9-13-2018

A Heuristic Method for Task Selection in Persistent ISR Missions Using Autonomous Unmanned Aerial Vehicles

Christopher C. Olsen

A Heuristic Method for Task Selection in Persistent ISR Missions using Autonomous Unmanned Aerial Vehicles

DISSERTATION

Christopher C. Olsen, Major, USAF

AFIT-ENY-DS-18-S-067

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENY-DS-18-S-067

A HEURISTIC METHOD FOR TASK SELECTION IN PERSISTENT ISR MISSIONS USING AUTONOMOUS UNMANNED AERIAL VEHICLES

DISSERTATION

Presented to the Faculty

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy in Aeronautical Engineering

Christopher C. Olsen, B.S.M.E., M.S.S.E.

Major, USAF

13 September 2018

AFIT-ENY-DS-18-S-067

A HEURISTIC METHOD FOR TASK SELECTION IN PERSISTENT ISR MISSIONS USING AUTONOMOUS UNMANNED AERIAL VEHICLES

DISSERTATION

Christopher C. Olsen, B.S.M.E., M.S.S.E.
Major, USAF

Committee Membership:

Dr. Donald L. Kunz
Chairman

Dr. Richard G. Cobb
Member

Dr. William P. Baker
Member

Dr. Scott L. Nykl
Dean's Representative

Abstract

The Persistent Intelligence, Surveillance, and Reconnaissance (PISR) problem seeks to provide timely collection and delivery of data from prioritized ISR tasks using an autonomous Unmanned Aerial Vehicle (UAV). In the literature, PISR is classified as a type of Vehicle Routing Problem (VRP), often called by other names such as persistent monitoring, persistent surveillance, and patrolling. The objective of PISR is to minimize the weighted revisit time to each task (called weighted latency) using an optimal task selection algorithm. In this research, we utilize the *average* weighted latency as our performance metric and investigate a method for task selection called the Maximal Distance Discounted and Weighted Revisit Period (MD^2WRP) utility function. The MD^2WRP function is a heuristic method of task selection that uses $n+1$ parameters, where n is the number of PISR tasks. We develop a two-step optimization method for the MD^2WRP parameters to deliver optimal latency performance for any given task configuration, which accommodates both single and multi-vehicle scenarios. To validate our optimization method, we compare the performance of MD^2WRP to common Traveling Salesman Problem (TSP) methods for PISR using different task configurations. We find that the optimized MD^2WRP function is competitive with the TSP methods, and that MD^2WRP often results in steady-state task visit sequences that are equivalent to the TSP solution for a single vehicle. We also compare MD^2WRP to other utility methods from the literature, finding that MD^2WRP performs on par with or better than these other methods even when optimizing only one of its $n+1$ parameters. To address real-world operational factors, we test MD^2WRP with Dubins constraints, no-fly zones in the operational area, return-to-base requirements, and the addition and removal of vehicles and tasks

mid-mission. For each operational factor, we demonstrate its effect on PISR task selections using MD^2WRP and how MD^2WRP needs to be modified, if at all, to compensate. Finally, we make practical suggestions about implementing MD^2WRP for flight testing, outline potential areas for future study, and offer recommendations about the conduct of PISR missions in general.

This is an exercise in fictional science, or science fiction, if you like that better. Not for amusement: science fiction in the service of science. Or just science, if you agree that fiction is part of it, always was, and always will be as long as our brains are only minuscule fragments of the universe, much too small to hold all the facts of the world but not too idle to speculate about them.

-Valentino Braitenberg, Experiments in Synthetic Psychology

Acknowledgements

Writing a dissertation requires thousands of hours of solitude. It's easy to forget, despite what it sometimes feels like, the pursuit of a doctorate does not take place in a vacuum. In fact, the end result would not be much to speak of if it did.

First and foremost, I'd like to thank my research advisor, Dr. Kunz, for being an open ear and, occasionally, making essential course corrections when my research began to wander. His guidance kept me on-track for graduating on-time.

I also extend deep thanks to my research committee members, Dr. Cobb and Dr. Baker, for lending a critical voice. Their critiques and suggestions greatly improved my body of work. Dr. Baker spent a great deal of his own time contributing to the periodicity proof in this document, while also maturing my mathematical view of the problem. There are several folks at AFRL/RQQ that deserve a hearty thanks. They spent their own research hours helping me understand the PISR problem so I could develop a thorough research plan.

Of course, when work extends into the night, the family bears the burden. I must thank my brilliant wife, an M.D. herself, for putting up with what has certainly not been the best version of myself for the last three years. It was her that told me to go for it, without hesitation, when I brought up the crazy idea of pursuing a Ph.D.. Lastly, though he wasn't born until near the end, I must also thank my son for giving me all the reason I needed to keep going.

Christopher C. Olsen

Table of Contents

Page

Page

List of Figures

List of Tables

A HEURISTIC METHOD FOR TASK SELECTION
IN PERSISTENT ISR MISSIONS
USING AUTONOMOUS UNMANNED AERIAL VEHICLES

I. Introduction

In the first decade of the 21st centry, concurrent advancements in computing hardware, navigation, and controls created the ideal conditions for the rapid rise in popularity of small, unmanned aircraft; commonly called "drones" in the mainstream, but more frequently referred to as Unmanned Aerial Systems (UAS) or Unmanned Aerial Vehicles (UAVs) in technical communities. Today, the physical size and cost of hardware continues to decrease and there is no apparent end in sight for UAV demand. Governments and businesses are hungry to explore and adopt new and practical UAV applications, from product delivery to national security.

Militaries were among the first to realize the benefits of unmanned aircraft and begin investing heavily in their development. The earliest UAVs were simply manned aircraft equipped with basic autopilots and deployed as airborne "torpedoes", decoys, or practice targets [6]. By the turn of the century, high-bandwidth satellite communications coupled with modern sensor technology enabled the U.S. to begin fielding Remotely Piloted Aircraft (RPAs) for airborne Intelligence, Surveillance, and Reconnaissance (ISR) missions over Afghanistan [7]. With high value unmanned systems such as the RQ-4 Global Hawk, MQ-1 Predator, and MQ-9 Reaper, the U.S. military has and continues to invest in RPAs for ISR and strike missions. While the current inventory of Department of Defense (DoD) unmanned systems provide mission capability with reduced cost and risk, they still require management by human

operators, either through direct flight control, sensor operation, or mission planning. Additionally, these assets, while certainly more attritable than manned aircraft, are nonetheless equipped with high value electronics; to put it plainly, their loss does not go unnoticed by commanders.

Thus, the stage is set to bring about the next era of unmanned aircraft, or to put it more precisely, to bring unmanned aircraft into the era of autonomy. The goal of autonomy is to further reduce the need for human oversight, such that the role of the human operator is simply to provide the autonomous agent with a goal. It is then up to the agent to decide how best to achieve the goal, even in the face of a changing mission environment. More will be discussed regarding autonomy and the control of autonomous vehicles in Ch. II. For now, the use of small, attritable, autonomous UAVs holds great promise for providing combatant commanders with low-risk and persistent ISR.

1.1 Motivation

The DoD Unmanned Systems Integrated Roadmap for FY2013-2038 advocates for the development of unmanned systems, to include UAVs, with autonomous and cognitive behavior. It specifically acknowledges their suitability for the Battlespace Awareness Joint Capability Area (JCA) [8],

> Battlespace Awareness is a capability area where unmanned systems in all domains have the ability to contribute significantly into the future to conduct ISR and environment collection-related tasks. Applications in this JCA include aerial, ground, surface sea, and undersea surveillance and reconnaissance. Today, these functions are performed by several systems across all domains and mission sets. In the future, technology will enable mission endurance to extend from hours to days and allow for long-endurance persistent reconnaissance and surveillance in all domains.

To realize this vision, concrete objectives must be derived from the high-level

abstractions in the Roadmap document. The goal of this dissertation is to investigate a practical decision-making algorithm for autonomous UAVs as one step toward "long-endurance persistent reconnaissance and surveillance", which we call persistent intelligence, surveillance, and reconnaissance (PISR).

We propose that a utility function is well-suited to serve as a basis for task selection in PISR. Utility-based decision-making stems from Utility Theory, an artificial intelligence concept which will be discussed in detail in Ch. II. In short, Utility Theory describes an agent decision-making process whereby decisions are based upon a utility value which is calculated from a utility function. The utility function takes the values of system state variables as input and outputs a corresponding utility value for taking the considered action. A rational agent pursues the decision yielding the highest utility.

A utility-based approach has many desirable attributes for PISR, which will be discussed at the end of Ch. II. Still, *choosing which state variables to include in a utility function and determining a mathematical relationship between them that results in desirable agent behavior is a significant challenge.*

One such utility function for PISR was proposed by Kalyanam[9], called the Maximal Distance Discounted & Weighted Revisit Period (MD^2WRP) algorithm. The derivation of MD^2WRP is presented in Ch. III. Characterizing agent decision-making under MD^2WRP and comparing its PISR mission performance to other task selection methods composes the main body of this research.

1.2 Research Questions, Scope, and Tasks

1.2.1 Research Questions.

Hypothesis: The MD^2WRP utility function can serve as the basis for PISR task selection decisions for single or multiple autonomous UAVs under a variety of opera-

tional constraints. Furthermore mission performance can be competitive with leading PISR task selection algorithms from the literature.

Research questions relating to this hypothesis are:

1. How do we define performance for PISR missions?
2. What is the relationship between the MD^2WRP parameters and how do they affect agent behavior?
3. How can the MD^2WRP parameters be optimized to maximize performance?
4. What is the underlying mathematical structure of MD^2WRP and how can it be used to predict PISR performance?
5. How can the MD^2WRP utility function be extended for use in multi-vehicle teaming?
6. How does the performance of MD^2WRP compare to other PISR solutions from the literature?
7. How does MD^2WRP perform in the presence of operational constraints?

1.2.2 Research Scope.

This research is focused on PISR task selection for autonomous agents (also frequently referred to as task scheduling). There are many facets to the control of autonomous agents. Task selection is the top-layer control, whose output is applied as input in the lower-layer control of trajectory generation. Our goal is to complete a rigorous study of task selection in support of the larger DoD research community objective, which is to develop a fully autonomous solution capable of executing all phases of a PISR mission.

The core results of this work are based in modeling and simulation under specific simplifying assumptions. In later phases of the work, some of the assumptions are lifted to simulate operational constraints likely to be encountered in the field. Theoretical work is also conducted, specifically in examining the mathematical structure of MD^2WRP and its implications on mission performance. Similarly, on the applied end of the research spectrum, the models used in this research were intentionally developed to mirror the existing autonomy software suite, Unmanned Systems Autonomy Services (UxAS), under development by Air Force Research Laboratory (AFRL). The goal is to maintain the relevance of the conclusions in this research to follow-on flight testing of MD^2WRP using UxAS.

1.2.3 Research Tasks & Ontology.

The following research tasks are defined in order to address our research questions:

1. **Define a performance measure for PISR missions.** Evaluate vehicle routing performance metrics from the literature and select one that is suitable for PISR to serve as a common metric for comparing MD^2WRP to other PISR methods.

2. **Characterize the MD^2WRP utility function.** Using both analysis and simple simulation scenarios, determine how MD^2WRP parameters affect vehicle behavior. Also, evaluate the transient and steady-state behavior of MD^2WRP in making task selections for PISR.

3. **Investigate optimization of the MD^2WRP parameters.** Propose a method to optimize the MD^2WRP parameters for any given task configuration.

4. **Develop a means for multiple vehicles to cooperatively use MD^2WRP for PISR task selection.** Modify MD^2WRP for use with multiple vehicles.

Evaluate different communication schemes for quantitative performance as well as suitability based on qualitative attributes.

5. **Compare MD^2WRP to other PISR methods.** Compare the performance of MD^2WRP, under the selected PISR performance measure, to other methods in the literature to include other utility-based approaches and combinatorial optimization solutions (*e.g.* the Traveling Salesman Problem).

6. **Evaluate MD^2WRP under operational factors.** Lift select simplifying assumptions in order to more realistically model real-world operational factors (*e.g.* the presence of no-fly zones). Develop methods to quantify the effects of operational constraints on MD^2WRP performance and propose methods to overcome operational challenges.

Figure 1.1 provides a visual mapping of how each research task stems from our hypothesis.

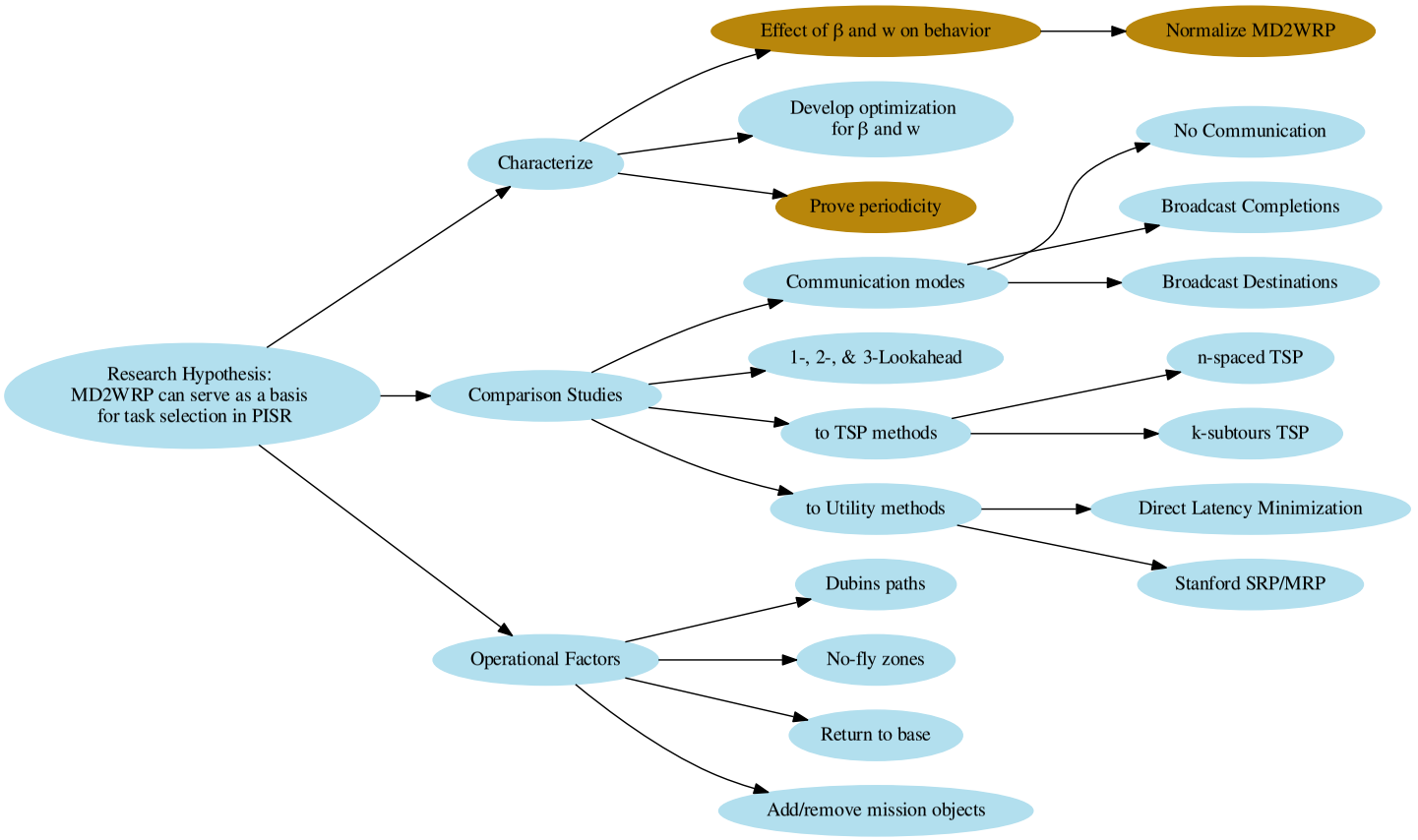


Figure 1.1. A visual mapping of research tasks and how they stem from the hypothesis. The yellow tasks only consider a single vehicle. Blue tasks consider multiple vehicles.

1.3 Assumptions

Assumptions are an important consideration in modeling and simulation research. We consider two categories of assumptions. The first category were supplied by the research sponsor (AFRL) according to their planned application.

- **Vehicles are fixed-wing UAVs with a dwell time of approximately six hours.**
- **Vehicles fly at constant speed and altitude.**
- **Missions last for 20000s, or approximately five and a half hours.**

The second type are simplifying assumptions. These assumptions apply to this entire work, unless specifically stated otherwise.

- **Vehicles travel Euclidean paths, that is, they fly between tasks in a straight line and have a zero turn radius.** For most real-world applications, tasks are likely to be located several kilometers apart, which is large compared to the vehicle's turning radius (less than 100 meters). So, the impact of the turn radius to flight time between tasks is negligible.
- **Vehicles have the capability to send and receive lossless communications across the entire simulated space.** While inter-vehicle communications are never guaranteed in real-world operations, the shortwave radios used in previous flight testing have proved to be reliable enough that this assumption ensures simulation results are meaningful for a majority of mission situations.
- **Vehicles may move freely between tasks. There are no path constraints.** Usually, PISR UAVs are operating in their own mission area. They are either far away from manned aircraft operations or operating at a lower flight level.

- **Vehicles are not required to return to base during the simulated scenario. They transmit their data back to the operations center with no distance or bandwidth limitations.** The PISR UAVs are equipped with satellite communication terminals for sending collected data back to the operations center in real-time.

- **The PISR tasks are point-searches, meaning they have zero service time, begin and end in the same location, and are considered complete as soon as a vehicle arrives.** Point-searches are good approximations for most PISR tasks. While some real-world tasks might include searching a road or a field (and thus have non-zero service time and different start/end locations), they can be modeled as a single point. In future research, MD^2WRP may be adapted to account for the specifics of non-point-search tasks.

1.4 Research Methodology

This research is primarily based in modeling and simulation. However, theoretical groundwork is also established to derive additional insight. Also, the work is done with an eye toward eventual flight testing, in an effort to ensure the results contained in this document are easily transitioned to the test range. Taken together, this document makes an attempt to pull the thread from theory to application, with an emphasis on the proof-of-concept that lies between.

The research is broken into three phases. The objective of the first phase is to characterize the MD^2WRP utility function. Using analysis and simulation, the intent is to understand how the MD^2WRP parameters influence agent behavior. Part of the characterization is selecting an appropriate performance metric for PISR and then exploring how MD^2WRP can be optimized to maximize performance under the chosen metric.

Once an understanding of MD^2WRP and its parameters is established, existing solutions for similar problems will be directly compared to MD^2WRP with the selected performance metric serving as a yard stick. Besides comparing MD^2WRP performance to other methods, this will also highlight the key features of each approach in terms of qualitative criteria such as scalability, complexity, and robustness. However, in order to compare MD^2WRP against other methods in the multi-vehicle case, we first explore three different communication modes for MD^2WRP.

Next, MD^2WRP is applied to a variety of scenarios designed to be representative of likely operational constraints. In each of these scenarios, individual simplifying assumptions are lifted to determine how MD^2WRP is affected by operational constraints as well as how it can be used in overcoming them.

1.5 Expected Contributions

While this work is specifically focused on the research and development of the MD^2WRP utility function for task selection in PISR, it is also expected to make broader contributions to the fields of vehicle routing and autonomous vehicle control.

We aim to develop *a scalable solution*. A significant advantage of utility functions for task selection is they are single, easy-to-evaluate algebraic expressions. The absence of a combinatorial optimization algorithm eliminates convergence issues and also results in quick decision making for the autonomous agent. The payoff is a task scheduling scheme that easily accommodates any practical number of tasks with little computational overhead. Additionally, in the future, these tasks may be more complex than simple point tasks; they may include line searches (*e.g.* road searches) or area searches which are beyond the capability of existing combinatorial algorithms for all but the simplest task configurations.

We also desire a task selection algorithm that is *suitable for a dynamic environ-*

ment. Because utility functions are simple algebraic expressions, they are quickly adapted to changing mission needs. Tasks may be added to or removed from the scenario without the need for a centralized recalculation of vehicle task assignments. Agent behavior can be set and adjusted with a small number of parameters to provide the desired behavior or to meet a performance requirement. Similarly, vehicles may be added or removed from service at any time. With an appropriate communication scheme, vehicles will automatically adjust their workloads to compensate for the number of UAVs and tasks.

1.6 Document Outline

The research hypothesis and related questions have been posed in this chapter, along with the list of research tasks. Chapter II surveys the existing literature for methods which have already been investigated for PISR. In Ch. III, the models and methods of this research will be described in detail. Chapter IV presents the detailed results and analysis conducted to answer the research questions and carry out the research tasks. Finally, Ch. V summarizes the major conclusions drawn from the results, describes the contributions of the research, and makes specific recommendations regarding MD^2WRP as a decision-making function for PISR task selection as well as recommendations for conducting PISR missions with autonomous UAVs in general.

II. Literature Review

2.1 Introduction

At its core, PISR is a type of Vehicle Routing Problem (VRP), wherein the subject vehicles consist of *autonomous* UAVs. This literature review will start by discussing what it means for a vehicle to be autonomous, especially in the context of DoD missions, and how the concept of control applies to such a vehicle. Next, we provide a brief overview of VRPs and their relationship to PISR. Then, we review several studies in PISR task selection that are based in two distinct methods: the Traveling Salesman Problem (TSP) and utility functions. Finally, the last portion of this chapter provides a survey delving deeper into the formulation and solution of some TSPs of special interest in PISR applications, as well as a discussion of the theory behind utility functions for autonomous decision making.

2.2 Autonomous Agents

2.2.1 Definition of Autonomy.

There are numerous definitions of autonomy and what constitutes an autonomous system. However, to pick a functional definition for discussion, the following from the DoD-sponsored 2012 Autonomy Research Pilot Initiative (ARPI)[10] is provided,

> Systems which have a set of intelligence-based capabilities that allow them to respond within a bounded domain to situations that were not pre-programmed or anticipated in the design (i.e., decision-based responses) for operations in unstructured, dynamic, uncertain, and adversarial environments. Autonomous systems have a degree of self-governance and self-directed behavior and must be adaptive to and/or learn from an ever-changing environment (with the human's proxy for decisions).

2.2.2 Control of Autonomous Agents.

From the above definition, it is clear that autonomous vehicles are expected to perform the mission with a minimal level of human oversight, but they also must act "within a bounded domain". Part of that bounded domain are the control laws which govern vehicle actions. To cage the literature review to follow, we present a brief overview of the various levels of autonomous vehicle control.

When an autonomous vehicle goes into a given environment to achieve a goal, there are three sequential decision making processes that must be considered: task selection, path selection, and path following. These are summarized in Fig. 2.1.

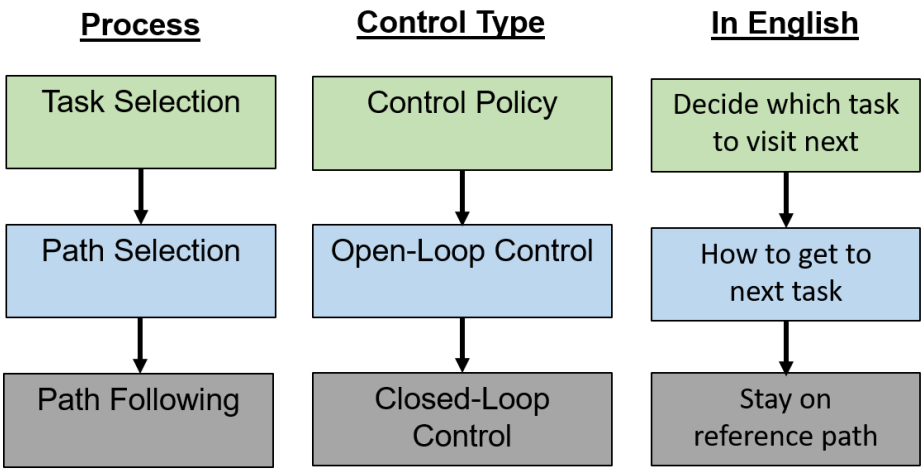


Figure 2.1. Levels of control for autonomous agents.

The first and highest level of decision making, and the one which is the focus of this research, is *task selection*. This is when the vehicle decides which tasks to do and in what order, frequently referred to in the literature as a control policy. Often, the goal is to find the optimal control policy through combinatorial optimization or with Artificial Intelligence (AI) tools, such as utility/reward functions or decision trees.

Once the vehicle has decided the order in which to accomplish tasks, it must determine how to physically move from its current location to the task location. This is

known as the path planning problem, or trajectory optimization, and is accomplished with open-loop control, meaning the control is executed in the absence of feedback. Two examples of path planning/optimal trajectory algorithms are the pseudospectral method[11] and A* search[12].

Finally, with the path decided, the agent must remain on the path as it moves to its destination. This is done with closed-loop (feedback) control by minimizing the error between the vehicle's actual trajectory and the planned (reference) trajectory[13].

2.3 PISR as a Vehicle Routing Problem

The VRP was first proposed in 1959 by Dantzig and Ramser[14]. It seeks to determine an optimal set of routes for a fleet of, m, vehicles that must visit a set of, n, customers starting from a depot. Optimality criteria are determined by an objective function, with the usual goal of minimizing the total distance traveled. Since its introduction, the VRP has been intensely studied due to its cost saving implications for a wide variety of industries.

In the terminology of computational complexity theory, the VRP is NP-hard[15], implying that exact solutions are, in general, not available. To find an exact solution the problem must be sufficiently small or a number of simplifying assumptions must be imposed. As such, many VRPs are typically approached with heuristic methods which yield good results for practical purposes.

The most broad classifications of VRPs are static/dynamic and deterministic/stochastic. For the static problem, all data are known to the planner *a priori*. For example, task locations are known up front and do not change. Thus the vehicle's path is set before leaving the depot. In the dynamic formulation, the problem data may change during the mission. For example, new service requests could appear at any time.

Independent of the static or dynamic nature, a problem may be deterministic or stochastic. In the deterministic formulation, there is no uncertainty in the problem data (*e.g.* the exact location of each customer is known) whereas in the stochastic formulation some uncertainty may exist (*e.g.* customer locations must be discovered).

For most of this work, we treat the problem of task selection in PISR as a *static-deterministic* VRP. It is static because the problem configuration does not change with time. The tasks to be surveilled, as well as their associated priorities, are known prior to the vehicle leaving the base. At the end of Ch. IV, we will briefly consider a dynamic problem, where the number of vehicles and tasks change during the mission. Our problem is deterministic because there is no uncertainty in the task locations or the time that must be spent at each task (in this research it takes zero time to service a task once the vehicle arrives).

2.4 Strategies for Task Selection in PISR

The problem of continually monitoring discrete points of interest (*i.e.* tasks) with one or more vehicles is referred to in the literature under many names including persistent monitoring, persistent surveillance, sweep coverage, and patrolling. In this research, we use the term persistent intelligence, surveillance, and reconnaissance, or PISR. Though the names differ, the core problem remains the same. Vehicles must select tasks in an optimal order so as to minimize the time tasks spend waiting for service. Previous studies have been based on two fundamental methods for task selection in PISR: the Traveling Salesman Problem and utility functions. We review several studies of each type in this section.

The TSP is a classic combinatorial optimization problem and has received a thorough and broad treatment in the literature. Many variants of the TSP have been studied for both single and multi-vehicle cases. Consequently, there exist a wide va-

riety of exact, approximate, and heuristic solutions to the many types of TSPs. A survey of the mathematical formulation of and solutions to some specific TSP types of special interest for PISR is included in Sec. 2.5. Below however, in presenting the application of TSPs to the PISR task selection problem, the details of the formulations and solutions for each type of TSP are not discussed. Instead, the focus is on how each TSP variant is implemented as a task selection solution for PISR.

Using utility functions as a basis for decision making by an autonomous agent (*i.e.* utility theory) stems from the field of AI. A brief introduction to utility theory is provided in Sec. 2.6. In this section, we focus on previous studies of PISR task selection that have implemented utility functions as the basis by which vehicles select tasks, but without discussing how those functions were developed or the underlying theory as to why utility functions are a valid approach.

2.4.1 TSP Methods.

The most common approach in the literature for determining the optimal task visit sequence in PISR is to solve one or more TSPs, with the tasks equating to nodes on a graph. The resulting tour(s) is/are assigned to one or more vehicles. The vehicles continually travel their assigned tours and service tasks, with new tours being generated and assigned if the number or location of tasks, or number of vehicles, change during the mission.

Chevaleyre conducts theoretical analysis for two multi-vehicle patrolling strategies based on the TSP[1]. The first is a cyclic strategy wherein vehicles are spaced along a closed TSP tour such that each task has equal revisit time. Chevalyere provides a proof for the worst-case revisit time for any task, given such a strategy. The second is a partition strategy where a TSP tour is divided into k subtours, where k is the number of agents. Again, a proof of the worst-case revisit time is demonstrated.

Figure 2.2 shows a simple example of cyclic TSP versus a k-subtour. Finally, the author conducts trials of each strategy on six different graphs to validate the analytical proofs on worst-case revisit time and to compare the performance of each strategy. The author concludes that the cyclic strategy results in lower average revisit times on all graphs except when the graph contains one or more long edges (*i.e.* the $d_{ij,\max}$ is large compared to the average d_{ij}), in which case the k-subtours strategy is the better choice.

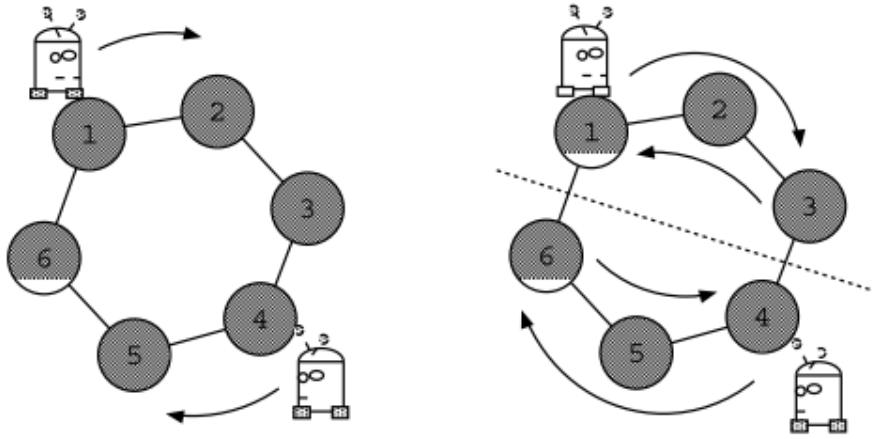
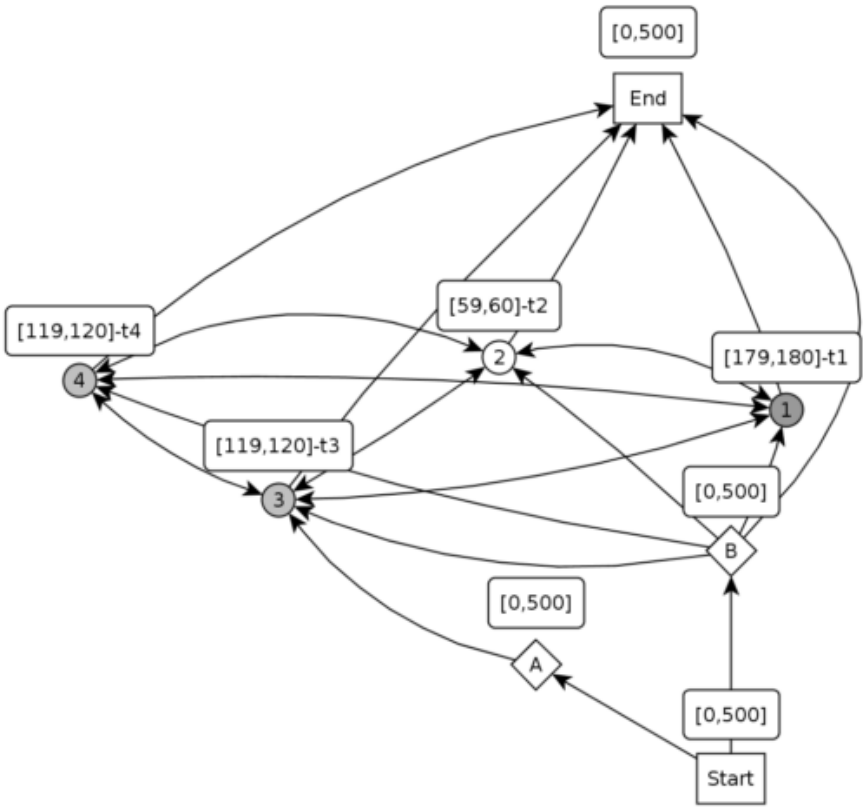


Figure 2.2. Left: Two vehicles on a cyclic TSP tour. Right: Two vehicles with subtours, based on the original TSP tour. (Image taken from [1].)

Stump and Michael[2] compare the cyclic TSP algorithm from Chevaleyre[1] to two different Vehicle Routing Problem with Time Windows (VRPTW) algorithms (see [16] for a treatment of the VRPTW). The first VRPTW algorithm, which the authors call "Next-Visit VRPTW", considers a single cycle of the vehicle. In other words, it creates a Hamiltonian tour (that is, each task may only be visited once per cycle) that satisfies all time window constraints (Fig. 2.3). The second algorithm, "Horizon VRPTW", attempts to generate an optimal route that visits all tasks multiple times over a defined time horizon, with higher priority tasks having more frequent visit window constraints. For example, a high priority task might require three visits

within the horizon, whereas a low priority task only requires one. The authors also impose a "return to base" constraint, since the quadrotor UAVs used in the research must recharge from time to time at a base station. They simulate the cyclic TSP and Next-Visit VRPTW in a scenario involving multiple task locations with a team of quadrotor UAVs over a period of six hours. The authors did not publish results for the Horizon VRPTW due to issues with the algorithm's ability to satisfy revisit period constraints. For the cyclic TSP strategy, five UAVs were used, whereas the Next-Visit VRPTW allows for anywhere from one to six UAVs to be employed at a given time, with the algorithm deciding the appropriate number to employ. Their results show the cyclic TSP strategy produces more frequent visits to the prioritized tasks, but that the Next-Visit VRPTW still meets the minimum revisit times. The authors note the advantage of the Next-Visit VRPTW is its ability to incorporate varying patrol periods while resulting in less total flight time. Essentially, Next-Visit VRPTW is able to meet requirements while conserving resources. Finally, the authors provide a defense of their exact, centralized solution, acknowledging the risk of a single point of failure and the imposed communication burden. Still, they argue, their approach provides a quantitative benchmark for assessing decentralized heuristic solutions that may be more robust.

Similar to Stump[2], Pasqualetti *et al.* explore a patrolling problem with prioritized tasks[17]. They also use the cyclic TSP strategy, which they refer to as the Equal-Spacing trajectory, as a benchmark. However, rather than achieving some required visit window for tasks as in [2], they propose the "Equal-Time-Spacing" trajectory which seeks to minimize the weighted revisit period. The Equal-Time-Spacing trajectory allows agents to hold their position at a task based on its priority so as to minimize weighted revisit time while keeping the time-spacing between vehicles equal. They show that, for a variety of priority sets, the Equal-Spacing al-



(a) Graph with Time-Windows by Priority

Figure 2.3. Diagram depicting the use of VRPTW to enforce priorities. High priority nodes have more frequent windows. (Image taken from [2].)

gorithm represents a conservative upper bound to the achievable performance of the Equal-Time-Spacing algorithm, which often results in lower refresh times. Next, the authors propose two different distributed control algorithms for implementing the Equal-Time-Spacing trajectory, each with differing communication constraints. Using 3 robots and a simulation with 35 tasks and a lab experiment with 6 tasks, the authors demonstrate that the distributed control algorithms eventually converge to the predicted centralized solutions.

In [18], a centralized and decentralized approach are taken to the persistent moni-

toring problem. (The decentralized algorithm is described below in the section on utility function strategies). The centralized algorithm, which the authors call CSWEEP, is a TSP-based approach in which each of the tasks to be monitored is considered a node. A classic TSP is solved and the resulting tour is partitioned into k subtours, much like in the partition strategy of Chevaleyre[1]. The mobile sensors then continually traverse their subtour, which provides an upper bound on the maximal revisit time to any given node.

The approach to persistent monitoring taken in [3] is to minimize the maximum weighted latency between task visits by assigning each task as the vertex of a graph, with the weight of each vertex corresponding to the priority of the task. The edges between vertices represent the travel time between them. They then define weighted latency as the time between consecutive visits to a vertex multiplied by the weight of the vertex. Since the problem of minimizing the maximum weighted latency of a closed tour that visits all nodes is NP-Hard, two approximation algorithms are proposed. The algorithms work by partitioning the graphs by node priority and then generating subtours that visit all nodes at least once, but visit high priority nodes more often. The generated subtours are then optimized by solving a TSP on the subset of vertices. See Fig. 2.4 for a depiction of the node partitioning scheme. The optimized subtours are traveled by the agent in sequence to accomplish the persistent monitoring task. The algorithms are tested on a variety of graphs with node counts on the order of several thousand. Their results show cost reductions of 40-70% over a standard TSP tour, with more cost savings achieved as the node count increases. Of course, the authors' algorithm is at a significant advantage since it allows for multiple visits to a high priority vertex before visiting those of lower priority, whereas a TSP tour must visit all other vertices before visiting a high priority vertex again. Still, their approach provides a way to handle task priority and establishes a metric for

gauging and bounding performance.

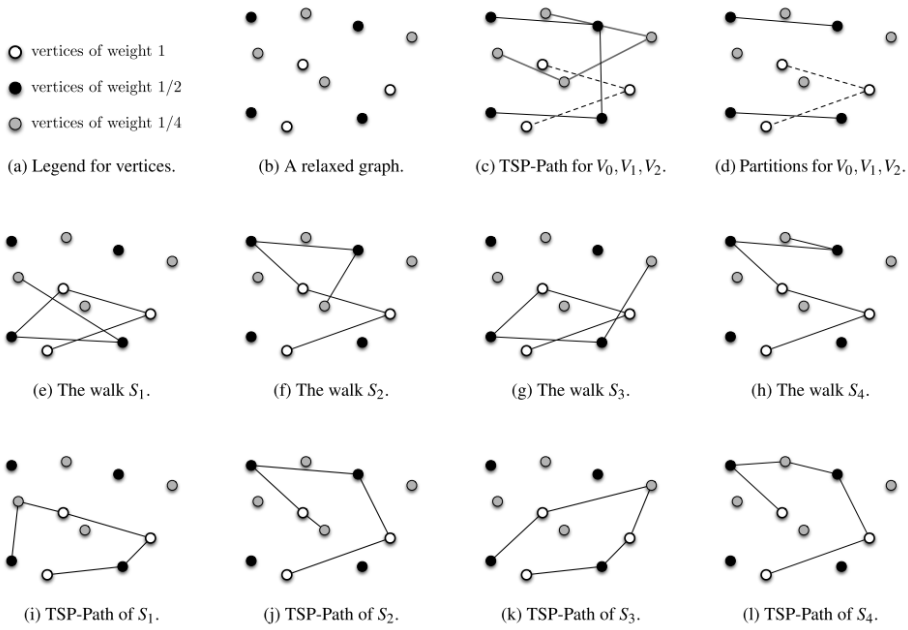


Figure 2.4. Example of the node partitioning algorithm from [3] (original image from the source), where multiple subtours are constructed with high priority nodes being included in more subtours than those of low priority.

The work of Smith and Rus[19] is similar to that from [3] above. They also define the maximum weighted latency metric (which they call "maximum urgency") for a set of prioritized tasks. They propose the *Partition-Tour* policy, which creates k partitions in the region containing all tasks, one for each vehicle, k. Then, the partitions are ordered according to a macro-TSP solution. Within each partition a small TSP is solved, with nodes of priority 1 being visited in every partition, but nodes of priority 2 only visited in $\frac{1}{2}$ the partitions, and so on such that nodes of priority l are visited in every $\frac{1}{l}$-th partition, where l is the priority of the node. Subtours are generated inside each partition in this fashion until all nodes have been visited at least once. Then, 2^{l-1} full tours are drawn, each connecting a set of subtours. The end result is a sequence of tours that, when executed by the k agents, is asymptotically optimal. Unfortunately, the *Partition-Tour* policy scales poorly with l, the number

of priority levels, so the authors introduce a computationally efficient heuristic based on the k-opt TSP improvement method[20, 21]. The heuristic solves for a TSP tour through all nodes, and then assigns vehicles to visit subsets of vertices along that tour, with higher priority vertices appearing in more subtours.

Another common approach for patrolling a set of n tasks with k vehicles is to partition the tasks into k clusters using a k-means clustering algorithm[22][23][24]. The k-means algorithm was first proposed by Stuart Lloyd for signal processing applications[25]. It has since been applied in numerous fields for statistical analysis. In PISR, if we assume each task is a "data point", we can use k-means to efficiently group tasks into clusters for easier servicing by a team of vehicles. K-means aims to assign a set of n data points to k clusters with the objective to minimize the "Within Cluster Sum of Squares (WCSS)", that is, the sum of the squared distances between the data points and the mean (centroid) of their assigned clusters. Solving the WCSS objective function is NP-Hard[26], so k-means proposes a heuristic minimization method. Of course, this results in sub-optimal solutions that are dependent upon the initial conditions of the algorithm. As such, many different initialization methods for k-means clustering have been explored. K-means++ is one of the most popular initialization methods because it has been shown to perform well on a wide variety of data sets[27]. From MacKay[28], to start the algorithm, each mean (the centroid of each cluster) must be initialized. The most basic initialization method is to generate k random values for each mean. The rest of the algorithm proceeds in two steps: an *assignment* step and an *update* step. In the assignment step, each data point is assigned to the nearest cluster. Then, in the update step, the centroid of all points in each cluster is calculated and becomes the new mean for that cluster. The two-step algorithm proceeds until data point assignments are unchanged, resulting in no changes to the means. While the basic k-means clustering algorithm

always converges to a fixed point, the random assignment of initial means can result in final clusters that are objectively bad with respect to the optimal clustering. K-means++ proposes an initialization procedure that is guaranteed to find a solution within $O(\log k)$ of the optimal[29]. The k-means++ initialization starts with selecting the first mean uniformly at random from the vector of data points. Then, for every data point, the distance between it and the nearest center in the set of centers that have already been created is calculated, called $d(x)$. A new mean is chosen from the set of points, with the probability of any point becoming a new mean based on a weighted probability distribution that is proportional to $d(x)^2$. This initial mean selection process is repeated until k means have been chosen. The algorithm then finishes using the basic two-step k-means clustering process. Once the PISR tasks have been partitioned with k-means clustering, each vehicle continually visits the tasks within its assigned cluster according to a Euclidean TSP solution on the subset of tasks. In this research, since each vehicle is traveling its own TSP subtour, we refer to the k-means multi-vehicle PISR method as "k-subtours".

2.4.2 Utility Function Methods.

The authors of [18], who implemented the TSP-based CSWEEP algorithm above, also consider a decentralized algorithm, which they name DSWEEP. Unlike CSWEEP, DSWEEP assumes no central planner is available and instead relies on inter-vehicle communication to provide information about the environment. Vehicles share their knowledge about task age (that is, the time elapsed since a task was last visited) and locally store the information in a sweep table. Using the sweep table, vehicles decide which task to visit next based on a utility calculation. The next-visit decision is performed iteratively within "hop" rings around the vehicle, with one hop defined by a user-defined travel time. The vehicle first looks within one hop for any tasks

that have yet to be visited by any vehicle. If only one exists, it is selected as the next destination. If multiple unvisited tasks exist, it chooses the nearest one. If all one-hop tasks have been visited previously, it looks at the impending sweep deadlines (defined as the task age plus the required sweep period) and, if any of the sweep deadlines is within one hop time, it marks that task as urgent. If multiple urgent tasks are found, it selects the one with the earliest sweep deadline. If no urgent tasks are found within one hop time, the vehicle extends its search to tasks within a two-hop radius. This process continues until a task is selected. The authors simulate DSWEEP on a map of 100 randomly generated tasks inside a 10 by 10-unit square, imposing three different sweep periods (of $T = 80, 120, 160s$), and allowing agents to exchange information if within a 2-unit distance of each other. The results show DSWEEP to outperform random task selection for all simulated scenarios, with DSWEEP maintaining 78% of nodes within their sweep deadlines for the $T = 80s$ scenario while only 51% of nodes were kept current when random task selection was used.

Machado *et al.*[4] investigate seven multi-vehicle patrolling architectures, with some architectures using task idleness to make utility-based decisions and others choosing destinations randomly. Architectures were also differentiated on other features, such as sensor range and type of communication between vehicles. Figure 2.5 provides a summary of the considered architectures. The authors adopt three primary performance criteria to compare the various architectures: average task idleness (average idleness among all tasks throughout the simulation), worst idleness (the highest idleness value achieved by any task), and exploration time (how long it takes for every task to be visited at least once). The most interesting aspect of the Machado results is the comparison between task idleness and random task selection. They show that random task selection performs much worse than using task idleness as a decision basis, which establishes idleness as a good heuristic for task selection in

PISR. Also, as might be expected, performance improves with increasing inter-vehicle communication and coordination.

Architecture Name	Basic Type	Communication	Next Node Choice	Coordination Strategy
Random Reactive	reactive	none	locally random	emergent
Conscientious Reactive			locally individual idleness	
Reactive with Flags		flags	locally shared idleness	
Conscientious Cognitive	cognitive	none	globally individual idleness	
Blackboard Cognitive		blackboard	globally shared idleness	
Random Coordinator		messages	globally random	central
Idleness Coordinator			globally shared idleness	

Figure 2.5. The seven architectures from [4] (Figure taken directly from the source).

A team at Stanford University investigated a utility function approach for both single and multi-vehicle PISR in [5] and later extended their work to laboratory flight testing in [30]. Their vehicles were tasked with minimizing the maximum age of any single graph node across a gridded network of nodes. Node ages are reset to zero when an agent visits. Figure 2.6 provides a sample plot of maximum age for a simple 2-node example.

The single-vehicle utility function employed to minimize maximum latency is,

$$V = \max_j \{T_j + w_0 \delta_{ij}\}, \qquad \forall j \in \{1 \ldots, n\}$$

where V is the value of the selected task, T_j is the age of candidate task j, δ_{ij} is the distance between current task i and j, and w_0 is a weight parameter with units of s/m. For the simple test case of a single vehicle and two nodes, the authors found $w_0 = -1/V_{UAV}$, where V_{UAV} is the constant velocity of the UAV, to be the optimal value. When more than two nodes are introduced, the authors acknowledge that using $w_0 = -1/V_{UAV}$ may not be optimal, and in that case they use an iterative sampling optimizer to approximate the optimal value for w_0.

The authors of [5] and [30] then extend their utility function to the multi-vehicle

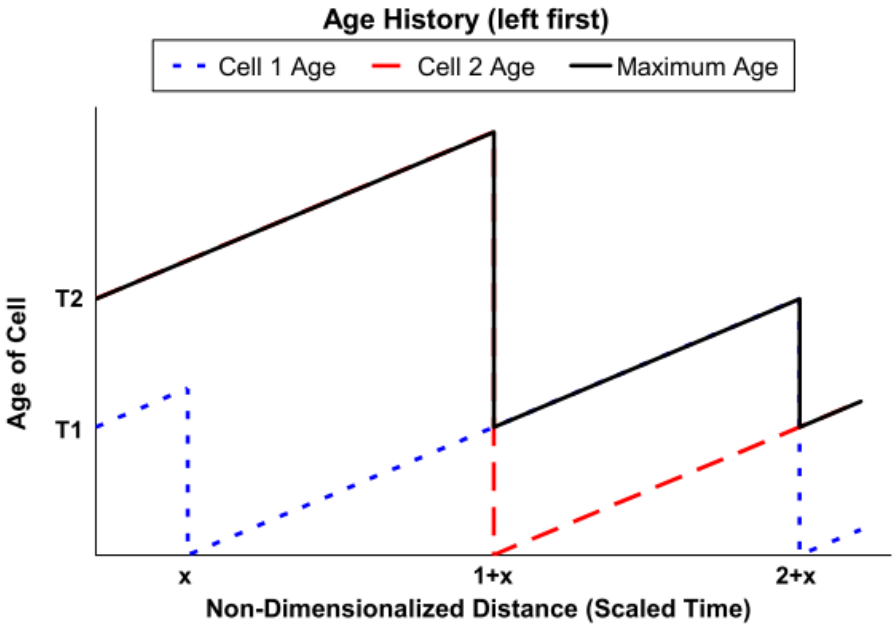


Figure 2.6. Notional progression of two node ages (in blue and red) as well as their sum, which is the maximum age of the network represented by the black line. (Figure adapted from [5]).

case by including an additional term,

$$V = \max_j \{T_j + w_0 \delta_{ij} + w_1 \min_{k \neq i}(\delta_{kj})\} \qquad \forall j, k \in \{1, \ldots, m\}$$

where w_1 is an additional positive weight parameter (s/m) and δ_{kj} is the distance between the k-th vehicle and task j. In other words, all other things equal, the multi-vehicle function encourages each vehicle to target nodes far away from other vehicles. As in the single vehicle case, the weight parameters w_0 and w_1 are optimized offline. The authors call the multi-UAV reward function the Multi-agent Reactive Policy (MRP), named for the reactive nature of vehicle coordination under the policy. They then compare the performance of MRP against a proactive Space Decomposition (SD) strategy for vehicle coordination. In SD, all nodes are divided optimally into k partitions, one for each vehicle, with a genetic algorithm. The partitioning is optimal

in the sense that it seeks to divide nodes among the vehicles such that maximum latency is minimized. Simulation results show that SD performs better, but as the number of vehicles increases, the performance of MRP approaches that of SD. As a final step, the paper goes on to impose simple dynamic constraints on the vehicles, limiting their turn radius. They then conduct several simulations with varying numbers of vehicles and minimum turn radii and compare the performance of the policy using Euclidean distances for δ_{ij} versus Dubins trajectories. In general, the policy using actual flight distances outperforms the Euclidean policy (so long as points are close enough together for dynamics to be a factor). However, an interesting result is that the performance gains derived from using actual distances over Euclidean decrease as the number of vehicles increases, regardless of the minimum turning radius. The authors explain this as an emergent behavior, resulting from the ability of other vehicles to "fill in the gaps" for each other.

2.5 Survey of the Traveling Salesman Problem

In this section we survey several variants of the well-known Traveling Salesman Problem, or TSP, that are of special interest for PISR applications. The TSP is a combinatorial optimization problem and special case of the general VRP. It asks, given a set of cities to be visited, in what order should a salesman visit the cities to minimize the total distance traveled? There are many variants of the TSP. Below, the classic, two-dimensional, Euclidean TSP along with three TSP variants of special interest to task selection in PISR will be discussed. Like the VRP, the TSP is NP-hard[31]. Still, due to its popularity, many TSP algorithms have been developed. These algorithms can find solutions for TSPs with node counts into the millions, often with solutions less than 1% from optimum[32].

The following terminology will be used to classify algorithms:

- *Exact* - finds the global minimum solution.
- *Approximate* - returns a solution with a worst-case bound; it approximates the global minimum to within some factor ϵ.
- *Heuristic* - generates locally optimal solutions that deliver empirically good results, but without any guarantee of being near or approaching the global minimum.

2.5.1 The 2D Euclidean TSP.

The Euclidean TSP can be formally defined in $\mathbb{R}^2$ as follows. Let the graph be $G(V, E)$, where $V = \{1...n\}$ represents the nodes and E the edges. Each edge, E, has an associated weight, d_{ij}, which corresponds to the distance between each V_j. Thus the agent must determine a closed path that visits each V_j exactly once (this restriction is called a Hamiltonian tour) while incurring the minimum cost, J, where J is the sum of all $d_{i,j}$ traveled.

The most common formulation of the TSP is in the form of an integer linear

program (ILP). The below ILP formulation was taken from [33]:

$$
\begin{aligned}
\text{Minimize} \quad & \sum_{i \neq j} d_{ij} x_{ij} && (1)\\
\text{subject to} \quad & \sum_{\substack{j=1 \\ n}} x_{ij} = 1, \qquad i = 1, \ldots, n, && (2)\\
& \sum_{i=1}^{n} x_{ij} = 1, \qquad j = 1, \ldots, n, && (3)\\
& \sum_{i,j \in S} x_{ij} \leq |S| - 1, \\
& S \subset V, 2 \leq |S| \leq n - 2, && (4)\\
& x_{ij} \in \{0, 1\}, \\
& i, j = 1, \ldots, n, \qquad i \neq j && (5)
\end{aligned}
\tag{2.1}
$$

where x_{ij} is a binary variable, equal to 1 if and only if the arc associated with d_{ij} is used in the solution. Constraints (2) and (3) are degree constraints, specifying that each vertex (or city) may only be entered once and exited once. Stated another way, this constraint ensures each vertex has a degree of exactly two. Constraint (4) eliminates solutions with subtours while (5) imposes the binary condition on variable x_{ij}.

2.5.1.1 Exact Algorithms.

One of the earliest and most common algorithms for solving Eq. 2.1 is the branch-and-bound (BB). Laporte provides an excellent qualitative description of BB algorithms:

> In the context of mathematical programming, they can best be viewed as initially relaxing some of the problem constraints, and then regaining feasibility through an enumerative process. The quality of a BB algorithm

is directly related to the quality of the bound provided by the relaxation. [33]

The relaxation in Eq. 2.1 is typically performed on constraint (4), which allows for subtours in the solution. A solution without subtours can then be found by solving an assignment problem, for which $O(n^3)$ solutions exist. The BB algorithm with subtour relaxation was used by Carpaneto and Toth in 1980 to solve 240-vertex TSPs in less than one minute[34]. In 1981, Balas and Christofides introduced constraints into the objective function via a Lagrangian approach and used BB to solve TSP instances of 5,000 vertices within 40 seconds and 500,000 vertices in just over 3.5 hours[35].

A more modern approach is to use the cutting plane method from linear programming (LP). This is often combined with a branching technique to form a branch-and-cut (BC) algorithm[36]. With the cutting plane method, the TSP is formulated as an ILP as in Eq. 2.1, but the binary constraint (5) is relaxed. The LP is then solved. If the optimal solution x^* consists of only ones or zeros, the optimal solution to Eq. 2.1 has been found. If x^* is not binary, a "cut", or linear inequality constraint, is added to the relaxed LP such that no integer solutions are eliminated while removing the current non-binary x^*.

Methods implementing the cutting plane technique are perhaps the best exact TSP algorithms known to date. The Concorde TSP Solver from University of Waterloo implements the BC algorithm as its primary solver, and has been used to solve TSP instances with up to 85,900 cities to optimality[37].

While BB and BC along with their numerous variations on relaxation are among the most popular exact TSP algorithms in the literature, it should be noted that many other exact algorithms have been introduced, including those based on shortest spanning trees, the shortest spanning arboresence bound, and the 2-matching lower bound. A detailed discussion of each of these algorithms is beyond the scope of this literature review, but Laporte provides brief descriptions and additional references in

[33].

2.5.1.2 Approximate Algorithms.

One of the earliest and most widely known approximate TSP algorithms for the symmetric, Euclidean TSP in $\mathbb{R}^2$ is that of Christofides. A symmetric TSP is one in which $d_{ij} = d_{ji}$. That is, the time to travel between i and j is the same regardless of the direction traveled. To start, Christofides finds a minimum spanning tree (MST) for the graph $G(V, E)$ describing the TSP. This can be done in $O(n^2)$. Since the resulting MST is not, in general, a Hamiltonian tour, Christofides then performs a minimum-cost matching algorithm ($O(n^3)$) on all nodes with degree 1 (nodes that are only connected to one edge). It can be shown that this algorithm provides a worst-case bound on the solution to any symmetric TSP in $\mathbb{R}^2$, with an approximation ratio of $\frac{3}{2}$. In other words, the cost of a solution generated by Christofides is guaranteed to cost no more than a factor of 1.5 times the optimum[31].

In 1996, Arora improved upon Christofides' $\frac{3}{2}$ approximation ratio[31]. Arora's algorithm, based in dynamic programming, achieves a worst-case bound of $(1 + \epsilon)$ for a cost of $n^{O(1/\epsilon)}$. With Arora's algorithm, a TSP solution can be found that is as close to the optimum as desired. While the author admits that the implementation is slow for even moderate values of ϵ, the algorithm opens the door for faster solutions via parallelization, since it naturally breaks up a single large TSP instance into many smaller instances. Arora's approach is also valuable for creating benchmark solutions against which speedier heuristic algorithms can be evaluated.

2.5.1.3 Heuristic Algorithms.

Christofides' and Arora's algorithms provide approximate solutions with a worst-case bound. In addition, many heuristics have been developed that are known to

provide "good" solutions in an empirical sense, but without a formal bound on performance. As an intuitive example, consider the Nearest Neighbor (NN) algorithm. In NN, a tour is constructed one edge at a time. An arbitrary vertex is chosen as the starting location and connected to the next closest vertex. That vertex is then connected to the nearest unconnected vertex and so on. The last vertex is connected back to the first. The complexity of NN is $O(n^2)$. As a variation, all n vertices can be considered as the starting point, which increases the cost to $O(n^3)$ but tends to yield better solutions[38].

The NN algorithm constructs a solution edge by edge, but some heuristics adopt a tour improvement approach. An example of this is the k-opt algorithm[20, 21]. With k-opt, an arbitrary initial tour is constructed. Then, k edges are removed from the initial tour and reconnected in all possible ways. If a shorter tour is found, it becomes the initial tour for the next iteration of k removals. The process is repeated for a set number of iterations, or until no improvements are found. Commonly, k is selected to be either 2 or 3, but Lin and Kernighan demonstrated an improved variation where k is chosen dynamically at the start of each iteration yielding better results[21].

Other examples of tour improvement heuristics are Ant Colony Optimization (ACO)[39] (a swarm intelligence algorithm by Dorigo) and simulated annealing (SA)[40], an optimization technique that emulates the natural annealing process of metals. Both ACO and SA are types of evolutionary algorithms, wherein an initial solution is constructed and incrementally improved through the random exploration of the state space.

2.5.2 The TSP with Time Windows.

In the standard TSP, the salesman is only interested in minimizing the total distance traveled. The only restriction is that each city is visited exactly once. What

if, however, some or all of the cities on the salesman's list had restrictions as to how early or late they could be visited? This variation is the Traveling Salesman Problem with Time Windows (TSPTW). It may also be referred to as the time-constrained TSP.

In the TSPTW, each node has an associated time window, which defines when the node can be visited, as well as a service requirement, q which represents some amount of goods that must be picked up or dropped off at that node. An agent (or agents) are then dispatched from a depot to visit the nodes. The agents may arrive at a node before the time window opens and wait there with no penalty, but it may not arrive after the time window has closed. Each agent also has a fixed capacity, Q, so that the net load from goods picked up and dropped off may not exceed Q. (Note that for PISR, Q can be considered infinite, since the "goods" being handled by the agent are data and we impose no data capacity limitations.) The most common objective is to minimize the total number of tours required to meet all demands with a set number of vehicles without exceeding vehicle capacity. The secondary objective is to minimize total distance traveled.

2.5.2.1 Exact Algorithms.

Despite the fact that the TSPTW has been shown to be an NP-complete problem[41], several exact algorithms have been proposed for simplified versions of the problem with varying degrees of success. One of the earliest was proposed by Christofides, Mingozzi, and Toth in 1981[42]. The authors implemented a branch-and-bound algorithm, using a state-space relaxation from a dynamic program to derive the lower bound. Their algorithm was demonstrated to be successful on a TSPTW instance with up to 50 nodes, so long as the time windows were sufficiently tight.

In a 1983 technical note, Baker formulated the TSPTW as an ILP and proved

the dual of his model to be a disjunctive graph model, for which solutions exist from scheduling theory[43]. By relaxing the dual, Baker's algorithm was able to solve TSPTW problems of up to 50-nodes, with the restriction that only a small percentage of the time windows could overlap.

Nearly a decade later, Dumas *et al.* applied a dynamic programming approach to the TSPTW, leveraging the time window constraints to eliminate large chunks of the problem state-space[44]. The authors were able to solve significantly larger problem sizes than in previous work, demonstrating their algorithm's effectiveness on an instance with 200 nodes and "fairly wide" time windows. Furthermore, if the geographical density of nodes was kept constant for increased problem sizes, they were able to solve problems of up to 800 nodes.

Tsitsiklis considers several special cases of the TSPTW, including when the number of nodes is bounded, the time windows open at $t = 0$, and the time windows are infinite after opening[45]. For each special case, the author either proves its NP-completeness or presents a forward dynamic programming algorithm that solves in polynomial-time.

2.5.2.2 Heuristic Algorithms.

Bräysy and Gendreau[16] provide a survey of heuristic and meta-heuristic algorithms for solving the TSPTW. (They use the terminology Vehicle Routing Problem with Time Windows (VRPTW)). They break the surveyed algorithms into two categories: route construction and route improvement methods. The original papers they reference are cited below.

In route construction, tours are constructed piecewise until a feasible solution is generated, with each sequential node selected based upon a cost minimization criteria. Node selection is further constrained by vehicle capacity and time windows. Solomon

proposed the simple "Giant-Tour" route construction heuristic in 1986[46]. It first solves a classic TSP with a single vehicle visiting all nodes, then splits the Giant-Tour into smaller tours until all nodes are visited and vehicle capacity constraints are satisfied.

One of the most popular route construction methods is the "Savings" heuristic, also proposed by Solomon[47]. The algorithm is initialized by servicing every node with its own route. It then sequentially evaluates the savings that would be realized from combining two routes and selecting the combination with the largest value in savings, subject to the feasibility of the combination (*i.e.* that satisfies time window and vehicle capacity constraints). The process is repeated until all nodes have been serviced and all routes are feasible.

Another technique is a variant on the nearest-neighbor algorithm from the classic TSP[47]. A route is constructed by first picking the closest customer to the depot with an eligible time window. Once the first customer has been serviced, the algorithm searches for the next-closest eligible customer. The algorithm continues until a vehicle reaches capacity, at which point it starts a new route with a new vehicle, or until all nodes have been serviced.

The last major class of route construction algorithms surveyed by [16] are the "sequential insertion heuristics". Routes are initialized with "seed" customers, who are selected based on some criteria, such as the farthest customer from the depot. Customers are then inserted into the seed customer's route until the vehicle capacity is reached or until no more time windows can be met. If any unserviced customers remain, a new seed customer is selected and a new route construction process is initiated. Many variations on the sequential insertion heuristic have been explored, with each variation applying different criteria for the selection of seed and insertion candidates[48, 49].

The second half of the Bräysy and Gendreau survey looks at route improvement methods, wherein an arbitrary route is constructed to initialize the algorithm. The initial tour is then improved, and feasibility conditions satisfied, through an iterative process. As [16] points out, there are four primary considerations in developing a route improvement heuristic: how the initial tour is selected, what criteria are used for making improvements, how improvements are accepted, and the algorithm stopping criteria. The most popular route improvement method, k-opt exchange, was already discussed in Sec. 2.5.1 for solving the Euclidean TSP. When applied to the TSPTW, if more than one route is used in the solution, edges are swapped within individual routes (intra-route swapping) to search for improvements. Variations on k-opt include inter-route swapping[50], swapping of customers[51, 52], and swapping of sets of customers[53].

Related to k-opt exchange, Koskosidis *et al.* propose a "cluster-first, route-second" algorithm[54]. Customers are first divided into clusters with a heuristic clustering algorithm. Each cluster is then serviced by its own route. The total solution is optimized by evaluating the exchange of customer pairs between clusters and updating the solution when lower cost routes are found.

Finally, in 1997 Shaw introduced a Large Neighborhood Search (LNS)[55]. In LNS, a subset of customers are removed from the initial route and then reinserted into the route, with the reinsertion position selected by a branch and bound algorithm. The customers selected for removal are chosen based on similarities in location (how near they are to each other), required service load q, and time windows. While LNS generates competitive TSPTW solutions, it does carry a high computational burden which limits its applicability to problem instances with a low number of customers per route.

The TSPTW has features that make it attractive as a tool for persistent moni-

toring problems. When coupled with multiple vehicles and subtours, it allows some tasks to be accomplished more frequently than others, corresponding to higher priority tasks having shorter and more frequent time windows.

2.5.3 The Weighted TSP (or The Minimum Latency Tour Problem).

What if all cities in the salesman's tour were not of equal importance? The salesman would then want to visit the more important cities earlier in the tour while still minimizing distance traveled to the extent possible. This introduces the TSP with prioritized vertices in what is known as the Weighted TSP (WTSP). Note: In the literature, the WTSP is more commonly referred to as the Minimum Latency Tour Problem (MLTP), but in the interest of maintaining consistent terminology, WTSP will be used here.

2.5.3.1 Exact Algorithms.

Because the WTSP is at least as hard as the TSP[56], there are no polynomial time solutions to the general problem. However, Blum *et al.* provide some exact solutions to special cases in their 1994 conference paper[56]. First, they provide a proof that a depth-first search solution exists when the nodes are vertices of a tree and all edges on the tree have unit length. The second exact solution comes from dynamic programming, where they consider the special case when"a good bound on the number of potential partial solutions" exists. Specifically, they prove that when all points are on a line, dynamic programming can provide a solution to the WTSP in $O(n^2)$. They also show that dynamic programming can produce a solution in $O(n^2)$ if the nodes are vertices of a tree of at most degree 3 (*i.e.* there are at most 3 nodes on the longest path between any two leaves).

More recently, a team from Shu-Te University in Taiwan introduced a hybrid

dynamic-programming/branch-and-bound algorithm to exactly solve the metric space WTSP[57]. They describe their algorithm as "dynamic programming with pruning" and demonstrate it on both random and real-world data sets consisting of node counts from 15 to 23. Their algorithm shows improvements over pure dynamic programming or branch-and-bound approaches and is able to solve instances of up to 26 nodes in about 100 seconds on a personal computer.

2.5.3.2 Approximate Algorithms.

In the same 1994 paper as above, Blum *et al.* provide the first constant-factor approximation for the WTSP in a metric space[56]. They prove an (α, β)-approximator algorithm (an algorithm in which the larger problem solution is stitched together from solutions to the partitioned problem) has an approximation ratio of 144.

Goemans and Kleinberg improved the WTSP approximation ratio using their own (α, β)-approximator algorithm in [58] to 21.55. In their conclusion, the authors cite the k-TSP solution of Garg[59], which can combine with their algorithm to further improve the approximation ratio to 10.78.

The approximation ratio is further reduced to 7.18 using $O(n \log n)$ calls by Archer, Levin, and Williamson[60]. Their method is based on the prize-collecting Steiner tree while performing calls to the Garg k-TSP. While a similar method was used in previous papers, the authors were able to improve the approximation ratio and running time over previous algorithms by taking advantage of special structures within the k-TSP sub-routine.

2.5.3.3 Heuristic Algorithms.

A meta-heuristic algorithm, which the authors call GILS-RVND, brings together three different heuristic algorithms in [61]. The meta-heuristic starts with a greedy

initial tour construction and then improves on the initial tour through node swapping and random perturbations. They use nine benchmark instances of the WTSP, some with up to 1000 customers, to demonstrate "good" performance. They also show that their algorithm finds the known optimal solutions for instances with up to 50 customers.

A group of students from Northeastern University in Shenyang, China proposed a modified version of the ant colony optimization algorithm for solving the WTSP in 2011[62]. By introducing node priority into the ant colony heuristic matrix, they were able to show that their algorithm performed almost as well as the best known approximation algorithms for 13 benchmark instances of the WTSP, though the authors did not present computation time data. One down side to the ant colony approach, is that the algorithm requires extensive exploration of parameter values in order to find the best solutions.

The WTSP is interesting in the context of PISR because it establishes two useful tools: a way to deal with tasks of varying priority and a metric (in the form of latency) for comparing the performance of different algorithms. Like all TSP solutions, however, the WTSP requires that each node be visited exactly once per tour. So while WTSP solutions do account for priority with node visit order, they do not allow for more frequent visits to high priority nodes.

2.5.4 The Dubins TSP.

All formulations and solutions to the TSP and its variants that have been discussed so far assume the traveling agent moves between nodes in a straight path. They do not consider the dynamics of the traveling agent which could limit its ability to follow the generated path solutions. In the case where the agent is a fixed-wing UAV, vehicle dynamics may have a significant effect on the feasibility of a Euclidean TSP solution,

notwithstanding that the Euclidean TSP solution is likely no longer optimal. When the distance between nodes is sufficiently small compared to the UAV turning radius (*i.e.* the distance between nodes is less than four times the turn radius[63, 64]), a more appropriate problem formulation is the Dubins TSP (DTSP). Note that when the inter-node distance is significantly larger than the UAV turn radius, the DTSP solution approaches that of the Euclidean TSP.

There are no exact algorithms for the DTSP. However, the problem of minimum time point-to-point trajectories for Dubins vehicles has been widely studied. Savla, Frazzoli, and Bullo leverage this previous body of work in their heuristic Alternating Algorithm (AA)[65]. The basis of AA is that solving the DTSP requires two steps: determining the order of node visits and assigning headings to the vehicle at each node. The AA starts by finding the optimal Euclidean TSP solution to the set of nodes, which fixes the order of visitation, and then generates a sub-optimal, yet cost-bounded, DTSP tour. The DTSP tour is formed by keeping the odd edges of the Euclidean TSP solution and replacing the straight-line even edges with minimum-length Dubins paths. The original node order from the Euclidean TSP solution is preserved. The authors go on to show that AA has a worst-case bound in terms of the Euclidean TSP optimal solution of $1 + \kappa[n/2]\pi r$, where κ is a constant of value < 2.658, n is the number of nodes, and r is the turn radius of the vehicle.

In [66] and [67], the authors propose an approximation algorithm for the DTSP based on arbitrarily fixing the required heading at all nodes to 0. An asymmetric distance matrix is then calculated based on the $n(n-1)$ Dubins travel distances between all nodes. This asymmetric TSP is then solved using a $\log n$ approximation algorithm. The resulting solution for a fixed heading assignment has an expected tour length within a factor of $\left(1 + \max\left[\frac{8\pi r}{D_{min}}, \frac{14}{3}\right]\right)\log n$, where D_{min} is the smallest Euclidean distance between two nodes. The authors go on to propose a Randomized Heading

(RH) version that is solved in the same fashion and improves expected solutions to be within $\left(1 + \frac{13.58r}{D_{min}}\right) \log n$ of the optimum. The RH algorithm is computationally demonstrated to outperform AA from [65] for problem sizes with $n > 10$[66].

A receding horizon approach is taken by Ma and Castanon in [68] and compared to AA[65] and RH[66]. Ma and Castanon evaluate three different receding horizon algorithms: a two-point (2PA), three-point (3PA), and three-point look-ahead (LAA). The 2PA assumes the initial vehicle heading is given and the destination terminal heading is free. Once the terminal heading is found, it is used as the initial heading in computing the next edge of the solution. The 3PA works in a similar manner, solving the Dubins path through three points and determining the heading for the midpoint and terminal nodes. The terminal node then becomes the initial node for another three-point path. In this way, the 3PA only solves Dubins paths between odd numbered nodes. Finally, the LAA solves a three-point Dubins path to determine the midpoint and terminal headings, but the vehicle only uses the solution until it reaches the midpoint, at which time a new three-point path is calculated with the previous midpoint solution becoming the new start location and heading. Through simulation results, the authors show that LAA outperforms AA, RH, 2PA, and 3PA, both for problems with predetermined node orders and when the node orders must be found by the algorithms. As an interesting aside, the 2PA and AA performed similarly in all trials.

An extension to the work on the RH algorithm in [66] is conducted in [69]. Instead of assigning random headings to nodes, the possible terminal headings to each node are discretized and represented by K nodes clustered around the original node. The Dubins distances are then calculated between pairs of nodes from separate clusters. The final tour is constructed by solving the resulting nK-node asymmetric TSP, which can be done with a heuristic or $\log n$ approximation algorithm. While this

discretization technique greatly increases the node count for even modest levels of discretization, the authors claim that a 100-node DTSP with 5 discretization levels (a 500-node asymmetric TSP) can be solved on a standard laptop in one minute. The authors show Monte Carlo simulation results comparing their K-headings algorithm with $K = [1, 5, 10]$ to AA, RH, and a Dubins implementation of NN. For all levels of discretization, K-headings resulted in the shortest average tour lengths among compared algorithms.

Aside from the approximate algorithms already discussed, heuristic approaches have also been studied. Particle Swarm Optimization (PSO) is applied to the DTSP by Kenefic in [70]. The visit order is determined by solving the Euclidean TSP while the particle headings are initialized using either the AA heuristic from [65] or by taking the average of the entry and exit headings from the Euclidean TSP solution, with the decision of which heading heuristic to use based upon how close vertices are with respect to the vehicle turning radius. Kenefic demonstrates his PSO approach on 10, 20, and 30 vertex graphs with varying amount of spacing between nodes. Due to the authors' use of the AA heuristic in initializing headings, the PSO solution converges to the AA solution as particle density increases.

More recently, in 2012, Yu and Hung proposed a Genetic Algorithm (GA) for solving the DTSP[71]. Like all GAs, it starts with a population of arbitrary solutions; in terms of the DTSP this means a population of randomly selected N-tuples corresponding to node order and heading angles (x_j, y_j, θ_j). Their GA then uses a combination of elitism (passing the best solutions directly into the next generation), roulette wheel (building a tour with random selections of nodes and headings), crossover (combining the solutions of two randomly-selected parents in the previous generation), and mutation (randomly permuting solution elements of some percentage of the population). They demonstrate their GA on graphs with node counts from 5 to

50, having varying degrees of node density, and show that, on average, it outperforms AA and RH.

2.6 Utility Theory

Up to this point, the surveyed literature has been motivated by viewing PISR as a problem in combinatorial optimization. From this perspective, thinking about PISR in terms of the Traveling Salesman Problem is natural. However, we can also think about PISR purely from the agent point of view. From this perspective, PISR can be viewed as an Artificial Intelligence problem. While the field of AI is vast and encompasses a wide range of problem sets and algorithms, one tool in the AI repertoire that is worth exploring is Utility Theory.

In their AI textbook, Russel and Norvig describe Utility Theory simply as a way to "represent and reason with preferences"[72]. Preferences, in turn, describe an agent's desire, or lack thereof, to be in a given state or take a certain action. The agent's preferences are represented with a utility function. In the usual formulation, actions and states either incur a reward (positive utility) or a penalty (negative utility). Once the agent calculates the utility of all possible action-state combinations, if it is rational, it will pursue the option with the highest utility.

Russel and Norvig provide a helpful example. Consider an agent trying to catch a flight. The desired state is to make the flight, which provides a reward, whereas missing the flight incurs a penalty. The available action is how early to leave the house, which implies a corresponding amount of time waiting at the airport. So, while leaving the house 24 hours in advance virtually guarantees making the flight, the long wait at the airport significantly reduces the utility of that decision. Conversely, leaving at the last possible moment may require no wait time, but the probability of missing the flight is unacceptably high. The agent's task, then, is to find the balance

between how early to leave for the airport and how long to wait, *i.e.* the decision with maximum utility.

A closer inspection of the above example sheds some light on the difficulties with utility-based decisions. Qualitatively, the flight catching model above is simple. If one starts to think about the implementation of such a model, however, the challenges become clear. What probability function should be used to determine the odds of making a flight? How large should the make/not-make reward be compared to the wait-time penalty? If the reward for making the flight is too large, then no amount of waiting will ever cancel out the reward and the best decision will always be to leave the house at the soonest possible moment, which is obviously undesirable.

Due to the difficulties addressed in the above example, *it is a significant challenge to design an appropriate utility function for agent decision making.* Still, utility functions offer attractive advantages over combinatorial optimization. They do not rely on the convergence of an algorithm and are not subject to computational complexity constraints as the number of agents or tasks increases. Furthermore, there is no need for a central planner to conduct route assignments since, in the simplest form at least, each agent makes its own decision about what to do. For these reasons, Utility Theory is worth exploring as an approach to task selection in PISR and is discussed in detail in Ch. III.

2.7 Summary

PISR has received several treatments in the literature. Most approaches formulate the problem as a type of TSP or use a utility function.

The TSP is computationally hard, especially with the addition of time windows, node priorities, asymmetry, and vehicle dynamics. The WTSP and TSPTW allow for the incorporation of task priority into the TSP formulation and the Dubins TSP

provides a good model when a UAV is the vehicle and tasks are close enough together such that vehicle dynamics become important. If the distance between tasks is large compared to the vehicle turning radius, however, the Euclidean TSP is sufficient.

Utility functions for agent decision making are fast, but lack the mathematical rigor and optimal guarantees provided by TSP. Crafting a useful utility function is difficult and requires selecting a good heuristic based on state variables. It may also be necessary to fine tune the utility function to arrive at the desired vehicle behavior.

Both TSP and utility methods allow for the incorporation of multiple vehicles. With TSP, however, the computational complexity of developing a solution grows rapidly with an increasing number of vehicles and tasks. Furthermore, computation and assignment of solutions must be done by a centralized planner, which places a premium on stable communication links; a luxury not always available in an operational environment. It is feasible that each vehicle could compute its own TSP solution based on estimates of the environment state. This, however, requires an accurate initial state synchronization and/or frequent exchanges of state information. The utility approach requires only minimal information flow, pertaining to which vehicles are accomplishing which tasks.

One significant advantage of TSP over utility functions is the guarantee of a performance bound. With a TSP solution, it is possible to get a worst-case bound on revisit times to any given task. It is also possible for the operator to easily locate a vehicle and predict where it will be at a given time, since the task visit sequence of each vehicle is predetermined.

When it comes to scalability and adaptability, the utility function approach has the advantage. Since all TSP methods are computationally hard, the algorithms may take a long time to converge, if they converge at all. Calculating utility values, on the other hand, is extremely fast. For example, if tasks or vehicles are added or removed, a new

(potentially complex) TSP solution must be calculated and assignments redistributed to vehicles. Using a utility function, the vehicles need only update their state variables to reflect the new environment and future decisions will take the new information into account. A summary of attributes for the TSP and utility function methods for task selection is presented in Table 2.1.

Table 2.1. The utility approach for task selection in PISR has many advantages over TSP

	TSP	Utility
Extension to Multi-Vehicle?	Yes	Yes
Architecture	Centralized	Decentralized
Computational complexity	High	Low
Guaranteed revisit times?	Yes	No
Communication demand	High	Low
Scalability	Low	High
Adaptability	Low	High

Due to the desire for a robust PISR solution, this research will pursue a utility function approach to task selection in PISR. The aim is to assess how much is lost in terms of performance by sacrificing the rigor of a TSP solution, and in turn, justifying that loss with gains in ease of implementation, scalability, and adaptability.

III. Methodology

3.1 Overview

In this chapter we discuss the general problem of task selection in PISR and define our performance criteria. We also formally define our proposed task selection method, the Maximal Distance Discounted and Weighted Revisit Period (MD^2WRP) utility function and provide its derivation. Finally, we provide an overview of our custom simulation environment and define our methodology for gathering the results discussed in Ch. IV. Detailed documentation of the simulation code is provided in Appendix A.

3.2 Performance Measures for PISR

In order to make comparisons between the various task selection methods for PISR, we define a performance metric based on task age, which is the length of time elapsed between consecutive visits to a task. Age-based metrics are a common means of measuring performance in the literature on PISR and similar problems[4, 3, 5]. Specifically, we use *average weighted latency* as the primary objective function which uses age as a basis, but multiplies each task age by a priority, which is a user-provided value. The average weighted latency, $\bar{L}$, is defined by,

$$\bar{L} = \frac{\sum_{k=0}^{m} \sum_{j=1}^{n} p(j) T_k(j)}{\Delta t \cdot m} \tag{3.1}$$

where k is the time step, m is the total number of time steps, n is the number of tasks, $p(j)$ is the priority of individual task j, $T_k(j)$ is the age of j at time step k, and Δt is the size of the time step. $\bar{L}$ is calculated *ex post facto* to determine the performance of the mission.

Equation 3.1 serves as a good metric for PISR because it considers both the transient and steady-state phases of latency development, which we show to be an important consideration in determining PISR performance. In this way, the $\bar{L}$ metric captures how vehicles will perform when viewing the mission as a whole, from start to finish.

While Eq. 3.1 serves as a quantitative criterion, we also wish for a solution that has certain qualitative attributes. In the spirit of autonomy, we desire a task selection method that is *adaptable*, without human intervention, to a wide range of scenarios. It should be able to accommodate single or multiple vehicles and numerous task configurations, allowing tasks to have varying priorities. The solution should be *scalable*; effective when the number of tasks or vehicles is small or large, whether tasks are in close proximity or far apart. Lastly, the solution should be *robust*, that is, resilient to the addition or removal of tasks and vehicles mid-mission, even if a vehicle loss occurs unexpectedly.

3.3 The Maximal Distance Discounted & Weighted Revisit Period

While we argue that the task selection utility functions from Ch. II are better suited for use in autonomous PISR applications than the TSP approaches, they still lack some features we desire or suffer from drawbacks of their own. Only the function from Ruan[73] allows for individual task weighting. The other functions all assume tasks of identical weight. For multiple vehicles, most require every vehicle to be aware of the location of all other vehicles throughout the mission, which may not be achievable. As for the function in Ruan, upon which we base our solution, making decisions solely on weight and future age is still troublesome. While we want to reward the vehicle for visiting tasks that have not been visited in a long time, we do not want to provide an incentive to travel to tasks that are far away. In other

words, we want the agent to spend as much time as possible accomplishing tasks, not traveling between them. Therefore, we propose a utility function that still uses future age as a basis for task selections, but that also discounts the reward received for tasks that are further away.

To this end, the Maximal Distance Discounted & Weighted Revisit Period (MD^2WRP) was proposed by Kalyanam of AFRL[9]. It is,

$$V = \max_j \left[e^{-\beta t_{ij}} w_j (T_j + t_{ij}) \right], \qquad \forall j \in \{1, \ldots, n\} \tag{3.2}$$

where V is the value the agent receives for accomplishing the selected PISR task, t_{ij} is the time to travel from current task i to candidate task j, β is a parameter that discounts utility based on travel time to a task, w_j is a weight parameter associated with task j, and T_j is the time since j was last visited, also referred to as the *age* of j. We add t_{ij} to T_j because we wish for the agent to consider the *future age* of the task, that is, what the age will be at the time of arrival. For the purposes of the derivation, we assume an agent with unit velocity, such that $t_{ij} = d_{ij}$, where d_{ij} is the distance between tasks i and j, selected from the user-provided task distance matrix, D. In simulations conducted later, we will specify a non-unit velocity for the agents. The benefits of using Eq. 3.2 in task selections for PISR will be explored through simulation. Results and analysis are presented in Ch. IV.

3.3.1 Derivation.

The MD^2WRP value function in Eq. 3.2 is a myopic policy derived from a dynamic programming formulation of PISR, which is presented here in a form adapted from the work of Kalyanam in [9].

Let each task, $j \in \{1, \ldots, n\}$ have an associated weight, $w_j > 0$. A vector, T, with entries $T(j)$, holds the time elapsed since task j was last visited by an agent. We

call T the vector of task ages. The system state can then be defined by the current location of the agent, $i \in \{1 \dots n\}$, and the age of all tasks, or $s = (i, T) \in S$, with S the set of all possible system states. Note that when the agent is at task i, $T(i) = 0$; in other words the age of task i is reset when it is visited by an agent. A control policy for the agent is defined as a mapping from the state $s = (i, T)$ to the set of control options: $U_i = \{1, \dots, i-1, i+1, \dots, n\}$. Note U_i does not allow the agent to select the task at which it is currently located. For an agent with unit velocity, the time required for the agent to travel from task i to target j can be defined as $d(i, j) > 0$ that satisfies the triangle inequality.

Now, for an agent in state $s = (i_0, T)$ that chooses to visit task $i_1 \in U_i$, the new state, $\bar{s}$ is a function of the current state, s and the selected task, i_1,

$$\bar{s} = f(s, i_1). \tag{3.3}$$

Or more precisely,

$$\bar{s} = (i_1, \bar{T}) \tag{3.4}$$

with $\bar{T}$ defined by,

$$\bar{T} = T + d(i_0, i_1)\mathbb{1} \qquad i_1 \in U_i, \bar{T}(i_1) = 0. \tag{3.5}$$

where $\mathbb{1}$ is a vector of ones in $\Re^n$.

Taken together, Eqs. 3.3-3.5 define the agent state transition as being a function of the current state, s, and selected task, i_1, where the new state, $\bar{s}$, has the vehicle located at task i_1 and the new task ages, $\bar{T}$, are updated as their previous age plus the distance the agent traveled during the state transition, $d(i_0, i_1)$, with the exception that the age of task i_1 is now zero.

From Ruan[73], we associate an immediate reward with state s and control $i_1 \in U_i$,

$$r(s, i_1) = w_{i_1}[T(i_1) + d(i_0, i_1)]. \tag{3.6}$$

This means the agent receives a reward for visiting task i_1 that is proportional to the age of i_1 *at the projected visit time*. The weight w_{i_1} is a parameter.

Next we establish a control policy, π, whose input is the current state and output is the task to select, i_1,

$$i_1 = \pi(s). \tag{3.7}$$

If the initial system state is $s_0 = (i_0, T_0)$, then the result of implementing π is a sequence of states: $s_1, s_2, \ldots$ with corresponding arrival times to tasks i_1, i_2 of $t^\pi_{i_1}, t^\pi_{i_2}, \ldots$. Note, the time the agent visits the first targeted task, $t^\pi_{i_1}$, is simply the travel time between the starting task i_0 and the task selected by policy π (that is, i_1) from the initial state s_0,

$$t^\pi_{i_1} = d(i_0, i_1) = d(i_0, \pi(s_0)) \tag{3.8}$$

and the time of visit to any subsequent task is the time of visit to the previous task plus the distance traveled,

$$t^\pi_{i_{k+1}} = t^\pi_{i_k} + d(i_k, i_{k+1}) = t^\pi_{i_k} + d(\pi(s_{k-1}), \pi(s_k)). \tag{3.9}$$

Therefore, combining Eqs. 3.6 and 3.9 while starting at state s_0 and following policy π, we can establish the *infinite horizon* cumulative discounted reward for the agent,

$$V^\pi(s_0) = \sum_{k=0}^{\infty} e^{-\beta t^\pi_{i_{k+1}}} w_{\pi(s_k)} \left[T(\pi(s_k)) + d(\pi(s_{k-1}), \pi(s_k))\right] \tag{3.10}$$

where, $V^\pi(s_0)$ is the total reward the agent receives when starting from state s_0 and following policy π, β is a travel distance discount parameter, $t^\pi_{i_{k+1}}$ is the time of visit to task i_{k+1}, $w_{\pi(s_k)}$ is the weight of the selected task, $T(\pi(s_k))$ is the age of the selected

task, and $d(\pi(s_{k-1}), \pi(s_k))$ is the distance between the current task and the selected task.

Ideally, we would like to compute a control policy that maximizes Eq. 3.10, that is, to find π such that,

$$V(s) = \max_{\pi} V^{\pi}(s), \qquad \forall s \in S. \tag{3.11}$$

We can represent this optimal value function, $V(s)$, using a dynamic programming recursion. Note that in Eqs. 3.12 and 3.13 to follow, the first term in the infinite horizon reward of Eq. 3.10 has been expanded, and that $V(f(s, i_1))$ represents the value of all future states.

$$V(s) = \max_{i_1 \in U_i} \{e^{-\beta d(i_0, i_1)} w_{i_1} \left[T(i_1) + d(i_0, i_1)\right] + V(f(s, i_1))\} \tag{3.12}$$

with the optimal policy in Eq. 3.12 determined by the maximizing control,

$$u(s) = \arg\max_{i_1 \in U_i} \{e^{-\beta d(i_0, i_1)} w_{i_1} \left[T(i_1) + d(i_0, i_1)\right] + V(f(s, i_1))\}. \tag{3.13}$$

Due to the curse of dimensionality[74], it is not in general possible to solve for the optimal control in Eq. 3.13. Instead, for the sake of practical application it makes sense to employ a myopic policy based on Eq. 3.13. We can do this with a zeroth order approximation, which effectively means ignoring the value of all future states, that is, setting $V(f(s, i_1)) = 0$. The resulting heuristic control policy is,

$$\bar{u}(s) = \arg\max_{i_{k+1} \in U_i} e^{-\beta d(i_k, i_{k+1})} w_{i_{k+1}} \{T(i_{k+1}) + d(i_k, i_{k+1})\} \tag{3.14}$$

and the associated value function is,

$$\bar{V}(s) = \max_{i_{k+1} \in U_i} e^{-\beta d(i_k, i_{k+1})} w_{i_{k+1}} \{T(i_{k+1}) + d(i_k, i_{k+1})\} \tag{3.15}$$

which is the same value function introduced in Eq. 3.2, noting that Eq. 3.2 is modified for simpler notation and generalized to allow for non-unit agent velocity.

3.3.2 Normalization.

One implementation challenge of MD^2WRP is the sensitivity of the distance discounting factor, represented by the exponential term in Eq. 3.2, to changes in the task geometry (that is, changes to the distance matrix, D). If travel time, t_{ij}, is measured in seconds (s), then we require the units of β to be $1/s$ in order to maintain a dimensionless exponent. Therefore if the magnitudes of t_{ij} were to change due to the application of a scalar multiplier to D, the magnitude of β would also need to change in order to maintain the same vehicle behavior (that is, the same task visit sequence). This makes selecting the MD^2WRP parameters very difficult.

A better way is proposed to implement Eq. 3.2 by normalizing according to the largest value in the distance matrix, $d_{ij,max}$, which becomes $t_{ij,max}$ when used with a constant velocity agent. The normalized version then becomes,

$$V = \max_j \left[e^{-\beta \frac{t_{ij}}{t_{ij,\max}}} w_j \frac{(T_j + t_{ij})}{t_{ij,\max}} \right], \qquad \forall j \in \{1, \ldots, n\}. \tag{3.16}$$

In this way, β is now a non-dimensional parameter whose value does not depend on the magnitude of t_{ij}. Additionally, we will always have $0 < \frac{t_{ij}}{t_{ij,\max}} \leq 1$, under the Euclidean travel assumption.

3.3.3 Using MD^2WRP to Minimize Latency.

In PISR missions, we wish to minimize average weighted latency, $\bar{L}$, under a given distance matrix, D. Unfortunately, directly solving for a tour to minimize $\bar{L}$ is a variant of the TSP and at least as hard as the TSP[56]. As such, no polynomial time solutions exist to the general problem. However, from [75], minimizing the time since tasks were last visited, which we can incentivize the agent to do with MD^2WRP rewards, is equivalent to minimizing the total task waiting time, or latency, of the system. We demonstrate the effectiveness of this strategy in Sec. 4.2.3, where we compare its performance to common TSP methods.

There is one key difference to note between the derivation of MD^2WRP and the use of MD^2WRP to minimize $\bar{L}$. In deriving MD^2WRP in Sec. 3.3.1, the objective was to maximize the total distance discounted value, $V(s)$, the agent received over an infinite number of task visits. We did this by selecting the optimal task visit order under a given D *assuming* β *and* $\mathbf{w}$ *were known.* This provided us with the basis for establishing the MD^2WRP policy for making task selections. However, in the context of minimizing $\bar{L}$, we must choose the β and $\mathbf{w}$ that will yield a task visit order with the lowest $\bar{L}$, depending on the mission specified in D.

3.4 Simulation Environment (PUMPS)

The Persistent Unmanned Monitoring and Patrolling Simulation (PUMPS) tool was developed by the author from scratch to fully meet the needs of this research. It is an object-oriented, modular, discrete-event simulation written in Python. The modularity allows the user to mix-and-match from among several types of routing, pathing, and communication for each vehicle. Because the primary focus of this research is on task selection for PISR, the simulation environment is discrete and only considers moments in time when a vehicle is collocated with a task, which covers

all key decision points (*i.e.* task selections) for the mission.

The tool is written primarily in Python 2.7.12, relying heavily on NumPy (from the SciPy scientific computing stack[76]) for the underlying data structure. It also takes advantage of third-party open-source software to handle more complex functions. A C library by Andrew Walker is used for generating optimal Dubins paths[77], which is accessed via a Python wrapper written by the same author[78]. To generate paths around polygon objects, the tool makes function calls to the TriPath Toolkit[79] (now called Triplanner), which is software based on Kallmann's academic publications for rapid, locally optimal trajectory generation with constraints[80, 81].

3.4.1 Architechture.

PUMPS is object-oriented and defines classes for each of the major components of a PISR mission. There are six classes: *Vehicle*, *Task*, *Routing*, *Pathing*, *Communication*, and *Database*. Each class along with its subclasses, attributes, and methods are described below.

3.4.1.1 The Vehicle Class.

Vehicle objects are derived from the *Vehicle* class, which is the primary class. Simulation events revolve around *Vehicle* objects. *Vehicle* objects use and act upon objects derived from other classes to simulate a full PISR mission. The attributes and methods of the *Vehicle* class are summarized in Fig. 3.1.

Below is a brief description of each attribute and method for *Vehicle* objects:

- *_indexer*. [int] A private attribute of each *Vehicle* object, primarily used for easy indexing of vehicles in various arrays used throughout the simulation.
- *ID*. [int] Each vehicle is assigned a unique ID in increments of 100. This is used for data presentation to make results easier to read for the user.

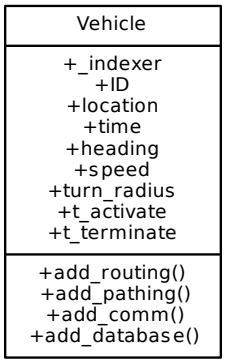


Figure 3.1. The attributes and methods of the *Vehicle* class in PUMPS.

- *location*. [Task object] The task at which the vehicle is currently located.
- *time*. [double] The time at which the vehicle arrived at its current location, in s, based on the time elapsed since $t_0 = 0$.
- *heading*. [double] The vehicle's heading just prior to arrival at its current location, in radians.
- *speed*. [double] The vehicle's constant velocity in m/s.
- *turn_radius*. [double] The vehicle's minimum turning radius in m, calculated based on *speed* and maximum bank angle, which are supplied by the user during setup.
- *t_activate*. [double] Time at which the vehicle enters the simulation, in s.
- *t_terminate*. [double] Time at which the vehicle exits the simulation, in s.
- *add_routing()*. Instantiates a *RoutingFactory* object and passes the routing preferences provided by the user at setup. The *RoutingFactory* object returns

the appropriate routing module and adds it to the *Vehicle* object. The routing module is accessed via the *Vehicle* object with *self.routing*.

- *add_pathing()*. Instantiates a *PathingFactory* object and passes the pathing preferences provided by the user at setup. The *PathingFactory* object returns the appropriate pathing module and adds it to the *Vehicle* object. The pathing module is accessed via the *Vehicle* object with *self.pathing*.

- *add_comm()*. Instantiates a *CommunicationFactory* object and passes the communication preferences provided by the user at setup. The *CommunicationFactory* object returns the appropriate communication module and adds it to the *Vehicle* object. The communication module is accessed via the *Vehicle* object with *self.comm*.

- *add_database()*. Adds a database module to the *Vehicle* object, which contains the database items requested by the user during setup. It is accessed via the *Vehicle* object with *self.database*.

3.4.1.2 The Task Class.

A *Task* object is created for each PISR task that may be visited during a mission. In the current version, PUMPS can only handle point-search tasks, that is, tasks that can be represented by a single point on the map. The *Task* class diagram is in Fig. 3.2.

Below is a brief description of each attribute for *Task* objects:

- *ID*. [int] Each task is assigned a unique ID beginning at 1, in increments of 1. The order of ID assignment is based upon the order in which the tasks were listed by the user during setup.

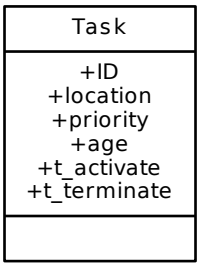


Figure 3.2. The attributes of the *Task* class in PUMPS.

- *location*. [(double, double)] The location of the task on the map in (x,y) coordinates, in m.
- *priority*. [int] The priority of the task. A higher priority results in faster accumulation of latency. The default priority is 1, with a higher integer value implying the task is more important.
- *age*. [double] The true age of the task, that is, the time elapsed since the task was last visited by any agent, in s.
- *t_activate*. [double] Time at which the task appears as eligible for vehicle visits, in s.
- *t_terminate*. [double] Time at which the task becomes ineligible for vehicle visits, in s.

3.4.1.3 The Routing Class.

Routing refers to the method by which vehicles select their next task. Each vehicle in a simulation run can utilize a different type of routing, which is specified by the

user during setup. The parent *Routing* class is a Python metaclass, which defines the structure of child classes but cannot be instantiated itself. Currently, there are two possible types of *Routing* objects: MD^2WRP and *Manual*. MD^2WRP routing selects the next task based on the MD^2WRP value function described in Sec. 3.3 while *Manual* allows the user to specify a static task visit sequence for the vehicle. The *Routing* class diagram is depicted in Fig. 3.3.

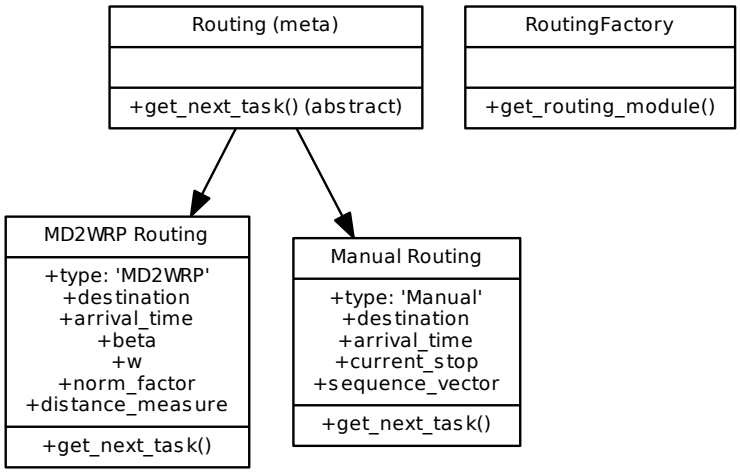


Figure 3.3. The attributes and methods of the *Routing* classes in PUMPS.

To add a routing type to a vehicle, a *Vehicle* object instantiates a *RoutingFactory* object and passes to it the routing preferences supplied by the user during setup. The *RoutingFactory* then uses the *get_routing_modules()* method to select the appropriate *Routing* subclass, instantiate a *Routing* object, and "load" it onto the *Vehicle* object (the vehicle's routing attributes and methods can then be accessed via the *Vehicle* object with *self.routing*).

Description of MD^2WRP *Routing* attributes and methods:

- *type*. [string] The type of routing, used primarily for the purpose of displaying

data and results to the user.

- *destination*. [Task object] The task to which the vehicle is currently headed.
- *arrival_time*. [double] The time the vehicle will arrive at the destination task, in s.
- *beta*. [double] The MD^2WRP distance discount factor.
- *w*. [$1 \times n$ double vector] The MD^2WRP weight for each task.
- *norm_factor*. [double] The value $t_{ij,\max}$ for computing task values with the normalized version of the MD^2WRP value function (see Sec. 3.3.2). Calculated as the largest value in the task distance matrix divided by the vehicle's constant velocity. Units of s.
- *distance_measure*. [string] Specifies how the distance between tasks is measured. Values: *Euclidean, Dubins, Tripath*.
- *get_next_task()*. Evaluates the MD^2WRP function using the specified parameters for all candidate tasks and returns the *Task* object with the highest value as the new vehicle destination.

Description of *Manual Routing* attributes and methods:

- *type*. [string] The type of routing, used primarily for the purpose of displaying data and results to the user.
- *destination*. [Task object] The task to which the vehicle is currently headed.
- *arrival_time*. [double] The time the vehicle will arrive at the destination task, in s.

- *current_stop*. [int] The index of the task at which the vehicle is currently located, according to the *sequence_vector*.

- *sequence_vector*. [$1 \times n$ int vector] The manually determined task visit sequence assigned to the vehicle.

- *get_next_task()*. Returns a *Task* object of the next task in *sequence_vector*, based on the vehicle's current location in the sequence, as the new vehicle destination.

3.4.1.4 The Pathing Class.

Pathing refers to how the vehicle travels between its current location and the destination task. As with routing, each vehicle can implement a different type of pathing. The *Pathing* class also implements a parent metaclass and offers three instantiable subclasses: *Euclidean_Pathing*, *Dubins_Pathing*, and *Tripath_Pathing*. *Euclidean_Pathing* is simple point-to-point travel, where the distance traveled is exactly equal to the Euclidean distance between the current task location and the destination task. *Dubins_Pathing* takes into account the vehicle's kinematic constraints (*i.e.* minimum turn radius) and current heading to generate the trajectory between tasks. *Tripath_Pathing* generates trajectories around polygon obstacles and is intended for scenarios where no-fly zones are in effect. The *Pathing* class diagram is in Fig. 3.4.

The *Pathing* class uses the same "factory" construct as the *Routing* class to generate and add a pathing module to each vehicle. Similarly, once loaded, *Pathing* attributes and methods can be accessed via the *Vehicle* object with *self.pathing*).

Description of *Euclidean_Pathing* attributes and methods:

- *type*. [string] The type of pathing in use.

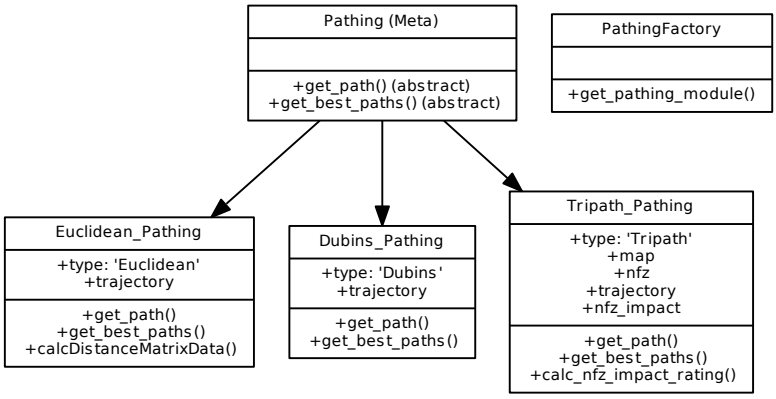


Figure 3.4. The attributes and methods of the *Pathing* classes in PUMPS.

- *trajectory*. [array of two (x,y) coordinates] Stores the vehicle's current trajectory to the destination task. For *Euclidean_Pathing*, this is simply the (x,y) coordinates of the current location and the destination task.

- *get_path()*. Provides the Euclidean path between two *Task* objects, in m.

- *get_best_paths()*. Provides the Euclidean distance to all candidate tasks given the vehicle's current location, in m.

- *calcDistanceMatrixData()*. A helper method used during simulation initialization to determine the longest and average values in the Euclidean distance matrix, in m.

Description of *Dubins_Pathing* attributes and methods:

- *type*. [string] The type of pathing in use.

- *trajectory*. [variable length array of (x,y) coordinates] Stores the vehicle's current Dubins trajectory to the destination task. Each (x,y) pair represents a

segment of the discretized Dubins path. Discretization step-size can be adjusted inside the Dubins calculator function[78].

- *get_path()*. Provides the optimal Dubins path between two *Task* objects, in m.
- *get_best_paths()*. Returns the optimal Dubins paths to all candidate tasks given the vehicle's current location, in m.

Description of *Tripath_Pathing* attributes and methods:

- *type*. [string] The type of pathing in use.
- *map*. [string] The name of the current task map. Used to tell the Tripath Toolkit which map to use in performing trajectory calculations.
- *nfz*. [int] Tells the Tripath Toolkit which no-fly zone (NFZ) instance to use in performing trajectory calculations. (Note: NFZs must be pre-loaded into Tripath Toolkit separately.)
- *trajectory*. [variable length array of (x,y) coordinates] Stores the vehicle's current trajectory to the destination task, taking into account obstacle avoidance provided by Tripath. Each (x,y) pair represents a segment of the vehicle path provided by Tripath Toolkit.
- *nfz_impact*. [double] Stores the NFZ "Impact Ratio". See Sec. 4.3.2 for a description of the Impact Ratio.
- *get_path()*. Provides the obstacle avoidance trajectory from Tripath between two *Task* objects, in m.
- *get_best_paths()*. Returns the obstacle avoidance trajectory from Tripath to all candidate tasks, given the vehicle's current location, in m.
- *calc_nfz_impact_rating()*. Returns the Impact Ratio of the NFZ.

3.4.1.5 The Communication Class.

There are three types of vehicle communication in PUMPS and it is not necessary for every vehicle to utilize the same type. If *No Communication* is used, the vehicle does not transmit the tasks it has accomplished or its destination information, but it can still receive communications from other vehicles and make decisions based on that information. *Broadcast Completions* sends other vehicles the task that was just serviced along with a timestamp. *Broadcast Destinations* shares with other vehicles the task that was just serviced, a timestamp, the next task the vehicle will visit, and its anticipated arrival time. Similar to the *Routing* and *Pathing* classes, the *Communication* class uses a factory object to generate and assign the appropriate communication module to each vehicle during initialization, based on user input during setup. *Communication* attributes and methods are accessed via the *Vehicle* object with *self.comm*. The *Communication* class diagram is shown in Fig. 3.5.

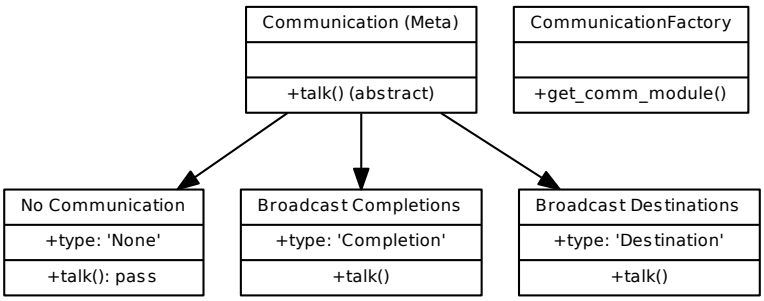


Figure 3.5. The attributes and methods of the *Communication* classes in PUMPS.

Description of *No Communication* attributes and methods:

- *type*. [string] The type of communication in use.
- *talk()*. A Python *pass* command, since the vehicle does not send data to other

vehicles.

Description of *Broadcast Completions* attributes and methods:

- *type*. [string] The type of communication in use.
- *talk()*. Updates the age of the task that was just serviced in *vehicle.database.age_tracker* of all other *Vehicle* objects.

Description of *Broadcast Destinations* attributes and methods:

- *type*. [string] The type of communication in use.
- *talk()*. Updates the age of the task that was just serviced in *vehicle.database.age_tracker* of all other *Vehicle* objects. Also updates *vehicle.database.vehicle_tracker* of all other vehicles to reflect the sending vehicle's destination task and projected arrival time.

3.4.1.6 The Database Class.

The *Database* class diagram is in Fig. 3.6. *Database* objects consolidate all data items that *Vehicle* objects track into a single module. In the current version of PUMPS, all vehicles track the same data using the mandatory attributes *age_tracker* and *vehicle_tracker*. However, adding optional *Database* attributes and methods is possible, if it is desired for vehicles to track more than just task ages and the activity of other vehicles. As with other functional modules, each vehicle could implement a different type of *Database* object.

Description of *Database* attributes:

- *age_tracker*. [$1 \times n$ double vector] The task ages as tracked locally by the vehicle. Since the vehicle only knows a task has been serviced when either it

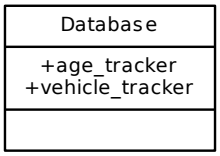


Figure 3.6. The attributes of the *Database* class in PUMPS.

accomplishes the task or receives a communication from another vehicle, the true age of a task is not necessarily the age reflected in *age_tracker*.

- *vehicle_tracker*. [$k \times 2$ mixed vector] Stores the destination task and projected arrival time of every other vehicle. *vehicle_tracker* is only updated if at least one vehicle is using the *Broadcast Destinations* communication mode, otherwise, task selection are made without considering the activity of other vehicles.

3.4.2 Data Flow and Algorithms.

In this section, we describe in broad terms how data flows through PUMPS from setup, to initialization, into the main simulation loop, and finally to the output of results. We also describe two of the more complex simulation algorithms in detail.

3.4.2.1 Simulation Setup.

For each simulation scenario, a trade configuration file must be supplied to PUMPS in the form of a Python Pickle. It is possible to run multiple trades in a row, as PUMPS will automatically execute a simulation run for each trade file in the working directory. The trade file pickle must contain the fields and formatted values as described below in order to successfully initialize:

- *'tradeID'* (int) A value to identify trades. Useful when multiple trades are conducted in a batch. (Ex: 1000)

- *'sim_length'* (1x2 list) Determines the length of the simulation. A simulation can be run until a given number of tasks have been visited or until a simulation time has been reached. The first entry in the list is the number of task visits and the second is the end simulation time. For example, if running 100 task visits the first entry in the list is an integer and the second entry is infinite ([100, float('inf')]). If running for a simulation time of $20,000s$, the first entry is infinite and the second entry is a float value ([float('inf'), 20000.0]). Alternatively, both entries can be set and the simulation will terminate at whichever condition is met first.

- *'task_geometry'* (string) A string to identify which task configuration to utilize from the *generateMapCoordinates.py* file, which stores the task location information for different scenarios along with a matching string ID.

- *'priorities_vector'* (1xn np array) An array of n float values, one for each task, to specify the task priorities. Priority determines how quickly tasks accumulate latency. (Ex: [1 1 1 1 3 1 1 2 1 1])

- *'init_ages_vector'* (1xn np array) An array of n float values, one for each task, that specifies the initial age of each task, in s. For most scenarios, all task ages start at zero, but it may be desired to "seed" the task ages to some non-zero value. (Ex: [0 0 0 100 250 0 0 0 0 0])

- *'task_activation_times_vector'* (1xn np array) An array of n float values, one for each task, that specifies the simulation time at which each task becomes active, in s. Inactive tasks will not be considered by vehicles during the routing

process. If all tasks will be active from the beginning, all values should be zero. (Ex: [0 0 0 1000 2500 0 0 0 0 0])

- *'task_termination_times_vector'* (1xn np array) An array of n float values, one for each task, that specifies the simulation time at which each task is terminated (that is, becomes inactive), in s. If all tasks will be active for the entire simulation, all values should be set to *float('inf')*.

- *'init_locations_vector'* (1xk np array) An array of k integer values, one for each vehicle, that specifies the task at which each vehicle will begin the simulation. (Ex: [1 1 3])

- *'init_headings_vector'* (1xk np array) An array of k float values, one for each vehicle, that specifies the initial heading of each vehicle in degrees. (Ex: [90 0 270])

- *'veh_speeds_vector'* (1xk np array) An array of k float values to specify the constant speed of each vehicle in m/s. (Ex: [20 25 30])

- *'veh_bank_angles_vector'* (1xk np array) An array of k float values to specify the maximum bank angle of each vehicle in degrees. (Ex: [30 30 45])

- *'veh_activation_times'* (1xk np array) An array of k float values to specify at what simulation time, in s, each vehicle becomes active. Inactive vehicles are ineligible to visit tasks until their activation time has been reached. An activation time of zero implies the vehicle is active at the start of the scenario. (Ex: [0 0 1000])

- *'veh_termination_times'* (1xk np array) An array of k float values to specify at what simulation time each vehicle is terminated (that is, becomes inactive),

in s. If a vehicle is to remain active for the entire simulation, use a value of *float('inf')*. (Ex: [float('inf') float('inf') 5000])

- *'routing_type'* (string) The type of routing for each vehicle. Routing refers to how vehicles select tasks. Current options are *MD2WRP* or *Manual*. *Manual* allows the user to specify a static task visit sequence for each vehicle. [Note: In the version of PUMPS in Appendix A, all vehicles must use the same type of routing. The capability for each vehicle to use a different routing type is planned.]

- *'beta'* (float) A float value specifying β for MD^2WRP routing. If MD^2WRP is not used, this value should be an empty list, []. [Note: In the version of PUMPS in Appendix A, all vehicles must use the same β but the capability for each vehicle to use a different β is planned.]

- *'ws_vector'* (1xn np array) An array of n float values to specify the MD^2WRP weight of each task. If MD^2WRP is not used, this value should be an empty list []. (Ex: [1 1 1 3 3 1 3 1 1 1]) [Note: In the version of PUMPS in Appendix A, all vehicles must use the same $\mathbf{w}$ but the capability for each vehicle to use different $\mathbf{w}$'s is planned.]

- *'distance_measure'* (string) A string to define the type of distance measurement to use in calculating travel times between tasks. Current options are *Euclidean* and *Dubins*. *Euclidean* will calculate travel times based off of the task distance matrix, regardless of the actual vehicle dynamics. *Dubins* will use the vehicle's min turning radius to calculate travel times. If the vehicle's max bank angle is 90deg, travel times are equal to those calculated by *Euclidean*. [Note: In the version of PUMPS in Appendix A, all vehicles must use the same type of distance measurement, but the capability for each vehicle to use a different

distance measure is planned.]

- *'tours_vector'* (list of k lists) A list containing k lists of task visit sequences, to be used with the *Manual* routing type. When the vehicle has visited every task in the sequence, it will start over. If *Manual* routing is not used, the value should be an empty list, []. (Ex: [[1 2 3], [4 5 6], [7 8 9 10]])

- *'veh_start_index_vector'* (list of k lists) A list containing k lists of integers, to be used with the *Manual* routing type. Specifies the *sequence index* where the vehicle will begin, not a task number. The corresponding task number of the specified sequence index must match the initial vehicle location specified in *init_locations_vector*. (Ex: [[0], [0], [1]]).

- *'pathing_type'* (list of lists) The type of pathing for each vehicle. Pathing refers to how vehicles travel between tasks. Current options are *Euclidean*, *Dubins*, and *Tripath*. With *Euclidean*, vehicles move between tasks in a straight line. *Dubins* enforces the vehicle min turning radius. *Tripath* allows vehicles to travel between tasks while avoiding no-fly zones. Use of *Tripath* requires additional arguments to specify the task geometry (a string) and no-fly zone shape.(an integer). (Ex: [['Euclidean']] or [['Tripath', 'clusters', 1]]). [Note: In the version of PUMPS in Appendix A, all vehicles must use the same type of pathing, but the capability for each vehicle to use its own is planned. Additionally, the use of *Tripath* requires additional setup of third-party software which is not covered here.]

- *'comm_mode'* (list of strings) The type of communication vehicles will use. Current options are *None*, *Completion*, and *Destination*. *Completion* means vehicles only share which task they have just completed and at what time. *Destination* means vehicles share the task they have just completed, the current time, their

next task, and what time they will arrive at the next task. (Ex: ['Destination']) [Note: In the version of PUMPS in Appendix A, all vehicles must use the same type of communication but the capability for each vehicle to use a different type is planned.]

- *'database_items'* (list of strings) The items that each vehicle will track during the simulation. This category is currently static with two mandatory items, *Age_Tracker* and *Vehicle_Tracker*. *Age_Tracker* is how the vehicle tracks the age of each task based on the information it has. *Vehicle_Tracker* is how the vehicle tracks what the other vehicles are doing and is used in calculating task ages from communicated data. (Ex: ['Age_Tracker', 'Vehicle_Tracker'])

The fields and values of the trade file as described above can be generated using a custom setup script. There should be one trade file for every scenario the user wishes to run.

3.4.2.2 Initialization.

The main simulation script is *runSim.py*, located in the root PUMPS directory. Once started, *runSim.py* looks for a trade file (in the form of a Python pickle) within the specified simulation directory. After opening the trade file, it unpacks the pickle into a single multi-field variable called *trade_config*. The *trade_config* variable is then used to initialize the simulation. First, the general simulation data is extracted, such as the trade ID and length of the simulation. Next, *trade_config* is passed to separate functions which instantiate the *Task* and *Vehicle* objects according to the trade configuration. *Task* and *Vehicle* objects are created within their respective functions and contain the appropriate attributes and methods outlined in Sec. 3.4.1. To complete initialization, a vector of each type of object is returned to the main simulation loop.

3.4.2.3 The Main Loop.

After initialization, the first step in the main loop is to decide which vehicle should make the next task selection, based on the vehicle with the earliest task arrival time. If multiple vehicles are arriving to their next task at the same time, the vehicle with the lowest ID takes priority.

Once the deciding vehicle has been selected, its location and time attributes are updated to reflect the current task and simulation time. At the same time, the age of the task that was just visited is set to zero in both the deciding vehicle's task *age_tracker* and within the age attribute of the visited *Task* object itself. The age attributes of all other *Task* objects are incremented by the time elapsed since the previous task visit. The ages of the tasks in the vehicle *age_tracker* are not yet incremented, as they are updated to reflect the current arrival time at the end of the main loop iteration.

Before the vehicle selects its next task, the current visit is added to the main data output variable, *visit_order*. The *visit_order* maintains a historical record of which tasks have been visited by which vehicles, the time they arrived, and the trajectory they flew en-route.

After task ages have been updated, the vehicle selects its next task according to the supplied routing method. Once a task has been selected, the vehicle calculates the path to the task according to the supplied pathing method. After the path calculation, the ages of all tasks in the vehicle *age_tracker* are increased by the time of the planned arrival at the next task less the current simulation time. Finally, the deciding vehicle communicates, sharing information about the task completed or the next task selected with all other vehicles according to the supplied communication mode.

With the task selection decision complete, the main loop updates the visit number

and simulation time before returning to the beginning of the loop to choose the next deciding vehicle. Tasks continue to be selected until the simulation termination conditions are reached. The *visit_order* data is saved to the simulation directory within a *trade_results* pickle, along with the task and vehicle vectors and other general simulation configuration information. All relevant simulation information is stored, making it easy to create a custom script for data analysis. Figure 3.7 provides a visual depiction of the main PUMPS simulation loop.

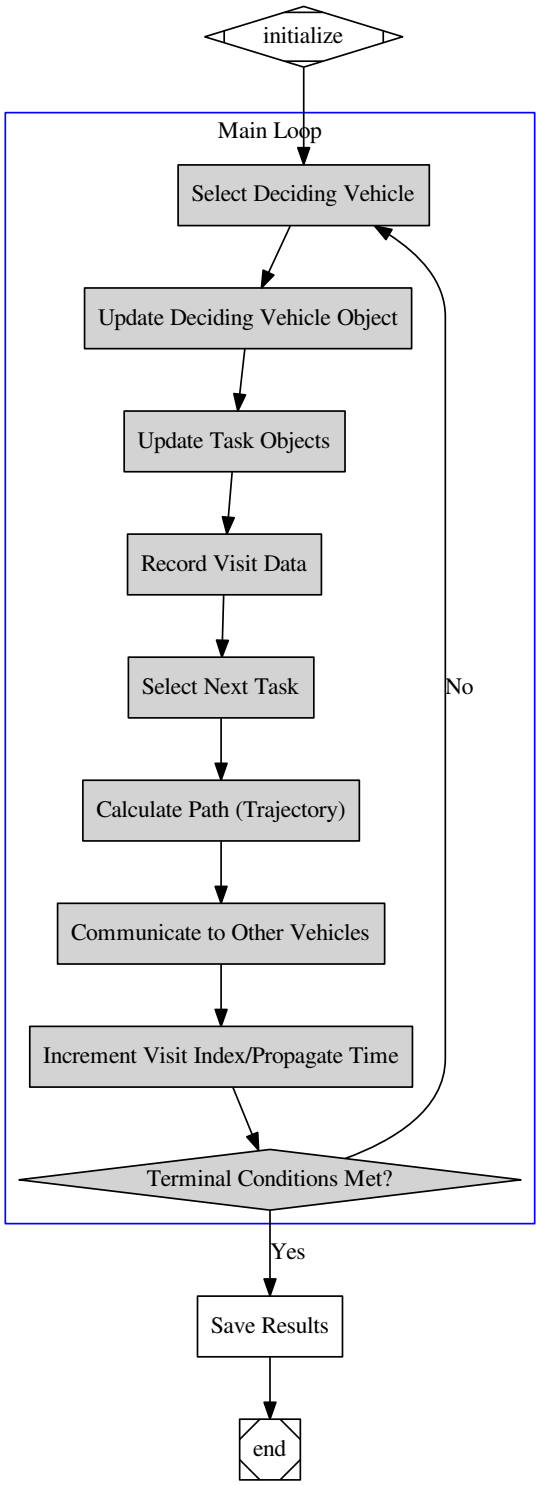


Figure 3.7. Data flow diagram for the PUMPS main loop, from initialization to termination.

3.4.2.4 Minor PUMPS Algorithms.

While the PISR task selection algorithms are the heart of PUMPS, several minor algorithms are also used. Two such algorithms are described below because they require design decisions that impact simulation results.

Finding the Shortest Travel Time with Dubins Pathing.

When the Dubins pathing module is selected, vehicles must calculate the shortest flight path to their destination subject to the motion constraints. PUMPS uses an optimal Dubins path calculator[78] which generates the path between two tasks, given the current heading *and desired arrival heading.* Of course, the optimal arrival heading is unknown. So, to determine the optimal path PUMPS uses a simple discretization strategy. The Dubins path for each arrival angle between 0 and 337.5 deg by steps of 22.5 deg is calculated and the shortest path is selected.

The choice of step size in the arrival heading discretization impacts the scenario results when Dubins paths are in use. A step size of 22.5 deg was selected because it provides the vehicle with ample options (16 arrival headings to choose from) while still being quick to compute. A finer step size provides the vehicle more flexibility, but the difference in vehicle trajectory between each option becomes less distinct while increasing computation time.

Calculating Task Ages with Multiple Vehicles.

If multiple vehicles are servicing tasks with the MD^2WRP routing module and the Broadcast Destinations (CxBD) communication mode is active, the calculation of task ages must be modified to reflect the activity of other vehicles. When a vehicle is calculating task values during its decision cycle, for each task under consideration it checks the information in its *vehicle_tracker* to see if another vehicle has commu-

nicated that it is en-route to that task.

The algorithm selected and used throughout this document considers two situations. If the other vehicle will arrive at the task before the deciding vehicle, the age of the task in the deciding vehicle's *age_tracker* is changed to reflect the arrival time of the deciding vehicle less the time of the interim visit. In the event multiple other vehicles are bound for the task under consideration, a check is performed such that only the vehicle with the arrival time closest to that of the deciding vehicle is taken into account.

If the other vehicle will arrive at the same time, or after, the deciding vehicle, the utility value of the task under consideration is set to zero. This prevents one vehicle from "cutting off" another in an attempt to eliminate redundant visits to a task. Another option that was considered but not implemented, was to allow the deciding vehicle to select a task even if another vehicle was en-route with an arrival time after the calculated arrival time of the deciding vehicle. However, the utility value of the task would be reduced. For example, the utility being received by the other vehicle could be subtracted from the utility calculated by the deciding vehicle. For simplicity and to encourage vehicle separation, this method was discarded in favor of the "zero utility" method.

3.5 Task Configurations

Six maps with different scale and task geometries are used throughout this document to demonstrate the behavior and performance of MD^2WRP. The first two maps are simple triangle configurations intended to make analysis of the basic properties of MD^2WRP easier. One triangle is equilateral and the other isosceles (Fig. 3.8).

The other four maps are designed to be representative of the way tasks might be arranged in an operational scenario. The four maps are presented in Fig. 3.9.

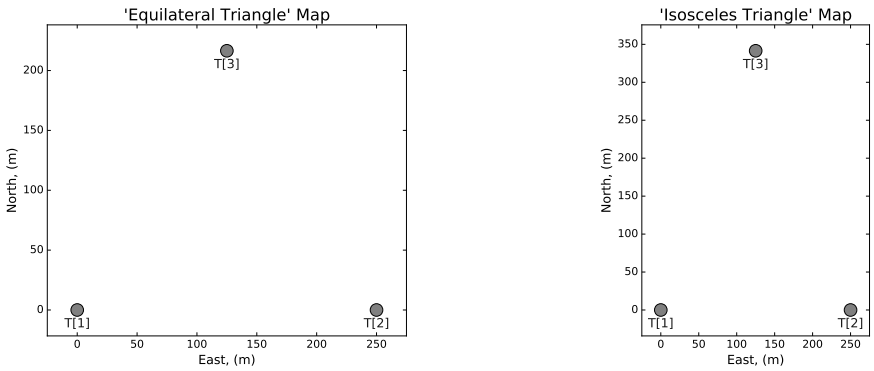


Figure 3.8. Simple triangular task configurations. These scenarios are useful for analyzing properties of the MD^2WRP utility function.

The Circle map represents a base perimeter defense mission. The Grid represents intersections to be monitored in an urban grid. The Random map is for a wide area surveillance mission across a large geographic region. Finally, the Clusters map represents several geographically separated areas of interest, such as a group of forward operating bases or small villages in a rural area.

3.6 Research Plan

To address the research questions in Ch. I, a plan consisting of three parts is presented below. Each part focuses on a different aspect of developing MD^2WRP from a theory to a practical utility function for task selection in PISR.

3.6.1 Characterization of MD^2WRP.

The first research step is to characterize the MD^2WRP function in Eq. 3.2 (or more precisely, the normalized version in Eq. 3.16). The desired outcome is first to understand how the parameters β and $\mathbf{w}$ affect vehicle *behavior* and then use that knowledge to develop a parameter optimization method that achieves the best *performance* (in terms of $\bar{L}$ from Eq. 3.1). We will also pursue analytical work re-

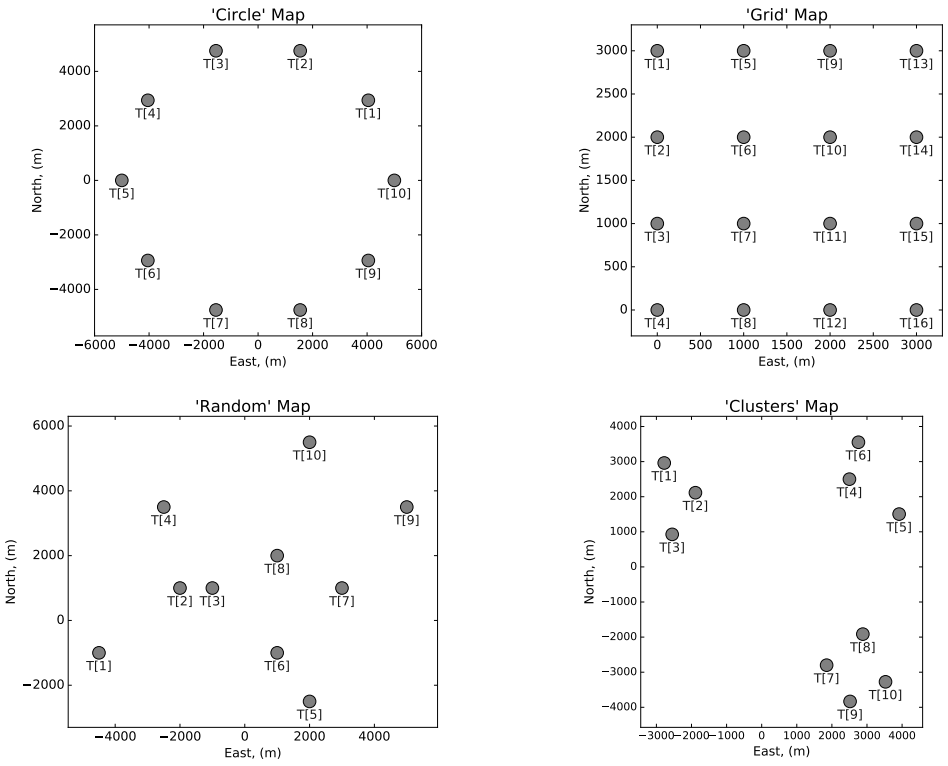


Figure 3.9. Four scenarios designed to represent how tasks might be distributed in various operational scenarios. The four configurations are Circle (top left), Grid (top right), Random (bottom left), and Clusters (bottom right).

garding the evolution of the task age vector, specifically investigating the transient and steady-state phases of the task visit sequences produced by MD^2WRP.

3.6.2 Comparison Studies of MD^2WRP.

We wish to conduct two types of MD^2WRP comparison studies. The first evaluates different versions of MD^2WRP itself. We evaluate MD^2WRP with multi-decision lookahead, where vehicles make task selections based on an increasingly longer decision horizon, from one to three decisions. Next, we explore MD^2WRP with multiple vehicles, testing three different inter-vehicle communication modes.

The second type of comparison is between MD^2WRP and other PISR methods from the literature. We compare single and multi-vehicle versions of the Traveling

Salesman Problem to MD^2WRP, from both a performance and qualitative perspective. We also compare MD^2WRP to other utility functions. These comparisons to alternative PISR methods serve two purposes, to validate our MD^2WRP parameter optimization method and to explore the benefits and trade-offs associated with each PISR method.

3.6.3 MD^2WRP and Operational Factors.

The third part of the research plan is to evaluate MD^2WRP in the presence of four operational factors: Dubins constraints on vehicle motion, no-fly zones, return-to-base requirements, and the addition/removal of vehicle/tasks mid-mission. For Dubins motion, we wish to determine at what point the travel time to a task should be calculated using a Dubins path rather than the simple Euclidean distance between tasks. To do this, we develop a method of changing the ratio of the vehicle turn radius to the average distance between tasks. When no-fly zones are added to the map, our goal is to understand when the presence of the no-fly zone impacts performance to a level that necessitates re-tuning of MD^2WRP. To this end, we develop a non-dimensional parameter called the *impact ratio*, which measures the level of interference a no-fly zone has on vehicle flight paths. When a return-to-base requirement is imposed, we explore how MD^2WRP can be modified and tuned to meet such a requirement and what the implications are on performance. Lastly, when mission objects are added or removed mid-mission, we use simulations to demonstrate the robustness of MD^2WRP to changes in the mission environment.

IV. Results

4.1 Characterization of MD^2WRP

The first major research task is to characterize the normalized version of the MD^2WRP function in Eq. 3.16. The desired outcome is to understand how the parameters β and $\mathbf{w}$ affect vehicle *behavior* and use that knowledge to optimize MD^2WRP for better *performance*. That is, we wish to find the parameter values that minimize $\bar{L}$ (Eq. 3.1). All characterization work is done under the Euclidean path assumption.

4.1.1 Effect of MD^2WRP Parameters on Vehicle Behavior.

Two toy problems are presented below. The goal of these simple examples is to highlight how the MD^2WRP parameters, β and $\mathbf{w}$, affect agent behavior. We begin with studying the effect of the task weights, $\mathbf{w}$, with a simple equilateral triangle task configuration. Then, we use an isosceles triangle configuration to examine β.

4.1.1.1 Task Weights (w).

To understand the effect of $\mathbf{w}$, three tasks are placed in an equilateral triangle (Fig. 4.1). Since the tasks are equally spaced and the vehicle travels Euclidean paths, β has no effect, as all tasks would be equally discounted. Hence, for this example β is zero. Instead, the objective is to highlight how each element of $\mathbf{w}$ (the individual task weights, w_j) affects vehicle task selection. We do this by performing trade studies using different w_j values. Data from four simulations are presented in Figs. 4.2 and 4.3. Figure 4.2 represents the visits per hour (vph) to each task, for each trade. Figure 4.3 shows the times at which a vehicle visited each task for Trades 1002 and 1003. Each trade consists of 100 task selections.

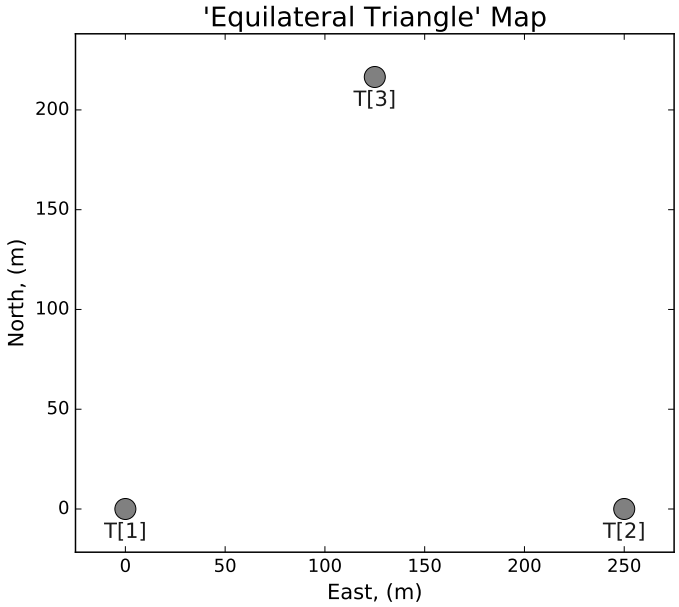


Figure 4.1. Three PISR tasks in an equilateral triangle configuration.

From Figs. 4.2 and 4.3, it is clear that increasing w_3 (weight of the top vertex) from 1.0 to 1.5 did not affect the overall revisit rate to any task. However, with $w_3 = 1.51$ a tipping point has been met or surpassed. Task 3 now has approximately double the vph of Tasks 1 and 2. So, is it possible to determine the exact value of w_3 that causes the shift? The answer is yes for this simple three-task problem.

With $\beta = 0$, $w_1 = w_2 = 1.0$, $w_3 = x$ and all $t_{ij} = \frac{d_{ij}}{v} = \frac{250m}{22m/s} = 11.4s$, we can reconstruct the vehicle's decision history to determine the value of x, that is, the value of w_3 that results in the behavior change. Going through the decisions one by one also serves as a good exercise to understand how task selections are made with utility functions. For the first decision, the vehicle will calculate the following values, assuming it begins at Task 1,

$$V_1 = w_1(T_1 + t_{1,1}) = (1.0)(0 + 0) = 0$$

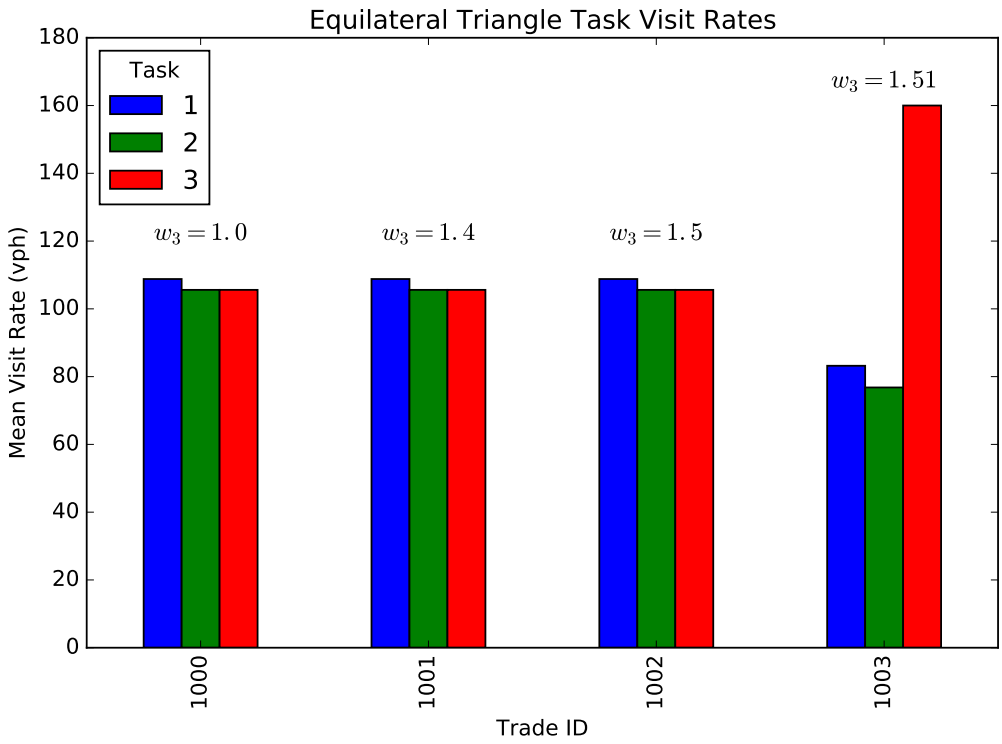


Figure 4.2. Visit rates from the equilateral triangle scenario, with varying weights applied to the top vertex (Task 3).

$$V_2 = w_2(T_2 + t_{1,2}) = (1.0)(0 + 11.4) = 11.4$$

$$V_3 = w_3(T_3 + t_{1,3}) = (x)(0 + 11.4) = 11.4x.$$

Clearly, we want to pick x greater than 1.0 since our goal is to increase the frequency of visits to Task 3. So, the vehicle selects Task 3 and on the second decision calculates,

$$V_1 = w_1(T_1 + t_{3,1}) = (1.0)(11.4 + 11.4) = 22.8$$

$$V_2 = w_2(T_2 + t_{3,2}) = (1.0)(11.4 + 11.4) = 22.8$$

$$V_3 = w_3(T_3 + t_{3,3}) = (x)(0 + 0) = 0.$$

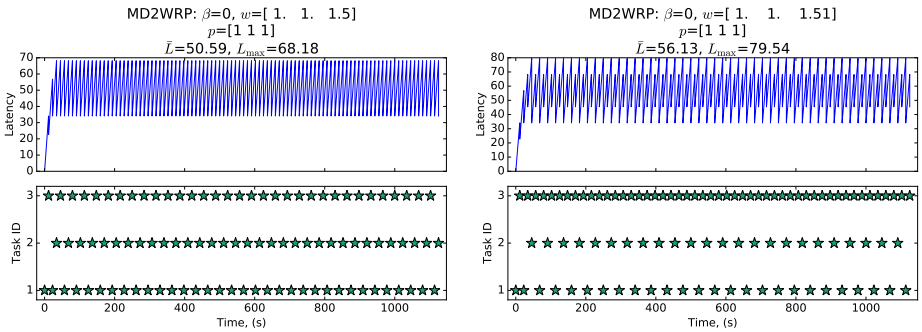


Figure 4.3. Times at which each task was visited by the vehicle for Trades 1002 ($w_3 = 1.5$) and 1003 ($w_3 = 1.51$).

Here, the vehicle could select either Task 1 or 2, since both have the same value. In this research, we set the vehicle to default to the task with lowest ID as a tie-breaker. So, Task 1 is selected and the third decision is calculated as,

$$V_1 = w_1(T_1 + t_{1,1}) = (1.0)(0 + 0) = 0$$

$$V_2 = w_2(T_2 + t_{1,2}) = (1.0)(22.8 + 11.4) = 34.2$$

$$V_3 = w_3(T_3 + t_{1,3}) = (x)(11.4 + 11.4) = 22.8x.$$

Since Task 2 has yet to be visited, its age is now old enough that it will be visited next unless a sufficiently large value is chosen for x. So the value of x needed to select Task 3 over 2 is,

$$22.8x > 34.2 \implies x > 1.5.$$

As long as $w_3 > 1.5$, Task 3 will dominate and be visited twice as often as Tasks 1 or 2. We can verify this by setting $x = 1.51$ and observing the next two decisions,

$$V_1 = w_1(T_1 + t_{3,1}) = (1.0)(11.4 + 11.4) = 22.8$$

$$V_2 = w_2(T_2 + t_{3,2}) = (1.0)(34.2 + 11.4) = 45.6$$

$$V_3 = w_3(T_3 + t_{3,3}) = (1.51)(0 + 0) = 0$$

and

$$V_1 = w_1(T_1 + t_{2,1}) = (1.0)(22.8 + 11.4) = 34.2$$

$$V_2 = w_2(T_2 + t_{2,2}) = (1.0)(0 + 0) = 0$$

$$V_3 = w_3(T_3 + t_{2,3}) = (1.51)(11.4 + 11.4) = 34.4.$$

The agent continues in a 1-3-2-3 pattern. At the risk of stating the obvious, further increases to w_3 do not result in increased visit frequency. This is because, with Euclidean travel, it is not possible to visit the same node twice in a row. The sum $(T_j + t_{ij})$ would always be zero for the task at which the vehicle is located. For the equilateral triangle scenario, Task 3 can be selected at most every other decision.

It is also worth noting that with w_3 set exactly to 1.5, the agent must arbitrarily decide between tasks, since this weight results in equal utility values for all tasks. This is undesirable since it forces the agent to make an arbitrary decision using a tie-breaker. Fortunately, in a real-world scenario where travel times are unpredictable and tasks are not placed in perfectly symmetrical configurations, the probability of two tasks having equal utility values is low.

The equilateral triangle scenario highlights an important aspect of MD^2WRP: applying a weight, w_j, to a task may not influence the vehicle's behavior. Instead, weights have bifurcation points. In a more complex scenario with multiple tasks and vehicles, identifying these bifurcation points for tasks is not trivial. This challenge will be addressed when we begin to optimize MD^2WRP parameters for performance in Sec. 4.1.4.

4.1.1.2 Distance Discount (β).

In the next scenario, we extend Task 3 northward to create an isosceles triangle (Fig. 4.4). By keeping $\mathbf{w} = \mathbb{1}$, the effect of the MD^2WRP travel time discounting term in Eq. 3.2, $e^{-\beta t_{ij}}$, can be studied by varying β. Similar to the equilateral triangle, Figs. 4.5 and 4.6 present the mean task visit rates and task visit times, respectively. There are 100 task selections for each trade.

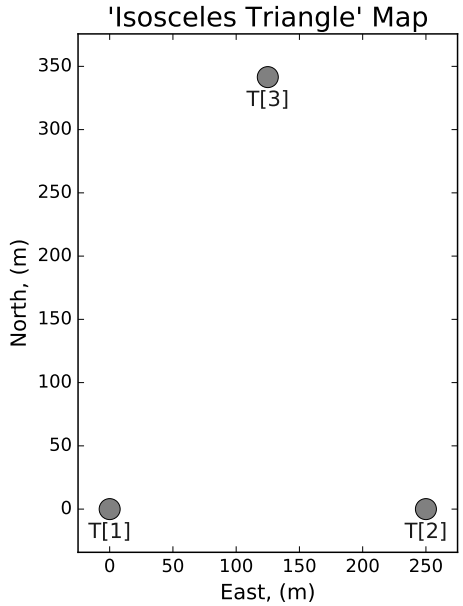


Figure 4.4. Three PISR tasks in an isosceles triangle configuration.

Figure 4.5 clearly demonstrates the effect of the β parameter. As β increases the vehicle spends more time servicing Tasks 1 and 2, since the reward for Task 3 has heavier discounting. The increasing sparsity of visits to Task 3 is evident in Fig. 4.6. When $\beta = 0.4$, the vehicle only selects Task 3 six times out of 100 decisions.

The isosceles triangle example demonstrates that a non-trivial range of β values exist for a particular task configuration. We use "non-trivial" in the sense that if β is too small, such as $\beta = 0.1$ in Fig. 4.5, it has no effect on task selections. Conversely, if β is too large, it results in one task being virtually ignored (as with $\beta = 0.8$ in Fig. 4.5).

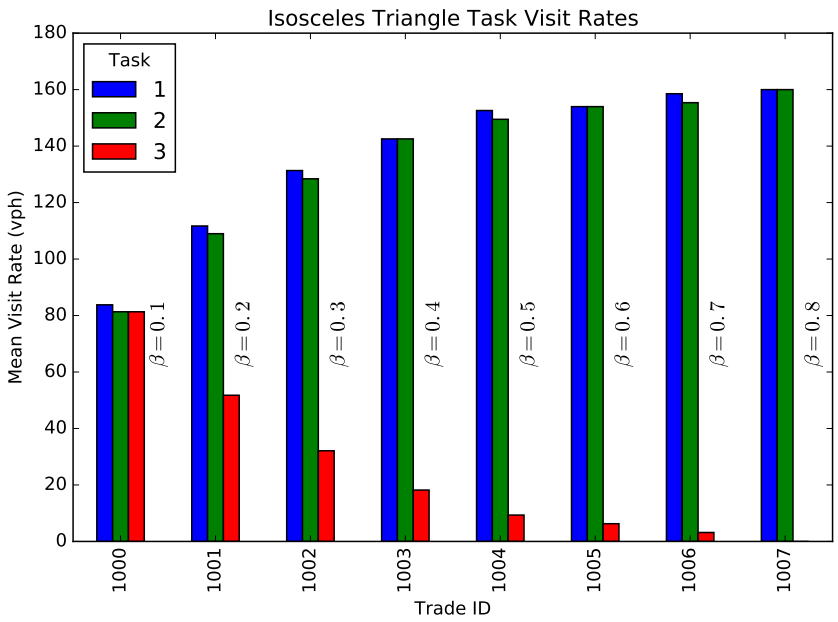


Figure 4.5. Task visit rates in the isosceles triangle scenario, with varying β values and $\mathbf{w} = \mathbb{1}$.

Unfortunately, identifying which values are within the non-trivial β range is difficult, since the range shifts as the task configuration changes. This is true even when two task configurations have similar shapes but different sizes, such as occurs when the D matrix is multiplied by a scalar. The solution to this conundrum is normalizing the MD^2WRP utility function, which is demonstrated next in Sec. 4.1.2.

4.1.2 The Value of Normalization.

In Sec. 3.3.2, the challenge of selecting β under a wide variety of task distributions is discussed. The challenge stems from the fact that as distance matrix, D, changes, so too does the effect β has on vehicle behavior. In other words, if D is multiplied by a scalar, a different task visit sequence will result even if β is unchanged. To resolve this issue, Eq. 3.16 was introduced, which is a normalized version of the original MD^2WRP utility function in Eq. 3.2. Equation 3.16 is reprinted below for

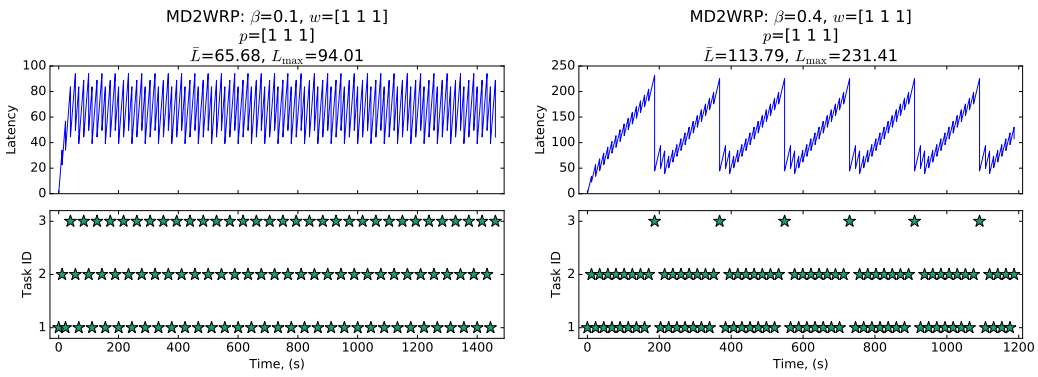


Figure 4.6. Times at which each task was visited by the vehicle in Trades 1000 ($\beta = 0.1$) and 1003 ($\beta = 0.4$) of the isosceles triangle scenario.

convenience.

$$V = \max_j \left[e^{-\beta \frac{t_{ij}}{t_{ij,\max}}} w_j \frac{(T_j + t_{ij})}{t_{ij,\max}} \right], \qquad \forall j \in \{1, \ldots, n\}.$$

Consider again Fig. 4.5. Note that the β values are small in magnitude and the range of non-trivial values is narrow. Any $\beta \leq 0.1$ will result in equal visit rates to all tasks, since the discount on travel distance is not sufficient to change the vehicle's default (*i.e.* as if $\beta = 0$) behavior. Likewise, any $\beta \geq 0.8$ will result in the vehicle virtually ignoring Task 3 for all 100 task selections. It is only within the range of $0.1 < \beta < 0.8$ that the vehicle exhibits interesting behavior. This is troublesome for two reasons. First, it makes it difficult to identify the correct magnitude and feasible range of β. Second, once the range is identified, it is only valid for a specific D. Any change in D will result in a new range of viable βs, perhaps with a different order of magnitude. Normalization, then, should make it possible to establish a general range of "well-behaved" βs to choose from, regardless of D.

As an illustrative example, the same isosceles triangle configuration used to generate Figs. 4.5 and 4.6 is simulated a second time, but using Eq. 3.16 to make task selections. The results are shown in Figs. 4.7 and 4.8.

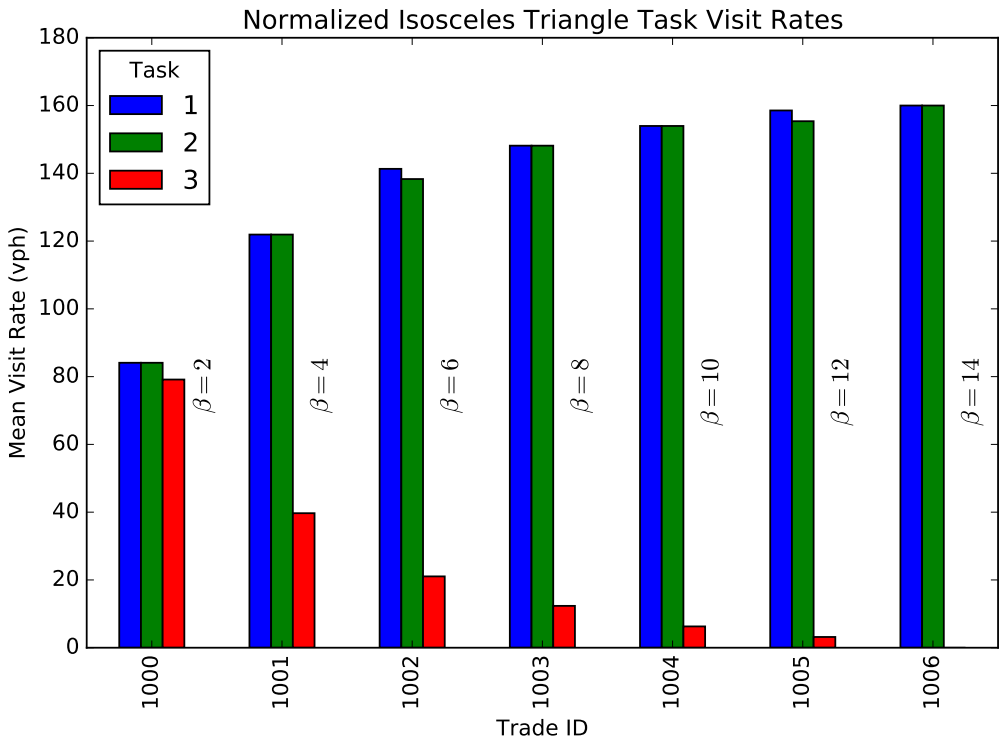


Figure 4.7. Visits per hour for the normalized isosceles triangle scenario (using normalized MD^2WRP), with varying β values and $\mathbf{w} = \mathbb{1}$.

The same general behavior is observed in Figs. 4.7 and 4.8 as in Figs. 4.5 and 4.6, with Task 3 receiving fewer visits as β increases. We also see there is still a non-trivial range of βs. Although now the range of β is approximately $2 < \beta < 14$. However, to demonstrate the real value of normalization, a third simulation is performed with an isosceles triangle of the same side length ratios, but 10 times larger (*i.e.* D for the isosceles triangle is multiplied by ten). The results of that simulation are not depicted here, because they generated visit rates and visit histories identical to those in Fig. 4.7 and Fig. 4.8, respectively. This result demonstrates that, as desired, the normalized MD^2WRP function makes it possible to apply the same β and get the same task visit sequence, even when the task distance matrix, D, is multiplied by a scalar.

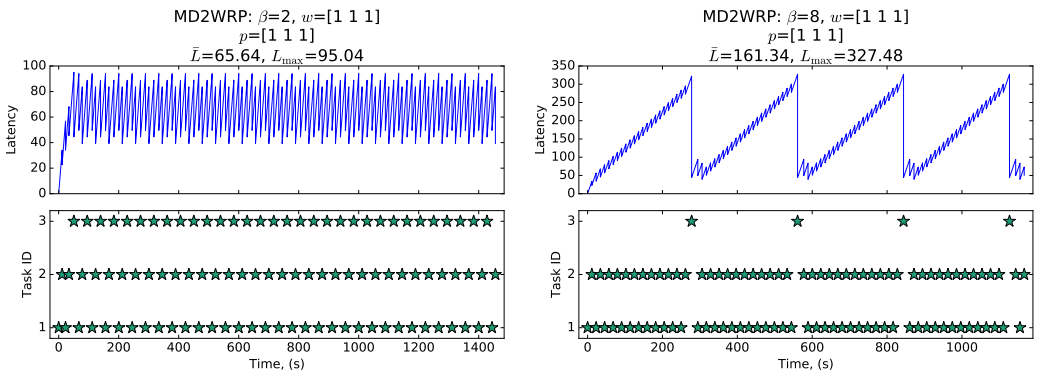


Figure 4.8. Times at which each task was visited by the vehicle for Trades 1000 ($\beta = 2$) and 1002 ($\beta = 8$) on the normalized isosceles triangle scenario.

4.1.3 Periodicity.

In this section we provide a proof that the steady-state behavior of MD^2WRP is periodic regardless of the initial state. We also provide simulation results to demonstrate the transition from a transient response to a periodic steady-state. Our simulations reinforce how the steady-state is always periodic, though the task visit pattern itself may vary based on the vehicle's starting location and the initial ages of the tasks. This proof was developed in cooperation with Dr. Kalyanam of AFRL and Dr. Baker of AFIT.

Definition 4.1.1. *Suppose we have n tasks. Let D be an $n \times n$ matrix with entries $d(i, j)$ representing the Euclidean distance between all tasks. Let S be the set of all states, where state s is defined by the current location of the agent, i, and a vector T of length n representing the ages of all tasks, that is, $s = (i, T)$. Let m be the total number of task visits, with each visit indexed by k, such that i_k represents the task at which the agent is located at visit k.*

To start, we show how the age vector, T, evolves with each agent decision, $k \in \{0, \ldots, m\}$. Suppose the initial age vector is T_0, then we can describe the task ages

at the time of the first visit, T_1, by,

$$T_1(j) = T_0(j) + d_{i_0,i_1}(1 - \delta_{i_1,j}) - \delta_{i_1,j}T_0(i_1), \qquad \forall j \tag{4.1}$$

where $j \in \{1 \dots n\}$.

Note that δ is the Kronecker delta. Equation 4.1 states that each component, j, of T is aged by the time required to move from task i_0 to i_1, except when $j = i_1$ (the age being updated belongs to the destination task), in which case the travel time is not added and the residual age of j is subtracted. Simply stated, the age of task j is reset to zero when it is visited by an agent. We can rewrite Eq. 4.1 in vector form as,

$$T_1 = T_0 + d_{i_0,i_1}(\mathbb{1} - \mathbf{e}_{i_1}) - \langle T_0, \mathbf{e}_{i_1} \rangle \mathbf{e}_{i_1} \tag{4.2}$$

where $\mathbb{1}$ is a vector of ones in $\Re^n$, $\mathbf{e}_{i_1}$ is the standard basis vector corresponding to task i_1, and $\langle T_0, \mathbf{e}_{i_1} \rangle$ is the standard dot product of T_0 and $\mathbf{e}_{i_1}$.

We can define the projection operator on vector $\mathbf{x} \in \Re^n$ as,

$$P_k \mathbf{x} = \langle \mathbf{x}, \mathbf{e}_k \rangle \mathbf{e}_k.$$

With the projection operator, Eq. 4.2 can be rewritten,

$$T_1 = T_0 + d_{i_0,i_1}(\mathbb{1} - \mathbf{e}_{i_1}) - P_{i_1} T_0 \tag{4.3}$$

and rearranged to,

$$T_1 = (I - P_{i_1})T_0 + d_{i_0,i_1}(\mathbb{1} - \mathbf{e}_{i_1}) \tag{4.4}$$

where I is an $n \times n$ identity matrix and the new state becomes $s_1 = (i_1, T_1)$. In

general we have,

$$T_{k+1} = (I - P_{i_{k+1}})T_k + d_{i_k,i_{k+1}}(\mathbb{1} - \mathbf{e}_{i_{k+1}}). \tag{4.5}$$

It can be shown that $\mathbb{1} - \mathbf{e}_{i_{k+1}} = (I - P_{i_{k+1}})\mathbb{1}$, thus,

$$T_{k+1} = (I - P_{i_{k+1}})(T_k + d_{i_k,i_{k+1}}\mathbb{1}). \tag{4.6}$$

So we can show that for visits $k = 0, \ldots, m$ and states $s_k = (i_k, T_k)$, and beginning with state $s_0 = (i_0, T_0)$,

$$T_1 = (I - P_{i_1})(T_0 + d_{i_0,i_1}\mathbb{1}) \tag{4.7}$$

with new state $s_1 = (i_1, T_1)$, and

$$\begin{aligned} T_2 &= (I - P_{i_2})(T_1 + d_{i_1,i_2}\mathbb{1}) \\ &= (I - P_{i_2})\left[(I - P_{i_1})(T_0 + d_{i_0,i_1}\mathbb{1}) + d_{i_1,i_2}\mathbb{1}\right] \\ &= (I - P_{i_2})(I - P_{i_1})T_0 + (I - P_{i_2})(I - P_{i_1})d_{i_0,i_1}\mathbb{1} + \\ &\quad (I - P_{i_2})d_{i_1,i_2}\mathbb{1} \end{aligned} \tag{4.8}$$

with new state $s_2 = (i_2, T_2)$. By induction, we have the age vector at visit m defined as,

$$T_m = \left(\prod_{k=1}^{m} B_k\right) T_0 + \sum_{l=1}^{m} d_{i_{l-1},i_l} \left(\prod_{k=l}^{m} B_k\right) \mathbb{1} \tag{4.9}$$

where $B_k = (I - P_{i_k})$ and $s_m = (i_m, T_m)$.

From Eq. 4.9, we see that the task ages at visit m depend exclusively on the initial age vector, T_0, and the entries of the D matrix, which are in turn selected by the control policy. Equation 4.9 allows for some interesting observations. First, however,

we state the following identities which can be proved by induction,

$$\prod_{k=1}^{m}(I - P_k) = I - \sum_{k=1}^{m} P_k \tag{4.10}$$

and

$$(I - P_k)^n = I - P_k, \qquad \forall n \geq 1, n \in \mathbb{N}. \tag{4.11}$$

Equation 4.10 states that the product of $I - P_k$ with $k = 1, \ldots, m$ is equivalent to I less the sum of each individual projection operator. Equation 4.11 simply states that repeated projection operators can be ignored.

Now, let us define $B_k = I - P_{i_k}$ and suppose that $\{1, 2, \ldots, n\} \subset \{i_k\}_{k=1}^{m}$ with $m \geq n$ (*i.e.* each task has been visited by the agent at least once). By Eq. 4.11, repeated projection operators can be ignored, so we have

$$\prod_{k=1}^{m} B_k = \prod_{l=1}^{n} B_l = \prod_{l=1}^{n}(I - P_l). \tag{4.12}$$

In other words, we only require one projection operator per task. Applying Eq. 4.10, we get

$$\prod_{l=1}^{n}(I - P_l) = I - \sum_{l=1}^{n} P_l. \tag{4.13}$$

Summing P_l for every task results in an $n \times n$ identity matrix, which yields

$$I - \sum_{l=1}^{n} P_l = I - I = 0 = \prod_{k=1}^{m} B_k. \tag{4.14}$$

Coming back to Eq. 4.9, suppose the agent has visited every task at least once such that $k = \tilde{m}$, then by Eq. 4.14 we have $\left(\prod_{k=1}^{\tilde{m}} B_k\right) = 0$ resulting in

$$T_{\tilde{m}} = \sum_{l=2}^{\tilde{m}} d_{i_{l-1}, i_l} \left(\prod_{k=l}^{\tilde{m}} B_k\right) \mathbb{1} \tag{4.15}$$

Note, the summation term with $l = 1$ has been discarded, since it would result in 0. If we look at the entries of vector T we see,

$$T_m(j) = \sum_{l=2}^{m} d_{i_{l-1},i_l} A_l \mathbf{e}_j \qquad \forall j \tag{4.16}$$

where $A_l \mathbf{e}_j$ is a coefficient of either 1 or 0, determined by the composite projection matrix acting on j at visit m. We see that $T_m(j)$ is composed of a linear combination of elements in D with a coefficient of 1 or 0.

Lemma 4.1.1. *(a) All task ages have an upper bound, M,*

$$T_k(j) < M_j \qquad \forall k = 1, \ldots, m, \forall j = 1, \ldots, n$$

(b) The state-space $S = \{s_0, \ldots, s_m\}$ is finite.

Proof. (a) We offer a proof by contradiction. Suppose for n tasks, the age of a single task j is unbounded. Then, as the age of task j increases to infinity, the value the agent receives for j approaches infinity, by Eq. 3.2. Since the infinite value of j would exceed the value of all other tasks, the agent would be forced to select j. Therefore, the age of j must have some upper bound, M.

(b) From (a), the ages of all tasks are bounded by M. From Eq. 4.16, given states $s_m = (i_m, T_m)$, the age of each task in T_m can only assume a value that is a linear combination of the elements in D (with a coefficient of 1 or 0), with lower bound 0 and upper bound M. Therefore, given a finite number of tasks, the state-space S is finite. □

Note that with Lemma 4.1.1(a) we now know that there necessarily exists some visit $k = \tilde{m}$ where the agent has visited every task at least once, which will result in Eq. 4.15.

Theorem 4.1.2. *For a given D and states s_k, and following myopic control policy π,*

$$i_{k+1} = \arg\max_{j \in U_i} e^{-\beta d(i_k,j)} w_j \left[T_k(j) + d(i_k, j)\right]$$

the steady-state sequence of task selections is periodic, where $i_k = \pi(s_{k-1})$ and $i_{k+1} = \pi(s_k)$.

Proof. The myopic control policy is deterministic so that any state $s_k \implies s_{k+1}$. That is, if an agent is located at task i_k with age vector T_k, the control policy will always result in the next agent task being i_{k+1} with corresponding age vector T_{k+1}. Invoking Lemma 4.1.1 (b), a finite state space requires that if an agent has a state progression of $s_0 \implies s_1 \implies \ldots \implies \hat{s_k} \implies s_k \implies s_{k+1}$, then at some point the agent must enter a state $\tilde{s_k}$ which returns the agent to some previously visited state, which we call $\hat{s_k}$. At this point, the agent enters a state feedback cycle which eventually returns the agent to state $\tilde{s_k} \implies \hat{s_k}$, the end result being a task visit order with a periodic structure. □

For a demonstration of periodicity, simulation results for the isosceles triangle with $\beta = 0$ and $\beta = 4$ are presented below. The weight vector is $\mathbf{w} = \mathbb{1}$ and 50 task selections are made ($m = 50$) with $V_{veh} = 22m/s$. Figure 4.9 includes both the visit histories of the vehicles as well as a plot of total latency. Latency is included here, not as a performance indicator, but because the latency "signal" makes it easy to identify the periodic portion of the visit sequence.

The isosceles triangle results show a clear periodic visit pattern. With $\beta = 0$ in Fig. 4.9, the pattern is simply, $\{1, 2, 3\}$. With $\beta = 4$, it takes longer for the pattern to repeat due to less frequent visits to Task 3. The new pattern is, $\{2, 1, 2, 1, 2, 1, 3\}$. Even though these scenarios are simple, the results hold regardless of the number of tasks or the values of the MD^2WRP parameters. Additional results are included

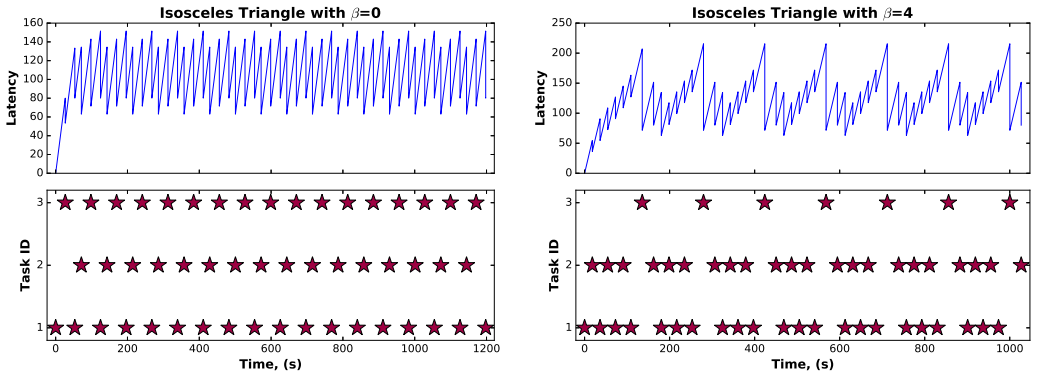


Figure 4.9. Demonstration of the periodicity of MD^2WRP under the isosceles triangle task configuration, with $\beta = 0$ (left) and $\beta = 4$ (right).

in Table 4.1. These were generated on the operational scenarios depicted in Fig. 3.9 from Ch. III.

Table 4.1. Visit pattern length (in number of tasks) and period for a variety of scenarios.

Map	β	Length	Period (s)
Isosceles Triangle	0	3	71.5
Isosceles Triangle	4	7	144.1
Circle	0	10	4140.9
Circle	8	10	1409.1
Clusters	0	10	2812.9
Clusters	4	22	1927.8
Grid	0	16	2042.3
Grid	5	16	831.2
Random	0	10	2795.2
Random	4	39	5294.2

One might ask, if the vehicle eventually enters a periodic pattern, why not determine the pattern and assign it as the vehicle route? While this idea seems reasonable, it undermines the purpose of decision making with utility functions. Specifically, it eliminates the element of adaptability that makes the utility approach desirable for an autonomous vehicle operating in a dynamic mission environment. If the vehicle's path will be predetermined, then other methods are better suited, such as those employing variants of the Traveling Salesman Problem, which we have already discussed

as having undesirable attributes for autonomous vehicle PISR.

While the task visit order produced by MD^2WRP is always periodic in the steady-state, different initial conditions may result in a different steady-state visit order. For a simple demonstration of the sensitivity to initial conditions, results are presented below in Sec. 4.1.3.1 and 4.1.3.2 for the isosceles triangle map. For each simulation, we use $\beta = 4.0$, $\mathbf{w} = \mathbb{1}$, $m = 50$, and $V_{veh} = 22m/s$.

4.1.3.1 Initial Conditions - Start Location.

In Fig. 4.10, results with the vehicle's starting location set to Task 1 (left) and Task 3 (right) are shown. A close inspection of the total latency for both scenarios shows that the latency curves are identical in the steady-state. Therefore, from a mission performance perspective, there is no difference in the two visit patterns once steady-state is achieved. In either case, Tasks 1 and 2 receive six combined visits for every visit to Task 3. The specific periodic visit order, however, changes. When the start location is Task 1, the periodic visit order is $\{2, 1, 2, 1, 2, 1, 3\}$, whereas starting at Task 3 produces $\{1, 2, 1, 2, 1, 2, 3\}$.

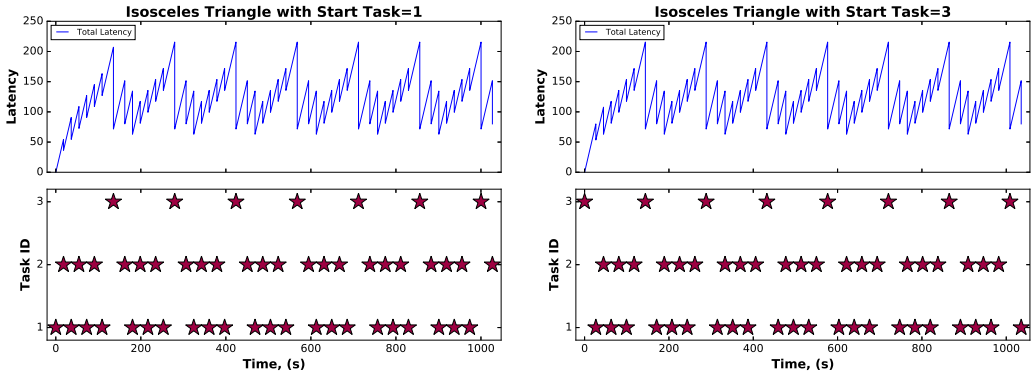


Figure 4.10. Visit patterns are dependent on vehicle starting location, but are always periodic in the steady-state.

4.1.3.2 Initial Conditions - Task Ages.

In Fig. 4.11, the initial normalized ages of the tasks are set to $T_0 = [0, .5, 3.7]$. The vehicle starts at Task 1 and it can be seen that the same steady-state latency curve emerges as in the left of Fig. 4.10, despite a slightly different transient. In this case, the steady-state visit pattern is the same as when all initial task ages are zero and the vehicle starts at Task 1, although this may not always be true in general. Interestingly, the transient period with a non-zero T_0 is shorter than with $T_0 = \bar{0}$.

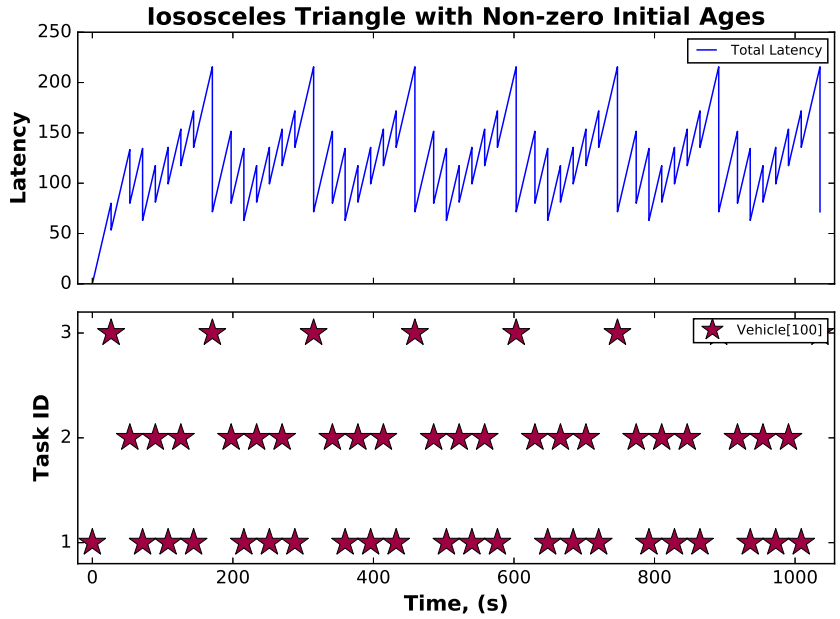


Figure 4.11. Visit patterns are dependent on initial task ages, but always periodic in the steady-state.

4.1.4 Optimizing β and w.

So far, the effects of the MD^2WRP parameters on vehicle behavior have been explored and it has been demonstrated that MD^2WRP produces steady-state visit patterns that are periodic. Before the performance of MD^2WRP is compared to other PISR methods, the parameters β and $\mathbf{w}$ must be optimized to provide the best

possible MD^2WRP performance.

Clearly, the search space for finding the optimal β and $\mathbf{w}$ is large, even for small problems. While normalization helps limit β to a relatively small range of viable values, each element of $\mathbf{w}$ can be any real positive number. An exact algorithm for finding the parameters that yield the global minimum latency is probably not possible. Still, a two-step heuristic method is proposed below that facilitates the selection of MD^2WRP parameters which yield empirically good results (as we will show with comparisons to other methods in Sec. 4.2) and provide some confidence that latency performance is within a local optimum, if not approaching the global minimum. The method below breaks the search into two phases: first the selection of β and then the weight vector, $\mathbf{w}$.

The parameter optimization method is again demonstrated on the isosceles triangle scenario. To enrich the example, Task 1 is given a higher priority. The priority vector is $\mathbf{p} = [3, 1, 1]$, such that the latency of Task 1 will increase three times faster than that of Tasks 2 or 3. Performance is based on average latency, $\bar{L}$, with maximum total latency, L_{max}, used as a tiebreaker.

4.1.4.1 Selecting β.

The search for β is conducted first since it is a single value. Also, its value will alter the effect of $\mathbf{w}$, which is a much larger space to search, so it makes sense to pick β first. For now, all tasks have an MD^2WRP weight equal to one ($\mathbf{w} = \mathbb{1}$). Due to normalization, the search range for β is relatively small and stable. Using the results in Fig. 4.7, a good place to begin the search for non-trivial β values is between 0-10. Values for β are simulated in increments of 0.25 with the latency performance of the top ten values for $m = 150$ task selections summarized in Table 4.2. The latency of the $\beta = 0$ case is shown for reference. Recall that $\beta = 0, \mathbf{w} = \mathbb{1}$ implies decisions are

based solely on the future age of tasks. The total latency curve and visit history for a vehicle with the optimal $\beta = 3.25$ (but using non-optimized $\mathbf{w} = \mathbb{1}$) is shown in Fig. 4.12.

Table 4.2. Top ten βs by best $\bar{L}$ performance.

β	$\bar{L}$	$L_{\max}$
3.25	101.90	182.86
2.75	104.66	182.86
2.25	104.94	182.86
2.50	104.94	182.86
3.50	105.10	182.86
3.75	105.18	182.86
3.00	105.83	172.53
2.00	106.60	182.86
1.75	106.60	182.86
4.25	106.68	196.29
0.00	106.98	182.86

The case with $\beta = 3.25$ provides the best performance in terms of $\bar{L}$, 4.7% better than with $\beta = 0$, which we use as our baseline policy. The reason $\beta = 3.25$ provides the best performance is because it strikes the appropriate balance in visit frequency between Task 1, with high-priority, and the low priority tasks. The vehicle visits Task 3 less often due to the distance discount from the exponential term, $e^{-\beta \frac{t_{ij}}{t_{ij,max}}}$. If β were smaller, the vehicle would visit Task 3 too frequently at the expense of significantly higher latency for Task 1. Conversely, if β were larger, Task 3 would not be visited enough and its latency would outgrow that of Task 1, despite its lower priority.

4.1.4.2 Selecting w.

With the best β selected, the next step is to determine the task weight vector, $\mathbf{w}$. This is more difficult than selecting β due to the size of the search space. In choosing weights, it should also be noted that the values themselves are not important, but

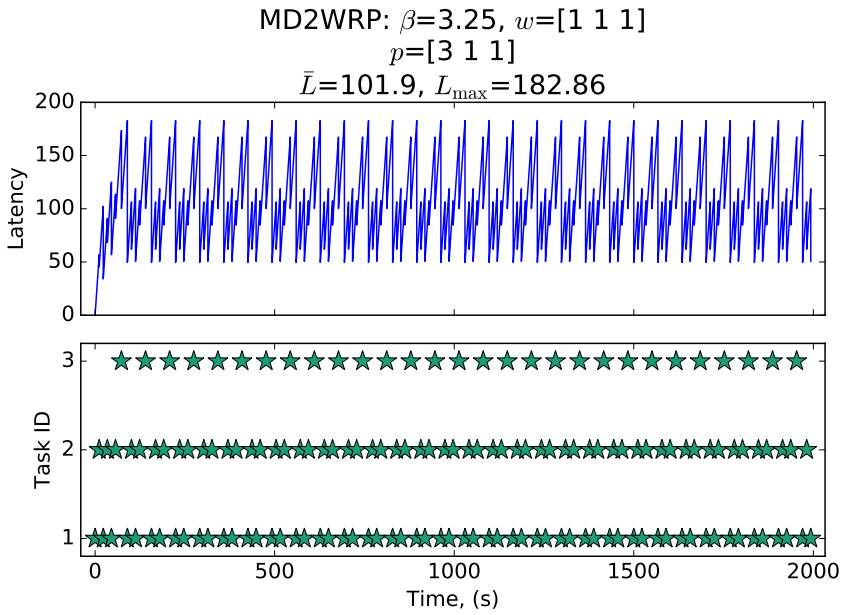


Figure 4.12. Total latency curve and task visit history for a vehicle operating on the isosceles triangle map with $\beta = 3.25$, the optimal β for this scenario.

the ratio between them.

The simplest way to test weights is to simulate numerous weight vectors. To begin, we explore 27 task weight combinations, allowing each task to have the weights $w_1 = \{1, 2, 3\}$, $w_2 = \{1, 2, 3\}$, and $w_3 = \{1, 2, 3\}$. The results for the top ten weight combinations by $\bar{L}$, using $\beta = 3.25$, are shown in Table 4.3. The latency curves and visit histories for two weight vectors of special interest, $\mathbf{w} = [3, 2, 3]$ and $\mathbf{w} = [3, 1, 1]$, are in Fig. 4.13.

Of the 27 weight vectors tested, the best performance is with $\mathbf{w} = [3, 2, 3]$, which results in a 9.6% improvement over the baseline policy ($\beta = 0$, $\mathbf{w} = [1, 1, 1]$), or an additional 4.9% improvement from the β-only optimization ($\beta = 3.25$, $\mathbf{w} = [1, 1, 1]$). Referring to the visit history on the left of Fig. 4.13, the vehicle achieves this performance by visiting Task 1 between every visit to Tasks 2 and 3. This makes intuitive sense because the high-priority of Task 1 warrants additional visits. What may not

Table 4.3. Performance of each w by $\bar{L}$ ($\beta = 3.25$).

w	$\bar{L}$	$L_{\max}$
$[3, 2, 3]$	96.70	160.13
$[3, 1, 3]$	97.03	160.13
$[2, 1, 2]$	97.03	160.13
$[3, 1, 2]$	97.03	160.13
$[3, 1, 1]$	99.87	162.20
$[2, 1, 1]$	99.87	162.20
$[3, 2, 2]$	99.87	162.20
$[1, 1, 1]$	101.90	182.86
$[2, 2, 3]$	106.60	182.86
$[1, 1, 3]$	106.98	182.86

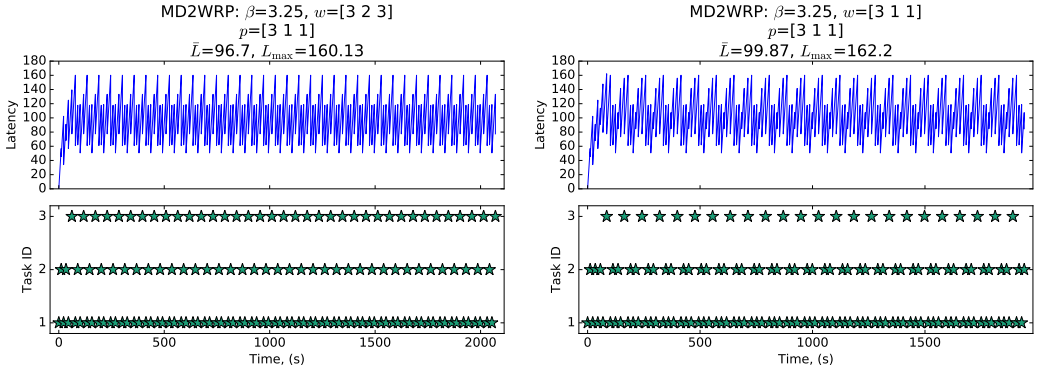


Figure 4.13. Latency curves and visit history with $\beta = 3.25$ and $\mathbf{w} = [3, 2, 3]$ (left) and $\mathbf{w} = [3, 1, 1]$ (right).

be intuitive, however, is the additional weights on Tasks 2 and 3, despite their lower priority. The reason is due to the discounting effect of β. Optimizing β alone did not yield the visit sequence required to achieve this performance. However, the additional weight on Task 3 partially offset the β discount to encourage a slightly higher visit frequency. Together, along with β, the weights in **w** adjusted the weight ratio between all tasks and worked in concert to deliver the best performance.

Unfortunately, optimizing **w** by testing weight combinations for every task has some significant limitations. As just demonstrated, the best performance may involve increasing the weight of some low priority tasks. For the isosceles triangle example,

this is not a problem. However, it becomes computationally impractical when the number of tasks is ten, or sixteen, such as with our operational scenarios depicted in Fig. 3.9. Assuming only a single task has an increased priority, testing even two priorities for every task requires simulating $2^{10} = 1024$ different weight vectors for ten tasks or $2^{16} = 65536$ for sixteen! In general, if r is the number of different priorities to test and n the number of tasks, there are r^n weight combinations to test.

To limit the search space, we can instead leave all low priority tasks with a weight of one and only increase the weight of the high-priority task. In that case, the best performance from Table 4.3 is with $\mathbf{w} = [3, 1, 1]$. Only modifying the weight of Task 1 reduces performance in favor of easier computation, but improvements in $\bar{L}$ are still realized. The improvement over the baseline policy becomes 6.6% and improvement over the β-only optimization is 1.9%. Overall, for this scenario we sacrifice 3% of our performance improvements, but reduce the number of weight combinations from 27 to 3.

Interestingly, what might have been the most intuitive choice, $\mathbf{w} = [3, 2, 1]$ (in other words, a $\mathbf{w}$ that matches $\mathbf{p}$), yields significantly worse results than leaving all weights at one. The value is not included in Table 4.3 because it is not in the top ten weight combinations. The $\bar{L}$ with $\mathbf{w} = [3, 2, 1]$ is 124.24. This reinforces that priority, established by the operator, and MD^2WRP weight, for optimizing vehicle performance, are independent concepts and that one is not necessarily a good guess for the other.

While the isosceles triangle example is simple, it reveals some truths that translate to larger, more complex scenarios. In general, only optimzing β, which is a quick and simple search, yields significant performance improvement over the baseline policy (*i.e.* $\beta = 0, \mathbf{w} = \mathbb{1}$). Weight optimization, on the other hand, does yield significant performance improvement if we test different weights for all tasks, which

comes with an increased computational burden. Limiting the weight increase to only the high-priority task(s) reduces the number of weight combinations to test, while still providing a performance increase. In Sec. 4.1.4.3, we demonstrate the two-step optimization method on more complex scenarios and only increase the weight of the single high-priority task.

From an operational standpoint, minimal effort should be spent on selecting a weight vector, since MD^2WRP is intended for use in a dynamic mission environment. The frequent introduction or removal of tasks and vehicles makes it impractical to continuously evaluate for an optimal weight vector. Therefore, if all tasks are of equal priority it is recommended to optimize β and use $\mathbf{w} = \mathbb{1}$. If some tasks have increased priority, only adjusting the weight of the high-priority tasks saves computation time while still improving performance.

The two-step optimization method introduced here is a simple brute force approach. For both β and $\mathbf{w}$, we simply search a discrete linear progression of values, simulating each until the values that results in the lowest $\bar{L}$ are found. While this method is simple, we demonstrate in Sec. 4.2 that it results in good performance that is competitive with other methods of task selection. In Sec. 5.2, we describe potential alternatives for optimizing β and $\mathbf{w}$ that may provide a more thorough search of the space, resulting in the discovery of better local minimums.

Lastly, it should be noted that it is possible to obtain any visit pattern under any task configuration solely through the manipulation of weights, whether distance discounting is used or not. Finding the weight vector that yields a specific visit pattern, however, is too difficult and of little value. Even if it were possible to precisely control visit patterns through weights, this would be detrimental to PISR with multiple vehicles. As will be discussed in Sec. 4.2, the real benefit of MD^2WRP is realized under a multi-vehicle scenario, where careful selection of β provides for the

emergence of decentralized cooperative behavior. If individual task weights are used in place of β, this cooperative behavior is lost.

4.1.4.3 Optimization Examples.

In this section, the two-step MD^2WRP optimization method is demonstrated on different task configurations and priority vectors. The purpose is to characterize the potential performance gains through optimization of β and $\mathbf{w}$ and to show the optimization methodology is effective regardless of the task configuration.

To simplify analysis, a priority vector is chosen for each map with one task given a priority of ten and all others set to one. This serves the dual purpose of limiting the search space for $\mathbf{w}$ (assuming that only weights for the high-priority task are adjusted) and accentuating the play between priorities and weights. In Table 4.4, for five task map and priority vector combinations, performance data is shown first for the baseline MD^2WRP ($\beta = 0, \mathbf{w} = \mathbb{1}$), then with optimal β and $\mathbf{w} = \mathbb{1}$, and finally with both β and $\mathbf{w}$ optimized. The percent improvement over the baseline policy is shown in the rightmost column.

In general, only optimizing β yields significant performance improvement over the baseline policy, more than halving $\bar{L}$ in every scenario tested. Weight optimization, on the other hand, only yields an additional 3-4% improvement in most cases. Occasionally, as seen for the Circle and Grid (with $p_{16} = 10$), the optimal weight of the high-priority task is equal to 1.0, so there is no difference between the β-only optimization and the optimization including w_j. Curiously, when the high-priority task is along the outermost edge of the map, adding a weight is detrimental to performance. The best choice is to leave the weight equal to one. This can be seen in the Circle map, where every task could be considered on the outermost edge, as well as the second Grid scenario where Task 16 is in the bottom-right corner. Performance decreases in

Table 4.4. $\bar{L}$ results for various maps, βs, and w's.

Map	$\mathbf{p_j}$	β	$\mathbf{w_j}$	$\bar{\mathbf{L}}$	% Improv.
Rand		0	$w_5 = 1.0$	26247	-
Rand	$p_5 = 10$	3.5	$w_5 = 1.0$	13229	50.0
Rand		3.5	$w_5 = 5.5$	11961	54.4
Circ		0	$w_1 = 1.0$	38926	-
Circ	$p_1 = 10$	2.0	$w_1 = 1.0$	13184	66.1
Circ		2.0	$w_1 = 1.0$	13184	66.1
Clust		0	$w_7 = 1.0$	26512	-
Clust	$p_7 = 10$	2.5	$w_7 = 1.0$	10362	61.0
Clust		2.5	$w_7 = 4.5$	9561	64.0
Grid		0	$w_6 = 1.0$	25071	-
Grid	$p_6 = 10$	5.0	$w_6 = 1.0$	9236	63.2
Grid		5.0	$w_6 = 8.0$	8090	67.7
Grid		0	$w_{16} = 1.0$	25002	-
Grid	$p_{16} = 10$	1.0	$w_{16} = 1.0$	8931	64.3
Grid		1.0	$w_{16} = 1.0$	8931	64.3

these situations because the increased frequency of visits to the outer high-priority task creates a large opportunity cost among all other tasks, with the net result of driving up $\bar{L}$. Therefore, an edge task would need an exceptionally high-priority to justify additional weight.

We can be confident that the optimized β and $\mathbf{w}$ in Table 4.4 are local optimums because either increasing or decreasing their values (to a point which changes the visit pattern) results in worse performance. The plots of $\bar{L}$ versus β and versus the weight of the high-priority task are shown for the Random map in Figs. 4.14 and 4.15, respectively.

4.2 Comparison Studies of MD^2WRP

The next research task investigates the performance of several variants of MD^2WRP and then looks at how MD^2WRP compares to other methods for PISR task selection, specifically TSP solutions and other utility functions. Among the self-comparisons, different communication modes are examined for the multi-vehicle case. We also

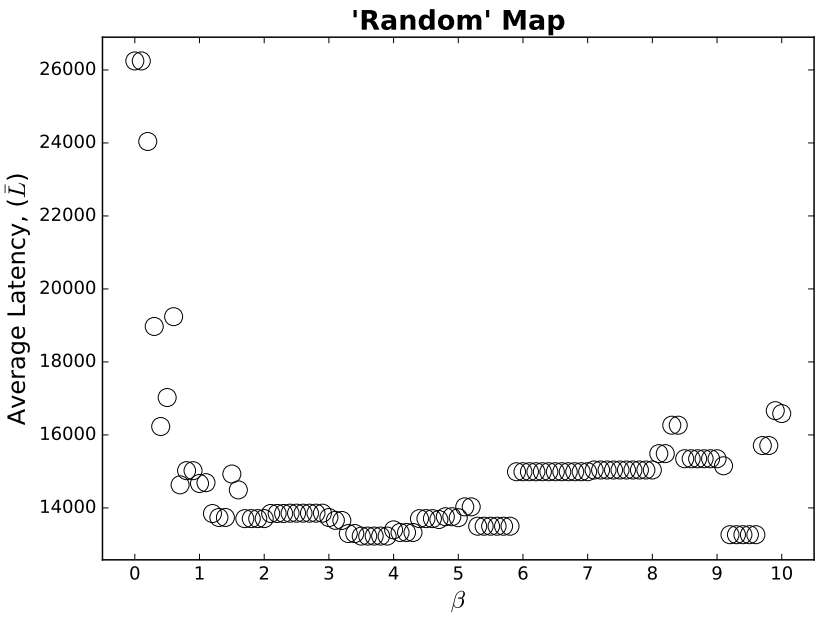


Figure 4.14. $\beta = 3.5 - 3.9$ **result in the best latency (w** $= 1$**).**

examine MD^2WRP with multiple decision lookahead. Then, MD^2WRP is compared to the cyclic and partition TSP strategies from [1], which we call n-spaced and k-subtours TSP, respectively. Finally, MD^2WRP is compared to the greedy direct latency minimization (DLM) utility function as well as the single-vehicle/multi-vehicle reactive policies (SRP/MRP) from [5].

4.2.1 MD^2WRP with Different Communication Modes.

To discuss multi-vehicle cooperation, one must address communication modes. There are many possibilities when it comes to communication amongst vehicles. However, in keeping with the spirit of simplicity that drove the development of MD^2WRP, only three basic modes are explored due to their minimalistic nature: no communication (CxNone), "Broadcast Completions" (CxBC), and "Broadcast Destinations" (CxBD).

We select the Clusters map using three vehicles to demonstrate each communica-

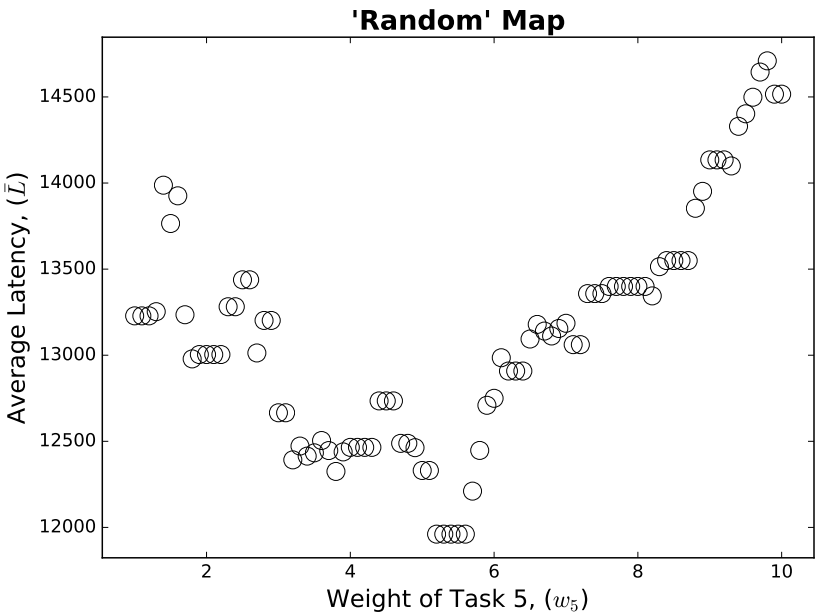


Figure 4.15. $w_5 = 5.2 - 5.6$ **result in the best latency** ($\beta = 3.5$)**.**

tion mode (bottom left of Fig. 3.9). The clusters scenario makes it easy to identify the advantages/disadvantages of each communication type due to the obvious opportunity for task partitioning. To further accentuate the characteristics of each mode, three different sets of initial conditions are used, each more challenging than the next: first with each of the three vehicles starting in separate clusters, then at separate tasks within the same cluster, and finally at the same task. We assume all tasks have equal priority and use $\mathbf{w} = \mathbb{1}$. For all three communication types we use $\beta = 5.0$. All task ages are initially zero.

4.2.1.1 No Communication (CxNone).

We begin with a scenario where communications are not available and each vehicle must make task selections independently. The CxNone mode provides a worst case baseline of performance to compare the CxBC and CxBD modes against for each of the three initial conditions described above.

From Sec. 4.1.3, it was shown that a single vehicle using MD^2WRP enters a periodic visit pattern. Given that conclusion, in the case of multiple vehicles operating in the same space as independent actors, each will eventually achieve a periodic pattern. This is indeed the case, as seen in Fig. 4.16. In this instance, the steady-state visit pattern for all three vehicles is the same, though this is not true in general.

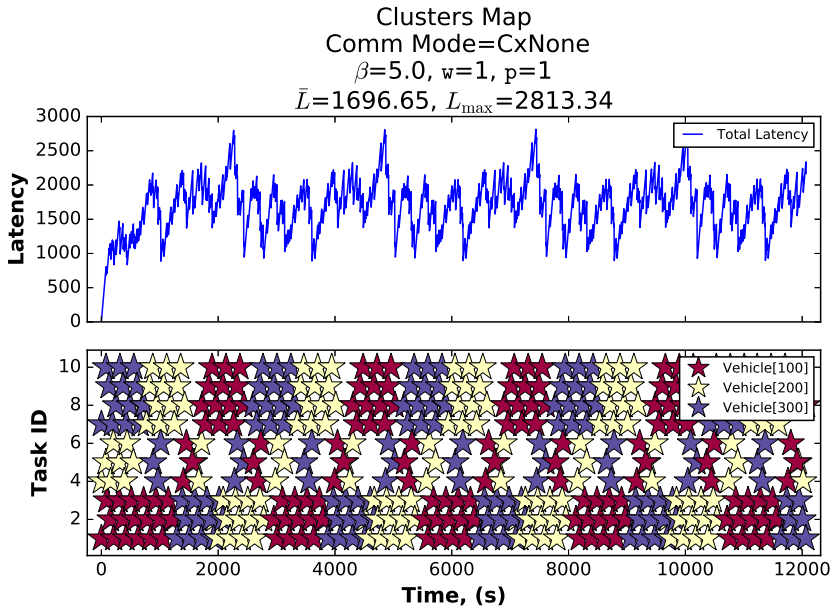


Figure 4.16. Three vehicles operating without communication. Each starts in a separate task cluster and eventually enters the same periodic pattern.

In Fig. 4.16, because the vehicles begin within separate clusters, they maintain a wide spacing resulting in relatively even coverage of tasks. The performance in terms of average weighted latency is $\bar{L} = 1696.65$ (Table 4.5). If vehicles begin at separate tasks within the *same* cluster, performance decreases drastically due to a tighter vehicle spacing ($\bar{L} = 5679.74$). A serious problem is encountered, however, when all vehicles start at the same task. Since all ages are initially zero, the vehicles begin in the same state, so every vehicle always makes the same task selection. The net result is the vehicles spending the entire simulation collocated. Since this defeats

the purpose of having multiple vehicles, it is essential in the CxNone mode that each vehicle start in a different state to ensure spacing between vehicles. Unfortunately, in an operational environment, initial vehicle states may not always be under the operator's control, since factors such as the base location may determine the first task visited.

Table 4.5. Performance of three vehicles on "Clusters" by start location ($\beta = 5$,$\mathbf{w} = \mathbb{1}$).

Start Locations	**Comm. Mode**	$\bar{L}$
Separate Clusters	CxNone	1696.65
	CxBC	1156.20
	CxBD	1156.20
Same Cluster	CxNone	5679.74
	CxBC	1810.38
	CxBD	1286.10
Same Task	CxNone	7266.59
	CxBC	7266.59
	CxBD	1308.83

4.2.1.2 Broadcast Completions (CxBC).

Sharing completion information is perhaps the simplest possible communication mode. Upon completing a task, each vehicle broadcasts the task ID and time of completion. All other vehicles update their table of task ages and use this information when the time comes to select their next task. In other words, this is as if all vehicles were making decisions from a single, shared database of task ages.

If the vehicles are initialized within the same cluster, but at different tasks, the CxBC mode significantly outperforms the no communication case, cutting latency from $\bar{L} = 5679.74$ to $\bar{L} = 1810.38$, a reduction of 68%. This performance increase is caused by an emergent cooperative behavior resulting from the implicit effect of the travel time discount factor, β. As seen in Fig. 4.17, the vehicles automatically partition themselves into separate clusters, where they remain for the duration of the

mission. It takes about 2100 seconds (35 minutes) for the vehicles to enter the final partitioned visit sequences.

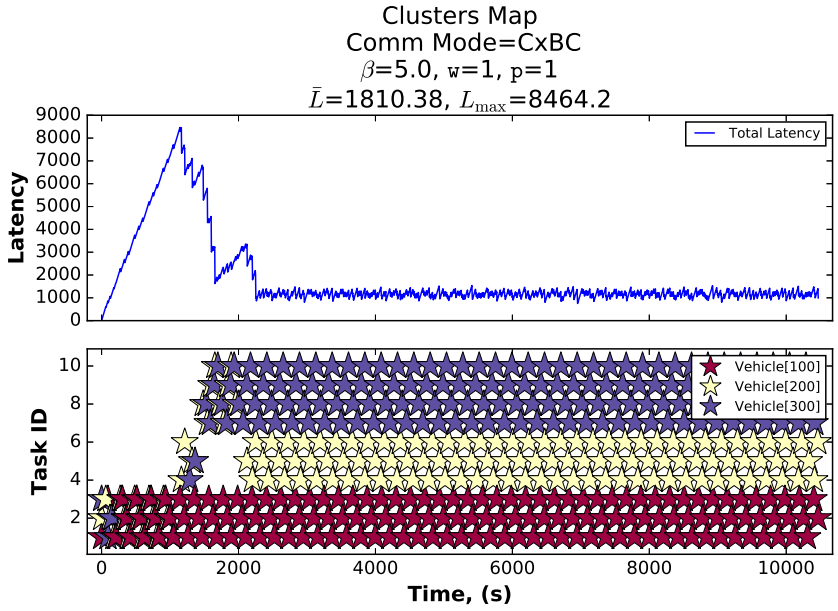


Figure 4.17. Three vehicles sharing completion data (CxBC) and starting at different tasks within the same cluster.

Of course, performance is better still if the vehicles start off in separate clusters, since vehicles begin already in their partitions. The effect of β ensures that the vehicles remain partitioned throughout the scenario.

As seen by their identical performance in Table 4.5, when the vehicles start at the same task, CxBC suffers from the same problem as CxNone; all vehicles visit the same tasks at the same time. At the start of the mission, every vehicle makes an initial task selection with all vehicles calculating utility using the same database. Since they are using identical information, they all select the same task. Upon arriving at the next task, they share which task they have completed (the same one), and again make their next selection using identical information. This is a major drawback of the CxBC mode. In the event two or more vehicles are at the same task with the

same age information (*i.e.* they enter the same state at the same time), they will "group up" for the remainder of the mission.

Despite the specific drawbacks of the CxBC mode, the automatic partitioning behavior demonstrated in Fig. 4.17 is a key benefit of MD^2WRP. The effect of β is to make each vehicle remain in its own cluster. Furthermore, utility values are never high enough to visit another cluster since the ages of those tasks are being reset by the other vehicles. If, however, one of the vehicles were to be lost, the increasing ages of the lost vehicle's tasks would eventually result in the remaining vehicles establishing a new division of tasks, after a transient period. Conversely, the introduction of a new vehicle would see a new task partition. The same could be said about the removal or addition of tasks. This adaptable automatic partitioning makes MD^2WRP ideal for uncertain, dynamic mission environments.

4.2.1.3 Broadcast Destinations (CxBD).

The destination sharing mode works as follows. After completing a task, vehicles select their next task and broadcast four pieces of information: the task ID and time of completion for the completed task plus the task ID and anticipated arrival time to the destination task. Using the destination information, vehicles are able to deconflict their task selections based on the activity of other vehicles. The CxBD mode slightly increases the complexity of the MD^2WRP task selection algorithm because the future activity of every vehicle must be accounted for when calculating task ages (see Sec. 3.4.2.4). Even so, the added complexity is compensated for with increased robustness, as the following scenario demonstrates.

The primary benefit of sharing destinations is that vehicles will automatically partition themselves even if they begin from the same state. In Fig. 4.18, all vehicles begin at Task 1 and are fully partitioned by 1000 seconds (about 17 minutes). In fact,

performance and partition time are only marginally worse for the "same task" start condition ($\bar{L} = 1308.83$) as for the "same cluster, different task" case ($\bar{L} = 1286.10$). As with the other communication methods, the best performance is achieved when the vehicles begin in separate clusters ($\bar{L} = 1156.2$).

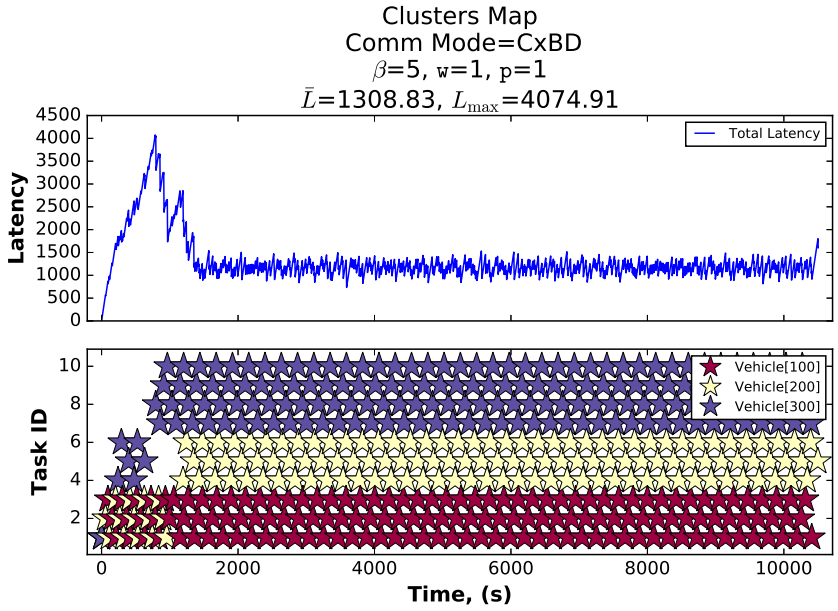


Figure 4.18. Three vehicles sharing destination data (CxBD) and starting at the same task.

The CxBD mode is the recommended form of communication and will be used for all comparisons in the remainder of this research. It allows vehicles to de-conflict their task selections while preventing vehicles from coalescing at the same tasks.

4.2.2 MD^2WRP with Multiple Decision Lookahead.

Recall that MD^2WRP is derived from the first term of the infinite horizon solution to PISR formulated as a dynamic programming problem (see Sec. 3.3.1). If the first two, or three, terms are taken from the infinite horizon solution, MD^2WRP becomes a first, or second, order approximation, equivalent to making decisions based on the

next two or three task visits. The result of a longer decision horizon should be a total tour *utility* value that approaches the optimal infinite horizon value. This is indeed the case and it is demonstrated in the first portion of this section. Later, we address whether decisions stemming from a better utility value approximation translate to better *performance*, that is reduced $\bar{L}$.

4.2.2.1 Utility from Multiple Decision Lookahead.

Recall the infinite horizon version of MD^2WRP from Eq. 3.10. It is reprinted here for convenience,

$$V^{\pi}(s_0) = \sum_{k=0}^{\infty} e^{-\beta t^{\pi}_{i_{k+1}}} w_{\pi(s_k)} \left[T(\pi(s_k)) + d(\pi(s_{k-1}), \pi(s_k)) \right].$$

First, we note a curious phenomenon that prevents the use of multiple decision lookahead when $\beta = 0$. With $\beta = 0$, the distance discount term, $e^{-\beta t^{\pi}_{i_{k+1}}}$, goes to one, such that the utility sum of all future decisions is no longer limited by a decaying exponential, but instead continues to increase with weighted task age, $w_{\pi(s_k)} \left[T(\pi(s_k)) + d(\pi(s_{k-1}), \pi(s_k)) \right]$. In other words, the longer a vehicle waits to accomplish a task, the more utility it receives! The end result is the task with the highest weighted age being continually pushed to the edge of the decision horizon, but never actually visited. Obviously this is counter to the intended behavior. So, in its current form, some value of β greater than zero must be used for MD^2WRP with multiple decision lookahead.

In Fig. 4.19, plots of the total tour utility under two different βs using 1-, 2-, and 3-Lookahead on the Random map are shown. For clarity, the tour utility curves are calculated after the fact, using Eq. 3.10, based on the final tour that was generated with a given lookahead; they do not reflect the actual utility values calculated for decisions during the simulation run. In the $\beta = 1.0$ plot, it is clear that increasing

the decision horizon results in a final tour with increased utility. Also notice that with $\beta = 1.0$ the final tour value does not plateau until about twenty decisions, but with $\beta = 5.0$ the plateau occurs in eight decisions. This is owing to the accelerated decay caused by a larger β in the exponential term of Eq. 3.10. What this means is the higher β, the less impact the steady-state has on final tour value. For example, with $\beta = 5.0$, the final visit pattern of 1-, 2-, and 3-Lookahead *are not* the same, but they have the same apparent tour utility because their first eight decisions *are* the same. The added utility of any decision beyond number eight is worth too little to appreciably impact the total tour utility.

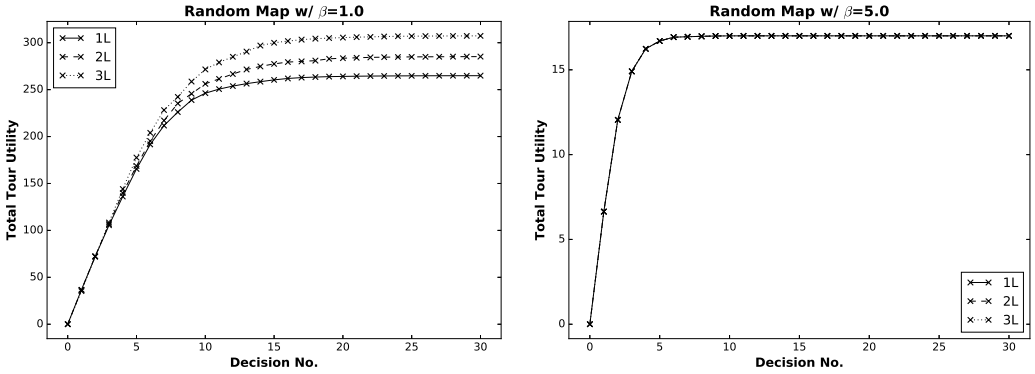


Figure 4.19. Under larger β the final tour utility approaches the limit sooner.

The final tour values under several different βs for 1-, 2-, and 3-Lookahead on the Random map are shown in Fig. 4.20. The tour values are normalized based on the largest tour value for a given β. As Fig. 4.20 depicts, a longer decision horizon always results in a better final tour utility value, but the effect is diminished as β increases.

4.2.2.2 Performance from Multiple Decision Lookahead.

Now we explore whether increasing the decision horizon through multiple lookahead translates to better performance in terms of $\bar{L}$. Of note, increasing the decision horizon changes the optimal β for a given task configuration. For example, $\beta = 8.5$

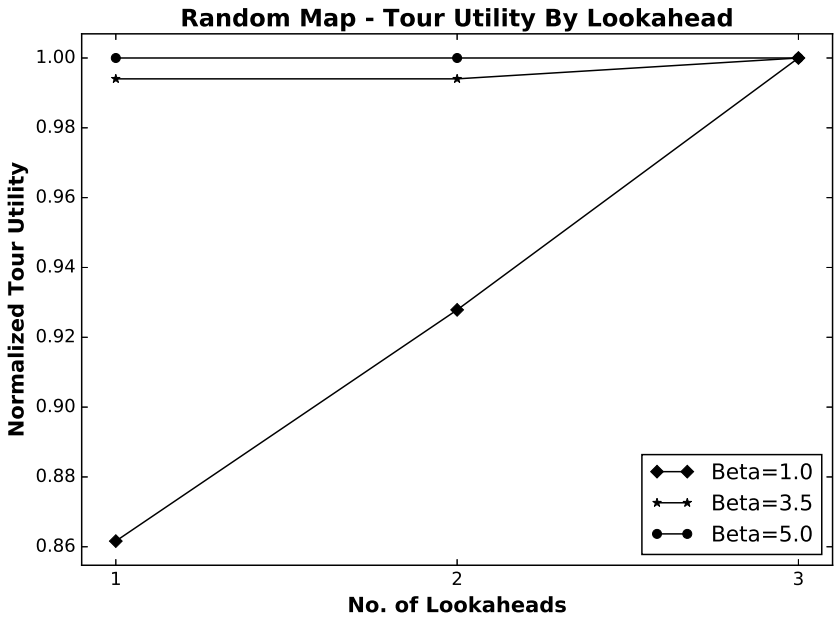


Figure 4.20. Increasing lookahead increases the final tour utility.

may be optimal with 1-Lookahead, but the optimal value for 3-Lookahead might be $\beta = 4.0$. As such, the data in Fig. 4.21 represents the latency under the optimal β for that level of lookahead.

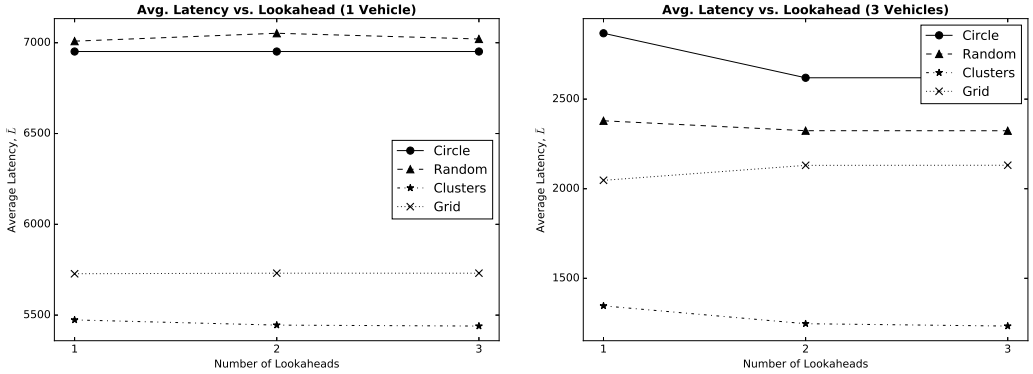


Figure 4.21. Multiple decision lookahead is more effective with multiple vehicles.

In the single vehicle case, there appears to be no correlation between multiple lookahead and improved $\bar{L}$ (left of Fig. 4.21). While 2- and 3-Lookahead deliver narrowly improved performance on the Clusters map, their performance is slightly

worse on the Random map. The extra computational cost of multiple lookahead is not recommended for a single vehicle, as it is just as likely to hurt performance as help it.

In the multi-vehicle case (right of Fig. 4.21), increased lookahead does provide a slight performance improvement. (Except in the case of the Grid map, which is likely attributable to the effect of arbitrary tie-breaking logic, which has been discussed previously.) The performance improvements due to lookahead using multiple vehicles is due to increased opportunities for task deconfliction, since a vehicle can account for the arrivals of other vehicles that would be beyond its decision horizon with only 1-Lookahead. Even so, the performance gains are marginal at best and the extra computational effort is likely not worth it. For this reason, MD^2WRP with 1-Lookahead is used in the remainder of this research for both single and multi-vehicle scenarios.

4.2.3 Comparison to TSP-based PISR.

In Chevaleyre[1] it was shown, for a single agent, that the cyclic strategy (which we refer to as n-spaced TSP) is optimal for the minimum latency tour problem. With n-spaced TSP, vehicles are evenly spaced along the same single-vehicle TSP tour and follow each other, such that every vehicle services every task. For multiple vehicles, the author goes on to demonstrate that a partitioning strategy (which we call k-subtours TSP) is better suited when the map contains one or more long edges. In this section, we compare MD^2WRP to both strategies using the four sample scenarios in Fig. 3.9. For reference, Fig. 4.22 depicts the single vehicle TSP solutions for each task map.

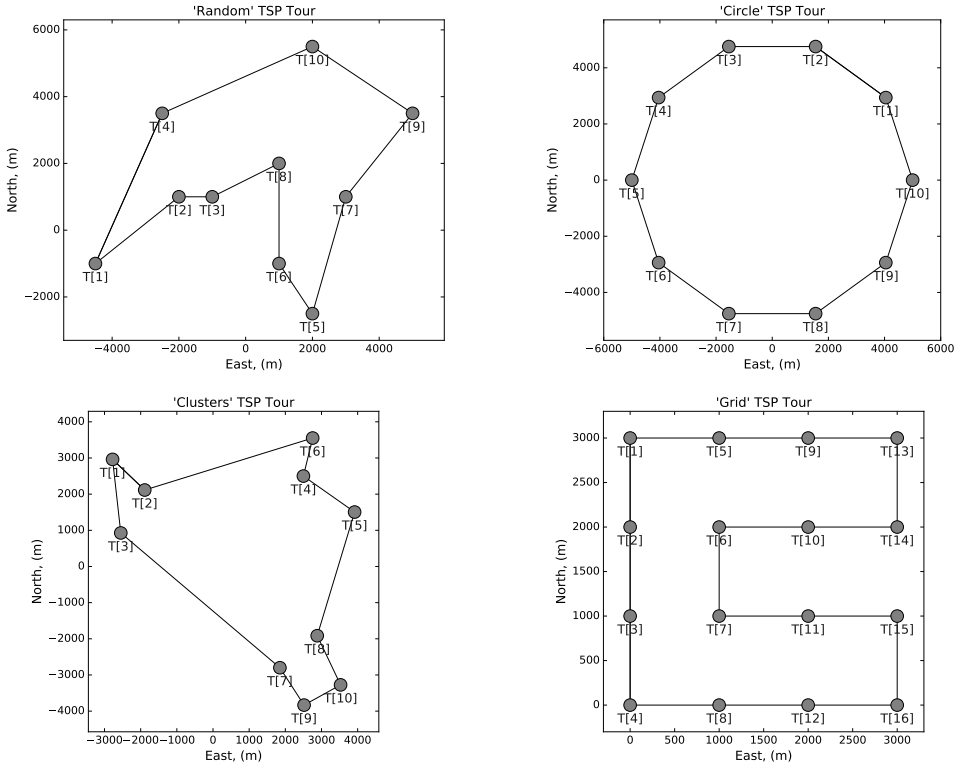


Figure 4.22. The single vehicle TSP solution for each task map.

4.2.3.1 n-spaced TSP.

The n-spaced TSP is perhaps the most intuitive approach to PISR. Here, we compare the $\bar{L}$ performance of n-spaced TSP to MD^2WRP with a varying number of vehicles. For multiple vehicles, we use the Broadcast Destinations communication mode (CxBD). The MD^2WRP parameter β has been optimized according to the process outlined in Sec. 4.1.4.1 and $\mathbf{w} = \mathbb{1}$. All task priorities are equal to one.

To maintain a fair comparison, the MD^2WRP vehicles begin in the same locations as those using n-spaced TSP with the caveat that the PUMPS tool can only handle vehicle locations that are collocated with tasks. For this reason, it is not possible to perfectly space vehicles along the TSP tour. Instead, the spacing is as close to equal as possible given the task configuration. Table 4.6 depicts the vehicle starting

locations for each of the four maps from Fig. 3.9 when one to five vehicles are used.

Table 4.6. Start locations for n-spaced TSP comparison.

Map	Start Locations
Circle	{1}, {1,6}, {1,4,7}, {1,4,6,9}, {1,3,5,7,9}
Clusters	{1}, {1,8}, {3,6,10}, {1,6,7,8}, {2,3,6,7,8}
Grid	{1}, {1,11}, {1,6,12}, {1,8,11,14}, {1,4,7,14,16}
Random	{1}, {1,7}, {1,5,10}, {2,4,5,9}, {1,4,7,8,10}

The results of the n-spaced TSP comparison are displayed in Fig. 4.23. Overall, MD^2WRP with CxBD is competitive with n-spaced TSP on all tested maps.

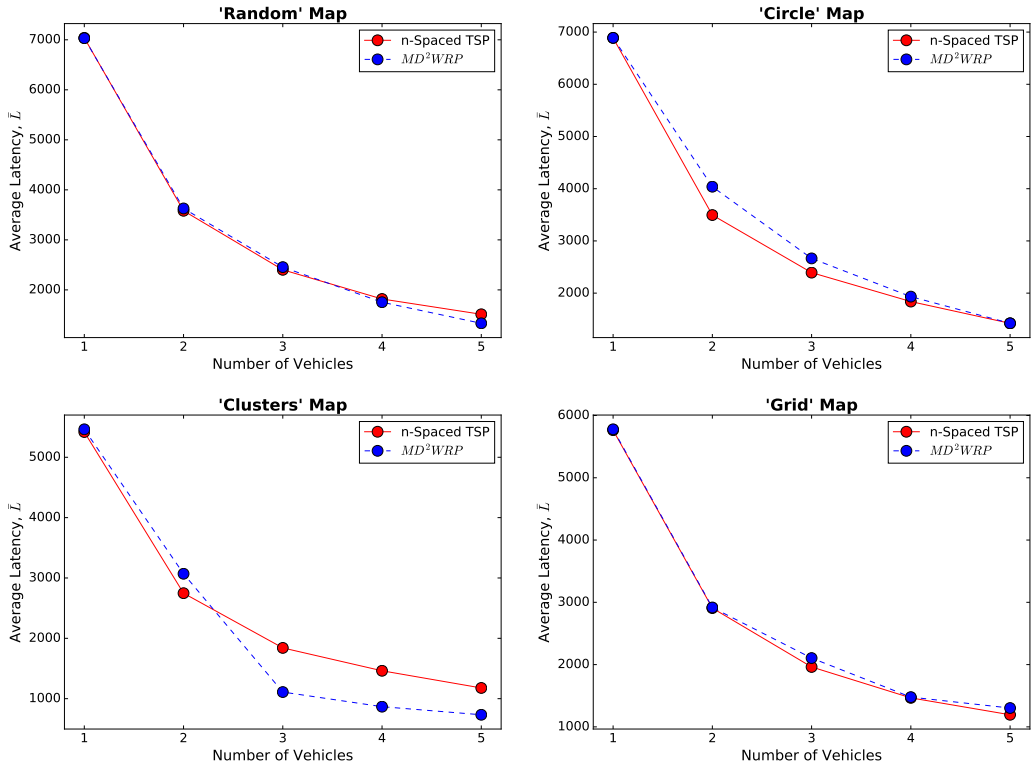


Figure 4.23. The tuned MD^2WRP is competitive with n-spaced TSP on a variety of task configurations.

Interestingly, for a single vehicle, the latency performance of MD^2WRP and n-spaced TSP is nearly equal regardless of task configuration. In fact, on the Circle, Grid, and Clusters maps, the MD^2WRP vehicle finds the TSP tour in the steady-

state with an optimal β. This is a powerful result as it demonstrates that the optimized MD^2WRP vehicle, using only utility values with 1-Lookahead for decision making, can achieve the same performance as a TSP vehicle whose path was generated with combinatorial optimization.

As the number of vehicles increases, n-spaced TSP narrowly edges out MD^2WRP except on the Clusters map when the number of vehicles is three or greater. This is due to the long edges in the Clusters map, a reflection of the Chevaleyre result mentioned previously. The automatic partitioning behavior of MD^2WRP results in at least one vehicle servicing every cluster. Hence, the number of long edges traveled between clusters is reduced, resulting in a more efficient vehicle allocation compared to the n-spaced TSP tour, in which every vehicle must travel between clusters.

The advantage of n-spaced TSP on the Circle and Grid maps with multiple vehicles is due to the arbitrary tie-breaker logic in the MD^2WRP algorithm. Take for example the Circle map with two vehicles: Vehicle 100 starts at Task 1 and Vehicle 200 at Task 6. From Task 1, Vehicle 100 receives equal utility for going to either Task 2 or 10. Since the lowest task number is given preference in the event two tasks have equal value, Task 2 is selected. Next, Vehicle 200 at Task 6 receives equal utility for going to Task 5 or 7, so it travels to Task 5. After both vehicles have arrived at their next task, Vehicle 100 at Task 2 selects a new task. Again, it receives equal value for both Task 3 and 1, since they are the same travel time and have the same age (the age of all tasks is now the time to travel from Task 1 to Task 2, t_{12}, except for Tasks 2 and 5, whose ages are now zero). Under the tie-breaker rules, Vehicle 100 chooses Task 1. Similarly, Vehicle 200 selects Task 4. From this point forward, both vehicles begin traveling around the circle clockwise. Except now, due to the tie-breaking decisions, there is only a two-task separation between the vehicles, instead of the original four task separation, which would have resulted in lower $\bar{L}$.

The n-spaced TSP method delivers good performance and is straightforward to implement. The relative simplicity makes n-spaced TSP an attractive option for PISR missions. Still, there are some limitations. For instance, the addition of different priorities among tasks will degrade n-spaced TSP performance, since a standard TSP solver has no way to include prioritized tasks (*i.e.* prioritized nodes on the graph). The MD^2WRP function, on the other hand, has the advantage of being able to adapt to priorities through manipulation of the weight vector, $\mathbf{w}$.

4.2.3.2 k-subtours TSP.

When multiple vehicles are employed, it may not make sense for every vehicle to service every task, especially when the map contains long edges[1]. Instead, transit time could be saved if the vehicles "divide and conquer", with each vehicle servicing a subset of tasks. This is the motivation for the k-subtours approach, where the map is divided into k clusters with k the number of vehicles. Each vehicle then travels a small TSP tour within its assigned cluster.

To generate the clusters, we use Matlab's k-means++ function from the Statistics and Machine Learning Toolbox. Use of k-means++ and k-means clustering as tools for generating vehicle tours in PISR are described in Sec. 2.4.1. We initialize the k-means++ algorithm 1,000 times and select the best local optimum found as the clustering solution (see Table 4.7).

The k-subtours and MD^2WRP vehicles once again begin in the same locations (*i.e.* the first task in each subtour from Table 4.7) and β is optimized while $\mathbf{w} = \mathbb{1}$. All priorities are equal. The $\bar{L}$ of MD^2WRP and k-subtours are shown for each map with one to four vehicles in Fig. 4.24. Note that the n-spaced and k-subtours approach are equivalent with a single vehicle.

Performance on the Random map is competitive, with a slight advantage to

Table 4.7. Partitions for k-subtours TSP comparison, generated with k-means++.

Map	No. of Veh.	Partitions
Circle	1	{1,2,3,4,5,6,7,8,9,10}
	2	{1,2,3,4,5}, {6,7,8,9,10}
	3	{1,2,3,4}, {5,6,7}, {8,9,10}
	4	{1,2,3}, {4,5,6}, {7,8}, {9,10}
Clusters	1	{1,2,6,4,5,8,10,9,7,3}
	2	{1,2,6,4,5,3}, {7,8,10,9}
	3	{1,2,3}, {4,5,6}, {7,8,10,9}
	4	{1,2,3}, {4,5,6}, {7,8}, {9,10}
Grid	1	{1,2,3,4,8,12,16,15,11,7,6,10,14,13,9,5}
	2	{1,2,3,4,8,7,6,5}, {9,10,11,12,16,15,14,13}
	3	{1,2,6,9,5}, {3,4,8,12,7}, {10,11,16,15,14,13}
	4	{1,2,6,5}, {3,4,8,7}, {9,10,14,13}, {11,12,15,16}
Random	1	{1,4,10,9,7,5,6,8,3,2}
	2	{1,2,3,4}, {5,6,8,10,9,7}
	3	{1,2,3,4}, {5,6,8,7}, {9,10}
	4	{1,2,3,4}, {5,6}, {7,8}, {9,10}

MD^2WRP. MD^2WRP naturally minimizes the time spent traversing long edges, due to the effect of β. The vehicles develop "loose" partitions, generally staying in their own region but occasionally "sharing" tasks with other vehicles (see Fig. 4.25 for an example in the two-vehicle case). Also, MD^2WRP vehicles are not limited to visiting each task only once per loop, as with k-subtours, which creates some performance gains.

In the Circle scenario, MD^2WRP outperforms k-subtours in every case except with a single vehicle, when both methods have the same performance. This is due to MD^2WRP adopting a cyclic visit pattern, with vehicles following each other around the circle as in the n-spaced method. For the Circle map, the cyclic method is more efficient than the partition method. This highlights the adaptability of MD^2WRP to changes in the number of tasks or vehicles. The agents are able to adapt to the most efficient coordination method without the need for explicit vehicle assignments.

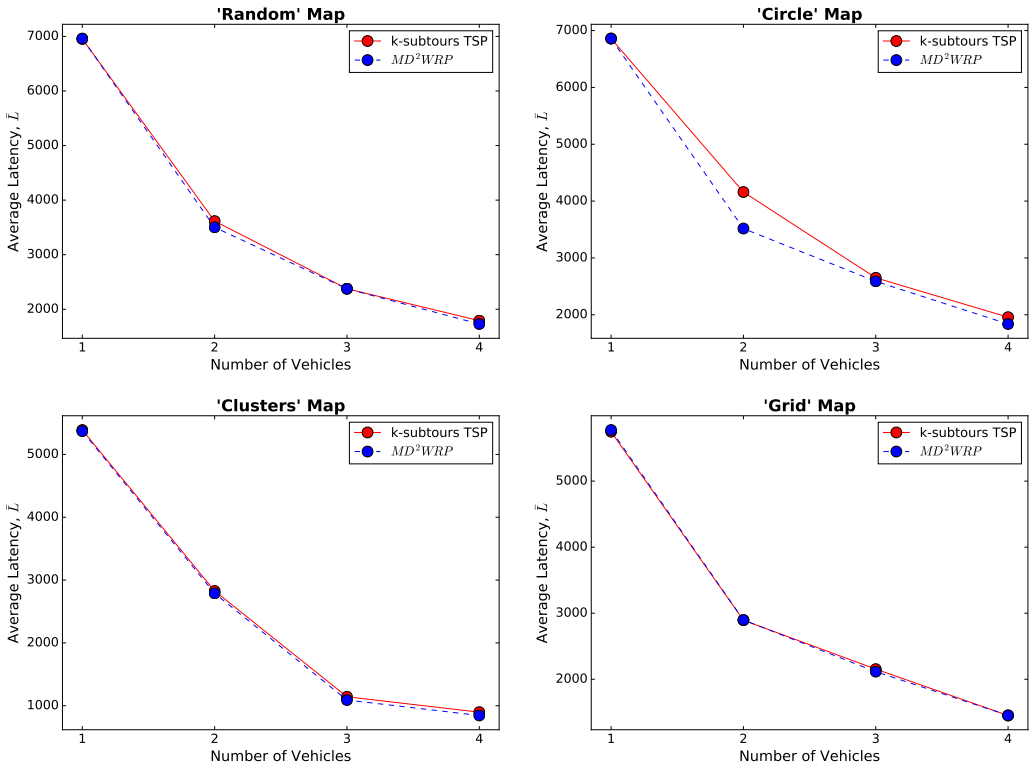


Figure 4.24. The tuned MD^2WRP consistently meets or exceeds the performance of k-subtours TSP.

On the Clusters map, k-subtours closes the performance gap on MD^2WRP where the n-spaced method fell short. The Clusters map contains several long edges, which n-spaced is ill-equipped to handle. From Fig. 4.24, the two methods are nearly equal for all numbers of vehicles with the map being partitioned in roughly the same way.

Finally, on the Grid map as with Clusters, the performance of both methods is nearly equal. Vehicles partition the tasks in similar ways.

The k-subtours TSP method provides a good alternative to n-spaced TSP, especially in situations where long edges make it desirable to divide responsibility for tasks among vehicles. Ideally, both methods would be available based on the particular scenario at hand. Just as with n-spaced, however, the partitions for k-subtours must be calculated offline and distributed to vehicles. Changes to the task configuration or the

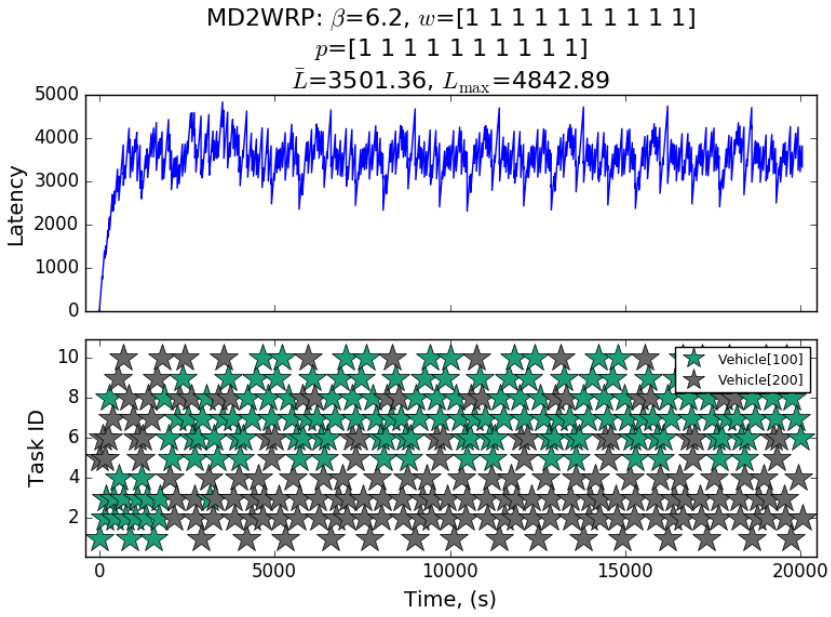


Figure 4.25. Two MD^2WRP vehicles on the Random map mostly divide the tasks between vehicles, but occasionally share tasks.

number of vehicles will require the calculation and distribution of new assignments. MD^2WRP, on the other hand, has the flexibility to cycle through tasks, partition them among vehicles, or adopt a different visit pattern altogether based on which provides the best performance.

4.2.4 Comparison to Other Utility-based PISR.

4.2.4.1 Direct Latency Minimization.

The Direct Latency Minimization (DLM) utility function was developed by the author for this research. It is a simple greedy algorithm that attempts to select tasks that will minimize total system latency. It calculates, for every candidate task, what the total system latency would be when the vehicle arrives. Whichever destination results in the lowest system latency is selected as the next task. Stated mathematically,

$$V = \min_j \sum_{k=1}^{n} p_k(T_k + t_{ij})), \qquad k \neq j, \forall j \in \{1, \ldots, n\} \tag{4.17}$$

where V is the value of the selected task, p_k is the priority of task k, T_k is the age of task k, and t_{ij} is the time to travel from the current location at task i to j. Note when the agent leaves task i, i will incur a latency cost during the transit to j, but the latency of j becomes zero when the vehicle arrives.

Though DLM is a more direct approach to maximizing PISR performance than MD^2WRP, it comes at the cost of more operations. Whereas MD^2WRP with 1-Lookahead requires $n-1$ operations per decision, DLM is on the order of n^2.

Before comparing DLM to MD^2WRP, a brief characterization of DLM is presented. In Fig. 4.26, the performance of a single vehicle using DLM is shown for four maps with varying degrees of lookahead. In general, performance improves as lookahead increases. Three decision lookahead provides the best performance in almost all cases but, owing to the n^2 nature of DLM, becomes computationally expensive with even a moderate number of tasks. In the PUMPS tool, 3-Lookahead on the Grid map of 16 tasks requires almost five minutes per decision, compared to less than 15 seconds per decision with 2-Lookahead. Thus, 2-Lookahead provides the best value in terms of the performance to computational cost ratio and will be the version of DLM implemented in the comparison to MD^2WRP below. Vehicles also communicate under DLM using the Broadcast Destinations (CxBD) communication scheme.

Figure 4.27 compares the performance of MD^2WRP to 2-Lookahead DLM for a varying number of vehicles across four maps. Vehicles always begin at Task 1. The MD^2WRP performance data was gathered under the previously recommended parameters, that is, using the CxBD communication mode, optimized β, and $\mathbf{w} = \mathbb{1}$. In most cases, MD^2WRP outperforms DLM. This is somewhat surprising since $\bar{L}$ minimization via MD^2WRP is a by-product of maximizing utility, whereas DLM

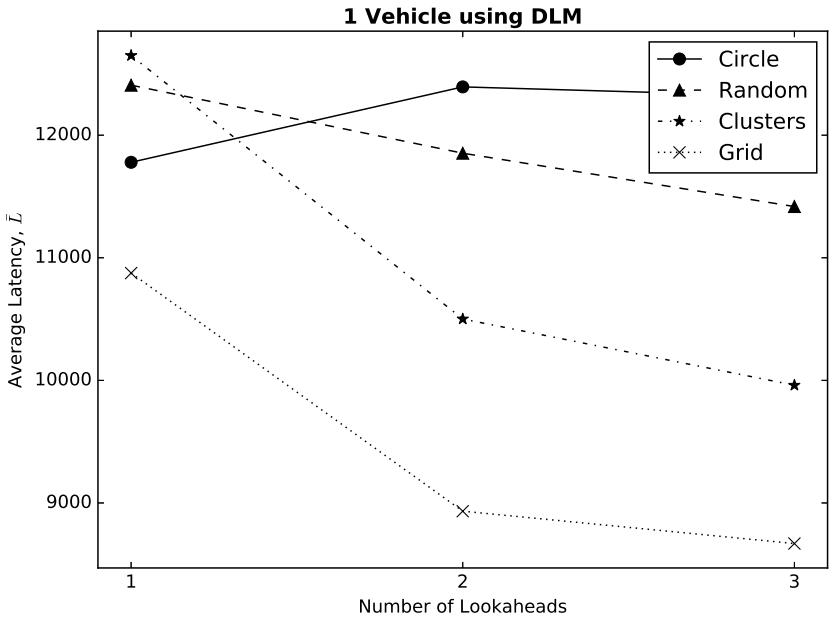


Figure 4.26. In most cases, the DLM utility function improves with an increasing decision horizon.

deliberately attempts to minimize latency. One should bear in mind, however, that MD^2WRP must be optimized to realize the best performance. With DLM, all an agent needs is a priority vector in order to work.

In Fig. 4.28, the latency curves and visit histories are provided for two data points from Fig. 4.27, specifically one point for each utility function from the Random map with 2 vehicles. These plots shed some light as to why the optimized MD^2WRP generally outperforms DLM. Notice that DLM has a high density of visits to Tasks 2 and 3, which are in close proximity to each other. These successive visits are a result of the algorithm's greedy nature. The vehicles find that successive visits between Tasks 2 and 3 result in minimal increases to system latency, as opposed to making lengthy trips to other tasks which would cause all task latencies to rise substantially, thus increasing system latency. However, the short-term gains of visiting Tasks 2 and 3 have a secondary effect with regards to the increase in total latency caused by the

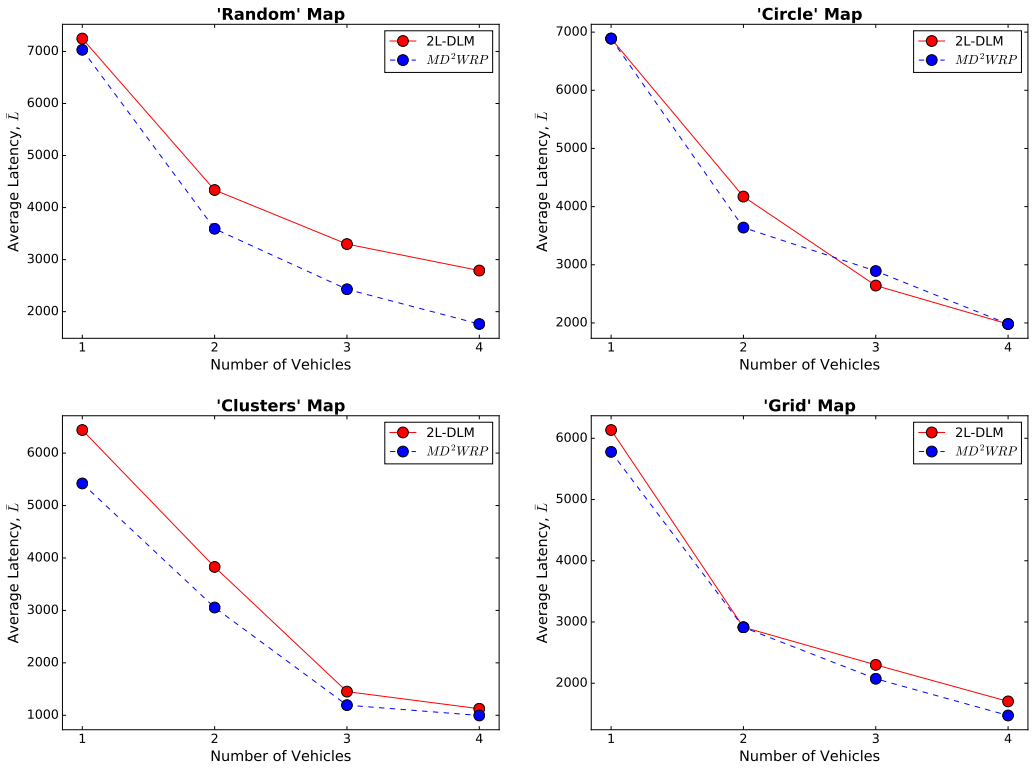


Figure 4.27. In most cases, the optimized MD^2WRP outperforms DLM.

increasing ages of the other tasks. Eventually, a point is reached where the ages of all other tasks have become so large that the vehicles find the best decision to minimize total latency is to reset their ages with visits.

Contrast the clustered task visit sequence of DLM with the relatively even distribution of visits in the MD^2WRP plot. Even though MD^2WRP is not directly attempting to minimize latency, it results in better system performance because it avoids the immediate reward pitfalls that are characteristic of greedy search.

Additionally, the DLM vehicles do not appear to exhibit the same kind of emergent cooperative behavior as the MD^2WRP vehicles. With MD^2WRP, in general, one vehicle services Task 1-4 and the other services Tasks 5-10, though there is some overlap. DLM, however, does not have this behavior. Both vehicles service all tasks throughout the simulation period.

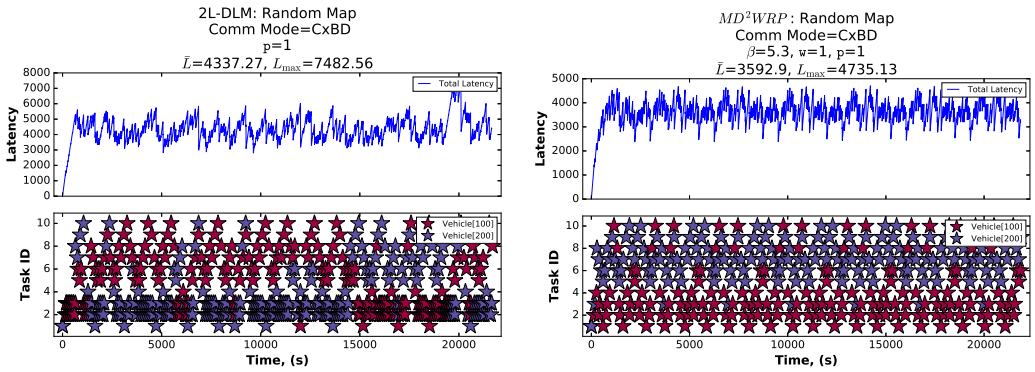


Figure 4.28. The MD^2WRP vehicle has a more evenly distributed visit history.

Despite its performance drawbacks, the DLM utility function has some attractive features. No optimization is required, so the operator can be certain of consistent performance despite changes to the task configuration or number of vehicles. Still, in most respects, MD^2WRP appears to be the better choice for PISR missions. It generates better latency performance while requiring less evaluations per utility function call.

4.2.4.2 The Stanford Single/Multi-Vehicle Reactive Policy.

In Ch. II, we referenced the work of Nigam and Kroo of Stanford University[5, 30], which proposed a utility function for PISR task selection. In this section, we implement modified versions of the Nigam and Kroo functions. Their single vehicle policy (which we refer to as the Single-Vehicle Reactive Policy, or SRP; the authors do not provide a name for their single vehicle policy) considers task age and weighted travel distance,

$$V = \max_j \{T_j + w_0 \delta_{ij}\}, \qquad \forall j \in \{1 \ldots, n\} \tag{4.18}$$

where V is the value of the selected task, T_j is the age of candidate task j, δ_{ij} is the distance between current task i and j (in m), and w_0 is a weight parameter with units of s/m. Note that w_0 must be negative, such that shorter travel distances are

preferred. The Multi-Vehicle Reactive Policy (MRP), was also introduced,

$$V = \max_j \{T_j + w_0 \delta_{ij} + w_1 \min_{k \neq i}(\delta_{kj})\} \qquad \forall j, k \in \{1, \ldots, m\} \tag{4.19}$$

where w_1 is an additional positive weight parameter (units of s/m) and δ_{kj} is the distance between the kth vehicle and task j, in m.

Both SRP/MRP and MD^2WRP use age as a basis for selecting tasks. They also consider travel distance, though their implementations differ. Whereas SRP/MRP directly applies a negative utility that increases linearly with travel distance, MD^2WRP uses t_{ij} in an exponential function to apply a value discount. They also differ in their approach to facilitating cooperation between vehicles. The SRP/MRP directly rewards vehicles for selecting tasks that are far away from other vehicles. On the other hand, MD^2WRP relies upon the exponential discount factor β to indirectly encourage separation of vehicles.

Before proceeding, a brief discussion on how SRP/MRP was implemented for this research is provided. The sole sources for reconstructing SRP/MRP were [5] and [30]. In those papers, vehicles were given a sensor radius and tasks were located within a network of cells. For this work, sensor radius is ignored. Tasks are serviced only when a vehicle is exactly collocated with a task. In turn, instead of a network of cells, tasks are located discretely in space; it is not possible for a vehicle to exist between tasks. The discrete nature of the simulation does not affect the calculation of task ages in Eqs. 4.18 and 4.19, but it does alter how the distance between target tasks and other vehicles, δ_{kj}, is calculated in Eq. 4.19. Nigam and Kroo state that to find δ_{kj}, "the UAVs need to know positions of all UAVs at all time steps". Since our simulator cannot determine the current location of vehicles that may be transiting between tasks during a given decision, the definition of δ_{kj} was altered to instead reflect the distance between the target task, j, and the current *destination task* of

vehicle k. In this way, MRP can utilize the same CxBD communication mode as MD^2WRP, adding consistency to the comparison. This implementation of δ_{kj} has the added benefit of requiring less information sharing between vehicles, since it is not necessary to receive an update on the location of every vehicle for each decision. Instead, each vehicle already knows the destination of every other vehicle due to CxBD. Also, it arguably provides better vehicle separation, since the location of the other vehicles is not as important as where they are going.

Nigam and Kroo also mention that $-1/V$, where V is the velocity of the vehicle, is a good place to start for optimizing the values of w_0 and w_1, though they acknowledge that for multiple tasks and vehicles these values are not necessarily optimal. They utilize an Iterative Sampling optimizer, which they developed, to determine optimal values of w_0 and w_1. We instead use a simple brute force method, performing numerous simulations across a linear distribution of w_0 and w_1 values to find the best combination. The authors also acknowledge that "the weights for different policies are allowed to be different, resulting in different policies for each UAV" and that "the optimization thus needs to be conducted for different number of UAVs too". For simplicity in this comparison, all vehicles use the same weight parameters, acknowledging that slight performance improvements may be possible by allowing each vehicle to use different weights. This assumption is based on the authors' own comment that "ideally, the weights would need to be re-optimized when the mission specifications (*e.g.*, the number of UAVs or size of target space) change, but the sensitivity of mission performance to such changes, using a fixed set of weights in found to be small."[30]. (Though not explored in this research, MD^2WRP may also be able to achieve better performance if each vehicle were allowed to use a different value for β and $\mathbf{w}$). Additionally, the authors indicate that w_0 should be a negative value, which makes intuitive sense as the longer travel distance becomes, the less utility should be avail-

able. However, they also seem to indicate that w_1 be negative. This is assumed to be an error, since a negative w_1 would result in larger negative utility for tasks far away from other vehicles, while tasks close to other vehicles would be less negative, making them preferred. In simulations to test the functionality of MRP, positive values of w_1 resulted in the intended effect of vehicles maintaining maximal spacing. Finally, the authors describe SRP/MRP as selecting the maximum of either calculated utility or zero, presumably to avoid negative utility. For this work, if all utility values are negative, the vehicle selects the least negative utility. This avoids a situation where all utility values become zero and a vehicle must resort to tie-breakers.

With the implementation described above, the SRP and MRP are simulated on the Random, Clusters, Circle, and Grid maps (Fig. 3.9) with a varying number of vehicles. All vehicles start at Task 1 for each simulation and all task priorities are equal. The latency performance is compared against MD^2WRP on the same scenarios in Fig. 4.29. The displayed data reflects the use of the optimized weight parameters (w_0, w_1) for SRP/MRP and optimal β for MD^2WRP with $\mathbf{w} = \mathbb{1}$.

Overall, the latency performance of SRP/MRP and MD^2WRP is comparable across all of the tested task configurations, with a slight advantage alternating between the two. Both functions exhibit automatic partitioning, though their specific task allocations, and the efficiency of those allocations, are not always the same given the same map and number of vehicles. For instance, on the Clusters map with three vehicles under SRP/MRP, the vehicles separate with each taking responsibility for its own cluster slightly quicker than with MD^2WRP. But on the Grid map with 3 vehicles MD^2WRP results in 3 distinct partitions (two 3x2 rectangles and a 1x4 line) while SRP/MRP creates no partitions at all (Fig. 4.30). Conversely, SRP/MRP with three vehicles on the Random map develop three distinct partitions while MD^2WRP does not.

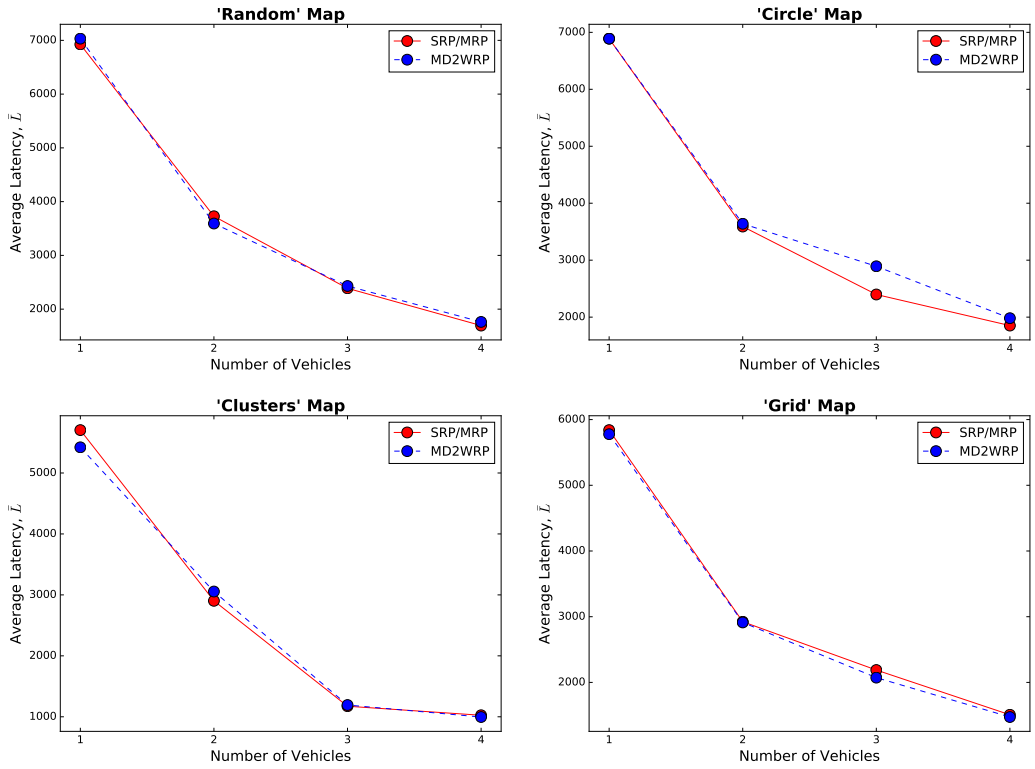


Figure 4.29. SRP/MRP and MD^2WRP deliver similar latency performance across all four maps.

Despite their different approaches to distance discounting and vehicle separation, SRP/MRP and MD^2WRP deliver similar performance when simulated on the same scenario. One advantage of MRP is that it explicitly rewards vehicles for maintaining separation, so the operator can be sure that vehicles are doing their best to remain in separate areas of the operational region. Conversely, vehicle spacing from MD^2WRP is implicit and stems solely from β and the fact that vehicles share task age information. While optimizing the two weight parameters in SRP/MRP was found to be a quick and straight-forward process, MD^2WRP requires optimizing only the single parameter, β. Along the same lines, β is dimensionless due to normalization. This makes it easier to select β regardless of the task configuration. The SRP/MRP weight parameters, w_0 and w_1, have units of s/m, so their optimal values may change when

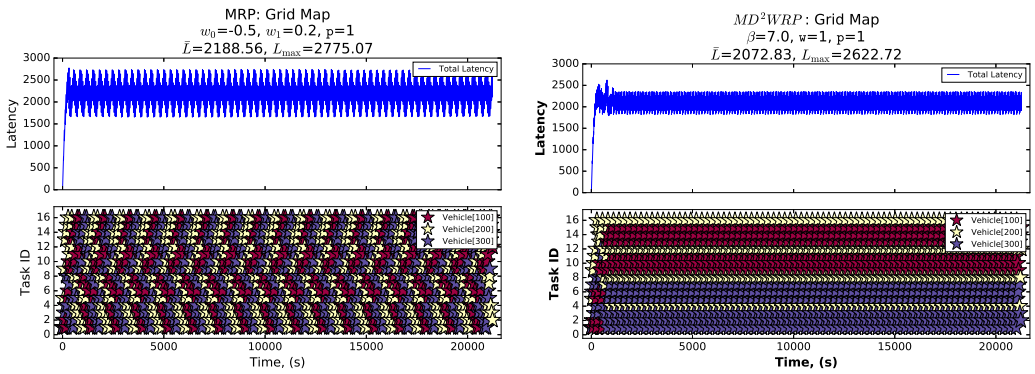


Figure 4.30. The MD^2WRP vehicles develop distinct partitions, although the latency performance is about the same.

the distance matrix, D, is multiplied by a scalar.

4.3 MD^2WRP and Operational Factors

In this section, we investigate how certain operational factors influence the behavior and performance of MD^2WRP vehicles, as well has how MD^2WRP can be adjusted to compensate for each factor. Four factors are addressed: Dubins constraints on vehicle motion, no-fly zones, return-to-base requirements, and the addition/removal of tasks/vehicles mid-mission.

4.3.1 Dubins Constraints on Vehicle Motion.

All results presented to this point have been with the assumption that vehicles move between tasks along Euclidean paths. This is a good approximation of vehicle motion so long as the distances between tasks are large relative to the turning radius of the vehicle. As the distance between tasks decreases, however, vehicle kinematics play an increasingly important role in calculating the travel times between tasks, which affects vehicle decision making under the MD^2WRP function. The goal of this section is to characterize how the Dubins path assumption influences vehicle

behavior and to determine when travel times based on Euclidean distances are no longer appropriate.

4.3.1.1 Characterization of MD^2WRP with Dubins Paths.

We begin with a simple scenario consisting of two tasks spaced $1000m$ apart. The vehicle's maximum bank angle is 30 degrees, which results in a minimum turning radius of $85m$ at a velocity of $22m/s$. The vehicle starts at Task 1 with an initial heading of 0 degrees (due east). We wish to explore variations in β while keeping $\mathbf{w} = \mathbb{1}$. Ten trades are conducted with β incrementing from 0.0 to 0.225. For this first example, the agent uses the non-normalized version of MD^2WRP for decision making (Eq. 3.2). This will emphasize the sensitivity of β to the task configuration and again motivate the need to normalize, especially under Dubins constraints. The results are shown in Figs. 4.31 and 4.32, which depict the mean visit rates to teach task for each trade and the task visit sequence for Trades 1003 and 1009.

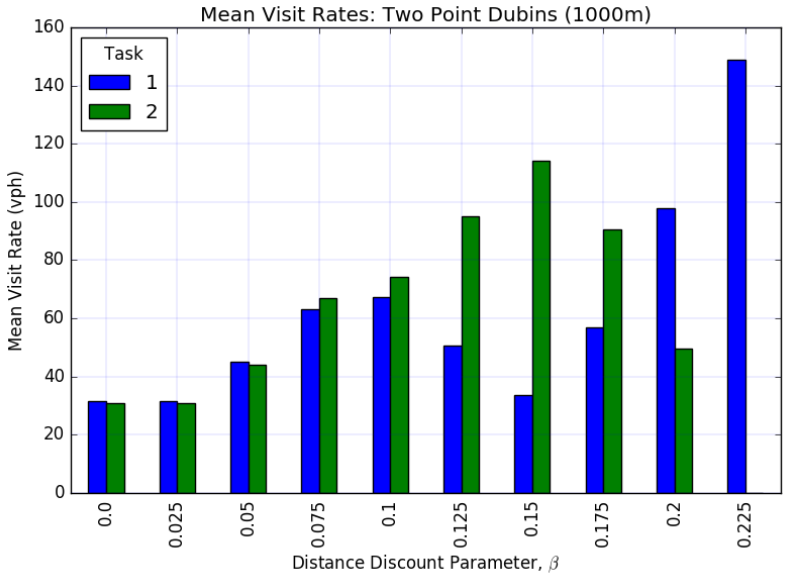


Figure 4.31. Visit rates between two tasks with Dubins motion as β increases.

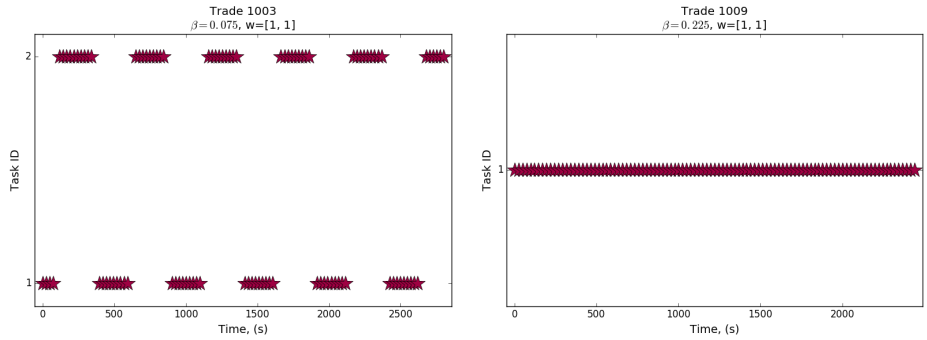


Figure 4.32. Task visit times for two trades of the two-point Dubins scenario (1000m spacing).

Unlike Euclidean paths, under Dubins it is possible for a vehicle to visit the same task twice in a row, since the future age term, $(T_j + t_{ij})$, is not automatically zero for a revisit. While T_j is still zero, the maximum turning radius makes t_{ij} non-zero. Therefore, it is possible for β to be large enough such that the UAV continually visits the same task for the entire mission. This is the case in Trade 1009 when $\beta = 0.225$ (right of Fig. 4.32). Over the course of 100 decisions, the UAV chooses to visit Task 1 every time.

The second interesting takeaway from the two-point Dubins scenario is that the visit *rate* to two tasks is essentially equal under a given β, even though the UAV may visit a single task multiple times in a row. This is shown by the equal blue and green bars in each trade of Fig. 4.31 and by the alternating visit patterns in Fig. 4.32. The apparent inequalities in visit rates are merely due to the simulation cutoff at 100 visits, which gives the appearance that one task is visited more often than another. Also note that the *absolute visit rate* increases as β increases, meaning the vehicle spends more time servicing tasks and less time in transit. This is reinforced by the fact that Trade 1003 completes 100 visits in just under 3000s, whereas Trade 1009 only takes about 2500s to service the same number of tasks (Fig. 4.32). The trajectory plots in Fig. 4.33 also illustrate this point. With Trade 1003, time is consumed as

the UAV traverses between tasks, whereas in Trade 1009 the UAV performs several shorter loop-backs of the same task.

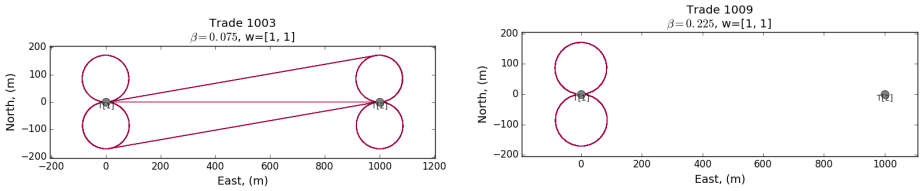


Figure 4.33. The flight trajectories for select trades of the two-point Dubins scenario.

4.3.1.2 Normalization with Dubins Paths.

Next, we wish to observe the effect of using the normalized MD^2WRP from Eq. 3.16 under Dubins constraints. Normalization in the Dubins case is interesting because, unlike the isosceles triangle examples with Euclidean paths from Sec. 4.1.1, two configurations with the same geometric ratios between tasks are in fact two different problems. In other words, they will yield different task visit sequences even if the vehicle uses the same β. This is attributed to the changing ratio of turn radius to $t_{ij,max}$ when a scalar multiplier is applied to the distance matrix, D.

As an example, we present the two-point scenario from above, but use Eq. 3.16 to make task selections. The results are shown in Figs. 4.34, 4.35, and 4.36 (which show the mean visit rates, task visit histories, and vehicle trajectories, respectively). Though the range of non-trivial β values has changed, the vehicle's general behavior is the same. When β is small, the vehicle alternates more frequently between tasks since the distance discount is minimal. As β grows, the vehicle begins repeating the same task more often before moving on.

Next, the distance between the two tasks is increased from $1000m$ to $5000m$ and we again use the normalized MD^2WRP utility function. The visits per hour for each trade is depicted in Fig. 4.37 and the task visit times in Fig. 4.38.

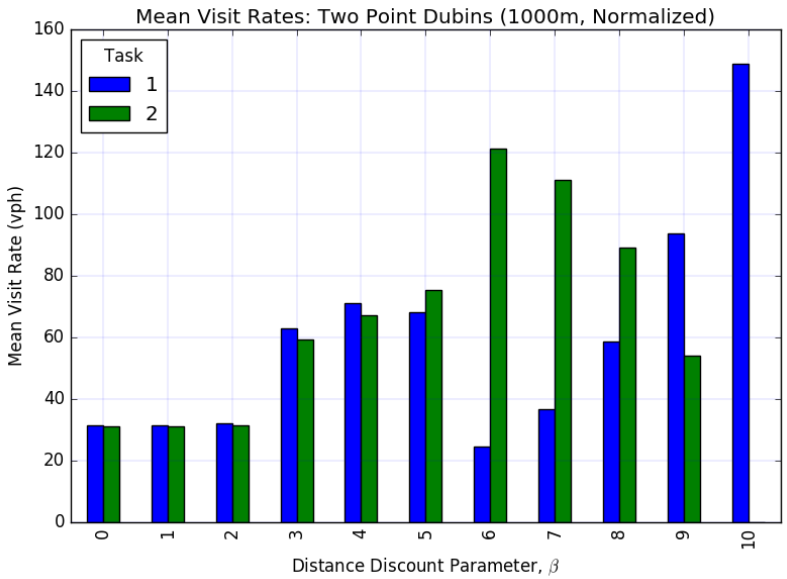


Figure 4.34. Visit rates between two tasks as β increases using normalized MD^2WRP with Dubins motion.

With the increased spacing between tasks, the non-trivial β range has changed. It is now approximately $3.00 \leq \beta \leq 5.25$. The shift in β is due to the change in the ratio of turn radius to $t_{ij,max}$. With a turn radius of 85m, the path distance for visiting the same task (a circle) is 534m. With normalization and $1000m$ spacing, this means the travel time to the same task is 0.534 versus 1.000 to travel to the distant task. For $5000m$ spacing, normalized travel time to the same task is now only

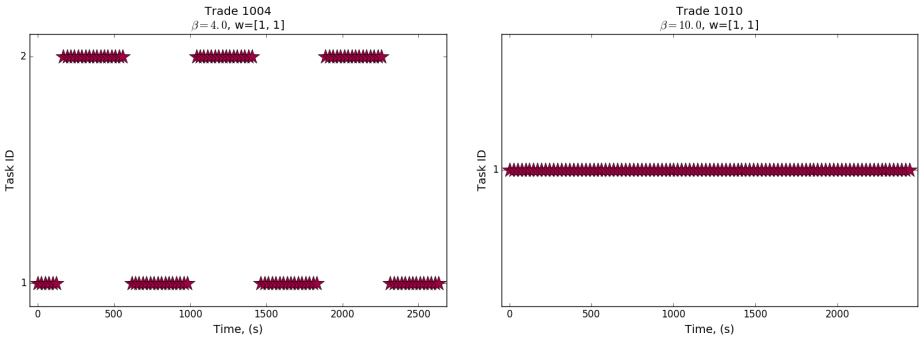


Figure 4.35. Task visit times for select trades of the two-point Dubins scenario under normalized MD^2WRP ($1000m$ spacing).

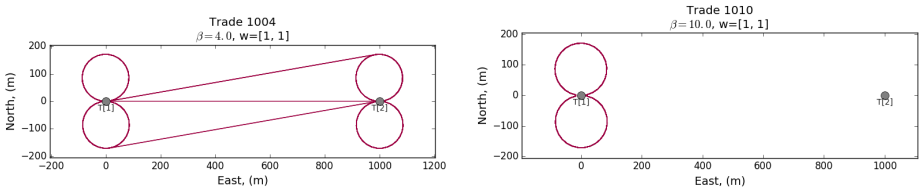


Figure 4.36. The flight trajectories for select trades of the two-point Dubins scenario under normalized MD^2WRP (1000m spacing).

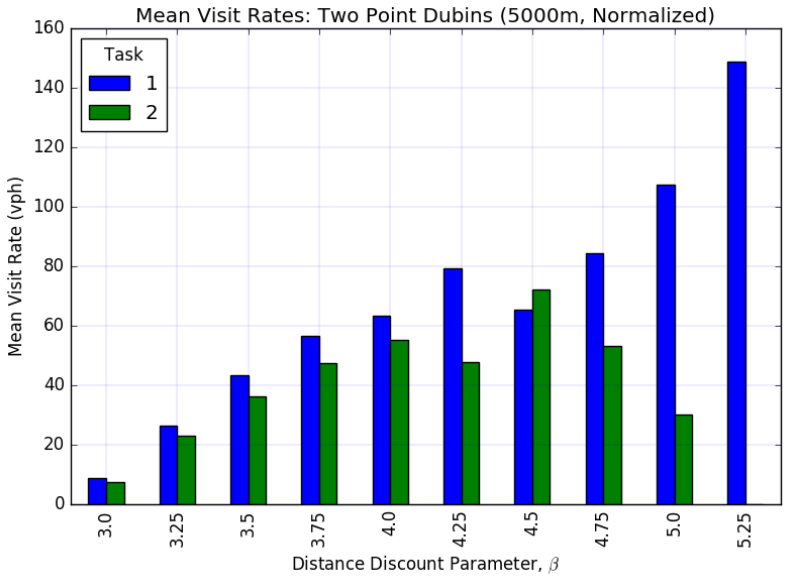


Figure 4.37. Visit rates between two tasks as β increases using normalized MD^2WRP with Dubins motion (5000m spacing).

0.107 while the distant travel time is still 1.000. Hence, it takes a larger β (> 3.0) to make revisiting the same task preferred over the longer trip. On the upper end, we see that it only requires $\beta = 5.25$ to make the vehicle never leave Task 1. This is caused by the relatively short travel time to Task 1, which results in a slow build up of age for Task 2. The simulation cutoff at 100 task visits prevents Task 2 from achieving a value greater than that of Task 1.

One final note regarding the Dubins path case. Recall that the normalizing value of $t_{ij,max}$ from Eq. 3.16 is based on the Euclidean distance between the two most distant tasks, such that traveling in a straight line between these two tasks gives

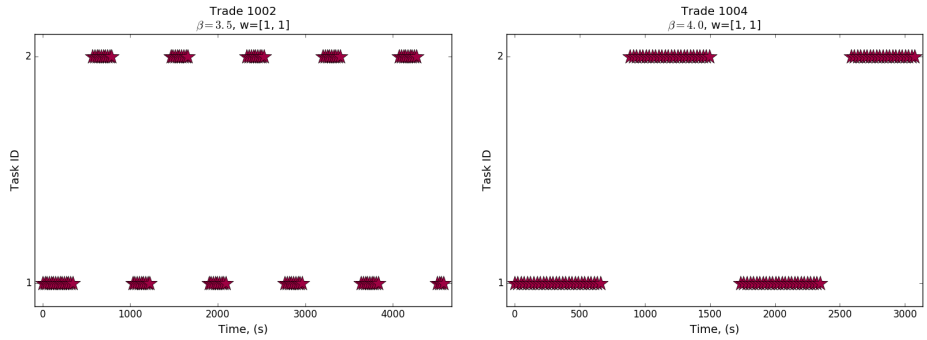


Figure 4.38. Task visit times for select trades of the two-point Dubins scenario under normalized MD^2WRP (5000m spacing).

$t_{ij} = 1.0$. Under Dubins constraints, it is possible that the true travel time between tasks can be greater than 1.0, due to turning. As the ratio of turn radius to $t_{ij,max}$, increases, the more t_{ij} may exceed 1. Therefore, the relative density or sparsity of task distribution under Dubins constraints also has an effect on the viable range of β values as well as performance implications, which is discussed next.

4.3.1.3 The Ratio of Turn Radius to t_{ij}.

In this section we consider two different measurement types for the distance between tasks, which are in turn used to calculate the value of t_{ij} in MD^2WRP. If we consider a case where the distance between tasks is large compared to the turning radius, the vehicle kinematic constraints add a negligible amount of travel time between tasks. In this case we can simply use the Euclidean distance. This is the preferred method of measuring distance because it only requires selection of a matrix entry, taking almost zero on-board computational resources. However, if the distance between tasks is equal to the turning radius, kinematics become an important player and using Dubins paths to measure distance is more accurate. Between these two extremes, there must exist a transition point where the Euclidean travel assumption becomes invalid.

To find the transition, we use a vehicle with a turn radius of 85m (a velocity of $22m/s$ and maximum bank angle of 30 deg) and the four maps from Fig. 3.9. We scale the maps by varying amounts to change the ratio of r/d, that is, the ratio of turn radius to average distance between tasks. We then simulate each case of r/d twice; first using Euclidean paths to calculate travel time and then using Dubins paths. The results for a single vehicle and three vehicles are displayed in Figs. 4.39 and 4.40.

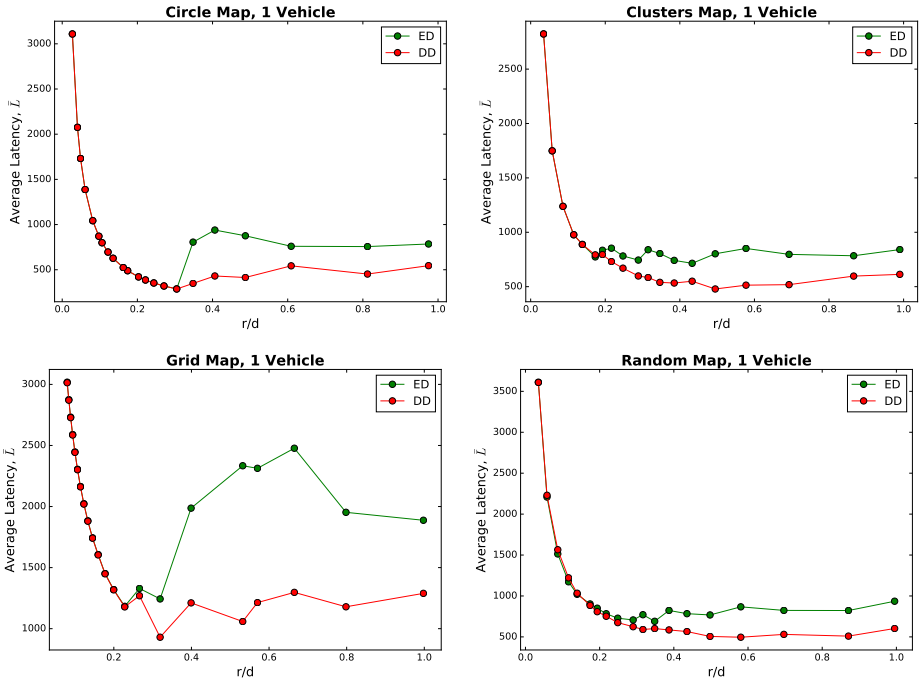


Figure 4.39. Comparison of performance using Euclidean distance versus Dubins path distance for a single vehicle.

Whether a single vehicle is used, or three vehicles, we see a bifurcation when r/d is between approximately 0.2 and 0.3 in all maps. In other words, if the vehicle turn radius is less than about 25% of the average distance between tasks, then there is little difference in performance between Euclidean and Dubins measurements. However, as the turn radius grows beyond 25% of average task separation, Dubins path measurements begin to outperform Euclidean, eventually becoming significantly better.

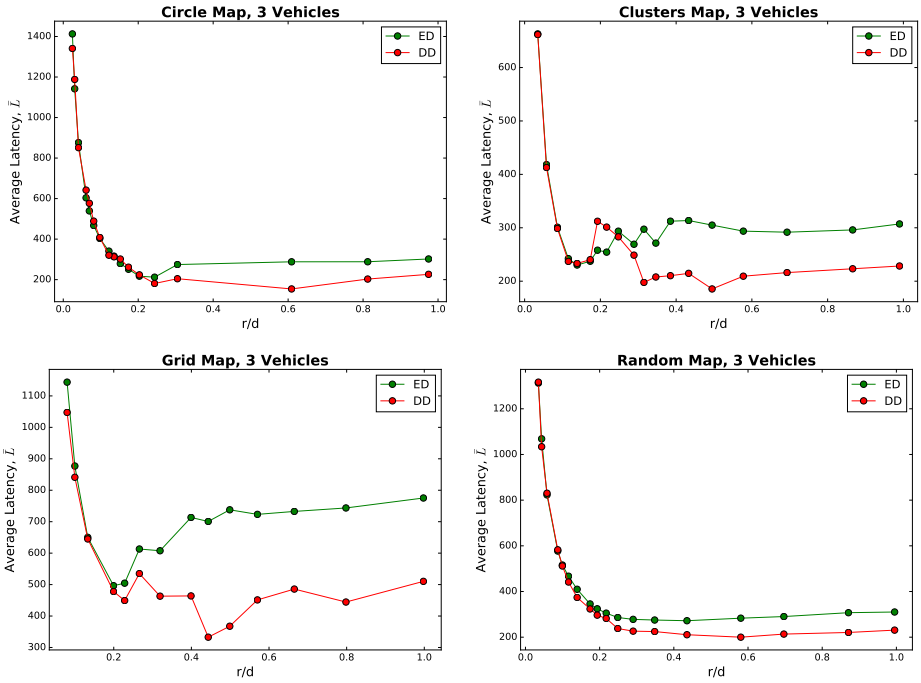


Figure 4.40. Comparison of performance using Euclidean distance versus Dubins path distance for three vehicles.

The simulation results presented here agree with the theoretical results for minimum Dubins paths between tasks. Specifically, if a destination point is located at a Euclidean distance from the vehicle of at least four times the minimum turn radius of the vehicle (or, in our notation, $r/d \leq 0.25$), then the the optimal path to the point can be constructed using only an arc with curvature equal to the minimum turn radius (a "C-segment") and a straight line (an "S-segment"), called a "CS" path. If the distance to the point is less than four times the vehicle turn radius (or $r/d > 0.25$), then the minimum Dubins path will be segments of type CCC, CSC, or a subset thereof[64, 63]. The implication is that Euclidean distance is a good approximation for the optimal Dubins path when $r/d \leq 0.25$, with an error proportionate to the extra path distance introduced by the C portion of the path, which agrees with our simulated results.

It should be noted that, for the results presented here, d represents the average

distance between all tasks. In other words, we use the average value of the Euclidean distance matrix, D, not including the diagonal zeros. This averaging method explains why we found the bifurcation point between Dubins and Euclidean measurements to lie between $r/d = 0.2$ and 0.3, while the theoretical results state 0.25 to be the true point. Other methods of calculating d are possible and could yield bifurcation points that are in better agreement with theory. Such methods include basing d on the median value in the distance matrix (not including diagonal zeros), the radius of a circle that circumscribes all tasks, or the distance between the geometric center of all tasks and the furthest task from that centroid.

4.3.1.4 Summary of Dubins Constraints Results.

Using Dubins paths to measure the travel distance between tasks introduces slightly more complexity into the MD^2WRP algorithm. Because the time to return to the current task is no longer zero, vehicles now have the option to visit the same task twice in a row. Additionally, because the ratio of the turn radius to the travel distance changes with each candidate task selection, the range of viable β values shifts for each decision.

The choice of whether to use Dubins paths or Euclidean distance to calculate the travel time between tasks is ultimately subject to design constraints, such as available CPU resources or user preference. Ideally, Dubins paths would always be used to measure travel distance, since they provide the greatest accuracy no matter the turn radius or the task configuration. However, if one can be reasonably certain that tasks will always be separated by distances that are large in comparison to turn radius (*i.e.* d is at least four times r), the Euclidean assumption for travel distance can save CPU time without sacrificing performance.

4.3.2 Presence of No-Fly Zones.

A common constraint in air operations is the presence of "no-fly" zones (NFZs). Airspace might be restricted in such a manner to maintain positive control around an airfield or other high traffic region, such as a training area. This may be especially important if there is a mix of manned and unmanned aircraft operating in the same space. Another reason a NFZ may be created is to keep friendly aircraft from getting too close to suspected enemy threats. Whatever the reason, airspace restrictions alter the mission environment. For MD^2WRP, this means parameters that were optimized for unrestricted airspace may no longer yield the best performance in the presence of NFZs. In this section, we attempt to quantify how various NFZ geometries affect MD^2WRP optimization.

We use each of the maps from Fig. 3.9 for our test. Vehicles are assumed to follow Euclidean paths (no kinematic constraints) and all vehicles begin the scenario at Task 1. Obstacle avoidance for vehicle paths is calculated using Tripath Toolkit[79] (now called Triplanner), which is software based on Kallmann's work in [80] and [81].

Our method of evaluating performance over numerous instances of NFZ is based on an *Impact Ratio* (IR),

$$IR = \frac{\bar{d}_{NFZ}}{\bar{d}}. \tag{4.20}$$

The IR is calculated by placing a rectangular NFZ on each map such that it interferes with a direct flight path between some of the tasks. The average distance between all tasks with the NFZ present ($\bar{d}_{NFZ}$) is divided by the unrestricted average distance between all tasks ($\bar{d}$) to provide the IR of the NFZ. In this way, a NFZ with $IR = 1$ has no effect on vehicle pathing, while any $IR > 1$ implies some degree of interference. To test over a range of Impact Ratios, NFZs of progressively larger size are created by stretching them along their primary axis.

For each map, we run a set of simulations for the case of one, two, and three

vehicles. For each case of vehicle number, we simulate on the range $1 \leq IR \leq 2$. We optimize and simulate MD^2WRP twice at each IR: first by using the optimal β given the unrestricted task configuration and then optimizing β in the presence of the NFZ. We refer to the original β as being "un-tuned" for the NFZ, whereas the re-optimized β is "tuned". Results are presented in terms of $\bar{L}$ vs. IR. In this way, we can see how various task and NFZ geometries interact to affect performance and gain insight as to when a NFZ is restrictive enough to warrant re-optimizing MD^2WRP.

4.3.2.1 The Clusters Map.

The first map we test is the Clusters map with a vertically oriented NFZ between the western three-task cluster and the two eastern clusters. A sample NFZ instance and the results for this scenario are shown in Fig. 4.41.

With a single vehicle, the performance of the un-tuned β begins to diverge slightly at $IR > 1.3$, with the slope of the curve increasing further from $IR = 1.6$. When $IR < 1.3$, there is little difference in performance. Re-tuning MD^2WRP is probably not necessary. Beyond $IR = 1.6$, failure to re-tune would result in drastic performance losses (increases in $\bar{L}$ of over 300%). Re-tuning for the NFZ, however, results in a linear increase in $\bar{L}$ with a relatively shallow slope. In other words, a single vehicle can adjust to a growing NFZ so as to minimize the impact to performance.

When a second vehicle is employed, re-tuning is not indicated until $IR > 1.6$, after which latency demonstrates an almost exponential increase without re-tuning. If both vehicles are re-tuned, the effects of the NFZ on performance are almost entirely eliminated, as indicated by the zero slope of the tuned curve. In other words, the vehicles compensate by positioning themselves around the NFZ.

The three vehicle results demonstrate an interesting phenomenon. Up to $IR = 1.5$, the tuned and un-tuned curves show about the same performance. When $IR > 1.5$,

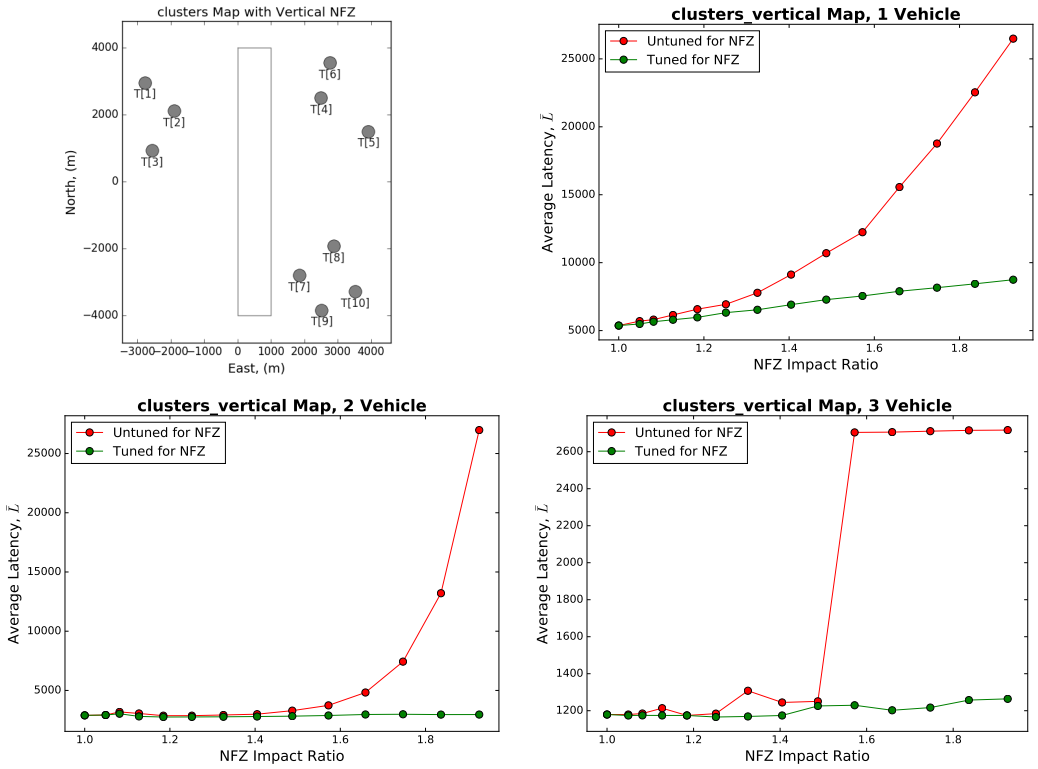


Figure 4.41. NFZ results for the Clusters map with a vertical NFZ between the western and eastern clusters.

the performance decreases in a step fashion, more than doubling the latency. The tuned curve remains nearly flat for all IR values. The reason for the jump in the un-tuned curve is due to the starting location of the agents with respect to the shape of the NFZ. Since all agents begin at Task 1, they are on the west side of the NFZ. When the NFZ is small enough ($IR \leq 1.5$), the un-tuned β is sufficient to successfully partition the vehicles around the NFZ. When the NFZ becomes too large ($IR > 1.5$), however, two vehicles become "trapped" to the west of the NFZ. The result is two vehicles servicing Tasks 1-3 while the other vehicle must service Tasks 4-10 alone (see the visit history in Fig. 4.42), which of course is a poor division of tasks resulting in poor latency performance. In the tuned case, the new β value ensures each vehicle "takes responsibility" for its own cluster, which is the optimal task division as it

effectively negates the effect of the NFZ.

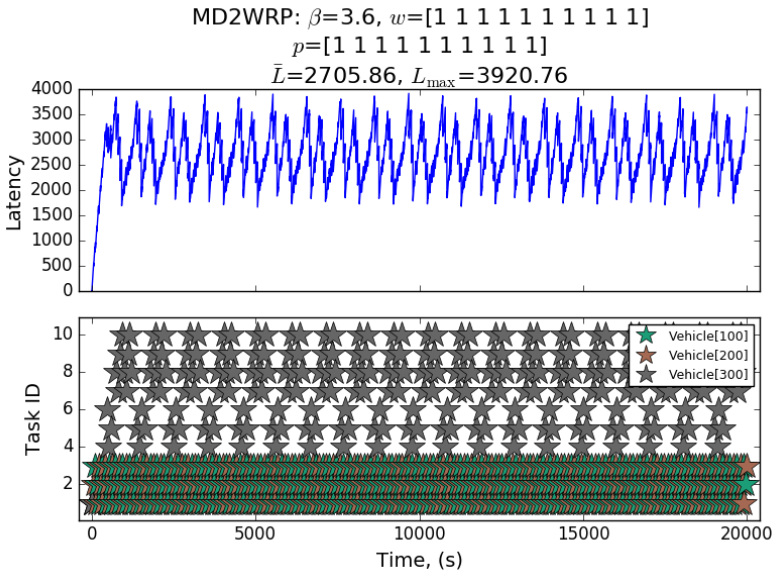


Figure 4.42. When the NFZ $IR > 1.5$ on the Clusters map, failure to re-tune β results in two vehicles becoming "trapped" on the west side of the NFZ.

We evaluate the Clusters map once again, but this time orient the NFZ horizontally between the northern and southern clusters. The objective is to ensure our IR method is sound regardless of the NFZ orientation. We also wish to see if the bifurcation points between the tuned and un-tuned curves are similar under different NFZ conditions. The new NFZ orientation and the simulation results are presented in Fig. 4.43.

The single and two-vehicle results are similar to the vertical case, with latency of the un-tuned curve increasing drastically for $IR > 1.6$. Again, the tuned curve shows a shallow slope for the single vehicle case and remains flat for two vehicles.

The three-vehicle curves with a horizontal NFZ look very similar to those of the two-vehicle results. The breakpoint for both is about $IR = 1.5$. With a horizontal NFZ, the step increase in latency seen for the vertical NFZ is eliminated, since all three vehicles begin north of the NFZ and there is only a single cluster of tasks to the

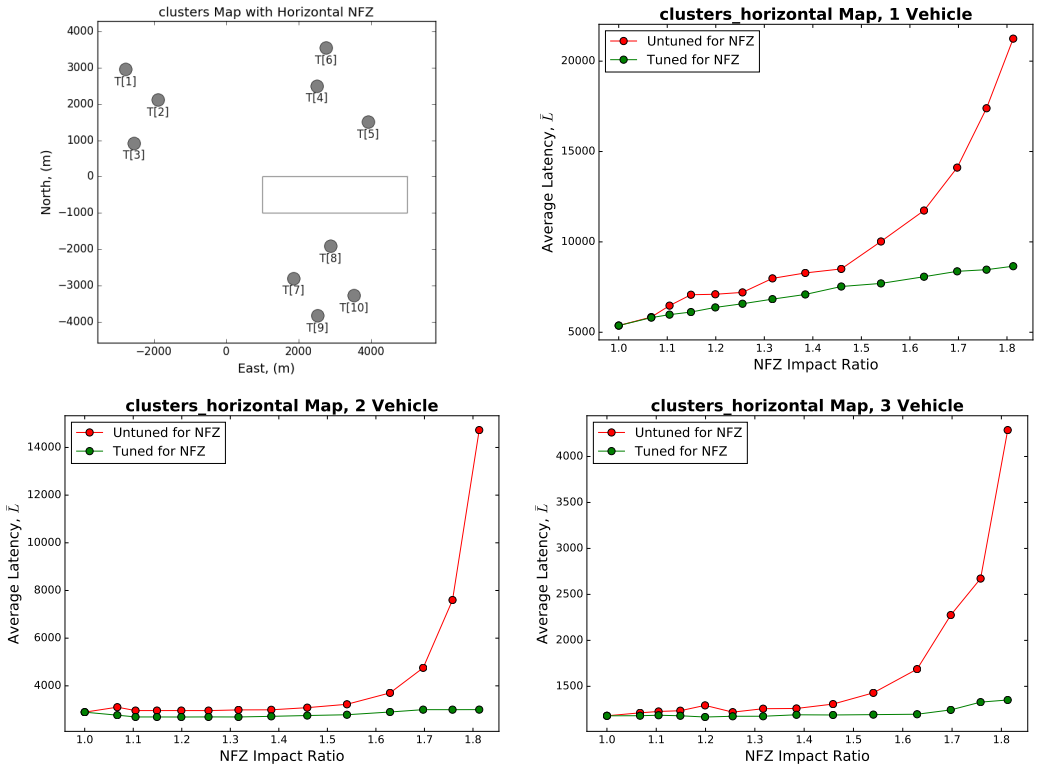


Figure 4.43. NFZ results for the Clusters map with a horizontal NFZ between the northern and southern clusters.

south, which removes the possibility of two vehicles being confined to a single cluster.

4.3.2.2 The Circle Map.

On the Circle map, we place a vertical NFZ to the west of Tasks 1, 9, and 10. A sample NFZ and the tuned versus un-tuned results are presented in Fig. 4.44.

The unique geometry of the circular task configuration makes the results interesting. The tuned and un-tuned performance for a single vehicle is identical until $IR = 1.6$, when the un-tuned curve begins to peel away from the linear tuned curve. Prior to $IR = 1.6$, both cases of β result in the vehicle visiting tasks in order around the circle and simply navigating around the NFZ. With $IR \geq 1.6$, the un-tuned vehicle begins to prefer tasks on whichever side of the NFZ it is currently on, servicing

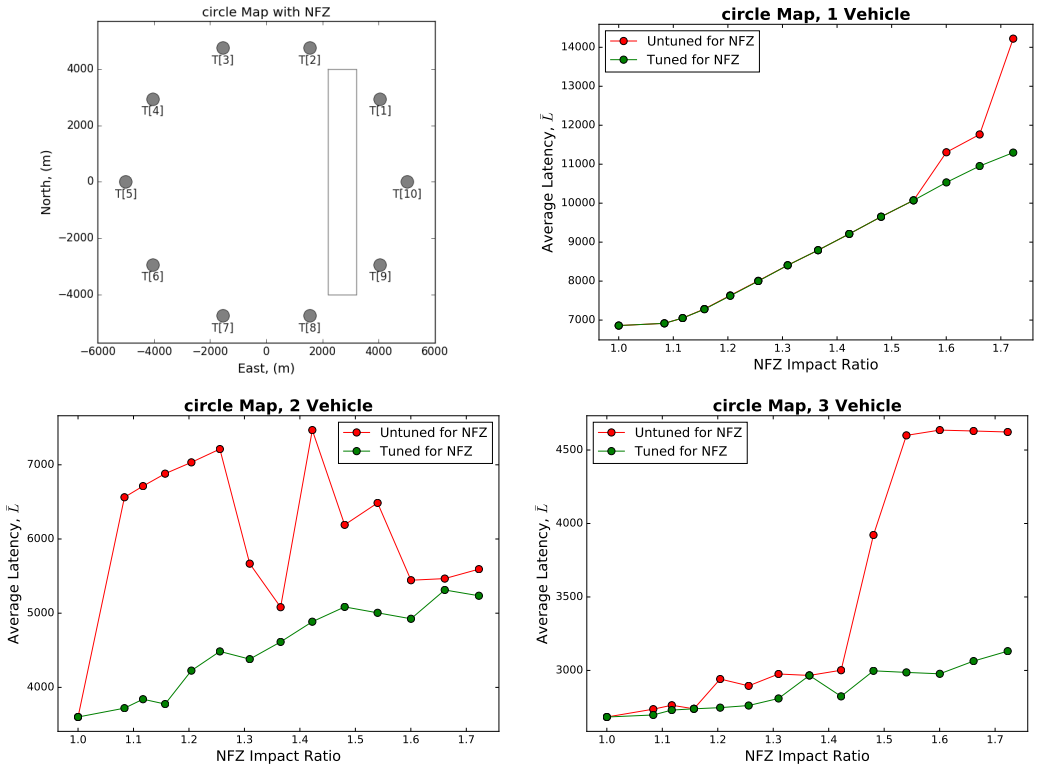


Figure 4.44. NFZ results for the Circle map.

those tasks multiple times before traveling around the NFZ to the other side of the map. This behavior creates high latency among the tasks on whichever side of the map the vehicle is not servicing. Tuning β for the NFZ, on the other hand, ensures the vehicle continues servicing each task in turn regardless of the NFZ size.

The two vehicle results present the best case for always re-tuning β out of all the scenarios tested. It is clear that, with the exception of a few IR values, the un-tuned vehicle performs significantly worse than the tuned β. The general reason is due to the difficult NFZ placement. As the NFZ grows, it splits the map unevenly, with 3 tasks east of the NFZ and 7 tasks to the west. This creates two groups of tasks of uneven size that are difficult for the two vehicles to share equitably.

When the $IR < 1.3$, the un-tuned vehicles "leap-frog" around the map, whereas the tuned vehicles follow each other along the same circuit, keeping a 5-task separa-

tion. The 5-task spacing yields the best latency. The dip in un-tuned latency near $IR = 1.4$ occurs because the particular instance of NFZ size and task geometry happens to result in the two vehicles following each other, as in the tuned case (though the latency is not quite as low due to a longer transient period). A further increase to the IR results in a third pattern; the vehicles take turns servicing the east and west sides of the map. This split servicing pattern also yields results significantly worse than the tuned result of 5-task spacing. Interestingly, as the IR continues to grow, the un-tuned β again yields the same pattern as the tuned spacing, with its performance suffering slightly from a longer transient period.

The Circle map with three vehicles displays curves that look similar to that of the Clusters map above with a vertical NFZ. That is, the tuned and un-tuned curves both deliver low latency values until $IR > 1.4$, when there is a step increase in the un-tuned latency. The reason is the same as in the Clusters map. All three vehicles start at Task 1, with two vehicles becoming "trapped" to the east of the NFZ as it grows in size. Re-tuning is necessary to evenly distribute the workload among the vehicles.

4.3.2.3 The Random Map.

The Random map results (Fig. 4.45) are similar to those of the Clusters map with horizontal NFZ. For a single vehicle, we see two bifurcation points in the un-tuned curve. The first is at $IR = 1.2$, where the un-tuned latency increases slightly over the tuned. The un-tuned curve tracks closely to the tuned curve with approximately a 500 latency offset until $IR = 1.6$, beyond which the un-tuned latency begins to increase at a faster rate. In both the two and three-vehicle cases, un-tuned and tuned latency are about equal until $IR > 1.6$. Beyond that point, the un-tuned latency begins to increase rapidly.

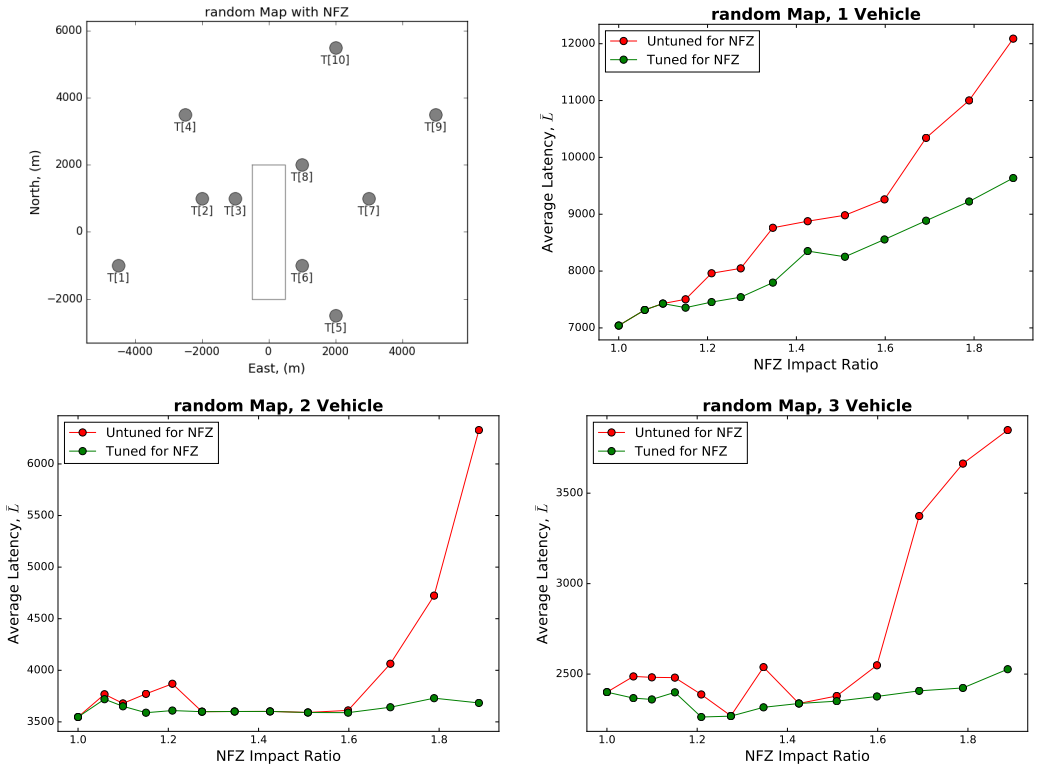


Figure 4.45. NFZ results for the Random map.

4.3.2.4 The Grid Map.

The last map we examine is the Grid map. The results are shown in Fig. 4.46. Like the Circle, the highly symmetric geometry of the Grid creates some interesting results. In the single vehicle case, the tuned and un-tuned performance are identical until $IR \approx 1.5$, with both vehicles visiting tasks according to the TSP tour while circumventing the NFZ. Beyond $IR = 1.5$, as in the Circle map, the un-tuned vehicle begins preferring tasks on whichever side of the NFZ it currently resides, causing the tasks on the opposite side to accumulate excessive latency.

The two-vehicle results show un-tuned and tuned performance that are competitive until $IR > 1.8$, with both βs encouraging the vehicles to split the task load evenly around the NFZ. One vehicle services Tasks 13-16 to the east of the NFZ and the

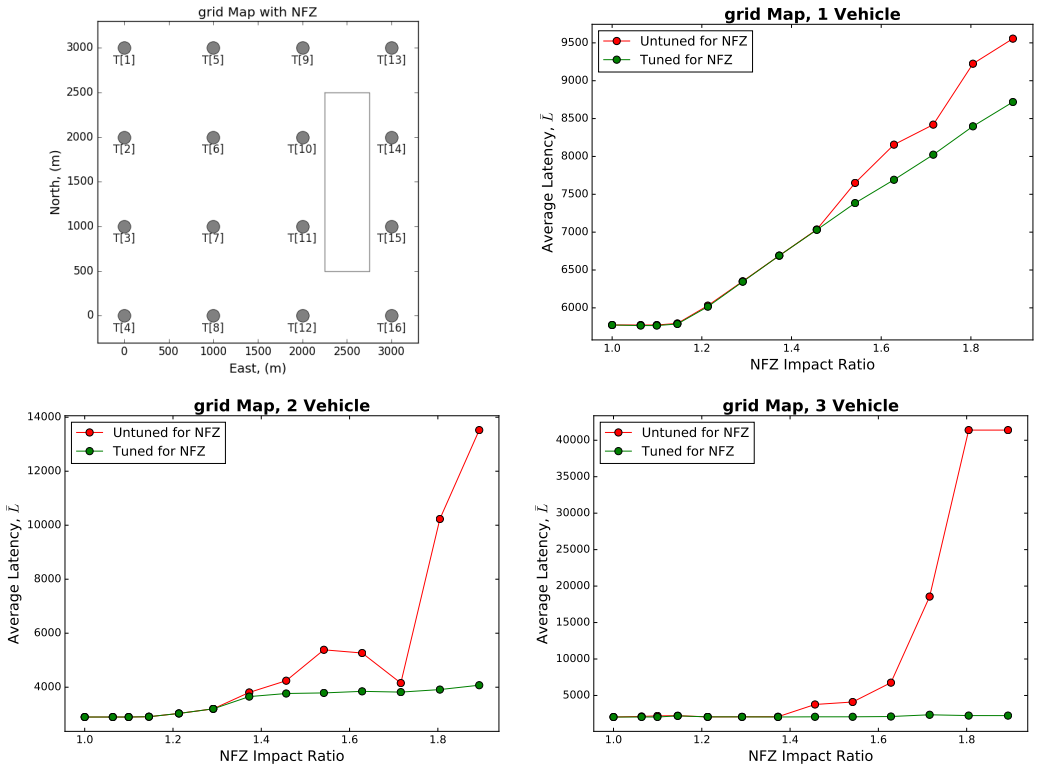


Figure 4.46. NFZ results for the Grid map.

other services all tasks to the west. When the IR exceeds 1.8, however, the un-tuned β results in both vehicles servicing all tasks, which is not ideal since both vehicles are making the long trip around the NFZ. Re-tuning brings the vehicles back to the east-west division of tasks.

As with most other scenarios, three vehicles on the Grid map have similar performance using both the tuned and un-tuned βs when $IR < 1.6$. For IRs greater than this, however, the latency spike of the un-tuned curve is the result of the distance discount being so large that none of the vehicles ever receive a reward large enough to warrant a visit to the other side of the NFZ. The end result is continually increasing latency for Tasks 13-16 until the end of the simulation.

4.3.2.5 Summary of NFZ Results.

In general, with the exception of the Circle map with two vehicles, the results for the scenarios we tested indicate that, if the calculated IR is less than 1.6, re-tuning of β is not strictly necessary. Performance will only suffer slightly, and in some cases be identical, to the performance using a β tuned for the NFZ. Of course, using the two-vehicle Circle map as a counter-example, the safest option is to always re-tune β when a NFZ is added or removed from the operational area. The development of a dynamic re-tuning algorithm to take into account changes to the mission environment would be a useful addition to the MD^2WRP software suite. This will be discussed further in Sec. 5.2.

These results are limited due to the small sample size of task configurations and NFZ placements that were tested. Also, we have only examined a single NFZ on each map but it is possible a given area could have more than one. While more testing could provide the confidence to draw broad conclusions, there are an infinite combination of task configuration with NFZs. Still, we have proposed a methodology (via the Impact Ratio, IR) which makes it possible to study the effects of NFZs on vehicle task selection, which could serve as a basis for more extensive testing, or at least allow the effects of a NFZ to be assessed for a specific mission.

4.3.3 Return to Base Requirements.

One of the primary benefits of the TSP approach to task selection is the guarantee it provides for task revisit times. If we consider one task to be the base, then we can guarantee that the vehicle will return to base (RTB) with a given frequency. An RTB criterion is useful to facilitate refueling or perhaps to dump collected data when long-haul communications are unavailable. For utility methods, such revisit rates are not necessarily guaranteed, since the vehicle is selecting tasks based on state variables.

However, since MD^2WRP provides a way to weight individual tasks (via $\mathbf{w}$), a mechanism does exist for encouraging more frequent visits to a "base" task without hard-coding an RTB command when the deadline is approaching. To test the ability of MD^2WRP to meet such an RTB requirement, we use two different schemes: one that varies the number of tasks and one that changes the relative placement of the base node to the PISR tasks.

4.3.3.1 Centrally Located Base with Varying Number of Tasks.

For the first test scenario, we place tasks in a circle of fixed $5000m$ radius, with the base "task" at the center. We vary the number of tasks from five to twelve (not including the base task). The idea is to explore how the weight of the base task, w_{base}, must change in order to guarantee the RTB deadline is met. We also wish to show how $\bar{L}$ is effected when weights are adjusted to satisfy the RTB requirement.

First, however, it is necessary to develop a method of selecting w_{base} that meets an RTB threshold while providing the best performance. To demonstrate the process we take a seven task RTB scenario as an example, but it should be noted that the same process for selecting w_{base} is used for all RTB maps.

The first step is to set $\mathbf{w} = \mathbb{1}$ and optimize β over a mission duration of $20000s$. In the event that multiple βs deliver the same $\bar{L}$, we select the lowest value. For the seven task example, $\beta = 2.6$ is optimal. Next, we simulate the scenario for a range of w_{base} values, from 1.0 to 10.0 in increments of 0.1. We sort the results by $\bar{L}$ with the objective of selecting w_{base} that never exceeds the RTB threshold while providing the best performance. Sample results are provided in Table 4.8 for an RTB threshold of $1200s$.

All weights below the horizontal line in Table 4.8 result in vehicle tours that satisfy a RTB requirement of $1200s$ or less. Of those that meet the criterion, we see

Table 4.8. Selection of w_{base} to meet an RTB threshold of 1200s.

$\mathbf{w_{base}}$	$\bar{\mathbf{L}}$	Max RTB (s)	Avg RTB (s)
1.0	6370.9	1835.1	1835.1
1.1	6373.6	1637.9	1637.9
1.2	6385.2	1637.9	1546.8
1.5	6449.6	1440.6	1370.2
1.6	6554.6	1440.6	1206.4
1.7	6571.5	1243.4	1185.4
1.8	6584.5	1243.4	1173.8
1.9	6768.3	1046.2	1006.8
2.2	6786.7	1046.2	997.0
2.4	6945.4	1046.2	902.8
3.0	7190.8	1046.2	816.1

that $w_{base} = 1.9$ provides the best $\bar{L}$. Even though w_{base} of 2.2, 2.4, and 3.0 also meet the RTB threshold, they place too much weight on the base, resulting in more frequent visits than necessary, as demonstrated by their lower average RTB times. Visiting the base too frequently is undesirable, since vehicles should spend as much time accomplishing PISR tasks as possible.

With a process for optimizing w_{base}, the next step is to explore how the value of the optimal w_{base} changes, both as the number of tasks change and as the RTB requirement becomes tighter. Figure 4.47 shows two curves of RTB criteria, one for 1200s and one for 900s, on a plot of w_{base} versus number of tasks.

As expected, the optimal value of w_{base} increases as the RTB requirement becomes tighter. Also, as the number of tasks increase, the value of w_{base} increases somewhat linearly. Though more data would be required on a wider variety of task configurations, these results show that it may be possible to extrapolate the required value of w_{base} when simulation data is not available for a specific scenario, or at least use this data to establish a good range of values for beginning the search.

Lastly, three performance curves are shown in Fig. 4.48 for a varying number of tasks. One curve is $\bar{L}$ when $w_{base} = 1.0$, which we include because it provides a

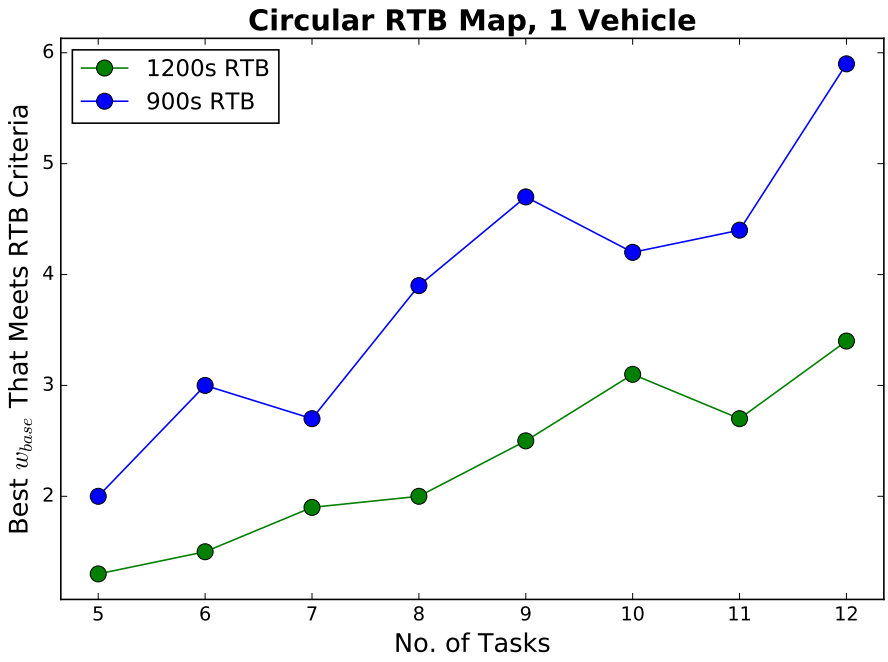


Figure 4.47. Best value of w_{base} as a function of number of tasks.

lower-bound on performance. (Recall that with $w_{base} = 1.0$ the vehicle is optimized for performance, but does not necessarily satisfy the RTB requirement). Each data point on the other two curves represent the performance on a simulation run for the given number of tasks with w_{base} optimized to meet either $1200s$ or $900s$ RTB criteria, while maximizing performance.

Of course, enforcing RTB criteria reduces performance. But we see that for a small number of tasks, less than about nine, the performance losses for $1200s$ RTB are minimal. This is a useful result, since we can guarantee a base visit every $1200s$ without losing too much time in servicing PISR tasks. Also, from nine tasks and up, we see the performance losses due to enforcing RTB criteria begin to level off, especially for $1200s$. The plateau is due to the decrease in spacing between tasks as more tasks are added to the fixed-radius circle. The vehicle has time to visit more tasks, because they are closer together, before returning to the base task at the center

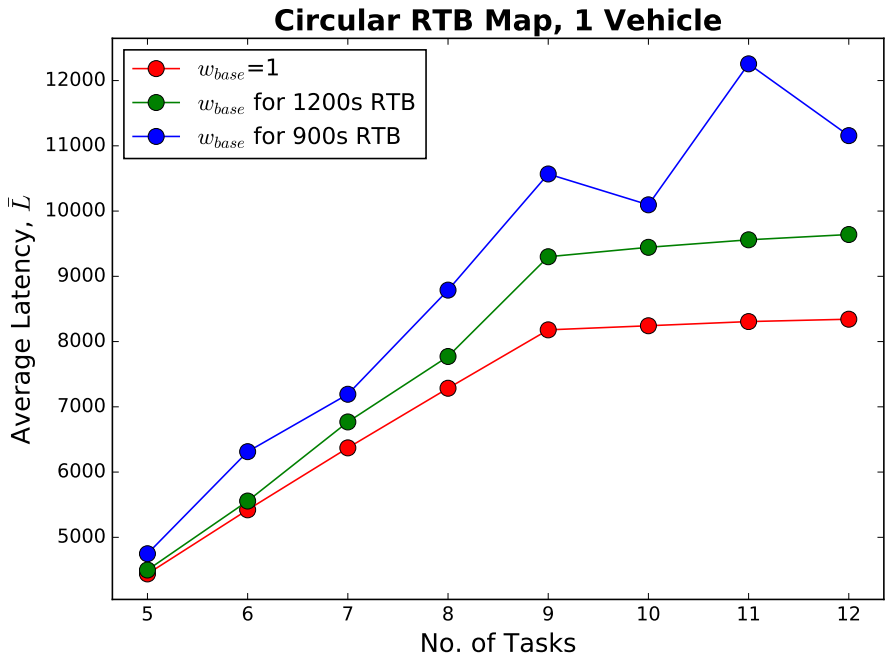


Figure 4.48. Average latency performance, $\bar{L}$, as a function of number of tasks.

of the circle.

The limited results presented here will require more substantiating data before any broad conclusions about the ability of MD^2WRP to achieve guaranteed RTB thresholds can be drawn. However, they do act as a proof of concept and reveal promise that RTB criteria can be met through manipulation of the MD^2WRP parameters as the number of tasks increases.

4.3.3.2 Relative Location of the Base to the Tasks .

In the second test scenario, we wish to evaluate how the relative location of the base task to the PISR tasks affects the performance of MD^2WRP and its ability to satisfy RTB criteria as well as how the required w_{base} changes. We use the 10-task Circle map and the Random map, each with an additional task which serves as the base task (the base is always Task 1). In order to quantify the placement of the base

relative to the tasks, we calculate the centroid of the task map. An offset is used to describe the location of the base. We consider a base located at the centroid to have an offset of 0% while a base located a distance from the centroid equal to the farthest task from the centroid has an offset of 100%. For both of our test maps, we collect data for a range of offset values from 0-100%.

Sample base locations are shown in Fig. 4.49 for the Circle map with 0, 40, and 90% offsets. As with the above results where we varied the number of tasks, for the base offset simulations we provide data in terms of optimal w_{base} that meets a given RTB threshold as well as $\bar{L}$ when RTB thresholds are met. For the $\bar{L}$ results, we again provide the performance data for $w_{base} = 1.0$ as a lower-bound comparison. These results are presented in Fig. 4.50.

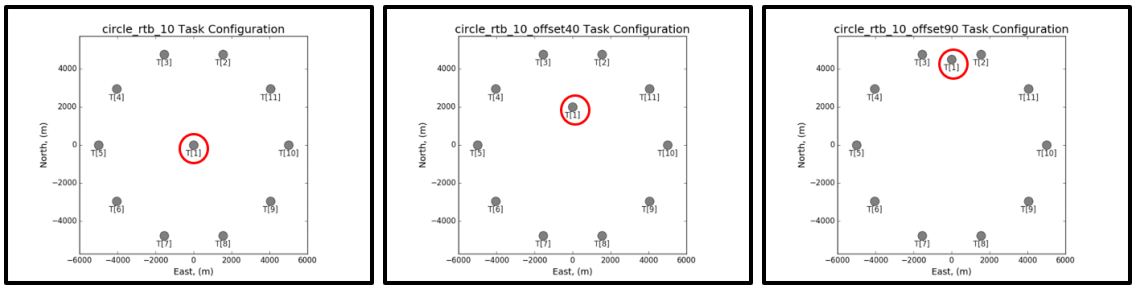


Figure 4.49. Sample base offsets for the Circle map (left to right - 0%, 40%, and 90%). The base task is circled in red.

For the case of a 1600s RTB requirement, additional weight is only required on the base node for offsets between 0-20%. With 30-100% offset, the base is close enough to the perimeter of the circle that the vehicle is able to visit the tasks and the base in a simple TSP circuit and still meet the RTB requirement. A 1300s RTB threshold is more difficult for the vehicle to meet. As the base gets closer to one side of the circle, increasingly large weights are required to draw the vehicle back to base when it is visiting tasks on the opposite side of the map.

In terms of performance, 30-100% offset with a 1600s RTB have the same per-

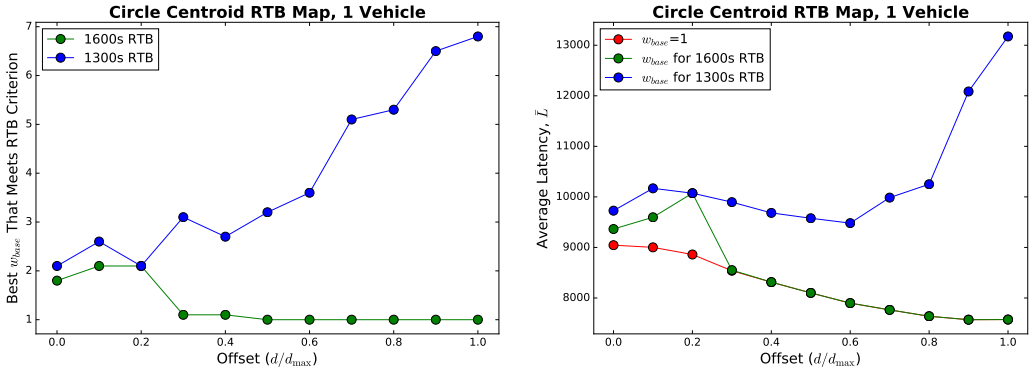


Figure 4.50. The required w_{base} to meet RTB thresholds for varying base offsets on the Circle map (left) and the performance given each RTB threshold is met (right).

formance as the case with $w_{base} = 1.0$, which makes sense given that we already determined that $w_{base} = 1.0$ is sufficient to meet the RTB threshold when the base is close to the perimeter of the circle. For the $1300s$ RTB, $\bar{L}$ is about $1000 - 2000$ higher between 0% and 60% offset, but begins to grow at a rapid rate when the offset is above this range, with $\bar{L}$ being about twice as high at 100% offset.

We also present results in terms of the RTB achieved versus w_{base} for offsets of 0, 40, and 90% (Fig. 4.51). For each instance of offset, we show the maximum, minimum, and average RTB time +/- standard deviation. For reference, the two RTB goal times are also plotted.

The data presented in Fig. 4.51 are derived from the same simulations as in Fig. 4.50, but when depicted in this way it is easy to see how increasing w_{base} reduces RTB times. It is also useful in determining what RTB deadlines are within the realm of possibility, as the relative location of the base to the tasks imposes physical constraints on how quickly the vehicle can RTB, regardless of w_{base}, as seen by the flattening of max and average RTB with increasing w_{base}. The inclusion of maximum, minimum, and average RTB on the chart show the variability in RTB metrics for different w_{base} values.

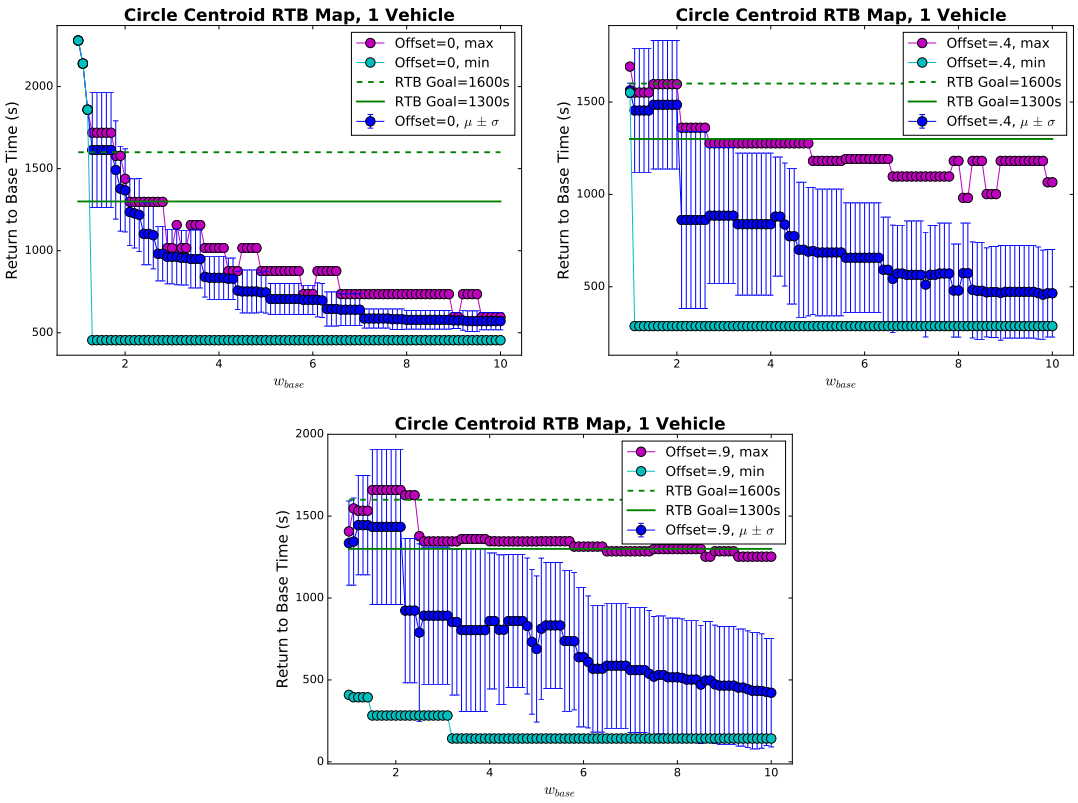


Figure 4.51. The RTB time as a function of w_{base}, for offsets of 0, 40, and 90% on the Circle map.

Figure 4.52 shows the visit and trajectory history for a vehicle tuned to meet 1300s RTB ($w_{base} = 6.5$) on the Circle map with a 90% offset (recall Task 1 is the base task). Figures 4.51 and 4.52 combined reveal the trade-offs required to meet an RTB criterion with MD^2WRP. From Fig. 4.51, the vehicle never takes longer than 1300s to RTB, however, the average RTB is much lower at 586s, indicating the vehicle actually returns to base much more often. We also see the standard deviation is quite large at $\pm$382s, so there is significant variability in the time between base visits. The minimum RTB is low, at 142s. From the visit history in Fig. 4.52, there are many quick returns to Task 1 during the transient period. This is where the minimum RTB occurs. During the steady-state, visits to the base occur in clusters, with 3-4 visits occurring relatively quickly, followed by a long interval while the vehicle visits tasks

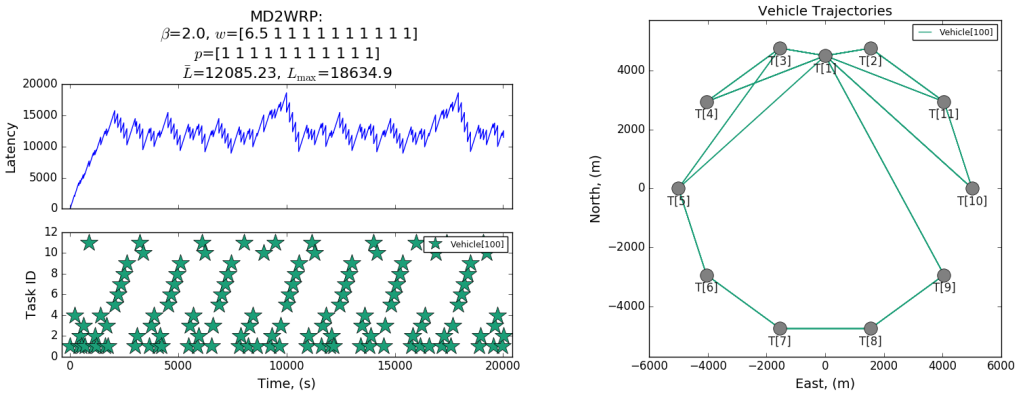


Figure 4.52. Left, vehicle visit history meeting a $1300s$ RTB threshold on the Circle map with a 90% base offset. Right, the vehicle trajectory history.

on the other side of the map. This is the cause of the low average RTB and the large standard deviation.

The trajectory history in Fig. 4.52 shows the "circuits" traveled as the vehicle reaches steady-state. The vehicle visits only 2-3 tasks before returning to base when it is near the base node, but takes a long route when visiting tasks far away. In this way, the vehicle can meet the RTB deadline while minimizing $\bar{L}$.

Another discussion point in Fig. 4.51 is the occasional increase of both maximum and average RTB times with increasing w_{base}, which is counter-intuitive. In some instances, w_{base} increases yet maximum RTB also increases, meaning the vehicle actually takes longer to RTB despite the base task offering a higher reward. Or similarly, w_{base} decreases with a corresponding decrease in average RTB - so less reward is gained, but the vehicle visits more frequently. How can this be? The answer lies in the sensitivity of vehicle decisions to the evolution of the task age vector. We use an example to illustrate, which is provided in Fig. 4.53 showing two different visit histories from the Circle map with 40% offset: one when $w_{base} = 8.2$ (left) and the other with $w_{base} = 8.3$ (right). These two simulation instances were chosen because they demonstrate this phenomenon well.

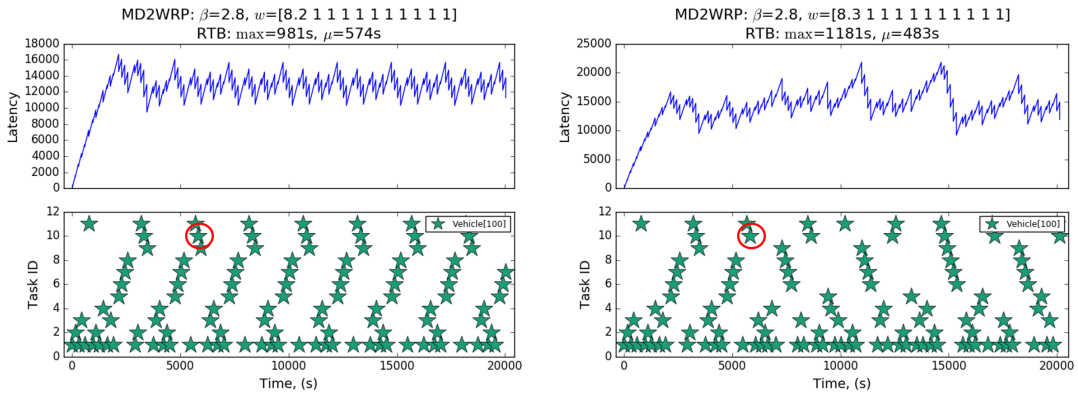


Figure 4.53. Vehicle visit histories for $w_{base} = 8.2$ (left) and $w_{base} = 8.3$ (right) on the Circle map with 40% offset.

The two visit histories are identical until the visit to Task 10 circled in red, which occurs around $6000s$. The vehicle with $w_{base} = 8.2$ goes from Task 10 to 9 whereas the $w_{base} = 8.3$ vehicle returns to the base (Task 1), due to the increased reward. From that point onwards, the task age vectors of the two simulations evolve differently. Though the base may have an increased reward, that reward is not necessarily enough to outweigh the task rewards under the new age vector in the same way it might have under the old one. While $w_{base} = 8.3$ does result in a lower average RTB ($574s$) than $w_{base} = 8.2$ ($483s$), $w_{base} = 8.3$ results in a longer period with no base visit ($1181s$). Meanwhile, $w_{base} = 8.2$ has a shorter maximum RTB ($981s$). In summary, under a consistent β, increases in w_{base} do not necessarily result in a monotonically decreasing maximum or average RTB.

Lastly, we look at the Random map. We present the same results as for the Circle map above. Sample base locations for the 10-task Random map are in Fig. 4.54.

The w_{base} required to meet RTB thresholds of $1000s$ and $1300s$ for offsets of 0-100% are shown on the left of Fig. 4.55 and $\bar{L}$ for each tested offset are on the right. The results are intuitive. As the base offset increases, the w_{base} required to meet the RTB threshold also increases. A centrally located base means the vehicle has less

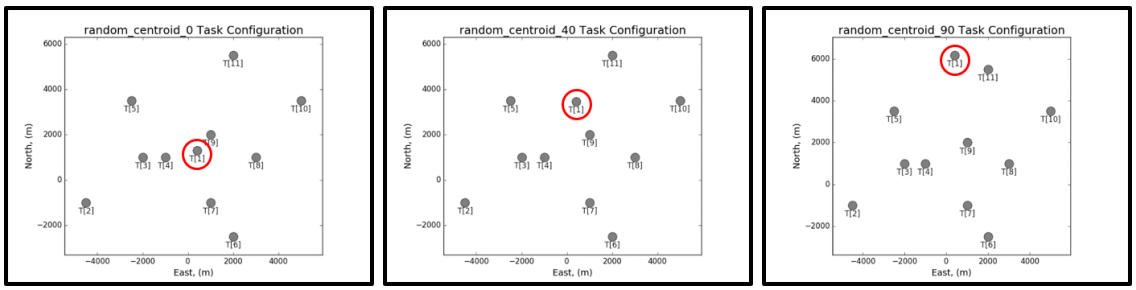


Figure 4.54. Sample base offsets for the Random map (left to right - 0%, 40%, and 90%). The base task is circled in red.

distance to the farthest located tasks, so there is less distance discounting from β and less w_{base} is required. When the base is located on the perimeter of the map, servicing tasks on the opposite side increases the RTB distance, so a higher w_{base} is necessary to overcome the larger distance discount. Of course, decreasing the RTB threshold further increases the required w_{base}. Similarly, increasing the base offset or decreasing the RTB threshold generally results in worse performance since the vehicle must stop servicing tasks more frequently in order to make it back to base before the deadline. When the offset is $\geq 80\%$, $\bar{L}$ begins to rise rapidly as making frequent trips back to base from the opposite side of the map impacts performance.

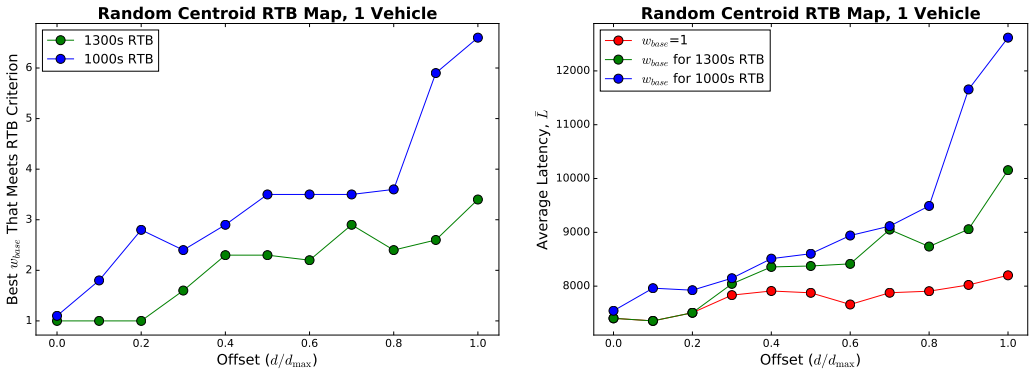


Figure 4.55. The required w_{base} to meet RTB thresholds for varying base offsets on the Random map (left) and the performance given each RTB threshold is met (right).

The plots of RTB versus w_{base} for offsets of 0, 40, and 90% are shown in Fig. 4.56.

The Random map plots look similar to the Circle map above, showing the same trends. In general, increasing w_{base} decreases the maximum and average RTB time, with higher w_{base} required to meet the RTB deadline as the offset increases. We also see the same disparity between maximum and average RTB. The w_{base} to achieve a maximum RTB threshold results in a significantly lower value of average RTB. This is because more frequent base visits occur when the vehicle is visiting tasks close to the base while RTB frequency is reduced when visiting distant tasks. The consequence of this is that achieving consistent revisit times to the base is difficult with MD^2WRP. This issue is discussed further along with possible solutions and alternatives in Ch. V.

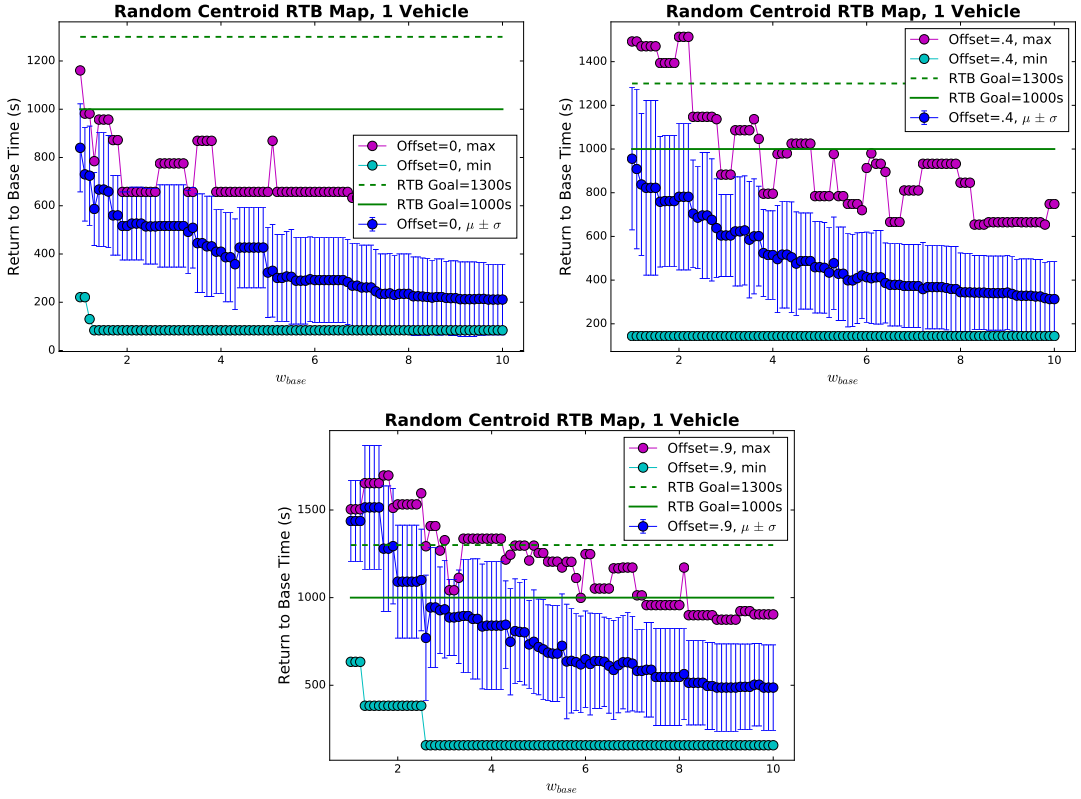


Figure 4.56. The RTB time as a function of w_{base}, for offsets of 0, 40, and 90% on the Random map.

4.3.3.3 Summary of Return to Base Results.

We demonstrated the feasibility of using MD^2WRP to enforce a return to base requirement by considering the base as another PISR task and manipulating its MD^2WRP weight, both as the number of tasks increases and as the relative position of the base to the tasks changes. Intuitively, as the number of tasks increases or as the base location becomes further removed from the centroid of all tasks, more MD^2WRP weight is required on the base task in order to meet a given RTB requirement. Similarly, as the time between required base visits becomes shorter, additional base weight is also required. Additionally, the performance penalty introduced by decreasing the required time between base visits becomes more severe when the base is located at a distance from the centroid that is further than 80% of the distance between the centroid of all tasks and the furthest task from the centroid.

4.3.4 Mid-Mission Addition and Removal of Vehicles and Tasks.

The utility function approach to PISR task selection is attractive over the TSP method because it does not require a centralized planner to assign routes to vehicles. Instead, each vehicle makes a real-time decision about which task to visit. Because of this decentralized nature, MD^2WRP is robust to the addition or loss of vehicles, intentional or not, and to the addition or removal of tasks mid-mission. In this section, we demonstrate such robustness with two examples.

The first demonstration concerns the addition or loss of vehicles during a mission. Once again, we use the Clusters map from Fig. 3.9. We set $\beta = 5.0$ and $\mathbf{w} = \mathbb{1}$. Vehicles communicate using the CxBD communication mode (or "Broadcast Destinatinos") described in Sec. 4.2.1.3. Two vehicles (Vehicle 100 and 200) begin the mission located at Task 1 and proceed to visit tasks until $5000s$. Their visit sequences are shown in the bottom half of Fig. 4.57, with each star representing a task visit.

The top half of Fig. 4.57 shows the total latency curve for the mission.

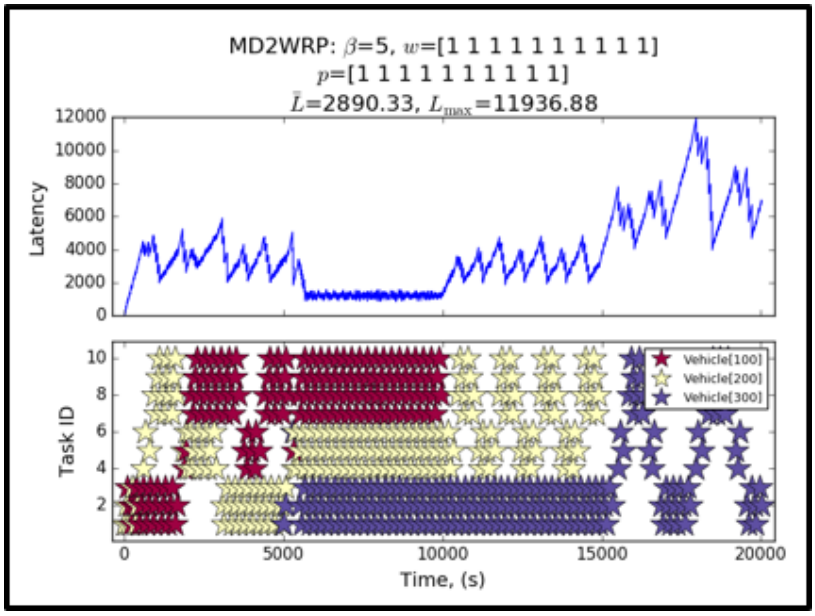


Figure 4.57. The total latency and task visit history of three vehicles on the Clusters map as vehicles are added and removed.

At 5000s mission time, Vehicle 300 is introduced, which also begins at Task 1. As the third vehicle begins servicing tasks and communicating its task completion and destination information, the other vehicles automatically adjust to its presence. Ultimately, each vehicle maneuvers to a separate cluster. This pattern continues until 10000s at which point Vehicle 100 goes offline. As the other vehicles continue to accomplish tasks they stop receiving updates from Vehicle 100. To compensate, they re-partition the tasks and begin servicing the cluster where Vehicle 100 had previously been. Finally, at 15000s Vehicle 200 is also removed, forcing Vehicle 300 to service all ten tasks by itself.

The second demonstration is of addition and removal of tasks mid-mission. We use the Clusters map with $\beta = 4.0$ and $\mathbf{w} = \mathbb{1}$. However, at $t = 0$, Tasks 7-10 are inactive. The number of vehicles remains constant at two, with both beginning at Task 1. The task visit history and total latency curves are shown in Fig. 4.58.

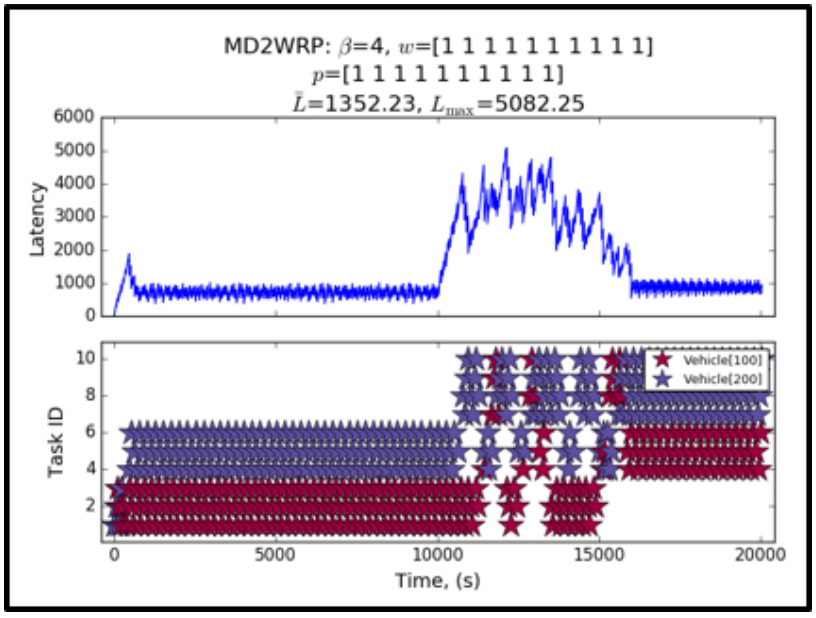


Figure 4.58. The total latency and task visit history of two vehicles on the Clusters map as tasks are added and removed.

Through 10000s, each vehicle services one of the two clusters, at which point Tasks 7-10 are activated. To adjust, the vehicles spread out and begin sharing all ten tasks. At 15000s, Tasks 1-3 are deactivated. The vehicles resume servicing one cluster each, though the clusters are different than at the beginning of the scenario.

It should be noted that the β values used in the demonstrations above are not necessarily optimal for the duration of the scenario. Any time a vehicle or task is added or removed, the optimal β for the scenario changes. So long as those changes are not too dramatic, continuing with the old β still results in good performance and vehicle separation, but task selections are no longer optimal. In order to ensure the best possible performance, it would be ideal to re-optimize β when there are changes to the number of vehicles, the number of tasks, or the location of tasks.

The above demonstrations show the power of using a decentralized task selection method with a simple communication scheme. Vehicles can readily adapt to a dynamic mission environment without operator intervention, whether those dynamics

are the result of deliberate operator actions, the unexpected failure of a vehicle, or the result of enemy interference.

4.4 Summary of Results

In this chapter, we performed an in-depth examination of the MD^2WRP utility function as a task selection method for PISR. We began in Sec. 4.1 by characterizing the MD^2WRP utility function. We explored how its parameters, β and $\mathbf{w}$, affected vehicle behavior and used that information to develop a normalized version of MD^2WRP, which facilitated optimization of the parameters using a simple two-step brute force method. We also proved that task selections under MD^2WRP eventually become periodic. Then, in Sec. 4.2, we explored different versions of MD^2WRP, including multiple decision lookahead and three different inter-vehicle communication modes. Moving forward with the best version of MD^2WRP, we compared its performance to four alternative methods for task selection, including those based on the TSP and other utility functions. Finally, in Sec. 4.3, we investigated the effects of four different operational factors on the performance of MD^2WRP, and proposed ways to characterize those effects and mitigate their performance impacts.

V. Conclusion

5.1 Conclusions from Results

From the results in Ch. IV, we can draw some conclusions about MD^2WRP that pave the way for real-world testing and application. First, to make MD^2WRP more intuitive to optimize, it is helpful to normalize travel time between tasks, t_{ij}, based on the time to travel between the two most distant tasks, $t_{ij,max}$. Normalizing in this way has two primary benefits: it ensures consistent behavior by vehicles using the same β and $\mathbf{w}$, even if they have different constant velocities, and it simplifies the search for the optimal β by making it a dimensionless parameter.

The behavior of vehicles under MD^2WRP is influenced by the parameters β and $\mathbf{w}$. We can view β as a way to alter a vehicle's "preference" for servicing nearby tasks versus those far away. Larger β encourages the former and smaller β the latter. Viewed another way, shifting β from large to small shifts the vehicle from locally-oriented to globally-oriented task coverage. It is important to note that the effect of β and $\mathbf{w}$ on vehicle behavior are subject to bifurcation points. Increasing or decreasing their values does not necessarily alter vehicle behavior. The parameters must change task values enough to alter the task visit sequence. On a related note, task visit sequences under MD^2WRP are always periodic in the steady-state, though changes to the initial conditions (*i.e.* the initial task ages or the vehicle start location) can change the final periodic visit sequence. This is important because a vehicle that begins servicing a specific task configuration under a different set of initial conditions than a vehicle that is already servicing tasks may not have the same performance even though the same parameters are used. Therefore, for maximum performance, all vehicles should be optimized based on their specific initial conditions before entering service.

In multi-vehicle scenarios, MD^2WRP provides implicit and decentralized cooperation with a simple data sharing scheme. Vehicles only broadcast updates immediately after completing a task. The updates consist of the task that was just completed along with a timestamp plus their next destination and an estimated arrival time. Other vehicles receive and store this information and make use of it when calculating task ages during their next task selection decision.

When optimizing MD^2WRP parameters, it is most efficient to optimize β before $\mathbf{w}$. Optimizing β is a single value and yields latency performance increases of 50% over using a utility function based solely on task age. In fact, for a single vehicle servicing tasks of the same priority, the optimization of β alone can often yield a steady-state visit sequence equivalent to the TSP solution. When tasks have different priorities, optimizing $\mathbf{w}$ for only the highest priority task(s) improves latency performance by an additional 3-5%. In the case of multiple vehicles, only optimizing β causes vehicles to automatically partition tasks, often yielding subtours that are equivalent to those generated by the k-means++ multiple TSP method. The result is MD^2WRP delivers performance on par with basic TSP solutions for PISR in both the single and multi-vehicle case. Additionally, when compared to other task selection utility functions from the literature, MD^2WRP performs just as well while needing only a single optimized parameter (β).

MD^2WRP can be adapted to overcome challenges introduced by operational factors. Vehicle kinematic constraints can be ignored without significant performance loss when calculating travel times between tasks so long as the turn radius is less than 25% of the maximum value in the task distance matrix. This lessens the burden of on-board computation since Dubins motion does not need to be calculated. When a no-fly zone is present, MD^2WRP can be tuned as if the NFZ were absent so long as the NFZ does not increase the vehicle's average travel distance by more than 50% ($IR < 1.5$).

If mission needs necessitate a return-to-base requirement, an RTB threshold can be met with minimal impact on performance when the base is located less than 80% of the distance between the centroid of all tasks and the farthest task from the centroid (offset < 0.8). If vehicles or tasks are added or removed mid-mission, MD^2WRP will ensure vehicles re-distribute the task workload without operator input, though the newly established visit sequences may no longer be optimal.

The results of this research provide a foundation for further testing and development of MD^2WRP, whether by simulation or flight test. There are many aspects to consider for task selection in PISR missions. This research answered some of the biggest and most important questions about using MD^2WRP for task selection, but also raised questions that require additional research to answer.

5.2 Future Work

The optimization method for the MD^2WRP parameters β and $\mathbf{w}$ described in Sec. 4.1.4 is a brute force approach, made possible by normalizing travel time based on the maximum value in the task distance matrix (D). We make the case, through comparisons with other methods, that good performance is possible by only optimizing β and the weight of the high priority task(s). However, we also show in Sec. 4.1.4 that it is possible to further increase performance by manipulating the weights of non-high priority tasks in unintuitive ways. Therefore, as an avenue of future study, we recommend developing a MD^2WRP parameter optimization method that conducts a more thorough search of the space in an attempt to find better local minimums that make use of more complex weight vectors ($\mathbf{w}$). For instance, genetic algorithms (GAs) are well-suited for optimization when many parameters are available. Implementing a GA poses a challenge, however, as our average weighted latency objective function (Eq. 3.1) requires a full mission simulation to be evaluated, which would make the

evaluation of thousands of different parameter sets infeasible. Therefore, a way would be needed to either simplify the evaluation of $\bar{L}$ or a new objective function would need to be selected. Another interesting approach, which would eliminate the need to fully simulate results in order to evaluate the objective function, is simulation-based learning. These methods, which were explored for utility function parameter optimization in [73], include Monte-Carlo, Temporal Difference, and Similar State Estimate Update. While the details of the specific methods differ, each make use of episodic "experience" from simulation results, which feed a reinforcement learning algorithm and iteratively update parameter values, eventually converging to locally optimal values.

Our focus in this work was to optimize the MD^2WRP parameters and use those to compare performance between task selection methods and under different operational factors. We were not so much interested in the actual parameter values, so long as performance was optimized. However, there is value in understanding how parameter values change, both with the number of vehicles and the task configuration. A closer look at these relationships may make it possible to create a table of "good" parameter values based on the number of vehicles and the distance matrix, eliminating the need to perform on-the-fly optimization.

Throughout this research, a constant theme has been the nature of the transient versus steady-state in PISR task selection. For the results in this dissertation, the decision was made to compare performance between vehicles based on average weighted latency, since that metric accounts for performance in both the transient and steady-state phases of task visit sequence development. This made sense because the UAVs intended for future flight testing of MD^2WRP had defined dwell times and it was unknown how long the transient period would last before steady-state for any given task configuration. However, as research continued, the question arose as to whether

it would be best to tune MD^2WRP parameters for performance based solely on the steady-state while ignoring the transient, since PISR missions take place over a long time horizon. This reasoning is logical, but in order to pursue it further a method must be developed to characterize the length of the transient as a function of MD^2WRP parameter values and the D matrix for the task configuration. The groundwork for this was laid in this document in proving the periodicity of MD^2WRP, which included formulating a mathematical structure for the evolution of the task age vector. Understanding how the age vector evolves is one piece of the puzzle. The other piece is placing bounds on task ages given a D and specific MD^2WRP parameters. Together, the evolution of the age vector and the maximum age for each task drive the task visit sequence toward a steady-state visit pattern. Using this knowledge as a start, it may be possible to analytically calculate the length of the transient, or at least determine an upper-bound.

A major assumption throughout this research was that every vehicle had the same constant velocity and MD^2WRP parameters. In reality, it is likely PISR will be performed by a heterogeneous team of UAVs with differing velocities. Future research could include developing a parameter optimization method for such a scenario, perhaps first focusing on the same MD^2WRP parameters but different speeds and then vice versa.

The final section of Ch. IV demonstrated scenarios where vehicles and tasks were added and removed mid-mission. Though MD^2WRP was shown to be robust by allowing the vehicles to continue to service all tasks despite changes in the mission objects, the original MD^2WRP parameters were no longer optimal when the number of vehicles or tasks changed. Since this scenario is entirely likely to come up in the field, it would be useful to develop an algorithm to facilitate mid-mission re-optimization of MD^2WRP parameters when mission objects are added or removed.

As was done for Dubins and no-fly zone constraints, it would be helpful to know how much benefit is derived from mid-mission retuning in terms of performance and when it is necessary or can be ignored.

All simulations conducted in this research assumed tasks were point searches. Vehicles begin and end the task in the same location and the task takes zero time to service. As soon as the vehicle arrives, the task is considered complete. This assumption is representative of some tasks that may be required for an actual PISR mission (*e.g.* take a picture of a specific coordinate), but it may also be necessary to search a non-point objective, such as a road or field. These tasks introduce further complexity, since vehicles may start the task in one location and end in another. Additionally, they take a non-zero amount of time to complete. MD^2WRP can be modified to accommodate these tasks relatively easily by introducing a term to account for task service time. How these new factors affect the conclusions of this research based on point search tasks, however, requires more research.

Finally, we have assumed that task priorities were provided by the operator. MD^2WRP was optimized to maximize performance given the assigned task priority vector ($\mathbf{p}$) and task configuration (D). In reality, before any task selection algorithm is implemented, selecting appropriate task priorities to provide the desired task coverage (that is, each task receiving approximately the desired number of visits during a given period of time) is a research problem unto itself. In short, if optimizing the task selection algorithm solves a problem, optimizing the task priorities ensures the right problem is being solved.

5.3 Contributions

This work has made several contributions to the field of persistent monitoring, or PISR. In Ch. II, a thorough literature review is presented that summarizes many

methods available for PISR and the strengths, weaknesses, and challenges associated with each. In addition to the various methods reviewed, some general considerations of PISR were also discussed, such as the interplay between the transient and steady-state. We noted that with PISR methods such as TSP, the task visit sequence was static but there was a transient and steady-state phase to latency development. Whereas with utility methods, both latency and the task visit sequence undergo transient and steady-state phases. These considerations are important in deciding which type of method to use and how it will perform in a real-world PISR mission.

The core of this research revolved around a specific method of PISR task selection, MD^2WRP. This dissertation took the previously published MD^2WRP theory and used it as a basis for developing MD^2WRP for practical use. We introduced a normalized form of MD^2WRP to make it easier to select parameter values, even when teams of heterogeneous vehicles are used. We characterized the effect of the parameters β and $\mathbf{w}$ on agent behavior and proved that MD^2WRP is periodic in the steady-state. To deliver the best performance, we proposed a two-step heuristic parameter optimization which was validated by comparing the optimized MD^2WRP performance to single and multi-vehicle TSP methods on a variety of task configurations.

In assessing the ability of MD^2WRP to adapt to operational challenges, we showed that MD^2WRP could accommodate multiple types of mission restrictions, including Dubins contraints on vehicle motion, no-fly zones, return-to-base requirements, and the mid-mission addition and removal of tasks and vehicles. While these conclusions are certainly contributions, the methods we developed to test three of the operational factors are contributions in their own right. For Dubins constraints, we introduced a method of scaling the task map to determine when the vehicle turn radius became significant in calculating task values. To test the effect of no-fly zones,

we developed the Impact Ratio, IR, which defined how much the no-fly zone interfered with vehicles traveling between tasks. Finally, in evaluating the effect of the relative location of a base to the PISR tasks, we presented an offset ratio based on the geometric centroid of all tasks.

Finally, the PUMPS tool, which was used to generate all of the simulation data in this document, was developed from scratch with the intent of providing a flexible code base for further testing of PISR task selection algorithms. It has a modular design allowing for each vehicle to use a different type of task selection, vehicle pathing, and communication. Tasks of different priorities can be placed in any configuration, with the possibility to add support for more complex tasks in the future, such as road or area searches. The PUMPS tool is freely available to the community to use or modify in future research, which could be extended to other domains such as dynamic sensor tasking for space applications or other logistic operations where TSP-like solutions are sought. The code for the current PUMPS version, as of the publishing of this document, is located in Appendix A. For the most up-to-date version of PUMPS, please visit the Github repository (https://github.com/Sacaraster/PISR-Simulation).

5.4 Recommendations

To conclude this work, we provide recommendations about how to implement MD^2WRP as a PISR task selection method with an eye toward flight testing, as well as general recommendations about conducting PISR missions. These recommendations are based on reviewing the literature surrounding PISR as well as our experiences conducting this research.

One of our first research tasks was to select a performance measure for PISR. Throughout this work we based our metric on both the transient and steady-state phases of latency development. However, the usefulness of this choice depends on

mission requirements. If the mission has a predefined timespan and the ages of all tasks are zero at the start of the mission, a transient phase will necessarily exist and is likely to have a significant impact on performance. In this situation, the average weighted latency serves as a good metric. However, if PISR is to be conducted indefinitely, with vehicles relieving each other and effectively continuing the mission from the same state (*i.e.* the task ages known to the old vehicle are passed on to the new vehicle), then a metric based solely on the steady-state may be more appropriate, such as minimizing the maximum total latency or the maximum latency of any single task once steady-state is achieved.

In addition to considering transient and steady-state, selecting the type of performance metric is also dependent upon the mission. Average weighted latency, as used in this research, provides performance that ensures all tasks are being serviced in accordance with their priorities, but it provides no guarantees as to how long some tasks may have to wait for service. Since it is an average, one task may receive many visits over a short period of time while another experiences a long delay before service, but the latency of the two tasks offset each other. For some use cases, this may be fine. However, minimizing maximum weighted latency places an upper bound on maximum idle time which provides a worst-case guarantee. Minimizing the maximum weighted latency can be done in one of two ways: as a summation of the latency across all tasks, which emphasizes minimizing latency from a system perspective, possibly at the expense of individual tasks, or by considering the maximum latency of any single task, which provides a firm upper bound on the latency of any single task but possibly introduces system-wide inefficiencies.

A major part of this research was developing an optimization method for MD^2WRP parameters given the number of vehicles and a specific task configuration. However, it may be desired for the sake of practicality to use predetermined parameters that

may be sub-optimal for a given scenario, but will work "well enough". Based on our findings and time spent running various scenarios, we recommend using $\beta = 5.0$ and $\mathbf{w} = \mathbb{1}$ as default parameter values. These settings often yield performance that is close to that of the optimized values, which is likely good enough for vehicles to be operationally effective. Table 5.1 summarizes the performance of the recommended β of 5.0 compared to the optimal β for the four operational scenarios in Fig. 3.9. The simulations in Table 5.1 assume a team of homogeneous vehicles, all vehicles beginning at Task 1, use of the CxBD communication mode, and a mission duration of $20000s$.

Table 5.1. $\bar{L}$ comparison between optimized and recommended β.

Map	1 Vehicle	2 Vehicles	3 Vehicles	4 Vehicles
Circle - Optimized	6859	3569	2682	1922
Circle - $\beta = 5.0$	6859	4152	2802	1957
Clusters - Optimized	5369	2867	1179	951
Clusters - $\beta = 5.0$	6955	3238	1215	1005
Random - Optimized	6960	3558	2391	1729
Random - $\beta = 5.0$	7107	3598	2455	1825
Grid - Optimized	5773	2896	2044	1458
Grid - $\beta = 5.0$	6459	2897	2288	1494

In order to maximize the benefit of MD^2WRP, some method of dynamically re-optimizing parameters should be included on flight software. In this research, parameters were optimized manually based on the number of vehicles and task configuration and then those values were assigned to vehicles. However, it should be possible to wrap the MD^2WRP task selection algorithm in an outer algorithm that calculates optimal parameter values anytime mission objects change, based on currently known state information. This would fully remove the user from the process, allowing more resources to be focused on data analysis rather than data acquisition.

We conducted several comparisons between MD^2WRP and various TSP solutions for PISR. The TSP solutions were useful for validating our MD^2WRP parameter

optimization method, but the process of coding the TSP solutions to make these comparisons also provided insight into the merits and drawbacks of utility versus TSP methods in general. While TSP methods require centralized coordination and combinatorial optimization algorithms, the basic methods we tested (n-spaced and k-subtours) are relatively simple and would probably be the preferred primary method in a mission where tasks are simple (*e.g.* point search tasks), mission objects are unlikely to change often, and satellite communication links are readily available in the event vehicle routes must be re-planned and re-distributed. Also, TSP methods provide guaranteed task revisit times and vehicles act in a predictable way, which might be desirable for users, if not posing some risk in terms of enemy awareness. Utility methods, on the other hand, would be preferred as the primary task selection method when tasks are complex (*e.g.* road searches), mission objects are introduced and removed frequently, and satellite communications are poor, since the distribution of vehicle routes is not required. Or, utility methods could serve as a secondary method of task selection when TSP methods fail or become infeasible. For instance, if long-haul satellite communications become unstable, vehicles could automatically switch to a utility method in order to continue to service tasks in the absence of centralized planning, instead relying on shortwave inter-vehicle communications. In that scenario, return-to-base requirements could be enforced to deliver the collected data.

The enforcement of a return-to-base (RTB) requirement with MD^2WRP was explored in Sec. 4.3.3. The conclusion was that meeting RTB criteria with MD^2WRP was possible, but doing so had the potential to introduce inefficiencies depending on the location of the base relative to the tasks. There is a variant of the TSP that enforces time windows on task visits that provides a guarantee on the RTB frequency while maximizing task visits (discussed in Sec. 2.5.2), but the algorithm quickly be-

comes intractable as the number of vehicles and tasks increases and may have no solution at all if the problem is not formulated with care. A simpler option that is compatible with MD^2WRP would be to enforce a hard-coded RTB requirement outside of the main MD^2WRP algorithm. For example, the RTB wrapper could continually track the travel time back to the base node from the current location, and if the next task visit would bust the deadline, the vehicle would RTB before visiting another task. In this way, the vehicles would continue to use MD^2WRP like normal, without including the base as a "task", but still maintain consistent visits to the base node.

Finally, this work has addressed the problem of how to select tasks for gathering PISR data, but how PISR data should be stored, accessed, and most importantly, used remains an open question. The purpose of PISR is to provide continual real-time surveillance but also to develop long term data about a region for trends analysis. If live streaming data is available from the vehicles, it should be accessible by an operator in real-time but also stored for future analysis. If data must be ferried back to base, operators should be notified when new data is available. The continual nature of PISR will result in large amounts of accumulated data. It would be helpful to store the data by both location and time, with a visual interface for easy navigation by a human user (software such as Google Earth or Systems Toolkit provide this capability) but the database should also provide interfaces for data mining AI routines, which excel at trends analysis.

Appendix A. PUMPS Code

The code for the current version of PUMPS, as of the time of this publishing, is provided below. This code is intended to serve as a reference for understanding the results presented in this document. It includes code for the main program as well as each class file. For a working and up-to-date version of the PUMPS tool that includes all external dependencies, setup files, and other auxiliary functions, please visit the Github repository at https://github.com/Sacaraster/PISR-Simulation.

A.1 Main

```
from __future__ import division


import shutil
import os
import sys
import math
import pickle
import dubins


import numpy as np


from generateMapCoordinates import generateMapCoordinates
from Classes.TaskClass import Task
from Classes.VehicleClass import Vehicle



def loadTaskConfig(trade_config):
```

```
task_geometry = trade_config['task_geometry']
x_coords, y_coords = generateMapCoordinates(task_geometry)
priorities_vector = trade_config['priorities_vector']
init_ages_vector = trade_config['init_ages_vector']
task_activation_times_vector =
↪  trade_config['task_activation_times_vector']
task_termination_times_vector =
↪  trade_config['task_termination_times_vector']

print ''
print '      Instantiating task objects ({} tasks, "{}"
↪  map)...'.format(len(x_coords), task_geometry)

task_vector = []
for index, task in enumerate(x_coords):
    taskID = index+1
    x_coord = x_coords[index]
    y_coord = y_coords[index]
    priority = priorities_vector[index]
    init_age = init_ages_vector[index]
    t_activate = task_activation_times_vector[index]
    t_terminate = task_termination_times_vector[index]
    print '         Task {} @ ({},{}), Priority={}'.format(taskID,
    ↪  x_coord, y_coord, priority)
```

```
            print '            Initial age={}, Activation time: {},
            ↪  Termination time:
            ↪  {}'.format(init_age,t_activate,t_terminate)
            taskObj = Task(taskID, x_coord, y_coord, priority, init_age,
            ↪  t_activate, t_terminate)  #INSTATIATE TASK OBJECT
            task_vector.append(taskObj)    #ADD OBJECT TO VECTOR OF TASK
            ↪  OBJECTS

        return task_vector, task_geometry

    def loadVehicleConfig(trade_config, task_vector):

        init_locations_vector = trade_config['init_locations_vector']
        init_headings_vector = trade_config['init_headings_vector']
        veh_speeds_vector = trade_config['veh_speeds_vector']
        veh_bank_angles_vector = trade_config['veh_bank_angles_vector']
        veh_activation_times = trade_config['veh_activation_times']
        veh_termination_times = trade_config['veh_termination_times']

        print ''
        print '      Instantiating vehicle objects ({}
        ↪  vehicles)...'.format(len(init_locations_vector))

        vehicle_vector = []
        for index, vehicle in enumerate(init_locations_vector):
            vehicleID = int((index+1)*100)
```

```
print '         Vehicle', vehicleID
init_location = init_locations_vector[index]
init_location = task_vector[init_location-1]   #re-assign
↪  init_location to be a task object
print '            Initial Task:', init_location.ID
init_heading = init_headings_vector[index]
↪  #stored in radians
print '            Initial
↪  Heading:',init_heading*(180/math.pi), 'degrees.'
↪  #output in degrees
veh_speed = veh_speeds_vector[index]
print '            Vehicle Speed:', veh_speed, 'meters/sec.'
veh_bank_angle = veh_bank_angles_vector[index]
↪  #stored in radians
print '            Vehicle Bank Angle: ',
↪  veh_bank_angle*(180/math.pi), 'degrees.'          #output
↪  in degrees
turn_radius = veh_speed**2/(9.807*math.tan(veh_bank_angle))
print '            Vehicle Turn Radius: ',
↪  np.around(turn_radius,1), 'meters.'
veh_t_activate = veh_activation_times[index]
print '            Vehicle Activation Time: ', veh_t_activate,
↪  'secs.'
veh_t_terminate = veh_termination_times[index]
print '            Vehicle Termination Time: ',
↪  veh_t_terminate, 'secs.'
```

```
        #Instantiate the vehicle object
        vehicleObj = Vehicle(index, vehicleID, init_location,
        ↪  init_heading, veh_speed, turn_radius, veh_t_activate,
        ↪  veh_t_terminate)

        vehicle_vector.append(vehicleObj)

    #Load the modules for each vehicle
    vehicle_vector = loadRoutingConfig(trade_config, vehicle_vector,
    ↪  task_vector)
    vehicle_vector = loadPathingConfig(trade_config, vehicle_vector,
    ↪  task_vector)
    vehicle_vector = loadCommConfig(trade_config, vehicle_vector)
    vehicle_vector = loadDatabaseConfig(trade_config, vehicle_vector,
    ↪  task_vector)

    return vehicle_vector

def loadRoutingConfig(trade_config, vehicle_vector, task_vector):

    print ''
    print '      Adding Routing modules...'

    routing_type = trade_config['routing_type']
    beta = trade_config['beta']
```

```
    ws_vector = trade_config['ws_vector']
    distance_measure = trade_config['distance_measure']
    tours_vector = trade_config['tours_vector']
    veh_start_index_vector = trade_config['veh_start_index_vector']

    routing_data = [routing_type, beta, ws_vector, distance_measure,
    ↪  tours_vector, veh_start_index_vector]

    for vehicle in vehicle_vector:
        vehicle.add_routing(routing_data, task_vector)
        print '         Vehicle {} Routing Data:'.format(vehicle.ID)
        vehicle.routing.print_routing_data()

    return vehicle_vector

def loadPathingConfig(trade_config, vehicle_vector, task_vector):

    print ''
    print '      Adding Pathing modules...'

    pathing_data = trade_config['pathing_type']

    for vehicle in vehicle_vector:
        vehicle.add_pathing(pathing_data)
        print '         Vehicle {} Pathing Data:'.format(vehicle.ID)
        try:
```

```
            vehicle.pathing.calc_nfz_impact_rating(pathing_data,
            ↪ task_vector)
        except:
            pass
        vehicle.pathing.print_pathing_data()

    return vehicle_vector

def loadCommConfig(trade_config, vehicle_vector):

    print ''
    print '      Adding Communication modules...'

    comm_mode = trade_config['comm_mode']

    comm_data = [comm_mode]

    for vehicle in vehicle_vector:
        vehicle.add_comm(comm_data)
        print '         Vehicle {} Comm Data: {}'.format(vehicle.ID,
        ↪ comm_data)

    return vehicle_vector

def loadDatabaseConfig(trade_config, vehicle_vector, task_vector):
```

```
    print ''
    print '      Adding Database modules...'

    database_items = trade_config['database_items']

    for vehicle in vehicle_vector:
        vehicle.add_database(database_items, vehicle_vector,
        ↪  task_vector)
        print '         Vehicle {} Database Items:
        ↪  {}'.format(vehicle.ID, database_items)

    return vehicle_vector


def main():

    ###################################################################
    ###   LOAD CONFIGURATION FILES AND INSANTIATE OBJECTS   #####
    ###################################################################
    #Argument supplying the location of the simulation configuration
    ↪  files
    sim_path = sys.argv[1]

    #Create the directory where simulation data will be saved
    sim_data_path = './Data/'
```

```
print '\nSimulation data will be saved to
↪  {}\n'.format(sim_data_path)

#open every pickle file in the directory
for file in os.listdir(sim_path):
    if file.endswith("_Config.pickle"):
        trade_config_pickle = '{0}{1}'.format(sim_path, file)
        print 'Opening \"', trade_config_pickle, '\"\n'
        trade_config = pickle.load(open(trade_config_pickle,
        ↪  "rb"))
        tradeID = trade_config['tradeID']
        print '**************************************'
        print '          Loading TradeID={}
        ↪  '.format(tradeID)
        print '**************************************'
        sim_length = trade_config['sim_length']
        sim_length_visits = sim_length[0]
        sim_length_time = sim_length[1]

        #Load the task parameters; returns a vector of task
        ↪  objects
        task_vector, task_geometry = loadTaskConfig(trade_config)

        #Load the vehicle parameters and all modules; returns a
        ↪  vector of vehicle objects
```

```
        vehicle_vector = loadVehicleConfig(trade_config,
        ↪  task_vector)

##############################################################
##############################################################

        np.set_printoptions(suppress=True)    #Don't print in
        ↪  scientific notation
        np.set_printoptions(threshold='nan')    #Don't truncate
        ↪  large arrays when printing

        ### MAIN SIM LOOP ###
        print ''
        print '    *****************************************'
        print '    ******    Beginning Simulation    **********'
        print '    *****************************************'

        visit_order = []
        task_ages = []
        visit_num = 1
        time = 0
        while ((visit_num < sim_length_visits+1) & (time <
        ↪  sim_length_time+1)):

            print '    Task Visit #[{}]\n'.format(visit_num)
```

```
# Decide which vehicle makes the next task selection,
↪ based on earliest arrival time
# Only vehicle's within their active window are
↪ considered
decider = min((vehicle for vehicle in vehicle_vector
↪ if (vehicle.t_terminate >
↪ vehicle.routing.arrival_time)),
    key=lambda x: x.routing.arrival_time)

print '   Vehicle', decider.ID, 'is selecting the next
↪ task.'
print '      Just arrived: Task {} @ {}
↪ secs.'.format(decider.routing.destination.ID,
↪ decider.routing.arrival_time)
print '      Vehicle heading: {}
↪ degrees'.format(decider.heading*(180/math.pi))

# Update deciding vehicle location (old destination
↪ is new location)
decider.location = decider.routing.destination
decider.time = decider.routing.arrival_time

# Zero-out age of visited task in deciding vehicle's
↪ age tracker
decider.database.age_tracker[int(
        decider.routing.destination.ID)-1]= 0.0
```

```
# Increment task object ages by travel time (time of
↪  this arrival less time of previous arrival)
↪
for task in task_vector:
    # Only tasks within their active window are
    ↪  incremented, all others have age '0'
    if ((decider.time >= task.t_activate) &
    ↪  (decider.time < task.t_terminate)):
        task.age = task.age + (decider.time-time)
        # A task does not begin accruing age until
        ↪  it's activation time
        if (time < task.t_activate):
            task.age = task.age -
            ↪  (task.t_activate-time)
    else:
        task.age = 0


# First, save task ages without setting visited task
↪  age to 0
task_age_vector = []
task_age_vector.append(decider.time)  #first entry in
↪  age vector is timestamp
for task in task_vector:
    task_age_vector.append(task.age)
task_ages.append(task_age_vector)
```

```
# Zero out age of task that vehicle just arrived at
task_vector[decider.routing.destination.ID-1].age = 0

# Now, save task ages again (@ +.01s) with age of
↪  visited task at 0
#   This is needed to perform the latency
↪  calculations in the analysis script
task_age_vector = []
task_age_vector.append(decider.time+.01)  #first
↪  entry in age vector is timestamp
for task in task_vector:
    task_age_vector.append(task.age)
task_ages.append(task_age_vector)

# Document vehicle task visits, trajectory
↪  information, and task ages
# visit_order format is [vehicleID, task, visit_time,
↪  trajectory]
visit_order.append([decider.ID, decider.location.ID,
↪  decider.time, decider.pathing.trajectory])

print '      Vehicle {} age tracker = \n
↪  {}'.format(decider.ID,
↪  np.around(decider.database.age_tracker, 3))
```

```
print '      True task ages = \n
↪  {}'.format(np.around(task_age_vector[1:], 3))
print '      Vehicle {} vehicle tracker =
↪  \n{}'.format(decider.ID,
↪  np.around(decider.database.vehicle_tracker, 3))

#Select the next task to visit
#   (Updates vehicle's destination)
decider.routing.get_next_task(decider, task_vector)

#Calculate path to selected task
#   (Updates trajectory, arrival_time, &
↪  current_heading)
decider.pathing.get_path(decider)

# Increment task ages in vehicle's own age tracker by
↪  travel time (time of the planned arrival less
↪  current time)
tij = decider.routing.arrival_time-decider.time
for task in task_vector:
    # Only tasks within their active window are
    ↪  incremented, all others have age '0'
    if ((decider.routing.arrival_time >=
    ↪  task.t_activate) &
    ↪  (decider.routing.arrival_time <
    ↪  task.t_terminate)):
```

```
                decider.database.age_tracker[task.ID-1] =
                ↪  decider.database.age_tracker[task.ID-1] +
                ↪  tij
                #A task does not begin accruing age until
                ↪  it's activation time
                if (decider.time < task.t_activate):
                    decider.database.age_tracker[task.ID-1] =
                    ↪  decider.database.age_tracker[task.ID-1]
                    ↪  - (task.t_activate-decider.time)
            else:
                decider.database.age_tracker[task.ID-1] = 0

        #Communicate
        decider.comm.talk(decider, vehicle_vector)

        #Update visit number and current simulation time
        ↪
        visit_num += 1
        time = decider.time

        print ''
        print '   Ready for next Task!\n'
        print '
        ↪  ****************************************\n'

    # #Save the final task visits for each vehicle
```

```
# for vehicle in vehicle_vector:
#     visit_order.append([vehicle.ID,
↪  vehicle.routing.destination.ID,
↪  vehicle.routing.arrival_time,
↪  vehicle.pathing.trajectory])

visit_order = sorted(visit_order,key=lambda x: x[2])
visit_order = np.array(visit_order, dtype=object)



print '   ****************************************'
print '   ******     Simulation Complete   **********'
print '   ****************************************'

#Display the visit history to screen
print ''
print '   Simulation Visit History:'
print(visit_order[:, 0:3])
print ''

#Save each trade into a pickle file...
print '   Pickling results...'
trade_results_pickle =
↪  '{0}Trade_{1}_Results.pickle'.format(sim_data_path,
↪  tradeID)
```

```
            trade_results = [visit_order, task_ages, task_vector,
            ↪  vehicle_vector, tradeID, task_geometry]
            pickle.dump(trade_results, open(trade_results_pickle,
            ↪  "wb"))
            print '   Results pickled.'


if __name__ == '__main__':

    main()
```

A.2 Classes

A.2.1 The Vehicle Class.

```
from __future__ import division
import random
import itertools
import math
import numpy as np
import dubins

from RoutingClass import RoutingFactory
from PathingClass import PathingFactory
from CommunicationClass import CommunicationFactory
from DatabaseClass import Database

class Vehicle:
```

```
"""A class for PISR vehicles."""

#The base vehicle class holds attributes of the physical vehicle only (e.g. speed and heading).
#Vehicles implement "modules" that perform other functions. For example, every vehicle loads a specific
#type of "Pathing" module, which is defined by the Pathing class. So a vehicle that flys Euclidean
#paths implements the Euclidean sublcass of the Pathing module.
def __init__(self, _indexer, ID, init_location, init_heading, speed, turn_radius, t_activate, t_terminate):

    self._indexer = _indexer   #since IDs are usually 100, 200, etc, this makes referencing vehicles easier
    self.ID = ID
    self.location = init_location   #a task object (vehicle is located at a task)
    self.time = t_activate                #current time for vehicle
    self.heading = init_heading
    self.speed = speed
    self.turn_radius = turn_radius
    self.t_activate = t_activate
    self.t_terminate = t_terminate
```

```
#Add the "Routing" module to the vehicle. This determines how
↪  the vehicle selects tasks.
def add_routing(self, routing_data, task_vector):
    routing_factory = RoutingFactory()
    self.routing =
    ↪  routing_factory.get_routing_module(routing_data, self,
    ↪  task_vector)

#Add the "Pathing" module to the vehicle. This determines how
↪  the vehicle travels between tasks.
def add_pathing(self, pathing_data):
    pathing_factory = PathingFactory()
    self.pathing =
    ↪  pathing_factory.get_pathing_module(pathing_data)

#Add the "Communication" module to the vehicle. This lets
↪  vehicles communicate.
def add_comm(self, comm_data):
    comm_factory = CommunicationFactory()
    self.comm = comm_factory.get_comm_module(comm_data)

#Store task and sister vehicle information based on the
↪  vehicle's type of "Database" module
def add_database(self, database_items, vehicle_vector,
↪  task_vector):
```

```
        self.database = Database(database_items, vehicle_vector,
        ↪  task_vector)
```

A.2.2 The Task Class.

```
class Task:

    """A class for PISR tasks"""

    def __init__(self, ID, x, y, pri, init_age, t_activate,
    ↪  t_terminate):
        self.ID = ID
        self.location = [x,y]
        self.priority = pri
        self.age = init_age
        self.t_activate = t_activate
        self.t_terminate = t_terminate
```

A.2.3 The Routing Class.

```
import math
import numpy as np
from abc import ABCMeta, abstractmethod
```

```
import PathingClass

class Routing(object):

    """A class for routing of PISR Vehicles."""

    __metaclass__ = ABCMeta

    @abstractmethod
    def get_next_task(self):
        raise NotImplementedError("You must implement a get_next_task
        ↪  method for this routing type!")

    @abstractmethod
    def print_routing_data(self):
        raise NotImplementedError("You must implement a
        ↪  print_routing_data method for this routing type!")

class MD2WRP_Routing(Routing):

    def __init__(self, vehicle, task_vector, beta, w,
    ↪  distance_measure):
        self.type = 'MD2WRP'
        self.destination = vehicle.location    # destination task (a
        ↪  task object)
        self.arrival_time = vehicle.time
```

```
        self.beta = beta
        self.w = w
        #generate a pathing object for calculating longest distance
        ↪  between tasks
        pathing_object = PathingClass.Euclidean_Pathing()
        avgDistance, longestDistance =
        ↪  pathing_object.calcDistanceMatrixData(task_vector)
        self.norm_factor = longestDistance/vehicle.speed
        # if distance_measure == 1:
        self.distance_measure = distance_measure
        # if distance_measure == 2:
        #     self.distance_measure = ['Dubins']
        # if distance_measure == 3:
        #     self.distance_measure = ['Tripath']

    def print_routing_data(self):
        print '            Type:', self.type
        print '            Beta:', self.beta
        print '            w:', self.w
        print '            Measure:', self.distance_measure
        print '            Norm Factor:', self.norm_factor


    def get_next_task(self, vehicle, task_vector):
```

```
#Calculate travel times to every task based on the type of
↪  distance measurement
#  (This is not necessarily the physical pathing of the
↪  vehicle)
pathing_factory = PathingClass.PathingFactory()
pathing_object =
↪  pathing_factory.get_pathing_module(vehicle.routing.distance_measure)
measured_times_and_headings =
↪  pathing_object.get_best_paths(vehicle, task_vector)
measured_times = measured_times_and_headings[:,1]

utilities = []
for index, tij in enumerate(measured_times):
    age_modifier = 0   #used to adjust the age of a task due
    ↪  to visits from other vehicles
    for other_arrival_index, other_arrival in
    ↪  enumerate(vehicle.database.vehicle_tracker[:, 0]):
        if ((other_arrival == index+1) & (other_arrival_index
        ↪  != vehicle._indexer)):
            #If another vehicle will arrive before me, reduce
            ↪  the age of the task under consideration
            if
            ↪  vehicle.database.vehicle_tracker[other_arrival_index,
            ↪  1] < (vehicle.time+tij):
                age_modifier = (vehicle.time+tij)- \
                        vehicle.database.vehicle_tracker[ \
```

```
                other_arrival_index, 1]
            #...but only reduce the age if the other
            ↪ vehicle's visit will result in a lower
            ↪ future age than at my arrival time
            if age_modifier <
            ↪ (vehicle.database.age_tracker[index]+tij):
                print '      *** Task {} age changed to
                ↪ {}s! (Interim visit)'.format(index+1,
                ↪ np.around(age_modifier,3), )
                age_modifier = -1*age_modifier +
                ↪ (vehicle.database.age_tracker[index]+tij)
                ↪ #negates all other age info
            else:
                age_modifier = 0
        #If another vehicle will be arriving after me (or
        ↪ at the same time), don't go to that task
        if
        ↪ vehicle.database.vehicle_tracker[other_arrival_index,
        ↪ 1] >= (vehicle.time+tij):
            age_modifier =
            ↪ (vehicle.database.age_tracker[index]+tij)
            ↪ #by making the effective (age + tij) term
            ↪ zero, utility=0
            print '      *** Task {} utility set to zero!
            ↪ (Conflict)'.format(index+1)
```

```
        utility = (math.exp(-vehicle.routing.beta* \
            (tij/vehicle.routing.norm_factor))*vehicle.routing.w[index]*
            ((vehicle.database.age_tracker[index]+tij- \
            age_modifier)/ vehicle.routing.norm_factor))*100000
        utilities.append(utility)
    for index, utility in enumerate(utilities):
        if vehicle.time < task_vector[index].t_activate:
            utilities[index] = 0
            print '      *** Task {} utility set to zero! (Not yet
             ↪  active)'.format(index+1)
        if vehicle.time >= task_vector[index].t_terminate:
            utilities[index] = 0
            print '      *** Task {} utility set to zero!
             ↪  (Terminated)'.format(index+1)

    print '      Calculated utilities for each task:'
    for index, task_utility in enumerate(utilities):
        print '         Task {} utility = {}'.format(index+1,
         ↪  task_utility)
    max_utility = max(utilities)

    selected_task = [index for index, utility in
     ↪  enumerate(utilities) if (utility == max_utility)]  #ID of
     ↪  selected task
    selected_task = task_vector[selected_task[0]]   #the actual
     ↪  task object
```

```
            print ''
            print '      Task {} has the highest utility.
            ↪  ({})'.format(selected_task.ID, max_utility)

            vehicle.routing.destination = selected_task

class Manual_Routing(Routing):

    def __init__(self, vehicle, seq_vector, veh_start_index_vector):
        self.type = 'Manual'
        self.destination = vehicle.location   # destination task (a
        ↪  task object)
        self.arrival_time = vehicle.time
        self.current_stop = veh_start_index_vector[vehicle._indexer]
        ↪  #not a task, but the index in the sequence
        self.sequence_vector = seq_vector[vehicle._indexer]

    def print_routing_data(self):
        print '            Type:', self.type
        print '            Sequence:', self.sequence_vector

    def get_next_task(self, vehicle, task_vector):
        print "      Selecting next task in the sequence,",
        ↪  self.sequence_vector
        print '      Currrent task:
        ↪  {}'.format(self.sequence_vector[self.current_stop])
```

```
            print '      Current task index: {}'.format(self.current_stop)

            #Increase the current_stop counter by 1
            self.current_stop += 1
            if self.current_stop > len(self.sequence_vector)-1:    #reset
            ↪  stop counter to 0 when at the end of the sequence
                self.current_stop = 0


            selected_task = self.sequence_vector[self.current_stop]
            print '      Next task: {}'.format(selected_task)
            print '      Next task index: {}'.format(self.current_stop)

            selected_task = task_vector[selected_task-1]    #the actual
            ↪  task object

            vehicle.routing.destination = selected_task


class RoutingFactory:
    def get_routing_module(self, routing_data, vehicle, task_vector):
        if routing_data[0] == 'MD2WRP':
            return MD2WRP_Routing(vehicle, task_vector,
            ↪  routing_data[1], routing_data[2], routing_data[3])
        elif routing_data[0] == 'Manual':
```

```
            return Manual_Routing(vehicle, routing_data[4],
            ↪  routing_data[5])
        else:
            raise NotImplementedError("Unknown routing type.")
```

A.2.4 The Pathing Class.

```
import os
import subprocess
import math
import dubins
import numpy as np
from abc import ABCMeta, abstractmethod

class Pathing(object):

    """A class for pathing of PISR Vehicles."""

    __metaclass__ = ABCMeta

    @abstractmethod
    def get_path(self):
        raise NotImplementedError("You must implement a get_path
        ↪  method for every Pathing type!")

    @abstractmethod
    def get_best_paths(self):
```

```
            raise NotImplementedError("You must implement a get_best_paths
            ↪  method for every Pathing type!")

        @abstractmethod
        def print_pathing_data(self):
            raise NotImplementedError("You must implement a
            ↪  print_pathing_data method for every Pathing type!")

    class Euclidean_Pathing(Pathing):

        def __init__(self):
            self.type = 'Euclidean'
            self.trajectory = []

        def print_pathing_data(self):
            print '            Type:', self.type

        def get_path(self, vehicle):

            x0 = vehicle.location.location[0]
            y0 = vehicle.location.location[1]

            x1 = vehicle.routing.destination.location[0]
            y1 = vehicle.routing.destination.location[1]

            #Calculate length of Euclidean path
```

```
        path_length = math.sqrt(math.pow(x1-x0, 2)+math.pow(y1-y0, 2))
        #Calculate trajectory
        trajectory = np.array([[x0, y0], [x1, y1]])

        #update vehicle states
        self.trajectory = trajectory
        vehicle.routing.arrival_time = vehicle.time +
        ↪  path_length/vehicle.speed
        vehicle.heading = 0

        print '      Travel time to Task {} =
        ↪  {}'.format(vehicle.routing.destination.ID,
        ↪  path_length/vehicle.speed)
        print '      Arriving @ {}
        ↪  secs.'.format(np.around(vehicle.routing.arrival_time, 3))
        print '      Arrival heading: {}
        ↪  degrees'.format(np.around(vehicle.heading*(180/math.pi),1))

    def get_best_paths(self, vehicle, task_vector):

        times_and_headings = []

        x0 = vehicle.location.location[0]
        y0 = vehicle.location.location[1]

        #For every candidate task...
```

```
for index, task in enumerate(task_vector):

    #Coordinates of candidate task
    x1 = task.location[0]
    y1 = task.location[1]

    #Calculate the distance between the current location and
    ↪  candidate task
    dist = math.sqrt(math.pow(x1-x0, 2)+math.pow(y1-y0, 2))

    #Convert distance to travel time
    time = dist/vehicle.speed

    #Save the travel time to each task
    times_and_headings.append([task.ID, time, 0])

times_and_headings = np.array(times_and_headings)

print '      Shortest measured times to each task:'
print ''
print times_and_headings[:,0:2]
print ''

return times_and_headings
```

```
    def calcDistanceMatrixData(self, task_vector):
        cxyVector = []
        for task in task_vector:
            x = task.location[0]
            y = task.location[1]
            cxy = x+y*1j
            cxyVector.append(cxy)
        cxyVector = np.array([cxyVector], dtype=complex)

        distanceMatrix = abs(cxyVector.T-cxyVector)

        longestDistance = np.max(distanceMatrix)
        avgDistance = np.sum(distanceMatrix)/ \
                ((distanceMatrix.shape[0]**2)-distanceMatrix.shape[0])
                ↪ #don't divide by diaganol entries, which are
                ↪ zero

        return avgDistance, longestDistance

class Dubins_Pathing(Pathing):

    def __init__(self):
        self.type = 'Dubins'
        self.trajectory = []

    def print_pathing_data(self):
```

```
        print '            Type:', self.type

    def get_path(self, vehicle):

        x0 = vehicle.location.location[0]
        y0 = vehicle.location.location[1]
        theta0 = vehicle.heading

        x1 = vehicle.routing.destination.location[0]
        y1 = vehicle.routing.destination.location[1]

        path_length_vector = []
        #Discretized arrival headings (try each of these and pick
        ↪  the one with the shortest travel distance)
        thetas = np.arange(0, 20, 1.25)*(math.pi/10)
        #If arriving at the current task at the current heading,
        ↪  slightly change arrival angle (prevents travel time of
        ↪  zero)
        for theta_index, theta in enumerate(thetas):
            if (int(vehicle.location.ID-1) ==
            ↪  vehicle.routing.destination.ID) & (theta0==theta):
                thetas[theta_index] = theta0 + 0.0174533    #add 1
                ↪  degree to arrivalangle
        for theta1 in thetas:
            #Cacluate the path length for given arrival angle
```

```
    path_length = dubins.path_length((x0, y0, theta0), (x1,
    ↪  y1, theta1), vehicle.turn_radius)
    path_length_vector.append(path_length)

#find the shortest travel distance for all calculated
↪  arrival heading options
min_dist = min(path_length_vector)
min_dist_index = np.argmin(path_length_vector)
arrival_heading = thetas[min_dist_index]

#Calculate trajectory to destination
trajectory, _ = dubins.path_sample((x0, y0, theta0), (x1, y1,
↪  arrival_heading), vehicle.turn_radius, 20)

#update vehicle states
self.trajectory = trajectory
vehicle.routing.arrival_time = vehicle.time +
↪  min_dist/vehicle.speed
vehicle.heading = arrival_heading

print '      Travel time to Task {} =
↪  {}'.format(vehicle.routing.destination.ID,
↪  min_dist/vehicle.speed)
print '      Arriving @ {}
↪  secs.'.format(np.around(vehicle.routing.arrival_time, 3))
```

```
        print '      Arrival heading: {}
        ↪  degrees'.format(np.around(vehicle.heading*(180/math.pi),1))

    def get_best_paths(self, vehicle, task_vector):

        times_and_headings = []

        #Coordinates of current location and current heading
        x0 = vehicle.location.location[0]
        y0 = vehicle.location.location[1]
        theta0 = vehicle.heading

        #For every candidate task...
        for index, task in enumerate(task_vector):
            path_length_vector = []
            #Coordinates of candidate task
            x1 = task.location[0]
            y1 = task.location[1]
            #Discretized arrival headings (try each of these and pick
            ↪  the one with the shortest travel distance)
            thetas = np.arange(0, 20, 1.25)*(math.pi/10)
            #If arriving at the current task at the current heading,
            ↪  slightly change arrival angle (prevents travel time
            ↪  of zero)
            for theta_index, theta in enumerate(thetas):
```

```
            if (int(vehicle.location.ID-1) == index) &
            ↪  (theta0==theta):
                thetas[theta_index] = theta0 + 0.0174533    #add 1
                ↪  degree to arrivalangle
        for theta1 in thetas:
            #Cacluate the path length for given arrival angle
            path_length = dubins.path_length((x0, y0, theta0),
            ↪  (x1, y1, theta1), vehicle.turn_radius)
            path_length_vector.append(path_length)
        #find the shortest travel distance for all arrival
        ↪  heading options
        min_dist = min(path_length_vector)
        min_dist_index = np.argmin(path_length_vector)
        heading = thetas[min_dist_index]
        #calculate travel time based on vehicle speed
        time = min_dist/vehicle.speed
        times_and_headings.append([task.ID, time, heading])

    times_and_headings = np.array(times_and_headings)

    print '      Shortest measured times to each task:'
    print ''
    print times_and_headings[:,0:2]
    print ''

    return times_and_headings
```

```
class Tripath_Pathing(Pathing):

    def __init__(self, task_geometry, nfz):
        self.type = 'Tripath'
        self.map = task_geometry   #tells Tripath which map is in use
        self.nfz = nfz             #tells Tripath which no-fly zone
        ↪ to use (an integer)
        self.trajectory = []
        self.nfz_impact = 0      #ratio of average travel distance
        ↪ with nfz to w/out nfz


    def print_pathing_data(self):
        print '            Type:', self.type
        print '            Map:', self.map
        print '            NFZ:', self.nfz
        print '            NFZ Impact Rating:', self.nfz_impact

    def get_path(self, vehicle):

        x0 = vehicle.location.location[0]
        y0 = vehicle.location.location[1]

        x1 = vehicle.routing.destination.location[0]
```

```
y1 = vehicle.routing.destination.location[1]


#Cacluate the path to the task
FNULL = open(os.devnull, 'w')    #This prevents a terminal
↪  window from popping up each time Tripath is called
subprocess.call(
        '/home/chris/Research/PISR_Sim_NGpp/Tripath_custom/bin/./setut
        ↪  {} {} {} {} {} {}'.format(
        x0, y0, x1, y1, vehicle.pathing.map,
        ↪  vehicle.pathing.nfz),
        cwd='/home/chris/Research/PISR_Sim_NGpp/Tripath_custom/bin/',
        stdout=FNULL, shell=True)
path_data = np.genfromtxt(
        '/home/chris/Research/PISR_Sim_NGpp/Tripath_custom/bin/path.txt',
        delimiter = ",")  #path_data is the trajectory data
xPath = path_data[:,0]
yPath = path_data[:,1]


#Calculate the length of the path
path_length = 0
for ind, entry in enumerate(xPath[0:-1]):
    path_length = path_length +
    ↪  math.sqrt(math.pow(xPath[ind+1]-xPath[ind],
    ↪  2)+math.pow(yPath[ind+1]-yPath[ind], 2))


#update vehicle states
```

```
        self.trajectory = path_data
        vehicle.routing.arrival_time = vehicle.time +
        ↪  path_length/vehicle.speed
        vehicle.heading = 0

        print '      Travel time to Task {} =
        ↪  {}'.format(vehicle.routing.destination.ID,
        ↪  path_length/vehicle.speed)
        print '      Arriving @ {}
        ↪  secs.'.format(np.around(vehicle.routing.arrival_time, 3))
        print '      Arrival heading: {}
        ↪  degrees'.format(np.around(vehicle.heading*(180/math.pi),1))


    def get_best_paths(self, vehicle, task_vector):

        times_and_headings = []  #note...Euclidean, so heading is
        ↪  always '0'

        #Coordinates of current location
        x0 = vehicle.location.location[0]
        y0 = vehicle.location.location[1]

        #For every candidate task...
        for index, task in enumerate(task_vector):
```

```
#Coordinates of candidate task
x1 = task.location[0]
y1 = task.location[1]

#Cacluate the path to the task
FNULL = open(os.devnull, 'w')    #This prevents a terminal
↪  window from popping up each time Tripath is called
subprocess.call(
    '/home/chris/Research/PISR_Sim_NGpp/Tripath_custom/bin/./setut
    ↪  {} {} {} {} {} {}'.format(
    x0, y0, x1, y1, vehicle.pathing.map,
    ↪  vehicle.pathing.nfz),

    ↪  cwd='/home/chris/Research/PISR_Sim_NGpp/Tripath_custom/bin/',
    stdout=FNULL, shell=True)
path_data = np.genfromtxt(
    '/home/chris/Research/PISR_Sim_NGpp/Tripath_custom/bin/path.txt',
    delimiter = ",")
xPath = path_data[:,0]
yPath = path_data[:,1]

#Calculate the length of the path
dist = 0
for ind, entry in enumerate(xPath[0:-1]):
```

```
                dist = dist +
                ↪  math.sqrt(math.pow(xPath[ind+1]-xPath[ind],
                ↪  2)+math.pow(yPath[ind+1]-yPath[ind], 2))

            #Convert distance to travel time
            time = dist/vehicle.speed

            #Save the travel time to each task
            times_and_headings.append([task.ID, time, 0])

        times_and_headings = np.array(times_and_headings)

        print '      Shortest measured times to each task:'
        print ''
        print times_and_headings[:,0:2]
        print ''

        return times_and_headings

    def calc_nfz_impact_rating(self, pathing_data, task_vector):
        #First, calculate the average distance between all tasks
        ↪  without the NFZ (Euclidean distances)
        cxyVector = []
        for task in task_vector:
            x = task.location[0]
            y = task.location[1]
```

```
        cxy = x+y*1j
        cxyVector.append(cxy)
    cxyVector = np.array([cxyVector], dtype=complex)
    distanceMatrix = abs(cxyVector.T-cxyVector)
    D_without_nfz = np.sum(
            distanceMatrix)/((distanceMatrix.shape[0]**2)
            -distanceMatrix.shape[0])   #don't divide by diaganol
             ↪  entries, which are zero

    #Second, calculate the average distance between all tasks
     ↪  taking into account the NFZ (Use Tripath)
    D_array = []
    for start_task in task_vector:  #for every task...
        for end_task in task_vector:  #to every task...
            #Coordinates of starting task
            x0 = start_task.location[0]
            y0 = start_task.location[1]

            #Coordinate of destination task
            x1 = end_task.location[0]
            y1 = end_task.location[1]

            #Caclulate distance between start and end task
            FNULL = open(os.devnull, 'w')   #This prevents a
             ↪  terminal window from popping up each time Tripath
             ↪  is called
```

```
            subprocess.call(
                    '../../Tripath_custom/bin/./setut {} {} {} {}
                    ↪  {} {}'.format(
                    x0, y0, x1, y1, pathing_data[1],
                    ↪  pathing_data[2]),
                    cwd='../../Tripath_custom/bin/', stdout=FNULL,
                    ↪  shell=True)
            path_data = np.genfromtxt(
                    '/home/chris/Research/PISR_Sim_NGpp \
                    /Tripath_custom/bin/path.txt',
                    delimiter = ",")
            xPath = path_data[:,0]
            yPath = path_data[:,1]
            dist = 0
            for ind, entry in enumerate(xPath[0:-1]):
                dist = dist +
                ↪  math.sqrt(math.pow(xPath[ind+1]-xPath[ind],
                ↪  2)+math.pow(yPath[ind+1]-yPath[ind], 2))
            D_array.append(dist)
    D_array = np.array(D_array)
    D_with_nfz =
    ↪  np.sum(D_array)/(D_array.shape[0]-len(task_vector))
    ↪  #don't divide by the zero entries of task x to task x

    #calculate nfz impact rating and save
    self.nfz_impact = D_with_nfz/D_without_nfz
```

```
class PathingFactory:
    def get_pathing_module(self, pathing_data):
        if pathing_data[0] == 'Euclidean':
            return Euclidean_Pathing()
        elif pathing_data[0] == 'Dubins':
            return Dubins_Pathing()
        elif pathing_data[0] == 'Tripath':
            return Tripath_Pathing(pathing_data[1], pathing_data[2])
        else:
            raise NotImplementedError("Unknown pathing type.")
```

A.2.5 The Communication Class.

```
import numpy as np
from abc import ABCMeta, abstractmethod

class Communication(object):

    """A class for communication of PISR Vehicles."""

    __metaclass__ = ABCMeta

    @abstractmethod
    def talk(self):
```

```
        raise NotImplementedError("You must implement a talk method
        ↪  for every Communication type!")

class No_Communication(Communication):

    def __init__(self):
        self.type = 'None'

    def talk(self, decider, vehicle_vector):
        pass

class Completion_Communication(Communication):

    def __init__(self):
        self.type = 'Completion'

    def talk(self, decider, vehicle_vector):
        for vehicle in vehicle_vector:
            if vehicle.ID != decider.ID:
                #Update sister vehicle age tracker's to account for
                ↪  the task just serviced by this vehicle
                #(This is how old the task will now be when the
                ↪  sister vehicle makes its next decision)

                ↪  vehicle.database.age_tracker[int(decider.location.ID-1)]
                ↪  = vehicle.routing.arrival_time-decider.time
```

```
                print ''
                print '      Broadcasted completion of Task {} @ {}
                ↪  secs.'.format(decider.location.ID, decider.time)

class Destination_Communication(Communication):

    def __init__(self):
        self.type = 'Destination'

    def talk(self, decider, vehicle_vector):
        for vehicle in vehicle_vector:
            if vehicle.ID != decider.ID:
                #Update sister vehicle age tracker's to account for
                ↪  the task just serviced by this vehicle
                #(This is how old the task will now be when the
                ↪  sister vehicle makes its next decision)

                ↪  vehicle.database.age_tracker[int(decider.location.ID-1)]
                ↪  = vehicle.routing.arrival_time-decider.time
                #Let the sister vehicles know which task this vehicle
                ↪  has just selected and what time it will arrive
                vehicle.database.vehicle_tracker[decider._indexer, 0]
                ↪  = decider.routing.destination.ID
                vehicle.database.vehicle_tracker[decider._indexer, 1]
                ↪  = decider.routing.arrival_time
```

```
            print ''
            print '      Broadcasted completion of Task {} @ {}
            ↪  secs.'.format(decider.location.ID, decider.time)
            print '      Broadcasted destination as Task {} @ {}
            ↪  secs.'.format(decider.routing.destination.ID,
            ↪  decider.routing.arrival_time)


class CommunicationFactory:
    def get_comm_module(self, comm_data):
        if comm_data[0] == 'None':
            return No_Communication()
        elif comm_data[0] == 'Completion':
            return Completion_Communication()
        elif comm_data[0] == 'Destination':
            return Destination_Communication()
        else:
            raise NotImplementedError("Unknown communication type.")
```

A.2.6 The Database Class.

```
import numpy as np


class Database:

    """A class for different information needs of PISR Vehicles."""

    def __init__(self, database_items, vehicle_vector, task_vector):
```

```
for entry in database_items:
    if entry == 'Age_Tracker':
        # age_vector = []
        # for task in task_vector:
        #     age_vector.append(task.age)
        # age_vector = np.array(age_vector)
        self.age_tracker = np.zeros(len(task_vector))
        for task in task_vector:
            self.age_tracker[task.ID-1] = task.age
    elif entry == 'Vehicle_Tracker':
        self.vehicle_tracker = np.zeros((len(vehicle_vector),
        ↪ 2))  # format: [destination_task, arrival_time]
```

Bibliography

1. Yann Chevaleyre. Theoretical analysis of the multi-agent patrolling problem. *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology, 2004. (IAT 2004).*, pages 302–308, 2004.

2. Ethan Stump and Nathan Michael. Multi-robot persistent surveillance planning as a vehicle routing problem. *IEEE International Conference on Automation Science and Engineering*, pages 569–575, 2011.

3. Soroush Alamdari, Elaheh Fata, and Stephen L. Smith. Persistent Monitoring in Discrete Environments: Minimizing the Maximum Weighted Latency Between Observations. pages 1–25, 2012.

4. Aydano Machado, Geber Ramalho, Jean Daniel Zucker, and Alexis Drogoul. Multi-agent patrolling: An empirical analysis of alternative architectures. In *Multi-Agent-Based Simulation II*, volume 2581, pages 155–170. 2003.

5. Nikhil Nigam and Ilan Kroo. Persistent Surveillance Using Multiple Unmanned Air Vehicles. *Aerospace Conference, 2008 IEEE*, pages 1–14, 2008.

6. H. R. Everett. *Unmanned Systems of World Wars I and II.* The MIT Press, 2015.

7. John Sifton. A Brief History of Drones. *The Nation*, feb 2012.

8. Department of Defense. Unmanned Systems Integrated Roadmap. Technical report, Department of Defense, 2013.

9. Steven Rasmussen, Krishnamoorthy Kalyanam, Satyanarayana Manyam, David Casbeer, and Christopher Olsen. Practical Considerations for Implementing an

Autonomous , Persistent , Intelligence , Surveillance , and Reconnaissance System. In *IEEE Conference on Control Technology and Applications*. IEEE, 2017.

10. Department of Defense. Autonomy Research Pilot Initiative (ARPI) Invitation for Proposals. 2012.

11. I. Michael Ross and Mark Karpenko. A review of pseudospectral optimal control: From theory to flight. *Annual Reviews in Control*, 36(2):182–197, 2012.

12. Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality af A*. *Journal of the ACM*, 32(3):505–536, 1985.

13. Katsuhiko Ogata. *Modern Control Engineering*. Prentice Hall, fifth edition, 2009.

14. G. B. Dantzig and J. H. Ramser. The Truck Dispatching Problem. *Management Science*, 6(1):80–91, 1959.

15. Tonci Caric and Hrvoje Gold. *Vehicle Routing Problem*. In-Teh, 2008.

16. O Bräysy and M Gendreau. Vehicle routing problem with time windows, Part I: Route construction and local search algorithms. *Transportation Science*, 39(1):104–118, 2005.

17. Fabio Pasqualetti, Joseph W. Durham, and Francesco Bullo. Cooperative patrolling via weighted tours: Performance analysis and distributed algorithms. *IEEE Transactions on Robotics*, 2012.

18. Mo Li, Weifang Cheng, and Kebin Liu. Sweep Coverage with Mobile Sensors. *IEEE Transactions on Mobile Computing*, 10(11):1534–1545, 2011.

19. Stephen L. Smith and Daniela Rus. Multi-robot monitoring in dynamic environments with guaranteed currency of observations. *Proceedings of the IEEE Conference on Decision and Control*, pages 514–521, 2010.

20. S. Lin. Computer solutions of the traveling salesman problem. *Bell System Computer Journal*, 44:2245–2269, 1965.

21. S. Lin and B.W. Kerninghan. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 21:498–516, 1973.

22. Majd Latah. Solving Multiple TSP Problem by K-Means and Crossover based Modified ACO Algorithm. *International Journal of Engineering Research & Technology (IJERT)*, 5(02):430–434, 2016.

23. San Nah Sze and Wei King Tiong. A Comparison between Heuristic and Meta-Heuristic Methods for Solving the Multiple Traveling Salesman Problem. *International Journal of Mechanical, Industrial Science and Engineering*, 1(1):27–30, 2007.

24. K S R College Technology Tiruchengode. Optimization of Non-Linear Multiple Traveling Salesman Problem Using K-Means Clustering , Shrink Wrap Algorithm and Meta-Heuristics. *International Journal of Nonlinear Science*, 9(4):171–177, 2010.

25. Stuart P. Lloyd. Least Squares Quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

26. D Aloise, A Deshpande, P Hansen, and P Popat. NP-hardness of Euclidean sum-of-squares clustering. *Machine Learning*, 75:245–249, 2009.

27. M.E. Celebi, H.A. Kingravi, and P.A. Vela. A comparative study of efficient intialization methods for the k-means clustering algorithm. *Expert Systems with Applications*, 40(1):200–210, 2013.

28. D J C MacKay. An Example Inference Task: Clustering. In *Information Theory, Inference and Learning Algorithms*, chapter 20, pages 284–292. Cambridge University Press, 2003.

29. David Arthur and Sergei Vassilvitskii. K-Means++: the Advantages of Careful Seeding. *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, 8:1027–1025, 2007.

30. Nikhil Nigam, Stefan Bieniawski, Ilan Kroo, and John Vian. Control of multiple UAVs for persistent surveillance: Algorithm and flight test results. *IEEE Transactions on Control Systems Technology*, 20(5):1236–1251, 2012.

31. Sanjeev Arora. Polynomial Time Approximation Schemes for Euclidean TSP and other Geometric Problems. In *Proceedings of 37th Conference on Foundations of Computer Science*, 1996.

32. William Cook. Solving TSPs: World TSP, 2013.

33. G Laporte. The traveling salesman problem: an overview of exact and approximate algorithms. *European Journal of Operational Research*, 59:231–47–231–47, 1992.

34. G Carpaneto and P Toth. Some new branching and bounding criteria for the asymmetric travelling salesman problem. *Management Science*, 26:736–743, 1980.

35. E Balas and N Christofides. A restricted Lagrangean approach to the traveling salesman problem. *Mathematical Programming*, 21:19–46, 1981.

36. Manfred Padberg and Giovanni Rinaldi. A Branch-and-Cut Algorithm for the Resolution of Large-Scale Symmetric Traveling Salesman Problems Published by : Stable URL : http://www.jstor.org/stable/20. *Society for Industrial and Applied Mathematics (SIAM) Review*, 33(1):60–100, 1991.

37. William Cook. Concorde TSP Solver, 2003.

38. D.J. Rosenkrantz, R.E. Stearns, and P.M. Lewis, II. An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing*, 6:563–581, 1977.

39. M. Dorigo and L.M. Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, 1997.

40. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.

41. Martin WP Savelsbergh. Local search in routing problems with time windows. *Annals of Operations research*, 4(1):285–305, 1985.

42. Nicos Christofides, A. Mingozzi, and P. Toth. State-space relaxation procedures for the computation of bounds to routing problems. *Networks*, 11(2):145–164, 1981.

43. Edward K Baker. An Exact Algorithm for the Time-Constrained Traveling Salesman Problem. *Operations Research*, 31(5):938–945, 1983.

44. Yvan Dumas, Jaques Desrosiers, Eric Gelinas, and Marius Solomon. An optimal algorithm for the traveling salesman problem with time windows. *Operations Research*, 43(2):367–371, 1995.

45. John N. Tsitsiklis. Special cases of traveling salesman and repairman problems with time windows. *Networks*, 22(3):263–282, 1992.

46. M.M. Solomon. On the Worst-Case Performance of Some Heuristics for the Vehicle Routing and Scheduling Problem with Time Window Constraints. *Networks*, 16:161–174, 1986.

47. M.M. Solomon. Algorithms for the Vehicle Routing and Scheduling Problems with Time Window Constraints. *Operations Research*, 35:254265, 1987.

48. W. Dullaert. A Sequential Insertion Heuristic for the Vehicle Routing Problem with Time Windows with Relatively Few Customers Per Route. 2000.

49. J.Y. Potvin and J.M. Rousseau. A Parallel Route Building Algorithm for the Vehicle Routing and Scheduling Problem with Time Windows. *European Journal of Operational Research*, (66):331340, 1993.

50. P. Prosser and P. Shaw. Study of Greedy Search with Multiple Improvement Heuristics for Vehicle Routing Problems. 1996.

51. F. Glover. Multilevel Tabu Search and Embedded Search Neighborhoods for the Traveling Salesman Problem. 1991.

52. F. Glover. New Ejection Chain and Alternating Path Methods for Traveling Salesman Problems. In O. Balci, R. Sharda, and S. Zenios, editors, *Computer Science and Operations Research: New Developments in Their Interfaces*, page 449509. Pergamon Press, Oxford., 1992.

53. I.H. Osman. Metastrategy Simulated Annealing and Tabu Search Algorithms for the Vehicle Routing Problems. *Annals of Operations Research*, 41:421452, 1993.

54. Y.A. Koskosidis, W.B. Powell, and M.M. Solomon. An Optimization-Based Heuristic for Vehicle Routing and Scheduling with Soft Time Window Constraints. *Transportation Science*, 26:69–85, 1992.

55. P. Shaw. A New Local Search Algorithm Providing High Quality Solutions to Vehicle Routing Problems. 1997.

56. Avrim Blum, Prasad Chalasani, Don Coppersmith, Bill Pulleyblank, Prabhakar Raghavan, and Madhu Sudan. The Minimum Latency Problem. In *STOC '94 Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 163–171, Montreal, Quebec, Canada, 1994. ACM.

57. Bang Y. Wu, Zheng N. Huang, and Fu J. Zhan. Exact algorithms for the minimum latency problem. *Information Processing Letters*, 92(6):303–309, 2004.

58. Michel Goemans and Jon M Kleinberg. An Improved Approximation Ratio for the Minimum Latency Problem. *Mathematical Programming*, 82(1):111–124, 1998.

59. N. Garg. A 3-Approximation for the Minimum Tree Spanning k Vertices. In *Proceedings of the 37th IEEE Symposium on Foundations of Computer Science*, pages 302–309, 1996.

60. Aaron Archer and David P Williamson. Faster approximation algorithms for the minimum latency problem. *Proc. of the 14th annual ACM-SIAM symposium on Discrete algorithms*, (i):88–96, 2003.

61. Marcos Melo Silva, Anand Subramanian, Thibaut Vidal, and Luiz Satoru Ochi. A simple and effective metaheuristic for the Minimum Latency Problem. *European Journal of Operational Research*, 221(3):513–520, 2012.

62. Jing Guan, Jiafu Tang, and Yang Yu. An Ant Colony Optimization for Weighted Traveling Salesman Problem and Analysis. In *Chinese Control and Decision Conference (CCDC)*, pages 3852–3857. IEEE, 2011.

63. Andrei M. Shkel and Vladimir Lumelsky. Classification of the Dubins set. *Robotics and Autonomous Systems*, 34(4):179–202, 2001.

64. Xavier Goaoc, Hyo-sil Kim, and Sylvain Lazard. Bounded-Curvature Shortest Paths through a Sequence of Points Using Convex Optimization. *Society for Industrial and Applied Mathematics (SIAM) Review*, 42(2):662–684, 2010.

65. Ketan Savla, Emilio Frazzoli, and Francesco Bullo. On the point-to-point and traveling salesperson problems for Dubins' vehicle. In *American Control Conference*, 2005.

66. J Le Ny and E Feron. An Approximation Algorithm for the Curvature-Constrained Traveling Salesman Problem. *Allerton Conf. on Communications, Control and Computing*, 2005.

67. Jerome Le Ny, Emilio Frazzoli, and Eric Feron. The curvature-constrained traveling salesman problem for high point densities. *Proceedings of the IEEE Conference on Decision and Control*, pages 5985–5990, 2007.

68. X Ma and D A Castanon. Receding horizon planning for Dubins traveling salesman problems. *Proceedings of the IEEE Conference on Decision and Control*, pages 5453–5458, 2006.

69. Jerome Le Ny, Eric Feron, and Emilio Frazzoli. On the Dubins Traveling Salesman Problem. *IEEE Transactions on Automatic Control*, 57(1):265–270, 2012.

70. R J Kenefic. Finding Good Dubins Tours for UAVs Using Particle Swarm Optimization. *Journal of Aerospace Computing Information and Communication*, 5(2):47–56, 2008.

71. Xin Yu and John Y. Hung. A genetic algorithm for the Dubins Traveling Salesman Problem. *2012 IEEE International Symposium on Industrial Electronics*, pages 1256–1261, 2012.

72. Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, New Jersey, third edition, 2010.

73. Sui Ruan, Candra Meirina, Feili Yu, Krishna R. Pattipati, and Robert L. Popp. Patrolling in A Stochastic Environment. In *10th International Command and Control Research and Technology Symposium*, 2005.

74. D. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, Belmont, MA, 3rd edition, 2005.

75. V Huynh, J Enright, and E Frazzoli. Persistent patrol in stochastic environments with limited sensors. *Proc. AIAA Conference on Guidance, Navigation, and Control*, (August):1–14, 2010.

76. Eric Jones, Travis Oliphant, Pearu Peterson, and Others. SciPy: Open source scientific tools for Python, 2001.

77. A. Walker. Dubins-Curves: an open implementation of shortest paths for the forward-only car, 2008.

78. A. Walker. PyDubins: code to generate and manipulate dubins curves, 2018.

79. Marcelo Kallmann. Graphsim Tripath Toolkit, 2010.

80. Marcelo Kallmann. Shortest Paths with Arbitrary Clearance from Navigation Meshes. *Proceedings of the Eurographics SIGGRAPH Symposium on Computer Animation SCA*, pages 159—-168, 2010.

81. M Kallmann. Dynamic and Robust Local Clearance Triangulations. *ACM Transactions on Graphics*, 33(5):17, 2014.

Vita

Major Christopher Olsen is a native Texan, born and raised in the Dallas-Ft. Worth metroplex. After graduating from Colleyville Heritage High School in 2003, he went on to attend Texas A&M University in College Station, Texas where he joined the Corps of Cadets and Air Force ROTC. He graduated in 2008 with an undergraduate in Mechanical Engineering and was commissioned into the Air Force as a Developmental Engineer.

His first assignment as a 2d Lt was to Wright-Patterson AFB, Ohio at the National Air and Space Intelligence Center (NASIC). At NASIC, he worked in the Space Analysis Squadron conducting technical analysis on space systems. While at NASIC, he earned his Space Operations badge through the National Security Space Institute (NSSI) in Colorado Springs, Colorado.

In 2011, he received orders to the National Security Agency (NSA) at Ft. Meade, Maryland where he was assigned to the 34th Intelligence Squadron (34 IS). He spent one year working crypto modernization in the Nuclear Communications branch of NSA before being deployed to Camp Leatherneck, Afghanistan in 2012. While deployed, he was given the role of a civil engineer. His primary duty was to aid the Afghan National Police (ANP) and Afghan National Army (ANA) in the construction of bases and checkpoints to provide regional security, with the ultimate goal of relieving U.S. Marines of security responsibilities. In October 2012, he redeployed to Ft. Meade, where he resumed work on Nuclear Communications and served as both a Branch Chief and Flight Commander.

In 2014, he was re-assigned to the 707th ISR Group (707 ISRG) as the commander's Plans and Programs officer. He helped create, document, and implement the commander's objective of reinvigorating the NSA-Air Force relationship.

While assigned to NSA, he also worked toward a M.S. in Systems Engineering from the Air Force Institute of Technology (AFIT), which he completed in December 2014. Shortly after, his application to AFIT as an in-residence Ph.D. student was accepted and, in September 2015, he began studies toward a Ph.D. in Aeronautical Engineering. His specialty is in Optimization and Controls.

Upon graduating from AFIT with his Ph.D., Maj Olsen is headed to the Air Force Research Laboratory, Aerospace Systems Directorate (AFRL/RQ) at Wright-Patterson AFB.

Maj Olsen has a wife, a curious son, a very energetic Australian Shepperd, and two standard house cats.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
14–08–2018	Doctorate Dissertation	Sept 2015 — Sept 2018

4. TITLE AND SUBTITLE

A HEURISTIC METHOD FOR TASK SELECTION
IN PERSISTENT ISR MISSIONS
USING AUTONOMOUS UNMANNED AERIAL VEHICLES

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Christopher C. Olsen, Maj, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENY-DS-18-S-067

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory, Aerospace Systems Directorate (AFRL/RQQC)
Attn: Amy Burns
2210 8th Street
WPAFB, OH 45433-7542
DSN 674-6542, COMM 937-904-6542
Email: amy.burns.3@us.af.mil

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

Additional sponsorship provided by Air Force Office of Scientific Research (AFOSR), 875 N. Randolph, Ste.325, Arlington, VA 22203, Comm: 703-696-7797, Email: info@us.af.mil

14. ABSTRACT

The Persistent Intelligence, Surveillance, and Reconnaissance (PISR) problem seeks to provide collection and delivery of data from prioritized ISR tasks using an autonomous Unmanned Aerial Vehicle (UAV). In this research, we investigate a method for selecting tasks called the Maximal Distance Discounted and Weighted Revisit Period (MD^2WRP) utility function. We develop a two-step optimization method for the MD^2WRP parameters for both single and multi-vehicle scenarios. We also compare the performance of MD^2WRP to other common methods for PISR task selection. We find that the optimized MD^2WRP function is competitive with these other methods. We also test MD^2WRP under simulated operational constraints. For each constraint, we demonstrate how MD^2WRP needs to be modified to compensate. Finally, we make practical suggestions about implementing MD^2WRP, outline areas for future study, and offer recommendations about the conduct of PISR missions in general.

15. SUBJECT TERMS

Autonomy, Persistent Monitoring, Patrolling, UAVs, Optimization

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Donald L. Kunz, AFIT/ENY
U	U	U	U	256	19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636, x4548; donald.kunz@afit.edu