TaxiWorld: Developing and Evaluating Solution Methods for Multi-Agent Planning Domains

by

Christopher White

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved June 2011 by the
Graduate Supervisory Committee:

Subbarao Kambhampati, Chair
Sandeep Gupta
Georgios Varsamopoulos

ARIZONA STATE UNIVERSITY

August 2011

ABSTRACT

TaxiWorld is a Matlab simulation of a city with a fleet of taxis which operate within it, with the goal of transporting passengers to their destinations. The size of the city, as well as the number of available taxis and the frequency and general locations of fare appearances can all be set on a scenario-by-scenario basis. The taxis must attempt to service the fares as quickly as possible, by picking each one up and carrying it to its drop-off location. The TaxiWorld scenario is formally modeled using both Decentralized Partially-Observable Markov Decision Processes (Dec-POMDPs) and Multi-agent Markov Decision Processes (MMDPs). The purpose of developing formal models is to learn how to build and use formal Markov models, such as can be given to planners to solve for optimal policies in problem domains. However, finding optimal solutions for Dec-POMDPs is NEXP-Complete, so an empirical algorithm was also developed as an improvement to the method already in use on the simulator, and the methods were compared in identical scenarios to determine which is more effective. The empirical method is of course not optimal - rather, it attempts to simply account for some of the most important factors to achieve an acceptable level of effectiveness while still retaining a reasonable level of computational complexity for online solving.

ACKNOWLEDGMENTS

I would first like to offer my gratitude to my director, Subbarao Kambhampati, for his patience and insight as he guided me in my research, and for providing me with necessary resources and information to which I would not otherwise have had access. It is due to his great willingness to provide assistance that I have been able to complete this thesis at all.

I would also like to thank J. Benton for his willingness to discuss my project and to share his own experience with me; the discussions I had with him were always valuable, and I am certain I would not have the level of understanding I do without his generosity.

Finally, Joshua Ferguson, whose work with TaxiWorld predates my own, was an extremely valuable resource as I learned how to work with the simulator. His knowledge of the many quirks and apparent chaos within the MATLAB code prevented me from being unnecessarily stonewalled on numerous occasions, and it is fair to say I might still be trying to tease apart its ineffable mysteries were it not for his help.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

TaxiWorld is a high-level simulation of a city, in which passengers need transportation, and a small number of taxis which patrol the city and are responsible for meeting the passengers' needs. The number of taxis in the fleet is variable, and can be configured as is desired for a given scenario. Requests for transportation are modeled as 'fares', which appear semi-randomly within the city, and are then available for a taxi to pick up and deliver to the fare's drop-off location. In addition, each fare has a time limit; if the fare is not collected by some taxi within the allotted amount of time, it will disappear, resulting in lost revenue for the taxi company. It is therefore of paramount importance that the taxis be dispatched efficiently, to maximize the number of fares they are able to deliver and increase the amount of profit that the company can collect. The simulation also includes a gas station, which is located outside of the city proper. Taxis must on occasion go to the gas station to refuel, so that they will be able to continue to provide service for the fares within the city.

Approach

My research was principally directed towards understanding Decentralized Partially-Observable Markov Decision Processes (Dec-POMDPs), and finding a way to encode the TaxiWorld domain using this model. Since the individual taxis can be considered to be separate agents, each of which must make autonomous decisions based on local information without explicit communication with the other taxis, this seemed like a good domain to use for this research. In learning about the structure of the simulator itself, it became clear that the behavior of the taxis in the simulator should actually be modeled using Multi-Agent Markov Decision Processes. For this reason, I developed a formalization of the TaxiWorld scenario using both models.

The second goal of my research was to develop a taxi-control algorithm for use within the Matlab simulation itself, to improve on the current algorithm being used. The Matlab simulation was actually developed prior to my own involvement in the project, so my principal problem was to first learn how to develop within the Matlab simulation environment; doing this revealed that the control algorithm in place fails to take into account a number of important factors in determining the behaviors of the fleet of taxis. To rectify this, I proceeded by first identifying the shortcomings of the original control strategy, and then trying to account for them in my improved version. Since I am more familiar with coding in Java than in Matlab, I found it helped to develop the taxi-assignment code externally in Java and call it from within the Matlab simulation; the results from that call are then used to generate orders for the fleet within Matlab.

In doing this research, I was able to learn the types of problems that are representable within different families of MDP models, and to use the Dec-POMDP and MMDP frameworks to model a fairly complex scenario. Through the development of my empirical model, I demonstrated that although finding an optimal policy for a problem may be intractable, it may be possible to leverage domain-specific properties to generate a solution strategy that achieves an acceptable level of success at significantly less computational cost. Because the cost-to-benefit ratio of developing and using a fully optimal solution would be extremely large, in many cases it might be far more practical to craft a reasonably efficient domain-specific solution than to formalize the problem and use a full-blown general solver to generate an optimal policy.

Related Work

The problem of coordinating multiple autonomous agents in accomplishing a set of goals is a non-trivial planning problem, especially if events beyond the agents' control can cause significant changes to the environment, thus invalidating or rendering obsolete previously generated plans. If the current state of the world is also only partially observable, the problem domain can become even more difficult. Planning algorithms that can effectively address such problem domains are of interest to the Artificial Intelligence community due to the inherent complexity possessed by problems such as these, and because of the transferability of a solution to one such problem to other similar problems.

Markov Decision Processes, in their many forms, have been considered in numerous papers and publications, due to their versatility and applicability in many problem domains. Because of these standardized formal models, any multi-agent planning domain may be addressed in a similar way to TaxiWorld, although the exogenous and stochastic goal arrivals are important factors that distinguish TaxiWorld from many other planning domains. Hubbe et al. [3] address the problem of stochastic goal arrival in a much smaller domain by sampling the distribution of possible future goal arrivals and choosing the action which will optimize the expected cost over the sampled futures.

A taxi-planning domain very similar to TaxiWorld is considered by Xu and Huang [12], but is notably different in that they allow taxis to carry multiple passengers at once (up to four). They attempt to solve their domain by using a decentralized algorithm in which taxis generate a set of possible plans and then communicate with other nearby taxis to determine which plan of action is best.

Ryan et al. [9] consider a similar domain in which unmanned aerial vehicles (UAVs) must coordinate to achieve a set of goals. The UAVs in this scenario coordinate by “bidding” on goals with the cost it would require them to achieve the goal; the UAV which can offer the lowest-cost bid for a goal takes on the assignment.

CHAPTER 2

SIMULATOR DESCRIPTION

TaxiWorld consists of a class framework which contains all of the components that comprise the city and its inhabitants. The simulation is carried out in "steps", which can be executed as quickly as the computer's processor can evaluate them. Each model component that appears within the simulation has its own "step" function, which is called by the simulator on each simulation cycle, and determines the component's behavior for that cycle. The objective of the simulator is to provide a test-bed for devising effective control strategies for servicing fares using the available taxis.

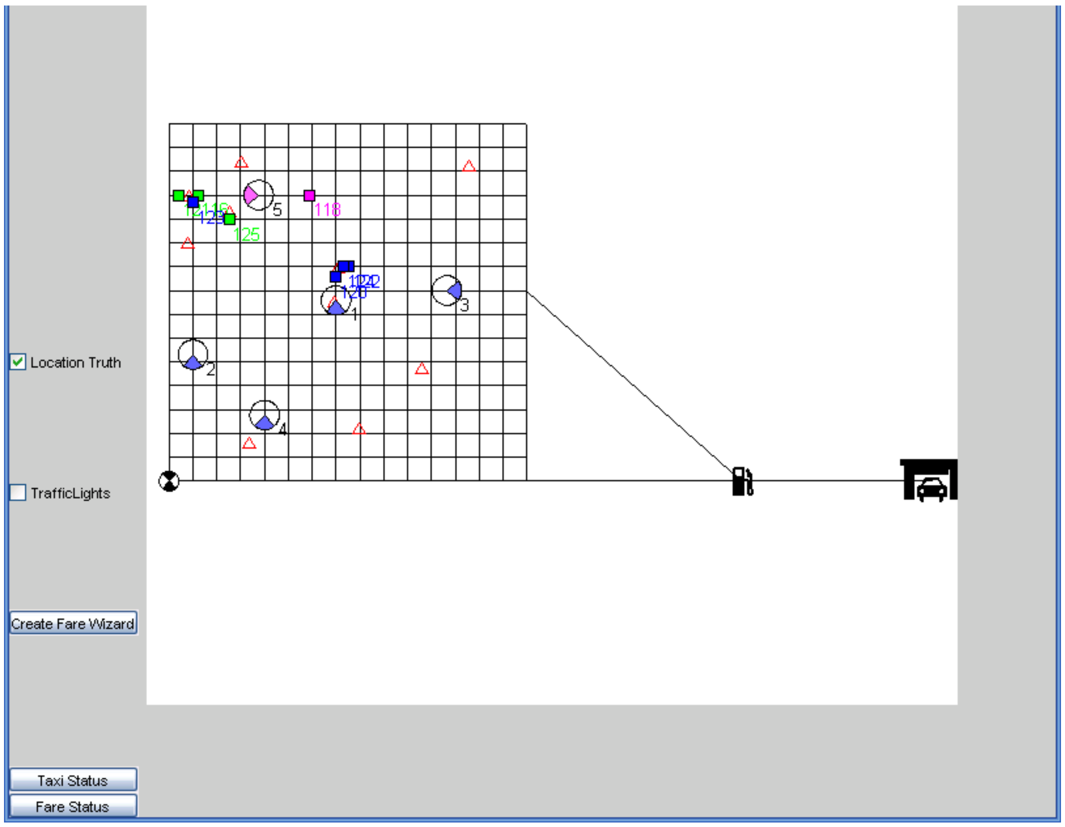


Figure 1: Example of the TaxiWorld Simulation

Simulation Components

Figure 1 shows what the simulator might look like while it is running. From this image it is possible to see the layout of the city, as well as the locations and positions of every fare, taxi, and point of interest in the scenario at any given time, as well as the locations of the gas station and the garage. Taxis are shown in the image as circles, containing a pie-slice-shaped, colored wedge, the point of which aims in the direction of the taxi’s motion. If the taxi is roaming or inactive, the wedge is shaded gray; if it is en route to pick up a fare, the wedge becomes blue; after retrieving the fare and before dropping it off, the wedge is magenta; when the taxi is on its way to refuel, the wedge becomes red. Fares are represented in the simulator as small squares. They have a similar coloring scheme to the taxis; when a fare first appears, it is green. Once a taxi is assigned to it, the fare becomes blue, and points that represent fare drop-off locations are shown in magenta. Points of interest are shown as red triangles on the map, and do not change throughout the course of the simulation. The gas station and the garage are represented respectively as a gas pump and a car inside a structure.

Each scenario is specified in its own file, which determines the size of the city by specifying the number of rows and columns (vertical and horizontal roads) it has. It also sets the locations and properties of all of the points of interest, as well as the size of the fleet of taxis that is available to service the city. The locations of any gas stations are also determined here, as well as any additional roads (called bridges) to connect the gas station to the city proper, since the gas station is located outside of the city proper.

Points Of Interest (POIs) are responsible for generating fares which can then be serviced by the taxi fleet. Each POI has a location within the city, as well as a radius. Together, the location and radius determine the portion of the city in which fares can be generated by the POI. A POI also has a frequency, which is the probability that on any given simulation cycle it will generate a new fare. The number of POIs within the city is scenario-specific, and the attributes of each POI can be configured individually if desired.

Fares are the goals for the agents in TaxiWorld. They are represented by a unique id, and have a pickup location, a drop-off location, and a time-to-live. The pickup location is the grid coordinate where the fare initially appears when spawned by a POI, and the drop-off location is the grid coordinate the taxi must take the fare to in order to complete the delivery and receive payment. The time-to-live is set when the fare is created (to a default value of 1300 simulation cycles), and will steadily decrease as the simulation runs. When the time-to-live reaches zero, the fare will expire, and it will no longer be possible for any Taxi to service it.

FareList is simply an organizational entity to make all of the information about fares in the city accessible from a single location within the program. The Dispatcher uses the FareList to keep track of the locations and life-spans of all of the currently-available fares, as well as to obtain the information it needs to make intelligent decisions regarding taxi deployment.

A TaxiState is a representation of all the information pertaining to a specific taxi. It contains the taxi's unique id, its location, its current fuel level, its current plan (the next fare it intends to collect, if there is one), the number of passengers currently in the vehicle, and the state of the taxi (a taxi can be active, inactive, refueling, or dead). The TaxiState can receive instructions to carry out, such as orders to pick up a fare or to go refuel.

The Dispatcher is one of the most important components of the simulator. It keeps track of all the fare locations and life-spans, and can send orders to the taxis based on the current conditions in the city and any changes that occur. It also has access to the FareList, which allows it to track changes to fare conditions and coordinate the taxi fleet's efforts in collecting them. The control algorithm I implemented is run from within the dispatcher’s step event, since it is the only component that not only has access to necessary information, but the ability to send out orders to all of the taxis as well.

CHAPTER 3

MDP MODEL BACKGROUND

Markovian models are useful for modeling problem domains in a discrete form such that it can be given to a solver, which can use the formal domain description to calculate an optimal policy for use in the domain. There are many different types of Markovian models that can be used, depending on the scenario under consideration; this thesis is primarily concerned with scenarios with multiple decision-making agents, and partial or full observability (this determines whether or not the agents have full knowledge of the current state). Several models are discussed here in order to demonstrate the differences between them, and to provide necessary background for understanding the choice of models for use with TaxiWorld. Figure 2 shows the relationships between the models under consideration.

MDPs are useful for situations where there is a single agent in a completely observable world. An MDP model can be used to generate a plan of action for a domain by considering all possible state configurations and solving for the best action to perform in each one. Such a mapping from valid state configurations to actions is called a “policy”, and if there is no other policy that does a better job of maximizing the agent’s goals, it is called an “optimal policy.” The agent can choose what action to do by looking at the current state and simply taking the action that the policy specifies. The state configuration serves as the “Markovian signal” – that is to say, it allows the agent, given a policy, to discern what action to do next without any knowledge of previous actions.

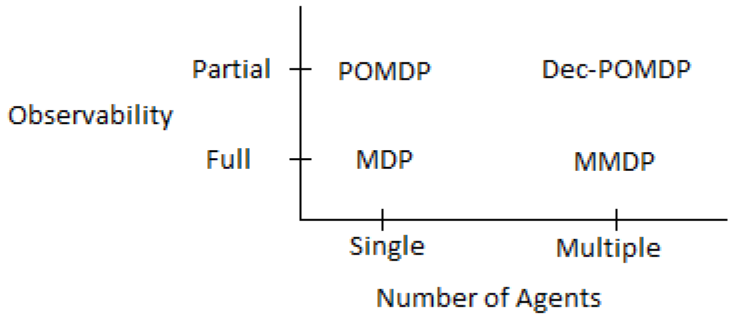


Figure 2: Families of MDP Models

Multi-agent MDPs (MMDPs) differ from MDPs in that there are multiple agents to perform actions, which means that a policy needs to specify an action for every agent (a "joint action") for each state. It is possible to reduce an MMDP to an MDP with a single agent by treating every possible combination of agent actions as singular joint actions, performed by one agent. This abstraction makes and MMDP practically identical to an MDP, except that it has a number of agent actions that is exponential in the number of agents from the original MMDP formulation.

Partially-Observable MDPs (POMDPs) can be used to solve for optimal policies when the world is not completely observable by maintaining a belief state about the condition of the world. This belief state effectively summarizes the action and observation histories, and serves as a Markovian signal to allow the agent to act in an optimal way. A POMDP policy is then a mapping of belief states to actions (instead of a mapping of states to actions).

Dec-POMDPs have the additional complexity over POMDPs in that there are multiple agents, each of which has the ability to make decisions on its own. Thus agents are required to take into account not just the environment, but the actions of the other agents in determining their own next move. An important distinction between Dec-POMDPs and MDPs or POMDPs is that agents in Dec-POMDPs have no Markovian signal (Oliehoek, “Decentralized POMDPs” 5). Any individual agent can only account for its own actions and observations, whereas the transition and observation functions are specified in terms of joint actions. The partially-observable nature of the problem also means that it is impossible to use the single agent “puppeteer” reduction from before, because no agent will necessarily know everything that other agents know. Because there is no known way to account for the influence of other agents, a Dec-POMDP policy must necessarily map full-length action histories to actions (Oliehoek, “Decision Making for Cooperative Agents” 32).

Finite-horizon Dec-POMDPs are known to be NEXP-Complete (and infinite-horizon Dec-POMDPs, like POMDPs, are undecidable) (Bernstein 3), which makes fully optimal planning algorithms utterly intractable for use in an online setting. Optimal plans for such scenarios are therefore normally computed ahead of time (offline) and executed as the simulation runs. General algorithms for solving Dec-POMDPs generally exhibit poor scalability, but special classes of Dec-POMDPs sometimes have a lower complexity that a more specialized algorithm can take advantage of to obtain more efficient solutions (Seuken and Zilberstein 40).

MMDP Model

An MMDP can be modeled as a tuple $<S, \boldsymbol{A}, T, R, h>$, where

S is the finite set of possible environmental states.

A is the set of possible joint actions.

T is the transition function,

R is the reward function, and

h is the horizon for the problem. This is, roughly speaking, the number of time-steps a solver has to find a solution, or the life-span of a specific instance of a problem.

Before developing a formal TaxiWorld representation, a few definitions are needed:

D = the set of taxis, {t_1, … t_n}.

L = the set of all possible map positions, where each *l* in *L* is represented as (x, y), where x and y are the horizontal and vertical coordinates of the position within the city, and x and y are both nonnegative real numbers.

n = the number of taxis in the fleet.

p = the number of POIS in the scenario.

F = a set of fares with the maximum number of fares that can be in the city at a given time for the scenario. This can be calculated for a given scenario if the number of POIs is known. Since each POI can generate at most one fare each simulation cycle, and each fare has a lifespan of at most 1300 cycles, the maximum number of fares that can be on the map at once is ($p \cdot 1300$).

f = the set of possible values for the amount of fuel a taxi can have in its tank. In the Matlab simulation, fuel level is simply modeled as a real number in the range (0, 10), but in order to represent TaxiWorld as a discrete model, it is necessary to represent the possible values as a discrete set of Boolean values rather than as a real number.

If we assume that the taxis are able to communicate instantly with each other with no cost penalty, we can formalize TaxiWorld as an MMDP as follows: $S = \{D \times L \times f \times F \times L\}$. In each state, each taxi can be in any valid location within the city with any amount of fuel in the tank, and each fare can be also be in any valid location within the city.

It is clear from this that the number of possible states $|S|$ is difficult to even conceive: consider, for example, a very simple TaxiWorld scenario with only two taxis, 100 possible map locations (far less than in any standard scenario), and a single point of interest, which has only 10 map locations within its radius of influence. For simplicity, we will assume that a taxi has only 10 possible fuel levels, but we will leave the fare time-to-live at its default value of 1300 time-steps. Even for such a comparatively small instance of TaxiWorld, the state space is quite large: each taxi can be in any of the 100 possible locations, and at any fuel level. This gives us $100 \cdot 100 \cdot 10 \cdot 10$, or 10^6 states before we even look at the fare distribution. Now, how many possible fare arrangements can there be? When the first fare is created, it can be in any of the 10 map locations near to the POI, for 10 possibilities. Once two fares are on the map, there can be $10 \cdot 10$ possible fare arrangements (since multiple fares can exist in the same space). Similarly for three or more fares; x fares can be arranged in 10^x possible ways. Since each fare has a lifespan of 1300, and the POI could theoretically generate a fare every single step for 1300 steps, we must consider the possibility of up to 1300 simultaneously-existing fares. Thus, accounting for every possible number of fares in every possible arrangement, there are $10^{1300} + 10^{1299} + \ldots + 10^{1}$, or $1.11 \cdot 10^{1300}$ possible fare arrangements in this simplified scenario. Considering both fare arrangements with taxi arrangements gives us a final state space with a cardinality of $10^{6} \cdot 1.11 \cdot 10^{1300} = 1.11 \cdot 10^{1306}$. This means that an optimal MMDP policy would have to specify the best action for every one of these states.

$\boldsymbol{A} = \{A_1 \times A_2 \ldots \times A_n\}$, where A_i is the set of possible actions for taxi i: $\{a_N, a_S, a_E, a_W, a_P, a_D\}$ – each taxi can move in each of the cardinal directions, and can pick up or drop off a fare, provided the correct conditions are met.

R = +1 for each time any taxi drops off a fare (any taxi performs action a_D). If this scenario is reduced to an MDP (treated as if there is a single “puppeteer” agent), then there will be 2^n actions which represent a 'drop-off' action for at least one taxi, so the reward function will need to specify the appropriate reward for each.

h – For testing the control algorithms, I chose a horizon of 50,000 steps.

T – The transition function must be able to account for all possible state changes due to actions by the taxis. It must also account for the stochastic appearance of fares that can happen in any simulation cycle.

There are a couple of ways to attempt to define such a transition function. The first possible approach is to define $T: S \times \boldsymbol{A} \rightarrow S$, so that the function takes as input a current state and a joint action (a move for every taxi in that time-step), and returns the resulting state. However, the problem with this approach is that it creates an incredibly large branching factor which is time-consuming for a planner to evaluate: for each state, each taxi can take any of up to six actions. This means that for each state, there would need to be a transition defined to every state that could result in the next step due to fare appearances, and for every one of these there would need to be transitions defined for each of the possible combinations of movements of the entire taxi fleet. The number of possible next states is quite large. Looking again at our simplified TaxiWorld domain from before, we can see that between the two taxis, there are 36 possible joint actions they might take. The single POI may also generate a fare at any of its 10 in-range locations. It is also possible that a fare will expire in the next step, but since this occurs deterministically, it will not affect the branching factor of the transition function. This means that we can have up to $36 \cdot 10 = 360$ possible next states for any given current state.

One method that can slim down the branching factor is to break apart the taxi moves and evaluate them separately, instead of evaluating them all at once. In the actual simulator, each taxi is technically (at a very low level) updated sequentially, and it is possible to do a similar thing with the states in this model. With this approach, you can define $T: S \times A \rightarrow S$, so you are only mapping single taxi actions, which decreases the branching factor (the number of possible next states) by a significant factor. Each simulation cycle can be represented in the Dec-POMDP as a sequence of n evaluation steps, and the POIs can be evaluated to see if they generate fares after the last taxi is planned at each cycle. This approach lowers the computational complexity for the planner at each step and can help to streamline the solving process.

DEC-POMDP Model

If the taxi drivers are assumed to be independent decision-making entities who work to service the city's fares in a decentralized way without implicit communication or knowledge of each other’s actions, then the domain can be modeled as a Dec-POMDP, which is similar to an MMDP, but has the additional complications that the domain is not fully observable, and agents cannot perform a joint belief update.

A Dec-POMDP can be represented as a tuple, $<D, S, \boldsymbol{A}, T, R, \boldsymbol{O}, O, h, \boldsymbol{I}>$, where S, $\boldsymbol{A}$, T, R, and h carry the same meaning as they do for an MMDP, and the additional symbols carry the following meanings:

D is the set of agents $\{1, \ldots n\}$.

$\boldsymbol{O}$ is the finite set of joint observations.

O is the observation probability function.

I is the initial state distribution at stage $t = 0$, and I is in the power set of S.

TaxiWorld could thus be represented as a Dec-POMDP as follows:

$D = \{t_1, \ldots t_n\}$ – simply the set of taxis.

$S = \{D \times L \times f \times F \times L\}$. In each state, each taxi can be in any valid location within the city with any amount of fuel in the tank, and each fare can be also be in any valid location within the city. The state representation can actually be identical to the MMDP version.

$\boldsymbol{A} = \{A_1 \times A_2 \ldots \times A_n\}$, where A_i is the set of possible actions for taxi i: $\{a_N, a_S, a_E, a_W, a_P, a_D\}$ – each taxi can move in each of the cardinal directions, and can pick up or drop off a fare, provided the correct conditions are met.

R = +1 for each time any taxi drops off a fare (performs action a_D).

O is the set of all fare and taxi locations, as well as the taxi's own fuel level.

O is the observation probability function. TaxiWorld actually does not have noisy observations, so the state of the city is observed with probability 1.

h – For testing the control algorithms, I chose *h* = 50,000 steps.

I – The initial state in TaxiWorld is simply the city with all of its POIs, and the taxis, which all start in the garage.

T – The transition function for the Dec-POMDP can be handled in the same way as for the MMDP, as it is still handling the same interactions.

Although the TaxiWorld scenario can be considered in the light of assumptions that allow it to be modeled as either an MMDP or a Dec-POMDP, the actual simulator is constructed under the reasonable assumption that there is a central dispatcher who can communicate with the taxis and coordinate their efforts, meaning that the domain as it is simulated (and as it is treated by the default planner in place in the simulator) is actually most accurately modeled as an MMDP.

From this discussion, it should be clear that while these models are quite powerful in their ability to capture representations for a wide variety of scenarios, their complexity renders them unwieldy for use in online settings in all but the simplest of problem domains. Although improving on the default TaxiWorld taxi-control algorithm is a stated goal of this thesis, the simulator is intended to be run in near-real time; it would be impractical to attempt to implement a fully-optimal planner to do this, especially since TaxiWorld can support any city configuration desired; the MDP policy would have to be re-computed for every new instance of the problem. Instead, I have implemented an online algorithm which attempts to be a closer approximation to the optimal policy than the default simulator control mechanism.

CHAPTER 4

CONTROL ALGORITHMS

The simulator already has a control algorithm in place to provide the taxis with direction. However, control is implemented in a fairly naïve way, and is therefore quite sub-optimal. Before considering any improvements that might be made to the control strategy, it will be useful to take a look at how control is currently implemented in the simulator.

Default Control Algorithm

The default control algorithm matches taxis to fares in a very straightforward way. Any time a taxi becomes available (e.g. when it enters the city proper at the beginning of the simulation, or after it drops off a fare), it is immediately assigned to the closest available fare. If multiple taxis become available at once, they will be assigned to fares in numerical order according to the taxi id numbers. Once a taxi is assigned to a fare, it will remain assigned to that fare until it either picks up the fare, or the fare expires and is no longer available to be retrieved by the taxi fleet. The taxi will then be assigned to the next available fare, if one exists.

There are multiple shortcomings with the control scheme as it stands. The default assignment strategy has a tendency to assign taxis to fares haphazardly, thereby losing a lot of time to inefficiency. For example, the numerically-ordered assignment means that the choice of which taxi collects a fare is determined by taxi id number, rather than by which taxi is in a better position to serve that particular fare. In addition, the inflexibility of the planner means that once a taxi is assigned to a fare—however sub-optimally—it can never be reassigned to a different fare unless it is no longer able to retrieve the fare (this will happen if the fare disappears due to exceeding its time limit). In a dynamic city where the distribution of fares is subject to constant change, a myopic approach like this is in no way acceptable.

Another consideration that is ignored is fare lifespan. Taxis are assigned to fares based on location, with no concern for when fares might be expiring. While it might be useful to attempt to prioritize fares based on lifespan (to target near-expired fares first and then come back for longer-lived ones), a perhaps more cogent threat to efficiency is the possibility of assigning a taxi to a fare it doesn't have time to reach before expiration. Such an assignment would result in a perfect waste of time while the taxi proceeds towards the fare, and then is eventually forced to find a new one. Since the taxis have a set speed at which they travel, and the time-to-live of each fare as well as its position relative to the taxi is known, it should be possible to calculate whether or not it is possible for a given taxi to reach its assigned fare in time.

Improved Control Algorithm

Initially, there were a number of possibilities under consideration for developing the improved control scheme. One of the most immediately obvious improvements was to make the assignment from taxis to fares in a more intelligent way, so fixing that was a first order of business. Another important change to make was to fix the unchangeable nature of fare assignments. In a city with constantly-changing conditions, the fleet's orders need to be flexible, so that taxis won't be doomed to pursue a sub-optimal course of action simply because the control system can't issue a change of plans. Another possibility, which would mostly be useful in situations where there are a lot of available fares, would be fare culling. If a fare is determined to be too far away, or there are enough nearby fares, it might make sense to simply ignore the distant fare until it expires. Finally, some consideration was given to the possibility of looking ahead to future actions – trying to determine (for a taxi that is already handling a fare) what would be a good plan to pursue once the current fare is dropped off.

The first two considerations are the principal ones that have been directly addressed. I formulated a more intelligent way of assigning taxis to fares, in order to minimize the average travel distance from any taxi to its fare. In this way, I can avoid the situation where, if multiple taxis are available when a fare appears, they are sub-optimally assigned based on id precedence. Instead, each available taxi is considered for the fare, and the one most able to service it is chosen for the job. Fare culling was not implemented directly, but seems to be an emergent behavior due to the way taxis are assigned to fares. If a fare is too far away, there will simply always be other available fares that are nearer to the taxis, and the far fare will never be assigned.

Since the appearance of fares can have a significant impact on the optimality of the current plan, it seemed that it would be necessary to evaluate and reassign taxis to more optimal fares from time to time, even while a taxi was on the way to pick up a fare (reassignment can only be done while en route to a fare, of course; once a taxi actually picks up a fare it should not change its plans until it reaches the drop-off point, so as not to take the passenger on a longer ride than necessary). It was thus necessary to determine when would be the most effective times to reevaluate the taxi assignments. Of course this reevaluation could be performed in every step, but this could become very expensive computationally. Since the taxi assignments depend at present only on the set of available fares, the natural approach is to perform the reassignment every time the list of fares changes, to wit: upon the appearance of a new fare, after a fare expiration due to elapsed time or after a fare is dropped off.

I also found that my algorithm seemed to have a propensity for stranding taxis in the city without fuel. The default algorithm does have a way of sending taxis to the gas station when needed; I was relying on this mechanism while simply overriding the fare assignments, but my changes were apparently preventing the refueling code from properly working. To overcome this, I added a check when performing reassignments to see if the taxi had less than 1 unit of fuel remaining (out of a maximum 10), in which case it would be ordered to refuel immediately before attempting to service any more fares.

Implementation

Before going into the details of how the assignment is made, it is necessary to review the data structures in use. Taxis and fares are both modeled as tuples of elements which includes an id, x and y coordinates, and an assignment.

The actual assignment code was implemented in a method called assignTaxis that is called from within the Matlab simulation. Before calling this method, each taxi is considered to see if it is currently available to be assigned (it

```
AssignTaxis{
    Takes two input parameters:
    taxis - a list of tuples with the ids and locations of all the taxis
    fares - a list of tuples with the ids and locations of all the fares

    for each taxi
        Build preference list by ordering fares by distance from the taxi

    for each taxi
        if taxi's most preferred fare is available
            Assign taxi to that fare
        else
            Enter conflict resolution

    return the list of taxis (which includes the assignments)
}

resolveConflict{
    Takes two taxis as input (the two taxis in contention)
    if either of the taxi's preferences are already assigned to other taxis, then include those
            taxis and their preferences as well
    repeat the above step until no additions are made
    Generate all joint assignments of taxis to fares using only those taxis and fares that are in
            the conflict
    Evaluate each joint assignment to determine the average taxi travel distance
    Choose and return the joint assignment with the smallest average travel distance.
}
```

Figure 3: Pseudocode for the Taxi Assignment Algorithm

is not currently dropping off a fare or low on fuel). Two lists are then constructed: the first (‘taxis’) contains the name and location of any taxi that is eligible for assignment, and the other (‘fares’) contains the names and locations of all currently available fares. The assignTaxis method is then called with these two lists as inputs; assignTaxis returns a single list containing the generated assignments. The assignTaxis method works as follows:

First, a “preference list” is constructed for each taxi, consisting of all of the available fares ordered by distance from that taxi. This is done simply by looping through the fares, calculating the distance to the current taxi in question, and ordering them appropriately within the taxi's preference list.

Next, the algorithm loops through the list of taxis in order, and attempts to assign each taxi to its most-preferred fare. If a taxi's most-preferred fare is available, a tentative assignment is made. If the fare is already assigned to another taxi, then the current taxi and the fare’s assigned taxi are determined to be “in conflict” over the fare, and both taxis must be evaluated to see which actually is in a better position to service that fare, taking other taxi and fare positions into account. The conflict resolution method is called, taking as its input the two conflicting taxis. It determines the best joint assignment for the involved taxis, and makes the assignments on its own. Once all taxis have been assigned to fares, the taxi list is returned to the caller (at this point, each taxi in the taxi list will have its “assignment” attribute assigned to the fare it is supposed to pursue).

The resolveConflict method is a little more complicated. It takes as its input the two taxis that are in conflict over a fare. It looks at the preferences for each taxi, and if any of the preferences are assigned to some taxi that is not already under consideration, that taxi is also added to the list. The new taxi’s preferences must then be analyzed as well to see if one of them is assigned to yet another taxi. This is continued until no more taxis can be added.

Next, the algorithm generates every possible valid joint assignment using the taxis under consideration and their preference lists. Each joint assignment is evaluated to find the average distance each taxi would have to travel under that assignment. The joint assignment with the lowest average travel distance is chosen, and taxi assignments are made accordingly. This ends the conflict resolution method.

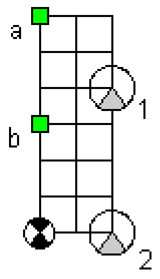
Complexity

The initial step of generating the taxi preferences lists is accomplished in $O(|T| \cdot |F|)$ time, where T is the set of taxis and F is the set of fares. Next comes the assignment/conflict resolution phase. Conflict resolution takes $O(|F|)$ to gather the necessary entities to consider (by looping through the fares under contention to build the lists, $O(|T \times F|)$ to generate all possible joint assignments, and then $O(|T \times F|)$ to evaluate and find the cheapest joint matching. Conflict resolution can be performed as many as $|T| - 1$ times in the course of running the assignment algorithm. Once the assignment phase is over, the resulting best joint assignment is returned to the caller. Thus the entire algorithm is accomplished in $O(|T| \cdot |F| + |T|(|T \times F| + |T \times F|))$ or $O(|T|(|T \times F|))$ time.

There are a couple of improvements that could be made to improve the efficiency of this algorithm. First, it would make sense to actively consider only the first two or three preferences for each taxi during conflict resolution. In this way it is possible to diminish the possibility that other taxis are drawn into the conflict resolution phase. This would also limit the exponential blowup in the number of possible joint assignments that are generated during this process.

Once fare resolution is performed, it would also be possible to prune the fare preferences list of any taxi that was assigned to a fare that was not its most-preferred. Since all more-preferred fares have already been considered and determined to result in sub-optimal joint assignments, they can be removed from the taxis preference list altogether. Scrubbing the suboptimal fares from the taxi's preference list in this way will also make it possible to continue doing conflict resolution with only the top two or three preferences, since the preferences will no longer contain fares that are ultimately sub-optimal for any taxi.

A small example will help to clarify the verbal description above: Consider the image to the right. In this scenario, there are two taxis (with ids '1' and '2') and two fares (with ids 'a' and 'b'). When the algorithm is called, it will first figure out the preferences list for each taxi. In this case, taxi '1' and taxi '2' are both closer to fare 'b'. Thus, the preference lists will appear as follows:



Taxi '1' preferences: [b][a]

Taxi '2' preferences: [b][a]

Next, it will attempt to assign each taxi to its most-preferred fare. Taxi '1' will be evaluated first, and assigned to fare 'b'. When taxi '2' is considered, it will start a conflict with taxi '1' over fare 'b', since taxi '2' also prefers 'b' over 'a'. Thus the two taxis will have to enter the “conflict resolution” phase. In this phase, all possible assignments of taxis to their preferred fares will be generated and evaluated for the average travel time they incur. The possible joint assignments generated are as follows:

Joint Assignment 1: {('1', 'a'), ('2', 'b')}

Average distance: (('1' to 'a' distance) + ('2' to 'b' distance)) / 2 = (4+5)/2 = 4.5

Joint Assignment 2: {('1', 'b'), ('2', 'a')}

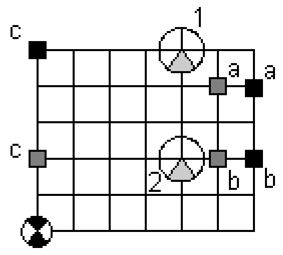
Average cost: (3 + 8)/2 = 5.5

In this case, joint assignment 1 has the more optimal average travel cost, and the conflict is resolved by assigning taxi '1' to fare 'a' and taxi '2' to fare 'b'. □

Optimality

For the example given above, the assignment generated by my algorithm is optimal, and better than the one that would be generated by the default algorithm in this situation (it would have paired ‘1’ with ‘b’ and ‘2’ with ‘a’). The assignment is optimal because there is no other assignment that can result in the collection of both available fares in less time. Furthermore, this will hold true regardless of the drop-off points for the fares, since the average time required for the drop-offs will be constant regardless of the initial fare assignment.

However, this algorithm is not guaranteed to be optimal in all situations. It will be optimal in situations such as the above, where the number of taxis is not exceeded by the number of fares, but in the highly-variable scenarios within the simulator, situations will arise for which my algorithm will produce sub-optimal results. Consider the image at right, where the eventual drop-off locations of the fares are shown in black. My algorithm would assign taxi ‘1’ to fare ‘a’ and taxi ‘2’ to fare ‘b’. Taxi ‘2’ will be assigned to fare ‘c’ after dropping off fare ‘b’, and taxi ‘2’ will have to travel 11 steps to complete its assignments, while taxi ‘1’ will be done after 3 blocks and have nothing to do for the rest of the time. An optimal assignment would be to send ‘2’ to ‘c’ while ‘1’ takes care of ‘a’ and then ‘b’; each taxi would have to travel for 7 steps, which means the optimal assignment will drop off the fares significantly quicker than my algorithm, although the average distance traveled by the taxis will still be the same.



The problem of assigning taxis to fares is related to the bipartite matching problem as explained in [4]; it would be possible to use bipartite matching algorithms to generate these assignments, by constructing two graphs that correspond to the set of taxis and the set of fares. Every taxi should be given an edge to each fare, with the weight of the edge equal to the inverse of the distance between the taxi and the fare. It would then be possible to use any bipartite matching algorithm to find a maximum matching, which would correspond to a valid assignment of taxis to fares. The assignments generated in this manner would still not be completely optimal however, since the matching will be optimized for the first set of assignments; it does not account for the fact that some taxis must service multiple fares if there are more fares than taxis.

Another important factor that makes my algorithm sub-optimal is that no attempt is made to account for the expected appearance of new fares. If the fare appearance distributions are known ahead of time (or if they can be approximated during simulation), then it would be possible to anticipate the possible arrival of fares and issue the taxis orders to move to a location that does not yet have a fare present if it is a known that a fare is likely to appear soon. Anticipatory actions such as this would be especially useful in situations where no fares are available, in order to minimize wait time when a fare does appear, but there is currently no mechanism in place to accommodate such strategies.

A fairly uncommon situation that would cause my algorithm to perform sub-optimally is if a taxi is assigned to a fare that is set to expire before the taxi is able to reach it. In such cases, the taxi will be wasting time until the fare expires or it is otherwise reassigned. An optimal strategy should be able to gauge the ability of a taxi to reach a fare before expiration and avoid making hopeless assignments, and my algorithm does not evaluate this. In practice however, such situations would be rare since taxis are assigned to the overall shortest-path pairings; it is unlikely that a taxi would be paired with a fare that is beyond its reach unless no other fares were present.

CHAPTER 5

EXPERIMENTAL SETUP

Experimental Metrics

In order to compare the two control algorithms to see if indeed the new one is an improvement over the original, it was necessary to come up with compelling metrics by which the two could be compared. The most important metric is naturally the number of fares that the taxi fleet is able to service in a given amount of time. However this only gives part of the picture, so the number of fares that are allowed to expire should also be recorded, since this allows us to calculate the “success ratio” of the taxi fleet (fares serviced / fares expired); a single number that can potentially be used to compare even between different city scenarios. Also recorded is the average fare wait time; how long on average a fare has to wait after appearing until it gets picked up. This metric will presumably mean less in high-fare-density scenarios, since a taxi should be able to find a nearby fare without too much difficulty most of the time, but in low-fare-density scenarios (in particular, scenarios where any reasonable algorithm is likely to be able to pick up most or all of the fares), the average fare wait time could provide an important indicator of an algorithm's efficiency compared to others.

One other metric that was considered is “Taxi Utilization”, or the proportion of the time each taxi spends actively trying to service fares, as opposed to roaming or refueling. While I did implement this metric in the simulator, I ultimately decided against using it, as it seemed a poor indicator of actual performance. Especially in low-fare-density environments, it could be misleading:

a highly efficient algorithm that quickly handles the available fares and then starts roaming while waiting for more fares to appear would have a lower taxi utilization than a less efficient algorithm which took more time to service the fares it had.

Experimental Design

Once the chosen metrics were implemented in the simulator, it was necessary to come up with appropriate testing scenarios in which to evaluate the algorithms. I performed testing primarily in two scenarios. The first test scenario, which we will call TwoTaxis, consists of an 8-by-8 city grid with a single point of interest to generate fares, and two taxis to make passenger deliveries. The second scenario, called ZedCity, is a much larger 16-by-16 grid, with ten POIs to provide fares and a 5-taxi fleet to service them.

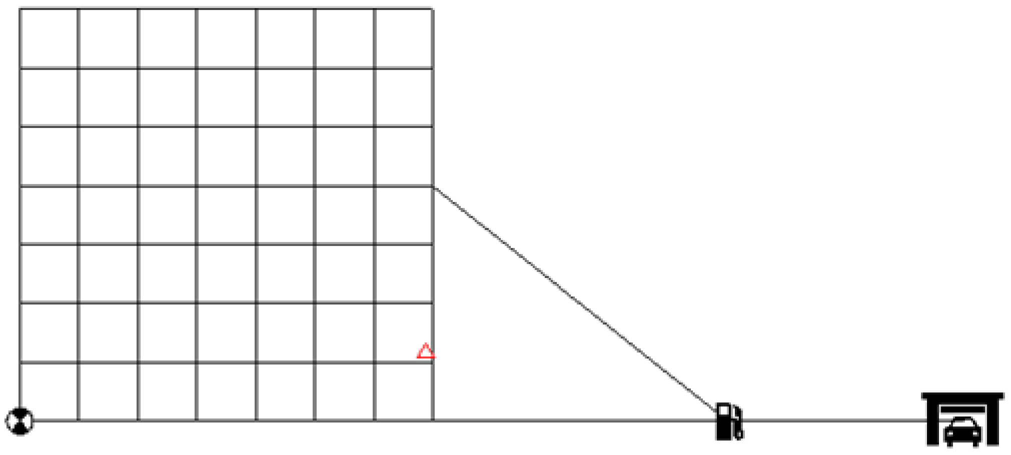


Figure 4: City Layout for the TwoTaxis Scenario

In order to ensure a more or less fair comparison of the performance between the algorithms tested, the "random" fare arrivals were rigged so that every algorithm would be confronted with the exact same set of fare appearances;

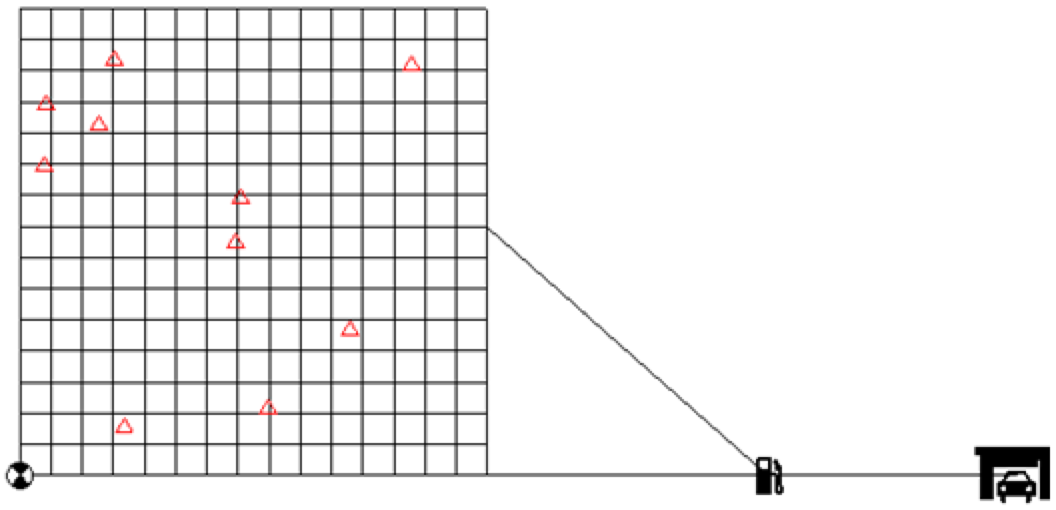


Figure 5: ZedCity Layout

the differences in performance are thus entirely resultant from the control algorithm itself, rather than being an artifact of different randomly-produced runs of fares.

Since the simulation itself can be run for an indefinite amount of time, it was necessary to come up with an appropriate evaluation window – a suitable amount of simulation time to get a fair representation of the efficacy of each algorithm. Each taxi can run for somewhere near 21,000 simulation cycles before needing to refuel, and it seemed like a good idea to include refueling trips in the test cases, so a time of 50,000 simulation cycles was chosen for each test run. This is enough time that each taxi in the fleet has to make two trips to the gas station, and allows for some buffer time besides. I used the dispatcher to collect all of the data necessary to evaluate these metrics, due to the fact that it has access to nearly all of the information in the simulation at any time.

CHAPTER 6

RESULTS

I discovered during testing that my reassignment method sometimes caused a taxi to forget to refuel. The default taxi AI has a mechanism to know when it must refuel, and I had not modified that aspect of the AI, but apparently my new code was interfering with it, causing taxis to occasionally strand themselves within the city. I accordingly added code to force the taxis to go to the gas station once their fuel fell below a certain level. For comparison, I ran the scenarios both with and without the modified refueling code. Each scenario was evaluated by each algorithm on three different fare-frequency settings to achieve a more comprehensive comparison. In the tables, the algorithms are represented by 'D' for the default control, 'I' for my improved variant, and 'F' for my algorithm with the modified fuel-aware code.

The table below summarizes the results of running each algorithm on the TwoTaxis scenario.

POI rate	.003			.005			.01		
Control	D	I	F	D	I	F	D	I	F
Serviced	70	77	74	106	117	117	149	119	155
Dropped	9	2	6	20	8	8	114	141	107
Avg Time	513.8	416.0	447.3	616.2	440.9	440.2	898.4	932.3	820.5

Table 1: Results for Default, Improved, and Fuel-aware algorithms in the TwoTaxis trials

In the scenario runs with the POI spawn rate set at .003 and .005, the improved algorithm performed better than the default algorithm, and the fuel-aware algorithm also did quite well. The fuel-aware method also did better on the

.01 setting, though this time the improved algorithm actually did worse, due to the stranding of one of the taxis after it ran out of fuel. It is also clear from this graph that the improved algorithm does significantly better regarding the average wait time, due to its effective prioritizing of the available fares.

The results for the ZedCity trials are likewise shown in the table below:

POI rate	.0005			.001			.002		
Control	D	I	F	D	I	F	D	I	F
Serviced	80	85	85	161	176	170	252	246	255
Dropped	13	7	7	33	19	24	125	133	124
Avg Time	593.7	469.4	469.4	717.1	519.9	546.2	859.6	819.8	794.4

Table 2: Results for the Default, Improved, and Fuel-aware algorithms in the ZedCity trials

In the ZedCity results there is a similar pattern of improvement over the default algorithm by the improved versions. There is also a similar trend in average fare wait times here as there was in the TwoTaxis scenario, with the improved algorithms showing significant improvement over the default algorithm.

From these results it is clear that in each of the scenarios considered, my modified algorithm outperformed the default on all fronts, achieving a higher success ratio with a shorter average wait time, reflecting the positive effect of the more efficient assignment algorithm. Of note is the fact that in scenarios where ‘I’ did not end up stranding a taxi, it seems to have done better than ‘F’. It is possible that this is could be caused by my modification being over-zealous in sending taxis to the gas station, depriving them of small amounts of fare-servicing time on each refueling cycle. This indicates that even the timing of refueling can be subject to improvement, and can have a noticeable effect on results.

CHAPTER 7

SUMMARY AND CONCLUSION

TaxiWorld provides an excellent example of a multi-agent planning domain; such domains are of great interest to the planning community, and good solving methods for such problems would find application in many different arenas. MDPs, in their various forms, provide powerful representational abilities which can be used to capture very complex domains in a form which can be fed to a solver to generate optimal policies; multiple such representations were considered for use with TaxiWorld, but it was determined that an empirical approach would be better suited to improve on the shortcomings of the default control mechanism in use on the simulator. After identifying several shortcomings, I presented a new approach that could improve on the default, and ran a number of simulation trials demonstrating that the new control scheme is in fact an improvement over the old one. Naturally, even this improved algorithm will still have shortcomings; possible future improvements were discussed as well, though an empirical approach will necessarily be only an approximation of the fully-optimal policy generated by an MDP solver.

In doing this research, I was able to learn the types of problems that are representable within different families of MDP models, and to use the Dec-POMDP and MMDP frameworks to model a fairly complex scenario. Through the development of my empirical model, I demonstrated that although finding an optimal policy for a problem may be intractable, it may be possible to leverage domain-specific properties to generate a solution strategy that achieves an acceptable level of success at significantly less computational cost. Because the cost-to-benefit ratio of developing and using a fully optimal solution (or even a theta-approximate one) would be extremely large, in many cases it might be far more practical to craft a reasonably efficient domain-specific solution than to formalize the problem and use a full-blown general solver to generate an optimal policy.

REFERENCES

1. Bernstein, D., Zilberstein, S., and Immerman, N. “The Complexity of Decentralized Control of Markov Decision Processes”. *16th Conference on Uncertainty in Artificial Intelligence*. 2000.

2. Hansen, et al. “Dynamic Programming for Partially Observable Stochastic Games”. *American Association for Artificial Intelligence*. 2004.

3. Hubbe, A., Ruml, W., Yoon, S., Benton, J., and Do, M. 2008. “On-line Anticipatory Planning”. *Workshop on a Reality Check for Planning and Scheduling under Uncertainty, ICAPS 2008*. 2008.

4. Karp, R. et al. “An Optimal Algorithm for On-line Bipartite Matching”. *Association for Computing Machinery*. 1990.

5. Lemons, Sofia. “Continual Online Planning”. *Twenty-Fourth AAAI Conference on Artificial Intelligence*. 2010.

6. Oliehoek, Frans. “Decentralized POMDPs”. 2011.

7. ---. “Decision Making for Cooperative Agents”. 2010.

8. “Formal Meta-framework to Evaluate System Readiness Under Critical Decision Making”. *Impact Lab, Arizona State University*. 2010.

9. Ryan, A., Tisdale, J., Godwin, M., Coatta, D., Nguyen, D., Spry, S., Sengupta, R., and Hedrick, J. K. “Decentralized Control of Unmanned Aerial Vehicle Collaborative Sensing Missions”. *Proceedings of the American Controls Conference*. 2007.

10. Seuken, S. and Zilberstein, S. “Formal models and algorithms for decentralized decision making under uncertainty”. *Journal of Autonomous Agents and Multi-agent Systems*. 2008.

11. Szer, et al. “MAA*: A Heuristic Search Algorithm for Solving Decentralized POMDPs”. *Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence (UAI)*. 2005.

12. Xu, Jin, and Huang, Zhe. “An Intelligent Model for Urban Deman-responsive Transport System Control”. *Journal of Software, Vol. 4*. 2009.