Task Partitioning for Distributed Assembly

A Thesis

Submitted to the Faculty

of

Drexel University

by

James Worcester

in partial fulfillment of the

requirements for the degree

of

Doctor of Philosophy in Mechanical Engineering and Mechanics

2015

Acknowledgments

I would like to thank my advisor, Ani Hsieh, for providing continual direction and encouragement. My collaborator Rolf Lakaemper was invaluable in providing the code for the Kinect. I would also like to thank the members of my committee, including Paul Oh, David Breen, Suhada Jayasuriya, Youngmoo Kim, John Lacontora, and Alan Lau. This research owes a debt to all the undergraduates who helped out along the way, most especially Joshua Rogoff, Mary Conrad and Zach Block. I am also grateful for the support and advice of the graduate students of SASLab, they are Bill Mather, Ken Mallory, Dean Galorowicz, Dennis Larkin, Matt Michini, and John Alexander. This work was supported by the National Science Foundation [CNS-1143941]; and the National Institute of Standards and Technology [ARRA-NIST-10D012].

Table of Contents

List of Tables

List of Figures

Abstract
Task Partitioning for Distributed Assembly

James Worcester
Advisor: M. Ani Hsieh

This thesis addresses the problem of how to plan a strategy for a team of robots to cooperatively build a structure, henceforth referred to as the distributed assembly problem. The problem of distributed assembly requires a range of capabilities for successful completion of the task. These include accurate sensing and manipulation using a mobile robot, the ability to continuously adhere to precedence constraints on placements, and the ability to guarantee static stability at every stage of construction.

The fundamental contribution of this work is to propose methods to address task allocation problems in the presence of constraints on task ordering. Algorithms are presented to partition 2- and 3-D assembly tasks into N separate subtasks that satisfy local and global precedence constraints between the assembly components. The objective is to achieve a partitioning that minimizes completion time by minimizing the workload imbalance between the robots, and maximizes assembly parallelization. Towards this objective four approaches are presented. The first is an approach where each robot runs a simultaneous Dijkstra's Algorithm with its own root. The second approach incorporates online workload balancing and error correction by adding a communication scheme and a scanning robot equipped with a visual depth sensor. The third approach addresses the task partitioning using an algorithm inspired by Ant Colony Optimization. Finally, the problem of cooperative manipulation for tasks that require close coordination is addressed. All approaches are tested in both simulation and experiment.

1. Introduction

Distributed autonomous assembly of general two (2D) and three dimensional (3D) structures is a complex task requiring robots to have the ability to: 1) sense and manipulate assembly components; 2) interact with the desired structure at all stages of the assembly process; 3) satisfy a variety of precedence constraints to ensure assembly correctness; and 4) ensure static stability and structural integrity throughout the assembly process. While the distributed assembly problem represents a class of tightly-coupled tasks that is of much interest in multi-robot systems (Chaimowicz et al., 2001), it is also highly relevant to the development of next generation intelligent, flexible, and adaptive manufacturing and automation. of different sizes, shapes, and weights onto a pallet; while over 60% of goods shipped worldwide is achieved by stacking them onto pallets, these tasks are still carried out mostly by humans (Balakirsky et al., 2010; Schuster et al., 2010b). Palletizing, construction, manufacturing and maintenance of locomotives, power turbines, etc. are all examples of 3D assembly where automation can significantly boost productivity and reduce worker injuries and operational costs.

The execution of tightly-coupled tasks by multi-robot teams has mostly focused on cooperative grasping and manipulation (Mataric et al., 1995; Fink et al., 2008). These works, however, do not address the challenges imposed by the need to satisfy specific precedence constraints during assembly process. These constraints are especially important in applications like automated palletizing, construction, manufacturing, infrastructure repair and maintenance since automated strategies must ensure correctness as well as stability of the resulting structures. While there has been significant focus in micro/nano-scale assembly (Klavins, 2007; Evans et al., 2010; Matthey et al., 2009; Rai et al., 2011), automated macro-scale assembly is gaining increased attention. Recent work in this area includes (Petersen et al., 2011; Yun et al., 2009; Yun and Rus, 2010; Stein et al., 2011; Heger and Singh, 2010;

Lindsey and Kumar, 2012).

In Werfel and Nagpal (2008); Petersen et al. (2011); Werfel et al. (2014), assembly is achieved through a combination of robots with limited sensing and actuation capabilities and assembly components capable of storing and communicating location information with the robots. The focus of these works is on designing a set of consistent local attachment rules that ensure completeness and correctness of the assembly, while obeying local constraints between pieces and avoiding unrecoverable situations. In Yun et al. (2009); Yun and Rus (2010); Stein et al. (2011); Schoen and Rus (2013), a workload partitioning strategy is presented to enable a team of robots to achieve parallel construction at the macro scale. The approach maintains a Voronoi decomposition of the structure based on the assembly robots' locations by minimizing the total difference in the masses of the assembly components in each cell. Failures of robots, ordering constraints, and changes to an existing structure are also addressed. Lastly, the synthesis of assembly instructions for special cubic structures by a team of quadrotors is discussed in Lindsey et al. (2011). While the assembly instructions can be executed by quadrotor teams of any size, correctness of the assembly strategy is achieved through serial execution of the assembly instructions. As such, this approach does not take advantage of the potential for parallelization afforded by a multi-robot team. In Schoen and Rus (2013), another strategy for task ordering to handle these precedence constraints is presented. This strategy also handles the problem of limited part availability. Lastly, the synthesis of assembly instructions for special cubic structures by a team of quadrotors is discussed in Lindsey et al. (2011). While the assembly instructions can be executed by quadrotor teams of any size, correctness of the assembly strategy is achieved through serial execution of the assembly instructions. As such, this approach does not take advantage of the potential for parallelization afforded by a multi-robot team. Amorphous materials are used in Napp and Nagpal (2014) as a way of handling potential obstacles within the construction site. Khoshnevis (2004) considers the problem of apply-

ing 3-D printing technology to building scale projects, although this still requires building a gantry taller than the building. D'Andrea (2011) demonstrates that robotic placement of parts can be made reliable enough to build large structures.

There has also been considerable work done on task partitioning from a swarm robotics perspective (Gerkey and Mataric, 2004). The focus from a swarm robotics perspective is on decentralization, limited perception and communication, and robustness to single robot failures. Pini et al. (2013) considers a foraging task, composed of independent tasks which do not have constraints on task ordering. In the hierarchical auction setup of Jones et al. (2011), precedence constraints are handled by having two sets of robots applied to a problem where fires need to be extinguished on a map of a town. Each fire truck plans a route to use for bidding on the fires, and part of this process is to determine how many rubble piles will need to be cleared from the proposed route. To do this it holds an auction for each rubble pile among the bulldozers (the second type of robot). This method allows for two types of tasks where one task can depend on any number of the other type of task. The distributed assembly problem requires a more general structure where any task may be dependent on another task, so long as there are no cycles in dependency. Another type of contraint is considered in Khaluf and Rammig (2013). In this work, time constraints are allowed, where each task must be completed by a certain deadline or have a cost imposed. Only soft deadlines (missing the deadline adds a cost) are considered, as they cannot guarantee that a hard deadline (missing the deadline implies failure of the task) will not be missed. This could potentially be used for task ordering, although guarantees that tasks would not happen out of order would be needed.

The problem of cooperative grasping and manipulation has been considered in Mataric et al. (1995); Sugar and Kumar (1999); Pereira et al. (2004); Fink et al. (2008); Michael et al. (2011); Yamashita et al. (2003); Spletzer et al. (2001); Chang et al. (2000). These works focus on the question of how a team of robots can work in close coordination to

accomplish a common task. I am interested in how these cooperative manipulation tasks affect the task partitioning problem. Specifically, given a larger team of robots, how subsets of this team get assigned to work in close coordination for a specific task and then rejoin the larger team.

Despite these successes, significant challenges still remain. First, existing macro-scale assembly strategies often reduces to a serialization of the assembly procedure despite employing multiple robots (Werfel et al., 2014; Lindsey and Kumar, 2012). While this ensures correctness of the resulting structure and safe execution, it can significantly hamper productivity by not exploiting parallelization in the assembly process. Second, existing strategies often rely on external sensors for localizing the assembly components (Lindsey and Kumar, 2012; Schoen and Rus, 2013), *i.e.*, stationary sensors mounted in the workspace. While such a strategy may be feasible for small work cell volumes, such an approach may be challenging for large workspaces since it would be difficult to provide enough coverage and accurate localization. This work will seek to provide an approach that minimizes construction time while addressing the precedence constraints likely to be found in an assembly problem.

1.1 Contribution

The main contribution of this work is to extend graph search techniques and evolutionary algorithms, in particular Ant Colony Optimization, to be applicable to the problem of distributed assembly. The objective throughout the approaches presented is to minimize completion time, usually by minimizing the related characteristic of workload imbalance. Minimizing this imbalance between robots will minimize the time robots spend idle, which will in turn minimize completion time. Various aspects of the problem are considered in detail, including task parallelization, preplanning, online workload balancing through exchanges of assigned tasks, communication schemes, and finally extending the approach

to handle more complicated arch-like structures that require close coordination between robots.

The approaches presented here can also be applied more broadly to other task partitioning problems involving complex constraints. One broad area would be paralell processing problems on how to assign jobs to multiple processors, where the tasks are not cleanly separable but have precedence constraints on ordering of tasks. Other areas within robotics would be warehouse problems requiring more complex interaction between robots. The best known warehouse solution is Kiva systems (http://www.kivasystems.com), where human packers are required to combine the items into one package. One of the obstacles to automating that process would be the uncertainty in what order items will arrive. By adding ordering constraints, the approaches presented here could ensure that items arrive in a specified order, making it possible to automate this process using palletizing software such as Schuster et al. (2010a). The most direct application of these approaches would be to an automated construction approach, using robots designed to perform specific routine construction activities, from pouring concrete to spreading mortar, in concert with humans performing more varied tasks. With either sensing to detect when the human tasks are completed or manual check-ins, the robots could plan a task ordering to accomplish their workload at appropriate times relative to related tasks.

1.2 Organization

Chapter 2 will review background material in graph search techniques, evolutionary algorithms, and particularly Ant Colony Optimization. As robotics is necessarily a multidisciplinary field, the goal of this chapter is to provide enough information so that readers unfamiliar with these topics will still find the rest of the thesis clear after reading this background. This chapter will not attempt to provide a comprehensive discussion of these topics, but rather will seek to introduce the most relevant aspects that will be important to

later discussion throughout this work.

Next, Chapter 3 lays out the problem to be solved, and introduces variables and representations that will be used throughout the thesis. This chapter will be limited to material that will remain relevant throughout the document, and will not include representations used only in specific chapters. Where necessary, later chapters will include additional information about changes to problem definitions or representations used.

Chapter 4 will address the partitioning of the assembly task into separate subtasks to be performed by individual robots. The objective is to achieve a partitioning that minimizes the workload imbalance between robots, similar to (Yun et al., 2009; Yun and Rus, 2010),. Specifically, the focus is on a class of structures where the precedence constraints between the various assembly components can be described by a graph, and construct an allocation strategy that explicitly accounts for the constraints and discrepancies in the size of the assigned tasks. Chapter 5 considers the partitioning of the heterogeneous robot team into assembly and scanning robots. Assembly robots will be tasked to assembly the desired 3D structure using a collection of assembly components of varying shapes and sizes, while scanning robots provide real-time visual feedback of the state of the structure during assembly. Additionally, this chapter will also include online workload balancing that does not violate local precedence constraints between assembly components. An advantage of the proposed workload balancing strategy is that it can be used with any pre-existing assembly plan modeled as a tree where the root node represents an assembly component located on the exterior of the desired structure. The main contribution of this chapter is a cooperative assembly framework that integrates the planning, sensing, and workload balancing into a single coordination architecture for teams of heterogeneous robots.

In Chapter 6, the problem is approached with an Ant Colony Optimization (ACO) (Dorigo et al., 1999; Sauter et al., 2002) based solution to the assembly partitioning problem, which I call Distributed Assembly by Ant Colony Optimization (DAACO). The pri-

mary contribution of this chapter is the modification of ACO to manage teams of cooperating agents, rather than to determine a policy for a single agent. Where traditional ACO uses groups of ants exploring the search space of solutions for a single ant, DAACO uses groups of teams of ants to explore the search space of solutions for a single team. The approach uses N teams of M ants each. Each team acts independently during a generation, but pheromone is shared across teams, allowing them to learn from each others' experience. Each team has a manager which directs the interaction of the team members, orchestrating a sequential node-claiming sequence.

Chapter 7 extends the online approach to handle the problem of more closely coordinated manipulation needed for some structures. Specifically, this chapter considers a class of archlike structures, where most partial assemblies require the support of one or more robots until later pieces are added. Building on the approach in Chapters 4 and 5, the assembly task is separated into subcomponents to enable parallel assembly by the assembly robots. This planning phase minimizes the workload imbalance between the assembly robots without violating local attachment constraints, given by the geometry of assembly tiles/components, and global precedence constraints, required for structural stability. During construction, robots coordinate online to ensure that each piece is made stable by later pieces before being released. This chapter will examine how this affects the minimum number of robots needed for assembly, as well as how to avoid potential deadlock scenarios. The main contribution of this chapter is a cooperative assembly framework that is capable of assembling more complicated structures that are not guaranteed to be statically stable for all intermediate states of assembly.

Simulation results and discussion are presented in Chapter 8. The focus here will be on how the approaches perform in terms of minimizing workload imbalance on different structure types, amount of communication needed, and how the approaches scale with the size of the structure and the number of robots in the team.

Experimental results are given and discussed in Chapter 9. This chapter provides the experimental validation for the approaches presented, and discusses some of the challenges inherent to this problem.

Some concluding remarks and directions for future work are given in Chapter 10. The main contribution of the work is discussed, the relation to the rest of the field is considered, and areas for extension are proposed.

2. Graph Search Strategies and Evolutionary Algorithms

In this chapter background material will be provided for graph search strategies, such as Dijkstra's Algorithm, as well as evolutionary algorithms, in particular Ant Colony Optimization.

2.1 Graph Search Algorithms

The fundamental purpose of a graph is to represent relations between pairs of objects or events. In this work, these objects will be parts of a structure being built. A graph, such as that in Figure 2.1, has a set of nodes or vertices, V, that represent individual objects or events, and edges, E, that represent some relation between them. In this work, edges are generally used to represent a direct connection between parts within the structure. While graphs are widely used for many purposes, this discussion will focus on just one of those areas: finding a path between a pair of nodes. As edges often represent a physical distance, the most direct application of this is finding a route between two points, and for some algorithms that becomes finding the shortest route.

Two basic methods of approaching this type of problem are depth-first search (DFS) and breadth-first search (BFS). In DFS, the search is conducted by following a path as far as it goes (or until it revisits a node already on the path), and then backing up and trying a different path until the target node is reached. An example of DFS can be seen in Fig. 2.1, with the nodes labeled in the order vistited. As this algorithm terminates as soon as a path to the target is found, it is not guaranteed to find the shortest path. In BFS, the algorithm first checks all neighbors of the starting node, then all neighbors of those nodes, and so on, as shown in Fig. 2.2. The downside of this is that it can be very demanding for memory usage. Additionally, if edges have weights on them, for example representing distance, this

approach is not guaranteed to find the shortest path, just a path with the fewest possible edges. Another way of looking at DFS and BFS is to consider the data structure holding the possible routes. BFS uses a queue, where the first node put into the queue is the first node taken out and looked at. In contrast, DFS uses a stack, where the last node placed into the stack is the first one removed and visited.

An algorithm that is guaranteed to find the shortest path from a start node to a separate goal node is Dijkstra's Algorithm. This algorithm is structured similarly to BFS, but changes how nodes are added to the queue when a new node is visited. In BFS, all new neighbors are added, with the ordering ignored. In Dijkstra's algorithm, the queue is replaced by a priority queue, where each node added is also given a priority, and the lowest priority node is the next to be removed from the queue and visited. For Dijkstra's Algorithm, this priority is the distance from the root (starting node). When a node is visited, it considers a path through that node to each of its neighbors. If that path gives a shorter distance to that neighbor than has been found previously, the neighbor is assigned the new shorter distance. This means that each node visited is guaranteed to have found the shortest path back to the root, as all unvisited nodes must have a higher distance, and therefore could not be part of a shorter distance (note that this requires non-negative edge weights). The algorithm does not terminate until the goal node is removed from the queue and visited, at which time this guarantee of having found the shortest path applies to the goal node as well. An example is shown in Fig. 2.3, note that the path to the node labeled 5 takes a route that has more edges but a lower cost than the alternate path through node 6.

2.2 Evolutionary Algorithms, Genetic Algorithms and Ant Colony Optimization

Many problems involve searching a high-dimensional space for an optimal solution, and often in these situations finding the global optimum is an NP-hard problem, so it is not practical to look for a guarantee of global optimality in those cases. This leaves us with

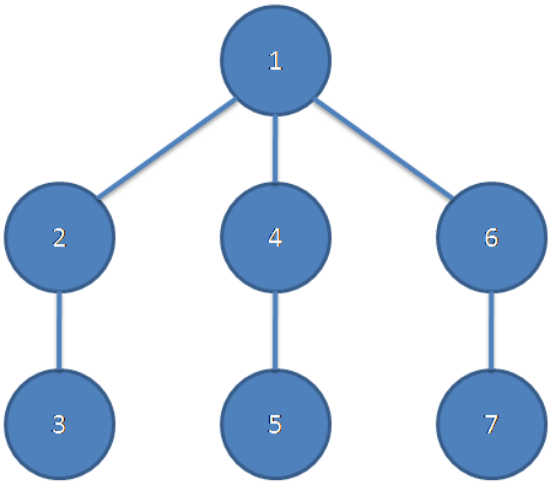


Figure 2.1: An example of a depth first search.

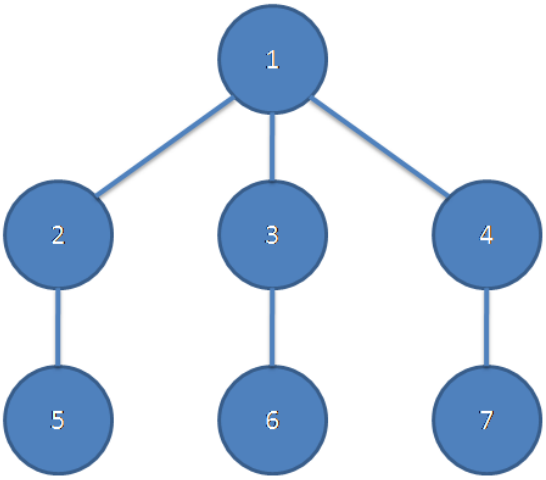


Figure 2.2: An example of a breadth first search

the question of how to approach these problems. There are many approaches available in the various techniques used in machine learning, but the focus here is on biologically-inspired evolutionary algorithms. Based on natural selection, these algorithms attempt to provide a framework where gradual improvement of the candidate solutions through many generations reaches a fairly good overall solution by the end of the process.

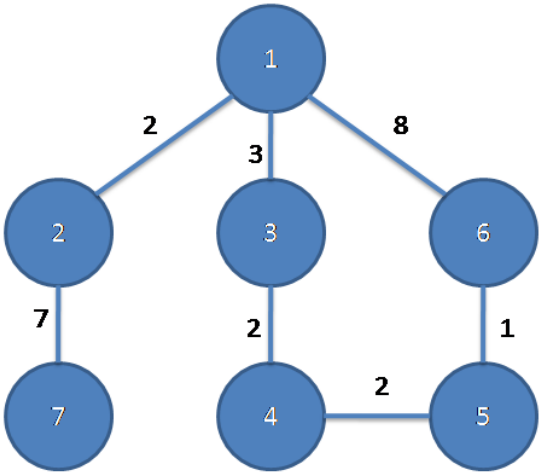


Figure 2.3: An example of Dijkstra's Algorithm

For any evolutionary algorithm, there are a few broad concerns. The first is how to define the fitness function, which determines what approaches will be kept for the next generation. This is analogous to an objective function, and how it is defined varies from problem to problem, but anything not included in the fitness function will not be optimized for in the outcome of the algorithm. Throughout the approaches discussed in this thesis, one central consideration will be the variance of the workload between robots. This is done in order to minimize completion time by ensuring that all robots have similar amounts of work.

Another consideration in evolutionary algorithms is to maintain diversity in the population, so that more of the search space is evaluated. A common failure mode for these algorithms is an early convergence into one local minimum, where the rest of the generations are spent on small refinements within that local minimum without ever discovering better areas of the search space. This is similar to the problem of overfitting in other machine learning algorithms, where the solution becomes too finely tuned to a particular data set and the broader picture is lost.

The most common type of evolutionary algorithm is a genetic algorithm. This works by having a population of candidate solutions with some number of ”genes” defining each solution. In each generation, the best candidates are kept and replicated with random changes made to some genes, which serves the same purpose as mutations in evolution. The exact way that these changes are made varies, and sometimes includes combining pairs of candidates to create a ”child” candidate with some propoerties of each parent. The purpose of combining two candidate solutions is to explore new areas of the search space that may be outside the current set of local minima represented by the candidate population.

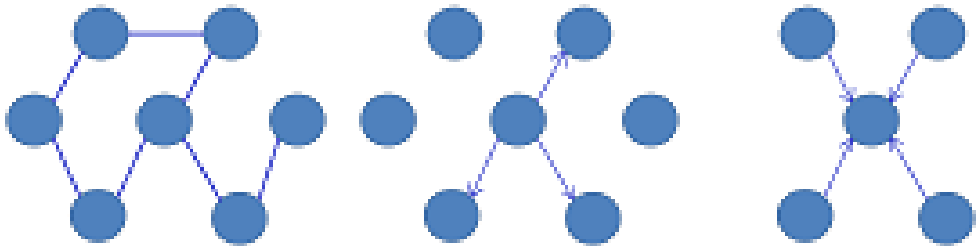
A more recent form of evolutionary algorithm is Ant Colony Optimization (ACO), which will be applied to the distributed assembly problem in Chapter 6. This algorithm takes inspiration from the behavior of ants using pheromone to communicate about paths to food. Similar to genetic algorithms, ACO is a biologically inspired algorithm that gradually refines candidate solutions through many generations to optimize performance on some objective function. The main difference between the two is that ACO stores the information in the simulated environment through depositing pheromone, while GA stores the information in the genes of the candidate population. Storing the information in the environment can lead to a more natural representation for tasks closely linked to a spatial representation, for example path finding. In the implementation presented here, the environment where pheromone is deposited is a graph representing the structure to be built. One advantage of this method is that information is shared by all candidate solutions rather than held within each candidate. However, this means there is still the danger of early convergence to one local minimum, so methods to maintain solution diversity are important for any implementation of ACO. A final advantage of ACO for this application is that the extension to a team of robots is more natural than it would be for GA. To apply GA to a team of robots, one needs to either have all robots share the same genes, or be faced with a dimensionality problem if each robot keeps it’s own genes. Since all candidates in ACO

already share information, which is dependent on position in the environment, a team of robots using an ACO strategy can differentiate their behavior based on their location more easily.

3. Problem Description

The approaches here will use a team of N homogeneous robots, each capable of transporting a single assembly component from a cache location to the assembly site. In Chapter 5, this is further subdivided into N_a assembly robots with the above capability, and N_s scanning robots, equipped with Microsoft Xbox Kinect visual depth sensors. Let M denote the number of distinct assembly components/tiles/nodes where t_i denotes a component/tile/node of type i. It will be assumed that each tile of type i can be described as a general polytope and that the robots know the geometries of the different tile types a priori. Furthermore, every tile of type i will have a fixed number of attachment sites. These attachment sites are locations where tiles can mate and lock onto other tiles. Let $\mathscr{W}$ denote the workspace and S denote a desired target structure. The structure-free portion of $\mathscr{W}$ is given by $\mathscr{W}_f = \mathscr{W} \setminus S$. Let $G_S = \{V_S, E_S\}$ denote the *structure graph* where each node in V_S represents each structural component that can be placed next to or on top of other component(s) by a single robot to form larger structures. An edge (u,v) exists in E_S if v can be reached from u through a path in $\mathscr{W}_f$ and vice versa. For every $(u,v) \in E_S$, a weight is assigned equal to the planar Euclidean distance between u and v. The *mass* of a node is the amount of time required to build the node. It is also assumed that S is finite in size or V_S is a finite set.

In this work, the impact of precedence constraints on the order of assembly for task partitioning will be considered. It is assumed the constraints are of the form $u \prec v$, or u must be built before v which represents a variety of assembly constraints, *e.g.*, different materials that must be combined in a specific sequence or the placement order of supporting components for structural stability during assembly. For a desired S, let a *constraint graph* be defined as a directed graph $G_C = \{V_C, E_C\}$ such that $V_C = V_S$ and $E_C = \{(u,v) | u \prec v\}$ and for every $(u,v) \in E_C$, u will be called a *support*, and v a *supported node*. In general, given S,



(a) (b) (c)

Figure 3.1: (a) A structure graph, navigation between nodes is only possible if an edge is present between those nodes. (b) A constraint graph requiring that the central node be built before the adjacent nodes. (c) An example of a structure that would not be in S_A. The displayed constraints state that each of the outer nodes must be placed before the central node. Assuming the robot can only move through the displayed spaces, a robot cannot reach the central node once all of its supports have been placed.

these constraints can be obtained by adding an edge for each node that is directly supported by another node (*i.e.,* must be placed after that node), derived from an AND/OR graph representation (Sanderson et al., 1990), or derived from a sequential assembly plan similar to (Grushin and Reggia, 2008). Finally, define a directed graph $G_R = \{V_R, E_R\}$, *i.e.*, the *route graph*, where $V_R = V_S$ and E_R is given by $E_R = \{E_S \setminus E_D\}$ with $E_D = \{(u,v)|\ (v,u) \in E_C\}$. The route graph represents all the viable paths within the structure that do not travel from a supported node to its support. It is assumed that there exists an obstacle-free and fully connected $\mathscr{W}$ prior to the assembly of S to simplify the motion-plans used to estimate the time to assembly for individual components and to ensure all components in S are reachable at some point in the assembly process.

The set of admissible structures S_A includes only structures whose G_R is strongly connected and G_C has no cycles. For 3-D structures, the limitations of the mobile manipulator adds the restriction that to be a member of S_A, each node must be reachable from $\mathscr{W}_f$ according to the geometry of the assembly robot. In this work, the objective is to partition S into N (or N_a in Chapter 5 subcomponents that can be assigned to individual robots for assembly. The resulting allocation should minimize the workload imbalance between robots

while satisfying the constraints in G_C. Furthermore, the desired decomposition of the task should result in robots spending an approximately equal amount of time on their tasks, while minimizing the time robots wait for one another.

Finally, it is assumed robots are able to localize within the workspace, identify and manipulate the components located at the parts/components cache, and are able to determine thier position relative to the structure throughout assembly. While this work is focused on ground mobile manipulators, the proposed partitioning strategies can be extended to other autonomous robots.

4. Dijkstra-Based Methods

This chapter proposes three variants of one algorithm whose inputs are G_S, G_C, and the following information for each node in the structure: location, time required to build, distance to cache, and whether that node is on the boundary of S, i.e. directly adjacent to $\mathscr{W}_f$. The outputs of the algorithms are N assembly sequences each specifying the order in which the components should be placed.

The base algorithm for the partitioning of S consists of three phases. Phase I of the algorithm begins by running a simultaneous Dijkstra's algorithm for each robot. Different from a standard Dijkstra's algorithm, this algorithm replaces the root with a set of starting roots, and runs a simultaneous Dijkstra's algorithm by finding at each step the shortest distance of any node to any root. The set of starting roots is found by determining the angular density of the structure about the center of mass, and then spacing the starting roots such that the wedge mapped out between any two starting roots has the same portion of the mass of the overall structure. This generates a set of trees Z_i for robots $i = 1, \ldots, N$, where it is guaranteed that every node's path to its root is the shortest path to any root (Cormen et al., 1990). This can also be viewed as a single-bid auction with the modification of allowing bids on all lots simultaneously rather than sequentially auctioning each node. The lowest bid on any node wins that node, and then the auction is restarted. One advantage of this approach over a Voronoi based approach is that it yields a spanning tree for each subtask, which is useful in planning the assembly order within each task. Having this tree structure also allows us to maintain a guarantee of an open path back to the root by only building leaves of the remaining tree as will be seen in Phase III.

To ensure that no constraints in G_C are violated, there is a check each time a node is added that its ancestor set does not include any of its supports. If an addition violates this condition, the edge in G_S that allowed this connection is removed. Because one node of

this edge is already in this tree, the only effect of removing that edge is to prevent this assignment. It is important to note that this addition to Dijkstra's algorithm removes the guarantee that each node has the shortest path to a root node, and as such the solutions generated by this part of the algorithm will not retain the shortest path property. These situations are resolved by running an A* search in G_S from the current node back to the root of the tree attempting to claim it, throwing out any routes that would go through the current node's support. The restriction to the class of structures S_A guarantees that a route will be found. The algorithm then reassigns the parents of each node in the path to follow this new route, allowing us to claim the node in question. This phase of the algorithm is summarized in Algorithm 4.0.1.

Algorithm 4.0.1 Algorithm 1 Phase I

```
while there are unclaimed nodes do
  node ←closest node to any root
  if no constraint is violated then
    assign node to the closest root's associated tree.
  else
    run A* from node to the closest root
    if A* succeeds then
      assign node and change parentage of all nodes along the discovered path
    else
      sever the connection used to reach this node and continue
    end if
  end if
end while
```

After Phase I, each partition consists of a spanning tree describing one portion of the task. By choosing the root nodes of the Z_i's to be nodes along the outside of S, it is possible to guarantee completion of the structure without robots becoming stuck in a partially built structure via the following strategy. At each assembly step, robots are restricted to building one of the leaves of their current tree, and removing that leaf from their tree. While the

current outlined strategy guarantees completeness and correctness, the overall performance may still be poor since there is no guarantee that the resulting tasks will be similar in size, as shown in fig. 4.1. This figure shows a scenario where phase I has led to an unbalanced distribution of work, since the root node on the left is closer to most of the structure.

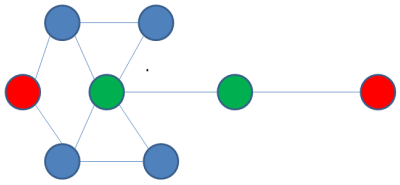


Figure 4.1: An example of phase I applied to a graph. The red nodes represent the roots of the two trees. One robot claims the green nodes, the other claims the blue. The green robot cannot claim any more nodes during phase I because it has distance 3 to all the blue nodes, while the blue robot has distance at most 2 to any of its nodes.

The main purpose of Phase II is to balance the distribution of work between the various trees via an exchange of nodes while ensuring no violations to the constraints specified by G_C. A weighted average of three criteria is used to evaluate potential trades. Those criteria are: 1) the total Euclidean distance along the path leading back to the root node; 2) the sizes of both the giving and receiving trees in relation to the average tree size; 3) the benefit of including both parts of a constraint within the same tree. The last criterion minimizes the number of delays due to robots pausing during construction to wait for others. The trade with the highest value is chosen at each step.

To determine which nodes to exchange between any two trees Z_i and Z_j, Algorithm 4.0.2 is used. This strategy alternates between having the smallest tree look for a node to take and having the largest tree look for a node to give away. In both cases, a set consisting of that node and all of its descendants is created to ensure that the tree structures for each robot are preserved. Thus, trades of individual leaves and trades of larger branches are both

considered. The process is repeated until no trade above a certain threshold value is found by either method, or until a hard limit on the number of trades is reached. This hard limit is included to guarantee termination of the algorithm, although in practice the number of trades was found to be significantly lower than the limits imposed.

Algorithm 4.0.2 Algorithm 1 Phase II

```
repeat
  tree ← smallest tree
  for i = 1 to N do
    score ← value of this tree taking node i
    if score < bestScore then
      bestScore ← score
      bestNode ← i
      newTree ← tree
      oldTree ← i.getTree()
    end if
  end for
  tree ← largest tree
  for i = 1 to N do
    score ← value of this tree giving away node i to the best candidate tree
    if score < bestScore then
      bestScore ← score
      bestNode ← i
      newTree ← tree chosen to receive node
      oldTree ← tree
    end if
  end for
  newTree takes bestNode from oldTree
until no trade is found
```

Once the task has been divided among the robots, all that remains is to specify how each robot should build its own subtask to best avoid delays resulting from split constraints. During construction, a path is guaranteed from every remaining node back to the root if the robot is restricted to building only leaves. As these leaves are built, they are removed from the structure, thus creating new leaves to be built. For a 2-D structure with robots smaller

than components, this represents the physical path of the robots, while in the experimental setup used in this work it represents the path of the end effector. To determine the order in which the leaves of a given tree should be assembled, a weighted average of three factors is used to give a value to each node. Priority will be given to nodes with lower values, though a node will not be considered for building until all of its child nodes and supports are built. The first is the distance from the center of the structure, to encourage robots to build outward from the center. This simplifies the collision avoidance problem as robots navigate to and from the cache and the next assembly location. The second criterion is based on the out degree of a node in G_C, counting only edges that cross a boundary between two tasks. This criterion gives preference for building nodes with a higher out degree in G_C early to provide as much time as possible for their higher number of supported nodes to be built. The final criterion is determined by the time at which a node's most recent support was built by another robot, ignoring the supports built by the robot itself. Maximizing this criterion avoids robots getting into a situation where they have to pause and wait for another robot to place a needed support.

Note that while the third criterion does change value during the construction, this criterion must settle to a final value before a node can be built, *i.e.* when the node's last support is placed. Therefore, rather than attempt to sort all nodes at the beginning of construction, the algorithm adds nodes to the queue as soon as they can be built, knowing that the evaluation of this node will no longer change after that point. In this implementation, a min-heap is maintained for each robot, which allows both adding a new element and removing the minimum element in $O(log\ m)$ time. The structure of a min-heap is a binary tree with a simple requirement that the value in each node be smaller than both of its children. The time taken to perform an addition to the heap or to remove the minimum element is primarily used to maintain this heap property. The value given to each node as it is added to the heap is a weighted average of the three criteria defined above.

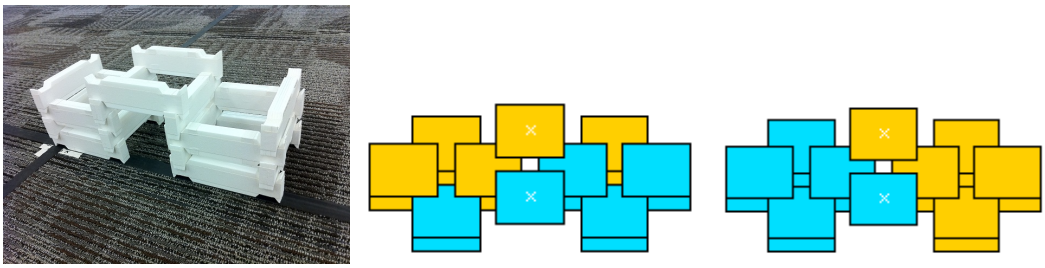
Each robot populates its initial heap with the set of leaves of its spanning tree except for any nodes that are supported nodes. Each robot begins its task by taking the minimum element of its heap, setting its current time based on the time that will be needed to build this node, and then finding all nodes that can now be built. The set of new buildable nodes generated whenever a node B is built will include the parent of B if it has no other unbuilt children, and any nodes supported by B that have no other unbuilt supports. Performing these checks is done by maintaining an array of the number of unbuilt children and number of unbuilt supports for each node. The construction sequence is generated by selecting at each step the robot R with the minimum current time, as that is the next robot to finish assembling an element. The algorithm then allows R to place the element at the root of its heap, adding all new nodes to be built to their respective heaps. This process is repeated until all robots have emptied their heaps, at which point construction is complete. This process is summarized in Algorithm 4.0.3.

Algorithm 4.0.3 Algorithm 1 Phase III

```
while at least one heap is not empty do
    heap ← heap from robot with minimum current time
    node ← heap.removeMinimum()
    if node is a support then
        decrement indices of supported nodes and add any that reach zero to their respective heaps
    end if
    if node is not the root then
        decrement childIndex of parent node and add that node to heap if it has no more children
    end if
end while
```

Next, two variations of this algorithm are considered to further improve its performance. To motivate the first change, consider the three-dimensional structure built out of Lincoln-log style blocks shown in Fig. 4.2(a). The structure has two towers connected by an over-

hanging bridge. The constraints on this problem are imposed by the nature of the materials used and the need to ensure static stability throughout construction. The constraints for this problem are that each node must be placed after the two nodes supporting it, thus a single node within the structure can be a part of four distinct constraints. The result of the initial algorithm is shown in Fig. 4.2(b). This is not the ideal allocation since there are more split constraints than necessary, which can cause delays during construction as one robot may need to wait for a placement by the other robot. In general, it is preferable to have the robots' sets of tasks as separated as possible to avoid these delays, such as in the plan shown in Fig. 4.2(c). The reason for the resulting allocation is because the initial allocation only considers the distance from the node to the root. Phase II is not enough to address this issue since there is no perceived gain to move a node once the tasks are the same size.



(a) (b) (c)

Figure 4.2: (a) The desired structure, composed of two towers with a connecting bridge. (b) Structure as built by Algorithm 1. Blue and orange represent the two robots, and the white x's represent their root locations. (c) The same structure as built by Algorithm 2. Note the decrease in potential conflicts between the robots.

To avoid these types of partitions, Algorithm 2 is proposed, which omits Phase II, and instead combines the node trading and the distance-based Dijkstra's algorithm by having the algorithm in Phase I use the full evaluation of a node's value introduced in Phase II of Algorithm 1. This means that for Algorithm 2, the Dijkstra's algorithm considers distance

to root, benefits from constraints, and the current size of the tree in forming the bid for each node from each tree. Now there is a situation where the bid for a specific node is fluctuating with every placement rather than only placements that provide a new route to that node. However, from this one gains the benefit of having the initial allocation attempting to avoid split constraints while also balancing the task sizes between the robots. Fig. 4.2(c) shows that this change leads to the desired result on the structure under consideration, where the two robots are able to minimize their interaction by working on separate towers until the time comes to build the bridge. One consequence of this change is increased dependence on the parameters chosen for the weighted average used to form the bids. That said, note that it is possible to recover the same result as Algorithm 1 if desired by simply setting the constraint and tree size weights to be very small relative to the weight applied to the distance from the node to the root.

The combination of Phases I and II into a single phase provides more foresight during Phase I, but does not preserve all the advantages of Phase II. This is because the node trading phase in Algorithm 1 is done after the initial allocation of Phase I and can correct for any discrepancies in size caused by unforeseeable irregularities in the shape of the structure. To address this, Algorithm 3 is introduced, which uses the improved initial allocation in Phase I of Algorithm 2, but still performs a separate Phase II to allow additional node trading. Comparing the results of Algorithms 2 and 3 allows us to determine if Phase II is redundant when Phase I is using the more complex criteria.

4.1 Analysis

In this section, the completeness and correctness properties of the partitioning strategy presented are considered. Correctness of the assembly sequence is guaranteed by the fact that each placement is made in accordance with the specified design, while completeness is addressed in Theorem 1.

Theorem 1 *Given $S \in S_A$, Algorithm 1, 2, or 3 will generate a complete plan in finite time.*

Proof 1 *The proof is composed of two parts. First, it will be shown that the trees generated in Phase I will include all nodes and then it will be shown that once the trees are generated, Phase III will generate a complete plan.*

The first part is proved by contradiction. Suppose that a node v in the target structure is not included in any of the trees generated by Phase I. There are only two cases in which this can occur: 1) no tree ever considers a path to v, or 2) A fails to find a path from v back to its associated root.*

If case 1 occurs, consider a path P that leads from v to any root r. Since a graph in S_A is strongly connected, such a path must exist. Along that path will be some point at which a node u not in any tree has a neighbor w in the tree associated with r. The Dijkstra's algorithm associated with r will have considered u when w was added, and this creates a contradiction.

Since A is guaranteed to find a path within a finite graph if one exists (Cormen et al., 1990), case 2 can only occur if there is no directed path in S_A from v to its associated root. Since a graph in S_A is strongly connected, such a path must exist, and again there is a contradiction. As such, v cannot be left out of the trees generated in Phase I. This concludes the first part of the proof. Further, since the node trading in Phase II passes nodes directly from one tree to another and thus cannot create an orphaned node. Since this analysis does not depend on the values of the edge weights, it is equally valid for all three algorithms.*

Again, the second part will be proved by contradiction. Suppose that there is a node v which does not get built. Since assembly continues until all heaps are empty, this means v was never added to a heap. The only way that can occur is if v never became buildable, which means either it still has unbuilt children, or it still has unbuilt supports. Since all

heaps are empty, this means that these children or supports were also not built, and consequently have their own unbuilt children or supports. Since any element of S_A is a finite structure, the only way this pattern can continue is if there is a cycle of parent-child and support relationships preventing each other from being built. Since a spanning tree cannot have a cycle of children, it must be the case that at least some of the relationships of this cycle must be supports. Since $S \in S_A$, G_C cannot have a cycle of supports, at least some of the relationships of this cycle must be parent-child. This means that there must exist a node which has a support in its ancestor set, which violates the criteria used in building the trees. Thus, since the trees from Phase I include all nodes, the assembly task will be completed in finite time.

5. Online Workload Balancing and Error Correction

This chapter introduces an online workload balancing strategy for the distributed assembly problem. The preplanning algorithm from the last chapter is modified by adding an online algorithm to enable robots to trade tasks for faster completion. A scanning robot equipped with a Microsoft Xbox Kinect is added to provide online error detection, so any missing parts can be detected and replaced.

The objective can then be stated as requiring the N_a assembly robots to build the desired structure S_d, which must be a member of S_A, without violating the constraint graph G_C. During this process, a balanced workload will be maintained via the online trading algorithm and any errors will be reported by the N_s scanning robots to be corrected by the assembly robots. All communications for a robot must be limited to the sets N_1 and N_2 defined for that robot.

5.1 Online Workload Balancing

This chapter extends the preplanning approach described above to include online workload balancing, accomplished by having each robot independently propose appropriate trades of tasks. For online trading to be added, it is necessary to consider what communications are allowed. For this purpose, two neighbor sets associated with each robot are defined. The first, N_1, is a static set of robots that the robot is allowed to make trades with. This is chosen to be the two robots with neighboring entrance nodes as defined by mapping the direction from the center to the entrance nodes onto a unit circle. The second, N_2, is a variable set encompassing all robots responsible for components adjacent to that robot's own components. This set N_2 is the set of robots that may place a support node that affects the current robot. The term *neighbors* will be used to describe the set N_1.

The preplanning approach generates a starting plan represented by a set of tree structures, one for each robot's subtask, with the restriction that the root node provides an exit from the structure. Each robot begins with full knowledge of the preplan, but may be unaware of changes made to that plan by other robots. The overall strategy used is to maintain accurate knowledge of N_1, which is the set of neighbors a robot is allowed to trade with. Outside this set, a robot has no knowledge of changes made to the plan unless they affect a robot in its N_1 set. These changes will be relayed by that robot, so no communication with robots outside N_2 is required. The only communications coming from outside N_1 are messages indicating that a support has been placed for an unbuilt node held by the robot, which necessarily comes from a robot in N_2. This class of messages could be replaced with a sensor capable of determining whether a specific support is present for a node that is otherwise ready to be built. By maintaining accurate knowledge of robots that can be trading partners, each robot is able to independently plan and propose beneficial trades. A robot will propose a trade whenever it is idle due to none of its task being currently buildable, or when its workload is below the average of its neighbors by at least twice the average build time for a node.

5.1.1 Node Trading Algorithm

The algorithm is based on Phase II of the preplanning algorithm, which is described in Chapter 4. Similar to that method, the online approach uses a weighted average of the distance from the root to the node being considered, the difference in task size between the giving and receiving robot, and whether taking the node being considered would put both parts of a constraint in the same task (which reduces the dependency between robots). By applying this formula to all neighbors of the current task, a robot can find the highest value trade it could currently make. For this application the algorithm has been revised to work in a distributed fashion, with each robot managing its own subtask and holding

as accurate a representation as possible of neighbors' subtasks. The method a robot uses to manage its own task is to maintain a min-heap structure containing all nodes that are ready to be built. For each node that has not yet been added to the heap, it maintains two variables describing the number of unbuilt children in the tree structure representation of the task, and the number of unbuilt supports (nodes that must be built before that node, generally because they provide an immediate physical support). After building a node itself or receiving a message from another robot building a node, the algorithm updates both of these variables accordingly. When both have a value of zero, indicating that the robot can build the node without blocking its access to other parts of the structure and all supports are in place, it adds that node to the heap. The heap priority is described by:

$$P = d_{ci} - c_{ki} + (w * t_{si}) \tag{5.1}$$

where P is the heap priority, d_{ci} is the distance from the center of the structure to node i, t_{si} is the timestamp the last support was built for node i by another robot, and w is a weighting factor. The variable c_{ki} represents the benefit from building this node in terms of how many nodes it supports. The t_{si} factor gives a higher score, which corresponds to a lower priority, to nodes with supports that were placed more recently. During this process, the robot frequently checks for incoming messages, and sends a message to all robots in set N_1 whenever it builds a node.

5.1.2 Communications Protocol

This section details the types of messages, when they are sent and how they are dealt with. Messages are passed with the following information: *destination robot, message type, source robot,* and *data*. The *destination robot* field is used as a tag for what robot should pick up this message, with two exceptions that will be discussed later. The *source robot* is the robot sending the message (necessary in order to acknowledge receipt of messages),

and the *data* field is of variable length depending on the type of message. The *message type* is interpreted as follows:

1. Sending robot built a node (data length 1)
2. Request to take a list of nodes (data length variable, first element gives length)
3. Answer to trade request (data length 1, yes or no)
4. Receipt of a new node along with the parent node it was attached to (data length 2, node and parent)
5. Sent a list of nodes to another robot (data length variable, specified by first element)
6. Built a node which acts as a support for a node held by receiving robot (data length 2, built node and supported node)
7. Acknowledgement of received message
8. Request for a resend of message type 1 if a certain node is built (data length 1)

Message types 2 through 5 are only sent to robots in the set N_1, while message types 1, 6, and 8 can also be sent to robots in the broader set N_2.

Throughout most of the task, these messages are aimed at a single robot. There are two exceptions which use a broadcast architecture, these are the coordination of the start and end of the experiment. In these cases special flags are used in place of the *destination robot* field, which indicate a different message structure then the norm. For the start case, there is no more information in the message, all robots start building as soon as they receive the message. For the end case, the rest of the message consists of a list of robots known to have finished. A robot done with its own task will only replace this message if they can add to its length (generally by adding their own id). When the list contains all IDs the robots stop. Coordination of finish times is necessary in order to be able to continue responding to

requests for build confirmations until all tasks have been completed. Both of these message types are relayed by each robot. This means that communication only needs to be possible with robots in N_2 in order for successful completion of an assembly task.

When a robot receives a type 1 (node built) message, it updates its representation of the sending robot's task, and checks whether the built node acts as a support for any of its own nodes. For type 2 (trade request), the robot checks that 1) it is the current owner of the requested nodes, 2) the nodes are not built, and 3) it is currently not building any of the requested nodes. If all of these conditions are satisfied, the robot responds with a message of type 3 saying 'yes', sends a message of type 5 to N_1 listing the nodes lost, and updates its representation of its own task. The robot, however, does not yet update its neighbor's task because it does not know where the new nodes will be attached. If the conditions are violated, it instead replies with a message of type 3 saying 'no'. For type 3 (answer to trade), if the answer is no, the robot cancels the proposed trade and returns to the main routine. If the answer is yes, the robot modifies its representations of both its own task and its neighbor's task, and then sends a message of type 4 to N_1, detailing where on the tree each of the new nodes is added. Upon receipt of a type 4 or type 5 message, robots modify their representation of the sending robot's task.

The last two message types usually come from robots in the set $N_2 \setminus (N_1 \cap N_2)$, and deal with supporting nodes. Message type 6 informs a robot that a support has been built for one of its nodes. When receiving this message, the robot first checks whether it still holds that node. If it has traded that node, the sending robot may not be aware of that, being outside N_1. If it has that node, it modifies its own representation and decrements the variable representing the number of unbuilt supports. If it does not, it forwards the message to whichever robot it traded the node to, which it knows because it is maintaining a full representation of robots in N_1. The other message is type 8, which is a request for confirmation that a certain node has been built. This message is sent when a robot has

an empty heap and can find nothing to take from its neighbors, but still has nodes with unbuilt supports. In this situation, a robot may have missed the message informing it that the support had been built and will request that message to be resent. When receiving a message of type 8, a robot checks whether the node being asked about was actually built. If so, the robot sends a message of type 1 to the requesting robot. If not, it does nothing.

Before building each node, a robot compares its own remaining task size with the average task size in its local neighborhood, which is defined as the set N_1 plus itself. If the robot's own task size is below the average by at least twice the average node build time, it searches among its representation of its neighbors for a valid trade, which it then requests. The other time it looks for a trade is when its own build heap is empty, indicating that it will be idle if it cannot find work to take from another robot. The benefit of looking for trades even when it still has its own work to do is that earlier trading allows more flexibility in what nodes are exchanged, because less of the structure has been built. Once the robot decides to look for a trade, it scores the possible trades according to the formula described above, considering relative task sizes, relative distances to the node in question, and the benefit a robot gets from holding both parts of a constraint itself. The purpose of this last criteria is to minimize the amount of cross-robot interaction. While the assembly robots are doing this, the scanning robots inspect the structure to discover missed placements, as described below.

5.2 Complexity

It is possible to determine bounds on the length of the experiment as well as the number of messages sent, which are provided in the following two theorems.

Theorem 2 *The length of the experiment will be $O(M * D)$, where M is the number of tiles and D is the maximum distance to the cache.*

Proof 2 *The only situation in which a robot will be idle while it has a non-empty task is if its remaining tasks have unsatisfied constraints. Because it was assumed that the constraint graph may not have cycles, there must be at least one unbuilt tile which does not have an unsatisfied constraint. Therefore, at least one robot, the robot possessing that tile, will still be working. The time to deliver a single tile will be twice the time taken to drive to the cache combined with a constant amount of time for pickup and placement. This time is* $O(D)$*. The worst case scenario would be a chain of constraints such that only one tile can be built at a time. In this case the time taken will be* $O(D)$ *times the number of tiles M, giving a total time of* $O(M * D)$*.*

Theorem 3 *With the exception of message type* 8 *and responses to it, the total number of messages sent will be* $O(M * N_a)$*.*

Proof 3 *Message types* 1 *and* 6 *will only be sent once per tile, except as a response to message type* 8*, and are thus directly bound by* $O(M)$*. Messages* 2 *through* 5 *are all bound by the number of trades. There are two cases in which tiles are traded, either when a robot is idle or when a robot has less work than its neighbors. If a robot is idle, it will immediately build any tile it receives in a trade, which prevents that tile from being retraded, since a robot cannot give away a tile being built. This limits the number of trades due to idle robots to* $O(M)$*, as each such trade results in a tile being built. A robot taking work from its neighbors cannot take more than* $M/2$ *tiles unless it is then giving those tiles away, since this would necessarily involve taking from a neighbor with fewer tiles. If it is giving those tiles away, this pattern can only be repeated* N_a *times before arriving back at the original robot. Since each successive robot has to have fewer tiles, it is not possible for this chain to repeat, as the first robot cannot have fewer tiles than itself. Therefore the total number of trades is* $O(M * N_a)$*, which limits message types* 2 *through* 5*. Message type* 7 *is sent once for each other message, and therefore increases the number of messages by a constant factor of two, which does not change the complexity. It is not possible to limit the number*

of messages of type 8*, as this will request message type* 1 *to be resent until it is successfully received.*

5.3 Online Error Correction

The VI-robot(s) is responsible for assigning the replacement of any missing tiles it discovers. It does this by managing an auction for each block that should have been placed but is absent. Each assembly robot sends a message to the VI-robot(s) after placing a tile. The VI-robot monitors these messages to maintain a state vector q, where q_i is 1 if the block has been placed and 0 otherwise. After each placement, the VI-robot reports a sensing vector q^s_j, where q^s_j is 1 if the block is definitely present, -1 if it is missing, and 0 if the presence or absence of the block cannot be determined. Then, if $q_i * q^s_j = -1$, a block that a robot claims to have placed is determined to be missing. Once the error has been detected, the scanning robot sends a message to inform the assembly robots that the block is missing and asks for bids to determine which robot will replace the missing block. Each robot then constructs a bid based on the following criteria:

$$b_i = w_i - A * c_i + B * d_{ij}, \tag{5.2}$$

where b_i is the bid of the i^{th} robot, w_i is the remaining workload of the i^{th} robot, c_i is the number of blocks still to be placed that are directly supported by the missing block, and d_{ij} is the distance between the missing block and the i^{th} robot's cache. The constants A and B are weights that can be optimized experimentally.

6. Ant Colony Optimization Methods

In this chapter Ant Colony Optimization methods will be applied to the problem. The preplanning Dijkstra-based method in Chapter 4 is deterministic, and only ever considers a single solution. In more complicated structures where the deterministic approach may not perform well, it would be useful to explore more of the potential solution space, for which ACO is a good candidate. The main difference between this approach and a standard ACO is that this approach will be dealing with a team of ants planning a candidate solution rather than a single ant constructing and scoring its own solution.

For this approach, it is assumed that there exist P teams of N ants, with each ant representing an assembly robot. Each edge (u, v) in E_{S_d} will be labeled by some amount of pheromone, which encourages an ant to claim u if it already has v, or vice versa. The more pheromone is present on that edge, the more likely u and v are to be assigned to the same task.

The set of starting nodes is chosen in two ways. For smaller structures, the set of starting nodes is determined by computing the angular density of the structure about the center of mass, and then spacing the starting nodes such that the wedge mapped out between any two starting nodes has the same portion of the mass of the overall structure, subject to the restriction that starting nodes must be on the exterior of the top surface of the structure. For the larger structure in simulation, starting nodes were chosen manually to test the effects of the distribution of starting nodes. The same set of starting nodes is then provided to all team managers so that pheromone can be meaningfully shared between teams, allowing them to learn from each others' experiences. The algorithm is then run for a prespecified number of generations. In each generation, the ants within a team will collectively divide the set of all tasks according to the pheromone present, then score their proposed solution according to a metric based on workload variance, deposit their own pheromone in proportion to the

quality of their solution, and finally reset their states in preparation for the next generation. This is shown in Alg. 6.0.1.

Algorithm 6.0.1 Algorithm Overview

```
0: choose starting nodes
  for i = 1 to number of generations do
    for j = 1 to P do
      Team j plans a task allocation (Alg. 6.1.1)
      Team j scores its solution
    end for
    for j = 1 to P do
      Team j deposits pheromone
      Team j resets its plan to hold only starting nodes
    end for
  end for
```

6.1 Baseline Strategy

The baseline strategy will be presented first, and will be labeled as DAACO (Distributed Assembly by Ant Colony Optimization). In each generation, the existing pheromone is used to plan a decomposition of the structure into a subset of tasks for each ant. This is achieved by each ant sequentially claiming a single node, continuing until no unclaimed nodes remain. An individual ant makes its decision on which node to claim based on summing the total amount of pheromone leading to each potential target node from all nodes currently part of its task set. That is, for each target node j, it computes a probability of taking that node according to equation 6.1:

$$p_j = ((\sum_i x_{ij}) + p_{min}) / \sum_k ((\sum_i x_{ik}) + p_{min}) \tag{6.1}$$

where p_j is the probability of claiming node j, x_{ij} is the pheromone on the edge between i and j, and p_{min} is a constant that provides a small chance of claiming each node, even in the absence of pheromone. The sums only consider nodes for j that are adjacent to the ant's current task set, and only consider nodes for i that are within the ant's current task set. By considering all edges leading to the target node, it becomes more likely that an ant will claim new tasks that share multiple adjacencies with the current task set, leading to a more compact overall task set for the ant.

The purpose of p_{min} is to introduce a possibility of exploring previously untried assignments. Effectively this provides a noise term to the exploration of the search space, giving a possibility of exiting a local minimum. The pheromone is globally initialized to a value of zero, meaning that initial decisions are chosen from a uniform distribution over all adjacent nodes. To increase solution diversity, an ordering of the ants is randomly generated by the team manager for each cycle (each ant claims at most one node during a cycle). The manager also determines when all nodes have been claimed, at which point the solution is scored. This process is summarized in Alg. 6.1.1.

Algorithm 6.1.1 Planning for a team

```
while There are unclaimed nodes do
    order = randomized permutation of ants within team
    for i = order do
        for k = neighbors of current task set do
            scores(k) = ∑ pheromone between k and current task set
        end for
        choose a node with probability proportional to scores(k) + p_min
        if Ant i found a claimable node then
            mark that node as claimed
        end if
    end for
end while
```

Between the planning and scoring phases, the pheromone graph is subjected to a global decay, removing a set fraction of the old pheromone each generation. This has the effect of attaching more importance to more recent generations. The reason for the timing between planning and scoring is to allow pheromone deposited in generation i to be used in generation $i+1$ before it is subjected to decay. This means it is possible to increase the decay to a point where all pheromone is erased before new pheromone is deposited, making each generation dependent only on the results of the immediately preceding generation. A decay rate of d means $(d*100)\%$ of the pheromone is removed at this step.

A metric based on the variance of the workload of each ant is employed by the team manager to score the solution. The metric is given by Eq. 6.2. This metric will provide a score scaled between 0 and 1, with a higher score indicating a better performance. This score is based on the entire team's performance, and is then passed to each individual ant to be used in depositing pheromone.

$$score = 1/(1+var(WL)) \tag{6.2}$$

After the team's manager has computed a team score, each ant deposits an amount of pheromone equal to the score to each edge connecting a pair of nodes within the ant's task set. Finally, at the end of each generation every ant is reset to contain only its assigned start node before the next generation begins.

After all generations are concluded, a solution is extracted from the population of ants by having one team run one last generation with a slight difference. Rather than stochastically choosing nodes according to the probabilities described above, each ant selects a node by deterministically selecting the node that would have the maximum probability, described by equation 6.3, where c is the choice made. This allows us to extract a solution that from the best knowledge the pheromone represents without adding the stochasticity

found in a typical generation.

$$c = argmax_j(\sum_i x_{ij}) \tag{6.3}$$

6.2 Variant Strategies

Several variants of the strategy are presented in this section:

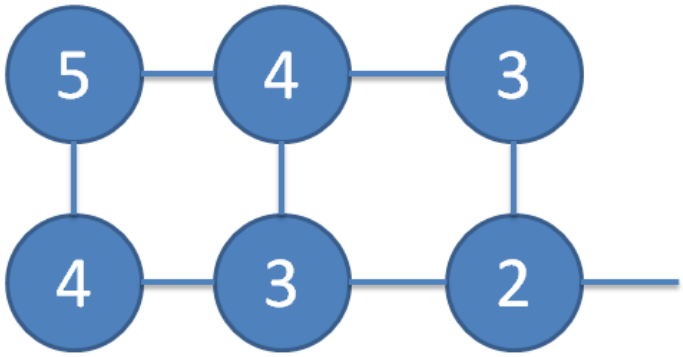
1. Directional DAACO (DAACO-D)
2. DAACO with stealing (DAACO-S)
3. DAACO with stealing and restrictive contiguity fix (DAACO-SCI)
4. DAACO with stealing and less restrictive contiguity fix (DAACO-SCII)

The first variant, called DAACO-D, provides an alternative way of depositing pheromone. As the baseline strategy deposits pheromone on all connections within the current task set, it does not designate a direction, so the pheromone graph is undirected. This means that an ant claiming a node on either side of an edge with a high value is likely to take the node on the other side. Since this may not be desirable behavior for nodes near one of the starting points, DAACO-D uses a directed pheromone graph. To determine where to deposit pheromone on this directed graph, Dijkstra's algorithm is run on an ant's complete task assignment to determine a distance back to the start for every node. Pheromone is then only deposited on edges leading from a lower distance to a higher distance.

The second variant, DAACO-S, allows ants to claim nodes that have already been taken by another ant, in order to more quickly reach an equal workload by not forcing an ant to stop taking nodes in a situation where all adjacent nodes have been claimed. In order to avoid ants repeatedly trading nodes back and forth, three restrictions are placed on this behavior. First, an ant will only consider stealing a node if there are no unclaimed nodes adjacent to its current task set. Second, an ant will only steal nodes if its current workload

is below the target workload (based on an equal distribution of the total). Finally, ant i will only steal a node from ant j if j has at least as much work as i. In order to encourage fewer stolen nodes in successive generations, an ant that loses a node will remove a fraction of the pheromone connecting it to that node.

When a node k is stolen, neighbors of k in the same task set can become orphans, with no path back to the root of the task set. This set of orphaned nodes will be called Q. To ensure that Q remains empty, two methods of ensuring task set contiguity for DAACO-S are proposed. Both are dependent on keeping track of the distance to the root node for each node in the task set (with each edge being assigned a distance of 1). The more restrictive method, DAACO-SCI, requires all neighbors of the stolen node within the same task set to have a lower distance to root, which ensures that removing that node cannot disconnect any neighbors. This is compared with a less restrictive method, DAACO-SCII, that checks each neighbor for alternate paths back to the root that do not use the removed node. This can take O(n*b) time, where n is the number of nodes and b is the branching factor. Since the normal process of claiming a node takes O(n) time to find neighbors of the existing task set, the time taken to do this contiguity check before stealing a node is not much different than the time taken to claim a free node, as long as the branching factor is low. For the experiments in this work, the branching factor does not exceed six. These methods are compared in Fig. 6.1(a). Note that in this example DAACO-SCI would be able to steal only the node with distance 5, since all others have a neighbor with a higher distance. In contrast, DAACO-SCII would be able to steal any node except the one with distance 2, since alternate paths back to the root can be found after any other node is removed.



(a)

Figure 6.1: An example task allocation, with each node labeled by its distance from the root.

7. Towards Cooperative Manipulation for Distributed Assembly

In this chapter we consider how to apply these methods to more complicated structures where partial assemblies of the structure are not guaranteed to be stable. This approach is similar in form to the initial Dijkstra-based method, but will require new representations for many of the variables to accommodate the new class of structures. First, the structure graph G_S will be changed for this chapter so that a set of nodes representing cache locations will be added, one for each robot. E_S will now be changed to represent distances back to the cache, so that an edge (u,v) represents the distance from cache node u to assembly node v, and no edge exists if neither u nor v is a cache node. This changes the emphasis from having contiguous tasks to focusing on distance back to the cache. The reason for this change is that contiguous tasks become impossible for archlike structures when robots will need to alternate placements to keep the structure stable.

In this chapter, the focus is on structures that require coordinated timing between robots, and could not be built by a single robot. In particular, arch-like structures where placements are not guaranteed to be stable when made, and must therefore be held in place until some later time when they become stable. Tthe first node will be called an *unstable node*, and the node which restores stability to the unstable node will be called a *stabilizing node*. It is assumed that every unstable node in a structure will directly support its stabilizing node. That is, when the next higher piece in the arch is placed, the lower node becomes stable. A structure S that fits this criteria will be called an *arch-like structure*. To approach this problem, the precedence constraints described above are replaced with a type of constraint that prohibits taking both of a pair of nodes u,v. So a robot that has claimed node u cannot claim v and vice versa. For a desired S, define a *constraint graph* as an undirected graph $G_C = \{V_C, E_C\}$ such that $V_C = V_S$ and E_C is defined as follows. An edge will be added to G_C in two situations. First, every unstable node is given an edge with its stabilizing node, so the

same robot cannot claim both. Second, for any node u which is a stabilizing node for more than one unstable node, edges are created between every pair of unstable nodes associated with u. Next, a directed graph called the *support graph* is created, with $G_U = \{V_U, E_U\}$ such that $V_U = V_S$ and E_U is given an edge from every node directly supporting another to the node it supports. This will be needed during assembly to verify when a node is ready for placement. The final graph is the *arch graph*, $G_A = \{V_A, E_A\}$, which denotes each pair of nodes with an archlike relationship. $V_A = V_S$ and E_A contains a directed edge from each node that must be held in place to any nodes that must be placed before it can be released. The node needing to be placed will be called an *arch support* of the unstable node being held.

The approach for the partitioning of S consists of three phases. Phase I of the algorithm begins by running a simultaneous Dijkstra's algorithm for each robot. Different from the initial method presented in Chapter 4, all edges lead to one of the cache locations, so the output focuses on minimizing travel distance for the robots rather than encouraging contiguous tasks. This can also be viewed as a single-bid auction with the modification of allowing bids on all lots simultaneously rather than sequentially auctioning each node. The lowest bid on any node wins that node, and then the auction is restarted. Bids are a weighted combination of a robot's current workload plus the distance to the node, defined by Equation 7.1. The winning bid for each round is then selected according to Equation 7.2. To ensure that no constraints in G_C are violated, there is a check for each bid that its current task does not include anything with a constraint on the node under consideration. If it does, that bid is set to infinity. This is shown in Algorithm 7.0.1.

$$B_{ij} = W_i + E_S(i, j) \tag{7.1}$$

$$B_{winner} = \underset{ij}{\operatorname{argmin}} B_{ij} \tag{7.2}$$

Algorithm 7.0.1 Phase I

```
while there are unclaimed nodes do
  for each robot and unclaimed node do
    if no constraint is violated then
      bid_ij ← W_i + E_S(i, j)
    else
      bid_ij ← inf
    end if
  end for
  node ←minimum bid for any robot
end while
```

The main purpose of Phase II is to balance the distribution of work between the various trees via an exchange of nodes while ensuring no violations to the constraints specified by G_C. Two criteria are considered when trading nodes: 1) the Euclidean distance back to that robot's cache; 2) the sizes of both the giving and receiving trees in relation to the average tree size. A weighted average of these criteria is used to evaluate any potential trade between two robots.

To determine which nodes to exchange between any two trees Z_i and Z_j, Algorithm 4.0.2 is still used, with no changes needed to this part.

Once the task has been divided among the robots, each robot is assigned its task and the experiment begins. Robots will prioritize their workload and continue to exchange workload online as described in the following sections.

7.1 Task Management

The online algorithm is based on Phase II of the preplanning algorithm, as described above, with the difference that this is evaluated during the course of construction. For this application the algorithm has been revised to work in a distributed fashion, with each robot managing its own subtask and holding as accurate a representation as possible of neighbors' subtasks. The method a robot uses to manage its own task is to maintain a min-heap

structure containing all nodes that are ready to be built. For each node that has not yet been added to the heap, it maintains a variable describing the number of unbuilt supports based on the support graph G_U (nodes that must be built before that node, generally because they provide an immediate physical support). After building a node itself or receiving a message from another robot building a node, the algorithm updates this variable accordingly. When it has a value of zero, indicating that all supports are in place, it adds that node to the heap. The heap priority is described by:

$$P = d_{ci} - c_{ki} + (w * t_{si}) \tag{7.3}$$

where P is the heap priority, d_{ci} is the distance from the center of the structure to node i, t_{si} is the timestamp the last support was built for node i by another robot, and w is a weighting factor. The variable c_{ki} represents the benefit from building this node in terms of how many nodes it supports. The t_{si} factor gives a higher score, which corresponds to a lower priority, to nodes with supports that were placed more recently. During this process, the robot frequently checks for incoming messages, and sends a message to all robots in set N_1 whenever it builds a node.

After a node is placed but before it is released, the robot checks for the presence of archlike constraints in the arch graph G_A. If any corresponding to the current node are found, it holds that node in place until the necessary node(s) are placed. This creates a possible deadlock situation where all robots are stuck holding an unstable node. This will be addressed in section 7.4.

7.2 Minimum Number of Robots

In Chapters 4 and 5, it was possible for one robot to build the structures, and the advantage to using multiple robots was faster completion of the structure. In this work, the

arch-like structures being considered cannot be built by a single robot, as there will be pieces that must be held in place while an arch is completed. This raises the question of how many robots are needed for any particular structure. It turns out that this number can be determined from looking at the constraint graph G_C.

Theorem 4 *The minimum number of robots that can build a structure is at most one greater than the degree of G_C.*

Proof 4 *This problem can be represented as a graph coloring problem on the constraint graph G_C. The graph coloring problem asks how many colors are needed to color every vertex of a graph such that no two neighboring vertices have the same color. The constraint graph has exactly this meaning, that no robot may hold two neighboring nodes. So, by representing each robot as a color, the answer to the graph coloring problem on G_C will give us the minimum number of robots needed to "color" the graph without violating any constraints. Brooks' theorem Brooks (1941) states that for any connected graph, the number of colors (robots) needed is at most one greater than the maximum degree of the graph. If the graph is not a complete graph or a cycle graph of odd length, the number of colors needed is equal to the maximum degree of the graph. As one cannot rule out these cases a priori, the result is that the minimum number of robots needed to build a structure with constraint graph G_C is at most one greater than the degree of G_C.*

7.3 Communications Protocol

This section details the types of messages, when they are sent and how they are dealt with. Some of these are the same as in Chapter 5, but several are new and some of the messages from that chapter are not used here. Messages are still passed with the following information: *destination robot, message type, source robot,* and *data*. The *destination robot* field is used as a tag for what robot should pick up this message, with two exceptions that

will be discussed later. The *source robot* is the robot sending the message (necessary in order to acknowledge receipt of messages), and the *data* field is of variable length depending on the type of message. The *message type* is interpreted as follows:

1 Sending robot built a node (data length 1).

4 Receipt of a new node along with the parent node it was attached to (data length 2, node and parent).

6 Built a node which acts as a support for a node held by receiving robot (data length 2, built node and supported node).

7 Acknowledgement of received message.

8 Request for a resend of message type 1 if a certain node is built (data length 1).

9 A node has disputed ownership, this message tells another robot to claim it.

11 Used to convey robot's availability to build.

14 Request to either build or reassign a node needed as an arch support.

15 Order to build a node, given by the robot which had ownership of the node to a robot that is currently idle.

Message types 1, 4, 6, 7, 8, and 9 are the same as they were described in Chapter 5.

Message type 11 is used to communicate robot availability throughout the task. This message is sent to all robots in N_2, which is the set of robots that may assign work back to this robot. A busy robot will send either 0 or a positive number, which are used for a bidding scheme in the event that all robots become stuck, to determine which robot will abandon its task and return to the cache. In the experiments conducted this situation does not arise. A value of -2 indicates the robot is available but has a non-empty heap, while a

value of -3 indicates an available robot with an empty heap. The next subsection explains the importance of this message type.

Message type 14 is used to state that a robot is requesting a certain node to be built, as an arch support for a node it is currently holding. When this happens, a message of type 15 is sent to an available robot to build that node. If no robot is available, the message will be sent as soon as one becomes available.

7.4 Avoiding deadlock

It was found that the best way to avoid deadlock situations where all robots are stuck waiting for arch supports was to keep around half the robots in reserve at all times, waiting for messages of type 15. This is done by checking the number of available robots in N_2 before any build determined by a robot's build heap. If too few robots are available, a robot will choose to wait rather than building something from its heap. Any available robot will always respond to a type 15 message, regardless of how many robots will still be available. This has the effect of concentrating the robots on a few parts of the structure at a time, so that arch supports will be quickly built and robots will spend less time holding an unstable node. Robots sending a value of -3 are chosen before those sending -2, so that if too many robots become available, the idle robots are more likely to find work in their heap.

8. Simulations

This chapter will present the implementation of results of each of the approaches discussed. Results for each approach will be followed with a discussion of significant findings.

8.1 Dijkstra-Based Methods

To evaluate the partitioning strategy presented in Chapter 4, this section will consider the partitioning of various 2- and 3-D structures in simulation. Nodes are defined by their global coordinates in $\mathscr{W}$, their mass, and the time required to navigate from the node to the nearest cache. In these simulations, it is assumed that the distance to the cache is large relative to intra-structure distances. As such, all nodes were assigned a cache distance of one, and for all but one of the cases, this was equal to construction time per node. Time required to assemble a node is given by the construction plus travel times to and from the cache.

All three algorithms were used to partition nine different structures. These are shown in Fig. 8.1. Recall that Algorithm 1 is the baseline strategy, Algorithm 2 combines Phases I and II by incoporating the additional criteria from Phase II into the initial Dijkstra's Algorithm, while Algorithm 3 uses this combined phase, but still keeps a separate Phase II to balance workload. For the structure in Fig. 8.1(c), the top four rows consist of parts that take 20 times longer to assemble than the rest. For the ones in Fig. 8.1(e) and 8.1(f) each element in the boxed column must be built before the element to its right. In Fig. 8.1(h), each boxed component must be built before its inward neighbor. The 8^{th} structure is shown in Fig. 4.2(a). The parameter values used to generate node scores for these simulations were 1 for distance to root, 1 for size of tree, and 10 for constraints kept together.

8.1.1 Results

Table 8.1 summarizes the results. Completion Time of the structure is set to the time when the final structure component is placed. Max. Difference is the difference in time between the first and last robot to finish. Ave. Wait Time measures the average amount of time robots were forced to wait during construction for a support placed by another robot. Split Constraints refer to the number of constraints divided between two different robots, *e.g.*, one robot builds the support for a part placed by another robot. In general, the more split constraints, the more likely it is that robots will have to wait for each other.

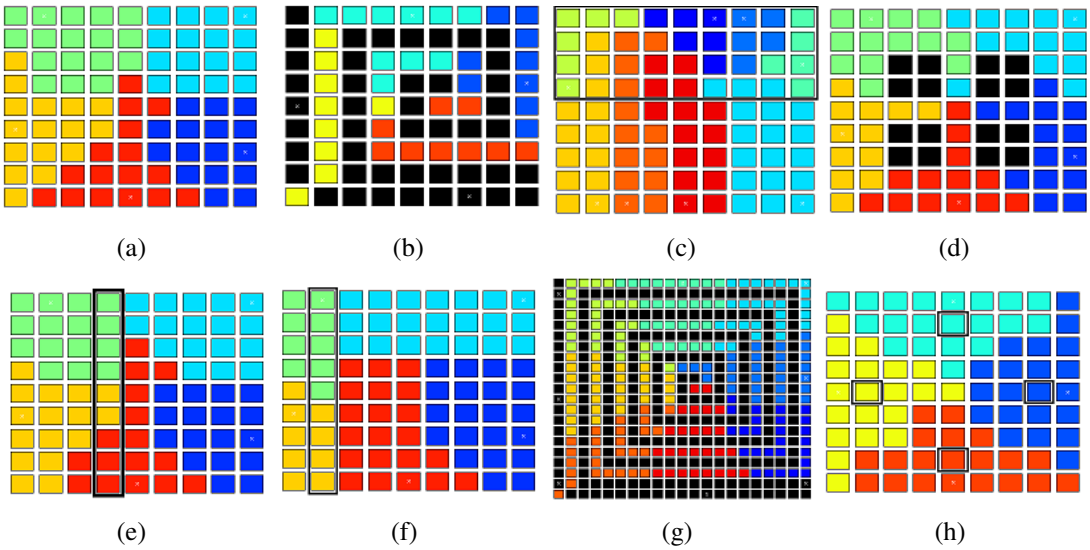


Figure 8.1: (a-g) Structures 1-7 and (h) Structure 9 partitioned by Algorithm 3. Black denotes empty space, while different colors represent different robots.

Table 8.1: Simulation results

Criterion	Algorithm	Structure Number								
		1	2	3	4	5	6	7	8	9
Completion Time	Algorithm 1	72	44	207	60	72	88	112	40	88
	Algorithm 2	72	44	170	60	76	96	116	40	88
	Algorithm 3	72	44	161	60	72	88	112	40	88
Max. Difference	Algorithm 1	4	8	171	12	8	48	8	0	4
	Algorithm 2	8	8	42	8	16	64	20	0	4
	Algorithm 3	4	8	29	8	8	48	4	0	4
Ave. Wait Time	Algorithm 1	0	0	0	0	0	0	0	0	0
	Algorithm 2	0	0	0	0	0	0	0	0	0
	Algorithm 3	0	0	0	0	0	0	0	0	0
Split Constraints	Algorithm 1	0	0	0	0	6	9	0	14	0
	Algorithm 2	0	0	0	0	5	8	0	2	0
	Algorithm 3	0	0	0	0	6	9	0	2	0

8.1.2 Discussion

Beginning with the Dijkstra-based algorithms, refer first to Table 8.1, where it can be seen that in all cases Algorithm 3 does at least as well as the other two by the time criteria, and in some cases is better. This is particularly true for case 3 which is the only case with heterogeneous assembly materials. While the difference in Max. Difference values between the three algorithms is harder to evaluate across the various cases, note that Algorithm 3 generally provides more equal distribution of work.

First, note that the sequencing approach in Phase III is sufficient to prevent robots from needing to wait for each other in these trials. This is because the approach is partially based on early placement of supports. Next, Algorithm 2 consistently results in minimizing the number of split constraints. Consider structure 8 where Algorithm 1 returns a solution with 14 split constraints, while Algorithms 2 and 3 both provide a solution with only 2 split constraints while yielding the same time performance. For structures 5 and 6, there appears to be a trade-off where allowing an additional split constraint yields a better time performance. Since Phase I is identical for Algorithms 2 and 3, this suggest that Algorithm

3 must be making a trade in Phase II that results in an additional split constraint to achieve a more equal distribution of work. Finally, the scalability of the approach with respect to the structure and team sizes is shown in Fig. 8.1(g) which shows the partitioning of a 230-node structure for a team of 8 robots.

8.2 Online Workload Balancing

Next, the node trading algorithm from Chapter 5 was implemented on a network of seven computers communicating over a wireless router, each simulating the activities of a single robot. Building a node was simulated by subjecting the robot to a delay randomly drawn from $\mathcal{N}(4,8)$, truncated at 0. In order to test the system's robustness to high levels of variability, a high standard deviation relative to the mean was used. The discussion will analyze two trends in the scaling of the problem. The first considers all seven simulated robots cooperatively building a structure that varies in size from 27 up to 512 nodes. The second examines a variable number of robots applied to the same structure of 512 nodes. In both cases, the primary focus is on how the completion time and the number of messages sent scale with the structure size and the number of robots.

The structures used for the experiments described above were cubes, built out of cubic pieces. The purpose of using a uniform structure is to isolate the effects of changes in structure size and number of robots. Table 8.2 provides results of three specific experiments to demonstrate the flexibility of the approach. Example 11 corresponds to the structure shown in Fig. 8.2, which is one of the structures the experimental testbed used in this work can build, built here by 4 robots. This demonstrates the approach's ability to handle heterogeneous building materials in a variety of configurations. Example 12 and 13 are each based on a cube of 512 nodes built with 7 robots. In Example 12, one of the robots is deliberately started with only a single node to force the approach to recover from a lopsided workload distribution. Example 13 has the added constraint that each node with

(a) (b)

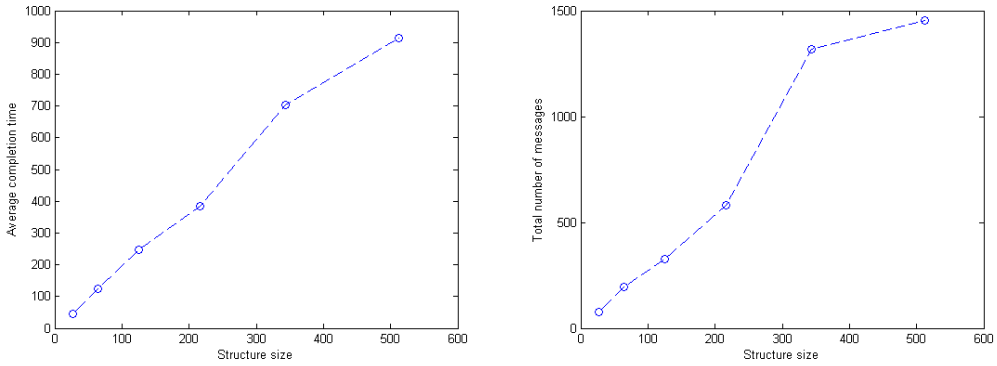
Figure 8.2: The structure used as example 11

a y coordinate equal to 4 must be built before the adjacent node with a y coordinate equal to 5. This is to demonstrate the ability to manage types of constraints other than those imposed by gravity.

8.2.1 Results

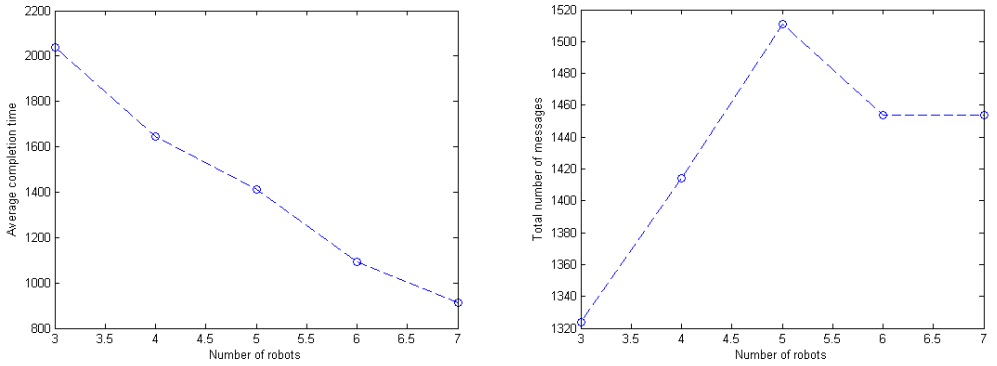
Table 8.2: Examples demonstrating flexibility

Criterion	Ex. 11	Ex. 12	Ex. 13
Structure size	43	512	512
# of robots	4	7	7
Ave. # of nodes per task	10.8	73.1	73.1
St. Dev. of # of nodes per task	3.30	35.4	37.6
Total # of nodes traded	15	55	15
Ave. wait time during construction	192.3	17.5	33.3
Ave. wait time after construction	11.8	236.4	775.1
Ave. completion time	302.7	1063.0	992.9
# of messages sent	508	1749	1615



(a) (b)

Figure 8.3: Graphs showing relatively linear growth in both completion time and number of messages with respect to structure size. Data listed in table 8.3.



(a) (b)

Figure 8.4: Graphs showing number of messages and completion time relative to number of robots. As the size of the team increases, the completion time drops linearly while the number of messages shows slight growth. Data listed in table 8.4.

Tables 8.3 and 8.4 summarize the following quantities for each experiment: structure size, mean and standard deviation for number of nodes built per robot, total number of nodes traded, average time spent waiting during construction, average time waiting after construction (waiting for other robots to finish), average completion time (not including time spent waiting for other robots to finish, but including any wait time during construction), number of messages sent, and number of messages that were not acknowledged (giving an idea of how many messages are actually being received).

Table 8.3: Variable size cubes built with seven robots

Criterion	Ex. 1	Ex. 2	Ex. 3	Ex. 4	Ex. 5	Ex. 6
Structure Size	27	64	125	216	343	512
Ave. # of nodes	3.9	9.1	17.9	30.9	49.0	73.1
St. Dev. of # of nodes	0.38	2.19	0.690	1.57	3.00	2.19
# of nodes traded	0	3	0	3	23	11
Ave. wait time during construction	3.0	4.5	2.4	2.5	92.7	22.7
Ave. wait time after construction	22.8	25.6	51.7	37.2	55.0	33.7
Ave. completion time	44.0	123.7	246.8	383.6	704.3	912.6
# of messages sent	76	193	325	579	1316	1454

8.2.2 Discussion

Next, this section will turn to the online workload balancing approach of Chapter 5, examining how communication and completion time scale with structure size and number of robots. Fig. 8.3 shows the approach being run with 7 robots on a variety of structure sizes. Examining the number of messages sent, it can be seen that the communication volume increases linearly with the size of the structure. Average finish time is also increasing

Table 8.4: Variable numbers of robots on a structure with 512 nodes

Criterion	Ex. 7	Ex. 8	Ex. 9	Ex. 10	Ex. 6
# of robots	3	4	5	6	7
Ave. # of nodes	170.7	128	102.4	85.3	73.1
St. Dev. of # of nodes	2.08	5.48	3.29	1.03	2.19
# of nodes traded	6	9	13	7	11
Ave. wait time during construction	1.1	3.9	60.8	38.1	22.7
Ave. wait time after construction	56.1	31.6	98.0	156.6	33.7
Ave. completion time	2038.2	1647	1413.7	1092.7	912.6
# of messages sent	1324	1414	1511	1454	1454

linearly with the size of the structure. This means that the rate at which the robots build is unaffected by the size of the structure. Also, it is worth noting that the amount of messages per time (number of messages divided by average completion time) is a constant relative to structure size, meaning there is no communication barrier to scaling this approach to larger structures.

Next, Fig. 8.4 shows how the approach scales with number of robots on a constant structure. First, the number of messages sent stays fairly constant across different numbers of robots, showing only a slight growth, so adding more robots does not cause any problem in communications. For the structure used, multiplying the number of robots used by the average completion time gives a relatively constant result. This means that adding an extra robot does not introduce significantly extra inefficiency in terms of idle robots waiting for placements. Certainly at some number of robots, this would stop being the case, but up through seven robots there has not yet been significant additional idling. For this approach, a hard limit on the number of robots that can be used is the number of valid root nodes (recall that the root needs to be viable as the last node placed on the structure). For the structure used, this would allow up to 64 robots to be applied to the problem.

Example 12 started one robot with a single tile to test the robustness of the algorithm to lopsided initial conditions. As shown in Table 8.2, this resulted in a large number of nodes

being traded,which is a result of nodes being moved towards the robot that started with a single node. This robot ended the experiment having built 25 nodes, roughly a third of the average size. This also resulted in an unusually high variance in number of nodes built during this experiment. Example 13 had a wall of constraints requiring that each node with a y coordinate of 4 be built before the corresponding node with a y coordinate of 5. This resulted in a bimodal distribution for number of robots built, with two robots building 128 nodes each and the other 5 all in the range 45-55. This is likely caused by an inability to trade across the wall of constraints imposed, which effectively divided the robots into two separate teams based on the locations of their roots.

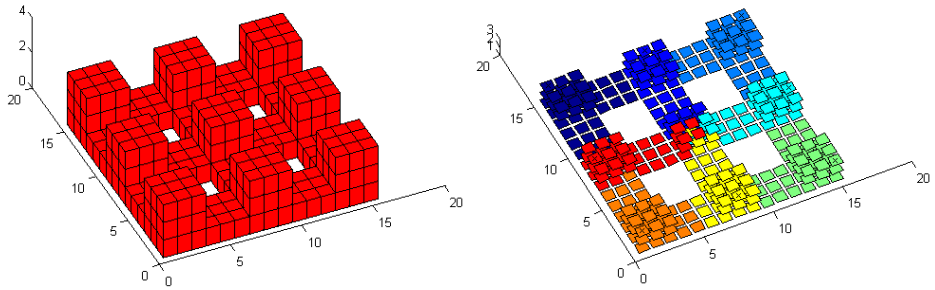
8.3 Ant Colony Optimization

Finally, the DAACO algorithms from Chapter 6 is tested on types of structures consisting of towers connected by paths, with holes between the paths where nothing is to be built, as well as a larger house structure. Both the number of robots dividing the structure and the decay rate of the pheromone is varied. Each map is also compared against the results of the algorithm described in Chapter 4 run on the same map. In this algorithm, the assembly task is initially divided by running Dijkstra's algorithm with multiple starting nodes, one for each robot. This leaves each robot with a tree representation of its task, where the root (the starting node) is guaranteed to be on the external boundary of the structure. By successively building leaves and removing built nodes from its tree, a robot can complete its task without the danger of becoming trapped in a partially built structure. The algorithm then goes through a node-trading phase which attempts to equalize the workload by exchanging leaves or branches while maintaining the tree property of each task. This is a deterministic algorithm, and hence generates only one solution for a given assembly task, in contrast to DAACO and its variants, which explore the solution space by varying pheromone levels.

Results are compared using the standard deviation of the workload between different

robots, with the goal being that this should be minimized in an effective plan. A more balanced workload should lead to faster completion of the structure as no robots will finish early and be left idle. To handle unexpected delays that may occur during construction, one could incorporate the online workload balancing approach discussed above after using ACO to create the initial plan. The base structure consists of 9 3x3 towers, built out of 351 pieces, shown in Fig. 8.5(a). Each reported result for DAACO and its variants is the average of five runs, with each run lasting 20 generations. If parameters are not explicitly stated, the experiment is done with 8 robots, using a decay rate of 0.1, and a node mass of 1 for each node.

8.3.1 Results

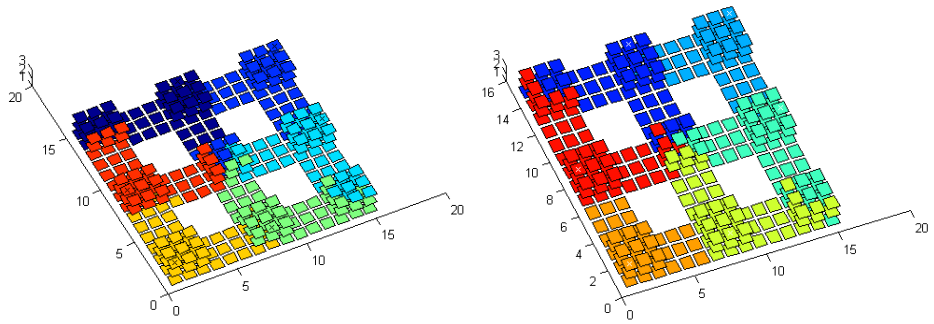


(a) (b)

Figure 8.5: (a) The base structure, consisting of nine towers connected by ground paths. (b) A typical decomposition, in this case by DAACO-S using 8 robots. Different colors represent the task sets of different robots.

First, the effects of using different numbers of robots to build this structure will be analyzed. For each number, DAACO, DAACO-S, DAACO-SCI, DAACO-SCII, DAACO-D, and the Dijkstra-based algorithm from Chapter 4 are applied. Results are shown in Figure 8.7(a) and Table 8.5. Each entry is the standard deviation of the workloads of the

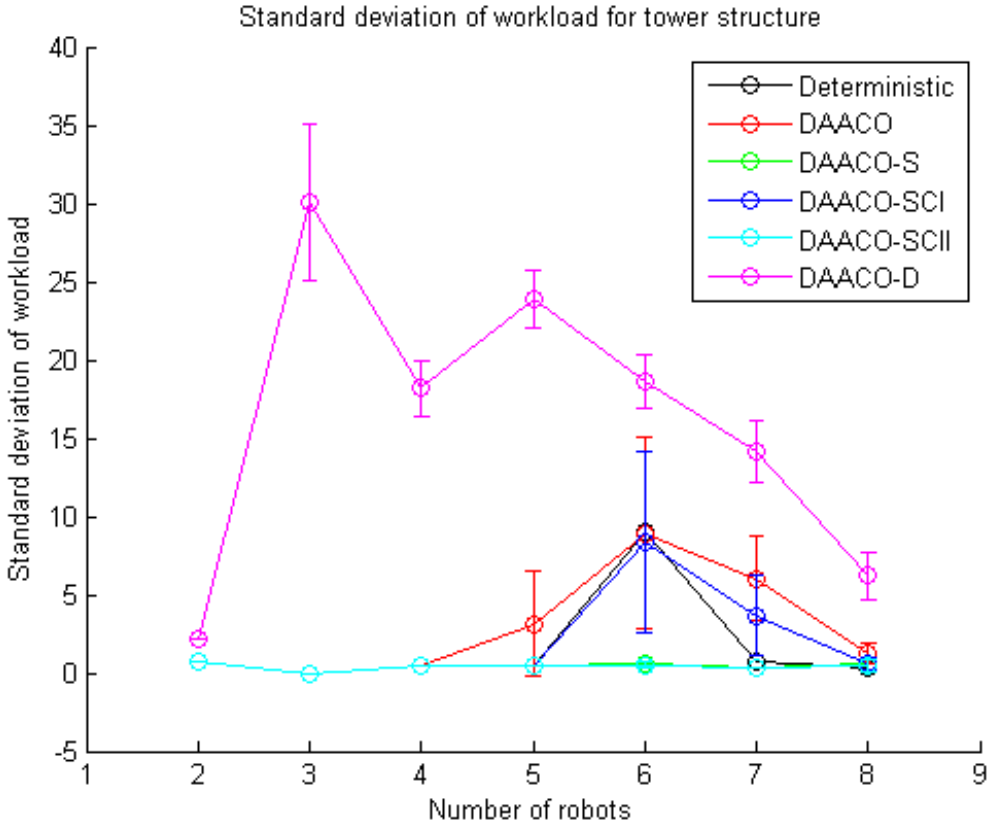
robots. In most cases the deterministic Dijkstra-based algorithm has the best performance, closely followed by DAACO-S, while the basic DAACO has higher workload variance, and DAACO-D is consistently the worst. A typical decomposition is shown in Fig. 8.5(b), generated by DAACO-S for eight robots. However, the deterministic algorithm, as it only generates one solution per problem, will occasionally encounter a situation that provides a poor result. One instance of this can be seen for 6 robots dividing the base structure. Although the deterministic algorithm generally produces slightly better results than DAACO-S, in this case DAACO-S vastly outperforms the poor solution chosen by the deterministic algorithm. These solutions are shown in Fig. 8.6.



(a) (b)

Figure 8.6: Solutions for six robots, as solved by DAACO-S (a) and the deterministic algorithm (b). Task sets are delineated by different colors.

Next, the algorithm's robustness to asymmetry is tested in the results of the two structures shown in Fig. 8.8. In one structure, one of the towers is much taller than the others, while the second is a 2-D structure with lopsided areas. Table 8.6 shows a direct comparison between these cases. Note that these asymmetric cases can cause problems for



(a)

Figure 8.7: Standard deviation of workload between robots when building the base structure, consisting of nine towers connected by ground paths.

Table 8.5: Performance as a Function of Team Size

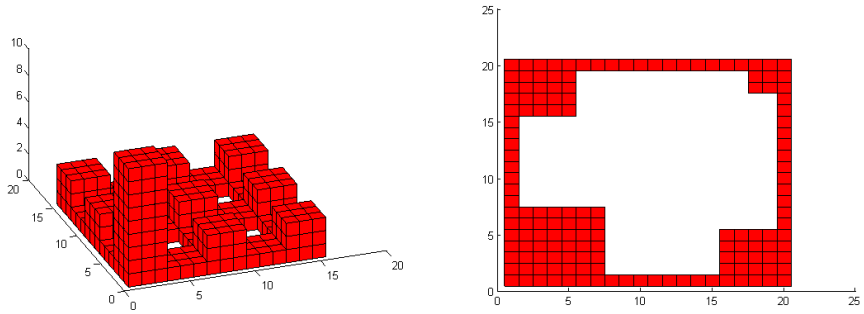
Algorithm	Number of Robots			
	2	3	4	5
Deterministic	0.71	0	0.50	0.45
DAACO	0.71 ± 0	0 ± 0	0.50 ± 0	3.15 ± 3.35
DAACO-S	0.71 ± 0	0 ± 0	0.50 ± 0	0.45 ± 0
DAACO-SCI	0.71 ± 0	0 ± 0	0.50 ± 0	0.45 ± 0
DAACO-SCII	0.71 ± 0	0 ± 0	0.50 ± 0	0.45 ± 0
DAACO-D	2.12 ± 0	30.11 ± 5.01	18.17 ± 1.81	23.87 ± 1.88
Algorithm	Number of Robots			
	6	7	8	
Deterministic	9.01	0.69	0.36	
DAACO	8.93 ± 6.09	6.01 ± 2.68	1.27 ± 0.69	
DAACO-S	0.55 ± 0	0.38 ± 0	0.59 ± 0.36	
DAACO-SCI	8.37 ± 5.77	3.66 ± 2.54	0.54 ± 0.40	
DAACO-SCII	0.52 ± 0.24	0.38 ± 0	0.49 ± 0.40	
DAACO-D	18.56 ± 1.70	14.14 ± 1.95	6.21 ± 1.51	

the deterministic algorithm, while DAACO-S still performs well. The plans generated for these structures are shown in Figs. 8.9 and 8.10.

Table 8.6: Results on Asymmetric Structures

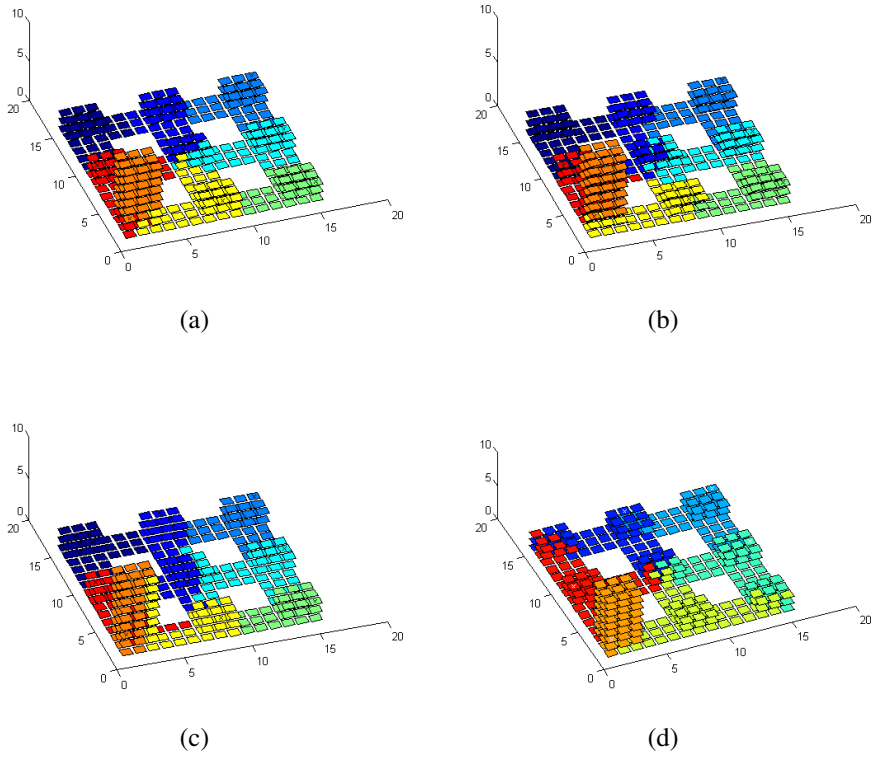
Algorithm	One Tall Tower	2-d asymmetric
DAACO	5.52 ± 1.87	5.62 ± 1.74
DAACO-S	0.75 ± 0.45	4.77 ± 1.38
DAACO-SCI	4.23 ± 2.14	5.07 ± 1.53
DAACO-SCII	0.64 ± 0.37	4.78 ± 1.70
DAACO-D	8.68 ± 1.32	9.41 ± 0
Deterministic Algorithm	6.41	5.42

Next, the algoritms are tested on a larger structure, which is a house consisting of 1360 nodes of three different shapes, shown in Fig. 8.11. The sloped roof pieces and



(a) (b)

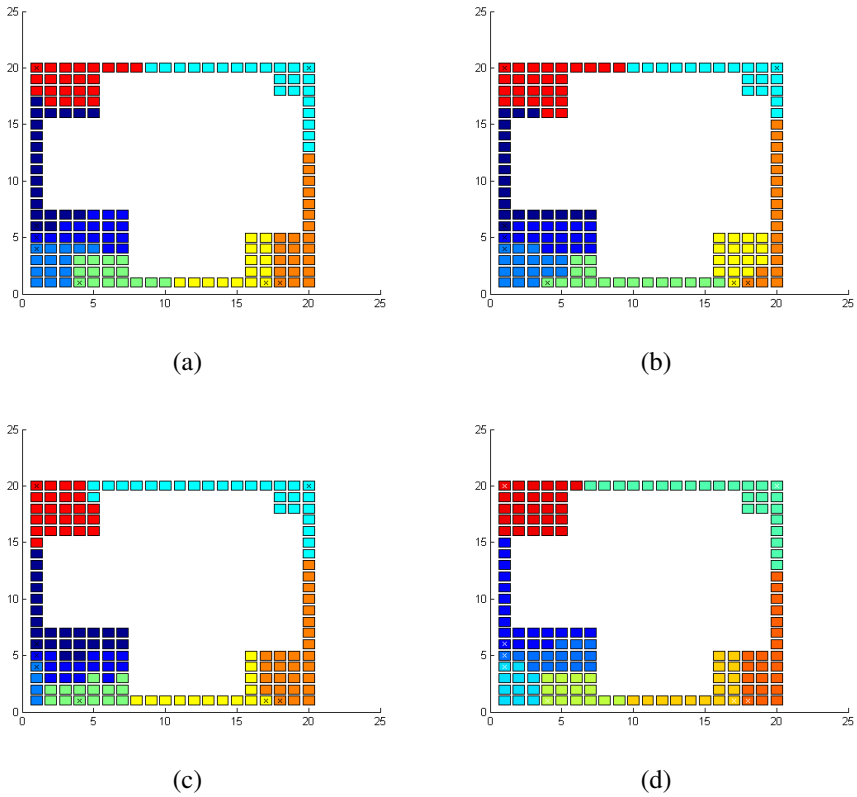
Figure 8.8: Two asymmetric structures. (a) Tower structure with one tower triple the height of the rest (b) 2-d structure with lopsided areas



(a) (b)

(c) (d)

Figure 8.9: Plans created for asymmetric tower structure by (a) DAACO, (b) DAACO-S, (c) DAACO-D, and (d) deterministic algorithm.



(a) (b)

(c) (d)

Figure 8.10: Plans created for 2-D structure with lopsided areas by (a) DAACO, (b) DAACO-S, (c) DAACO-D, and (d) deterministic algorithm.

corner pieces are given a mass of $1/2$, while cubic pieces have a mass of 1. Results are given in Table 8.7 and Table 8.8, and show that the best performers on this larger problem tend to be the deterministic strategy and DAACO-D. Comparing the two contiguity fixes, DAACO-SCI and DAACO-SCII, with the basic DAACO-S allows us to determine if any performance is lost in terms of workload balance by adding these restrictions. DAACO-SCII is usually similar in performance to DAACO-S, while DAACO-SCI has more cases where performance differs significantly from DAACO-S.

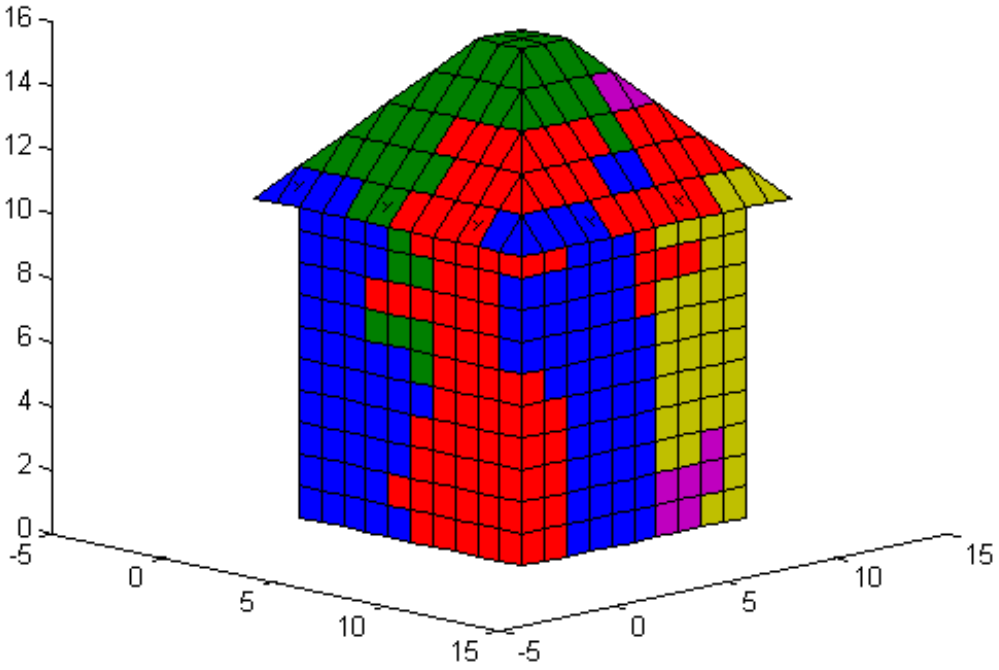
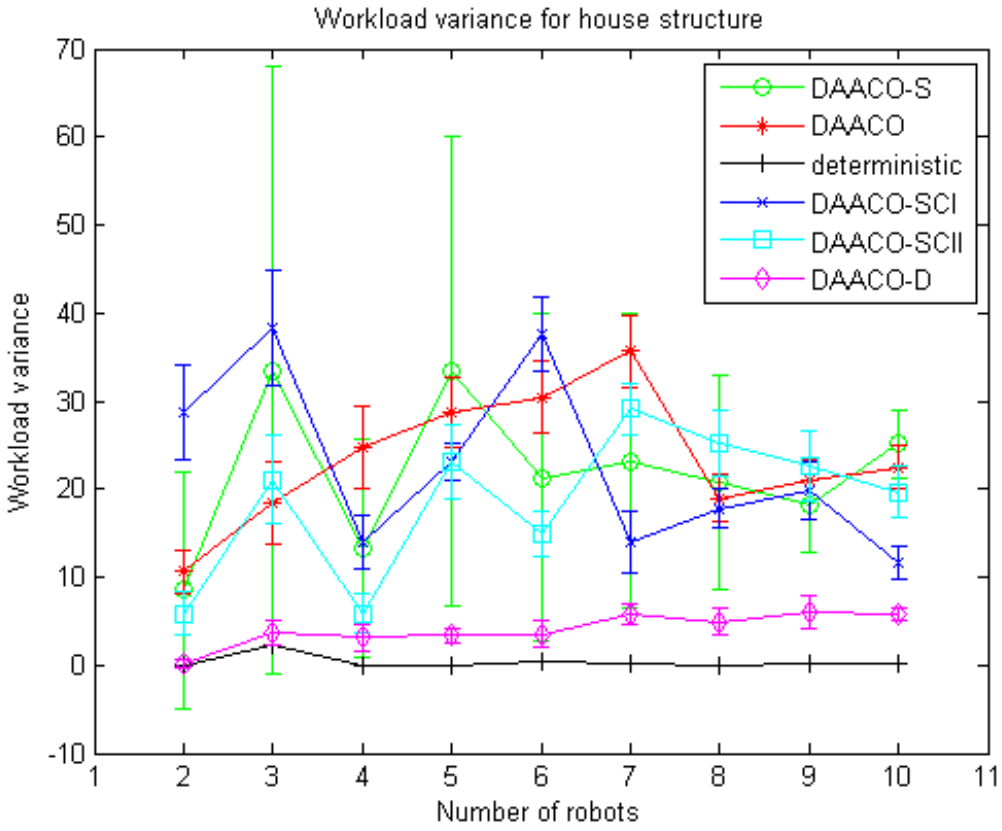


Figure 8.11: The larger structure, showing a plan for 10 robots determined by DAACO-SCII.

Varying the decay rate of the pheromone (what percentage evaporates between each

generation) has little effect on the resulting standard deviation of the workload, as shown in Table 8.9. These results were generated on the base structure.



(a)

Figure 8.12: Workload variance between robots when building the larger structure, a house of 1360 nodes.

Table 8.7: Workload Variance on Larger Structure with Clustered Start Nodes, Average of 5 Runs

Algorithm	Number of Robots			
	3	4	5	6
Deterministic	1.3	7.3	2.5	3.5
DAACO	132 ± 11.9	95.4 ± 4.9	92.4 ± 3.3	73.4 ± 6.6
DAACO-S	198 ± 133	172 ± 104	72.3 ± 23.3	61.1 ± 27.0
DAACO-SCI	233 ± 14.0	148 ± 10.5	133 ± 8.5	69.0 ± 6.3
DAACO-SCII	97.7 ± 121	117 ± 64.7	111 ± 55.2	69.9 ± 36.7
DAACO-D	1.1 ± 0.8	2.8 ± 1.4	2.5 ± 1.6	4.9 ± 1.3
Algorithm	Number of Robots			
	7	8	9	10
Deterministic	1.2	0.9	0.1	0
DAACO	57.0 ± 3.8	46.9 ± 4.0	41.9 ± 3.2	31.7 ± 2.6
DAACO-S	55.7 ± 19.5	48.0 ± 18.1	38.4 ± 14.8	46.1 ± 18.7
DAACO-SCI	68.0 ± 4.6	52.4 ± 4.2	40.4 ± 3.9	33.5 ± 3.5
DAACO-SCII	71.9 ± 20.1	65.6 ± 19.5	44.3 ± 14.3	38.7 ± 6.0
DAACO-D	4.6 ± 1.0	5.0 ± 1.0	5.1 ± 1.2	58.7 ± 0.0

8.3.2 Discussion

Very different results are obtained for the two types of structures tested. The larger structure, while having the advantage of testing the algorithms at a larger scale, may actually not be as difficult a test because of how many connections each cube has. The smaller structure has portions where there is only a single path, allowing more potential conflict between robots. In the more connected house structure, it is easier for robots to find paths around each other to continue claiming nodes.

The results on the tower-based structures show that allowing ants to steal nodes from each other greatly improves the performance of DAACO-S, giving it comparable performance to the Dijkstra-based algorithm from Chapter 4 in most cases, and superior performance in some. This performance appears to be largely independent of how much pheromone decay is present. Breaks in symmetry tend to be problematic for the deterministic approach, but are handled well by DAACO-S. When looking at the data for the

Table 8.8: Workload Variance on Larger Structure with Scattered Start Nodes, Average of 5 Runs

Algorithm	Number of Robots			
	3	4	5	6
Deterministic	2.3	0	0	0.3
DAACO	18.4 ± 4.6	24.7 ± 4.7	28.7 ± 4.0	30.4 ± 4.1
DAACO-S	33.5 ± 34.6	13.3 ± 12.4	33.4 ± 26.6	21.3 ± 18.6
DAACO-SCI	38.3 ± 6.6	14.0 ± 3.0	23.1 ± 2.2	37.7 ± 4.2
DAACO-SCII	21.1 ± 5.1	5.9 ± 2.3	23.2 ± 4.2	15.0 ± 2.6
DAACO-D	3.7 ± 1.5	3.1 ± 1.6	3.4 ± 0.8	3.5 ± 1.5
Algorithm	Number of Robots			
	7	8	9	10
Deterministic	0.2	0	0.1	0.2
DAACO	35.7 ± 4.1	18.9 ± 2.7	21.0 ± 2.3	22.5 ± 2.4
DAACO-S	23.2 ± 16.7	20.8 ± 12.2	18.1 ± 5.3	25.1 ± 3.8
DAACO-SCI	14.0 ± 3.6	17.8 ± 2.3	19.8 ± 3.2	11.6 ± 1.9
DAACO-SCII	29.1 ± 2.9	25.2 ± 3.7	22.6 ± 3.9	19.7 ± 3.0
DAACO-D	5.8 ± 1.1	4.9 ± 1.5	6.0 ± 1.9	5.8 ± 0.8

Table 8.9: Performance with Different Decay Rates

Algorithm	Decay Rate				
	0	0.1	0.5	0.75	1
DAACO	5.11 ± 2.57	5.96 ± 2.52	6.28 ± 2.40	6.22 ± 2.35	6.42 ± 3.21
DAACO-S	0.52 ± 0	0.62 ± 0.40	1.09 ± 0.79	0.91 ± 0.90	0.76 ± 0.61
DAACO-SCI	3.48 ± 2.30	3.92 ± 2.76	4.31 ± 2.35	4.34 ± 2.71	4.47 ± 2.40
DAACO-SCII	0.73 ± 0.57	0.71 ± 0.44	0.82 ± 0.35	0.87 ± 0.50	0.95 ± 0.95
DAACO-D	8.54 ± 1.07	8.67 ± 1.35	8.59 ± 1.00	8.64 ± 0.99	8.61 ± 1.31

larger structure, the deterministic strategy and DAACO-D become the best performers. DAACO-SCII shows similar performance to DAACO-S for most trials, meaning that the more flexible method of guaranteeing contiguity does not seem to impose additional costs. DAACO-SCI, on the other hand, shows more significant deviation from DAACO-S, as a result of its more restrictive rule for vetoing trades.

DAACO-D shows behavior that differs with the number of tasks. With fewer tasks, DAACO-D is outperformed by DAACO, possibly because of the larger solution space that

results from having twice as many pheromone markers, since each pair of directed edges has separate pheromone, while an undirected edge only needs one value. Another potential reason may be because not adding pheromone between nodes at the same distance from the starting point weakens the incentive to tightly cluster the tasks chosen, and allows claiming of more distant tasks which can block off other ants, leaving them with smaller workloads. However, on the larger structure with 1360 nodes, DAACO-D shows superior results to the other variants of DAACO. This may be because the increase in solution space is more significant on the smaller structures, where DAACO and DAACO-S are able to do a more exhaustive search in the smaller space. On the larger structure, the size of the solution space is much larger relative to the number of solutions explored, so the number of samples becomes a more important factor than the size of the solution space. By distinguishing between the direction of the claim, DAACO-D is able to give a preference to claiming nodes closer to the starting point, which may yield more compact task sets, giving each robot more time to claim the nodes in its neighborhood before other robots begin to interfere. This advantage may be able to make up for the discrepancy in solution space size for a larger structure where a more exhaustive search is not possible.

Overall, the performance with scattered starting nodes was consistently superior to the clustered starting node case, showing that the choice of starting nodes is a significant factor in the performance of this task.

One aspect of this algorithm worth noting is that it can be used to maintain connectivity back to an external node, similar to the Dijkstra-based algorithm. By requiring the starting points to be on the exterior of the structure and limiting newly claimed nodes to be adjacent to already possessed nodes, DAACO, DAACO-D, DAACO-SCI, and DAACO-SCII provide contiguous tasks that contain part of the exterior of the structure. This means each robot can plan a way to build its own part that avoids being trapped in a partially complete structure. As discussed in Hsieh and Rogoff (2010), when using Voronoi methods, robots

can have entirely internal tasks, leaving them no way to escape the structure if other robots build a wall around them.

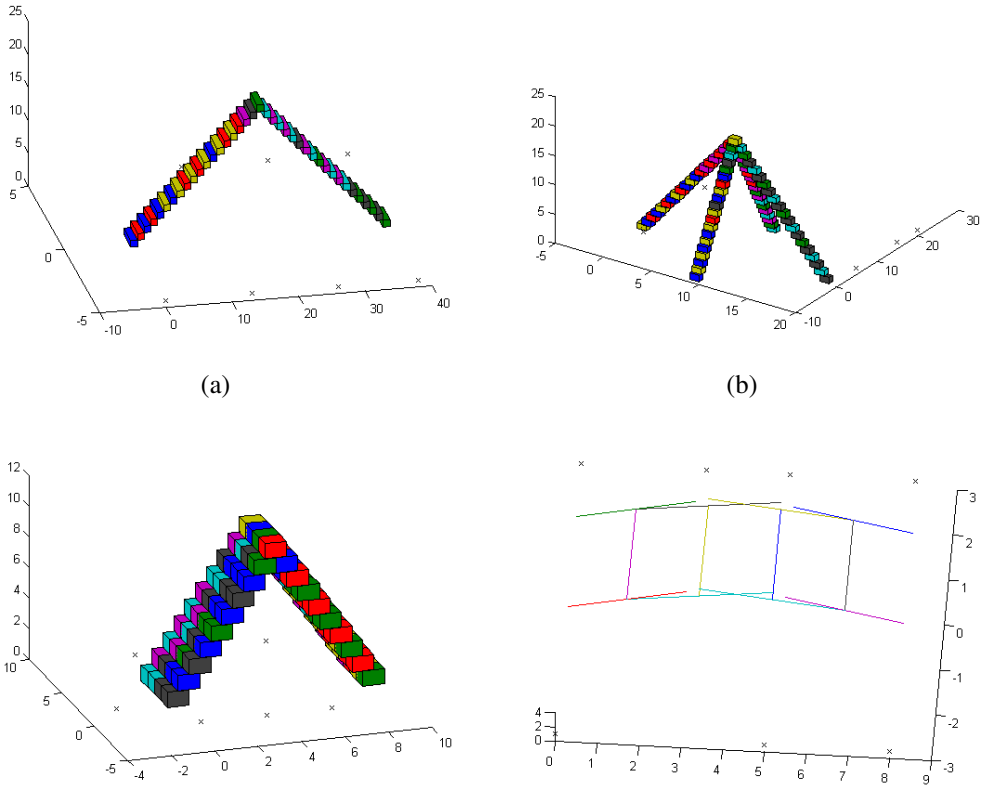
DAACO-SCI and DAACO-SCII were implemented in order to maintain this connectivity. The cost of this was quite high for DAACO-SCI, as it shows results similar to DAACO, implying that it is simply blocking most attempts to steal a node. However, it was found that DAACO-SCII achieved similar performance to DAACO-S, while still maintaining contiguity.

8.4 Cooperative Manipulation

8.4.1 Results

The approach is tested on four types of archlike structures, shown in Fig. 8.13. The structure types are a basic arch, an arch with four legs, a tunnel made of independent arches, and a Da Vinci bridge. A Da Vinci bridge http://www.leonardodavincisinventions.com/leonardo-da-vinci-models/leonardo-da-vincis-self-supporting bridge/, shown in Figure 8.14, is a structure designed to stand without any type of fasteners, and is of interest here as it is an archlike structure with more complicated support relationships than a typical arch. For each structure type, four quantities are considered: number of nodes built per robot, finish time of the structure, wait time per robot, and number of messages per robot. These figures are given for the arch in Table 8.10, the four legged arch in Table 8.11, the tunnel in Table 8.12, and finally the Da Vinci bridge in Table 8.13.

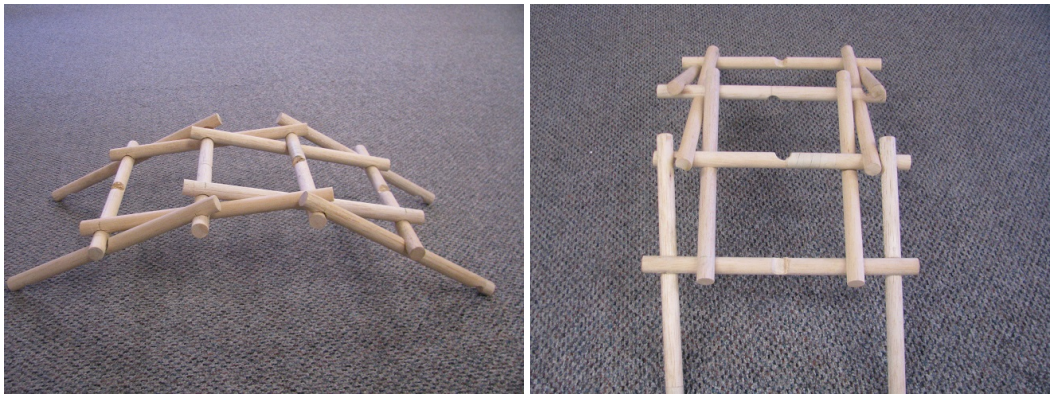
The trends for each structure type with increasing numbers of robots are compared in four graphs. First, Figure 8.15(a) shows that the number of nodes built by each robot declines as expected as the number of robots increases. One interesting phenomenon here is that the standard deviation of nodes built for the standard arch is increasing as the number of robots increases. What is occurring here is that some of the later robots are being given almost no work throughout the course of the experiment. As the first available robot id is



(a) (b)

(c) (d)

Figure 8.13: (a) Basic arch structure. (b) Four legged arch. (c) Tunnel composed of independent arches. (d) Da Vinci bridge.



(a) (b)

Figure 8.14: Side (a) and front (b) views of a Da Vinci bridge.

chosen to build arch supports, when there are more robots than needed, robots with higher ids are generally left idle. This increase starts at the same number of robots as the increase in finish time, discussed below, which reinforces the idea that this is caused by having extra, idle robots.

Table 8.10: Results for Arch

Criterion	Number of Robots		
	3	4	5
Nodes Built	14.0 ± 1.7	10.8 ± 0.5	8.8 ± 1.3
Finish Time	755.7 ± 3.0	489.6 ± 11.0	380.1 ± 6.6
Wait Time	560.9 ± 50.3	332.7 ± 52.7	254.1 ± 27.7
Number of Messages per Robot	1451 ± 149	929 ± 48	831 ± 38
Criterion	Number of Robots		
	6	7	8
Nodes Built	7.5 ± 1.4	6.6 ± 2.4	5.8 ± 2.6
Finish Time	389.3 ± 11.7	434.4 ± 15.8	442.8 ± 26.8
Wait Time	261.2 ± 56.6	324.2 ± 26.9	356.1 ± 57.4
Number of Messages per Robot	931 ± 36	1082 ± 70	1198 ± 84

Table 8.11: Results for 4 Legged Arch

Criterion	Number of Robots			
	5	6	7	8
Nodes Built	16.8 ± 2.4	14.2 ± 2.6	12.2 ± 2.3	10.8 ± 2.2
Finish Time	1141 ± 94	1079 ± 51	1917 ± 21	1988 ± 155
Wait Time	958 ± 167	734 ± 103	1445 ± 100	1385 ± 332
Number of Messages per Robot	2233 ± 391	2209 ± 242	3640 ± 510	2990 ± 1181

Figure 8.15(b) examines the relation of finish time to number of robots for each structure type. The general pattern is an initial decrease in finish times followed by an increase

Table 8.12: Results for Arch Tunnel

Criterion	Number of Robots		
	3	4	5
Nodes Built	26.3 ± 2.5	20.0 ± 0.8	16.6 ± 2.9
Finish Time	1257.7 ± 0.7	808.7 ± 4.0	1181.8 ± 91.0
Wait Time	839.5 ± 42.3	481.8 ± 49.1	1037 ± 189.3
Number of Messages per Robot	1862 ± 51	1489 ± 30	3226 ± 780
Criterion	Number of Robots		
	6	7	8
Nodes Built	13.7 ± 1.4	12.3 ± 1.5	11.5 ± 1.8
Finish Time	759.5 ± 38.7	827.2 ± 10.6	909.1 ± 14.4
Wait Time	459.2 ± 84.4	538.2 ± 77.2	593.0 ± 105.0
Number of Messages per Robot	1719 ± 167	1984 ± 84	2211 ± 89

Table 8.13: Results for Da Vinci Bridge

Criterion	Number of Robots			
	5	6	7	8
Nodes Built	3.4 ± 1.1	3.0 ± 0.5	2.7 ± 0.7	2.5 ± 0.5
Finish Time	275 ± 23	185 ± 40	184 ± 45	201 ± 37
Wait Time	280 ± 129	210 ± 48	248 ± 43	187 ± 67
Number of Messages per Robot	508 ± 184	410 ± 118	420 ± 182	523 ± 187

as more robots are added past a certain number for each structure. This contrasts strongly with the findings for earlier approaches presented here, where increasing the number of robots consistently decreases completion time of the structure.

Next, the average wait times per robot are given in Fig. 8.15(c). Unsurprisingly, this closely follows the results for completion time, as the main factor that affects completion time is how long each robot spends waiting. Finally, the number of messages sent per robot is given in Fig. 8.15(d), which is also strongly correlated with completion times.

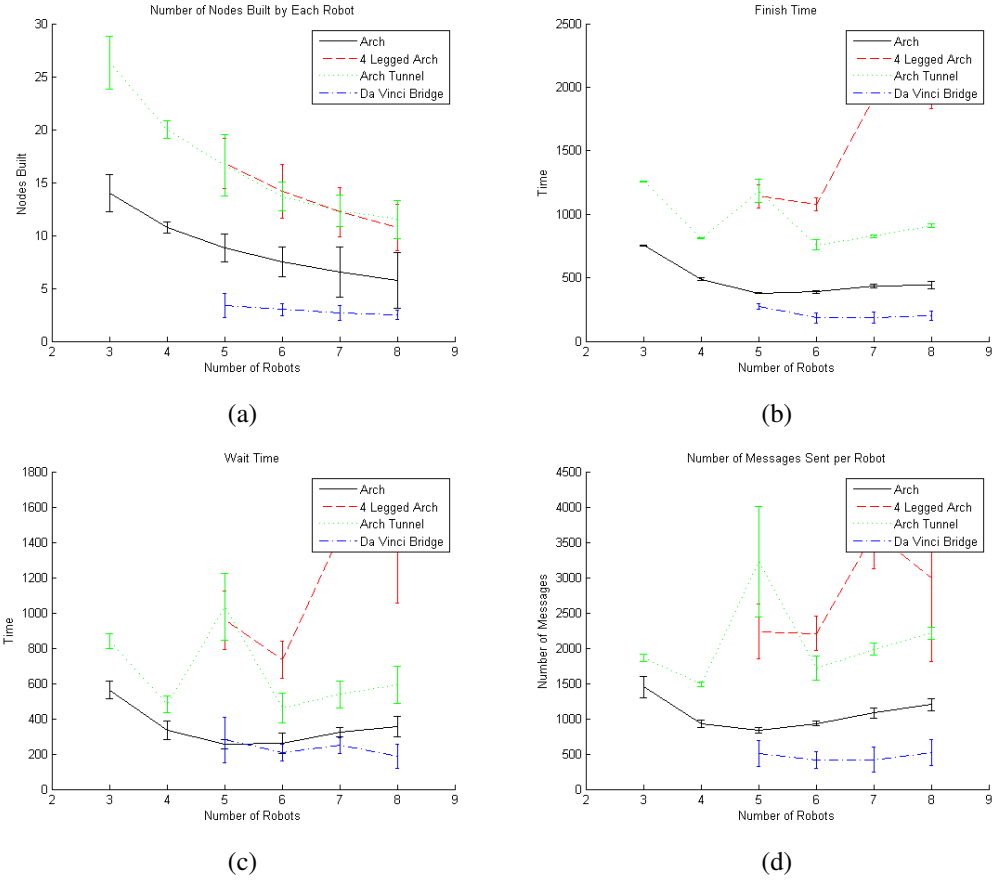


Figure 8.15: (a) Number of nodes built per robot for each structure type. (b) Finish times for each structure type. (c) Total wait times for each structure type. (d) Number of messages per robot for each structure type.

8.4.2 Discussion

One of the most interesting trends in the results is the increase in finish times when more robots are added. As observed above, this seems to be strongly correlated with an increasing number of messages. The reason that this is occurring is that the most common type of message are the availability messages, which are frequently resent to ensure that all robots can make an accurate decision as to which robot to assign work to. As this message is sent to all robots in N_2, the number of these messages is proportional to the size of N_2, which for these experiments is equal to the number of robots in the system. This could be solved with a broadcast architecture that does not involve sending a separate message to each robot, although this does not allow for acknowledgements of these messages to be sent, as the number of acknowledgements of each availability message would again be proportional to the number of robots. It also does not allow a smaller N_2 for larger teams. This introduces a trade-off, where one can either accept a penalty for increasing the size of N_2, or accept a higher probability of missed messages, potentially leading to inconsistencies among robots about task assignments. For the number of robots being considered, it was decided that the first cost was more acceptable. For a larger swarm of robots, one could potentially solve this issue by adding more sensing to individual robots to allow them to perceive if another robot has placed the node they are attempting to build.

However, in most cases this increase is fairly gradual, overwhelmed at first by the effect of adding more robots to further split the workload. This seems to indicate that the benefit of adding robots quickly levels off for these archlike structures. This is likely because of the much more restricted space of nodes to build at any given time. If a robot builds a node that does not act as an arch support for a node being held by another robot, it will immediately become stuck waiting for its own arch support, thus limiting the pool of available robots.

All structures adhere very closely to the requirement established by Theorem 4. The arch and arch tunnel both have a constraint graph with a degree of three one step below

the top of the arch, and indeed it takes a minimum of three robots to build these structures. The four legged arch has a degree of five one step below the top, and requires a minimum of five robots. The Da Vinci bridge has a degree of four in G_C, and takes five robots before the algorithm is able to divide the workload. As the theoretical bound is one higher than the degree of G_C, this does not violate Theorem 4. Interestingly, the Da Vinci bridge can in fact be built by three robots if one robot is assigned all of the cross pieces. Note that this also does not violate Theorem 4, as that only establishes a maximum on the number of robots that may be required to build the structure.

Although many of the nodes are built by assignment rather than according to the original division of workload, that original division is still necessary for correct completion of the structure. Assigning each node to be the responsibility of a single robot provides a guarantee that that robot will not complete its task until it knows that all of its nodes have been built. This means that while it may be possible for some builds to be repeated if there is a failure in communications, the approach will never terminate with an incomplete structure. The possibility of a repeat build due to communications failure can be handled by locally sensing whether that node has already been placed. Also, the initial assignment allows the assignment of a needed arch support to be determined by the initial owner of that node, which can then remove that work from its own task at the same time it assigns the task to another robot. This again reduces the possibility of repeated builds due to dropped messages. Having examined the results of the different approaches, the next step is to verify that these methods will work in experiment.

9. Experimental Validation

The proposed distributed assembly strategies were implemented on the Drexel SASLab's multi-robot assembly testbed. The testbed consists of two mini-mobile manipulators (M3 robots), or $N_c = 2$, shown in Figure 9.5, each equipped with an iRobot Create base, a Crustcrawler 5 DOF arm, 802.11b wireless communication, and a Hokuyo URG laser range finder (LRF). The laser range finder is first used to detect the position and orientation of the tile with respect to the robot. The arm is equipped with a 1 degree of freedom 2 finger gripper. The arm is then commanded to pick up or place the block at the given position. Tiles to be picked up are individually placed at fixed locations in the workspace. Since the geometry of each tile and the desired structure is known a priori, the desired placement position and orientation of each tile can be computed in relation to the base of the structure as detected by the laser range finder. Further, since each tile directly above another is defined as being constrained by the supporting tile, the robots assume the tiles above the current tile have not been placed yet, allowing us to assume an obstacle free path for the manipulator. Two types of tiles are used, the first are interlocking Lincoln-Log style blocks, and the second type attach and self-align using magnets. To test online error correction, some experiments include one scanning robot equipped with a iRobot Create base and a Microsoft Kinect visual depth sensor. This robot follows a fixed rectangle surrounding the workspace of the assembly robots. Overhead localization for the robots was provided using two to four visual cameras.

Each robot was given the global position of the structure's center and the positions of their respective parts cache.

To test error correction, each robot was assigned a preplanned assembly plan as described, with no node trading allowed. The assembly plans consisted of a list of tile identifiers in the computed assembly order. Distributed implementation of the plan was achieved

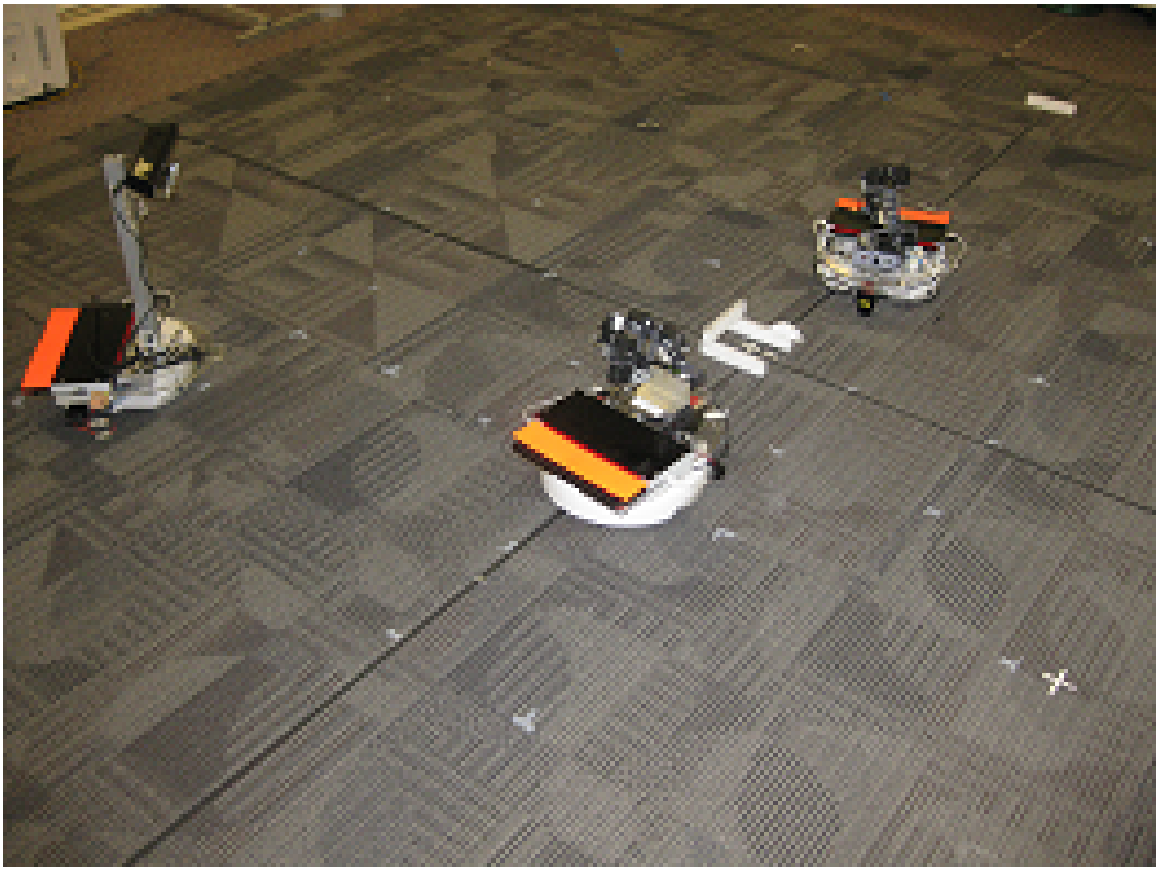
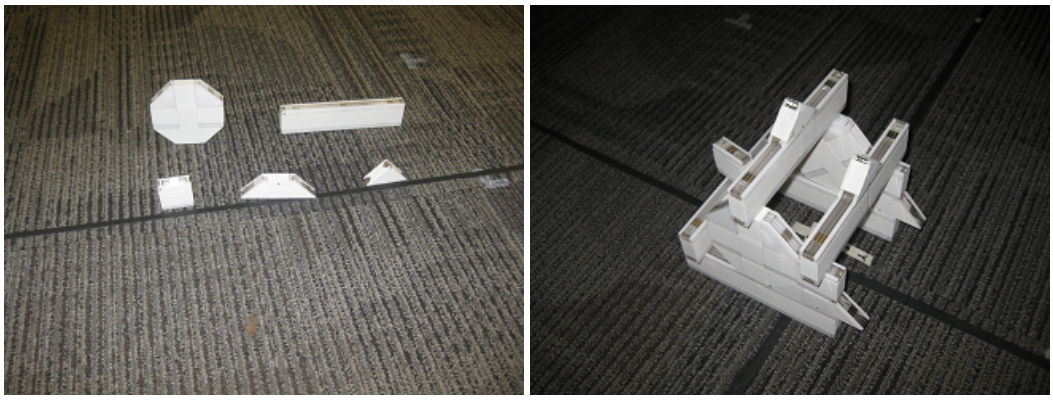


Figure 9.1: Team of two assembly robots and one VI-robot with a raised Kinect around a partially completed structure.



(a) (b)

Figure 9.2: (a) Sample assembly tiles. (b) Desired structure to be assembled.

by encoding the immediate supports for each component in the plan to ensure robots wait for the placement of a missing support tile by another robot before placing their parts.

The assembly tiles were placed in predefined cache locations in the workspace. These experiments will test the distributed assembly of 3D structures composed of 5 distinct tile types.

9.1 Results

9.1.1 Dijkstra-Based

To compare the variants of the Dijkstra-based preplanning strategy, Algorithms 1-3 were used to partition the assembly task for the structure shown in Fig. 4.2 for a 2-robot team. Each robot was given the global position of the structure's center and the positions of their respective parts cache. The assembly parts were Lincoln-Log style plastic interlocking blocks, as in Fig. 4.2(a). Each robot was assigned their respective assembly plans determined by Algorithms 1-3. The assembly plans consisted of a list of node identifiers in the computed assembly order. Distributed implementation of the plan was achieved by encoding the immediate supports for each node in the plan to ensure robots wait for the

placement of a missing support by another robot before placing their parts.

Five experimental trials were executed for each plan. The time required by each robot for assembly and the total distance traveled by each robot are shown in Table 9.1. It was noted that Algorithms 2 and 3 produced the same plans, which will be referred to as Plan 2 (Fig. 4.2(c)), and the plan generated by Algorithm 1 as Plan 1 (Fig. 4.2(b)). Similar to the simulation results, the experimental results show that neither plan resulted in any waiting time. This suggests that the sequencing phase (Phase III) is sufficiently robust to accommodate real-world uncertainties.

The cache positions used for this task were located along the long axis of the structure. This means that Plan 1, where each robot had to place two blocks in the tower farther from its associated cache, will require more travel time. This is supported by the results in Table 9.1 where Plan 1 performs slightly worse.

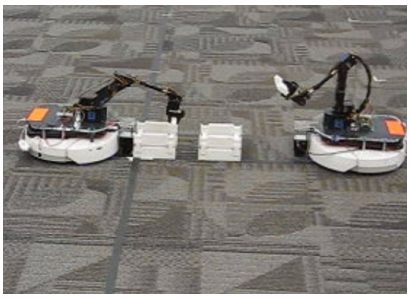


Figure 9.3: Two M3 robots working on a structure.

9.1.2 Online Workload Balancing and Error Correction

Next, the online workload balancing and error correction was tested. The assembly parts were plastic tiles of various shapes and sizes (side lengths from $4-17$ *cm*), each with

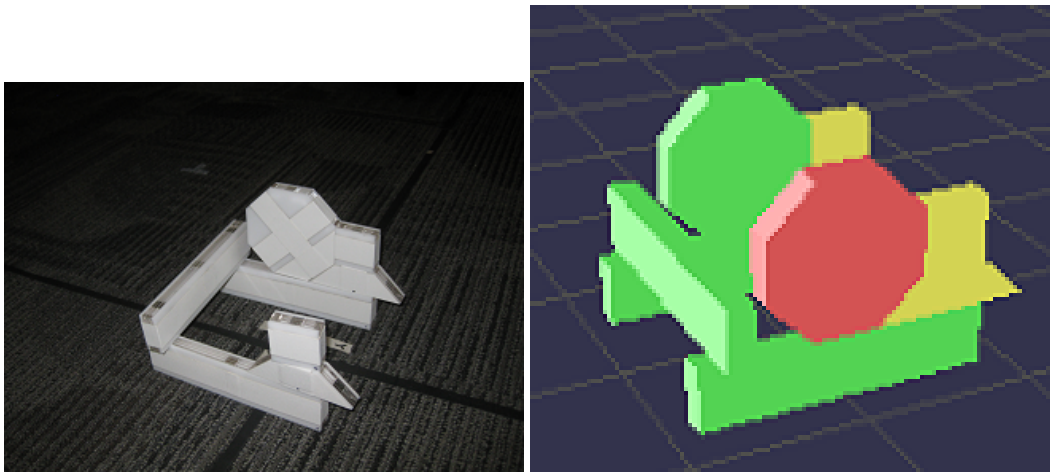
Table 9.1: Experimental results

Measure	Plan	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
Finish Time (s)	Plan 1	1069	1065	1030	1109	1174	1089
	Plan 2	1028	1037	1060	1021	1027	1035
Time Difference (s)	Plan 1	97	117	96	160	142	122
	Plan 2	11	2	144	100	12	54
Distance Traveled (m)	Plan 1	32.10	32.25	32.18	32.33	32.22	32.22
	Plan 2	31.58	31.30	31.27	31.32	31.34	31.36

a given set of magnetic attachment sites (see Figure 9.2(a)). Figure 9.2(b) shows the desired structure used for the experiment. To simulate missed placements, random assembly tiles were removed at various times during the assembly process.

Fourteen experimental trials were run on the scanning robot for the desired structure shown in Figure 9.2(b). During each trial, one or more random assembly tiles were removed at different parts of the assembly process. Figure 9.4 shows the results of one of the experimental trials where the missing tile was successfully detected by the scanning robot. Out of twenty-two removed blocks, the scanning robot was able to successfully detect twelve of the missing tiles and reported undetermined for the other ten. There were no false positives during these trials, and only one false negative where a tile was reported as missing when it was actually present. The smaller tiles (square and triangle) were always reported as undetermined, while the larger tiles were always detected as missing after they had been removed in these trials.

Table 9.2 summarizes the assembly partition obtained at the start of an experimental trial for each robot. The tiles allocated to each robot are shown in the order in which they are supposed to be placed. Table 9.3 shows the updated assembly allocation as tiles are removed during the experiment, including the workload reallocation after the detection of errors.



(a) (b)

Figure 9.4: ((a) Tile removed. (b) Missing tile reported by the scanning robot.

Robot 1	Tile ID	**Robot 2**	Tile ID
Long Rectangle	3	Trapezoid	7
Trapezoid	6	Octagon	5
Octagon	4	Square	8
Square	9	Square	10
Long Rectangle	11	Long Rectangle	12
Triangle	13	Triangle	14

Table 9.2: Initial Allocation for the 3D Structure in Fig. 9.2(b)

In the next set of experiments the node trading was included to test for a reduction in assembly time compared to the preplanned approach. Tests were also done with one robot being given artificially longer assembly times to simulate the effects of a less efficient robot. For this, the manipulation was replaced with a timed delay drawn from a specified distribution. The typical manipulation times were measured to have a mean of 69.0 seconds and a variance of 2.5 seconds. The first set of experiments is done with this scaled down by a factor of 10. For the second set of experiments, robot 2 keeps the same distribution while robot 1's times are scaled up to half the original times (5 times that of robot 2). This is

Robot 1	Tile ID	**Robot 2**	Tile ID
Long Rectangle	3	Trapezoid	7
		Removed tile	7
Trapezoid	6		
Octagon	4	Trapezoid	7
Removed tile	4		
Square	9	Octagon	5
		Removed tile	5
Octagon	4	Square	8
		Removed tile	8
		Octagon	5
Long Rectangle	11	Square	10
Triangle	13	Long Rectangle	12
Removed tile	13		
Square	8	Triangle	14
		Triangle	13

Table 9.3: Allocation After Detection of a Missing Tile.

done in order to create a situation where the slower robot will need to give work to the faster robot in order to minimize completion time. Another benefit of running the experiments this way is that the unreliable placement does not need to be corrected with a scanning robot, allowing the node trading to be analysed on its own. Table 9.5 shows results for the two sets of experiments. From the results, it does not seem that the higher manipulation time was sufficient to cause any trades to occur for such a small structure. In fact, one of the trials with the higher manipulation time finished before the faster manipulation time. Since this was adding about 20 seconds per placement for 5 placements, it should have added 100 seconds to the completion time for the slower robot. However, the variation in time taken to navigate and use the LRF to align with the cache and structure seems to have been large enough to wash out the effect of a higher manipulation time. Larger experiments are needed to see the effects visible in the simulations.

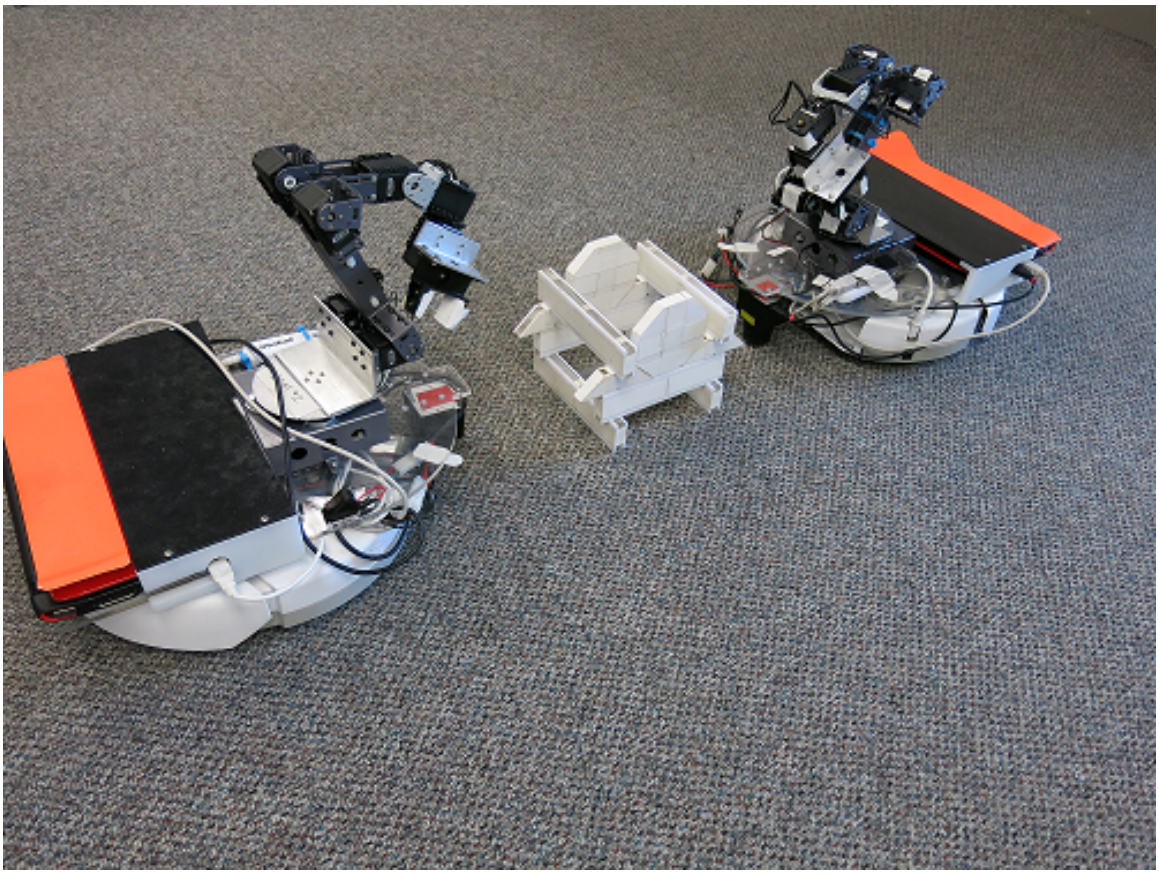


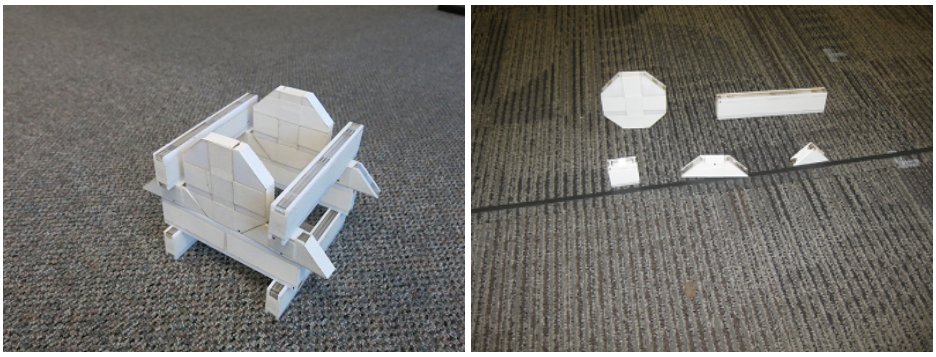
Figure 9.5: Two assembly robots with a completed structure.

9.1.3 Ant Colony Optimization

In the final set of experiments, each robot was assigned their respective assembly plans determined by DAACO-S. The assembly plans consisted of a list of block identifiers in the computed assembly order. Distributed implementation of the plan was achieved by encoding the immediate supports for each component in the plan to ensure robots wait for the placement of a missing support block by another robot before placing their parts.

Each assembly block was placed in the cache in the order specified by the generated plan. For this experiment, Figure 9.6 shows the desired structure for the experiment and the parts available.

7 full trials were conducted, with 2 robots placing 5 blocks each, along with an ad-



(a) (b)

Figure 9.6: (a) The target structure. (b) The parts available in the experimental setup.

ditional 3 half trials with only robot 2, for a total of 85 attempted placements. Of these, there were 31 failures, 15 of those occuring during pickup, 16 occurring during placement. This yields a 64% rate of successfully retrieving a piece and placing it correctly. The time to pick up and place a block averaged 246.9 seconds with a standard deviation of 18.3 seconds. The highest time observed over the 85 placements was 309.6 seconds, while the lowest was 209.7 seconds. The time for one robot to complete its task averaged 1230.1 seconds with a standard deviation of 71.4 seconds. Completion times are given in Table 9.4. These rates clearly do not allow for correct assembly of large-scale structures, but could be improved upon by adding more accurate sensing and more reliable manipulation.

9.2 Discussion

The execution of complex tasks by a team of heterogeneous robots in a complex and dynamic environment with limited resources poses significant challenges. Most existing assembly strategies do not explicitly address the impact of sensing and actuation noise

Table 9.4: Experiment Completion Times in Seconds

Trial	Robot 1	Robot 2
1	1192.8	1169.1
2	1157.4	1202.5
3	1153.9	1253.1
4	1258.0	1214.2
5	1162.8	1272.6
6	1180.1	1188.3
7	1293.6	1256.2
8	N/A	1424.4
9	N/A	1205.6
10	N/A	1327.0

Table 9.5: Two M3 robots with variable manipulation times

Criterion	Same speed	Different speed	Different speed, no trading
Structure Size	9	9	9
Ave. # of nodes	4.5	4.5	4.5
St. Dev. of # of nodes	0.71	0.71	0.71
Total # of nodes traded	0	0	0
Ave. wait during construction	0.6	0.4	0.6
Ave. wait after construction	165.8	220.4	156.3
Ave. completion time	1007.0	1066.4	953.0

on the performance of a team of autonomous robots tasked to assemble complex three dimensional structures in an actual physical space.

The experimental setup involved two types of real-time on-board sensing: 1) the ability to localize the individual assembly tiles for pick-up and placement by the assembly robots, and 2) the ability to determine the state of the assembly structure during the entire assembly process. In both cases, the relative small size of the assembly tiles in relation to the sensing and actuation precision of the actuators and sensors used in the system posed significant engineering challenges. However, the ability to overcome these limitations at the small scale suggests that one can be more confident in the performance of the algorithms when employed on larger full scale systems.

10. Conclusions and Future Work

The main contribution of this work is to extend graph search strategies, specifically Dijkstra's Algorithm, and evolutionary algorithms, specifically Ant Colony Optimization, to address a distributed task involving constraints on task ordering. As the focus in this work has been on distributed assembly, the approaches detailed have included methods that are specific to the challenges in that task, notably in the types of constraints considered, the communication needed for online workload balancing, and consideration of special types of structures. However, the general form of these approaches could be adapted to other applications having similar task relationships, such as job assignment in parallel processing, or pallet planning for an automated warehouse.

The area of distributed assembly is still fairly new, and has a number of promising approaches in it. A central issue is that many of these approaches are limited to specific hardware and types of structures, which makes comparisons between approaches difficult. Another problem is that these approaches do not all define their goals the same way, which makes comparisons impossible unless a common objective can be agreed upon. In this thesis the objective used has been either minimizing completion time, or the stand-in objective of balancing workload for preplanning approaches, which is intended to give plans that will lead to minimizing completion time in experiment. This is similar to the objective function used by Daniela Rus's group (Yun et al., 2009; Yun and Rus, 2010; Stein et al., 2011; Schoen and Rus, 2013) that is primarily dependent on balancing workload as well. However, some other approaches, most notably Justin Werfel's work on TERMES (Werfel and Nagpal, 2008; Petersen et al., 2011; Werfel et al., 2014), does not present any objective beyond guaranteeing successful completion of the structure. Given the present state of the field, it seems to me that each of these approaches has something valuable to offer and should be continued, although it would be helpful to agree on a common set of challenges

to aim for. In this thesis an attempt was made to extend the approach to handle situations involving unstable partial structures, this seems like a necessary step towards actual automation of construction work. Other potential directions include more heterogeneity of construction tasks, such as spreading mortar, laying bricks, and pouring concrete with a team of robots of varying capabilities. There is also a human-robot interaction problem that must be solved for automated construction, as early steps would likely involve specialized tasks for a robotic team which then must coordinate with humans performing other tasks in the same environment.

For the approaches presented here, it seems that in most cases the approach based on Dijkstra's Algorithm performs fairly well, especially if augmented by online workload balancing, although in practice communication difficulties remain a concern, and the approach must be resilient to communication failures. The final approach presented here addressed this to some extent, implementing a scheme where failures in communication will result in redundant builds rather than missing pieces, since local sensing before placement can resolve a redundant build assignment by returning the part and labeling it built. The DAACO approaches were able to improve on the performance of the deterministic algorithm in some cases, particularly those with less symmetry, but this comes at a cost of considerably more computation during preplanning. Whether this is worthwhile would depend on the application, though in most construction tasks there is likely sufficient time between the final blueprints and breaking ground on the site for DAACO to run enough generations to be worthwhile. The best approach of those presented here would be DAACO-SCII for preplanning combined with the online workload balancing approach during actual construction.

There are many areas that could be expanded in future work. For the deterministic preplanning approach, one direction for future work is to enable the algorithm to change the entrance locations during Phase II to allow more flexibility in workload exchange between

robots. For the online error correction, a possible direction for future work is to improve the visual feedback system to provide more detailed assessment of the state of the assembly structure. In particular, the reduction of false negatives by visually inspecting the structure via different viewpoints. A second direction for future work is to extend the visual feedback system to enable identification of incorrect assembly placements as well as missing tiles. The path of the scanning robot, which is currently a rectangle around the workspace, could also be optimized to ensure each node gets more frequent views. Expanding the size and reliability of the experimental testbed will allow more informative results to be seen. Finally, in both simulation and experiment it is worth noting that in several cases significant idle time is seen at the end of the experiment, meaning that robots that have completed their own tasks do sometimes have a long delay before the last robot finishes. This means that there is still room to gain from adding a mechanism for a robot to split its task in half, giving half to an idle robot. The complication with this will be revising the approach to allow two or more robots to share a root node, which will require all of them to confirm they are finished before one is allowed to build the root. Another way would be to split off a subtree with its own valid root node. For DAACO and its variants, one area for future work is to apply an idea from Simulated Annealing, and allow p_{min} to decay over time, starting with a high value to encourage early exploration of the solution space, with the best solution being further refined as the noise represented by p_{min} decreases.

Several directions for extension of the cooperative manipulation task are possible. First, I am interested in extending the approach to apply also to structures that require pieces too large for a single robot to carry. This should be possible within this approach by having the first robot to arrive at such a piece assign one of the available robots to assist, in the same way that arch supports are assigned. Another direction for future work is to handle the broken robot case. For the approach to be robust to the loss of a robot will require solving two problems. First, the work assigned to that robot must be claimed and managed by

another robot when a robot becomes completely unresponsive. Second, all robots in the N_2 of the broken robot must be notified that a specific robot has been deemed nonresponsive, as the number of robots that choose to stay available for assigned work is dependent on the total number of functioning robots in N_2. Even then, if the number of functioning robots dips below the number given by Theorem 4, the structure may not be buildable by the remainder.

Another direction for future work would be to expand the experimental system to larger structures with larger teams of robots. In addition to being a more direct test of the assembly plans being generated, this would allow the methods to be compared using completion time of the structure as well as workload variance. These approaches could also be applied to other hardware platforms, including those used in Lindsey et al. (2011), Petersen et al. (2011), and Bolger et al. (2010).

Finally, I would like to directly compare these methods against other approaches on the same structures. This has been difficult because many approaches are tied to specific hardware or types of structures, which limits the comparisons that can be done.

Bibliography

S. Balakirsky, F. Proctor, T. Kramer, P. Kolhe, and H. I. Christensen. Using simulation to assess the effectiveness of pallet stacking methods. In Simulation, Modeling, and Programming for Autonomous Robots, pages 336–349, Berlin, Germany, Sep 2010.

A. Bolger, M. Faulkner, D. Stein, L. White, S.K. Yun, and D. Rus. Experiments in decentralized robot construction with tool delivery and assembly robots. In IEEE/RSJ International Conference on Intelligent Robots and Systems, 2010.

R.L. Brooks. On colouring the nodes of a network. In Proceedings of Cambridge Philosophical Society, volume 37, pages 194–197, 1941.

L Chaimowicz, T Sugar, V Kumar, and M F M Campos. An Architecture for Tightly Coupled Multi-Robot Cooperation. In Proc. IEEE Int. Conf. on Rob. & Autom., pages 2292–2297, Seoul, Korea, May 2001.

K.S. Chang, R. Holmberg, and O. Khatib. The augmented object model: cooperative manipulation and parallel mechanism dynamics. In International Conference on Robotics and Automation (ICRA00), volume 1, pages 470–475, Apr 2000.

T.H. Cormen, C.E. Leiserson, and R.L. Rivest. Introduction to Algorithms. MIT Press, 1990.

R. D'Andrea. Flight assembled architecture, 2011. URL `http://raffaello.name/dynamic-works/flight-assembled-architecture/`.

M. Dorigo, G. Di Caro, and L. M. Gambardella. Ant algorithms for discrete optimization. Artificial Life, pages 137–172, 1999.

William C. Evans, Grgory Mermoud, and Alcherio Martinoli. Comparing and modeling distributed control strategies for miniature self-assembling robots. In in the Proc. of the 2010 Int. Conf. on Robotics and Automation (ICRA10), pages 1438–1445, Anchorage, AK, May 2010.

J Fink, M Ani Hsieh, and V Kumar. Multi-robot manipulation via caging in environments with obstacles. In Proc. IEEE International Conference on Robotics and Automation (ICRA08), pages 1471–1476, Pasadena, CA, May 2008.

B. Gerkey and M. Mataric. A formal analysis and taxonomy of task allocation in multi-robot systems. International Journal of Robotics Research, 23(9):939–954, 2004.

Alexander Grushin and James A. Reggia. Automated design of distributed control rules for the self-assembly of prespecified artificial structures. Robotics and Autonomous Systems, pages 334–359, 2008.

F Heger and S Singh. Robust robotic assembly through contingencies, plan repair and re-planning. In Proceedings of ICRA 2010, May 2010.

M. Ani Hsieh and Josh Rogoff. Complexity measures for distributed assembly tasks. In Proc. of the 2010 Performance Metrics for Intelligent Systems Workshop (PerMIS09), Baltimore, Maryland, Sept 2010.

http://www.kivasystems.com. Kiva systems website.

http://www.leonardodavincisinventions.com/leonardo-da-vinci-models/leonardo-da-vincis-self-supporting bridge/.

E.G. Jones, M.B. Dias, and A. Stentz. Time-extended multi-robot coordination for domains with intra-path constraints. Autonomous Robots, 30(1):41–56, 2011.

Y. Khaluf and F. Rammig. Task allocation strategy for time-constrained tasks in robots swarms. In European Conference on Artificial Life (ECAL'13), Sept 2013.

B. Khoshnevis. Automated construction by contour crafting related robotics and information technologies. Journal of Automation in Construction Special Issue: The best of ISARC 2002, 13:5–19, 2004.

E Klavins. Programmable Self-Assembly. In Control Systems Magazine, volume 24, pages 43–56, August 2007.

Quentin Lindsey and Vijay Kumar. Distributed construction of truss structures. In Proc. of Workshop on the Algorithmic Foundations of Robotics (WAFR), 2012.

Quentin J. Lindsey, Daniel Mellinger, and Vijay Kumar. Construction of cubic structures with quadrotor teams. Robotics: Science and Systems, June 2011.

M J Mataric, M Nilsson, and K Simsarian. Cooperative Multi-Robot Box-Pushing. In Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS95), pages 556–561, Pittsburgh, Pennsylvania, August 1995.

L Matthey, S Berman, and V Kumar. Stochastic Strategies for a Swarm Robotic Assembly System. In Proc. 2009 IEEE International Conference on Robotics and Automation (ICRA09), pages 1953–1958, Kobe, Japan, May 2009.

N. Michael, J. Fink, and V. Kumar. Cooperative manipulation and transportation with aerial robots. Autonomous Robots, 30(1):73–86, Sept 2011.

N. Napp and R. Nagpal. Robotic construction of arbitrary shapes with amorphous materials. In International Conference on Robotics and Automation (ICRA'14), 2014.

G A S Pereira, V Kumar, and M F M Campos. Decentralized Algorithms for Multirobot Manipulation via Caging. In International Journal of Robotics Research, volume 23, pages 783–795, Nice, France, December 2004.

K. Petersen, R. Nagpal, and J. Werfel. Termes: an autonomous robotic system for three-dimensional collective construction. In Robotics: Science and Systems VII, 2011.

G. Pini, A. Brutschy, C. Pinciroli, M. Dorigo, and M. Birattari. Autonomous task partitioning in robot foraging: an approach based on cost estimation. Adaptive Behavior, 21(2): 118–136, 2013.

V. Rai, A. van Rossum, and N. Correll. Self-assembly of modular robots from finite number of modules using graph grammars. In Proceedings of the International Conference on Intelligent Robots and Systems (IROS11), San Francisco, CA, Sep 2011.

A. C. Sanderson, H. Zhang, L. S. Homen De Mello, Amla Arthur C. S, Arthur C. S, Hui Zhang, Hui Zhang, Luiz S. Homem De Mello, and Luiz S. Homem De Mello. Assembly sequence planning. AI Magazine, 11:62–81, 1990.

J. A. Sauter, R. Matthews, H. V. D. Parunak, and S. Brueckner. Evolving adaptive pheromone path planning mechanisms. In Autonomous Agents and Multi-Agent Systems (AAMAS02), pages 434–440, Bologna, Italy, 2002.

T.R. Schoen and D. Rus. Decentralized robotic assembly with physical ordering and timing constraints. In International Conference on Intelligent Robots and Systems (IROS'13), Nov 2013.

M. Schuster, R. Bormann, D. Steidl, S. Reynolds-Haertle, and M. Stilman. Stable stacking for the distributor's pallet packing problem. In Intelligent Robots and Systems (IROS10, pages 3646–3651, 2010a.

Martin Schuster, Richard Bormann, Daniela Steidl, Saul Reynolds-Haertle, and Mike Stilman. Stable stacking for the distributor's pallet packing problem. In Proc. 2010 Int. Conf. on Intelligent Robots and Systems (IROS10), Taipei, Taiwan, Oct 2010b.

J. Spletzer, A.K. Das, R. Fierro, C.J. Taylor, V. Kumar, and J.P. Ostrokowski. Cooperative localization and control for multi-robot manipulation. In International Conference on Intelligent Robots and Systems (IROS01), volume 2, pages 631–636, Oct 2001.

D. Stein, R. Schoen, and D. Rus. Constraint-aware coordinated construction of generic structures. In IEEE International Conference on Intelligent Robots and Systems (IROS11), San Francisco, CA, Sep 2011.

T Sugar and V Kumar. Multiple Cooperating Mobile Manipulators. In Proc. 1999 IEEE International Conference on Robotics and Automation (ICRA99), pages 1538–1543, Detroit, Michigan, May 1999.

J Werfel and R Nagpal. Three-dimensional construction with mobile robots and modular blocks. In International Journal of Robotics Research, volume 27, pages 463–479, March 2008.

J. Werfel, K. Petersen, and R. Nagpal. Designing collective behavior in a termite-inspired robot construction team. Science, 343(6172):754–758, Feb 2014.

A. Yamashita, T. Arai, J. Ota, and H. Asama. Motion planning of multiple mobile robots for cooperative manipulation and transportation. IEEE Transactions on Robotics and Automation, 19(2):223–237, Apr 2003.

S. K. Yun and D. Rus. Adaptation to robot failures and shape change in decentralized construction. In Proc. of the Int. Conf. on Robotics & Automation (ICRA10), pages 2451 – 2458, Anchorage, AK USA, May 2010.

S. K. Yun, M. Schwager, and D. Rus. Coordinating construction of truss structures using distributed equal-mass partitioning. In Proc. of the 14th International Symposium on Robotics Research, Lucerne, Switzerland, Aug-Sept 2009.