VISITING PEBBLES ON RECTANGULAR GRIDS
Coordinating Multiple Robots in Mobile Fulfilment Systems

by

Cornelis Francois van Eeden

Submitted in partial fulfilment of the requirements for the degree

Master of Engineering

in the

Department of Mechanical Systems Engineering

Faculty of Engineering

UNIVERSITY OF MIYAZAKI

September 2017

ABSTRACT

VISITING PEBBLES ON RECTANGULAR GRIDS
Coordinating Multiple Robots in Mobile Fulfilment Systems

by

Cornelis Francois van Eeden

Supervisor:	Associate Professor Geunho Lee, Ph.D.
Department:	Mechanical Systems Engineering
University:	University of Miyazaki
Degree:	Master of Engineering
Keywords:	Pebble Motion on Graphs, Multi-Agent Path Finding, Starvation-freedom, Mobile Fulfilment Systems, Automatic Guided Vehicle

Over the past decade, the application of multi-robot systems to e-commerce distribution centres for pick and pack operations has become a billion dollar business. Multi-agent path finding and coordination is one of the key performance affecting sub-systems of the overall robotic order fulfilment process. The purpose of multi-agent path finding and motion coordination is to plan and coordinate the motions of multi-vehicle systems such that all vehicles reach their assigned goals safely. Much research has focussed on solving the multi-agent path finding problem in a general way. As a result, researchers have considered a system wide goal state where all vehicles are at their goal destinations in some final time. In this work, an algorithm is designed specifically for order fulfilment in e-commerce. Though the algorithm is designed for order fulfilment, it is generally applicable to point-to-point transport in autonomous multi-vehicle systems. In point-to-point transport, all robots do not necessarily need to be at their destination locations in the same final time. This is the key assumption of this work. Designing the algorithm based on the correct assumptions for its application domain allows for an elegant solution. The resulting solution is referred to as the visiting pebble motion on rectangular grids. The algorithm is efficient and analytical performance guarantees are given. More specifically, an asynchronous, starvation-free, semi-decentralized, scalable multi-agent path finding algorithm is presented. This thesis takes an incremental approach to developing the solution, starting with a high level of abstraction and gradually progressing to a low level so as to facilitate industrial adoption. As a result, the final algorithm takes the constraints of real robot dynamics and collision avoidance into account and is capable of operating under asynchronous conditions while providing analytical performance guarantees.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

A∗	“A-star” path-finding algorithm
BFS	Breadth First Search
CPF	Cooperative Path Finding
GUI	Graphical User Interface
MAPF	Multi-Agent Path Finding
MFS	Mobile Fulfilment System
MRPP	Multi-Robot Path Planning
MVS	Multi-Vehicle Systems
PMG	Pebble Motion on Graphs

MATHEMATICAL SYMBOLS & NOTATION

Symbols:

A	Required grid area
a	Aspect ratio of a grid
a_i	Aspect ratio of s_i
c	Number of moves associated with unblocking a robot's path
D	Diameter of a graph
d_i	Diameter of s_i
E	Edge set of a graph
f_i	Fraction of nodes contained in s_i
G	A graph
g	Asymptotic worst-case growth rate
g_r	Required moves to a robot from it's current node to it's destination node
h	Height of a grid
k	Number of robots on a graph
l	Distance between adjacent nodes
N	Number of nodes in a graph
n_i	Number of nodes on s_i
n_s	Number of rectangular sections in a map
n_{turns}	Number of required turns
o_i	Current node of r_i
p_i	Goal node of r_i
r	Rotation envelope radius
r_i	Robot i
s	Inter-robot separation distance
s_i	Rectangular section i
V	Vertex set of a graph
w	Width of a grid
δ	Distance from start node
ρ	Traffic density

Operators:

$A \leftarrow B$	Assignment operator. The value of B is assigned to A
$A \rightarrow B$	A implies B
$f(\cdot)$	"function of"
$O(\cdot)$	"Order of" / Big O class
$w(\cdot)$	"Waits for" relationship
$\lvert \cdot \rvert$	Cardinality of a set
$\{\cdot\}$	Set

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER 1 INTRODUCTION

1.1 PROBLEM STATEMENT AND OVERVIEW

In 2006, Kiva Systems introduced the first commercially implemented autonomous Multi-Vehicle System (MVS) for pick and pack operations in e-commerce distribution centres (commonly referred to as warehouses). Their solution is referred to as a Mobile Fulfilment System (MFS). At the time, existing research in multi-agent systems did not satisfactorily address many of the problems they faced. As a result, mostly "textbook implementations of standard algorithms" were used [2].

In recent times, many researchers have addressed the issue of multi-agent path finding, which is one of the central components of MFS. The performance of a MFS is heavily dependent on the path finding algorithms used. Though many researchers claim that their work is applicable to applications such as MFS, their assumptions are seldom amenable to the underlying motion coordination problem specific to MFS. Furthermore, many works do not consider the constraints imposed by industrial adoption. Most existing research assumes that robots are initially at random positions and all robots reach their goal destinations at some final time. In MFS, there is no requirement that all robots must reach their destinations at the same time. Instead, it is only required that each robot eventually reaches its destination at least once during the course of execution.

Little is known about the algorithms used in real MFS. From the available literature, the algorithms used are not guaranteed to be deadlock-free and performance guarantees are not provided. Most works in the field of Autonomous Ground Vehicles (AGVs) assume that planning is done by a central unit and robots execute the plan under supervisory control. The planning algorithms usually make strong assumptions on the synchrony of the AGVs. Asynchrony is, however, prevalent in any real system and, therefore, has to be accounted for. Naturally, the following questions arise:

- Is it possible to ensure that all robots reach their assigned destinations regardless of traffic volume?
- Is it possible to distribute the computational load in such a way that real-time implementation is possible and scalable to large teams of robots?
- Is it possible to design algorithms that do not require supervisory control and strong synchrony assumptions on the system?

1.2 THESIS OBJECTIVES

The objectives of this thesis are to:

1. Show that MFS has unique requirements in terms of motion coordination and that existing methods do not provide satisfactory performance guarantees, robustness and scalability.
2. Determine the appropriate design requirements for motion coordination in MFS, taking industrial adoption into account.
3. Design robust and scalable algorithms tailored for MFS which deliver high performance.
4. Implement the algorithms in an appropriate simulator.
5. Provide analysis of the designed algorithms in order to provide worst-case performance guarantees and assess the completeness of the algorithms.
6. Provide experimental data and discussions to support the analysis and design decisions.

1.3 THESIS CONTRIBUTIONS

This thesis provides the following contributions:

1. The specific motion coordination requirements for MFS are determined.
2. An algorithm is described that solves the motion coordination problem specifically for MFS.
3. The asymptotic worst case upper bound on the required number of moves is analysed and verified by experiments.
4. Analysis of the algorithms finds them to be complete for a subset of graphs that are appropriate to MFS workspaces.
5. A parallel version of the algorithm is provided which allows multiple robots to move simultaneously.
6. The provided algorithms are shown to be scalable. Semi-decentralized implementation leads to low computational effort per robot.
7. An asynchronous version of the parallel algorithm is provided and implemented in a continuous time simulation.
8. The starvation-freedom of the algorithms is proven. This guarantees that all tasks will eventually complete, regardless of traffic density, as long as a single unoccupied grid position is available in the workspace.

1.4 THESIS OUTLINE

Chapter 2 introduces MFS and the open-source ALPHABET SOUP simulator. Though MFS presents many opportunities to study various dynamic resource allocation problems, it is shown that existing motion coordination algorithms do not adequately solve the problems specific to MFS and, therefore, deserve foremost attention. The scope of this work is thus limited to motion coordination in MFS and related works are surveyed in Chapter 2.

Chapter 2 introduces the popular 15 puzzle which is closely related to the problems posed by automated manufacturing and warehousing systems. The generalization of the 15 puzzle to any arbitrary layouts is referred to as “pebble motion on graphs”, where the workspace is represented by an undirected graph. Chapter 2 concludes with a proposal to alter the assumptions of the classical pebble motion problem to suit the MFS scenario. The rest of this work incrementally develops this idea in order to arrive at a solution that is appropriate for industrial adoption.

The altered assumptions for the MFS pebble motion problem are dubbed “visitors’ rules”. Chapter 3 introduces the visitors’ rules for pebble motion on rectangular grids when a single unoccupied space is available. Chapter 4 extends the work from chapter 3 to the case where more than a single unoccupied space is available. Chapter 5 provides a parallel version of the algorithms developed in chapter 3 and 4. Chapter 6 provides a continuous time version of the algorithm presented in Chapter 5 that is capable of operating under asynchronous conditions. Finally, chapter 7 provides a summary of this work and proposes possible complementary further work.

CHAPTER 2 LITERATURE STUDY

2.1 INTRODUCTION

In recent times, wireless communication devices, micro-processors and sensors have become powerful and inexpensive enough to enable the use of autonomous mobile robots in various real world applications. Given these advances, it is expected that, in the future, thousands of autonomous robots will be able to complete complex tasks cooperatively. As such, multi-robot systems has been the focus of much research. Even though research has demonstrated large systems of cooperative robots, the application domains thus far have not been everyday events.

Until recently, “Real, mundane applications with more than a few vehicles have been lacking.” [2]. This changed in 2006, when the KIVA MFS demonstrated the commercial feasibility of large-scale autonomous systems in a pick and pack warehousing environment. MFS is a new type of automated storage and part-to-picker order picking system that is well-suited to e-commerce distribution centres which experience strong fluctuations in demand and carry a wide variety of small products [17].

Currently, MFS represents the state-of-the art in commercially implemented MVS. Although MFS technology has reached a level of maturity that allows commercial adoption, many interesting and challenging research areas remain leaving much room for improvement of these systems.

2.2 CHAPTER OBJECTIVES

The objectives of this chapter are to provide the reader with an overview MFS and its components, introduce the problem of motion coordination in MFS, introduce and discuss the relevant approaches for solving motion coordination in MFS and, finally, show that the existing approaches leave room for further research.

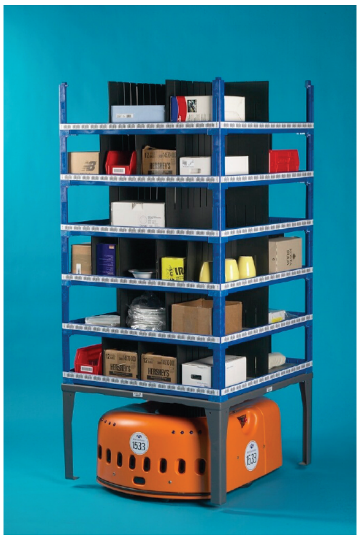
2.3 THE MOBILE FULFILMENT SYSTEM

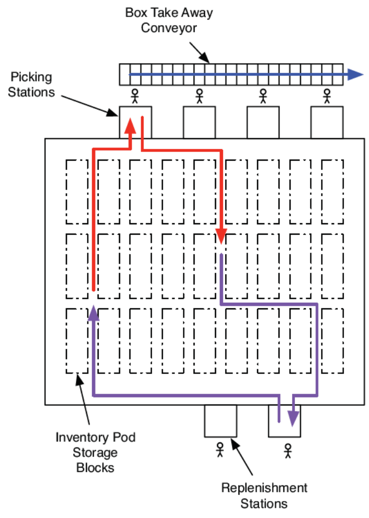
The conventional system of order fulfilment requires pick/pack workers to walk through the distribution centre with a list of inventory and retrieve ordered items by hand. Traditionally, warehouse automation came in the form of inflexible carousels and conveyor belts which require significant capital investment and take time to implement. It was the inflexibility, cost and complexity of the existing automated materials handling systems that inspired Mick Mountz to found KIVA systems in 2003 [9]. Mountz was first exposed to the realities of automated material handling equipment during his time at the failed online grocery home delivery company Webvan.

In MFS, items of inventory are stored in shelves called mobile storage pods. Instead of sending workers to fetch the desired items, mobile robots are employed to retrieve the pods that contain the items. The height of the robots is low enough to fit under the pods as can be seen in Figure 2.1(a).

As seen at the top of Figure 2.1(b), human pick workers are responsible for picking the items off the pods. The storage pods have barcodes that mark every location within the pod. Every item that is removed from the pods is scanned using laser scanners. In this way the warehouse management software keeps track of the inventory levels. Similarly, when the storage pods are low on stock, they are replenished at the replenishment stations shown at the bottom of Figure 2.1(b). Upon replenishment, every product is scanned into the system so that the relevant pod and storage location are associated with the product being carried. In this manner, the warehouse management software keeps track of where the products are in the warehouse.

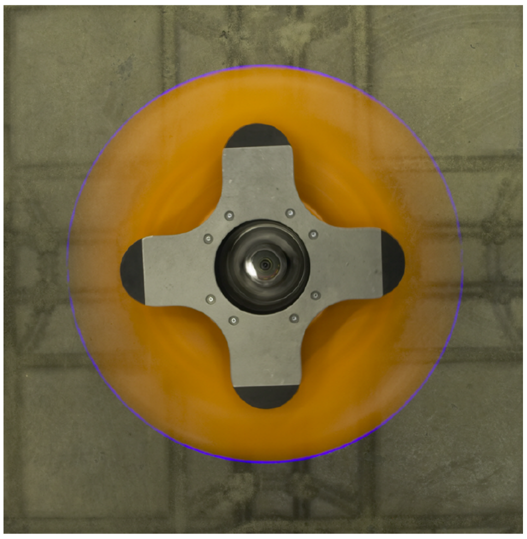
The robots employ a screw mechanism to lift the pods. The screw mechanism is actuated by a dedicated geared motor. As shown in Figure 2.2(a), the robots counter-rotate their lifting devices whilst rotating themselves under the storage pods. This causes the screw to extend upwards and lift the storage pods off the ground whilst maintaining a zero angular velocity between the lifting device and the base of the pod. The ability to counter-rotate the screw mechanism also serves a second purpose: the pods are purely translated during transport; whereas the robots rotate and translate to transport them. This


(a) An inventory pod being carried by a mobile robot



(b) A typical KIVA MFS warehouse layout

Figure 2.1. Components of the KIVA solution [40]


(a) Rotating robot counter-rotating its lift device [41]

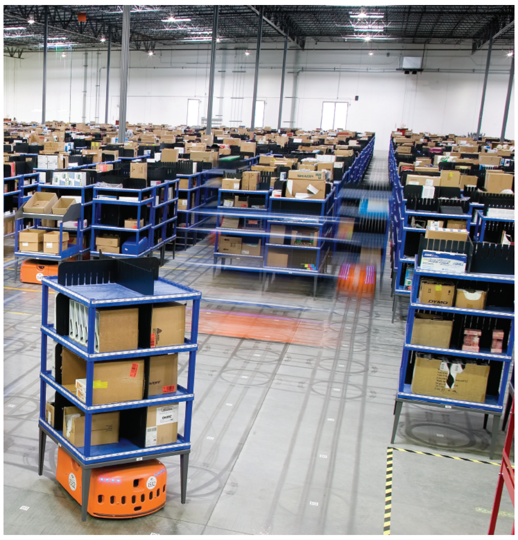

(b) A section of a KIVA equipped warehouse [4]

Figure 2.2. The KIVA solution in motion

allows for compact storage, as purely translational square pods have a smaller storage footprint than rotational ones.

At the centre of Figure 2.2(a), an upward facing camera is seen to be mounted on the robot. This camera is used to identify the storage pods by scanning 2D barcodes placed underneath them. Similarly, robots localize themselves in the warehouse using a downward facing camera (not shown) and fiducial markers on floor as seen in Figure 2.2(b).

The robots' traction system is of the well-studied differential drive type. In fact, apart from the lifting system, the robots are fairly simple. The robots are "...to a warehouse what taxis are to a city. The complexity of the warehouse is truly expressed in the warehouse control software that runs on the servers." [40].

In MFS, the warehouse control software has two functions, namely:

1. Resource allocation: the software assigns orders to picking stations, robots to pods, and pods to either picking or replenishment stations.

2. Assistance in coordinating robot motion: robots use a space reservation system to reserve sections demarcated by the fiducial markers. The reservation status of these sections are communicated via standard Wi-Fi routers and is disseminated by the servers. The space reservation system ensures that two robots cannot occupy the same section at the same time, thereby preventing collisions.

The servers manage the resource allocation in a centralized way. On the other hand, the motion planning is semi-decentralized: given the information supplied from the servers, robots independently plan and execute their motions [4].

The robots are equipped with bumper pressure and infrared sensors. Because of the space-reservation system, the sensors carried by the units are a safety measure and are not used under normal operating conditions. These sensors serve to avoid injury to humans who wander into the working area of the robots, or prevent damage to products that accidentally fell from the pods.

According to field data and customer testimonials, the benefits of MFS [9] include:

1. Efficiency: due to low lighting and air-conditioning needs, the cost of recharging the robots are recovered and electricity bills are reduced.

2. Accuracy: due to the scanning system, erroneous shipments are reduced.

3. Flexibility and (rapid) scalability: the required infrastructure for expansion includes robots, pods and 2D fiducial markers. This keeps the time and cost for instalment of extra capacity low in comparison to fixed structures required by other automation approaches.

4. Quality of life: warehouse employees have a better working environment due to reduced noise levels (compared to all conveyor systems), less physical fatigue and increased safety.

5. Redundancy: failure of a robot, pod or even picking station does not cause the system to halt. Instead, the capacity of the system is temporarily affected depending on which part of the system has failed. This also eliminates the need for system-wide downtime.

6. Reduced training time: the required training time is very short, typically 2 to 3 days for new employees.

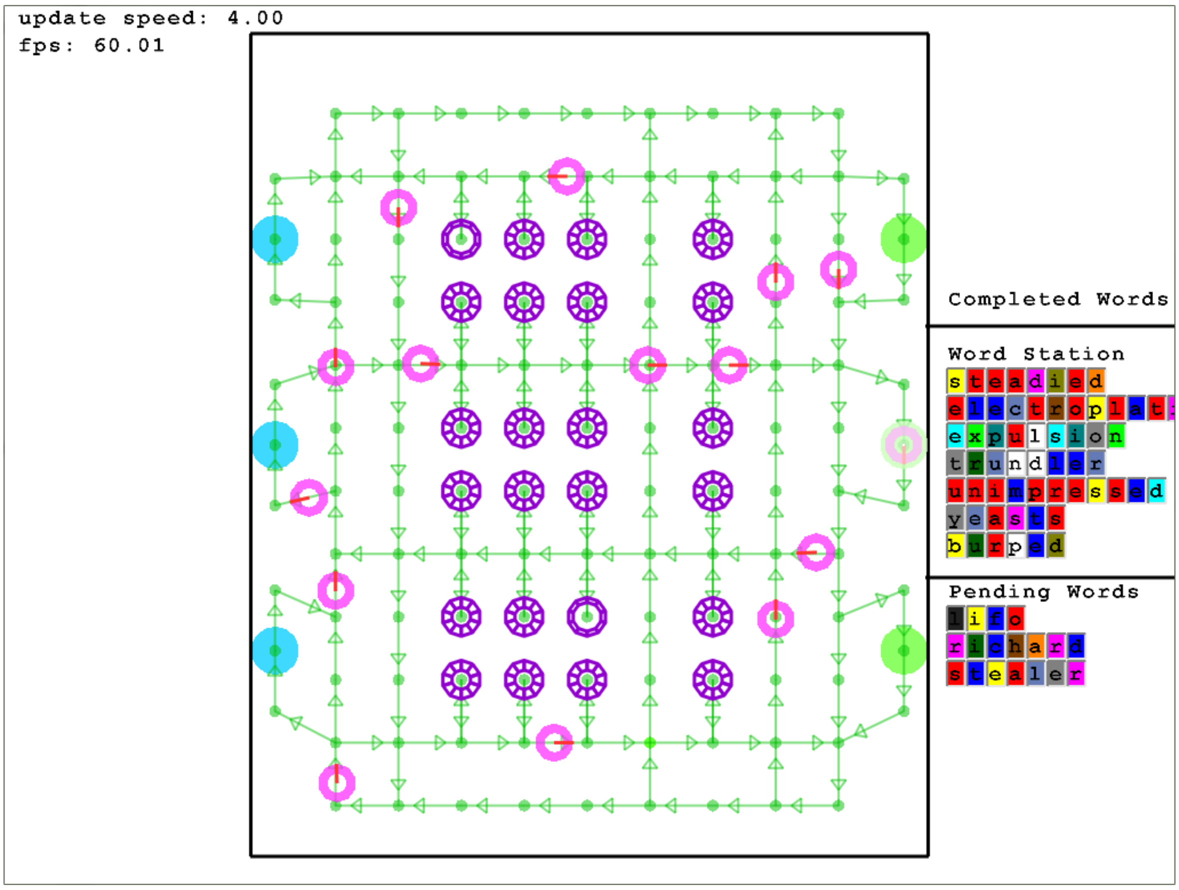
7. Increased productivity: MFS has been found to increase productivity 2 to 3 times in pick-to-conveyor operations.

A MFS typically has a payback period of 2-3 years [2]. Until very recently, KIVA was the only company supplying this kind of materials handling technology. Up to February of 2012, KIVA had managed to raise $33M in funding. However, in 2012 Amazon acquired KIVA for $775M and subsequently re-branded to Amazon robotics. Amazon decided to stop providing the technology to competitors, so the contracts with existing KIVA customers were not renewed after they ran out. It is estimated that there are around 30000 robots currently in Amazon's service [8], making it clear that the acquisition resulted in a gap in the market.

Only very recently did new start-ups such as OTTO MOTORS [5] and fetch robotics [6] emerge to fill the needs of industry with technologies that resemble the KIVA system. Several robotics companies are announcing that their products are ready for real-world applications [7], but the success of these systems has yet to be established.

2.4 ALPHABET SOUP AND POSSIBLE RESEARCH DIRECTIONS

ALPHABET SOUP [42] is an open source simulator written in the Java programming language, introduced by Hazard, Wurman and D'Andrea. Its intended purpose is to study 1) resource allocation and 2) motion coordination in the context of MFS. In ALPHABET SOUP, items of inventory are represented by coloured tiles with letters on them. Orders are represented by words taken from a dictionary: every order represents a list of letters to be assembled at a picking station and shipped. A modified version of ALPHABET SOUP's user interface is shown in Figure 2.3.



```
update speed: 4.00
fps: 60.01
```

Completed Words

Word Station

steadied
electroplat
expulsion
trundler
unimpressed
yeasts
burped

Pending Words

lifo
richard
stealer

Figure 2.3. Initial configuration of warehouse and robots in ALPHABET SOUP

Warehouses are cost centres. Therefore, the key objective in managing a warehouse is to minimize the cost of operation. The authors hoped that releasing ALPHABET SOUP would facilitate research into coordination algorithms which maximize the sustainable order completion rate whilst minimizing the amount of required resources. With this goal in mind, the authors [2, 40, 41, 42] propose the following questions in terms of resource allocation:

- Where should storage pods be stored in the warehouse?
- Which pods should be taken to which stations?
- In which pods should new items be stored?
- To which picking stations should each order be assigned to?
- To which replenishment stations should incoming products be assigned to?

ALPHABET SOUP has minimal implementations of coordination algorithms. Based on their industrial experience with the KIVA system, the authors expect that coordination algorithms should offer one to two orders of magnitude of improvement in terms of the order completion rate.

The authors of ALPHABET SOUP make the observation that, though much research has been done on multi-agent coordination, a great deal thereof has been in the context of contrived problems. The motions of the robots must be such that collisions and congestion are avoided while facilitating high product throughput. In the multi-agent research community, much focus has been placed on solving the difficult problem of navigation in unknown environments. In contrast, in the context of MFS, mobile robots operate in known and controlled environments.

In spite of the relative simplicity of the MFS's motion coordination requirements and the apparent availability of multi-agent motion coordination research, the motion coordination in ALPHABET SOUP consists of standard A^* [27] path-finding and reactive obstacle avoidance. The reports generated from the provided release of ALPHABET SOUP show very high collision rates. In practice, even a single collision in the warehouse could have catastrophic effects. These collisions are probably due to an erroneous implementation. Regardless of the cause of the collisions, the motion coordination scheme deserves replacement in lieu of the abundance of alternative approaches.

2.5 SCOPE

In spite of MFS's commercial success [8], it is unclear how these large robot teams coordinate their movements. Therefore, although MFS encompasses many rich research areas, the scope of this work is limited to the study of appropriate motion coordination algorithms for use in MFS.

2.6 COORDINATING MOTION IN MOBILE FULFILMENT SYSTEMS

In this section, the available literature for coordinating MVS is studied. In addition, the approaches that are most amenable to the requirements of MFS are determined.

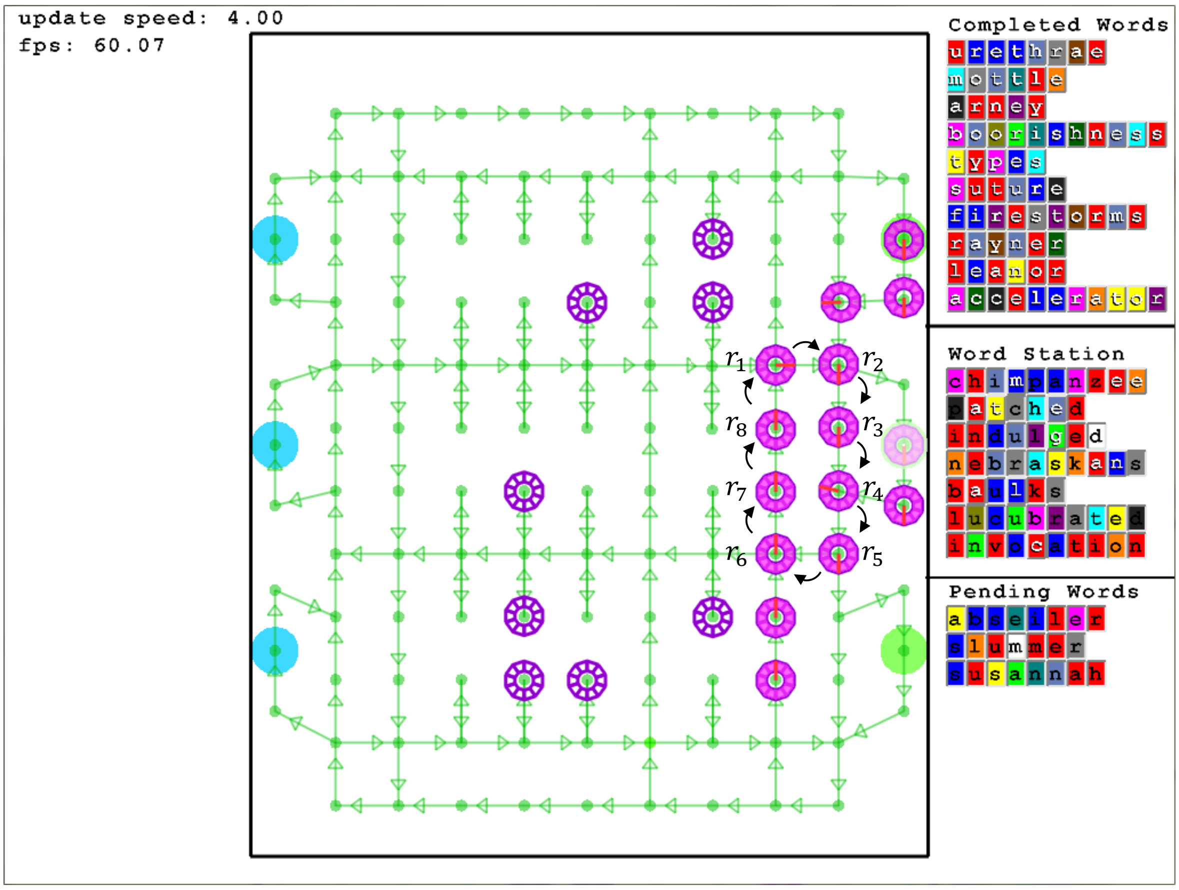
2.6.1 Resource Allocation Approaches

In the previous section, a modified version of ALPHABET SOUP was introduced. This version is used to demonstrate how an infinite wait, called a deadlock, can occur in MFS. In this version of ALPHABET SOUP, the following changes were made:

- Visualization of the underlying directed graph used for path finding and space reservation.
- Directed 'highways' to avoid head-to-head conflicts between robots, as introduced in [26].
- Implementation of a simple space reservation system to avoid collisions.

The space reservation system works as follows: before a robot approaches a node, it must reserve the node first. Only nodes that are not already reserved may be reserved. If the node is already reserved, the robot waits until the node is freed before attempting to reserve it. When two robots compete to reserve a node, ties are broken by a pre-defined priority in the server system. Robots free their previously reserved node once it reaches the next node on its path.

Introduction of so called highways (by using a directed underlying graph) resolves deadlocks that would occur from head-to-head conflicts. As seen in Figure 2.4, a deadlock could also result due to loops in the underlying graph. Let $w(\cdot)$ denote a "waits for" relationship and $r_1, r_2 \ldots r_8$ denote robot 1 to robot 8. It is easy to see that $r_1 = w(r_2), r_2 = w(r_3)$ ... $r_8 = w(r_1) \rightarrow r_1 = w(r_1)$.



```
update speed: 4.00
fps: 60.07
```

Completed Words
urethrae
mottle
arney
boorishness
types
suture
firestorms
rayner
leanor
accelerator

Word Station
chimpanzee
patched
indulged
nebraskans
baulks
lucubrated
invocation

Pending Words
abseiler
slummer
susannah

Figure 2.4. Deadlocked configuration of warehouse and robots

It is a known requirement that the motion coordination of multiple vehicles should be deadlock free [18]. It is generally accepted that the deadlocks should be automatically resolved [3]. Some recent research in multi-agent coordination do not address these issues [43, 44], while others specifically focus on them [19, 45, 46, 47, 48].

Banker's algorithm [55] is a well known algorithm originally developed to prevent deadlocks in distributed computing systems with shared resources. In [19, 45, 46], authors adapt these ideas for use in MVS. In this paradigm, space is regarded as a shared resource. The key idea is to check that the system never enters an 'unsafe' state by ensuring that every requested action is safe before allowing it. An unsafe state is one where the allocated resources exceed the available resources. In the situation shown in Figure 2.4, this could for example have been achieved by preventing r_1 from entering the loop before r_8 had exited it. In [19, 47], the authors devise a check to ensure that the system never enters a state that will inevitably lead to a deadlocked state.

Another approach is to model the MVS as a Petri net [47, 48], a tool used in the analysis of distributed discrete dynamical systems. The authors provide little detail about the completeness and efficiency of their approach and also do not discuss its algorithmic complexity.

The works mentioned here show that ideas from distributed computing can be applied to MVS. These ideas include deadlocks, live-locks and starvation-freedom. The notion of a deadlock has already been introduced as a system state where all robots wait infinitely for another robot to move. Live-locks are similar to deadlocks because they prohibit the system from completing its assigned tasks. A live-locked situation is one where robots move in an oscillatory manner. This implies that, even though robots are moving, no robot is actually sequentially progressing towards its goal. The notion of live-locks leads to the important system property of starvation-freedom. If a robot does not reach its goal destination in a finite time, it is said to be starving. Therefore, starvation-freedom of the system implies that all robots will eventually complete their tasks (reach their goal destinations).

It is important to note that in all the works cited in this subsection, the path planning is done off-line and deadlock avoidance is regarded as a pure scheduling problem. Because the path planning is done off-line, live-locks cannot occur and deadlock freeness implies starvation-freedom. Live-locks are a problem that occur when robots myopically try to resolve conflicts by re-planning their paths. Roozbehani and D'Andrea [26] present an example of a live-lock situation, which is discussed later in this chapter.

In distributed systems, critical sections of code are those that may only be accessed by one process at a time. In other words, allowing any one process to execute a critical section mutually excludes all other processes from executing it. Solving this mutual exclusion (mutex) problem entails the design of a method that brackets the critical sections of code by use of an entry and exit protocol. By analogy, if the processes are represented by robots in MVS and critical sections of code are represented by graph vertices, then the space-reservation system represents the entry and exit protocol. The mutual exclusion property therefore naturally leads to a safety property: occupancy of a node by any one robot mutually excludes occupancy by any other robot. As a result, mutex ensures that collisions will never occur. In order to avoid trivial solutions to the mutex problem, where no process ever executes any critical section, a liveness property is also required. Starvation-freedom is such a liveness property [56].

It is important to note that the notions of these distributed system concepts have been adapted to suit the MVS scenario. For example, in the study of mutex algorithms, starvation-freedom is guaranteed when a process that wants to access a critical section obtains mutex to access it within a finite time [56]. Eventually granting a robot access to a desired node in MVS does not necessarily guarantee progress because this does not exclude the live-lock scenario. This implies that eventual access to a shared resource does not guarantee progress in MVS in the same way that it does for distributed computing systems. Therefore, the ideas from distributed systems are applicable, but their exact definitions have been adapted to suit the scenario at hand.

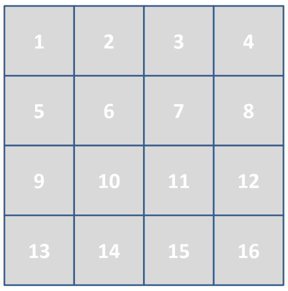
Finally, the resource allocation approaches mentioned here assume that the vehicles leave the workspace after completion of their missions. In practice, this would require that every robot's path to terminate in a reserved parking space. Such additional parking spaces consume space in the warehouse where it is a valuable commodity. Furthermore, these approaches model the continuous time dynamics of the system as a discrete dynamical system in the scheduling phase. This leads to a high accuracy tracking performance requirement during the execution phase [3] and should be accompanied by a synchronization method in the case where contingencies occur or the tracking performance is not good enough.

2.6.2 Multi-Agent Path Finding

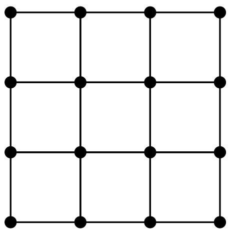
Abundant literature is available that addresses the problem of Cooperative Path Finding (CPF) [32, 34, 35, 36, 52], Multi-Agent Path Finding (MAPF) [13, 14, 15, 37], Multi-Robot Path Planning (MRPP) [16, 38, 51] and Pebble Motion on a Graph (PMG) [21, 34, 39]. CPF, MAPF, MRPP and PMG all address the problem of moving a set of objects from their initial locations to their goal locations without collisions in a spatially constrained environment [52]. In CPF, the said objects can be either agents or robots; in MAPF, the objects to be moved are agents; and, in MRPP, it is robots. Robots are agents; however, agents are not necessarily robots. In this work, the proposed algorithms are specifically intended for robots. This implies that the algorithms specifically need to account for their physical embodiment in space and time. Agents do not necessarily require these aspects to be taken into account, depending on the context of their manifestation.

Of all the definitions, PMG is the most general. In PMG, pebbles, which is an abstraction of any arbitrary object, are to be moved. PMG is a generalization of the famous 15 puzzle. In the 15 puzzle [10], 15 tiles are placed on a 4×4 grid, leaving one open space. The 15 puzzle board is numbered in row-major order to indicate the destination positions of each tile as shown in Figure 2.5(a). The board can conveniently be represented by an undirected graph as shown in Figure 2.5(b), where the nodes represent allowable board positions. The tiles are such that they can slide vertically or horizontally along the board and cannot be lifted from it. The allowable moves between positions can be represented as graph edges between the nodes. The game starts out in row-major configuration. To play the game, the tiles are first randomly shuffled. The objective of the game is to restore the board to its initial configuration as shown in Figure 2.6(a).

It has long been known that odd permutations of the puzzle are impossible to solve and all even permutations are solvable [23]. An example of an unsolvable configuration is shown in Figure 2.6(b). The first attempt to generalize the analysis of the 15 puzzle to arbitrary graphs was given by Wilson [20]. Wilson gives an efficient procedure for deciding the solvability (also called feasibility) for bi-connected graphs with one unoccupied vertex, but does not provide an analysis of the number of moves required to solve the puzzle. It was Kornhauser [1, 34] who introduced PMG and named it. Kornhauser gives an analysis for the case where there are less pebbles than vertices on any arbitrary graph. His results include a P-time feasibility decision procedure and the provided solution method requires $O(N^3)$ moves to reach a solution, where N is the number of nodes in the graph.

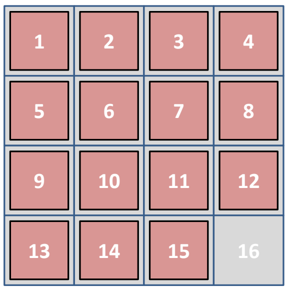


(a) 15 puzzle board indicating grid positions

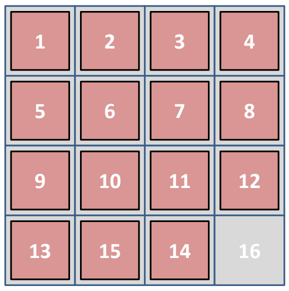


(b) A graph representation of the board

Figure 2.5. The 15 puzzle board and its graph representation



(a) A solved instance of the 15 puzzle



(b) An unsolvable instance of the 15 puzzle

Figure 2.6. Instances of the 15 puzzle

In PMG, algorithms are said to be complete if they are able to provide a solution to any problem for which a solution exists. As an example, works that require the underlying graph to be bi-connected [20, 50] are not complete because solvable instances of problems exist that do not manifest in bi-connected graphs. Kornhauser's analysis is, therefore, complete as it is capable of solving any solvable instance and provides a feasibility check. Though it would be useful to solve this problem optimally, both the problem of minimizing the number of moves when only one agent is allowed to move at a time, and minimizing the makespan when agents are allowed to move simultaneously have been shown to be NP-hard [39, 54].

As a result of its NP-hardness, optimal approaches scale very badly with the number of agents. Some very recent approaches [52] minimize the makespan (number of turns) with the critique that sub-optimal methods have dramatically longer solution sequences in densely populated scenarios. Considering the target application domain of MFS, it is observed that overfilling the environment with robots unavoidably leads to reduced throughput and should be avoided. However, providing for algorithms that are capable of solving densely populated scenarios and are always deadlock free is desirable for the sake of robustness. The solution length of sub-optimal approaches are likely to be close to optimal in their intended use due to low traffic density. Therefore, the solution length of sub-optimal approaches is permissible in this application domain and provide a scalable alternative to optimal ones.

Kornhauser's work gives a sub-optimal algorithm to the PMG problem. Despite the fact that their results are subsets of Kornhauser's [49], it was pointed out that sub-optimal solutions that are complete for certain sub-classes of graphs have recently been published in top-tier journals [24, 25, 30, 31]. It is speculated that this oversight occurred due to the fact that Kornhauser's work is not described in a single document and he also does not provide algorithmic descriptions. It was noted that it would require significant effort to construct algorithms from the proofs in his works. In response, recent attempts at reconstructing complete solutions have been given, notably that of de Wilde [15].

Similar to the works discussed in the subsection 2.6.1, the definition of discrete time steps assumes that all robots move at the same time. This means that a centralized planner and supervisory controller will be used, which implies synchronous execution. In fact, recent research [53] details the various issues with real-world adoption of MAPF and other centralized planning methods. In contrast to approaches referred to in the previous section, if the PMG solver is able to find a solution, it is live and deadlock free. This is, however, not necessarily the case for all CPF, MAPF and MRPP methods.

Finally, some research [11] studies cases where robots do not have unique identification numbers and moving any one of a number of anonymous robots to a specified location is acceptable. In the MFS context, tasks are assigned to a specific robot which will need to visit specific locations (specific pods need to go to specific stations). While it is true that, in the resource allocation phase, robots can arbitrarily be assigned tasks, once a task is assigned, they are no longer anonymous.

2.6.3 Adaptive Highways on a Grid

One work that is of notable importance, due one of the co-author's industrial experience in development of the KIVA MFS, is that of Roozbehani and D'Andrea [26] called "Adaptive Highways on a Grid". Amazon and KIVA combined have had over 14 years and large budgets to develop and improve their robot coordination scheme. Due to Amazon's decision to keep the technology to themselves, not much is known about how the robots are coordinated. It is speculated that this work is the only known work to detail the motion coordination in industrially adopted MFS.

Furthermore, it is the only work that describes the dynamic model of the robots in detail. The robots are governed by a second order dynamic model, with a maximum acceleration and velocity constraint. The robots act as finite state machines which must transition through a stationary state in order to switch between linear and rotating motions.

A central hub is employed to communicate space reservations and trajectories between the robots. This hub imposes a fixed ordering on the robots in terms of communication, space reservation and path planning. As implemented in subsection 2.6.1, robots must reserve space before using it. The space is partitioned into cells that are large enough contain the vehicle envelope completely. Once again the cells are represented as vertices on a graph and the allowable moves between them are represented by the connecting edges. This results in a four-connected grid type map.

This work is furthermore unique in the sense that robots are not required to visit a single destination, but rather a set of randomly selected locations that are updated over time. In other words, robots cyclically visit random locations on a grid and it is not required that they occupy their target locations at the same time, as assumed in subsections 2.6.1 and 2.6.2. This allows for solving the problem of cyclically sending each robot in a group of robots to some target location to either pick up or deliver inventory as typically required in MFS.

The authors address the problem of finding a set of trajectories that maximize the average speed of the vehicles in real time. This is achieved using a two-layer control architecture in which the trajectory optimization is done locally and information is distributed via the central hub. Local planners generate discrete trajectories given the information about other robots' trajectories from the central hub, and a low level controller provides a continuous execution in order to implement and preserve the properties of the discrete plan.

Although their approach is well-suited to MFS, there are two aspects that could be improved. Firstly, some attempt is made to avoid deadlocks in an ad hoc manner, but their approach is not provably starvation free. Secondly, the algorithmic implementation of their overall scheme is not clearly stated. The ALPHABET SOUP implementation in subsection 2.6.1 is an instance of the "Fixed Highways Algorithm" without any deadlock handling mechanism.

2.7 RESEARCH GAP

Resource allocation approaches [19, 45, 46, 47, 48] are not complete and require conservative means to accommodate robots that have completed their tasks. The computational requirement for deadlock checking and its centralized nature limit the scalability of these approaches. Furthermore, these approaches often prune away a lot of the usable solution space in a conservative way leading to long solutions in terms of makespan.

Recent sub-optimal PMG approaches are complete and provide starvation free solutions at low computational cost. However, for both optimal and sub-optimal approaches, the underlying assumption that all robots end at their final destinations is not appropriate to MFS. Firstly, letting completed robots stand idle while all robots are not yet complete incurs opportunity costs: the completed robots could be assigned new tasks and do useful delivery work instead of waiting. In order to avoid the costs of being idle, cyclic task assignment would induce re-computation upon every robot reaching its destination and a large portion of the computation is wasted by calculating parts of the plan that will inevitably be overridden later. Apart from incurring unnecessary computation, additional priority orderings would be required to ensure liveness of the solution: to ensure all robots eventually reach their destinations.

The approaches discussed in subsection 2.6.2 also do not account for robots in the environment that have no destination node. Furthermore, these approaches do not account for the case where all robots are to visit the same destination at different times. Therefore, the feasibility tests as discussed by Yu and Rus [12], where it is assumed that all pebbles must be at their destination node in the final time, are not necessarily appropriate in the MFS context. Consider, for example, the “un-solvable” 15 puzzle in Figure 2.6(b). If the requirement is such that every tile on the board reaches its destination at least once, the puzzle is no longer un-solvable and a sequence of moves can easily be shown that brings tiles 14 and 15 to their respective grid positions. These new rules for the 15 puzzle are appropriate to MFS. It is thus concluded that adapting the PMG approach to MFS is not an elegant solution and the requirements of MFS should be designed into the coordination algorithm from inception.

CHAPTER 3 THE 15 PUZZLE WITH VISITORS' RULES

3.1 INTRODUCTION

In order to solve the problem of multi-robot motion coordination in MFS efficiently, algorithms that are tailored to its needs are required. Existing approaches do not scale well with large numbers of robots and do not always guarantee starvation-freedom. The underlying assumptions of the intended application dictates the design decisions made in the development phase of coordination algorithms. Therefore, adapting existing approaches to MFS does not provide an efficient solution to the motion coordination problem.

3.2 MFS MOTION COORDINATION REQUIREMENTS

Given the feasibility of current commercial MFS and the surveyed literature, the following requirements must be met:

1. Collisions are catastrophic and are not allowed.

2. The movements of the robots are governed by second order dynamics and a finite state machine: they have to transition through an instantaneous stationary state in order to switch between linear and rotating motions. This implies that they have to come to a halt in order to change direction.

3. The system should be deadlock free.

4. It is preferable for robots to operate independently and asynchronously in order to avoid enforcing supervisory control systems with good tracking performance requirements.

5. Fiducial markers and server systems provide the required infrastructure for robots to localize themselves and communicate with each other.

6. When a robot is assigned a task, it must eventually be completed, where-after it may or may not receive a new task.

7. Idle robots are allowed in the environment and must not (catastrophically) hinder the system's performance.

8. In order to maximize throughput, minimum makespan solutions are preferable; however, solution quality is secondary to deadlock-freeness, robustness and scalability.

Addendum A discusses the notion of discrete rotations on graph cycles and train-like motions. It is chosen not to allow such motions due to the technical difficulties that arise with their implementation on real robots.

3.3 VISITORS' RULES

As discussed in subsection 2.6.2, previous works that originated from analogies with the 15 puzzle adopt the assumption that robots start in an initial configuration and end in a final configuration. However, in MFS robots are required to visit their assigned destinations at least once and can thereafter move to different locations without compromising their goal: to move items between storage locations and picking or replenishment stations.

With this goal in mind, new rules to the 15 puzzle are proposed: tiles are initially at random board positions. Every tile is assigned a destination board position. The objective of this new 15 puzzle is to generate a sequence of moves such that every tile visits its destination node at least once. Once a tile has reached its destination node it is regarded as having completed its mission and is allowed to move to any other node. This version of the game is referred to as the "visitors' 15 puzzle" and the new rules as "visitors' rules".

Under visitors' rules, the destination node of a tile does not necessarily need to be unique. In fact, all tiles can be sent to same location and the assignment can still be solvable. If the board of the visitors' 15 puzzle is restricted to rectangular four-connected grids, all instances of the visitors' 15 puzzle are solvable. In other words, the cost associated with the feasibility check can be eliminated in this case.

Tiles in the 15 puzzle are analogous to robots in the MFS workspace. Therefore, tiles are referred to as robots in the rest of this thesis. Similarly, the 15 puzzle board is analogous to the MFS workspace and will therefore be referred to as the workspace. If the board has a height of h and width of w, a more general instance of the 15 puzzle is obtained. This puzzle is referred to as the $h \times w$ puzzle.

3.4 PROBLEM STATEMENT

G is a graph of N vertices with $k = N - 1$ robots numbered $1, ..., k$ on distinct vertices. The robots are constrained to move on the vertices V and edges E in a graph $G(V, E)$. Each robot r_i has a current position $o_i \in V$ and a goal position $p_i \in V$. Time is divided into discrete steps. At each time step a robot can either move to a neighbouring location or wait in its current location. A move consists of transferring a robot from its current vertex to an adjacent unoccupied one. The problem is to find a sequence of actions for all r_i that will move it to p_i at least once.

3.5 PROPOSED SOLUTION

In the 15 puzzle, only one unoccupied node exists. The unoccupied node is referred to as a "swap-space". In order to play the 15 puzzle with visitors' rules, the following procedure is executed for every robot, one after the other:

First, a route is planned from o_i to p_i. Once the route is determined, the robot has to move to its destination. While en route, the way can either be clear or blocked. If the way is clear, it implies that the swap-space is at the next node on the current robot's route. In this case, the robot simply swaps its position with the swap-space.

However, the robot could be required to move other robots out of its way. In the case that the current robot's path is blocked, it plans a path from the swap-space to the next node on its route. This path is referred to as the "swap-path". It is a requirement that the swap-path circumvents the current node of the robot. In order to achieve this, the robot's current node is excluded from the graph while the swap path is planned. Once the swap-path is found, starting from the swap-space, robots swap their positions sequentially with the swap-space along the swap path until the swap-space is at the next node on the current robot's route.

Mission completion means that a robot has reached its destination vertex. After every move, the robot that has moved is checked for mission completion. If the robot has reached its destination, it is added to a set of completed robots. This can happen either by serendipity: as a result of being on another robot's swap-path, or by intent: as a result of the robot's own planned movements. When all robots are contained in the completed set, each robot has reached its target destination at least once.

The outlined procedure provides a solution to the $h \times w$ puzzle.

3.6 IMPLEMENTATION

The pseudo-code of the proposed algorithm is as shown in Algorithm 3.1 below. The planned path contained in *Route* is represented as a list of graph vertices $\in V$. *Route* is planned using a standard implementation of the well-known A^* [27] algorithm. The underlying graph G is implemented using an adjacency list as it represents a sparse four-connected grid. *Swap-space* represents the node location of the swap-space and is a global variable that is visible to all sub-routines.

The notation: $A \leftarrow B$, indicates that the value of B is assigned to A. Algorithm 3.1 makes use of two sub-routines "Plan_swap" and "Execute_swap" shown in Algorithms 3.2 and 3.3 respectively. In Algorithm 3.2, the swap-path is planned. In order to circumvent the *Robot*'s currently occupied node, *Current_Node* is excluded during the planning step. Once again, A^* is used to plan this path.

Algorithm 3.1 Solve

```
add all robots to Pebbles_queue
while Pebbles_queue has elements do
    if Robot has not yet completed its mission then
        Current_node ← get Robot's current position
        Goal_node ← get Robot's goal position
        Route ← plan a path from Current_node to Goal_node
        while Route is not empty do
            Next_node ← remove next node from Route
            if Next_node is not occupied then
                move Robot to Next_node
                Swap-space ← Current_node
            else
                Swap-path ← Plan_swap(Current_node, Next_node)
                Execute_swap(Swap-path)
            end if
        end while
    end if
end while
```

Algorithm 3.2 Plan_swap

```
Input: Current_node, Next_node
    set Current_node to excluded
    Swap_path ← plan route from Swap-space to Next_node
    set Current_node to included
Output: Swap_path
```

Algorithm 3.3 Execute_swap

```
Input: Swap_path
    while Swap_path is not empty do
        To_swap_node ← remove next node from Swap_path
        To_swap_robot ← get robot at To_swap_node
        move To_swap_robot to Swap-space
        Swap-space ← To_swap_node
    end while
```

In Figures 3.1 and 3.2, robot 1's mission is completed using the proposed algorithm. Figure 3.1 shows that robot 1's path is blocked while executing Algorithm 3.1. It subsequently plans a swap-path using Algorithm 3.2 and then successively executes the moves required by the swap-path using Algorithm 3.3. In the final step of Figure 3.1, robot 1 progresses by a single move in the next iteration of the outer loop in Algorithm 3.1.

Figure 3.2 shows the overall process of moving robot 1 to its destination. The first and last images represent the state of the board before robot 1 starts moving and after it reaches its destination respectively. The second image shows the initial planned path. All subsequent steps include the required swap path and do not show the process of the individual swaps, but rather the result after every termination of the sub-routine shown in Algorithm 3.3.

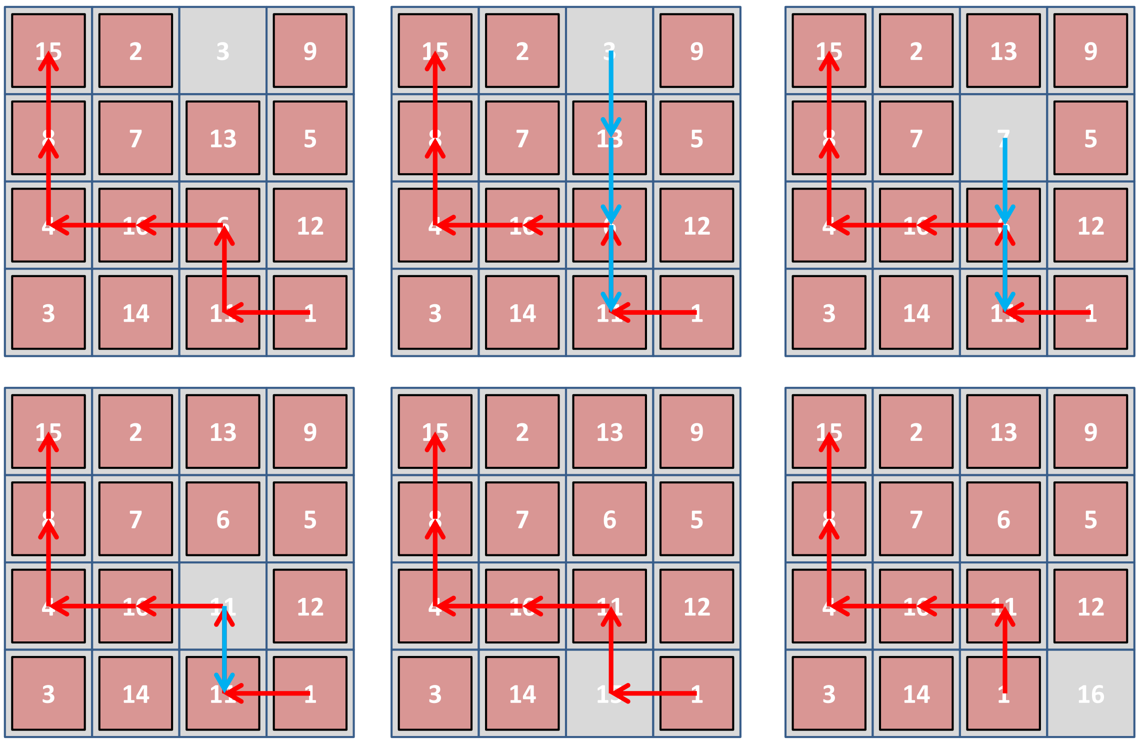


Figure 3.1. Robot 1 plans its route and progresses by one move

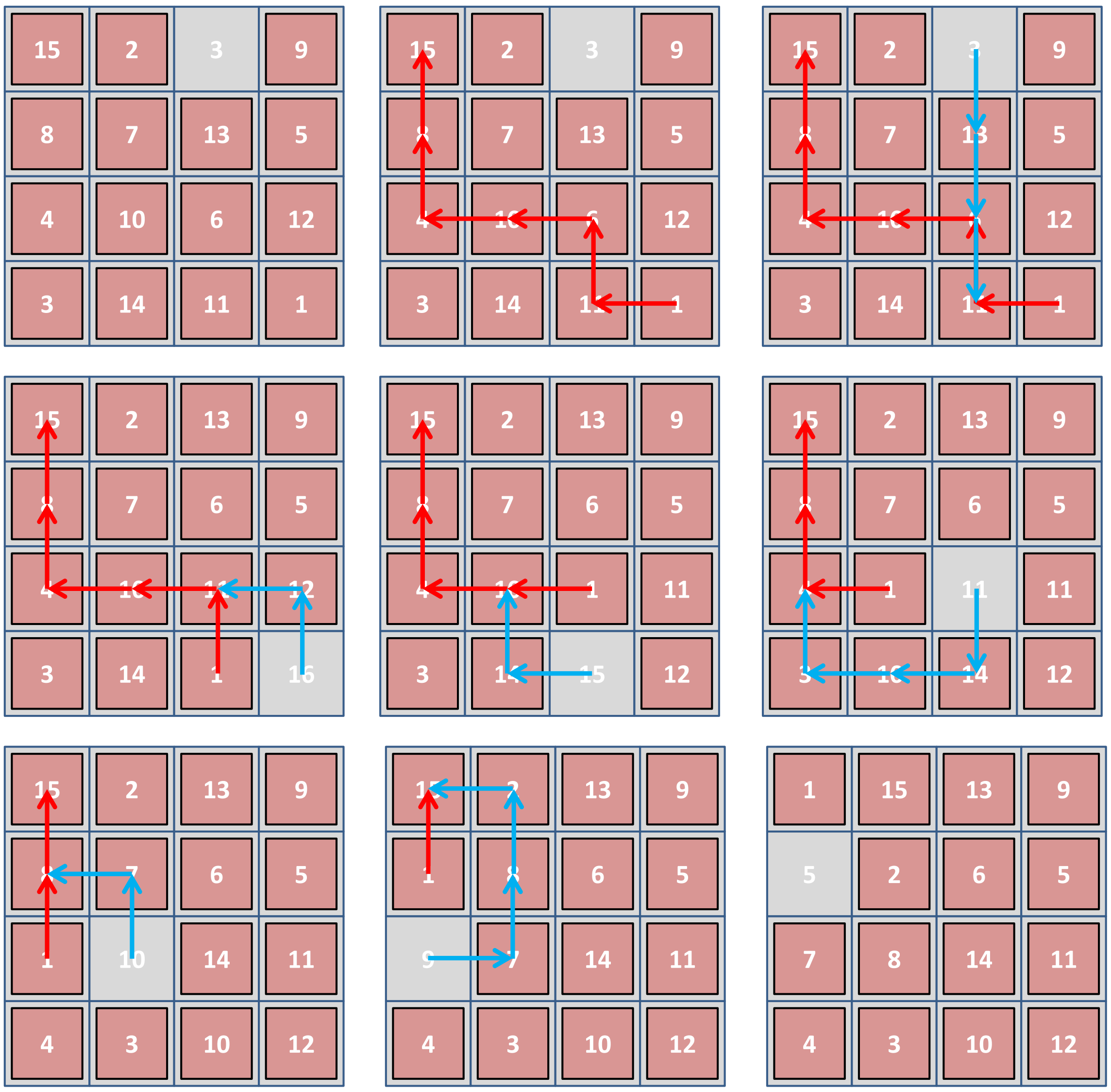


Figure 3.2. Robot 1 completes its mission by intent

3.7 RESULTS AND DISCUSSION

3.7.1 Text-based simulation environment

In order to validate the proposed algorithm, a simulation environment was developed in the Java programming language. The environment supports simple text based output so that the mission completion of the robots can be visualized, as shown in Figure 3.3.

In this implementation, the target destination of each robot is determined by its unique identification number and the row-dominant numbering of the board for consistency with the original 15 puzzle. However, robots can be assigned any destination without affecting the solvability of the visitors' $h \times w$ puzzle. In later chapters, other destination assignments will be showcased.

The initial board state can be seen in Figure 3.3(a). Robot 12 is already in its target position in the starting board state; this is allowed and is taken as an instance of mission completion by serendipity. "Pebbles Queue" indicates a ordered list of robots that have not yet been completed. The use of the term "Pebbles" is from the PMG nomenclature. Robots are removed from the "Pebbles Queue" in the order shown. This order is arbitrary as it does not affect the game-play under visitors' rules. The completed set is not ordered as it is implemented using a hash set.

A robot is removed from the "Pebbles Queue" and added to the "Completed" set when it reaches its destination. Once all the robots have been added to the completed set, the game terminates as shown in Figure 3.3(b). Coincidentally, in Figure 3.3(b) robots 1,2,3,4,13 and 14 are at their destinations upon termination of the game. Once again, this is not a requirement under the visitors' rules.

3.7.2 Complexity and completeness

For robots moving on grids, the asymptotic worst case growth rate g of the proposed algorithm is given in Addendum B and found to be $g = O(N^{\frac{3}{2}})$ where N is the number of nodes in G. Addendum C shows that this result extends to the case where the underlying graph is composed of multiple rectangular grids where every pair of intersecting rectangular sections share at least two nodes. Addendum D shows that the proposed algorithm is complete for bi-connected graphs. As discussed in subsection 2.6.2, this result was also obtained by Wilson [20]. The result of these three Addenda can be summarized as follows:

1) As long as the underlying graph is biconnected, the proposed algorithm is guaranteed to provide a solution.

2) However, some bi-connected graphs can lead to very inefficient motions if one of the paths that connect two sub-graphs is very long. The asymptotic analyses in Addenda B and C rely on the

```
Moving pebble: 9
Pebbles queue: 6 14 10 8 5 1 13 11 12 7 4 15 3 2
Completed: 12

[ 5] [ 7] [11] [14]

[13] [ 3] [15] [ 1]

[ 8] [ 6] [ 2] [12]

[ 4] [  ] [ 9] [10]
```

(a) The starting board state

```
Moving pebble: 2
Pebbles queue:
Completed: 9 6 14 10 8 5 1 13 11 12 7 4 15 3 2

[ 1] [ 2] [ 3] [ 4]

[ 9] [  ] [ 5] [11]

[ 6] [ 7] [10] [12]

[13] [14] [ 8] [15]
```

(b) The final board state

Figure 3.3. Console window text-based output for the 15 puzzle with visitors' rules

assumption that both paths connecting any two sub-graphs are short. The class of graphs composed of rectangular grids that share at least two nodes between any two intersecting sections always have short pairs of paths between any two of their sub-graphs. The said class of graphs well describes real warehouses. Therefore, even though the proposed algorithm is not complete in general, it is complete for an appropriate class of graphs that is easy to enforce in real applications. As long as the user enforces such a graph, the performance guarantees hold in terms of the number of required moves for all robots to reach their destinations at least once.

In other words, the analysis specifies the design guidelines for the underlying graphs for which the proposed algorithm 1) will work and 2) will be efficient. For the specific sub-class of graphs, the described performance is achieved; however, it is easy to show other counter-examples where the performance guarantees do not hold. For example, the algorithm achieves a $O(N^3)$ upper bound when the graph consists of a single graph cycle.

3.7.3 Results

The developed simulation environment allows for testing of the proposed algorithm. The simulations show that all robots eventually reach their target destinations as desired as shown in Figure 3.3. To verify the analytical result, it is compared with data obtained from experiments as shown in Figure 3.4. The experiments were conducted by creating a $h \times h$ grid and allowing the game to be played to completion whilst counting the number of moves. The grid dimension h was initialized to 2 and incrementally increased to 75 which yields 5625 nodes.

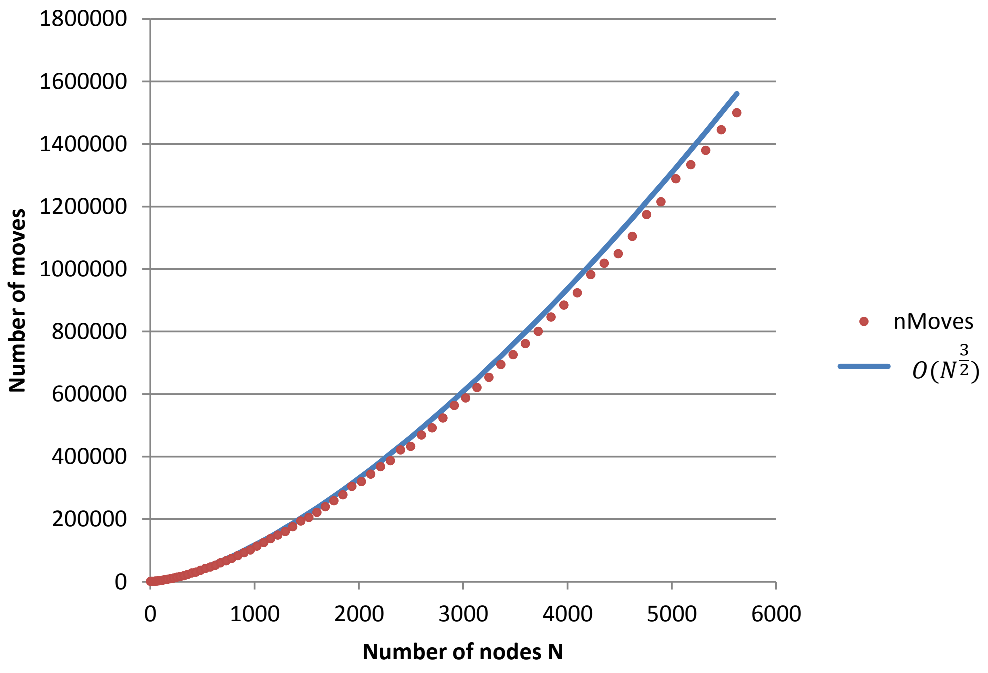


Figure 3.4. Moves versus board size for the visitors' $h \times h$ puzzle

3.7.4 Discussion

The analytical growth rate of the algorithm is shown by the blue curve and the simulation result is shown by the maroon dots in Figure 3.4. The analytical growth rate and simulation result are seen to have the same shape, which suggests that the analysis is correct. Though it is not shown, the environment supports rectangular boards of any dimension. As discussed in section 3.6 the location of the swap-space is a global variable that is updated with every move. Therefore, this version of the simulator does not support more than one open-space.

3.8 CONCLUSION

This chapter has introduced the requirements of a motion coordination scheme for use in MFS. The notion of visitors' rules for the 15 puzzle allows for an analogy to be made between the 15 puzzle and the MFS motion coordination requirements. An algorithm was proposed, implemented, analysed, evaluated and discussed. The analysis shows that the proposed algorithm is complete and efficient for a class of graphs that is appropriate in industrial MFS settings. The experiments show that the proposed algorithm allows all robots to reach their target nodes at least once, as desired. The analytical and experimental results show good correlation. The proposed algorithm scales well with the number of nodes in the underlying graph, even in this case where there are $k = N - 1$ robots. It is noted that the current algorithm does not allow for cases where more than one unoccupied node is available, which will be addressed in the following chapter.

CHAPTER 4 EXTENSION TO MULTIPLE SWAP SPACES

4.1 PROBLEM STATEMENT

In the previous chapter, an algorithm that solves the MFS motion coordination problem in the case where only one unoccupied node exists was introduced. An algorithm that allows for the single swap-space scenario is robust against high traffic volumes and must be deadlock free. It is, however, highly unlikely for a single swap-space scenario to occur in industry. In industry, the robot traffic density will generally be low, as high traffic density leads to reduced throughput. It is, therefore, desirable to have an algorithm that can both solve single and multi-swap-space scenarios. This chapter is dedicated to extending the work from the previous chapter so that multiple unoccupied spaces can be accommodated.

The previous problem statement is thus modified to the case where G is a graph of N vertices with $k < N$ robots numbered $1, ..., k$ on distinct vertices.

4.2 PROPOSED SOLUTION

According to Skiena [57], the Breadth First Search (BFS) was first found to give the shortest path by Moore [29]. The A^* search from the swap-space to the next node on the robots path can easily be replaced with a BFS from the next node on its path to the nearest unoccupied space. This allows the algorithm to be extended to the case where more than one unoccupied node is available.

The updated version of Algorithm 3.1 is shown in Algorithm 4.1. Previously, the global variable *Swap-space* was updated after every move operation. In the updated algorithm, it is no longer necessary to keep track of *Swap-space*'s position. In Algorithm 4.1, the sub-routine Plan_swap has been replaced with Find_swap shown in Algorithm 4.2 which implements the BFS.

In Algorithm 4.2, *Swap-space* is no longer used. Also, it is assumed that a function reverse$(\cdot)$ is available. This function reverses the order of the nodes in a path such that it is given from sink to source instead of from source to sink. As a result, the swap-path initially contains an unoccupied node at its head and the next node on the robot's path at its tail.

Algorithm 3.3 also requires modification to the form given in Algorithm 4.3. This is due to the fact that the global variable *Swap-space* needs to be replaced with the local variable *Unoccupied* which keeps track of the swap-space on the swap-path, and is initially obtained from the head of the swap-path. Algorithm 4.3 could be used to replace Algorithm 3.3 in the previous chapter if the swap-path given by the A^* implementation contains the swap-space at its head. In the previous chapter, it was assumed that the swap-path does not include the unoccupied node.

4.3 RESULTS

The proposed changes were made to the simulation environment. Figure 4.1 shows that the workspace contains multiple unoccupied spaces and was able to proceed from its initial to its final state using the proposed changes. Previously, it was claimed that the implementation was capable of solving any $h \times w$ instance of the game, but was not shown. In this experiment, a 4×5 workspace was used in order to showcase this functionality.

Robot 9 is still in the "Pebbles Queue" in the final state, due to the way that the program removes elements from the queue. Robot 9 was serendipitously completed and added to the completed set; however, it was never removed from the "Pebbles Queue" because all robots were contained in the completed set before it was scheduled for removal. This behaviour does not affect the correctness of the program.

It is expected that lower traffic densities result in fewer required moves, as less moves are spent long

Algorithm 4.1 Solve_multi

```
add all robots to Pebbles_queue
while Pebbles_queue has elements do
    if Robot has not yet completed its mission then
        Current_node ← get Robot's current position
        Goal_node ← get Robot's goal position
        Route ← plan a path from Current_node to Goal_node
        while Route is not empty do
            Next_node ← remove next node from Route
            if Next_node is not occupied then
                move Robot to Next_node
            else
                Swap-path ← Find_swap(Current_node, Next_node)
                Execute_swap(Swap-path)
            end if
        end while
    end if
end while
```

Algorithm 4.2 Find_swap

```
Input: Current_node, Next_node
    set Current_node to excluded
    Swap_path ← find a route from Next_node to the nearest open space using BFS
    Swap_path ← reverse(Swap_path)
    set Current_node to included
Output: Swap_path
```

swap paths and more are spent progressing towards the robot destinations. In addition, when the traffic density is reduced, there are less robots that need to visit their destinations, which should have an additional linearly decreasing effect on the required number of moves. Experiments are performed to verify this expectation.

The experiments consist of generating random starting positions for k robots and allowing the game to proceed to completion. The number of robots investigated is $k = 1, ..., N-1$. In addition to the

Algorithm 4.3 Execute_swap_multi

```
Input: $Swap\_path$
  $Unoccupied \leftarrow$ remove first node from $Swap\_path$
  while $Swap\_path$ is not empty do
    $To\_swap\_node \leftarrow$ remove next node from $Swap\_path$
    $To\_swap\_robot \leftarrow$ get robot at $To\_swap\_node$
    move $To\_swap\_robot$ to $Unoccupied$
    $Unoccupied \leftarrow To\_swap\_node$
  end while
```

```
Moving pebble: 2
Pebbles queue: 11 6 12 7 15 8 10 4 14 13 5 1 3 9
Completed: 11 1

[ 1] [ 7] [ 4] [15] [14]

[10] [ 9] [13] [  ] [ 6]

[11] [ 8] [ 5] [  ] [  ]

[ 3] [  ] [ 2] [12] [  ]
```

(a) The starting workspace state

```
Moving pebble: 3
Pebbles queue: 9
Completed: 11 2 6 12 7 15 8 10 4 14 13 5 1 3 9

[ 1] [11] [ 3] [ 4] [ 5]

[ 6] [ 9] [  ] [ 8] [10]

[  ] [ 7] [  ] [14] [15]

[  ] [ 2] [12] [13] [  ]
```

(b) The final workspace state

Figure 4.1. Text-based output with multiple unoccupied spaces

effect of the traffic volume, the effect of the aspect ratio of the grid also affects the number of required moves. To investigate the effect of the aspect ratio of grids independently from the number of nodes, numerous experiments with the same number of nodes but different aspect ratios are required. To this end, 144, which is a comparatively small number, has many integer factors. Therefore, $N = 144$ is a good choice as it provides many row/column combinations. The results of the experiment are shown in Figure 4.2.

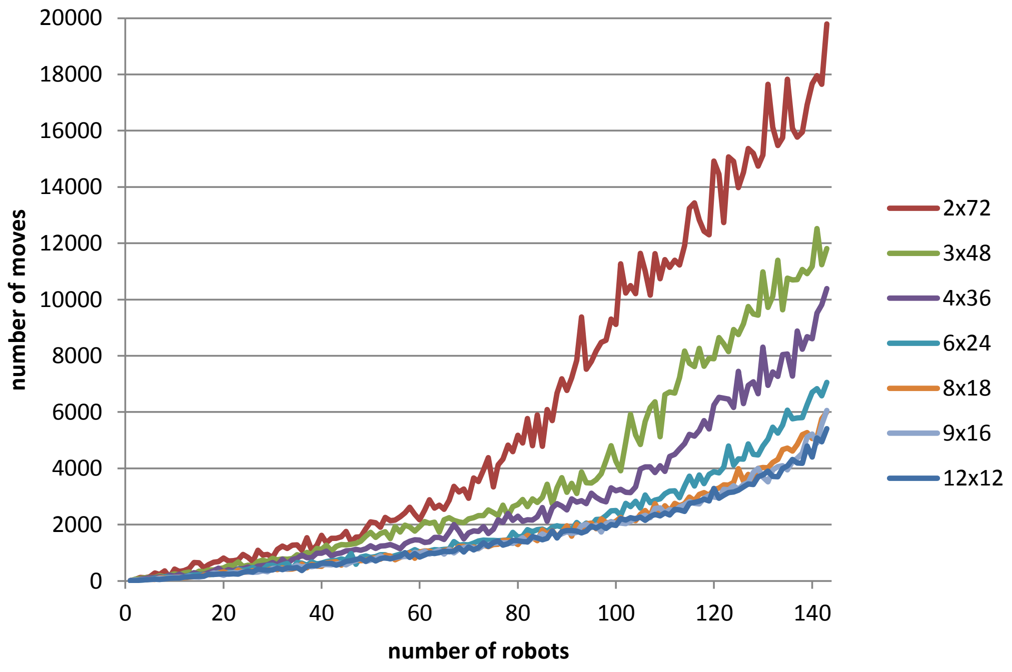


Figure 4.2. Number of moves versus number of robots for different workspace sizes

4.4 DISCUSSION

The expectation that the number of moves decreases as the traffic decreases is validated by the experimental results. The results show that the number of moves scales sub-linearly between the number required for a single robot and the number required for $k = N - 1$ robots.

The aspect ratio is taken as $a = h/w$. Decreasing the aspect ratio is seen to have the effect of increasing the number of moves. However, the effect is not linear. The 8×18 workspace results in $a = 0.44$ and has comparative performance to the best faring case of the 12×12 workspace with $a = 1$, whilst the 2×74 with an aspect ratio of $a = 0.028$ workspace results in 3.65 times the number of moves compared to the 12×12 workspace for the case where $k = N - 1$. The increase in the number of required steps can be attributed to the fact that the average path length in grids with extremely low aspect ratio are significantly longer than for square grids.

4.5 CONCLUSION

The visitors' algorithm of Chapter 3 has been extended to the case where more than a single unoccupied space is available. Experiments show that the modification is successful. The resulting program behaviour is as expected in the sense that the required number of moves decreases when the number of unoccupied spaces increases. Grids that are severely non-square are seen to increase the required number of steps significantly.

In this and the previous chapter, the number of moves has been the primary feature under investigation. The advantage of employing numerous robots to perform tasks is that multiple tasks can be performed simultaneously. This results in increased throughput. Up to this point, this advantage has not yet been exploited as robots completed their tasks consecutively. The following chapter is dedicated to developing a visitors' algorithm that allows for simultaneous task execution. This is referred to as parallel execution. In the context of parallel execution for MFS, the number of moves becomes less important. Instead, throughput becomes the metric of concern.

CHAPTER 5 PARALLEL EXECUTION

5.1 INTRODUCTION

In the previous chapters, it was assumed that the plans for all robots are computed and executed in series. One of the main benefits of employing multiple robots is the increased throughput gained by parallel execution of tasks. Therefore, in this chapter an algorithm is developed that implements the visitors' rules in the case where robots are allowed to complete their tasks in parallel.

5.2 PROBLEM STATEMENT

In the case of parallel task execution, all robots are allowed to move simultaneously. This gives rise to the possibility of collision due to two or more robots attempting to move to the same location at the same time. Collision avoidance can effectively be achieved with the use of a space reservation system. The space reservation system gives rise to the problem of choosing precedence when multiple robots have conflicting goals. As discussed in subsection 2.6.1, improper implementation of the precedence handling mechanism could lead to starvation.

5.3 PROPOSED SOLUTION

The execution is adapted such that all robots are allowed to move simultaneously. In this context, every discrete time step represents a turn. In the context of the previous chapters, only one robot was moved in every turn. Therefore, previously, the notion of a move and a turn were equivalent. In this chapter, these two concepts are separate and there can be multiple moves in every turn.

With the aim of achieving a well defined precedence handling mechanism, all robots are initially assigned a priority. Every turn can be broken down into phases. The phases of the turns are described below. The first phase is the planning and intention propagation phase. At the start of each turn, all robots plan their routes and determine the next node to move to in order to progress towards their goals.

Every robot maintains a priority queue of instructions. The priority of elements in this queue is determined by the priority of the instructing robot. The instructions instruct robots to move to a neighbouring node, or stay put in its current location. If the next node on a robot's route is not blocked, the robot adds an instruction, from itself, to move to that node into its priority queue. In other words, if the next node on a robot's path is not occupied, the robot instructs itself to move to the node.

If, however, the next node on the robot's route is occupied, the robot instructs other robots to give way while instructing itself to stay put in its current location. The method of determining these instructions is based on the work from the previous chapters: every robot whose route is blocked plans a swap-path, and instructs only the robot in the swap-path that is currently neighbouring the swap-path's unoccupied node to move to the unoccupied node. This marks the end of the first phase, where all robots accumulate instructions into their priority queues.

The next phase is the requesting phase. After the planning and propagation phase, all robots choose the highest priority instruction from their queues and request, based on this instruction, to either move or stay where they are. Nodes also maintain priority queues of instructions. Every node is responsible for granting robots access to itself. From the node's perspective, the instructions received from robots are therefore requests. Every node receives the requests sent out by the robots and accumulates them into its priority queue. The ordering in this priority queue is also based on the instructing robot's priority. Once the robot has sent its request to a node, its instructions priority queue is cleared.

In the final phase, all requests at the nodes are granted and the robots make the necessary moves. Every node grants the highest priority request in its priority queue and thereafter clears all other requests from its queue. This gives the robots authority to move to the node that granted the request. If the node that granted the request is the one that was already occupied by the robot, the robot stays put. Due to the method by which requests are sent to nodes, the requests are such that no occupied node receives a request, unless the requesting robot is already at the node. In other words, only one robot is allowed to move to an unoccupied node at a time. Robots can only move to unoccupied nodes or remain stationary in their current nodes. Therefore, no collisions can occur.

Finally, all granted moves are made and the process repeats until all robots have reached their destinations at least once.

5.4 IMPLEMENTATION

The main program is shown in Algorithm 5.1. In this chapter, the priority assigned to the robots is equal to their identification number. Lower identification numbers represent higher priorities. As in previous chapters, the robots' routes are planned using the A^* algorithm. Algorithm 5.1 invokes three sub-routines representing its three phases. The subroutines are Propagate_intentions, Make_requests and Make_moves represented by Algorithms 5.2, 5.3 and 5.4 respectively.

Using an object-oriented approach, the instructions can be represented as request objects. A request is an object that has an instructing robot, instructed robot and a node towards which the robot is being instructed. Therefore, every time the keyword 'instructs' is used, a request object is created and added to the instructed robot's priority queue.

Algorithm 5.2 uses the subroutine from Chapter 4 described in Algorithm 4.2 to plan the swap-path. The logic of Algorithm 5.2 is similar to that of Algorithms 3.1 and 4.1, but robots now go through an accumulation and requesting phase before moving. The reason robots need to add instructions from themselves to their own queues is so that their priorities can be taken into account. A robot's priority only comes into play if they manifest in the priority queues of the nodes and robots. A robot is allowed to instruct higher priority robots than itself. If the higher priority robot has not completed its mission, it will not request the lower priority instruction because its own instruction with a higher priority is also in its priority queue. On the other hand, if the high priority robot has completed its mission, it stops making its own requests and will obey lower priority robots.

In Algorithm 5.3, the robots' requests are sent to the nodes and the robots' priority queues are emptied. Similarly, in Algorithm 5.4, nodes grant the requests and robots make their moves, after which all remaining requests are cleared from the nodes' priority queues.

Algorithm 5.1 Parallel_solve

```
assign all robot priorities
while not all robots have completed their missions do
    for all Robots that have not yet completed their missions do
        Current_node ← get Robot's current position
        Goal_node ← get Robot's goal position
        Route ← plan a path from Current_node to Goal_node
        Next_node ← remove first node from Route
        Propagate_intentions(Robot, Current_node, Next_node)
    end for
    Make_requests()
    Make_moves()
end while
```

Algorithm 5.2 Propagate_intentions

```
Input: Robot, Current_node, Next_node
if Next_node is not occupied then
    Robot instructs itself to Next_node
else
    Robot instructs itself to Current_node
    Swap-path ← Find_swap(Current_node, Next_node)
    Unoccupied ← remove first node from Swap_path
    To_swap_node ← remove next node from Swap_path
    To_swap_robot ← get robot at To_swap_node
    Robot instructs To_swap_robot to Unoccupied
end if
```

5.5 RESULTS

Figure 5.1 shows three robots completing their tasks in parallel using the proposed algorithm. In every sub-figure, the end result of the three phases of Algorithm 5.1 is shown. In other words, for every sub-figure, the previous sub-figure shows its initial state. From Figures 5.1(a) to 5.1(c), from turn 0 to turn 2, there are no conflicts between the robots and they all progress towards their goals.

Algorithm 5.3 Make_requests

```
for all Robots do
    if Robot's priority queue is not empty then
        Request ← poll Robot's priority queue
        To_node ← get destination node from Request
        Add Request to To_node's priority queue
    end if
    empty Robot's priority queue
end for
```

Algorithm 5.4 Make_moves

```
for all Nodes do
    if the Node's priority queue is not empty then
        Request ← poll Node's priority queue
        Granted ← get instructed robot from Request
        move Granted to Node
        check whether Granted has completed its mission
    end if
    remove all of Node's requests
end for
```

The nodes are named according to their grid positions. Rows are numbered from top to bottom and columns from left to right. For example, the node in the third row from the top and the second column from the left is written as [3,2].

In turn 3 shown in Figure 5.1(d), robot 1 instructs robot 2 to move out of the way. Also, robot 2 instructs robot 1 to move out of the way. This instruction is added to robot 1's priority queue along with an instruction from itself to stay at grid position [1,2]. When robot 1 selects the highest priority instruction from its priority queue, the resulting instruction is to to stay at [1,2]. Because [1,2] is occupied, no other robot will request to move there and robot 1 is granted access to stay. Conversely, robot 2 also added an instruction from itself to stay at [1,1], but the highest priority instruction in its priority queue was from robot 1 to move to [2,1], which is the instruction that was requested by robot 2. Because [2,1] was not being requested by any other robot, robot 2 is granted access to move there and subsequently does so. Meanwhile, robot 3 is able to continue unhindered, and progresses closer to its goal.

In the planning and propagation phase of turn 4, robot 1 and 2 request to move to the node at grid position [1,1]; however, because of robot 1's higher priority, [1,1] selects robot 1 as the highest priority request from its priority queue. Robot 1 is granted access to [1,1] and robot 2 has to wait for the next turn because its request was not granted. Robot 1 and robot 3 reach their destination nodes, determined by the row major numbering of the grid. Robot 1 and 3 have thus completed their assigned missions and become idle.

The failed requests are deleted in every turn. Therefore, robot 2's request from turn 4 to move to [1,1] is not 'remembered' in turn 5. In turn 5, robot 2 gives two new instructions: the first is for itself to stay at [2,1] and the second is for robot 1 to move to [1,2]. Because robot 1 and 3 have become idle, they cease to give any instructions. However, they are still able to make requests. As a result, any instruction given by robot 2 is requested by the instructed robot and is granted by the requested node. Therefore, robot 2 waits at [2,1] while robot 1 clears the way by moving to [1,2].

In turn 6, robot 2 moves to [1,1]. In turn 7, robot 2 once again instructs robot 1 to move out of the way. In turn 8, robot 2 reaches its destination, thereby ending the execution of the algorithm.

For the random initial configuration shown, the minimum number of moves is 10. The minimum number of turns is 5. The number of moves and turns generated by the proposed algorithm is 14 and 8 respectively. The optimality of the proposed algorithm was not considered in its design and the algorithm does not perform optimally in terms of the number of moves or turns. However, the number of turns obtained is lower than the minimum number of moves, indicating that the algorithm effectively exploits the parallelism of the system.

From observation of the result presented in Figure 5.1, it is expected that the parallel execution should result in a significantly higher number of moves. Figure 5.2 shows the resulting number of moves for a 12×12 workspace using both the series and parallel algorithm. The experiments for series and parallel execution were conducted from different random starting configurations in each case. It is clear from the figure that the effects of parallel execution slightly increases the number of moves for cases with more than 100 robots. Let ρ denote the traffic density, defined as $\rho = k/N$. Remembering that the intended purpose of the algorithm is for use in low traffic density situations and noticing that

```
Completed:

[  ] [  ] [  ] [ 1]

[  ] [  ] [  ] [  ]

[ 2] [  ] [  ] [  ]

[  ] [  ] [  ] [ 3]
```

(a) turn=0, moves=0

```
Completed:

[  ] [  ] [ 1] [  ]

[ 2] [  ] [  ] [  ]

[  ] [  ] [  ] [ 3]

[  ] [  ] [  ] [  ]
```

(b) turn=1, moves=3

```
Completed:

[ 2] [ 1] [  ] [  ]

[  ] [  ] [  ] [ 3]

[  ] [  ] [  ] [  ]

[  ] [  ] [  ] [  ]
```

(c) turn=2, moves=6

```
Completed:

[  ] [ 1] [  ] [ 3]

[ 2] [  ] [  ] [  ]

[  ] [  ] [  ] [  ]

[  ] [  ] [  ] [  ]
```

(d) turn=3, moves=8

```
Completed: 1 3

[ 1] [  ] [ 3] [  ]

[ 2] [  ] [  ] [  ]

[  ] [  ] [  ] [  ]

[  ] [  ] [  ] [  ]
```

(e) turn=4, moves=10

```
Completed: 1 3

[  ] [ 1] [ 3] [  ]

[ 2] [  ] [  ] [  ]

[  ] [  ] [  ] [  ]

[  ] [  ] [  ] [  ]
```

(f) turn=5, moves=11

```
Completed: 1 3

[ 2] [ 1] [ 3] [  ]

[  ] [  ] [  ] [  ]

[  ] [  ] [  ] [  ]

[  ] [  ] [  ] [  ]
```

(g) turn=6, moves=12

```
Completed: 1 3

[ 2] [  ] [ 3] [  ]

[  ] [ 1] [  ] [  ]

[  ] [  ] [  ] [  ]

[  ] [  ] [  ] [  ]
```

(h) turn=7, moves=13

```
Completed: 1 3 2

[  ] [ 2] [ 3] [  ]

[  ] [ 1] [  ] [  ]

[  ] [  ] [  ] [  ]

[  ] [  ] [  ] [  ]
```

(i) turn=8, moves=14

Figure 5.1. Parallel completion of missions for 3 robots

$\rho = 100/144 = 0.69$ is a high traffic density, this slight increase in the number of moves is not of critical concern.

Figure 5.3 shows the number of moves and number of turns versus the number of robots in a 12×12 workspace for the parallel algorithm. The number of moves and turns are equal in the case where only a single swap-space is available. The algorithm behaves as if it were a series algorithm in this case. However, the number of turns drastically reduces as more unoccupied nodes are introduced. Once again, when there is only a single robot in the workspace, the series and parallel execution are

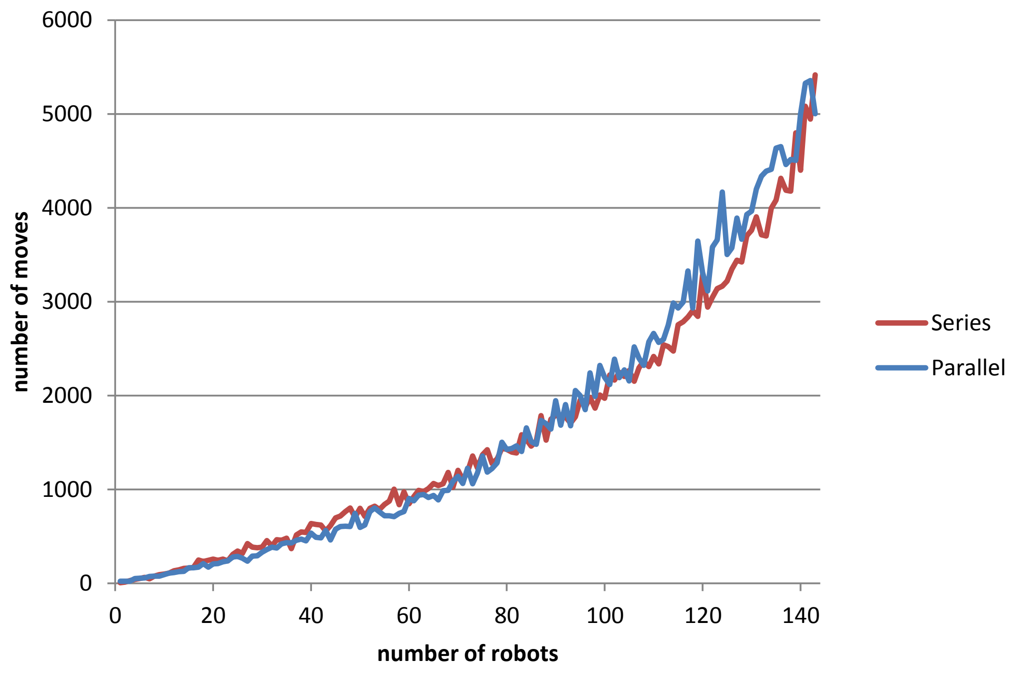


Figure 5.2. Comparison of series and parallel moves on a 12×12 workspace

equivalent. Intuitively, it is expected that the number of turns should be half that of the number of moves when two swap-spaces are available. Similarly, if 3 swap-spaces are available 3 times less turns than moves are expected, and so on. This type of decrease is hyperbolic. Most of the effect of the hyperbolic decrease happens with the introduction of the first couple of swap-spaces. For example, $\frac{100\%}{5} = 20\% \rightarrow 80\%$ decrease for introduction of 5 additional swap spaces. The hyperbolic decrease is plotted in Figure 5.3 and shows an optimistic estimate for the number of turns, given the number of moves.

There is, however, an upper bound for the number of moves that can be made in a single turn. When the number of unoccupied spaces is less than the number of robots, the maximum number of moves per turn is equal to the number of unoccupied spaces. If, however, the number of swap-spaces is more than the number of robots, the maximum number of moves per turn is equal to the number of robots. Furthermore, there are factors that negatively affect the ideal hyperbolic decrease. Firstly, the progress made by low priority robots can be undone by high priority robots if their swap-paths cross each other. Secondly, robots could choose congested routes while better alternatives are available, leading them to sit and wait when they could be making progress instead. Therefore, the theoretical upper bound on the number of moves per turn can never be reached in practice.

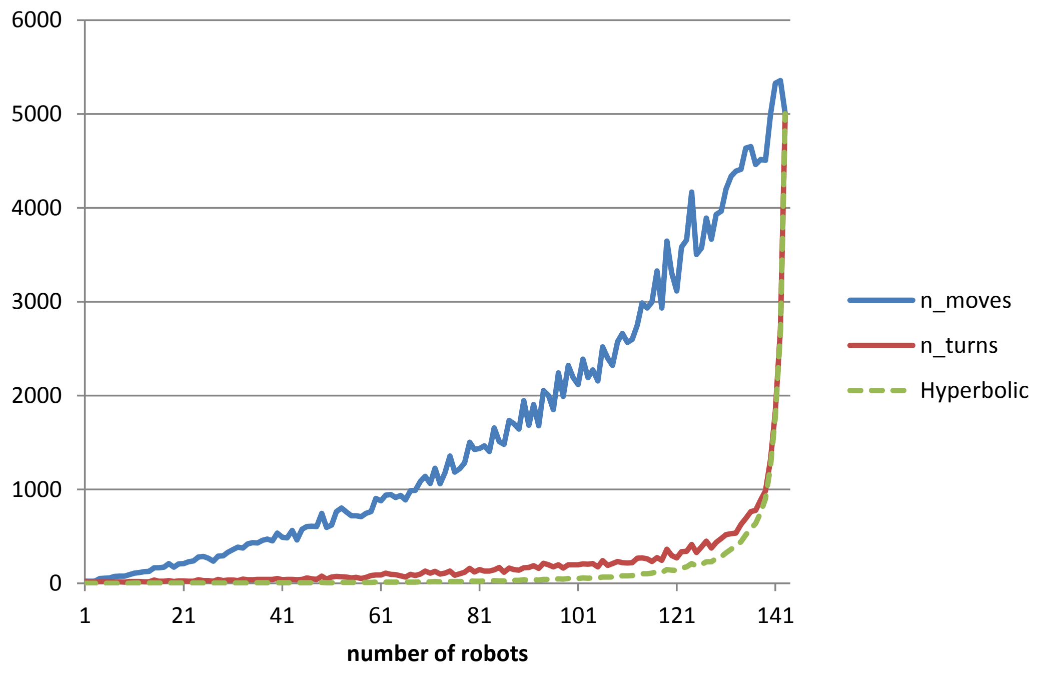


Figure 5.3. Moves and turns on a 12×12 workspace for parallel execution

One measure of parallelism is the number of moves per turn. To investigate the parallelism, 10 experiments were performed for every possible traffic density on a 12×12 grid as shown in Figure 5.4. The blue data points represent the number of moves divided by the number of turns for every experiment. The upper bound is indicated by the solid maroon line. The experiments show that the algorithm achieves the ideal parallelism when the number of unoccupied nodes or the number of robots is small. When the number of unoccupied spaces and robots is the same, the highest possible upper bound is achieved, but the delivered performance is not close to the ideal in this regime due to the effects explained in the previous paragraph. In spite of the ideal parallelism not being obtained in all cases, the algorithm fares very well for the first 5 unoccupied spaces achieving roughly 80% reduction in number of turns as expected by the hyperbolic decrease shown in Figure 5.3. The variance of the experiments tends to zero as the number of unoccupied nodes approaches one.

For $k = N - 1$ the execution of the series and parallel algorithm is the same. Though the execution is equivalent in these two cases, the associated computational cost for a centralized planner to plan the execution is not. In the parallel algorithm, all robots re-plan their routes and swap-paths in every turn, whereas the series algorithm only requires the route to be planned once per robot and the swap-path once for every time the swap-space is brought onto a robot's route. The re-planning in every turn

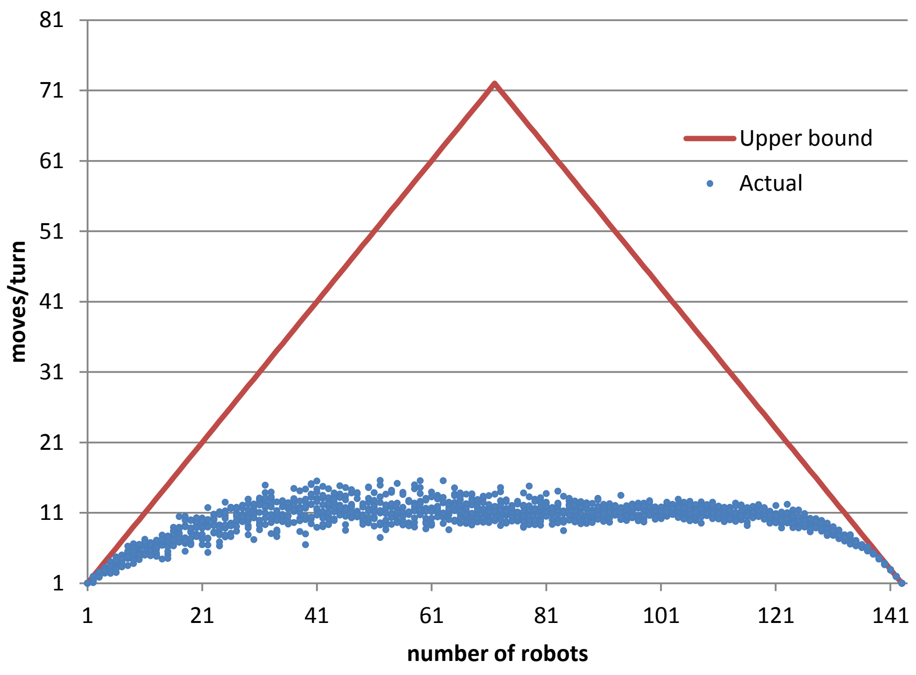


Figure 5.4. Moves per turn versus number of robots on a 12×12 workspace

for the parallel algorithm is required in order to handle contingencies when higher priority robots affect the executions of lower priority robots. In Addendum E, Equations (E.18) and (E.22) give the asymptotic rates on the required number of computations for the series and parallel case as $O(N^{\frac{5}{2}})$ and $O(N^{\frac{7}{2}} \log N)$ respectively. Figure 5.5 shows the comparative run times and theoretical bounds for square grids with increasing numbers of nodes for the two algorithms. The provided analytical bounds on the number of computations are conservative and do not appear to be tight as they eventually diverge from the experimental data.

The state of the workspace is represented by the knowledge of which node every robot is currently occupying, entering or deserting. Consider the case where there are $k < N$ robots, each with its own processor, that are provided with the state of the workspace and plan their routes and swap-paths independently. In this case, the computational load for parallel execution is only that of the BFS and A^* per robot which is $O(N \log N)$. Assuming that all robots synchronously start their planning phases, this implies an overall $O(N \log N)$ time associated with the planning phase for all robots. In other words, before every moving phase a $O(N \log N)$ waiting time is expected.

This is significantly shorter than the centralized serial time of $O(N^{\frac{5}{2}})$. In the serial case, a single $O(N^{\frac{5}{2}})$

planning phase is required after which all robots execute the plan without pausing. It would also be possible to shorten this planning time by starting execution before the entire plan is available.

Finally, the required number of moves remains $O(N^{\frac{3}{2}})$ regardless of whether the series or parallel algorithm is employed. It is thus concluded that Figure 5.5 represents the run times for simulating the two algorithms for the case where a centralized planner is used, but there exists an opportunity to significantly improve the computational time associated with the parallel algorithm when deployed on real robots.

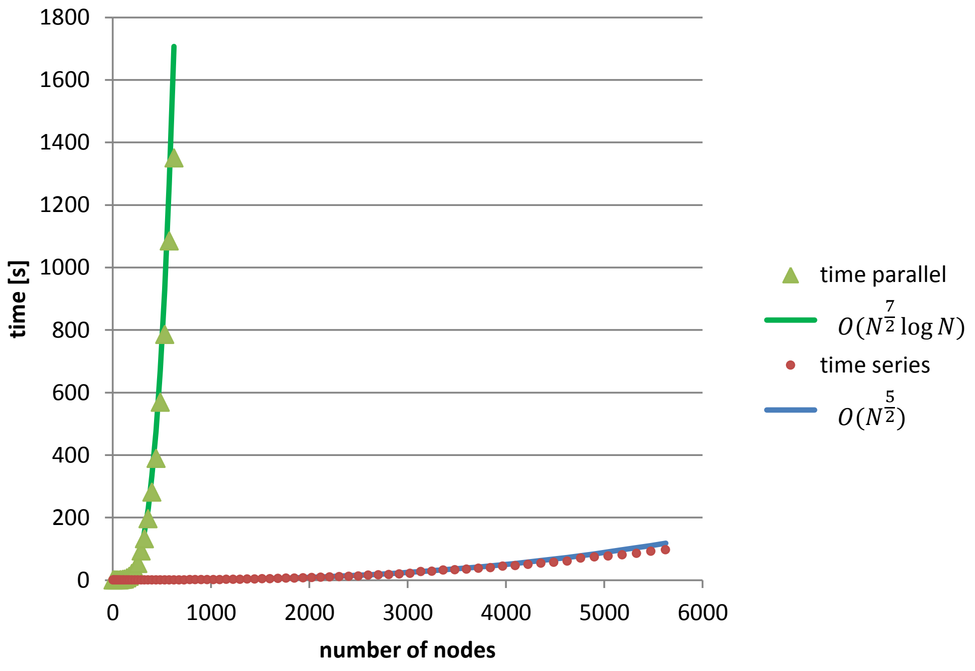


Figure 5.5. Series and parallel run times on square grids with $k = N - 1$

The effect of a on the number of turns can be observed in Figure 5.6. The trend is the same as for the series execution: the number of turns increases non-linearly as the aspect ratio decreases. This leads to square grids being the ideal geometry for maximum efficiency.

From Equation (B.6), the limiting behaviours of the number of moves and, therefore the number of turns n_{turns} are proportional to a multiplier that is function of the aspect ratio a. By setting the multiplier equal to 1 for $a = 1$, the following relation can be written:

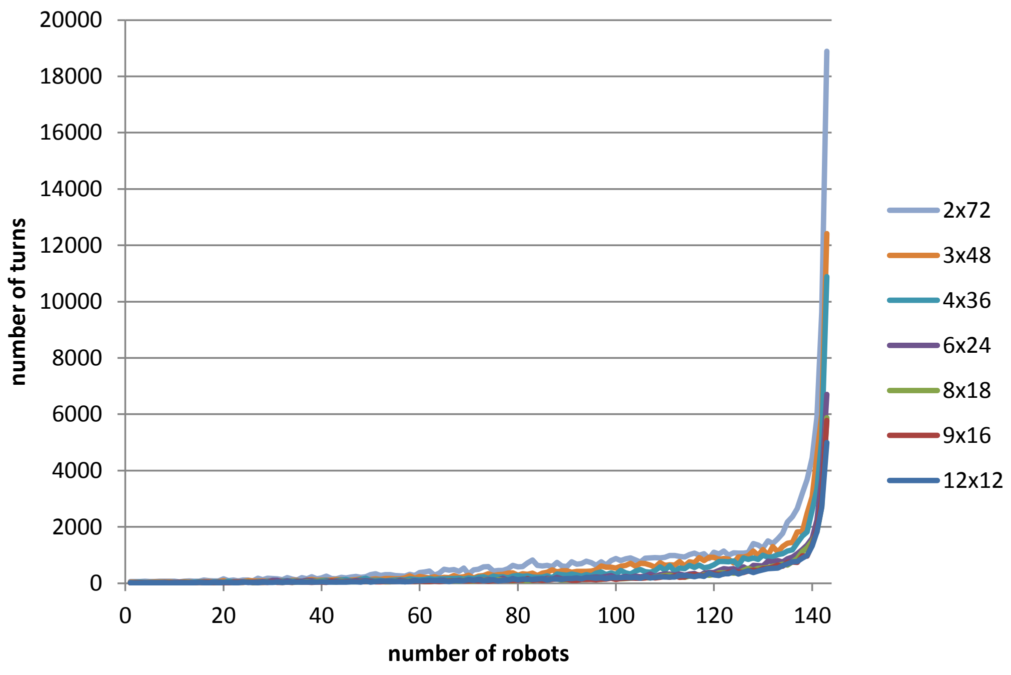


Figure 5.6. Comparison of turns on $h \times w$ workspaces for parallel execution

$$n_{turns} \propto \frac{1}{2}(a^{-\frac{1}{2}} + a^{\frac{1}{2}}) \tag{5.1}$$

Equation (5.1) shows that there exists a singularity at $a = 0$. Figure 5.7 shows a plot of Equation (5.1). In sub-section 4.4, it was mentioned that the effect of the aspect ratio becomes significant at low values of a. This can be seen from Figure 5.7 where for $a = 0.41$ the number of turns is expected to be just 10% higher, but for values of $a < 0.41$ the number of turns increases rapidly as $a \to 0$.

5.6 DISCUSSION

For an assignment of tasks, the term "makespan" is used to refer to the total time required to complete all the tasks. The makespan is equal to the number of moves in the serial case and equal to the number of turns in the parallel case. The results show that parallelism is successfully exploited as the makespan is significantly lower for the parallel algorithm than for the series algorithm in general.

In this chapter, the priorities of the robots were set equal to their identification numbers, which are

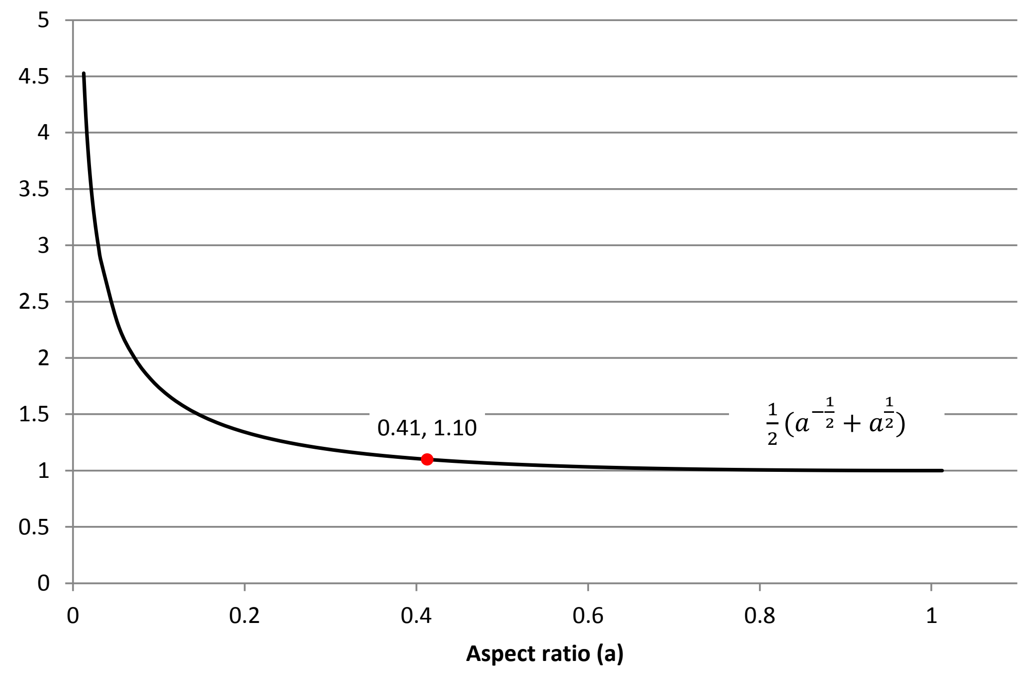


Figure 5.7. The effect of the aspect ratio on the number of moves/turns

arbitrarily assigned. The priorities can, however, be assigned in any way as long as no two priorities are equal. The implementation of robot priorities results in the highest priority robot dictating the movements in the workspace. If only one swap-space is available, a single robot dictates the entire workspace until its mission is complete. Once the dictating robot completes its task, it ceases to give instructions and the robot with the next highest priority assumes control over all robots in the workspace, including the previous dictating robot. If the traffic density is low, the dictated movements tend to be local to the dictating robot. Conversely, if the traffic is dense, it leads to far field dictation.

Interference in the swap path of higher priority robots by lower priority robots can only assist the higher priority robots. The interfering robot's only allowable interference is to move a swap-space onto the dictating robot's swap-path. This swap-space then becomes the closest open space to the dictating robot and the lower priority robot loses its control over it.

The nodes are said to grant or deny requests, which means that they act as robots. The nodes can in fact be seen as simple robots. Every node, however, does not require its own processor or any other type of physical embodiment, apart from the fiducial markers. The nodes can in practice be represented by software robots which can be simulated by a single or multiple servers in the warehouse.

5.7 CONCLUSION

In its current form, the parallel algorithm can be implemented in a synchronous fashion. All robots are required to depart from their starting nodes and reach their destination nodes at the same time. To achieve this, a supervisory controller with good tracking performance would be required. Even under good tracking performance, discrepancies are always present due to communication delays and variances in the robot hardware. If the nodes are used as control points such that each robot is only allowed to advance when all other robots have reached their respective nodes, the executions of the robots is synchronized. This equivalent to a “synchronization barrier” employed in the design of distributed systems, as discussed by Raynal [56]. In the next chapter, the work from this chapter is extended to the continuous time domain under the assumption of asynchronous execution.

CHAPTER 6 ASYNCHRONOUS EXECUTION

6.1 INTRODUCTION

In previous chapters, time is divided into discrete steps. In every discrete time step, robots either stay at their current position, or transition to a neighbouring node. All transitioning robots do so in the same time step, leading to synchronous execution. In real systems, robots exist in continuous time where bodies must accelerate and decelerate to move from point to point. The hardware required to enforce these accelerations is subject to variances which leads to variances between robots in the time required to move between points. Therefore, algorithms need to account for this inherent asynchrony when applied in practice.

The work from the previous chapter can be synchronized by employing a synchronization barrier. Implementing a synchronization barrier could result in a system wide failure if a single robot fails and never reaches its goal. In other words, the stringent requirements on the synchronization of all robots is safe, but is not robust. Furthermore, system-wide waiting is regular side-effect of the synchronization barrier. Therefore, the synchronization barrier incurs an opportunity cost due to lost potential progress.

An algorithm that effectively handles the inherent asynchrony in the system is desirable. Such an algorithm should require minimal communication and dependence amongst robots and should also provide a starvation-freedom guarantee. This chapter is devoted to extending the work to a continuous time, asynchronous version of the visitors' algorithm.

6.2 PROBLEM STATEMENT

In the previous chapter, the planning, requesting and moving phases are synchronized. That is, all robots complete their planning before requesting starts, all nodes grant access before moving starts and all moves are made before the next cycle starts. For the system to predictably solve the visitors' problem, synchronization barriers are required for each of these phases. In other words, timing assumptions on the underlying processes are required to ensure safety and starvation-freedom.

The phases, when abstracted to the asynchronous case, can be thought of as states. In the synchronous case, all robots are in the same state at the same time. In the concept of asynchronous execution, there is no underlying timing assumption on the processes. All processes can execute at different speeds and state transitions can occur at any point in time for a given robot. In an asynchronous system, it is possible for different robots to be in different states at the same time. An algorithm that is suitable for the asynchronous case must ensure that the robots and nodes behave safely under the assumption that the robots are in different states at the same time. Apart from the safety requirement, the system must also be starvation free in order to ensure that all assigned tasks are eventually completed.

By synchronizing all moves in the system, the synchronization barriers synchronize the system such that there are turns. Under the assumption of asynchrony, the notion of a turn is no longer defined as there is no system-wide synchronized start or end of a turn.

6.3 PROPOSED SOLUTION

In an asynchronous system, there are no defined system-wide phases for planning, requesting, granting requests and motion. It has to be decided when to execute each of these operations.

To ensure the safety property, nodes grant only a single request at a time. The requests are only granted when the node is empty. Every node represents a section of the workspace. Nodes are only set to empty once the robot that was granted access to it previously has fully exited the section. Exiting a section is defined as having reached a neighbouring node. This means two or more robots cannot head to the same point in space at the same time and thus collisions are prevented. While a node is occupied, being moved to or being deserted, it cannot grant access and simply accumulates requests. Granting of access is therefore done either when a node is set to empty or when it is empty and receives its first request. Furthermore, when a node grants a request, all other requests at that node are declined. The declined robots receive an expiry notice that must be relayed to the instructing robot. Motion starts when a robot is granted access to a node and ends when it reaches the node that granted the access.

Assigning a task to a robot means that the robot gets a mission. To ensure starvation freedom, any robot that has not completed its mission must always have an intention to do so. The intention is communicated to other robots in the form of an instruction and is known as the pending instruction. As before, robots can instruct themselves too. Robots immediately clear their pending instructions upon receiving an expiry notice from a node.

Planning is done initially when the robot receives its task, or upon expiry of its pending instruction. Instructions expire when they are either granted or declined. A robot's pending instruction can be declined in one of two ways: 1) a requested node grants access to another robot, or 2) an instructed robot requests another robot's instruction.

When the instruction is declined by a robot, the declining robot also sends expiry notifications to all other instructing robots. The reason for expiry for all but one robot is that their instructions have been declined. For a single robot, its instruction expires because the instructed robot is granted access to the instructed node. Expiry notifications trigger replanning for the instructing robots. The state of the workspace continually evolves, which could render old plans null and void. As a result, replanning is done frequently.

All robots, whether their missions are complete or not, must receive and act on their instructions. Acting on an instruction means to request access to the node that a robot is being instructed to. To clearly define a precedence relation, as before, all robots are assigned a unique positive integer priority. When robots have not completed their missions, nodes decline instructions from lower priority robots. Conversely, when robots' missions are complete, nodes grant instructions from robots even if their priorities are lower than the instructed robot.

Requesting is done when the first instruction is received, and is updated upon receiving new instructions. To avoid conflicting situations where robots are granted access to more than one node at a time, they only request access to a single node at a time. As robots continuously receive new instructions, an instruction of higher priority than the requested instruction might arrive. Therefore, robots are required to update their requested instructions continuously as higher priority instructions arrive. When updating the requested instruction, the previously sent request has to be withdrawn from its node in order to prevent receiving access to more than a single node. When the robots' requests are accepted or rejected, the instructing robot must be notified to ensure that it has a pending instruction as long as its mission is not complete. It is the requesting robot's responsibility to relay the expiry notification to the instructing robot when it is declined by a node.

6.4 IMPLEMENTATION

Section 2.4 introduces ALPHABET SOUP: an open source MFS simulator. ALPHABET SOUP provides a continuous time framework to simulate the robot dynamics and is therefore modified for the purposes of this chapter. In these experiments, all word and letter stations as well as buckets are removed and the workspace is represented by a rectangular grid.

Figure 6.1 shows how an arbitrary node and robot's states are synchronized when the robot enters and leaves the node. The node has four distinct states: "empty", "being entered", "occupied" and "being exited". The robot has three distinct states: "waiting", "moving" and "halted". In Figure 6.1, the robot first halts and waits at a neighbouring node before moving toward the node in question. The robot then halts and waits at the node for some time before moving to a neighbouring node. Finally, the robot halts and waits at the neighbouring node at which point the node in question is set to empty.

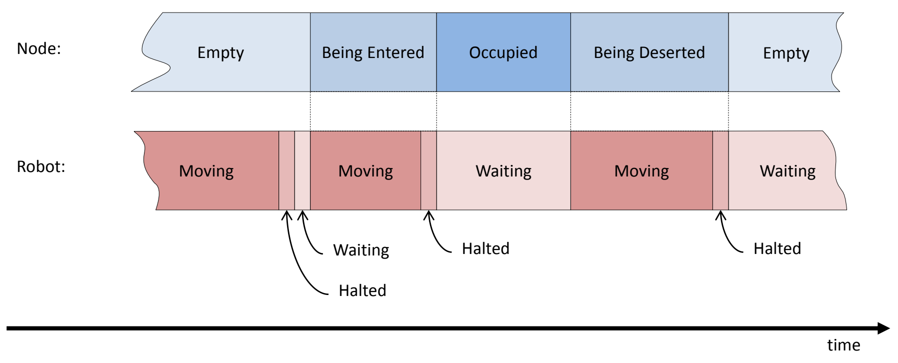


Figure 6.1. Robot/node synchronization of states

Figure 6.2 shows the three states of the robot. The halted state serves two purposes: 1) it prepares the robot to enter the waiting state, and 2) sets its previous node to empty so that other robots can approach the node. Also shown in Figure 6.2, when the robot reaches its next node and comes to a halt, a "set empty" notification is sent to its previous node to indicate that it has successfully exited the node. The code contained in the dashed box in Figure 6.2 manifests on every robot. The variable *Robot* is therefore local to every robot, but for every robot it is global in terms of all its sub-routines. In other words, *Robot* always refers to the robot executing the algorithm.

It is indicated in Figure 6.2 that the robot must be granted access before transitioning from the waiting to the moving state. Conversely, the transition from halted to waiting is unconditional. Within the waiting state shown in Figure 6.2, three sub-states exist: "granted", "planning and propagation" and "declined". The details of the waiting state are shown in Figure 6.3. Tracing through Figure 6.3, it is clear that if the robot is not "granted", not "not idle with a pending request" and not "declined", it continuously checks these three conditions until one of them changes and triggers one of the three sub-state transitions. This happens continually until the "granted" sub-state is triggered, which prepares the robot for the transition to its moving state. Furthermore, the continuous checking, together with the planning sub-state, ensures that a robot always has a pending instruction, unless it has completed its mission and was not assigned a new one.

Robots only plan and communicate their intentions while stationary in the waiting state. The planning procedure is shown in Algorithm 6.1 and is based on the ideas from the previous chapters: The robot plans the route from its current node to its destination node. If the next node on this route is unoccupied, it requests to go there; if the next node is occupied, the robot looks for a swap-path so that its route can become unblocked by instructing other robots to move out of the way along the swap-path. Because robots are now operating in continuous time, the "being entered" and "being deserted" node states have to be taken into consideration. In the planning phase "being entered" is treated as being occupied and "being deserted" is treated as being empty. Algorithm 6.1 employs the Find_swap procedure shown in Algorithm 4.2, but with the important predicate that the BFS looks for the first node that is either empty or being deserted. This version of the algorithm is denoted as Find_swap*. Similarly, when determining which robot to instruct, the instructed robot can either be entering the node or occupying it. Therefore, both robots coming into a node or stationary at a node can take requests from robots that need to use the node. However, departing robots are never instructed.

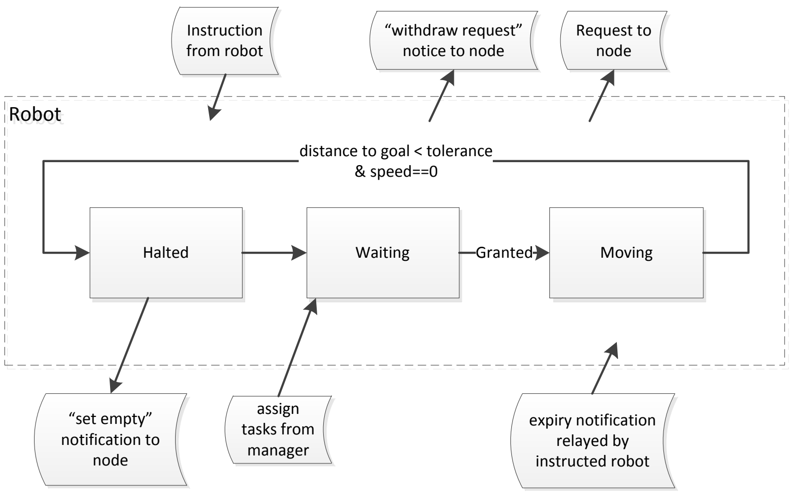

Instruction from robot
"withdraw request" notice to node
Request to node
Robot
distance to goal < tolerance & speed==0
Halted
Waiting
Granted
Moving
"set empty" notification to node
assign tasks from manager
expiry notification relayed by instructed robot

Figure 6.2. Robot state flow

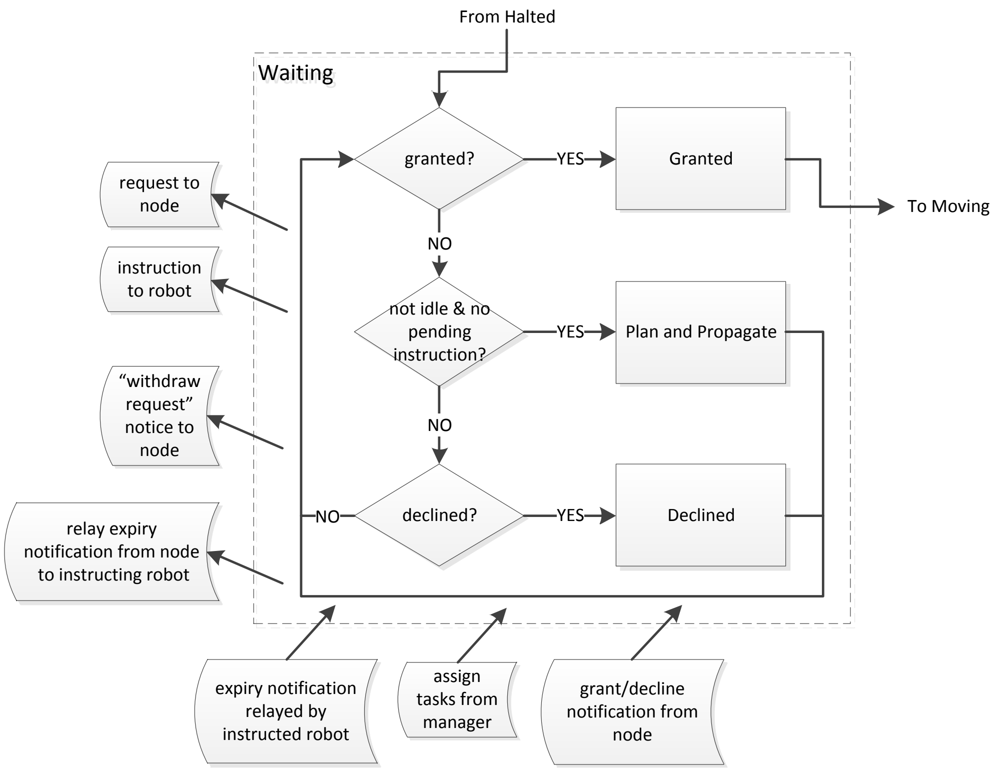

From Halted
Waiting
granted?
YES
Granted
To Moving
NO
not idle & no pending instruction?
YES
Plan and Propagate
NO
declined?
YES
Declined
NO
request to node
instruction to robot
"withdraw request" notice to node
relay expiry notification from node to instructing robot
expiry notification relayed by instructed robot
assign tasks from manager
grant/decline notification from node

Figure 6.3. Robot waiting state

Algorithm 6.1 Plan_and_propagate

```
Current_node ← get Robot's current node
Goal_node ← get Robot's destination node
Route ← plan a path from Current_node to Goal_node using $A^*$
Next_node ← remove first node from Route
if Next_node is empty or being deserted then
    Instruction ← Robot instructs Robot to Next_node
    set Robot's pending instruction to Instruction
    add Instruction to Robot's instructions priority queue
    Update_requested(Robot)
else if Next_node is occupied or being entered then
    Swap-path ← Find_swap$^*$(Current_node, Next_node)
    Unoccupied ← remove first node from Swap_path
    To_swap_node ← remove next node from Swap_path
    To_swap_robot ← get the occupying or entering robot at To_swap_node
    Instruction ← Robot instructs To_swap_robot to To_swap_node
    set Robot's pending instruction to Instruction
    add Instruction to To_swap_robot's instructions priority queue
    Update_requested(To_swap_robot)
end if
```

Robots maintain a priority queue of instructions. This allows the robots to accumulate instructions when moving and accept multiple instructions from different robots. The highest priority instruction from each robot's queue is sent to its corresponding node. When a new instruction is given to a robot, the robot's requested instruction is updated in order to ensure that the requested instruction is always the one with the highest priority. The procedure by which the requested instruction is updated is shown in Algorithm 6.2. Firstly, if the robot has a requested instruction, it withdraws the instruction from its corresponding node. The robot also clears its requested instruction, but does not remove it from its instructions priority queue. Next, it polls its priority queue for the highest priority instruction. If the corresponding node of the polled instruction has become occupied or is being entered by another robot, the instructing robot's instruction has failed and it is notified with an expiry notification. Furthermore, because the instruction has been polled, it is no longer in the instructed robot's instructions priority queue. If, however, the corresponding node is empty or being deserted, the instructed robot re-enters the instruction into its priority queue and sets the instruction as its requested instruction. The robot keeps polling its instructions priority queue until it finds a non-failed instruction or empties the queue.

Algorithm 6.2 Update_requested

```
Input: To_update_robot
  Withdraw_requested(To_update_robot)
  InstructionsPQ ← get To_update_robot's priority queue of instructions
  while InstructionsPQ has elements do
    Updated ← poll InstructionsPQ
    Node ← get Updated's node
    if Node is occupied or being entered then
      Instructing ← get Updated's instructing robot
      Send Instructing an expiry notification
    else
      add Updated to InstructionsPQ
      Set To_update_robot's requested instruction to Updated
      break while
    end if
  end while
```

Both the “Halted” and “Granted” subroutines manage boolean variables that indicate states and sub-states as well as other local variables that hold node addresses for next, previous and current nodes. In addition, the nodes have similar variables for keeping track of their states and which robots are occupying, entering and exiting them. The required assignments for these variables are intuitive and easy to implement in practice, but clutter the pseudo-code. Therefore, these details are not shown in the pseudo-code for the sake of brevity and clarity.

Algorithm 6.3 shows the procedure for the granted sub-state. Once granted, the robot’s pending instruction is cleared. The granted node removes all of its other requests from its priority queue and sends an expiry notification to all declined robots. The granted robot also withdraws its requested instruction from the node using the routine in Algorithm 6.4. Finally, the granted robot clears its instructions priority queue and sends expiry notifications to all robots that gave it instructions.

When executing Algorithm 6.4, a robot removes itself from the requested node’s requests priority queue. The robot also clears its requested instruction. This withdraw operation prevents the robot

Algorithm 6.3 Granted
```
  $Granted \leftarrow Robot$'s granted waypoint
  Clear *Robot*'s pending instruction
  Remove all *Granted*'s requests and send expiry notifications to declined robots
  Withdraw_requested(*Robot*)
  Clear *Robot*'s instructions priority queue and send expiry notifications to declined robots
```

from re-requesting the granted node again next time that its requested instruction is updated. This is critical as it ensures that the robot will not be granted access to a node that is not one of its direct neighbours. In Algorithm 6.4, real robots send a "withdraw request" notification to the node with every invocation of Algorithm 6.4. The sent message is also indicated in Figures 6.2 and 6.3. The ALPHABET SOUP implementation does not simulate these communications. In the ALPHABET SOUP implementation, the "withdraw request" is represented by removing the requested instruction directly from the requested node's priority queue. Similarly, "send expiry notification" in Algorithms 6.2 and 6.3 is represented in ALPHABET SOUP by directly setting the expired robot's pending request to *null*.

Algorithm 6.4 Withdraw_requested
```
Input: To_withdraw_robot
  $Requested \leftarrow$ get *To_withdraw_robot*'s requested instruction
  $Node \leftarrow$ get *Requested*'s node
  Send withdraw request to *Node*
  Clear *To_withdraw_robot*'s requested instruction
```

Requests result from instructions. As the instructed robot is always the one making the requests, the node informs the instructed robot of its success or failure. Upon notification, the instructed robot removes the failed instruction from its instructions priority queue. However, the instructing robot must also be notified of the expiry of its pending instruction. The instructed robot therefore has the task of relaying the grant or decline notice as an expiry notification to the instructing robot. In this sense, instructions are propagated and, when they expire, a notification is back propagated. Therefore, both grant and decline notifications translate to expiry notifications that are relayed to the instructing robot. Finally, the denied robot updates its requested instruction using the same procedure as before. The flow of these messages between nodes and robots is depicted in Figures 6.2 and 6.3. If a robot acts on its own instruction, it relays the expiry notification to itself.

Algorithm 6.5 Declined

```
  Requested ← get requested instruction
  Instructing ← get Requested's instructing robot
  Clear Instructing's pending instruction
  InstructionsPQ ← get robot's priority queue of instructions
  Remove Requested from InstructionsPQ
  Update_requested(Robot)
```

In the moving state, a bang-bang controller controls the robot's heading, acceleration and deceleration. Firstly, the heading is simply set towards the target node. ALPHABET SOUP currently does not model the rotational acceleration and velocity of the robots, changing its heading is therefore instantaneous. This would be a problem if robots were following curved trajectories; however, because robots simply move in straight lines and only change their heading when stationary, the underlying dynamics are equivalent to omni-wheel robot dynamics. Once the heading is set, the robot accelerates at its maximum acceleration until it reaches its maximum velocity, if space allows. The robot then cruises at its maximum velocity until it is its minimum stopping distance away from the target, at which point it starts a maximum deceleration to zero. If the space between the nodes does not allow the robot to reach its maximum velocity, the robot simply goes from maximum acceleration to maximum deceleration, leading to a triangular velocity profile between the start and end points.

Apart from the routines executed by the robots, the nodes are also required to perform certain functions. The control flow of the nodes is shown in Figure 6.4. Requests are sent to the nodes as messages from the robots. If a node is empty and has no other requests, it immediately grants the first request it receives. This results in a first come, first served behaviour when there are no conflicting requests. The nodes notify granted and declined robots by sending them messages as shown in Figure 6.4. The nodes keep a priority queue of requests. The nodes accumulate requests when being moved to or being deserted by adding the requests to their priority queue. Requests are also granted when nodes are set empty. The nodes are set empty by robots that were previously granted access to them and have arrived at their neighbouring nodes after departure. The nodes become empty when they receive a "set empty" notification from the robot. Once the node is empty, it grants the highest priority request from its priority queue. Every time a robot is granted access to a node, the node empties its requests priority queue while notifying all robots whose instructions have failed as well as the single robot whose instruction has succeeded. When robots no longer need access to the node, they send a "withdraw request" notification. Upon receiving this notification, the node removes the request of the said robot from its priority queue.

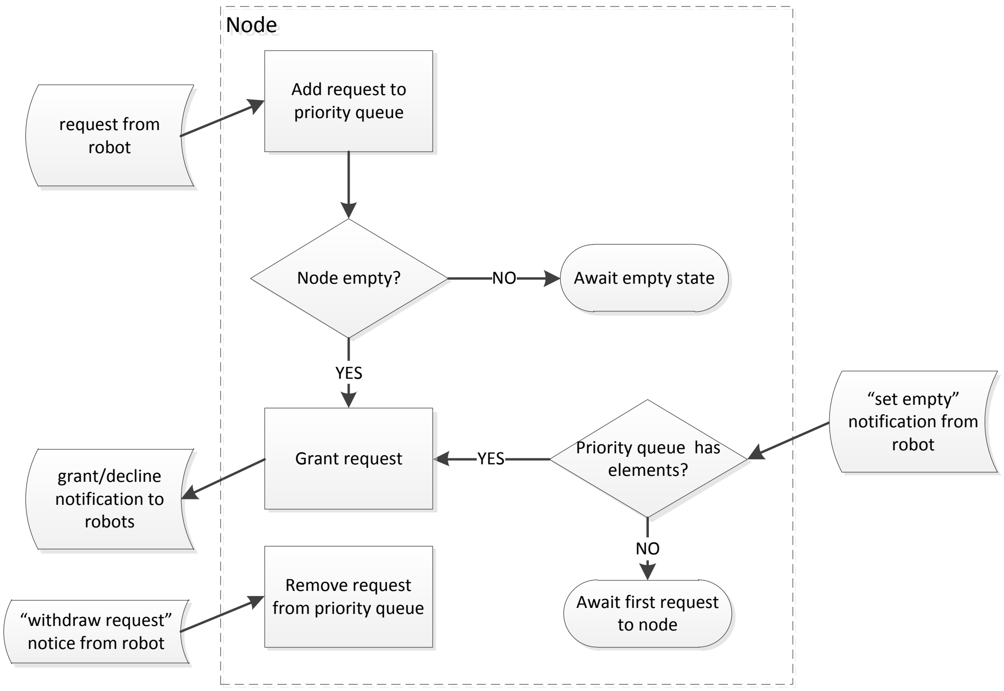

Node
request from robot
Add request to priority queue
Node empty?
NO
Await empty state
YES
"set empty" notification from robot
Grant request
YES
Priority queue has elements?
grant/decline notification to robots
NO
Remove request from priority queue
"withdraw request" notice from robot
Await first request to node

Figure 6.4. Node control flow

The Grant_request sub-routine is shown in Algorithm 6.6. Requests are polled from the priority queue until the queue is empty or a request is granted. Every polled request is tested for its' feasibility. Requests are feasible when the instructed robot is not already granted by another node, and the instructing robot has higher priority than the instructed robot, or when the instructed robot is idle. The first feasible request from the priority queue is granted. Once a request is granted, all other requests, whether they are feasible or not, are removed from the node and expiry notifications are sent to the declined robots.

6.5 STARVATION-FREEDOM

The picking stations can be viewed as clients in the system. The clients cannot afford to wait indefinitely for a product to arrive. In the system under consideration, every robot in the system is executing the process of visiting its destination node. If every process eventually successfully terminates, the clients

Algorithm 6.6 Grant_request

```
while Node's priority queue is not empty do
    Request ← poll Node's priority queue
    Instructed ← get instructed robot from Request
    Instructing ← get instructing robot from Request
    if (Instructing's priority < Instructed's priority OR Instructed is idle) AND Instructed is not
    already granted then
        Send granted notification to Instructed
        break
    else
        Send declined notification to Instructed
    end if
end while
remove all of Node's requests and send declined notifications to their instructed robots
```

do not wait indefinitely for their products. System-wide starvation-freedom is therefore essential in this context and is proved in this section.

Every robot has a unique integer priority. Smaller integers have higher priorities. Every task has a unique integer task number. Every time a new task is created, its number is incremented. Every time a robot is assigned a task, its priority is set equal to the task number. Specifically, the following two cases are considered: 1) every robot is assigned a single task and 2) every robot is assigned a task initially and, when its task is complete, it receives a new one.

Theorem. Every assigned task eventually successfully terminates.

Proof. Consider the highest priority robot in the workspace, which is referred to as the critical robot. The critical robot's instructions are unconditionally obeyed. Every time the critical robot's pending instruction expires, a new pending instruction is created. The new instruction is created before the previous one is completed because completing an instruction takes longer than planning and communicating it. The time for a robot to transition between two nodes is assumed to be finite. The series of instructions generated by the critical robot therefore lead to a series of granted requests, which progress an open space towards the next node on its path if it is blocked. If the next node on the critical robot's path is not blocked, it is guaranteed access to the next node. The waiting time between assignment of a task and its completion for the critical pebble monotonically decreases as time progresses. As a result, the critical robot is guaranteed to eventually complete its assigned task. Upon completion of the critical robot's task, another robot becomes critical. Therefore, all robots eventually complete their missions.

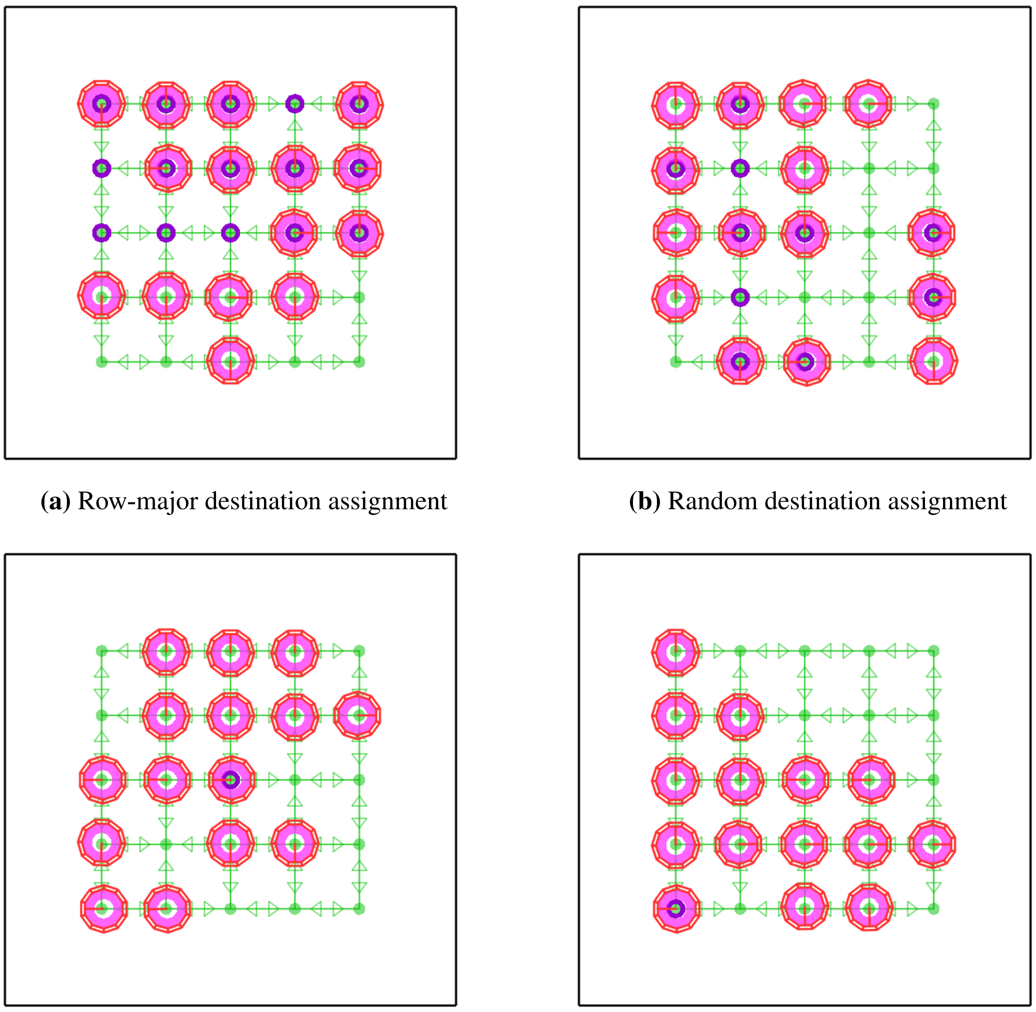
The route of the critical robot is re-planned with every instruction expiry. The length of the route must remain unchanged or reduce in length when re-planning. Similarly, the swap-path may also be re-planned with every instruction expiry. The length of the swap-path must monotonically decrease between subsequent replanning steps.

6.6 RESULTS AND DISCUSSION

Experiments are performed to assess the efficacy of the proposed method. In the experiments, every robot is initially randomly placed at a unique node on the grid. The robots are assigned destination nodes that are to be visited. In Figure 6.5, for the purpose of visualization, the robots colour the destination nodes purple upon visiting them. In addition, robots that have completed their tasks are highlighted with a red border.

The assigned destination nodes in previous chapters were determined by the robot identification number and row-major numbering of the nodes on the grid. This is the original destination assignment of the 15 puzzle and was chosen for the development of the algorithms in this work. Figure 6.5(a) shows the resulting final state for 15 robots on a 5×5 grid. It is clear that all robots have completed their tasks as they are all highlighted with a red border. Counting the number of visited nodes shows 15 highlighted in purple. Therefore, every robot visited its unique target location at least once. The robots are not at their assigned locations at the time when the last robot reaches its target node as this is not a requirement of the visitors' rules.

In contrast with the original 15 puzzle rules, the destinations of the robots need not necessarily be unique. The proposed algorithm is capable of solving any destination assignment for any number of robots as long as a single space is left unoccupied in the workspace. As an example, the destination assignments are made randomly in Figure 6.5(b). The random target assignment leads to more than one



(a) Row-major destination assignment

(b) Random destination assignment

(c) Single central node destination assignment

(d) Single eccentric node destination assignment

Figure 6.5. Various destination assignments successfully completed

robot visiting the same destination, this is revealed by counting the purple nodes and comparing it to the number of robots. Only 10 distinct nodes were visited, but all 15 have completed their tasks.

Under the assumption that more than one robot may be assigned the same destination, the original 15 puzzle rules immediately deem such an assignment infeasible. As opposed to being an optional criterion, this kind of assignment necessitates the assumption that robots need not be at their final location at the final time. Furthermore, this kind of assignment occurs frequently in MFS, for example, when more than a single robot is sent to the same picking station.

In fact, all robots can be sent to the same single node as shown in Figures 6.5(c) and 6.5(d). Any node can be chosen for the assignment, for example in Figure 6.5(c), the central node was chosen and, in Figure 6.5(d), an eccentric node was chosen. In each case, all robots can be seen to have successfully reached their target nodes at least once. Although there is a very large number of possible task assignments, the presented samples are representative of the algorithms' capabilities.

When a robot is transitioning between two nodes, it cannot be stopped midway. If another robot instructs it away from the node that it is heading to, the robot adds the instruction to its priority queue and evaluates its priority queue upon arriving at the next node. If the instructing robot has the appropriate priority, it is possible for the instructed robot to return to its previous node immediately upon arrival at its next node. It is observed that some robots experience oscillations in the case where all robots are sent to the same target node. This is due to the fact that robots do not check whether all robots on their swap-paths are of lower priority before giving instructions. As a result, the first couple of instructions along the swap-path are granted until one of them become rejected. When this happens, the robot at the interface between the allowed and rejected instruction obeys the swap-path instruction, followed by its own instruction, followed by the swap-path instruction again and so on. This is the mechanism that causes the oscillations. The oscillations are not desirable, but do not affect the starvation-freedom of the algorithm.

The purpose of the proposed method is to facilitate starvation-free delivery in MFS. In order not to constrain the study to a single warehouse layout, random cyclic task assignments on various rectangular grids are studied. The assignments are cyclic from the robots' viewpoint: once a robot completes a task, it receives a new one. Firstly, a version of the simulator is developed for debugging and observing the robot behaviour. In this simulator, the robots pause at their target nodes for $1.5s$ while highlighted with a red border as displayed by the Graphical User Interface (GUI) shown in Figure 6.6. This visualizes the experiment in an efficient manner. Asynchrony is introduced into the system by setting the maximum velocity and acceleration of each robot to a bounded random number. Despite the asynchrony, the robots are found to complete their assigned tasks successfully in a finite time. In real systems, communication delays and failures are present, which have not been simulated. The proposed algorithm has proven itself under asynchronous conditions and it is expected that it could be robust against these delays and failures.

It should be noted that the tasks are not required to be completed in a specific order. Figure 6.7 shows the completion order versus task number for various numbers of robots. In the case where there is only

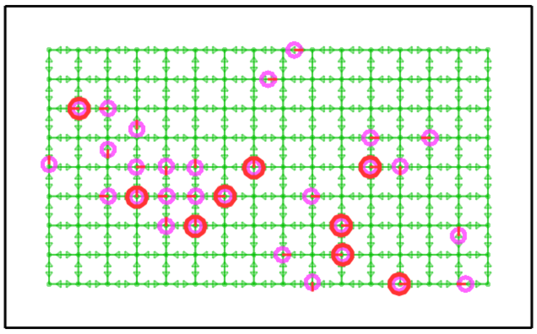


Figure 6.6. GUI for cyclic mission completion with 30 robots on a 9 × 16 grid

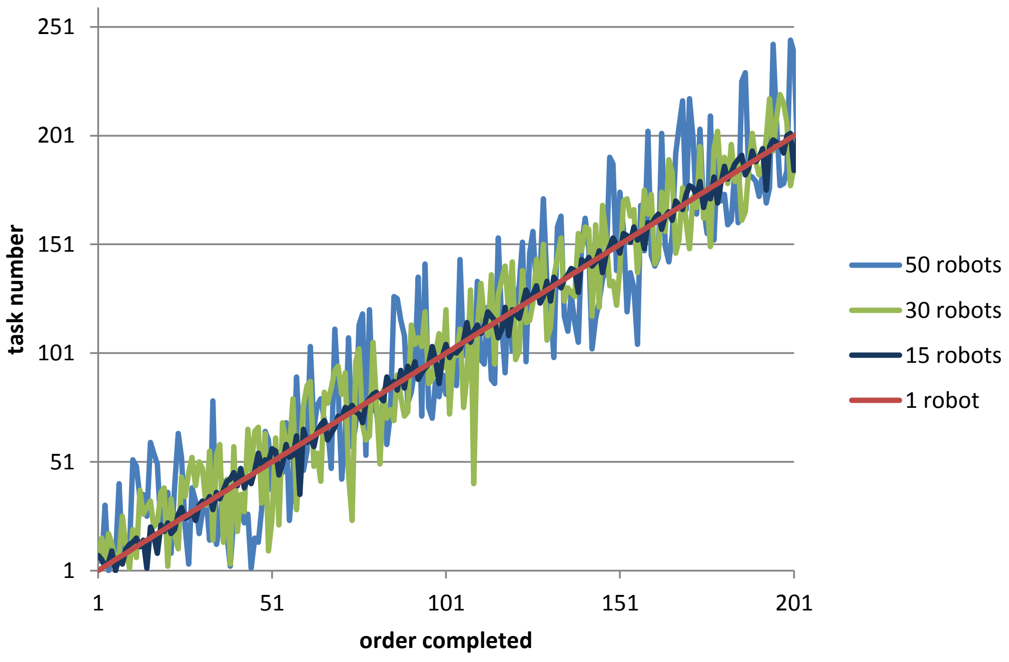


Figure 6.7. Task number versus order of completion

a single robot, the tasks are completed in the order that they are assigned; however, as the number of robots increases, it becomes more likely that a low priority robot will be close to its destination without any obstructions. As a result, the number of tasks that are completed in a different order than they were assigned increases as the number of robots increase. Despite the fact that the completion is not completely ordered, every task is guaranteed to eventually complete. Furthermore, the underlying trend in Figure 6.7 is linearly increasing, which indicates that orders are completed roughly in the order that they are assigned.

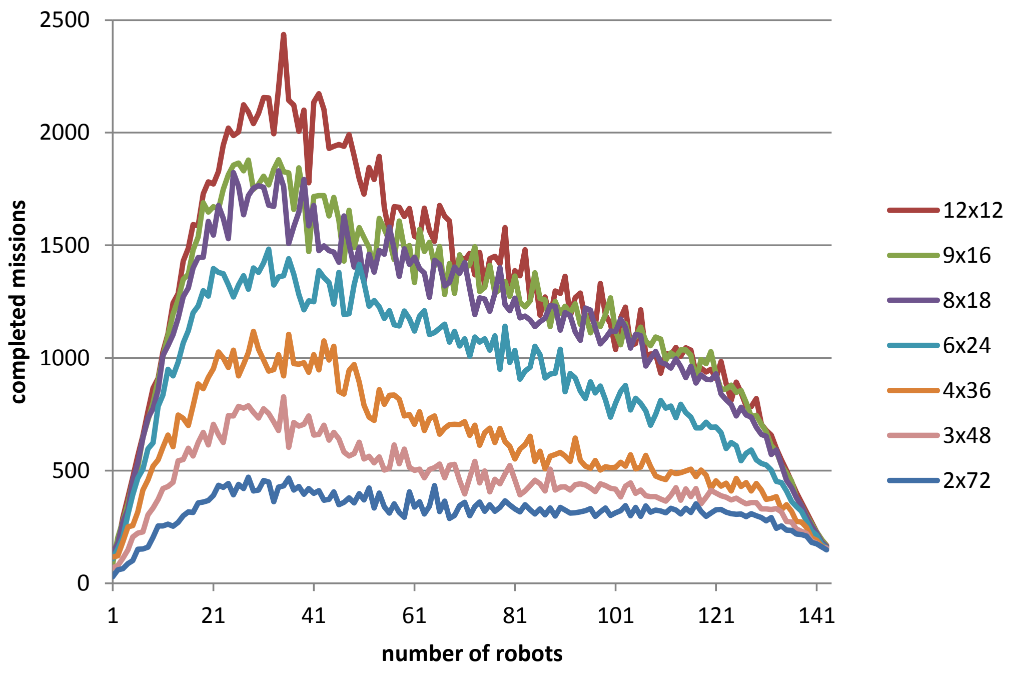
In MFS, the product throughput is to be maximized. The number of completed missions in a fixed period of time serves as a proxy for the throughput. It is expected that the throughput depends on the number of robots and aspect ratio of the grid. In previous experiments, the GUI was used to verify the correct implementation of the proposed algorithm. In the following experiment, the GUI is suppressed in order to accelerate the speed at which trials can be performed. The robots no longer pause at their target nodes; they are immediately assigned a new task and proceed to advance towards it. In the experiment, nodes are placed $1.3m$ apart, robots accelerate at $1.3m/s^2$ and have a maximum velocity of $1.3m/s$. Robots have a radius of $0.4m$ which leaves a tolerance of $0.5m$ between adjacent robots on the grid. The results were obtained using a laptop with an Intel Core i7 with $2.9GHz$ clock-speed and $16GB$ of memory. For the sake of repeatability, the robot velocities and accelerations are not randomized. Once again, the rectangular grid with 144 nodes is chosen to facilitate investigation of the effect of the aspect ratio. For every aspect ratio, the number of robots is swept from 1 to 143. For every number of robots, a trial of 3600 simulated seconds is performed while measuring the required CPU time and the number of completed tasks.

The number of completed missions versus number of robots is shown in Figure 6.8(a). The variance in the result is due to the randomness of the task assignment and initial placement. For all aspect ratios, the maximum throughput is obtained for about 30 robots. This gives a traffic density of $\rho = 30/144 = 0.21$. The highest throughput is achieved when the workspace is square and performance gradually decreases as the aspect ratio decreases. In cases of severely low aspect ratios, the difference in number of completed missions between the optimal traffic density and sub-optimal densities is less pronounced than in cases with aspect ratios closer to $a = 1$. Even though the algorithm is capable of allowing 143 robots to simultaneously use the workspace, for the square grid 30 robots complete around 13 times more tasks in a fixed period of time. Therefore, under normal circumstances the MFS warehouse should not be overfilled to traffic densities exceeding $\rho = 0.21$. In spite of this, the proposed algorithm provides the guarantee of starvation-freedom for $k = 1, 2...N-1$ robots and is therefore robust.

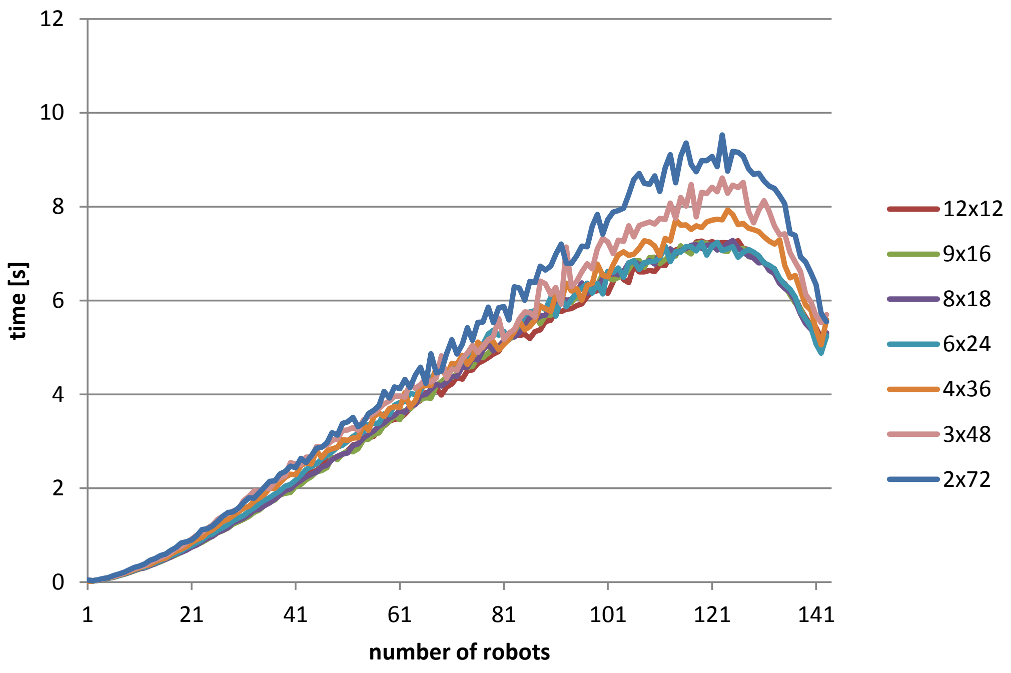
As the aspect ratio becomes smaller, more CPU time is required as seen in Figure 6.8(b). However, all comparative simulation times are in the same order of magnitude, with at most a 32% difference between them. In the high traffic density simulations, the simulation runs about 360 times faster than real time. In simulation, the continuous time dynamics of all the robots in the system, their routes and swap-paths are centrally computed by the CPU. In reality, every robot has its own processor and their routes and swap-paths will be planned independently on k processors. In addition, the robots do not need to simulate their dynamics in the real implementation. It is therefore safe to conclude that the proposed algorithm is fast enough to be implemented in real time. Counter-intuitively, the required CPU time decreases when the traffic density exceeds $\rho = 116/144 = 0.81$. The overall simulation time is dependent on many factors including route and swap-path planning. The time required for route and swap-path planning is dependent on the traffic density, and from the observed results this requirement decreases as the traffic density exceeds $\rho = 0.81$. This is probably due to robots replanning less frequently as a result of their instructions and requests being held in node and robot priority queues for longer intervals before they expire. Finally, it is observed that the simulation times for four of the experiments are almost identical in Figure 6.8(b). Only for aspect ratios below $a = 6/24 = 0.25$ does the run time increase significantly.

6.7 CONCLUSION

The proposed algorithm successfully allows the visitors' problem to be solved in continuous time under severe asynchrony. A proof of starvation-freedom of the algorithm has been provided and extensive simulations show that the algorithm is robust. All target node assignments are feasible and robots can be assigned to go to any node without affecting the starvation-freedom of the algorithm. It has been shown that low traffic density is essential for high throughput and that the algorithm continues to function in severely congested scenarios. It has also been shown that small grid aspect ratios adversely affect the throughput. Finally, the algorithm is shown to be computationally permissible as the simulation is two orders of magnitude faster than real time for the case with 143 robots. The simulation run times are merely an indication of the computational permissibility of the algorithm. Real robots will independently perform planning on k processors leading to increased computational permissibility and scalability.



(a) Number of completed missions versus number of robots for 3600 seconds of simulated time



(b) Required CPU time versus number of robots for 3600 seconds of simulated time

Figure 6.8. Cyclic mission completion in simulated continuous time

CHAPTER 7 CONCLUSION

In this chapter, the findings of this work are summarized, performance guarantees and their limitations are given and the general applicability of the visitors' rules is considered. Lastly, suggestions for further work are given.

7.1 SUMMARY

Most of the proposed approaches to multi-robot path planning presented thus far are not amenable to MFS. The approaches that are amenable to MFS either give no starvation-freedom and asymptotic performance guarantees or are computationally expensive. In this work, an algorithm for the visiting pebble motion on rectangular grids has been proposed. Rectangular grids and workspaces composed of multiple rectangular grids provide an appropriate representation for MFS workspaces.

A serial planning algorithm for the case where only a single swap-space is available has been presented. The proposed algorithm has been extended to the case where multiple unoccupied spaces are available. Further extension allows the algorithm to be executed in parallel with the assumption that all moves are synchronous. Finally, an asynchronous continuous time version of the algorithm is provided.

7.2 GUARANTEES AND LIMITATIONS

The proposed algorithm does not allow rotations and train-like motion for the sake of safety and real-world implementability. The algorithm is capable of finding a solution to any instance on a bi-connected graph as long as a single unoccupied space is available. Any target node assignment is feasible under the proposed algorithm, including assigning multiple robots to the same node. The required number of moves is $O(N^{\frac{3}{2}})$ on rectangular grids, when there is one unoccupied space in the workspace. When more than a single unoccupied space is available, the required number of turns in the synchronized case initially decreases hyperbolically, leading to significant decreases in the makespan with introduction of only a few unoccupied spaces. Robots can plan their routes and swap-paths independently if provided with the state of the workspace. In this case, the computational load for a robot to decide its next move is only that of the BFS and A^* per which is $O(N \log N)$. The asymptotic rate for the number of moves is valid as long as the workspace consists of rectangular sections that share at least two nodes between every intersecting pair of sections. The algorithm is shown to be starvation-free, which means that every assigned task will eventually complete.

7.3 INTENDED USE

The algorithm is intended for use in low traffic density scenarios around $\rho = 0.2$, as this is where maximum throughput is achieved. However, by virtue of the starvation-freedom guarantee, the algorithm is robust to high traffic densities and can keep delivering products as $\rho \to 1$, as long as a single unoccupied space is available.

7.4 APPLICABILITY

Even though the focus is specifically on the requirements of MFS, these requirements can be seen to be applicable to numerous real world MVS problems. The visitors' rules for the pebble motion on graphs are applicable to any problem where point-to-point transport is the objective.

7.5 FURTHER WORK

Optimality did not enjoy high priority in the design of the proposed algorithm. As a result, the swap-paths frequently drive lower priority robots further away from their targets which leads to sub-optimal behaviours. Robots could also benefit from attempting to avoid each other altogether when planning their routes. Therefore, optimization of the proposed algorithm is a likely direction for future work. In addition, it was reported that the asynchronous algorithm leads to noticeable oscillations in the case where all robots are sent to the same node. Further refinement of the algorithm should be capable of remedying this.

It is expected that the asynchronous visitors' algorithm will be resilient to communication delays and failures. However, communication delays and failures have not been simulated in this work and the communications were assumed to be ideal. Future work could study the communications in more detail.

The algorithms have not been implemented in the presence of storage pods, replenishment and picking stations. Further work is required to realize such an implementation. Inclusion of the storage pods presents new difficulties because robots can pass under the pods when not carrying buckets. The swap-path planning algorithm should take into account whether a robot is carrying a storage pod or not in order to obtain an efficient algorithm.

The current node reservation system only attempts to reserve one node ahead on the robots' paths. This leads to robots that stop at every node on their way to their targets. In other words, a toothed velocity profile results from the current space reservation mechanism. This is sub-optimal from a makespan point of view. In future work, the space reservation system can be modified so that the robots only accelerate and decelerate when necessary so that the velocity profile is not toothed. Furthermore, the current version of ALPHABET SOUP does not implement rotational accelerations and velocities for the robots. This can be seen as a realistic assumption if the robots are omni-wheeled units. However, the robots used in most industrial applications are differential drive units. Further work could implement the differential drive dynamics of the robots in order to simulate the industrial conditions more accurately.

Robot breakdowns were not considered in this work. The proposed algorithms could effectively treat breakdowns by simply excluding the nodes where breakdowns are blocking the robot paths. If the resulting graph remains bi-connected, the algorithms would be able to find a solution, otherwise a complete algorithm is required. The presented algorithms are complete for bi-connected graphs, but are in general not complete. Completeness is a desirable property for any multi-agent motion coordination algorithm. The provided visitors' rules form the basis from which complete algorithms can be derived in future work.

REFERENCES

[1] D. M. Kornhauser, G. L. Miller, and P. G. Spirakis, “Coordinating pebble motion on graphs, the diameter of permutation groups, and applications,” Master’s thesis, M. I. T., Dept. of Electrical Engineering and Computer Science, Cambridge, 1984.

[2] P. R. Wurman, R. D’Andrea, and M. Mountz, “Coordinating hundreds of cooperative, autonomous vehicles in warehouses,” *AI magazine*, vol. 29, no. 1, pp. 9–19, 2008.

[3] H. Andreasson, A. Bouguerra, M. Cirillo, D. N. Dimitrov, D. Driankov, L. Karlsson, A. J. Lilienthal, F. Pecora, J. P. Saarinen, A. Sherikov, and T. Stoyanov, “Autonomous transport vehicles: Where we are and what is missing,” *IEEE Robotics Automation Magazine*, vol. 22, no. 1, pp. 64–75, March 2015.

[4] E. Guizzo, “Three engineers, hundreds of robots, one warehouse,” *IEEE spectrum*, vol. 45, no. 7, pp. 26–34, 2008.

[5] Otto motors. Clearpath Robotics, Inc. (Last accessed 2017/08/01). [Online]. Available: https://www.ottomotors.com

[6] fetch robotics. (Last accessed 2017/08/01). [Online]. Available: http://www.fetchrobotics.com

[7] E. Ackerman. (2017, apr) Fetch robotics introduces burly new freight robots. IEEE Spectrum. (Last accessed 2017/08/01). [Online]. Available: http://spectrum.ieee.org/automaton/robotics/industrial-robots/fetch-robotics-introduces-burly-new-freight-robots

[8] K. Bhasin and P. Clark. (2016, jun) How amazon triggered a robot arms race. Bloomberg. (Last accessed 2017/07/19). [Online]. Available: http://www.chicagotribune.com/bluesky/technology/ct-amazon-distribution-center-robots-20160629-story.html

[9] M. Wulfraat. (2012, feb) Is kiva systems a good fit for your distribution center? an unbiased consultant evaluation. MWPVL International Inc. (Last accessed 2017/07/19). [Online]. Available: http://www.mwpvl.com/html/kiva_systems.html

[10] J. Slocum and E. W. Weisstein. "15 Puzzle.". MathWorld–A Wolfram Web Resource. (Last accessed on 2017/07/19). [Online]. Available: http://mathworld.wolfram.com/15Puzzle.html

[11] A. Adler, M. de Berg, D. Halperin, and K. Solovey, "Efficient multi-robot motion planning for unlabeled discs in simple polygons," *Algorithmic Foundations of Robotics XI: Selected Contributions of the Eleventh International Workshop on the Algorithmic Foundations of Robotics*, pp. 1–17, 2015.

[12] J. Yu and D. Rus, "Pebble motion on graphs with rotations: Efficient feasibility tests and planning algorithms," *Algorithmic Foundations of Robotics XI: Selected Contributions of the Eleventh International Workshop on the Algorithmic Foundations of Robotics*, pp. 729–746, 2015.

[13] G. Sharon, R. Stern, M. Goldenberg, and A. Felner, "The increasing cost tree search for optimal multi-agent pathfinding," *Artificial Intelligence*, vol. 195, pp. 470–495, 2013.

[14] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, "Conflict-based search for optimal multi-agent pathfinding," *Artificial Intelligence*, vol. 219, pp. 40–66, 2015.

[15] B. de Wilde, A. W. ter Mors, and C. Witteveen, "Push and rotate: a complete multi-agent pathfinding algorithm," *Journal of Artificial Intelligence Research*, vol. 51, pp. 443–492, 2014.

[16] M. R. K. Ryan, "Exploiting subgraph structure in multi-robot path planning," *Journal of Artificial Intelligence Research*, vol. 31, pp. 497–542, 2008.

[17] T. Lamballais, D. Roy, and M. B. M. de Koster, “Estimating performance in a robotic mobile fulfillment system,” *European Journal of Operational Research*, vol. 256, no. 3, pp. 976–990, 2017.

[18] M. Mansouri, H. Andreasson, and F. Pecora, “Hybrid reasoning for multi-robot drill planning in open-pit mines,” *Acta Polytechnica*, vol. 56, no. 1, pp. 47–56, 2016.

[19] S. A. Reveliotis and E. Roszkowska, “Conflict resolution in free-ranging multivehicle systems: A resource allocation paradigm,” *IEEE Transactions on Robotics*, vol. 27, no. 2, pp. 283–296, 2011.

[20] R. M. Wilson, “Graph puzzles, homotopy, and the alternating group,” *Journal of Combinatorial Theory, Series B*, vol. 16, no. 1, pp. 86–96, 1974.

[21] K. Solovey and D. Halperin, “k-color multi-robot motion planning,” *The International Journal of Robotics Research*, vol. 33, no. 1, pp. 82–97, 2014.

[22] P. Surynek, “Solving abstract cooperative path-finding in densely populated environments,” *Computational Intelligence*, vol. 30, no. 2, pp. 402–450, 2014.

[23] W. W. Johnson and W. E. Story, “Notes on the "15" puzzle,” *American Journal of Mathematics*, vol. 2, no. 4, pp. 397–404, 1879.

[24] M. Peasgood, C. Clark, and J. McPhee, “Complete and scalable multi-robot roadmap coordination planning,” *IEEE Transactions on Robotics*, vol. 24, no. 2, pp. 283–292, 2008.

[25] K.-H. C. Wang and A. Botea, “Mapp: a scalable multi-agent path planning algorithm with tractability and completeness guarantees,” *Journal of Artificial Intelligence Research*, vol. 42, pp. 55–90, 2011.

[26] H. Roozbehani and R. D’Andrea, “Adaptive highways on a grid,” *Robotics Research*, vol. 70, pp. 661–680, 2011.

[27] P. E. Hart, N. J. Nilsson, and B. Raphael, “A formal basis for the heuristic determination of minimum cost paths,” *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[28] E. W. Dijkstra, “A note on two problems in connexion with graphs,” *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[29] E. F. Moore, “The shortest path through a maze,” in *Proceedings of the International Symposium on the Theory of Switching*. Harvard University Press, 1959, pp. 285–292.

[30] R. J. Luna and K. E. Bekris, “Push and swap: Fast cooperative path-finding with completeness guarantees,” in *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*, 2011, pp. 294–300.

[31] M. M. Khorshid, R. C. Holte, and N. R. Sturtevant, “A polynomial-time algorithm for non-optimal multi-agent pathfinding,” in *Proceedings of the Fourth Annual Symposium on Combinatorial Search*, 2011, pp. 76–83.

[32] P. Surynek, “Towards optimal cooperative path planning in hard setups through satisfiability solving,” in *Proceedings of the 12th Pacific Rim international conference on Trends in Artificial Intelligence*, 2012, pp. 564–576.

[33] T. S. Standley, “Finding optimal solutions to cooperative pathfinding problems,” in *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010, pp. 28–29.

[34] D. Kornhauser, G. Miller, and P. Spirakis, “Coordinating pebble motion on graphs, the diameter of permutation groups, and applications,” in *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*, 1984, pp. 241–250.

[35] P. Surynek, “Mutex reasoning in cooperative path finding modeled as propositional satisfiability,” in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2013, pp. 4326–4331.

[36] T. S. Standley and R. Korf, “Complete algorithms for cooperative pathfinding problems,” in *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, 2011, pp. 668–673.

[37] B. de Wilde, A. W. ter Mors, and C. Witteveen, “Push and rotate: cooperative multi-agent path planning,” in *Proceedings of the International Conference on Autonomous Agents and multi-agent systems*, 2013, pp. 87–94.

[38] M. R. Ryan, “Graph decomposition for efficient multi-robot path planning,” in *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, 2007, pp. 2003–2008.

[39] D. Ratner and M. K. Warmuth, “Finding a shortest solution for the $n \times n$ extension of the 15-puzzle is intractable,” in *Association for the Advancement of Artificial Intelligence*, 1986, pp. 168–172.

[40] J. Enright and P. R. Wurman, “Optimization and coordinated autonomy in mobile fulfillment systems,” in *Proceedings of the 9th AAAI Conference on Automated Action Planning for Autonomous Mobile Robots*, 2011, pp. 33–38.

[41] R. D’Andrea and P. Wurman, “Future challenges of coordinating hundreds of autonomous vehicles in distribution facilities,” in *Proceedings of the IEEE International Conference on Technologies for Practical Robot Applications*, 2008, pp. 80–83.

[42] C. J. Hazard, P. R. Wurman, and R. D’Andrea, “Alphabet soup: A testbed for studying resource allocation in multi-vehicle systems,” in *AAAI Workshop on Auction Mechanisms for Robot Coordination*, 2006, pp. 23–30. [Online]. Available: research.csc.ncsu.edu/alphabetsoup

[43] A. W. ter Mors, “Conflict-free route planning in dynamic environments,” in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2011, pp. 2166–2171.

[44] V. R. Desaraju and J. P. How, “Decentralized path planning for multi-agent teams in complex environments using rapidly-exploring random trees,” in *Proceedings of the IEEE International Conference on Robotics and Automation*, 2011, pp. 4956–4961.

[45] L. Kalinovcic, T. Petrovic, S. Bogdan, and V. Bobanac, “Modified banker’s algorithm for scheduling in multi-agv systems,” in *Proceedings of the IEEE International Conference on Automation Science and Engineering*, 2011, pp. 351–356.

[46] V. Bobanac and S. Bogdan, “Routing and scheduling in multi-agv systems based on dynamic banker algorithm,” in *Proceedings of the 16th Mediterranean Conference on Control and Automation*, 2008, pp. 1168–1173.

[47] T. Petrovic, “Path assignment and resource allocation control in reconfigurable multi-vehicle system,” in *Proceedings of the IEEE Conference on Control Applications*, 2014, pp. 1789–1794.

[48] M. Kloetzer, C. Mahulea, and J. M. Colom, “Petri net approach for deadlock prevention in robot planning,” in *Proceedings of the 18th IEEE Conference on Emerging Technologies and Factory Automation*, 2013, pp. 1–4.

[49] G. Röger and M. Helmert, “Non-optimal multi-agent pathfinding is solved (since 1984),” in *Proceedings of the Fifth Annual Symposium on Combinatorial Search*, 2012, pp. 173–174.

[50] P. Surynek, “A novel approach to path planning for multiple robots in bi-connected graphs,” in *Proceedings of the IEEE International Conference on Robotics and Automation*, 2009, pp. 3613–3619.

[51] J. van Den Berg, J. Snoeyink, M. C. Lin, and D. Manocha, “Centralized path planning for multiple robots: Optimal decoupling into sequential plans,” in *Robotics: Science and systems*, vol. 2, no. 2.5, 2009, pp. 2–3.

[52] P. Surynek, “Makespan optimal solving of cooperative path-finding via reductions to propositional satisfiability,” (2016,oct), arXiv preprint. Last accessed 2017/06/10. [Online]. Available: http://arxiv.org/abs/1610.05452

[53] H. Ma, S. Koenig, N. Ayanian, L. Cohen, W. Hoenig, S. Kumar, T. Uras, H. Xu, C. Tovey, and G. Sharon., “Overview: Generalizations of multi-agent path finding to real-world

scenarios," (2017,feb), arXiv preprint. Last accessed 2017/07/19. [Online]. Available: https://arxiv.org/abs/1702.05515

[54] O. Goldreich, "Finding the shortest move-sequence in the graph-generalized 15-puzzle is np-hard," 1984, laboratory for Computer Science, Massachusetts Institute of Technology, Unpublished manuscript.

[55] E. W. Dijkstra, "Een algorithme ter voorkoming van de dodelijke omarming," n.d., circulated privately. [Online]. Available: http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD108.PDF

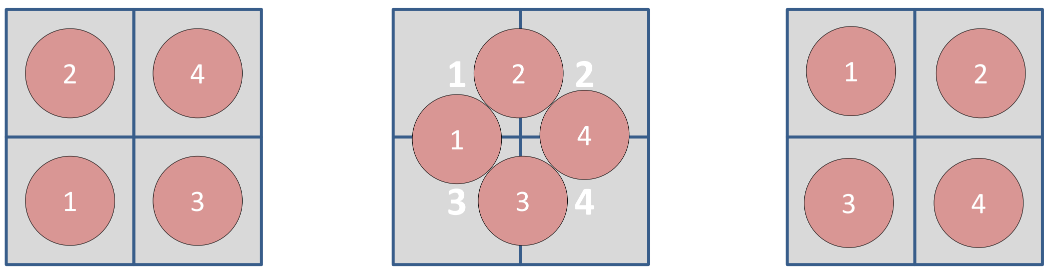
[56] M. Raynal, *Concurrent Programming: Algorithms, Principles, and Foundations*, 1st ed. Springer-Verlag Berlin Heidelberg, 2013.

[57] S. S. Skiena, *The Algorithm Design Manual*, 2nd ed. Springer-Verlag London, 2008.

ADDENDUM

ADDENDUM A ROTATIONS AND TRAIN-LIKE MOTIONS

Recent research proposes that groups of robots on a cycle in a graph should be allowed to undergo "rotations" [12, 15, 33, 37]. These works consider the robots to move in discrete time steps. A rotation along a cycle is represented by an instantaneous transition from the discrete instant shown in Figure A.1(a) to that of Figure A.1(c) for four robots. However, if the robots are represented by squares on a grid (like for example in the 15 puzzle) this is physically not possible unless the grid cells are much larger than the robots. Generally, the grid cells are just large enough to contain the robots as to make efficient use of floor space and allow for reservation based collision avoidance. These difficulties arise because robots exist not in discrete time-steps but in continuous time. Robots also occupy space, which space can be represented by the rotation envelope with a radius r as depicted by the blue circle in Figure 2.2(a). Therefore, the rotation along a cycle, embodied in continuous time and 2D space is accompanied by an instant of either collision or minimum separation as shown in Figure A.1(b).



(a) Configuration before rotation **(b)** Minimum separation instant **(c)** Configuration after rotation

Figure A.1. Robots undergoing a rotation without space reservation

To analyse the separation properties of such a system of robots, consider the case where robots occupying nodes i and j simultaneously move toward nodes j and k respectively as shown in Figure A.2. Assuming that both robots depart at the same time and accelerate at the same rate, the distance δ from their respective start nodes is the same for both robots.

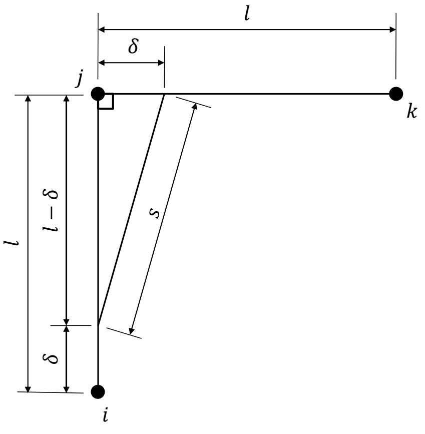


Figure A.2. Separation between two robots under simultaneous motion.

By the Pythagorean theorem, the separation $s = f(\delta)$ between the robots is given by:

$$s = \sqrt{2\delta^2 - 2l\delta + l^2} \tag{A.1}$$

Taking the distance between nodes l as a constant, Equation (A.1) can be viewed as the square root of a second order polynomial. By inspection, the polynomial is convex and therefore has a minimum in the interval $\delta \in (0,l)$. To find the minimum of s, the derivative is taken:

$$\frac{ds}{d\delta} = \frac{2\delta - l}{\sqrt{2\delta^2 - 2l\delta + l^2}} \tag{A.2}$$

The minimum separation $s_{\min}$ occurs at δ^*, which is found by setting $\frac{ds}{d\delta} = 0 \rightarrow \delta^* = \frac{1}{2}l$. Substituting δ^* into Equation (A.1) leads to $s_{\min} = \sqrt{\frac{1}{2}}l$.

Assuming that all robots have a rotation envelope with radius r, in order to avoid collisions it needs to be ensured that $s(\delta) > 2r, \delta \in [0,l]$. Therefore $s_{\min} > 2r \rightarrow l > 2^{\frac{3}{2}}r$. If a $h \times w$ grid that allows for rotations is desired, the minimum required area A' is given as:

$$A' = (w2^{\frac{3}{2}}r)(h2^{\frac{3}{2}}r) \tag{A.3}$$

$$A' = 2^3r^2wh \tag{A.4}$$

Conversely, if rotations are not allowed, the required area becomes:

$$A = (w2r)(h2r) \tag{A.5}$$

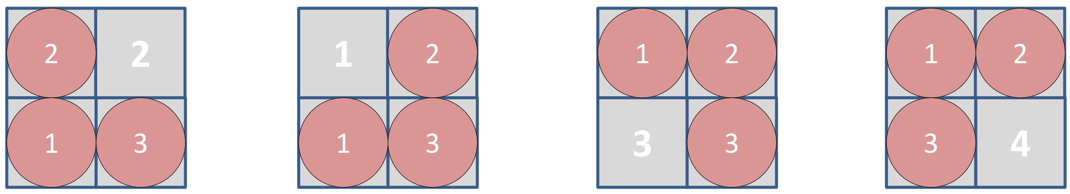
$$A = 2^2 r^2 wh \tag{A.6}$$

It is therefore clear that

$$A' = 2A \tag{A.7}$$

This shows that at least double the area is required to allow rotations without collisions. In order to visualize the magnitude of the difference in space requirements, Figures A.2 and A.3 are drawn to scale with respect to each other and have the same envelope radius r. Warehouse floor space is a valuable asset which should be efficiently utilized; therefore, allowing rotations is costly in this regard.

In order to enable rotations as in Figure A.2, the reservation strategy must be abandoned: robots are allowed to approach nodes that are occupied or being deserted by other robots. This introduces serious risks and technical difficulties in order to ensure safety: precise tracking performance would be required in the case of a centralized control strategy. On the other hand, the reservation based collision avoidance strategy has proven itself to be technically feasible and safe. Similarly, other research [22] assumes train-like motions where robots can move along a straight line without requiring an unoccupied space. The reservation based collision avoidance strategy does not allow for rotations or train-like motions as it requires a cell to be empty before it can be reserved.



(a) $turn = 0$ **(b)** $turn = 1$ **(c)** $turn = 2$ **(d)** $turn = 3$

Figure A.3. Robots progressing along a graph cycle with space reservation.

ADDENDUM B COMPLEXITY ON RECTANGULAR GRIDS

The proposed algorithm in Chapter 3 provides a solution to the visitors' puzzle on a $h \times w$ grid with one unoccupied node. The grid is shown in Figure B.1. The upper bound on the number of moves required to solve a specific instance of this puzzle is of interest, as it governs the worst-case required time to complete a given task assignment in industrial applications.

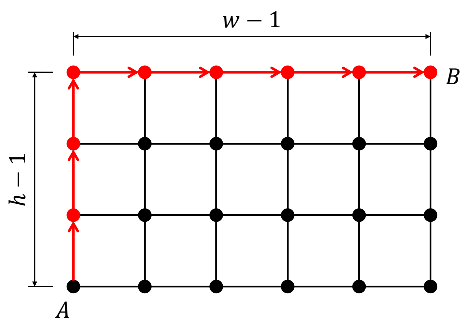


Figure B.1. $w \times h$ grid and its diameter.

The number of nodes N on the grid is:

$$N = wh \tag{B.1}$$

The diameter d of a graph is the longest shortest path in the graph. The diameter of a grid, shown as the red path in Figure B.1, is given as:

$$d = h + w - 2 \tag{B.2}$$

The aspect ratio a of the grid is defined as:

$$a = h/w \tag{B.3}$$

Substitution of Equation B.3 in Equations B.2 and B.1 yields:

$$d = w(1+a) - 2, \tag{B.4}$$

$$\text{and } w = a^{\frac{1}{2}} N^{\frac{1}{2}} \tag{B.5}$$

which leads to:

$$d = N^{\frac{1}{2}}(a^{-\frac{1}{2}} + a^{\frac{1}{2}}) - 2 \tag{B.6}$$

For an arbitrary robot r_i, the distance to move from its starting node to its destination node is denoted as g_r. In the worst case, every r_i has to travel the entire distance d, thus g_r is given by:

$$g_r = (d-1) + cd \tag{B.7}$$

The first term in Equation B.7 is the cost associated with moving the swap-space onto the first node in r_i's route. The second term accounts for moving r_i along its route. The coefficient c accounts for the moves associated with repeatedly circumventing r_i to get the swap-space back onto r_i's route after each of its moves. The grid is represented by the graph $G(V,E)$. Let G' be such that:

$$G' = G - \{n\} \tag{B.8}$$

where $n \in V$. Let P be the shortest path in G' between any of n's neighbours in G. For a four-connected grid, the property holds that $|P| <= 4$. Because $c = 1 + |P|$, it follows that, for any rectangular grid, it is valid to assume:

$$c = O(1) \tag{B.9}$$

The number of robots on the grid is denoted as k. The maximum number of moves g to get k robots from their start to destination vertices is therefore given as:

$$g = kg_r \tag{B.10}$$

In the worst case, the traffic density is very high and there is only one unoccupied node on the grid, which gives $k = N - 1$. By substitution, this leads to:

$$g = (N^{\frac{3}{2}} - N^{\frac{1}{2}})(a^{-\frac{1}{2}} + a^{\frac{1}{2}})(1+c) - N(2c+3) + 2c + 3 \tag{B.11}$$

which clearly shows that:

$$g = O(N^{\frac{3}{2}}) \tag{B.12}$$

ADDENDUM C COMPLEXITY ON MAPS COMPOSED OF RECTANGULAR GRIDS

Consider a warehouse floor composed of n_s rectangular sections. Two sections are said to intersect each other if they share at least one node. An arbitrary section is denoted by s_i. The smallest allowed dimensions of a sub-section are $w \geq 2, h \geq 2$. Assume all pairs $\{s_i, s_j | i \neq j\}$ of intersecting sections share at least two nodes as shown in Figure C.1, then the assumption shown in Equation (B.9) that $c = O(1)$ holds.

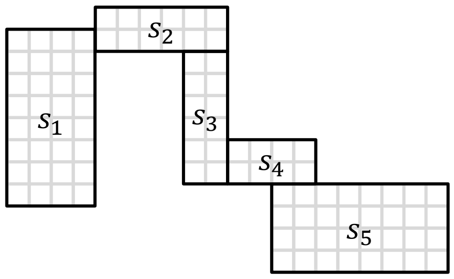


Figure C.1. A warehouse map composed of multiple rectangular grids

Each s_i consists of n_i nodes, has a diameter of d_i and aspect ratio a_i. The diameter of the composite graph consisting of all s_i is D. The diameter of a graph composed of rectangular grids is always smaller than the sum of the diameters of the grids:

$$D \leq \sum_{i=1}^{n_s} d_i \tag{C.1}$$

Where d_i is given by Equation (B.6) with addition of the subscript. The total number of nodes in G is N:

$$N = \sum_{i=1}^{n_s} n_i \tag{C.2}$$

The number of nodes in s_i is a fraction f_i of N, which can be written as:

$$n_i = f_i N \tag{C.3}$$

Previously, Equation (B.6) showed for a single rectangular grid:

$$d = N^{\frac{1}{2}}(a^{-\frac{1}{2}} + a^{\frac{1}{2}}) - 2 \tag{C.4}$$

Adapting Equation (C.4) for the diameter of a section and substitution of Equation (C.3) gives:

$$d_i = N^{\frac{1}{2}} f_i (a_i^{-\frac{1}{2}} + a_i^{\frac{1}{2}}) - 2 \tag{C.5}$$

Whereby Equation (C.1) can be re-written as:

$$D \leq N^{\frac{1}{2}} \sum_{i=1}^{n_s} f_i (a_i^{-\frac{1}{2}} + a_i^{\frac{1}{2}}) - 2n_s \tag{C.6}$$

Previously for a single rectangular grid, it was shown in Equation (B.7) that the cost to move a single robot from its start to its destination is in the worst case given by:

$$g_r = (d-1) + cd \tag{C.7}$$

From the knowledge of Equation (C.1), in this case Equation (C.7) becomes :

$$g_r \leq (D-1) + cD \tag{C.8}$$

Equation (B.10) remains unchanged:

$$g = k g_r \tag{C.9}$$

Substitution of Equation (C.8) into Equation (C.9) and letting $k = N - 1$ yields:

$$g \leq (N-1)(D(1+c) - 1) \tag{C.10}$$

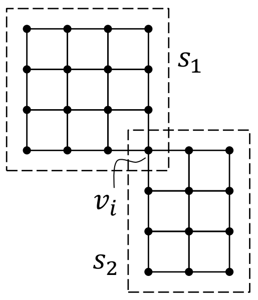
By substitution of Equation (C.6), this finally gives:

$$g \leq (N^{\frac{3}{2}} - N^{\frac{1}{2}})(1+c) \sum_{i=1}^{n_s} f_i (a_i^{-\frac{1}{2}} + a_i^{\frac{1}{2}}) - N + 1 \tag{C.11}$$

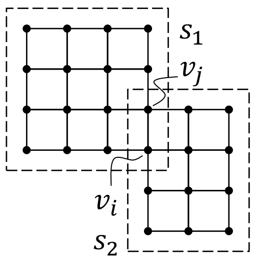
Which is $O(N^{\frac{3}{2}})$. This shows that the complexity of the proposed algorithm in Chapter 3 is the same for rectangular grids and any graph composed of multiple rectangular grids such that all pairs of intersecting sections share at least two nodes.

ADDENDUM D   COMPLETENESS

Consider the case where there is a graph $G(V,E)$ with N nodes and $k = N - 1$ robots at distinct vertices $\in V$. G consists of two intersecting bi-connected components s_1 and s_2. In Figure D.1(a), suppose that a robot is attempting to move from a vertex v_2 in section s_2 to a vertex v_1 in s_1, such that $v_1, v_2 \neq v_i$, using the algorithm in Chapter 3. If the robot is at vertex v_i on its way to a vertex in s_1, but s_1 has no swap-space, then the robot will attempt to move a swap space into s_1 from s_2. However, the swap-path may not pass through vertex v_i as this would reverse the robot's progress. The robot thus considers $G - \{v_i\}$, which is a disconnected graph, when planning the swap-path. Because $s_1 - \{v_i\}$ and $s_2 - \{v_i\}$ are the two connected components of $G - \{v_i\}$, there exists no path between them. The robot is therefore stuck at v_i and the algorithm has failed.



(a) A graph that is not bi-connected



(b) A biconnected graph

Figure D.1. Singly connected and biconnected graphs

Conversely, if s_1 and s_2 share at least two vertices v_i and v_j, for any vertex $v_m \in V$, the graph $G - \{v_m\}$ is connected and a swap-path will always be found for the swap-space between any pair of sub-graphs in G. This is the definition of a bi-connected graph and the proposed algorithm is thus complete for bi-connected graphs.

ADDENDUM E TIME COMPLEXITY OF THE SERIES AND PARALLEL ALGORITHMS

Measuring the time taken to solve the same size experiments using the series and parallel visitors' algorithms yields significantly different run times. The differences are the result of the re-calculation of the swap-paths and routes in every turn for every robot in the parallel algorithm. In this section, the asymptotic rates for both algorithms are analysed in order to understand how they scale with regards to their input size.

E.1 TIME COMPLEXITY OF BFS

The complexity of BFS can be written as:

$$g_{BFS} = O(|V| + |E|) \tag{E.1}$$

where $|V|$ and $|E|$ are the cardinalities of the vertex and edge sets respectively. On rectangular grids of dimension $h \times w$, the number of vertices:

$$|V| = wh \tag{E.2}$$

$$= N \tag{E.3}$$

and number of edges:

$$|E| = w(h-1) + (w-1)h \tag{E.4}$$

$$= 2wh - w - h \tag{E.5}$$

By manipulation of the definition of the aspect ratio, given in Equation (B.3):

$$h = aw \tag{E.6}$$

$$w = a^{-\frac{1}{2}} N^{\frac{1}{2}} \tag{E.7}$$

$$\therefore\ |E| = 2N - N^{\frac{1}{2}}(a^{-\frac{1}{2}} + a^{\frac{1}{2}}) \tag{E.8}$$

$$\text{and } |E| = O(N) \tag{E.9}$$

which gives:

$$g_{BFS} = O(N + O(N)) \tag{E.10}$$

$$= O(N) \tag{E.11}$$

E.2 TIME COMPLEXITY OF A^*

In the worst case, the time complexity of A^* is equivalent to Dijkstra's algorithm [28]. In the case where A^* is implemented using a priority queue and the graph is represented as a adjacency list, the time complexity of A^* can be written as:

$$g_{A^*} = O((|V| + |E|)\log|V|) \tag{E.12}$$

By noting that both $|V|, |E| = O(N)$, this becomes:

$$g_{A^*} = N \log N \tag{E.13}$$

E.3 TIME COMPLEXITY OF THE SERIES VISITORS' ALGORITHM

In the case where the $k = N - 1$, the order of the number of robots is:

$$g_k = O(N) \tag{E.14}$$

By inspection of Equation (B.6), taking the length of the robot's path to be equal to the diameter of the grid gives the order of the length of the path:

$$g_d = O(N^{\frac{1}{2}}) \tag{E.15}$$

For the series visitors' algorithm, the cost of A^* is incurred once per robot. The cost of BFS is incurred for every node along the robot's path. The complexity of the series algorithm can therefore be written as:

$$g_{series} = g_k(g_{A^*} + g_d g_{BFS}) \tag{E.16}$$

$$= O(N)(N\log N + O(N^{\frac{1}{2}})O(N)) \tag{E.17}$$

$$= O(N^{5/2}) \tag{E.18}$$

E.4 TIME COMPLEXITY OF THE PARALLEL VISITORS' ALGORITHM

The number of turns is equal to the number of moves when $k = N - 1$ and, therefore, the scaling of the number of turns is given as:

$$g_{turns} = O(N^{3/2}) \tag{E.19}$$

For the parallel algorithm, the cost of the BFS and A^* is incurred for every robot in every turn. The asymptotic rate for the parallel algorithm therefore becomes:

$$g_{parallel} = g_k g_{turns}(g_{A^*} + g_{BFS}) \tag{E.20}$$

$$= O(N)O(N^{\frac{3}{2}})(N\log N + O(N)) \tag{E.21}$$

$$= O(N^{\frac{7}{2}}\log N) \tag{E.22}$$