1-12-2016

Improving Spatially Distributed Multiagent Pathfinding Using Cooperative JPS

Sanjay Renukamurthy
University of Windsor

IMPROVING SPATIALLY DISTRIBUTED MULTIAGENT PATHFINDING USING COOPERATIVE JPS

By

SANJAY RENUKAMURTHY

A Thesis

Submitted to the Faculty of Graduate Studies

Through the School of Computer Science

In Partial Fulfillment of the Requirements for

the Degree of Master of Science at the

University of Windsor

Windsor, Ontario, Canada

2016

Improving Spatially Distributed Multiagent Pathfinding using Cooperative JPS

by

Sanjay Renukamurthy

APPROVED BY:

Dr. Myron Hlynka, External Reader
Department of Mathematics and Statistics

Dr. Imran Ahmad, Internal Reader
School of Computer Science

Dr. Scott Goodwin, Advisor
School of Computer Science

January 5, 2016

DECLARATION OF ORIGINALITY

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

ABSTRACT

The Multiagent Pathfinding problem (MAPF) applies in fields such as video games, robotics, warehouse management, etc. MAPF is mainly concerned with routing units while avoiding collision. A recent approach by Wilt et al. to MAPF for maps with narrow corridors spatially partitions maps into High Contention (HCA) and Low Contention areas (LCA). A modified Cooperative A* is used in LCA.

In our approach we introduce a new algorithm by combining "Cooperative" and "*Jump Point Search*" (JPS) to traverse through the LCA. JPS is modified to handle the multiagent environment by incorporating a new stopping rule to identify between HCA and LCA called "*forced selection*". As JPS jumps from node to node, we introduce a "*backtracking mechanism*" to avoid collision. We evaluate our algorithm against Wilt et al.'s algorithm on real video game maps and demonstrate significate improvements in terms of makespan, solution time and failure-rate.

DEDICATION

To my loving family:

Father: Renuka murthy

Mother: Kavitha

Brother: Ajay

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Dr. Scott Goodwin, for giving me the opportunity to get an exposure of the research field. It has been a great experience working under his guidance. His constant motivation, support and faith guided me in successful completion of my thesis.

I would like to thank my internal reader Dr. Imran Ahmad for his support and showing interest towards my research. I am also thankful to my external reader Dr. Myron Hlynka for being flexiable about the schedule time of my defence.

I give my sincere thanks to Mrs. Karen Bourdeau, the graduate secretary, who always supported and helped me when ever I needed any assistance in various academic issues and providing a GA position.

Lastly, I would like to thank my family and friends for their utmost faith in me and for supporting me through the ups and downs of my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF ACRONYMS

JPS – Jump Point Search

RTS – Real Time Strategy

NPC – Non Player Control

GPS – Golbal Positioning System

MAPF – Multiagent Pathfinding

CBS – Conflict Based Search

CT – Conflict Tree

LRA* - Local Repair A*

COWHCA* - Conflict Orientented Windowed Hiearchical Cooperative A*

WHCA* - Windowed Hiearchical Cooperative A*

RWT – Reduced Wait Time

SDP – Spatially Distributed Pathfinding

HCA – High Contention Area

LCA – Low Contention Area

DM – Direction Map

DV – Direction Vector

MV – Movement Vector

OD – Operator Decomposition

ID – Independence Detection

MGS – Maximum Group Size

HCA* - Hierarchical Cooperative A*

ICTS – Increasing Cost Tree Search

SPP – Simple Pairwise Prusing

EPP – Enhanced Pairwise Prusing

REPP – Repeated Enhanced Pairwise Prusing

P&S – Push and Swap

CHAPTER 1: INTRODUCTION

Pathfinding problem can be found in many fields from video game industry to ware house management. The general problem which is addressed here is to find an optimal path for a unit from its start node to goal node on a graph representation of map. Based on application in use, the above problem statement can be used for different usage such as in field of video game they require the problem to be solved in minimum time, but in other application such GPS, the problem would be to find a safe and short to the goal. Consider a scenario in one of the Real Time Strategy (RTS) games, where the Non Player Control (NPC) has to find a path from its start to goal position real quickly to give a realistic feel to the gamer while playing. In figure 1, a NPC unit as to travel from tower to its house, so the start position is tower and goal position is house. Even though the unit could have travel through the river, but that would not be a realistic path. So the unit crosses the bridge and reaches its destination. In here we want an algorithm which could solve the pathfinding problem quickly.

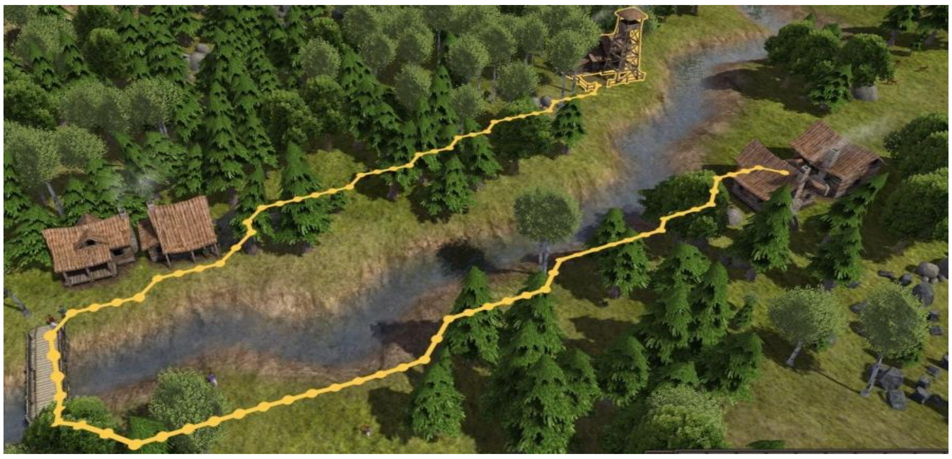


Figure 1: Single agent path on a video game [1]

Similarly we could find the same problem with different objective for solving in application like GPS. Consider the figure 2, where a vehicle has to travel from Denver to Ohio, he can either take route 1(4->5->6->1) or route2 (3-> 2->1), even though the route2 is shorter compared to route1 the driver takes a longer route1 because it’s safer then shorter route2. In this case the main criterion of finding the path from Denver to Ohio is to find a safest path rather than shortest path.

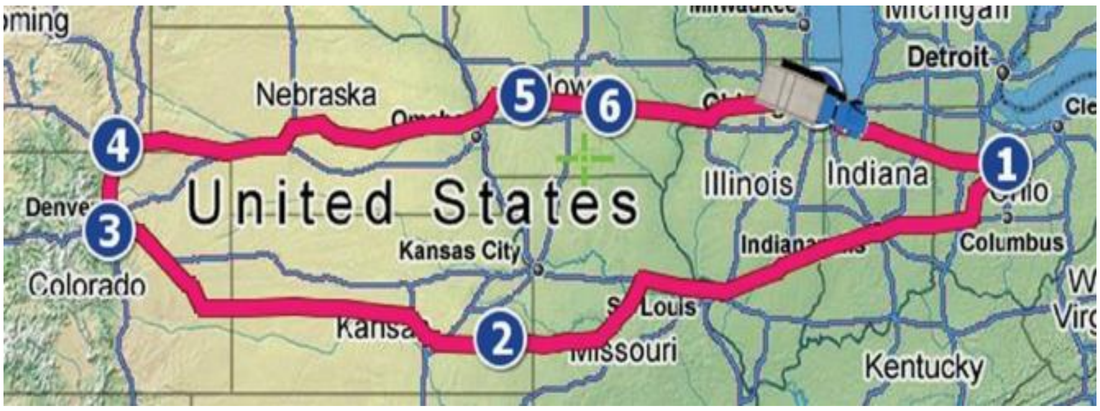


Figure 2: Two routes for Truck driver from start point ‘Denver’ to destination ‘Ohio’ [2]

The three basic elements used to solve a pathfinding problem in any of the fields are graph representation of map, heuristic to guide the unit to goal and a search algorithm to find the route for the unit.

1.1 Graph representation of map:

The maps can broadly divided into grids and hierarchical techniques that are widely used in many real world applications to find path. Map represented as grids consists of polygons grouped together to form a map. Consider a graph G(V, E) where V is set of vertices and E is set of edges. The continuous connected graph G, can be represented by placing a vertex at the centre of polygon or at each of its corners. In a map where the vertex is at the center of polygon, the edges would be the connection between polygons and in case where vertex is at the corners, the edges can be sides of the polygon. The only disadvantage of using a grid over hierarchical techniques is that it consumes a lot of memory to store the entire map.

1.1.1 Grids

Grids are connected graph with vertices and edges to represent a map. Each of the polygon on the map is called as the tile/node based on the application it is being used. There are two popular grid representation firstly regular grids where all the nodes are formed using the same kind of polygons and irregular grids where the map could be represented with different types of polygons.

1.1.1.1 Regular Grids

Regular grids are the ones where the entire maps are represented using uniformed polygons such as square, hexagon and triangle. The regular grids are one of the famous map representations which are used in many fields like video games such as Pac-man, Pokémon games, sim city etc. and robotics where the mars rover robots used regular grids for their exploration [3]. Since our work mainly concerns with square grids, we have explained it below.

Square grid is one of most popular regular grids which are used in many fields because of its simplicity in creation and finding the distance between square nodes. There are two types of movements for the units on the map. First being 4-way movement, where the units can move only horizontal and vertical directions from its node as show in figure 3(a) and 8-way movement, apart from horizontal and vertical moves, unit can move diagonally as well, as shown in figure 3(b). Most of the algorithms such Dijkstra’s [4], A* [5], Jump Point Search (JPS) [6] could be used on the square grids to find the path for units. JPS uses the symmetrical natural of square grids to reduce number of nodes expanded during the search.

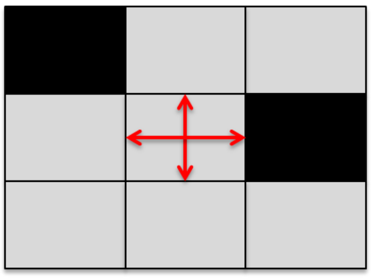


Figure 3(a) 4-way movement on square grid

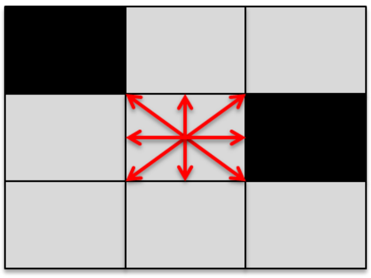


Figure 3(b) 8 way movement on square grid

1.2 Heuristic:

Most of the search algorithm used for solving Pathfinding problem uses a heuristic function to direct the unit during node exploration based on the information from the heuristic function. A perfect heuristic function is the one in which it never over estimates the distance from start to goal node are called as "*admissible*" heuristic functions and the heuristic function which either over estimates or under-estimates the distance are called as "*non-admissible*" heuristic functions. Starting from the simple diagonal heuristic function to Manhattan distance heuristic function can be used in search algorithms.

As we are using grid maps, the heuristic functions explained below are based only on square grid maps. Manhattan distance heuristic function is perfect to be used on a square grid that allows 4 directions movement. Euclidean is best suited on square grid that allows any direction movement.

1.2.1 Manhattan Distance:

Manhattan distance is considered as a standard heuristic on square grids. The minimum cost of moving from one node to its adjacent node is set as cost D and used in cost function. In simple case, the cost value D is set to 1.

```
Function heuristic (node) =
dx = abs(node.x – goal.x)
dy = abs(node.y – goal.y)
return D * ( dx + dy) [7]
```

In the above function 'dx' represents the horizontal distance between the node and goal and 'dy' represents the vertical distance between the node and goal. To obtain an admissible heuristic the value of D plays an important part; where by cost value of D must be set a minimum value. By managing the value of D we could generate an optimal path for a search algorithm. The figure 4 shows a path generated by a search algorithm on a single unit using Manhattan distance.

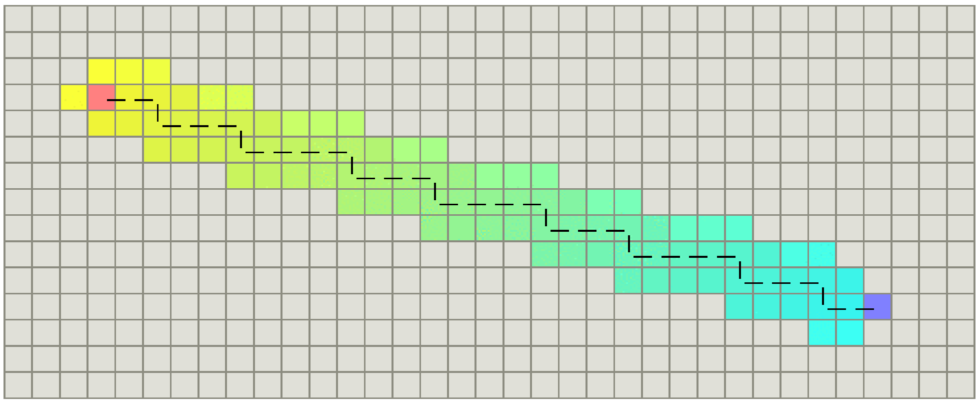


Figure 4: Search path calculated using Manhattan distance [7]

1.2.2 Euclidean Distance:

When a unit is allowed to travel in any-angle on grid map, then Euclidean distance is best suited to handle it. Euclidean distance is based on the Pythagoreans theorem of finding the hypotenuse of right angled triangle. So the Euclidean distance is the sqrt(dx*dx + dy*dy)* D. The heuristic function is given below:

```
Function heuristic (node) =
dx = abs(node.x – goal.x)
dy = abs(node.y – goal.y)
return D * sqrt( dx*dx  +  dy*dy)[7]
```

In the above heuristic function, ‘dx’ represents the horizontal distance between the node and goal and ‘dy’ represents the vertical distance between the node and goal. In the figure 5, we show the path generated from one a search algorithm using Euclidean Distance heuristic. The path obtained using Euclidean distance is much smaller compared to Manhattan distance.

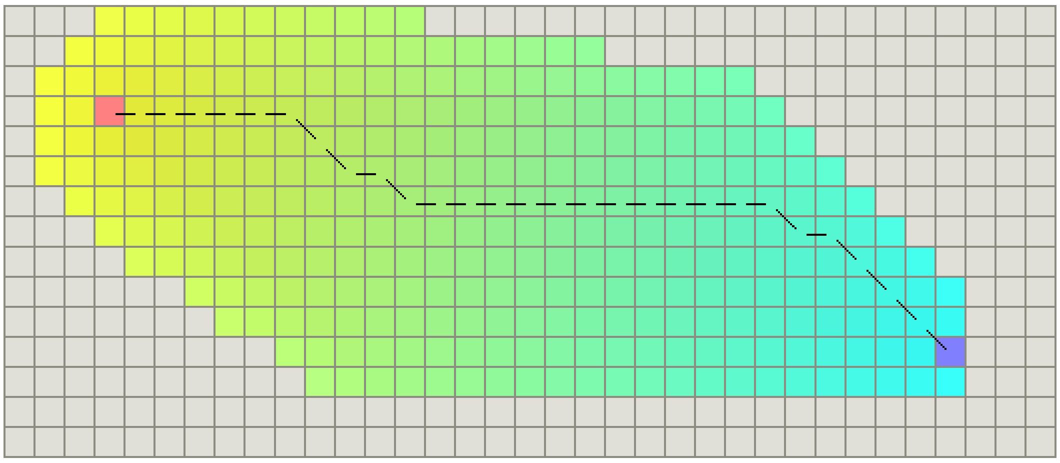


Figure 5: Search path calculated using Euclidean Distance [7]

1.3 Search Algorithms:

One of the key elements in tackling the pathfinding problem is the usage of search algorithms that helps find the optimal route from any of start position on the map to goal position while avoiding the collision of obstacles. There are many types of search algorithms that could be categorised based on number of units the search algorithm can handle. Single agent pathfinding algorithms can handle only one unit at a time while Multiagent pathfinding algorithms can handle many units at a time.

1.3.1 Single Pathfinding Algorithm:

As the name says single agent pathfinding algorithms consists of only one unit. Single agent pathfinding algorithm is used to find an optimal path for a unit from its start position to goal position on a map using an efficient heuristic while avoiding collision of obstacles. There are many types of single agent pathfinding algorithm starting from Dijkstra's to Jump Point Search. The main aim of single agent pathfinding algorithms is to find an optimal path for a unit with minimum computation time and memory overhead. Dijkstra's algorithm is one of the earlier pathfinding algorithms that find the shortest path for a unit. Dijkstra's algorithm is considered as one of the complete and optimal algorithm because it finds the path if the path exists. Dijkstra's algorithm ran on a weighted graph from start node to goal node, here the neighbor nodes from start node is search recursively until it reach the goal node. A* was an upgrade to Dijkstra's algorithm in a way that it reduced the total number of explored nodes on the graph using heuristic functions. A* is a best first type of algorithm which produces a shorter and more effective path compared to Dijkstra's algorithm. Both types of grids can be used as map for searching the path for A* algorithm. Over the years there have been many variants A* algorithm such theta* which use line of sight checks on map to find any-angle path to the destination, Weighted A* that uses the weights while selecting the neighbor node with least heuristic function to reach the destination with least computation time compared A* but finds a non-optimal path, Jump Point Search considers symmetry breaking technique to reduce the number of nodes explored by the unit and thus reducing the computation time of search, but Jump Point Search could only be implemented on square grids. There are many cost functions used to measure a single agent pathfinding algorithm such as *pathlength* that is total number of nodes between the start to goal node, *computation time* of search algorithm and number of *nodes expanded* on the map.

1.3.2 Multiagent Pathfinding Algorithm:

Multiagent Pathfinding Algorithm (MAPF) finds the routes for more than one unit. MAPF can be defined as finding the route to all the units on the provided map without collision among the units and obstacles. There are number of domains which require the MAPF such as commercial gaming industry where multiple units in the game has to find the route to their respective destination, in warehouse management, military where the robots have to find the routes to their respective destinations while avoiding collision between the robots. There are two main variants of MAPF such as centralized approach and decentralized approach. The centralized approach consists of a central controller which manages all the units on map and finds route to all units by knowledge sharing.

The Centralized approach is considered to be complete and optimal solution. A complete algorithm will find a solution if one exists and optimal algorithm will find a solution that is best. One of first algorithms under centralized approach was introduced by Svestka et al. [8] which takes a coordinated approach among the robots and the path is found using probabilistic roadmap (to find feasible path for all nodes without collision). First step in their approach creates a roadmap for single robots and stored in a data structure, following this a composite roadmap is generated for all robots and finally all the routes are retrieved from the data structure. In 2008 Ryan [9] introduced an abstraction approach of dividing the graph into subgraphs and using these subgraphs to find the paths for units in smaller level. They proved their algorithm is complete and produces an optimal path for units. Standley [10] introduced a first complete and admissible technique to solve the MAPF problem, where they proposed an "*operator decomposition*" technique to reduce the branching factor of MAPF algorithm. An "*independence detection*" technique which allows units to retain their optimal paths, thus making the entire solution to be optimal. Sharon et al. [11] introduced a Conflict Based Search (CBS) which uses a high level Conflict Tree (CT) where each node represents the conflicts generated between the units and lower single agent search. By using these two techniques the algorithms produces an optimal and complete solution. One of cost function used in centralized approach is sum-of-cost that is the summation of time steps of all the units. Finding the minimum sum-of-cost is considered to be NP-Hard problem.

Decentralized approach divides the MAPF problem into single agent searches and collision is avoided based on the previous agents search path. Stout [12] introduced a decentralized approach called "*Local Repair A**" (LRA*) where the individual paths for all agents is generated using A* and collision is avoided by rerouting the path for lower priority unit during conflicts. The rerouting process in LRA* is very expensive in terms of computational time for the solution. To avoid the above problem Silver [13] introduced a "*Cooperative A**" which uses a space-time data structure called "Reservation Table" to avoid collision between the units. There have been many other algorithms which use map abstraction [14] and map decomposition [15] for solving the MAPF problem. Bnaya et al. [16] improved the "*Windowed Hierarchical Cooperative A**" (WHCA*) [13] in terms of solution quality by effective placing the window only during the conflicts. They proposed a "*Conflict-Oriented Windowed Hierarchical Cooperative A**" (CO-WHCA*) algorithm with both online and offline approach. Even though CO-WHCA* produce better solution quality compared to WHCA* but computational time increases. Saeidianmanesh [17] introduced a "*Reduced Wait Time*" (RWT) algorithm to reduce the overall solution time of search by grouping the units with more waiting time in a narrow corridor and taking an alternate route for the grouped units. The algorithm was able decrease the solution time but gets in trouble in terms of pathlength of units. There are many cost function for decentralized approach such as *makespan* that is to find worst pathlength among units, *fuel* that is the total amount of pathlength or time taken by all the units which is similar to *sum-of-cost* but fuel does not consider the wait move of units, individual cost of units.

1.4 Problem Statement:

Multiagent pathfinding problems occur in many fields such as video games, robotics, warehouse management, military, GPS etc. A path is found for each of the unit on the map while avoiding collision between the units and obstacles. There has been a lot research which has done to address the above problem, but the scenario where the units as to travel through a narrow corridor are still not addressed efficiently. The "*Spatially distributed Muliagent Path Planning*" [18] is one of the algorithm which tried to address the problem of MAPF travelling through narrow corridor. The results obtained using SDP algorithm with "*Cooperative A**" [13] produces a better results compared to state-of-art algorithms.

We introduce a novel algorithm called "*Cooperative JPS*" which is incorporated with SDP framework to produce better results compared to the standard "*SDP framework with Cooperative A**" [18] in terms of *makespan, solution time, failure-rate*.

1.5 Motivation:

The problem of efficient traversal through a narrow corridor can be found in many fields. Considering the gaming industry where the units have to travel from its start position to its goal position, we need to find an optimal path while avoiding collision. In Real-Time Strategy (RTS) games such as StarCraft, non-player Controls (NPC) has to find a path from its base to the base of player. NPC in some maps have to travel through a narrow bridge to reach player's base. By using the traditional MAPF algorithms the units take too long which gives advantage to the human opponent. To provide a realistic feel to the game, the algorithm used to solve MAPF problem through narrow corridor must be really fast and produce effective paths. The SDP algorithm [18] try to address the problem, but the individual pathlength and individual solution time was considerably larger. This motivated us to create a new algorithm on the SDP framework which could produce better results in terms of individual pathlength and solution time which helps the game more playable for gamers. The above mentioned case is one of the examples for MAPF problem through narrow corridor, we could see the results from our algorithm can be used in other fields such as robotics, military, GPS etc.

1.6 Thesis Claim:

By incorporating Cooperative JPS in SDP framework to traverse units in LCA, we saw a significate improvement for cost functions such as makespan, solution time and failure rate when compared with SDP framework with Cooperative A*.

1.7 Thesis Outline:

At the beginning of the thesis, a problem statement was presented that speaks about the standard pathfinding problem and its application in various fields. In chapter 1, an introduction to different elements involved in solving pathfinding problem was proposed. In chapter 2 a brief literature review on single agent and multiagent pathfinding algorithms is introduced. Chapter 3 contains a detailed description of "*Spatially Distributed Multiagent path planning*" [18] algorithm in detail. Chapter 4 presents the proposed approach about the Cooperative JPS and its impact on different layers in SDP framework. Chapter 5 discuss the experiments that we carried out on a benchmark maps. In this chapter, we discuss the experiments setup along with results obtained with the comparison of our approach and existing SDP framework. Results and discussions on how the performance of our approach with SDP framework was improved is shown in chapter 6, followed by concluding remarks on entire research and some of future work which could be done on our approach along with SDP framework on a whole.

CHAPTER-2: LITERATURE REVIEW

2. Literature Review:

This section tries to give insight into some of important works which has been done to address the pathfinding problem. The pathfinding problem can be broadly classified into single agent pathfinding and multi agent pathfinding. Research into pathfinding initially started by solving for single agent, and then researchers started looking into multi agent pathfinding as it is slightly complicated compared to single agent pathfinding as it is NP hard problem. [19]

2.1 Single Agent Pathfinding Algorithms:

Single agent Pathfinding problem is to find the route for a single unit from its start position to its goal position avoiding collision with the obstacles on a map such as grids (triangular, square, hexagonal, octagonal), waypoints and navigational mesh. There have been many single agent pathfinding algorithms over time, starting from dijkstra's algorithm to jump point search. We would concentrate only on the algorithms which are relevant to our work.

2.1.1 A*:

A* algorithm is one of the efficient single agent pathfinding algorithm introduced by Hart et al. [5]. It is a graph based search algorithm which tackles the above mention problem of avoiding obstacles and finding optimal path for the unit. A* can be considered as a combination of best first search and Dijkstra's algorithms as it explores the adjacent nodes similar to dijkstra's but only considers the shortest among the nodes to goal using a heuristic estimator as best first search. Heuristic function are used to find the distance between two nodes on a weighted map (i.e. pre-set weight between two adjacent nodes, usually for horizontal and vertical nodes its set as 1 and for diagonal nodes its set to 1.4). A* uses one heuristic function such as

Manhattan distance, Euclidean distance in terms of h(n) where ‘n’ is the current node and h(n) is the distance from ‘n’ to the goal position. The evaluation function used by A* is show below:

$$f(n) = g(n) + h(n)$$

Where g (n) = Distance from start node of unit to the current node ‘n’

h(n) = Heuristic distance from current node ‘n’ to goal node of unit

f(n) = Overall distance from start node to goal node travelling via current node ‘n’

Using the above evaluation function A* selects the smallest f(n) among all the discovered neighbouring nodes

Cases of A* Algorithm based on Heuristic Function:

1. If h(n) = 0, then A* = Dijkstra’s algorithm.
2. If h(n) < g(n) then A* is guaranteed to find the shortest path.
3. If h(n) = g(n) then A* will follow only the best path never expanding anything making it very fast which very rare. This is the perfect scenario.
4. If sometimes h(n) > g(n) then A* is not guaranteed to find the shortest path.
5. If only h(n) plays a role in finding the best path A* turns to greedy best first search.

Pseudo code for A* algorithm:

1. Create a Graph G formed using the start node N0.
2. Push the start node N0 into the OPEN list. CLOSED list shall be empty at this point, f(n) = 0 + h’(n).

3. If N is the destination node the goal has been reached and the path is obtained by tracing the pointers from N0 to N.
4. If not, Expand N, and generate a set S of its successors that are not already ancestors of N and add them to the OPEN list.
5. Place the above set of successor S to N on the open list and attach a pointer to N from each successor node now in the OPEN list.
6. For each member of the successor node set S either on the OPEN or CLOSED list, redirect its pointer through N if that is the best path to the successor s. For each member of the set S on the CLOSED list, redirect the pointers of each of its descendants in graph G so they backwards along the best paths found so far to these descendants.
7. Reorder the OPEN list in order of increasing f values.
8. Go to step 3.

   Closed List - Nodes already explored in the graph G.

   OPEN List - Nodes to be explored in the graph G.

To explain the A* algorithm with an example, consider a square grid with only 4-way selection of grid and cost to travel to adjacent node is 1. As from figure 3, the start node is ‘A’ (At node 1) and goal node is ‘B’ (At node 11). The black blocks on the grid are non-traversal node or obstacles. When the algorithm starts, we add the start node to open list, before removing the node for evaluation we add the start node closed list as it is already explored. We find all neighbouring nodes to 1 i.e. 4 and 2 are selection and by using the evaluation function we calculate the h(n) and g(n) values for both node. So the f(n) values for node 4 and node 2 comes up to 4. Now we add them back to open list to and select the neighbour node with least f(n) value, since both node 4 and node 2 have same f(n) we will select the node which is on top in open list for evaluation. We repeat the above process until we reach the goal node 'B' (i.e. 11)

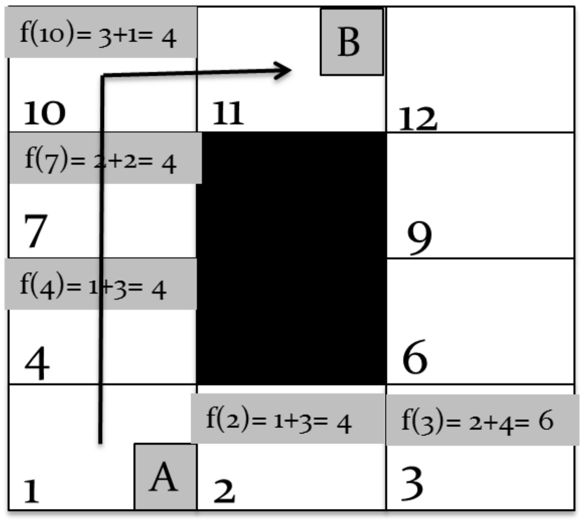


Figure 6.Working diagram of A algorithm*

By selecting a perfect heuristic function, A* algorithm can produce an optimal shortest distance from start to goal node and the algorithm is complete as it can find the path if it exists on the map.

2.1.2 Jump Point Search:

Jump Point Search (JPS) is one latest algorithm introduced by Harbour et al. [6] which addresses the problem of single agent pathfinding problem. JPS is an online symmetry breaking algorithm which eliminates most of the repetitive paths from start node to goal node. They consider a concept that moving from start node at bottom right corner of a 3x3 grid to goal node at top left corner, we could either move left-left-left-up-up-up or up-up-up-left-left-left. Since both paths are have same distance we can consider only one of them and by doing reduce the time of exploring. (In the above example we have an obstacle at centre of grid). As JPS works only with repetitive paths we can only use the algorithm on a grid maps.

The main focus of JPS is to identify the jump point nodes by recursively pruning from selective neighbor node from current node. Similar to A* algorithm, the same evaluation function is used to move the unit towards its goal node.

$$f(n) = g(n) + h(n)$$

Rather than adding all the adjacent neighbors from current node to open list, JPS will selectively pick the neighbors based on two lengths. Considering the figure 7(a), current node which is being is expanded is 'x', its neighbors(x) = 1,2,3,6,7,8,9 and the parent of 'x' is p(x) = 4. We select the neighbors based two lengths. First the length from parent node p(x) to a neighbor node 'n' going via 'x' and the second length from parent node p(x) to neighbor node 'n' not going through 'x'. From the figure 7(a), the node 5 is not pruned because the length from p(x) to 5 via x is (len (4, x, 5) = 2) less than length from 4 to 6 not going through 5 (len (4, ! x, 5) = 2.8). Figure 7(b) shows the example of diagonal pruning rule.

The condition to not select a neighbor for straight move and diagonal move is give below:

Straight move -> len (p(x), !x, n) <= len (p(x), x, n)

Diagonal move -> len (p(x), !x, n) < len (p(x), x, n)

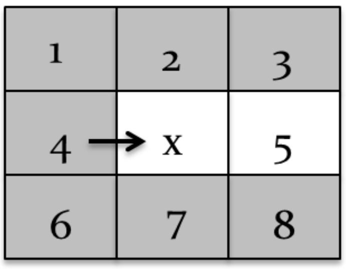


Figure 7(a): Straight pruning rule

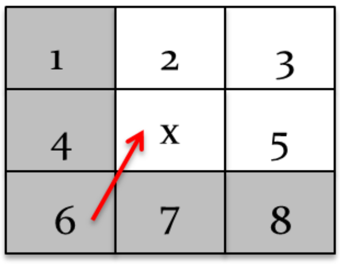


Figure 7(b): Diagonal pruning rule

Once we identify the neighbors we apply two pruning rules recursively to all the neighbors based on their position to current node:

Straight pruning rule for Forced neighbor:

When there is a straight neighbor of current node ‘x’, we continuous move in that direction until we encounter the goal node, an obstacle or a forced neighbor for the expanded node.

Consider the figures 8, if node ‘y’ is the goal node we stop the search, if ‘y’ is an obstacle we pass a null value which says that the route from the neighbor useless and when there is a forced neighbor ‘z’ we stop the pruning and pass back ‘y’ as jump point to ‘x’. The forced neighbor is identified based on two conditions; first it should not be a natural neighbor of ‘y’ and second the length from ‘x’ to ‘z’ via ‘y’ must be less then length of ‘x’ to ‘z’ not going via ‘y’ i.e. len(x, y, z) < len(x, !y, z)

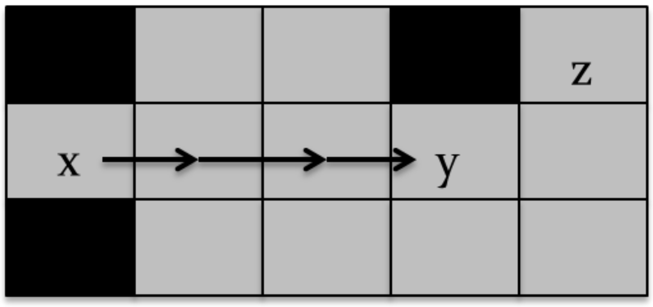


Figure 8: Forced neighbor for straight move

Diagonal pruning rule:

When there is a diagonal neighbor of current node ‘x’, we continuous move in diagonal direction until we encounter the goal node, an obstacle or a forced neighbor for the expanded node. Figure 9 shows an example of Forced neighbor for diagonal move.

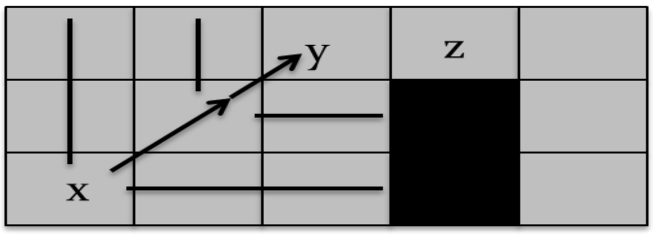


Figure 9: Forced Neighbor for diagonal move

Once we have the jump point nodes for current node ‘x’ we add them to open list and select the node which as least f(n) value and repeat the process of pruning until we get the goal node.

JPS is proved to extremely fast in terms solution time compared to A* as it expands fewer nodes and results show that JPS is 10 times faster than A* algorithm. The JPS is also proved to be optimal as it produces the shortest path for a unit and it requires very less memory.

2.2 Multiagent Pathfinding Algorithms:

Multiagent Pathfinding problem (MAPF) deals with finding the routes to all the units from their respective start node to their respective goal node while avoiding collision between the units and obstacles on a map. Over the years there have been many algorithms addressing MAPF problem using two standard approaches. Centralized approach consists of a centralized controller monitoring all the units to reach their respective goal nodes while avoiding collision. Decentralized approach sub divides the problem into single agent runs to reach goal node while avoiding collision by communicating between the units.

2.2.1 Decentralized Approach:

One of the earliest decentralized approaches was introduced by Stout [12], where the A* algorithm was ran on all the units individually considering only of its current neighbor unit. Once all the route for units is generate, the unit tracks back route to check for collision. If there exists a collision with other unit, the current unit just reroutes the unit from the node previous to collision node by running A* algorithm. The same process is repeated for all the units Since A* algorithm is run for every collision there is massive impact on CPU usage.

To avoid the problem of running A* for every collision Silver [13] introduced a new algorithm called "*Cooperative A**". A* algorithm is ran on individual units on a three dimensional space-time, and considering the routes of other units. The individual routes of units are stored in a data structure "*reservation table*" which contains both node on the path and time on which it is being occupied. So while searching the route for other units, the nodes on the reservation table will become untraversable at that particular time. When there is a collision between the units, the unit currently being searched uses a "wait" move. Until the node required by the unit will not be available, the unit waits at the previous node before collision. Consider the figure 10(a), with two units 'A' and 'B' on square grid map with 4 way travel. The start node and goal node of 'A' is S1(0, 0) and G1(3, 3) respectively. And for unit 'B', the start node is S2(0, 1) and goal node is G2(3, 1). When cooperative A* is used on above map, A* algorithm is ran on unit 'A' to generate the path, the same path is stored in a reservation table along with its time. While running A* on unit 'B' all the expanded nodes from its start node till it reaches the goal node is cross checked against the reservation table. Since there is no collision the two reach their in optimal time and path. Consider a similar example as above in figure 10(b), where unit 'A' as same start and goal node, but unit 'B' as start node S2(0, 2) and goal node G2(2, 0). Initially the path for unit 'A' is stored in reservation table along with its time, while finding the path for unit 'B' the one expanded node (1,1) at time 2 is occupied by unit

‘A’ , so the unit ‘B’ waits at node before collision at (0,1) and moves to node (1,1) when it becomes available.

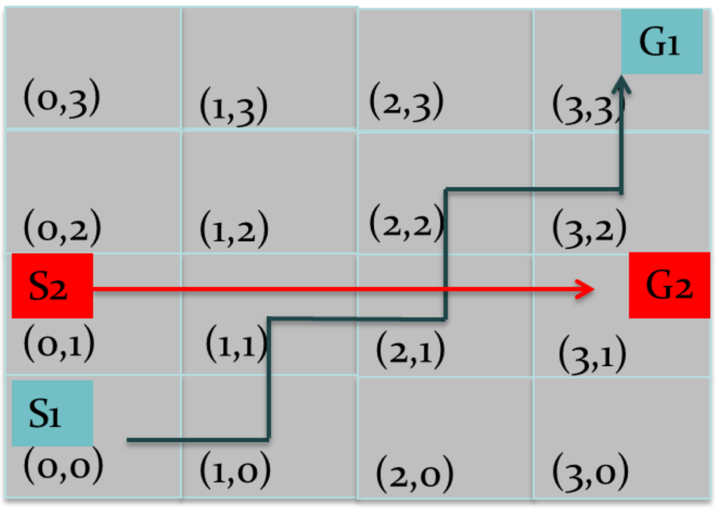


Figure 10(a): Cooperative A without collision*

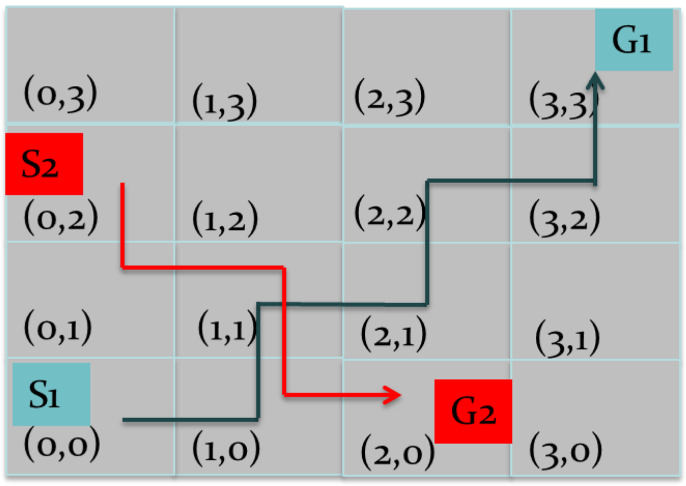


Figure 10(b): Cooperative A with collision*

Cooperative A* has some drawbacks in terms of termination of units i.e. the units would become inactive after reaching the goal and block other units, there is no prioritization, on when each unit is ran which may impact the units which has longer path and efficiency of algorithm as the entire path of unit is calculated on a three dimensional space time state.

The above problem in cooperative A* were addressed by Silver [13] with a "*Windowed Hierarchical Cooperative A**". Here a window of predefined depth is used while finding the path for individual units i.e. unit's search is partially stopped when the window limit is reached, thus allowing the units to be prioritized based on duration of usage of A* algorithm. By partitioning the search for a unit efficiency of entire algorithm is increased in terms of time. And finally by using the window, the units which reach their goal node are still active as long as the window limit is reached.

Jansen et al. [14] introduce an implicit cooperative pathfinding using direction maps (DM) which are built on a map. An abstract map is used to run all the units individually to capture the path and direction of travel which is later used in DM. Direction maps are data structure with collection of all the direction vectors (DV). Direction vectors are vectors which give direction to each unit within each traversal of node, its value ranges from zero to one. The author's also use movement vectors (MV) to capture the individual movement between the nodes on the map which could be any of the 8 directions. Planning of DM is done just the opposite to A* algorithm where f-cost is used to expand the nodes, while in DM the cost to travel to adjacent node is changed on both nodes. The main objective of DM is used to find the path with lower cost when compared shorter path. Thus while traversing on a DM the units with same direction are grouped together to move to their respective goal nodes The author's propose the following formulation using the dot product of DV and MV which ranges from -1 to 1 for movement from node 'a' to node 'b'

$$Wab + 0.25 \cdot Wmax\ (2 - \mathrm{DV\ a} \cdot \mathrm{MV\ ab} - \mathrm{DV\ b} \cdot \mathrm{MV\ ab})$$

And `*Wmax*' is the penalty for units which move in opposite direction to DV , `*Wab*' is the cost of moving the unit from edge `*a*' to `*b*', `MV ab' is the movement vector for moving from node `*a*' to `*b*', `*DV a*' is direction vector associated with node `*a*' and `*DV b*' is direction vector associated with edge `*b*'. Once the DM is built a single unit, the next unit can use the previous DM to travel on the map. The author's state that the dynamic direction maps can be used for learning process has the direction map will be constantly updated with movements of all units.

One of the most recent works addressing the MAPF problem is proposed by Bnaya et al. [16], where they introduce a upgrade to Silver [13] "*Windowed Hierarchical Cooperative A**"*(*WHCA*) called "*Conflict oriented Windowed Hierarchical Cooperative A**" (CO-WHCA*). WHCA* does not consider conflicts between units in some cases where the window is used for each unit and space-time node is reserved for the entire path in the reservation table till the window limit. The space-time node reserved in reservation table may not have any conflicts with the other units. Secondly in case where the conflict may occur at Window + 1 node for unit, by then the unit may be physical impossible to avoid the collision. CO-WHCA* address the above drawbacks by placing the window only when the conflict occurs. Window is placed only during the conflicts as in case of WHCA* a predefined length of window size is utilized. One of the conflicting units is selected as a conflict owner and that unit is allowed to use the window and reserve space-time node in reservation. Following the initial cycle, after the first reservation table is not erased as in case of WHCA* and the previous reservation table is utilized while finding routes for next units. Thus by managing the window only during the conflicts the CO-WHCA* produces better solution in terms of success-rate, solution cost and time compared to WHCA*.

The latest algorithm for solution MAPF problem is introduced by Saeidianmanesh [17] called "Reduced Wait Time" (RWT). Here all the units on a map have same direction of travel i.e. on a square grid map of 5x5; all the units have start nodes on left-hand side and their goal nodes on right-hand side. The main focus of the RWT algorithm is used to reduce the waiting time of units in a narrow passage where only few units can pass and rest of units as to wait for their turn to pass. To reduce the overall time of all units, RWT propose to divide the group of units when there is a shared passage (i.e. two ways to reach the destination). So the some group of units take an alternate route then the optimal route to reach their destination. By doing so the overall solution time is decreased but takes non optimal route to destination. Consider the figure 11, where there is shared passage 'A' and 'B', if there are 20 units entering the large corridor, the RWT algorithm will send 10 units through passage 'A' and rest of units through passage 'B' thus reducing overall solution time.

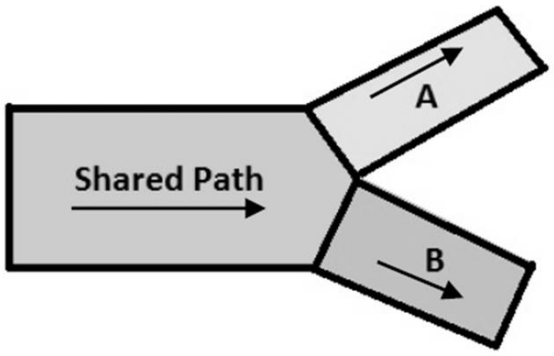


Figure 11: Passage with path 'A' and 'B'[17]

2.2.2 Centralized Approach:

Most of the algorithms using the centralized approach produce an optimal solution to MAPF problem but fail as the number of units increase on the map. One of predominate algorithm was introduced by Standley et al. [10], where they introduce an "*operator decomposition*" (OD) technique to reduce the branching factor of a standard A* algorithm which is ran on multi agent environment. By reducing the number of operation for each od nodes on every timestep the OD reduces number of nodes which would selected during a standard A* run. Even though OD greatly reduces the number of explored node space, the algorithm will be still exponential. To tackle this, they introduce independence detection (ID) which utilizes the idea of decentralized algorithm, by running the units independent to other units. After this process, they group the units with conflicts and units without. Thus concentrating only on the conflicting units algorithm reduces overall the solution time. There algorithm produces an admissible, optimal and complete solution to MAPF problem. Standley et al. [20] further improved the solution quality by introducing a Maximum Group Size (MGS) algorithm, which is used to set a maximum size for groups which are created during ID process, thus allowing conflicting groups to not combine by find a alterative path. Neither OD+ID nor MGS algorithms could produces optimal solution for a MAPF, so they introduced an Optimal Anytime algorithm by using MSG algorithm, where in after finding the solution, the algorithm continues to run until it finds an optimal solution or until the algorithm terminates. They compared their algorithm against Hierarchical Cooperative A* (HCA*) to see their algorithm outperform HCA* in terms of solution quality.

Sharon et al [11] introduce the pruning technique to their previous work called "Increasing Cost tree Search" (ICTS), where they used increasing cost tree (ICT). By partitioning the ICT into High level tree which stores all the independent paths of units and low level tree where they compare the unit's path with high level tree to avoid collision and to find optimal solution. There were 4 pruning technique which was used to improve the solution quality of ICTS algorithm. First was the "Simple Pairwise Pruning" (SPP), considering a pair of agents ai and aj in the list of agents from the multi agent problem. Once selected ignoring the rest of the agents the route for ai and aj is found with the cost for reaching their respective destination assumed as

Ci and Cj. So if there is no immediate solution to the above problem of two agent search space of MDDij, then the corresponding ICT(n) node can be considered is not a goal. Second “Enhanced Pairwise Pruning” (EPP), by modifying the SPP, the pairwise pruning can be improved to perform in worst case also. By modifying the searching strategy of SPP from depth first search to breadth first search and also by modifying the single agent MDD's, the EPP removes the invalid nodes from all the individual MDD. So the k-agent search (higher level search) is improved. Third “Repeated Enhanced Pairwise Pruning” (REPP), the EPP is continuously iterated to check until there is no solution for a pair of agent's a_i and a_j or until there is no single agent MDD to further iterate in the ICT. Fourth “m-agent Pruning”, where a group of agents ranging from $2 < m < k$ can be pruning using the m-agent pruning which search the m-agent MDD search space using depth first search strategy. So if there is no solution for the above m-agent pruning. By implementing the pruning techniques the normal ICST was completed out performed by the ICST with pruning technique.

One most recent works on MAPF problem using centralized approach was done by Mors et al. [21], where they improved the “push and swap” (P&S) [22] algorithm. The “push” process is used to move the units towards their respective goals and “swap” process allows swapping the units without altering the configuration of units. P&S as some shortcoming on some of instance, during the “swap” process, the 2 swapping units must a node with degree greater than 3 to make perfect swap, otherwise the one of unit gets stuck or in other instance as to longer route to its destination. To address the above problems and to produce a complete and optimal solution, they introduced a “push and rotate” algorithm where in the algorithm is divided into three phases. In first phase all the disjoint parts are found in the graph and named sub problems by taking into account of number of unoccupied vertices and on whether a unit from one location can be moved to another location in the graph. In the second phase, each unit is assigned a sub problem depending on the number of empty vertices surrounding the sub problem and on whether the units can be moved out of the sub problem easily. Finally the third phase is to prioritize the units placed in different sub problems so that the solution can be easily found for the units while present in bottle neck. Thus solving the entire instances which were not addressed by P&S and producing a complete algorithm.

2.3 Summary

In this chapter, we have examined some of the relevant papers to our work. We introduced both the single agent pathfinding and multiagent pathfinding algorithms. We also examined the two approaches which are employed by various researchers to address the MAPF problem called centralized approach and decentralized approach. In the next chapter we are going to explain in detail the most relevant algorithm Spatially distributed Multiagent Path planning [18] algorithm which is used to address MAPF problem while managing the transversal of units through narrow corridors.

CHAPTER-3: SPATIALLY DISTRIBUTED MULTIAGENT PATH PLANNING

To address the issue of traversing of units through the narrow corridors in MAPF problem, Wilt et al [18] introduced a "*Spatially Distributed Multiagent Path Planning*" algorithm. The given map is partitioned into Low Contention Area (LCA) such as open fields in video games, rooms for cleaning robots etc and High Contention Area (HCA) such as narrow hallway for warehouse robots, bridges in video games etc. Each of the areas consist of controllers that are responsible for their respective areas and have knowledge of their area in terms of number of units, obstacles etc. There is a 1-to-1 mapping between the controllers and areas on the map.

3.1 Spatial Distribution of Map:

Spatial distribution is dividing the map graph G (V, E) with controllers C_1 to C_k where each controller consists of subset of V. The edges connecting within a controller are called as internal edges and edges connecting between the controllers are referred to as transition edges. Each controller has the knowledge of its respective area which is the topology of area and configuration of units within it. There are two movements of units within a controller that are internal moves and transition moves. Internal moves are classified into 3 types: first to transfer the unit to its goal with target macro, second to transfer a unit from current controller to next controller and third to accept the unit from other controllers. Transition moves are the single step movement of units through transition edge from one controller to next. The figure 12 shows the two controller1 and controller2. The double ended arrow between the controllers represents the transition move of unit showing units can travel in both directions. One of the cases of internal move which is to transfer a unit to its goal 'G' is also shown.

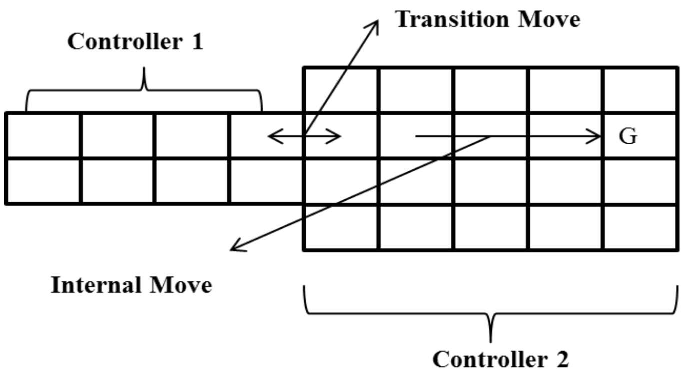


Figure 12: Spatial distribution of map into controllers and transition move between controllers and internal move

Since each controller has the configuration knowledge of units with it, there is a need for each controller to know all the units which are arriving and departing from it. A heuristic guidance is generated by running an individual search for all unit using A*. This high level path would help the controller to transfer the unit to appropriate controller and to accept a unit from an appropriate controller. There could be cases where the units start and goal nodes are within a single controller then the high level path would consists single controller. In figure 13 we show the high level paths for two units U1 and U2 with S1, G1 and S2, G2 as start node and goal node respectively. There are three controllers with name controller 1(C1), controller 2 (C2) and controller 3(C3). U1 with high level path C1->C2->C3 and U2 with C3 as both its start and goal nodes are within C3.

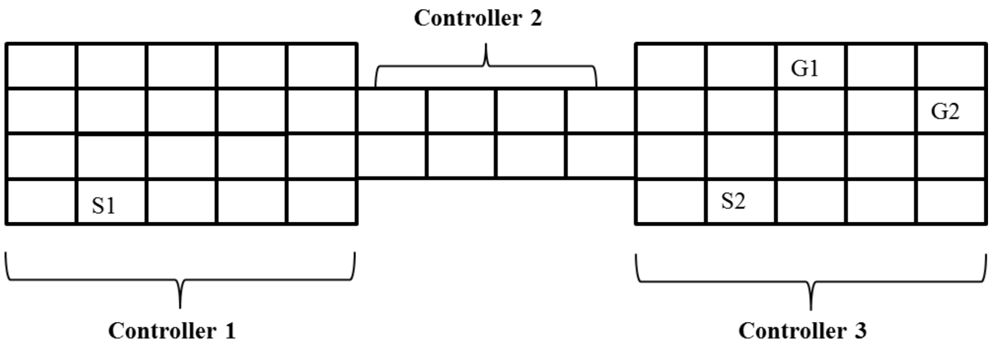


Figure 13: High level paths for units U1 and U2

3.2 High Contention Area:

All the narrow corridors, bridges, narrow hallways on a given map will be HCA's. Central core area and buffering area which allows units to wait together forms a HCA. For a unit to travel through the HCA first it has to arrive at one of the nodes in buffering area then it is moved to central core area and through the opposite buffering area. To avoid collision between the units within the HCA, it is divided into inbound and outbound areas. Based on the direction of travel a unit can take either the inbound or outbound area.

The central core area of HCA is identified on a given map. A pattern of 2, 3 and 4 nodes wide and 8 nodes long both vertical and horizontal is moved over the map to find all the central core areas for each of the HCA. All three horizontal patterns with 4 nodes long showing both inbound and outbound direction are shown in figure 14(a), 14(b) and 14(c).

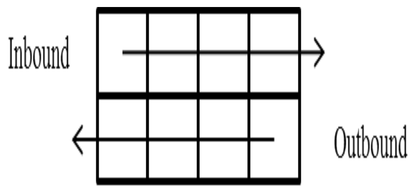


Figure 14(a): 2 wide central core HCA

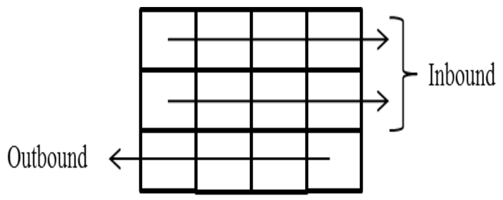


Figure 14(b): 3 wide central core HCA

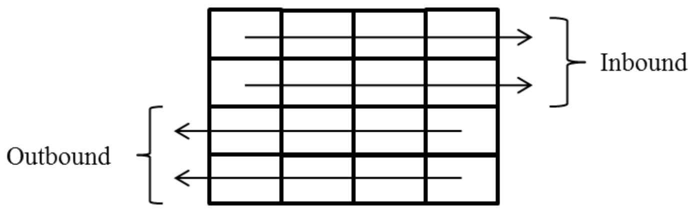


Figure 14(c): 4 wide central core HCA

Once the central core area for each HCA is identified, the buffering area must be created around the central core area for each HCA. The size of 13 nodes is used for buffering area. Since all units cannot travel through central core there must be a buffering/waiting area surrounding the central area which allows the units wait for their turn to move. The first step in this process is to divide the central core area of HCA, for hortizonal cental core the left to right will be inbound and opposite will be outbound. Following this the first N=13 cells on same side of the direction being considered will be selected starting from central core area of HCA using Breadth first search (BFS). Now the HCA is created on the map. Figure 15 shows a HCA with 2 size wide and 4 long vertical central core area represented with 'C' cells. Using BFS on both inbound and outbound direction the buffering area is created around the central core area which are represeneted with grey cells.

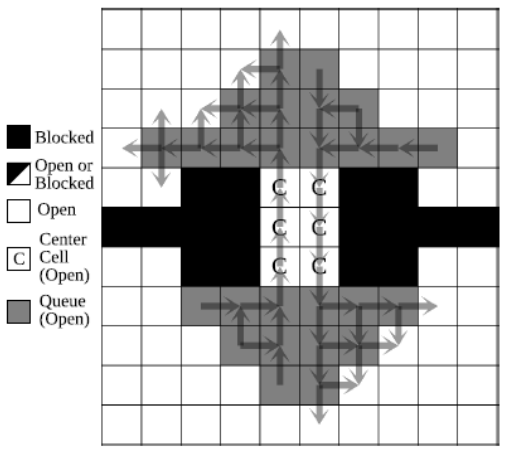


Figure 15: High Contention Area [18]

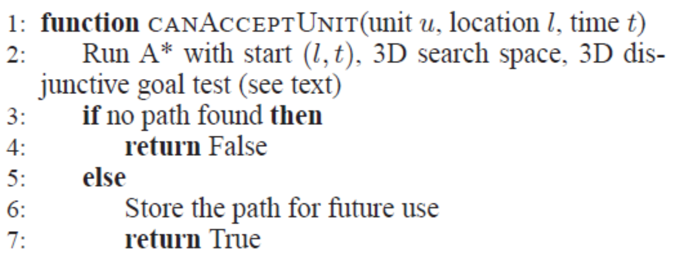
Each cell in the HCA is given a BFS depth value starting from the first buffering cell for each direction through the central core area and to other side of central core area of HCA. In the figure 15, for inbound direction i.e. from down to up the BFS value for first buffering cell will be 0 and for outbound direction i.e. from up to down the lowest BFS depth will be at the top buffereing cell. By using BFS depth of cells in HCA, the units are traversed through the HCA. There are some simple rules to avoid deadlock and unit starvation such as the prority is always given to unit with longer time in the controller by moving other units to a empty cell with lower BFS depth value, there is no dead on collision of units as the BFS depth value in each of the direction is different value. Units that are on its way out of HCA can either be transferred to adjacent cell of next controller (only if the cell is avilable) or taken to end of HCA and then transfer to next controller depending on availability.

3.3 Low Contention Area:

Once the HCA's are found on map, the rest of area's can be considered as LCA's. The main responsibility of controller of LCA is to transfer the unit to next HCA controller or to send the unit to its local goal node. Since LCA's are open area's with few obstcales, a modified Cooperative A* [13] is ran for all the units. As the map is spatially distributed, the standard Cooperative A* [13] cannot be used because of the following problems. In standard Cooperative A* [13] there is a preset goal node for each unit and the unit can arrive at its goal at any time, but in spatially distributed map, the arrival time of unit to its destination cannot be gurantee. So they have considered the destination of a unit as a disjuctive destinations that are locations adjacent to existing controller to its next controller. So there could be series of nodes along each of controller before reaching its actual destination. The other part to consider is the time, as the availability of nodes to transfer a unit would not be available at a particular time on a controller. So the disjuctive destination to be considered as goal state of unit the controller must accept the unit at a arrival time. Once the units reach their respective goal node, in standard Cooperative A* [13] they become inactive i.e. they just sit at their goal nodes. Thus making the node not available to units on its optimal path. In modified Cooperative A* even after reaching the goal node the unit will be active by replanning a route to goal node, thus allowing other units to use its goal node.

3.4 Spatially Distributed Algorithm:

Spatially distributed framework is used to partition the given into High Contention Area (HCA) and Low Contention Area (LCA). Each area shows a 1-to-1 mapping between the controllers. As stated in the earlier discussiones each controller communicate with other controller to negotiate the transfer of unit. So there are two macros to handle the communication in both HCA and LCA controllers with respect to unit to be transferred, node where unit is accepted and time of arrival of unit. The pseudo code for both LCA and HCA for accepting a unit is given below. The HCA accepts the units at particular node and time only if it can keep the previous promise made to other units. If two units request the same node at same time, then the priority is given to the unit which asked first and other unit as time wait the transition node for its turn. As LCA controller's main responsibilities is to accept unit from HCA and transfer either to local goal node or to next HCA controller, in pseudocode just modified Cooperative A* is ran for the particular unit. The location parameter can either represent a local goal node or the disjunctive destination for next HCA controller.



```
function canAcceptUnit(unit $u$, location $l$, time $t$)
    Run A* with start $(l, t)$, 3D search space, 3D dis-
    junctive goal test (see text)
    if no path found then
        return False
    else
        Store the path for future use
        return True
```

Algorithm 1: Low Contention Area Accepting Unit [18]

```
function canAcceptUnit(unit $u$, location $l$, time $t$)
    $t_c \leftarrow$ current time
    $\kappa_p \leftarrow$ previous acceptance commitments    ▷ Triples $(u', l', t')$, with $t_c \leq t'$
    $\kappa \leftarrow \kappa_p \cup \{(u, l, t)\}$
    $\sigma \leftarrow$ current unit configuration in this HCA
    $t_s \leftarrow t_c$    ▷ Initialize simulation time
    while $\kappa \neq \emptyset$ do
        for all $(u', l', t') \in \kappa$ with $t' \leq t_s$ do
            if $l'$ is occupied in $\sigma$ then
                return False
            Remove $(u', l', t')$ from $\kappa$
            Add unit $u'$ at location $l'$ in $\sigma$
        Update $\sigma$    ▷ Relaxed simulation of internal routing one time step ahead (see text)
        $t_s \leftarrow t_s + 1$
    return True
```

Algorithm 2: High Contention Area Accepting Unit [18]

3.5 Summary:

In this chapter we have examined the working of "*Spatially Distributed Multiagent Path Planning*" (SDP) which address the problem of traversing the units through the narrow corridor called High Contention Area's in a MAPF problem. To do the search for units in Low Contention Area's (LCA) the authors have proposed a modified Cooperative A* which is one of earliest algorithm to solve MAPF problem. In next chapter we propose a novel algorithm called Cooperative Jump Point Search (Cooperative JPS) which is ran on LCA to find path for units. We see the impact of Cooperative JPS on the entire SDP framework and modification made to both Cooperative and Jump Point Search algorithms to adapt to SDP framework.

CHAPTER 4: PROPOSED APPROACH

The problem of finding the paths for all the units on the map from their respective start and goal positions is referred to as MAPF problem. Many researchers have addressed the MAPF problem over past few years. Wilt et al [18] tackled the MAPF problem with a scenario where the units have to travel through a narrow corridor on the map. They partitioned the map into High Contention Area (HCA) and Low Contention Area's (LCA). HCA are the narrow corridors, bridges, narrow hallway etc, and LCA are the open areas on the map. Each area has its own controller having the local knowledge of its area and units inside it. A modified Cooperative A* [18] was used to traverse the units through their paths in LCA and a Breadth First Search was used in HCA. Since Cooperative A* [13] is one of the oldest MAPF algorithm, we have proposed a novel algorithm called Cooperative JPS which is a combination of Jump Point Search (JPS) [6] and the cooperative nature from cooperative A* [13] algorithm. We have introduced our new algorithm in place of Cooperative A* on the Spatially Distributed Pathfinding framework (SDP) [18] to find the paths of units in LCA. In rest of chapter, we introduce all the techniques which was employed to combine the Cooperative and JPS. We also see the adjustments done to Cooperative JPS to accommodate inside SDP framework.

4.1 Cooperative JPS:

Cooperative JPS is the combination of JPS [6] and Cooperative nature seen in Cooperative A*. As we have explained in literature review on the working of JPS [6] and Cooperative A* [13], the JPS [6] uses the symmetry breaking techniques to reduce the number of nodes explored by a unit in a single agent environment. Instead of probing all the nodes surrounding the unit, JPS uses the couple of pruning rules and stopping rules to find the next available node on the path for a unit called "*jump* *points*”. Thus by using the jump points the JPS finds the path for a unit from its start node to goal node. Cooperative A* [13] uses a 3D space and time data structure called “*Reservation table*” to store the paths of units in multiagent environment. The collision is avoided by constant lookup into reservation table for each unit. A wait macro is used during the collision. The unit which has path already is given priority to occupy the node and the other colliding unit would have to wait until the node becomes available. Since the reservation table requires the entire path of units, we have introduced a “*Backtracking mechanism*” to get all the nodes in the path for each unit, as JPS [6] will only give the jump points from start to goal nodes. While finding a path for a unit using JPS [6], the algorithm need to differentiate between High Contention Area and Low Contention Area, so we have proposed a new stopping rule for JPS [6] algorithm called “*Forced Selection*”. In some cases the solution time becomes more important than finding an optimal path for a unit, so by just using wait macro we cannot achieve that. Thus we present a new “*Side-way Movement*” macro along with wait macro to improve the solution time and effective avoid collision between units. To decrease the overall solution time of the entire SDP algorithm with Cooperative JPS, we use JPS [6] instead of A* algorithm to find the high level path which acts as a heuristic guide for controllers. We explain all the above mentioned techniques in details later in the chapter.

4.1.1 Backtracking Mechanism:

The *backtracking mechanism* is very important part when combining JPS [6] with cooperation in multiagent environment. As the reservation table requires the entire path for each unit to be stored in the table, we propose a *backtracking mechanism* to find the entire path generated using the JPS [6] algorithm. As mentioned earlier in the chapter, the JPS [6] will only generate the jump points from start node to goal node as a path. Thus we need to use a *backtracking mechanism* to get all nodes in the path between the jump points. While storing the expanded nodes from the JPS [6] algorithm to the reservation table, we consider the current node from where we do the search and each jump node expanded by JPS [6] to find all the nodes between the current node to each expand node using *backtracking mechanism*. We need to pass the direction of travel node using below equation in both ‘x’ and ‘y’ direction.

DirectionX = (jumpNode.x – currentNode.x)/max(abs(jumpNode.x – currentNode.x), 1)

DirectionY = (jumpNode.y – currentNode.y)/max(abs(jumpNode.y – currentNode.y), 1)

The pseudo code for *backtracking mechanism* is presented below:

```
Algorithm 3 Function backtracking
Require: p: parent node, j: jump node, d: direction

distanceBetweenNodes = euclidenDistance(p, j)
 a[distanceBetweenNodes] =  null // create an array of size of
distanceBetweenNodes
while distanceBetweenNodes ≠ 0 do
        n = step(p, d)
        distancebetweenNodes = distancebetweenNodes – 1
        a[distancebetweenNodes] = n
return a
```

Algorithm 3: Backtracking mechanism in Cooperative JPS

For each jump node generated by the JPS [6] algorithm, we pass the jump node along with the parent of jump node to the function *backtracking mechanism* to generate all the nodes in between the parent node and jump node. Based on the direction of travel of unit we get each node from the parent node by constantly incrementing in the direction of travel till we reach the jump node. Once we have all the nodes, we check the availability of each node and store it in reservation table.

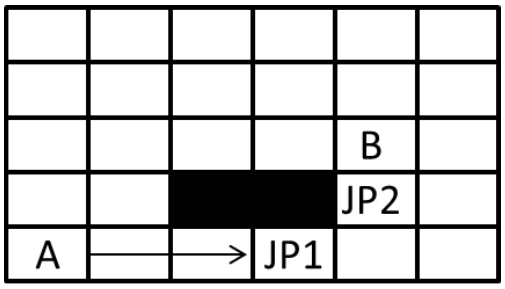


Figure 16: Backtracking Mechanism Example

In figure 16 we have a unit with start node ‘A’ and goal node ‘B’ on an 8-way square grid. When JPS [6] is used to find the path the above unit, the path generated will be represented as A->JP1->JP2->B which would only contain the jump points between nodes ‘A’ and ‘B’ discarding all the nodes between the nodes on the path. Using the above *backtracking mechanism* on each unit and for each expanded nodes, in our example the first jump point from start node is JP1. Once we have JP1, we run the backtracking mechanism to find nodes between the parent of JP1, i.e, ‘A’ and add it to the reservation table. After applying backtracking mechanism on above unit we have all the nodes from start node ‘A’ to goal node ‘B’ via JP1 and JP2.The same process applied to all units and for each expanded node.

4.1.2 Forced Selection:

Existing JPS [6] algorithm already has two stopping rules one for horizontal or vertical straight movement and one for diagonal movements. Just by using these stopping rules JPS [6] algorithm cannot differentiate between the High Contention Area (HCA) and Low Contention Area (LCA). Since Spatially Distributed Pathfinding framework (SDP) [18] have controllers for each area which allows an effective transfer of units from a narrow corridor to open areas. We have introduced a new stopping rule called “*Forced Selection*” which allows the unit to be transferred from one controller to other. As we already have high level path for each unit which acts as a heuristic guide for controllers on arrival of units to its area. By utilizing the high level path, the JPS [6] algorithm is stopped forcefully at one of the transaction nodes in HCA. Depending on the direction of travel of unit from its LCA to HCA, we first block all the inbound or outbound directed nodes in HCA. Once done, the search is stopped and search algorithm generates a transaction node as an expanded node.

The pseudocode for *forced selection* macro is presented below:

```
Algorithm 4  Function ForcedSelection
Require: n: expanded node, hlp: High level path of unit

if hlp contain a HCA then
    HCA = select all HCA in hlp
    while till there is no HCA do
        if n ∈ HCA
            return true
return false
```

Algorithm 4: Forced Selection in Cooperative JPS

For all the nodes expanded by JPS [6], we need to check whether the node belongs to a HCA or a LCA, so for each unit we would generate the High level area which represents all the areas which would be travelled by the unit. So for units travelling from LCA to HCA, we require transferring control of unit to HCA controller, to do so we have the forced selection macro which checks each expanded node for its correct belonging area. When we encounter a node which is part of HCA, we stop the search and transfer the control to that HCA controller.

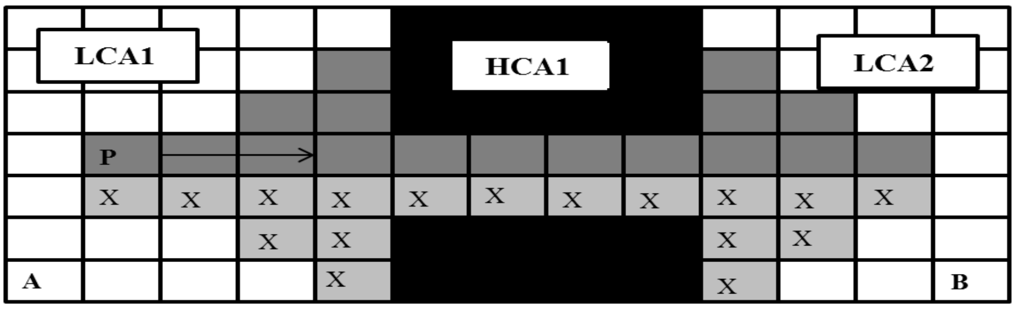


Figure 17: Forced Stopping on SDP framework

In figure 17 consider a unit ‘Z’ with start node ‘A’ and goal node ‘B’ on 8-way square grid map spatially partitioned into HCA and LCA. The greyed part in the figure represents the HCA1 and the area side of it is represented as LCA1 and LCA2. First we need to block either the inbound nodes or outbound nodes in HCA1 based on the direction of travel which can be obtained by the high level path for unit ‘Z’ that can be represented as LCA1->HCA1->LCA2(i.e. one of nodes in each area to represent the entire area). Using the high level path, we apply a forced selection by stopping the JPS [6] algorithm at a transition node ‘P’. The forced selection is implemented while expanding the JPS algorithm which checks for the only for the nodes in HCA, so the first recognised node in HCA is selected for transferring the control of unit from LCA1 controller to HCA1 controller.

4.1.3 Collision Avoidance:

In some real time scenario such as video games where the solution time is more important than actual pathlength, we cannot just use the ‘wait’ macro used in Cooperative A* [13] for collision avoidance. So we propose new collision avoidance technique called “*Side-Way Movement*” in Cooperative JPS. When there is collision between the units, the first preference is given to side-way movement than wait macro. In *side-way movement* the low priority unit i.e. unit which is trying to occupy a node which is already occupied by unit in collision, will deviate from its optimal path and occupy one of the nodes on either of side of collision node. There are two variants to *side-way movement* first being the straight movement of unit with either horizontal or vertical movement and second is diagonal movement. Consider a unit ‘Z’ with ‘N’ nodes as path i.e. $n_1....n_g$, where n_1 being the start node and n_g being the goal node which would be generated after the backtracking process. Consider a collision node n(x, y) on the path of unit ‘Z’ then for a straight movement we would consider either the node n(x-1, y) or n(x+1, y) for horizontal movement and for vertical movement we would either consider n(x, y -1) or n(x, y+1) as save node and move the unit to one of the nodes to avoid collision. When the collision node 'n' is at the diagonal travel of unit then we would consider either the nodes n(x, y-1) and n(x+1, y) or n(x-1, y) and n(x, y+1) as save nodes to move the unit to avoid collision. Both diagonal and straight side-way movement for collision node n(x, y) is shown in figure 18(a) and figure 18(b) respectively.

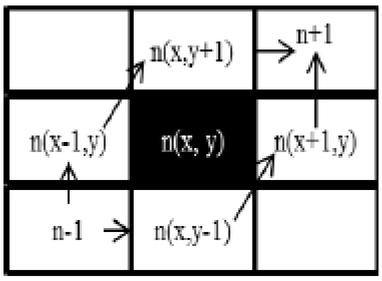


Figure 18(a): Diagonal side-way movement

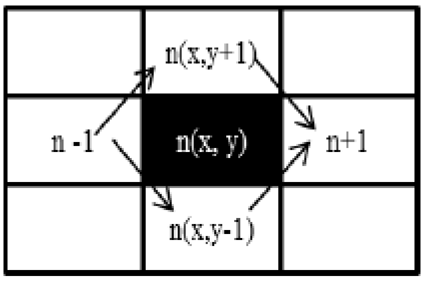


Figure 18(b): Straight horizontal side-way movement

Depending upon the above discussion we can conclude with 4 main cases:

Case 1: Straight side-way movement (either horizontal or vertical)

Collision node – n(x, y)

Horizontal side-way movement – n(x-1, y) or n(x+1, y)

Vertical side-way movement – n(x, y-1) or n(x,y+1)

Case 2: Diagonal side-way movement

Collision node – n(x, y)

Diagonal side-way movement – n(x-1, y) and n(x, y+1) or n(x, y-1) and n(x+1, y)

Case 3: When the side-way movement is not available

Collision node – n(x, y)

Wait node – (n-1) waiting node will always be the node previous to collision node

Case 4: When both side-way movement and wait node is not available

Collision node – n(x, y)

Wait node – ((n-1)-1) waiting node will be two nodes way from the collision node

Waiting node – we will iterate from (n-1) till its parent node to find the waiting node

Below is the pseudocode for collision avoidance techniques used in Cooperative JPS.

```
Algorithm 5 Function CollisionAvoidance
Require: c: Collision Node, d : direction of unit, p: previous node to 'n',t: time of collision

  n = sideWayMovement(c, d)  // based on direction of travel generates the pair of evading
                                                  node/nodes
if n ≠ 0 then
        x = Select one of evading node/nodes
        x.time = t          //for diagonal evading nodes the time 't' is added  appropriately
        return x
if n = 0 then
        if waiting at p is available then
                p.time = t                  // we create a new node at 'p' with updated time t
                return wait(p)
        if waiting at p is not available then
                q = parent of node p
                r = all nodes from p to q
                t = t -1                    // since waiting node could be two space before 'c'
                for all z ∈ r do
                        if wait(z) is available then
                                z.time = t
                                return wait(z)
                t = t -1
```

Algorithm 5: Collision Avoidance in Cooperative JPS

When we encounter a collision node, the first step is to find the pair of evading node/nodes using *side-way movement* function and return one of the available evading node/nodes. The algorithm for *side-way movement* is presented in algorithm 4. In some cases the evading node/nodes are blocked or not available then we resort to wait action at node previous to collision node. But when the number of units on the map is very high there could be chances that both *side-way movement* nodes and waiting node be unavailable, than we iterate from the previous node to collision node to its parent node to find a waiting node.

```
Algorithm 4 Function sideWayMovement
Require: c: collision node, d: direction of travel of node

if d is straight horizontal then
        n1 = c(x-1, y)
        n2 = c(x+1, y)
        return n1 and n2
if d is straight vertical then
        n1 = c(x, y -1)
        n2 = c(x, y+1)
        return n1 and n2
if d is diagonal then
        n1 = c(x-1, y) and c(x, y+1)
        n2 = c(x, y-1) and c(x+1, y)
        return n1 and n2
```

Algorithm 6: Side-way movement in Cooperative JPS

4.1.4 High level Path using JPS:

Instead of using an expensive A* algorithm to generate the high level path for individual units, we use JPS [6] to increase the overall solution time of spatially distributed pathfinding algorithm with cooperative JPS. In figure 19; we have two unit ‘A’ with start node S1 and goal node G1 and unit ‘B’ with start node S2 and goal node G2. Unit ‘A’ starts from LCA1 and ‘B’ starts from LCA2. So when JPS algorithm is used to find the paths for both units individual, we pick only one node in each area to represent the entire area for high level path consideration. In HCA we pick the central core nodes for consideration depending on the direction of travel of units. So in our example the High level path for unit ‘A’ will be S1->H1->G1 and for unit ‘B’ it will be S2->H2->G2.

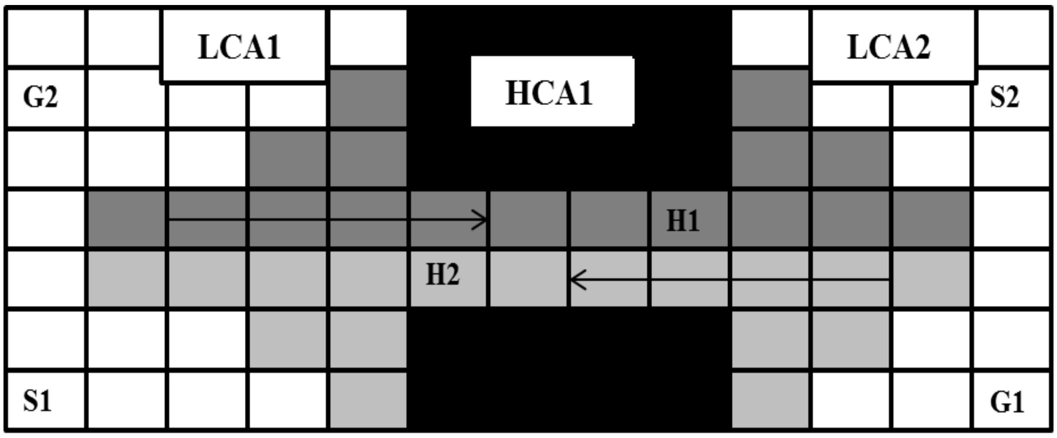


Figure 19: High level path using JPS for two units on SDP framework

4.2 Summary:

In this chapter we introduced a novel MAPF algorithm called Cooperative JPS which is implemented on SDP framework. We showed various techniques required to incorporate JPS [6] into a Cooperative environment on SDP framework. We proposed a backtracking mechanism for JPS [6] algorithm to find all the nodes in the path for units. Using the backtracking mechanism we introduce a new stopping rule for JPS [6] to differentiate between HCA and LCA. A collision avoidance technique called side-way movement was introduced along with wait macro in existing cooperative A* [13] to increase the solution time of individual units. And finally to reduce the solution time of overall SDP with cooperative JPS algorithm we used JPS [6] instead of A*. In the following chapter we compare our work with existing SDP with Cooperative A* algorithm [13] to measure the performance of our work in terms of *makespan, overall solution time, failure-rate.*

CHAPTER-5: EXPERIMENTAL SETUP AND RESULTS

We ran our experiments on an ASUS G46V laptop with Intel core i5 processor and RAM capacity of 8GB. We compared our Spatially Distributed Pathfinding algorithm with Cooperative JPS with the existing Spatially Distributed Pathfinding algorithm with Cooperative A* [13]. We would like to thank the authors for providing their code for our experiments. There work was completely written in JAVA. We have used the benchmark maps for one of the famous games from Blizzard gaming company called Dragon Age: Origins [23].

For our experiments we have considered 8 different maps from Dragon Age: Origins game with High Contention Area (HCA) ranging from 0 to 20 and number of open nodes for units traversal from 925 to 30236. We could not experiment on maps with higher open nodes because of laptops constraints. The maps used are square grids which allows both 4-way and 8-way traversal.

Start and goal nodes for each of the unit were randomly generated only on the LCA. Since HCA would usually be a narrow corridor, no start or goal nodes are placed. It would not be practical to place either the start or goal node in HCA because in any real life scenario such as a bridge we won’t have a parking lot, the bridge just acts as a passage for vehicles to move.

The results are represented starting for maps with least HCA to maps with higher value. A total of 530 problem instances were considered for our experiments, with 5 iterations run for each map which was enough to obtain accurate average results. The cost functions used in our experiments are makespan, overall solution time of algorithm, failure rate, individual Pathlength of unit causing the makespan.

5.1 orz704d.map:

The orz704d map has zero HCA in it so the entire map acts as one big LCA. Since there is only one LCA our work will just use Cooperative JPS and modified Cooperative A* in Wilt et al [18] work. There are 2097 open nodes for units traverse on the map. We started with 10 units gradually increasing by 10 units till 100 units. The figure 20(a) shows the comparison of makespan between our work and existing SDP with Cooperative A* [13]. As from the results our work completely dominates the existing work by around 59.2%. In figure 20(b) we have compared the overall solution time between the algorithm and by the result we see a decrease in solution time by 33%. Number of units failing in each of instance is shown in figure 20(c). The table 1(a) shows the actual values of makespan and solution time for individual unit causing the makespan. We also cumulated the failure rate of units over each run in table 1(b).

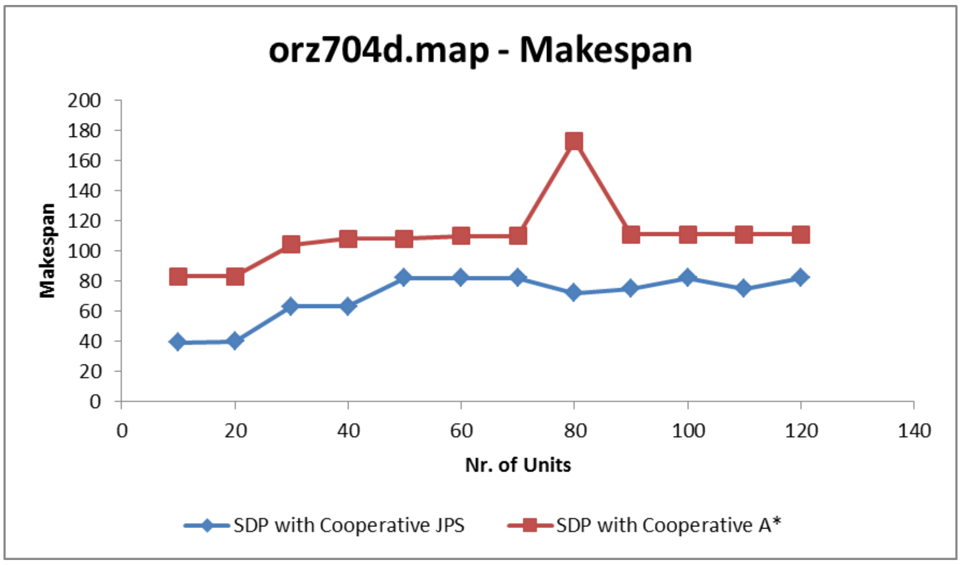


Figure 20(a): makespan generated for SDP with Cooperative JPS and SDP with Cooperative A on map orz704d.map*

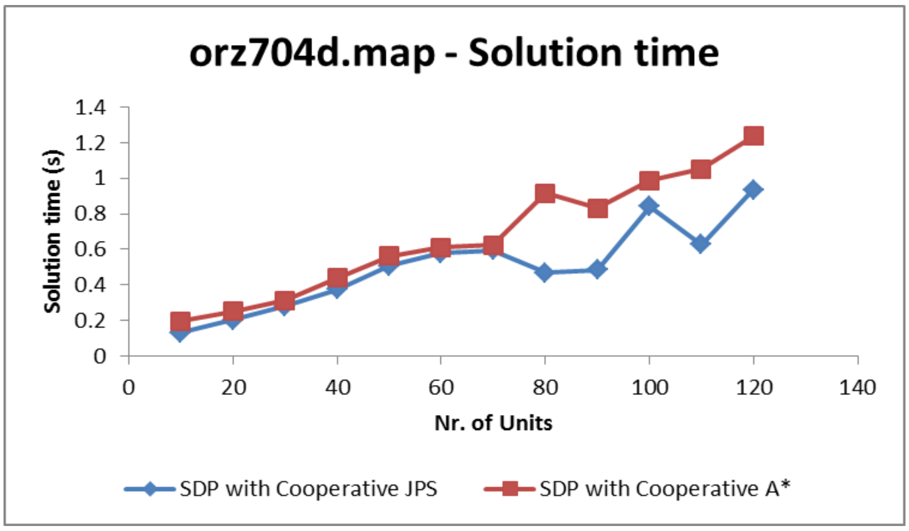


*Figure 20(b): Solution time generated for SDP with Cooperative JPS and SDP with Cooperative A*on orz704d map*

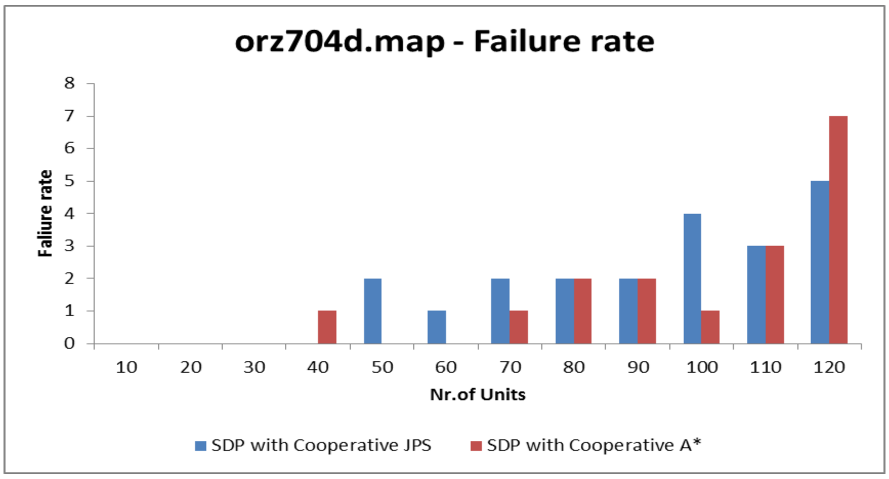


*Figure 20(c): Number of Failed units for SDP with Cooperative JPS and SDP with Cooperative A*on map orz704d over each instances*

Number of units	SDP with Cooperative A*			SDP with Cooperative JPS		
	Makespan		Solution time (seconds)	Makespan		Solution time (Seconds)
	Pathlength	Unit		Pathlength	Unit	
10	83	2	0.197	39	6	0.131
20	83	2	0.25	40	6	0.203
30	104	29	0.312	63	25	0.281
40	108	35	0.438	63	25	0.375
50	108	35	0.562	82	37	0.509
60	110	35	0.61	82	37	0.578
70	110	35	0.625	82	37	0.594
80	173	79	0.915	72	46	0.467
90	111	86	0.833	75	46	0.486
100	111	86	0.986	82	37	0.84
110	111	86	1.052	75	46	0.628
120	111	86	1.238	82	37	0.936

Table 1(a): Makespan and solution time on map orz704d

Number of units	Number of failed units	
	SDP with Cooperative A*	SDP with Cooperative JPS
10	0	0
20	0	0
30	0	0
40	1	0
50	0	2
60	0	1
70	1	2
80	2	2
90	2	2
100	1	4
110	3	3
120	7	5

Table 1(b): Number of failed units on map orz704d

5.2 den204d.map:

The den204d map consists of one HCA and as 15788 open nodes. With 10 units for each instances; a total of 250 units is ran on the map. The figure 21(a) shows makespan between the two algorithms with almost similar decrement in makespan of about 59.2%. We saw a massive decrease in solution time by about 117.12% as there are two large LCA on the map that allows our algorithm to perform better in figure 21(b). Similar to previous we also see a decrease in number of failed units in figure 21(c). The tables 2(a) and 2(b) gives the unit which is causing the makespan along with makespan and solution time for particular group of units.

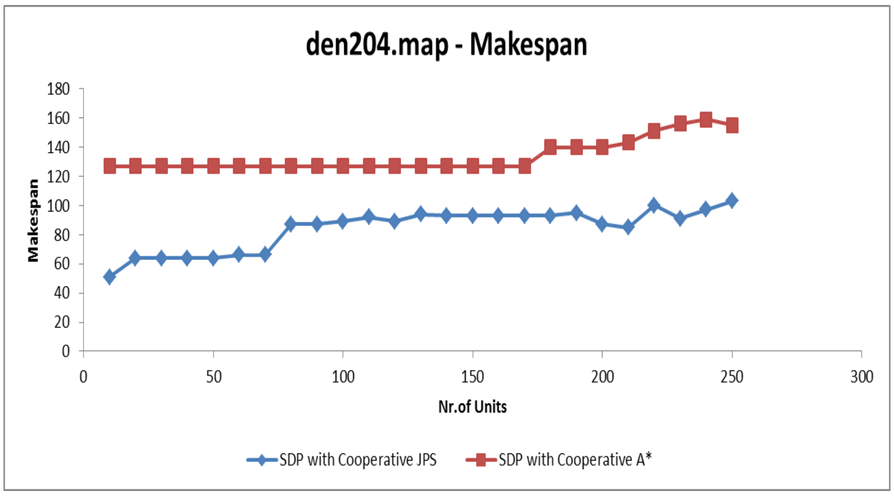


Figure 21(a): makespan generated for SDP with Cooperative JPS and SDP with Cooperative A on map den204.map*

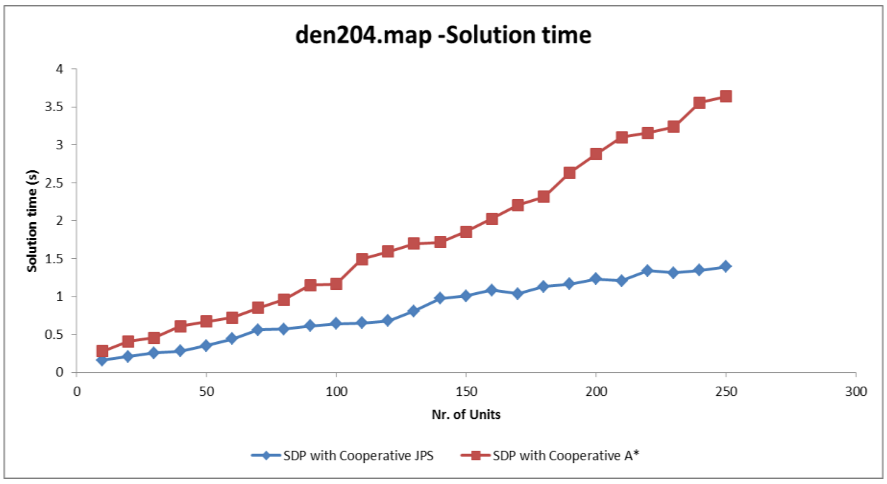


*Figure 21(b): Solution time generated for SDP with Cooperative JPS and SDP with Cooperative A*on den204d.map*

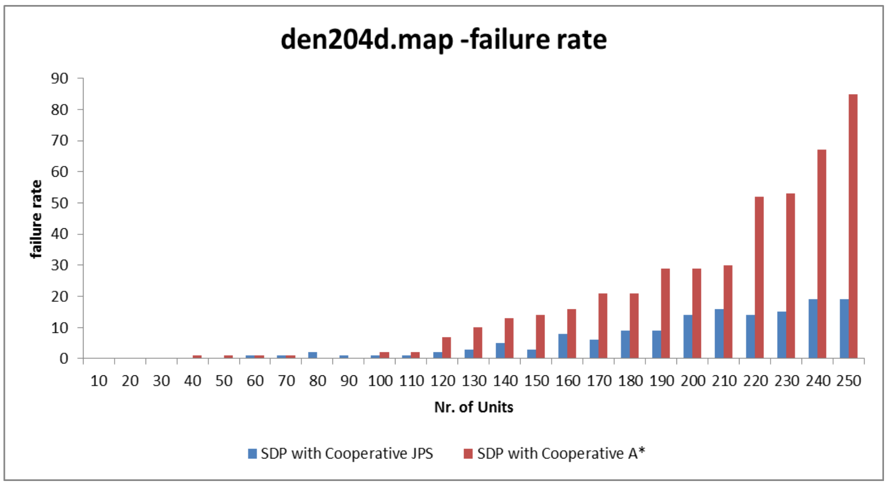


*Figure 21(c): Number of Failed units for SDP with Cooperative JPS and SDP with Cooperative A*on map den204d over each instances*

Number of units	SDP with Cooperative A*			SDP with Cooperative JPS		
	Makespan		Solution time (seconds)	Makespan		Solution time (Seconds)
	Pathlength	Unit		Pathlength	Unit	
10	127	1	0.282	51	6	0.1585
20	127	1	0.406	64	6	0.206
30	127	1	0.454	64	6	0.255
40	127	1	0.609	64	6	0.28
50	127	1	0.672	64	6	0.353
60	127	1	0.723	66	50	0.439
70	127	1	0.849	66	50	0.561
80	127	1	0.958	87	50	0.569
90	127	1	1.15	87	50	0.612
100	127	1	1.163	89	60	0.64
110	127	1	1.492	92	50	0.653
120	127	1	1.593	89	60	0.68
130	127	1	1.699	94	50	0.806
140	127	1	1.716	93	50	0.974
150	127	1	1.853	93	50	1.008
160	127	1	2.025	93	50	1.084
170	127	1	2.208	93	50	1.035
180	140	173	2.315	93	50	1.131
190	140	173	2.633	95	50	1.163
200	140	199	2.878	87	75	1.232
210	143	100	3.102	85	98	1.206
220	151	170	3.157	100	60	1.341
230	156	170	3.238	91	98	1.313
240	159	170	3.556	97	60	1.343
250	155	243	3.639	103	60	1.393

Table 2(a): Makespan and solution time on map den204d

Number of units	Number of failed units	
	SDP with Cooperative A*	SDP with Cooperative JPS
10	0	0
20	0	0
30	0	0
40	1	0
50	1	0
60	1	1
70	1	1
80	0	2
90	0	1
100	2	1
110	2	1
120	7	2
130	10	3
140	13	5
150	14	3
160	16	8
170	21	6
180	21	9
190	29	9

200	29	14
210	30	16
220	52	14
230	53	15
240	67	19
250	85	19

Table 2(b): Number of failed units on map den204d

5.3 den401d.map:

We gradually increase the number of HCA as well the number of open nodes on the maps by 11 and 11456 respectively. The experiments conducted are similar to the earlier maps, but we see some anomalies as we increase the HCA along with the increase in number of units on the map. On maps with higher HCA number we observe that when the number of units on the map is smaller our work tends to have the solution time close to existing algorithm, but stabilizes with increase in number of units. In figure 22(a) we see the makespan on map den401d between 2 algorithms with our work showing closer makespan to original work [22] at 15.57%. We still see a good decrement in terms of solution time with 75.49% in figure 22(b). Den401d is one of the maps consisting of long hallways as LCA, thus making our work less immune to failure compared to modified Cooperative A* in SDP [18] as shown in figure 22(c). The tables 3(a) and 3(b) gives the unit which is causing the makespan along with makespan and solution time for particular group of units.

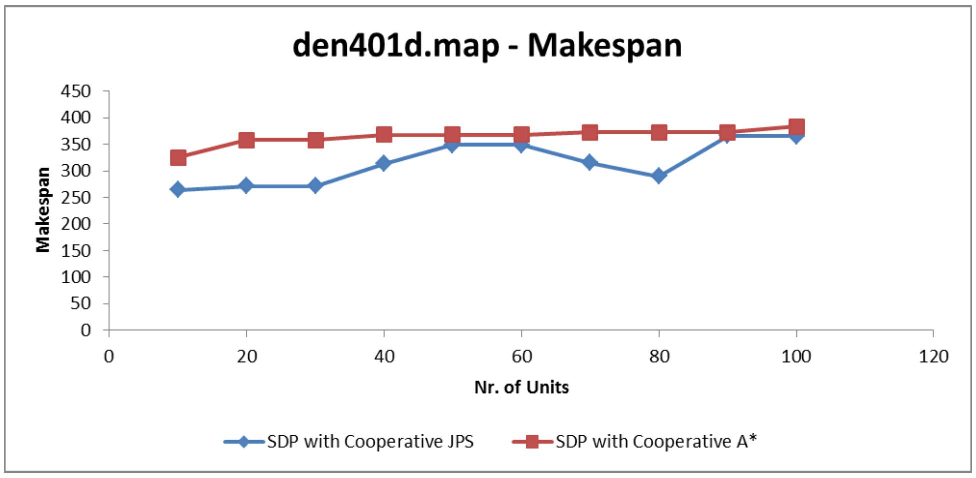


Figure 22(a): makespan generated for SDP with Cooperative JPS and SDP with Cooperative A on map den401d.map*

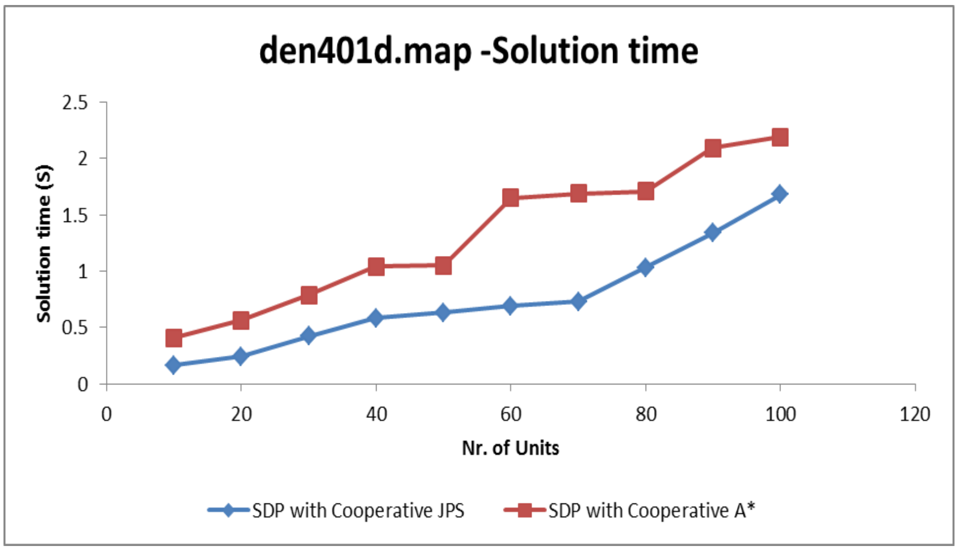


*Figure 22(b): Solution time generated for SDP with Cooperative JPS and SDP with Cooperative A*on den401d.map*

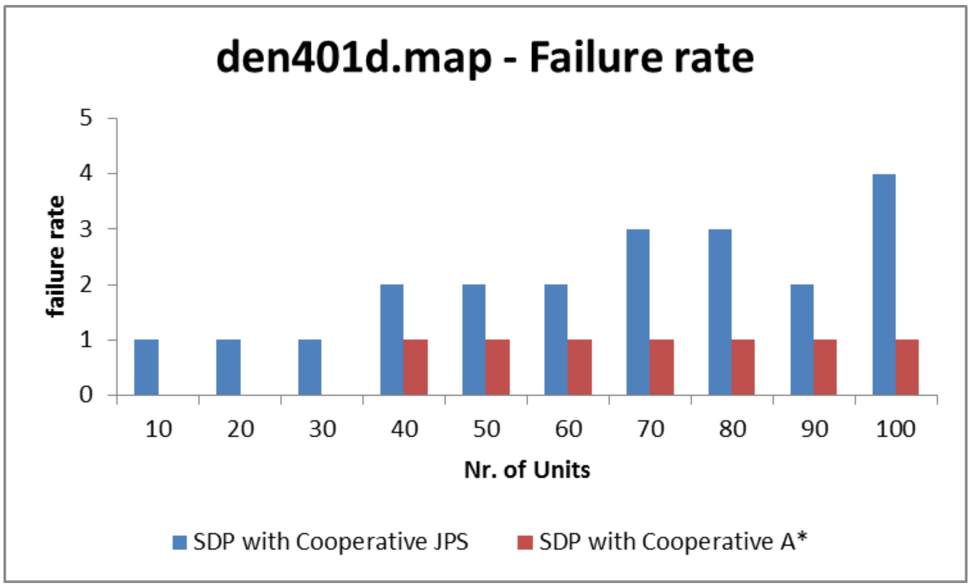


*Figure 22(c): Number of Failed units for SDP with Cooperative JPS and SDP with Cooperative A*on map den401d over each instances*

Number of units	SDP with Cooperative A*			SDP with Cooperative JPS		
	Makespan		Solution time (seconds)	Makespan		Solution time (Seconds)
	Pathlength	Unit		Pathlength	Unit	
10	325	0	0.409	264	0	0.164
20	358	10	0.564	271	15	0.243
30	358	10	0.79	271	13	0.423
40	368	10	1.044	313	33	0.583
50	368	10	1.05	349	37	0.634

60	368	10	1.649	349	37	0.69
70	372	10	1.69	315	33	0.73
80	372	10	1.71	290	77	1.03
90	372	10	2.093	366	33	1.34
100	383	10	2.193	365	33	1.68

Table 3(a): Makespan and solution time on map den401d

Number of units	Number of failed units	
	SDP with Cooperative A*	SDP with Cooperative JPS
10	0	1
20	0	1
30	0	1
40	1	2
50	1	2
60	1	2
70	1	3
80	1	3
90	1	2
100	1	4

Table 3(b): Number of failed units on map den401d

5.4 den505d.map:

The map den505d is one largest in terms of both HCA and open nodes with 20 and 30236 respectively. We start our rans with 20 units and increasing with the same amount till 360 units. In figure 23(a) we present the makespan for both algorithms with our algorithm showing a better result with decreased makespan of about 22.61%. As mentioned earlier with increased HCA our work tends to slightly more solution time, as we use the blocking mechanism to earlier units. In figure 23(b) we see a solution time on map den505d with mean decrement of about 30%. The failure rate is completely proportional to the number of units, so as the number of units increase we see a massive failure rate on existing approach compared to our work in figure 23(c). The failed units in most cases are the units which have already reached the goal and helping other units to have an optimal path through its goal node. The tables 4(a) and 4(b) gives the unit which is causing the makespan along with makespan and solution time for particular group of units.

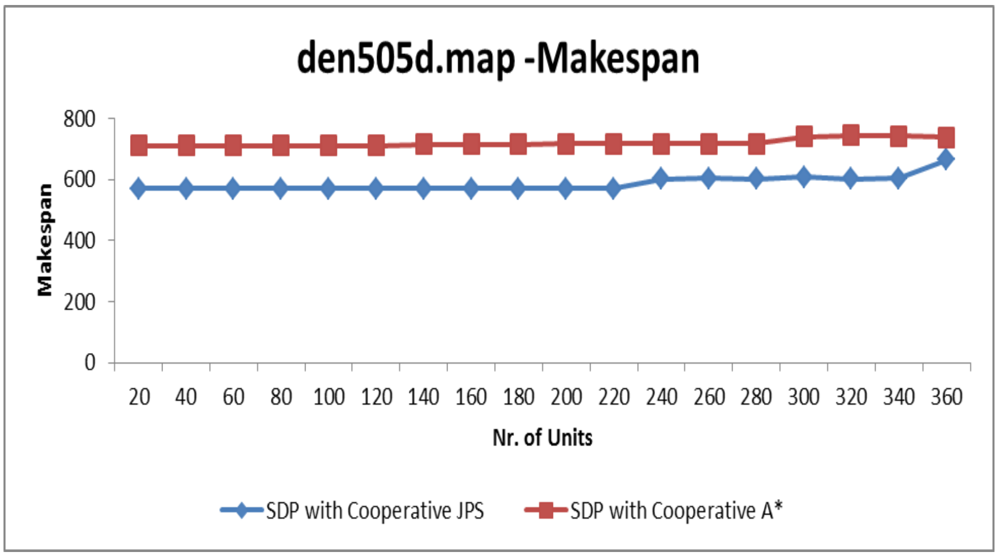


Figure 23(a): makespan generated for SDP with Cooperative JPS and SDP with Cooperative A on map den505d.map*

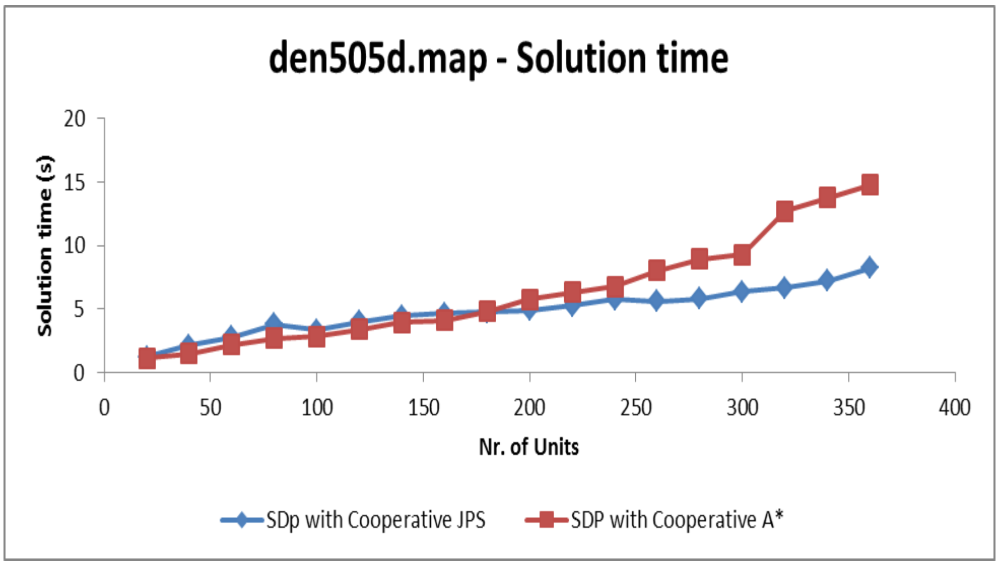


*Figure 23(b): Solution time generated for SDP with Cooperative JPS and SDP with Cooperative A*on den505d.map*

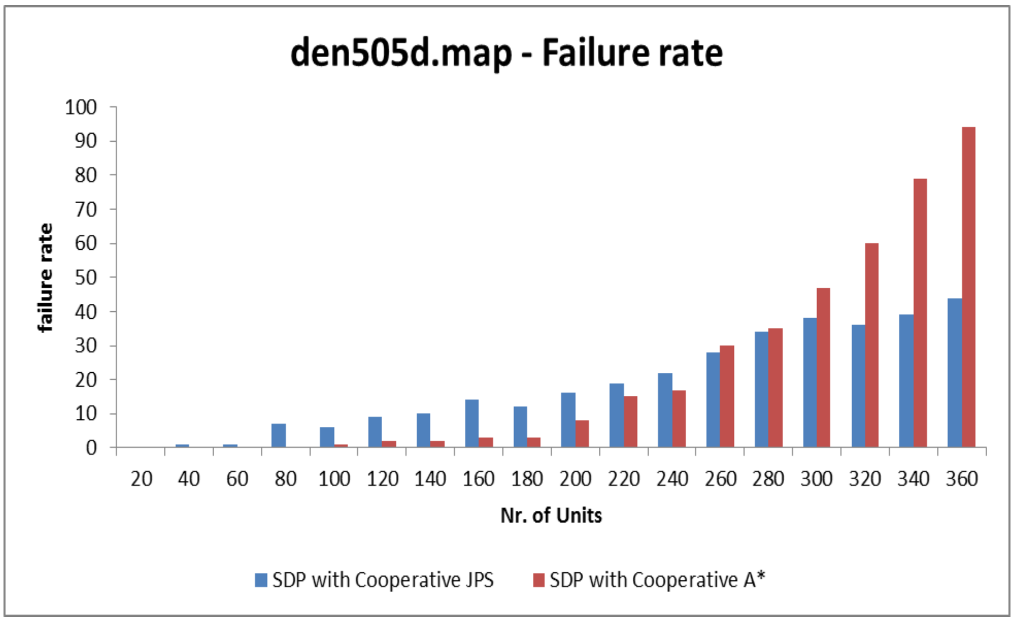


*Figure 23(c): Number of Failed units for SDP with Cooperative JPS and SDP with Cooperative A*on map den505d over each instances*

Number of units	SDP with Cooperative A*			SDP with Cooperative JPS		
	Makespan		Solution time (seconds)	Makespan		Solution time (Seconds)
	Pathlength	Unit		Pathlength	Unit	
20	711	10	1.123	571	10	1.246
40	711	10	1.46	571	10	2.141
60	711	10	2.177	571	10	2.788
80	711	10	2.713	571	10	3.775
100	711	10	2.847	571	10	3.325
120	711	10	3.408	571	10	3.986
140	715	10	3.964	571	10	4.427
160	715	10	4.095	571	10	4.631
180	715	10	4.792	571	10	4.761
200	717	10	5.743	571	10	4.891
220	717	10	6.289	571	10	5.23
240	717	10	6.773	603	115	5.76
260	717	10	8.032	604	115	5.604
280	717	10	8.902	603	115	5.805
300	741	283	9.274	609	283	6.37
320	744	283	12.686	603	115	6.66
340	743	283	13.748	604	115	7.23
360	739	283	14.772	665	115	8.23

Table 4(a): Makespan and solution time on map den505d

Number of units	Number of failed units	
	SDP with Cooperative A*	SDP with Cooperative JPS
10	0	0
20	0	1
30	0	1
40	0	7
50	1	6

60	2	9
70	2	10
80	3	14
90	3	12
100	8	16
110	15	19
120	17	22
130	30	28
140	35	34
150	47	38
160	60	36
170	79	39
180	94	44

Table 4(b): Number of failed units on map den505d

The results for the remaining 4 maps are provided in Appendix 1.

5.2 Summary:

In this we presented the results of our work while comparing with SDP with Cooperative A* [18]. We observed that our work out performs the existing approach in terms of makespan, solution time and failure rate on maps with smaller HCA with decrement of makespan of about 59.2%, average solution time of both the maps of about 75.09% and failure rate being very less compared to existing work. As we increase the HCA we saw having smaller makespan of about 18.78% and solution time of about 52.74%. The failure rate tend to be close to existing work when the number of units is less but becomes stabilized as the number of units is increased.

In the next chapter we will provide the conclusion statement of our thesis along with some future work which could be done on both Cooperative JPS and SDP framework to improve the overall the framework with Cooperative JPS.

CHAPTER-6: CONCLUSION AND FUTURE WORK

In our thesis, we have proposed a new novel algorithm called Cooperative JPS, which is the combination of Cooperative behaviour in Cooperative A*[18] and one of fastest single agent pathfinding algorithm Jump Point Search [6]. By combining Cooperative and JPS algorithms we have taken advantage of both the algorithms. To implement JPS in a multiagent environment we proposed macro’s such as Backtracking mechanism which allows capturing all the nodes between the parent node and the Jump point from JPS that are placed in the 3D space/time data structure called reservation table of Cooperation and new collision avoidance technique which allows a side-way movement for units during collision. After doing this we introduce a forced selection macro to Cooperative JPS to recognise between High Contention Area (HCA) and Low Contention Area (LCA) on SDP framework.

The main motivation for our work was to address the Multiagent Pathfinding Problem (MAPF) for units traversing on narrow corridor which as much usefulness in real work like GPS, video games, warehouse management etc. So we introduced Cooperative JPS algorithm on SDP framework that was used to address the above problem.

By using a new Cooperative JPS algorithm on SDP framework compared to existing SDP with Cooperative A* we saw a massive improvement in terms of makespan and solution time as per the results. On maps with open LCA’s we got better failure rate then on narrow corridors of LCA where we saw a slightly degrade on failure rate. We conducted our experiments on benchmark video game map of Dragon Age: Origins with HCA ranging from 0 to 20 and open nodes ranging from 925 to 23572. The maximum number units ran on one of the map was about 360.

From the results we saw improvement in terms of makespan of about 59.2% and solution time of about 75.09% on lower HCA maps and makespan of about 18.78% and solution time of about 52.74% on higher HCA maps. The failure rate were also very less on most of the maps. From our results we conclude that by incorporate Cooperative JPS in SDP framework we saw a significant improvements in terms of makespan, solution time and failure rate.

In our future work, we would like to improve the Cooperative JPS to handle dynamic environment in scenarios where the map could change while running the pathfinding algorithm such as in first person shooter games where a bridge could be destroyed that would make the units planning the route through the bridge unavailable. By introducing a modified Reduced Wait Time [17] algorithm into Cooperative JPS we would be able to do effective reroute of units in terms of solution time; when the HCA becomes destroyed. We would like to improve the SDP framework, by dynamic assigning the size of buffering area which could increase the solution time of overall algorithm. Our approach consumes more time while finding routes for units travelling in a narrow LCA hallway, so we would like to modify the JPS to handle the above scenario. One of main drawbacks of JPS is that it is only compliable for square grid maps; in future we would like to modify the JPS which would be capable to find paths for units in other regular grids.

REFERENCES:

[1]http://www.shiningrocksoftware.com/wp-content/uploads/2013/11/LongPath-1024x576.jpg

[2] http://www.randmcnally.com/product/intelliroute-tnd-510#features

[3] J. Carsten, A. Rankin, D. Ferguson, and A. Stentz, “Global planning on the mars exploration rovers: software integration and surface testing,” Journal of Field Robotics, vol. 26, no. 4, pp. 337–357, 2009.

[4] Skiena, S. "Dijkstra's Algorithm." Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica, Reading, MA: Addison-Wesley (1990): 225-227.

[5] Hart, Peter E., Nils J. Nilsson, and Bertram Raphael. "A formal basis for the heuristic determination of minimum cost paths." Systems Science and Cybernetics, IEEE Transactions on 4.2 (1968): 100-107.

[6] Harabor, Daniel Damir, and Alban Grastien. "Online Graph Pruning for Pathfinding On Grid Maps." AAAI. 2011.

[7] http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#heuristics-for-grid-maps

[8] Švestka, Petr, and Mark H. Overmars. "Coordinated path planning for multiple robots." Robotics and autonomous systems 23.3 (1998): 125-152.

[9] Ryan, Malcolm Ross Kinsella. "Exploiting subgraph structure in multi-robot path planning." Journal of Artificial Intelligence Research (2008): 497-542.

[10] Standley, Trevor Scott. "Finding Optimal Solutions to Cooperative Pathfinding Problems." AAAI. Vol. 1. 2010.

[11] Sharon, Guni, et al. "Pruning techniques for the increasing cost tree search for optimal multi-agent pathfinding." Fourth Annual Symposium on Combinatorial Search. 2011.

[12] Stout, B. 1996. Smart moves: Intelligent pathfinding. Game Developer Magazine April 1996.

[13] D. Silver. Cooperative Pathfinding. In AIIDE, pages 117– 122, 2005.

[14] Jansen, M. R., and Sturtevant, N. R. Direction maps for cooperative pathfinding. In AIIDE (2008).

[15] Wang, Ko-Hsin Cindy, and Adi Botea. "MAPP: a scalable multi-agent path planning algorithm with tractability and completeness guarantees." Journal of Artificial Intelligence Research (2011): 55-90.

[16] Bnaya, Zahy, and Ariel Felner. "Conflict-Oriented Windowed Hierarchical Cooperative A*." Robotics and Automation (ICRA), 2014 IEEE International Conference on. IEEE, 2014.

[17] Saeidianmanesh, Mehdi. Group Pathfinding Using Group Division. Diss. Concordia University, 2015.

[18] Wilt, Christopher, and Adi Botea. "Spatially Distributed Multiagent Path Planning." Twenty-Fourth International Conference on Automated Planning and Scheduling. 2014.

[19] Yu, Jingjin, and Steven M. LaValle. "Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs." AAAI. 2013.

[20] Standley, Trevor, and Richard Korf. "Complete algorithms for cooperative pathfinding problems." IJCAI. 2011.

[21] de Wilde, Boris, Adriaan W. ter Mors, and Cees Witteveen. "Push and rotate: cooperative multi-agent path planning." Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems. International Foundation for Autonomous Agents and Multiagent Systems, 2013.

[22] Luna, Ryan, and Kostas E. Bekris. "Push and swap: Fast cooperative path-finding with completeness guarantees." IJCAI. 2011.

[23] Sturtevant, Nathan R. "Benchmarks for grid-based pathfinding."Computational Intelligence and AI in Games, IEEE Transactions on 4.2 (2012): 144-148.

APPENDIX 1:

Results of rest of maps:

1) isound1.map: HCA is 2 and open nodes is 2976

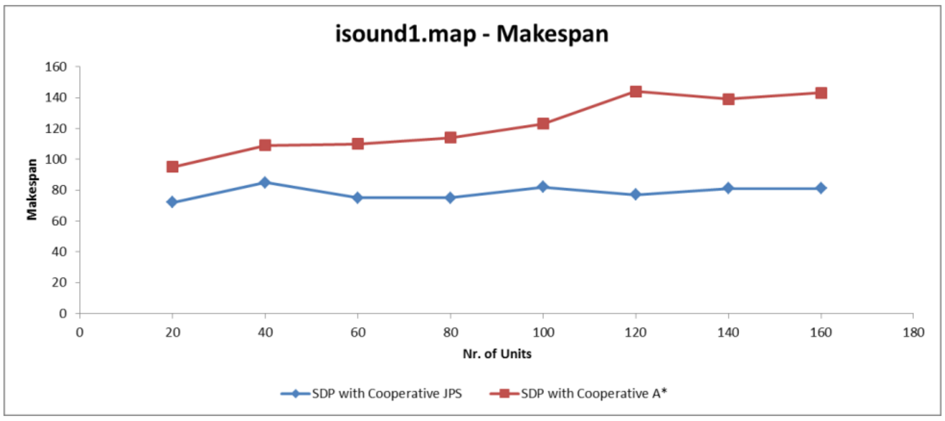


Figure 1(a): makespan generated for SDP with Cooperative JPS and SDP with Cooperative A on map isound1.map*

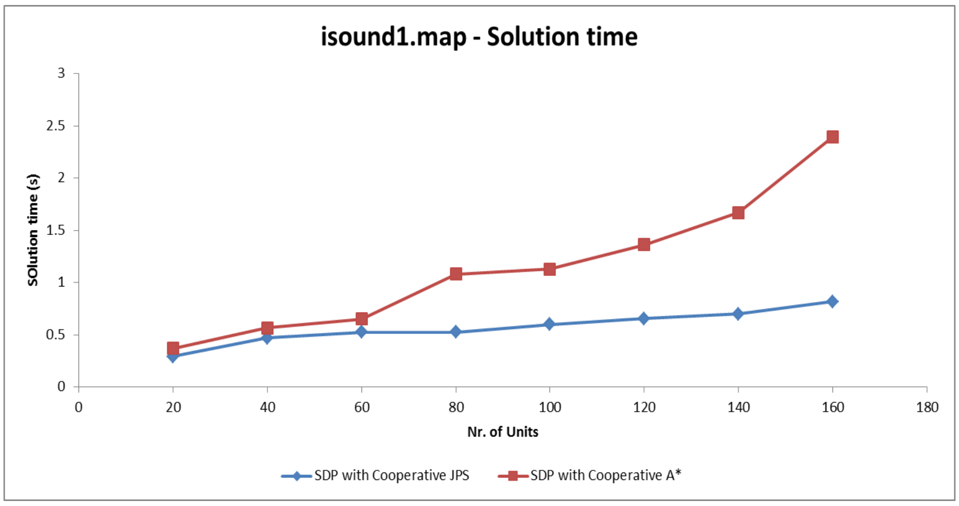


Figure 1(b): Solution time generated for SDP with Cooperative JPS and SDP with Cooperative A on map isound1.map*

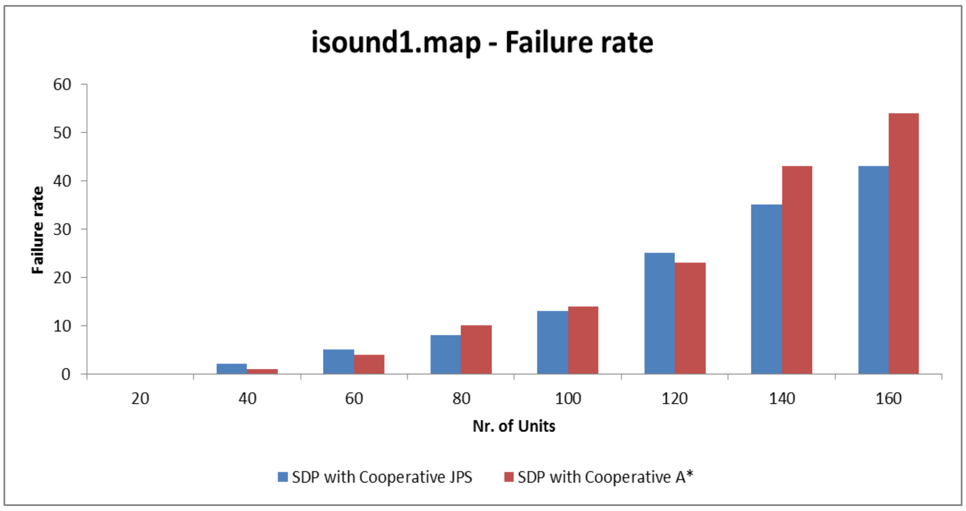


Figure 1(c): Failure rate generated for SDP with Cooperative JPS and SDP with Cooperative A on map isound1.map*

Number of units	SDP with Cooperative A*			SDP with Cooperative JPS		
	Makespan		Solution time (seconds)	Makespan		Solution time (Seconds)
	Pathlength	Unit		Pathlength	Unit	
20	95	14	0.367	72	3	0.291
40	109	30	0.566	85	30	0.47
60	110	30	0.652	75	3	0.525
80	114	30	1.082	75	3	0.525
100	123	99	1.127	82	30	0.598
120	144	99	1.362	77	99	0.655
140	139	130	1.671	81	130	0.698
160	143	66	2.395	81	130	0.818

Table 1(a): Makespan and Solution time on map isound1

Number of units	Number of failed units	
	SDP with Cooperative A*	SDP with Cooperative JPS
20	0	0
40	1	2
60	4	5
80	10	8
100	14	13
120	23	25
140	43	35
160	54	43

Table 1(b): Number of failed units on map isound1

2) **den405d.map:** HCA is 3 and open nodes is 925

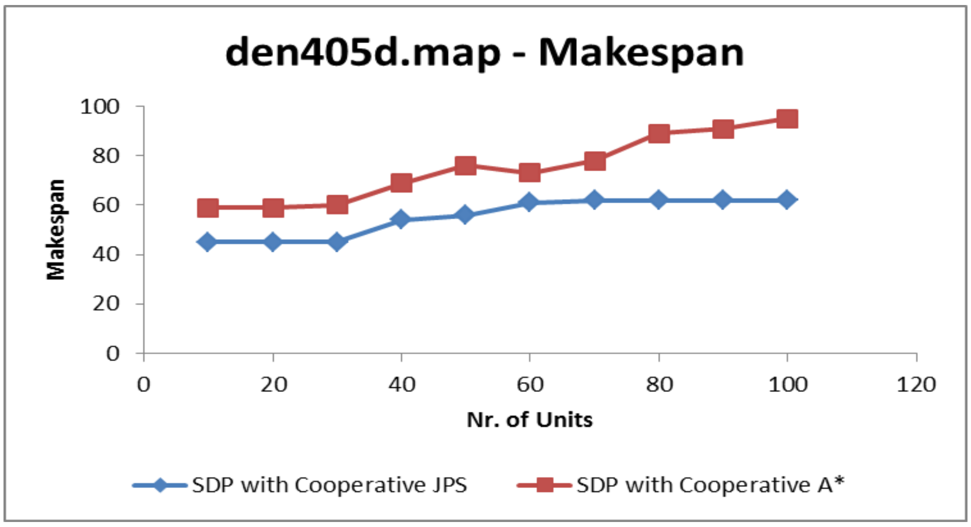


Figure 2(a): makespan generated for SDP with Cooperative JPS and SDP with Cooperative A on map den405d.map*

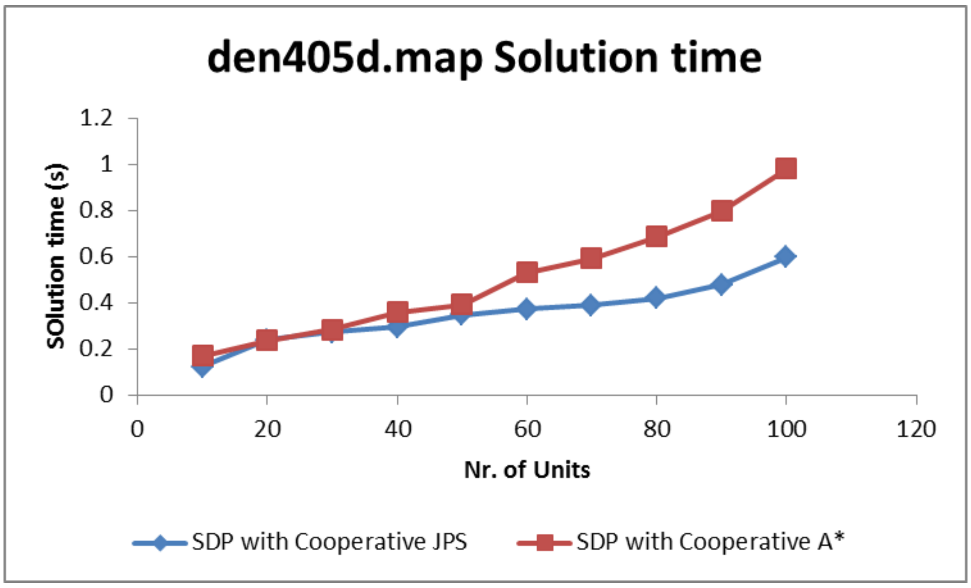


Figure 2(a): Solution time generated for SDP with Cooperative JPS and SDP with Cooperative A on map den405d.map*

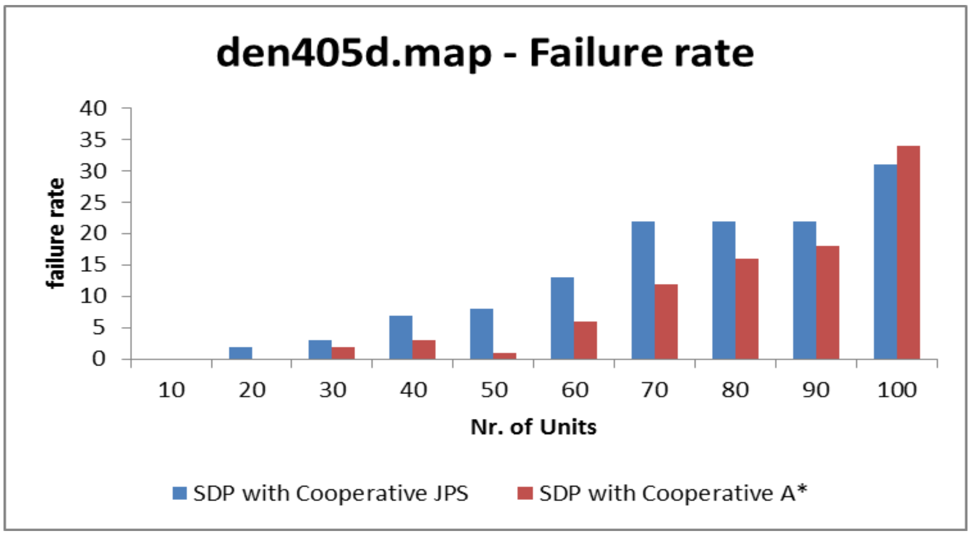


Figure 2(c): Failure rate generated for SDP with Cooperative JPS and SDP with Cooperative A on map den405d.map*

Number of units	SDP with Cooperative A*			SDP with Cooperative JPS		
	Makespan		Solution time (seconds)	Makespan		Solution time (Seconds)
	Pathlength	Unit		Pathlength	Unit	
10	59	0	0.172	45	0	0.123
20	59	0	0.238	45	0	0.24
30	60	2	0.285	45	0	0.273
40	69	34	0.36	54	34	0.296
50	76	34	0.392	56	28	0.35
60	73	53	0.532	61	53	0.375
70	78	38	0.594	62	53	0.39
80	89	52	0.687	62	53	0.42
90	91	52	0.8	62	53	0.48
100	95	52	0.984	62	53	0.6

Table 2(a): Makespan and Solution time on map den405d

Number of units	Number of failed units	
	SDP with Cooperative A*	SDP with Cooperative JPS
10	0	0
20	0	2
30	2	3
40	3	7
50	1	8
60	6	13
70	12	22
80	16	22
90	18	22
100	34	31

Table 2(b): Number of failed units on map den405d

3) **orz601d.map:** HCA is 6 and open nodes is 1890

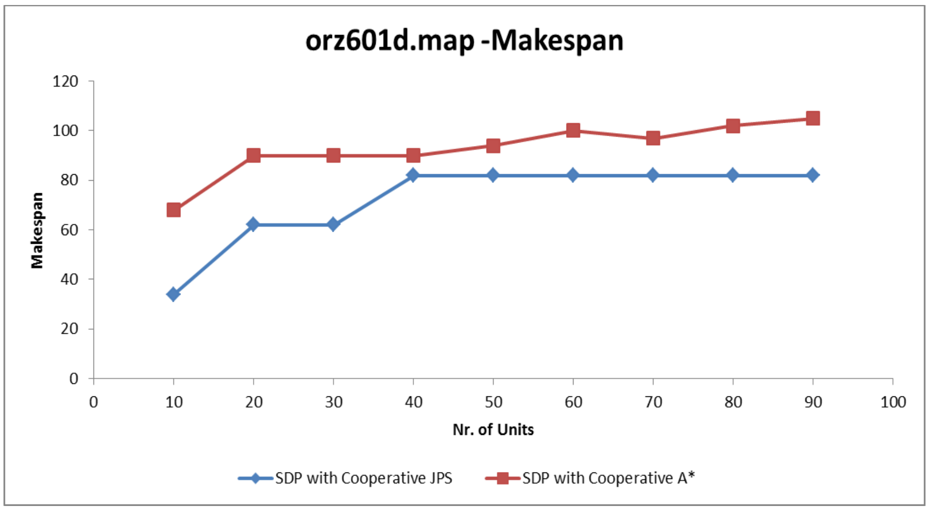


Figure 3(a): makespan generated for SDP with Cooperative JPS and SDP with Cooperative A on map orz601d.map*

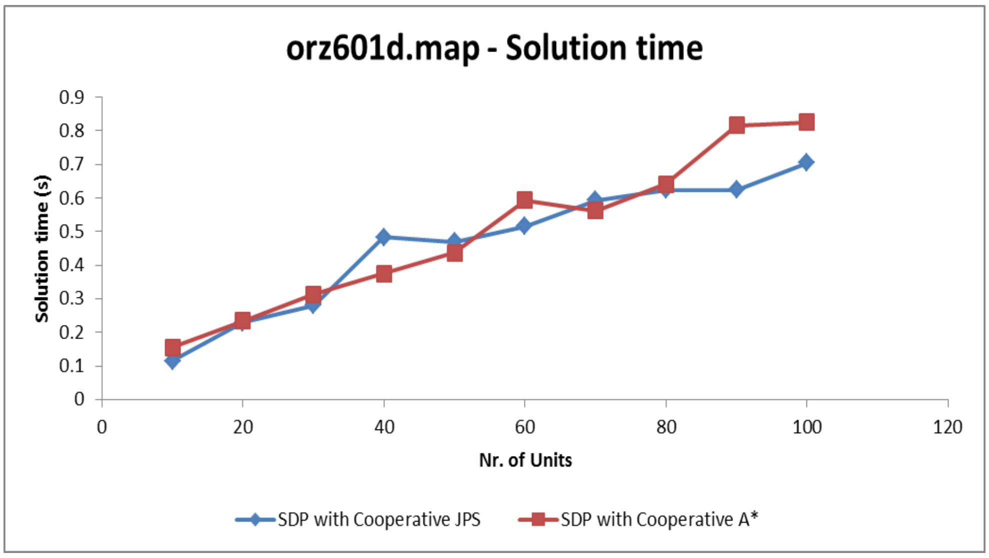


Figure 3(b): Solution time generated for SDP with Cooperative JPS and SDP with Cooperative A on map orz601d.map*

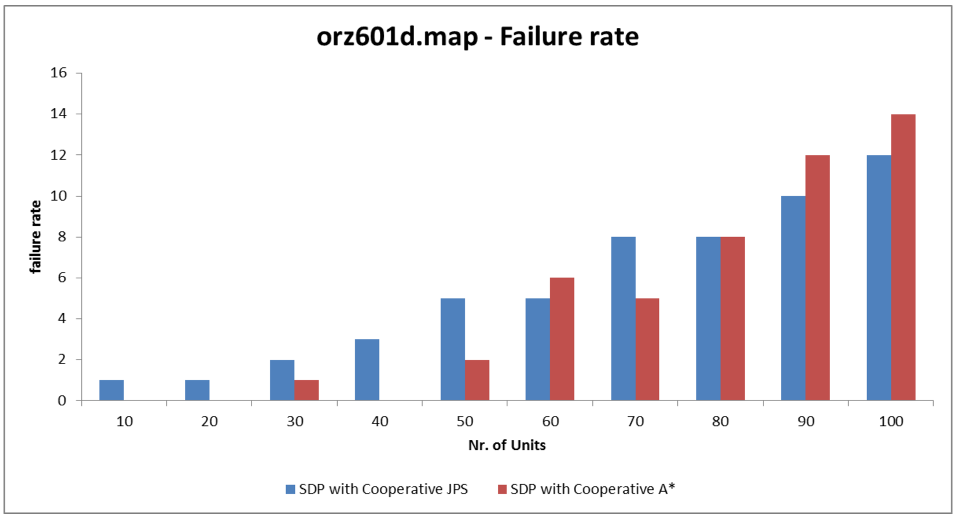


Figure 3(c): Failure rate generated for SDP with Cooperative JPS and SDP with Cooperative A on map orz601d.map*

Number of units	SDP with Cooperative A*			SDP with Cooperative JPS		
	Makespan		Solution time (seconds)	Makespan		Solution time (Seconds)
	Pathlength	Unit		Pathlength	Unit	
10	68	5	0.156	34	1	0.115
20	90	12	0.234	62	16	0.23
30	90	12	0.313	62	16	0.281
40	90	12	0.375	82	37	0.484
50	94	12	0.437	82	37	0.469
60	100	21	0.593	82	37	0.516
70	97	59	0.562	82	37	0.594
80	102	71	0.641	82	37	0.625
90	105	71	0.816	82	37	0.625
100	111	71	0.825	82	37	0.704

Table 3(a): Makespan and Solution time on map orz601

Number of units	Number of failed units	
	SDP with Cooperative A*	SDP with Cooperative JPS
10	0	1
20	0	1
30	1	2
40	0	3
50	2	5
60	6	5
70	5	8
80	8	8
90	12	10
100	14	12

Table 3(b): Number of failed units on map orz601

4) **hrt201d.map:** HCA is 15 and open nodes is 23572

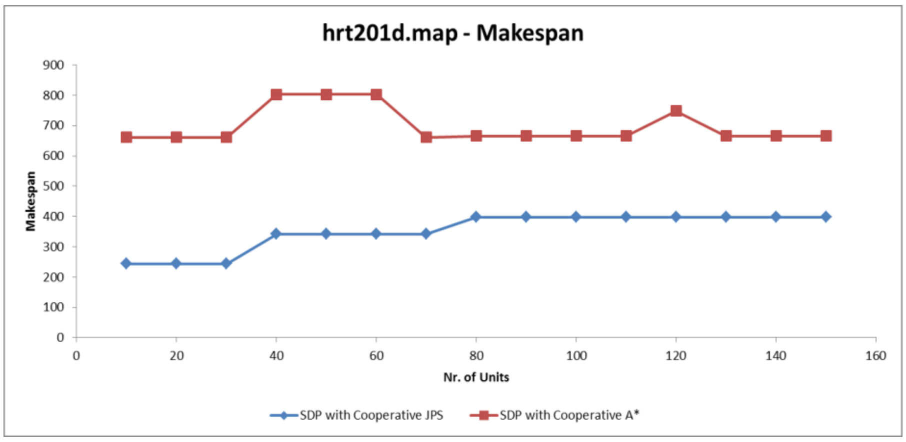


Figure 4(a): makespan generated for SDP with Cooperative JPS and SDP with Cooperative A on map hrt201d.map*

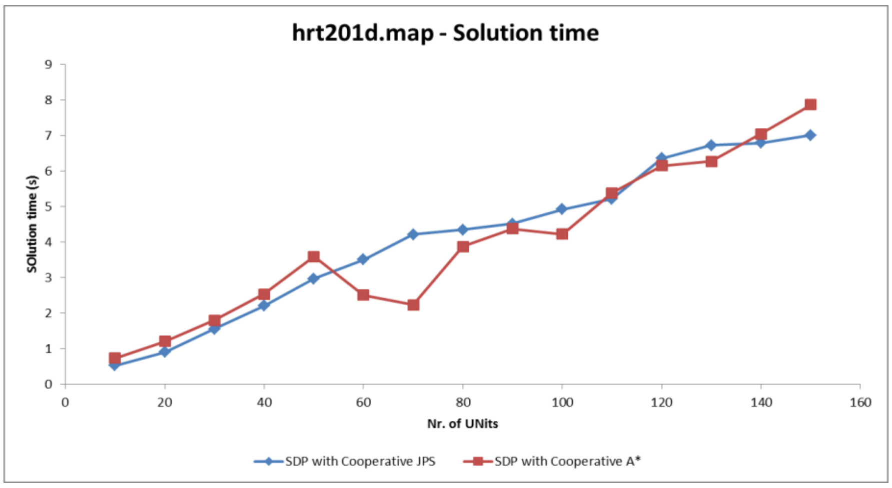


Figure 4(b): Solution time generated for SDP with Cooperative JPS and SDP with Cooperative A on map hrt201d.map*

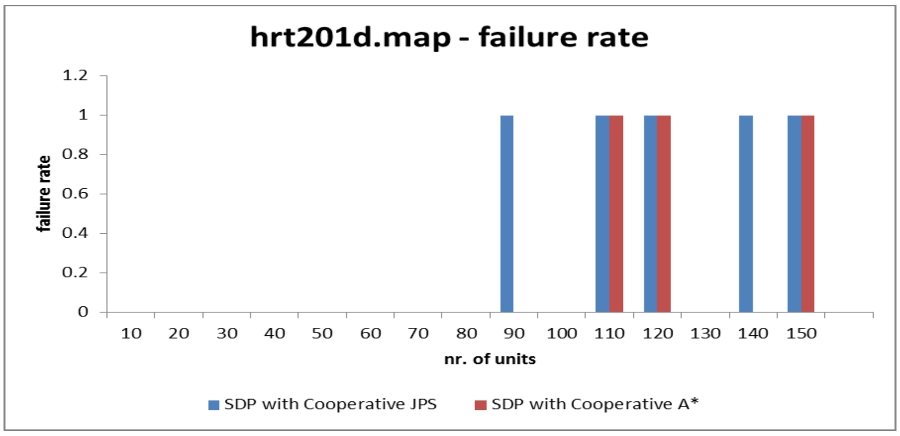


Figure 4(c): Failure rate generated for SDP with Cooperative JPS and SDP with Cooperative A on map hrt201d.map*

Number of units	SDP with Cooperative A*			SDP with Cooperative JPS		
	Makespan		Solution time (seconds)	Makespan		Solution time (Seconds)
	Pathlength	Unit		Pathlength	Unit	
10	661	6	0.729	244	2	0.524
20	661	6	1.203	244	2	0.903
30	661	6	1.793	244	2	1.555
40	803	39	2.534	342	31	2.199
50	803	39	3.593	342	31	2.968
60	803	39	2.503	342	31	3.501
70	661	6	2.227	342	31	4.216
80	666	6	3.876	398	70	4.341
90	666	6	4.377	398	70	4.512
100	666	6	4.218	398	70	4.92
110	666	6	5.381	398	70	5.21
120	748	50	6.152	398	70	6.36
130	666	6	6.268	398	70	6.72
140	666	6	7.04	398	70	6.79
150	666	6	7.87	398	70	7.01

Table 4(a): Makespan and Solution time on map hrt201

Number of units	Number of failed units	
	SDP with Cooperative A*	SDP with Cooperative JPS
10	0	0
20	0	0
30	0	0
40	0	0
50	0	0
60	0	0
70	0	0
80	0	0
90	1	0
100	0	0
110	1	1
120	1	1
130	0	0
140	1	0
150	1	1

Table 4(b): Number of failed units on map hrt201

VITA AUCTORIS

NAME: Sanjay Renukamurthy

EDUCATION: Master of Science, Computer Science

University of Windsor, Windsor ON, Canada

PUBLICATION

CONFERENCE: FLAIRS 2016

TITLE: Spatially Distributed Multiagent Pathfinding

Using Cooperative Jump Point Search

STATUS: Submitted 11-16-2015