SUBOPTIMAL CONFLICT BASED SEARCH FOR MULTI AGENT PATH FINDING

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

İLHAN YOLDAŞ KARABULUT

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

SEPTEMBER 2019

Approval of the thesis:

SUBOPTIMAL CONFLICT BASED SEARCH FOR MULTI AGENT PATH FINDING

submitted by **İLHAN YOLDAŞ KARABULUT** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** _______________

Prof. Dr. Halit Oğuztüzün
Head of Department, **Computer Engineering** _______________

Prof. Dr. Faruk Polat
Supervisor, **Computer Engineering, METU** _______________

Examining Committee Members:

Prof. Dr. İsmail Hakki Toroslu
Computer Engineering, METU _______________

Prof. Dr. Faruk Polat
Computer Engineering, METU _______________

Prof. Dr. Ahmet Cosar
Computer Engineering, THKU _______________

Date:

ABSTRACT

SUBOPTIMAL CONFLICT BASED SEARCH FOR MULTI AGENT PATH FINDING

Karabulut, İlhan Yoldaş
M.S., Department of Computer Engineering
Supervisor: Prof. Dr. Faruk Polat

September 2019, 43 pages

Multiagent pathfinding is a problem faced in many fields including computer games, simulation software, and robotics. These applications generally require complete and real-time solutions. The class of methods which solves the problem optimally and completely spends much more time than real-time solutions. The algorithms to solve the problem within expected time limits, are mostly incomplete and do not guarantee an optimum solution. In this thesis, we proposed a method that improves the Conflict Based Search algorithm which is complete and optimum. In this way, we combined the features of two different classes of solutions to develop an algorithm that can be used in real-life problems. Using a heuristic-based approach, we developed an algorithm which is complete, time-efficient and producing near-optimal solutions. We compared our method with CBS and suboptimal algorithms experimentally and showed the advantage of our proposed method.

Keywords: Path Finding, Multiagent Agent Path Findig, Search-Based Solutions,

Optimal Solvers, Suboptimal Solvers

ÖZ

ÇOK UNSURLU YOL BULMA İÇİN EN UYGUNA YAKIN ÇÖZÜMLÜ ÇATIŞMA TABANLI ARAMA

Karabulut, İlhan Yoldaş
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Faruk Polat

Eylül 2019 , 43 sayfa

Çok unsurlu yol bulma, bilgisayar oyunları, simülasyon yazılımları ve robotik gibi birçok alanda kaşılaşılan bir problemdir. Bu alanlar genellikle her örnek için ve gerçek zamanlı çözümlere ihtiyaç duyar. Sorunu optimum ve her örnek için çözen yöntemler, gerçek zamanlı çözümlerden çok daha fazla zaman harcamaktadır. Beklenilen zaman sınırları içerisinde çözüm bulan yöntemler ise çoğunlukla tüm örnekleri çözemiyor ve optimum çözümden uzaklar. Bu tezde, Conflict Based Search algoritmasını geliştirerek, tüm örnekler için sonuç veren ve optimum bir yöntem önerdik. Bu şekilde, sahada kullanılabilecek bir algoritma geliştirmek için iki farklı çözüm sınıfının özelliklerini birleştirmiş olduk. Sezgisel yaklaşım tabanlı bir yöntem kullanarak, her örnek için çalışan, zaman açısından verimli ve optimum çözümlere yakın sonuçlar veren bir algoritma geliştirdik. Yöntemimizi CBS ve optimum olmayan algoritmalarla karşılaştırdık ve yöntemimizin avantajlarını gösterdik.

Anahtar Kelimeler: Yol Bulma, Çok Unsurlu Yol Bulma, Arama Tabanlı Çözümler,

En Uygun Çözümler, En Uyguna Yakın Çözümler

To my family

ACKNOWLEDGMENTS

First of all, I would like to thank my supervisor Prof. Dr. Faruk Polat. Beside his supervision and guidance, He patiently encouraged me to complete this thesis work at every challenging moment.

Also, I want to thank my family and friends for their constant support.They are always with me through this journey, and I never felt alone.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ABBREVIATIONS	
MAPF	Multi-agent Path Finding
LRA*	Local Repair A Star
HCA*	Hierarchical A Star
WHCA*	Windowed Hierarchical A Star
CO-WHCA*	Conflict-Oriented Windowed Hierarchical Cooperative A*
FAR	Flow Annotation Re-planning
CBS	Conflict-Based Search
CT	Constraint Tree
SIC	Sum of Individual Cost
IDE	Integrated Development Environment

CHAPTER 1

INTRODUCTION

Single-agent pathfinding is the calculation of a route from a start position to an end position for a moving agent commonly modeled as a path search from one node to another in a graph. For this problem mostly search-based solutions are used. One of the commonly used methods is the A* algorithm [1]. A* searches graph nodes based on a cost function which is f = g + h, where g stands for the cost spent to reach parameter node, h is a heuristic value which estimates the cost to reach the goal. With an admissible heuristic, A* guarantees producing optimum solutions for single-agent pathfinding problems.

In the case of multi-agent path finding (MAPF) problems, the aim is transporting more than one agent from their start to goal nodes on the same graph. While doing this, a MAPF solver must consider the constraints like no two agents can be at the same graph node at the same time. For MAPF problems optimality is minimizing a cost function which is the sum of the path lengths of all agents. Because the state space grows exponentially with the number of agents, solving a MAPF problem optimally is NP-Hard[2] [3]. Some application domains including computer games, simulations, and robotics require real-time solutions. Optimal algorithms can not be computed in real-time as the problem size grows. That's why suboptimal solutions are acceptable.

In this thesis, we proposed a complete and time-efficient algorithm by modifying an existing optimal algorithm. While doing this, although we do not guarantee the optimality feature, the algorithm that we proposed reached competitive total path cost results compared to other suboptimal algorithms. Our method based on CBS algorithm [4]. The proposed algorithm creates a tree that consists of information gathered from conflicts of single-agent paths. By searching this tree according to the cost function, CBS tries to find an optimum solution. We changed this search mechanism so that it finds a solution in a shorter way by using a heuristic approach similar to A*. With this modification, we developed a complete, time-efficient but suboptimal solution.

The proposed method compared with CBS and some suboptimal algorithms which belong to different solution families and results are presented in the experimental work. In the experimental work, we used several graph configurations with varying number of agents.

We organized this thesis as follows. In chapter 2, we defined the problem with its constraints and classified the solution methods. In the next chapter, related work is given and evaluated by pointing at strong and week points of the algorithms. Chapter 4 covers the detailed explanation of the proposed method. The next chapter includes the experimental setup and input generation method. Then experimental results and evaluation of the algorithms are presented. The last chapter summarizes the thesis work and states the future research directions.

CHAPTER 2

BACKGROUND

Multi-agent pathfinding can be defined as planning routes from start position to goal position for more than one agent commonly modeled as a path search from one node to another in a graph. In this chapter, the fundamentals of the problem setup are explained. Definitions that are used in this thesis are also given.

2.1 Problem inputs

An environment and an agent list are the inputs of the MAPF problem. The environment mostly defined as a graph. Vertices of the graph define locations in which an agent can be placed at any time and edges define roads between vertices of the graph. Some of the vertices of the graph can be defined as obstacles that agents can not be placed at any time. Capacity can be settled for vertices which define the number of agents that can be placed on a vertex at the same time.

The agent list consists of a set of agents with start positions and goal positions. The aim is to find a path from start position to goal position for every agent without any conflict.

Formally;

An environment is a graph G:(V,E) where V=$\{v_0,...,v_n\}$ is the set of vertices and E=$\{e_1,...,e_t\}$ is the set of edges. e_i=(v_x,v_y) defines edge between two vertex.

A=$\{a_0,...,a_k\}$ is the set of agents, where a_i=(v_s,v_g) and v_s is initial node of agent i, v_g is the goal node of agent i.

2.2 Problem Outputs

The output of a MAPF problem instance is a path list. A path is a set of ordered vertices. Vertices in a path indicate positions of the agent at every time step from the start position to the goal position. A solution to the MAPF problem must satisfy constraints explained in the next section.

2.3 Constraints

The main constraint of a MAPF problem is that the number of agents on any vertex at any time must not be more than the capacity of that vertex. For simplicity, the capacity constraint is not used in this work. In other words, capacity assumed to be 1 and there must not be more than 1 agent on a vertex at a time.

2.4 Solution Classifications

There are different classifications of the MAPF problem solvers. Centralized and decentralized solvers are one of these classifications. Decentralized solvers calculate every agent's paths individually, then tries to fix collisions using some messaging mechanisms between agents. The information set can be limited based on the specific problem requirements. Decentralized solvers are more convenient to solve on distributed systems. For a centralized solver, a state contains all agents' positions. The solver finds the next positions of all agents for a state transition. Centralized solvers know every required information about all agents so a messaging mechanism is not needed [5].

Another classification is online and offline solvers. Online methods divide the solution into smaller time intervals, apply "solving" and "moving" phases for time intervals by proceeding. In "solving" phase paths are calculated, in "moving" phase paths are realized for every agent. Time intervals can be predefined or based on conflicts. Solving and moving phases are repeated until all paths are found for all agents. Offline solvers calculate paths from start positions to goal positions before any moving phase. Online solvers are commonly used in real-life applications[6].

CHAPTER 3

RELATED WORK

In this chapter, different solution alternatives are given in a classified way. Leading algorithms of different classes will be explained in detail with outstanding features. Elaborated algorithms will also be the subject of experimental results in later chapters.

Before explaining the solution methods, the A* algorithm explained first because the MAPF algorithms commonly based on this single-agent solver.

3.1 A* Algorithm

A* is a single agent pathfinding algorithm which is optimum, complete and time-efficient [1]. The search-based algorithm works by conducting best-first search on a tree using cost function f(n) = g(n) + h(n). g(n) is the cost from start node to node n and h(n) is the estimated cost form n to the goal. A* uses an open list and a closed list. Graph nodes are pushed to open list from start node and every time the node with minimum f(n) value is fetched. Goal test is executed on the fetched node if it is the goal node, a path from start to expanded node is returned else the neighbors of that node are pushed to the open list. The node is added to the closed list. Every time a node is pushed to the open list, it is checked that the node is not in the closed list and there is not any node in the open list whose f(n) value lower than the node. In this way, it is prevented to consider the same node again and again. If there is not any node left in the open list, before reaching to the goal node, the problem instance is unsolvable. Calculation of heuristic h(n) depends on the problem type but with an admissible heuristic meaning that never overestimating the cost, A* guarantees finding an optimal solution if one exists.

3.2 Sub-Optimal Search based MAPF Solvers

Sub-optimal MAPF algorithms find a solution time-efficiently by sacrificing optimality and/or completeness. Search based solutions attempt to find a path with a search algorithm for every agent without considering any other agents in the environment and then repair conflicts using different methods [6][7][8]. A family of algorithms that are commonly used in computer games is Local Repair A* (LRA*). In the basic implementation of LRA*, for every agent, paths are calculated from start positions to goal positions as any other agent does not exist in the environment, then while agents follow their paths if any conflict occurs remaining paths are recalculated [6]. This basic form of algorithm causes many deadlocks and cycles so different techniques are used to fix them. There is not an action taken before a conflict occurs by LRA* based algorithms. Hierarchical A* (HCA*) and its online version Windowed Hierarchical A*(WHCA*) solve this problem using reservation table[7]. HCA* keeps a reservation table that holds agents' feature positions calculated previously. Path calculation for an agent is done by considering reservation table entries recorded for agents while their paths are calculated. WHCA* defines a window size to hold reservations in a time interval and shifts that interval as agents move. Lastly Conflict-Oriented Windowed Hierarchical Cooperative A* (CO-WHCA*) which is a variant of WHCA* creates reservations around conflicts [8]. We worked on CO-WHCA* as a benchmark algorithm because it comes to the forefront with being efficient in terms of time, so a detailed explanation of CO-WHCA* as follows.

3.2.1 Conflict-Oriented Windowed Hierarchical Cooperative A*

Pseudo-code for CO-WHCA* [8] is given in Algorithm 1. CO-WHCA* algorithm is a sub-optimal search-based approach similar to Windowed Hierarchical Cooperative A* (WHCA*). As WHCA*, CO-WHCA* uses a fixed size reservation window. CO-WHCA* algorithm has two phases which are planning phase and moving phase. In the planning phase, paths are planned to some pre-defined distance. In the moving phase, agents walk through the planned path. Unlike WHCA*, CO-WHCA* does not keep a reservation window for all time and positions. The reservation window is only kept for a time interval that starts a pre-defined time before the collision and ends a predefined time after collision (Algorithm 1 lines 8,9). Furthermore, CO-WHCA* makes reservations for only one of the agents that subject to collision (Algorithm 1 line 7). CO-WCHA* places reservation table such that the conflict time comes to the center of the reservation table, in this way the planer has become aware of conflict before some time. Keeping the reservation table for only conflict regions and a single agent per collision makes CO-WCHA* memory efficient compared to the WHCA*. Despite all these improvements, selecting one of the agents subject to a collision is problematic. Because only one of the agents uses the reservation table, the selection of agents affects the solution quality. Moreover, this selection prevents finding a solution in some cases. As we show in the experimental results, success rates are substantially low due to the prioritization problem. To solve prioriti issues, an almost brute force method is proposed named CO-WCHA*P. CO-WCHA*P tries all different priority orderings and selects the one with the minimum sum of costs. Although this method improves solution lengths, its improvement on success rates is negligible and the time is required to calculate became extremely huge. [9][10][11][12]

Algorithm 1 CO-WHCA*

```
1: reset T
2: while some agents are not at their goal do
3:     for each agent $a_i$ do
4:         planPath($a_i, T$)
5:     end for
6:     $c = (A, v, t) \leftarrow$ findFirstConflict
7:     $a_c \leftarrow$ conflictOwner(c)
8:     $w_{start} \leftarrow$ movePosition(c)
9:     $w_{end} \leftarrow$ reservePosition(c)
10:    reserve $w_{start}...w_{end}$ st-points as $a_c$ in T
11:    for each agent $a_i$ in paralel do
12:        move $a_i$ to $w_{start}$
13:    end for
14: end while
```

3.3 Sub-Optimal Rule Based Solvers

Sub-optimal rule-based methods define some rules to calculate a path from start positions to goal positions. Algorithms that belong to this class are generally time-efficient. Because actions are taken when just a conflict occurs, produced solutions are not guaranteed to be optimal. Rule-based methods are complete for a class of graphs. Pebble motion[11] and Tree-based agent swapping strategy (TASS)[10] are belongs to rule-based methods. Push and Swap[13] and Push and Rotate [9] are more applicable methods to real-life problems. Push and Rotate is an extension of Push and Swap. The algorithm extends Push and Swap by grouping agents and prioritizing agent groups. Our experiments showed that the Push and Swap algorithm has satisfactory completeness results so we used Push and Swap in our experiments. Push and Swap is a rule-based suboptimal algorithm that is time-efficient and complete for the graphs that have two unoccupied vertices.

3.3.1 Push and Swap

The Push and Swap is a complete algorithm that is classified as a rule-based sub-optimal solution[13]. The algorithm works by calculating agents' paths one by one. For every agent, ignoring other agents, the shortest paths are calculated. The path is traversed until the path is blocked by another agent. When an agent comes across, the closest empty node is found. The blocked agent's position is considered as obstacle. Agents which are already reached at the goal positions are considered as obstacles as well. If a reachable empty node is found, the blocking agent is pushed to this node. If It is not found, the process continues with Swap primitive. The agents which are already at goal positions are ignored and an empty node that has at least two empty neighbors is found. The blocked agent and blocker agent swaps their positions at the empty node. If this kind of a node can not be found the process halts. In this way, the current agent can move its way. During the swap process, if it is reached to another blocking agent, push and swap primitives are repeated. To reach the swap position some agents can be moved from their positions. At the end of the swap process, these agents are returned to their initial positions. In this way, the swap primitive changes the positions of only two agents. This process continues until the parameter agent reaches its goal position and is continued with the next agents. When all agents are reached to their goal positions the process returns the solution. Because the method finds the paths one by one and the swap primitive cause too many agents to move, Push and Swap generates paths that are much longer than optimum ones. However, the algorithm can be evaluated as successful in terms of time-efficiency and completeness.

3.4 Hybrid Solvers

Hybrid solvers use search mechanisms on maps that are divided into regions or that are given flow directions using some rules. For Instance, graphs are divided into known forms like cliques and halls, the paths between these subgraphs are found and used for constraints of the problem[14][15]. Methods define flows using behaviors of previously moved agents [16] or define traffic flows in a preprocessing step as Flow Annotation Re-planning(FAR) [17]. The FAR algorithm prevents head-on collisions by defining traffic flows causes some deficiencies. We chose FAR in our experiments which is commonly used in the field. The algorithm is reviewed in the next subsection.

3.4.1 Flow Annotation Re-planning

[17] suggested a novel approach to solve multi-agent pathfinding problem named as Flow Annotation Re-planning .In a preprocessing step, the algorithm generates a new graph named Flow-Annotated Search Graph by assigning flow directions to the rows and columns of the input grid graph. In this way, the algorithm tries to prevent head-to-head collisions. The flow annotated search graph guarantees every two locations that are available from each other in the input graph remain accessible. Unidirectional connections induce dead-end paths and impassible gateways. the flow-annotated graph allows bi-directional or diagonal flows in these situations. Giving flows to rows and columns cause longer paths and it creates unnatural behaviors for agents other than vehicles. Similar to other suboptimal algorithms FAR starts with finding independent paths from start positions to goal positions for each agent ignor-

ing other agents. The algorithm uses a collision-deadlock prevention mechanism that enforces agents to move along straight lines if possible. Even so, there can be collisions, hence a simple reservation mechanism is used. Agents reserve the nodes on their paths for some time steps. If an agent can not reserve for a time step, it does not move for that time step. To create a hierarchy and prevent from dead-locks, in successive time steps priority is changed between agents with horizontal movement and agents with vertical movements. Not using re-planning during collision avoidance provides time-efficiency but local reservation mechanism fails for narrow gateways. Lastly, the algorithm uses the node density notion for solving the deadlock problem. When more than one agent waits for each other, the algorithm detects node with the maximum number of agents intend to pass through and the agent on this node moved away. This process repeated for more than one agent until the deadlock is broken down. Due to using a graph with flows and reservation mechanism, paths created by FAR are much longer than optimum ones. The algorithm reaches a solution spending less time than other suboptimal solvers but fails at finding a solution for graphs includes narrow gates and for large numbers of agents.

3.5 Optimal Solvers

Finding optimal solutions to MAPF problems have PSPACE-hard complexity because problem space is expanded exponentially as the number of agents grows [2] [3]. Different approaches were proposed to solve the MAPF problem optimally. One alternative is to define states to include all agents positions and then employ a single agent pathfinding problem using A* [18]. To speed up this process, some methods are proposed [19][20]. Another approach is finding paths for every agent individually and holding collisions in different data structures. By searching on these data structures algorithms try to find collision-free paths [5][4].

Although there are prominent algorithms that overcomes different aspects, there is no algorithm that is superior in all types of problem instances. Conflict-Based Search [4] is a recently proposed method that outperforms on many optimal solvers in small-sized dense graphs with a large number of agents. We modified the CBS algorithm for creating a time-efficient method. A detailed explanation of CBS is given in the next subsection.

3.5.1 Conflict-Based Search for Optimal Multi-Agent Path Finding

Conflict-Based Search (CBS) algorithm [4] is a centralized multi-agent pathfinding algorithm. CBS traverses a high-level search tree consists of constraints that are defined by collisions. CBS finds a path for every agent which does not violate constraints associated with it. A constraint is a three-tuple (a_i,n,t) where a_i is the i^{th} agent, n is a node in the graph and t is the time point. It means that a_i is involved in a conflict at the node n at time t. A constraint forbids an agent to be located on a node n at time t. Constraints are created by the result of a conflict which occurs when two or more agent located on the same node at the same time. Edge conflicts are not considered by the algorithm because of retaining simplicity. Edge conflicts can be handled similar to node conflicts with a minor change. The algorithm uses a two-level search mechanism named as *high-level search* and *low-level search*. High-level search is based on creating a binary search tree called Constraint Tree (CT). CT maintains constraints defined by conflicts. The high-level search starts with pushing the root node to the open list (Algorithm 2, line 5). Root node includes an empty list of constraints and a solution found by the low-level search. In high-level search, CT is traversed using the best-first search for the Sum of Individual Cost (SIC)[4] function so while the open list is not empty the node with lowest SIC value is popped (Algorithm 2, line 7). The Popped node will be the parent of newly created CT nodes. The solution contained by the popped node is examined to find whether there is a conflict or not (Algorithm 2, line 8). If the solution is conflict-free, it is an optimal solution to the MAPF problem instance. When a conflict found constraints are defined for every agent which is subjected to the conflict (Algorithm 2, line 14). For every constraint, a new CT node generated. The set of constraints belongs to the new CT node includes the constraints of the parent node and newly created constraint (Algorithm 2, line 14). To find a solution for the new CT node, a low-level search is invoked for only the constrained agent. The parent CT node's solution is updated with the newly calculated path (Algorithm 2, line 16). The last step is calculating SIC value for the new CT node and is pushing the new CT node to the open list (Algorithm 2, line 17,20). For low-level search, any single agent pathfinding algorithm can be used including that constraints are handled. CBS uses A* search and constraint handling is done by avoiding adding a node to A*'s open list if the agent is forbidden to locate at that node. When a conflict with more than two agents occurs, two different approaches can be used. The first alternative is to generate a new CT node for every agent by violating the binary feature of the CT. Otherwise, the situation is handled by creating new nodes for only the first two of the conflicting agents leaving other conflicts to be handled in the lower levels of the CT. Because both methods have similar complexity the second method is preferred in the algorithm.

For MAPF problems, time-efficiency is an important concern because in most cases real-time solutions are required in real-life applications. Although CBS has better experimental results than other optimal algorithms, still it does not sense using CBS in real-time applications. In the next chapter, a modified version of CBS will be introduced. Completeness will be the outstanding feature of this new algorithm while it can compete with other suboptimal methods in terms of time efficiency.

Algorithm 2 High level of CBS

```
Require: Graph,AgentList
 1: // construct root node
 2: initial constraints = Ø
 3: solution = run low level search (find individual paths for all agents)
 4: cost = sum of individual cost for root node's solution
 5: insert root node to open list
 6: while open list not empty do
 7:     $P \leftarrow$ node with least cost from open list // best node
 8:     Validate P //search for a conflict.
 9:     if P is a conflict free node then return path list in P // P has a solution
10:     end if
11:     $C \leftarrow$ first conflict $(a_i, a_j, v, t)$ in P
12:     for all agent $a_i$ in C do
13:         // construct new node
14:         constraints $\leftarrow$ constraints in P + $(a_i, v, t)$
15:         solution $\leftarrow$ solution in P
16:         Update solution by invoking low level search for $a_i$
17:         cost = sum of individual cost for new node's solution
18:         // end of new node construction
19:         if new node's cost $< \infty$ then
20:             insert new node to open list
21:         end if
22:     end for
23: end while
```

CHAPTER 4

SUBOPTIMAL CONFLICT-BASED SEARCH FOR MULTI-AGENT PATH FINDING

In this chapter, modifications that make CBS algorithm time-efficient and complete but suboptimal will be introduced. We named the proposed method as Suboptimal Conflict-Based Search (S-CBS) for multi-agent path finding.

4.1 Modifications

CBS has two levels of search; high-level search and low-level search. The high-level search is done to find a conflict-free node. The node with the minimum cost is examined whether it has a conflict or not and it is removed from the high-level tree. If a conflict is found low-level search is triggered for one of the conflicting agents. Time efficiency of the CBS algorithm is highly related to the number of high-level and low-level searches. Searching for the node with the least cost in the high-level search tree guarantees optimality but this is a costly process.

The first modification we made on CBS aims to solve the problem executing a minimum number of high-level searches. It takes place in selecting a node from the open list. S-CBS selects the node that has the minimum number of conflicts (Algorithm 3, line 7) instead of the node with the best cost. Because a node with a solution has no conflict, this modification leads us directly to the solution. But the found solution may not be optimum.

CBS examines nodes whether there is a conflict or not. When a conflict is detected the search is terminated. We need a new set of data which is conflict count for modifying

Algorithm 3 High Level of Suboptimal CBS

```
Require: Graph,AgentList
 1: // construct root node
 2: initial constraints = Ø
 3: solution = run low level search (find individual paths for all agents)
 4: cost = sum of individual cost for root node's solution
 5: Validate the paths in root node until the time that longest path ends // Store conflict count and first conflict
 6: if root node has no conflict then return path list in root // root has a solution
 7: end if
 8: insert root node to open list
 9: while open list not empty do
10:     $P \leftarrow$ node with lowest conflict count from open list
11:     $C \leftarrow$ first conflict $(a_i, a_j, v, t)$ in P
12:     for all agent $a_i$ in C do
13:         // construct new node
14:         constraints $\leftarrow$ constraints in P + $(a_i, v, t)$
15:         solution $\leftarrow$ solution in P
16:         Update solution by invoking low level search for $a_i$
17:         cost = sum of individual cost for new node's solution
18:         if new node's cost $< \infty$ then
19:             Validate the paths in new node until the time that longest path ends // Store conflict count and first conflict
20:             if new node has no conflict then return path list in new node // new node has a solution
21:             end if
22:             insert new node to open list
23:         end if
24:     end for
25: end while
```

CBS. Conflict count is used for selecting nodes with the minimum number of conflict from the high-level search tree. So, when examining a node, even if a conflict is found the process is continued until the time that the longest path ends. In this way, conflict count for every node is found but as CBS algorithm, the first conflict is used for creating new nodes (Algorithm 3, line 19).

The last modification is changing the order of validating and pushing processes. CBS algorithm pushes newly created nodes to the open list without checking whether a conflict exists or not (Algorithm 2, lines 19-20). The nodes with the lowest costs are fetched from the open list and are examined until a conflict-free node is found (Algorithm 2, lines 7,8). This way, CBS guarantees optimality. Suboptimal CBS validates nodes when just they are created and before pushing them to open list (Algorithm 3, line 19). This modification is needed because suboptimal CBS uses conflict count while selecting nodes from the open list and conflict count is calculated in the validation process. By this modification, If a conflict-free node is encountered, the solution is returned without any further exploration. So the time consumption to find a solution is decreased highly with this modification.

By these modifications, a complete and time-efficient algorithm is proposed. Because the S-CBS algorithm searches the high-level tree until a conflict-free node is found, the proposed method guarantees completeness. Optimality is sacrificed by not fetching the best node. The experimental results show that the modified version of CBS has advantages over other suboptimal algorithms in terms of success rates while being competitive in time.

4.2 Example 1: S-CBS Outperforms CBS

Figure 4.1 [4] contains a sample MAPF problem. In this example, taxi 1 and taxi 2 should pick up passengers 1 and 2, respectively. A taxi can move in four directions (left, right, up, down) or stands still avoiding obstacles with a unit cost. CBS can not efficiently solve this problem instance because of the open area in the middle of the graph.

Figure 4.2 show the high-level tree constructed by CBS algorithm. Calculations start

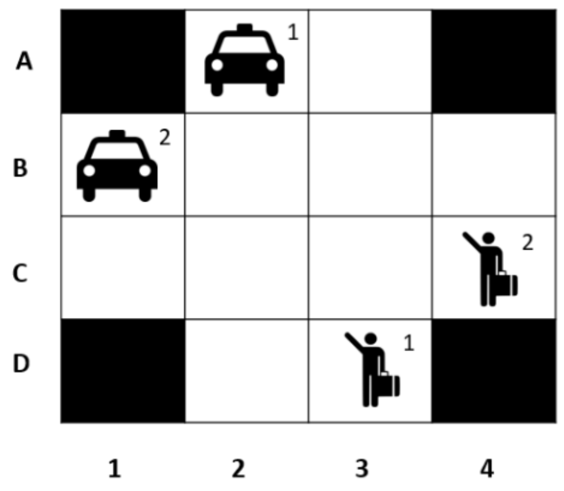

Figure 4.1: Example on Suboptimal CBS Outperforms CBS

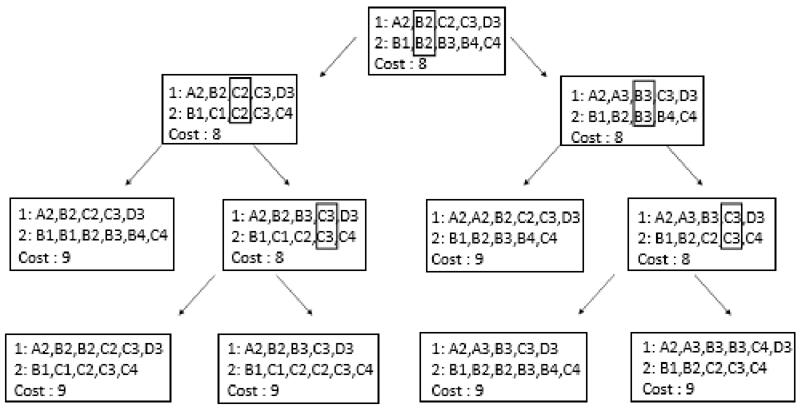

Figure 4.2: High level tree for Example in Figure 3.1

with a root node which has a total cost of 8. In the root node, CBS finds solutions for two agents by using a low-level search. As seen in Figure 4.2, both agents occupy the same location B2 so being in that position at time 1 should be constrained and new high-level nodes should be created. The root node is popped from the open list and is validated whether a conflict exists or not (Algorithm 2 lines 7,8). Left child is created for forbidding agent 2 to locate on B2 at time 1 and the right child is created for forbidding agent 1 to locate on B2 at time 1. Because the costs of two nodes in the open list are the same, the left child is popped. This child is validated and a conflict in place C2 at time 2 is found. For this conflict again two children nodes are created with costs 9 and 8. At this point, there are three nodes in the open list, one with cost 9 and two with cost 8. CBS selects one of the nodes with cost 8 and this process goes until there is no node with cost 8 in the open list. Then a node with cost 9 is popped from open list and validated. Because the node with cost 9 is valid, process halts and returns solutions. To solve this problem instance, CBS has to do 5 high-level searches, one for every node with cost 8 and 12 low-level searches, two for the root node and one for every child node.

For the same problem instance, S-CBS validates the root node and creates first two child node of the tree in the same way with CBS. While validation, S-CBS counts conflicts for these children. The right child of the root node is popped because it has less conflict. Two children nodes are created with costs 9 and 8. S-CBS validates these nodes before pushed into the open list. Because the node with cost 9 is a valid solution, further exploration is not needed. S-CBS solves the problem with a total of 2 high-level searches and 5 low-level searches. For this problem instance, both solutions have a cost of 9. This case shows that S-CBS has advantages in graphs with open spaces compared to CBS. Both algorithms create solutions that have the same cost and clearly S-CBS solves the instance in less time.

4.3 Example 2: S-CBS Outperform CO-WHCA*

The prominent feature of the S-CBS over other suboptimal solutions is its completeness. Figure 4.3 shows a simple example where CO-WHCA* does not find a solution but Sub-Optimal CBS does. This instance shows a case that one agent's goal position

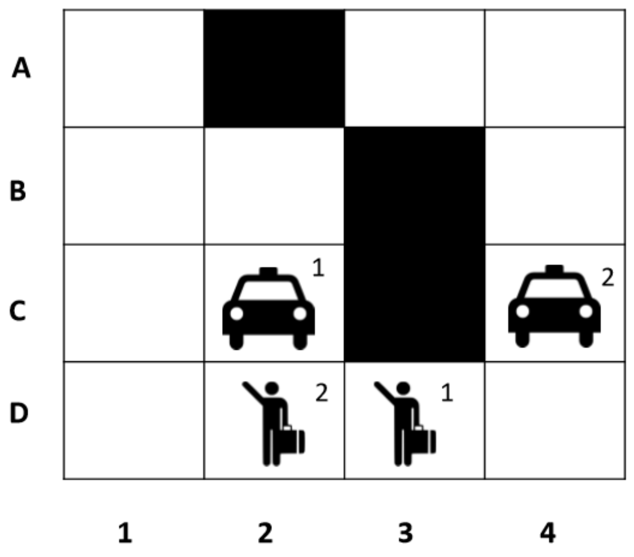

Figure 4.3: Example on Sub-Optimal CBS Outperforms Other Sub-Optimal Solutions

is on the way of the other agent. Agent (Taxi) 1 starts from position C2 and tries to reach D3 and agent (taxi) 2 starts from position C4 and tries to reach to D2. Individual optimal solutions of agent 1 and agent 2 conflicting on position D3. Agent 1 should wait a time step before reaching the goal position. When two agents conflict on D3 at time 2, S-CBS creates two new CT nodes. One for forbidding agent 1 to locate on D3 at time 2 and one for forbidding agent 2 to locate on D3 at time 2. Then tries both cases till a solution found. In this case, the node that agent 1 is forbidden reaches a solution. For the same problem instance, CO-WHCA* is affected by the prioritization rule directly. CO-WHCA* starts with finding individual paths of both agents and finds a conflict on D3 at time 2 same as S-CBS. But in this case, CO-WHCA* selects one agent to fill the reservation table. When agent 1 is selected to reserve D3 for time 1 and reaches the goal without any wait action, there is no other way for agent 2 to complete its path. Because CO-WHCA* uses random prioritization 1, fails for this sample and for the instances that have similar issues.

CHAPTER 5

EVALUATION AND RESULTS

In this chapter, a performance evaluation of the proposed solution will be given. An algorithm superior in every aspect of the problem domain does not exist, so, methods were evaluated with a wide range of test cases. In this way, the advantages of different algorithms were specified. Experiments were conducted with a varying number of agents on maps with different sizes, densities, and characteristics. Magnitudes of the numerical results of CBS affected the visual representations of the results of other suboptimal algorithms badly, so, we presented results of CBS separated from suboptimal methods. All algorithms were implemented using C++ with Visual Studio IDE. The PC running on 64-bit Windows 10 operating system with specifications 2.4 GHz i7 processor, 8 GB RAM is used for conducting all of the test cases.

First, the generation of maps and test cases will be explained in detail. Second, performances of S-CBS and other algorithms will be presented.

5.1 Input Generation

Two different sets of maps were used for the experiments. The first one is grid maps that are randomly generated and the other is the Dragon Age Origins (DAO) game maps [21]. For the randomly generated maps, obstacle settings was used as a parameter. Tree different obstacle ratios were used; obstacle-free, %20 and %50 obstacle. For DAO, ost003d, den520d and brc202d maps were used. These maps are widely used in the field [4][8] and with their different sizes and characteristics, they are convenient for comparing MAPF algorithms. These characteristics will be given in the results and discussion section in detail.

An input for a MAPF problem instance contains a map and a number of agents with their unique start and goal positions. When start and goal positions are generated randomly, the agent set can contain agent groups that their paths completely independent with each other. The number of independent groups affects the evaluation. To prevent these effects, all agents' paths should be dependent on each other. To create agents that are depended on each other, the method proposed in [5] was used. A number of agents are created and single-agent paths are calculated by ignoring other agents. Then, these paths are evaluated by checking whether there are conflicts or not. Agents that their paths are conflicted with each other are grouped. If the number of agents in a group reaches a number that will be used in experiments, using this agent group, a MAPF instance is created.

For every map configuration, test cases with a different number of agents were created. For every agent count, 10 different MAPF instances were generated. The average of the results for these 10 instances was represented.

5.2 Results and Discussion

In this section, Experiment results will be given. We used three different performance metrics that are *completeness*, *optimality* and *time efficiency*.

- Completeness : We represented completeness metric as the ratio of instances solved by the algorithm. Test cases for all map configurations contain 10 solvable inputs. For every agent count, obstacle ratios and maps, completeness ratios were presented.

- Optimality : A solution for a MAPF problem instance includes paths from start positions to goal positions for every agent. An optimum solution has the minimum total path cost possible. In our experiments, every move and wait action count unit cost. For an agent that reached to the goal position and never moved again, wait actions were not added in the path cost. Except for the FAR algorithm, it is allowed that agents follow each other. An agent can move to a position that is left by another agent at the same time. FAR algorithm forbids

	CBS		S-CBS	
Number of Agents	Time(ms)	Path Length	Time(ms)	Path Length
3	0	17	0	18
4	1	25	0	26
5	3	31	1	32
6	33	33	1	48
7	17	47	1	49
8	52	52	3	64
9	180	65	3	68
10	63	75	4	79
11	270	79	4	82
12	625	85	3	87
13	674	83	2	86
14	928	98	5	101
15	1398	107	8	111
16	992	107	7	110
17	2294	117	9	120

Table 5.1: Running times and total path lengths on 8x8 obstacle free grid map.

this kind of action in the definition. The cost of a solution is the sum of the path costs for all agents.

- Time Efficiency : Time efficiency is an important metric because real-life applications require real-time solutions. A MAPF algorithm is time-efficient compared to another algorithm if it consumes less time to reach a solution. Suboptimal methods can be claimed to offer real-time solutions but this is not the case for optimal algorithms. Because CBS spent long times for reaching a solution especially in DAO maps, we used five minutes of time limit.

5.2.1 Experimental Results on 8x8 Grid Maps

5.2.1.1 Comparison with CBS

Tables 5.1, 5.2 and 5.3 shows the experimental results for S-CBS and CBS on 8x8 grid maps. Both methods are complete and solved all instances of test cases, so completeness results were not presented. In the case of time efficiency, results show that S-CBS has a huge advantage. For all of the map configurations, S-CBS consumed

	CBS		S-CBS	
Number of Agents	Time(ms)	Path Length	Time(ms)	Path Length
3	0	22	0	22
4	1	28	0	29
5	3	34	1	36
6	4	42	1	43
7	3	50	1	52
8	38	63	1	67
9	24	58	2	64
10	192	75	3	79
11	279	84	3	91
12	338	82	4	86
13	477	101	4	106
14	642	96	4	104
15	294	101	4	105
16	418	108	2	113
17	469	115	4	123

Table 5.2: Running times and total path lengths on 8x8 grid map with %20 obstacle.

much less time than CBS. The gap between time consumption values became smaller for the maps with more number of obstacles. For example, in obstacle-free maps with 17 agents, CBS consumed approximately 250 times more time than S-CBS, but in maps with %20 obstacle and 17 agents, the difference was approximately 120 times. The increase in agent counts causes both algorithms to consume more time. Results show that increase ratios of time consumption values for CBS are much more than S-CBS. S-CBS created a little longer paths for all map configurations. We see the obstacle ratio has not distinct effect on the difference of path lengths created by CBS and S-CBS. The results show that the proposed method achieves the aim of creating a complete, time-efficient but suboptimal algorithm.

5.2.1.2 Comparison with CO-WHCA*, FAR, Push and Swap

We run S-SBC and 3 competitive suboptimal algorithms; CO-WHCA*, FAR and Push and Swap on different maps.

	CBS		S-CBS	
Number of Agents	Time(ms)	Path Length	Time(ms)	Path Length
3	0	17	0	18
4	1	28	0	28
5	5	32	1	33
6	12	45	2	45
7	13	56	1	56
8	56	52	2	64
9	606	61	70	65

Table 5.3: Running times and total path lengths on 8x8 grid map with %50 obstacle.

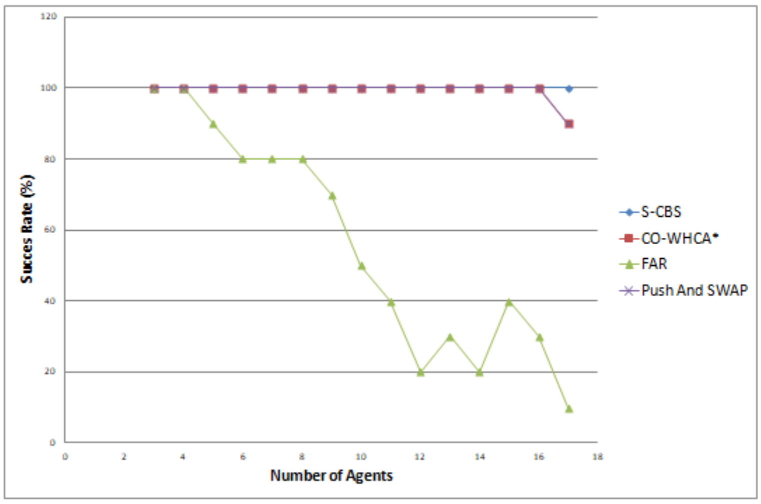

Figure 5.1: 8x8 obstacle free grid map, number of agent v.s. success rates

Completeness

Figures 5.1, 5.2 and 5.3 show success rates for tests conducted on 8x8 grid maps with obstacle free, %20 obstacle ratio and %50 obstacle ratio configurations respectively. Because algorithms other than S-CBS could not solve any instances, we limit the agent count with 9 for the maps with %50 obstacles. For S-CBS, we see that in all map configurations and for all agent numbers, any of the test cases failed. For CO-WHCA* success rates decrease considerably with the increase of agent counts and obstacle ratios. An increase in the number of agents seems to be less effective on completeness results of the Push and Swap. As in the map configuration with %50 obstacle, higher ratios of obstacles decrease success rates because it makes it challenging to find swap positions. The FAR algorithm affected severely from an

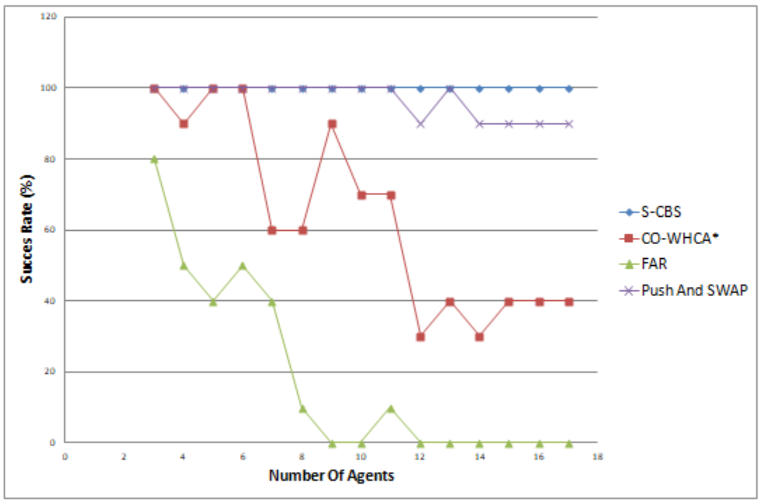

Figure 5.2: 8x8 grid map with %20 obstacle, number of agent v.s. success rates

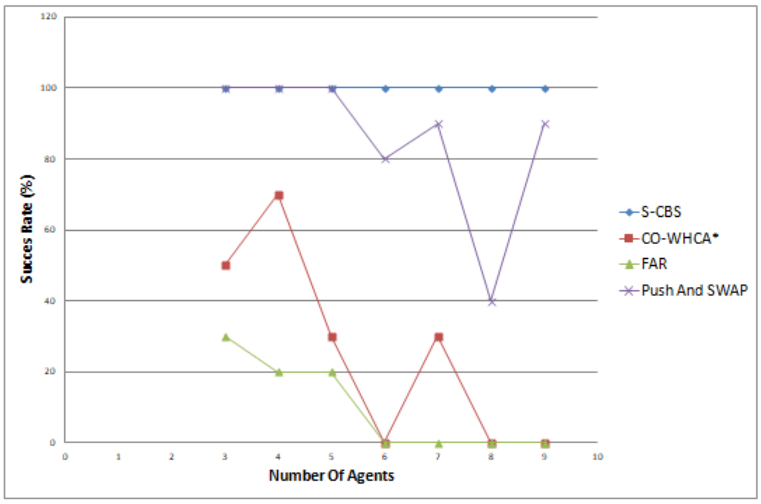

Figure 5.3: 8x8 grid map %50 obstacle, number of agent v.s. success rates

increase in both agent number and obstacle ratios. The results show that S-CBS has advantages over the other algorithms in terms of completeness.

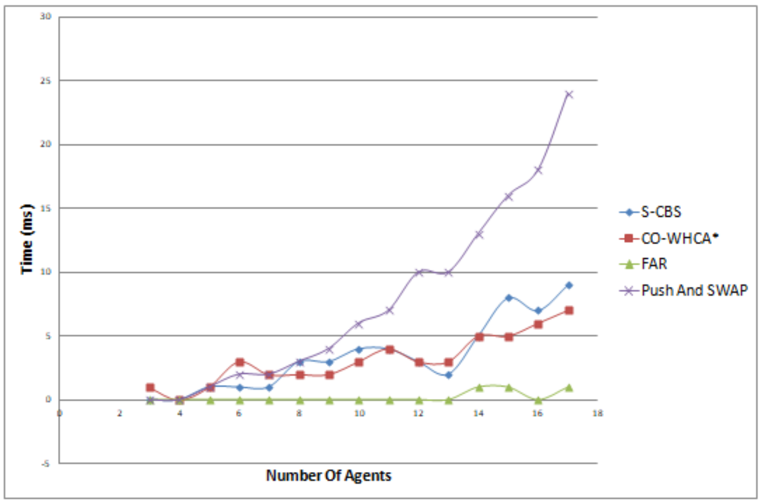

Figure 5.4: 8x8 obstacle free grid map, number of agent v.s. run time

Time Efficiency

The results presented in Figures 5.4, 5.5 and 5.6 show that S- CBS solves test cases consuming similar time with CO-WHCA*. With a simple reservation mechanism, the FAR algorithm recorded better time scores than other suboptimal algorithms. The Push and Swap algorithm consumed more time than other algorithms. The differences in obstacle ratios did not change the time-efficiency relations between algorithms. The results of test cases on %50 maps are affected by the number of instances that a solution found. Because algorithms were able to solve a different number of instances, time consumption values on %50 maps were not distinctive.

Optimality

Figures 5.7, 5.8 and 5.9 show the total path lengths results for suboptimal algorithms. CO-WHCA* and S-CBS produced similar results in terms of solution quality. The FAR algorithm that uses flow annotated graph creates paths like traffic flows, so the solution costs are more than those of search-based algorithms. Push and Swap algorithm calculates agent's paths one by one. This feature causes adding wait actions

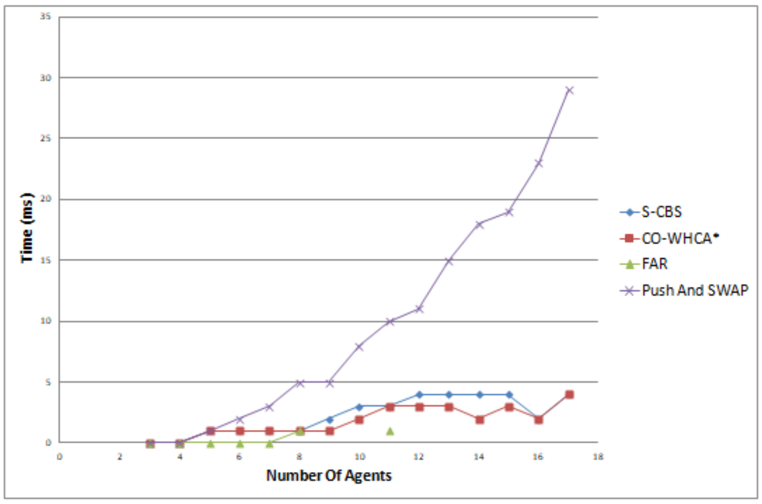

Figure 5.5: 8x8 grid map with %20 obstacle, number of agent v.s. run time

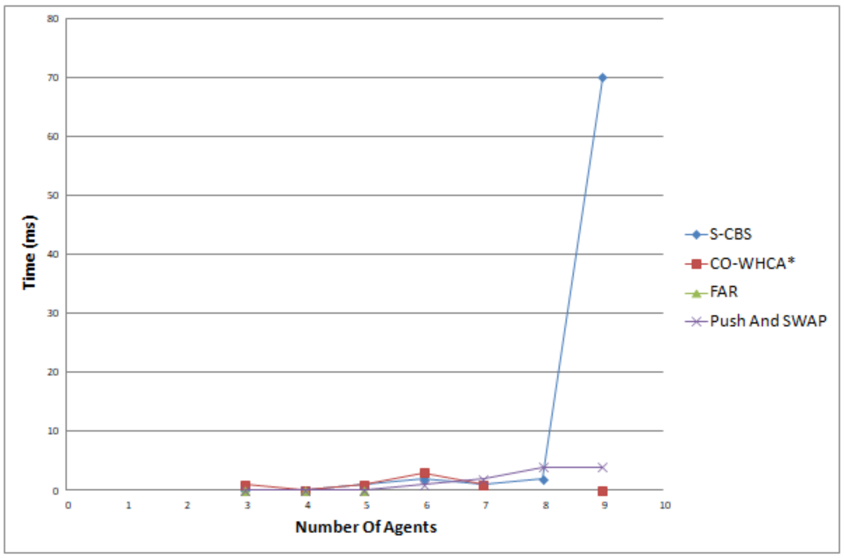

Figure 5.6: 8x8 grid map %50 obstacle, number of agent v.s. run time

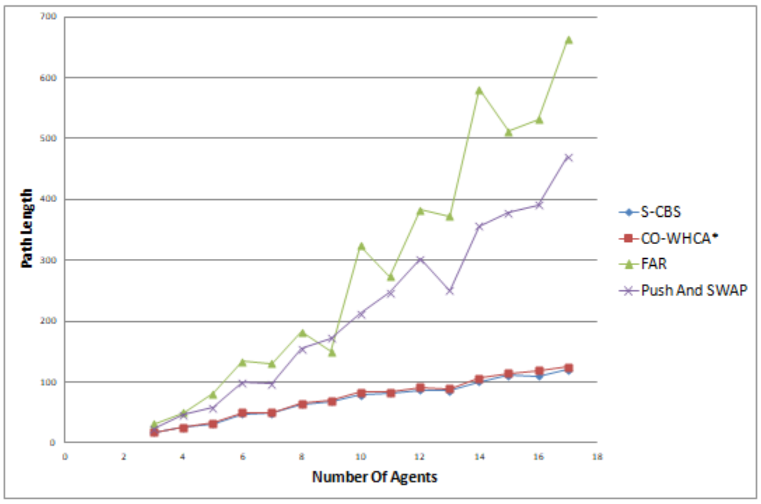

Figure 5.7: 8x8 obstacle free grid map, number of agent v.s. path length

to the agent whose path is not currently calculated, so Push and Swap generated longer paths. As we claimed, the S-CBS algorithm calculate near-optimal solutions for MAPF problem instances.

5.2.2 Experimental Results on Dragon Age: Origin Maps

5.2.2.1 Comparison with CBS

The results generated by CBS for test cases on DAO maps are not presented because the CBS algorithm could not generate a solution for most of the test cases in 5-minute. S-CBS solved almost all of these test cases within time limits. This situation shows that we improved CBS considerably in terms of time-efficiency. For the test cases which were solved by CBS, S-CBS calculated slightly longer paths. However, S-CBS has competitive optimality results compared to other suboptimal algorithms.

5.2.2.2 Comparison with CO-WHCA*, FAR and Push and Swap

Completeness

Figures 5.10, 5.11 and 5.12 show success rate results for the tests conducted on DAO maps ost003d, den520d and brc202d. These maps have sizes of 194x194, 257x257,

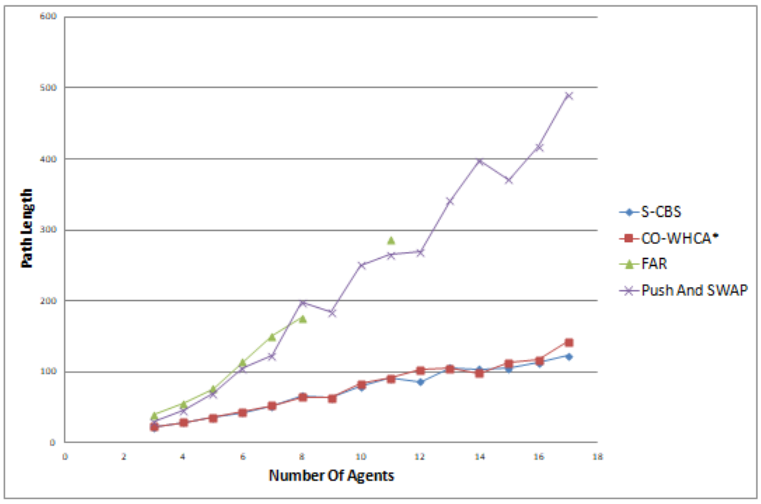

Figure 5.8: 8x8 grid map with %20 obstacle, number of agent v.s. path length

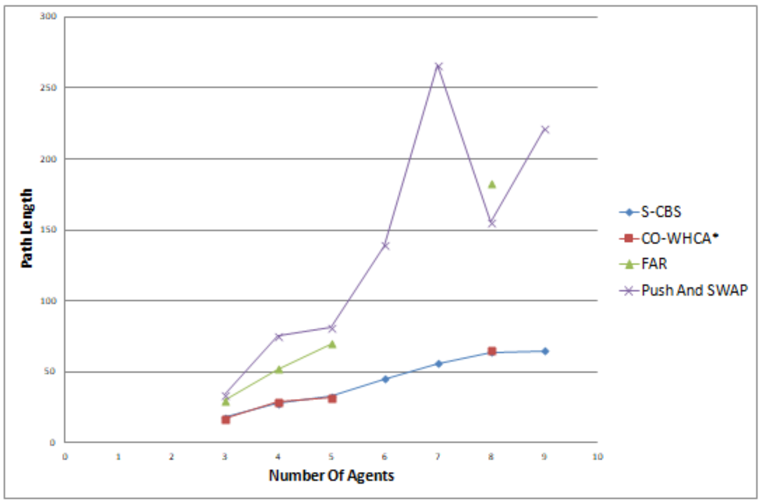

Figure 5.9: 8x8 grid map %50 obstacle, number of agent v.s. path length

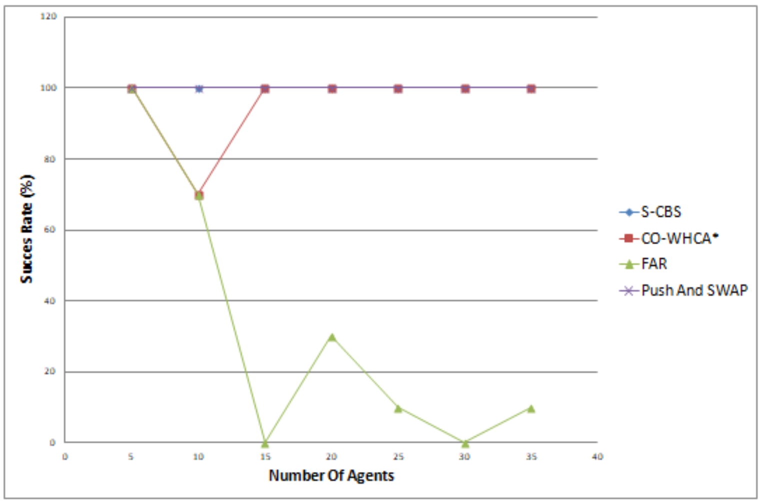

Figure 5.10: Dragon Age:Origins Ost003d map, number of agent v.s. success rates

and 530x530 respectively. CO-WHCA* generated acceptable completeness results for the test cases on the ost003d map, which has open spaces. For the den520d and brc202d, success rates of search-based algorithm decrease because these maps have narrow gates and passageways. The FAR algorithm, indipendent of map sizes and obstacle ratios, was effected badly from the increase in the number of agents. Similar to the test conducted on 8x8 grid maps, FAR achieved the worst completeness results on DAO maps. Push and Swap is the only algorithm that solves all instances of the test cases on DAO maps. Because these maps have big open spaces, Push and Swap was able to swap positions easily. On the brc202d map, S-CBS could not calculate solutions within time limits for test cases with 25 agents. For the maps ost202d and den520d, Suboptimal CBS solved all of the test cases. Suboptimal CBS generated better completeness results except the Push and Swap algorithm.

Time Efficiency

Experimental results for CBS showed that CBS could not calculate a solution in the given time limit for the maps with bigger sizes and high densities of agents. Although S-CBS solved almost all test cases on DAO maps, S-CBS calculated solutions consuming more time then suboptimal solutions. Figures 5.13, 5.14 and 5.15 shows time consumption results on DAO maps. CBS performed poorly on test cases with a high

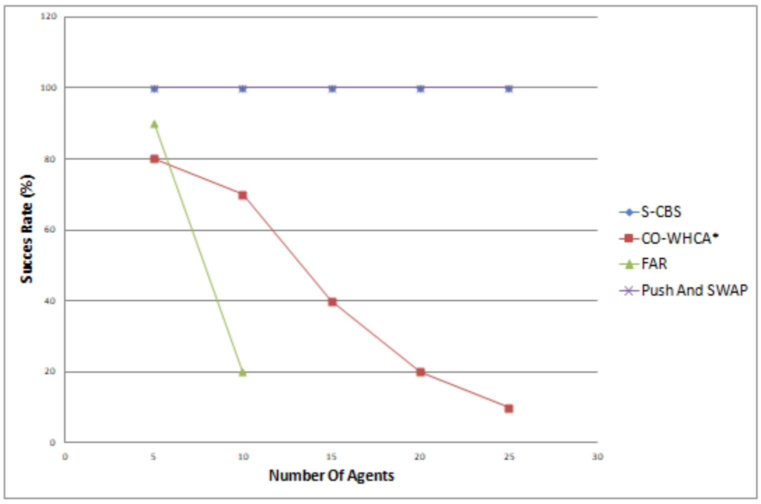

Figure 5.11: Dragon Age:Origins Den520d map, number of agent v.s. success rates

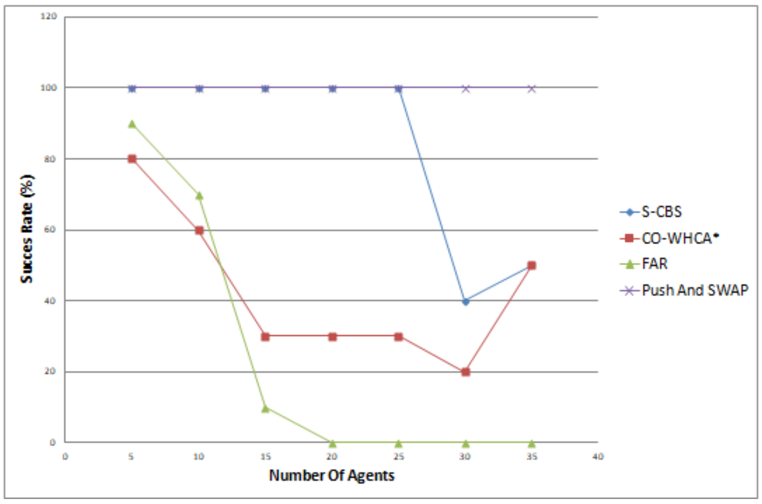

Figure 5.12: Dragon Age:Origins Brc202d map, number of agent v.s. success rates

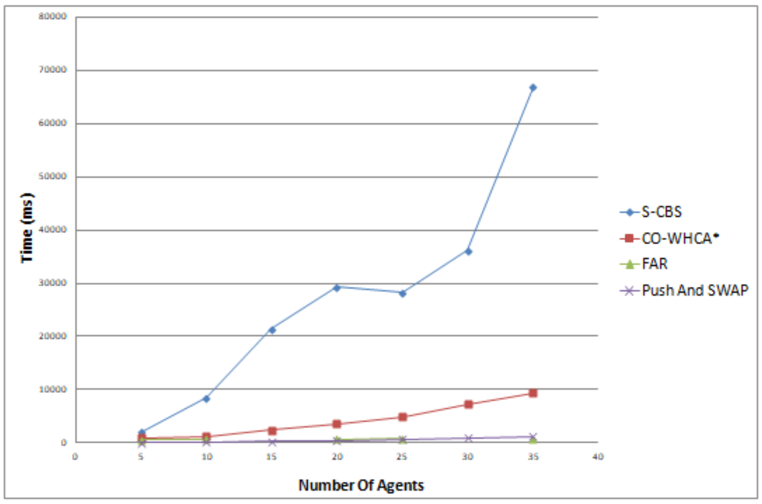

Figure 5.13: Dragon Age:Origins Ost003d map, number of agent v.s. time

number of agent. CO-WHCA* achieved superiority on S-CBS. Push and Swap and FAR reached solutions consuming less time than search-based algorithms.

Optimality

Tests conducted on DAO maps showed that changing map sizes is not affected the relations between methods in terms of solution qualities. Figures 5.16, 5.17 and 5.18 shows optimality results. As in the 8x8 maps S-CBS and CO-WHCA* generated similar total path lengths. Compared to search-based algorithms, Push and Swap performed poorer performances. FAR creates the biggest total path lengths on every map and agent count configurations. As we claimed, the S-CBS algorithm achieved competitive optimality results on DAO maps.

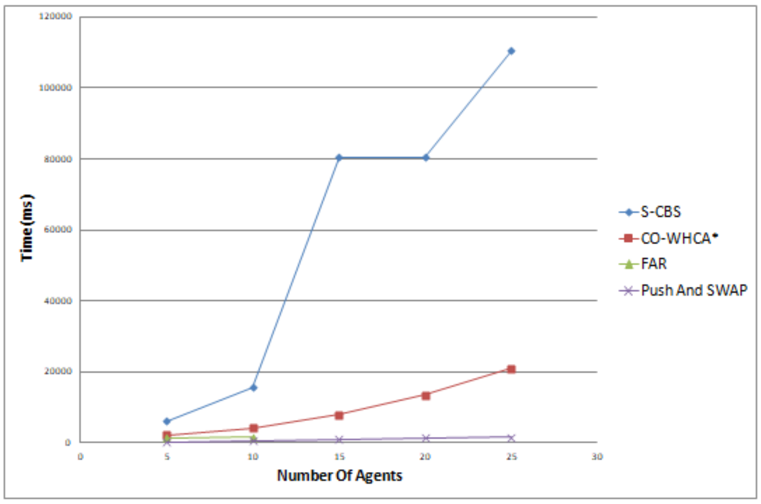

Figure 5.14: Dragon Age:Origins Den520d map, number of agent v.s. time

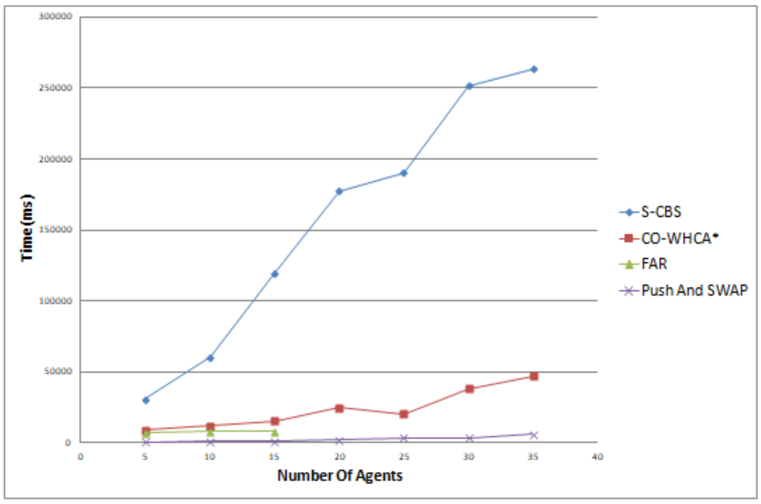

Figure 5.15: Dragon Age:Origins Brc202d map, number of agent v.s. time

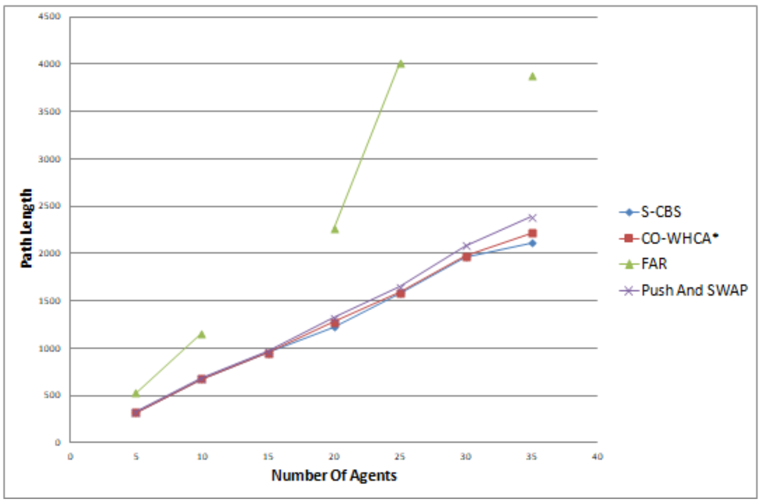

Figure 5.16: Dragon Age:Origins Ost003d map, number of agent v.s. path length

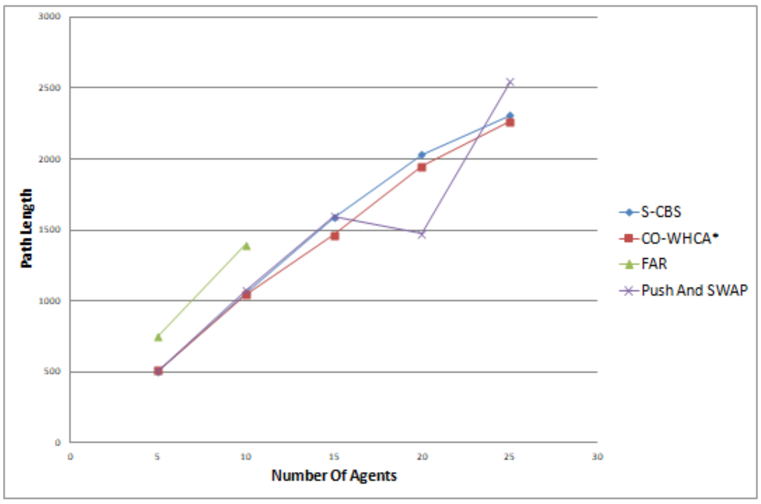

Figure 5.17: Dragon Age:Origins Den520d map, number of agent v.s. path length

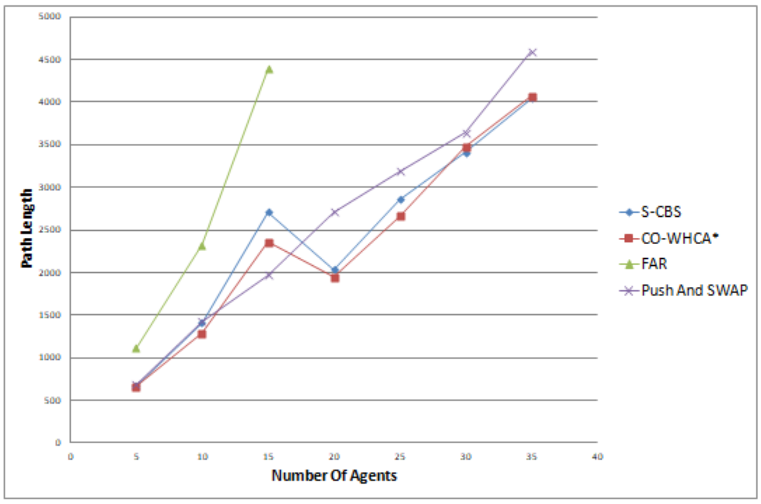

Figure 5.18: Dragon Age:Origins Brc202d map, number of agent v.s. path length

CHAPTER 6

CONCLUSION AND FUTURE WORK

Existing research on MAPF showed that optimal and complete solutions could not be computed in real-time as the problem size grows. Besides, suboptimal methods do not guarantee completeness. Completeness is an essential requirement for the real-life applications. Motivated from these requirements, we proposed a MAPF algorithm which is complete, time-efficient, and suboptimal by modifying an optimal algorithm.

Being one of the best optimal algorithm, CBS creates a high-level tree that contains nodes which are created for conflicts of the agents' paths. By searching high-level tree, CBS finds optimal conflict-free paths for every agent. Because CBS searches for an optimum solution, it considers nodes with least costs first. So, reaching a solution takes longer than suboptimal algorithms. In this thesis work, a suboptimal version of CBS algorithm named S-CBS is developed. S-CBS algorithm is based on the truth that a high-level tree node of CBS must be conflict-free to be a solution. Instead of the node with the least cost, S-CBS searches for the node with the least number of conflicts. This modification is made by counting future conflicts that a node will produce. This count is used for fetching a node from the high-level tree. Moreover, pushing and validating order is changed for reaching a solution immediately. Instead of searching for the best solution, the algorithm returns when a solution is found. By these modifications, S-CBS becomes a suboptimal, complete and time-efficient algorithm.

Because the algorithm proposed is suboptimal, we compared S-CBS with other suboptimal methods. Experiments were conducted on suboptimal algorithms. We evaluated two search-based algorithms (WHCA* and WHCA*P), a rule-based algorithm (Push and Swap), and a hybrid algorithm (FAR). Experiments were conducted with varying number of agents and obstacle ratios on maps with different sizes. Optimality, completeness, and time efficiency metrics were used for evaluation criteria. The results showed that S-CBS algorithm is competitive in optimality and time efficiency and has significant superiority in terms of completeness, especially in maps with small sizes. In these maps, it is seen that S-CBS created smaller path lengths than Push and Swap which achieved the closest completeness results with S-CBS. In addition to experiments on small-sized maps, DAO maps which have bigger sizes were used. In these maps, S-CBS produced competitive completeness and optimality results. Although considerably better than CBS, S-CBS spent more time than other suboptimal algorithms to reach a solution.

In this work, we achieved the aim of creating a suboptimal but complete and time-efficient algorithm. Decreasing time consumptions in bigger sized maps can be one of the works that will be considered in the future. For this, S-CBS can be converted to an online algorithm. In other words, a time frame can be defined, and the calculations can be done within this time frame. After the solution found for this time frame, next time frames can be processed. In this case, for preserving completeness, proper modifications should be done. With this work, we showed that optimal and complete algorithms could be modified to achieve different performance goals. Another direction can be identifying different algorithms that can be modified by using this point of view and comparing them with each other.

REFERENCES

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

[2] J. E. Hopcroft, J. T. Schwartz, and M. Sharir, “On the complexity of motion planning for multiple independent objects; pspace-hardness of the" warehouseman’s problem",” *The International Journal of Robotics Research*, vol. 3, no. 4, pp. 76–88, 1984.

[3] R. A. Hearn and E. D. Demaine, “Pspace-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation,” *Theoretical Computer Science*, vol. 343, no. 1-2, pp. 72–96, 2005.

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

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

[6] A. Zelinsky, “A mobile robot exploration algorithm,” *IEEE Transactions on Robotics and Automation*, vol. 8, no. 6, pp. 707–717, 1992.

[7] D. Silver, “Cooperative pathfinding.,” *AIIDE*, vol. 1, pp. 117–122, 2005.

[8] Z. Bnaya and A. Felner, “Conflict-oriented windowed hierarchical cooperative a*,” in *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pp. 3743–3748, IEEE, 2014.

[9] B. de Wilde, A. W. ter Mors, and C. Witteveen, “Push and rotate: cooperative multi-agent path planning,” in *Proceedings of the 2013 international conference*

on Autonomous agents and multi-agent systems, pp. 87–94, International Foundation for Autonomous Agents and Multiagent Systems, 2013.

[10] M. M. Khorshid, R. C. Holte, and N. R. Sturtevant, "A polynomial-time algorithm for non-optimal multi-agent pathfinding," in *Fourth Annual Symposium on Combinatorial Search*, 2011.

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

[12] R. Luna and K. E. Bekris, "Efficient and complete centralized multi-robot path planning," in *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pp. 3268–3275, IEEE, 2011.

[13] R. J. Luna and K. E. Bekris, "Push and swap: Fast cooperative path-finding with completeness guarantees," in *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

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

[15] M. Ryan, "Constraint-based multi-robot path planning," in *2010 IEEE International Conference on Robotics and Automation*, pp. 922–928, IEEE, 2010.

[16] R. Jansen and N. Sturtevant, "A new approach to cooperative pathfinding," in *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 3*, pp. 1401–1404, International Foundation for Autonomous Agents and Multiagent Systems, 2008.

[17] K.-H. C. Wang, A. Botea, *et al.*, "Fast and memory-efficient multi-agent pathfinding.," in *ICAPS*, pp. 380–387, 2008.

[18] T. S. Standley, "Finding optimal solutions to cooperative pathfinding problems.," in *AAAI*, vol. 1, pp. 28–29, 2010.

[19] A. Felner, M. Goldenberg, G. Sharon, R. Stern, T. Beja, N. Sturtevant, J. Schaeffer, and R. Holte, "Partial-expansion a* with selective node generation," in *Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

[20] M. Goldenberg, A. Felner, R. Stern, G. Sharon, N. Sturtevant, R. C. Holte, and J. Schaeffer, “Enhanced partial expansion a,” *Journal of Artificial Intelligence Research*, vol. 50, pp. 141–187, 2014.

[21] N. Sturtevant, “Benchmarks for grid-based pathfinding,” *Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 2, pp. 144 – 148, 2012.