Multi-agent path finding using time-extended graphs using auctions

Simanta Barman
University of Minnesota
Twin Cities, United States
barma017@umn.edu

Angel Sylvester
University of Minnesota
Twin Cities, United States
sylve057@umn.edu

Ebasa Temesgen
University of Minnesota
Twin Cities, United States
temes021@umn.edu

Michael W. Levin
University of Minnesota
Twin Cities, United States
mlevin@umn.edu

Maria Gini
University of Minnesota
Twin Cities, United States
gini@umn.edu

ABSTRACT

The multi-agent path finding (MAPF) problem considers the problem of assigning a set of agents to a set of targets and finding collision free paths to the targets. Even for a discrete environment, homogeneous agents, and no kinematic constraints the problem is proven to be NP-hard for minimization of certain objectives. Considering the complexity of the problem some previous research has developed sub-optimal solution algorithms to reduce runtime. We approach the MAPF problem in a similar fashion focusing on obtaining guaranteed collision free paths for each agent. Our approach relies on using auction-like mechanisms on a time extended graph of the environment. We present detailed description of the construction of the time extended graph, a label setting algorithm to find the shortest paths in that graph and an auction-like mechanism to ensure collision free paths for each agent. We also prove that we can exploit several properties of the time extended graph to reduce both the space complexity of the time extended graph and the time complexity of finding shortest paths in that graph. Along with implementation details, we present numerical results on common metrics to evaluate the quality of the solutions for several problem instances with changes in the auction mechanism. Our method is a generalization of the cooperative A* algorithm if we use a heuristic in our label setting algorithm which has been used to tackle the MAPF problem for large problem instances.

KEYWORDS

Path planning, multi-agent systems

ACM Reference Format:
Simanta Barman, Angel Sylvester, Ebasa Temesgen, Michael W. Levin, and Maria Gini. 2024. Multi-agent path finding using time-extended graphs using auctions. In *ACM Conference, Washington, DC, USA, July 2017,* IFAAMAS, 9 pages.

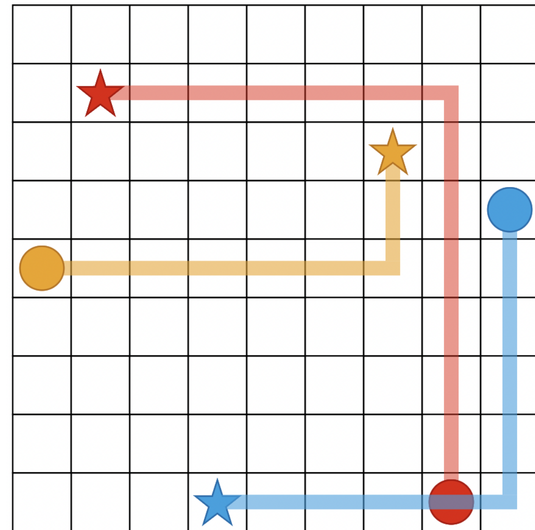


Figure 1: A MAPF problem instance with 3 agents (circles) and 3 targets (stars). (Source: wiki/Multi-agent_pathfinding)

1 INTRODUCTION

Multi-agent path finding (MAPF) [15] is an important problem in several domains including Artificial Intelligence (AI), robotics, computer science, and operations research. The general MAPF problem has multiple agents in a shared environment with each agent having a unique start location. The goal for each agent is to find the best paths to their destinations while avoiding collisions with other agents or obstacles in the environment. For a single agent this problem just reduces to finding the shortest path in the environment. However, multiple agents' presence in the shared environment and navigating around each other requires better planning and coordination. Treating the agents as dynamic obstacles causes the environment to change each time an agent moves. This means that the shortest path found in the static environment is not sufficient to solve the MAPF problem. Furthermore, at each time step the paths need to be recomputed considering the changes in the environment to avoid collisions. This rerouting can quickly become computationally expensive, which makes the problem difficult to solve for large problem instances [10].

ACM Conference, , July 2017, Washington, DC, USA. © 2024 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). This work is licenced under the Creative Commons Attribution 4.0 International (CC-BY 4.0) licence.
…$ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
…$15.00

This problem has many applications in the real world [1, 9, 16]. MAPF is one of the most important problems in robotics and autonomous systems where coordination and navigation of multiple robots is required. Scenarios like warehouse automation, autonomous traffic intersection management, self-driving cars, drone fleets and anything that requires collaborative robotics, where multiple agents need to operate concurrently in a shared environment, requires solving this problem [3, 7]. In transportation systems solutions to MAPF can be used to optimize traffic routing, manage traffic flows and prevent congestion in a system optimal way. Furthermore, ride sharing services like Uber, Lyft etc. can use MAPF to dispatch and route vehicles to maximize service coverage in an area, minimize wait times and maximize profits [4]. Supply chain operations like moving goods within and between warehouses can also benefit from MAPF solutions. Several other uses of MAPF include planetary surface exploration, disaster response, search and rescue, air traffic control etc.

Companies with warehouse automation needs are particularly interested in this problem because of the potential savings a collision free MAPF solution can provide. Automating warehouses reduces the need for human labor just to move items in the fulfillment centers. Performing the task with robots frees up human labor for more complex tasks. The deployment of robots for these tasks also reduces the risk of injury to human workers, reduces order fulfillment time resulting in millions of dollars in savings for companies. Companies like Amazon, Walmart, and Target are already using robots in their warehouses and are heavily investing in this technology [11].

Along with continuous environment other important practical considerations are heterogeneity of agents and kinematic constraints which make the problem even more difficult. To simplify these problems the environment is usually assumed to be discrete. However, even with discrete environments most variants of MAPF have been proven to be NP-hard and thus are computationally intractable to solve to optimality when the problem instances are large [18]. The main objective of this project is to explore whether an extension and optimization of the cooperative MAPF algorithm is tractable to solve the MAPF problem for larger problem instances while guaranteeing collision free paths for each robot.

2 RELATED WORK

Important considerations for practical implementations of MAPF solutions include continuous environment, heterogeneous agents, agent malfunction, unexpected obstacles and being able to resolve other unforeseen circumstances [6, 20]. In an effort to simplify the problem the environment is assumed to be discrete. However, even with that simplification most variants of MAPF are NP-hard problems and thus computationally intractable to solve to optimality for large problem instances. Optimality can however be defined differently based on the context of the problem. Optimality is usually defined as minimizing the maximum time required to complete all the tasks for all the agents (makespan) or minimizing the total time required to complete all the tasks or reach the destinations. Minimization of the total traveled distance, energy expenditure, etc. can also be considered while measuring optimality. Existing MAPF methods use a variety of techniques like reductions to problems from satisfiability, mixed integer linear programming, or answer set programming, all of which are computationally intractable for problems of large instances where the number of agents and the environment size are large. Existing methods include extensions of the A^* algorithm [14], increasing cost tree search [13], conflict-based search [12], constraint programming [2] and other.

The multi-agent path finding (MAPF) problem is a fascinating problem with many applications in robotics, logistics, and transportation [8]. The proposed research will advance the state of the art in MAPF and in the field of AI. Different variants of this problem exist, but a general solution method may not be sufficiently efficient for all variants. For example, if the environment is continuous the collision detection problem can be solved differently (e.g., in real-time as in [5]). By projecting a ray in time from each robot to their goals and checking for intersections, collisions can be detected and the trajectories of the robots changed accordingly. This time-to-collision (TTC) method is more efficient but requires a continuous environment. The proposed method in this paper is similar, however, unlike a TTC force based method, this proposed method can be applied to discrete environments.

Since most MAPF problem variants for discrete environments are NP-hard, it is important to develop and test different solution methods. Adding other constraints, such as limitations on the energy usage for each robot, and considering their kinematic constraints, such as vertical movements of the robots in a 3D environment, would make the problem even more challenging but applicable to a larger set of problems. All these variants of the problem are of great interest to the robotics field and the industry. Furthermore, another interesting research direction is to study how communication between robots and learning-based techniques can be used to improve the performance of the solution methods. However, first, we need an efficient and guaranteed collision avoidance method. This project will explore that direction.

3 METHODOLOGY

3.1 Problem Statement

Consider the problem of navigating a team of agents represented by the set R of size $|R|$ from some initial positions to their destinations in a grid. The agents have already been assigned their destinations on a one-to-one basis. After each robot reaches its destination, it will be assigned a new destination automatically. The problem is to find a collision-free path for each agent from its current position to its destination. While collision avoidance is an important constraint, the main objective is to plan paths for the agents so that they can reach their destinations as soon and efficiently as possible. The grid-world can have static obstacles that the agents need to consider while planning their paths. The agents are restricted to certain movements.

Define the set of actions for each agent as $A = \{\uparrow, \downarrow, \rightarrow, \leftarrow, W\}$ where the straight arrows represent the actions that move an agent to its neighboring cell. W is the wait action at the current cell. Each of these actions takes 1 time step. Since other robots also move in the environment and no collision is allowed, the robots are considered to be dynamic obstacles to each other.

Let $G = (V, E)$ be the graph representation of the 2D grid-world environment where V is the set of all vertices and E is the set of all

edges. The set of vertices V can further be divided into two sets of vertices:

(1) Usable vertex V_u,
(2) Vertex where a static obstacle is present V_s.

We can reduce the size of the graph by removing the vertices in V_s since they are always impassable. We redefine the set of vertices to have only the usable vertices, $V = V_u$. In other words we remove the impassable vertices from the original environment. Define a function $P : (r, t) \rightarrow v \in V$ that returns the vertex robot r occupies at time t. Vertex collision then can be represented as the existence of a robot pair $(r_1, r_2) \in R \times R$ such that $P(r_1, t) = P(r_2, t)$ at $t \in T$ where T is the set of timesteps until the planning horizon $|T|$. Edge collision can be represented as $P(r_1, t) = P(r_2, t)$ and $P(r_1, t + 1) = P(r_2, t + 1)$ for some robot pair $(r_1, r_2) \in R \times R$ at $t \in T$. Vertex and Edge collisions are not allowed. An action $a \in A$ is considered valid if action a can be performed without any static or dynamic collision. To summarize, the problem is to find a sequence of actions $\pi_r = (a_1, a_2, \ldots, a_n)$ for each agent $r \in R$ such that the actions do not result in any collision and minimizes $\sum_{r \in R} |\pi_r|$ and the agents reach their destinations as efficiently as possible. Here, efficiency means using the least amount of actions. Since each action require 1 time step, the number of actions is the number of timesteps required to reach the destination. More concisely, the optimal solution would minimize the size of the sequence of actions $|\pi_r|$ for each robot $r \in R$ while ensuring that no collision occurs.

3.2 Solution quality and metrics

The metrics used to measure the quality of a solution for this specific variant of Multi-agent Path Finding (MAPF) problem are either the sum (flow-time) or the maximum (makespan) of the number of actions required by each robot to reach their destinations. Since each robot is assigned a new task or destination as soon as they reach their current destination and the number of these destinations are infinite for each robot, using the makespan or flowtime as a metric is not appropriate in their usual definitions. Because the definition of both makespan and flowtime require a fixed number of tasks or destinations for each robot. We will consider the first $|R|$ destinations for the $|R|$ robots as the first MAPF problem instance. In this problem instance, each robot $r \in R$ will have their own unique start and goal vertice. After a robot reaches its destination, the robot will be assigned a new destination which will be considered the next instance of the MAPF problem for which route planning will be done for all the robots.

Definition 3.1 (Makespan). The makespan of a solution is the maximum number of actions required by any agent to reach its destination.

$$\text{makespan} = \max_{r \in R} |\pi_r| \tag{1}$$

Definition 3.2 (Flow-time). The flow-time of a solution is the sum of the number of actions required by each agent to reach its destination.

$$\text{flowtime} = \sum_{r \in R} |\pi_r| \tag{2}$$

Definition 3.3 (Optimal solution). An optimal solution is a combination of collision-free paths for each agent that minimizes the makespan or the flow-time.

Solving this problem to optimality is computationally intractable because of the proven strong NP-hardness of the MAPF problem [17, 19]. The NP-hardness of the MAPF problem is proven in the literature by reducing the boolean satisfiability problem to the MAPF problem. There exist some methods in the literature that can find optimal solutions using mixed integer linear programs or SAT solvers. Custom algorithms like conflict-based search or increasing cost tree search can also produce optimal solutions. They are special cases of constrained optimization programs and can be implemented inside an MILP. However, due to the NP-hardness of the problem, the computational time increases exponentially with the number of agents and the size of the environment. Intuitively, what makes the problem hard is the fact that one agent's path can influence the optimal path of another agent. Therefore, combinations of paths for all the agents need to be evaluated. This requires enumerating all the possible combinations of paths for all the agents which becomes computationally expensive. Even though the current literature can prune the search space of the combination of paths for some paths, the problem is still computationally intractable for large problem instances.

In this paper we will only focus on finding a collision free solution quickly without enumerating all combinations of paths.

3.3 Solution approach

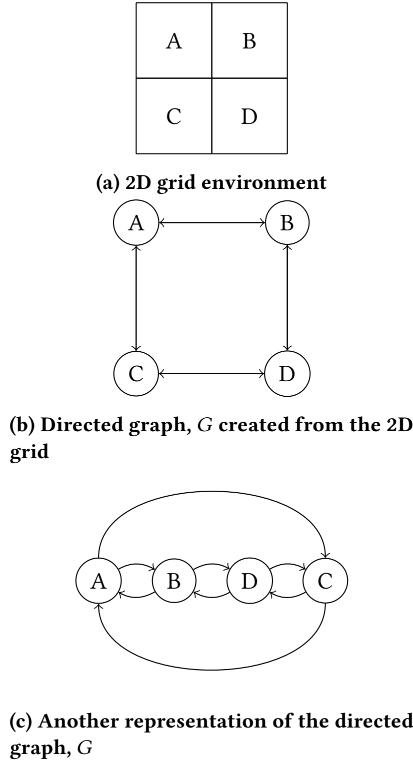
The original graph $G = (V, E)$ can be extended in the time dimension and the new space-time graph can be represented as $G^e = (V^e, E^e)$. Then the time-extended graph G^e can be used to plan paths for the agents. Paths in the time-extended graph can be used to check for collisions. If a collision is detected, then the paths can be replanned to avoid them. Reserving nodes and edges for different agents in the time-extended graph can also guarantee that no collision occurs.

3.4 Construction of the time-extended graph

First $|T|$ copies of the graph will be made. For each copy the vertices will be labeled with a unique label that consists of the vertex id and the timestep. Then the edges will be created between the vertices of the same id in different timesteps.

(1) Create $|T|$ copies of the graph G. Let the set of copies be $G^c = \{G_t \; \forall t \in T\}$.
(2) For each copy $G_t = (V_t, E_t) \in G^c$, label each vertex $v \in V_t$ by (v, t). Construct a set containing all these vertices $\cup_{t=1}^{|T|} V^t$ and call it V^e.
(3) If there is an edge $(u, v) \in E$ in the original graph G, then create an edge (u_t, v_{t+1}) and insert it in E_t to represent using the edge will take 1 time step in the time extended graph. Also, create another edge from the vertex (u, t) to the vertex $(u, t + 1)$ in G_t to represent waiting in the same vertex. Construct a set containing all these edges $\cup_{t=1}^{|T|} E^t$ and call it E^e.

Notice that for a naive implementation of graph storage, the space complexity of the time-extended graph G^e is $O(T(2|V| +$



(a) 2D grid environment

(b) Directed graph, G created from the 2D grid

(c) Another representation of the directed graph, G

Figure 2: Original graph, G

$|E|)$) where the original graph G has $|V|$ vertices and $|E|$ edges and the planning horizon is T. This is because the time extended graph G^e has T copies of the original graph vertices V and edges E. Additionally, each copy has $|V|$ extra edges from a vertex from the previous time step to itself at the next time step. However, this space complexity can be reduced further for our purposes which will be discussed later in the paper.

3.4.1 Useful properties of the time-extended graph.

THEOREM 3.4. *The time-extended graph G^e is a Directed Acyclic Graph (DAG).*

PROOF. Since in the time extended graph G^e, the edges exist only from timestep t to the next timestep $t+1$. Therefore, the graph is only traversable in the direction of time which make the edges directed. Therefore, the time extended graph is a Directed Acyclic Graph (DAG). □

Since the time-extended graph G^e is a DAG and the nodes are connected in the forward direction of time, a natural topological ordering of the nodes based on the time labels exist. This topological ordering can be exploited to find a shortest path without the use of a priority queue to store unvisited nodes.

THEOREM 3.5. *There exists a path from any vertex $(u, t) \in V^e$ to any other vertex $(v, t + m) \in V^e$ in the time-extended graph G^e where m is greater than or equal to the shortest path length from u to v in the original graph G assuming a path from u to v exists in the original graph G.*

PROOF. Consider any path from vertex u to vertex v in the original graph G. Let the path be $P = (u, v_1, v_2, \ldots, v_n, v)$. The described construction procedure for the time-extended graph has all the edge connection only in the time direction ($t \rightarrow t + 1$). Therefore, the equivalent path in the time-extended graph G^e will be $P^e = ((u, t), (v_1, t + 1), (v_2, t + 2), \ldots, (v_n, t + n), (v, t + n + 1))$. In this path no waiting is considered therefore it is the shortest path from (u, t) to $(v, t + n + 1)$. The length of the shortest path from vertex u to v in the original graph G is $n + 1$ which is equal to the length of the shortest path from vertex (u, t) to $(v, t + n + 1)$ in G^e. Since waiting in the same vertex is also allowed, other valid paths from vertex u to v exist. Let waiting at some vertex results in a path in the time extended graph G^e represented by $P^e_w = ((u, t), (v_1, t{+}1), (v_2, t{+}2), \ldots (v_n, t{+}n), (v_n, t{+}n{+}1), (v, t{+}m))$. Here, the vertex v_n is the vertex where waiting occurs which results in a longer path of length m where $m > n + 1$. □

Now since multiple paths from the same vertex to the same vertex exist in the time-extended graph G^e, even if we reserve some path for an agent, other paths can still be used by other agents. However, careful reservation of the paths needs to be performed.

THEOREM 3.6. *A valid topological ordering of the vertices V^e in the time-extended graph G^e can be constructed by using any random ordering of the vertices V from the original graph G while ensuring that the vertices at time $t' > t$ always appear after the vertices in time t in the topological ordering.*

PROOF. At time $t = 0$, the number of incoming edges or in degree of any vertex is 0. Therefore, any random vertex from time $t = 0$ can be inserted in the topologically ordered list. After inserting all the vertices from time $t = 0$, the vertices at $t = 0$ and the outgoing edges from the vertices can be deleted from the graph G^e. This results in all the vertices at time $t = 1$ to have an indegree of 0. Therefore, we can repeat the same process and insert any random vertex from time $t = 1$ until all the vertices from time $t = 1$ are inserted in the topologically ordered list. We can repeat this process to create a valid topological ordering until some time horizon T is reached. □

3.4.2 Shortest paths in the time-extended graph. Since the time-extended graph G^e is a DAG, a topological ordering of the vertices exists based on the time labels. Multiple paths also exist between the same pair of vertices. The source of an agent is given by the vertex $P(r, 0)$ for all $r \in R$. The following algorithm can be used to find the shortest path from the source vertex to all other destination vertices in the time-extended graph G^e in linear time.

Note that Algorithm 1 does not require the use of a priority queue to store the unvisited vertices. Instead, a topological ordering of the vertices is used to update the labels and the predecessors of the vertices. Any vertex that has the time label t has a lower topological ordering than any vertex that has the time label greater than t. Additionally, the vertices at time $t = 0$ does not have any incoming edges. Therefore, any random ordering of the vertices at time $t = 0$ is a valid topological order. However, the vertices at time greater than t need to appear later in the topological ordering. This is ensured in line 9 of the algorithm 1 where all the nodes in one time step are visited before moving to the next time step.

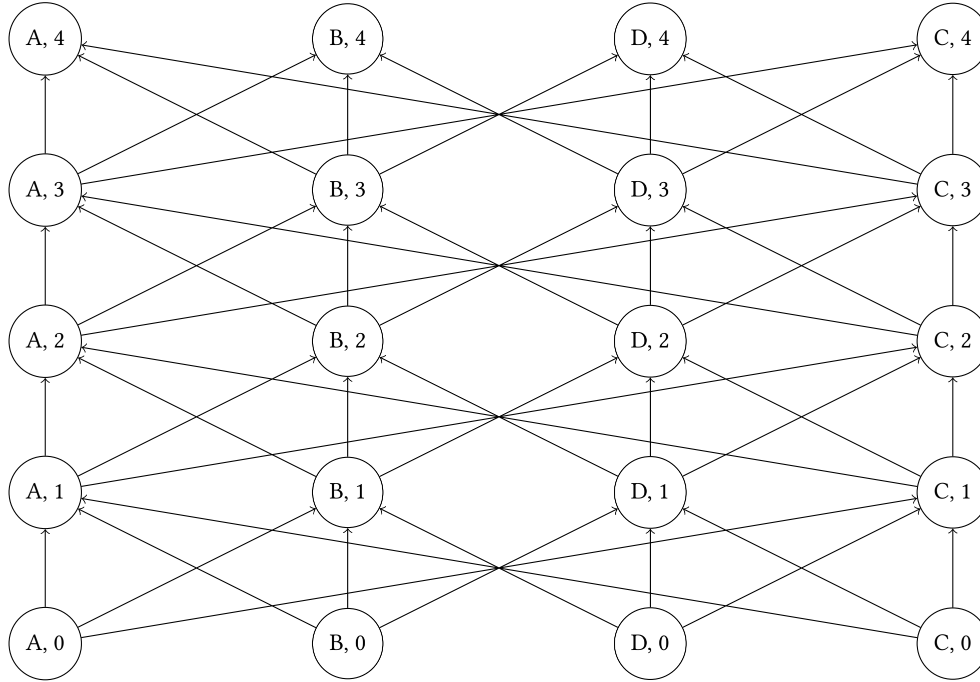


Figure 3: Time-extended graph, G^e

Algorithm 1 Shortest path in the time-extended graph G^e

1: **procedure** LabelSetting($G^e, P(r, 0)$)
2: **for** $(i, t) \in V^e$ **do**
3: $q_{(i,t)} \leftarrow (-1, -1)$ ▷ Predecessor of vertex (i, t)
4: $L_{(i,t)} \leftarrow \infty$ ▷ Label of vertex (i, t)
5: **end for**
6: $L_{P(r,0)} \leftarrow 0$ ▷ Label of the source vertex
7: $V \leftarrow \{v : \forall (v, _) \in V^e\}$ ▷ V = original graph vertices
8: $t \leftarrow 0$
9: **while** $t < T$ **do**
10: **for** $i \in V$ **do**
11: **for** $((i, t), (j, t + 1)) \in$ outgoingEdges(i, t) **do**
12: **if** $L_{(j,t+1)} > L_{(i,t)} + 1$ **then**
13: $L_{(j,t+1)} \leftarrow L_{(i,t)} + 1$
14: $q_{(j,t+1)} \leftarrow (i, t)$
15: **end if**
16: **end for**
17: **end for**
18: Terminate if a path to $(\text{sink}(r), t)$ is found
19: $t \leftarrow t + 1$
20: **end while**
21: **end procedure**

Lines 9 and 10 together iterate over all the time expanded nodes in a topological order only once. For each node $(v, t) \in V^e$ the outgoing edges are visited only once as well to update the labels and the predecessors. Therefore, the time complexity of the label setting algorithm is $O(|V^e| + |E^e|)$ which is linear in the size of the time-extended graph G^e.

Algorithm 2 Constructing the shortest path from the predecessors and the final t

1: **function** ConstructPath($G^e, P(r, 0), t$)
2: $path \leftarrow Vector()$
3: $v \leftarrow \text{sink}(r)$
4: **while** $v \neq P(r, 0)$ **do**
5: $path \leftarrow$ path.pushBack((v, t))
6: $v \leftarrow q_v$
7: $t \leftarrow t - 1$
8: **end while**
9: $path \leftarrow$ path.pushBack($(P(r, 0), t)$)
10: **return** $path.reverse()$
11: **end function**

3.4.3 Collision-free path planning in the time-extended graph. Now for each agent $r \in R$, the path planning can be done in the time extended graph G^e. Each agent $r \in R$ at time $t \in T$ will occupy the vertex $P(r, t)$. We are given the assignment of the agents to their destinations. Therefore, we already know the destinations of the robots. Let the destination of robot r be the vertex sink(r). Then the path planning problem simplifies to just finding the shortest paths for agent $r \in R$ from the vertex $P(r, 0)$ to the vertex $(\text{sink}(r), \min_{t \in T}(t))$ in the time extended graph G^e while ensuring no collisions. Collision-free paths can be ensured by reserving the paths for each agent $r \in R$.

For each agent $r \in R$, finding the shortest path from its origin $P(r, 0)$ to its destination $(\text{sink}(r), \min_{t \in T} t)$ and then deleting the vertices and edges of the shortest path from the time extended graph G^e will let other agents to plan paths using the new graph. This

process will ensure that no collision can occur. Even though this algorithm ensures that no collision will occur, it does not guarantee that the solution is optimal. This problem can be formulated as a mixed integer linear program (MILP). However, solving the MILP for a large instance of the problem is computationally intractable. We will explore whether an auction mechanism can be used to find a better ordering for the agents to plan their paths.

This reservation procedure require maintaining a set of reserved vertices V_r and a set of reserved edges E_r that cannot be used while trying to find shortest paths for the other agents. To determine which robot gets to reserve paths first we will use an auction like mechanism. Each agent will submit their bids without knowing the bids of the other agents. For the bids, each agent will submit their shortest path length. Then based on the type of auctioneer, the winner will be decided. The advantage of using an auction-like mechanism is that if we just change the type of the auctioneer, the result of the auction will be different. This will result in different orderings of the agents based on which the agents would get to reserve their paths. This reservation ordering can have significant impact on the quality of the solution because reserving a path for some agent can end up removing a better path for another agent. For this project, the following types of auctioneers were implemented:

(1) Random auctioneer: The random auctioneer will randomly select a winner from the set of bidders. This will ensure that the ordering of the agents will be random.
(2) Bid minimizer auctioneer: The bid minimizer auctioneer will select the agent with the minimum bid as the winner.
(3) Bid maximizer auctioneer: The bid maximizer auctioneer will select the agent with the maximum bid as the winner.

Note that this auction mechanism can be interpreted as generalized version of the prioritized planning [14] where orderings of the robots depends on the type of auctioneer.

The following algorithm can be used to find the shortest paths for each agent $r \in R$ in the time extended graph G^e and then to reserve the paths.

Algorithm 3 Collision-free path planning in the time-extended graph G^e

```
1: procedure AUCTION(G^e, R, θ)                          ▷ θ is the type of auctioneer
2:     Q_a ← R
3:     while Q_a ≠ ∅ do
4:         for r ∈ Q_a do
5:             LabelSetting(G^e, P(r, 0))                 ▷ Update labels and predecessors
6:             t'_r ← the shortest path length for agent r
7:             Find a shortest path from P(r, 0) to (sink(r), min_{t=t'_r}^{T} {t})
8:             Bid the length of the shortest path as b_r
9:         end for
10:        r_w ← selectWinner({b_r ∀r ∈ R}, θ)    ▷ Depends on auctioneer type θ
11:        reservePath(G^e, r_w)
12:        Q_a ← Q_a \ {r_w}
13:    end while
14: end procedure
```

The reservePath$(\cdot, \cdot)$ procedure require some additional considerations. In addition to reserving used vertices and edges, an additional edge needs to be reserved to ensure the avoidance of swap collisions. For example, consider the environment with vertices A and B where an agent wants to move from A to B and another agent wants to move from B to A. If the first agent reserves the edge $((A, 0), (B, 1))$ the second agent can still use the edges $((B, 0), (A, 1))$ which results in a swap collision. To avoid this type of collision, for reservation of an edge $((u, t), (v, t+1))$ we also reserve the edge $((v, t), (u, t+1))$ which ensures the avoidance of such collisions.

3.4.4 Space complexity of the time-extended graph. The time extended graph can be constructed in polynomial time since we only need to create T copies of the original graph and then connect edges using the procedure described in a previous section. However, the space complexity of the time extended graph can be $O(T(2|V|+|E|))$ with a naive implementation. However, this space complexity can be reduced to $O(|V| + |E| + (|R| - 1)P_m)$ by using a more efficient implementation where $P_m = \max_{r \in R} |\pi_r|$.

For the reserved nodes and edges, we need to maintain two hash sets V_r and E_r that will contain all the reserved nodes and edges. Other than that we only need to store the original graph G and use that as an implicit representation of the time extended graph G^e. Using an adjacency list (we will implement it using a set adjSet(v) because we do not need to preserve any ordering of the nodes) representation of the original graph G, we can store the original graph in $O(|V|+|E|)$ space. We only need to store the original graph G and the sets of reserved nodes V_r and E_r. The reserved nodes and edges will also have a time associated with them to indicate which node or edge is reserved at what time. Each robot can reserve only reserve at most the time indexed nodes and edges in its path. If the maximum path length of all robots is $P_m = \max_{r \in R} |\pi_r|$ then the total number of reserved nodes and edges must be asymptotically equal to the number of robots times P_m. The last robot will not need to reserve it's path Therefore, the total reserved nodes and edges would have the size asymptotically equal to $(|R| - 1)P_m$. Therefore, the space complexity of storing the time extended graph G^e is $O(|V| + |E| + (|R| - 1)P_m)$. Note that we are assuming that the path lengths for each robots path are the same and equal to the largest path. However, in practice different robots may have different path lengths making the storage slightly more efficient.

Both adding and removing a vertex and an edge to the original graph require inserting or removing an element in an unordered set which takes $O(1)$ time. We will only need to be careful while iterating over the outgoing edges of a node in the time-extended graph. For a given node $(v, t) \in V^e$ to find the outgoing edges, we can iterate over all of node v's outgoing edges $(v, u) \in E$ in the original graph G and check whether the edge $((v, t), (u, t+1))$ is in the set of reserved edges E_r which can be done in $O(1)$ time. We will only need to iterate over all node v's outgoing edges and check whether they are in E_r. Since the maximum number of adjacent edges of a node in a $2D$ grid is 4 this check can be done in $O(1)$ time. If the edge is not in the set of reserved edges E_r, then we can add the edge to the set of outgoing edges of the node (v, t) and return the set. Therefore, more precisely the outgoing edges of a node $(v, t) \in V^e$ can be found in $O(|E|)$ using the following:

$$\text{outgoingEdges}(v, t) = \{(e = (v, t)(u, t+1)) : (e \notin E_r) \quad \forall (v, u) \in \text{adjSet}(v)\} \tag{3}$$

Similarly, outgoing vertices or the forward star $\Gamma^+(v, t)$ can be found using the following:

$$\Gamma^+(v, t) = \{(u, t + 1) : (v, t) \notin V_r \wedge (u, t + 1) \notin V_r \quad \forall (v, u) \in \text{adjSet}(v)\} \tag{4}$$

Instead of iterating over all the edges and vertices in the original graph before checking whether they are reserved, we use the stored information in adjacency sets about outgoing edges for a vertex in G. By using the adjacency sets of the orignal graph's nodes that contain the information about which vertices are connected we can use eq. (3) and eq. (4) to find the outgoing edges and vertices efficiently.

4 NUMERICAL RESULTS

A random 2D grid environment was generated with 20 rows and 20 colums. The number of agents was selected to be 20. A planning horizon of 40 seconds was chosen. The three implemented auctioneers were used to find different ordering of the agents before reserving their paths. The following results show the quality of the results based on the type of auctioneer used. For the quality metrics, the makespan and the flowtime were used. There were no collisions in any of the solutions

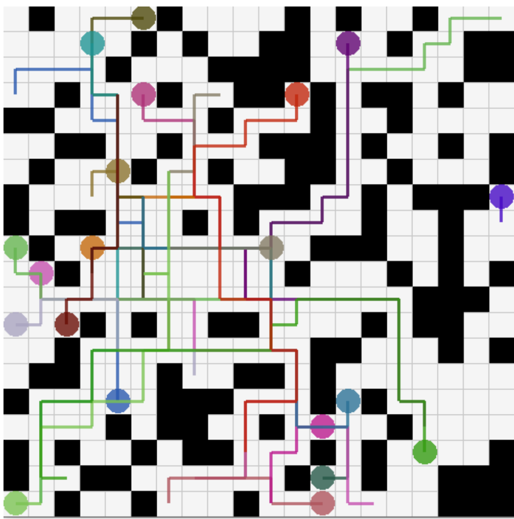


Figure 4: A 20×20 2D grid environment with 20 robots and a planning horizon of 40 seconds. The circles are the robots and the lines are the planned collision-free paths.

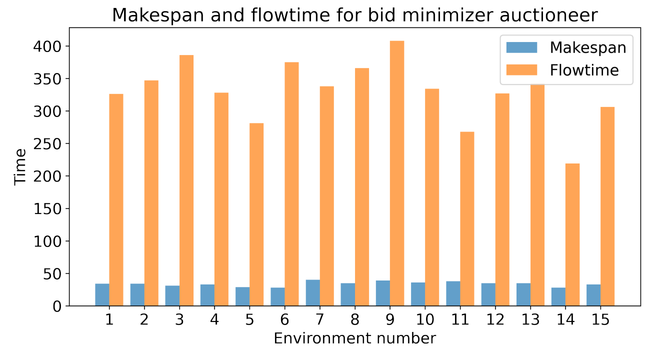
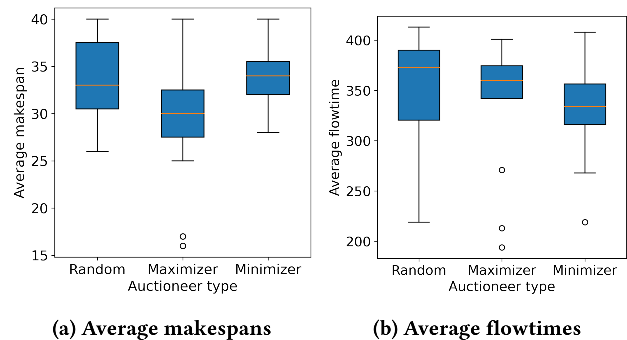


Figure 5: Makespans and flowtimes for the bid minimizer auctioneer for different environments



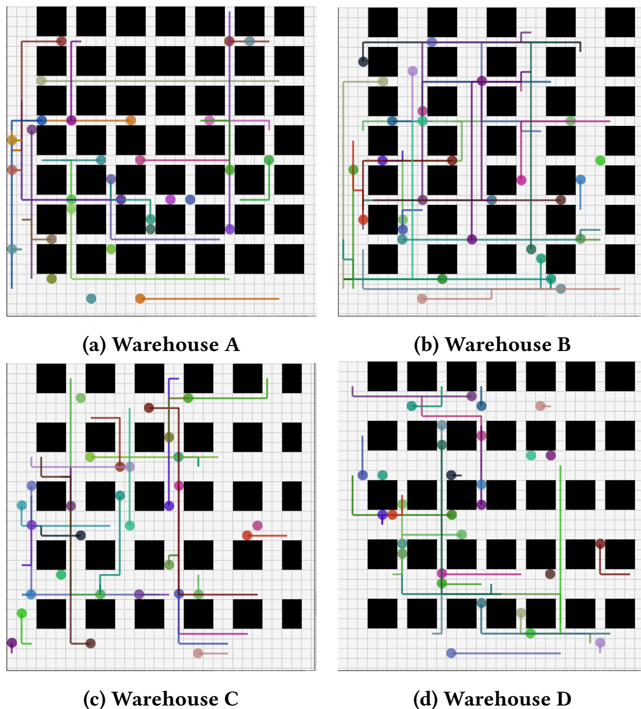
(a) Average makespans **(b) Average flowtimes**

Figure 6: Average makespans and flowtimes for different types of autioneer. (Averaged over 15 different environments.)

Figure 5 shows the different makespans and flowtimes for 15 different environments for the bid minimizer auctioneer. We also ran the same scenarios with bid maximizer and random auctioneer. Figure 6 shows the average makespans and the flowtimes for different types of auctioneers averaged over 15 different environments. Figure 6 shows that the average flowtime is minimum for the bid minimizer auctioneer compared to the other two auctioneers. However, the bid minimizer auctioneer can create an ordering of agents following which may result in deletion of valid paths for other agents. Because the bid minimizer auctioneer chooses the agent with the minimum bid as the winner, that agent will reserve its shortest path first. This reservation may result in removal of paths for other agents. To deal with this problem, the bid maximizer auctioneer can be considered. The idea is that the agent with the maximum bid or shortest path size does not have many options which is why it's bid is the highest. Therefore, securing the path for the agent with the highest bid or shortest path length may help with this problem. Figure 6 shows that the average flowtime increases for the whole system when the bid maximizer auctioneer is used. Bid maximizer also managed to minimize the average makespan the most in the generated scenarios. The reason bid maximizer auctioneer may have performed better than bid minimizer may be that using bid minimizer reserving shortest path or making locally greedy choices lead to worse performance. However, more environments should be tested before reaching any concrete conclusion. The other auctioneer that was implemented is the random auctioneer. The random auctioneer does not have any bias towards any agent and randomly selects an agent as the winner. The average makespan and the flowtime for the random auctioneer is higher than both of the other auctioneers. Therefore, random ordering of the agents may not be a good idea.

We also tested several warehouse like environments shown in Figure 7. The environments here are 30×30 $2D$ grids. For these

scenarios we used 30 agents in the problem instance and randomly generated valid start and goal positions for these robots. Valid start and end locations only mean that there exist at least one path from the start to the end position. This ensures that each robot has at least one path that it can use to reach its goal. We extended the graph in the time dimension for 50 seconds for these problem instances. Using the proposed approach we were able to find collision free paths for all the robots.



(a) Warehouse A **(b) Warehouse B**

(c) Warehouse C **(d) Warehouse D**

Figure 7: Collision-free path planning in warehouse-like 30×30 grid environments with 30 robots.

5 CONCLUSIONS AND FUTURE WORK

The main objective of this project was to develop a collision free and optimized approach to solve a variant of the MAPF problem. The proposed approach using time-extended graphs ensures no collision. Extending the graph in the time dimension requires the storage of a larger graph. This project explored whether better storage and more efficient computation schemes for the time-extended graph are possible. The time-extended graph was found to have certain properties. Properties of the time-extended graph like it being a DAG, guarantee the existence of paths, if paths exist in the original graph. An efficient way to find a topological order of the vertices was proved. These properties were exploited to find the shortest paths more efficiently using a label setting algorithm. The time complexity of the label setting algorithm is shown to be in linear time. The space complexity of the time-extended graph was also explored. Compared to just a naive implementation of the time-extended graph, a more efficient data-structure was proposed that has the space complexity equivalent to the space complexity of storing the original graph and sum path lengths of all robots. Additionally, whether auction-like mechanisms can be used to find higher-quality solutions was also explored. This project found that the bid minimizer auctioneer minimized the average flowtime the most compared to the other auctioneers. However, the bid minimizer auctioneer may result in deletion of valid paths for other agents. To address this problem, a bid maximizer auctioneer was also implemented. A random auctioneer was also implemented and found to perform worse than both of the other auctioneers in terms of the makespan and the flowtime minimization.

For future work, several improvements can be considered. For example, certain portions of the grid can be ignored before expanding the original graph if those portions are too far from an agent and its target. The currently proposed label setting algorithm expands all nodes and acts like uniform cost search. However, adding an admissible heuristic to expand nodes in the direction of the sink vertices can speed up the label setting algorithm, which in turn will speed up the shortest path finding. Using a heuristic, an A* algorithm can be designed. Insights were obtained into whether learning techniques and heuristics may also help solve the problem more efficiently with collision-free guarantees. Further, reinforcement learning can be used to find better combinations of paths. After selecting the next action for an agent, the reward can be immediately calculated if there is a collision. Otherwise, a rollout-like approach can be used to find the makespan or the flowtime, which can be used as the reward. Another way to use reinforcement learning would be find the optimal solution for smaller portions of the grid using an optimal solver like MILP or CBS. Then using the optimal solution, the RL agent can be trained. Even though the problem is NP-hard, an approximation algorithm may be developed to find better solutions with guarantees close to ϵ optimality based on the environment and the number of agents. Furthermore, how well the agents can do with selfish planning can also be explored. Analysis of similar techniques already exist in the transportation engineering literature. Applying those techniques in this context could be a new research direction that has not been explored in the MAPF literature as far as the authors are aware.

REFERENCES

[1] Anton Andreychuk and Konstantin Yakovlev. 2018. Two techniques that enhance the performance of multi-robot prioritized path planning. *arXiv preprint arXiv:1805.01270* (2018).

[2] Roman Barták, Neng-Fa Zhou, Roni Stern, Eli Boyarski, and Pavel Surynek. 2017. Modeling and Solving the Multi-agent Pathfinding Problem in Picat. In *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, Boston, MA, USA, 959–966. https://doi.org/10.1109/ICTAI.2017.00147

[3] S.-H. Chan, J. Li, G. Gange, D. Harabor, P. Stuckey, and S. Koenig. 2022. Flex Distribution for Bounded-Suboptimal Multi-Agent Path Finding. In *AAAI Conference on Artificial Intelligence (AAAI)*. AAAI, Washington, DC 20004, 9313–9322.

[4] Shushman Choudhury, Kiril Solovey, Mykel J. Kochenderfer, and Marco Pavone. 2021. Coordinated Multi-Agent Pathfinding for Drones and Trucks over Road Networks. *CoRR* abs/2110.08802 (2021).

[5] Julio Godoy, Stephen Guy, Maria Gini, and Ioannis Karamouzas. 2020. C-Nav: Distributed Coordination in Crowded Multi-Agent Navigation. *Robotics and Autonomous Systems* 133 (Nov. 2020), 103631.

[6] Florence Ho, Ana Salta, Ruben Geraldes, Artur Alves Gonçalves, Marc Cavazza, and Helmut Prendinger. 2019. Multi-Agent Path Finding for UAV Traffic Management.

[7] C. Leet, J. Li, and S. Koenig. 2022. Shard Systems: Scalable, Robust and Persistent Multi-Agent Path Finding with Performance Guarantees. In *AAAI Conference on Artificial Intelligence (AAAI)*. AAAI, Washington, DC 20004, 9386–9395.

[8] C. Leet, C. Oh, M. Lora, S. Koenig, and P. Nuzzo. 2023. Task Assignment, Scheduling, and Motion Planning for Automated Warehouses for Million Product

Workloads. In *IEEE International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 7362–7369.

[9] Hang Ma, Jingxing Yang, Liron Cohen, TK Kumar, and Sven Koenig. 2017. Feasibility study: Moving non-homogeneous teams in congested video game environments. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, Vol. 13. 270–272.

[10] Mike Phillips and Maxim Likhachev. 2011. SIPP: Safe interval path planning for dynamic environments. *2011 IEEE International Conference on Robotics and Automation* (2011), 5628–5635. https://api.semanticscholar.org/CorpusID:16210040

[11] Oren Salzman and Roni Stern. 2020. Research Challenges and Opportunities in Multi-Agent Path Finding and Multi-Agent Pickup and Delivery Problems. In *Proceedings of the 19th International Conference on Autonomous Agents and Multi-Agent Systems* (Auckland, New Zealand) *(AAMAS '20)*. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 1711–1715.

[12] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. 2015. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence* 219 (2015), 40–66. https://doi.org/10.1016/j.artint.2014.11.006

[13] Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. 2013. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence* 195 (2013), 470–495. https://doi.org/10.1016/j.artint.2012.11.006

[14] David Silver. 2021. Cooperative Pathfinding. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* 1, 1 (Sep. 2021), 117–122. https://doi.org/10.1609/aiide.v1i1.18726

[15] R. Stern, N. R. Sturtevant, A. Felner, S. S. Koenig, H. Ma, T. Walker, J. Li, D. Atzmon, L. Cohen, T. K. S. Kumar, E. Boyarski, and R. Bartak. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*. AAAI, Washington, DC 20004, 151–159.

[16] Peter R. Wurman, Raffaello D'Andrea, and Mick Mountz. 2008. Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses. *AI Magazine* 29, 1 (Mar. 2008), 9. https://doi.org/10.1609/aimag.v29i1.2082

[17] Jingjin Yu. 2016. Intractability of Optimal Multirobot Path Planning on Planar Graphs. *IEEE Robotics and Automation Letters* 1, 1 (2016), 33–40. https://doi.org/10.1109/LRA.2015.2503143

[18] Jingjin Yu and Steven LaValle. 2013. Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs. *Proceedings of the AAAI Conference on Artificial Intelligence* 27, 1 (Jun. 2013), 1443–1449. https://doi.org/10.1609/aaai.v27i1.8541

[19] Jingjin Yu and Steven LaValle. 2013. Structure and intractability of optimal multi-robot path planning on graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 27. AAAI, Washington, DC 20004, 1443–1449.

[20] Lifeng Zhou and Pratap Tokekar. 2021. Multi-robot Coordination and Planning in Uncertain and Adversarial Environments. *Current Robotics Reports* 2, 2 (April 2021), 147–157. https://doi.org/10.1007/s43154-021-00046-5