Survey of the Multi-Agent Pathfinding Solutions

Research · January 2021

DOI: 10.13140/RG.2.2.14030.28486

CITATION
1

READS
1,823

2 authors:

Erwin Lejeune
Coalescent Mobile Robotics
5 PUBLICATIONS **3** CITATIONS
SEE PROFILE

Sampreet Sarkar
CEA Tech
5 PUBLICATIONS **7** CITATIONS
SEE PROFILE

ECOLE CENTRALE DE NANTES
MASTER IN CONTROL AND ROBOTICS
EMBEDDED REAL-TIME SYSTEMS PATH
BIBLIOGRAPHY PROJECT

A Survey of the Multi-Agent Pathfinding Problem

Authors:
Erwin Lejeune
Sampreet Sarkar

Supervisor:
Loïg Jezequel
Asst. Professor at LS2N
Real Time Systems Research Group

A bibliographic study delivered as a requirement for the validation of the second year of the Master's degree

January 21, 2021

***Abstract*—Multi-Agent Path Finding (MAPF) is the planning problem that consists in planning routes for multiple agents, avoiding both collision among themselves and with their environment. This research topic deals with Artificial Intelligence (AI), Multi-Agent Systems (MAS) and Autonomous Robotics and there is a growing interest around it for relatively new applications that appeared along with autonomous mobile vehicles: warehouse management, drone delivery. Here, we will focus on path finding and not discuss task planning, comparing solutions for finding collision-free paths.**

***Index Terms*—Multi-Agent, Collision avoidance, path planning, centralized planning, decentralized planning, unmanned aerial vehicles, unmanned ground vehicles**

I. Introduction

The single-agent path finding problem has been studied extensively throughout various fields such as robotics, artificial intelligence or automatic control. There has been various solutions with their strengths and weaknesses presented by researchers and the usual solution is found using graph search algorithms [1] or sample-based search [2], which both can use informed [3] or uninformed [4] searches. The Multi-Agent Path Finding *(MAPF)* problem is more complex and require more elaborated solutions for it. We can try and understand intuitively, why this is such a complex problem when we consider the environment presented in Figure 1. We can observe that the main challenges that we would be facing is with coordination between the various robots, having to constantly update the locations of the obstacles and having to compute the paths for each robot such as they avoid collision with themselves, as well as with the obstacles.

The MAPF problem resolved sequentially by every agent is NP-hard [5]. The general description of the problem involves the proposal of paths *(optimal or sub-optimal)* for each agent *(mobile robots, Unmanned Aerial Vehicles (UAVs), game characters, etc.)* from a starting position to a goal position in 2-D or 3-D space, whilst avoiding obstacles, which might be static or dynamic. The applications of MAPF are very diverse, and perhaps one of the best examples that we can consider is that of the air traffic control at any airport. In the process of trying to automate the planning, we observe that standard algorithms like A* and RRT, which are the go-to solutions for single-agent path planning, do not perform very well. They result in the exponential growth of the state-space with the number of agents. In such cases, a global search is often impractical, even if there are only a small number of units in consideration. With an increase in the research and development of swarm-robotic applications, we have to also consider collective actions for a fleet of units and convey the appropriate messages to generate collision-free paths for each individual unit, and the fleet in turn will collectively accomplish the task.

One of the main purpose of MAPF problems is to be able to generate paths that are free of conflicts. In the context of UAVs, the solution turns out to be effective designing of Conflict Detection and Resolution *(CDR)* algorithms [6]. In such a scenario, the algorithm has to generate multiple UAV paths, while maintaining a minimum distance of separation required for safe operation. In a more general context, conflict detection and resolution may be done online or offline, depending upon the agents and the environment they are interacting in. However, through the course of this document, we will see that sometimes a mix of online and offline approach is the most favourable.

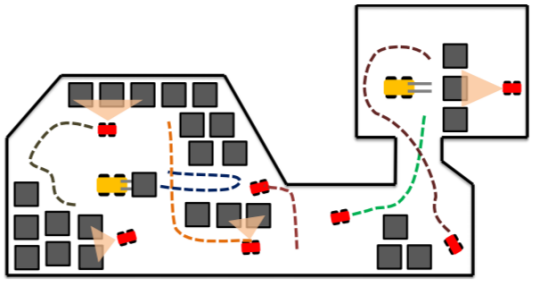


Fig. 1. Multi-Robot team operating in a constrained warehouse environment

The construction of the optimal strategy for the agent would require some knowledge about the problem — a description of the agent, and how it interacts with the environment, a description of the domain or the environment the agent will function in, and a problem statement, which presents the current state of the agent, and the final state the agent wants to be in. However, in further pondering of the problem statement, it is not enough: the current states of the other agents in the environment is also a necessary piece of information. One can think of some approaches we might take to solve this problem, namely manual abstraction of the search space, and decomposition of the bigger search problems into multiple smaller search problems. These approaches have both strengths and weaknesses [7]. The general trend to these types of problems have been to somehow balance the completeness and optimality factors, while still retaining enough performance to be practical.

II. The Multi-Agent Path Finding Problem

Multi Agent Path Finding (MAPF) is the problem of planning paths for agents to reach their targets from their start locations, such that the agents do not collide while executing the plan. A standard description of the search space with respect to low altitude airspace domain has been provided in [6], as well as the methods to solve the issue from a UAV perspective in a dense city. Similarly, the authors in [8] have provided a similar algorithm for the domain of mobile robots, where they dealt with the collision avoidance and path planning problems by enforcing a minimum distance of separation between pairs of robots. The assumptions that the authors have worked under were that the robots have been independently assigned a set of tasks in an unordered manner. This becomes important when we consider the scenario

with respect to swarm robotics, where a group of robots collectively accomplish a task or a set of tasks. If the robots are to operate as a swarm, then care must be taken to ensure that we have a system of acknowledgement that a certain task has indeed been completed by a robot in the swarm. This approach, while raising complexity of the general algorithm, ensures that we have a certain synchronisation in the operation.

The main challenges of having to solve such a problem can be summarized as follows:

- **Collision Avoidance**: The main challenge in the solution of the MAPF problem happens to be in the management of dynamic obstacles in the environment. Static obstacles provide a certain sense of serenity to the solution, in the sense that their location in the environment is not subject to change during the time of operation, however dynamic obstacles pose two distinct levels of disintegration of the overall performance of the planner. In the first case, if the obstacles move about in relatively simple mannerisms, and can be dealt with by adding simple maneuvers to the plan *(as limit switches to sense an immediate obstacle)*, however the problem becomes more complex when the environment is poorly known, or the obstacles may arrive in a sporadic manner. This problem has been explained in great detail in [9], which dealt with the notion of autonomy in the deep space probes made by ESA (European Space Agency).

- **Real-Time Performance**: This particular challenge is general for all kinds of planning — not usually limited to the MAPF domain. In mission-critical applications, being able to produce accurate results is accompanied by the challenge of doing so on time. As a result, obtaining solutions of a NP-hard problem becomes extremely difficult to manage for such applications which have to guarantee real-time results and a certain level of autonomy. Most applications nowadays include a certain degree of human intervention along with autonomous capabilities, however the ultimate aim of the researching scientists is to be able to guarantee real-time performance along with complete autonomy.

- **Domain Dependency** : Along with the above mentioned challenges, this particular point is also a prevalent challenge in simple AI Planning, which can be extrapolated into the field of MAPF as well. The idea, simply put, is to be able to design "one planner to solve all" problems. However, the challenge quickly piles up from there, as it is almost impossible to write one central planner which can take into account ALL the types of planning problems there exist in today's world, and is able to accept the planning problems of tomorrow. In such cases, it is often advisable to be able to break off from the central planner notion, and instead to write planners which are domain specific — *i.e.,* they will be optimized for performance in only one specific domain.

III. Representation of the MAPF Problem

This paper uses an extension of the representation in [10], which is a classical way of describing the MAPF problem. The purpose in MAPF problems is to find non-interfering, collision-free paths for several agents. The MAPF representation we use is, for a group of n agent $\phi = \{\varphi_{i=1}^{n}\}$, a quadruple $\varpi = [SG, \{A_i\}_{i=1}^{n}, I, G_i]$ where:

- SG is a state-graph where each states are connected through actions;
- $I \subseteq SG$ encodes the initial state of the system;
- $G \subseteq SG$ defines a set of goals;
- A_i is the set of actions that can be performed by agent ϕ to go from one state to another (e.g. figure 2);

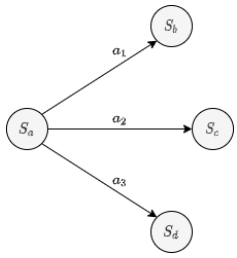


Fig. 2. Graphical representation of states and actions

A solution is a partially ordered sequence of actions such that each action in the solution is associated with a single agent. A plan is a set of states starting at I and ending at G. With each actions in the problem, a cost is associated, the more costly the action the less optimal the path: an optimal plan is a plan with the minimum cost among all existing plans for the defined problem. A cost estimation from one state to the goal state is a Heuristic, it directs the search towards the minimum cost path, reducing the depth of the search. In the single-agent A* algorithm [11], the path returned by the search is said to be optimal if the heuristic is admissible (the estimation should always underestimate or estimate exactly the cost to the goal).

The question we will move forward with and try to answer is — given a graph of the environment where there are n agents, with each agent having a initial state, a final state, and a list of available actions to choose from, how do we deliver a path for each of the n agents?

IV. Segmentation of MAPF methods

Traditional multi-agent path planning takes two main approaches [7]. A centralised approach that incorporates a global planner to compute all agents' paths simultaneously, which is in theory optimal but in practice has repellent complexity and doesn't scale up to many units. On the other hand, a decentralised method significantly lowers computations by

decomposing the problem into several sub-problems. [12] A typical approach first computes paths individually, ignoring all other units, then handles the collision avoidance as they come. This is often much faster but yields sub-optimal solutions and loses completeness.

A. Centralized Planning for Distributed Plans (CPDP)

In this category of multi-agents planning, there is one planner and several agents, it focuses on the control and the coordination of actions performed by several agents in a shared environment [13]. Here, the order of tasks isn't important and partial planners can distribute sub-plans. The agents negotiate, bid to the global centralized planner to resolve conflicts and execute their plan.

B. Distributed Planning for Centralized Plans (DPCP)

DPCP involves a complex planning process that is distributed among the agents, who cooperate and communicate by sharing their objectives and their own representation of the planning problem (i.e. state-graph, timed automaton...). The primary objective is to build a valid plan. All agents generate their portions of sub-plans, if one of the agents doesn't satisfy the given constraints, a rollback is performed by switching to a new partner agent. All the sub-plans are in the end synchronized into one global plan [14]. It can lead to incompatibility of goals (i.e. for one agent's objective to be valid, another agent can't reach his goal), sub-plans representations (i.e. each agent has a state-graph representation except one, who has a timed automaton) being different from one agent to the other...

C. Distributed Planning for Distributed Plans (DPDP)

DPDP is often more efficient, but doesn't offer the guarantee of a complete plan. Each agent produces a plan either independently from the others (agent-driven) or by being directed by a common goal (goal-driven). There are three main solutions for this category of MAPF: Plan Fusion, Iterative Planning, Negotiated Planning (e.g. [15]). Plan Fusion require the plans to all have the same representation for their execution to reach the desired goal. Conflicts can appear between plans but the resolutions of those conflicts is performed by an analysis of the interactions: analysis of the pre/post conditions operators; if all the preconditions are satisfied at the same timestamp, then the plans are executed in parallel, otherwise one should try to order them. If none of those options are possible, the conflicts isn't solvable and the plan will fail.

Iterative Planning suggests that at each steps of a plan, all agents propose the next step to everyone else. The prioritized planning is distributed and the resolution of the conflicts are recognized and handled in real-time (while executing the partial plans). It requires to be able to rebuild a plan dynamically.

V. Survey of CPDP

A. Safe Interval Path Planning

The conventional approach towards solving the dynamic obstacle based path planning problems is to treat the obstacles as static within a small time frame, and treat it as a static planning problem within that time frame. However, this adds additional time complexity to the problem. A novel solution has been provided in [16], which works under the assumption that there can only be a finite number of safe intervals*(a time interval where there exists no collision and which if extended by even one time step, would result in collision)* to operate with. This allows for the algorithm to constrict the search space and thus provide improvements in the performance. This approach can be thought of as a state minimization approach for those who are familiar with Model Checking and Formal Verification techniques — albeit with a focus on a very specific domain.

A configuration from the robot's perspective is a set of non-time variables, which generally for a mobile robot is *(x, y, θ)*, which provide us with vital information about the robot, such as where it is right now, in which direction it is facing, and allows us to draw meaningful inferences by extrapolating from this data such as where the robot might be one time step from now. Aside from defining the notion of a safe interval, the authors have also defined the concept of a collision interval, *i.e.,* the time interval during which there is a conflict in the configuration, generally the possibility of collision with the dynamic obstacles. By convention, the collision interval must be preceded and followed by a safe interval. This approach minimizes the total number of good states in the search space to a set of safe and collision intervals, where one safe interval could be a rather long collection of good configurations of the robot.

The algorithm works under the assumption that it always has a list of dynamic obstacles, their radii, and their trajectories*(list of points showing the change in configuration in the robot with time)*. This helps to generate a collision map of the obstacles across time. Another key assumption which the algorithm works under is that the robot is able to wait *in situ*. This is not a point of concern for most types of mobile robots, other than bicycle-like mobile robots, which would fall if they were commanded to stand in place.

Once the planner has been provided with the location of the agents and the collision map, it is not too difficult to perform graph search like A^* as shown below. We need a specialized method for obtaining the successors to the current state, called getSuccessors(s). the method applyMotion(m, s) will simply activate the motion m for the current configuration s. Similarly, the methods startTime(s) and endTime(s) gives us the starting and ending times for the interval respectively. This is important as the authors have focused mainly on a time-optimal algorithm.

```
def getSuccessors(s):
    successors = []
    for m in available_motions(s):
        #We apply the motion and see how much it
    adds to the time boundaries
        cfg = applyMotion(m, s)
        time_m = time taken to execute action m
        t_start = time(s) + time_m
        t_end = endTime(interval(s)) + time_m
```

```
        #si = safe interval
        for si in cfg:
            #Check for valid configurations
            if (startTime(si) > t_end) or (endTime(
    si) < t_start)
                continue

            t = time(cfg[si])
            if t == None:
                continue

            successors.append(cfg[i])
    return successors

g(s_start) = 0
opened_nodes = None

opened_nodes.append(s_start)
f(s_start) = h(s_start)

while(s_goal is not expanded):
    sort(opened_nodes, key=f())
    s = opened_nodes[0]
    successors = getSuccessors(s)

    for s' in successors:
        if s'.visited = False :
            f(s') = g(s') = inf

        if g(s') > g(s) + c(s, s'):
            g(s') = g(s) + c(s, s')
            updateTime(s')
            f(s') = g(s') + h(s')
            opened_nodes.append(s')
```

Listing 1. A^* pseudocode using the SIPP approach

The algorithm has shown promising results in real-world implementations as well, having being implemented on the PR2 robot on top of the ROS Navigation stack. What was incredible was the performance in tight spaces, which do not provide much maneuverability to the robot, where the robot chooses to wait for the human to pass by moving away from the path and resuming its journey to the goal once the human has passed. This shows promise for implementation in areas with high human-machine interaction, such as factory floors or hotel lobbies.

B. Conflict Based Search (CBS)

The state-space spanned by A^* is exponential with respect to the number of agents. When there is only a single agent, the graph expands linearly. Conflict based search (CBS) aims to solve the MAPF problem by decomposing this search space into a large number of constrained single agent path finding problems. Each of these problems can be solved in linear time, only proportional to the size of the graph, and the length of the solution. As a result, we can observe that the number of agents will contribute exponentially to the length of the final plan.

The CBS algorithm works in two levels — the *high level*, where it aims to discover new conflicts and develop new constraints to resolve the conflicts, and at the *low level*, it strives to generate a path for each single agent in accordance to the new constraints.

The algorithm searches through a *Conflict Tree* at the high level, which is a binary tree consisting of multiple nodes, as shown in Figure 3. A node can be thought of as a data structure containing three fields, a set of constraints, a set of solutions, and a cost variable, as shown in Figure 4. The length of the set of constraints and solutions will depend on the number of agents. A node in the conflict tree is considered a goal node when its corresponding solution is valid, *i.e.,* the paths for all agents are without conflict.

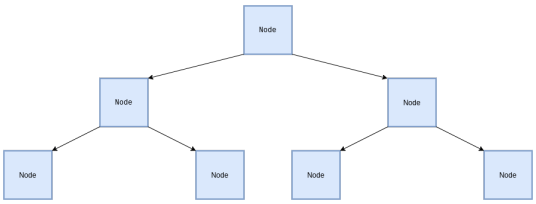


Fig. 3. Constraint Tree for the CBS algorithm

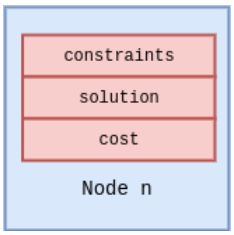


Fig. 4. A single node from the constraint tree

This graph is passed on to the low level, which traverses it node by node to take in an agent and its corresponding set of constraints. The low level search has a very simple job — to find a path for the agent that satisfies all the other constraints, while actively ignoring the influence of other agents. This is returned to the high level search, which validates the solution against other agents by checking that no two agents are in the same space in the same time step. If this process is validated, then the Node containing that solution is termed to be the goal node. The pseudocode of the high-level of the CBS algorithm is presented in the Listing below:

```
Input : MAPF instance
R.constraints = 0
R.solutions = individual paths using the low-level
    search
R.cost = SIC(R.cost)

put R in OPEN

while OPEN is not empty, do:
    P = best nodes from OPEN
    Validate the path until conflict occurs

    if P has no conflict, then:
        return P.solution

    C = first conflict(a_i, a_j, v, t) in P

    for each agent a_i in C, do:
        A = new_node
        A.constraints = P.constraints + (a_i, s, t)
        A.solution = P.solution
        update A.solution from low-level(a_i)
        A.cost = SIC(A.solution)
```

```
        Insert A in OPEN
```

Listing 2. High level of CBS

C. Enhanced Conflict Based Search (ECBS)

A sub-optimal variant to the CBS algorithm we experienced earlier has been proposed in [6], where the focus has been on providing a "quick" solution, rather than the "best" solution as was being provided by CBS. Given a parameter of optimality w, the method of distributing w over a range $[w_L, w_H]$ has been a problem which has predominantly remained domain dependent. Also, we have to address the very real concern of having to maintain the same level of optimality at all stages of the search, which is a tedious task to say the least. Enhanced CBS*(ECBS)* tackles these problems to by proposing a low level focal search with optimality w. For every node i in a set of opened nodes*(borrowing the explanation of opened nodes from the context of A*)*, we have a minimum value $f_{min}(i)$ of the f value in the CBS low-level search for each agent a_i. This value $f_{min}(i)$ presents us with a lower bound on the cost of the optimal path. For each node n in the CT we came across in Figure 3 and k agents in the environment, we can assert the following about the Lower Bound on the node n, given by $LB(n)$:

$$LB(n) = \sum_{i=1}^{k} \cdot f_{min}(i) \leq n.cost \tag{1}$$

This assertion holds true because CBS did not work under the assumptions of lower bounds. The low level of ECBS returns two pieces of information to the high level — the cost*(n.cost)* and the Lower Bound on the node*(LB(n))*. With n being a node in the CT which has already been opened at the high level, we can see that the minimum value from the set of Lower Bounds $LB = min(LB(n) \forall n \in$ opened_nodes projects a global lower bound to the solution of the entire problem. The set of nodes for the focal search in ECBS will be a subset of those in the opened_nodes set, and can be selected as follows:

$$focal_{nodes} = \{n | n \in opened_{nodes}, n.cost \leq LB \cdot w\} \tag{2}$$

Since LB is a lower bound on the problem itself, we see that it must contain all nodes that are in accordance with the optimality factor w, guaranteeing the fact that the maximum cost of the solution would never theoretically exceed the value $w \cdot C^*$, where C^* is the size of the problem. The advantage of ECBS over CBS is that it allows for a greater deal of flexibility at the high level while the low level provides low cost solutions. This makes it more adjustable and welcoming to quicker yet sub-optimal solutions. Since ECBS is a systematic search, it is imperative that it will eventually find a solution if there exists one, which leads ECBS to be complete.

D. Cooperative A*

One of the most cited methods for solving the Cooperative Path-finding problem was proposed in [17] as the Cooperative A* *(CA*)* algorithm. It is based on the A* algorithm [3], and proposes a tractable algorithm that builds a search space using time as an additional dimension. Each agent can be in a defined state at time t, and reserves that time t. No other agent can enter this state at this time t, ensuring the conflict resolution as all the searches progress. As a result, each cell in the reservation table is marked as an obstacle for a brief period of time, and can be used by the other agents later on in the search. Note that as there is no re-planning in CA*, which only ensures cooperation among agents and will not handle dynamic obstacles by design. We can see in Figure 5 how the items in the reservation table fade with time.

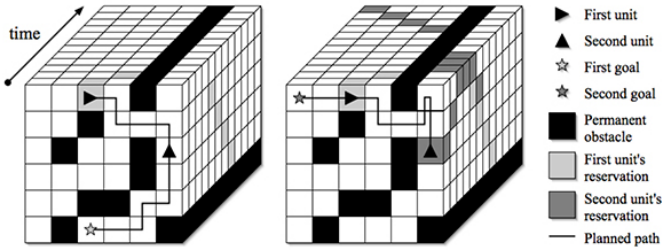


Fig. 5. Graphical representation of how the cooperative A* search takes place along the time dimension [17]

The Cooperative A* algorithm accepts any admissible heuristic, and the default go to for this algorithm would be the Manhattan Distance, owing to its simplicity and ease of computation. For two points (x_1, y_1) and (x_2, y_2) in a 2D plane, the Manhattan Distance can be computed as $d_{Manhattan} = |x_1 - x_2| + |y_1 - y_2|$. However, care should be taken that for environments which are difficult to navigate through, using the Manhattan Distance might not be the most informative heuristic.

E. Local-Repair A*

Local Repair A* [17] is a set of algorithms where each agent tries to search for an optimal path to its respective goal, ignoring the effect of all other agents except its immediate neighbours, until a collision can be foreseen. Just before the agent is to move into the zone of collision, *i.e.,* the common state where the collision was supposed to occur between two or more agents, the agents recompute their path from that specific point the environment to the goal. Loops in the computed path are highly undesirable, and hence the *agitation level* of the algorithm *(similar to a penalty)* is increased each time the algorithm is compelled to find another solution. Random noise is then added to the distance heuristic function proportional to the level of agitation. This is done in order to encourage agents to find new paths each time, and in doing so, eventually overcome the obstacles that may lie in their path.

It should be noted that Local Repair A* proposes its fair share of problems, especially in environments which are a little difficult to navigate. If a certain region is surrounded with many obstacles which are constantly moving, then the

computation of the complete A* search at each step would introduce a lot of time delay, which in turn would affect the performance.

VI. Survey of DPDP

A. Rapidly-exploring Random Trees (RRT) based methods

RRT [18] consists in points randomly generated and connected to the closest available node. Each time a vertex is created, the algorithm checks that the vertex is located in a free space. It also checks that the connection between the vertex and its closest neighbor avoids any collision with obstacles. The algorithm ends when a node is generated within the goal region, or a given threshold is surpassed. As seen in Figure 6, an issue of sample-based algorithm is their unpredictability: without a hard threshold, it could run infinitely without generating a node in the goal region, and since we generate the nodes randomly, the only guarantee RRT provides is that if you generate an infinite number of random nodes and there exists a path to the goal, you will find that path. Note than in RRT, this path will - in general - not be optimal, because the nearest node in the graph from our randomly generated node isn't guaranteed to be the best candidate.

```
goal = g # region that identifies success
counter = 0 # keeps track of iterations
threshold = n # threshold number of iterations
graph = G(V, E) # graph containing edges and
    vertices, initialized as empty
while counter < threshold:
    new  = random_position()
    if not is_free(new) == True:
        continue
    nearest = nearest(graph, new) # find nearest
    vertex
    link = chain(new, nearest)
    graph.append(link)
    if new in goal:
        return graph
return graph
```

Fig. 6. RRT Pseudocode

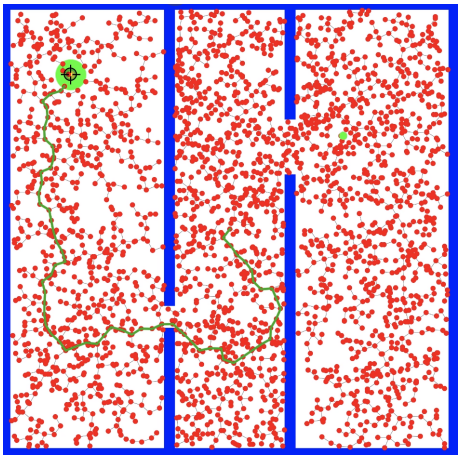


Fig. 7. Graph generated by RRT [19]

A drawback from RRT is the fact that the path found is not optimal (e.g. fig 7), as mentioned previously. Even though we keep exploring the search space, a better path will not be found. This issue is addressed in RRT* [20]. V sampled from the obstacle-free region, and edges E that connect these vertices together are being maintained by RRT* as it tries to find this solution. Every time a random node is generated, the first step of the algorithm is to optimize the graph by using this node. RRT is a variant of RRT that rewires the tree as better paths are discovered. After rewiring the cost has to be propagated along the leaves, to evaluate if new paths are better or worse. The procedure is described in Figure 8.

```
goal = g # region that identifies success
rad = r
counter = 0 # keeps track of iterations
threshold = n # threshold number of iterations
graph = G(V, E) # graph containing edges and
    vertices, initialized as empty
goal_satisfied = False
while counter < threshold and not goal_satisfied:
    new = random_position()
    if not is_free(new) == True:
        continue
    nearest = nearest(graph, new)
    cost(new) = distance(new, nearest)
    best, neighbors = find_neighbors(graph, new, rad
    )
    link = chain(new, best)
    for n in neighbors:
        if cost(new) + distance(new, n) < cost(n):
            cost(n) = cost(new) + distance(new, n)
            parent(n) = new
            graph += {new, n}
    graph += link
    if new in goal :
        goal_satisfied = True
return graph
```

Fig. 8. RRT* Pseudocode

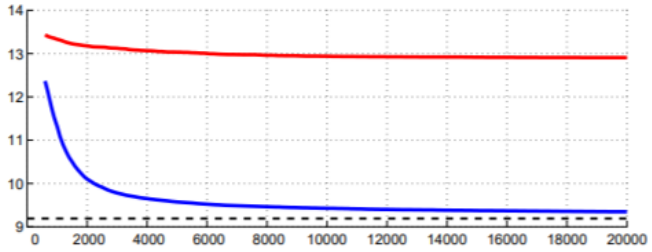


Fig. 9. Path Cost w.r.t. number of generated nodes for RRT (Red) and RRT* (Blue) [21]

RRT* has more overhead, but it will always find the optimal path among the opened nodes (e.g. figure 10). Apart from ensuring probabilistic completeness, RRT* also guarantees asymptotic optimality (e.g. figure 11, showing a larger number of generated nodes leads to a better path that is still optimal among the generated nodes) - which is in contrast to its predecessor RRT, as shown in Figure 9. It has been proven

mathematically that it reaches the said solution in infinite time [20].

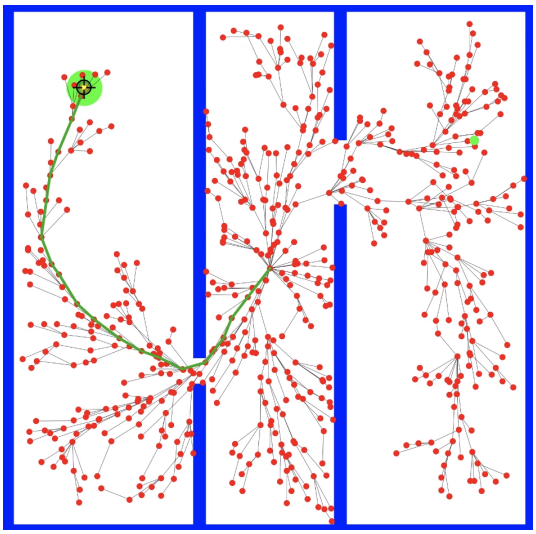


Fig. 10. State of the Graph generated by RRT* the first time the goal is reached [19]

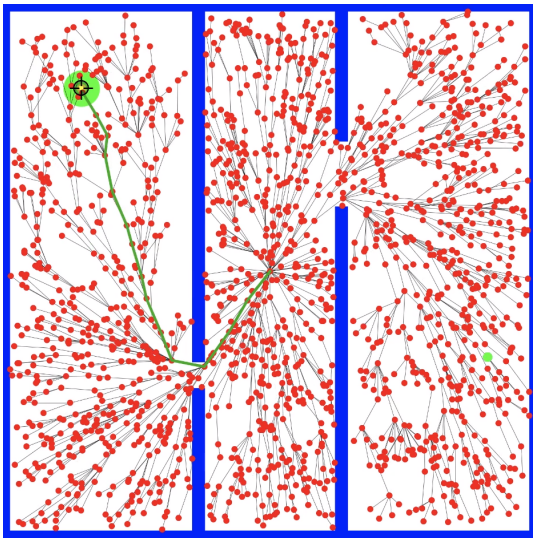


Fig. 11. State of the Graph generated by RRT* after exploring 500 more nodes [19]

RRT and RRT* serve as basis for enhancements used to solve the MAPF problem. Indeed, each agent can use RRT and move towards a goal and re-plan [22] in case an obstacle is blocking the path during the motion towards generated waypoints. Dynamic re-planning with RRT* allows to obtain new optimal paths each time a problem is met along the way. While [22] uses dynamic obstacles as a research topic, it can be applied to the MAPF problem as a distributed approach, where each agent treat each other as dynamic obstacles. While the original RRT algorithm is a fast way to plan paths in complex, high-dimensional spaces, such as CL-RRT* (Closed-Loop Rapidly-exploring Random Trees Star) [23] and its extension DMA-RRT (Distributed Multi-Agent Rapidly-exploring Random Trees) [24] are much more useful as online path planners.

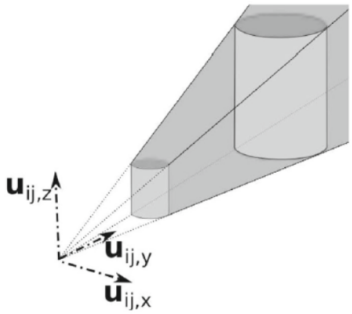
CL-RRT introduces the use of closed-loop prediction in RRT. While the replanning method can supposedly handle dynamic obstacles, highly unstable dynamic obstacles may lead - in a real-life application with kinematic constraints - to vehicle dynamics singularities with valid paths that require unfeasible manoeuvres. In CL-RRT, an input to the controller is sampled, and a prediction is made using the vehicle model for the controller to compute a predicted trajectory. The planner then manages to create smooth trajectories in an efficient manner while allowing the planner to explore cluttered, narrow spaces.

While all the former RRT-based algorithms presented before can be used to solve the MAPF problem by replanning dynamically, DMA-RRT has been developed to solve the MAPF problem. A merit-based token passes a coordination strategy and updates the planning order based on if any given agent is in a better or worse position to replan. It results in a more optimal global cost and team performance. Cooperative DMA-RRT introduces the ability for each agent to modify another's plan to select the most optimal one - cost wise and performance wise.

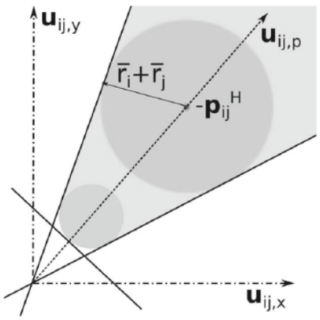
B. Reactive Approaches

In reactive approaches, a local planner (controller) handles dynamic obstacles. Even though very susceptible to deadlocks and not guaranteeing optimal results performance wise, their scalability and simplicity still makes them relevant. These approaches are often classified in two categories: rule-based and optimization-based. Optimal control laws such as an adaptive PID local planner [25] and potential fields [26] methods are considered rule-based and have good results in low speed and low agent density environments. This subsection will focus on two optimization based approaches, that produces better and more relevant results.

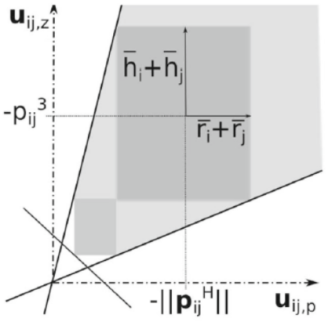
1) Velocity Obstacle: Three optimizations method were proposed in [27]: distributed convex, centralized convex and centralized non-convex. The first two methods minimize a singular and joint quadratic function subject to quadratic and linear time constraint. Those approaches scale well but do not guarantee the global optimum. The third method ensures a global optimum as it explores the entire state space, but scales poorly. In these methods, the robot to robot collision check is introduced as a constraint in the optimization representation. Neighbour agents are modelled as velocity obstacles, and a shared responsibility between robots, called reciprocity is introduced. In a situation where two robots are heading towards each other such as Figure 12, half of the responsibility for collision avoidance is taken by the first agent and the rest is taken by the second one. Each robot has an independent feedback loop executing, allowing it to control its new velocity based on its observations.



(a) Velocity obstacle in relative velocity



(b) Projection onto the horizontal plane



(c) Projection onto a vertical plane

Fig. 12. Interaction between two agents a_i and a_j from a_i's point of view: u_i and u_j represent their respective velocities. The grey zone is the area excluded by the optimization constraints [27]

2) Nonlinear Model Predictive Control (NMPC): Collision avoidance can be added to a NMPC model efficiently, although the high computational use and the complexity of implementing a complex dynamic system model to it prevents it from being a fast implemented method, as opposed to RRT methods or A* methods that can just let every robot handle their own dynamics. In the approach from [28], the trajectory tracking and collision avoidance is solved by a single optimization solver. This is one of the state-of-the-art solution with the best results. The NMPC controller solves a finite horizon optimal control problem (OCP) at each step, using a cost for a deviation from the pre-computed trajectory and a penalty for deviating from the control input. A hard constraint is set for the distance between agents and a set of admissible control inputs is defined. A collision cost $J\_c(x(t))$ is also introduced based on a smooth collision function:

$$J_c(x(t)) = \sum_{j=1}^{N_agents} \frac{Q_{c,j}}{1 + exp\kappa_j(d_j(t) - r_{th,j}(t))} \quad (3)$$

where $d\_j(t)$ is the Euclidean distance to the j_th agent, $Q\_c, j > 0$ is a tuning parameter, $\kappa\_j > 0$ is the smoothness factor of the cost function and r_th, j is a threshold distance between the agents.

Figure 13 shows all agents' motions in a collision-free scenario where each agent has to join the hub (starting point) of another agent. The discontinuous lines shows the shortest path to their goal, and the continuous lines shows the path they followed.

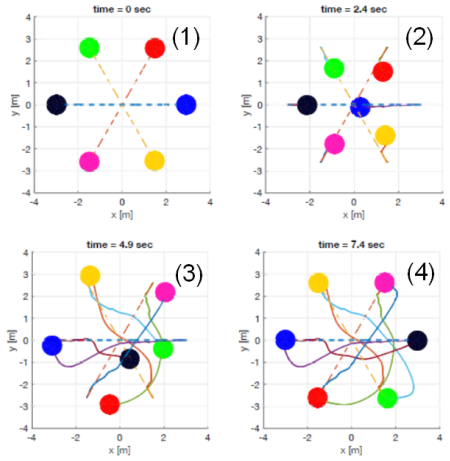


Fig. 13. Simulation of MAPF solving with NMPC [28]

VII. Implementation

This section highlights the performance of one algorithm from each categories of Multi-Agents planners: Conflict-Based Search and Non Linear Predictive Control. We will benchmark the implementation of CBS from [29], and our own implementation of NMPC [30]. As centralized solutions tends to explode with the number of agents and the size of the search space, reactive distributed solutions should not. Note that [30] does not feature concurrency yet and agents resolve conflicts on each time step iteratively. It is although possible to fetch the total computation time of each agent, which emulates a system with the algorithm implemented on each robot. Introducing concurrency should greatly solve any performance issue with large number of agents, if one has some.

A. Performance assertions

The main data used is running time w.r.t. number of agents, and running time w.r.t. search space (greatest x and y among start and goal positions for all agents in the NMPC case). All data displayed have ran the same scenario in simulation: hub swaps. Each agent goal is another's starting point, similar to Figure 13.

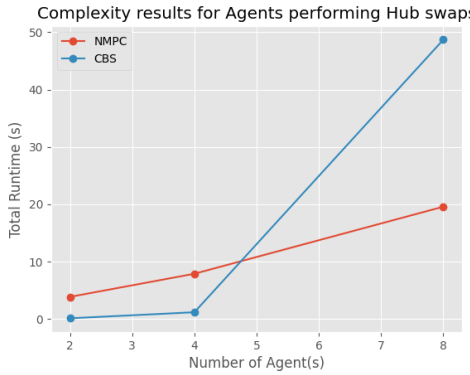


Fig. 14. Total Runtime (s) w.r.t. Number of Agents for NMPC [30] and CBS [29], for the same scenario and the same search space

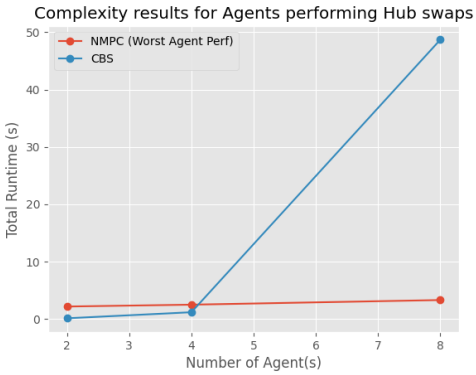


Fig. 16. Total Runtime (s) w.r.t. Number of Agents for the worst performing NMPC [30] Agent and CBS [29], for the same scenario and the same search space

In Figure 14, the properties mentioned in V and VI appear to be verified. A larger amount of data would result in the same tendency: in CBS, each agents has to search the whole search space while in NMPC, each agent resolves locally their predicted conflicts, making it non-deterministic but fairly less expensive computationally.

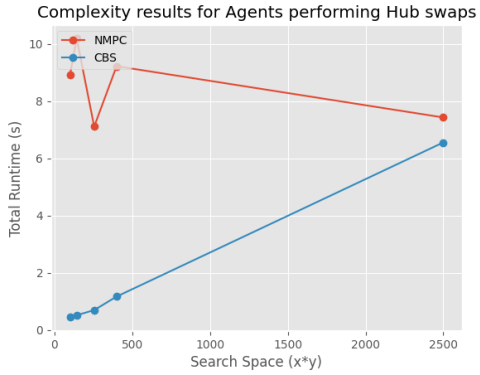


Fig. 15. Total Runtime (s) w.r.t. Search Space Size for NMPC [30] and CBS [29], for the same scenario and the same number of agents

The above plot 15 highlights the independence of the performance of the NMPC [30] algorithm with respect to the size of the search space. As mentioned in the previous section, it performs a local search and doesn't expand to the whole search space: it reacts to dynamic events in its horizon, making it highly relevant in crowded environments. For the centralized planner, the larger the search space the longer it will take to compute a path, due to the completeness of the algorithm.

Figure 16 shows the major upside of NMPC: as — in a real-world multi-agent system — all robotic entities would embed the algorithm, they would run it in parallel. The runtime of finding a path to the goal then isn't the total runtime of the NMPC simulation, but the computation time of each agent. Here, we displayed the worst performing agent for each dot. It shows in reactive navigation, the number of agents does not make the computation time explode, as they all perform actions individually. What highly increases the runtime of a search for an agent is the amount of obstacles it finds on its way to the goal.

VIII. Open Questions

ROS, or the Robot Operating System is a game changer in Robotic software development. At its core, ROS is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms. It is now widely used both in Academia and the Industry, and emerges as a standard in Robotics. Its open source dimension allows it to grow fast while keeping a quality standard maintained by competent team from various horizons. One of the main stack of ROS is the Navigation Stack [31]. They offer ready-to-use packages for autonomous navigation, in static and dynamic environments. They deemed the sole use of a reactive navigation approach not the best one and decided to implement a hybrid solution: they use Dijkstra for a first global planning to the goal to get a roughly approximate path if no obstacles are met along the way. From the path generated, they create waypoints from path pruning, and proceed to run the local planner Dynamic Window Approach (DWA) [32] to navigate to each waypoints, handling dynamic obstacles along the way.

There are certainly advantages to each methods. While CBS doesn't scale well, Enhanced CBS has proved [6] to be capable of handling a high number of agents in a Flight Management System, and Centralized methods lets one choose lower computational power for embedded agents and only focus on computational power for the central unit that performs the planning. On the other hand, reactive and replanning approaches requires more than a minimal configuration for the agent, and lacks determinism in terms of optimality and complexity. A hybrid approach may be solving a few of these issues and an extension to this research should assess those.

IX. Conclusion

Optimal single agent path finding is tractable (e.g. using the A* algorithm) while optimal (according to flowtime, makespan). Centralized MAPF is NP-hard [5]. The choice of the algorithm to apply depends on various factors related to the instance of the problem. Felner et al. [33] showed an interesting comparison between the main centralized approaches. The results emphasize that there is no universal winner. However, as the authors confirm, a more systematic comparison between existing solvers is needed in order to understand which algorithm performs better given the initial problem settings. Moreover, when generalizing centralized MAPF to real-world scenarios different kinds of issues arise. According to Felner et al. [33], more research should be done to adapt existing MAPF solvers to real-world domain. For instance, classical MAPF formulations ignores the fact that robot movements are subject to kinematic constraints [34]. Distributed approaches assess these issues and solves the scalability issue of centralized approaches, but it comes with flaws: even though their CPU and memory requirements are significantly lower, existing decentralized methods (e.g. [35], [17], VI-B) are incomplete and provide no criteria to distinguish between problems that can successfully be solved and problems where such algorithms fail. Further, no guarantees are given with respect to the running time, the memory requirements, and the quality of the computed solutions. Local (reactive) avoidance also brings downside because it only reacts to obstacles by anticipating what will happen in the next few time units. In very constrained environments such as a corridor where two agents meet, one of them should either go backwards or have made the precomputed choice to let the other pass through first. These local minima are very dangerous as reactive algorithms would likely not being able to resolve such a situation. Removing the long term planning cripples the cooperative aspect of MAPF. Such limitations should be addressed in future research and should tackle the costs of the efficiency of dynamically decoupled (decentralized) planners.

References

[1] Abi See Steve Mussmann. Graph search algorithms. https://cs.stanford.edu/people/abisee/gs.pdf.

[2] S. Persson and Inna Sharf. Sampling-based a* algorithm for robot path-planning. *The International Journal of Robotics Research*, 33:1683–1708, 11 2014.

[3] Ronit Patel and Maharshi Pathak. Comparative analysis of search algorithms. 06 2018.

[4] Kozen D.C. Depth-first and breadth-first search. 1992.

[5] Jingjin Yu and Steven M. LaValle. Structure and intractability of optimal multi-robot path planning on graphs. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, AAAI'13, page 1443–1449. AAAI Press, 2013.

[6] Florence Ho, Ana Salta, Ruben Geraldes, Artur Goncalves, Marc Cavazza, and Helmut Prendinger. Multi-agent path finding for uav traffic management. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, pages 131–139, 2019.

[7] Giovanni Giardini and Tamas Kalmar-Nagy. Centralized and distributed path planning for multi-agent exploration. volume 3, 08 2007.

[8] Subhrajit Bhattacharya, Maxim Likhachev, and Vijay Kumar. Multi-agent path planning with multiple tasks and distance constraints. In *2010 IEEE International Conference on Robotics and Automation*, pages 953–959. IEEE, 2010.

[9] Klaus Schilling, J De Lafontaine, and Hubert Roth. Autonomy capabilities of european deep space probes. *Autonomous Robots*, 3(1):19–30, 1996.

[10] Shaull Almagor and M. Lahijanian. Explainable multi agent path finding. In *AAMAS*, 2020.

[11] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[12] Jean-Claude Latombe. *Robot motion planning*, volume 124 of *The Kluwer international series in engineering and computer science*. Kluwer, 1991.

[13] Ping Xuan and Victor Lesser. Multi-agent policies: From centralized ones to decentralized ones. pages 1098–1105, 01 2002.

[14] D. Corkill. Hierarchical planning in a distributed environment. In *IJCAI*, 1979.

[15] Hsu MC., Chang P.HM., Wang YM., and Soo VW. Multi-agent travel planning through coalition and negotiation in an auction. *Intelligent Agents and Multi-Agent Systems*, vol 2891, 2003.

[16] Mike Phillips and Maxim Likhachev. Sipp: Safe interval path planning for dynamic environments. In *2011 IEEE International Conference on Robotics and Automation*, pages 5628–5635. IEEE, 2011.

[17] David Silver. Cooperative pathfinding. In *Proceedings of the First AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'05, page 117–122. AAAI Press, 2005.

[18] Steven M. Lavalle. Rapidly-exploring random trees: A new tool for path planning. Technical report, 1998.

[19] Aaron Becker and Li Huang. Random tree, rapidly-exploring random tree and rapidly-exploring random tree star live demos. https://www.wolframcloud.com/objects/demonstrations/RapidlyExploringRandomTreeRRTAndRRT-source.nb, 2018.

[20] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *International Journal of Robotic Research - IJRR*, 30:846–894, 06 2011.

[21] Emilio Frazzoli. Principles of autonomy and decision making. https://ocw.mit.edu/courses/aeronautics-and-astronautics/16-410-principles-of-autonomy-and-decision-making-fall-2010/lecture-notes/MIT16_410F10_lec15.pdf.

[22] Devin Connell and Hung Manh La. Dynamic path planning and replanning for mobile robots using RRT. *CoRR*, abs/1704.04585, 2017.

[23] Y. Kuwata, J. Teo, S. Karaman, Gaston A. Fiore, Emilio Frazzoli, and J. How. Motion planning in complex environments using closed-loop prediction. 2008.

[24] V. R. Desaraju and J. P. How. Decentralized path planning for multi-agent teams in complex environments using rapidly-exploring random trees. In *2011 IEEE International Conference on Robotics and Automation*, pages 4956–4961, 2011.

[25] K. L. Anderson, G. L. Blankenship, and L. G. Lebow. A rule-based adaptive pid controller. In *Proceedings of the 27th IEEE Conference on Decision and Control*, pages 564–569 vol.1, 1988.

[26] Rainer Palm and Abdelbaki Bouguerra. Navigation of mobile robots by potential field methods and market-based optimization. 09 2011.

[27] Javier Alonso-Mora, Tobias Naegeli, R. Siegwart, and P. Beardsley. Collision avoidance for aerial vehicles in multi-agent scenarios. *Autonomous Robots*, 39:101–121, 2015.

[28] Mina Kamel, Javier Alonso-Mora, Roland Siegwart, and Juan I. Nieto. Nonlinear model predictive control for multi-micro aerial vehicle robust collision avoidance. *CoRR*, abs/1703.01164, 2017.

[29] Ashwin Bose. Multi agent path planning. https://github.com/atb033/multi_agent_path_planning, 2019.

[30] Erwin Lejeune. Pymapf: a python library for multi-agent pathfinding. https://github.com/APLA-Toolbox/pymapf, 2021.

[31] Navigation Working Group. Ros navigation stack. http://wiki.ros.org/navigation.

[32] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. The dynamic window approach to collision avoidance. *Robotics Automation Magazine, IEEE*, 4:23 – 33, 04 1997.

[33] Ariel Felner, R. Stern, S. E. Shimony, Eli Boyarski, Meir Goldenberg, G. Sharon, Nathan R Sturtevant, G. Wagner, and Pavel Surynek. Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges. In *SOCS*, 2017.

[34] Wolfgang Hönig, T. K. Satish Kumar, Liron Cohen, Hang Ma, Hong Xu, Nora Ayanian, and Sven Koenig. Multi-agent path finding with kinematic constraints. In *Proceedings of the Twenty-Sixth International Conference on International Conference on Automated Planning and Scheduling*, ICAPS'16, page 477–485. AAAI Press, 2016.

[35] Ko-Hsin Wang and Adi Botea. Fast and memory-efficient multi-agent pathfinding. pages 380–387, 01 2008.