Shadoks Approach to Low-Makespan Coordinated Motion Planning

LOÏC CROMBEZ, LIMOS, Université Clermont Auvergne, France
GUILHERME D. DA FONSECA, LIS, Aix-Marseille Université, France
YAN GERARD, LIMOS, Université Clermont Auvergne, France
ALDO GONZALEZ-LORENZO, LIS, Aix-Marseille Université, France
PASCAL LAFOURCADE, LIMOS, Université Clermont Auvergne, France
LUC LIBRALESSO, LIMOS, Université Clermont Auvergne, France

This paper describes the heuristics used by the Shadoks[1] team for the CG:SHOP 2021 challenge. This year's problem is to coordinate the motion of multiple robots in order to reach their targets without collisions and minimizing the makespan. It is a classical multi agent path finding problem with the specificity that the instances are highly dense in an unbounded grid. Using the heuristics outlined in this paper, our team won first place with the best solution to 202 out of 203 instances and optimal solutions to at least 105 of them. The main ingredients include several different strategies to compute initial solutions coupled with a heuristic called Conflict Optimizer to reduce the makespan of existing solutions.

CCS Concepts: • **Theory of computation** → **Design and analysis of algorithms**; *Computational Geometry*.

Additional Key Words and Phrases: multi agent path finding, heuristics, motion planning, shortest path

ACM Reference Format:
Loïc Crombez, Guilherme D. da Fonseca, Yan Gerard, Aldo Gonzalez-Lorenzo, Pascal Lafourcade, and Luc Libralesso. 2022. Shadoks Approach to Low-Makespan Coordinated Motion Planning. *ACM J. Exp. Algor.* 0, 0, Article 0 (2022), 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

We explain some heuristics used by the Shadoks team to win first place in the CG:SHOP 2021 challenge that considers a coordinated motion planning problem in the two-dimensional grid $\mathbb{Z}^2$. The goal is to move a set of n labeled unit squares called *robots* between given start and target grid cells without collisions.

More formally, the input consists of a set of *obstacles* O and a set of n *robots* $R = \{r_1, \ldots, r_n\}$. Each obstacle is a lattice point and each robot r_i is a pair of lattice points s_i, d_i respectively called *start* and *target*. A path P_i of length m is a

[1]The team name comes from the animated television series Les Shadoks https://en.wikipedia.org/wiki/Les_Shadoks.

Authors' addresses: Loïc Crombez, loic.crombez@uca.fr, LIMOS, Université Clermont Auvergne, Clermont-Fd, France; Guilherme D. da Fonseca, guilherme.fonseca@lis-lab.fr, LIS, Aix-Marseille Université, Marseille, France; Yan Gerard, yan.gerard@uca.fr, LIMOS, Université Clermont Auvergne, Clermont-Fd, France; Aldo Gonzalez-Lorenzo, aldo.gonzalez-lorenzo@univ-amu.fr, LIS, Aix-Marseille Université, Marseille, France; Pascal Lafourcade, pascal.lafourcade@uca.fr, LIMOS, Université Clermont Auvergne, Clermont-Fd, France; Luc Libralesso, luc.libralesso@uca.fr, LIMOS, Université Clermont Auvergne, Clermont-Fd, France.

sequence of $m+1$ lattice points $p_i(t)$ for a time $t = 0, \dots, m$. A solution of makespan m is an assignment of *paths* P_i of length m to each robot r_i satisfying the following constraints.

(1) $p_i(0) = s_i$ and $p_i(m) = d_i$ for $1 \leq i \leq n$,
(2) $\|p_i(t) - p_i(t-1)\| \leq 1$ for $1 \leq i \leq n$ and $1 \leq t \leq m$,
(3) $p_i(t) \notin O$ for $1 \leq i \leq n$ and $0 \leq t \leq m$,
(4) (*collision constraint*) $p_i(t) \neq p_j(t)$ for $i \neq j$ and $0 \leq t \leq m$, and
(5) (*overlap constraint*) if $p_i(t) = p_j(t-1)$, then $p_i(t) - p_i(t-1) = p_j(t) - p_j(t-1)$. This constraint comes from the robots being square shaped, in order to avoid corner collisions in the continuous movement of the robots.

The objective of the problem is to minimize the makespan[2] m. A trivial lower bound is obtained by ignoring constraints (4) and (5) and finding shortest paths. For more details about the problem, see the overview [9] and the related paper [8].

The challenge CG:SHOP 2021 provided 203 instances containing between 10 and 9000 robots, out of which 202 of our solutions were the best ones among all the 17 teams who participated. To our surprise, we succeed in finding 105 solutions that match the trivial lower bound.

Literature review. The *multi-agent path finding* problem (MAPF) has been well studied over the last 20 years. This problem occurs in many industrial applications that involve agents that have to reach destinations without colliding with each other [19], for instance, in automated warehouses [16], autonomous vehicles, and robotics [2]. Well-studied approaches include search-based solvers (for instance: HCA* [24]) that route the agents one by one according to a predefined order. When an agent is routed, it "reserves" times and locations. Then, the algorithm routes the next agent until every agent is routed. Such search-based solvers can also route many robots at the same time, thus having for each time-step up to 5^n possibilities where n is the number of robots simultaneously routed together (for instance, Enhanced Partial Expansion A* [13]). There also exist rule-based solvers that identify scenarios and apply rules (predetermined patterns) to move agents (for instance, the push-and-rotate algorithm [31]). Some algorithms model the MAPF problem using network flows [28], integer linear programs [33], or SAT instances [26, 27].

One of the most popular methods to solve the MAPF problem is the *conflict-based search* (CBS) [23]. This method starts by solving a relaxation of the original problem, in which agent collisions are ignored. This relaxation is relatively easy to solve, as it consists of running a shortest-path algorithm for each agent. If the resulting plan contains a time t and coordinate c where two agents r_1, r_2 collide, then the algorithm forbids either r_1 or r_2 from being at the coordinate c at time t. This results in a search tree that is explored until it is depleted (thus finding the optimal solution for the problem). This method has been reported to achieve excellent results and multiple improvements have been made. Some examples of improvements include a better estimate of the remaining cost [10], merge-and-restart and conflict prioritization [3], some suboptimal variants [1], and some techniques such as a branch-and-cut-and-price algorithm [15]. For more information about the multi-agent path finding problem, we refer the reader to surveys about the MAPF variants and instances [25] and about the MAPF solvers [11].

At the beginning of the challenge, we tried some of the aforementioned approaches (notably CBS) to solve the challenge instances. To our surprise, CBS did not perform well. Indeed, the challenge instances are much denser than the ones in the literature. The classical MAPF instances are usually sparse in the number of agents (there are from 2 to 120 agents placed in grids with over 100,000 cells in the classical instances [25], while challenge instances contain hundreds or thousands of agents placed in grids never larger than 100×100). This structural difference has a dramatic

[2]The challenge also considered the objective of minimizing the sum of the distances, but we did not optimize our solutions for this version of the problem.

effect on the performance of CBS in our experiments. We tested the open-source MAPF solver libMultiRobotPlanning [3] with a memory limit of 16GB and it fails to find solutions for most of the challenge instances (except for some instances with less than 50 robots, namely `small_000` and `small_free_000`).

In this paper, we present several new ideas we used in the competition. We implemented a plethora of different techniques to find initial solutions to different kinds of instances, depending mostly on the size and density of the instance, as well as the presence of obstacles. A key element of our approach was to improve these initial solutions into low-makespan solutions. For this task, we introduced a novel heuristic that we call the *Conflict Optimizer*, which may very well be adapted to other optimization problems. Our code is available on github https://github.com/gfonsecabr/shadoks-robots. Some of our results in the competition were obtained by running our solvers on an instance for several weeks.

The remainder of the paper is organized as follows. In Section 2 we consider the problem of obtaining feasible solutions of moderate makespan. These solutions are optimized later on, with the techniques described in Section 3. Details on implementing the algorithms are described in Section 4. Section 5 describes the results we obtained for some challenge instances and presents a comparison with the strategies used by other teams. Concluding remarks are presented in Section 6.

2 INITIAL SOLUTIONS

Feasibility is guaranteed for the challenge instances since the number of obstacles is finite and every start and target are located in the unbounded region of space. In this section, we show how to obtain feasible solutions with a moderate makespan. We divide the heuristics in two categories. In Section 2.1, we compute the solution one step at a time, considering multiple robots simultaneously. In Section 2.2, we compute the solutions one robot at a time.

The heuristics of the first category are not guaranteed to find a solution, but when they do they often find solutions of lower makespan than those of the second category. The algorithms of the second category are guaranteed to find a solution, but the resulting makespan may potentially be high.

2.1 Step by Step Computation

The problem of finding a solution for coordinated motion planning in a given number of steps can be modeled as an Integer Linear Problem (ILP) or equivalently as a SAT problem (see [26, 27] and references therein). While applying such an approach is intractable even for small instances, it can be adapted to find an initial solution. The general idea of the *Greedy* solver is to plan only a small number k of steps for the robots such that the overall distance to the targets decreases as much as possible. Then, we move each robot by one step and repeat this procedure until all robots reach their respective targets.

Our ILP model considers a Boolean variable $x_{r,P}$ for each robot r and for each possible path P of length k starting at the position of the robot. Constraints of having one and only one path per robot and avoiding obstacles and collisions between robots are easily expressed as linear inequalities. The objective function we maximize is the sum of all the variables with weight

$$\text{weight}(r, P) = \Big(\delta_r(p(0)) - \delta_r(p(k))\Big) \cdot \Big((\delta_r(p(0)))^2 + 1\Big),$$

where $p(0)$ and $p(k)$ are the first and the last positions of the path P and $\delta_r(p)$ is the obstacle-avoiding distance from a point p to the target of the robot r. The first factor encourages the solution to push the robots towards their targets, since it is better to get closer to the target. The second factor prioritizes moving robots that are farther from their targets,

[3] https://github.com/whoenig/libMultiRobotPlanning

and we add one so that robots that are already at their target position are encouraged to remain there (otherwise, since their weight is zero, they would be free to move to any position).

In practice, we set $k = 3$ and we only perform the first step of the planned moves, so that the robots can *anticipate* the moves of the other robots. Using the CPLEX [7] library to solve these problems, we can handle instances with up to roughly 200 robots. Note that this Greedy algorithm is not guaranteed to find a solution. For example, it fails to solve instances with *corridors*. The reason is that two robots may enter an infinite loop pushing each other back and forth inside the corridor, as the weight function gives higher priority to the robot that is currently farther away from its target, which may alternate between the two robots.

2.2 Robot by Robot Computation

The algorithms in this section compute the solution one robot at a time using an A* search in two steps. Each robot is assigned an intermediate position called storage and defined later on. First, we use the A* search to compute paths to the storage, one robot at a time. Then, we use A* search again, to compute a path from the initial position to the target position. The order of the robots to compute these paths is an important element that will be detailed later. The path to the intermediate position serves to guarantee that a feasible path exists. Next, we present more details of this approach, and invite the reader to see Figure 1 for an illustration of the definitions.

We refer to the *bounding box* as an integer axis-aligned rectangular region containing all the start positions, target positions, and obstacles inside its strict interior (not on the boundary). Given a set of obstacles and a bounding box, the *depth* of a position p is the minimum obstacle-avoiding distance from p to a position outside the bounding box. One may note that, being in the strict interior of the bounding box, the start and target positions have depth at least two. By increasing the size of the bounding box, we can set the minimum depth of the start and target positions, as well as the obstacles to any value $b \geq 2$. We refer to the positions of depth less than b as the *border* of the bounding box. Indeed the subsequent algorithms require a way for a robot to go around all start and target positions and the border provides such a way. A larger value of b prevents congestion among robots traveling through the border and may help reduce the makespan. However, setting b to a large value may also increase the makespan by making robots travel too far away from the start position before moving to the target position. A value of b from 2 to 4, depending on the algorithm and the instance, seems to be a good compromise in our experience, where instances with more robots tend to benefit from slightly larger values of b.

All algorithms in this section are based on a *storage network* N. A storage network is a set N of positions outside a predetermined bounding box such that for every position p in N, there exists a path that avoids all other positions of N and goes from p to some point in the bounding box. Each robot r_i is assigned to a distinct element of N, called the *storage* of r_i.

Initially, we set the path of each robot to be stationary at the start position. We sort the robots by *increasing start depth* and for each robot in order, we use A* search to find the shortest path from start to storage, replacing the previous stationary path. The order by which the robots are sorted guarantees that such a path exists.

The A* search happens in 3-dimensional space, where each robot state has integer coordinates of the form (x, y, t) for position coordinates x, y and time t. There are 5 possible *movements*, all of which increase t by one unit. One movement keeps the position x, y unchanged, while the other 4 movements increment or decrement one of the two coordinates. A movement is feasible if it does not violate any of the problem constraints, considering the currently defined path of the other robots.

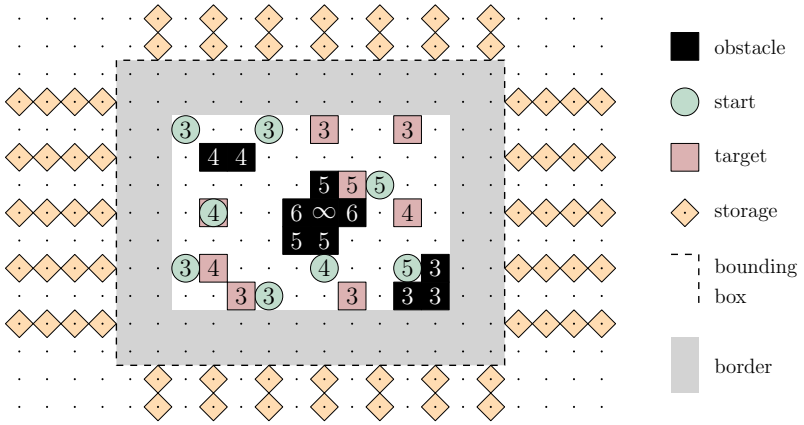

Fig. 1. Bounding box and storage network, with the depth of start positions, target positions, and obstacles written inside the cells.

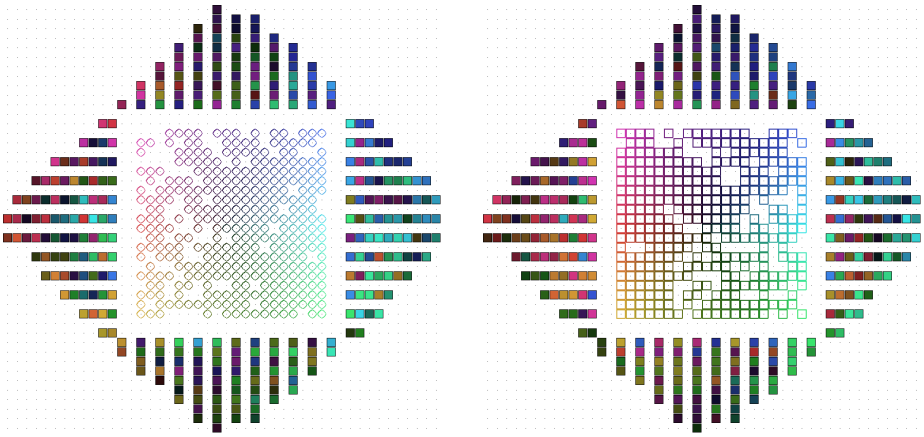

Fig. 2. Cross storage network for the `small_free_016` instance colored based on start and target locations, respectively.

After finding paths from start to storage for every robot, we proceed to the next phase of the algorithm. We now sort the robots by *decreasing target depth*. Again, the order of the robots guarantees that a path from storage to target exists. However, we do not compute such a path. Instead, we compute a path from start to target directly, whose existence is guaranteed by the existence of a path from start to storage and another one from storage to target. The following paragraphs describe the design of four different storage networks.

Cross. In the *Cross* strategy, we define the storage network N as the set of columns of even x coordinate lying directly above or below the bounding box and the set of rows of even y coordinate lying directly to the left or right of the bounding box, hence the name Cross. Then, we compute a maximal cardinality matching between the robots and N. We tried both minimum-weight matching and greedy matchings, minimizing a weight function that considers the distance from start to storage as well as the distance from storage to target. In the greedy matching version, robots are assigned a storage ordered by decreasing start-to-target distance. The result is represented in Figure 2. Start positions are represented by a hollow circle, target positions are represented by a hollow square and the storage positions are

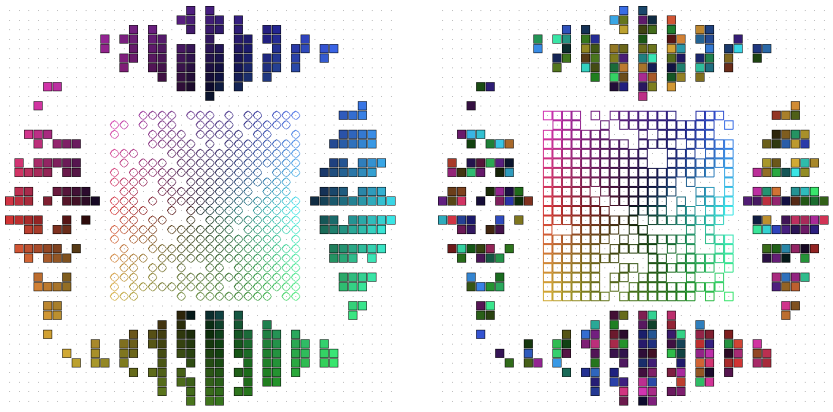

Fig. 3. Cootie Catcher storage network for the `small_free_016` instance colored based on start and target locations, respectively.

represented by a slanted solid square. Each robot is assigned a different color following a rainbow pattern based on either the start or the target position to illustrate how the storage assignments are made.

Cootie Catcher. The previous strategy works very well for small or sparse instances. However, the different directions of the flow of robots from start to storage make the solutions inefficient for large and dense instances. The *Cootie Catcher* strategy computes the storage using only the start location, in order to better exploit parallel movement of the robots. The storage network shape consisting of four diamonds is presented in Figure 3. When applied to instances without obstacles, the strategy is guaranteed to find a path from start to storage using at most $w/2 + O(1)$ steps, where w is the largest bounding box side. Surprisingly, this strategy also works well for many instances with obstacles.

Dichotomy. The weakness of the previous method is that robots may be assigned storage in a location that is opposite to the direction from start to target. Furthermore, the parallel movement of the robots makes it unlikely that a robot will be able to take any significant shortcuts before it reaches the storage. In order to exploit parallel movements while taking the target location into consideration, we developed the *Dichotomy* strategy. The strategy only works for instances without obstacles.

We translate the coordinate system so that the origin is the center of the bounding box. The robots are partitioned into two sets called *left side* and *right side* according to the sign of the x-coordinate of the target location. Left-side robots are assigned storage with positive x-coordinate while right-side robots are assigned storage with negative x-coordinate, as represented in Figure 4.

The algorithm performs the following steps, described only for the robots with non-negative y-coordinate for simplicity, as the other half is analogous.

(1) Each robot goes up from start position (x, y) to position $(x, 2y)$.
(2) If the robot target is on the right side, the robot moves up one more row. At this point, the even y rows contain left-side robots and the odd y rows contain the right-side robots.
(3) If a right-side (resp. left-side) robot is still inside the bounding box, then it moves to the right (left) as far as needed to leave enough space for the other robots to its left on the same row to move out of the bounding box. Otherwise, a right-side (left-side) robot moves right (left) in order to leave enough space for the robots on the same row to move to a position of positive (negative) x-coordinate.

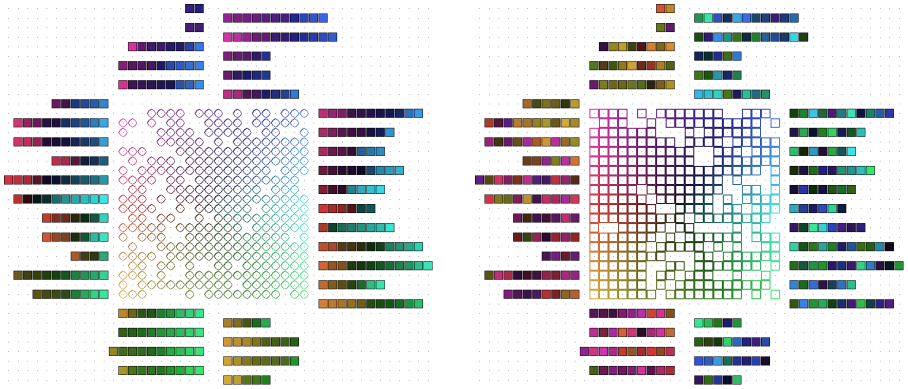

Fig. 4. Dichotomy storage network for the `small_free_016` instance colored based on start and target locations, respectively.

Going from start to storage takes at most $3w/2 + O(1)$ movements for a $w \times w$ bounding box. Instead of sorting the robots by decreasing target depth as usual, we sort the robots by absolute value of the target x-coordinate and then determine the paths from start to target using A* as usual.

Escape. This strategy focuses on instances with obstacles, especially on dense instances where the obstacles create bottlenecks to the passage of the robots. The goal of the *Escape* strategy is to move all robots outside the bounding box as quickly as possible. To do so, we move the robots by blocks (a *block* is a large polyomino of robots), making efficient use of parallel movements.

The Escape strategy divides the non-obstacle grid cells inside the bounding box into *layers* and each layer is partitioned into blocks. The first layer is defined so that the grid cells located in it can reach the outside of the bounding box by following a straight line. To make sure that there are no intersections between any two paths taken by robots located in the first layer, the layer is divided in blocks. The robots in each block move at the same time in parallel. Then, the second layer is defined. It consists of blocks adjacent to the first layer, ideally containing as many cells as possible, that will move in a straight line to the first layer. Again, all robots located in the same block will move towards the first layer in parallel motion. Then, in the same way, a third layer is defined, consisting of blocks that will move into the first or second layer. Layers are added until every robot is located in a layer, as represented in Figure 5.

We used a greedy algorithm to define the layers. To compute layer k, the algorithm iteratively looks at each block b_i of the previous layers and looks for the largest block not yet assigned to a layer that can move to b_i in a straight line. We partially redefined the layers manually for the most complicated instances and the ones where the algorithm produced unsatisfying results. As shown in Figure 6, outside the bounding box, only two out of every three consecutive rows and columns are used for storage in order to create traveling corridors for the future movements.

3 IMPROVING SOLUTIONS

In this section, we discuss the two heuristics that we used to reduce the makespan of a given feasible solution. The first heuristic makes local changes to the solution, which remains feasible throughout the process, and possibly reduces the makespan. The second heuristic destroys the feasibility of the solution and either finds another solution of reduced makespan, or no feasible solution at all. Throughout, let m be the makespan of the input solution.

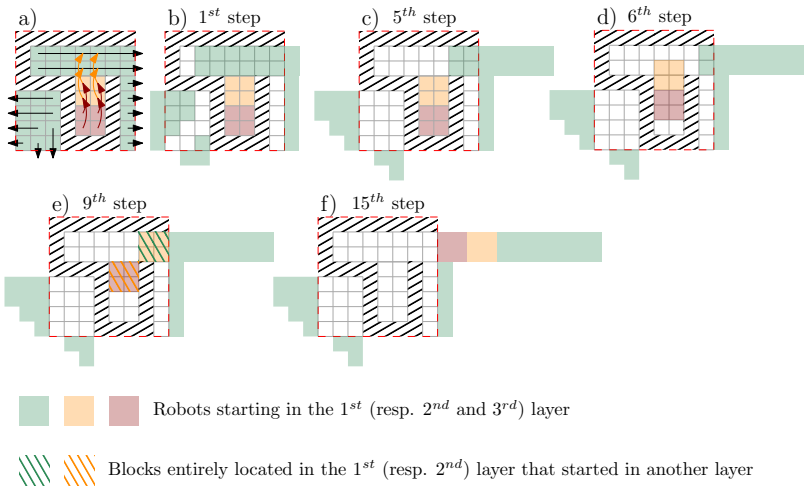

Fig. 5. Escape strategy. a) In green (resp. orange and red), the first (resp. second and third) layer. The orange arrows show where the second layer will move, the red arrows show the same for the third layer. b-c) The robots in the first layer are moving, but the robots of the second and third layer are still stuck. d) The robots from the second layer are now free to move towards the first layer, this also allows the robots from the third layer to move. e) The robots that used to be located in the third layer, are now located in the second layer and are waiting for their path to be clear of robots. f) Final placement outside the bounding box.

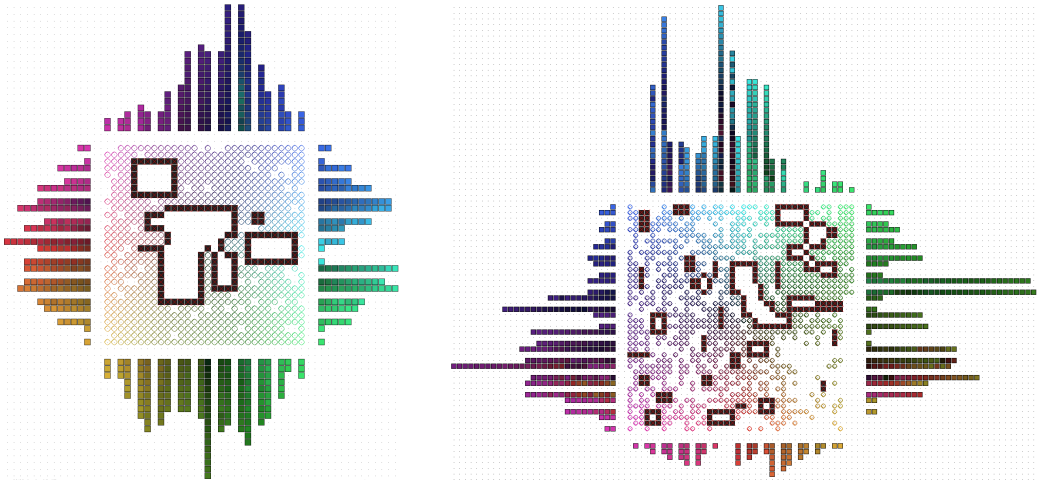

Fig. 6. Escape storage network for the `medium_007` and `buffalo_003` instances, respectively.

Feasible Optimizer. The idea of the *Feasible Optimizer* is the following. We iteratively remove the path of a robot r from the solution, and then use the A* algorithm to find a new (hopefully different) path for r. The A* algorithm may be tuned in several ways to produce different paths, and we do so in such a way that the makespan of the solution never increases and also that a robot is only allowed to move at time m if it already did so in the original path. This way, not only the makespan but also the number of robots moving at time m never increase. Next, we list some examples on how to modify the A* search.

- Find the path from start to target that reaches the target as quickly as possible but break ties using the sum of random weights given to each grid cell the robot passes through.
- Reversing the direction of time and then finding a path from target to start that reaches the start as quickly as possible. In the original time direction, that means that the robot will remain at the start for as long as possible.
- In the reversed case, force the robot to stay at target for a certain number of steps.

Conflict Optimizer. The previous optimization strategy may take very long to reduce the makespan because it relies on chance to move a robot away from the path another robot would rather take. Next, we describe a more aggressive approach that leaves the feasible solution space and works far better than we expected. The algorithm uses a modified A* search that allows for a robot to go over another robot's path, which we call a *conflict*. We start by creating a *queue* with all the robots that move at makespan time m. While the queue is not empty, we repeat the following procedure for a robot r popped from the front of the queue.

(1) Erase r's path.
(2) Find a path for r from start to target that arrives no later than time $m - 1$ and minimizes the sum of the weights of the conflicting robots.
(3) Add all conflicting robots to the queue.

Let $q(r)$ be the number of times r has been popped out of the queue. We define the *weight* of a robot r as $1 + (q(r))^2$. This weight function gives some incentive to converge to a feasible solution, preventing robots from repeatedly finding paths that conflict with each other. Notice that r's path is only cleared once it is popped out of the queue. This way, a robot r tries to prevent parts of its previous path from being used by other robots, despite the fact that r will need to find a new path. This little detail makes a big difference, as more likely a big portion of the new path of r will coincide with its previous path.

For sparse or small instances, the Conflict Optimizer can even be used to compute solutions from scratch by choosing an initial makespan and putting all the robots in the queue. This approach fails to find solutions for most instances, though. Hence, we used the Conflict Optimizer only to optimize solutions obtained using the algorithms described in Section 2.

4 IMPLEMENTATION ASPECTS

In this section, we describe different techniques used to efficiently implement and apply the previously described heuristics. All heuristics have been executed multiple times, extensively using randomization whenever possible. Furthermore, the problem has several types of symmetry that have been exploited to find more (potentially better) solutions. First, we note that applying rotations (by multiples of 90 degrees) does not change the problem. Second, we note that reversing the start and target positions also does not change the problem. Hence, we applied different rotations as well as reversing the start and the target of the instances.

Multiple executions were used to produce over ten thousand solution files total. All solution files were saved with a timestamp on the file name. That allowed us to find initial solutions that would optimize better. Even though we only optimized the solutions for makespan, this large volume of solutions allowed us to obtain solutions with a sufficiently low sum of the distances to obtain the third place in that category. Developing tools to efficiently organize and view all of these solutions was an important part of the team strategy. These tools include a viewer, an editor, and a utility to list and copy solution files meeting certain criteria.

The heuristics have been coded in C++, most of which is available publicly on github[4]. The tools have been coded in python. We executed the code on several Linux machines, both personal computers and high performance computing clusters at the LIS and LIMOS laboratories.

The A* search is in the heart of most of our heuristics. Hence, a lot of work has been done to improve its performance. Sometimes we used deterministic A*, breaking ties by the coordinates of the position, but more often we used a randomized A* algorithm where ties are broken by the sum of the weights of the positions in the path, which are assigned randomly. The A* algorithm needs a distance function as a lower bound and a collision detection, which are described next.

Distance queries. The A* algorithm is guided by a lower bound to the distance to target. In the case without obstacles, the lower bound we used is simply the L_1 distance, which can be calculated in $O(1)$ time. While this lower bound is still valid for instances with a set O of obstacles, it is inefficient because it does not take the obstacles into account. Instead, we used the obstacle-avoiding L_1 distance.

Calculating the obstacle-avoiding L_1 distance from scratch is a slow process. Since this computation happens many times during the execution of our heuristics, it is essential to be able to compute it quickly. To this purpose, we need to use a data structure to compute the distance query(p) from a query point p to a fixed target (in our case, given at preprocessing time). Existing data structures for the problem [5, 6] seem hard to implement. Instead, we designed a simple data structure that takes $O(\log w)$ query time for obstacles inside a square of side w. The storage requirement may potentially be close to w^2, but in our case it is significantly less, generally close to $O(|O|)$.

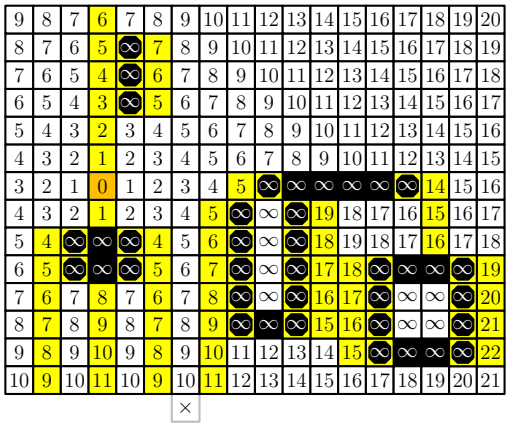

Fig. 7. Distance with obstacles. Only yellow positions are stored. Black regions represent obstacles and the corners are painted yellow if the value of infinity is stored.

Given two consecutive points $(x, y), (x + 1, y)$ on the same line we have query(x, y) − query$(x + 1, y) \in \{-1, 0, 1\}$. Furthermore, if x is to the left (resp., right) of the bounding box, then query(x, y) − query$(x + 1, y) = 1$ (resp. $= -1$). Hence, for each line y in the bounding box we only store the points (x, y) such that query$(x, y) \neq$ (query$(x - 1, y)$ + query$(x + 1, y))/2$, as shown in Figure 7. All the remaining queries for line y can be calculated by interpolating or extrapolating these stored values, which can be located in $O(\log w)$ time using binary search on a sorted vector.

[4] https://github.com/gfonsecabr/shadoks-robots

Queries for a point (x, y) above or below the bounding box are answered by using the closest line of the bounding box and the fact that $\text{query}(x, y) = \text{query}(x, y') + |y - y'|$. For example, to query the square marked by a $\times$ in Figure 7, we would add 1 to go one line above, and then interpolate between 9 and 11 to obtain $1 + (9 + 11)/2 = 11$.

Collision detection. Fast collision detection between two robots is a key point to the performance of the A* algorithm. We used an internal storage Hopscotch hash table implemented by Thibaut Goetghebuer-Planchon and distributed under the MIT license [12]. Given a position and time, we stored the robot in that position (or the list of robots in that position for the Conflict Optimizer). Hence, collision detection reduces to a small number of hash table lookups.

Avoiding Conflict Optimizer stalls. The Conflict Optimizer is arguably the most significant contribution of this work. However, it may stall at sub-optimal solutions. To reduce this problem, we may use the following approaches. (i) Reverse start and target as well as the paths in the solution. (ii) Use the Feasible Optimizer to shuffle the solution. (iii) Use randomized paths in the A* search. (iv) Insert the robots that conflict with the same robot in the queue using a random order.

5 EXPERIMENTAL RESULTS

Tables 1 and 2 show the makespan obtained using different heuristics on some selected challenge instances and the makespan lower bound. The Feasible Optimizer column corresponds to the best optimization it obtained starting from different solutions. The Conflict Optimizer column corresponds to the optimization of the solution obtained by the Feasible Optimizer.

Figure 8 shows the improvement obtained by the Conflict Optimizer over one hour of execution as well as over several weeks. In the case of one hour of execution, we did not attempt to avoid the conflict optimizer stalls. However, the data represented at the bottom of the figure has been produced during the challenge, and is subject to both automatic and manual attempts to avoid stalls. We note that near the end of the challenge, some solutions kept improving very slowly with the Conflict Optimizer: some instances with a few thousand robots such as `sun_007`, `clouds_008`, and `large_free_007` were consistently giving 1 unit of makespan improvement for every 10 to 20 hours of computation throughout weeks.

Comparison with other teams. The two other teams `UNIST` [32] and `gitastrophe` [17] on the podium of the CG:SHOP 2021 challenge used a two-phase strategy similar to ours: first compute an initial solution and then optimize it.

The other two teams also used initial solutions computed through a storage network. The existence of a solution is also guaranteed by using the depth of the start and target positions. We noticed that `gitastrophe` used the same trick we did: first compute a partial solution going from start to storage and then compute new paths going from start to target since the existence of such paths is guaranteed. `gitastrophe` also used a minimum weight matching to assign each robot to a storage position. The main difference between the teams during this phase is in the choice of the storage network (points within pairwise L_∞ distance at least 2 for `gitastrophe` and points having even coordinates for `UNIST`). Among the different storage networks that we used (Cross, Cootie Catcher, Dichotomy, Escape), we noticed that none of the four is always the best one.

For the optimization of an initial solution, there are again some similarities, but the differences are more significant. The standard strategy is to randomly remove the paths of a sample of robots before recomputing them with the hope of an improvement. `gitastrophe` experimented with different ways to choose the samples: according to their makespan, relative distance, or conflicts with a given robot. `UNIST` used a sample of only one robot as we did in our *Feasible*

			initial solution				optimizer		lower
instance	n	w	Greedy	Cross	Cootie C.	Dichotomy	Feasible	Conflict	bound
small_free_002	40	10	**17**	22	27	22	17	**15**	**15**
small_free_003	70	10	**20**	31	27	26	20	**16**	14
small_free_010	200	20	**34**	46	54	45	33	**32**	**32**
small_free_015	280	20	·	**60**	68	65	51	**40**	32
small_free_016	320	20	**63**	68	77	68	60	**47**	36
medium_free_007	630	30	148	**89**	103	95	81	**60**	52
medium_free_009	800	40	**93**	97	124	109	81	**71**	**71**
medium_free_012	1000	50	·	**114**	125	127	96	**94**	**94**
microbes_004	1250	50	·	**132**	159	135	125	**91**	**91**
buffalo_free_003	1440	60	·	**149**	165	158	125	**87**	78
london_night_005	1875	50	·	179	190	**173**	157	**124**	92
universe_bg_005	2000	50	·	194	198	**177**	173	**141**	82
galaxy_c2_008	3000	100	·	**198**	258	234	168	**163**	**163**
large_free_004	3938	75	·	274	276	**256**	240	**204**	127
large_free_005	5000	100	·	**260**	316	293	252	**184**	**184**
large_free_007	6000	100	·	**297**	343	325	295	**236**	189
sun_009	7500	100	·	424	395	**361**	354	**345**	187
large_free_009	9000	100	·	514	440	**391**	378	**374**	182

Table 1. Makespan of different heuristics for selected instances without obstacles.

			initial solution				optimizer		lower
instance	n	w	Greedy	Cross	Cootie C.	Escape	Feasible	Conflict	bound
small_005	63	10	**27**	28	32	37	25	**20**	18
sun_000	143	20	**32**	39	46	61	29	**27**	**27**
small_011	183	20	**56**	60	70	67	48	**40**	37
small_016	276	20	·	**67**	72	79	57	**43**	36
medium_005	407	30	·	119	110	**106**	94	**74**	58
london_night_002	825	50	·	**149**	162	165	142	**94**	84
microbes_002	958	50	·	**111**	135	173	97	**89**	**89**
clouds_001	912	50	·	**117**	138	159	94	**83**	**83**
medium_014	1165	40	·	180	**161**	180	161	**151**	73
algae_004	1113	50	·	**139**	160	191	121	**84**	79
buffalo_004	1404	60	·	**136**	164	195	120	**104**	**104**
large_003	1906	100	·	**172**	224	250	**154**	**154**	**154**
large_004	2034	100	·	431	391	**381**	·	**381**	185
large_005	3223	75	·	398	**310**	317	·	**299**	141
universe_bg_007	3820	100	·	**224**	289	323	202	**184**	**184**
large_007	4706	100	·	753	497	**491**	497	**471**	215
microbes_008	5643	100	·	**329**	359	425	322	**279**	188
algae_009	7311	100	·	500	**439**	441	·	**421**	176
large_009	8595	100	·	398	**387**	566	·	**352**	176

Table 2. Makespan of different heuristics for selected instances with obstacles. Note that the Dichotomy algorithm does not work with obstacles and has been replaced by the Escape algorithm.

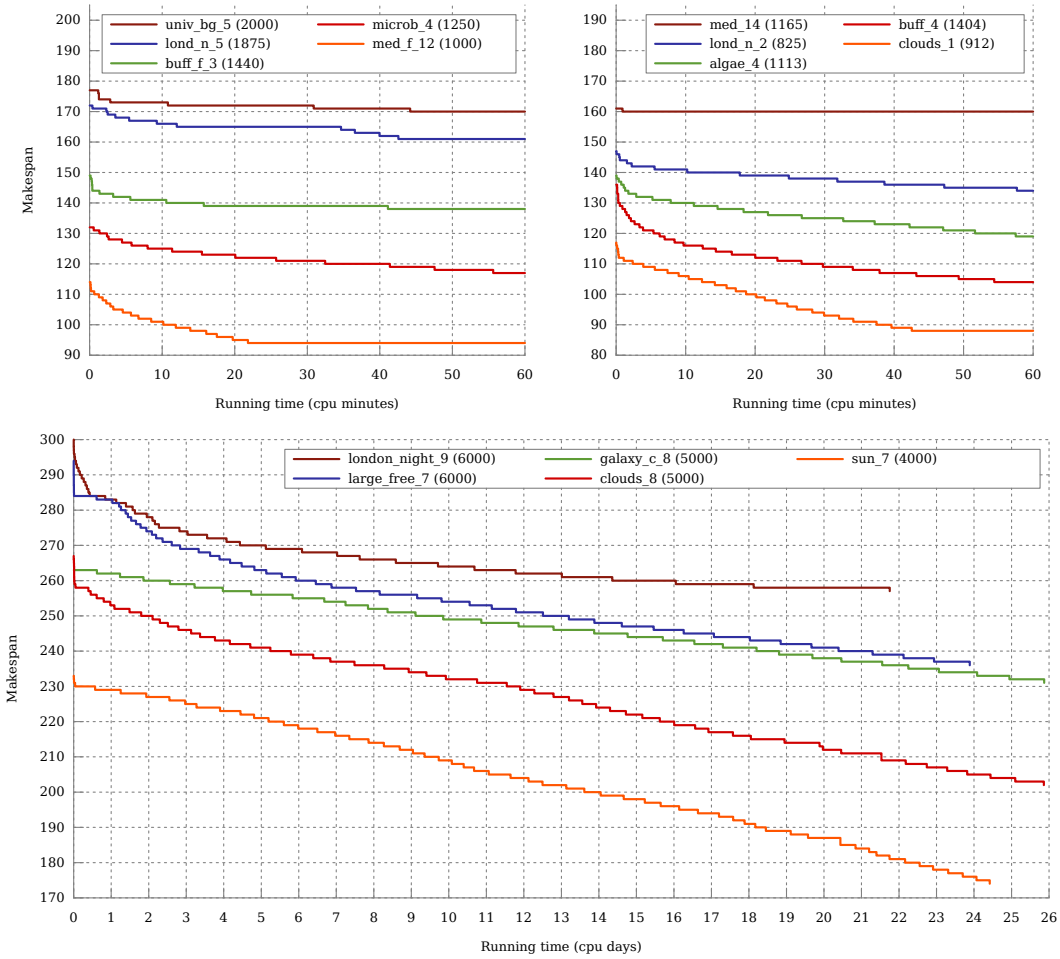

Fig. 8. Improved makespan over computation time using the Conflict Optimizer.

Optimizer but `UNIST` also added an original simulated annealing optimization step. Neither `UNIST` nor `gitastrophe` have used an optimization algorithm where the sample of robots to recompute evolves dynamically as in our *Conflict Optimizer*. The reason could be that this strategy destroys the solution feasibility without any guarantee of recovery. However, the Conflict Optimizer is probably the main ingredient that gave us a dramatic advantage over the other teams.

6 CONCLUSION AND PERSPECTIVES

We developed several algorithms to solve the coordinated motion planning instances of the CG:SHOP2021 challenge, obtaining low-makespan solutions. Considering how dense the instances are, we were surprised that our algorithms found optimal solutions (matching the trivial lower bound obtained by routing each robot independently) for 105 out of 203 challenge instances. Furthermore, our team obtained the best solution found for 202 out of 203 challenge instances.

We attribute this success to the large variety of algorithms that we used. The 3-step *Greedy* algorithm cannot solve most instances, but when it does, it provides solutions that are better than the robot by robot solutions. In the robot by robot solutions, we used several different kinds of storage networks, adapting to the peculiarities of each instance. Besides the four storage networks described herein, a few others have been tested, including the possibility of storage inside the bounding box, as well as restricting the movement direction in chosen cells in order to create one-way lanes. These other networks did not yield any noticeable benefit, though. Optimizing the solutions is a key ingredient. The *Feasible Optimizer* helps, but about half of the optimal solutions that we found were produced by the *Conflict Optimizer*. We were surprised how well this algorithm works, while not surprised that the higher the robots density is, the lower is its efficiency.

We should keep in mind that the algorithms to find initial solutions have been designed for the challenge instances. A natural question is to determine if the strategy that we used remains suitable for other types of instances, such as the ones arising in the following three scenarios:

(1) The start and the target positions are located in different areas. For example, consider an instance with the start positions in the square $[-size, -1] \times [-size, size]$ and the target positions in $[1, size] \times [-size, size]$, where $size$ is a parameter determining the size of the different storage spaces. We separate the start and the target positions with a vertical line of obstacles at coordinates $(0, y)$ for $|y| > door$, where $door$ is a parameter determining the dimension of the corridor connecting them. This scenario corresponds to the practical problem of moving the content of a warehouse to another location close by.
(2) A set of obstacles form a bounding box around the start and the target positions. An example of such instance is an instance with obstacles around the square $[0, size] \times [0, size]$ and all the start and the target positions located in this square. This scenario differs from the challenge instances since there is no way to make the robots escape the bounding box. A well-known special case of this family of problems is the 15-puzzle, where 15 robots represented by squares have to be reconfigured in a 4×4 grid. These puzzles have been introduced by Noyes Chapman in 1874 and became a craze in the 1880s in the US [30]. Mathematical results showing that there does not always exist a solution date back to 1879 [14]. The computation of the shortest solution for this class of puzzles is NP-hard [20]. More generally, in addition to its playfulness, this scenario corresponds to the practical problem of reorganizing a warehouse without much free space.
(3) At last, we can imagine a set of obstacles defining a bounding box with a small number of free squares (doors) connecting the interior and the exterior of this area. Start and target positions may be both inside and outside the bounding box. Some of the most difficult instances of the CG:SHOP 2021 challenge were of this kind, with all the start and target positions in the bounding box and with only a few doors to go outside. This scenario corresponds to the practical problem of loading and unloading a warehouse.

A key tool that we used for computing the initial solution is a storage network. Storage networks could potentially be used in scenarios 1 and 3, but may be very inefficient for scenario 3 due to a bottleneck at the doors. In scenario 2, it may be possible to adapt the storage network idea, as long as the density is low enough. Tackling these theoretical problems is a possible direction for future works. Another direction is to fully go to practice by addressing concrete applications of MAPF in warehouses managed by Kiva or Quictrons robots systems. The combinatorial problems of order picking in Robotic Mobile Fulfillment Systems are different from the classical MAPF. The start and target positions of the robots are not necessarily constrained but the robots have to pass through a picking area, for filling the order

streaming without wasting time [4, 18, 21, 22, 29]. Efficient heuristics for MAPF are surely a part of the solution but playing with the other variables than the paths opens many other perspectives.

The Conflict Optimizer, however, is not restricted to the coordinated motion planning problem. For example, we can use the same strategy to optimize the vertex coloring of a graph. Given a graph $G = (V, E)$ and a k-coloring $c : V \to \{1, \ldots, k\}$, we initialize the queue with the vertices of color k. Then, until the queue becomes empty, we pop a vertex v from the queue and color it with a color at most $k - 1$ while minimizing the sum of weight of the conflicting vertices and adding the conflicting vertices to the queue. If the queue is empty, then we succeeded in coloring the graph using $k - 1$ colors. This strategy is one of the main strategies used by the `Shadoks`, `Gitastrophe`, and `LASAOFOOFUBESTINNRRALLDECA` teams to respectively win first to third places in the CG:SHOP 2022 challenge, which consists of vertex-coloring the intersection graphs of line segments in the plane.

7 ACKNOWLEDGMENTS

We would like to thank Hélène Toussaint, Raphaël Amato, Boris Lonjon, and William Guyot-Lénat from LIMOS, as well as the Qarma and TALEP teams and Manuel Bertrand from LIS, who continue to make the computational resources of the LIMOS and LIS clusters available to our research. We would also like to thank the challenge organizers and other competitors for their time, feedback, and making this whole event possible.

The work of Loïc Crombez has been sponsored by the French government research program "Investissements d'Avenir" through the IDEX-ISITE initiative 16-IDEX-0001 (CAP 20-25). The work of Guilherme D. da Fonseca is supported by the French ANR PRC grant ADDS (ANR-19-CE48-0005). The work of Yan Gerard is supported by the French ANR PRC grants ADDS (ANR-19-CE48-0005), ACTIVmap (ANR-19-CE19-0005) and by the French government IDEX-ISITE initiative 16-IDEX-0001 (CAP 20-25). The work of Aldo Gonzalez-Lorenzo is supported by the French ANR PRC grant COHERENCE4D (ANR-20-CE10-0002). The work of Pascal Lafourcade is supported by the French ANR PRC grant MobiS5 (ANR-18-CE39-0019), DECRYPT (ANR-18-CE39-0007), SEVERITAS (ANR-20-CE39-0005) and by the French government IDEX-ISITE initiative 16-IDEX-0001 (CAP 20-25). The work of Luc Libralesso is supported by the French ANR PRC grant DECRYPT (ANR-18-CE39-0007).

REFERENCES

[1] Max Barer, Guni Sharon, Roni Stern, and Ariel Felner. 2014. Suboptimal Variants of the Conflict-Based Search Algorithm for the Multi-Agent Pathfinding Problem. In *21st European Conference on Artificial Intelligence, ECAI 2014*, Vol. 263. 961–962. https://doi.org/10.3233/978-1-61499-419-0-961

[2] Roman Barták, Jirí Svancara, Vera Skopková, David Nohejl, and Ivan Krasicenko. 2019. Multi-agent path finding on real robots. *AI Communications* 32, 3 (2019), 175–189. https://doi.org/10.3233/AIC-190621

[3] Eli Boyarski, Ariel Felner, Roni Stern, Guni Sharon, Oded Betzalel, David Tolpin, and Solomon Eyal Shimony. 2015. ICBS: The Improved Conflict-Based Search Algorithm for Multi-Agent Pathfinding. In *Eighth Annual Symposium on Combinatorial Search, SOCS 2015*. 223–225. http://www.aaai.org/ocs/index.php/SOCS/SOCS15/paper/view/10974

[4] Nils Boysen, Dirk Briskorn, and Simon Emde. 2017. Parts-to-picker based order processing in a rack-moving mobile robots environment. *European Journal of Operational Research* 262, 2 (2017), 550–562. https://doi.org/10.1016/j.ejor.2017.03.053

[5] Danny Z. Chen, Rajasekhar Inkulu, and Haitao Wang. 2016. Two-point L_1 shortest path queries in the plane. *Journal of Computational Geometry* 7, 1 (2016), 473–519. https://doi.org/10.20382/jocg.v7i1a20

[6] Danny Z. Chen and Haitao Wang. 2013. L_1 Shortest Path Queries among Polygonal Obstacles in the Plane. In *30th International Symposium on Theoretical Aspects of Computer Science, STACS 2013*, Vol. 20. 293–304. https://doi.org/10.4230/LIPIcs.STACS.2013.293

[7] IBM ILOG CPLEX. 2009. V12. 1: User's Manual for CPLEX. *International Business Machines Corporation* 46, 53 (2009), 157.

[8] Erik D. Demaine, Sándor P. Fekete, Phillip Keldenich, Henk Meijer, and Christian Scheffer. 2019. Coordinated motion planning: Reconfiguring a swarm of labeled robots with bounded stretch. *SIAM J. Comput.* 48, 6 (2019), 1727–1762. https://doi.org/10.1137/18M1194341

[9] Sándor P. Fekete, Phillip Keldenich, Dominik Krupke, and Joseph S. B. Mitchell. 2021. Computing Coordinated Motion Plans for Robot Swarms: The CG: SHOP Challenge 2021. *CoRR* abs/2103.15381 (2021). arXiv:2103.15381 https://arxiv.org/abs/2103.15381

[10] Ariel Felner, Jiaoyang Li, Eli Boyarski, Hang Ma, Liron Cohen, T. K. Satish Kumar, and Sven Koenig. 2018. Adding Heuristics to Conflict-Based Search for Multi-Agent Path Finding. In *28th International Conference on Automated Planning and Scheduling, ICAPS 2018*. 83–87. https://aaai.org/ocs/index.php/ICAPS/ICAPS18/paper/view/17735

[11] Ariel Felner, Roni Stern, Solomon Eyal Shimony, Eli Boyarski, Meir Goldenberg, Guni Sharon, Nathan R. Sturtevant, Glenn Wagner, and Pavel Surynek. 2017. Search-Based Optimal Solvers for the Multi-Agent Pathfinding Problem: Summary and Challenges. In *Tenth International Symposium on Combinatorial Search, SOCS 2017*. 29–37. https://aaai.org/ocs/index.php/SOCS/SOCS17/paper/view/15781

[12] Thibaut Goetghebuer-Planchon. 2020. A C++ implementation of a fast hash map and hash set using hopscotch hashing. https://github.com/Tessil/hopscotch-map.

[13] Meir Goldenberg, Ariel Felner, Roni Stern, Guni Sharon, Nathan R. Sturtevant, Robert C. Holte, and Jonathan Schaeffer. 2014. Enhanced Partial Expansion A*. *Journal of Artificial Intelligence Research* 50 (2014), 141–187. https://doi.org/10.1613/jair.4171

[14] Wm. Woolsey Johnson and William E. Story. 1879. Notes on the '15' Puzzle. *American Journal of Mathematics* 2, 4 (1879), 397–404. https://doi.org/10.2307/2369492

[15] Edward Lam, Pierre Le Bodic, Daniel Damir Harabor, and Peter J. Stuckey. 2019. Branch-and-Cut-and-Price for Multi-Agent Pathfinding. In *International Joint Conferences on Artificial Intelligence IJCAI 2019*. 1289–1296. https://doi.org/10.24963/ijcai.2019/179

[16] Jiaoyang Li, Andrew Tinka, Scott Kiesel, Joseph W. Durham, T.K. Satish Kumar, and Sven Koenig. 2020. Lifelong Multi-Agent Path Finding in Large-Scale Warehouses. *CoRR* abs/2005.07371 (2020). arXiv:2005.07371 https://arxiv.org/abs/2005.07371

[17] Paul Liu, Jack Spalding-Jamieson, Brandon Zhang, and Da Wei Zheng. 2021. Coordinated Motion Planning Through Randomized k-Opt (CG Challenge). In *37th International Symposium on Computational Geometry, SoCG 2021*, Kevin Buchin and Éric Colin de Verdière (Eds.), Vol. 189. 64:1–64:8. https://doi.org/10.4230/LIPIcs.SoCG.2021.64

[18] Yiming Liu, Mengxia Chen, and Hejiao Huang. 2019. Multi-agent Pathfinding Based on Improved Cooperative A* in Kiva System. In *fifth International Conference on Control, Automation and Robotics ICCAR 2019*. 633–638. https://doi.org/10.1109/ICCAR.2019.8813319

[19] Hang Ma, Sven Koenig, Nora Ayanian, Liron Cohen, Wolfgang Hönig, T. K. Satish Kumar, Tansel Uras, Hong Xu, Craig A. Tovey, and Guni Sharon. 2017. Overview: Generalizations of Multi-Agent Path Finding to Real-World Scenarios. *CoRR* abs/1702.05515 (2017). arXiv:1702.05515 http://arxiv.org/abs/1702.05515

[20] Daniel Ratner and Manfred Warmuth. 1986. Finding a Shortest Solution for the NxN Extension of the 15-Puzzle is Intractable. In *Fifth AAAI National Conference on Artificial Intelligence AAAI 1986*. 168–172.

[21] Adrien Rimélé, Philippe Grangier, Michel Gamache, Michel Gendreau, and Louis-Martin Rousseau. 2021. E-commerce warehousing: learning a storage policy. *CoRR* abs/2101.08828 (2021). arXiv:2101.08828 https://arxiv.org/abs/2101.08828

[22] Adrien Rimélé, Philippe Grangier, Michel Gamache, Michel Gendreau, and Louis-Martin Rousseau. 2021. Supervised learning and tree search for real-time storage allocation in Robotic Mobile Fulfillment Systems. *CoRR* abs/2106.02450 (2021). arXiv:2106.02450 https://arxiv.org/abs/2106.02450

[23] 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

[24] David Silver. 2005. Cooperative Pathfinding. In *First Artificial Intelligence and Interactive Digital Entertainment Conference*. 117–122.

[25] Roni Stern, Nathan R. Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne T. Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, T. K. Satish Kumar, Roman Barták, and Eli Boyarski. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In *12th International Symposium on Combinatorial Search, SOCS 2019*. 151–159. https://aaai.org/ocs/index.php/SOCS/SOCS19/paper/view/18341

[26] Pavel Surynek. 2019. Unifying Search-based and Compilation-based Approaches to Multi-agent Path Finding through Satisfiability Modulo Theories. In *28th International Joint Conference on Artificial Intelligence, IJCAI 2019*. 1177–1183. https://doi.org/10.24963/ijcai.2019/164

[27] Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski. 2016. Efficient SAT Approach to Multi-Agent Path Finding Under the Sum of Costs Objective. In *22nd European Conference on Artificial Intelligence, ECAI 2016*, Vol. 285. 810–818. https://doi.org/10.3233/978-1-61499-672-9-810

[28] Jiri Svancara and Pavel Surynek. 2017. New Flow-based Heuristic for Search Algorithms Solving Multi-agent Path Finding. In *Ninth International Conference on Agents and Artificial Intelligence - Volume 2: ICAART,*. 451–458. https://doi.org/10.5220/0006184504510458

[29] Cristiano Arbex Valle and John E Beasley. 2021. Order allocation, rack allocation and rack sequencing for pickers in a mobile rack environment. *Computers & Operations Research* 125 (2021), 105090. https://doi.org/10.1016/j.cor.2020.105090

[30] Wikipedia. 2021. 15 puzzle — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=15%20puzzle&oldid=1058438932. [Online; accessed 27-December-2021].

[31] Boris de Wilde, Adriaan W. ter Mors, and Cees Witteveen. 2014. Push and rotate: a complete multi-agent pathfinding algorithm. *Journal of Artificial Intelligence Research* 51 (2014), 443–492. https://doi.org/10.1613/jair.4447

[32] Hyeyun Yang and Antoine Vigneron. 2021. A Simulated Annealing Approach to Coordinated Motion Planning. In *37th International Symposium on Computational Geometry, SoCG 2021*, Vol. 189. 65:1–65:9. https://doi.org/10.4230/LIPIcs.SoCG.2021.65

[33] Jingjin Yu and Steven M. LaValle. 2013. Planning optimal paths for multiple robots on graphs. In *2013 IEEE International Conference on Robotics and Automation*. 3612–3617. https://doi.org/10.1109/ICRA.2013.6631084