Scaling up MAPF

Daniel D. Harabor
Monash University

http://harabor.net/daniel

Recap

Things we want from a MAPF solver:

- Compute collision-free plans
- For as many agents as possible
- Computed as **fast** as possible
- While maximising throughput (i.e., optimise the efficiency of each individual agent).

Optimal algorithms can reliably solve MAPF problems with 150+ agents. But it's not enough.

Application examples

An amazon parcel sortation centre.

Application examples

Robots drop parcels into sorted bins, for onward delivery.

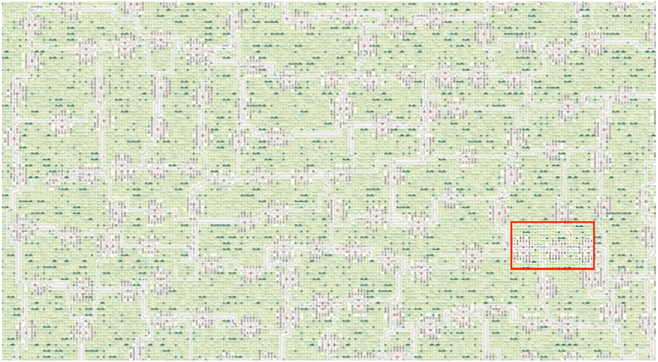
Application examples

Up to 800 agents can be in operation at the same time.

Application examples

Flatland Challenge is a industry sponsored rail planning competition.

Application examples



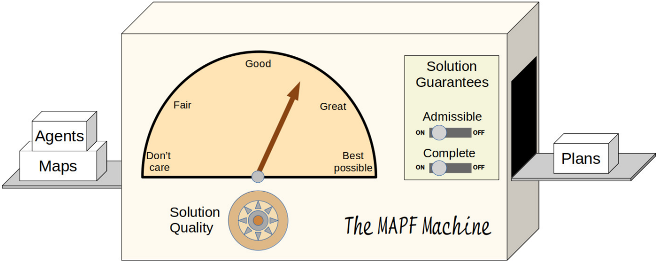
Several thousand agents can be on the map at the same time.

Tradeoffs

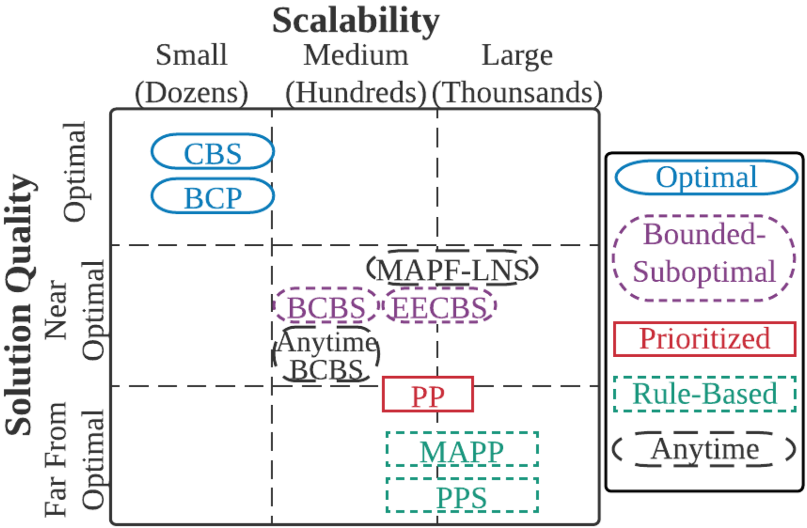
We can scale up (and speed up) existing MAPF algorithms by relaxing some strict requirements.

Tradeoffs

We can scale up (and speed up) existing MAPF algorithms by relaxing some strict requirements.



Approaches that have been considered



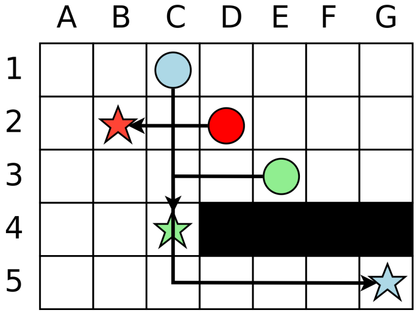
Bounded Suboptimal MAPF

Daniel D. Harabor
Monash University

http://harabor.net/daniel

Recap: Conflict-Based Search

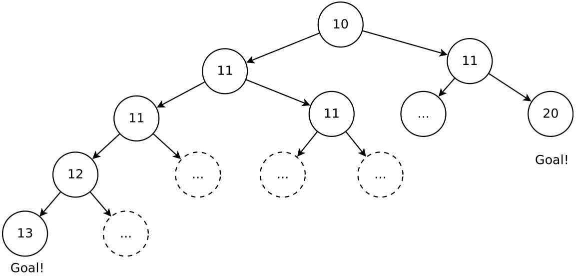
Let's try to solve this MAPF instance using CBS



Bounded Suboptimal Search

Idea: Accept any plan whose cost C is not more than some fixed amount larger than C^*, the cost of the optimal plan.

- $C \leq w \times C^*$ where $w \geq 1$ (w-admissible)
- $C \leq \epsilon + C^*$ where $\epsilon \geq 0$ (ϵ-admissible)



In this CT a $w = 2$ suboptimal plan is available but not within reach for CBS.

How do we compute bounded suboptimal plans?

We use a variant of FOCAL Search [Pearl and Kim, 1982] for the high- and low- level of CBS.

In FOCAL search the frontier comprises two lists:

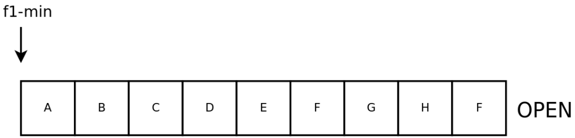
- OPEN, sorted by $f_1(n) = g(n) + h(n)$ (strict admissibility).
- FOCAL $\subseteq$ OPEN, sorted by $f_2(n)$ (possibly inadmissible).

How do we compute bounded suboptimal plans?

We use a variant of FOCAL Search [Pearl and Kim, 1982] for the high- and low- level of CBS.

In FOCAL search the frontier comprises two lists:

- OPEN, sorted by $f_1(n) = g(n) + h(n)$ (strict admissibility).
- FOCAL $\subseteq$ OPEN, sorted by $f_2(n)$ (possibly inadmissible).



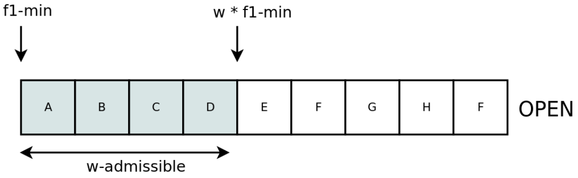
With a conventional OPEN list we always expand the node with minimum f_1

How do we compute bounded suboptimal plans?

We use a variant of FOCAL Search [Pearl and Kim, 1982] for the high- and low- level of CBS.

In FOCAL search the frontier comprises two lists:

- OPEN, sorted by $f_1(n) = g(n) + h(n)$ (strict admissibility).
- FOCAL $\subseteq$ OPEN, sorted by $f_2(n)$ (possibly inadmissible).



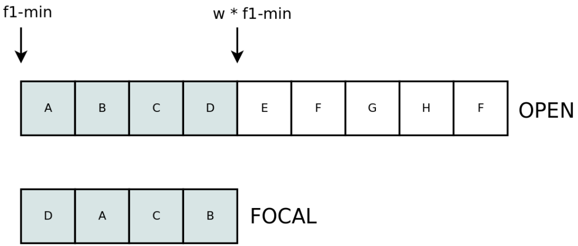
Notice however that many nodes may satisfy the bounded-suboptimal criteria

How do we compute bounded suboptimal plans?

We use a variant of FOCAL Search [Pearl and Kim, 1982] for the high- and low- level of CBS.

In FOCAL search the frontier comprises two lists:

- OPEN, sorted by $f_1(n) = g(n) + h(n)$ (strict admissibility).
- FOCAL $\subseteq$ OPEN, sorted by $f_2(n)$ (possibly inadmissible).



FOCAL allows us to expand the bounded suboptimal nodes in any order, f_2

How do we compute bounded suboptimal plans?

We use a variant of FOCAL Search [Pearl and Kim, 1982] for the high- and low- level of CBS.

In FOCAL search the frontier comprises two lists:

- OPEN, sorted by $f_1(n) = g(n) + h(n)$ (strict admissibility).
- FOCAL $\subseteq$ OPEN, sorted by $f_2(n)$ (possibly inadmissible).

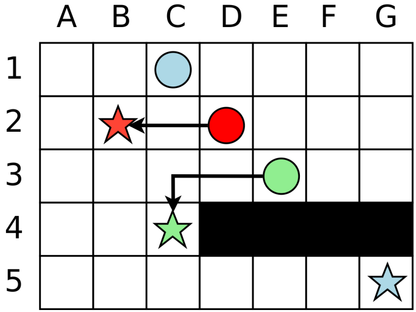
On termination, FOCAL search can return:

- failure (no plan exists) **or**
- a bounded-suboptimal plan, π, and
- (optionally) a best-known lower-bound $LB = \min f_1 \leq C^*$

Modified low-level search

We use FOCAL Search plan single-agent paths. This helps to reduce the number of collisions in a CBS CT node. We have:

- $f_1 = g + h$ (with h = manhattan distance) and $w = 1.5$
- f_2 = minimum number of conflicts.

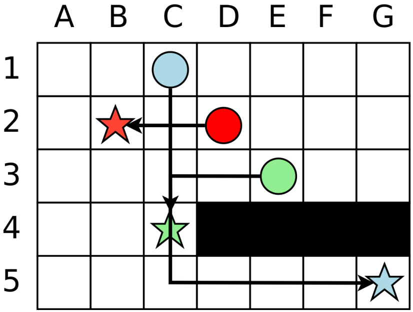


We need to re-plan the blue agent.

Modified low-level search

We use FOCAL Search plan single-agent paths. This helps to reduce the number of collisions in a CBS CT node. We have:

- $f_1 = g + h$ (with $h =$ manhattan distance) and $w = 1.5$
- $f_2 =$ minimum number of conflicts.

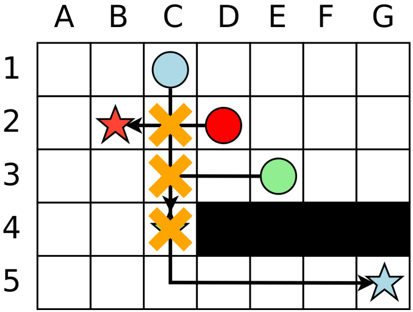


This plan is f_1-optimal but blue is incompatible with red and green

Modified low-level search

We use FOCAL Search plan single-agent paths. This helps to reduce the number of collisions in a CBS CT node. We have:

- $f_1 = g + h$ (with $h =$ manhattan distance) and $w = 1.5$
- $f_2 =$ minimum number of conflicts.

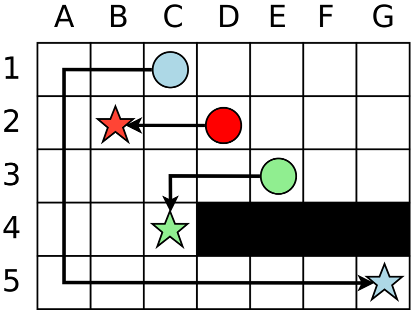


High-level CBS will need to split to resolve these problems.

Modified low-level search

We use FOCAL Search plan single-agent paths. This helps to reduce the number of collisions in a CBS CT node. We have:

- $f_1 = g + h$ (with $h =$ manhattan distance) and $w = 1.5$
- $f_2 =$ minimum number of conflicts.



FOCAL search will return this path which w-suboptimal and conflict-free.

The situation so far...

We use FOCAL search inside CBS: to compute a path π_i for each individual agent $a_i \in A$. Thus, for each CT node we have:

- $c(\pi_i) \leq w \times \pi_i^*$
- $\displaystyle\sum_{a_i \in A} c(\pi_i) \leq w \times C^*$

The situation so far...

We use FOCAL search inside CBS: to compute a path π_i for each individual agent $a_i \in A$. Thus, for each CT node we have:

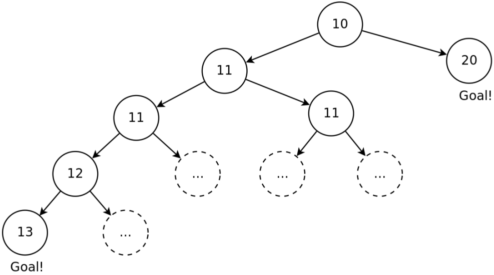
- $c(\pi_i) \leq w \times \pi_i^*$
- $\sum_{a_i \in A} c(\pi_i) \leq w \times C^*$

This approach is indeed bounded suboptimal, but we can do better...

The situation so far...

We use FOCAL search inside CBS: to compute a path π_i for each individual agent $a_i \in A$. Thus, for each CT node we have:

- $c(\pi_i) \leq w \times \pi_i^*$
- $\sum_{a_i \in A} c(\pi_i) \leq w \times C^*$

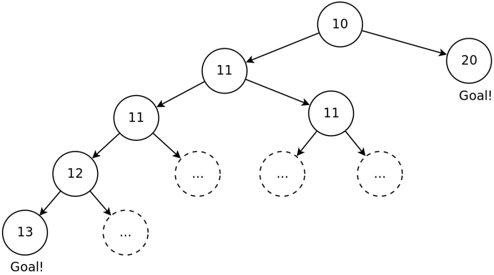


In this CT a $w = 2$ suboptimal plan is available immediately.

The situation so far...

We use FOCAL search inside CBS: to compute a path π_i for each individual agent $a_i \in A$. Thus, for each CT node we have:

- $c(\pi_i) \leq w \times \pi_i^*$
- $\sum_{a_i \in A} c(\pi_i) \leq w \times C^*$



Yet the high-level CBS expands nodes in f-min order!

Modified high-level search

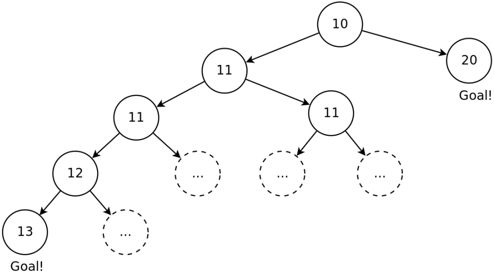
We use FOCAL Search to explore the CBS Conflict Tree. We have:

- $f_1 = \sum_{a_i \in A} LB_i(a_i)$
- $f_2 =$ minimum number of pairwise conflicts

Modified high-level search

We use FOCAL Search to explore the CBS Conflict Tree. We have:

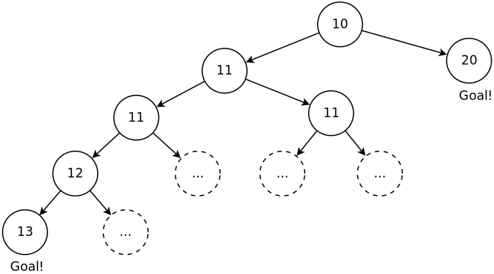
- $f_1 = \sum_{a_i \in A} LB_i(a_i)$
- $f_2 =$ minimum number of pairwise conflicts



Modified high-level search

We use FOCAL Search to explore the CBS Conflict Tree. We have:

- $f_1 = \sum_{a_i \in A} LB_i(a_i)$
- $f_2 =$ minimum number of pairwise conflicts

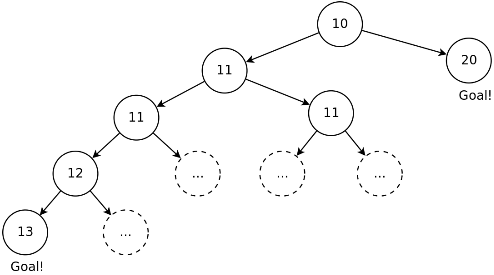


Here $\min f_1 = 10$ and $w(= 2) \times \min f_1 = 20$

Modified high-level search

We use FOCAL Search to explore the CBS Conflict Tree. We have:

- $f_1 = \sum_{a_i \in A} LB_i(a_i)$
- $f_2 =$ minimum number of pairwise conflicts



CBS is now free to expand the shallow solution!

Modified high-level search

We use FOCAL Search to explore the CBS Conflict Tree. We have:

- $f_1 = \sum_{a_i \in A} LB_i(a_i)$
- $f_2 =$ minimum number of pairwise conflicts

CBS with high-level FOCAL search is itself bounded suboptimal. But for best results, we combine it with low-level FOCAL search.

Enhanced CBS

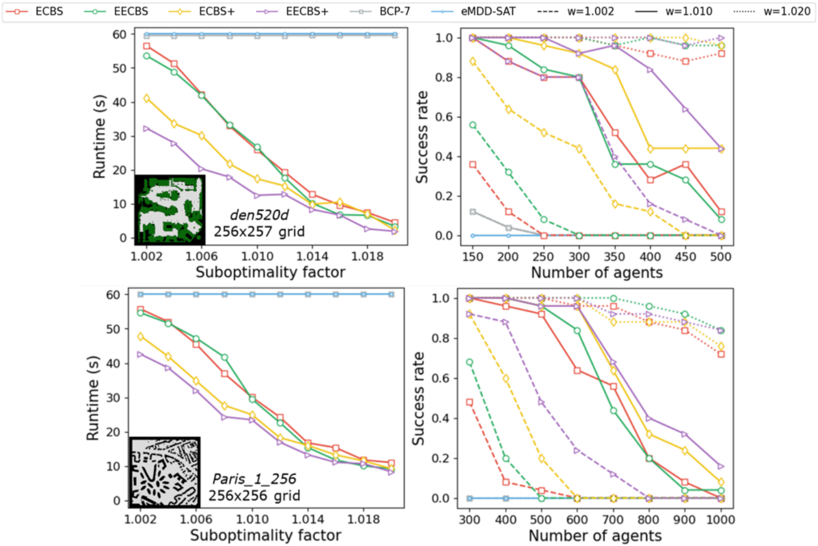
The combination of FOCAL search, at the high- and low- level, is known as **Enhanced CBS** [Barer *et al.*, 2014].

- This an influential workhorse algorithm for MAPF.
- Easily handles hundreds of agents with only small suboptimality.
- Further improved with heuristics and constraints from optimal CBS

Recent developments in this area:

- Explicit Estimation CBS (EECBS) [Li *et al.*, 2021c] investigates new high-level search strategies for ECBS (the frontier is now comprised of three lists instead of two).
- Flex distribution [Chan *et al.*, 2022] for ECBS (FEECBS) investigates how the available suboptimality ("flex") can be divided up amongst the different agents.

Some results for ECBS [Li *et al.*, 2021c]



Rule-based MAPF

Daniel D. Harabor
Monash University

http://harabor.net/daniel

Rule-based Search Algorithms

Idea: Develop a policy that decides how collisions between agents will be resolved. i.e., who has priority where, and when.

Rule-based solvers are generally iterative algorithms:

- Agents are processed sequentially and in order
- Each agent plans one or more steps towards its target
- Later agents cannot derail the plans of earlier agents
- After every agent is planned, the current state is updated
- The process continues until success (unless terminated sooner)

The main ideas

Two types of approaches exist in this space:

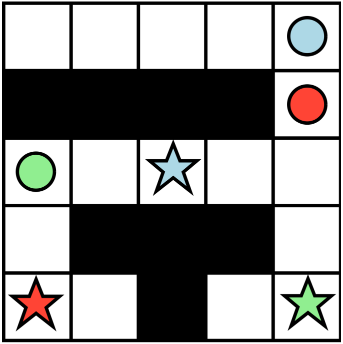
- Move operators
- Temporal reservation methods

In this section we look at examples of each type:

- The PUSH move operator (Okumura variant [Okumura *et al.*, 2019])
- Fixed-order prioritised planning

The Push move operator

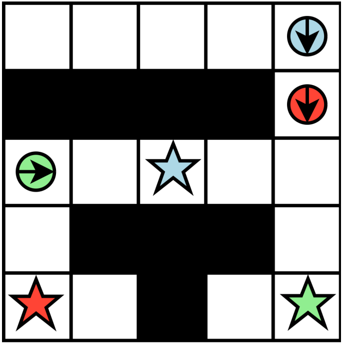
Idea: Resolve collisions by recursively PUSHING other agents aside



In this problem the agents must traverse a narrow passage.

The PUSH move operator

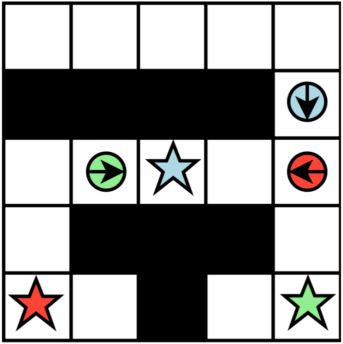
Idea: Resolve collisions by recursively PUSHING other agents aside



Each iteration every agent plans one step towards its target.

The PUSH move operator

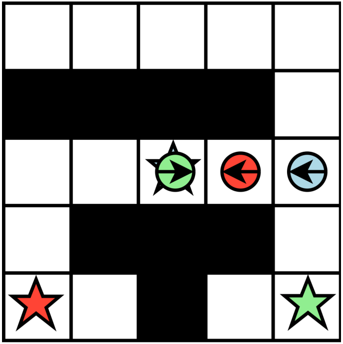
Idea: Resolve collisions by recursively PUSHING other agents aside



The agents are planned sequentially.

The PUSH move operator

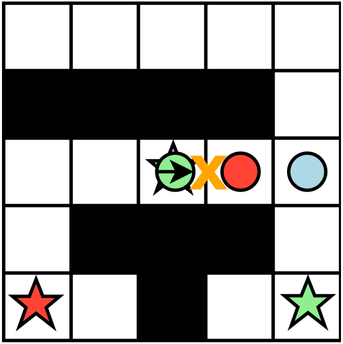
Idea: Resolve collisions by recursively PUSHING other agents aside



Each can move to any adjacent location that is not already claimed.

The PUSH move operator

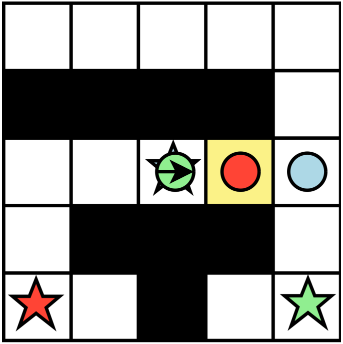
Idea: Resolve collisions by recursively PUSHING other agents aside



Here we plan Green first. But the tile ahead is occupied by Red.

The PUSH move operator

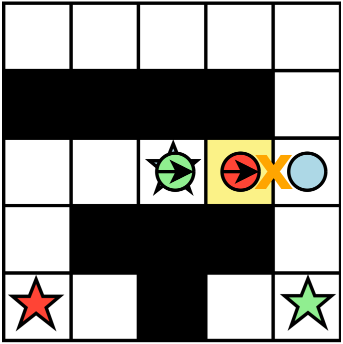
Idea: Resolve collisions by recursively PUSHING other agents aside



Since Green has priority, it claims the tile.

The PUSH move operator

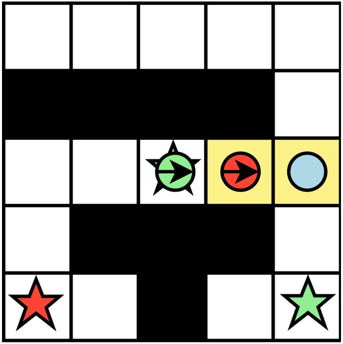
Idea: Resolve collisions by recursively PUSHING other agents aside



(Recurse) Red plans next. It tries to step aisde, but the way is blocked by Blue.

The PUSH move operator

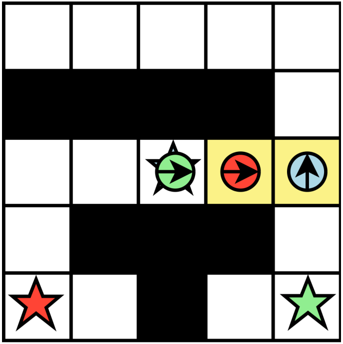
Idea: Resolve collisions by recursively PUSHING other agents aside



Since Red has priority, it claims the tile.

The PUSH move operator

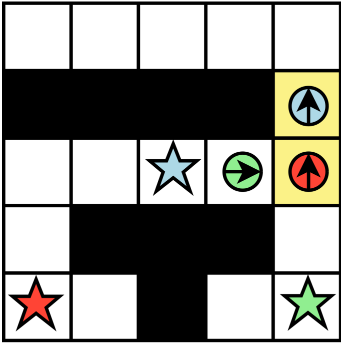
Idea: Resolve collisions by recursively PUSHING other agents aside



(Recurse) Blue plans next and is forced to step aside for Red.

The PUSH move operator

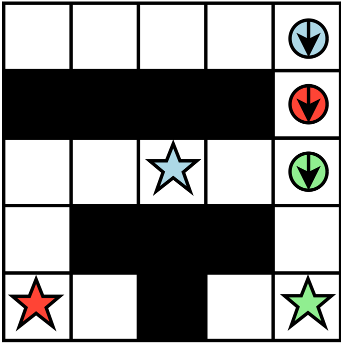
Idea: Resolve collisions by recursively PUSHING other agents aside



In the next iteration Green once again pushes Red (pushes Blue).

The PUSH move operator

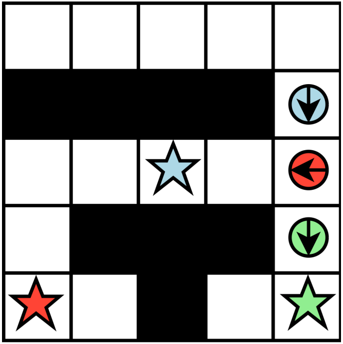
Idea: Resolve collisions by recursively PUSHING other agents aside



Once Green passes, all agents can resume toward their targets.

The PUSH move operator

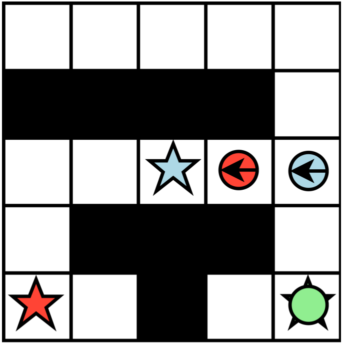
Idea: Resolve collisions by recursively PUSHING other agents aside



Once Green passes, all agents can resume toward their targets.

The PUSH move operator

Idea: Resolve collisions by recursively PUSHING other agents aside



Once Green passes, all agents can resume toward their targets.

The PUSH move operator

Idea: Resolve collisions by recursively PUSHING other agents aside

...

Analysing PUSH

Pros:

- Simple
- Extremely fast (linear time per iteration)
- Scales to thousands of agents
- Can sometimes produce good quality plans

Analysing PUSH

Pros:

- Simple
- Extremely fast (linear time per iteration)
- Scales to thousands of agents
- Can sometimes produce good quality plans

Cons:

- Incomplete

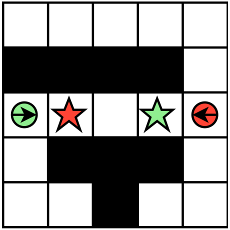
Analysing Push

Pros:

- Simple
- Extremely fast (linear time per iteration)
- Scales to thousands of agents
- Can sometimes produce good quality plans

Cons:

- Incomplete



Push fails here due to livelock

Analysing PUSH

Pros:

- Simple
- Extremely fast (linear time per iteration)
- Scales to thousands of agents
- Can sometimes produce good quality plans

Cons:

- Incomplete
- Only somewhat aware of the objective function

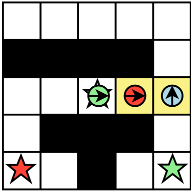
Analysing Push

Pros:

- Simple
- Extremely fast (linear time per iteration)
- Scales to thousands of agents
- Can sometimes produce good quality plans

Cons:

- Incomplete
- Only somewhat aware of the objective function



Here Blue could move up or down. But one is much worse than the other.

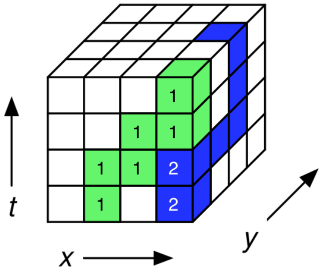
Recent development for move operators

PUSH forms the basis for a very scalable MAPF algorithm known as Priority Inheritance with Backtracking (PIBT) [Okumura *et al.*, 2019]. This is a very capable solver. Recent versions can compute good quality solutions for up to thousands of agents.

PUSH can also be combined with other similar move operators, such as SWAP, to derive more powerful and complete MAPF algorithms. The state of the art is a called PUSH-AND-ROTATE [de Wilde *et al.*, 2014].

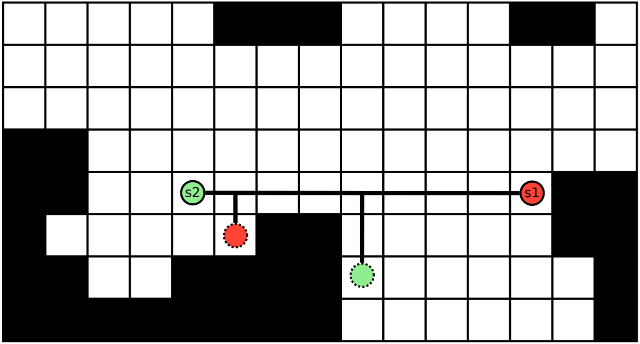
The Reserve operator

Idea: use a reservation table to coordinate k future timesteps of individual agents.



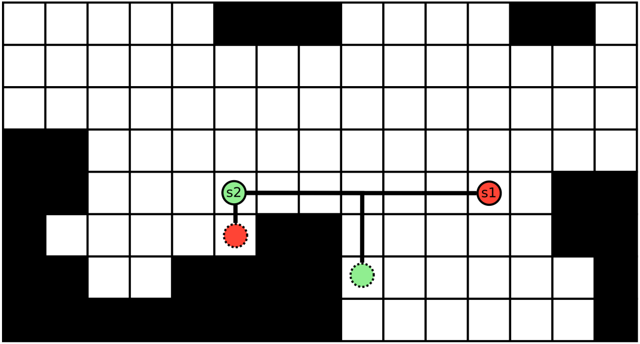
This (3d) data structure tells the (x, y) position of an agent for the upcoming timesteps in the interval $[t, t+k]$

Preventing collisions



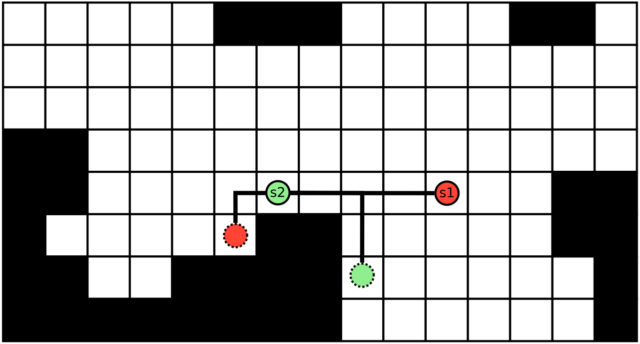
Sometimes two (individually planned) agents compete for the same location.

Preventing collisions



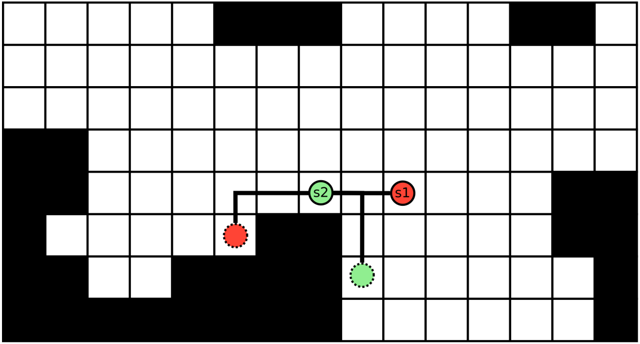
Sometimes two (individually planned) agents compete for the same location.

Preventing collisions



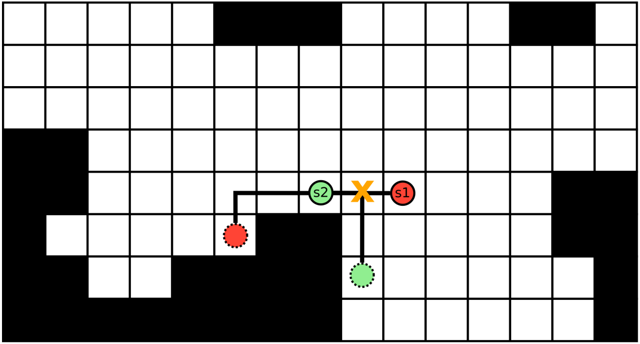
Sometimes two (individually planned) agents compete for the same location.

Preventing collisions



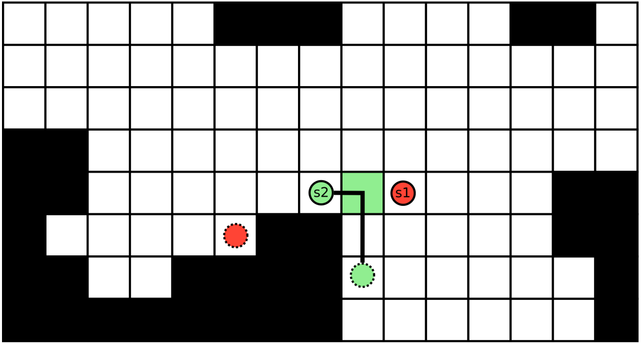
Sometimes two (individually planned) agents compete for the same location.

Preventing collisions



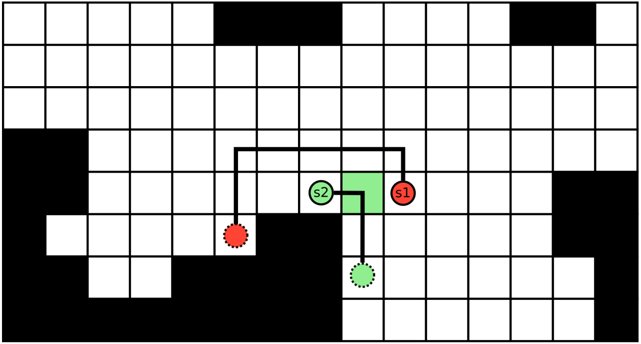
Sometimes two (individually planned) agents compete for the same location.

Preventing collisions



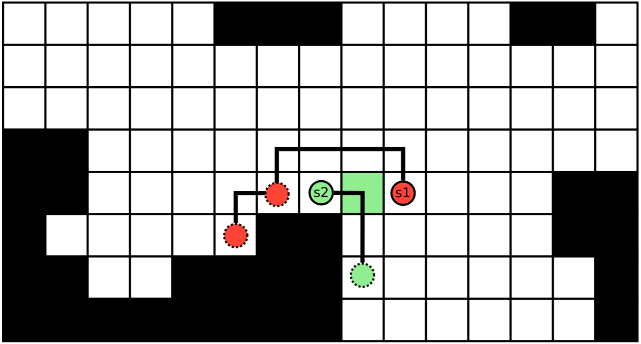
We set $2 > 1$ which gives agent 2 priority at the conflict location

Preventing collisions



Agent 1 meanwhile must find a new path that avoids the conflict location.

Preventing collisions



Usually we replan to the goal node. Another option is to search only to the next waypoint on the old (individually optimal) path. This is faster, but could be worse (cost, aesthetic appeal).

Generalising RESERVE

RESERVE-1 works well for both edge and vertex collisions. But we can also decompose it to derive a family of related techniques.

The main components are:

- Agent selection (who gets priority?)
- Reserve horizon (how many steps ahead to reserve?)
- Replanning strategy (to the target? to a waypoint?)

Generalising Reserve

Reserve-1 works well for both edge and vertex collisions. But we can also decompose it to derive a family of related techniques.

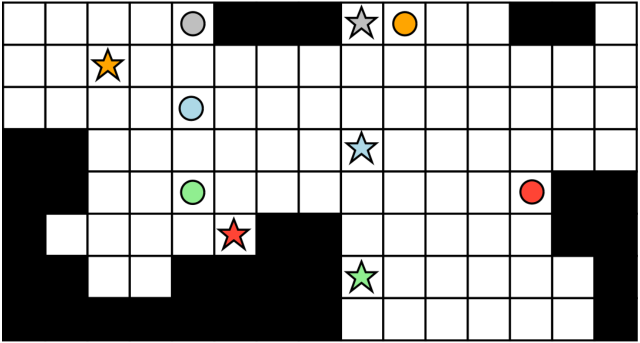
The main components are:

- Agent selection (who gets priority?)
- Reserve horizon (how many steps ahead to reserve?)
- Replanning strategy (to the target? to a waypoint?)

Various instantiations of this family are widely used in practice.

Prioritised Planning [Erdmann and Lozano-Pérez, 1987]

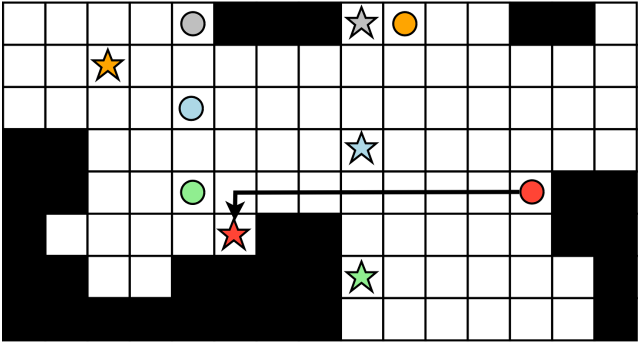
Idea: Reserve the (spatio-temporal) shortest path of each agent.



We plan the agents one by one.

Prioritised Planning [Erdmann and Lozano-Pérez, 1987]

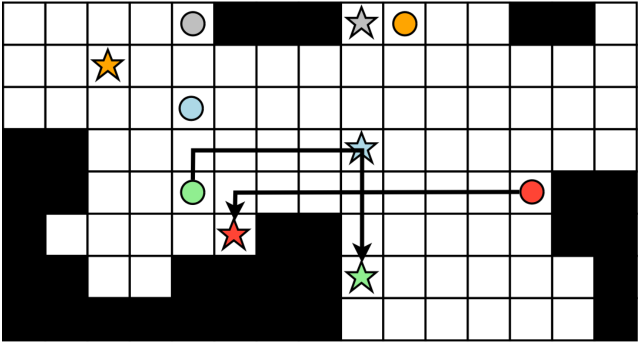
Idea: Reserve the (spatio-temporal) shortest path of each agent.



Red goes first and reserves its individually optimal path.

Prioritised Planning [Erdmann and Lozano-Pérez, 1987]

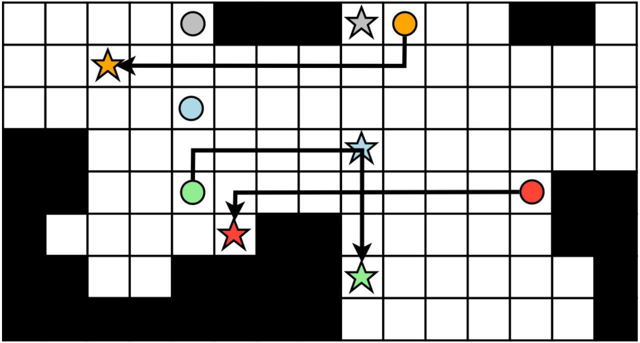
Idea: Reserve the (spatio-temporal) shortest path of each agent.



Green is next. It must avoid the path of the higher-priority Red agent.

Prioritised Planning [Erdmann and Lozano-Pérez, 1987]

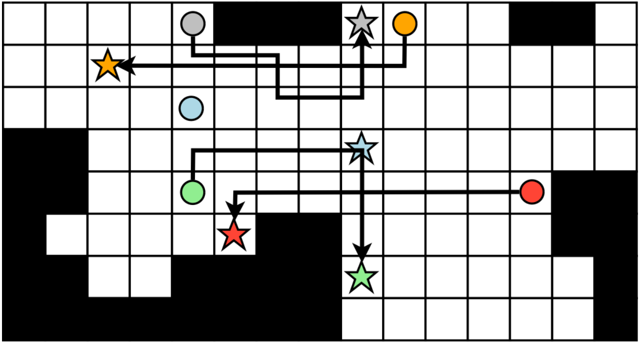
Idea: Reserve the (spatio-temporal) shortest path of each agent.



Each subsequent agent avoids all those previously planned.

Prioritised Planning [Erdmann and Lozano-Pérez, 1987]

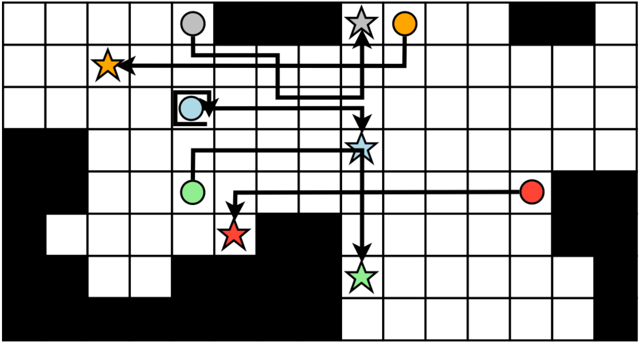
Idea: Reserve the (spatio-temporal) shortest path of each agent.



Notice high priority agents never wait for any low priority agents.

Prioritised Planning [Erdmann and Lozano-Pérez, 1987]

Idea: Reserve the (spatio-temporal) shortest path of each agent.



If all agents have a feasible path, the problem is solved.

Pros and Cons

Prioritised Planning is extremely popular in practice.

Pros:

- Simple (one agent at a time)
- Fast (one agent at a time!)

Cons:

- Not optimal
- Not complete in general (certain problem setups avoid this)
- Not clear how to find a good ordering

Recent developments in this area

Priority-Based Search (PBS) [Ma *et al.*, 2019] is CBS-like planner that fixes priorities between two agents only in the event of a conflict. This strategy is much more effective than conventional ordering strategies.

Rolling Horizon Collision Resolution (RHCR) [Li *et al.*, 2020] is an iterated priority planner that reserves only k steps in advance (cf. the entire path). Authors report large performance gains for teams of up to 1000 agents.

Large Neighbourhood Search

Daniel D. Harabor
Monash University

http://harabor.net/daniel

The story so far

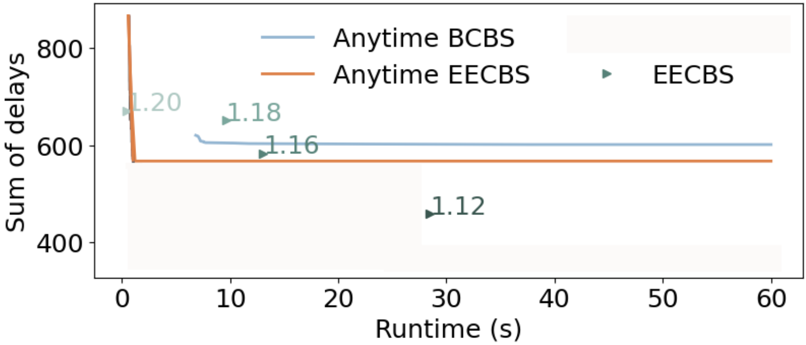
We want MAPF solvers that are fast, scalable and maximise throughput.
The approaches we have seen so far:

- Compute feasible solutions fast (e.g., PIBT)
- Compute close-to-optimal solutions more slowly (e.g., EECBS)
- Scale to many hundreds (even thousands) of agents.

But they are all strongly-coupled, all-or-nothing solvers:

- They tackle the whole problem, in one shot.
- They return a single solution, or failure.

The story so far

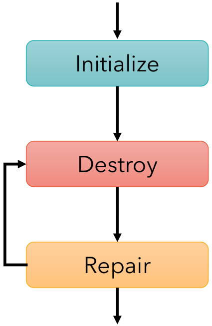


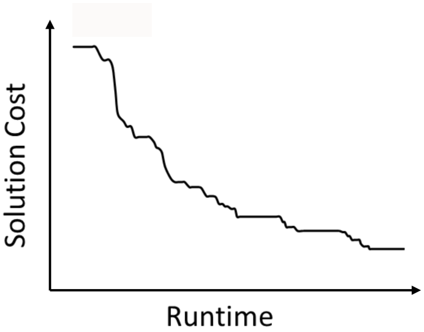
Behaviour of anytime optimal and near-optimal CBS variants
(this experiment: 150 agents on 32x32 map with 20% random obstacles)

Large Neighbourhood Search for MAPF (MAPF-LNS2)

Idea:

Solve MAPF problems incrementally; improve the solution over time.





Initialize

Assign collision-minimising paths to each agents.

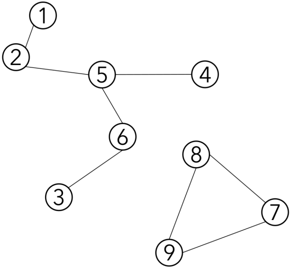
Here, we adapt a version of prioritised planning:

- Plan a path for the first agent (random order)
- Plan a path that **minimises** the number of collisions with the planned path.
- Repeat steps 1 and 2 until every agent has a path

NB: The first solution could also be computed with any other solver [Li *et al.*, 2021a].

Destroy

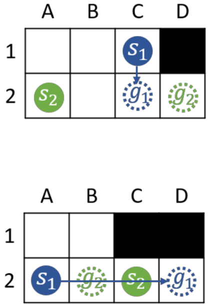
Select a subset of agents that are in collision. We switch between several different strategies that try to identify highly coupled sets of agents.



Collision neighbourhood is based on a conflict graph. We choose k agents from a random selected connected component.

Destroy

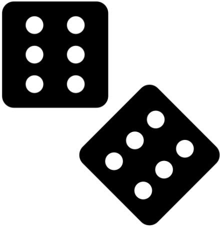
Select a subset of agents that are in collision. We switch between several different strategies that try to identify highly coupled sets of agents.



Failure neighbourhood. We choose an in-collsion agent a_i and then select other agents that prevent a_i from completing its plan, collision-free.

Destroy

Select a subset of agents that are in collision. We switch between several different strategies that try to identify highly coupled sets of agents.



Random neighbourhood. We choose k agents at random.

Repair

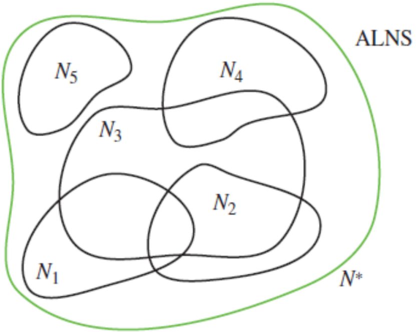
Assign new collision-minimising paths to the agents in the destroy neighbourhood.

Here, we use the same prioritised planning method as before:

- Plan a path for the first agent (random order)
- Plan a path that **minimises** the number of collisions with the planned path.
- Repeat steps 1 and 2 until every agent has a path

Adaptive Neighbourhood Selection

LNS tracks how effectively a neighbourhood improves the objective (# collisions) and switches between them dynamically.

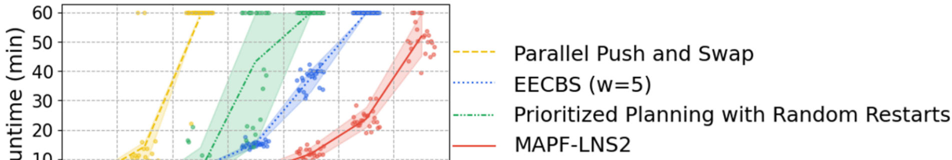


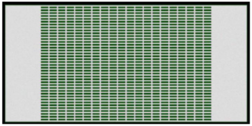
Each neighbourhood N_i is assigned a weight w_i that tracks its recent success: in reducing the number of collisions. We select N_i with probability $\frac{w_i}{\sum_j w_j}$.

Results

In this experiment we test scalability on a **single warehouse map** We compare MAPF-LNS2 with a variety of suboptimal solvers:

- Prioritized Planning (random restarts)
- Parallel Push and Swap (rule-based solver)
- EECBS (bounded suboptimal)





warehouse-20-40-10-2-2.map
340x164 grid with 38,756 empty cells

Recent developments in this area

LNS is a workhorse meta-heuristic in Operations Research [Shaw, 1998]. Effectiveness depends strongly on good strategies for **destroy** and **repair**

- MAPF-LNS first appears in [Li *et al.*, 2021a]. This version starts from an initially feasible plan and iteratively improves.
- Later work (MAPF-LNS2) [Li *et al.*, 2022] improves performance and adds support for infeasible first solutions.
- Other recent work considers how ML can be used to derive new types of destroy heuristics [Huang *et al.*, 2022].

Variants of MAPF-LNS obtained **first place** at the 2020 and 2021 NeurIPS Flatland Challenge [Li *et al.*, 2021b].

Summary and Challenges

Daniel D. Harabor
Monash University

http://harabor.net/daniel

MAPF is an important problem

The problem MAPF problem asks us to coordinate a team of moving agents.

Studied by multiple communities of interest

- Artificial Intelligence
- Robotics
- Industrial practitioners

Key enabler for a variety of important and emerging applications:

- Warehouse fulfilment
- Mail sortation
- Pipe routing
- Aircraft towing
- Autonmous intersections
- Computer games

MAPF problems are tricky to solve

Finding feasible solutions to MAPF problems is **tractable**. But finding optimal solutions (and close approximations) is **hard**.

Practitioners have competing demands:

- Plans should be computed fast
- But maximise an objective function

Additional complications:

- Agent kinematics
- Execution uncertainty
- Operational constraints
- 3D Environments

There has been massive progress in MAPF

We have developed strong tools to address some of the core difficulties that makes MAPF problems hard.

- Symmetry breaking consttaints
- Strong heuristic bounds
- More efficient search-based solvivng techniques

Compared to just a few years ago:

- Optimal search: from dozens of agents to 150+
- Bounded suboptimal: from hundreds to 1000+
- Suboptimal: near-optimal many thousands of moving agents

Things that are still hard

Many opportunities exist for further improvement

- Continuous space and time
- Execution-time failures
- Motion Planning
- Online MAPF and
- Multi-agent Pickup and Delivery

Great topics for PhD theses!

Would you like to know more?

Community website: `http://mapf.info`.

Conferences:

- AAAI and IJCAI (general AI)
- AAMAS (multi-agent AI, relatively general)
- ICRA and IROS (general robotics)
- International Conference on Planning and Scheduling (ICAPS)
- International Symposium on Combinatorial Search (SoCS)

References I

Max Barer, Guni Sharon, Roni Stern, and Ariel Felner.
Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem.
In *Seventh Annual Symposium on Combinatorial Search*, 2014.

Shao-Hung Chan, Jiaoyang Li, Graeme Gange, Daniel Harabor, Peter J Stuckey, and Sven Koenig.
Flex distribution for bounded suboptimal multi-agent path finding.
In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2022.

Boris de Wilde, Adriaan ter Mors, and Cees Witteveen.
Push and rotate: a complete multi-agent pathfinding algorithm.
J. Artif. Intell. Res., 51:443–492, 2014.

M. A. Erdmann and T. Lozano-Pérez.
On multiple moving objects.
Algorithmica, 2:477–521, 1987.

References II

Taoan Huang, Jiaoyang Li, Sven Koenig, and Bistra Dilkina.
Anytime multi-agent path finding via machine learning-guided large neighborhood search.
2022.

Jiaoyang Li, Andrew Tinka, Scott Kiesel, Joseph W Durham, TK Satish Kumar, and Sven Koenig.
Lifelong multi-agent path finding in large-scale warehouses.
In *AAMAS*, pages 1898–1900, 2020.

Jiaoyang Li, Zhe Chen, Daniel Harabor, P Stuckey, and Sven Koenig.
Anytime multi-agent path finding via large neighborhood search.
In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2021.

Jiaoyang Li, Zhe Chen, Yi Zheng, Shao-Hung Chan, Daniel Harabor, Peter J Stuckey, Hang Ma, and Sven Koenig.
Scalable rail planning and replanning: Winning the 2020 flatland challenge.
In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, pages 477–485, 2021.

References III

Jiaoyang Li, Wheeler Ruml, and Sven Koenig.
EECBS: A bounded-suboptimal search for multi-agent path finding.
In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 12353–12362, 2021.

Jiaoyang Li, Zhe Chen, Daniel Harabor, P Stuckey, and Sven Koenig.
Mapf-lns2: Fast repairing for multi-agent path finding via large neighborhood search.
In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2022.

Hang Ma, Daniel Harabor, Peter Stuckey, Jiaoyang Li, and Sven Koenig.
Searching with Consistent Prioritization for Multi-Agent Path Finding.
In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 7643–7650, 2019.

Keisuke Okumura, Manao Machida, Xavier Défago, and Yasumasa Tamura.
Priority inheritance with backtracking for iterative multi-agent path finding.
In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 535–542. ijcai.org, 2019.

References IV

Judea Pearl and Jin H Kim.
Studies in semi-admissible heuristics.
IEEE transactions on pattern analysis and machine intelligence, (4):392–399, 1982.

P. Shaw.
Using constraint programming and local search methods to solve vehicle routing problems.
In *CP*, pages 417–431, 1998.