Tractable Multi-Agent Path Planning on Grid Maps

Ko-Hsin Cindy Wang and **Adi Botea***

NICTA† and The Australian National University
{cindy.wang|adi.botea}@nicta.com.au

Abstract

Multi-agent path planning on grid maps is a challenging problem and has numerous real-life applications. Running a centralized, systematic search such as A* is complete and cost-optimal but scales up poorly in practice, since both the search space and the branching factor grow exponentially in the number of mobile units. Decentralized approaches, which decompose a problem into several subproblems, can be faster and can work for larger problems. However, existing decentralized methods offer no guarantees with respect to completeness, running time, and solution quality.

To address such limitations, we introduce MAPP, a tractable algorithm for multi-agent path planning on grid maps. We show that MAPP has low-polynomial worst-case upper bounds for the running time, the memory requirements, and the length of solutions. As it runs in low-polynomial time, MAPP is incomplete in the general case. We identify a class of problems for which our algorithm is complete. We believe that this is the first study that formalises restrictions to obtain a tractable class of multi-agent path planning problems.

1 Introduction

Path planning is important in many real-life problems, including robotics, military applications, logistics, and commercial games. Single-agent path planning, where the size of the state space is bounded by the map size, can be tackled with a search algorithm such as A* [Hart *et al.*, 1968]. In multi-agent problems, however, both the number of states and the branching factor grow exponentially in the number of mobile units. Hence, despite its completeness and solution optimality guarantees, a centralized A* search has little practical value in a multi-agent path planning problem, being intractable even for relatively small maps and collections of mobile units.

Scalability to larger problems can be achieved with decentralized approaches, which decompose an initial problem into a series of searches [Silver, 2006; Wang and Botea, 2008]. However, existing decentralized methods are incomplete and provide no criteria to distinguish between problems that can be successfully solved and problems where such algorithms fail. Further, no guarantees are given with respect to the running time and the quality of the computed solutions.

Addressing such limitations is a central motivation for our work. We introduce MAPP (multi-agent path planning), a tractable algorithm for multi-agent path planning on grid maps. Given a problem with m traversable tiles and n mobile units, MAPP's worst case performance is $O(m^2n^2)$ for the running time, $O(m^2n)$ for the memory requirements, and $O(m^2n^2)$ for the solution length. Depending on the assumptions on the input problems, the bounds can be even lower, being linear in m instead of quadratic, as we discuss in Section 5. As it runs in low polynomial time, MAPP is incomplete in the general case. However, it is complete for a class of problems which we define in Section 3, and extend in Section 6.

MAPP keeps its running costs low by eliminating the need for replanning. A path $\pi(u)$ is computed at the beginning for each unit u. No other searches are required at runtime. A *blank travel* idea, inspired from sliding tile puzzles, is at the center of the algorithm. A unit u can progress from a current location l_i^u to the next location l_{i+1}^u on its path $\pi(u)$ only if a *blank* is located there (i.e., l_{i+1}^u is empty). Intuitively, if the next location is currently occupied by another unit, MAPP tries to bring a blank along an *alternate path*, which connects l_{i-1}^u and l_{i+1}^u without passing through l_i^u. When possible, the blank is brought to l_{i+1}^u by shifting units along the alternate path, just as the blank travels in a sliding tile puzzle. The ability to bring a blank to the next location is key to guarantee a unit's progress. See formal details in Section 4.

Among several existing methods to abstract a problem map into a search graph, including navigation meshes (e.g., [Tozour, 2002]), visibility points (e.g., [Rabin, 2000]), and quadtrees (e.g., [Samet, 1988]), we focus on grid maps. Besides being very popular and easy to implement, grid maps are well suited to multi-agent problems. Their search graphs contain more nodes to cover all locations of the traversable space, offering more path options to avoid collisions.

*We thank Nick Hay, Malte Helmert, Jussi Rintanen, and our reviewers for their valuable feedback.

†NICTA is funded by the Australian Government's *Backing Australia's Ability* initiative.

2 Related Work

Traditional multi-agent path planning takes two main approaches [Latombe, 1991]. A *centralised* method incorporates a single decision maker. For example, planning for all units simultaneously, as in a centralised A* search, or using techniques such as biased-cost heuristics [Geramifard *et al.*, 2006] to detect and resolve collisions as a whole. Although theoretically optimal, in practice this approach has a prohibitive complexity and cannot scale up to many units.

On the other hand, a *decentralised* method can significantly reduce computation by decomposing the problem into several subproblems. This typically involves first planning the units' paths independently, then handling the interactions along the way; it is often much faster but yields suboptimal solutions and loses completeness. Examples include computing velocity profiles to avoid collisions [Kant and Zucker, 1986], or assigning priorities [Erdmann and Lozano-Perez, 1986]. Local Repair A* (LRA*) [Stout, 1996] replans using expensive A* searches with every collision. Ryan [2008] introduces a complete method that combines multi-agent path planning with hierarchical planning on search graphs with specific sub-structures such as stacks, halls, cliques and rings.

Other recent decentralised methods in a grid-based world include using a direction map structure [Jansen and Sturtevant, 2008] to share information about units' directions of travel, so later units can follow the earlier ones. The improved coherence leads desirably to reduced collisions. Silver's cooperative pathfinding method [2006] uses a reservation table and performs windowed forward searches on each unit, based on a true distance heuristic obtained from an initial backward A* search from each target. In Wang and Botea's work [2008], units follow a flow annotation on the map when planning and moving, then use heuristic procedures to break deadlocks. Methods such as these scale up to a number of units well beyond the capabilities of centralised search. However, they come with no formal characterisation of the running time and the quality of solutions in the worst case. They lack the ability to answer beforehand whether a given problem would be successfully solved, which is always important in the case of incomplete algorithms. We address such issues in the following sections.

3 Problem Statement and Definitions

A *problem* is characterized by a map and a non-empty collection of mobile units U. Units are homogeneous in speed and size. Each unit $u \in U$ has an associated start-target pair (s_u, t_u). Two distinct units can share neither a common start nor a common target. The objective is to navigate all units from their start positions to the targets while avoiding all fixed and mobile obstacles. A *state* contains the positions of all units at a given time.

We assume that only straight moves in the four cardinal directions can be performed (4-connect grid). All our solutions can be extended to 8-connect grids (*octiles*), since the standard practice is to allow a diagonal move only if an equivalent (but longer) two-move path exists. Therefore, any problem with diagonal moves can be reduced to a problem with only straight moves, at the price of possibly taking longer paths.

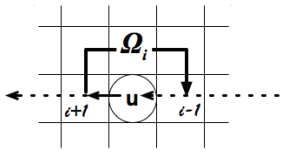


Figure 1: An example of an alternate path, Ω_i, connecting locations $i-1$ and $i+1$ along the path $\pi(u)$ for a unit u.

Introducing diagonal moves could reduce the path length but they have the potential drawback of becoming blocked more often than straight moves on crowded maps, due to the physical impossibility of squeezing past two orthogonal neighbours. A better study of this trade-off is left as future work.

Definition 1 *A problem belongs to the class* SLIDEABLE *iff for each unit $u \in U$ a path $\pi(u) = (l_0^u, l_1^u, \ldots, l_{k_u}^u)$ exists, where $l_0^u = s_u$, $l_{k_u}^u = t_u$, and k_u denotes the length of the path $\pi(u)$, such that all the following conditions are met:*

1. *Alternate connectivity: For each three consecutive locations $l_{i-1}^u, l_i^u, l_{i+1}^u$ on $\pi(u)$, an alternate path $\Omega_i^{\pi(u)}$ (or simpler, Ω_i^u) exists between l_{i-1}^u and l_{i+1}^u that does not go through l_i^u, as illustrated in Figure 1.*
2. *In the initial state, l_1^u is blank (i.e. unoccupied).*
3. *Target isolation:*
   (a) $(\forall v \in U \setminus \{u\}) : t_u \notin \pi(v)$*; and*
   (b) $(\forall v \in U, \forall i \in \{1, \ldots, k_v - 1\}) : t_u \notin \Omega_i^v$.

Unless otherwise mentioned, in the rest of this paper, we assume that the input problem belongs to the SLIDEABLE class and that the computed paths $\pi(u)$ are fixed throughout the solving process. Given a unit u, let $\text{pos}(u)$ be its current position, and let $\text{int}(\pi(u)) = \{l_1^u, \ldots, l_{k_u-1}^u\}$ be the interior of its precomputed path $\pi(u)$.

Definition 2 *We define the* private zone $\zeta(u)$ *of a unit as follows. If $pos(u) = l_i^u \in int(\pi(u))$, then $\zeta(u) = \{l_{i-1}^u, l_i^u\}$. Otherwise, $\zeta(u) = \{pos(u)\}$.*

The set of units U is partitioned into a subset S of *solved units* that have already reached their targets, and a subset A of *active units*. Initially, all units are active. After becoming solved, units do not interfere with the rest of the problem (as ensured by the target isolation condition). As shown later, solved units never become active again, and do not have to be considered in the remaining part of the solving process.

Definition 3 *The* advancing condition *of an active unit u is satisfied iff its current position belongs to the path $\pi(u)$ and the next location on the path is blank.*

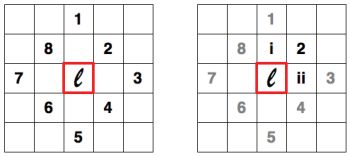
Definition 4 *A state is* well positioned *iff all active units have their advancing condition satisfied.*

4 The MAPP Algorithm

As illustrated in Algorithm 1, for each problem instance, MAPP starts by computing a path $\pi(u)$ for each unit u to its target (goal), constructing and caching alternate paths Ω

Algorithm 1 Overview of MAPP.

1: **for** each $u \in A$ **do**
2: compute $\pi(u)$ and Ω's (as needed) from s_u to t_u
3: **if** path computation failed **then**
4: **return** problem is not in SLIDEABLE
5: **while** $A \neq \emptyset$ **do**
6: do progression step
7: do repositioning step if needed



a b

Figure 2: (a) The eight locations two moves away from l. (b) Two two-move paths from l to location 2 go through i and ii.

along the way. Note that all paths π and alternate paths Ω need to satisfy the conditions in Definition 1. If the *for* loop in lines 1–4 succeeds, MAPP can tell that the instance at hand belongs to SLIDEABLE, for which MAPP is complete. The remaining part is a series of two-step iterations (lines 5–7).

A *progression step* advances active units towards their targets. As shown later, each progression step brings at least one active unit to its target, reducing A and ensuring that the algorithm terminates, reaching the state where all units are solved. A progression could result in breaking the advancing condition of one or more active units, if any remain. The objective of a *repositioning step* is to ensure that each active unit has its advancing condition satisfied before starting the next progression step. Note that a repositioning step is necessary after every progression step except for the last.

4.1 Path Pre-computation

For each problem instance, we pre-compute each path $\pi(u)$ individually. To ensure that paths satisfy the alternate connectivity condition (Definition 1), we modify the standard A* algorithm as follows. When expanding a node x', a neighbour x'' is added to the open list only if there is an alternate path between x'' and x, the parent of x'. By this process we compute each path $\pi(u)$ and its family of alternate paths Ω simultaneously. To give each neighbour x'' of the node x' a chance to be added to the open list, node x' might have to be expanded at most three times, once per possible parent x. Therefore, $O(m)$ node expansions are required by A* search to find each π path. Equivalently, computing a π path could also be seen as a standard A* search in the extended space of pairs of neighbouring nodes (at most four nodes are created for each original node).

Since alternate paths depend only on the triple locations, not the unit, we can re-use this information when planning paths for all units of the same problem. This means that the alternate path for any set of three adjacent tiles on the map is computed only once, and cached for later use. Given a location l on the map, there are at most eight locations that could be on a path two moves away on a four-connect grid. As shown in Figure 2a, these eight locations form a diamond shape around l. For each of the four locations that are on a straight line from l (locations 1, 3, 5, 7), we precompute an alternate path that avoids the in-between location and any targets. For each of the other four locations (labeled 2, 4, 6, 8), we need to compute two alternate paths. For example, there are two possible paths between l and 2 that are two moves long: through i or ii (Figure 2b). We need one alternate path to avoid each intermediate location. In summary, we precompute at most 12 paths for each l. For at most m locations on a map, we need $\frac{12m}{2} = 6m$ alternate paths (only need one computation for each triple, since an alternate path connects its two endpoints both ways).

A possible optimization is to reuse alternate paths for SLIDEABLE problems on the same map. Alternate paths that overlap targets in the new problem need to be re-computed. With large number of units, the overhead in verifying and recomputing alternate paths may outweigh the savings.

4.2 Progression

Algorithm 2 shows the progression step in pseudocode. At each iteration of the outer loop, active units attempt to progress by one move towards their targets. They are processed in order (line 2). If unit v is processed before unit w, we say that v has a higher priority and write $v < w$. The ordering is fixed inside a progression step, but it may change from one progression step to another. The actual ordering affects neither the correctness nor the completeness of the method, but it may impact the speed and the solution length. The ordering of units can be chosen heuristically, e.g. giving higher priority to units that are closer to target. Thus, these units get to their target more quickly, and once solved they are out of the way of the remaining units in the problem.

At the beginning of a progression step, one *master unit* $\bar{u}$ is selected. It is the unit with the highest priority among the units that are active at the beginning of the progression step. The status of being the master unit is preserved during the entire progression step, even after $\bar{u}$ becomes solved. At the beginning of the next progression step, a new master unit will be selected among the remaining active units.

Lines 3–15 show the processing of u, the active unit at hand. If u has been pushed off its precomputed path as a result of blank travel (see details later), then no action is taken (lines 3–4). If u is on its path but the next location l^u_{i+1} is currently blocked by a higher-priority unit v, then no action is taken (lines 5–6). Lines 7 and 8 cover the situation when unit u has been pushed around (via blank travel) by higher-priority units back to a location on $\pi(u)$ already visited in the current progression step. In such a case, u doesn't attempt to travel again on a previously traversed portion of its path, ensuring that the bounds on the total travelled distance introduced later hold. Otherwise, if the next location l^u_{i+1} is available, u moves there (lines 9–10). Finally, if l^u_{i+1} is occupied by a smaller-priority unit, an attempt is made to first bring a blank to l^u_{i+1} and then have u move there (lines 11–13). When u moves to a new location l^u_{i+1} (lines 10 and 13), a test is performed to check if l^u_{i+1} is the target location of u.

Algorithm 2 Progression step.

```
1: while changes occur do
2:   for each u ∈ A in order do
3:     if pos(u) ∉ π(u) then
4:       do nothing {u has been pushed off the track as a
         result of blank travel}
5:     else if ∃v < u : l^u_{i+1} ∈ ζ(v) then
6:       do nothing {wait until l^u_{i+1} is released by v}
7:     else if u has already visited l^u_{i+1} in current progres-
       sion step then
8:       do nothing
9:     else if l^u_{i+1} is blank then
10:      move u to l^u_{i+1}
11:    else if can bring blank to l^u_{i+1} then
12:      bring blank to l^u_{i+1}
13:      move u to l^u_{i+1}
14:    else
15:      do nothing
```

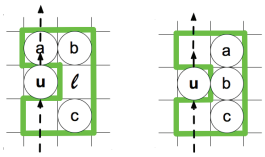


Figure 3: At the left, u is blocked by a. A blank is found at location l along the Ω-path (outlined in bold). At the right: by sliding b and then a along Ω, the blank is brought to l^u_{i+1}.

If this is the case, then u is marked as solved by removing it from A and adding it to S.

Bringing a blank to l^u_{i+1} (lines 11 and 12) works as follows. A location $l \in \Omega^u_i$ is sought with the following properties: (1) l is blank, (2) none of the locations from l to l^u_{i+1} (including these two ends) along Ω^u_i belongs to the private zone of a higher-priority unit, and (3) l is the closest (along Ω^u_i) to l^u_{i+1} with this property. If such a location l is found, then the test on line 11 succeeds. The actual travel of the blank from l to l^u_{i+1} along Ω^u_i (line 12) is identical to the movement of tiles in a sliding-tile puzzle. Figure 3 shows an example before and after blank traveling. The intuition behind seeking a blank along Ω^u_i is that, often, l^u_{i-1} remains blank during the time interval after u advances to l^u_i and until the test on line 11 is performed. This is guaranteed to always hold in the case of the master unit $\bar{u}$, since $l^{\bar{u}}_{i-1}$ belongs to $\zeta(\bar{u})$ and no other unit can interfere with $\zeta(\bar{u})$. The following result characterizes the behaviour of $\bar{u}$ more precisely. The proof is rather straightforward from the previous algorithm description, and we skip it for the sake of space.

Lemma 5 *As long as the master unit $\bar{u}$ is not solved, it is guaranteed to advance along $\pi(\bar{u})$ at each iteration of the outer ("while") loop in Algorithm 2. By the end of the current progression step, at least $\bar{u}$ has reached its target.*

The following result is useful to ensure that a progression step always terminates, either in a state where all units are solved or in a state where all remaining active units are stuck.

Lemma 6 *Algorithm 2 generates no cycles (i.e., no repetitions of the global state).*

Proof: Below we show a proof by contradiction. Assume that there are cycles. Consider a cycle and the active unit u in the cycle has the highest priority. Since no other unit in the cycle dominates u, it means that the movements of u cannot be part of a blank travel triggered by a higher priority unit. Therefore, the movements of u are a result of either line 10 or line 13. That is, all u's moves are along its path $\pi(u)$. Since $\pi(u)$ contains no cycles, u cannot run in a cycle. □

4.3 Repositioning

By the end of a progression step, some of the remaining active units (if any left) have their advancing condition broken. Recall that this happens for a unit u when either $\text{pos}(u) \notin \pi(u)$ or u is placed on its precomputed path but the next location on the path is not blank. A repositioning step ensures that a well positioned state is reached (i.e., all active units have the advancing condition satisfied) before starting the next progression step.

A simple and computationally efficient method to perform repositioning is to undo part of the moves performed in the most recent progression step. Solved units are not affected. For the remaining active units, we undo their moves, in reverse order, until a well positioned state is encountered. We call this strategy *reverse repositioning*.

Lemma 7 *If the reverse repositioning strategy is used at line 7 of Algorithm 1 (when needed), then all progression steps start from a well positioned state.*

Proof: This lemma can be proven by induction on the iteration number j in Algorithm 1. Since the initial state is well positioned (this follows easily from Definitions 1 and 4), the proof for $j = 1$ is trivial. Assume that a repositioning step is performed before starting the iteration $j+1$. In the worst case, reverse repositioning undoes all the moves of the remaining active units (but not the moves of the units that have become solved), back to their original positions at the beginning of j-th progression step. In other words, we reach a state s that is similar to the state s' at the beginning of the previous progression step, except that more units are on their targets in s. Since s' is well positioned (according to the induction step), it follows easily that s is well positioned too. □

4.4 Example

A simple example of how MAPP works is illustrated in Figure 4. There are two units, ordered as $a < b$. In (i), as a and b progress towards their targets, a becomes blocked by b. In (ii), a blank is brought in front of a by sliding b down Ω^a_i (outlined in bold); as a side effect, b is pushed off its path. At the end of the current progression step (iii), a reaches its target. In the repositioning step (iv), b undoes its previous move to restore its advancing condition. In the next progression step (v), b reaches its target. The algorithm terminates.

5 Worst-case and Best-case Analysis

We give here bounds on the runtime, memory usage, and solution length for the MAPP algorithm on a problem in SLIDE-

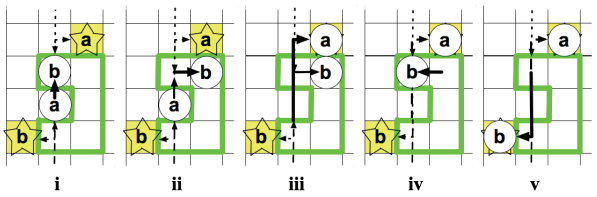


Figure 4: Example of how MAPP works.

ABLE with n units and a map of m traversable tiles. We examine the worst case scenario in each case, and also discuss a best-case scenario at the end. Assessing the algorithm's practical significance is also very important, but it is not the topic of this paper.

We introduce an additional parameter, λ, to measure the maximal length of alternate paths Ω. In the worst case, λ grows linearly with m. However, in many practical situations, λ is a constant, since the ends of an Ω path are so close to each other. Our analysis discusses both scenarios.

Theorem 8 *Algorithm 1 has a worst-case running time of $O(n^2m)$ when λ is a constant, or $O(n^2m^2)$ when λ grows linearly with m.*

Proof: As outlined in Section 4.1, each *single-agent* A* search for π expands nodes linear in the size of the map, taking $O(nm)$ time for all n units. Each A* search for an alternate path Ω expands $O(\lambda^2)$ nodes, so the A* searches for all Ω's take $O(m\lambda^2)$ time. If λ grows linearly with m, we note that the number of states in the A* search space is linear in m, giving $O(m^2)$ time.

In a single progression step, outlined in Algorithm 2, suppose blank travel is required by all n units, every move along the way except the first and last moves, and each operation brings the blank from the location behind (l^u_{i-1}) to the front (l^u_{i+1}). Since the length of π paths is bounded by m and the length of alternate paths Ω is bounded by λ, the total number of moves in a progression step is within $O(nm\lambda)$, and so is the running time of Algorithm 2.

Clearly, the complexity of a repositioning step cannot exceed the complexity of the previous progression step. In the worst case, the size of A reduces by one at each iteration in lines 5–7 of Algorithm 1. So MAPP takes $O(n^2m\lambda)$ time to run, which is $O(n^2m)$ when λ is constant and $O(n^2m^2)$ when λ grows linearly with m. □

Theorem 9 *The maximum memory required to execute MAPP is $O(nm)$ when λ is a constant, or $O(nm^2)$ when λ grows linearly with m.*

Proof: Caching the possible Ω paths for the entire problem as described in Section 4.1 takes $O(m\lambda)$ memory. The A* searches for the π paths are performed one at a time. After each search, π is stored in cache, and the memory used for the open and closed lists is released. The A* working memory takes only $O(m)$ space, and storing the π paths takes $O(nm)$ space. Overall, path computation across all units requires $O(nm + m\lambda)$.

Then, in lines 5–7 of Algorithm 1, memory is required to store a stack of moves performed in one progression step, to be used during repositioning. As shown in the proof of Theorem 8, the number of moves in a progression step is within $O(nm\lambda)$. So, the overall maximum memory required to execute the program is $O(nm\lambda)$, which is $O(nm)$ when λ is a constant and $O(nm^2)$ when λ grows linearly with m. □

Theorem 10 *The total distances travelled across all units is at most $O(n^2m)$ when λ is a constant, or $O(n^2m^2)$ when λ grows linearly with m.*

Proof: As shown previously, the number of moves in a progression step is within $O(nm\lambda)$. The number of moves in a repositioning step is strictly smaller than the number of moves in the previous progression step. There are at most n progression steps (followed by repositioning steps). Hence, the total travelled distance is within $O(n^2m\lambda)$. □

Corollary 11 *To store the global solution takes $O(n^2m)$ memory when λ is a constant, or $O(n^2m^2)$ when λ grows linearly with m.*

We discuss now a best case scenario. MAPP computes optimal solutions in the number of moves when the paths π are optimal and all units reach their targets without any blank traveling (i.e., units travel only along the paths π). An obvious example is where all paths π are disjoint. In such a case, solutions are *makespan* optimal too. As well as preserving the optimality in the best case, the search effort in MAPP can also be smaller than that spent in a centralised A* search, being n single-agent $O(m)$ searches, compared to searching in the combined state space of n units, with up to $\frac{m!}{(m-n)!}$ states and a branching factor of 5^n.

6 A Closer Look at Class SLIDEABLE and Beyond

In this section we discuss advantages, limitations and extensions of the class SLIDEABLE.

Fortunately, all conditions in the definition of class SLIDEABLE refer to the computed paths π. This gives us some freedom to actively search for a path that satisfies those conditions. Among the possible paths between two locations s and t, we can try, for example, to avoid planning paths through targets of other units, and single-width tunnels where the alternate connectivity condition is harder to satisfy. Furthermore, when searching for the π paths, we can actively try to minimise the overlapping between paths. This way, we could avoid beforehand many potential collisions and detours, reducing both the running time and the quality of the solutions.

Examples of problems that do not belong to SLIDEABLE, as shown in Figure 5, include cases where two or more targets are located inside a narrow open tunnel and units come from opposite ends, or where many targets are located next to each other in a cluster. The existence of special cases that MAPP cannot solve is quite normal, given the algorithm's low polynomial-time upper-bound.

The SLIDEABLE class can be extended without affecting MAPP's completeness. The three conditions in Definition 1 are kept simple for the sake of clarity. Each of these can be relaxed. The relaxations require small and computationally cheap changes to the algorithm. We show the intuitions for

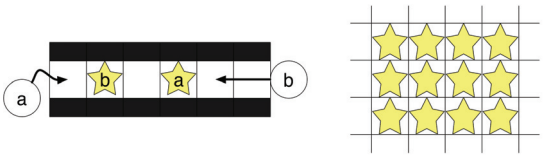


Figure 5: Two challenging cases. Left: two targets (denoted by stars) inside a narrow tunnel. Right: a cluster of targets.

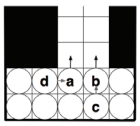


Figure 6: After units a and b make their first moves, initial blanks are created for c and d, and so on.

extending the first two conditions, and a more detailed explanation for the third.

We can relax the first condition on alternate connectivity based on the fact that a segment of a path $\{l_i^u, \ldots, l_j^u\} \subseteq \pi(u)$ does not require alternate paths if no other paths or alternate paths intersect with any of the segment's locations: $(\forall v \in U \setminus \{u\}, \forall x \in \{1, \ldots, k_v\})$: $\{l_i^u, \ldots, l_j^u\} \cap (\pi(v) \cup \Omega_x^v) = \emptyset$. This way, no blank traveling is required along $l_i^u \ldots l_j^u$, since u is the only unit that traverses that part of the map.

We can also relax the second condition, which requires all units to have an initial blank, to have only *some* units meeting this requirement. Figure 6 illustrates an example where initially only two units, a and b, have a blank in front. But as they move forward, blanks are created at their vacated positions, satisfying the initial blank condition for units c and d. As c and d make their first moves, the blanks get carried back again, allowing the units behind to start moving in turn.

The third condition (target isolation) can be relaxed as follows. Assume some targets do interfere with some paths: $\exists u, v \in U, t_u \in \Pi(v)$, where $\Pi(v) = (\pi(v) \cup \bigcup_{x=1}^{k_v - 1} \Omega_x^v)$, violating the target isolation condition. In such cases, we define a partial ordering $\prec$ such that $v \prec u$ when the target of u is on $\Pi(v)$. Problems where $\prec$ produces no cycles can be solved by a version of MAPP with two small modifications: (1) the total ordering $<$ inside a progression step is consistent with $\prec$: $v \prec u \Rightarrow v < u$; (2) if $v \prec u$, then u cannot be marked as solved (i.e. moved from A to S) unless v has already been marked as solved. So even if u arrives at its target first, v can get past u by performing the normal blank travel. Following that, u can undo its moves back to target in the repositioning step, as outlined in Section 4.3.

7 Conclusion

In multi-agent path planning, traditional centralised and decoupled approaches both have shortcomings: the former faces an exponentially growing state space in the number of units; the latter offers no guarantees with respect to completeness, running time and solution length. We have identified conditions for a class of multi-agent path planning problems on grid maps that can be solved in polynomial time. To solve such problems, we introduced an algorithm, MAPP, with low polynomial complexity in time ($O(n^2m^2)$), space ($O(nm^2)$), and solution quality ($O(n^2m^2)$). The upper bounds can be even better (linear in m instead of quadratic), depending on the assumptions on the input problems. We also discussed extensions to more general classes of problems where the algorithm's completeness is preserved.

A detailed empirical evaluation, which is important for establishing MAPP's practical applicability, is a main future work topic. Also, we plan to investigate heuristic enhancements that could be added to the algorithm. We will study the performance ratio of MAPP compared with optimal solutions. We want to define a measure of how tightly coupled units are in large multi-agent pathfinding problems, and to use this measure to refine our theoretical study and to design heuristic enhancements.

References

[Erdmann and Lozano-Perez, 1986] M. Erdmann and T. Lozano-Perez. On Multiple Moving Objects. In *ICRA*, pages 1419–1424, 1986.

[Geramifard *et al.*, 2006] A. Geramifard, P. Chubak, and V. Bulitko. Biased Cost Pathfinding. In *AIIDE*, pages 112–114, 2006.

[Hart *et al.*, 1968] P. Hart, N. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Trans. on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[Jansen and Sturtevant, 2008] R. Jansen and N. Sturtevant. A New Approach to Cooperative Pathfinding. In *AAMAS*, pages 1401–1404, 2008.

[Kant and Zucker, 1986] K. Kant and S. W. Zucker. Toward Efficient Trajectory Planning: The Path-Velocity Decomposition. *IJRR*, 5(3):72–89, 1986.

[Latombe, 1991] J.-C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, 1991.

[Rabin, 2000] S. Rabin. A* Speed Optimizations. In Mark Deloura, editor, *Game Programming Gems*, pages 272–287. Charles River Media, 2000.

[Ryan, 2008] M. R. K. Ryan. Exploiting Subgraph Structure in Multi-Robot Path Planning. *JAIR*, pages 497–542, 2008.

[Samet, 1988] H. Samet. An Overview of Quadtrees, Octrees, and Related Hierarchical Data Structures. NATO ASI Series, Vol. F40, 1988.

[Silver, 2006] D. Silver. Cooperative pathfinding. *AI Programming Wisdom*, 2006.

[Stout, 1996] B. Stout. Smart Moves: Intelligent Pathfinding. *Game Developer Magazine*, October/November 1996.

[Tozour, 2002] P. Tozour. Building a Near-Optimal Navigation Mesh. In Steve Rabin, editor, *AI Game Programming Wisdom*, pages 171–185. Charles River Media, 2002.

[Wang and Botea, 2008] K.-H. C. Wang and A. Botea. Fast and Memory-Efficient Multi-Agent Pathfinding. In *ICAPS*, pages 380–387, 2008.