**Problem Definitions and Solutions**

Applications benefiting from solutions to the MAPF and MAPD problems are often large, dynamic, continuous spaces throughout which many complex interactions must occur to realize a desired outcome. In real-world applications there are many concerns to be aware of which tend to cloud the central problem of finding optimal paths. Examples include robot turning radius, positional tolerances, momentum, acceleration, carry weight, and so on.

Fundamentally, solutions in MAPF and MAPD problems are designed to find collision-free paths for all agents while maximizing some metric which relates to the performance of the system. To begin developing a solution in such spaces it is best to restrict the number of confounding issues, starting from a basis of the most critical and well-defined issues. By implementing a few key assumptions, the problem space is reduced significantly. The two most useful changes to make to the problem statement involve discretization of spatial and temporal concerns.

With a discretized space, collisions occur at points in space, rather than needing to be evaluated on the whole spectrum of motion. This reduces the description and evaluation of motion to simple ideas of position rather than needing to consider speed, orientation, or turning radius.

With a discrete description of time, agent actions can be evaluated on a simple basis of the number of timesteps required to complete the action. This allows evaluations of the system state to be made on a consistent basis without concerns about synchronization and timing needing to be settled.

Of course, this also means that solutions found in discrete spaces cannot be blindly applied to the real-world, continuous, situation. However, with a method for approaching the process established, it becomes easier to augment the solution to handle kinematic restrictions and respect tolerances in motion [1].

With the above restrictions, it becomes possible to express the problem spatially in terms of a mathematics construction called a *graph*. In doing so, the problem can be formally stated and solutions from other areas of mathematics may be applied using the model. Finding the shortest path, among other problems of graph traversal, is a well-studied problem with many useful results and techniques. A graph is therefore the standard model used to solve many problems in pathfinding, including MAPF and MAPD problems.

**Defining Multi-Agent Problems**

A graph is composed of two primary objects: *vertices* and *edges*. Abstractly, a vertex (also called a node) is a representation of some *thing* which is possible to reach via some *process*. It could be a state, a location, or an object.

An edge (also called a link) is used to represent a connection between two vertices. Such a relationship could be the path walked by a person to reach location A from location B, the set of actions taken in a system to reach state A from state B, or the machining process used to create a part from stock material. An edge can be said to be directed if it is only possible to move from vertex A to vertex B along the edge, and not from B to A. Otherwise, it is said to be undirected. An edge may also have a weight, or cost, associated with it, possibly representing the expense of driving from city A to city B along a particular route.

These two components form the non-linear structure called a *graph* and are typically expressed as a graph composed of a set of vertices and a set of edges . The edge set is built from a subset of vertices which are connected. An edge is represented by an unordered pair of connected vertices. For this work, the graph is constructed with the following definition:

where

This construction disallows the existence of multiple edges connecting a node, which would be called a *multigraph*. Further, the representation of edges as a set rather than an ordered pair means that an edge is agnostic to ideas of direction. If the edge was an ordered pair, the graph would be considered *directed*. The set-builder notation for the edge set also contains an assertion that an edge cannot connect a node to itself. Without this restriction, the graph is called a graph *with loops*. The above definition, used throughout this work, is therefore termed a *simple undirected* graph *without loops*.

The exclusion of graphs with loops is adhered to for simplicity of the graph’s structure, but the restriction is not necessarily required. In fact, the ability of an agent to travel from its current location to the same location (but one timestep into the future, effectively a waiting move) will be critical in allowing other agents to maneuver with minimal disruption.

A few additional components are needed to fully define an MAPF or MAPD problem. A set of agents contains information about how many agents are in the problem’s system and their individual properties. The task set contains tasks which have not yet been completed, driving the solving of the problem. The task set must be finite in order for a solution to exist but is not a fixed quantity. Tasks may be freely added and removed from the set in an on-line fashion, simulating an infinite set. For the purpose of formality, two mapping functions and are used to define the positions of all agents and tasks. In this way, a multi-agent problem is defined:

where

* ,
* .

With a formal expression of the problem, it becomes possible to formally express conflicts. Figure 1 shows the two basest cases of conflict possible in this construction of the MAPF and MAPD problems: a vertex conflict (left) and a swapping conflict (right). Each expresses the idea that some resource (in this case, space) is being used by more than one agent, which in the real-world system causes an undesired collision.

Further conflicts exist but in general there is no unified requirement for certain conflicts to be observed throughout the literature [2].

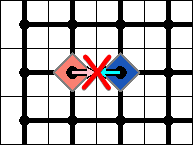
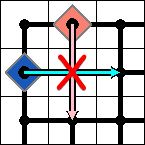


Figure 1: Two conflicts agents may experience when moving within a multi-agent problem instance, rendered in FleetBench. Left: Vertex conflict. Right: Edge conflict.

Formally expressing these conflicts in the context of MAPF and MAPD systems is done by representing a moment in time where two agents will occupy the same space. In the typical construction each agent has a series of positions it intends to inhabit, separated by timesteps, called a plan. The space-time position of agent for any given node at any timestep is given as .

A vertex conflict arises when two agents attempt to occupy the same node at the same time:

Swapping conflicts occur when two agents attempt to use the same edge E to reach new nodes. Because an edge connects only two nodes, the agents are attempting to swap node positions and would collide on the way to their planned locations:

With these restrictions in place, it is now possible to evaluate which nodes and edges are available for use at any given time. Actions which would cause vertex or swapping conflicts are marked off during search and traversal.

In MAPF problems, objectives or tasks are expressed as the need for an agent (which may or may not be a particular agent from the set of agents in the system) to be in a specific location. The MAPD problem is an augmentation of the MAPF statement to include a two-part task. This task includes a demand for an agent to be present at some location for a “pickup” action, followed by a need for the same agent to be in a location afterwards to perform a “delivery” action. Therefore, any agent in an MAPF or MAPD problem with an assigned task will have some objective or target node to reach, ideally in the fewest number of timesteps possible.

With these definitions and restrictions in place, it is possible to begin seeking optimal paths.

**Basic Approaches**

The single-agent pathfinding (SAPF) problem has been thoroughly investigated in literature for some time. The SAPF problem shares similar concepts with the MAPF or MAPD problems, in that a path must be found for each agent. However, the results do not generalize to the multi-agent case—single-agent solutions make no effort to avoid other agents, by definition. There are sufficiently many similarities that solutions to the SAPF problem can be adapted as a starting point for addressing multi-agent problems.

**The A\* Algorithm**

One of the most useful results in the study of graph traversal is the A\* algorithm [3]. This algorithm can be viewed as an extension of Dijkstra’s algorithm, which finds the shortest path between nodes (in terms of edge costs) via an exhaustive search. A\* augments this approach with the use of a heuristic that guides the search by attaching an idea of proximity to the goal to each node explored, called a node’s *h-Score*. Choosing the next nodes to be explored using the best h-Score guides the algorithm to explore paths which approach the goal, until they cannot be further advanced. So long as certain qualities about the heuristic being used are guaranteed, A\* returns provably optimal paths without over-processing the graph [4]. The primary requirement for the heuristic function to do so is that it does not overestimate the distance to the goal.

Several simple and intuitive heuristics exist. Dijkstra’s algorithm is functionally equivalent to the A\* algorithm when the heuristic always returns 0, making no effort to distinguish best nodes. The Manhattan distance is a good model for 4-neighbor connected spaces, while the Chebyshev distance is well-suited for 8-neighbor connected spaces. In a continuous space, the Euclidean distance may be used.

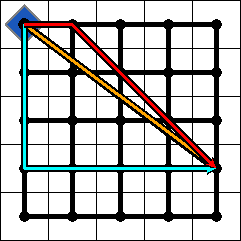


Figure 2: Three different measures of distances on a grid rendered in FleetBench: Euclidean (orange), Manhattan (cyan), and Chebyshev (red).

Each heuristic provides some quantitative value describing the distance between two points in their respective geometries and each is considered to be valid for use in the A\* algorithm, provided they do not overestimate distance. For example, while the Euclidean distance may be employed in a 4-neighbor connected space (a straight line to the goal will always have less or equal length to any Manhattan distance), the Manhattan distance is not a usable heuristic in continuous spaces as it will overestimate the distance to the goal. In this work, the graph is assumed to describe a system with a Manhattan geometry, where a node may have up to four neighbors. Further, the edge cost for each movement is assumed to be the same, such that an agent exerts the same effort in moving any direction.

Two other scoring mechanisms are employed to inform the search. A node’s *g-Score* is the length of the best path found to reach a node from the starting position. This is not always the best possible path, so a node’s *g-Score* may be updated over the course of the algorithm as improvements to the path are found. A node’s *f-Score* is the sum of a node’s *g-Score* and *h-Score*, representing the estimate of the total length of a path which passes through the node on its way to the goal node.

There are five subroutines called during the execution of A\* which are defined and identified here for ease of reference:

* returns an estimate of the distance between nodes and , determined by using an appropriate heuristic function.
* returns the set of nodes which are connected to node via an edge. This function may be augmented to include the node in the returned values if an agent may be interested in waiting in the same position.
* returns the edge cost of a movement from to , which in many cases is equivalent to the function’s result but may not always be. With the assumption that all edge costs are the same this function does not perform any calculation, but it may easily be augmented.
* returns a Boolean *True* value if the input node is the goal node, and Boolean *False* otherwise.
* is used when the goal node is explored by the algorithm. Using saved *g-Score* data, the optimal path to from is reconstructed, returning the ideal path to the goal.

The operating procedure for the A\* algorithm is described in Algorithm 1. Lines 2-6 describe the setup for the algorithm. The search is primarily driven by the *open* set, which acts as a priority queue containing all nodes which are available for exploration, ordered in favor of the least remaining heuristic distance from the node to the goal. The first node added to this set is the starting position, whose *g-Score* is clearly zero. At this point in time, *g-Scores* for other nodes are unknown and assumed to be infinite. The *f-Score* for any node is simply the calculated heuristic distance, which for all nodes other than the start node is unknown.

|  |  |  |
| --- | --- | --- |
| **Algorithm 1** A\* | | |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | **if** IsGoal(: |
|  | BuildPath( |
|  | **return** |
|  | **for** : |
|  | Cost |
|  | **if** |
|  |  |
|  |  |
|  |  |
|  | **if** **not in** |
|  |  |
|  | **return** |

So long as there is at least one node in the open set, the search has not exhausted potential options, and continues with the process of removing the node from the open set and evaluating it on lines 8-9. Should the removed node turn out to be the goal node, the algorithm recursively checks its memory of parent nodes, reconstructing the path with the lowest g-Score to the node and returning it in lines 10-12. Otherwise, the algorithm obtains a list of all nodes which share an edge with the current node, which are called neighbors.

On lines 13-20, for each neighbor a *g-Score* is calculated which combines the *g-Score* of the current node with the edge cost of traveling from the current node to the neighbor. If this cost is lower than the currently stored *g-Score* for the neighbor node, then an improved path from the start to the neighbor node has been found, and the current node is recorded as the best way to reach the neighbor node in the *parent* datastructure. The neighbor node updates its *f-Score* by adding the new *g-Score* together with the heuristic distance from the goal. If the neighbor is not already included in the open set for exploration, then it is added to the open set and the process repeats. This algorithm can only end in success, with an optimal path found (line 12), or failure in the case that no such path is possible because all accessible nodes have been explored and the open set is empty (line 21).

This algorithm forms the basis for all pathfinding operations described in this chapter. There are alternatives which offer different properties and advantages such as D\* Lite, which maintains a memory of paths and adapts to changes in the graph [5]. Ultimately, however, the choice of path planning algorithm is a degree of freedom for the designer and does not significantly impact the procedures outlined in Chapter ?.

**LRA\***

Naively planning paths for all agents in a system quickly turns out to be insufficient in many cases. For example, two agents which must pass each other could very easily plan intersecting paths which result in a vertex or swap conflict as shown in figure Y.

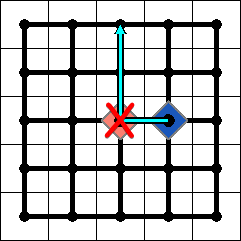
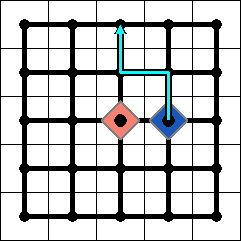
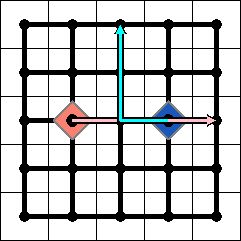


Figure 3: The LRA\* replanning procedure. On the left, two agents plan optimistic paths to their goals. A collision is detected, and the lower priority agent (blue) plans a new route, avoiding the immediate neighbor.

A simple approach, which is widely used in the videogame industry, is to allow A\* to execute until a collision is inevitable [6]. At the timestep where a collision is detected, the agent whose movement is blocked because of the collision instead re-plans its path by running a new instance of the A\* pathfinding algorithm. The starting node is the current location, while the goal node remains the same. To avoid the collision, the ***Neighbors*** function is modified so that the agent is unable to consider nodes in its immediate proximity which would lead to a collision. To do this it needs access to the system state at the current timestep to evaluate the position of other agents in the system: . The updated function is described in Algorithm 2.

|  |  |
| --- | --- |
| **Algorithm 2** Neighbors function for LRA\* | |
|  |  |
|  |  |
|  | Neighbors(): |
|  | **if** Position |
|  |  |
|  | **return** |

This approach is not very powerful for a host of reasons. The replanning procedure means that in densely populated graphs agents frequently undergo recalculation of their entire paths. The low amount of foresight given by the simple ***Neighbors*** function augmentation also frequently results in cycling or jostling behaviors in which agents shuffle back and forth attempting to find routes around each other. As the number of agents in the bottleneck increases, the situation grows to take arbitrarily long to resolve, never guaranteeing a solution will be found [6]. The algorithm is provably unable to resolve bottlenecks at all in certain graph conditions and provides no predictive power to prevent its use in such cases. These behaviors are immediately obvious in experimentation and quickly prove to yield insufficient results. A greater degree of inter-agent cooperation is needed.

**Multi-Agent Approaches**

A\* serves as a powerful tool for finding paths in any given instance of a graph traversal problem but, as discussed in the previous section, cannot be naively used to solve problems in which multiple agents share the same space.

This section presents the strategies employed by two families of algorithms which attempt to solve multi-agent problems. The first is called Windowed Hierarchical Cooperative A\* (WHCA\*), which combines the principles of three algorithms into one and is presented in [6]. The second family is built on a principle called Token Passing (TP). Its authors also present a set of criteria for determining whether an instance of an MAPD problem is guaranteed to be solvable [7].

These approaches will be adapted via the procedure laid out in Chapter ? as examples of the methodology so as to provide points of comparison. Their usage within the framework laid out in Chapter ? to produce experimental data in Chapter ?? will reveal each approach’s advantages, restrictions, and demonstrate the value of this work.

**Windowed Hierarchical Cooperative A\***

Windowed Hierarchical Cooperative A\* (WHCA\*) is a combination of three algorithms presented by David Silver in his 2005 paper “Cooperative Pathfinding” [6]. Building directly from the A\* and LRA\* implementations, Silver develops additional augmentations which enable agents to be aware of each other’s intentions, better analyze the space in which they travel via altering the heuristic function of A\* and increase the flexibility and computational speed of the pathfinding algorithm via the addition of a windowing function.

See Appendix ??? for details on the algorithm’s implementation within FleetBench.

**Cooperative A\***

Cooperative A\* (CA\*) is the first algorithm which achieves a degree of direct agent cooperation. By implementing a reservation table which is stored at some central authority which all agents may access, agents are able to keep track of each other’s plans. The reservation table is a representation of the graph structure which is augmented with a third dimension: time. Agent paths through the system are represented by a series of n-tuples with increasing time depth composed from the position and the time at which an agent will occupy that position. By including the time dimension the reservation table has predictive power which is accessible by any agent intending to plan a path.

To make use of this shared information, the A\* algorithm must be modified in several ways:

* Found paths need to be logged into the reservation table.
* Some notion of time depth must be included in the searching strategy, increasing for each new node explored. To do this, ***BuildPath*** must be expanded to include the time in its characterization of a node.
* The ***Neighbors*** subroutine must be modified to only return nodes which are unreserved in the next time step. It should also return the agent’s current node, if it is not reserved in the next time step (this is the *wait* action). This new routine is named ***FreeNeighbors***.
* In the case of a replan, the old plan must be removed from the reservation table so as not to falsely impact the performance of new searches. This procedure is implemented in a new routine, ***Replan***.

With these changes implemented, the algorithm achieves a degree of cooperation among agents such that no agent should be able to find a path which intersects another agent. This may mean that an agent is unable to find any path to the goal. In such cases, the agent must be allowed to not act, which exposes a weakness in the algorithm that can be felt in densely populated graphs.

Should an agent fail to find a path, it has no default behavior to fall back on. Its inability to create a plan may interfere with the plans of other agents. This behavior also indicates a high degree of ordering sensitivity: the first plans have the least restrictions, while the last plans may be impossible to create. Naively implementing a similar solution to LRA\* for agents which are disrupted by another agent’s failure to find a path, the ***Replan*** routine is introduced.

The CA\* process is described by Algorithm 3.

|  |  |  |
| --- | --- | --- |
| **Algorithm 3** Cooperative A\* (CA\*) | | |
|  | **function** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | **if** IsGoal(: |
|  | BuildPath( |
|  |  |
|  | **return** |
|  | **for** FreeNeighbors |
|  | Cost |
|  | **if** |
|  |  |
|  |  |
|  |  |
|  | **if** **not in** |
|  |  |
|  | **return** |
|  | **function** FreeNeighbors |
|  |  |
|  | **for** Neighbors: |
|  | **if** ( |
|  |  |
|  | **return** |
|  | **function** Replan |
|  |  |
|  | CAStar |
|  | **return** |

**Hierarchical Cooperative A\***

Silver notes that the choice of heuristic may cause problems in more challenging environments, where searches generate complicated paths which are vulnerable to being replanned as the dynamics of the system cause interruptions. He proposes the use of simple hierarchy to represent the search space abstractly, reducing the difficulty of the search operation. This new algorithm, in combination with an optimization in the form of a reversed-direction A\* search which is kept in memory, constitutes the Hierarchical Cooperative A\* (HCA\*) algorithm.

The key to the hierarchy is that it is an abstraction of the graph’s state which does not include other agents or obstacles. The distance from starting position to the goal is therefore a perfect estimate. This clearly cannot overestimate the distance to the goal, making it a sufficient heuristic to assure optimality in A\* searches.

The proposed hierarchy replaces the ***HDistance*** subroutine with a search in the reverse direction, from goal to current position, which ignores the presence of agents and obstacles in the graph. Employing an A\* search in place of the standard ***HDistance*** subroutine means that the distance from the currently evaluated node to the goal is a more precise estimate. This increase in accuracy reduces the number of search operations needed by better guiding the agents CA\* search toward the goal.

Further, the results of this reversed search are kept in memory as an optimization to avoid performance hits during operation due to the extra searches being performed. As an agent advances further along its plan, the hierarchical search data remains relevant because it is advancing further into already known data. Because the motion of agents does not impact the hierarchical search the search data is never invalidated by agent activity. If the properties of the graph change, then the stored data cannot be assured to be accurate.

In combination, these modifications to the heuristic function of the CA\* search are termed Reverse Resumable A\*. The procedure is outlined in Algorithm 4, serving as a drop-in replacement for the ***HDistance*** routine in CA\* (Algorithm 3). In this case, the goal is still the target node for the agent, but the reversed direction of the search means that the starting position is the goal. The search progresses toward the agent’s current position. When it is completed, the *g-Score* of the search from target to current position is equivalent to the hierarchical distance from agent to target and is returned.

Before executing the search procedure in lines 9-22, the algorithm checks to see if it has already stored the appropriate *g-Score* in its closed set of nodes. If the data is already available, the algorithm immediately returns the distance (lines 7-8). Declarations of the scoring, open, and closed sets must be moved out of the function in order to remain in memory after calls to the routine, as shown in lines 1-5.

|  |  |  |
| --- | --- | --- |
| **Algorithm 4** Reverse-Resumable A\* (RRA\*) | | |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | **if** |
|  | **return** |
|  |  |
|  |  |
|  |  |
|  | **if** IsGoal(: |
|  | **return** |
|  | **for** : |
|  | Cost |
|  | **if** |
|  |  |
|  |  |
|  |  |
|  | **if** **not in** |
|  |  |
|  | **return** |

**Windowed Hierarchical Cooperative A\***

A final alteration is made to solve a set of practical concerns with the previous algorithms. By imposing a windowing restriction that prevents the algorithm from searching too deeply the searching time is potentially reduced dramatically, time is not wasted evaluating potential collisions which may not occur in a more loosely scheduled system, and the sensitivity of the algorithm to agent ordering is significantly reduced [6]. This final change, taken together with the changes found in CA\* and HCA\* comprises the Windowed Hierarchical Cooperative A\* (WHCA\*) algorithm.

The windowing restriction is an alteration of the ***IsGoal*** function which compares the current search depth to the window size parameter , whose value is to be selected by the designer. If the search depth is equal to or greater than the size of the window, the search is terminated in the same fashion as if the agent had found a complete route to the goal node. The replacement algorithm, called ***Finished-WHCA***, is shown in Algorithm 5.

|  |  |
| --- | --- |
| **Algorithm 5** Finished function for WHCA\* | |
|  | Finished-WHCA |
|  | **return** |

Because of the HCA\* implementation providing a heuristic which guides the search in the direction of the optimal path toward the goal, forward progress is still assured so long as there is some path to the goal. Effectively, this means that for search depths less than the size of the window, the agent is navigating the base state of the graph where it considers other agents plans. Beyond the window the path is equivalent to the hierarchical abstraction from HCA\*. Only the base graph search path is logged into the reservation table of CA\* so as to avoid unnecessarily obstructing other agents with the theoretical path given by the abstraction—which is not guaranteed to be followed.

Notably, this implementation of WHCA\* is designed to solve the MAPF problem. When an agent reaches its goal, it is assumed that its objective is to remain on the goal as much as possible, moving only to allow another agent to reach its own goal. However, the warehousing environment is much more analogous to the MAPD problem. This leaves the process vulnerable to the same problems present in LRA\* because it is essentially the same at its core: new plans are made to avoid agents on a regular basis without deeply considering the disruptive impacts of doing so or what should occur when an agent has no possible actions to take according to the reservation table. To implement WHCA\* in an MAPD scenario, additional care must be taken to process such cases in a manner which does not terminate the simulation of the problem. The general collision resolution strategy employed in this case is described in Appendix ?.

**Token Passing**

A second family of approaches tackles the MAPD problem directly by making assertions about the class of problems which are solvable and therefore being provably complete, but not necessarily optimal. These algorithms employ a Token Passing strategy in which the token is a block of memory which represents combined knowledge of the state of the system at the current time step and into the future. By passing the token to agents one at a time, the agents can find and plan paths in a decoupled fashion before passing the modified token back to the central authority.

**Well-formed Problems and Completeness Guarantees**

The authors argue and prove that for a certain class of MAPD problems, solutions may be guaranteed [7]. Because of this, any algorithm which makes proper use of the conditions defining MAPD problems of this type can be said to be *complete* such that it will not fail to eventually find a solution to the problem.

The concept of an endpoint is introduced. An endpoint is defined as a node in which an agent could freely rest until the end of the time horizon. This amounts to an extension of the reservation table which reserves a node for an agent for all known timesteps in the future. This definition must be applied to pickup nodes, delivery nodes, and designated locations for agent parking in order to assure solutions can be found. Three sets of endpoints are defined:

* Vendpoints, which contains all nodes satisfying the endpoint definition.
* Vtask, which contains all pickup and delivery nodes.
* Vntask, which contains all endpoints which are not pickup or delivery nodes.

According to the authors, to guarantee the existence of a solution for a given MAPD problem, several prerequisites must be met:

1. The number of tasks in the system is finite. This does not preclude the addition of tasks in an on-line fashion, merely that the act of searching for a task does not take arbitrarily long.
2. Vntask must contain at least as many non-task endpoints as there are agents in the system.
3. For any two endpoints, a path in the graph exists which does not require passing over another endpoint.

MAPD problems satisfying these requirements are considered “well-formed” and solutions are guaranteed when using the related algorithms, presented in the next sections. Optimality of solutions is not guaranteed, as greedier algorithms may find faster solutions while sacrificing the completeness guarantee. Figure 4 presents three instances of an MAPD problem which demonstrate the second and third conditions. The instance on the left shows a well-formed problem: there are two agents, two non-task endpoints, and no path to any endpoint requires passing over another. The middle instance is not well-formed as there are fewer non-task endpoints than there are agents which violates the second requirement. The instance on the right is also not well-formed because the path to the delivery endpoint from a pickup endpoint is required to cross over an endpoint, violating the third requirement.

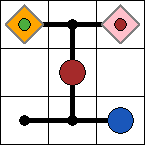
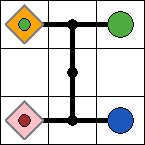
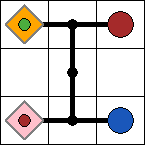


Figure 4: Three MAPD instances. Green circles indicated pickup endpoints. Blue circles indicate delivery endpoints. Brown nodes indicate non-task endpoints. Two agents, shown as diamonds, are in the problem space.

**Token Passing**

Token Passing (TP) is very similar to CA\* in that it uses reservations to govern the set of nodes which are accessible to the agent seeking a path. However, using the endpoint definition, the reservation table must now reserve an agent’s endpoint for all timesteps into the future. The authors prove that an agent is only able to plan paths ending in an endpoint, so the final node in any agent’s path is not allowable into the path of any agent’s search operations. This imposes a restriction on the set of tasks to which an agent can be assigned: no task whose pickup node or delivery node are the end of another agent’s path in the token may be considered a valid assignment.

Unlike WHCA\*, TP takes advantage of a precompute step which executes before the act of solving the problem begins. This is only possible on maps which remain static during the solving process, else the information from the precompute step would become unreliable. During the preprocessing step distances and paths from every node in the graph to each endpoint are found. Because an agent operating in this algorithm will only ever find paths toward an endpoint, this data may also be used in place of the ***HDistance*** routine for notions of distance from an agent’s goal. For large graphs this may take a significant amount of time but need only be done once, yielding results which are reusable by the algorithm for the lifetime of the problem. This shortest-path data is stored in the problem data as .

The primary benefit of this preprocessing is that it becomes trivial to select the task with the nearest pickup location, significantly reducing the amount of travel time before an agent is performing useful work. The act of finding this task is given as a named subroutine, ***BestTask***, in Algorithm 6. By using the stored data from the preprocessing step, this process is made efficient. This information cannot be used to quickly generate the path an agent will take while executing the task, however, because the agent must still be conscious of other agents and avoid collisions.

|  |  |  |
| --- | --- | --- |
| **Algorithm 6** Finding Nearest Task | | |
|  | **function** BestTask |
|  |  |
|  | **return** |

Pathfinding operations in TP are classed as one of two types; a search for a path to a task endpoint, called ***Path1***, or a search for a path to a non-task endpoint, called ***Path2***. If the agent is content to remain in its current position, the trivial path denoted ***Stay*** is found, and the agent waits in its current position. ***Path2*** is executed when the agent is not able to complete any task, while also unable to remain in its current position due to the path of another agent. In either case, the pathfinding is done in a manner identical to CA\*, using the augmented reservation table which respects an agent’s intent to rest forever in a location when it reaches the end of its planned path. If it is ever impossible to find a path, the agent will use its ability to remain in place, waiting until path can be found for its assigned task.

The complete TP algorithm is described in Algorithm 7. Unlike other algorithms presented in this chapter, TP is not designed exclusively to find paths through the system state. It also performs operations on the set of tasks in the system before seeking paths. A named routine, called ***Preprocess***, is introduced to represent the precomputation step, which returns all information necessary to define , including all computed distances (line 1). From then on, TP is executed continuously, using ***Update*** to add new tasks to the task set (lines 3-4). It first assigns the best task in the system to each agent, removing the task from the task set and planning an optimal path (lines 5-11). If there is no assignable task, an agent should be allowed to rest in placeso long as its current position is not the endpoint of another task in the system to avoid future blockages (lines 12-13). Otherwise, the agent should navigate to a non-task endpoint to make space for other agents (lines 14-15). At the end of the loop, the simulation advances by executing agent plans and incrementing the system timestep (line 16).

|  |  |  |
| --- | --- | --- |
| **Algorithm 7** Token Passing (TP) | | |
|  | Preprocess |
|  |  |
|  | **while** |
|  | Update |
|  | **for** |
|  |  |
|  | **if** |
|  |  |
|  | Assign to |
|  |  |
|  | Path1 |
|  | **else if** |
|  | Stay |
|  | **else**: |
|  | Path2 |
|  | All agents execute plans; |
|  | **function** Path1 |
|  | **if**  reached |
|  | CAStar |
|  | **else**: |
|  | CAStar |
|  | **function** Path2 |
|  |  |
|  | CAStar |
|  | **function** Stay |
|  |  |
|  |  |

**Token Passing with Task Swaps**

TP is simple and efficient in many cases but shows an algorithmic inefficiency in the way tasks are assigned: a task which may be completed much more quickly by an agent later in the priority queue may be claimed by an agent which would complete the task more slowly.

It is possible to further optimize the selection of tasks by enabling agents to exchange tasks when the time to complete the task is reduced in doing so. To preserve the pickup and delivery analogy, agents may only swap task assignments before an agent has interacted with the pickup portion of the task. Before that occurs, a task is considered to be “unexecuted”. Afterwards, the task is being “executed” and can no longer be handed off to another agent. This procedure is implemented in Token Passing with Task Swaps (TPTS).

The primary concern in expressing this process as a single algorithm is the recursive nature in which task swaps must occur. An agent may be better suited to complete task than , but if is not able to make its way to a free endpoint from its current position, the task swap should fail and should be allowed to continue with its previously planned actions. This process is represented by the ***GetTask*** procedure in Algorithm 8.

|  |  |  |
| --- | --- | --- |
| **Algorithm 8** Token Passing with Task Swaps (TPTS) | | |
|  | Preprocess |
|  |  |
|  | **while** |
|  | Update |
|  | **for** |
|  | GetTask |
|  | All agents execute plans; |
|  | **function** GetTask |
|  |  |
|  | **while** |
|  |  |
|  |  |
|  | **if** no agent assigned to |
|  | Assign to |
|  | Path1 |
|  | **return** |
|  | **else**: |
|  |  |
|  | agent assigned to |
|  | Path |
|  | Unassign from ; Remove from |
|  | Path1 |
|  | **if** : |
|  | GetTask |
|  | **if** : |
|  | **return** |
|  | **else**: |
|  |  |
|  | **if** Position |
|  | Path2 |
|  | **if** |
|  | **return** |
|  | **else**: |
|  | **if** |
|  | Stay |
|  | **else**: |
|  | Path2 |
|  | **return** |
|  | **return** |

Once again, the data found by preprocessing the graph before starting to solve the MAPD problem is reused to supply distance information to the ***BestTask*** and ***HDistance*** routines (lines1-2). TPTS is then executed continuously, seeking assignments for free agents using ***GetTask*** (lines 5-6). As before, a subset of viable tasks is taken from the set of all tasks, evaluated based on the reservations of other agents in the system (line 9). One by one, tasks in this subset are evaluated and assigned to agents if they do not have an assigned agent, or an attempt is made to swap task assignments between the current agent and the one currently assigned to the task (lines 10-26). Assignments are made assuming that the task swap will succeed but the information before the swap occurs must be stored in case the swap fails. If the current agent would outpace previously assigned agent on the way to the pickup node, then the swap takes place by recursively calling ***GetTask*** until no more tasks can be assigned to agents in the recursion. This can end when all tasks in the viable task set are assigned (lines 32, 38) or when an agent is offered a task with no assigned agent (line 16). If either of these occur, then the recursion on line 24 resolves until the primordial ***GetTask*** call returns.

So long as the requirements for well-formedness are met, the authors of TP and TPTS guarantee that each algorithm solves all MAPD instances [7].

[1] W. Hoenig *et al.*, “Multi-Agent Path Finding with Kinematic Constraints,” *Proc. Int. Conf. Autom. Plan. Sched.*, vol. 26, pp. 477–485, Mar. 2016, doi: 10.1609/icaps.v26i1.13796.

[2] R. Stern *et al.*, “Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks,” *Proc. Int. Symp. Comb. Search*, vol. 10, no. 1, pp. 151–158, Sep. 2021, doi: 10.1609/socs.v10i1.18510.

[3] P. E. Hart, N. J. Nilsson, and B. Raphael, “A Formal Basis for the Heuristic Determination of Minimum Cost Paths,” *IEEE Trans. Syst. Sci. Cybern.*, vol. 4, no. 2, pp. 100–107, Jul. 1968, doi: 10.1109/TSSC.1968.300136.

[4] R. Dechter and J. Pearl, “Generalized best-first search strategies and the optimality of A\*,” *J. ACM*, vol. 32, no. 3, pp. 505–536, Jul. 1985, doi: 10.1145/3828.3830.

[5] S. Koenig and M. Likhachev, “Fast replanning for navigation in unknown terrain,” *IEEE Trans. Robot.*, vol. 21, no. 3, pp. 354–363, Jun. 2005, doi: 10.1109/TRO.2004.838026.

[6] D. Silver, “Cooperative Pathfinding,” *Proc. AAAI Conf. Artif. Intell. Interact. Digit. Entertain.*, vol. 1, no. 1, pp. 117–122, Sep. 2021, doi: 10.1609/aiide.v1i1.18726.

[7] H. Ma, J. Li, T. K. S. Kumar, and S. Koenig, “Lifelong Multi-Agent Path Finding for Online Pickup and Delivery Tasks.” arXiv, May 30, 2017. Accessed: Nov. 05, 2023. [Online]. Available: http://arxiv.org/abs/1705.10868