Multi-Agent Path Finding Problem and its Applications in Agriculture

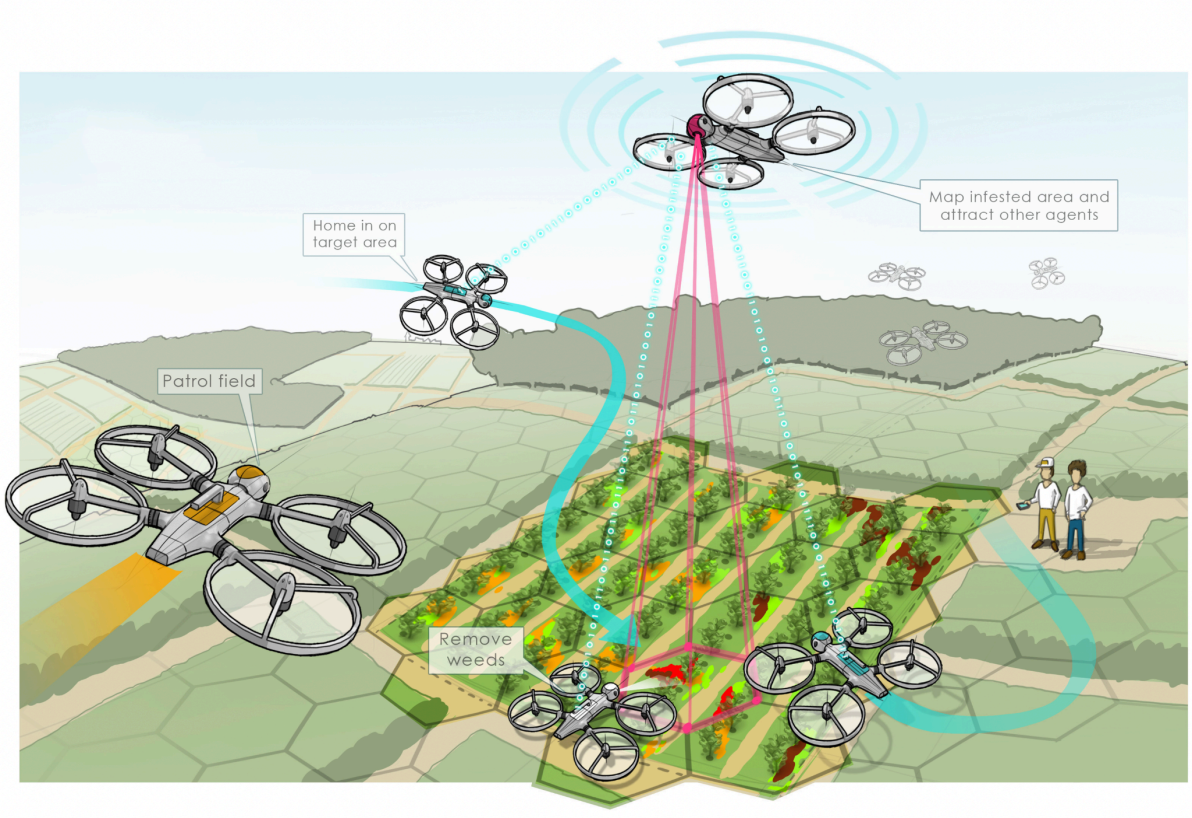
AI61005: Artificial Intelligence Foundations and Applications

Term Report

Aviral Jain (18AG3AI08) | Devansh Arora (18AG3AI09)

Introduction

The ever-increasing global population demands more food, and with limited supply of crops, the agricultural industry needs a revolution to fulfill the demands. Precision agriculture is nowadays the name given to technological developments for assessing and responding to local variations and emergencies in crop health on farms. It involves collecting, interpreting and acting on data which has potential to be automatized with robotic systems. Satellites or robots (especially UAVs or ground robots) are used to collect such images over farm lands. With the help of such data, farmers can take smart decisions and act ahead of time. Since agricultural fields have complex topologies and different constraints, the problem of optimized path routing of these swarms is important to be tackled. A current research area is improving this automation, in particular managing robot swarms for precision agriculture.



This problem statement could be mapped to the Multi-agent pathfinding domain.

Multi-agent pathfinding is to find the paths for multiple agents from their current locations to their target locations without colliding with each other, while at the same time optimizing a cost function, such as the sum of the path lengths of all agents. It is a generalization of pathfinding. These path-planning problems are NP-hard. This means that the solution space grows very rapidly with the problem size, and there is no efficient algorithm for finding an exact optimal solution. Therefore in general it is best to search for an approximate best solution. At present, there exist several algorithms for multi-agent path planning problems, like the Nearest Neighbor Algorithm, Ant Colony Optimization, Conflict Based Search, A*, Bellman-Held-Karp, etc. However, such algorithms are typically incomplete; in other words, not proven to produce a solution within polynomial time. We plan to explain and use the Conflict Based Search algorithm to solve the Multi-agent path-finding problem.

Problem Modelling

The swarm agents used can be Unmanned Aerial Vehicles (UAVs) or Ground Robots. To formulate the problem, we need to keep in mind that ground robots are subject to facing obstacles in their path, whereas UAVs are not. Therefore, in general, it is best to search in a grid world with obstacles (for ground robots), so that it can be easily generalized to a no-object continuous world model (UAVs).

Continuous World vs Grid world with Obstacles

Black squares represent trees, black round labels represent crops in a line, while blue labels in the left bottom of the field represent the agent's initial position.

Each point on the 2D plane has a unique position (x, y). Agents are modeled as point particles too. Their state is given by their current 2D position (x, y) and their linear velocity υ, which is uniform for all agents. The agents always started at position (0, 0), where they had to return afterward. Each state is defined as the agent's location (x, y) at some point in time *t*.

Some of the following assumptions were made to further confine the scope of the problem:

1. Number of agents: One up to five agents can be employed.
2. Agent types: There is only one type of agent, i.e. ground robot.
3. Origin: The agents start at a common location.
4. Destination: All agents must return to their start locations.
5. Scenario: The field layout and Points of Interest (POI) do not change, which is typical for an agricultural field on a typical day or even week.
6. Observability: Global, i.e. the field layout and all the POIs are known in advance.
7. Space: Agents can move freely in a two-dimensional space.
8. Interactions: The time needed for performing agriculture-related tasks at each interest point (e.g. pruning, spraying, crop collection, taking photos, etc.) is ignored. Interactions between agents (i.e. collisions) are also ignored.

Now, we shall briefly look at the Conflict Based Search (CBS) algorithm.

Conflict-Based Search (CBS) Algorithm

CBS is a **two-level algorithm** that converts the problem into the single 'joint agent' model. At the high level, a search is performed on a Conflict Tree (CT) which is a tree based on conflicts between individual agents. Each node in the CT represents a set of constraints on the motion of the agents. At the low level, fast single-agent searches are performed to satisfy the constraints imposed by the high-level CT node. In many cases, this two-level formulation enables CBS to examine fewer states than A* while still maintaining optimality. Single-agent pathfinding is the problem of finding a path between two vertices in a graph. Solving pathfinding problems optimally is commonly done with search algorithms based on the A* algorithm. Such algorithms perform a best-first search that is guided by $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the shortest known path from the start state to state n and $h(n)$ is a heuristic function estimating the cost from n to the nearest goal state. If the heuristic function h is admissible, meaning that

it never overestimates the shortest path from n to the goal, then A* (and other algorithms that are guided by the same cost function) are guaranteed to find an optimal path from the start state to a goal state, if one exists.

The following definitions will be used in our remaining discussion:

• We use the term path only in the context of a single agent and use the term solution to denote a set of k paths for the given set of k agents.

• A **constraint** is a tuple (a_i, v, t) where the agent a_i is prohibited from occupying vertex *v* at time step *t*.

During the course of the algorithm, agents will be associated with constraints. A consistent path for agent a_i is a path that satisfies all its constraints. Likewise, a consistent solution is a solution that is made up of paths, such that the path for any agent a_i is consistent with the constraints of a_i.

• A **conflict** is a tuple (a_i, a_j, v, t) where agent a_i and agent a_j occupy the vertex *v* at time point *t*. A solution (of k paths) is valid if all its paths have no conflicts. A consistent solution can be invalid if, despite the fact that the individual paths are consistent with the constraints associated with their agents, these paths still have conflicts.

The key idea of CBS is to grow a set of constraints and find paths that are consistent with these constraints. If these paths have conflicts and are thus invalid, the conflicts are resolved by adding new constraints. CBS works on two levels. At the high level, conflicts are found and constraints are added. The low level finds paths for individual agents that are consistent with the new constraints.
Next, we describe each part of this process in more detail.

The Constraint Tree (CT)

At the high level, CBS searches for a tree called the constraint tree (CT). Each node *N* in the CT consists of :

1. A set of constraints ($N_{constraints}$). Each of these constraints belongs to a single agent. The root of the CT contains an empty set of constraints. The child of a node in the CT inherits the constraints of the parent and adds one new constraint for one agent.

2. A solution ($N_{solution}$), a set of k paths, and one path for each agent. The path for the agent a_i must be consistent with the constraints of a_i. Such paths are found by the low-level search.

3. The total cost (N_{cost}) of the current solution (summed over all the single-agent path costs). This cost is referred to as the ***f-value*** **of node N**.

Node N in the CT is a goal node when N's solution is valid, i.e., the set of paths for all agents has no conflicts. The high level performs a best-first search on the CT where nodes are ordered by their costs. In our implementation, ties are broken in favor of CT nodes whose associated solution contains fewer conflicts. Further ties were broken in a *FIFO* manner.

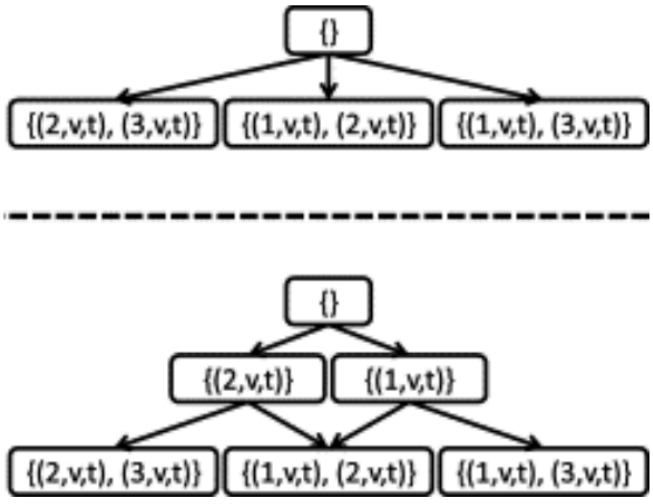
Processing a node in the CT

Given the list of constraints for node N of the CT, the low-level search is invoked. The low-level search (described in detail below) returns one shortest path for each agent, a_i, that is consistent with all the constraints associated with a_i in node N. Once a consistent path has been found for each agent (with respect to its own constraints) these paths are then validated with respect to the other agents. The validation is performed by iterating through all time steps and matching the locations reserved by all agents. If no two agents plan to be at the same location at the same time, this CT node N is declared as the goal node, and the current solution ($N_{solution}$) that contains this set of paths is returned. If, however, while performing the validation, a conflict $C = (a_i, a_j, v, t)$ is found for two or more agents a_i and a_j, the validation halts and the node is declared a non-goal node.

Resolving a conflict

Given a non-goal CT node N whose solution $N_{solution}$ includes a conflict $C_n = (a_i, a_j, v, t)$ we know that in any valid solution, at most one of the conflicting agents (a_i and a_j) may occupy vertex *v* at time *t*. Therefore, at least one of the constraints(a_i, v, t) or (a_j, v, t) must be added to the set of constraints in $N_{constraints}$. To guarantee optimality, both possibilities are examined and node N is split into two children. Both children inherit the set of constraints from N. The left child resolves the conflict by adding the

constraint (a_i, v, t) and the right child adds the constraint (a_j, v, t). Note that for a given CT node N, one does not have to save all its cumulative constraints. Instead, it can save only its latest constraint and extract the other constraints by traversing the path from N to the root via its ancestors. Similarly, with the exception of the root node, the low-level search should only be performed for agent a_i which is associated with the newly added constraint. The paths of other agents remain the same as no new constraints are added for them. In conflicts of $k > 2$ agents, it may be the case that while performing the validation between the different paths a k-agent conflict is found for $k > 2$. There are two ways to handle such k-agent conflicts. We can generate k children, each of which adds a constraint to k−1 agents (i.e., each child allows only one agent to occupy the conflicting vertex v at time t). Or, an equivalent formalization is to only focus on the first two agents that are found to conflict, and only branch according to their conflict. This leaves further conflicts for deeper levels of the tree. This is illustrated in the Figure below. The top tree represents a variant of CT where k-way branching is allowed for a single conflict that includes k-agents for the case where $k=3$. Each new successor adds $k-1$ (=2) new constraints (on all agents but one). The bottom tree presents a binary CT for the same problem. Note that the bottom middle state is a duplicate state, and if duplicate detection is not applied there will be two occurrences of this node instead of one. As can be seen, the size of the deepest layer in both trees is identical. The complexity of the two approaches is similar, as they both will end up with k nodes, each with $k-1$ new constraints. For simplicity, we implemented and described only the second option.



(a_i, a_j, v_1, v_2, t) where two agents "swap" locations (a_i moves from v1 to v2 while a_j moves from v_2 to v_1) between time step t to time step t+1. An edge constraint is defined as (a_i, v_1, v_2, t), where agent a_i is prohibited from starting to move along the edge from v_1 to v_2 at time step t (and reaching v_2 at time step *t+1*). When applicable, edge conflicts are treated by the high level in the same manner as vertex conflicts.

When validating the two-agent solution given by the two individual paths, a conflict is found when both agents arrive at vertex *D* at time step 2. This creates a conflict $(a_1, a_2, D, 2)$. As a result, the root is declared as a non-goal, and two children are generated in order to resolve the conflict. The left child adds the constraint $(a_1, D, 2)$ while the right child adds the constraint $(a_2, D, 2)$. The low-level search is now invoked (line 23) for the left child to find an optimal path that also satisfies the new constraint. For this, a_1 must wait one time step either at A_1 or at S_1 and the path $\langle S_1, A_1, A_1, D, G_1 \rangle$ is returned for a_1. The path for a_2, $\langle S_2, B_1, D, G_2 \rangle$ remains unchanged in the left child. The total cost for the left child is now 7. In a similar way, the right child is generated, also with a cost of 7. Both children are inserted into OPEN. In the next iteration of the while loop, the left child is chosen for expansion, and the underlying paths are validated. Since no conflicts exist, the left child is declared a goal node and its solution is returned as an optimal solution.

Low level: Find paths for CT nodes

The low-level search is given an agent, a_i, and the set of constraints associated with a_i. It performs a search in the underlying graph to find an optimal path for agent a_i that satisfies all its constraints while completely ignores the other agents. The search space for the low-level search has two dimensions: the spatial dimension and the time dimension. Any single-agent pathfinding algorithm can be used to find the path for agent a_i, while verifying that the constraints are satisfied. We implemented the low-level search of CBS with **A*** which handled the constraints as follows. Whenever a state *(v,t)* is generated where *v* is a location and *t* a time step and there exists a constraint (a_1, v, t) in the current CT (high-level) node, this state is discarded. The heuristic we used is the shortest path in the spatial dimension, ignoring other agents and constraints. Duplicate state detection and pruning (DD) speeds up the low-level procedure. Unlike single-agent pathfinding, the low-level state space also includes the time dimension and dynamic

'obstacles' caused by constraints. Therefore, two states are considered duplicates if both the position of a_i and the time step are identical in both states.

Discussion

The implementation of the above can be found here: https://github.com/jainaviral898/mapp-cbs-aifa

Although the situations studied are static in the sense that the agents know the overall plan in advance, they can become dynamic (particularly) in the case of UAVs whose POIs are based on daily remote sensing observations, leading to precision agriculture. As a result, making the most use of path-planning algorithms is critical, and the time it takes for them to execute has an impact on the practicality of any option.

In future work, we can spend more time on the implementation of other multi-agent path planning algorithms in order to make a more fair judgment of our methods. Furthermore, we can take into account the actual limits of UAVs (e.g., maximum flying time) and ground robots (e.g., maximum distance traveled in a number of hours). For UAVs and ground robots, this will necessitate the adoption of more realistic physical models. Finally, a dynamic scenario can be investigated in which the agents have no prior knowledge of the area and must combine their ad-hoc knowledge to solve the optimization issue in the most efficient manner feasible.

References

1. MULTI-AGENT PATH PLANNING OF ROBOTIC SWARMS IN AGRICULTURAL FIELDS (N. Botteghi, A. Kamilaris, L. Sinai, B. Sirmacek)
2. Improved Heuristics for Multi-Agent Path Finding with Conflict-Based Search Jiaoyang Li , Ariel Felner, Eli Boyarski , Hang Ma, and Sven Koenig.