Multiagent Pathfinding
AAAI Technical Report WS-12-10

Independence Detection for Multi-Agent Pathfinding Problems

Trevor Standley
Google Inc.
340 Main St.
Venice, CA 90291
trevorstandley@gmail.com

Abstract

Problems that require multiple agents to follow non-interfering paths from their current states to their respective goal states are called multi-agent pathfinding problems (MAPFs). In previous work, we presented Independence Detection (ID), an algorithm for breaking a large MAPF problem into smaller problems that can be solved independently. Independence Detection is complete and can be used in combination with both optimal and approximation algorithms. This paper serves as an introduction to Independence Detection and aims to clarify its details.

Introduction

Pathfinding, or planning a route to a destination that avoids obstacles, is a classic problem in AI. When only a single agent is present, the problem can usually be effectively solved using the A* algorithm (Hart *et al.* 1968). Unfortunately, standard search algorithms such as A* quickly become intractable with multiple agents. Multi-agent pathfinding has applications in robotics, aviation, and vehicle routing (Wang and Botea 2008; Svestka and Overmars 1996), and is becoming increasingly important in modern video games.

Problem Formulation

Examples of cooperative pathfinding problems include planning the motions of multiple robotic arms, each of which must accomplish a separate goal without moving into one another; scheduling trains in a rail road network without sending a pair of trains on a collision course; and deciding actions for automobiles approaching an intersection so that each may pass through safely and quickly (Dresner and Stone 2004). For the sake of clarity and simplicity, we will describe Independence Detection using an eight-connected grid world like the one in Figure 1.

We use the problem formulation from our previous works (Standley 2010; Standley and Korf 2011), in which each agent occupies a single cell of the grid world and has a unique destination. During a single timestep, each agent can either wait or move to any of its eight adjacent cells if that cell is free. A cell is free if it does not contain an obstacle and will not be occupied by another agent at the end of the timestep. The cost of a single agent's path is the total number of timesteps that the agent spends away from its goal, and the cost of the entire solution is the sum of all single-agent path costs. Diagonal moves are allowed even when the two cells on the opposing diagonal are not free, for example, agent 5 in Figure 1 can immediately move to its destination. Agents arranged in a cycle are allowed to simultaneously follow one another, resulting in a rotation of the agents that does not require an empty cell. For example, agents 1, 2, 3, and 4 can simultaneously move to their destinations in a single timestep. However, transitions in which agents pass through each other, including diagonally crossing, are prohibited even when those agents never occupy the same position during the same timestep. For example, agents 6 and 7 cannot simultaneously move to their destinations.

Copyright © 2012, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

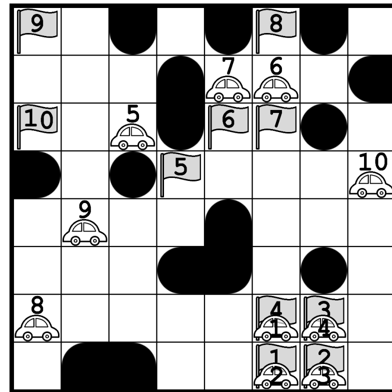

Figure 1: A small grid world instance. Cars represent initial agent positions. Flags represent destinations. Obstacles are in black.

Related Work

Multi-agent pathfinding problems have usually been solved in one of two ways in the literature. In global search approaches, such as those in (Surynek 2009; Ryan 2008; Svestka and Overmars 1996; Sharon *et al.* 2011), the entire set of agents is treated as a single entity and paths are found for all agents simultaneously. Alternatively, in decoupled approaches, such as those in (Silver 2005; Wang and Botea 2008; Jansen and Sturtevant 2008a; 2008b), paths are found for each agent one at a time, and informa-

tion about the paths of other agents is used to ensure that no paths conflict. Global search approaches typically have the advantage of being *complete*, meaning that they will always eventually find a solution to any problem if a solution exists, but are often intractable for even small numbers of agents. On the other hand, decoupled approaches are fast, but usually incomplete.

We presented Independence Detection in (Standley 2010) and refined it in (Standley and Korf 2011). Independence Detection (ID) decomposes a problem instance into independent subproblems and can do so without compromising optimality. ID uses a more costly global search algorithm only on subproblems for which no such independence can be found. Independence Detection can be both complete and optimal when coupled with a complete and optimal global search algorithm. Alternatively, ID offers a trade-off between solution quality and computation time without sacrificing completeness.

To our knowledge, all optimal algorithms in the literature either make use of the Independence Detection algorithm, or have difficulty solving problems with more than 20 agents. In contrast, optimal algorithms that make use of ID can often solve problems with more than 100 agents in seconds.

Independence Detection

ID is an algorithm that is used in conjunction with a complete search algorithm such as A*, OD (Standley 2010), or ICTS (Sharon *et al.* 2011). Since these search algorithms are all exponential in the number of agents, they are effective only for small numbers of agents. In order to solve larger problems, ID partitions the agents into several smaller **travel groups** in such a way that the optimal paths found for each independent travel group do not conflict with the paths of other travel groups. Therefore, the paths for all travel groups constitute a solution to the entire problem.

To explain Independence Detection, we will first describe a simple algorithm that detects some kinds of independence in problems in which agents' paths are less likely to interfere with one another. Then, we will describe two important refinements, each of which drastically improves the algorithm's performance.

Simple Independence Detection

A simple independence detection (SID) algorithm works as follows: First, assign each agent to its own travel group. Then, find a path for each agent's travel group. Some of these paths probably contain conflicts with the paths of other travel groups. By simulating the actions of every agent following these preliminary paths, we can discover conflicts among the paths. We then merge the first two travel groups whose paths are found to conflict and find a new set of paths for the merged travel group using the complete global search algorithm. We can repeat this process of merging two conflicting travel groups into a larger travel group until no conflicts are found. We can be sure that this algorithm will terminate because in the worst case, it will merge all the agents into a single travel group and find paths for this group using the global search algorithm. There are three eventualities. First, SID could find a set of travel groups that are independent, in which case the algorithm terminates with a valid solution. Second, the algorithm could find a travel group for which no solution can be found, in which case the problem has no solution. Finally, the algorithm could combine all of the agents into a single travel group, in which case SID didn't find any independence and must delegate the entire problem to the global search algorithm.

The pseudocode for this simple independence detection algorithm is copied from (Standley 2010) as Algorithm 1.

Algorithm 1 Simple Independence Detection

```
1: assign each agent to a singleton travel group
2: plan a path for each travel group with A*
3: repeat
4:     simulate execution of all paths until a conflict occurs
5:     merge two conflicting travel groups into a single travel
       group
6:     use global search to generate paths for the new travel group
7: until no conflicts occur
8: solution ← paths of all travel groups combined
9: return solution
```

Refinement 1: Illegal Move Table

Because the running time of the algorithm is dominated by the time to plan paths for the largest travel group, performance can be improved drastically by avoiding unnecessary merges. This is our intuition behind our first refinement. Instead of always merging conflicting travel groups, a more sophisticated algorithm can find an alternative set of paths for one of the groups, one that avoids the paths of the other conflicting group.

If we wish to ensure optimality, the alternative paths for a travel group must have the same total cost as the initial paths for that travel group. We refer to this as the optimality constraint of ID. To satisfy this requirement, the global search algorithm can be given a cost limit, so that it does not consider paths of greater cost.

The alternative paths are only useful if they obviate the need to merge the two groups, so the new paths must avoid all possible conflicts with the other travel group. Thus, the global search algorithm is also given a table called the **illegal move table**, which records which moves are illegal at which timesteps. The illegal move table is similar to the reservations table used in Silver's HCA* (Silver 2005) except that it stores illegal moves rather than illegal tiles. When the global search algorithm is considering a move for an agent, it consults the illegal move table at the appropriate timestep to determine whether that move would result in yet another conflict with the other travel group. It can treat such moves as illegal.

For example, if the paths of two travel groups G_1 and G_2 conflict, we can try to replan the paths of G_1. First, we would populate the illegal move table based on the transitions that agents in G_2 make when following G_2's current paths at every timestep. When our global search algorithm tries to find another set of paths for G_1, it only considers moves that are not in the illegal move table and therefore

will not conflict with the moves of agents in G_2. If a new set of paths is found, then the conflict between these two groups has been resolved. However, since we have made some moves illegal, the algorithm may not be able to find an alternative set of paths. In this case, we can try to replan the paths of G_2.

With this refinement, the algorithm vastly outperforms the simple independence detection algorithm because it is able to recover from poorly chosen paths.

Refinement 2: Conflict Avoidance Table

Giving our algorithm the foresight to avoid future conflicts also drastically improves performance.

Whenever the global search algorithm finds paths for a travel group, it is important for the algorithm to avoid other agents' paths in order to reduce the likelihood of future travel group merges and replans. Toward this goal, the full ID algorithm uses a table similar to the illegal move table, which is called the **conflict avoidance table**. This table stores all the current tentative moves of all agents not in the travel group being planned.

We want our global search algorithm to generate paths containing as few moves in conflict with the conflict avoidance table as possible. Toward this end, the algorithm should have multiple priorities. If the overall algorithm must achieve optimal solutions, then the first priority is to return a solution with the lowest cost. Among paths with the lowest cost (there are often many paths with optimal cost) the algorithm should choose a path with the fewest conflict avoidance table violations. If optimality is not as important as running time, then avoiding conflict table violations should be the first priority.

For A*-like global search algorithms this can be easily achieved. In such algorithms, tie-breaking between nodes with the same minimum $f(n)$ cost is usually done by choosing the node with the lowest $h(n)$ to achieve the best performance. To avoid future conflicts, the algorithm can keep track of another number for every node in addition to $h(n)$ and $g(n)$, denoted $v(n)$. $v(n)$ represents the number of conflicts with existing paths that occur on the best path to a node n. For each child node c that results from making move m from its parent node p, we set $v(c) = v(p) + 1$ whenever m conflicts with the conflict avoidance table, and $v(c) = v(p)$ otherwise. The general search algorithm can then break ties in $f(n)$ by the lowest $v(n)$. This tie-breaking strategy ensures that among the many sets of optimal paths, the paths returned during replanning will have the fewest conflicts with the current paths of agents outside the group (Standley 2010). If optimality is not the primary concern, the algorithm can just as easily break $v(n)$ ties with $f(n)$ instead, which results in a faster, yet suboptimal algorithm.

Global search algorithms like ICTS can probably achieve this by keeping a cutoff value for the number of conflict table violations in each node of the top level search and expanding nodes in order of lowest cost, and then lowest $v(n)$ to obtain optimal solutions.

Using the conflict avoidance table should not be overlooked. Our experiments below show a speedup of at least as many orders of magnitude as our first refinement.

Full Independence Detection

The full ID algorithm starts by assigning each agent to its own travel group. It then finds an initial path for each travel group independently, guided by the conflict avoidance table. Next, ID looks for conflicts within its current set of paths. Upon detecting a conflict, ID attempts to find an alternative set of paths for one of the conflicting travel groups, ensuring that the new paths do not conflict with the other travel group. If this fails, it repeats this process with the other of the conflicting travel groups. If both attempts to find alternative paths fail, ID merges the conflicting groups and cooperatively plans a set of paths for the new travel group. All paths are planned with a constantly updated conflict avoidance table to minimize future conflicts.

The pseudocode for this full independence detection algorithm is adapted from our prior work (Standley 2010).

Algorithm 2 Independence Detection

```
1: create a singleton travel group for each agent
2: plan optimal paths for each travel group with A*
3: call Resolve Conflicts with all paths
4: solution ← paths of all groups combined
5: return solution
```

Algorithm 3 Resolve Conflicts

```
1: fill conflict avoidance table with every path
2: repeat
3:     simulate execution of all paths until a conflict between two
       travel groups G1 and G2 occurs
4:     if G1 and G2 have not conflicted before then
5:         fill illegal move table with the current paths for G2
6:         find alternative paths for G1 that do not conflict with G2
           and satisfy cost limits
7:         if failed to find such paths then
8:             fill illegal move table with the current paths for G1
9:             find alternative paths for G2 that do not conflict with
               G1 and satisfy cost limits
10:        end if
11:    end if
12:    if failed to find alternative paths for both G1 and G2 then
13:        merge G1 and G2 into a single group
14:        cooperatively plan new group with OD
15:    end if
16:    update conflict avoidance table with changes made to paths
17: until no conflicts occur
```

It's important to keep in mind that under our formulation, when agents reach their goals, they can still conflict with other agents who may try to pass through their goals while they are stopped on them. ID must try to avoid such conflicts, which can be accomplished by filling in the illegal move table and conflict avoidance table appropriately for all timesteps after agents have reached their goals.

Optimizations

Group Order for Replanning

Choosing which travel group to replan first when two groups collide is an obvious thing to optimize. In preliminary experiments we found that it works well to first replan the travel

group that took the least time to plan initially, and that it is better than simply choosing the travel group with the fewest agents.

Initial Path Choice

Another important and helpful modification of ID involves the initial paths used. ID starts with a path for each agent, and then resolves the conflicts among those agents. There can be many choices for the initial paths, however, and a choice with fewer initial conflicts will lead to fewer replans and group merges. To achieve this, we find a path for every agent twice in the initialization step. We first find a path for every agent in turn while avoiding conflicts with the paths of all previously planned agents whenever possible, using the conflict avoidance table. On the second pass, we then find another path for every agent that tries to avoid conflicts with the newest path found for every other agent (either from the first pass or the current second pass). This ensures that the final path we find for every agent maximally avoids a path found for every other agent on either the first pass or the second pass.

Prioritize Avoiding Large Groups

Also, when finding paths, some travel groups are more important to avoid than others. Large groups in particular are harder to replan, and if replanning fails, we risk creating an even larger travel group. By storing a cost for every move in the conflict avoidance table, and incrementing $v(n)$ by the cost at every node expansion rather than simply incrementing by 1, we can effectively bias the algorithm to avoid large travel groups. In preliminary experiments, we set the cost to the group size, and achieved modest performance improvements.

Avoid Futile Replanning

When an agent collides with an agent that is stopped at its goal, it's impossible to find an alternative path for the second agent's travel group that has the same optimal cost and avoids this collision, because all paths for the second agent's travel group will still result in the second agent being stopped at its goal. Therefore, replanning the second agent's group is futile, and we can avoid wasting time recomputing new paths for such groups. Again, this modification results in modest improvements in running time.

Experiments

All of our experiments were run on an Intel Core i7 @ 2.4GHz using benchmarks like those proposed in (Silver 2005) and used in (Standley 2010; Standley and Korf 2011): 32x32 grids were generated with random obstacles (each cell is an obstacle with 20% probability). Each agent was placed in a random unique location with a random unique destination.

Just like (Standley 2010; Standley and Korf 2011), we use what we call the *performance curve* of an algorithm on a set of instances to convey how well an algorithm performs on a set of instances. We run each algorithm on each instance in the set and record the time taken to solve each instance.

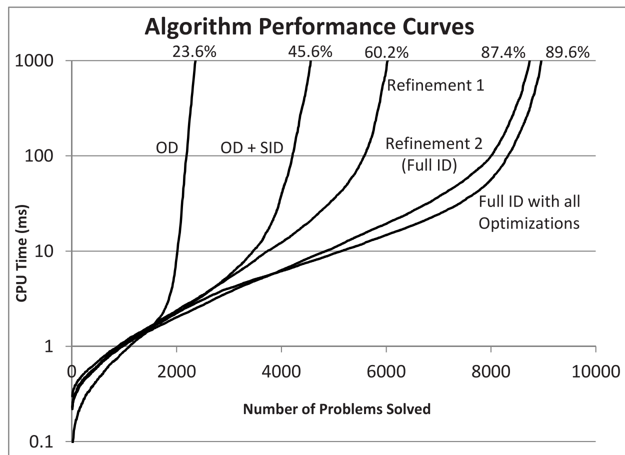

Figure 2: Performance curves for various refinements of independence detection.

For each algorithm, we sort the instances based on the time taken by that algorithm, and plot the results. The index of each instance in the sorted sequence is plotted along the x-axis, and the time taken to solve that instance is plotted along the y-axis. Note that the ith instance is a different problem instance for each algorithm's performance curve. Therefore, a performance curve shows the total number of instances an algorithm would solve if limited to a certain amount of time per instance.

The performance curves in Figure 2 were generated on 10,000 problems with a random number of agents between 2 and 60. Operator decomposition on its own was able to optimally solve only 23.6% of problems in one second. Adding simple independence detection nearly doubled the number of problems solved, bringing it to 45.6%. Refinement 1, which adds replanning with an illegal move table, brings the total up to 60.2% (an increase of about 32%). Even more drastic improvements were seen when the collision avoidance table was introduced (i.e., refinement 2), increasing the number of problems solved to 87.4% (an *additional* improvement of about 45%). Since we chose problems having a wide range of difficulty levels, solving the larger problems is substantially harder, highlighting the importance of refinement 2.

Adding all four optimizations described in the last section allow us to solve a modest 2.5% more problems. This might not sound like much, but it would require 2.5 times longer to solve that many extra problems without the additional optimizations. Further experimentation revealed that choosing which travel group to attempt to plan first was the most effective optimization, allowing the algorithm to solve 70 more problems than it would with just the other optimizations. Prioritizing avoiding collisions with large groups allowed 48 more problems to be solved than with just the other optimizations. Using better initial paths helped to solve an extra 27 problems, but it is more effective when ID is used to find approximate solutions. Finally, avoiding futile replanning was not effective at helping to solve more problems, but did reduce the time needed to solve problems by an average

of about 15%.

Future Work

Independence detection has proven to be a powerful technique for breaking large MAPF problems into more manageable pieces, but the algorithm requires that there are travel groups whose paths are entirely independent. A better algorithm might be able to take advantage of independence that exists before and after a conflict.

Conclusion

In this paper, we described Independence Detection in greater detail than in our previous work. First, we gave a simple algorithm for detecting independence, SID. We refined the algorithm with the addition of illegal move tables, which allow the algorithm to recover from unfortunate initial path choices. We further refined the algorithm using conflict avoidance tables, which allow the algorithm to make better path choices in the first place, thereby often avoiding future conflicts. Finally, we proposed four simple optimizations that further increase the algorithm's performance, including three that have never before been published. We showed that our second refinement, using collision avoidance tables, was the most effective, and we quantified the effectiveness of our other refinements and optimizations.

References

Kurt M. Dresner and Peter Stone. Multiagent traffic management: A reservation-based intersection control mechanism. In *AAMAS*, pages 530–537, 2004.

Alborz Geramifard, Pirooz Chubak, and Vadim Bulitko. Biased cost pathfinding. In John E. Laird and Jonathan Schaeffer, editors, *AIIDE*, pages 112–114. The AAAI Press, 2006.

Peter Hart, Nils Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, February 1968.

M. Renee Jansen and Nathan R. Sturtevant. Direction maps for cooperative pathfinding. In *AIIDE poster*, 2008.

M. Renee Jansen and Nathan R. Sturtevant. A new approach to cooperative pathfinding. In *AAMAS 2008 Volume 3*, pages 1401–1404, 2008.

Malcolm R. K. Ryan. Exploiting subgraph structure in multi-robot path planning. *JAIR*, 31(1):497–542, 2008.

Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. The increasing cost tree search for optimal multi-agent pathfinding. In Walsh (2011), pages 662–667.

David Silver. Cooperative pathfinding. In *AIIDE*, pages 117–122, 2005.

Trevor Scott Standley and Richard E. Korf. Complete algorithms for cooperative pathfinding problems. In Walsh (2011), pages 668–673.

Trevor Standley. Finding optimal solutions to cooperative pathfinding problems. In *AAAI*, pages 173–178, 2010.

Pavel Surynek. An application of pebble motion on graphs to abstract multi-robot path planning. In *ICTAI*, pages 151–158, 2009.

P. Svestka and M. H. Overmars. Coordinated path planning for multiple robots. Technical Report UU-CS-1996-43, Department of Information and Computing Sciences, Utrecht University, 1996.

Toby Walsh, editor. *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*. IJCAI/AAAI, 2011.

Ko-Hsin Cindy Wang and Adi Botea. Fast and memory-efficient multi-agent pathfinding. In *ICAPS*, pages 380–387, 2008.

K.-H. C. Wang and A. Botea. Tractable Multi-Agent Path Planning on Grid Maps. In *Proceedings of the International Joint Conference on Artificial Intelligence IJCAI-09*, pages 1870–1875, Pasadena, CA, USA, 2009.