Complete Algorithms for Cooperative Pathfinding Problems

Trevor Standley[1] and **Richard Korf**
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095
{tstand, korf}@cs.ucla.edu

Abstract

Problems that require multiple agents to follow non-interfering paths from their current states to their respective goal states are called cooperative pathfinding problems. We present the first *complete* algorithm for finding these paths that is sufficiently fast for real-time applications. Furthermore, our algorithm offers a trade-off between running time and solution quality. We then refine our algorithm into an anytime algorithm that first quickly finds a solution, and then uses any remaining time to incrementally improve that solution until it is optimal or the algorithm is terminated. We compare our algorithms to those in the literature and show that in addition to completeness, our algorithms offer improved solution quality as well as competitive running time.

1 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]. When the problem contains multiple agents, however, care must be taken to avoid computing a solution that leads any subset of agents to conflict, for example, one that requires two agents to occupy the same space at the same time. Cooperative 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, but because all known fast algorithms are incomplete, players must be notified when the algorithm fails and manually fix the issue.

2 Problem Formulation

There are many distinct types of cooperative pathfinding problems, but the algorithm we present is broadly applicable. 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 railroad 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, simplicity, and comparison to existing algorithms, the testbed for our algorithm will be an eight-connected grid world like the one in Figure 1.

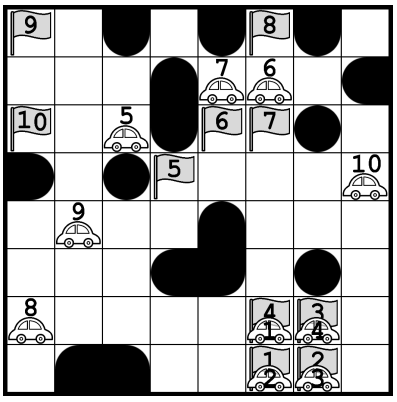

Figure 1: A small grid world instance. Cars represent initial agent positions. Flags represent destinations. Obstacles are in black.

We use the problem formulation from our previous work [Standley, 2010], 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 it 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 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 diagonal 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.

[1] current affiliation: Google Inc.

3 Related Work

Cooperative pathfinding problems have usually been solved in one of two ways in the literature. In global search approaches, the entire set of agents is treated as a single entity and paths are found for all agents simultaneously. Alternatively, in decoupled approaches paths are found for each agent one at a time, and information 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 incomplete.

One example of a decoupled approach is HCA* [Silver, 2005]. HCA* employs a reservation table for timestep-location pairs. The algorithm chooses a fixed ordering of agents, and plans a path for each agent in turn that avoids conflicts with previously computed paths by checking against the reservation table. Unfortunately, in over half of our benchmark instances, some agents never reach their destinations because the paths found for previous agents in the fixed order can make finding paths for subsequent agents impossible. Using a windowed search, Silver's WHCA* mitigates this problem at the cost of solution quality and running time.

Other decoupled attempts establish a direction for every grid position and encourage or require each agent to move in that direction at every step [Wang and Botea, 2008; Jansen and Sturtevant, 2008a; 2008b]. These methods reduce the chance of computing conflicting paths by creating the analog of traffic laws for the agents to follow. While these methods are effective at reducing the number of incompatible paths, paths are still computed greedily one agent at a time. Therefore, this strategy also leads to incomplete algorithms. Moreover, the speed advantage of these algorithms over HCA* comes at the cost of solution quality because many valid paths are pruned or penalized if they don't obey the traffic laws.

Unfortunately, it is PSPACE-hard to find *any* solution to many cooperative pathfinding problems [Hopcroft *et al.*, 1984; Hearn and Demaine, 2005], so research in this area tends to focus on incomplete algorithms like the ones above.

However, there are a few complete algorithms in the literature [Surynek, 2009; Ryan, 2008; Svestka and Overmars, 1996]. These algorithms abstract the search space so that solutions can be found with a global search algorithm for the abstraction. These algorithms are complete, but usually have high solution costs. Furthermore, the time and memory requirements of these algorithms limit their applicability.

In our prior work, we presented another complete algorithm that is also optimal [Standley, 2010]. We proposed two techniques. Independence Detection (ID) decomposes instances into independent subproblems when doing so would not compromise optimality, and only uses a more costly but admissible search algorithm, Operator Decomposition (OD), when no such independence can be found. Although this combination is both complete and optimal, it has a couple of drawbacks. First, since it only aims for optimal solutions, its running time is prohibitively expensive for many real-time applications. Second, prematurely terminated runs of the algorithm only return conflicting paths. The present work builds on our OD+ID algorithm, and overcomes these drawbacks.

4 OD+ID

In our prior work, we started by defining a state space representation for cooperative pathfinding problems that is optimized for A* called OD. The general idea is that each search node contains a position for every agent, and subsequent nodes are generated by changing the position of a single agent. We showed that OD drastically reduces the number of nodes generated when compared to A*, because pruning can take place even before all agents have been assigned a move. A* with the OD state space representation is a global search algorithm, which we refer to simply as OD. We couple OD with our independence detection algorithm, ID, which partitions the agents into several smaller groups, and finds optimal paths for each group with OD. The algorithm ensures that the paths found for each group are optimal and that the paths for agents in separate groups do not conflict. Taken as a whole, the groups' paths constitute an optimal solution to the cooperative pathfinding problem.

4.1 Independence Detection

We present the details of independence detection because they are important for developing the algorithms in this paper. As a first step toward an independence detection algorithm, we presented a simple way to discover independent partitions of agents [Standley, 2010]: Start by independently finding a path for each agent and look for collisions within those paths. Group the first two agents found to be in conflict and find a new path for the group using a global search A* algorithm such as OD. Repeat this process of merging two conflicting groups into a larger group and recomputing a path for the new group until there are no more conflicts.

Because the running time of the algorithm is dominated by the time to plan paths for the largest group, performance can be improved drastically by avoiding unnecessary merges. The ID algorithm can sometimes avoid merging two conflicting groups by finding a new optimal path for one of the groups. In order to ensure optimality, the new paths for a group must have the same total cost as the initial paths. We refer to this as the optimality constraint of ID. To satisfy this requirement, OD is given a cost limit, and does not consider paths of greater cost. We also want the new path found to not conflict again. Thus, OD is also given a table called the illegal move table. When OD expands a node, it consults the illegal move table at the appropriate timestep to determine which of the node's children would result in a path with another conflict, so that it may discard these children.

If, for example, two groups G_1 and G_2 conflict, and ID decides to replan the path of G_1, ID would populate the illegal move table with the transitions that agents in G_2 make when following G_2's current paths at every timestep. When OD expands a node while replanning the path for G_1, it checks the table at that node's timestep, and only considers moves that do not conflict with the moves of G_2's agents during that

timestep. If a path is found, then the conflict between these two groups has been resolved.

During these replans, it is important that the algorithm find a path that creates the fewest conflicts with other agents' paths, so that future merges and replans are less likely to occur. Toward this goal, ID uses a table similar to the illegal move table, which is called the conflict avoidance table. This table stores the moves of all other agents for every timestep. In A* 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, $v(n)$, which 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 algorithm then breaks ties first by the lowest $v(n)$, then by the lowest $h(n)$. 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].

ID starts by assigning each agent to its own group. It then finds an initial path for each group independently. Next, ID looks for conflicts within its current set of paths. Upon detecting a conflict, ID attempts to find an alternative path for one of the conflicting groups, ensuring that the new path does not conflict with the other group. If this fails, it repeats this process with the other conflicting group. If both attempts to find alternative paths fail, ID merges the conflicting groups and cooperatively plans a path for the new 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 1 Independence Detection

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

5 Approximate Algorithms

Many applications, especially those in digital entertainment, demand real-time performance, and it is often acceptable to sacrifice solution quality to satisfy this demand. There have been many algorithms suggested for this purpose, and some are described in Section 3. Unfortunately, we know of no existing algorithm that is both efficient and complete, so human intervention is often required in real-time applications. However, OD+ID can be easily extended to create an algorithm that is not only complete, but also offers a tradeoff between solution quality and running time.

6 Complete Approximate Algorithms

Two constraints embedded in OD+ID ensure that its solutions are optimal. The first is the previously mentioned optimality constraint of ID, which is implemented by giving OD a cost limit for finding alternative paths. The second constraint is that OD always expands nodes in non-decreasing order of $f(n)$ cost.

Algorithm 2 Resolve Conflicts

```
1: fill conflict avoidance table with every path
2: repeat
3:     simulate execution of all paths until a conflict between two
       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
```

Dropping both constraints produces a fast and complete algorithm. Without the optimality constraint of ID, OD will search for alternative paths of any length rather than just optimal paths, making it much more likely for an alternative path to be found so that the conflicting groups do not have to be merged.

As long as nodes on the open list are expanded in the order of lowest $f(n)$ first, and $h(n)$ is always a lower bound on the node's distance to the goal, A* will return an optimal solution [Hart *et al.*, 1968]. In dropping the second constraint, we propose a modification to A* and thus OD. Instead of expanding nodes with lowest $f(n)$ first, we expand nodes with lowest $v(n)$ first, and break ties in favor of lowest $f(n)$. Now OD will be guaranteed to return paths that minimally conflict with the current paths of other groups instead of paths with the lowest cost. Nevertheless, among the many paths with the fewest conflicts, OD will find one with the lowest cost.

In order to produce even higher-quality solutions, we observe that we can dynamically drop these constraints, which leads to an algorithm that offers the time-quality trade-off mentioned above. We introduce a parameter called the *maximum group size* (MGS). Consider a situation during the execution of ID in which we encounter a conflict between two groups G_1 and G_2 each containing s_1 and s_2 agents respectively. ID will try to find an alternative path for G_1 that does not conflict with G_2. In order to avoid creating groups larger than the MGS, we drop the two constraints whenever $s_1 + s_2 > MGS$. This leads to a spectrum of approximations. If the MGS is one, this algorithm will always drop the constraints (because merging would always result in a group of size two or larger). If the MGS is greater than one, the algorithm will take longer because it must find optimal paths for larger groups of agents, but this allows it to produce lower-cost paths. We drop the constraints when finding initial single-agent paths if and only if the MGS is one.

Note that it is still possible for the algorithm to produce a group containing more agents than the maximum group size. This will happen when no appropriate alternative paths of any length exist for either of the two groups in a given conflict. In this case, independence detection will merge the groups as usual to maintain completeness. Fortunately, this does not happen very often as there are a combinatorial number of possible alternative paths, and only one is needed to resolve the conflict.

We call this algorithm the maximum group size algorithm (MGS). We denote an algorithm with a maximum group size of x as MGSx. Note that if x is sufficiently large, the algorithm behaves exactly like OD+ID.

7 Optimal Anytime Algorithms

The ideal algorithm for real-time applications is an optimal anytime algorithm. Optimal anytime algorithms eventually find optimal solutions, but these algorithms can also be stopped before completion because they maintain an ever-improving solution that they can return at any time. Unfortunately, neither OD+ID nor the MGS algorithm presented above can solve the problem in this way. Once given a maximum group size, the MGS algorithm spends an unspecified amount of time to return a result, and there are no meaningful intermediate results.

A simple way that the MGS algorithm can be adapted to be an anytime algorithm is inspired by iterative deepening [Korf, 1985]. First, we run MGS1. After it has found a solution, we run MGS2 if time remains. We continue to incrementally increase the maximum group size until time has run out. At that time, we return the highest-quality solution found so far. If we make the assumption that the running time of OD is roughly exponential in the number of agents in its group and does not depend on the number of agents in other groups, and we also assume that MGSx never calls OD with groups larger than x, then the overhead for the last unfinished iteration and unused previous iterations does not depend on the number of agents in the problem.

We call this the simple anytime algorithm. This algorithm is useful, but we can mitigate the overhead of unused and unfinished iterations.

In order to make use of the work done on previous iterations, we would like to reuse the old groupings found, as well as the paths for as many of those groups as possible. We would like to maintain the invariant that at the end of an iteration, all groups with sizes less than the MGS have optimal paths so that the algorithm eventually arrives at an optimal solution. To accomplish this, we must have a way of knowing which groups already have optimal paths. Fortunately, if we label each group with a lower bound on the cost of an optimal path for that group, we can tell which groups have optimal paths. The initial lower bound for each singleton group G can simply be $h(g)$. During the execution of the algorithm, we can sometimes update these lower bounds. Groups with costs equal to their lower bounds are known to have optimal paths for that group.

When groups are merged, the algorithm must cooperatively plan a path for the new group. If the size of the new group is less than the current maximum group size, then we know that the cost of the new path will be optimal and the lower bound can be updated. Otherwise, the lower bound for the group will be the sum of the lower bounds for the two merged groups.

We still iteratively increase the maximum group size starting from one, but now we keep the groupings and paths of the previous iteration along with the lower bound for each group. Every time we increase the maximum group size, we can easily tell which groups might not have optimal paths by simply comparing the lower bound for each group with the cost of the current set of paths for that group. We can then start finding new optimal paths for each group that might not currently have optimal paths, but that is smaller than the maximum group size. After new optimal paths are found for a group, we can update the lower bound for that group. We can also resolve any conflicts that the new set of optimal paths creates in the same way as the ID algorithm. If an optimal path has been found for a group as a result of increasing the maximum group size, then we must maintain this optimality during future replans of this group. When we are done resolving the conflicts introduced by enforcing optimality on a group with size less than the MGS, we update the highest-quality solution found so far if our new solution is indeed of lower cost.

This method allows information obtained in previous iterations to be used in subsequent iterations, which reduces the time of subsiquent iterations. Furthermore, the best solution so far is updated many times within an iteration, thereby mitigating the overhead of the last unfinished iteration. A lower bound on the cost of the optimal solution for the entire problem can be calculated by adding the lower bounds for all groups. Once the total lower bound is equal to the cost of the best solution found, the algorithm can terminate with an optimal solution. If the algorithm is terminated early, it can return not only the best solution encountered, but also a lower bound, which indicates how much the solution could improve if the algorithm continued.

We refer to this algorithm as the optimal anytime (OA) algorithm. We use OAt to refer to the algorithm when it is terminated at time t.

Algorithm 3 Optimal Anytime Algorithm

```
1: call MGS1 to get initial paths and groupings
2: for all G in groups do LB(G) ← h(G)
3: set best solution to the MGS1 solution
4: MGS ← 2
5: repeat
6:    for all G in groups do
7:       if LB(G) < COST(G) and SIZE(G) < MGS then
8:          find an optimal path for G with OD
9:          LB(G) ← COST(G)
10:         call a modified Resolve Conflicts
11:         update best solution
12:      end if
13:   end for
14:   MGS ← MGS + 1
15: until sum(LB(G)) = COST(best solution)
```

The pseudocode for our optimal anytime algorithm is presented as Algorithm 3. For the modified Resolve Conflicts,

OD will prioritize nodes with lowest $v(n)$ and ignore cost limits if the combined sizes of the conflicting groups is greater than the MGS, and line 9 was not executed for the group. When merging two groups, the modified Resolve Conflicts will also set the lower bound of the new group to be the sum of the lower bounds of the old groups.

8 Other Optimizations

Another helpful modification to both OD+ID and our approximation algorithms 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 in turn while avoiding collisions with the paths of all previously planned agents whenever possible, using the collision avoidance table. We then repeat this process while also avoiding collisions with paths found on the previous iteration for agents whose paths have not yet been planned in the current iteration. This ensures that the final path we find for every agent avoids a path found on either iteration 1 or 2 for every other agent. We use this modification in our experiments for OD+ID and all of our algorithms.

9 Experiments

All of our experiments were run on an Intel Core i7 @ 2.6GHz using benchmarks like those proposed in [Silver, 2005] and used in [Standley, 2010]: 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], 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. 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.

Figure 2 shows the performance curves of several approximate algorithms on instances that took less than a second to solve. We randomly generated two sets of 10,000 benchmark instances each. Every instance in the first set had 150 agents. Every instance in the second set had 250 agents. OD+ID, MGS1, MGS2, and HCA* were run on these instances. OD+ID did not solve a single instance in either set and is not shown. HCA* solved less than half of the instances in the first set, and only 32 instances in the second set. However, MGS1, our coarsest algorithm, could solve nearly all of the instances in under a second each. Even MGS2 could solve more instances in the first set than HCA* in the time limit even though it generates much better solutions. Since

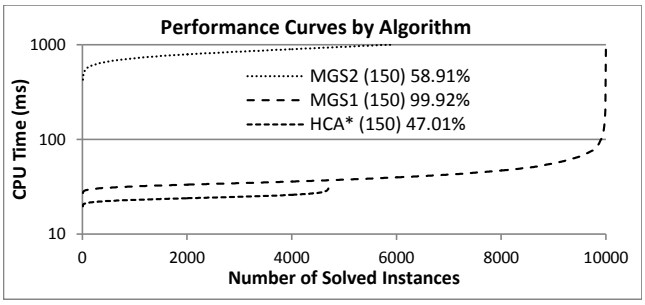

Figure 2: Performance curve for the three approximate algorithms on two sets of 10,000 very large instances each.

MGS1 finds at least two paths for each agent, MGS1 took an average of 74.8% longer to solve the instances that were solved by both MGS1 and HCA*. Still, MGS1 spent an average of only 0.28 ms per agent when solving these instances.

Furthermore, the MGS algorithm scales nicely to larger problems. In a similar experiment with 250 agents, MGS1 solved 94.41% if instances in under a second each, while HCA* solved only 0.32%.

In order to determine the quality of the solutions produced by our approximate algorithms, 2,500 instances with 60 agents each were randomly generated. Three optimal algorithms, OD+ID, the simple anytime algorithm (SA), and our optimal anytime algorithm (OA), were each given 30 seconds to optimally solve each instance. When any algorithm was successful, the optimal solution cost was recorded. To validate our implementations, we verified that whenever two or more algorithms optimally solved the same instance, their solutions had the same cost. Although there were only 60 agents in each instance, only 1,817 of these 2,500 instances were solved by any optimal algorithm. These 1,817 solved instances are used to compare the solution quality of our approximate algorithms. We refer to these solved instances as our reference instances.

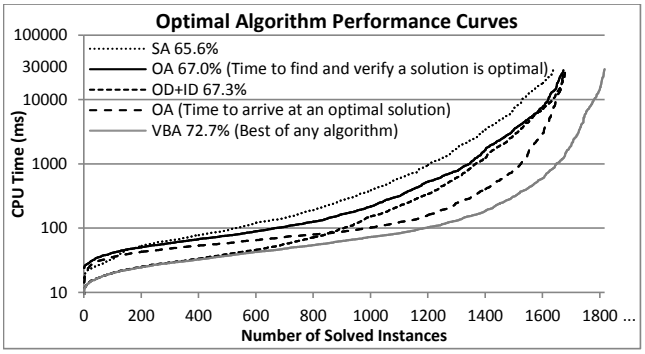

Figure 3: Performance curve and percent solved for three optimal algorithms on 2,500 instances with 60 agents each, plotted on a log scale.

Figure 3 shows the performance curves for our optimal algorithms on these 2,500 instances. We plot the performance of OA for finding an optimal solution and for finding *and* verifying an optimal solution separately. The time to find an optimal solution is the time taken to first encounter that solu-

tion, which was usually before the algorithm knew that it was optimal, because its optimality could not be verified until the lower bound matched its cost. The performance of OA for finding and verifying an optimal solution is much better than that of the simple anytime algorithm and nearly as good as OD+ID. However, the data show that our algorithm usually finds an optimal solution even before OD+ID. We have also plotted the minimum time taken of any algorithm on each instance, called the Virtual Best Algorithm (VBA). Since VBA solves more instances within a third of the time limit than any of the algorithms by itself, an algorithm that ran all three algorithms in parallel would improve the state of the art for finding optimal solutions.

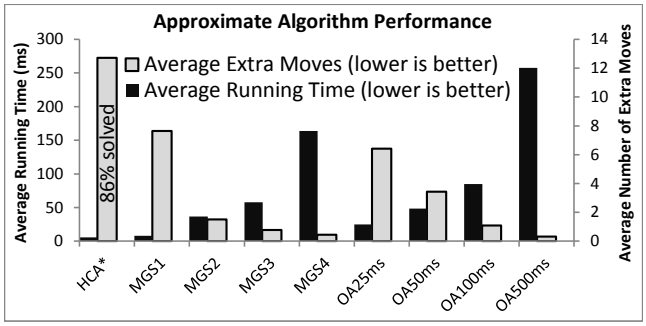

Figure 4: The solution quality and time for our MGS algorithms, our anytime algorithm, and HCA* on our reference instances.

Four levels of the MGS algorithm and HCA* were run on our 1,817 reference instances. We also allowed OA to run for 25ms, 50ms, 100ms, and 500ms on the same instances. The average solution quality and running times are plotted in Figure 4. Although HCA* ran faster than our MGS algorithms, it did not solve all of the reference instances. Furthermore, the solution quality of HCA* was worse than the solution quality of every MGS algorithm. We see that the paths generated by the anytime algorithm quickly approach the optimal paths, and that the algorithm can generate high-quality paths in an amount of time suitable for applications in digital entertainment. However, the MGS algorithms are a better choice if one is only concerned with the solution quality obtained in a specific *average* amount of time per instance, because they have less overhead.

10 Limitations

None of our algorithms scale well in highly constrained situations such as sliding tile puzzles. In prelminary experiments with 4-connected grids, MGS1 solved 100% of problems with 8 agents on 3x3 grids, but only 15% of problems with 15 agents on 4x4 grids in under 5 minutes each. In highly constraind situations, even MGS1 can detect little independence, and OD must find paths for large groups of agents.

11 Conclusion

We described the general problem of cooperative pathfinding, and showed that our prior optimal approach suggests complete approximation algorithms with time-quality tradeoffs. We also showed that these algorithms perform better than existing approximation algorithms on a set of randomly generated benchmark instances, and that they are capable of quickly and reliably finding solutions to instances with as many as 250 agents on a 32x32 grid. We then refined our complete approximation algorithms into an optimal anytime algorithm using a low-overhead iterative deepening approach. We showed that the anytime algorithm is not only competitive with OD+ID for finding optimal solutions, but also can be terminated early to result in high-quality approximate solutions.

Acknowledgments

This research has been supported by NSF grant No. IIS-0713178. We also thank Dawn Chen for her many edits, suggestions, and discussions.

References

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

[Hart *et al.*, 1968] 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.

[Hearn and Demaine, 2005] Robert A. Hearn and Erik D. Demaine. PSPACE-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. *TCS*, 343(1-2):72–96, 2005.

[Hopcroft *et al.*, 1984] J.E. Hopcroft, J.T. Schwartz, and M. Sharir. On the Complexity of Motion Planning for Multiple Independent Objects; PSPACE-Hardness of the Warehouseman's Problem. *IJRR*, 3(4):76–88, 1984.

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

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

[Korf, 1985] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

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

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

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

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

[Svestka and Overmars, 1996] 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.

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