Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10)

Finding Optimal Solutions to Cooperative Pathfinding Problems

Trevor Standley
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095
tstand@cs.ucla.edu

Abstract

In cooperative pathfinding problems, non-interfering paths that bring each agent from its current state to its goal state must be planned for multiple agents. We present the first practical, admissible, and complete algorithm for solving problems of this kind. First, we propose a technique called operator decomposition, which can be used to reduce the branching factors of many search algorithms, including algorithms for cooperative pathfinding. We then show how a type of independence common in instances of cooperative pathfinding problems can be exploited. Next, we take the idea of exploiting independent subproblems further by adding improvements that allow the algorithm to recognize many more cases of such independence. Finally, we show empirically that these techniques drastically improve the performance of the standard admissible algorithm for the cooperative pathfinding problem, and that their combination results in a complete algorithm capable of optimally solving relatively large problems in milliseconds.

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 is usually effectively solved using the A* algorithm (Hart, Nilsson, and Raphael 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, or more abstractly, a solution in which agents access more of a resource than is currently available. While cooperative pathfinding is becoming increasingly important in modern video games, it has many applications outside of entertainment such as robotics, aviation, and vehicle routing (Wang and Botea 2008; Svestka and Overmars 1996).

Unfortunately, it has been shown to be PSPACE-hard to compute any set of paths that bring each agent to its goal state without conflict in several cooperative pathfinding domains, even those without obstacles (Hopcroft, Schwartz, and Sharir 1984; Hearn and Demaine 2005), and the standard admissible algorithm for cooperative pathfinding problems has a branching factor that is exponential in the number of agents (Silver 2005). It is for these reasons that existing research on these problems focus on algorithms for finding good solutions as often as possible rather than developing optimal and complete algorithms. Our algorithm is complete, however, and can find optimal solutions to nontrivial problems, often in milliseconds.

This paper is divided into the following sections. First, we describe related work in this area. Then, we describe one particular formulation of the cooperative pathfinding problem. Next, we elaborate on the standard admissible algorithm and our two improvements. Finally, we present our experimental results and conclusion.

Related Work

Silver (2005) describes an algorithm which he claims is widely used in the video games industry, Local Repair A* (LRA*). In LRA*, a path is computed for each agent independently, and conflicts are not discovered until execution. If the algorithm discovers that following a solution one step further results in a conflict, the algorithm re-plans the path of one of the conflicting agents, disallowing only the move that would immediately result in a conflict. Unfortunately LRA* often results in cycles and deadlock. Silver's solution to this problem, Hierarchical Cooperative A* (HCA*), focuses on reducing the number of future re-plans by creating a reservation table for future timesteps. The algorithm chooses an ordering of agents, and plans a path for each agent that avoids conflicts with previously computed paths by checking against the reservation table. While the solutions found by HCA* are significantly shorter than those computed by LRA*, the greedy nature of this algorithm still results in suboptimal solutions. Furthermore, in 4.3% instances, some agents never reach their destinations. Finally, this technique offers no option to spend more computation time to increase either the number of solutions found or solution quality.

Other 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

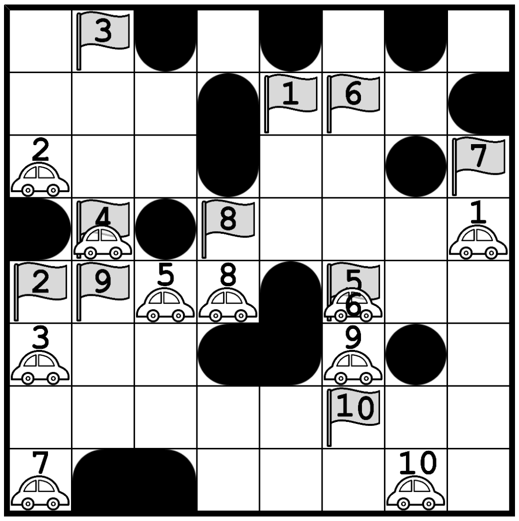

Figure 1: A small random grid world instance. Cars represent initial agent positions. Flags represent destinations. Obstacles are in black.

for the agents to follow. While these methods are effective at reducing the number of incompatible paths, they limit the types of paths that agents can follow, and thus virtually guarantee that the solutions found will be suboptimal. Moreover, as paths are still computed one agent at a time, without regard to the consequences that a choice of path can have on the paths available to future agents, this strategy also suffers from the problem of deadlock, and therefore some agents may never reach their destinations.

Finally, existing centralized approaches such those presented in (Ryan 2008) and (Svestka and Overmars 1996) are complete, but do not aim for low solution cost. Furthermore, the time and memory requirements of these algorithms limit their applicability to problems containing only 100 positions and less than ten agents.

Problem Formulation

There are many distinct types of cooperative pathfinding problems, and each can be solved using a variant of our algorithm. One example is planning the motions of robotic arms, each of which must accomplish a separate goal without moving into one another or monopolizing a resource. 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) are also cooperative pathfinding problems. 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.

In this formulation, each agent occupies a single cell of the grid world, and has a unique destination. During a single timestep, each agent either waits, or moves to any of its free adjacent cells. A cell is free if it does not contain an obstacle. Diagonal moves are allowed even when the two tiles adjacent to both the starting tile and the target are obstacles. The cost of a single agent's path is the total number of timesteps that agent spends away from its goal, and the cost of the entire solution is the sum of all path costs. Two agents must not occupy the same tile at the same time.

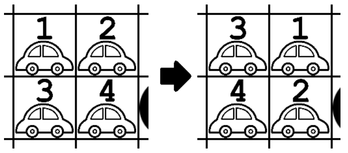

Figure 2: Our formulation of the cooperative pathfinding problem allows rotational moves, which result in each agent occupying a previously occupied cell.

Moreover, 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. In addition, we allow rotational transitions like the ones depicted in Figure 2.

The Standard Admissible Algorithm

The standard admissible algorithm for this problem is A* with the following problem representation: A state is an n-tuple of grid locations, one for each of n agents (Wang and Botea 2008). The standard algorithm considers the moves of all agents simultaneously at a timestep, so each state potentially has 9^n legal operators. Each of these legal operators is a solution to the constraint satisfaction problem in which each agent must be assigned a move from $\{N, NE, E, SE, S, SW, W, NW, \text{and } wait\}$, and there are constraints between sets of legal moves. For example, these legal moves must not lead two agents to pass through each other. In order to generate these operators, a backtracking search is employed to efficiently find all solutions to the CSP at every node expansion. In practice, these CSPs are not difficult, and most of the 9^n combinations of moves are legal. The cost of each operator is the number of agents not stopped on their goals. An admissible heuristic, like the one presented in a later section, is always coupled with this algorithm.

While this method can be used to solve some trivial problem instances, each node expansion can add approximately 59k nodes (approximately 1.5MB in our implementation) to the open list when run on an instance with only 5 agents.

In many of the successors of a given node, the majority of agents have moved farther from their goals, resulting in nodes that have heuristic values as much as n higher than the heuristic values of their parents. There are typically an exponential number of such unpromising nodes, and A* generates them all, placing each onto the open list.

Operator Decomposition

In the standard algorithm, every operator advances a state by one timestep and changes the position of every agent. We propose a new representation of the state space in which each timestep is divided so that agents are considered one at a time and a state requires n operators to advance to the next timestep. In this representation, a state not only contains a

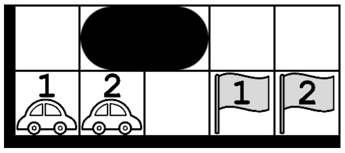

Figure 3: An example in which naïve OD would fail.

position for every agent, but also contains up to one move assignment for every agent.

An operator in this representation consists of assigning a move to the next unassigned agent in a fixed order, leaving the moves of the remaining agents to descendant nodes within the same search. The new state representation has a branching factor of 9 rather than 9^n, but because each operator only advances a single agent rather than every agent, the depth of a goal node increases by a factor of n. We call this technique A* with operator decomposition (OD) because the standard operators are decomposed into a series of operators.

Under this representation, there are two conceptually different types of states. We refer to *standard* states as those in which no agent has been assigned a move, and *intermediate* states as those in which at least one agent has been assigned a move. Assigning a move to the last unassigned agent in an intermediate state results in a standard state which has advanced to the next timestep. Note that while these states are conceptually different, A* search treats them equivalently, so intermediate nodes derived from different standard nodes can be expanded in any order.

For a given state, we refer to a *post-move* position as the position that an agent will have after it follows the assigned move, and a *pre-move* position as the position an agent has before it follows an assigned move. We may refer to the positions of agents without assigned moves as either pre-move or post-move positions.

Note that the $h()$ and $g()$ values for intermediate nodes can be different from those of their standard ancestor nodes because some of the post-move positions may be different. Therefore, A* can sometimes avoid expanding intermediate nodes because of cost. The standard algorithm, however, cannot because it uses backtracking search to solve the CSPs which has no concept of cost. It can only prune configurations that are illegal.

Unfortunately, OD will not guarantee optimal solutions if the implementation is naïve. Consider the situation depicted in Figure 3. If the algorithm considers agent 1 first, then it might conclude that it has only two legal moves $(N, wait)$. However, the optimal solution requires that agents 1 and 2 both make the move E first. The problem with this naïve OD is that it is too strict. It restricts the types of moves an agent can make on the basis of every other agent's post-move position. The algorithm should require that a new move assigned to an agent be consistent with the existing assignments of moves to agents in a node, but the algorithm should allow agents to move into spaces occupied by agents who are yet to be assigned moves within a node.

Consider again the example of Figure 3. When OD is used correctly, A* starts by placing three nodes onto the open list because agent 1 can be assigned N, E, or $wait$. Note that

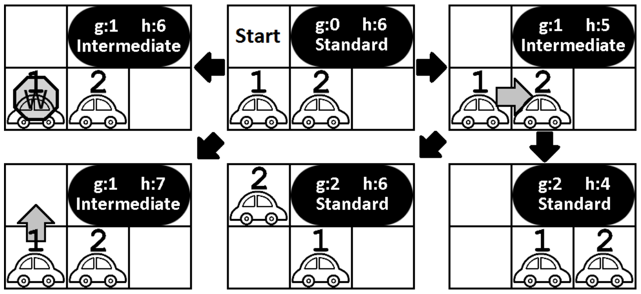

Figure 4: The search tree of A* with OD for one. Pre-move positions are shown. Move assignments are marked with arrows.

both agents have the same post-move position in the node in which agent 1 is assigned E. This is desirable because agent 2 has not yet been assigned a move. Then, when that node is chosen for expansion, A* considers assigning agent 2 the move E. The algorithm then checks to see that the move assignment $[1 : E, 2 : E]$ is legal (because agent 1 has *already* been assigned a move in this node) and places the resulting child node onto the open list. The operators $wait$ and W are also checked, but discarded (because the move assignment $[1 : E, 2 : wait]$ results in two agents overlapping, and the move assignment $[1 : E, 2 : W]$ results in two agents passing through each other.

We see in Figure 4 that OD generates five nodes during the first timestep in this example. On the other hand, the standard algorithm generates eight successor nodes from the start state because $[wait, wait]$, $[wait, E]$, $[wait, NW]$, $[N, wait]$, $[N, E]$, $[N, W]$, $[E, E]$, and $[E, NW]$ are all legal standard operators from the start state. For both algorithms, a node in which both agents have made the move E is expanded next.

When coupled with a perfect heuristic and a perfect tie-breaking strategy, A* search generates $b \times d$ nodes where b is the branching factor, and d is the depth of the search. Since the standard algorithm has a branching factor of approximately 9^n and a depth of t (the number of timesteps in the optimal solution), A* search on the standard state space generates approximately $(9^n)t$ nodes when coupled with a perfect heuristic. A* with OD, however, will generate no more than $9nt$ nodes in the same case because its branching factor is reduced to 9, and its depth only increases to $n \times t$. This is an exponential savings with a perfect heuristic.

OD is capable of generating every legal successor of a standard node for any chosen order of agents by assigning the appropriate move to each agent in turn. Therefore, A* with OD is admissible and complete for all choices of agent order due to the admissibility and completeness of the standard A* algorithm.

A* makes use of a closed list to detect duplicate nodes in order to avoid redundant node expansions. Like many cooperative pathfinding problems, the state space of the eight-connected grid world variant grows only polynomially with depth (although it is exponential in n). Therefore, because the number of nodes in the search tree grows exponentially with depth, the duplicate detection is indispensable.

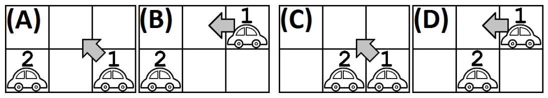

Figure 5: Two pairs of similar intermediate states.

The intermediate descendants of a standard node form a tree rooted at the standard node because a move assignment to an agent cannot be changed until all agents have been assigned moves. Because duplicate intermediate nodes would have duplicate standard ancestors nodes, A* will never encounter duplicate intermediate nodes. Therefore, it is reasonable to put only standard nodes onto the closed list. Empirically, this modification reduced the size of the closed list by nearly 93% while improving running time in preliminary experiments.

Alternatively, we can sometimes safely treat nodes that are not equal, but have the same post-move positions, as duplicates. Observe in Figure 5 that both (A) and (B) have the same post-move positions. Also, the set of possible moves for agent 2 that do not conflict with agent 1's move is the same in both (A) and (B). These nodes will therefore have the same set of standard descendants, and the algorithm can safely treat them as duplicates. However, agent 2 cannot legally move NE in state (C), but it can in state (D), so treating these states as duplicates would be incorrect even though they have the same post-move positions. This example also shows the importance of storing move assignments in addition to positions in a state. Enabling the algorithm to treat nodes like (A) and (B) as duplicates is well worth the overhead and is used in our experiments.

The Heuristic

A* makes use of an estimate of the true distance to the goal for every state. This allows A* to intelligently search only a relevant subset of the search space. If this heuristic never overestimates the cost from a state to a goal state, it is called admissible, and A* will return optimal solutions.

One admissible heuristic for a state in any cooperative pathfinding problem is the sum of the costs of the optimal single-agent paths for all agents. Conceptually, these distances could be obtained by running a single-agent search for every agent whenever A* generates a node. However, we can simply run a breadth-first search from an agent's goal position to every free grid position to precompute a table of the single agent distances for every agent at every position. During the A* search with OD, the change in heuristic value from a state to a child state is due to the move assignment for a single agent. The change in heuristic value is the difference between the table entry for the agent's post-move position and the table entry for the agent's pre-move position. So the heuristic can be updated using two table lookups. One could also use Reverse Resumable A* (RRA*) (Silver 2005) to fill in the table values, which improved the efficiency of the breadth-first search option by a small additive constant for each agent in our preliminary experiments. RRA* is used in our experiments.

Independent Subproblems

Although operator decomposition allows the algorithm to avoid considering a substantial portion of the search space, the resulting algorithm is still exponential in the number of agents. It would be ideal if one could plan a path for each agent independently. This seems to be the intuition behind most inadmissible algorithms developed up to this point (Silver 2005; Jansen and Sturtevant 2008b; Dresner and Stone 2004; Wang and Botea 2008). A slightly more general idea involves independently planning optimal paths for disjoint and exhaustive subsets of the agents. If these paths were all found to not interfere with one another, then the total solution obtained would be optimal.

In general, we would like to have an algorithm for partitioning the agents into disjoint sets in such a way that the optimal paths computed for each set of agents do not interfere with the others. This way, we can be sure that the combined solution is optimal.

In order to discover such a partitioning, we developed the simple independence detection (SID) algorithm, which assumes that the paths for all agents will be independent, and cooperatively recomputes the conflicting paths using an optimal algorithm like operator decomposition if the assumption fails. Because the planning time is exponential in the size of the largest group, the largest replan dominates the running time.

Algorithm 1 Simple Independence Detection

```
1: assign each agent to a singleton group
2: plan a path for each group
3: repeat
4:     simulate execution of all paths until a conflict occurs
5:     merge two conflicting groups into a single group
6:     cooperatively plan new group
7: until no conflicts occur
8: $solution \leftarrow$ paths of all groups combined
9: return $solution$
```

To simplify the explanations below, we only consider the case in which conflicting groups are singleton groups (the extension to general groups is straightforward).

We can improve SID by making the following observation: Each agent will usually have many optimal paths to its goal, and whether two agents are found to be independent depends on which two paths the algorithm finds initially. Therefore, we can sometimes avoid merging the groups of two agents whose paths conflict by finding another optimal path for either agent. In order to ensure optimality, the new path for an agent must have the same cost as the initial path. Furthermore, the new path must not conflict with the other agent. To realize these goals, we can give the search algorithm a cost limit and an illegal move table which is populated just before finding an alternate path.

If two agents a_1 and a_2 conflict and we decide to replan the path of a_1, we would populate the illegal move table with the transitions a_2 makes when following its current path at every timestep. When the search algorithm expands a node while replanning the path for a_1, it checks the table at that node's timestep to see which possible moves will not conflict with a_2's move for that timestep. If a path is found, then the conflict between these two agents has been resolved.

During these replans, it is important that the algorithm finds a path that creates the fewest conflicts with other agents. This can be achieved using a table similar to the illegal move table, called the conflict avoidance table, which stores the moves of all other agents for every timestep. A* will typically have a choice between several nodes with the same minimum $f()$ cost to expand first, and tie breaking is usually done in favor of nodes with the lowest $h()$ value to achieve the best performance. Alternatively, each node can keep track of the number of conflict avoidance table violations that have occurred on the path leading up to this node, and break ties in favor of nodes with the fewest violations, only relying on the $h()$ to break ties when nodes have an equal violation count as well as $f()$ cost. This method ensures that the path returned during the replan will lead to the fewest future conflicts of any optimal path.

This is the intuition behind our improved independence detection algorithm (ID), which starts by assigning each agent to its own group. It then finds an initial path for each agent independently. Upon detecting a conflict between the current paths of two agents, the algorithm attempts to find an alternate optimal path for one of the conflicting agents, ensuring that the new path does not conflict with the other agent. If this fails, it tries this process with the other conflicting agent. Finally, if both attempts to find alternate paths fail or two agents that have conflicted before are found to conflict again, the algorithm merges the conflicting groups and cooperatively plans a path for this new group without a cost limit or illegal move table. All plans are found with a conflict avoidance table which is kept up to date so that every path always results in the fewest number of future conflicts.

Algorithm 2 Independence Detection

```
 1: assign each agent to a group
 2: plan a path for each group
 3: fill conflict avoidance table with every path
 4: repeat
 5:     simulate execution of all paths until a conflict between two
        groups $G_1$ and $G_2$ occurs
 6:     if these two groups have not conflicted before then
 7:         fill illegal move table with the current paths for $G_2$
 8:         find another set of paths with the same cost for $G_1$
 9:         if failed to find such a set then
10:             fill illegal move table with the current paths for $G_1$
11:             find another set of paths with the same cost for $G_2$
12:         end if
13:     end if
14:     if failed to find an alternate set of paths for $G_1$ and $G_2$ then
15:         merge $G_1$ and $G_2$ into a single group
16:         cooperatively plan new group
17:     end if
18:     update conflict avoidance table with changes made to paths
19: until no conflicts occur
20: $solution \leftarrow$ paths of all groups combined
21: return $solution$
```

These independence detection algorithms do not solve pathfinding problems on their own. They simply call a search algorithm many times on subproblems allowing that algorithm to solve many smaller problems rather than the full problem. The independence detection algorithms can use the standard algorithm or operator decomposition as the needed search algorithm. ID will merge two groups and run a search algorithm on the combined group if it cannot find nonconflicting paths for both groups. Therefore ID is complete when coupled with a complete search algorithm. Both OD and the standard algorithm are complete.

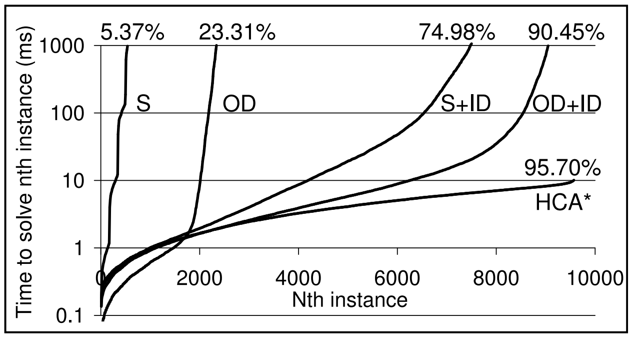

Figure 6: Running time of each instance and percent of problems solved for each algorithm.

Experiments

Experiments were run on an Intel Core i7 @ 2.8GHz using benchmarks like those proposed in (Silver 2005): 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.

The inadmissible algorithm, hierarchical cooperative A* (HCA*), from (Silver 2005), and admissible combinations of the standard algorithm (S), operator decomposition (OD), and independence detection (ID) were run on the same 10,000 instances with a random number of agents chosen uniformly between 2 and 60. Figure 6 shows the running time of each algorithm on instances that took less than a second to solve, sorted in ascending order of solution time for each algorithm. The performance of the standard algorithm was highly dependent on the number of agents in the problem which lead to the stair-step shape observed in the graph. When ID is used, OD not only helps on easy problems, solving them more quickly, but more importantly allows the algorithm to solve the harder problems. OD+ID optimally solved 90.45% of the problems in under a second each. Note that the one second time limit is arbitrary and can be increased in case of hard instances.

Only 69 instances were not solved by either OD+ID or HCA*. Moreover, OD+ID solved 361 out of the 430 problems that HCA* did not solve even though the problems had to be solved optimally for OD+ID. In the problems that were solved by both algorithms and had more than 40 agents, the optimal solutions were an average of 4.4 moves longer than the the heuristic lower bound, while the solutions produced by HCA* were an average of 12.8 moves longer than the heuristic lower bound.

ID partitions a problem instance into subproblems, and

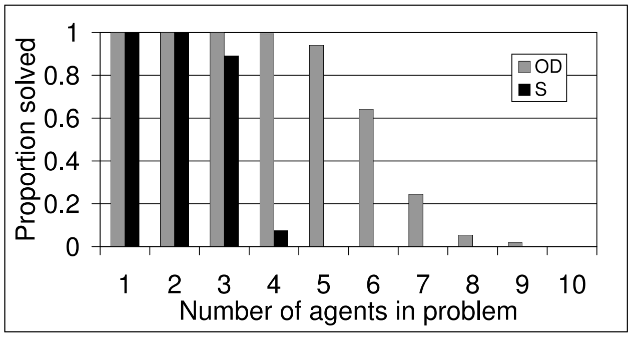

Figure 7: Proportion of largest non-independent subproblems given by ID solved within two seconds by OD and S.

the subproblem with the highest number of agents typically dominates the computation time for the instance. Experiments show that the strongest indicator of running time on these instances was indeed the number of agents in the largest non-independent subproblem solved by OD. An exponential regression yielded the equation $t = 0.465e^{1.14s}$ where t is time (ms) taken for OD+ID to solve a problem, and s is the number of agents in the largest non-independent subproblem of ID. The regression equation had an R^2 of 0.77. Figure 7 shows the results of OD and S on the largest such subproblem solved during each of the above 10,000 instances. In this experiment, OD and S are given two seconds to solve each subproblem regardless of what time was spent solving other subproblems before reaching this subproblem. Since ID could not detect independence within these subproblems, these problems are more challenging for OD and S than random instances with the same number of agents. The ability of the standard algorithm to solve these problems degrades quickly with problem size. In contrast, not only can operator decomposition solve larger problems reliably, but its ability to solve these problems also degrades more slowly with problem size.

Conclusion

This paper describes the general problem of cooperative pathfinding. We discussed the pitfalls of modern incomplete and inadmissible algorithms for the cooperative pathfinding problem, and identified two improvements to the standard admissible algorithm: Operator decomposition, which was used to reduce the branching factor of the problem, was effective at allowing the algorithm to consider only a small fraction of the children of each node, and independence detection, which often allowed the paths of groups of agents to be computed independently, without sacrificing optimality. The largest improvement was gained by exploiting this independence. On the other hand, operator decomposition provided a significant benefit even when no independence was present in the problem, and scaled more gracefully to larger problems than the standard admissible algorithm. We also began characterizing the quality of optimal solutions, and the running time of our algorithm.

While both independence detection and operator decomposition are applicable to cooperative pathfinding problems in general, we presented a specific example of cooperative pathfinding which was modeled after typical video game designs, and used this problem to test the presented algorithms as well as to clarify the algorithm descriptions. The results of the tests show that the standard admissible algorithm can be substantially improved. Furthermore, we showed that when the techniques are used in combination, the algorithm can be made practical.

Acknowledgments

This research has been supported by NSF grant No. IIS-0713178 to Richard Korf. We also sincerely thank Richard Korf and Dawn Chen for their many edits, suggestions and discussions. Artwork by Dawn Chen.

References

Dresner, K., and Stone, P. 2004. Multiagent traffic management: A reservation-based intersection control mechanism. In *In The Third International Joint Conference on Autonomous Agents and Multiagent Systems*, 530–537.

Geramifard, A.; Chubak, P.; and Bulitko, V. 2006. Biased cost pathfinding. In Laird, J. E., and Schaeffer, J., eds., *AIIDE*, 112–114. The AAAI Press.

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

Hearn, R. A., and Demaine, E. D. 2005. PSPACE-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. *Theor. Comput. Sci.* 343(1-2):72–96.

Hopcroft, J.; Schwartz, J.; and Sharir, M. 1984. On the Complexity of Motion Planning for Multiple Independent Objects; PSPACE-Hardness of the Warehouseman's Problem. *The International Journal of Robotics Research* 3(4):76–88.

Jansen, M. R., and Sturtevant, N. R. 2008a. Direction maps for cooperative pathfinding. In Darken, C., and Mateas, M., eds., *AIIDE*. The AAAI Press.

Jansen, R., and Sturtevant, N. 2008b. A new approach to cooperative pathfinding. In *AAMAS*, 1401–1404. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems.

Ryan, M. R. K. 2008. Exploiting subgraph structure in multi-robot path planning. *J. Artif. Int. Res.* 31(1):497–542.

Silver, D. 2005. Cooperative pathfinding. In Young, R. M., and Laird, J. E., eds., *AIIDE*, 117–122. AAAI Press.

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

Wang, K.-H. C., and Botea, A. 2008. Fast and memory-efficient multi-agent pathfinding. *AAAI 2008*.

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