Proceedings of the First
Artificial Intelligence and Interactive
Digital Entertainment Conference
(AIIDE-2005)

Cooperative Pathfinding

David Silver

Department of Computing Science
University of Alberta
Edmonton, Canada T6G 2E8
silver@cs.ualberta.ca

Abstract

Cooperative Pathfinding is a multi-agent path planning problem where agents must find non-colliding routes to separate destinations, given full information about the routes of other agents. This paper presents three new algorithms for efficiently solving this problem, suitable for use in Real-Time Strategy games and other real-time environments. The algorithms are decoupled approaches that break down the problem into a series of single-agent searches. Cooperative A* (CA*) searches space-time for a non-colliding route. Hierarchical Cooperative A* (HCA*) uses an abstract heuristic to boost performance. Finally, Windowed Hierarchical Cooperative A* (WHCA*) limits the space-time search depth to a dynamic window, spreading computation over the duration of the route. The algorithms are applied to a series of challenging, maze-like environments, and compared to A* with Local Repair (the current video-games industry standard). The results show that the new algorithms, especially WHCA*, are robust and efficient solutions to the Cooperative Pathfinding problem, finding more successful routes and following better paths than Local Repair A*.

Keywords: path planning, search techniques, group behaviour

Introduction

Pathfinding is a critical element of AI in many modern game genres. Real-Time Strategy games present particularly large challenges to pathfinding algorithms (Pottinger 1999), due to the presence of multiple agents on a congested, dynamic map. Although the classic A* algorithm (Hart, Nilsson, & Raphael 1968), can route a single agent to its destination, *multi-agent pathfinding* (Erdmann & Lozano-Perez 1987) must be used when multiple agents are present, to avoid collisions between the agents. A* can be adapted to reroute on demand (Stout 1996), a procedure known as *Local Repair*. However, this algorithm is known to be inadequate in many respects (Zelinsky 1992). On difficult maps with narrow passageways and many agents, these pathfinding systems break down and give rise to bottlenecks, deadlock, and cyclic repetition. In many cases game designers are forced to work around the shortcomings of the pathfinding system in order to ensure good behaviour (Pagan 2001). This paper introduces new algorithms for dealing with multi-agent pathfinding more robustly and effectively, suitable for use in challenging, real-time environments.

Real-Time Strategy games may have differing requirements for the multi-agent pathfinding system. In *Cooperative Pathfinding* each agent is assumed to have full knowledge of all other agents and their planned routes. In the complementary problem, *Non-Cooperative Pathfinding*, agents have no knowledge of each other's plans, and must predict their future movements. Finally, in *Antagonistic Pathfinding* each agent tries to reach its own goal whilst preventing other agents from reaching theirs. This paper focusses exclusively on the Cooperative Pathfinding problem.

Latombe (1991) and Fujimura (1991) break down the possible approaches to multi-agent pathfinding into two categories. *Centralised* planning takes account of all agents at once, and can find all possible solutions. However, the problem is PSPACE-hard (Hopcroft, Schwartz, & Sharir 1984) and the search space in a typical game is prohibitively large for this approach to be practical. For example, given n agents in a 32x32, 4-connected grid there are approximately 10^{3n} different states with a branching factor of 5^n ($N, E, S, W, wait$ for each agent). The *decoupled* or *distributed* approach decomposes the task into independent or weakly-dependent problems for each agent (Erdmann & Lozano-Perez 1987). Each agent can then search greedily for a path to its destination, given the current state of all other agents. Four decoupled algorithms are presented in this paper, one existing algorithm and three new ones.

Local Repair A*

Local Repair A* (LRA*) describes a family of algorithms widely used in the video-games industry. Each agent searches for a route to the destination using the A* algorithm (Hart, Nilsson, & Raphael 1968), ignoring all other agents except for its current neighbours. The agents then begin to follow their routes, until a collision is imminent. Whenever an agent is about to move into an occupied position it instead recalculates the remainder of its route. The basic algorithm corresponds to the brute-force replanner described by Zelinsky (1992).

Cycles are possible (and indeed common) using this algorithm, so it is usual to try and add some modifications to

escape such problems. One possibility, used here, is to increase an agent's *agitation level* every time it is forced to reroute. Random noise is then added to the distance heuristic in proportion to the agitation level. As the agents behave increasingly more randomly it is hoped that they will escape from the problematic area and try different routes.

Local Repair A* is known to have several severe drawbacks in difficult environments (Zelinsky 1992; Stout 1996; Pottinger 1999). If bottlenecks occur in crowded regions, they may take arbitrarily long to be resolved. Whilst caught in a bottleneck, agents constantly reroute in an attempt to escape, requiring a full recomputation of the A* search almost every turn. This leads to visually disturbing behaviour that is perceived as unintelligent. Each change in route is made independently, leading to cycles in which the same location may be visited repeatedly in a loop. The remainder of this paper develops new algorithms to overcome these problems by the use of cooperative search.

Cooperative A*

Cooperative A*(CA*) is a new algorithm for solving the Cooperative Pathfinding problem. The task is decoupled into a series of single agent searches. The individual searches are performed in three dimensional space-time, and take account of the planned routes of other agents. A $wait$ move is included in the agent's action set, to enable it to remain stationary. After each agent's route is calculated, the states along the route are marked into a *reservation table*. Entries in the reservation table are considered impassable and are avoided during searches by subsequent agents.

The reservation table represents the agents' shared knowledge about each other's planned routes. It is a sparse data structure marking off regions of space-time. The choice of data structure is independent from the state space of the agents themselves. In general, individual agents may vary in speed or size, and the reservation table must be capable of marking off any occupied region.

A simple implementation, used here, is to treat the reservation table as a 3-dimensional grid (two spatial dimensions and one time dimension). Each cell of the grid that is intersected by the agent's planned route is marked as impassable for precisely the duration of the intersection, thus preventing any other agent from planning a colliding route. Only a small proportion of grid locations will be touched, and so the grid can be efficiently implemented as a hash table, hashing on a randomly distributed function of the (x, y, t) key.

It is important to note that any decoupled, greedy algorithm that precalculates the optimal path will not be able to solve certain classes of problem. This can happen when a greedy solution for one agent prevents any solution for another agent, for example see Figure 1. In general, such algorithms are sensitive to the ordering of the agents, requiring that sensible priorities to be selected for good performance, for example using Latombe's *prioritized planning* (1991).

Any admissible heuristic can be used in CA*. The base case is to use the Manhattan distance. However, this can give poor performance in more challenging environments. Ideally, a better heuristic should be used to help reduce the computation.

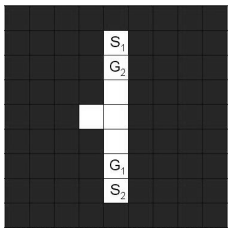

Figure 1: This simple problem cannot be solved by Cooperative A*. One agent must navigate from S_1 to G_1 whilst the other attempts to get from S_2 to G_2.

Hierarchical Cooperative A*

There are two generic methods for improving a heuristic based on abstractions of the state space. The first approach is to precompute all distances in the abstract space, and store them in a pattern database (Culberson & Schaeffer 1994). However, this is not feasible in Cooperative Pathfinding because the map is dynamic and the goal can change at any time. The second generic method is to use Hierarchical A* (Holte *et al.* 1996). With this approach the abstract distances are computed on demand, which is more appropriate in a dynamic context. The hierarchy in this case refers to a series of abstractions of the state space, each more general than the previous, and is not restricted to spatial hierarchy. Holte notes that the choice of hierarchy is critical, and that large hierarchies may in fact perform worse than small, simple hierarchies.

Hierarchical Cooperative A* (HCA*) uses a simple hierarchy containing a single domain abstraction, which ignores both the time dimension and the reservation table. In other words, the abstraction is a simple 2-dimensional map with all agents removed. Abstract distances can thus be viewed as perfect estimates of the distance to the destination, ignoring any potential interactions with other agents. This is clearly an admissible and consistent heuristic. Furthermore, the inaccuracy of the heuristic is determined only by the difficulty of interacting with other agents (how much the agent must deviate from the direct path to avoid other agents).

One of the issues in Hierarchical A* is how best to reuse search data in the abstract domain. If searches are performed from scratch each time around, then overall performance may in fact be reduced (Holte *et al.* 1996). Holte introduces 3 different techniques for reusing search data in his paper. A fourth approach is introduced here, which is to use a Reverse Resumable A* (RRA*) search in the abstract domain.

The RRA* algorithm executes a modified A* search in a reverse direction. The search starts at the agent's goal G, and heads towards the agent's initial position O. Instead of terminating at O, the search continues until a specified node N is expanded. A well-known property of A* with a consistent heuristic is that the optimal distance from the start to a node is known once that node is expanded (Nilsson 1980). As the search executes in a reverse direction, this means that the optimal distance from N to G is known once the RRA* search completes. The Manhattan distance is used as a heuristic for the RRA* search, meeting the consistency

requirements.

Unlike previous work in this area (Zhou & Hansen 2004), the RRA* algorithm is used to calculate the abstract distance on-demand. Whenever an abstract distance is requested from N to G, RRA* checks whether N exists in its Closed table. If so, the optimal distance of that node is already known and can be returned immediately. If not, the RRA* search is resumed until node N is expanded. Pseudocode for the RRA* procedure is shown in algorithm 1.

Algorithm 1 Reverse Resumable A*

```
 1: procedure INITIALISERRA*(O, G)
 2:     G.g ← 0
 3:     G.h ← MANHATTAN(G, O)
 4:     Open ← {G}
 5:     Closed ← ∅
 6:     RESUMERRA*(O)
 7: end procedure

 8: procedure RESUMERRA*(N)
 9:     while Open ≠ ∅ do
10:         P ← pop(Open)
11:         Closed ←(add)— P
12:         if P = N then
13:             return success
14:         end if
15:         for all Q ∈ reverse(SUCCESSORS(P)) do
16:             Q.g ← P.g + COST(P, Q)
17:             Q.h ← MANHATTAN(Q, O)
18:             if Q ∉ Open and Q ∉ Closed then
19:                 Open ←(add)— Q
20:             end if
21:             if Q ∈ Open and f(Q) < f(Q in Open)
                    then
22:                 Open ←(update)— Q
23:             end if
24:         end for
25:     end while
26:     return failure
27: end procedure

28: procedure ABSTRACTDIST(N, G)
29:     if N ∈ Closed then
30:         return g(N)
31:     end if
32:     if RESUMERRA*(N) = success then
33:         return g(N)
34:     end if
35:     return +∞
36: end procedure
```

HCA* is just like CA* with a more sophisticated heuristic, using RRA* to calculate the abstract distance on demand. If the shortest path to the destination is clear of agents then the initial call to RRA* should contain the distances to all required nodes. This is achieved by reversing the ordering of the successor function in RRA* to ensure that ties are broken in the same direction (e.g. a left fork in the forward direction should become a right fork in the reverse direction).

If there are other agents on the path to the destination then HCA* will get pushed away from the shortest path. In this case, new nodes will be encountered and the RRA* search must be continued until they are reached. Each resumption RRA* will expand nodes in concentric rings of equidistance from the shortest path until the required node has been found.

Windowed Hierarchical Cooperative A*

One issue with the previous algorithms is how they terminate once the agents reach their destination. If an agent sits on its destination, for example in a narrow corridor, then it may block off parts of the map to other agents. Ideally, agents should continue to cooperate after reaching their destinations, so that an agent can move off its destination and allow others to pass.

A second issue is the sensitivity to agent ordering. Although it is sometimes possible to prioritise agents globally (Latombe 1991), a more robust solution is to dynamically vary the agent order, so that every agent will have the highest priority for a short period of time. Solutions can then be found which would be unsolvable with an arbitrary, fixed agent order.

Thirdly, the previous algorithms must calculate a complete route to the destination in a large, three-dimensional state space. With single agent searches, planning and plan execution are often interleaved to achieve greater efficiency (see for example Korf's Real-Time Heuristic Search (1990)), by avoiding the need to plan for long-term contingencies that do not in fact occur. WHCA* develops a similar idea for cooperative search.

A simple solution to all of these issues is to *window* the search. The cooperative search is limited to a fixed depth specified by the current window. Each agent searches for a partial route to its destination, and then begins following the route. At regular intervals (e.g. when an agent is half-way through its partial route) the window is shifted forwards and a new partial route is computed.

To ensure that the agent heads in the correct direction, only the cooperative search depth is limited to a fixed depth, whilst the abstract search is executed to full depth. A window of size w can be viewed as an intermediate abstraction that is equivalent to the base level state space for w steps, and then equivalent to the abstract level state space for the remainder of the search. In other words, other agents are only considered for w steps (via the reservation table) and are ignored for the remainder of the search.

To search this new search space efficiently, a simple trick can be used. Once w steps have elapsed, agents are ignored and the search space becomes identical to the abstract search space. This means that the abstract distance provides the same information as completing the search. For each node N_i reached after w steps a special terminal edge is introduced, going directly from N_i to the destination G, with a cost equal to the abstract distance from N_i to G. Using this

trick, the search is reduced to a w-step window using the abstract distance heuristic introduced for HCA*.

In addition, the windowed search can continue once the agent has reached its destination. The agent's goal is no longer to reach the destination, but to complete the window via a terminal edge. Any sequence of w moves will thus reach the goal. However, the WHCA* search will efficiently find the lowest cost sequence. This optimal sequence represents the partial route that will take the agent closest to its destination, and once there to stay on the destination for as much time as possible.

In general, the edge cost function for WHCA* is:

$$\text{COST}(P,Q) = \begin{cases} 0 & \text{if } P = Q = G, t < w \\ \text{ABSTRACTDIST}(P,G) & \text{if } t = w \\ 1 & \text{otherwise} \end{cases}$$

An additional benefit of windowing is that processing time can be spread across all agents. By staggering the windows appropriately, searches can be smoothly interleaved. With n agents and a window size of w, recalculating routes at the midpoint of each window, only $2n/w$ searches need be performed per turn. If a turn consists of many frames, then the resumable search naturally breaks down further and can be spread across multiple frames.

Finally, the RRA* search results can be reused for each consecutive window. This requires each agent to store its own Open and Closed lists. An initial RRA* search is performed for each agent from its original position O to its goal G. For each subsequent window, the RRA* search is resumed to take account of any new nodes encountered. For consistency, it must continue to search towards the agent's original position O, and not the agent's current position. This means that expansions will take place in concentric rings of equidistance about the original shortest path. The efficiency of this approach will reduce if agents are forced far away from the original path. However, the savings from reuse should in general outweigh any additional expansions caused by deviation.

Experimental Results

To test the algorithms, a series of 10 challenging, maze-like environments were generated. Each environment consisted of a 32x32 4-connected grid, with impassable obstacles randomly placed down in 20% of the grid locations. Any disconnected subregions were additionally filled with obstacles to guarantee a fully connected map. Agents were placed into this environment with randomly selected start positions and destinations, subject to the constraint that no two agents shared the same start position or destination. An example environment is shown in Figure 2.

An agent was considered successful if it was able to reach its destination within 100 turns. If the agent exceeded this time limit, was unable to find a route, or collided with any other agent, then it was considered to be a failure. The average success rates of each algorithm are shown in Figure 3. With few agents, all of the algorithms are able to achieve 100% success rates. However, with more crowded maps Local Repair A* begins to struggle and 20% of the agents fail

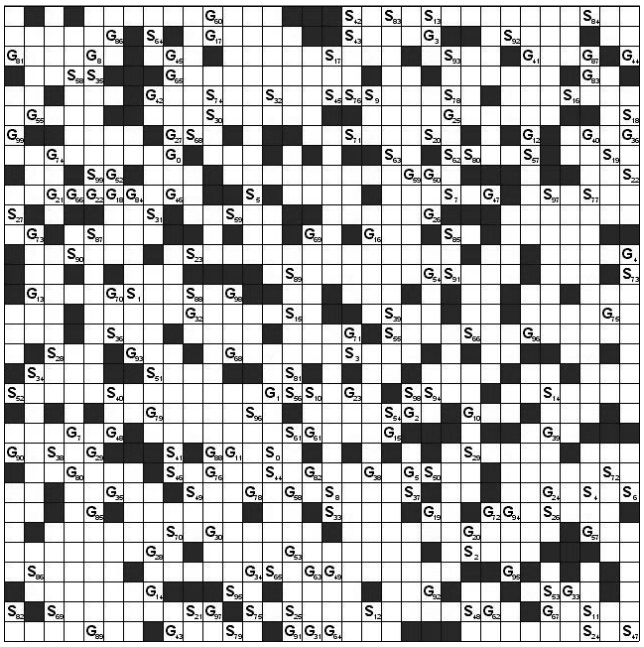

Figure 2: An example environment. Agents must navigate from S_i to G_i.

to reach their destinations. A large throng of agents tends to build up in a key bottleneck, and the high agitation level of the agents is insufficient to escape the deadlock. When using any of the three Cooperative A* algorithms, agents are able to step aside and allow others to pass, dealing comfortably with bottleneck situations. Even with 100 agents, under 2% of the agents fail to reach their destination when using WHCA* with a window size of 16 (window size is indicated in parentheses in the figures).

The path length of each agent's route was also measured. This is the number of turns that elapse before an agent first reaches its destination, or equivalently the length of the path through space-time. This is important since agents may $wait$ for many turns when using Cooperative Pathfinding. Average path lengths for each agent are shown in Figure 4. The shortest path is also shown, assuming no other agents are present. This provides a lower bound on the optimal path length achievable by any algorithm. With few agents on the map, all the algorithms find routes which are close to the optimal bound. As the number of agents is increased, interactions begin to occur and agents must deviate from the shortest path. Using Local Repair A* these deviations can be significant, with path lengths of more than twice the optimal lower bound when 100 agents are present. In contrast CA* and HCA* deviate by just 20% from the optimal lower bound.

The path length is only one indicator of route quality. In addition, it is desirable for agents to follow routes which have the appearance of intelligence and consistency. An approximate estimate of the route quality is the number of cycles occurring. A cycle was counted every time an agent revisited a grid location that it had previously visited in the

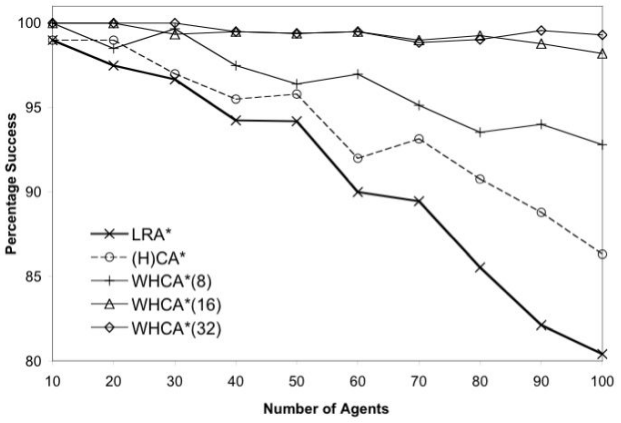

Figure 3: Percentage of agents that successfully reach their destination

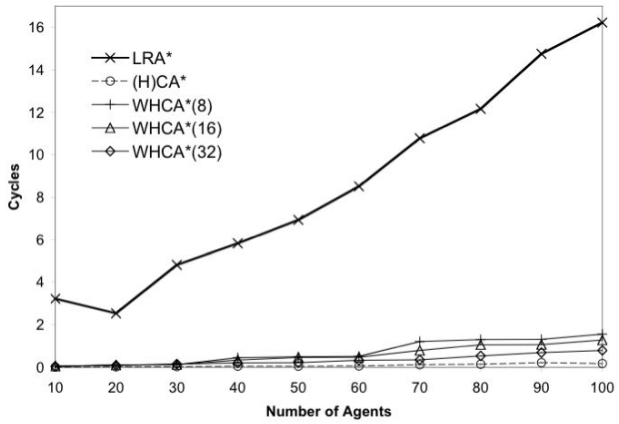

Figure 5: Average number of cycles in each agent's route

same trial. The average number of cycles for each agent is shown in Figure 5. CA* and HCA* produce very few cycles, since they precompute the entire route cooperatively. However, they are occasionally forced to reroute once agents have reached their destination, as they have no with ongoing cooperation. WHCA* does have ongoing cooperation, but reroutes at regular intervals according to the window. Despite the regular recalculations, the paths maintain an overall consistency, guided towards the destination by the abstract heuristic. This keeps cycles down to an acceptable frequency (averaging at most 1.5 cycles per agent with 100 agents). Local Repair A* produces more than 10 times this number of cycles, corresponding to the large number of agents that mill around aimlessly in the bottleneck regions.

The processing time for each algorithm is broken down into two separate components. The total processing time to calculate initial routes for all agents is shown in Figure 6. After agents have begun to follow their routes, the total processing time across all agents is calculated every turn. For real-time games, the maximum processing time for any turn is particularly important, shown in Figure 7. All times were measured on a 1.2GHz Pentium 4 processor. Local Repair A* is fast to initialise and to use, although it decreases in efficiency once many agents are present, due to more frequent rerouting. CA* and HCA* are prohibitively slow to initialise as they must perform full-depth cooperative searches in space-time.

The efficiency of WHCA* depends on the window size. A smaller window size requires the least initialisation but must reroute the most frequently, although at little cost. A larger window size must precalculate a larger proportion of the route, rerouting rarely but with a higher cost. In practice window sizes of 8, 16 and 32 have similar ongoing costs, but an initial calculation time that increases with window size. Using a window size of 16, 100 agents can initialise in under 100ms with a maximum ongoing cost of around 50ms per turn.

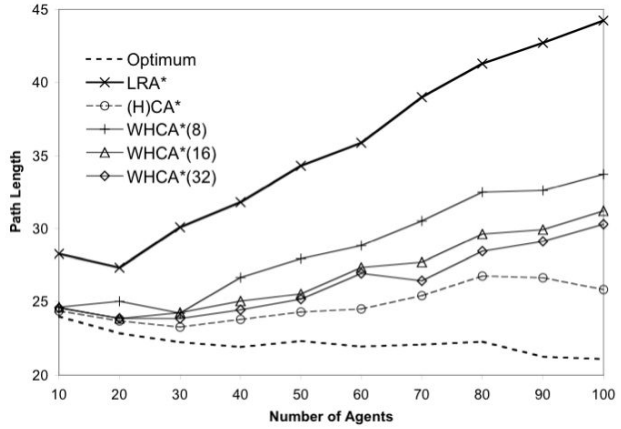

Figure 4: Average path length for each agent

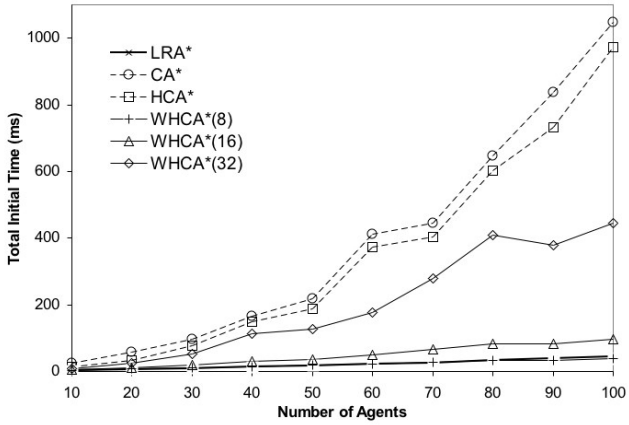

Figure 6: Total initial calculation time for all agents

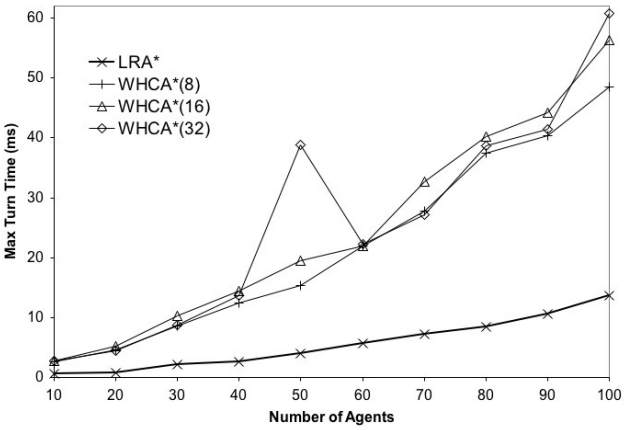

Figure 7: Maximum total calculation time for all agents on any turn

Discussion

The cooperative pathfinding methods are more successful and find higher quality routes than A* with Local Repair. Unfortunately, the basic CA* algorithm is costly to compute, taking over a second to calculate 100 routes. Using a hierarchical heuristic reduces the cooperative search cost, but with an overhead in the abstract domain. Although slightly lower than CA*, the initial cost of HCA* is still too great for real-time applications. By windowing the search to a fixed depth, the precalculation can be reduced to under 1ms per agent. The abstract search is spread over the duration of the route, guiding the cooperative search quickly towards the destination. The ongoing cost of WHCA* is under 0.6ms per agent, suitable for real-time use.

The size of the window has a significant effect on the success and performance of the algorithm. With a large window, WHCA* behaves more like HCA* and the initialisation time increases. If the window is small, WHCA* behaves more like Local Repair A*. The success rate goes down and the path length increases. The window size parameter thus provides a spectrum between Local Repair A* and HCA*. An intermediate choice (16 in these environments) appears to give the most robust overall performance. In general, the window size should be set to the duration of the longest predicted bottleneck. In Real-Time Strategy games groups of units are often moved together towards a common destination. In this case the maximum bottleneck time with cooperative pathfinding (ignoring units in other groups) is the number of units in the group. If the window size is lower than this threshold, bottleneck situations may not be resolved well. If the window size is higher, then some redundant search will be performed.

Conclusion

Local Repair A* may be an adequate solution for simple environments with few bottlenecks and few agents. With more difficult environments, Local Repair A* is inadequate and is significantly outperformed by Cooperative A* algorithms. By introducing Hierarchical A* to improve the heuristic and using windowing to shorten the search, a robust and efficient algorithm, WHCA*, was developed. WHCA* finds more successful routes than Local Repair A*, with shorter paths and fewer cycles. It is efficient both at calculating routes and for ongoing processing, with a run-time cost that is suitable for use in real-time video-games.

Although this research was restricted to grid environments, the algorithms presented here apply equally to more general pathfinding domains. Any continuous environment or navigational mesh can be used, so long as each agent's route can be planned by discrete motion elements. The grid-based reservation table is generally applicable, but reserving mesh intersections may be more appropriate in some cases. Finally, the cooperative algorithms may be applied in dynamic environments, so long as an agent's route is recomputed whenever invalidated by a change to the map.

References

Culberson, J., and Schaeffer, J. 1994. Efficiently searching the 15-puzzle. Technical report, Department of Computer Science, University of Alberta.

Erdmann, M., and Lozano-Perez, T. 1987. On multiple moving objects. *Algorithmica* 2:477–521.

Fujimura, K. 1991. *Motion planning in Dynamic Environments*. New York, NY: Springer-Verlag.

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

Holte, R. C.; Perez, M. B.; Zimmer, R. M.; and MacDonald, A. J. 1996. Hierarchical A*: Searching abstraction hierarchies efficiently. In *AAAI/IAAI, Vol. 1*, 530–535.

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. *International Journal of Robotics Research* 3(4).

Korf, R. E. 1990. Real-time heuristic search. *Artificial Intelligence* 42(2-3):189–211.

Latombe, J. 1991. *Robot Motion Planning*. Boston, MA: Kluwer Academic.

Nilsson, N. J. 1980. *Principles of Artificial Intelligence*. Morgan Kaufmann.

Pagan, T. 2001. Where's the design in level design? *Gamasutra*.

Pottinger, D. 1999. Implementing coordinated movement. *Game Developer Magazine*.

Stout, B. 1996. Smart moves: Intelligent pathfinding. *Game Developer Magazine*.

Zelinsky, A. 1992. A mobile robot exploration algorithm. *IEEE Transactions on Robotics and Automation* 8(6).

Zhou, R., and Hansen, E. A. 2004. Space-efficient memory-based heuristics. *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04)* 677–682.