Fast and Memory-Efficient Multi-Agent Pathfinding

Ko-Hsin Cindy Wang and **Adi Botea**

NICTA and the Australian National University
{cindy.wang|adi.botea}@nicta.com.au

Abstract

Multi-agent path planning has been shown to be a PSPACE-hard problem. Running a complete search such as A* at the global level is often intractable in practice, since both the number of states and the branching factor grow exponentially as the number of mobile units increases. In addition to the inherent difficulty of the problem, in many real-life applications such as computer games, solutions have to be computed in real time, using limited CPU and memory resources.

In this paper we introduce FAR (Flow Annotation Replanning), a method for multi-agent path planning on grid maps. When building a search graph from a grid map, FAR implements a flow restriction idea inspired by road networks. The movement along a given row (or column) is restricted to only one direction, avoiding head-to-head collisions. The movement direction alternates from one row (or column) to the next. Additional rules ensure that two locations reachable from each other on the original map remain connected (in both directions) in the graph. After building the search graph, an A* search is independently run for each mobile unit. During plan execution, deadlocks are detected as cycles of units that wait for each other to move. A heuristic procedure for deadlock breaking attempts to repair plans locally, instead of running a larger scale, more expensive replanning step.

Experiments are run on a collection of maps extracted from BALDUR'S GATE[1], a popular commercial computer game. We compare FAR with WHCA*, a recent successful algorithm for multi-agent path planning on grid maps. FAR is shown to run significantly faster, use much less memory, and scale up to problems with more mobile units.

Introduction

Multi-agent path planning involves navigating units from their starting positions to their respective goals, whilst going around any static obstacles and other moving units along the way. The problem is important in many real-life applications, including motion planning in robotics (Bennewitz, Burgard, & Thrun 2002), air traffic control (Pallottino *et al.* 2007; Tomlin, Pappas, & Sastry 1998), vehicle routing (Sharma *et al.* 2007), disaster rescue (Kitano *et al.* 1999), and computer games (Silver 2006; Buro & Furtak 2004).

[1]http://www.bioware.com/games/baldurs_gate

Multi-agent pathfinding is much more challenging than the single agent case (Pearl 1984), being a PSPACE-hard problem (Hopcroft, Schwartz, & Sharir 1984; Reif 1979). The number of states and the branching factor grow exponentially with the number of mobile units on a map, as units move simultaneously. Often, a global search is intractable even for a small number of mobile units. Approaches to address this include manually abstracting a search space, when the actual topology allows it (Ryan 2008), and decomposing a problem into smaller searches (Silver 2006). Replanning is often used to handle undesired events such as collisions or deadlocks between mobile agents. Trading the method completeness and the solution optimality for improved performance is a typical feature of decentralised approaches, including the method described in this paper.

In many applications such as computer games, path planning problems have to be solved in real time (Bulitko *et al.* 2007). Often, a search algorithm gets limited access to the CPU and memory resources, which are allocated with higher priority to other game modules such as the graphics engine. A path planning algorithm able to scale up to many agents while using reasonably small resources can make a significant contribution to the overall quality of a game.

Contributions

In this paper we introduce FAR (Flow Annotation Replanning), a new method for multi-agent path planning on grid maps. To avoid head-to-head collisions and to reduce the branching factor in search, FAR builds a *flow-annotated search graph* inspired by two-way roads. The movement along a row (or column) is restricted to only one direction. Two adjacent rows (or columns) have different directions, similarly to the lanes on a two-way road. The idea is applied to all rows and columns, covering the entire map with virtual criss-crossing roads. Additional rules are implemented to guarantee that any two locations connected through a path on the original grid map are still connected in both directions in the flow-annotated graph. An A* search is independently run in the flow-annotated graph for each mobile unit. As soon as a search is completed, the computed path is cached and the memory used by the open and the closed list is released. During the execution of the planned paths, deadlocks are detected as cycles of agents that wait for each other to move. A heuristic method attempts to solve deadlocks locally instead of resorting to a more expensive replanning step.

FAR is evaluated empirically on a set of maps extracted from BALDUR'S GATE, a popular commercial computer game. We compare FAR with WHCA*, an algorithm for multi-agent path-planning on grid maps. The original WHCA* (Silver 2006) has recently been enhanced with spatial abstraction (Sturtevant & Buro 2006). We use the enhanced version as a benchmark. In experiments, FAR is significantly faster, requires less memory, and scales up to problems with more mobile units.

The rest of this paper is structured as follows. First we review related work. Next we introduce the problem addressed in this work. A detailed description of FAR follows, after which we present the experimental work. Finally we conclude and outline future work.

Related Work

There is a rich body of literature on multi-agent path planning and great variety in the exact problem that each algorithm is designed to solve. Multi-agent pathfinding work can be grouped into two types of methods. A centralised approach (Barraquand & Latombe 1991; LaValle & Hutchinson 1998) has a single global decision maker for all agents, is theoretically optimal but, as discussed before, it does not scale up to many agents due to a prohibitive complexity. A decoupled (decentralized) approach decomposes the problem into several subproblems. The latter approach is faster but yields suboptimal solutions and loses the completeness. One such example is prioritised planning (Erdmann & Lozano-Perez 1986), which uses prioritisation to assign an order in which the objects move.

The direct application of A* to multiple agent planning is a game industry standard procedure called Local Repair A*, which performs an expensive full A* for every replan, outlined in (Silver 2006). It has drawbacks such as an intensive CPU usage (Pottinger 1999), bottlenecks and cycles (Alexander 1992; Stout 1996).

In Silver's work on Cooperative Pathfinding (2006), units perform *windowed* planning from start to goal in a grid-based world. A backwards A* search is initially run for each agent. The results of these searches (i.e., full open and closed lists) are cached and used as a heuristic guidance in subsequent windowed planning steps, so that each unit knows fully the routes planned by other units. Silver's algorithm *Windowed Hierarchical Cooperative A** (*WHCA**) was later enhanced by Sturtevant and Buro, by combining it with spatial abstraction to improve on WHCA*'s significant memory usage (Sturtevant & Buro 2006).

Ryan 2008 introduces a method for multi-agent path planning. His method is complete but restricted to specific search graphs, which can be decomposed into structures such as chains or rings. A problem is hierarchically decomposed into a global level and a collection of local problems, one for each subgraph such as a chain or a ring.

Automated spatial abstraction has demonstrated its success in path-finding. As map sizes scale up, searching in a reduced space, as opposed to the large, ground-level space, shows much more effective results. Techniques for abstraction include HPA* (Botea, Müller, & Schaeffer 2004), PRA* (Sturtevant & Buro 2005). Such methods build abstracted search graphs from a grid map with the goal of speeding up the search in single-agent search. In contrast, our method of abstracting a grid map with flow annotation aims at reducing collisions in multi-agent search.

Besides grid maps, many other types of methods have been proposed to convert a map into a search graph. Another commonly used representation of the environment is the roadmap, which is a connectivity graph of the map space, as is used for example in (Ryan 2008). First and second order Voronoi graphs can be combined to compute a Multi-agent Navigation Graph (MaNG) for global path planning, applied to crowd simulation in computer graphics (Sud *et al.* 2008). In robotics, motion planning can be modelled in a 2D plane of objects (Lumelsky & Harinarayan 1997). Bounding boxes are used in multi-robot path coordination (Leroy, Laumond, & Siméon 1999). Other geometric approaches can model the world as a collection of objects such as polygons (Arikan, Chenney, & Forsyth 2001).

Problem Definition

This work assumes that the problem topology is represented as a grid map. Arguably, grid maps are the most popular approach to abstract a real-world navigation map into a search space. The map is discretized into a grid of atomic locations called tiles. Adjacency relationships are defined in eight directions, four cardinal and four diagonal. Tiles with eight movement directions are also called octiles. Each tile is either *accessible* (traversable) or *blocked* by a permanent obstacle.

Each mobile agent occupies exactly one accessible tile at a time. A tile cannot host more than one agent at a time. An accessible tile is *free* at a given time if no agent occupies it. Agents move between two adjacent positions instantaneously, from one discrete time step to the next. A movement can be performed only if (1) the destination is free at the current time, and (2) no other agent plans to occupy it at the next time step. There is one additional constraint for diagonal moves. They are allowed only if at least one of the two tiles that separate the starting tile from the target is free at the current time (Figure 1a). Otherwise, it would be physically impossible to squeeze the agent through two diagonal locations that are each occupied either by an obstacle or by an agent (Figures 1b and 1c). The distance travelled is $\sqrt(2)$ for a diagonal move, and 1 for a cardinal move.

The standard way to build a search graph from an octile grid map is to define one node for each accessible tile. Undirected edges are defined between adjacent nodes, except for two diagonal tiles separated by two blocked tiles. (See the next section for a search graph enhanced with flow annotation.) In a problem instance, each agent is associated with exactly one start and one target node (tile). A node location cannot host more than one starting position (or more than one target). We assume homogenous agents and uniform speed. After an agent reaches its target, it does not disappear from the map. When an agent sitting on its target interferes

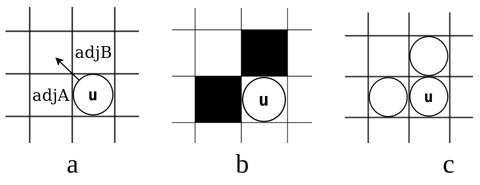

Figure 1: (a) To move diagonally, u checks the 2 directly adjacent neighbouring tiles, labelled $adjA$ and $adjB$, are not both blocking. The cases (b) and (c) are examples where u cannot make the diagonal move.

with other mobile units, they need to collaborate to solve the interference. A problem is solved when all mobile units sit on their target locations.

The FAR Method

Our goal is to achieve a scalable algorithm while keeping CPU and memory requirements low. FAR starts with a preprocessing step where a grid map is abstracted into a *flow-annotated search graph*, a structure enhanced with flow restrictions to avoid head-on collisions and to reduce the branching factor in search. Next, a complete A* search is run for each unit independently, computing a path that ignores the other agents on the map. Plan execution starts as soon as a path is computed for each agent. A strategy based on *waiting* and *reservation* attempts to avoid deadlocks. A deadlock contains two or more agents that wait for each other in a cycle. When deadlocks cannot be avoided, a deadlock breaking procedure attempts to repair plans locally instead of a larger-scale replanning step. The rest of this section contains details on the method's main steps.

Flow-Annotated Search Graph

As outlined in the previous section, grid maps are typically converted into undirected search graphs, where moving between two adjacent locations is allowed in both directions. In contrast, we build a directed graph where the navigation flow is better controlled. First, a flow annotation is imposed on the map. Then additional rules are applied to preserve the map *connectivity* in the new graph. The connectivity is preserved if two locations reachable from each other on the original map are still reachable in both directions in the annotated search graph. Initially no diagonal moves are considered. Relatively few diagonal edges might need to be added later, as will be explained.

The flow annotation limits the traffic on each row (column) to only one direction. The idea is expanded to cover the entire grid such that nodes in alternating rows have the same horizontal flow, and alternating columns flow in the same vertical direction, as illustrated in Figure 2. This can be thought of as an extension of a road system, where we cover the entire grid with crisscrossing virtual roads. The path to a target is likely to involve a longer route with more turns to account for the imposed flow, just like how a driver navigating to her destination in a city has to obey the local traffic flow by following the roads, driving on the correct

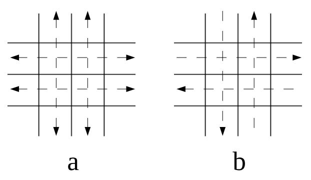

Figure 2: A classical map at the left and an annotated map at the right. For clarity, diagonal moves are not shown.

side of the road, and making turns along the way. In particular, going to a neighbour that used to be adjacent in the undirected graph, but is located at the node *against the flow* in the flow restricted setting, is now done in at least 3 steps instead of 1.

Flow annotation replaces undirected (i.e., bi-directional) edges with directed ones. To preserve the map connectivity, we use additional rules to ensure that each two adjacent nodes remain connected in both directions via a path of one or more steps. It can be easily proven that if any two adjacent nodes are connected in both ways (local connectivity) then any two locations connected on the undirected map remain connected in the flow-annotated graph (global connectivity). We prefer to keep adjacent locations connected through *short* paths. Otherwise, the quality (length) of paths computed on a flow-annotated map might suffer significantly.

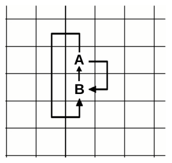

Figure 3: Local connectivity.

Consider the nodes A and B in Figure 3. The flow defines a direct transition from B to A. If at least one of the two paths from A to B, shown in the figure, is obstacle free, then the local connectivity between A and B is established and no additional measures are required. On the other hand, if both paths are blocked by obstacles, we repair the local connectivity between A and B, even if another longer path might connect A to B outside this neighbourhood. As mentioned before, shorter paths are better.

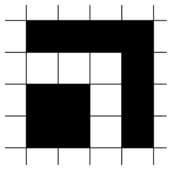

Figure 4: A single-width tunnel.

When restoring the connectivity, a measure as simple as making the edge between A and B bi-directional again will do. For a uniform treatment of nodes on the map edge and

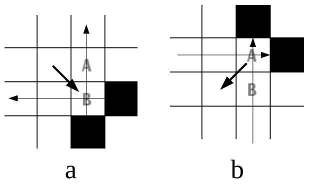

Figure 5: Sources (a) and sinks (b).

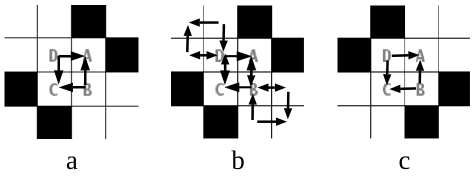

Figure 6: In (a), A and C are sinks. They are fixed by making four edges bidirectional as shown in (b). In (c), B and D are sources.

internal nodes, we assume that the map is bordered by a layer of blocked cells. A (single-width) tunnel on a map is a chain of nodes (locations) such that each internal node is connected to exactly two neighbours in cardinal directions. A tunnel can be open to both ends (Figure 4) or only to one end. Edges between the nodes of a tunnel are made bi-directional so that every two nodes along the tunnel are reachable from each other.

On certain patterns of blocked cells, such as the *source* and *sink* nodes shown in Figure 5, we are able to restore the connectivity by adding new diagonal edges. A source is a corner node with two outgoing edges and no incoming edges. To allow mobile units to visit such nodes, we add an incoming diagonal edge. A sink is a corner node with two incoming edges but no outgoing edges. A new outgoing diagonal edge allows units that visit a sink not to get trapped. The diagonal in each case is added when it does not come from another source, or lead into another sink. Figure 6a presents a pattern that creates two sink nodes, A and C. The two sinks will not be fixed by having diagonals leading into each other. Instead, making a pair of orthogonal edges bidirectional at either diagonal end, as shown in Figure 6b, does the trick. Figure 6c shows a pattern with two source nodes. It is handled in a similar manner.

The intention is to always add new edges in resolving special cases where some default incoming or outgoing flows are blocked off, either by inserting additional diagonals or reverting back to bidirectional edges. We never delete edges from the default flow assignment. Once local connectivity is maintained, global connectivity follows directly.

The limitation of using only the local configuration of obstacles to revise the annotation is that we can not exploit the overall topology. A “twisted ladder” of trajectories results from the slanted narrow passageway, as shown in Figure 7. It would have been better to enable unidirectional diagonal edges in such cases, following the same annotation idea. We revisit problems produced from such closely interwoven collision points at the end of the deadlock section.

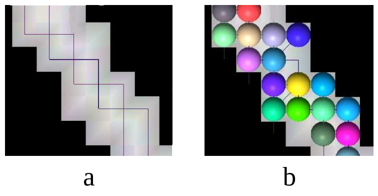

Figure 7: The “twisted ladder” trajectory.

Executing Plans

Just as the roads in real life are designed to prevent *head-on* car crashes, imposing flow restrictions as outlined in the previous section achieves the same effect. Again, similarly to roads, at junctions, where our “virtual roads” intersect, *side-on collisions* could still happen quite easily unless our mobile units communicate their intended moves beforehand. Our strategy for achieving such collaborative behaviour is founded on two main ideas. Firstly, we try to avoid situations that lead to replanning. Secondly, when replanning has to be done, we try a local, cheap computation instead of an expensive, large-scale search.

A simple approach that turned out to be effective in reducing potential collisions is to keep paths as straight as possible, since fewer turns reduce the chance for side-on collisions. To favor straighter paths, the A* search that computes a path for a mobile unit has an additional criterion to order the nodes in the open queue. As in standard A*, the main ordering criterion is the $f = g + h$ value. Ties are broken by favoring the successors which continue the current path prefix in a straight line.

To provide inter-unit cooperation during plan execution, we adopted the temporal reservation idea from (Silver 2006) such that each unit is able to make its next intended moves known to all other units. Our approach differs however from Silver’s: instead of integrating reservation in an expensive online search down to a fixed depth, our units reserve a number of steps ahead on their independently pre-computed paths. The basic idea is that all units must reserve a node prior to moving there. All units make reservations for k steps ahead (where k is a parameter). At the completion of those k moves, the next k steps have to be reserved to continue the execution of the path. A unit can start moving only if it has successfully reserved the next bunch of k steps - unless it is less than k steps away from its target. The node at step $i \leq k$ in the current sub-sequence can be reserved only if the unit can make the move to the previous location at step $i - 1$. In an area of high density, where multiple paths intersect the same, “busy” node, neighbouring units coming from orthogonal directions could compete for the same next node. When such conflicts occur, we first allow horizontal movements to have priority, then verticals, alternating in the ensuing time steps. In this way, we impose a temporal flow regulation on top of the spatial one that is annotated on the map. This measure achieves a similar effect as having traffic lights at road intersections.

Once a unit arrives to its target it stays there by returning the wait action in all subsequent time steps unless it blocks another unit's path. In the latter case, our method forces the blocking unit to take a step away from its target. The blocking unit then uses A* to path-plan back to its original blocking location, always following the annotated flow.

Another case where the need for replanning arises is when a unit is found to be critical in breaking a *deadlock*. To break a deadlock, the critical unit inserts a few detour moves into its plan. This is described in detail in the following section.

Deadlocks

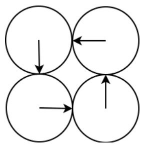

Figure 8: Deadlock: a cycle of 4 units.

The deadlocks we consider are the *circular wait* situations described in (Coffman, Elphick, & Shoshani 1971). Figure 8 shows an example where four units wait for each other to move in a cycle. Assume they have reached the shown configuration after arriving at the end of their respective k-step reservations. Recall from the restriction stated in our problem definition that a move is not allowed if the target is occupied at the current time, even if the occupying agent plans to leave the position. Since the next locations of the four agents are occupied at the current time step, they are unsuccessful in making new reservations, so each of them waits for one time step before trying again to reserve a new sub-path of k steps. The units are waiting for one another, and no one is able to make further progress. With many units placed on a map, deadlocks can happen quite easily, and propagate quickly as the units in a deadlock could block more units behind them.

When addressing deadlocks, the first step is to identify whether a deadlock has occurred. The condition that triggers the deadlock detection algorithm is that a unit u cannot move at the current step because the next node on its planned path is blocked by a unit who is not on its target. In this case, the deadlock detection needs to be launched. The procedure recursively builds a chain of agents, each waiting for the next one to move, starting from unit u. The check terminates either when it comes across a unit wanting to move to an unoccupied node, in which case no deadlock was detected, or when a unit that is already seen is re-encountered. In the later case a deadlock is discovered. Launching the deadlock detection procedure every time a unit u satisfies the triggering condition is a good trade-off. In most cases, if there is no deadlock then the procedure should terminate quickly, as the detection reaches an unoccupied node. When there is a deadlock, it is better to identify it sooner and deal with it before it propagates into a massive gridlock.

After identifying all the units in a deadlock, we force a unit called a *critical unit* to diverge temporarily off its pre-planned trajectory, as explained later. The desired effect is

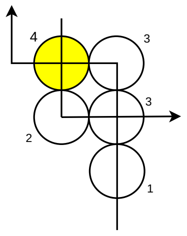

Figure 9: Five units in a deadlock with the node densities labelled at each location. The coloured unit sits on the highest density node.

that a number of units close to the critical unit are now able to pass through given the freedom from the operated change, ideally leading to more units being free to pass through. After each deadlock breaking maneuver, we call the detection procedure again. If no deadlock is found, then the job is done satisfactorily. Otherwise the original deadlock might have been reduced, rather than eliminated completely, so we repeat the process by selecting a new critical unit.

Selecting a critical unit is based on a *node density* heuristic. The density of a node is the number of computed paths that pass through the node. It is continuously updated such that each time a unit moves to its next location, the density count of the previous node is decremented by 1. With each local replan, the count in nodes involved in detours or step-aside trajectories are incremented accordingly. Since the unit in the deadlock who is sitting on the highest density node is likely to be blocking the largest number of units, moving it away might produce the greatest effect on the deadlock (Figure 9). Thus, among the units that are able to move aside, the one whose position has the largest node density is selected as the critical one.

The deadlock breaking procedure works as follows. First, the critical unit takes one step away from the deadlock. In most cases (e.g., when no diagonal edges have been added), there are 2 outgoing flows at where the critical unit sits. One direction is that unit's intended move, which goes towards the deadlock (otherwise, the agent wouldn't be part of a deadlock), and hence there is only one other location to step aside to. At the next time step, the critical unit will replan its way back to the previous position. The journey back takes at least three time steps in general. Meanwhile, units blocked by the critical unit have a chance to pass through.

The procedure outlined above has shown to be quite effective in practice. Only in cases of narrow passageways that become too congested (Figure 7b), a single change made by the critical unit might produce a very limited effect due to the restrictive space and repeated collision points that prevent units from moving along quickly. Then the deadlock breaking works slowly. Fortunately, the modularity of FAR's approach allows different sub-procedures to be implemented in the future for improvement.

Experimental Results

Our experiments were run on a 2GHz Intel Core Duo MacBook laptop with 1GB of RAM. We used the 10 largest maps (the size of traversable terrain range from 13765 to 51586 tiles) from the BALDUR'S GATE map collection, a standard data set used in several previous studies (e.g., Botea, Müller, & Schaeffer 2004). These grid maps are available at `http://users.rsise.anu.edu.au/~cwang/`. For each map, we grow the number of mobile units N by 100 at a time. For each N, we generate 10 problems with the start and target locations randomly placed on the map. The time limit is set to 10 minutes per problem. For each problem, the measured performance parameters are the total search time, the total distance travelled by all units, the maximum size of the open and closed lists in A* search, and the total number of nodes expanded. For each N, we count how many out of the 10 problems are solved successfully.

We compared FAR with two variations of WHCA*(w, a) (Sturtevant & Buro 2006), one with diagonals and the other without (note that Silver's original WHCA* considered cardinal movements only). The parameters are $w = 8, a = 1$, a combination that seems to work best in the authors' experiments. Currently, parameter k in FAR is set independently from the grid map at hand. A good value that was empirically found is $k = 3$. For a better understanding of the results, a brief explanation of WHCA*'s main steps is necessary. WHCA* starts with a backwards search from the target to the starting position for each agent. As a result, the perfect distance from each visited node to the target is computed. These distances will be used as a heuristic guidance in subsequent fixed-horizon (i.e., windowed) searches (the h score in A*). The closed and open lists of all initial backward searches have to be kept in memory such that the search can be extended to compute the distance to target for more nodes as required. Every new unit added to the map introduces a new pair of an open and a closed list to be stored in memory.

Detailed results for one map, selected to be representative for the results on all other maps, are shown in Figure 10. A summary of the results on all maps is contained in Table 1. Consider Figure 10 first. In terms of memory usage, the requirements of both WHCA* variants grow linearly with the number of agents, whereas FAR stays about constant since it throws away its open and closed lists each time a unit completes its initial A* search. The space requirements for FAR are bounded by the size of the map, just as in a single agent problem. The memory requirements of WHCA* grow linearly in the number of agents.

In terms of total search time and total nodes expanded, WHCA* faces an exponential growth. In addition to the original backwards searches, WHCA* continuously performs windowed searches to find the next steps in each agent's path. The total search effort in FAR, dominated by its initial A* searches, grows linearly with the number of agents. Replanning is done locally and only when it is necessary to resolve deadlocks. A useful outcome from flow regulation, in addition to reducing collisions, is that the branching factor in search is also reduced.

As expected, when comparing the total distance travelled by all units to reach their targets, the only algorithm that enables diagonal movements entirely computes the shortest solutions. Compared to this, the overhead in solution quality observed in FAR is reasonably low. The average solution length ratio between WHCA* with diagonals and FAR is 86%. On the other hand, FAR computes better solutions than WHCA* without diagonals. Finally, notice that FAR scales up to problems with more mobile units.

Table 1 summarizes the results on all maps. The three different algorithms scale up to different number of units. To enable a direct comparison between the numbers reported for each algorithm, the data included in all but the last column correspond to the number of agents N where *all* algorithms solve at least one problem. The last column displays the actual highest number of units that each algorithm can scale up to. Each number in columns 3–6 is averaged over all instances solved for the corresponding number of agents N. The results are consistent with the previous analysis. FAR is much better in terms of memory and CPU requirements. The differences in total travelled distance are small, with FAR coming between the two versions of WHCA*. FAR generally scales up to problems with more units. All methods, being decentralised and therefore incomplete, occasionally fail to solve some instances, especially as N increases. Recall that the data summarized in the table come from the hardest problems solved by all 3 algorithms, corresponding to the largest N where each succeeds at least once.

A closer look at FAR's performance shows that many of the observed failures are caused by targets placed inside single-width tunnels. Extending FAR to handle such situations better is a main direction for future work. Note that these cases are not always solvable in WHCA* either, if the window size w is not big enough.

Conclusion

We have presented a new algorithm, FAR, that uses approaches such as flow annotation and local plan repair measures to solve difficult multi-agent pathfinding problems. Our results have demonstrated that such procedures can be very effective in many cases. FAR solves problems more quickly and requires less memory than WHCA*. FAR can also solve problems with larger number of units in almost all cases. Like many other approaches in the literature, we trade the completeness for an improved efficiency. Arguably, this is an inherent trade-off to be made in large multi-agent pathfinding problems, given the difficulty of the problem.

In future work, in addition to handling the single-width tunnels as mentioned before, there are a number of extensions we plan to develop to further improve the performance of FAR. It would be worth performing two complete A* searches in pre-processing. The first would produce a "global traffic report", whereas the second would use this information to avoid potentially congested areas. We plan to structure the flow annotation information hierarchically, to better exploit the topological structure of a map. More ideas on achieving better performance in FAR include enhancing the deadlock breaking heuristic. For example, allow the unit that is closest to escaping from a deadlock to push units out of its way.

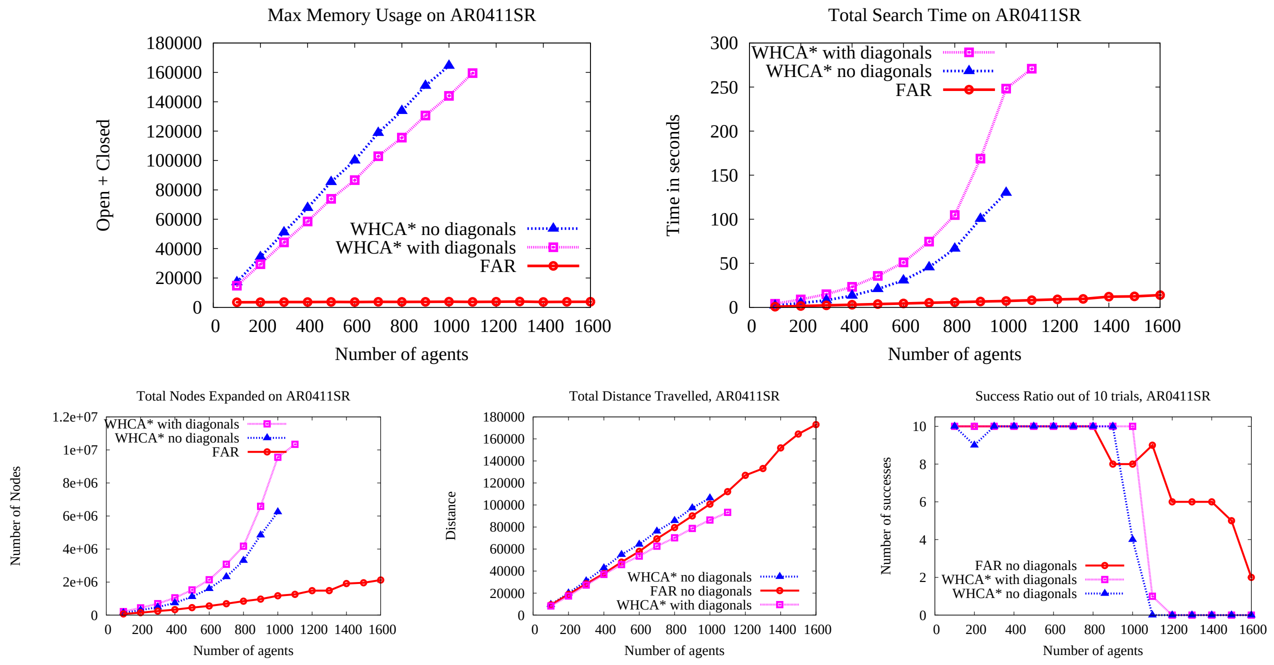

Figure 10: Results from experiments on BALDUR'S GATE map AR0411SR. Note that, for readability, the legend within each graph corresponds to the ordering of the graph lines.

Acknowledgment

NICTA is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs.

Many thanks to Nathan Sturtevant for providing the HOG[2] (Hierarchical Open Graph) framework, and his help with our understanding of the program. Thanks also to Phil Kilby, Jussi Rintanen, Kee Siong Ng and Nick Hay for their many helpful comments.

References

Alexander, Z. 1992. A mobile robot navigation exploration algorithm. *IEEE Transactions of Robotics and Automation*.

Arikan, O.; Chenney, S.; and Forsyth, D. A. 2001. Efficient multi-agent path planning. In *In Proceedings of the Eurographic Workshop on Computer Animation and Simulation*.

Barraquand, J., and Latombe, J.-C. 1991. Robot motion planning: a distributed representation approach. *International Journal of Robotics Research*.

Bennewitz, M.; Burgard, W.; and Thrun, S. 2002. Finding and optimizing solvable priority schemes for decoupled path planning techniques for teams of mobile robots. *Robotics and Autonomous Systems*.

Botea, A.; Müller, M.; and Schaeffer, J. 2004. Near optimal hierarchical path-finding. *Journal of Game Development*.

[2]`http://www.cs.ualberta.ca/~nathanst/hog.html`

Bulitko, V.; Bjornsson, Y.; Luvstrek, M.; Schaeffer, J.; and Sigmundarson, S. 2007. Dynamic Control in Path-Planning with Real-Time Heuristic Search. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*.

Buro, M., and Furtak, T. 2004. RTS games and real-time AI research. In *Proceedings of the Behaviour Representation in Modeling and Simulation Conference*.

Coffman, E. G.; Elphick, M.; and Shoshani, A. 1971. System deadlocks. *ACM Comput. Surv.*

Erdmann, M., and Lozano-Perez, T. 1986. On multiple moving objects. In *IEEE International Conference on Robotics and Automation*.

Hopcroft, J. E.; Schwartz, J. T.; 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*.

Kitano, H.; Tadokor, S.; Noda, H.; Matsubara, I.; Takhasi, T.; Shinjou, A.; and Shimada, S. 1999. Robocup-rescue: Search and rescue for large scale disasters as a domain for multi-agent research. In *In Proceedings of the IEEE Conference on Systems, Men, and Cybernetics*.

LaValle, S., and Hutchinson, S. A. 1998. Optimal motion planning for multiple robots having independent goals. *IEEE Transactions on Robotics and Automation*.

Leroy, S.; Laumond, J.-P.; and Siméon, T. 1999. Multiple path coordination for mobile robots: A geometric algorithm. In *International Joint Conference on Artificial Intelligence*.

Lumelsky, V. J., and Harinarayan, K. R. 1997. Decentralized motion planning for multiple mobile robots: The cocktail party model. *Autonomous Robots*.

Pallottino, L.; Scordio, V. G.; Frazzoli, E.; and Bicchi, A. 2007.

Map	# traversable tiles	Algorithm	Total Search Time (s)	Total Distance Travelled	Memory Usage	Total Nodes Expanded	Solved out of 10 Trials	*Max # Units Solved*
AR0700SR	51586	FAR	**41**	172061.9	**19674**	**3190181**	**7**	**1400**
		WHCA* (no diagonals)	114	175738.6	572697	5155596	**7**	1000
		WHCA* (with diagonals)	259	**151371.4**	431613	8913488	**7**	1000
AR0500SR	29160	FAR	**22**	121900.0	**11101**	**1682771**	**5**	**≥1500**
		WHCA* (no diagonals)	80	126048.4	335264	3901146	**5**	1100
		WHCA* (with diagonals)	203	**105869.8**	268609	7059405	**5**	1000
AR0300SR	26950	FAR	**26**	115862.6	**17577**	**2721906**	**5**	**≥ 1500**
		WHCA* (no diagonals)	87	121810.3	278323	4079344	4	800
		WHCA* (with diagonals)	143	**100950.8**	228299	5606509	**5**	900
AR0400SR	24945	FAR	**26**	135986.3	**9775**	**2238156**	**8**	**≥1500**
		WHCA* (no diagonals)	139	138264.8	305127	6573640	4	1100
		WHCA* (with diagonals)	303	**114727.9**	260119	11117252	3	1100
AR0602SR	23314	FAR	**14**	123954.4	**7904**	**2101600**	7	**≥1500**
		WHCA* (no diagonals)	116	128571.3	243843	5512319	6	900
		WHCA* (with diagonals)	198	**103827.0**	204235	7119173	**9**	1000
AR0414SR	22841	FAR	**25**	111644.1	**11557**	**1756200**	**10**	**≥2000**
		WHCA* (no diagonals)	101	115891.5	379538	4557328	**10**	1100
		WHCA* (with diagonals)	240	**99540.7**	308056	9289430	**10**	1000
AR0204SR	15899	FAR	**9**	87634.6	**4229**	**808477**	7	**≥2000**
		WHCA* (no diagonals)	108	94329.2	181732	4990668	**9**	1200
		WHCA* (with diagonals)	263	**76194.8**	164138	9427872	8	1100
AR0307SR	14901	FAR	**4**	77647.0	**3133**	**663099**	4	**≥1500**
		WHCA* (no diagonals)	109	84438.3	118077	5435017	3	1000
		WHCA* (with diagonals)	203	**67776.0**	103695	7530998	**8**	1100
AR0411SR	14098	FAR	**7**	100767.8	**3874**	**1171818**	8	**1600**
		WHCA* (no diagonals)	130	106335.3	164531	6247997	4	1000
		WHCA* (with diagonals)	248	**86315.0**	143990	9554315	**10**	1100
AR0603SR	13765	FAR	**8**	102169.2	**4561**	**1126541**	7	**≥1300**
		WHCA* (no diagonals)	112	109636.0	190422	5985135	1	800
		WHCA* (with diagonals)	214	**85622.5**	158887	8906368	**8**	900

Table 1: Results table for the 10 largest maps from BALDUR'S GATE. The winner in each case is shown in bold.

Decentralized cooperative policy for conflict resolution in multi-vehicle systems. *IEEE Transactions on Robotics*.

Pearl, J. 1984. *Heuristics - Intelligent Search Strategies for Computer Problem Solving*. Addison–Wesley.

Pottinger, D. C. 1999. Coordinated unit movement. *Game Developer Magazine*.

Reif, J. 1979. Complexity of the mover's problem and generalizations. In *Proceedings of IEEE Symposium on Foundations of Computer Science*.

Ryan, M. R. K. 2008. Exploiting Subgraph Structure in Multi-Robot Path Planning. *Journal of Artificial Intelligence Research*.

Sharma, V.; Savchenko, M.; Frazzoli, E.; and Voulgaris, P. G. 2007. Transfer time complexity of conflict-free vehicle routing with no communications. *The International Journal of Robotics Research*.

Silver, D. 2006. Cooperative pathfinding. *AI Programming Wisdom*.

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

Sturtevant, N. R., and Buro, M. 2005. Partial pathfinding using map abstraction and refinement. In *Association for the Advancement of Artificial Intelligence (AAAI)*.

Sturtevant, N. R., and Buro, M. 2006. Improving collaborative pathfinding using map abstraction. In *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*.

Sud, A.; Andersen, E.; Curtis, S.; Lin, M. C.; and Manocha, D. 2008. Real-time path planning in dynamic virtual environments using multiagent navigation graphs. *IEEE Transactions on Visualization and Computer Graphics*.

Tomlin, C.; Pappas, G.; and Sastry, S. 1998. Conflict resolution for air traffic management : A study in muti-agent hybrid systems. *IEEE Transactions on Automatic Control*.