Biased Cost Pathfinding

Alborz Geramifard and **Pirooz Chubak** and **Vadim Bulitko**
Department of Computing Science, University of Alberta
Edmonton, Alberta, T6G 2E8, Canada
{alborz|pchubak|bulitko}@cs.ualberta.ca

Abstract

In this paper we introduce the Biased Cost Pathfinding (BCP) algorithm as a simple yet effective meta-algorithm that can be fused with any single-agent search method in order to make it usable in multi-agent environments. In particular, we focus on pathfinding problems common in real-time strategy games where units can have different functions and mission priorities. We evaluate BCP paired with the A* algorithm in several game-like scenarios. Performance improvement of up to 90% is demonstrated with respect to several metrics.

Keywords: Multi-Agent Pathfinding; Biased Cost Pathfinding; Real-time Heuristic Search

Introduction

Multi-agent and group pathfinding is an area of active research in the heuristic search and games communities. Modern real-time strategy games challenge existing single agent pathfinding algorithms (Botea, Müller, & Schaeffer 2004; Bulitko & Lee 2005; Korf 1993; Koenig 2004; Sturtevant & Buro 2005) and have given rise to specialized multi-agent pathfinding algorithms (Silver 2005; 2006). As multi-agent optimal pathfinding has been proven to be a PSPACE-hard problem (Hopcroft, Schwartz, & Sharir 1984), heuristic search algorithms are a popular way of trading off computational time and solution optimality.

Most algorithms on multi-agent pathfinding fall into two categories: *centralized*, in which all paths are computed jointly by a central unit, and *distributed*, which lets each unit decide on its path and resolve potential collisions locally. In this paper we will focus on *centralized* pathfinding since in most real-time strategy (RTS) games complete information on all units is available at all times. The rest of the paper is structured as follows: first we detail the problem tackled in this work. Then, the novel algorithm, called Biased Cost Pathfinding (BCP), is introduced as an approach to port single agent pathfinding methods to multi-agent environments. The next section presents results of an empirical study situated in several RTS game-like scenarios. The paper will be concluded by discussion section.

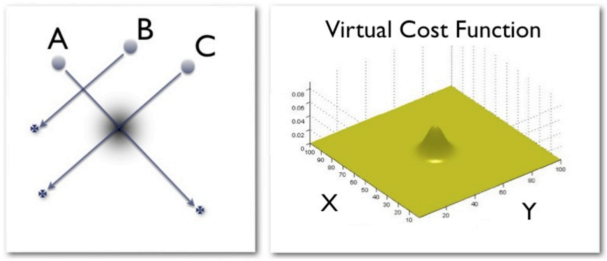

Figure 1: a) A pathfinding algorithm plan for the movement of three units. The paths of A and B intersect but they do not collide, But the paths of A and C intersect in the middle point at the same time, so there is a collision. b) A virtual cost function based on the central collision point. (μ = Collision coordination, σ = Number of colliding units)

Problem Formulation

In this paper we focus on multi-agent pathfinding for cooperating units of equal movement speed but of different priorities. The environment is an 8-connected deterministic gridworld, fully observable by all units. Collisions occur as a unit tries to move into a position occupied by another unit or when two or more units try to move into the same position at the same time. Each unit has a priority assigned to it which represents its importance in an in-game scenario. For example, injured units or special units (heros) can have higher priorities. Heuristic used is Euclidian distance.

Proposed Approach

One concern of any multi-agent pathfinding problem is to identify collisions. On the other hand, resolving all collisions can be potentially time consuming. In our approach, collisions are detected and prevented as much as planning time allows. This may result in suboptimal paths for units but lets them cooperate by coordinating their planned paths. As shown in Figure 1.a, in each iteration paths for each unit are found through an arbitrary pathfinding algorithm (e.g., A^*) without considering collisions with other units. Afterwards the exact collision points can be computed for all paths. In the example of Figure 1, the intersection between A and B is not a collision because by the time unit B reaches that point, unit A has passed it already. However, units A and C will collide at the center point. In order to resolve collisions, a biased cost function[1] is defined on such points for all of the colliding units except the unit with the highest priority. This encourages all colliding units to replan their paths except the one which will be given the highest priority. Assuming unit C has the highest priority, Figure 1.b shows a Gaussian function centered on the collision point which will cause unit A to replan its path. After one iteration the set of virtual cost functions for each agent will be updated. By adding these functions to the actual heuristic values, the path finding algorithm will find new paths for each unit. This will decrease the probability of having the same collisions on the next pathfinding trial. This process can be done iteratively until the time limit is reached or no more collisions are detected.

The BCP algorithm is shown in Figure 2. On line 3, for each unit a limited path considering the virtual heuristic values (h') from previous iterations is planned. The information on the paths is inserted into a hash table called *CollisionDetector*. After finishing the planning, all collisions are computed from the hash table. Each collision is defined by its position (x, y) and the set of units colliding in it. On line 12, the unit with the highest priority is omitted from the list. This allows that specific unit to occupy the position on the next iteration and discourage the rest of the colliding units from getting close to that collision point. Each agent maintains its own set of virtual cost functions. In line 14, the new virtual heuristic is added to the rest of the colliding units. This virtual cost is modeled as a Gaussian function with the mean at the collision location and the variance relative to the number of colliding units. On the next iteration the new paths are generated taking all virtual cost functions into account. The result returned in line 23 is the set of paths for all units with the least number of collisions that has been found so far. h' is being computed for any given position and unit by extracting the list of virtual cost functions assigned to that unit. Then all of the virtual functions are computed for the given position and summed up.

We expect BCP to help units (especially prioritized ones) reach their goals faster compared to original single agent pathfinding method. On the other hand BCP has a higher computational complexity due to repeated planning trials and heuristic refinement.

Empirical Evaluation

We situated our empirical evaluation in a recent RTS game-like open source testbed (Sturtevant 2005). It allowed us to set up controlled experiments based on scenarios from commercial games. We picked A^* as the basic pathfinding algorithm as it is commonly used in commercial games. The small scale of these scenarios enabled complete path computation. In order to increase the speed of the algorithm, the effective areas of the Gaussian functions are limited to 2σ and if one collision is detected the rest of the path is ignored. All tests have been done on a computer with an Intel P4 3.0 GHz CPU and 512 MB of RAM.

[1]This function will be used later in order to compute virtual heuristic values (h').

BCP
```
0   Colliding ← True
1   **While** time is available and Colliding **do**
2     **For** each unit $i$ on the map **do**
3         $p$ ← Limited path from the start to the goal of unit $i$
          with maximum length $k$ considering heuristic as $h + h'$
4         reset the time: $t \leftarrow 0$
5         **For** each position $n$ on path $p$ **do**
6             CollisionDetector.add($n, t, i$)
7         **end for**
8     **end for**
9     $C$ ← CollisionDetector.getCollisions()
10    **For** each collision $c$ in $C$ **do**
11        $A \leftarrow c$.units()
12        Delete the unit with highest priority from $A$
13        **For** each unit $i$ in A **do**
14            VirtualHeuristic.add($i, c.x, c.y, c.size$)
15        **end for**
16    **end for**
18    **if** C is not empty
19        Colliding ← True
20    **else**
21        Colliding ← False
22  **end while**
23  **return** set of paths with least number of collisions
```

Figure 2: Pseudo-code for BCP algorithm

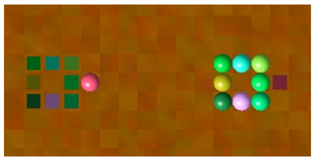

Figure 3: A sample configuation of units for Rescue the Hero scenario.

Rescue the Hero

For the first set of experiments we simulated a scenario commonly found in most RTS games. Suppose a high priority unit (called *hero* hereafter) is badly injured and the user wants to return it to the base while enemy units are attacking it. The user also sends a group of units to defend the hero and stop the enemies. Unfortunately, the hero collides with its friendly units and while struggling to find a new path, is eliminated by the attackers. Figure 3 shows this scenario in simulation. The unit on the left side (hero) wants to reach a location on the right side, while support units want to come left in order to secure the hero. In our experiments the hero had difficulties passing through the support units using A^* method. Also, support units lacked a good cooperation between themselves. However, the BCP agents anticipated forthcoming collisions with the hero and started to drift away from their straight path in the middle. As the middle units planed to shift to the bottom, the bottom most units cooperated with them and started their shift to the bottom. When the hero passed the group, they moved close and reached their goals.

In Figure 4 the results of A* and BCP algorithms are

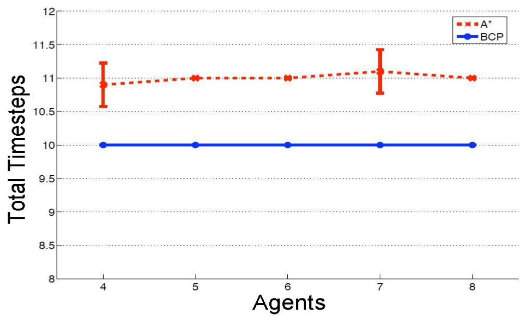

Figure 4: The comparission of A^* and BCP heros in terms of total number of steps taken by each of them to get their goal in scenario one. Results averaged over 10 experiments.

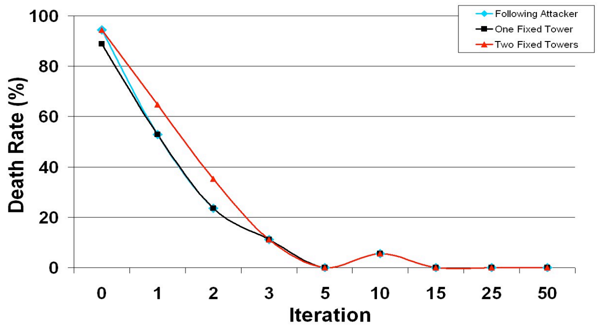

Figure 5: Death rate of the hero in three extensions of scenario one based on the number of iterations averaged over 18 samples.

shown for the aforementioned scenario with different numbers of support units positioned randomly in a 3×3 matrix at the right side. Each graph shows the time steps needed for the hero unit to reach its goal, averaged over 10 samples. The BCP hero could reach its target in 10 time steps in all cases. This means that all of the other units considered its priority and avoided any kind of collision with the hero while reaching their goal. On the other hand, the A^* hero almost always collided with other units and failed to reach its goal in 10 time steps. In all cases the required time for BCP method did not exceed half a second.

Rescue the hero: Extensions

In order to show the effectiveness of BCP in practical environments, the previous scenario has been extended into three different sub-scenarios. In the first one, the hero is fleeing from a critical situation with a low hit-point (HP) count while being chased by a ranged attacker. On each cycle if the hero is in the attacker's range it will lose one HP. If the hero stands still, an additional HP is deducted. For the next two extensions we put respectively one and two static attackers which would decrease the HP of the hero by one on each cycle if the hero is in their range. Figure 5 shows the result of all extensions. The horizontal axis illustrates the number of iterations used for BCP. Note that the first number represents the initial A^* algorithm. The vertical axis shows the death rate of the hero averaged over 18 runs with random starting positions of the hero in a 3×7 rectangle. In order to make the random tests interesting, we set the HP of the hero to the mean of all damages it took on all iterations of BCP. As Figure 5 depicts, by increasing the number of iterations the death rate of the hero decreases in all cases.

Discussion

In small-scale scenarios inspired by common situations in real-time strategy games, BCP improved A^* results for cooperative multi-agent pathfinding. Specifically, the hero unit was able to leave enemy territory without colliding with ally units. Consequently, the damage it incurred during the evacuation sequence was minimized and in the last test case the death rate has been reduced by 90%. On the down side, BCP adds overhead to the A^* computation. While scale-up experiments are in progress to establish its limits, one can speed up BCP by capping the number of iterations (m) appropriately. This allows a game AI designer to trade off pathfinding performance for computational efficiency.

Acknowledgments

We greatly appreciate the insightful comments of David Silver, Nathan Sturtevant and Anna Koop.

References

Botea, A.; Müller, M.; and Schaeffer, J. 2004. Near optimal hierarchical path-finding. In *Journal of Game Development*, volume 1, 7–28.

Bulitko, V., and Lee, G. 2005. Learning in real time search: A unifying framework. *Journal of Artificial Intelligence Research*, Volume 24.

Hopcroft, J. E.; Schwartz, J. T.; and Sharir, M. 1984. On the complexity of motion planning for multiple independent objects: Pspace-hardness for the warehousman's problem. In *International Journal of Robotics Research*, volume 3, 76–88.

Koenig, S. 2004. A comparison of fast search methods for real-time situated agents. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 864–871.

Korf, R. E. 1993. Real time heuristic search. In *Artificial Intelligence*, volume 27, 97–109.

Silver, D. 2005. Cooperative pathfinding. In *Proceedings of the 1st Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Silver, D. 2006. Cooperative path-planning. In *AI Programming Wisdom*.

Sturtevant, N., and Buro, M. 2005. Partial pathfinding using map abstraction and refinement. In *AAAI, Pittsburgh*.

Sturtevant, N. 2005. Hierarchical open graph. In *http://www.cs.ualberta.ca/ nathanst/hog/*.