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 explain the problem we are facing in more details. This will be followed by related works on multi-agent pathfinding in different domains. Then, BCP will be introduced as a solution to import single agent pathfinding methods to Multi-Agent Systems (MAS). It will be analyzed in terms of computational cost. Empirical results obtained from simulation in different maps and scenarios will illustrate the efficiency of our method. These results are evaluated afterwards. We will also talk about the possible extensions to our approach in the future work section. The conclusion part will wrap up the paper.

Problem Formulation

In an RTS game the player has to control many units with different abilities and speeds. In this paper we focus on multi-agent pathfinding for cooperating units of equal movement speed but of different priorities. The pathfinding task is defined by assigning each unit start and target positions. The following assumptions further shape the problem:

- The environment is an 8-connected gridworld. It is deterministic and fully observable by all units. The units may not move if they try to move to an obstacle or another unit's position.
- No two units can have the same start position or target.
- 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. In the latter case only one of the units is able to move successfully and the rest remain in their previous positions. If a unit is in position s at time t it will occupy this position for the times t and $t+1$. If it plans to move to position s' at time $t+1$, and the move is successful, this unit will also occupy position s' at time $t+1$
- Eight moves of an equal cost available to each unit are: $\leftarrow$, $\nwarrow$, $\uparrow$, $\nearrow$, $\rightarrow$, $\searrow$, $\downarrow$ and $\swarrow$.
- 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. Priority of a unit can be adjusted dynamically as it takes on different roles during a game.
- Heuristic used is Euclidian distance.

Related Work

Multi-agent pathfinding has been a challenging problem in robotics, studied over a number of years (Clark, Rock, & Latombe 2003; Guo & Parker 2002). Guo and Parker tackled a multi-robot pathfinding problem via the concept of a local sensory area. These robots find a path individually and keep following it. Before colliding with other units they will be identified in each other's sensory area, so they can communicate and resolve the possible collisions through real-time interaction. This method is well suited to robotic problems since usually each unit does not have a complete

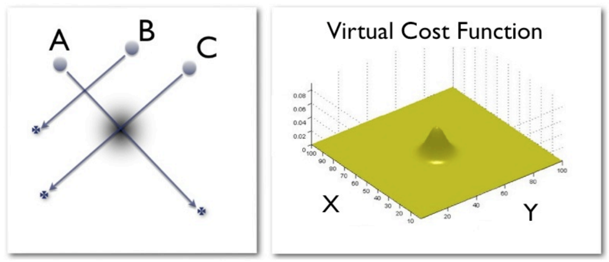

Figure 1: a) A pathfinding algorithm plan for the movement of three units. The paths of A and B intersect but they do not collide, But the paths of A and C intersect in the middle point at the same time, so there is a collision. b) A virtual cost function based on the central collision point. (μ = Collision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

knowledge of the environment. However, in RTS games *centralized* path finding is often a more effective approach as it takes advantage of complete knowledge of the environment at all times.

Current methods used by game developers do not try to solve the whole pathfinding problem as a single task. Instead, the units' initial paths are computed on an individual basis without considering projected paths of other units. Upon a collision, new paths are computed taken the current positions of all units into account but, again, ignoring projected paths. Such collisions lead to additional travel and planning cost reducing the overall pathfinding efficiency. In particular, in fast-paced RTS games, the time units waste colliding with each other can strongly influence the outcome of a battle. This problem manifests itself frequently in cluttered environments such as enemy bases or bridges or mountain paths.

Recently, an algorithm called *Cooperative Pathfinding* was poised to address these problems (Silver 2005; 2006). In this algorithm, the geographical search space is extended in the time dimension. This allows memory-sharing units to make reservations in both space and time thereby preventing collisions. The larger search space of this approach presents problems as the map size and the number of units scale up.

Proposed Approach

Overview

One concern of any multi-agent pathfinding problem is to identify collisions and avoid them as much as possible. But on the other hand, resolving all collisions can be potentially time consuming and thus adversely affect the performance in fast-paced RTS games. In our approach, collisions are detected and prevented as much as planning time allows. This may result in sub optimal paths for units but let them cooperate via 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 in 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. Therefore, on the next iteration, units are discouraged to traverse this location. 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. In each iteration the new cost function would be computed using all of the previous extra cost functions and the Euclidian distance. BCP takes its name from the use of biased cost functions. For the sake of presentation clarity we start with the basic algorithm and follow the exposition with two enhancements.

Basic Algorithm

The BCP algorithm is shown in Figure 2. On line 3, for each unit a limited path considering the virtual heuristic values (h') is planned. The information on the paths is inserted into a hash table called *CollisionDetector*. After finishing the planning part, 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. Figure 3 illustrates the way that h' function is being computed. For any given position and unit the list of virtual cost functions assigned to that unit is extracted (line 1). Then all of the virtual functions are computed for the given position and summed up in line 4.

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. In the next section we analyze this overhead.

[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: BCP algorithm in a high level pseudo-code

```
h'(n, i)
 1  $G$ ← VirtualHeuristic.Gaussians(i)
 2  $S \leftarrow 0$
 3  For each function $f$ in $G$ do
 4    $S \leftarrow S + f(n.x, n.y)$
 5  end for
 6  return S
```

Figure 3: Virtual heuristic method (h') which is called with position n and unit i.

Theoretical Analysis

Computational complexity

The *CollisionDetector* and *VirtualHeuristic* objects can be implemented using hashing tables. This means inserting each element is done in constant time. Finding collisions can be done incrementally as we insert new data into the *CollisionDetector*. Therefore, the complexity of one iteration of BCP which is finding a path for all of the units is upper bounded as follows:

$$O\Big(N(A + K) + CN\Big) = O\Big(N(A + K + C)\Big)$$

Here N is the number of units on the map; A is the time complexity of pathfinding algorithm used by BCP (e.g., A^*); K is the limit for the path (Figure 2, line 3) and C is the maximum number of collisions in all iterations. Normally, the computational complexity of the underlying algorithm (i.e. the A^* here) dominates the sum, therby reducing it to $O(NA)$. This means that BCP's *trial* complexity is the same order as planning paths individually for all of the N units. The total cost is determinied by a number of trials BCP performes. Additionally computing the virtual heuristic values[2] for each position contributes further planning cost. In the worst case on each iteration the number of virtual cost functions could be increased by K, which means all units chose the same path. Assuming m iterations to be completed successfully, this will add an extra cost of $O(\frac{Km(m-1)}{2})$ on each node expanded during pathfinding algorithm for all m iterations. In order to reduce the extra cost, two enhancements are made to the BCP.

Enhancement One: Trimmed-Gaussian functions

The effective radius of each Gaussian function is dependent on its covariance. Since outside of this range the virtual heuristic values are infinitesimal, we can decrease the computational cost substantially by cutting the effective area of each Gaussian function to a limited bound.

$$g^*(x, y) = \begin{cases} g(x, y) & \|x + y\| \leq 2\sigma \\ 0 & \text{otherwise.} \end{cases}$$

The $g^*(x, y)$ trimmed-Gaussian function is used henceforth in place of g.

Enhancement Two: Limited collision detection

In our settings, the first detected collision renders the rest of the process meaningless as replanning is due. Therefore, it would be sufficient to find the first collision of each unit and discard the rest of their path on each iteration. This reduces the worst case bound on number of Gaussian functions that can be added for a unit on each iteration to one which reduces the excessive node computation for each node to $O(\frac{m(m-1)}{2})$. Since after detecting the first collision for one unit the rest of them would be ignored, the number of collisions per trial is no longer a good evaluation parameter for the set of paths. This means, the BCP method (Figure 2, line 18) should return the last found path.

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. Small scale of these scenarios enabled complete path computation ($k = \infty$).

Scenario one: 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 back to the base while enemy units are attacking it. The user sends a group of units to defend the hero and stop the enemies (Figure 8). Unfortunately, the hero collides with its friendly units and while struggling to find a new path, is eliminated by the attackers. Figures 4 and 5

[2]The new heuristic value for each position is the sum of original heuristic value and the virtual heuristic value: $h'' = h + h'$.

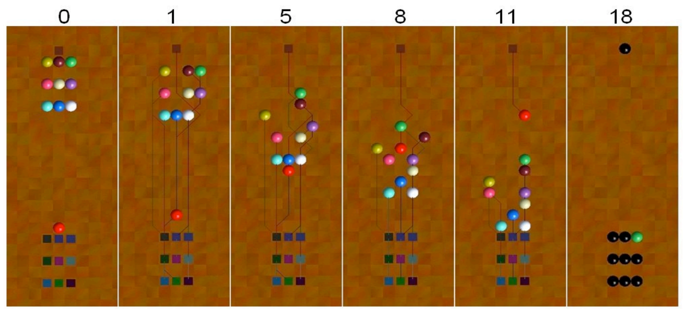

Figure 4: Scenario 1: The top nine units are intending to go down while the hero unit of a higher priority should get to its goal but the lack of cooperation makes it difficult for it to pass through. A^* is used.

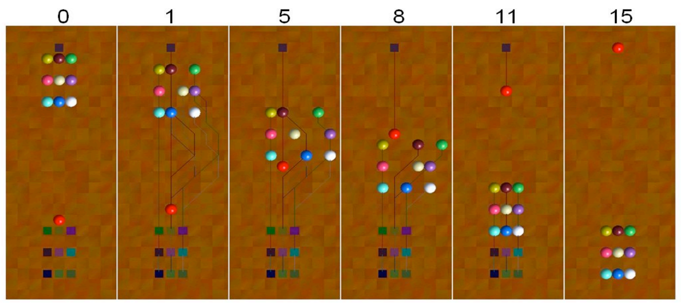

Figure 5: Scenario 1: Good cooperation between units makes it possible for the hero unit to pass through the group without changing its straight line path. *BCP* is used

show the simulation view of A^* and BCP methods for this scenario in our testbed. From left to the right, screenshots shows the simulation at time steps 0, 1, 5, 8, 11 and at the goal state. Circles and squares denote units and their goals respectively. The red unit on the bottom (hero) wants to reach a location on the top, while nine support units want to come down in order to secure the hero. It is clear that the hero has difficulties passing through the other units using A^* method. Also, support units lack a good cooperation between themselves. This behavior can be seen in Figure 4 on times 5,8, and 11. Figure 5 shows application of BCP to the same problem. Three top units in the same column as the hero, realized forthcoming collisions with the hero and started to drift away from their straight path in the middle (Figure 5, time 5). As they plan to shift to the right, the rightmost units cooperate with them and start their shift to the right (Figure 5, time 8). When the hero passes the group, they will move close and reach their goals.

In Figure 6 the results of A* and BCP algorithms are shown for scenario one with different numbers of units coming down positioned randomly in a 3×3 matrix at the top. The hero is positioned in the same column and is 10 moves away from its goal. Each graph shows the sum of time steps needed for all units to reach their goal, averaged over 10 samples. Overall, the BCP method is better than A^* algorithm, though the difference is not statistically significant.

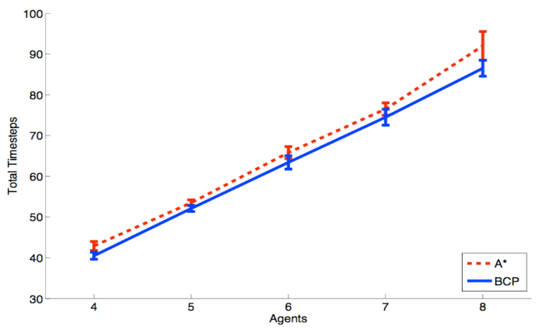

Figure 6: The comparison of A^* and BCP units in terms of total number of steps taken by all units in scenario one. Results averaged over 10 experiments.

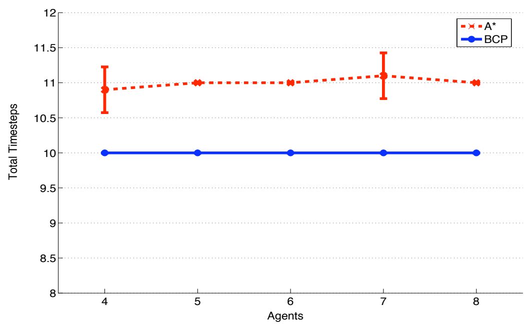

Figure 7: The comparission of A^* and BCP heros in terms of total number of steps taken by each of them to get their goal in scenario one. Results averaged over 10 experiments.

This is because the units generated on a 3×3 matrix can be initialized in a way so that they will not collide to each other, so both algorithms will return the same optimal results. Considering this fact, we can reliably claim that BCP units could reach their goals faster or with the same speed (in special cases). Note that as the number of units increases, the gap between two algorithms becomes more noticeable. This promises the scalability of BCP. Figure 7 shows similar data for the hero unit. The BCP hero could reach its target in 10 time steps in all cases (Hence zero variance in the graphs). 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. Table 1 shows the first move delay which is the time used by all of the pathfinding iterations of BCP for all units on the first move. These pathfinding times are less than half a second which is important in real time strategy games.

Scenario one: Extensions

In order to show the effectiveness of BCP in practical environments, scenario one 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

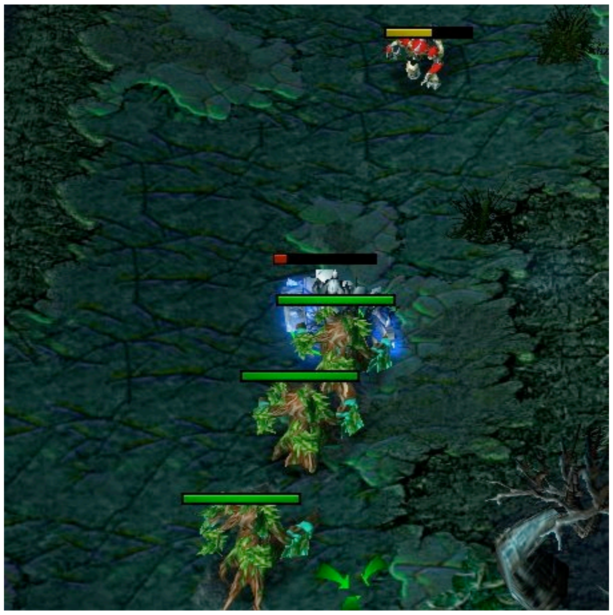

Figure 8: A screenshot of WarCraft III by Blizzard Company. The hero in critical situation has a hard time in order to retreat because of allied units while being chased by an attacker.

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 9 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 9 depicts, by increasing the number of iterations the death rate of the hero decreases in all cases. It was interesting that in all extensions we experienced a slight increase on the 10th iteration. After reviewing the simulations we realized that in general, avoiding some collisions might result in new collisions which might lead to a lower efficiency but if time allows, collisions will be resolved eventually.

Scenario two: Groups going through each other

In order to see the scalability of BCP to more complicated environments, we setup another scenario which is similar to scenario one, but has two groups of units who want to pass through each other (Figure 10). Each group can include zero to nine units located in a 3×3 matrix randomly. This increases the complexity of the path finding since now more cooperation is needed to resolve collisions. Results are shown in Figure 11 and Table 1. The total number of units is shown on horizontal axis. The gap between BCP and A^* methods grows as the size of the problem increases. At the same time the first-move delay is kept below 0.6 seconds.

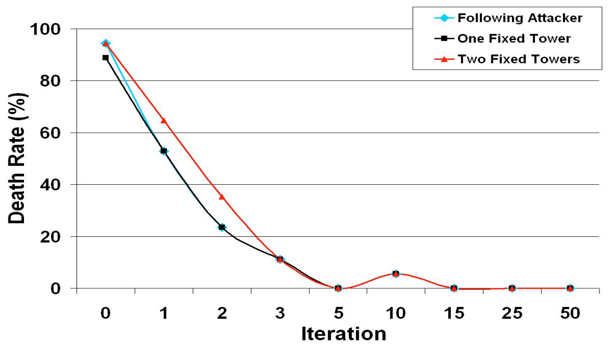

Figure 9: Death rate of the hero in three extensions of scenario one based on the number of iterations averaged over 18 samples.

Scenario 1		Scenario 2	
Units	1st move delay (ms)	Units	1st move delay (ms)
4	236	8	315
5	254	9	434
6	323	10	396
7	329	11	375
8	329	12	543
		13	458
		14	442
		15	470

Table 1: The averaged 1st move time delay of BCP algorithm in both scenarios for all of the units.

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.

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.

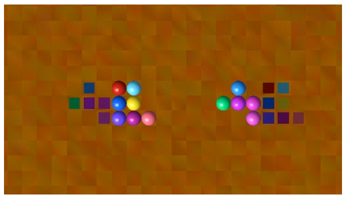

Figure 10: A snapshot of HOG simulator for a sample configuration of scenario two of 12 units. Two groups of units intend to go through each other in order to reach their goal.

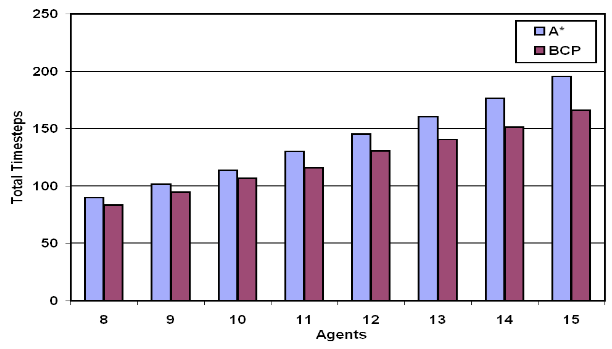

Figure 11: Comparison of BCP and A^* methods in scenario two. Bars show the total number of time step needed for all units in order to reach their goals. Still as the complexity of the task increases the distance between BCP and A^* becomes larger.

Future Work

As we discussed earlier, BCP is a general path finding algorithm that can be combined with various single agent search methods. As a future work we will investigate application of BCP to state-abstraction path refinement A* (Sturtevant & Buro 2005). This may also help us find collisions and add virtual heuristics at higher abstraction levels, resulting in a lower computation and memory cost. Another extension to our work lies in comparing BCP with other group pathfinding algorithms like *Cooperative Pathfinding* (Silver 2005). We expect to have less optimal solutions, but better time and space complexities.

Conclusions

Real-time strategy games provide a challenging testbed. In particular, optimal real-time multi-agent pathfinding is intractable and requires approximate heuristic algorithms. Classic methods such as A^*, while widely used in single-agent pathfinding, prove inadequate for cooperative multi-agent pathfinding as performance is undermined by numerous collisions among the units. The novel algorithm proposed in this paper addresses this problem by iteratively resolving collisions in the planning phase. This is done via biasing the heuristic function of potentially colliding units to repel them from collision locations and force them to prefer alternative routes. The advantage of this approach is that it can be used with *any* existing path-finding algorithm based on a heuristic function. Experimental results demonstrate BCP's effectiveness in several scenarios commonly found in RTS games. Performance gains of up to 90% are observed over the A^* search.

Acknowledgments

We greatly appreciate support of David Silver for his valuable feedbacks about the main ideas of this paper. We also want to thank Nathan Sturtevant for his support about HOG environment and Anna Koop for proof reading the paper. This research was supported by iCore, Alberta Ingenuity, NSERC, University of Alberta and RLAI lab.

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.

Clark, C.; Rock, S. M.; and Latombe, J.-C. 2003. Motion planning for multiple mobile robot systems using dynamic networks. In *Proceedings of the International Conference on Robotics and Automation*.

Guo, Y., and Parker, L. 2002. A distributed and optimal motion planning approach for multiple mobile robots.

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/*.