Tactical Path Planning

January 31, 2011

1 Problem

Path planning and navigation are critical aspects of most video games. Agents need to be able to traverse their virtual world in order to interact with each other and the player. In general, the problem of pathing has long been solved in games, with A* being the method of choice. However, this merely accomplishes the task of moving about the world without stumbling into walls. More can be done to improve the observed intelligence of game agents. In addition to the passability of the virtual world, agents should also be capable of pathing around threats presented by other game agents within the world. Also, game agents should have the option of pathing closer to designated "safe" areas in a map, such as those populated by allies or those that are far removed from threats. With that in mind, this problem also has some relevance to military movements or those done by automated machinery, such as exploratory robots. Military movement and robots such as the Mars rover could potentially benefit from a pathing algorithm that takes into account threats and safe areas.

2 Related Works

From a military perspective, there are many related works to this project. Martin Oxenham and Philip Cutler ad- dress a similar issue in a paper focused on the navigation of aircraft to targets near the presence of prohibited areas [1]. While this is similar to the problem at hand, their work is mostly focused on threat elimination rather than avoidance and is quite specific to the ideas of tactical air combat on a curved surface (the Earth).

More directly related to this project is the work of Haiquing Wang et al on the tactical planning of multiple units. Their project is on a much larger scale however, and is mostly interested in multiple agents operating together in more complex interactions such as trapping and escorting [3]. The most similar work is an overview of the AI in the game Killzone which covers concepts such as tactical position evaluation for multiple purposes [2]. The article details methods for evaluating threats at arbitrary positions and pathing around these threats as well as more advanced techniques specific to the game's genre.

3 Approach

For this project, we implemented a variation of Bidirectional A* in C#, using XNA for the GUI interface and its object/ray collision detection. Because A* finds the optimal path, it wasn't exactly what we needed. An optimal path in an environment filled with threats would essentially treat them as impassable obstacles. We wanted more leeway in the path, allowing an agent to pass trade increased threat for decreased distance. However, A* did provide a good starting framework.

To explain, the heuristic of A* is defined as f(x) = g(x) + h(x), where g(x) is the cost to travel to the node x, and h(x) is the estimated cost to go from x to the goal. One of the requirements of A* is that h(x) must never overestimate the cost of going from x to the goal node. However, we quickly found that, in order to properly utilize threatening and safe areas, it was necessary that h(x) sometimes overestimate the cost in order to bias the search against a route. If it didn't, the algorithm would, in some cases, ignore the bias against threats and towards safe areas. More importantly though, without this change, A* would expand many more nodes than it would in an environment without threats..

To bias this quasi-A* algorithm, now dubbed Better Safe Than Sorry (BSTS), we implemented h(x) as a direct-line from node x to the goal node. This line is then intersected against a list of of threats and allies to see if any of them lie along the line and modify its cost. Each threatening and safe area intersected yeilds a

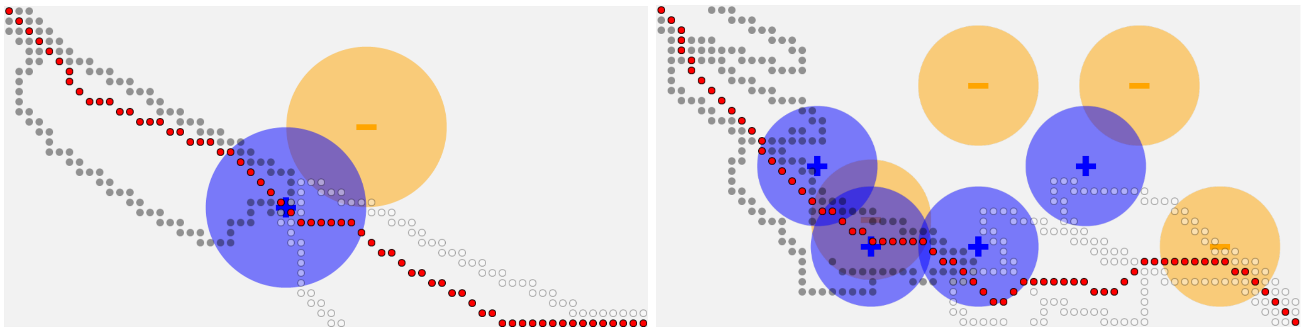
(a) One threat and one ally. (b) Four threats and four allies.

Figure 1: Example environments and searches across them. Pluses indicate allies while minuses indicate threats. Dark grey nodes are the final frontier of the forward A*, while light grey nodes are the frontier of the reverse A*. The red path denotes the returned path.

straight-line distance measuring the amount of threat along the path. These distances are subtracted from the path's Euclidean distance, and then divided by a constant c_1. The distance travelled through threatening areas is then multiplied by a second constant $c_2$and added back into the running total. Summed together, c_1 and c_2 equal 1.

$h(x, goal) = [dist(x, goal) - threatDist(x, goal) - safeDist(x, goal)] * c_1 + threatDist(x, goal) * c_2$

Once node x was chosen to be expanded, g(x) is calculated as the cost of getting to from the origin node to predecessor node of x, called y, plus the direct line cost from x to y, using the same algorithm that is used to calculate h(x).

$g(x) = g(y) + h(x, y)$

4 Evaluation

Firstly, due to the alterations made to the A* algorithm, the paths retuned by the search were not guaranteed to be optimal. This manifested as inconsistent, sub-optimal segments in usually localized parts of the return path. However, the cost calculations allow trading distance for threat and vice versa, so a path segment that traverses a threat is not automatically a bad segment. We occasionally found it difficult to eyeball the effectiveness of our algorithm as a result and would have to alter the environment and generate slightly different paths to determine how it behaved.

The more blatant sub-optimality present in the paths is a result of our non-admissible BSTS heurisitic. When ran using an admissible heuristic, the search operated as one would expect - including a drastic increase in node expansion. Bidirectional A* using only Euclidean distance as a heuristic quite often explored half the environment as it tried to find routes around the threats. As one of our intial objectives was to minimize the run-time of our search, expanding half the graph was unacceptable. As Tables 1 and 2 show, the tradeoff on optimality generated substantial savings on node exploration. Table 2 also demonstrates the cost volatility that can sometimes occur when many agents are present in nearby vicinity. It was a particularly bad case, where the path woud jump between two options based on the resolution of the discretized space.

Even with our trade-off, the paths returned by the search are usable. Figure 1 contains captures of some example searches. They are reasonably good paths, and the values for the constants in the BSTS heuristic can be adjusted to prefer more or less exposure to threats.

5 Discussion

Some of the minor inconsistencies in the paths were caused by artifacts left over from two separate searches in the Bidirectional A*. Sub-optimal bumps in the path are sometimes present where the two paths meet and are subsequently joined. These could have been removed with a more robust search tree intersection algorithm, but as the intricacies of Bidirectional search were not the primary focus of the project, we settled for a fast, dirty solution.

Nodes Explored	Total Nodes	Percentage Explored
100	512	19.5%
404	2048	19.7%
1758	8192	21.5%

Table 1: Node exploration with a single threat and ally (two total agents).

Nodes Explored	Total Nodes	Percentage Explored
62	512	12.1%
357	2048	17.4%
1600	8192	19.5%

Table 2: Node exploration with four threats and allies (eight total agents).

To visit once again on our inadmissible heuristic, it often branches off from the returned path as it attempts to fire rays that minimize (or maximize) the intersected chord lengths with threats (or allies). While this leads to sections of the graph that are explored and then abandoned, it still gave us cheaper results than an admissible heuristic. This is particularly applicable to video game agents, as there may be many path requests and a player will notice if an agent stands around while waiting for a path return. The node expansion percentages, in general, seemed to be fairly constant as the resolution of the discretized space increased. Table 1 demonstrates this nicely, and is the general case witnessed when testing various configurations. This means that the expansion costs of the algorithm are linear, based on the number of nodes in the space. We were unable to determine the effects of agent count on expansion costs, as it very much depended on the arrangement of the threats.

One consistency amongst our tests was that BSTS attempted to go around obstacles before actually meeting them; firing lines through threats did not account for distance from the threat, and a node close to the target would suffer the same threat as one at a distance. Ideally, in the future, the heuristic would also take into account distance from target, lessening the threat and allowing a path to "weave" in between threat targets rather than trying to circle around all threats at once.

Primarily, we learned that a different heuristic can drastically affect a search's exploration. We knew this conceptually before, but the search space differences we witnessed as changes were made to the project were of greater import than expected. Additionally, we found that a greedy algorithm can be used to get you 'close enough' results if you're willing to settle for less than perfection. If additional work on this project were undertaken, we would like to observe the interactions of threats areas and hard obstacles, particularly when we allow obstacles to block the a threat's influence area. We would also like to experiment with threat area of multiple geometric shapes in the same space. Improvements could still be made on speed, perhaps using a hierarchical search.

References

[1] M. G. Oxenham and P. Cutler. Accomodating obstacle avoidance in the weapons allocation problem for tactical air defense. In *Proc. 9th Int Information Fusion Conf*, pages 1-8, 2006.

[2] R. Straatman, W. van der Sterren, and A. Beij. Killzone's ai: dynamic procedural combat tactics. In *Game Developer's Conference*. Citeseer, 2005.

[3] Haiqing Wang, O. N. Malik, and A. Nareyek. Multi-unit tactical pathplanning. In *Proc. IEEE Symp. Computation Intelligence and Games CIG 2009*, pages 349-354, 2009.