# Game engines report

This report will explain the method and conclusion of testing 5 path finding algorithms and then draw conclusions from the results. The algorithms are as follows: Dijkstra, Astar(A\*), pre-calculated Dijkstra, scent mapping (Aliens vs predator) and rectangular symmetry reduction(RSR).

## Method

### Algorithm overviews

**Dijkstra/Shared Dijkstra**This expands from the player searching in every direction checking the next cell with the lowest cost and storing the parent cell for each cell, when the bot is found it it looks at the parent cells and uses them as the path

The shared version expands its search to fill the map and then lets the other bots use the search data to find their own paths meaning only one search is required for n bots

**Astar**  
Similar to Dijkstra except a heuristic is used which predicts the distance from each point to the goal, this is used to determine the next best cell to expand favouring directions towards the goal

**Pre-calculated Dijkstra**This uses Dijkstra to calculate the path from every point to every other and stores it in an array. Every non wall cell is iterated through and then Dijkstra is used to expand from that cell. The next step towards that cell form every other is stored in the array. With that map a bot can look up the next cell to travel to so that it can reach any goal

**Scent Mapping (AvP)**An array of the map is used and the players scent is added to the array. The array is then updated with every cell being equal to one less that its highest neighbour. A buffer is used to ensure uniform spreading. Bots can look for the next highest scent value near them to travel to, this should eventually lead to the player.

**Rectangular Symmetry Reduction**This method fills the map with rectangles and prunes the interior of these rectangles so the bot only has to move along edges or across them to find the best path to the player. By removing the inside of the rectangles while allowing them to be traversed it removes most of the symmetrical paths across the map. Then the method used Astar to traverse the new pruned map.

### Variables

When testing methods of pathfinding against each other common variable need to be used to ensure that comparisons are fair and relevant. Below are a list of variables that will be tested and which algorithms they apply to.

|  |  |  |
| --- | --- | --- |
| Variable | Applicable algorithms | Justification for variable and exclude algorithms |
| Time to find the path | Astar, Dijkstra, RSR | Pathfinding algorithms like most processes are preferred to be as fast as possible to allow other processes to be completed.  Pre-calculated Dijkstra has all of the paths already found and the only time required would be time to look up the next position to move to. The AvP method follows a “scent” from the player and so doesn’t actually find a path but instead moves towards the highest scent value. |
| Cells opened in finding a path | Astar, Dijkstra, RSR | With less cells opened the algorithm has been more efficient in choosing a direction to search to find a path and has used less resources. AvP and pre-calculated Dijkstra do not use open lists when path finding. |
| Cells closed in finding a path | Astar, Dijkstra, RSR | Cells closed represent those searched but weren’t the goal, the less nodes closed the less unnecessary calculations were performed. . AvP and pre-calculated Dijkstra do not use closed lists when path finding. |
| Length of path to goal | All | When pathfinding the best route is often wanted, if the path is longer than it needs to be then the algorithm is inefficient and so could run better. AvP doesn’t create a path for its bots but one can be generated to find the distance the bot would travel to get to the goal. |
| **Variables that can affect the above variables** | | |
| Number of bots | All | Multiple bots are often required in applications to path find and so finding out how well they perform with multiple bots could inform a decision in a game for example.  Note: multiple instances of all bots can be used but some algorithms have been built specifically to share some information giving them large advantages over their un altered versions, while not a feature of the algorithm it is used where available. |
| Map size | All | Applications using pathfinding such as games will have maps of different sizes and so the algorithms need to be tested with a variety of map sizes from 20 by 20 to 200 by 200 |
| Map density | All | Some algorithms might favour empty or dense maps which can affect their performance in various applications. Maps will vary from empty to mazes with one cell wide corridors |

### Time measurement

Some variables will require the time taken to be measured, these processes could take less than a millisecond and so the Stopwatch class will be used to measure times. This option was chosen over the DateTime class as the DateTime class is built for handling the time while the Stopwatch was built to accurately measure elapsed time (Microsoft).

### Tests

Each algorithm will be tested with 1, 4, 16, 64 and 256 bots on 20x20, 50x50 and 100x100 maps which are empty, low density and dense.

## References

Microsoft https://msdn.microsoft.com/en-us/library/system.diagnostics.stopwatch%28v=vs.110%29.aspx