**Optimizing A\* with lookahead and graph pruning**

**By Chloe Walters**

We are often faced with the problem of finding an optimal path between two points on a uniform grid consisting of wall tiles and empty tiles where only orthogonal movement is permitted. A special implementation of A\* can be created to take advantage of the grid-structure of our graph. Furthermore, because we are on a uniform grid, we can perform two different optimizations that can speed up the search. The first optimization we consider is inspired by jump point search (JPS).[1] JPS is designed for diagonally-connected grids, but we can use a similar method to optimize our pathfinding on an orthogonally-connected grid, we call this optimization lookahead. The second optimization we can make is graph pruning using cellular automata.[2] We compare performance between A\*, lookahead, pruned A\*, and pruned lookahead.

**Lookahead**

Our typical A\* algorithm involves storing a heap of unsearched nodes, and repeatedly popping the node with the lowest f value, which is the sum of the distance travelled (g) and the heuristic (h). our heuristic is the Manhattan distance to the destination, this is admissible on a uniform grid where all costs are 1. When we pop a node of cost f, we check its neighbours and add each unsearched neighbour to the heap. In lookahead, we can make the following optimizations:

1. We can skip checking the parent node, since we have already visited it.

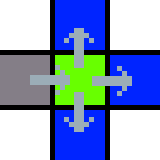


Figure - skipping the parent node

1. If a node diagonally “behind” us is empty and unsearched, we can skip the nodes on that “side”, since the parent node has an equally good path to these nodes that it can take later.

A picture containing symbol, pixel, graphics

Description automatically generated

Figure - skipping a side node, the node below us does not need to be searched because the node to our lower left can search it later

1. If any node adjacent to the current node has a smaller heuristic, we can choose one of these nodes to immediately check after the current node. This works because we know that fcurrent was a minimum value of the heap, and for any child fchild = gchild + hchild = (gcurrent + 1) + (h­current ± 1) = fcurrent + 1 ± 1, so fchild >= fcurrent, so if we find a node such that fchild = fcurrent, we know that it must be a minimum value of the heap, so we can safely choose it as the next node to search without adding it to the heap.

**Cellular automata pruning**

Given the dimension of the grid (d) we can prune the graph with two simple rules:

1. If a cell is a dead end (2d - 1 orthogonally adjacent walls), we can safely prune it.
2. If a cell is a corner (at least d orthogonally adjacent walls that are themselves diagonally adjacent), and part of a 2d grid of empty cells, we can safely prune it.

Since any tile outside the grid is impassable, we can count these as walls for the purposes of pruning. Additionally, we can count pruned tiles as walls for the purposes of pruning, which allows further pruning.

A picture containing colorfulness, screenshot, graphics, line

Description automatically generated

Figure 3 - rules for pruning illustrated on a 2d grid, blue marks a cell as irrelevant, black marks a cell as a wall, we can prune the green cell in the left and middle cases but not the right case as it could block off an important passage

The pseudocode for our pruning algorithm looks like this:

Prune(graph, position):

If cell at position is already a wall or pruned, or it is the start or end cell, return

Get all the cells of all diagonally and orthogonally adjacent neighbours to position

If should\_prune(neighbours):

Mark cell at position as pruned

Call Prune on all neighbours of position

Then we simply call Prune on every cell. Of course, if we want to optimize this algorithm, we can trivially replace the recursion with either pushes and pops to a stack or iterating over the entire graph repeatedly.

Since each neighbour contains one bit of information (wall/pruned or not), we can optimize should\_prune by packing the neighbours into an index and using a lookup table.

A picture containing line, diagram

Description automatically generated

Figure 4 - bit-indices of each adjacent cell

We use some rust code to generate the lookup table for a 2d grid:

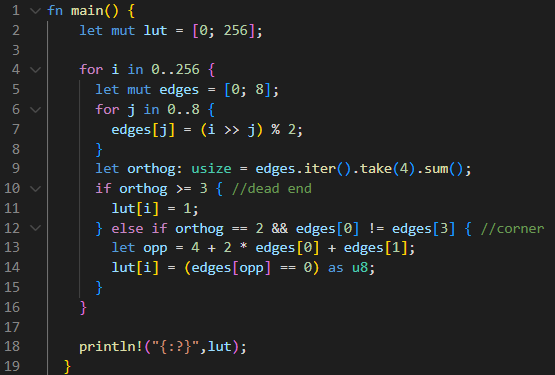


Figure 5 - lookup table calculation

In our analysis, we know our start and end points ahead of time, so we can avoid pruning them. In another scenario, we might not know how to avoid pruning away our start and end points. If we prune away the start or end point, we can solve it in two ways, either we can use flood fill to un-prune all pruned nodes around the point, or we can permit the pathfinder to move from a pruned node to another pruned node.

**Complexity Analysis**

We assume the heuristic calculation is O(1), and so not a factor in time complexity calculations. In the worst case on an arbitrary graph, A\* is equivalent to Dijkstra’s algorithm and must search O(bd) nodes, where b is the branching factor of the graph and d is the distance from the starting node to the goal. [3] In our application, the graph is known to be a uniform grid, which reduces the worst case performance to O(dn) nodes, where n is the dimensionality of our grid. On an arbitrary graph, the average case time complexity is O(b\*d) where b\* is the modified branching factor given by the heuristic, which must be experimentally determined, and can approach 1 for very accurate heuristics, we can do a similar experiment to determine time complexity on our grid, O(d\*n), where d\* must be experimentally determined.[3] In addition to the number of nodes visited, we must also consider the cost of adding and removing nodes from the heap, which in our case is a binary heap, and so has O(log(m)) time complexity, where m is the number of nodes added, d\*n. This means our total time complexity is O(d\*nlog(d\*n)). We choose to statically allocate a closed set equal to the size of the graph, so our memory complexity is O(V), where V is the number of vertices, if this became a limiting factor we could instead use a hash map, reducing our memory complexity to O(d\*nlog(d\*n)), or we could use the graph itself to path, although this would have a time complexity cost.

Both of our optimizations reduce the number of nodes that must be searched by a constant factor, this does not change our overall time complexity, so we expect a constant increase in speed from each optimization, rather than an increase that grows with the size b\*.

Lookahead uses strictly less memory than A\*. The memory usage of pruning depends on whether we’re permitted to modify the graph in-place or must allocate a new graph, if we must allocate a new graph then the memory complexity of pruning is O(V), where V >> d\*nlog(d\*n).

Pruning has an additional time complexity factor: the pruning process itself, which is O(V), which will be much larger than our pathfinding complexity. In a real world situation we would likely do many paths, which would amortize this one time cost. However if we truly only wanted one path, pruning could potentially cost us much more time than it saves. We will compare both the cost of pruning + pruned A\*, and the cost of pruned A\* alone, to measure these two cases. A secondary factor in pruning is the fact that it is a very simple process, this means that it could be offloaded to the gpu in a real world scenario, which could trivialize the cost.

**Graph setup**

For simplicity, we choose a grid of dimension 2, and we choose to path from the top left corner of the grid to the bottom right corner, this make the solutions easy to visualize and allows us to profile very large grids without running out of memory. Generalizing to higher dimensions and arbitrary dimensions should be trivial.

We chose to use Perlin noise to generate an initial distribution of walls on the graph. We then set the start and end tiles to be passable, and use flood fill to find empty cells and punch holes in the walls until all empty cells are connected.

**Results**

We compared the performance of A\*, lookahead, pruned A\*, and pruned lookahead on various grid sizes. We also recorded the number of nodes discovered, expanded, and pushed to the heap for each algorithm, to determine whether our assumptions about lookahead and pruning being constant factor reductions in number of nodes visited is true.

Lookahead proved to be very effective in low density maps as can be seen in the figures below, in this example A\* took 220x as long, and pushed 384x as many nodes to the heap.

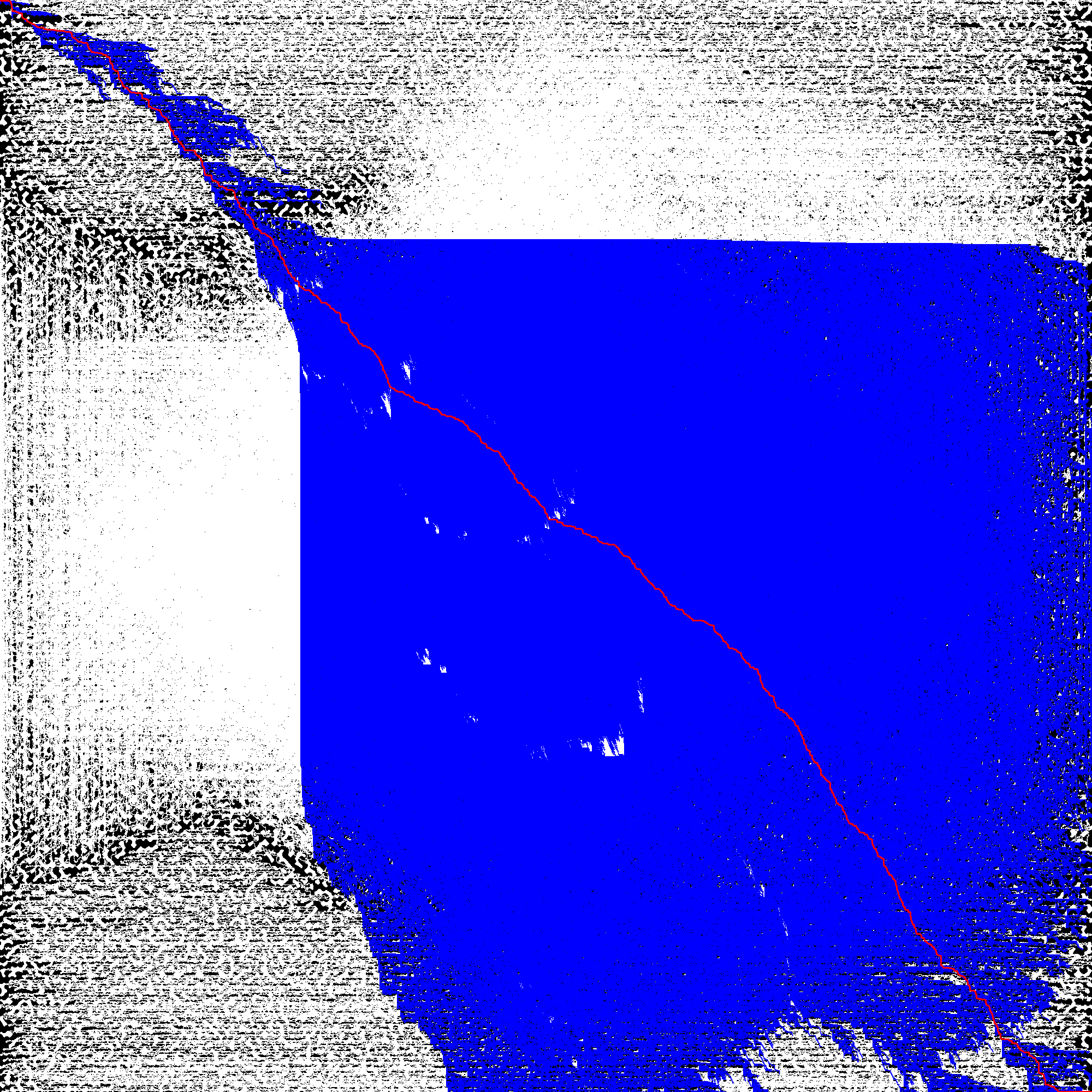


Figure 6 - A\* explores most of an empty graph, path is marked in red, expanded nodes are marked in blue

A picture containing text, screenshot, line, rectangle

Description automatically generated

Figure 7 - lookahead ignores huge sections of an empty graph and still generates an optimal path

In higher density maps, lookahead’s performance is not as stark, but pruning is able to remove many nodes as can be seen here:

A picture containing pixel, colorfulness, graphics, green

Description automatically generated A picture containing pattern, green, colorfulness, graphics

Description automatically generated

Figure 8 – pruning (illustrated in green) is very effective on dense graphs. In same cases (as on the left) it can even solve the maze on its own

**Pruning performance**

We used a noise threshold of -0.4, to make a very dense map, then we tested pruning time and number of generations required to finalize at various graph sizes and noise scales.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Graph size (wxh), scale | 2500, 0.1 | 5000, 0.1 | 10000, 0.1 | 2500, 0.01 | 5000, 0.01 | 10000, 0.01 |
| Runtime(s) | 2.35 | 0.47 | 0.1 | 0.06 | 0.36 |  |
| Generations | 88 | 101.6 | 110.2 | 634.6 | 654.8 |  |

**Discussion**

Lookahead provided consistently better performance than A\* in all contexts,

**Further optimizations**

Lookahead has the strong tendency to move in straight lines, it should be possible to give A\* some of this same tendency using the same logic, this would be interesting as it would give us some of the optimizations of lookahead with less of the overhead.

We talked about reducing memory usage by replacing the closed set array with a hashmap, we could also consider further reducing the number of heap pushes in lookahead by replacing the next variable with a stack, this would allow us to process all nodes whose heuristics are smaller rather than just one, resulting in less nodes pushed to the heap.

A common optimization in A\* pathfinding is to tiebreak node priority by lower h value (higher g value), this would make it reach the goal sooner while still generating optimal paths.

We could consider more advanced pruning rules that look at a larger area, for example, we could cast rays in all directions from a node until they hit a wall and prune the node if none of the empty tiles the rays visited have a wall on two opposite sides.

A picture containing symbol, pixel

Description automatically generated

Figure 9 - another possible pruning rule, if all blue nodes are empty

We could also consider more pre-processing steps, such as constructing a series of shortcut markers to mark different nodes.

A final optimization we could make would be to replace the heap with a list of stacks, a stack at index n would store nodes with f = n, this reduces the (amortized) time complexity of pushing and popping from O(log(n)) to O(1).

**References**

1. Harabor, D. and Grastien, A., 2011, August. Online graph pruning for pathfinding on grid maps. In *Proceedings of the AAAI Conference on Artificial Intelligence* (Vol. 25, No. 1, pp. 1114-1119).
2. Ericson, C. (n.d.). Aiding pathfinding with cellular automata. Retrieved from <https://realtimecollisiondetection.net/blog/?p=57>
3. "A\* search algorithm." In Wikipedia. Retrieved from [https://en.wikipedia.org/wiki/A\*\_search\_algorithm](https://en.wikipedia.org/wiki/A*_search_algorithm)