# 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 on various grid.

## 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.

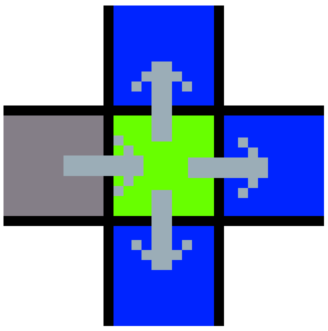


*Figure 1 - skipping the parent node*

2. 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.

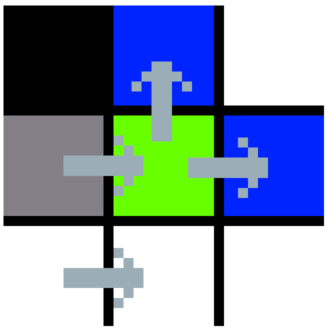


*Figure 2 - skipping a side node, the node below us does not need to be searched because the node to our lower left can search it later*

3. 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 $f_{current}$ was a minimum value of the heap, and for any child $f_{child} = g_{child} + h_{child} = (g_{current} + 1) + (h_{current} \pm 1) = f_{current} + 1 \pm 1$, so $f_{child} >= f_{current}$, so if we find a node such that $f_{child} = f_{current}$, 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 $2^d$ 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.

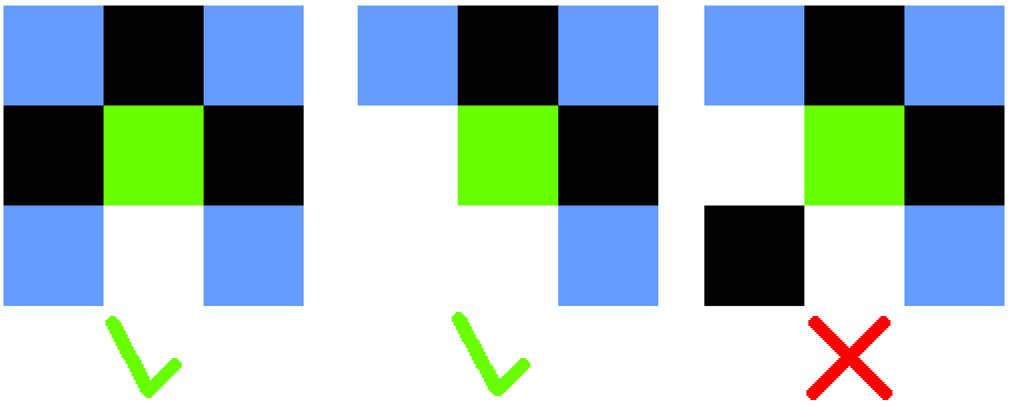


*Figure 3 - rules for pruning illustrated on a 2d grid, blue marks a cell as irrelevant, black marks a cell as a wall, we can prune the green cell in the left and middle cases but not the right case as it could block off an important passage*

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.

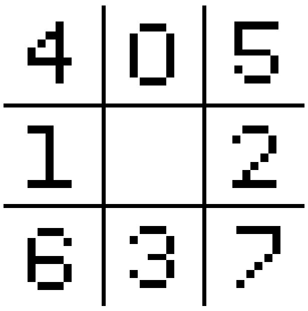
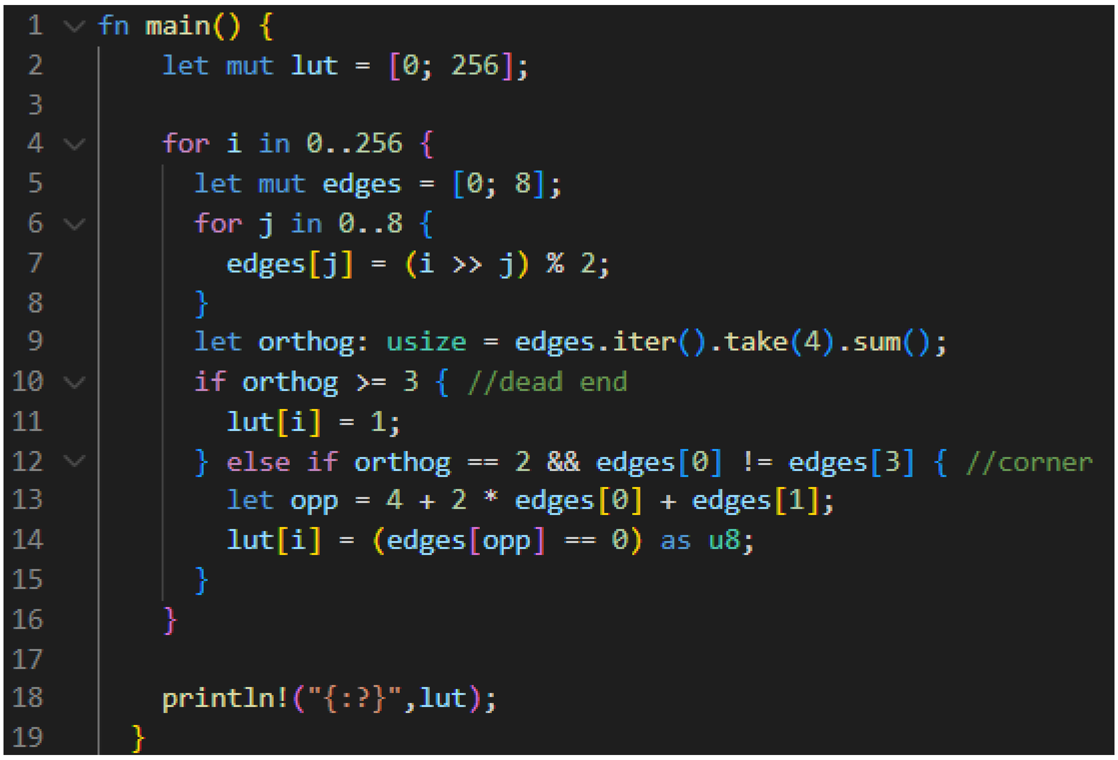


*Figure 4 - bit-indices of each adjacent cell*

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



```rust
fn main() {
    let mut lut = [0; 256];

    for i in 0..256 {
      let mut edges = [0; 8];
      for j in 0..8 {
        edges[j] = (i >> j) % 2;
      }
      let orthog: usize = edges.iter().take(4).sum();
      if orthog >= 3 { //dead end
        lut[i] = 1;
      } else if orthog == 2 && edges[0] != edges[3] { //corner
        let opp = 4 + 2 * edges[0] + edges[1];
        lut[i] = (edges[opp] == 0) as u8;
      }
    }

    println!("{:?}",lut);
  }
```

*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($b^d$) 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($d^n$) 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 n* 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 $O(d^{n^*}\log(d^{n^*}))$. 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 $O(d^{n^*})$, 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 d.

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 $V >> d^{n^*}\log(d^{n^*})$.

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.

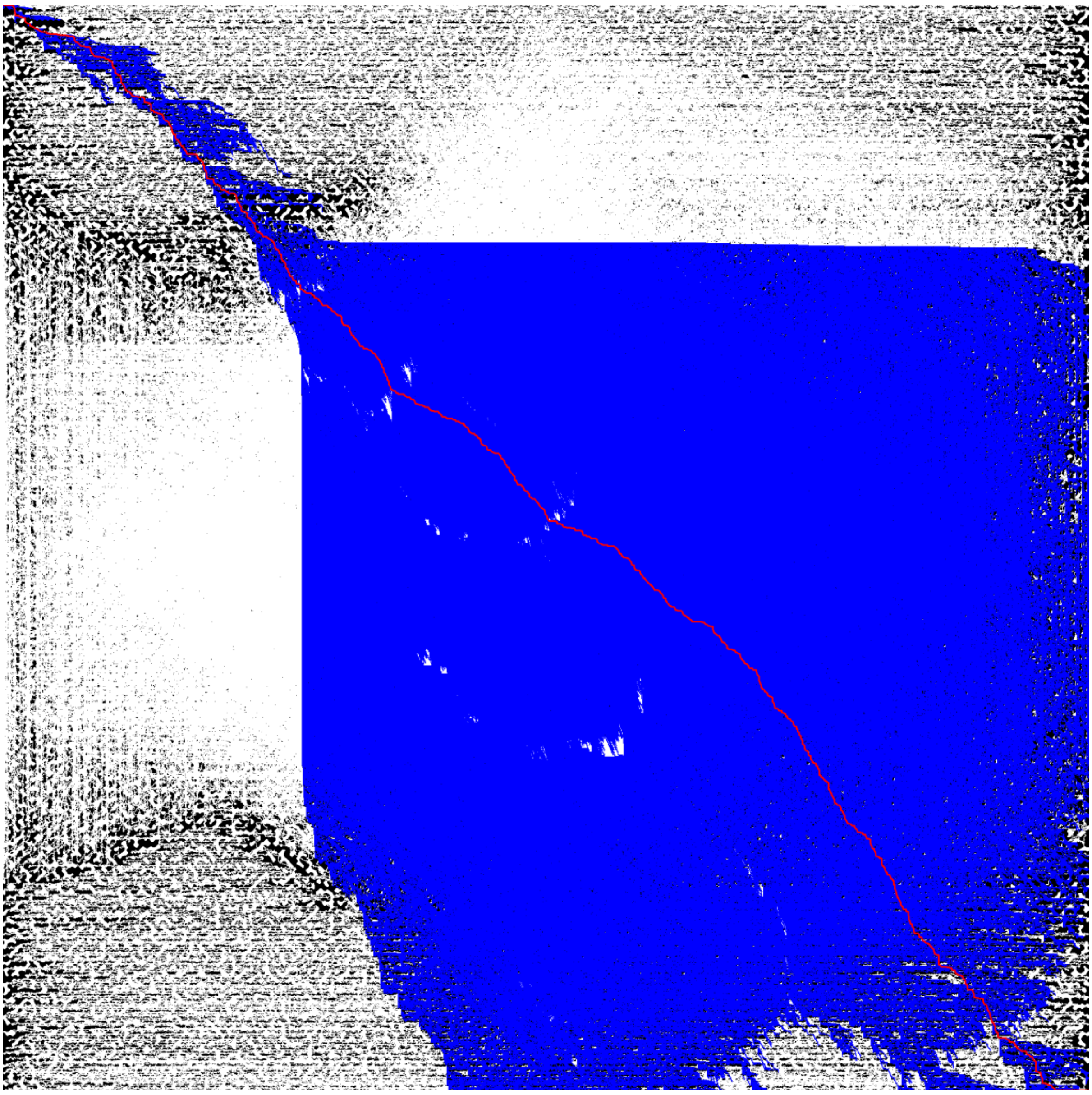


*Figure 6 - A* explores most of an empty graph, path is marked in red, expanded nodes are marked in blue*

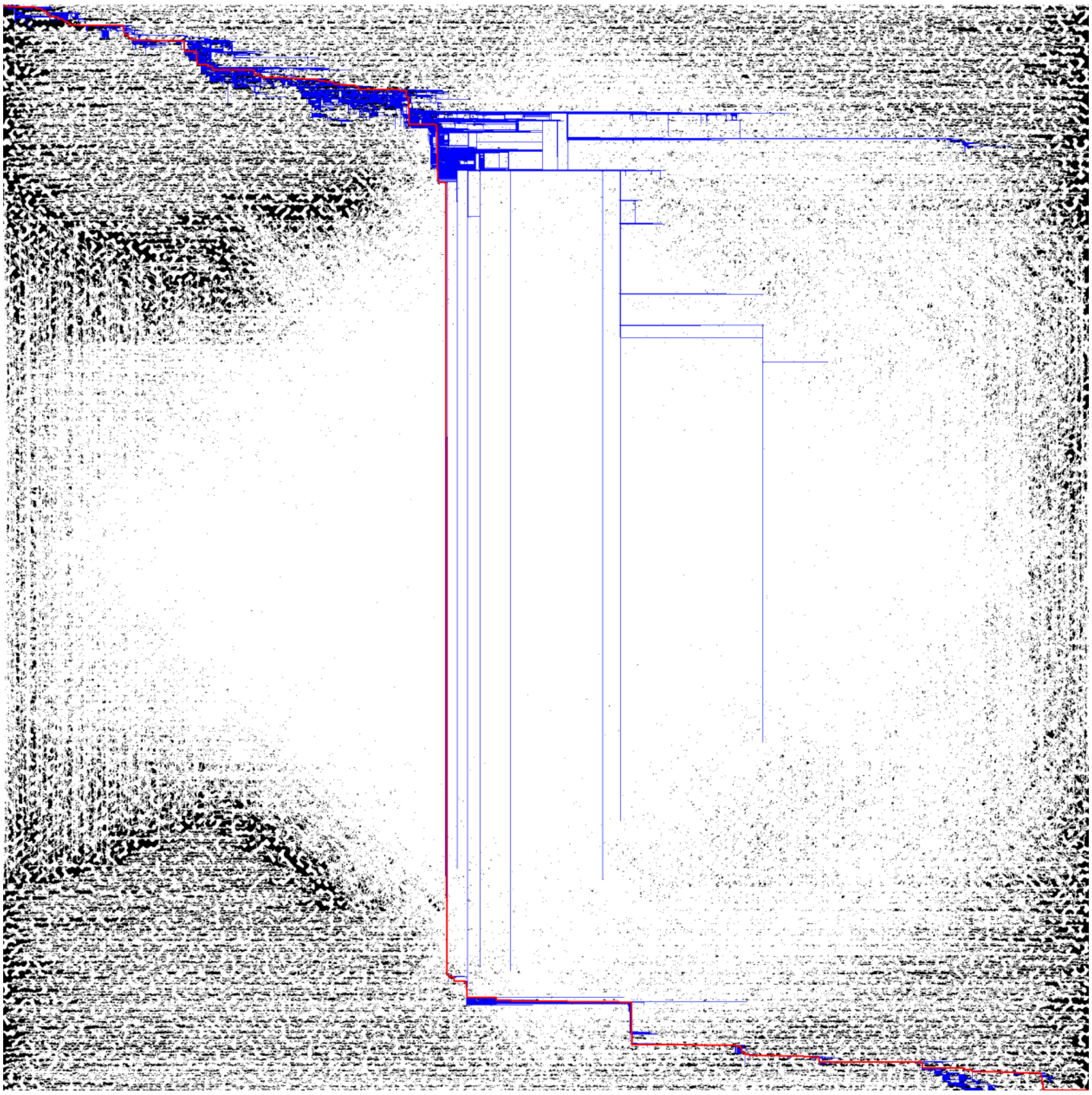


*Figure 7 - lookahead ignores huge sections of an empty graph and still generates an optimal path*

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

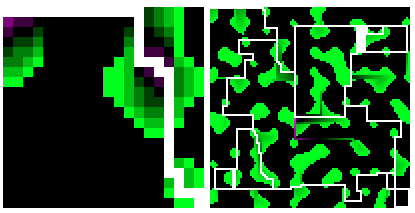


*Figure 8 – pruning (illustrated in green) is very effective on dense graphs. In same cases (as on the left) it can even solve the maze on its own*

**Pathfinding performance** We compared the performance of our algorithms on various graphs, averaged over 10 runs. We recorded the number of nodes discovered, the number of nodes pushed to the heap, the number of nodes expanded, and the largest the heap got during the algorithm's runtime.

| Parameters | A* | Lookahead | Pruned A* | Pruned Lookahead |
|---|---|---|---|---|
| Size: 1000x1000 Noise scale: 0.01 Noise Threshold: -0.4 | Runtime: 0.0115s Discovered: 4772.1 Pushed: 2179.4 Expanded: 1193.1 Max Heap: 379.0 | Runtime: 0.00505s Discovered: 3634.4 Pushed: 47.2 Expanded: 1211.5 Max Heap: 1.7 | Runtime: 0.00207s Discovered: 934.9 Pushed: 234.5 Expanded: 233.8 Max Heap: 0.8 | Runtime: 0.00036s Discovered: 700.7 Pushed: 7.6 Expanded: 233.6 Max Heap: 0.4 |
| Size: 5000x5000 Noise scale: 0.01 Noise Threshold: -0.4 | Runtime: 0.288s Discovered: 302778 Pushed: 145151 Expanded: 75694 Max Heap: 5800 | Runtime: 0.107s Discovered: 226985 Pushed: 11093 Expanded: 75661 Max Heap: 59 | Runtime: 0.0120s Discovered: 19559 Pushed: 5565 Expanded: 4889 Max Heap: 74 | Runtime: 0.00843s Discovered: 14654 Pushed: 866 Expanded: 4885 Max Heap: 5 |
| Size: 1000x1000 Noise scale: 0.01 Noise Threshold: 0 | Runtime: 0.0506s Discovered: 54582 Pushed: 27062.4 Expanded: 13645.7 Max Heap: 7077.3 | Runtime: 0.0157s Discovered: 31673 Pushed: 159.5 Expanded: 10557.7 Max Heap: 3.2 | Runtime: 0.00637s Discovered: 3333.3 Pushed: 1284.3 Expanded: 833.4 Max Heap: 211.3 | Runtime: 0.00183s Discovered: 926.6 Pushed: 1.8 Expanded: 308.9 Max Heap: 1.1 |
| Size: 5000x5000 Noise scale: 0.01 Noise Threshold: 0 | Runtime: 0.921s Discovered: 650761 Pushed: 321431 Expanded: 162690 Max Heap: 30926 | Runtime: 0.236s Discovered: 485632 Pushed: 4375 Expanded: 161877 Max Heap: 55 | Runtime: 0.078s Discovered: 63428 Pushed: 28750 Expanded: 15857 Max Heap: 3390 | Runtime: 0.026s Discovered: 46640 Pushed: 501 Expanded: 15546 Max Heap: 7 |
| Size: 20000x20000 Noise scale: 0.01 Noise Threshold: 0 | Runtime: 12.351s Discovered: 6967775 Pushed: 3448410 Expanded: 1741943 Max Heap: 457329 | Runtime: 3.905s Discovered: 5097715 Pushed: 116562 Expanded: 1699238 Max Heap: 444 | Runtime: 0.961s Discovered: 993017 Pushed: 464067 Expanded: 248254 Max Heap: 75178 | Runtime: 0.330s Discovered: 569790 Pushed: 6546 Expanded: 189930 Max Heap: 82 |
| Size: 1000x1000 Noise scale: 0.01 Noise Threshold: 0.4 | Runtime: 0.05283s Discovered:104740 Pushed: 52882.8 Expanded: 26185.1 Max Heap: 15003.3 | Runtime: 0.03565s Discovered:124803 Pushed: 177.2 Expanded: 41601.3 Max Heap: 4.2 | Runtime: 0.04298s Discovered: 86668 Pushed: 43639.2 Expanded: 21667.1 Max Heap: 12285.6 | Runtime: 0.00920s Discovered: 22878 Pushed: 62.5 Expanded: 7626.1 Max Heap: 1.2 |
| Size: 5000x5000 Noise scale: 0.01 Noise Threshold: 0.4 | Runtime: 1.550s Discovered:3245323 Pushed: 1623317 Expanded: 811330 Max Heap: 433123 | Runtime: 0.4586s Discovered:2351219 Pushed: 4005.3 Expanded:783740.0 Max Heap: 54.8 | Runtime: 1.290s Discovered:3498192 Pushed: 1747998.4 Expanded:874548.1 Max Heap:465230.6 | Runtime: 0.353s Discovered:1638462 Pushed: 2761.7 Expanded:546154.3 Max Heap: 31.2 |
| Size: 1000x1000 Noise scale: 0.05 Noise Threshold: 0 | Runtime: 0.0463s Discovered: 59858 Pushed: 28456 Expanded: 14964 Max Heap: 2695 | Runtime: 0.0145s Discovered: 45565 Pushed: 1119 Expanded: 15188 Max Heap: 47 | Runtime: 0.00595s Discovered: 12784 Pushed: 4966 Expanded: 3196 Max Heap: 445 | Runtime: 0.00254s Discovered: 9292 Pushed: 238 Expanded: 3097 Max Heap: 17 |
| Size: 5000x5000 Noise scale: 0.05 Noise Threshold: 0 | Runtime: 0.792s Discovered: 611844 Pushed: 289042 Expanded: 152961 Max Heap: 11621 | Runtime: 0.249s Discovered: 457679 Pushed: 15737 Expanded: 152560 Max Heap: 431 | Runtime: 0.088s Discovered: 114307 Pushed: 45021 Expanded: 28577 Max Heap: 1698 | Runtime: 0.038s Discovered: 85378 Pushed: 2943 Expanded: 28459 Max Heap: 78 |
| Size: 20000x20000 Noise scale: 0.05 Noise Threshold: 0 | Runtime: 15.824s Discovered:22611053 Pushed: 10661338 Expanded: 5652763 Max Heap: 79219 | Runtime: 5.849s Discovered:16924472 Pushed: 873358 Expanded: 5641491 Max Heap: 8331 | Runtime: 1.799s Discovered:3531610 Pushed: 1370952 Expanded: 882902 Max Heap: 13223 | Runtime: 0.910s Discovered:2664482 Pushed: 114625 Expanded: 888161 Max Heap: 905 |

**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, averaged over 10 runs.

| size, scale | 2500, 0.1 | 5000, 0.1 | 10000, 0.1 | 2500, 0.01 | 5000, 0.01 | 10000, 0.01 |
|---|---|---|---|---|---|---|
| Runtime(s) | 0.1 | 0.47 | 2.35 | 0.06 | 0.36 | 1.79 |
| Generations | 88 | 101.6 | 110.2 | 634.6 | 654.8 | 974.4 |

## Discussion

Our data showed lookahead provided consistently better performance than A* in all contexts, typically two to three times faster. Pruned lookahead still outperformed pruned A*, but the difference was not as pronounced. Pruning made a more dramatic difference, often able to speed up A* by a factor of 8 to 10, and lookahead by a factor of 5 to 8.

Comparing maps of different sizes showed that our beliefs about algorithmic complexity were mostly correct. Pruned algorithms performed slightly better at larger grid sizes, although this seems to be a quirk of our small grid sizes that diminishes when going to even larger sizes, rather than a trend. If we compare the number of nodes discovered, pushed, or expanded, we find a consistent trend, each doubling of the grid side leads to ~1.38 more nodes searched, this gives us an experimentally determined n* of 1.38, a significant improvement over n = 2. If we put this experimentally determined n* into our time complexity formula we get $O(d^{1.38}\log(d^{1.38}))$. This formula is within bounds of our observed runtimes.

With each quadrupling of graph size (double in both directions), prune time increased by about 5-fold, this is likely due to a decrease in cache coherency and an increase in pruneable nodes in larger grids, without these factors it would likely be linear with the size of the graph. On the grid sizes we tested, pruning generally took longer than pathing, but as grids grow larger, pathfinding should outpace pathfinding due to its larger time complexity, making pruning worthwhile even when not amortized or offloaded to the gpu.

## Further optimizations

Lookahead has the strong tendency to move in straight lines, it should be possible to give A* some of this same tendency by simple reorder of node checking, 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 hash map, 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.

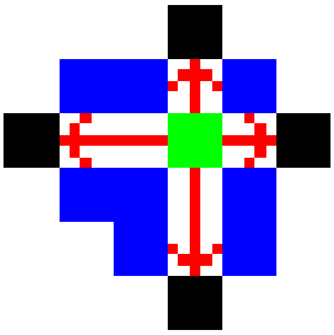


*Figure 9 - another possible pruning rule, if all blue nodes are empty*

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