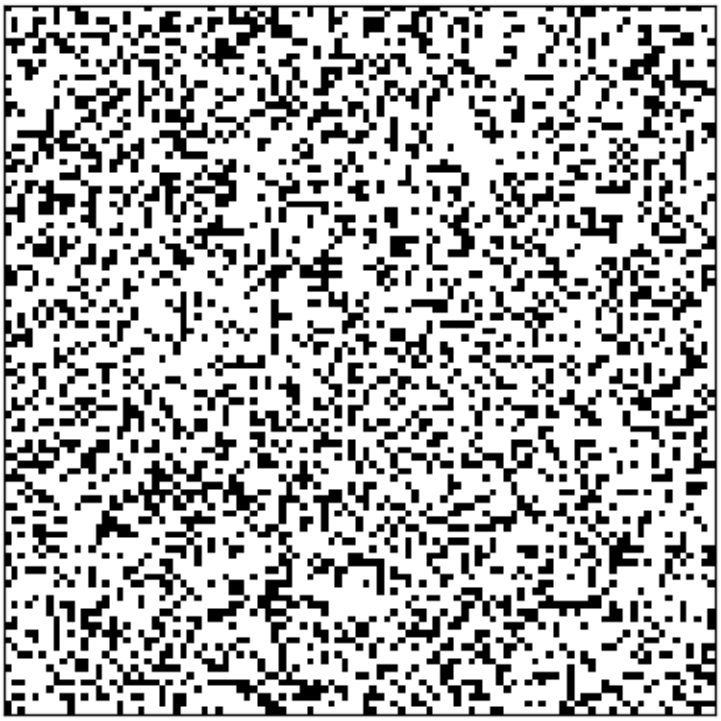
0 Setup Your Environments

We created a simple maze environment by randomly assigning each cell to either be unblocked (0) or blocked (1) using weighted probabilities. The following figure is an example of a grid world we generated.



1 Understanding the Methods

(a) The first move of the agent for the example search is to the east rather than north because it gives us a smaller heuristic. The Manhattan distance from E3 to T is 2, whereas the Manhattan distance from D2 to T is 4. Since they both have a cost of 1, the f value for E3 is smaller. It is not until the agent actually traverses to the E3 cell that it becomes aware that there is a wall located at E4, at which point it will run another A* search. In a finite grid world, there are a finite number of cells. Furthermore, if we do not consider blocked cells, every cell is reachable from any other cell. However, because of blocked cells, we divide the finite grid world into smaller finite regions, where each cell in a given region can access every other cell in that region. When running any A* search, a closed list of expanded cells is kept, so no cell can be expanded twice. Therefore, in a given A* search, every cell in the same region as the agent will eventually be expanded if the target is not reached. If the agent and the target are in the same region, then the agent will find the path to the target in finite time. If the agent and the target are not in the same region, then the agent will expand the finite number of unblocked cells in its region and discover it is impossible to reach the target.

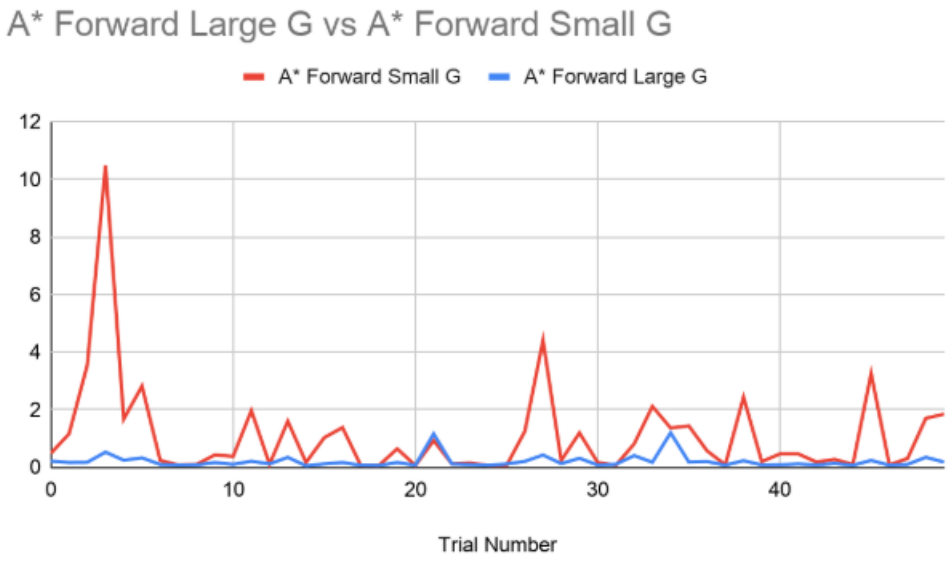
The way Repeated A* works is that the blocked cells are slowly discovered after every iteration of A*. Every iteration will end in either blocked cells being discovered or the target being reached. Since there are finite number of blocked cells in a finite grid world, all the blocked cells surrounding the region the agent is in will be discovered in finite time, and thus the extent of the region the agent is in will be known, and thus, by the above logic, the agent will either reach the target or discover that it is impossible in finite time.

(b) *Proof.* We know that the total number of moves the agent will make is

equal to the total number of times A* is called (call it t) multiplied by the total number of moves for each A* call (call it m). Each time A* is called, it moves to a new unblocked cell. Therefore, the number of times A* is called is bounded above by the total number of unblocked cells. Furthermore, since A* never expands a given cell twice, so the number of moves it makes in each A* call is bounded above the total number of unblocked cells.

Thus, if the number of unblocked cells is n, we know that $t \leq n$ and $m \leq n$, so $t \times m \leq n \times n$, as desired □

2 The Effects of Ties



The above chart shows our data for comparing a tiebreaker which favors large g-values with a tiebreaker which favors small g-values. We used runtime in seconds as a measurement. The red line represents the tiebreaker which favors a smaller g-value and is consistently greater than the tiebreaker which favors a larger g-value.

The reason behind this can be seen by looking at the following figure. If Repeated Forward A* favors a smaller g-values value then it will expand the diagonal cells in order, given that there are no obstacles in the grid like in Figure 9. If a larger g-values is favored then it will expand cells in the order of how it would traverse using the Manhattan Distance. In the case of an unblocked grid such as Figure 9, the larger g-values favor works out better because it will expand less nodes as opposed to smaller g-values value which will expand each single one. A larger g-values would be much faster for Repeated Forward A* as opposed to a smaller g-values favor which will expand much more nodes.

Figure 9: Third Example Search Problem

small g(x) favor

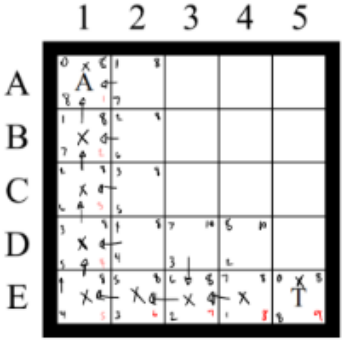
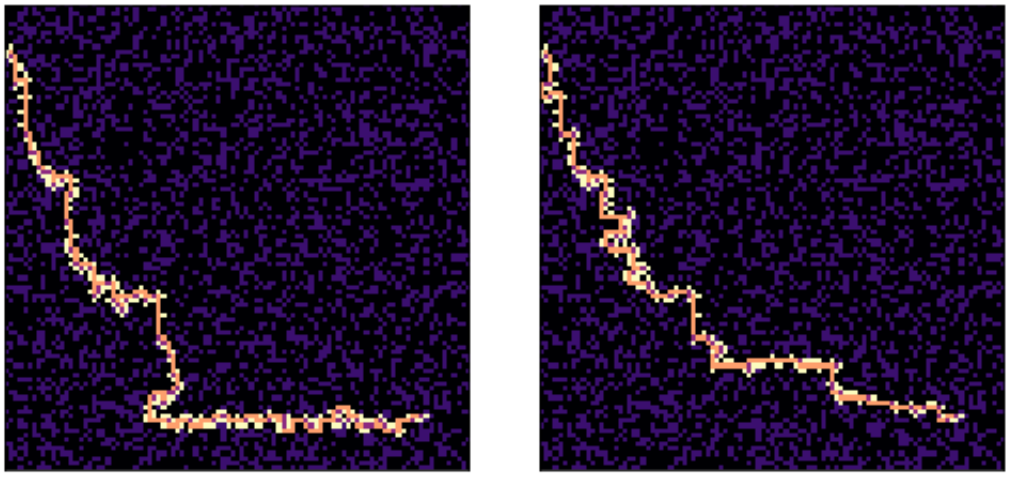


Figure 9: Third Example Search Problem

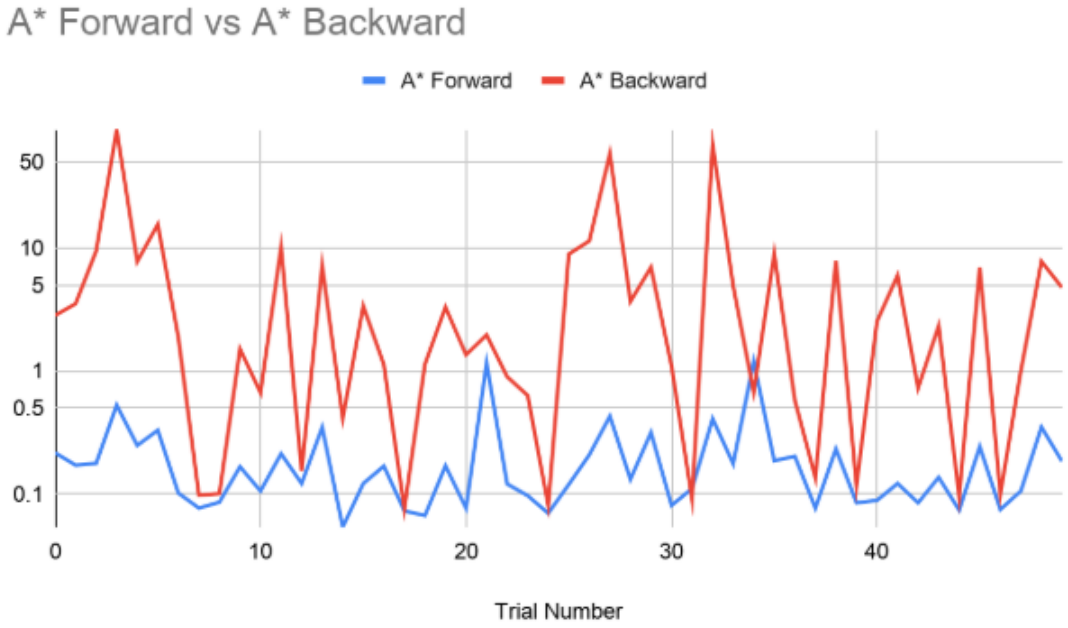
large g(x) favor

Figure 1: Example of A*Forward Small G Path vs A*Forward Large G Path



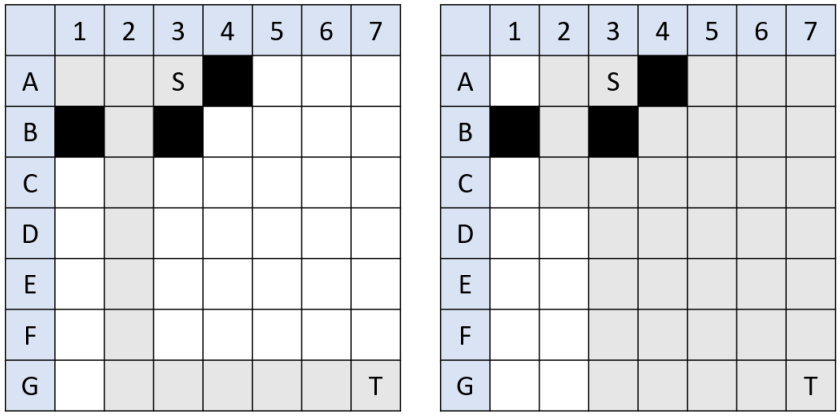
The images above show the difference of using either a small g-value or large g-value tiebreaker. A small g-value tiebreaker expands more cells on average than a large g-value tiebreaker.

3 Forward vs. Backward



The above chart shows our data for comparing Repeated Forward A* and Re-

peated Backward A*. The chart compares them using runtime in log(seconds). We used logarithms because the Repeated Backward A* was consistently significantly slower than Repeated Forward A*. We could explain Repeated Backward A*'s slower run speed as a result of it having to re-expand many more nodes whenever it encounters a wall on its journey to finding a path to the start node. If we are starting the A* search near the start point, then we can begin to formulate a path around them much faster than if the A* search started near the goal point.



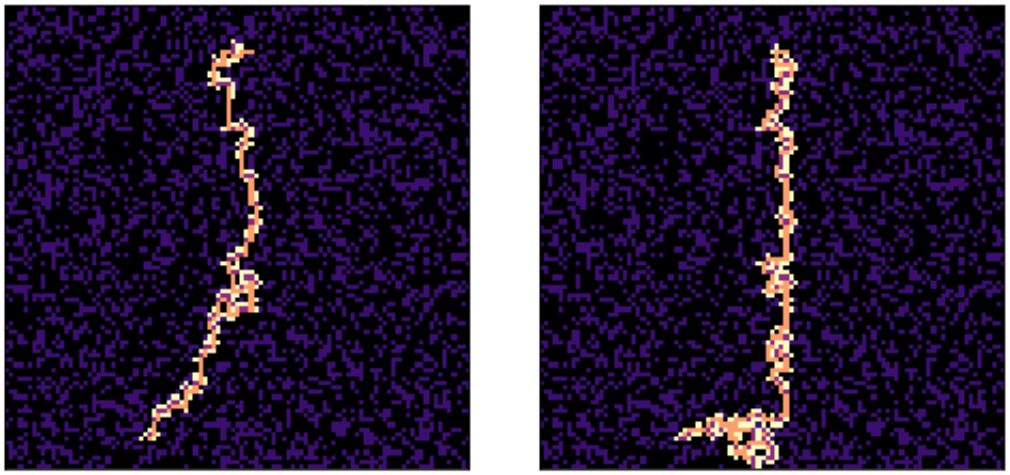
. Forward A* Backward A* .

In the diagram above, the nodes that Forward A* will expand is shown on the left while the nodes that Backward A* will expand is shown on the right. Forward A* will expand way less nodes than Backward A* because it will see the blocked cells first as opposed to Backward A* that will expand many nodes on its initial path based on the Manhattan Distance. Backward A* will start expanding from its position at G7 going upwards all the way to A7 and then expand more nodes going left until it expands A5 and realizes that there is a wall at A4. It will then start expanding from B6 until it reaches and expands B4 where it sees a wall again at B3. Backward A* will then try to expand C6, and once it expands C3, it will see that C2 has a higher

f-value, leading it to expand every node that remains not expanded from columns 3 to 7 because they will all have a smaller f-value (actually having an f-value of 10 based on the Manhattan Distance where they all just don't account for the blocked cells on the way to the start node) than anything in column 2. Finally, it will expand C2 and find a viable path that will lead it to A3 where the start node is located.

Now if a Backward A* was performed on a grid of 101 x 101 cells with many more blocked cells as the project requires, it is no wonder that it will perform significantly worse than Forward A* in most occasions.

Figure 2: Example of A*Forward Large G Path vs A*Backward Large G Path



The images above show the different paths taken by A*Forward and A*Backward.

4 Heuristics in the Adaptive A*

Proof. To prove that Manhattan distances are consistent in grid worlds in which the agent can move only in four main conditions we must define a

couple of things: A consistent heuristic is heuristic such that:

$$h(s) \leq c(s, a, s') + h(s')$$

and

$$h(g) = 0$$

where s is the parent node and $c(s, a, s')$ is the action cost of travelling from node s to node s'.

The Manhattan Distance formula is:

$$h(s) = |X_s - X_{sgoal}| + |Y_s - Y_{sgoal}|$$

where X denotes the index of the rows, Y is the index of the columns, s is the current state of the agent, and $sgoal$ is the goal state. We first consider when $s = sgoal$. In this case, $h(s) = |X_{sgoal} - X_{sgoal}| + |Y_{sgoal} - Y_{sgoal}| = 0$, which aligns with our definition of a consistent heuristic. Next, we consider an arbitrary s. Suppose s and s' are neighboring cells in the grid world. Thus, we know that

$$h(s) - h(s') = (|X_s - X_{sgoal}| + |Y_s - Y_{sgoal}|) - (|X_{s'} - X_{sgoal}| + |Y_{s'} - Y_{sgoal}|)$$

Since, s and s' are directly, adjacent, we know that either $X_s = X_{s'}$ or $Y_s = Y_{s'}$ (but not both). Without loss of generality, assume that $X_s = X_{s'}$, so $|Y_s - Y_{s'}| = 1$ because the two cells are directly adjacent. Thus,

$$h(s) - h(s') = |Y_s - Y_{sgoal}| - |Y_{s'} - Y_{sgoal}| \leq |Y_s - Y_{sgoal} - Y_{s'} + Y_{sgoal}|$$

$$= |Y_s - Y_{s'}| = 1 = c(s, a, s')$$

Which means that $h(s) \leq h(s') + c(s, a, s')$, as desired. □

Proof. Next it is time to address the Adaptive Heuristics. The Adaptive A*

updates heuristics based on the previous A* search. The new heuristic is generated by the following formula:

$$h_{new}(s) = g(sgoal) - g(s)$$

Where the $g - values$ are from the previous iteration of A*. We know that by definition of h_{new}, $h_{new}(sgoal) = g(sgoal) - g(sgoal) = 0$ which satisfies one of the requirements of a consistent heuristic. In order to address the other condition, we proceed by cases.

Case 1: s and s' have both been expanded in the previous A* search.
By definition of h_{new} we know that

$$h_{new}(s) = g(sgoal) - g(s)$$

$$h_{new}(s) = g(sgoal) - g(s')$$

We also know that $g(s') - g(s) \leq c(s, a, s')$. This statement is true because the two nodes are adjacent and the action cost between them is 1 which is the smallest possible action cost between any two arbitrary nodes in the grid world. Therefore, any A* search which expands both of these nodes will likely expand one as the parent of the other, in which case the difference of their path costs would be 1. If they were not parents, then their path costs would not differ by more than 1 because otherwise being each others parent would have a lower path cost. We now manipulate this statement. First we subtract $g(s')$ from both sides:

$$-g(s) \leq -g(s') + c(s, a, s')$$

Then we add $g(sgoal)$ to both sides:

$$g(sgoal) - g(s) \leq g(sgoal) - g(s') + c(s, a, s')$$

We can then substitute using the equations for $hnew(s)$ and $hnew(s')$ we defined earlier

$$hnew(s) \leq hnew(s') + c(s, a, s')$$

which satisfied the definition of a consistent heuristic, as desired.

Case 2: s had been expanded in the previous A* but s' had not.
We know that

$$h_new(s) = g(Sgoal) - g(s)$$

and $h(s')$ is the Manhattan distance from it to the goal.
We know that s' was visited in the previous A* search because it is adjacent to a node which was expanded, namely s. Since it was visited but not expanded, its f-value must be greater than or equal to the f-value of any node which was expanded in the previous search (because A* prioritizes the expansion of smaller f-values). We also know that $h(sgoal) = 0$ (which we proved above), so $f(sgoal) = g(sgoal) + 0 = g(sgoal)$. Since $sgoal$ was expanded in the previous search, we know that $f(s') \geq f(sgoal)$, and thus $f(s') \geq g(sgoal)$. Using the same logic from case 1, we also know that $-g(s) \leq -g(s') + c(s, a, s')$, so we can add these two inequalities together to yield

$$g(sgoal) - g(s) \leq f(s') - g(s') + c(s, a, s')$$

By definition of f-value, $f(s') = g(s') + h(s')$, so $h(s') = f(s') - g(s')$. We substitute to get

$$g(sgoal) - g(s) \leq h(s') + c(s, a, s')$$

By definition of $h_{new}(s)$, $h_{new}(s) = g(sgoal) - g(s)$, so we substitute to get

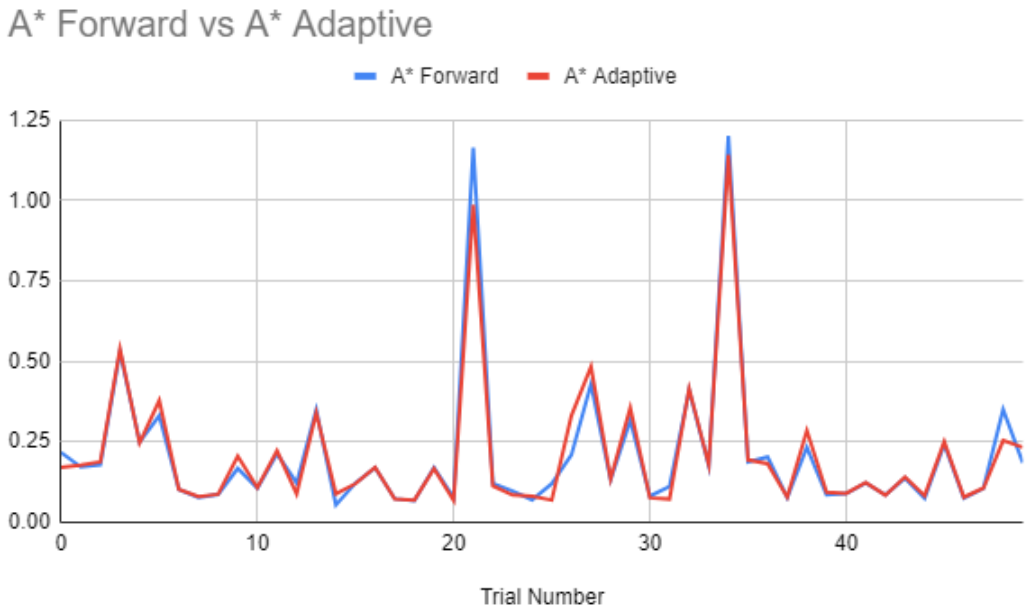
$$h_{new}(s) \leq h(s') + c(s, a, s')$$

which satisfies the definition of a consistent heuristic, as desired.

Case 3: Neither s nor s' have been expanded previously
This will not use the new heuristic because it has not been expanded and the grid has stayed the same or no solution was found. It will still use the old heuristic based on the Manhattan Distance which we have proved is consistent.

These three cases are sufficient to prove that the heuristics in the adaptive A* are consistent, as desired. □

5 Heuristics in the Adaptive A*

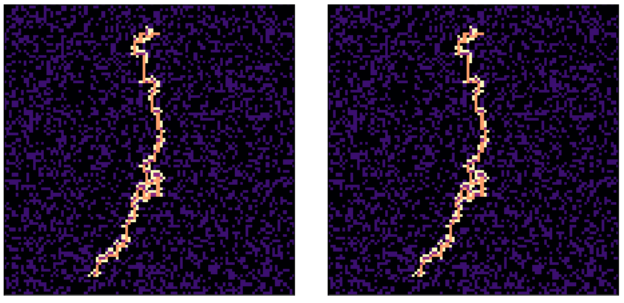


The above chart shows our data for comparing Repeated Forward A* and Adaptive A*. The chart compares them using run time in seconds. The two implementations ran in extremely similar times, with Adaptive A* having the slight edge in a few cases (especially the two times when the run time was over a second). With data so similar, we suspect that some of the fluctuations may also have been due to simply the CPU's memory usage at

a particular moment. We also noticed that more often than not, Adaptive A* would run ComputePath fewer times than Forward A*, but we suspect that because of the additional for loop that Adaptive A* uses to update the heuristics, even though the algorithm may be more efficient the run time may have been slightly increased due to this additional requirement.

The reason that Adaptive A* is slightly faster is because after each time it runs ComputePath, the heuristics of all the cells that were expanded are updated to a more accurate, but still increased heuristic. Rather than using the Manhattan distance as the heuristic, the program uses the distance it would have taken for the agent to get to the goal from the cell given its current knowledge of the blocked cells. For example if there is a cell which is at the end of a long "corridor" then the only possible path from the cell to the goal may not be the simple Manhattan distance, but rather would likely involve exiting through the long corridor. Therefore, the new heuristic would take that lengthy exit into account, and it would be far less likely for the A* search to attempt going down the corridor. The reason that Adaptive A* may not be as effective for us is because the grid worlds we generated had random blocked cells, meaning fewer walls and blocked off sections and more mini obstacles the agent would need to work around.

Figure 3: Example of A*Forward Path vs Adaptive A*Forward Path



The images above show the paths taken by A*Forward and Adaptive A* Forward when using a large g-value tiebreaker. They have extremely similar paths.

6 Memory Issues

So in order to reduce memory consumption, we could theoretically minimize the amount of data which needs to be stored per cell. Our implementation stores a lot of data in each cell in order to make the code simpler, but in order to conduct an A* search, we really only need a g-value, an h-value, a parent indicator, and a search value (which is needed to maintain a level of efficiency in the A* search). Therefore, we should minimize the size of each of these pieces of data. The parent indicator only needs to be 2 bits, because there are only 4 directions the parent could be in. For the other 3 values, they are all bounded above by the number of cells in the gridworld, so we can store them as data points which are $\log_2(w)$ rounded up bits long, where w is the number of cells in the grid world.

When considering a 1001×1001 grid world, there are 1002001 cells. $\log_2(w)$ rounds up to 20, so our three values can each be 20 bits, thus we need 62 bits for each cell.

$$62 \text{ bits} \times 1002001 = 62124062 \text{ bits} = 7.7765507.75 \text{ Bytes} = 7.766 \text{ MB}$$

If we are limited by 4MB (32 million bits) we must assess how many cells we could handle. The size of a cell could be found using the equation

$$w(3 \log_2 w + 2) \leq 3.2e7$$

This inequality results in a maximal whole number w of $w = 541118$ However, we must round down to one below the nearest power of 2 because we look for the ceiling of the $\log_2(w)$, so we get $2^{19} = 524288$. Thus, with 4MB we can handle a grid world that contains 524,287 cells.