COMP3600 Final Project M3

Jackson Kilrain-Mottram (u6940136)

November 2020

1 Description

This project will build a ray casting implementation, with the intention of extending it beyond the raw single colour, basic rectangular polygon fills. Texturing will be added to allow for more comprehensive rendering akin to that of the id tech 1 engine that Wolfenstein 3D is built on. Additionally, path solving will be implemented with the A* path finding algorithm in order to find the most efficient path from a starting point to an end point in a level, assisting the player in fast traversals of a given level.

2 Functionalities

2.1 Hash table

As the raycaster is handling all rendering functionality, this includes loading, referencing and retrieving textures. In order to handle the textures efficiently, rather than storing a full instance of them on each wall instance, a reference is stored in the form of the texture name. This is then used to retrieve the texture instance via a hash table matching a low length string to a texture instance.

In this instance, we will be using a string of maximum length 32, allowing for amortized O(1) hashing as per the usual implementation of a hashmap/hashtable. It's important to note that only standard non-extended ASCII values as described by the default char data type in C/C++ on an english locale. In order to reduce collisions with low-sized tables, a modified version of rolling polynomial and FNV-1a hashing is used. The standard implementation of rolling polynomial uses continuous offsets as a factor as a prime power.

2.2 A* path finding

A large part of the project is using the A* algorithm and extending the base heuristic to account for polygons in the range of the player frustum. In order to do this, a base implementation of the A* path finding algorithm needs to be used. The A* algorithm modifies Dijkstra's path finding to use not only forward (current location relative to end target) heuristic but also the reverse relative heuristic from current location to starting location. This simple idea is expressed through the equation , where G is the path $F = G + H$ distance between the current position and the starting position, H is the estimated distance between the current position and the end position and F is the total cost, the sum of the two.This simple formula allows for relative evaluation at every note making choosing the next position to try much easier since the F cost scales almost inversely when moving away from the target.

Using the A* algorithm in this project is a fairly straightforward choice, notably because it works well with graph style data. In this case the graph is grid aligned fixed width and height squares. Something to note is that this uses conditional neighbouring and not Manhattan distance 1neighbours. This is the case as it allows for diagonal movement and the conditional part allows for evaluation of relative wall position in the squares around, preventing movement between two diagonally connected walls. As the map is a grid structure, this will require adjusting the heuristic to work on a constant connectivity basis with a slightly higher bias for diagonal traversals as they are slightly longer.

Figure 1 provides an example of this conditional invalidation for diagonal movement compared to normal movement availability:

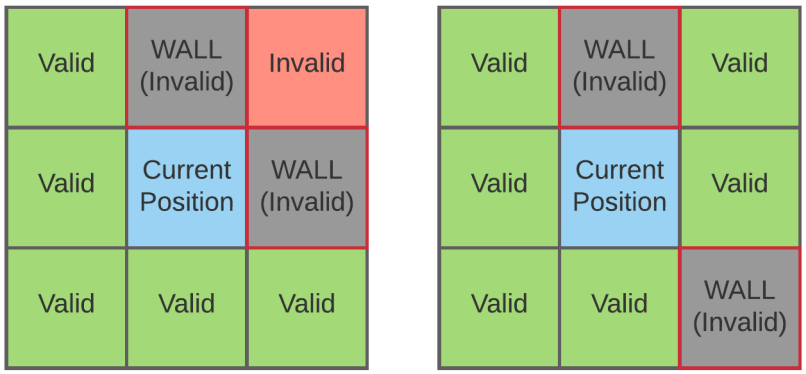


Fig (1): Examples of neighbour configurations with diagonal invalidation

2.3 Min heap

In order to facilitate the finding of the shortest route within a map, the A* algorithm depends on a minimum first priority queue. Min heaps employ a structure that guarantees the smallest element within the tree will always be at the root node, thus providing the correct circumstance for a minimum first priority queue.

3 Theoretical Time Analysis

3.1 Hash Table

Before performing an analysis on the hash table implementation, we first need a few assumptions and statements about the string keys. Note that we using bounded strings of maximum length 32. Additionally, we will assume uniform distribution of possible string values as each character has equal probability of occurring in the strings and also each length of string has equal probability of occurring.

In order to reduce the overall redundancy between the average cases of **insert**(), **get**() and **remove**(), I will provide the average case proof for identifying whether the given key exists or not in $\mathcal{O}(1+\alpha)$ time below and then each of the analysis of the average case for each function will be based on the specific operations performed as a result of finding or not finding the element.

Lemma 1: Average Case of Identifying Element Existance

As we have bounded length strings (max 32), we can assume uniform distribution of values as:

$$\forall v \in [0 \cdots -1] \quad \mathbf{P}(h(k) = v) = \frac{1}{m}$$

As it is uniformly distributed, the collision probability is:

$$\mathbf{P}_{k_1 \neq k_2}\{h(k_1) = h(k_2)\} = \frac{1}{m}$$

Using the above, we can show that the time complexity for searching for a given key, is on average $\mathcal{O}(1+\alpha)$ where $\alpha = \frac{n}{m}$. In order to prove the above time complexity, we first prove the two cases:

- When the key has been found
- When the key has not been found

For both cases, we will assume that hashing a key takes $\mathcal{O}(1)$ time.

1. Key found

In the case that a key has been found to match a given entry, then the analysis is on the linked list associated with that bucket. Here the average number of elements to search before finding the exact entry in the list with index $h(k)$ is $\frac{n}{m}$. As such the average time can be expressed as $\mathcal{O}(1 + \frac{n}{m}) = \mathcal{O}(1 + \alpha)$

2. Key not found

In the case of the key having been found, the average number of elements to check prior to the (k, v) being found is the same as the average amount of elements added to the hash table with identical hash values. In other words is the

average amount of elements in a linked list at an arbitrary bucket.

We can define a indicator random variable to analyse this probability. We define it as follows:

$$X_{ij} = \begin{cases} 1 & h(k_i) = h(k_j) \\ 0 & \text{otherwise} \end{cases}$$

Computing the collision probability is done by taking the expectation value of X_{ij}, resulting in:

$$E\left[X_{ij}\right] = \mathbf{P}\left(X_{ij} = 1\right) = \frac{1}{m}$$

From here we define the average number of elements in a list based on hash collision as a nested sum where the outer sum expresses the existing keys and the inner sum the keys to match:

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} X_{ij}\right)\right]$$

Applying the linearity of expectation, we can bring it to the inner sum:

$$= \frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} E\left[X_{ij}\right]\right)$$

$$= \frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} \frac{1}{m}\right)$$

Equating the outer sum, we get n:

$$= \frac{1}{n} \cdot n + \frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n} \frac{1}{m}$$

$$= 1 + \frac{1}{n \cdot m}\sum_{i=1}^{n}\sum_{j=i+1}^{n} 1$$

Here we can simplify the inner sum as the total iterations will start at $i + 1$ and end at n, totaling $n - 1$ iterations

$$= 1 + \frac{1}{n \cdot m}\sum_{i=1}^{n}(n - 1)$$

Using the linearity of summations and commutative behaviour of minus, we can split this into two separate sums over each term:

$$= 1 + \frac{1}{n \cdot m}\left(\sum_{i=1}^{n} n - \sum_{i=1}^{n} 1\right)$$

$$= 1 + \frac{1}{n \cdot m}\left(n^2 - \frac{n(n+1)}{2}\right)$$

$$= 1 + \frac{2n - (n+1)}{2m}$$

$$= 1 + \frac{n}{2m} - \frac{1}{2m}$$

$$= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}$$

Now we have an expression for the average items in a given linked list, we can use this in conjuction with the $\mathcal{O}(1)$ required by the hash function:

$$\mathcal{O}(1) + \mathcal{O}\left(1 + \frac{\alpha}{2} - \frac{\alpha}{2n}\right)$$

$$= \mathcal{O}(1 + \alpha)$$

As we can see, in both cases of the key being found or not found, the resulting time complexity resolves to $\mathcal{O}(1 + \alpha)$, in the average case

–END LEMMA–

3.1.1 Insertion

Best Case

In order to show the best case is $\mathcal{O}(1)$, we first need to establish that the hash function is asymptotically $\mathcal{O}(1)$. In the case of the hash function used in this implementation, the strings are of a length ≤ 32. As such we can say that hashing any given satisfying string will always be 32:

- Lines $193 - 196$ are of complexity $\mathcal{O}(1)$
- Lines $197 - 200$ will always have a fixed number of iterations between $[0 \dots 32]$ and as such will be a constant number of operations upper bounded in the best case as $\mathcal{O}(32) = \mathcal{O}(1)$
- Line 201 is of complexity $\mathcal{O}(1)$

From this analysis, we can conclude that in the upper-bounded best case for hashing this will be of time complexity:

$$3 \times \mathcal{O}(1) = \mathcal{O}(1)$$

Now, we can analyse the **insert**() method:

- Line 136 is a call to the **validateKeySize**() function which is a series of $\mathcal{O}(1)$ statements, and as such is $\mathcal{O}(1)$ asymptotically
- Lines $138 - 140$ are all $\mathcal{O}(1)$ assignments
- Line 142 is a call to the **getSequentialNonNull**() function will execute a single loop cycle since upon evaluating the loop after the first iteration the value of **bucket** will be a nullptr, therefore this will be $\mathcal{O}(1)$
- Lines $144 - 150$ are all $\mathcal{O}(1)$ assignments or checks
- Line 151 is a call to the **processBucketLinking**() function which will be $\mathcal{O}(1)$ since the provided **prev** argument will be a nullptr, immediately assigning the value to the bucket location in the table.
- Line 152 is of complexity $\mathcal{O}(1)$

Using the above analysis, we can see that this equates to the following expression:

$$\begin{aligned} &\mathcal{O}(1) + \mathcal{O}(1) + \mathcal{O}(1) + \mathcal{O}(1) + \mathcal{O}(1) + \mathcal{O}(1) \\ &= 6 * \mathcal{O}(1) \\ &= \mathcal{O}(1) \end{aligned}$$

Therefore in the best case scenario the **insert**() method will have time complexity of $\mathcal{O}(1)$

Average Case

Using the above proof for the average case of identifying the existence of a given element, at this point, the last operation to perform is to overwrite the found value or insert into an empty place, designates by the nullptr returned from the **getSequentialNonNull**() call. In either case this is an $\mathcal{O}(1)$ operation.

As we can see, in both cases of the key being found or not found, the resulting time complexity resolves to $\mathcal{O}(1 + \alpha)$, which is the overall time complexity of the **insert**() function in the average case

Worst Case

Analysing the worst case of insertion takes place with the entire table being filled bar a single slot, however all elements are in the same linked list. This implies that all elements are hashed to the same slot.

Let our element to be placed in the table be designated as v with key k, we can say the the existing list will be of length $n - 1$. In this regard the insertion algorithm will retrieve the linked list at index $h(k)$. Next a call to the **getSequentialNonNull**() will result in a total of n inerations, as it loops over all $n - 1$ elements in the list include the last nullptr, totalling n operations of complexity $\mathcal{O}(1)$:

$$\begin{aligned} &1 + \sum_{i=1}^{n} \mathcal{O}(1) \\ &= \mathcal{O}(n) \end{aligned}$$

Lastly the element will be inserted at the location will the nullptr, which is a constant time operation. Asymptotically, this does not change the existing $\mathcal{O}(n)$ complexity, and as such will remain as the final complexity.

Therefore, the time complexity of inserting an element in the worst case is $\mathcal{O}(n)$.

3.1.2 Retrieval

Best Case

With the understanding that string hashing is indeed $\mathcal{O}(1)$, we can use this in the analysis of the **get**() function. For this analysis, the best case is one where there is a single element in the hash table matching exactly that which we are looking for.

1. Lines $98 - 100$ are of complexity $\mathcal{O}(1)$ as they are Boolean checks and a error throw
2. Line 102 makes a call to **validateKeySize**() which is trivially $\mathcal{O}(1)$ by the same justification as lines $98 - 100$
3. Lines $104 - 105$ are $\mathcal{O}(1)$ since the first is a hash calculation which we already know is constant time and the second is vector index retrieval which is garuanteed to be constant time.
4. Lines $107 - 108$ will only execute once as the while loop initial condition is fulfilled and the **if** condition immediately satisfies when matching the keys.
5. Lines $109 - 110$ will also be constant as they are an assignment and return respectively. This is trivially $\mathcal{O}(1)$ by definition.

Combining the above time complexities into a single expression equates to:

$$2 \cdot \mathcal{O}(1) + \mathcal{O}(1) + 2 \cdot \mathcal{O}(1) + 2 \cdot \mathcal{O}(1) + 2 \cdot \mathcal{O}(1)$$

$$= \mathcal{O}(1)$$

Therefore the time complexity of retrieval in the best case is $\mathcal{O}(1)$

Average Case

Using the above proof for the average case of identifying the existence of a given element, the last operation is to return the found value, which is an $\mathcal{O}(1)$ operation, resulting in no overall change to the time complexity.

As we can see, in both cases of the key being found or not found, the resulting time complexity resolves to $\mathcal{O}(1 + \alpha)$, which is the overall time complexity of the **get**() function in the average case

Worst Case

Analysing the worst case of retrieval takes place with the entire table being filled, where all elements are hashed to the same slot. The element that is being sought after to retrieve is in the last slot of this linked list.

Let our element to be placed in the table be designated as v with key k, we can say the the existing list will be of length n. In this regard the insertion algorithm will retrieve the linked list at index $h(k)$. Next a call to the **getSequentialNonNull**() will result in a total of n inerations, as it loops over all n elements in the list, totalling n operations of complexity $\mathcal{O}(1)$:

$$1 + \sum_{i=1}^{n} \mathcal{O}(1)$$

$$= \mathcal{O}(n)$$

Lastly the element in the last last slot will be returned as a reference, which is a constant time operations. Asymptotically, this does not change the existing $\mathcal{O}(n)$ complexity, and as such will remain as the final complexity.

Therefore, the time complexity of retrieving an element in the worst case is $\mathcal{O}(n)$.

3.1.3 Deletion

Best Case

Similar to the best cases for insertion and retrieval, the best case for the deletion method, **remove**(), centers around having a single element in the table that matches the key we are looking to delete for. Once, again we start the analysis with the statement that hashing a given key is $\mathcal{O}(1)$ bounded.

- Line 157 is a call to the **validateKeySize**() function which is a series of $\mathcal{O}(1)$ statements, and as such is $\mathcal{O}(1)$ asymptotically

- Lines $159 - 161$ here the hash is calculated, a previous node is instantiated and the bucket related to the hash is retrieved, all taking $\mathcal{O}(1)$ time.
- Line 163 is a call to the **getSequentialNonNull**() function will execute a single loop cycle since upon evaluating the loop after the first iteration the value of **bucket** will be a nullptr, therefore this will be $\mathcal{O}(1)$
- Line 165 will run but not satisfy the expression (in $\mathcal{O}(1)$ time), as such lines $166 - 167$ wont execute, so these are ignored.
- Line 169 is a call to the **processBucketLinking**() function which will be $\mathcal{O}(1)$ since the provided **prev** argument will be a nullptr, immediately assigning the value to the bucket location in the table.
- Lines $170 - 171$ are de-allocating the element and decrementing the $\mathbf{element_count}$ attribute, both of which are $\mathcal{O}(1)$ bounded.

Using the above line by line analysis we end up with the following expression as a sum of all complexities:

$$\mathcal{O}(1) + 3 \cdot \mathcal{O}(1) + \mathcal{O}(1) + \mathcal{O}(1) + \mathcal{O}(1) + 2 \cdot \mathcal{O}(1)$$

$$= \mathcal{O}(1)$$

Therefore the time complexity of deleting in the best case is $\mathcal{O}(1)$ bounded

Average Case

Using the above proof for the average case of identifying the existence of a given element, at this point, the last operation to perform is to delete the pointer the found value which resulted from the call to **getSequentialNonNull**(). The alternative when no element is found is to return immediately since the condition on line 165 will be satisfied, in either case this is an $\mathcal{O}(1)$ operation, resulting in no overall change to the time complexity.

As we can see, in both cases of the key being found or not found, the resulting time complexity resolves to $\mathcal{O}(1 + \alpha)$, which is the overall time complexity of the **remove**() function in the average case

Worst Case

Using the above proof for the average case of identifying the existence of a given element, the last operation is to return the found value, which is an $\mathcal{O}(1)$ operation, resulting in no overall change to the time complexity.

As we can see, in both cases of the key being found or not found, the resulting time complexity resolves to $\mathcal{O}(1 + \alpha)$, which is the overall time complexity of the **get**() function in the average case

Worst Case

Analysing the worst case of deletion takes place with the entire table being filled, where all elements are hashed to the same slot. The element that is being sought after to delete is in the last slot of this linked list.

Let our element to be placed in the table be designated as v with key k, we can say the the existing list will be of length n. In this regard the insertion algorithm will retrieve the linked list at index $h(k)$. Next a call to the **getSequentialNonNull**() will result in a total of n inerations, as it loops over all n elements in the list, totalling n operations of complexity $\mathcal{O}(1)$:

$$1 + \sum_{i=1}^{n} \mathcal{O}(1)$$

$$= \mathcal{O}(n)$$

Lastly the element in the last last slot will be deleted and replaced with a nullptr, which is two constant time operations. Asymptotically, this does not change the existing $\mathcal{O}(n)$ complexity, and as such will remain as the final complexity.

Therefore, the time complexity of delete an element in the worst case is $\mathcal{O}(n)$.

3.2 Min Heap

All operations besides **heapifyUp**() and **heapifyDown**() are $\mathcal{O}(1)$ as they either perform only boolean checks, assignments or chained $\mathcal{O}(1)$ calls. Note that the implementation of min heap uses a vector, which has the same properties of $\mathcal{O}(1)$ for both insertion and deletion.

3.2.1 Insertion

The insert function is named **emplace()** within the MinHeap class.

Best Case

In the best case scenarios the heap is empty and as such insertion is equivalent to inserting an element into an array at index 0. This results in a time complexity of $\mathcal{O}(1)$.

Average Case

Deriving the average case insertion for a binary heap (in general, invariant of min or max), is based around the distribution of values and probabilistic positioning from a distribution. Suppose some value V is chosen to be inserted from a distribution bounded by existing inserted nodes. The structure of the tree is as follows:

- $\frac{n}{2}$ nodes at at height h
- $\frac{n}{4}$ nodes are at height $h-1$
- $\frac{n}{8}$ nodes are at height $h-2$

  $\vdots$
- 1 node is at the root $(h-h)$

In order to consider the average case, we need to weight each of these cases as probabilistic comparisons and heapifying swaps.

- $P(V) = \frac{1}{2}$ to be at level h using 1 comparison and no heapify calls
- $P(V) = \frac{1}{4}$ to be at level h using 2 comparison and 1 heapify calls
- $P(V) = \frac{1}{8}$ to be at level h using 3 comparison and 2 heapify calls

  $\vdots$

We can express this as a an infinite series of weighted terms:

$$1 \cdot \frac{1}{2} + 2 \cdot \frac{1}{4} + 3 \cdot \frac{1}{8} + \dots$$

Which equates to the infinite sum:

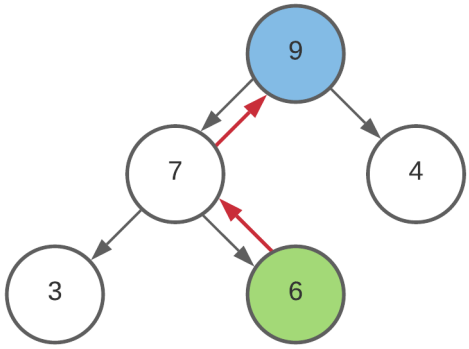
$$\sum_{i=0}^{\infty} \frac{i}{2^i}$$

Here we note that this sum converges to 2 when taken as $i \to \infty$. As such we can conclude that on average it will take a time of $\mathcal{O}(2)$ which is asymptotically equivalent to $\mathcal{O}(1)$.

Worst Case

1. Line 75 is $\mathcal{O}(1)$
2. Line 76 references **heapifyUp()** which is $\mathcal{O}(\log(n))$, as proved below
3. Line 77 is $\mathcal{O}(1)$

The method **heapifyUp**() traverses the tree from a node upwards progressively swapping nodes if the current key is less than that of the parent. In the maximal case, this traversal will take place from leaf node to root node.



Blue: root, green: node in question, red: path

This is a traversal of $\log(n)$ nodes equal to the height of the tree, h. As such we can conclude that the time complexity from traversing from leaf to root is:

$$\mathcal{O}(h) = \mathcal{O}(\log(n))$$

Therefore the total time complexity of **heapifyUp**() is:

$$\mathcal{O}(1) + \mathcal{O}(\log(n)) + \mathcal{O}(1)$$

$$= \mathcal{O}(\log(n))$$

3.2.2 Deletion

The delete function is named **pop()** within the MinHeap class.

Best Case

In the best case, the heap contains a single element. This is equates to a single comparison between the key argument and the root element and then the equivalent of array removal at index 0. Therefore the best case time complexity is $\mathcal{O}(1)$.

Worst Case

1. Lines $90 - 94$ are $4 \times \mathcal{O}(1) = \mathcal{O}(1)$
2. Line 95 references **heapifyDown()** which is $\Theta(\log(n))$, as proved below

With the heap property of a balanced binary tree, a maximal sub-problem is encountered when the bottom level is half full. This case translates to:

$$n = 1 + \textbf{LST.size()} + \textbf{RST.size()}$$

Where h is the height of the tree, **LST.size()** and **RST.size()** are defined as:

$$\textbf{LST.size()} = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

$$\textbf{RST.size()} = \sum_{i=0}^{h-1} 2^i = 2^h - 1$$

Using this with the original equation we obtain:

$$n = 1 + \textbf{LST.size()} + \textbf{RST.size()}$$

$$n = 1 + \left[\sum_{i=0}^{h} 2^i\right] + \left[\sum_{i=0}^{h-1} 2^i\right]$$

$$n = 1 + \left[2^{h+1} - 1\right] + \left[2^h - 1\right]$$

$$n = 2^h(2 + 1) - 1$$

$$2^h = \frac{n+1}{3}$$

Using the above result and substituting into LST, we can see that:

$$\textbf{LST.size()} = 2^{h+1} - 1$$

$$= 2^h \times 2 - 1$$

$$= \frac{2n+1}{3} - 1 \leq \frac{2n}{3}$$

Therefore the total time for **heapifyDown()**, by substituting the above, is:

$$T(n) \leq T\left(\frac{2n}{3}\right) + C$$

$$= T\left(\frac{n}{1.5}\right) + C$$

Here, we can apply the master theorem with $a = 1, b = 1.5, f(n) = n^{\log_b(a)}$. Note that:

$$f(n) = n^{\log_b(a)} = n^{\log_{1.5}(1)} = n^0 = 1$$

As such we can write this in the form of case 2:

$$T(n) = \Theta\left(f(n)\log(n)\right)$$

$$= \Theta\left(\log(n)\right)$$

Therefore, the total time complexity of **heapifyDown**() is:

$$4 \times \mathcal{O}(1) + \Theta(\log(n))$$

$$= \mathcal{O}(1) + \Theta(\log(n))$$

$$= \Theta(\log(n))$$

Or more generally with respect to tree height:

$$= \mathcal{O}(h)$$

3.3 A* Path finding

In regards to the A* search algorithm, the worst case is an empty $n \times n$ grid, with an incurred cost 1 when traversing between Von Neumann neighbours (Manhattan distance 1). For the extended neighbourhood of Moore neighbours, a diagonal cost of $\sqrt{2}$ (direct distance from current position to diagonally connected grid locations) is considered.

3.3.1 Best Case

The best case is a general best case for all grid/graph search algorithms, where the start and end nodes are in the same Moore neighbourhood. As such the algorithm will be $\Theta(1)$ As the algorithm will perform a single traversal and evaluate neighbours to find immediately that the end is a direct neighbour of the start.

Note that the **emplace**() and **pop**() calls to the min heap, will be in the best case of an empty heap and single element heap respectively. Both of these resulting in $\mathcal{O}(1)$ time complexity for the calls.

This is invariant of the amount of nodes in the graph and will always take the same amount of iterations (only 1). Therefore, the best case is $\Theta(1)$.

Lemma 2 A heuristic function is said to be consistent, or monotone, if its estimate is always less than or equal to the estimated distance from any neighbouring vertex to the goal, plus the cost of reaching that neighbour. That is:

$$h(N) \leq c(N, P) + h(P)$$

$$h(G) = 0$$

Where:

- h is the consistent heuristic function
- N is any node in the graph
- P is any descendant of N
- G is any goal node
- $c(N, P)$ is the cost of reaching node P from N

Reference: `https://en.wikipedia.org/wiki/Consistent_heuristic`

–END LEMMA–

It is important to note that the heuristic used by A* to calculate priorities of traversal is monotone as each time a node is queried and subsequently calculated for cost, it is already minimised relative to the cost function.

Therefore, for a graph (grid) with cost function:

$$c'(N, P) = c(N, P) + h(P) - h(N)$$

and a consistent h, A* is equivalent to breadth-first search on the grid using Dijkstra's algorithm. This results in the neighbours with Manhattan distance 1 being considered first, then the missing Moore neighbours in succession.

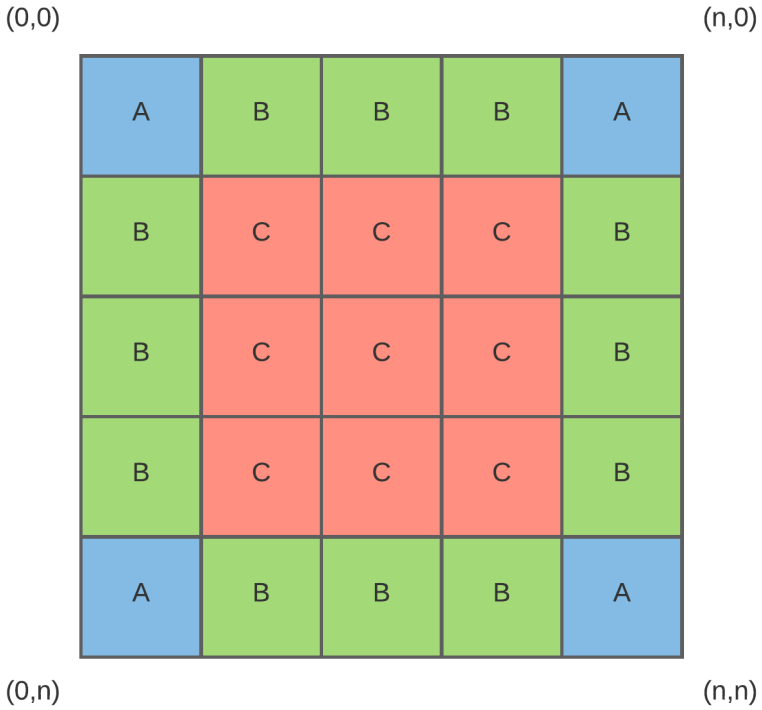
3.3.2 Average Case

In order to analyse the average case time complexity, there are a few assumptions and declarations that need to be made before hand:

- We will refer to the grid as a graph for simplicity and accuracy in terms of connected points (verticies)
- Let $|E|$ and $|V|$ be the number of edges and the number of vertices in the graph respectively.
- The heuristic will be ignored in the analysis as it is monotone, providing no additional overhead to the calculations involved in the traversals.

In order to analyse the average case, we first need to determine the average connectivity of a graph in a grid formation. A grid a connected graph by definition, but has particular casing for maximum connection between nodes.

In the minimum connected case, the connections are the same as any connected graph, $|E_{\min}| = n - 1$ edges. However, for the maximum connections this is different. Below is a diagram of the connectivity relations for an arbitrary $n \times n$ grid:



There are 3 classes of connnectivity with the grid structure, these are marked as A,B and C. The connectivity of class A is $|A_E| = 3$, class B is $|B_E| = 5$ and class C is $|C_E| = 3$. Note that in the graph there are always 4 class A nodes, $4(n-2)$ class B nodes and $(n-2)^2$ class C nodes.

Expressing this as an equation, we get:

$$\begin{aligned} |E_{\max}| &= 4|A_E| + 4(n-2)|B_E| + (n-2)^2|C_E| \\ &= 4(3) + 4(n-2)(5) + (n-2)^2(8) \\ &= 8n^2 - 12n - 60 \end{aligned}$$

Now that we have both the minimal and maximal connectivity of an arbitrary grid, we can compute the average connectivity as:

$$\begin{aligned} |E_{\text{avg}}| &= \frac{|E_{\max}|}{|E_{\min}|} \\ &= \frac{8n^2 - 12n - 60}{n-1} \\ &= 8n - 4 - \frac{64}{n-1} \end{aligned}$$

It is also important to note the respective probabilities of any given node being a certain class. We can classify these probabilities as a quantity in terms of n^2 (total node count):

$$X_i = \begin{cases} \frac{4}{n^2} & V_i == A \\ \frac{4n-8}{n^2} & V_i == B \\ \frac{(n-2)^2}{n^2} & V_i == C \end{cases}$$

In the implementation, all lines before the while loop ate of complexity $\mathcal{O}(1)$ and as such will be ignored. Considering the loop itself, there are a total of $|E_{\text{avg}}|$ iterations on average, where ℓ is the time complexity body of the loop.

$$\sum_{i=1}^{|E_{\text{avg}}|} \ell_i$$

Here we can expand ℓ in terms of average neighbours to consider. Here all statements other than the for loop are $\mathcal{O}(1)$ and will be ignored since they don't change the derivation. In order to account for average neighbours, we need the expectation value of X_i as described above:

$$\begin{aligned} E[X_i] = P(X_i) &= \frac{\left(3 \cdot \frac{4}{n^2}\right) + \left(5 \cdot \frac{4n-8}{n^2}\right) + \left(8 \cdot \frac{(n-2)^2}{n^2}\right)}{\frac{4}{n^2} + \frac{4n-8}{n^2} + \frac{(n-2)^2}{n^2}} \\ &= \frac{8n^2 - 12n + 4}{n^2} \end{aligned}$$

Now that we know the average neighbours considered in the for loop, we can use this as a replacement for the expectation value of $E[X_i]$ in ℓ:

$$\ell_i = \sum_{j=1}^{\frac{8n^2-12n+4}{n^2}} C$$

From here we can evaluate the sum as just the total sum of constant time operations:

$$\begin{aligned} &\sum_{i=1}^{|E_{\text{avg}}|} \frac{8n^2 - 12n + 4}{n^2} \\ &= |E_{\text{avg}}| \cdot \frac{8n^2 - 12n + 4}{n^2} \\ &= \left(8n - 4 - \frac{64}{n-1}\right) \cdot \frac{8n^2 - 12n + 4}{n^2} \\ &= \frac{16\left(2n^2 - 3n - 15\right)(2n-1)}{n^2} \end{aligned}$$

Evaluating this expression asymptotically, for $n \to \infty$, this expression is bounded by $\mathcal{O}(n)$:

$$\begin{aligned} &\lim_{n\to\infty} \frac{\frac{16\left(2n^2-3n-15\right)(2n-1)}{n^2}}{n} \\ &= \lim_{n\to\infty} \frac{16\left(2n^2 - 3n - 15\right)(2n-1)}{n^3} \\ &= 64 \end{aligned}$$

Additionally, we can show that the time complexity exceeds $\mathcal{O}(\log(n))$ and as such the closest bound is $\mathcal{O}(n)$:

$$\begin{aligned} &\lim_{n\to\infty} \frac{\frac{16\left(2n^2-3n-15\right)(2n-1)}{n^2}}{\log(n)} \\ &= \lim_{n\to\infty} \frac{16\left(2n^2 - 3n - 15\right)(2n-1)}{n^2 \log(n)} \\ &= \infty \end{aligned}$$

As such in the average case, the A* has $\mathcal{O}(n)$ time complexity, though we can re-write this in terms of V and E_{avg} as $(|V| + |E_{\text{avg}}|) \cdot \log(|V|)$. We need to show that the original complexity grows at the same rate as the suggested equivalent, for this to occur it must satisfy: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = C$, where C is some constant.

$$\lim_{n\to\infty} \frac{\frac{16\left(2n^2-3n-15\right)(2n-1)}{n^2}}{\left(\frac{8n^2-12n+4}{n^2} + \left(8n - 4 - \frac{64}{n-1}\right)\right) \cdot \log\left(\frac{8n^2-12n+4}{n^2}\right)}$$

Simplifying this expression leaves us with:

$$= \lim_{n\to\infty} \frac{4\left(2n^2-3n-15\right)(2n-1)(n-1)}{(2n^4-n^3-20n^2+4n-1)\cdot(\log(8n^2-12n+4)-2\log_2(n))}$$

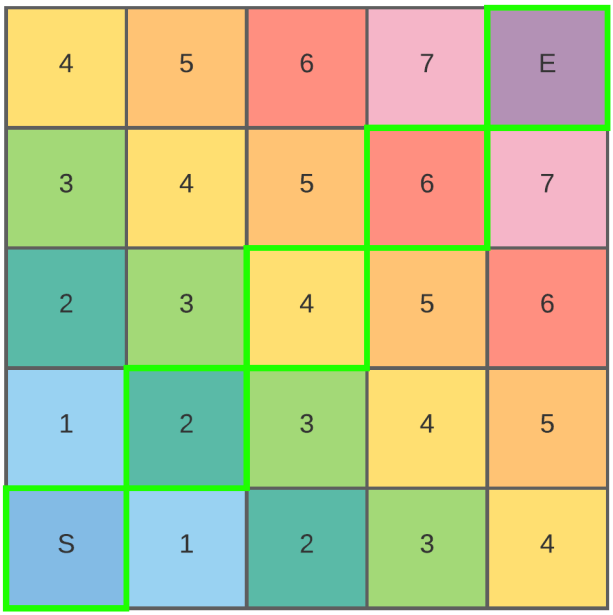
Applying the limit:

$$= \frac{8}{3}$$

Therefore, the final expression in terms of V and E_{avg} is $\mathcal{O}((|V|+|E_{\text{avg}}|)\cdot\log(|V|))$ which is much more descriptive for the grid than $\mathcal{O}(n)$

3.3.3 Worst Case

For a 5×5 grid with start S and end E, this would result in the following traversal order (Outlined in green is the path found):



As the heuristic is admissible and the degradation of the algorithm in the worst case, we can say that the worst case running time is

$$\mathcal{O}(n^2)$$

4 Empirical Time Analysis

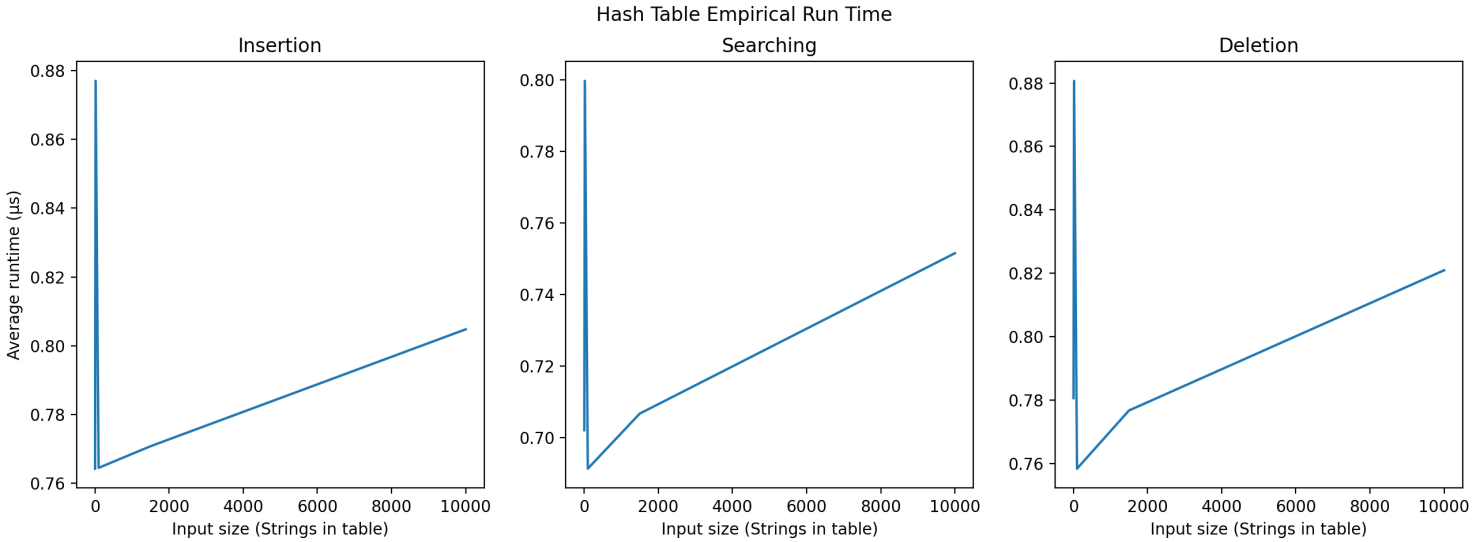
As a side note, the values you are about to see are not fake, this took a ton of time to run. It is important to note how many CPU cycles take place relative to processor speed. In this case, this was tested on a 1.4GHz i5-8257U CPU with 8GB 2133MHz LPDDR3 RAM, this equates to approximately 1400 CPU cycles per nanosecond, this metric will be used in the following analysis.

4.1 Hash Table

Each test performs n number of insertions, searches and deletions a total of $10,000$ times. This totals, in the largest case, $100,000,000$ individual calls of the relevant method in insertion, deletion and searching.

It is important to note how many CPU cycles take place relative to processor speed. In this case, this was tested on a 1.4GHz i5-8257U CPU with 8GB 2133MHz LPDDR3 RAM, this equates to approximately 1400 CPU cycles per nanosecond, this metric will be used in the following analysis.

Number of strings	Insertion [µs]	Searching [µs]	Deletion [µs]
5	0.737102	0.683428	0.755392
20	0.97075	0.990546	0.97659
100	1.00387	0.912066	1.00299
1500	0.78047	0.710211	0.778716
10000	0.781799	0.720695	0.786963



Although these graphs may seem extremely wild and not close to constant time, but relative to actual execution on the CPU and the deviation between tests, this is very close to constant time. Here we can note the largest difference in each case is:

- Insert: 0.112857 µs equating to approximately 158 CPU cycles
- Search: 0.108436 µs equating to approximately 152 CPU cycles
- Delete: 0.122384 µs equating to approximately 171 CPU cycles

With this we can see that the actual deviation between inserting 5 elements and $10,000$ elements with $10,000$ trials is on average 158, 152 and 171 CPU cycles. This is extremely low and well within reason for justification of $\mathcal{O}(1)$ time complexity.

In the interest of a more statistically significant metric, we can calculate the standard deviation of each method of the hash table. The formula for the standard deviation is:

$$\rho = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \mu)^2}{N}}$$

Where:

- ρ is the standard deviation
- N is the size of population
- x_i is the i^{th} element of the population
- μ is the population mean

Here N is defined as the amount of differing input sizes we used; 5. Additionally, x_i is defined as the i^{th} element of $\forall x \in \overline{T}$ where $\overline{T}$ is the set of average runtimes. We also define μ as:

$$\mu = \frac{\sum_{j=1}^{N} x_j}{N}$$

From this we can evaluate each of the variables according to the method, note that $N = 5$ for all methods:

Insertion

Using the insertion data, we can calculate μ as:

$$\mu = \frac{\sum_{j=1}^{5} x_j}{5}$$

$$= \frac{0.737102 + 0.97075 + 1.00387 + 0.78047 + 0.781799}{5}$$

$$= \frac{4.273991}{5} = 0.8547982$$

Now we can calculate the sum at the top of the fraction, which we will designate as S:

$$S = \sum_{i=1}^{N} (x_i - \mu)^2$$

$$= \sum_{i=1}^{5} (x_i - 0.8547982)^2$$

$$= (0.737102-0.8547982)^2+(0.97075-0.8547982)^2+(1.00387-0.8547982)^2+(0.78047-0.8547982)^2+(0.781799-0.8547982)^2$$

$$\approx 0.06037$$

Using this in the standard deviation formula, we get:

$$\rho = \sqrt{\frac{S}{5}} = \sqrt{\frac{0.06037}{5}}$$

$$\approx 0.10988 \ \mu s$$

$$\approx 153.832 \text{ CPU cycles}$$

As we can see, this standard deviation is very much inline with what we'd expect to see with constant time deviation.

Searching

Using the same process for searching we evaluate μ as:

$$\mu = \frac{0.683428 + 0.990546 + 0.912066 + 0.912066 + 0.720695}{5}$$

$$= \frac{4.218801}{5} = 0.8437602$$

Where S is:

$$S = \sum_{i=1}^{5} (x_i - 0.8437602)^2$$

$$= (0.683428-0.8437602)^2+(0.990546-0.8437602)^2+(0.912066-0.8437602)^2+(0.912066-0.8437602)^2+(0.720695-0.8437602)^2$$

$$\approx 0.07172$$

Finally, calculating ρ, the standard deviation, as:

$$\rho = \sqrt{\frac{S}{5}} = \sqrt{\frac{0.07172}{5}}$$

$$\approx 0.11977 \ \mu s$$

$$\approx 167.678 \ CPU cycles$$

Deletion

Using the same process for searching we evaluate μ as:

$$\mu = \frac{0.755392 + 0.97659 + 1.00299 + 0.778716 + 0.786963}{5}$$

$$= \frac{4.300651}{5} = 0.8601302$$

Where S is:

$$S = \sum_{i=1}^{5} (x_i - 0.8601302)^2$$

$$= (0.755392-0.8601302)^2+(0.97659-0.8601302)^2+(1.00299-0.8601302)^2+(0.778716-0.8601302)^2+(0.786963-0.8601302)^2$$

$$\approx 0.05692$$

Finally, calculating ρ, the standard deviation, as:

$$\rho = \sqrt{\frac{S}{5}} = \sqrt{\frac{0.05692}{5}}$$

$$\approx 0.10669\ \mu s$$

$$\approx 149.366\ CPUcycles$$

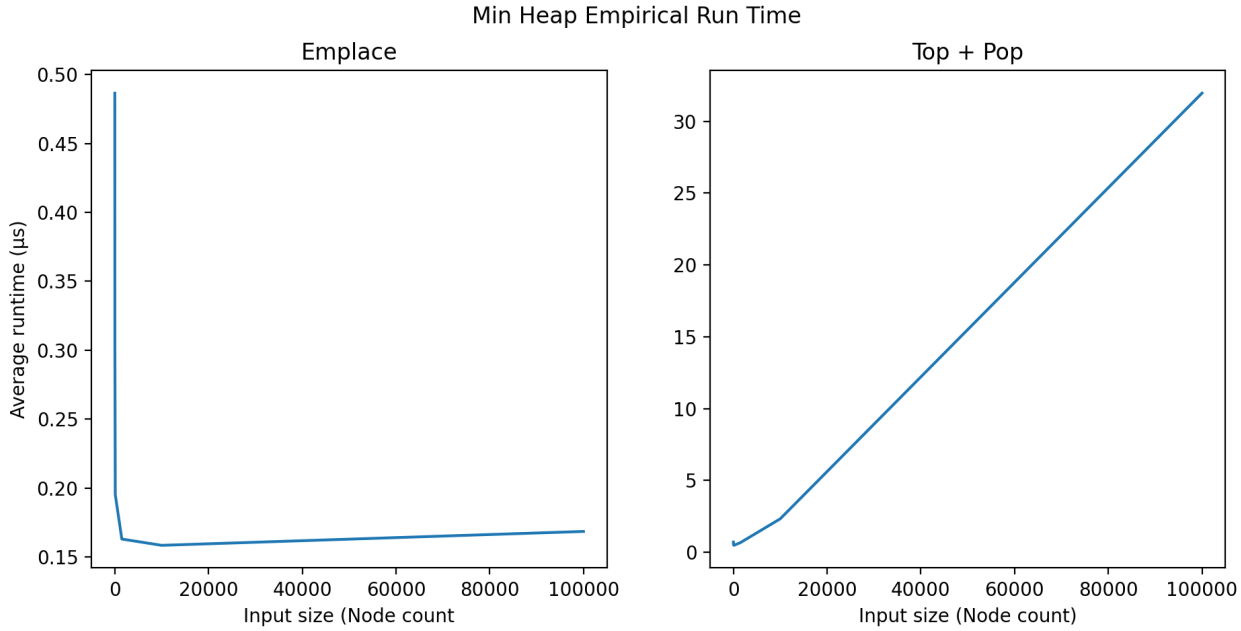
As we can see, for all three methods, the standard deviation is very small, to the point where it can be accounted for in inherit uncertainty of the OS. Such things as interrupts, process usage spiking, process niceness changes, large scale memory management, etc.

We can conclude from this that the time complexity of the hash table is indeed $\mathcal{O}(1)$ in the average case.

4.2 Min Heap

Each test performs n number of insertions, searches and deletions a total of $10,000$ times. This totals, in the largest case, $1,000,000,000$ individual calls of the relevant method in insertion, deletion and searching.

Node Count	**Emplace [µs]**	**Top + Pop [µs]**
5	0.486602	0.710968
20	0.305427	0.557348
100	0.195178	0.482622
1500	0.162943	0.670009
10000	0.15839	2.32839
100000	0.168463	31.9559



The results of the min heap are extremely close to that of the theoretical analysis. You'll note that there is an initial spike in the emplace graph, we can account for this simply as inaccuracy with lower node count makes the emplacement of a node inconsistent.

Overall, we can see that the **emplace**() method is indeed constant time relevant to the node count and the **top**()+**pop**() methods are of linear time complexity.

In order to grasp the variance involved, we will perform the same analysis as with the hash table, using the standard deviation. Note that, however, we will exclude the lowest test case of the **emplace**() method as is it an outlier but also it's inaccuracy comes not from the algorithm but inconsistent call scenarios.

$$\mu = \frac{\sum_{j=1}^{N} x_j}{N}$$

Emplace

Utilising the emplace data, excluding the flaky first set, we can calculate μ as:

$$\mu = \frac{\sum_{j=2}^{N} x_j}{5}$$

$$= \frac{0.305427 + 0.195178 + 0.162943 + 0.15839 + 0.168463}{5}$$

$$= 0.1980802$$

Now we can calculate the sum at the top of the standard deviation formula, as S:

$$S = \sum_{j=2}^{N} (x_j - \mu)^2$$

$$= \sum_{j=2}^{5} (x_i - 0.1980802)^2$$

$$= (0.305427-0.1980802)^2+(0.195178-0.1980802)^2+(0.162943-0.1980802)^2+(0.15839-0.1980802)^2+(0.168463-0.1980802)^2$$

$$= 0.01521$$

Using this in standard deviation formula:

$$\rho = \sqrt{\frac{S}{5}} = \sqrt{\frac{0.01521}{5}}$$

$$\approx 0.05517\ \mu s$$

$$\approx 77.238 \text{ CPU cycles}$$

This deviation of 77.238 CPU cycles is very low and more than within acceptable bounds to account for inconsistencies in the underlying frameworks and systems the test was run on. As such we can conclude that the experimental data does indeed reflect the $\mathcal{O}(1)$ time complexity of the theoretical analysis. **Top + Pop**

In the case of the top and pop data, standard deviation is not going to tell us much about the overall performance. Instead we will perform linear regression on the data set and calculate approximates for intermediate node sizes.

First lets define the formulas that are needed, the equation of a line:

$$y = b_0 + b_1 x$$

Where

$$b_0 = \frac{(\sum y) \cdot (\sum x^2) - (\sum x) \cdot (\sum x \cdot y)}{n \cdot (\sum x^2) - (\sum x)^2}$$

$$b_1 = \frac{n \cdot (\sum x \cdot y) - (\sum x)(\sum y)}{n \cdot (\sum x^2) - (\sum x)^2}$$

Let us define out variables as follows: Substituting the relevant values in the formulas for b_0 and b_1, we obtain:

Subject	**x**	**y**	**x · y**	**x²**	**y²**
1	5	0.710968	3.55484	25	0.505475497
2	20	0.557348	11.14696	400	0.310636793
3	100	0.482622	48.2622	10000	0.232923995
4	1500	0.670009	1005.0135	2250000	0.44891206
5	10000	2.32839	23283.9	100000000	5.421399992
6	100000	31.9559	3195590	10000000000	1021.17954481
$\sum$	111625	36.705237	3219941.8775	10102260425	1028.0988931470001

$$b_0 = \frac{(36.705237) \cdot (10102260425) - (111625) \cdot (3219941.8775)}{n \cdot (10102260425) - (111625)^2}$$

$$= \frac{11379851059.4082}{10102260425n - 12460140625}$$

$$b_1 = \frac{n \cdot (3219941.8775) - (111625) \cdot (36.705237)}{n \cdot (10102260425) - (111625)^2}$$

$$= \frac{c \cdot 3219941.8775 - 4097222.08012\ldots}{n \cdot 10102260425 - 12460140625}$$

Here, our sample size, n, is 6. Equating the above with this substitution, we get:

$$b_0 = 0.23632$$

$$b_1 = 0.00031$$

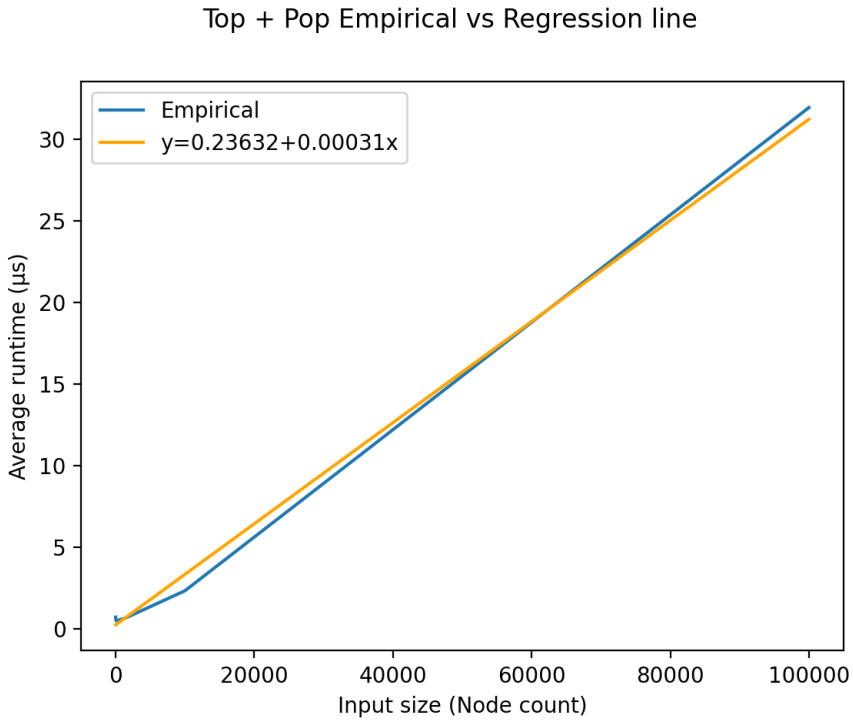
Now we can substitute the values of b_0 and b_1 into the original linear equation:

$$y = 0.23632 + 0.00031 \cdot x$$

Now we can input various node counts as the value x, to calculate the runtime in μs. We will use the values 1, 2500, 40,000, 87,000 and 100,000:

-
- 1: $y = 0.23632 + 0.00031 \cdot 1 = 0.23663$
- 2500: $y = 0.23632 + 0.00031 \cdot 2500 = 1.01132\ \mu s$
- 40,000: $y = 0.23632 + 0.00031 \cdot 40000 = 12.63632\ \mu s$
- 87,000: $y = 0.23632 + 0.00031 \cdot 87000 = 27.20632\ \mu s$
- 100,000: $y = 0.23632 + 0.00031 \cdot 100000 = 31.23632\ \mu s$

This results in the following line, which is an extremely close approximation to $y = n$ or $\mathcal{O}(n)$:



As such we can conclude that the experimental results do indeed reflect the theoretical runtime of $\mathcal{O}(n)$.

4.3 A* Path Finding

Testing the path finding algorithm requires creating 5 different maps, doing this required a randomly generated maps for varying sizes. In order to achieve this, a modified version of a grid aligned procedural dungeon generator was used with varying width, height, room count and connections.

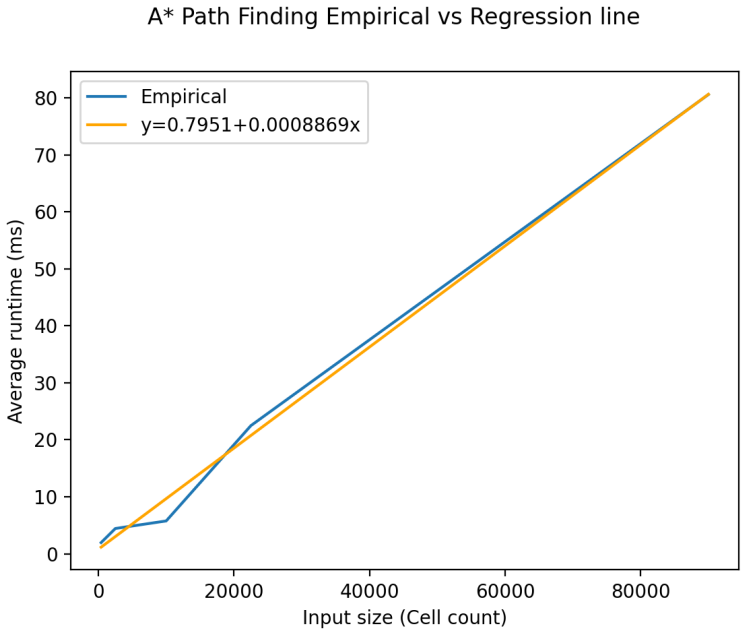
It is important to recognize that given the structure of the maps used, these files get very big very quickly, to the point where a $10,000 \times 10,000$ grid is somewhere in the range of 20GB of data. In light of this, the map sizes that are used in the tests are:

- 20×20 grid, totalling 400 cells
- 50×50 grid, totalling 1500 cells
- 100×100 grid, totalling 10,000 cells
- 150×150 grid, totalling 22,500 cells

- 300×300 grid, totalling $90,000$ cells

From the above configurations, 5 maps have been generated and tested with the path finding algorithm. Below we can see the results of these tests:

Cell Count	**Path Found [ms]**
400	1.95493
2500	4.42906
10000	5.75341
22500	22.4798
90000	80.5743



Immediately, we can see there seems to be a linear relationship between the two from the fact that there is an approximate scalar of 0.0009 relating the two. In order to see how close this relation is, we can perform the same linear regression as in the Min Heap.

In performing this regression, we achieve the following linear equation:

$$7 = 0.7951 + 0.0008869 \cdot x$$

As we can see from the b_0 and b_1 scalars, this is extremely close to a perfect $\mathcal{O}(n)$ linear relationship. It should be noted that on the low end of the experiments, this has some inconsistency, though not outside of a reasonable level.

We can conlude that the algorithm is indeed $\mathcal{O}(n)$, or more descriptively $\mathcal{O}((|V| + |E_{avg}| \cdot \log |V|))$