Shortest Path Heuristic

MET CS 526 – Data Structure and Algorithms

Spring 2 – 2018

Final Project

Michael Melvin

**Introduction**

In many branches of science, including computer science, artificial intelligence, and mathematical optimization, a heuristic is a technique designed for solving a problem more quickly when classic methods are too slow, or for finding an approximate solution when classic methods fail to find any exact solution. This is achieved by trading optimality, completeness, accuracy, and/or precision, for speed. It is basically a shortcut.[[1]](#footnote-1) In this project, we utilize this strategy to implement two shortest path algorithms for traversing a graph from one node to another in the graph. A graph is a way of representing relationships that exist between pairs of objects called nodes or vertices, together with a collection of connections between these object, called edges, which can be weighted or unweighted, and a graph can be overall directed or undirected.[[2]](#footnote-2) In our study, we use an undirected graph. Graphs have many applications, including but not limited to mapping, networking, traffic, circuitry, and the internet. Along with the graph, we also use stacks, heap-based priority queues, hash maps, and hash sets to implement the algorithms. Next, we will take a detailed look at the aforementioned data structures and explain why they were chosen for their specific role in the algorithm.

**Data Structures**

*Graphs:*

A graph is a collection of vertices and edges, so an ADT to represent a graph would need to provide functions necessary for the creation of, access to, iteration through, removal of, and counting of the vertices and edges in a particular graph. With that in mind, we use the Graph ADT with the following methods[[3]](#footnote-3):

* **numVertices():** Returns the number of vertices of the graph.
* **vertices():** Returns an iteration of all the vertices of the graph.
* **numEdges():** Returns the number of edges of the graph.
* **edges():** Returns an iteration of all the edges of the graph.
* **getEdge(u, v):** Returns the edge from vertex u to vertex v, in one exists, otherwise return null. For an undirected graph, there is no difference between **getEdge(u, v)** and **getEdge(v, u)**.
* **endVertices(e):** Returns an array containing the two endpoint vertices of the edge e. If the graph is directed, the first vertex is the origin and the second is the destination.
* **opposite(v, e):** For edge e incident to vertex v, returns the other vertex of the edge. An error occurs if e is not incident to v.
* **outDegree(v):** Returns the number of outgoing edges from vertex v.
* **inDegree(v):** Returns the number of incoming edges from vertex v. For an undirected graph, this returns the same value as does out degree.
* **outgoingEdges(v):** Returns an iteration of all outgoing edges from vertex v.
* **incomingEdges(v):** Returns an iteration of all incoming edges from vertex v. For an undirected graph, this returns the same value as does **outgoingEdges(v)**.
* **insertVertex(x):** Creates and returns a new Vertex storing element x.
* **insertEdge(u, v, x):** Creates and returns a new Edge from vertex u to vertex v, storing element x. An error occurs if there already exists an edge from u to v.
* **removeVertex(v):** Removes vertex v and all its incident edges from the graph.
* **removeEdge(e):** Removes edge e from the graph.

There are different ways of implementing the Graph ADT, an edge list, an adjacency list, an adjacency map, and an adjacency matrix. The edge list implementation uses an unordered list of all of the edges of the graph. This doesn’t allow for efficient access to a particular edge, or access to all of the edges of a particular vertex. The adjacency list maintains a separate list of each vertex’s edges, allowing for the more efficient access to all edges incident a vertex. The adjacency map is similar, but it maintains a map to the edges, with the vertex as the key. This allows O(1) access to a specific edge. The adjacency matrix maintains an n x n matrix for a graph with n vertices. The elements of the matrix storing the edges. This allows O(1) access to specific edges, but O(n2) space, while all other implementations use O(m + n) space. The running times of the methods above for each of these implementations are shown in the table below, where n is the number of vertices, m the number of edges, dv the degree of vertex v.[[4]](#footnote-4)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Method** | Edge List | Adj. List | Adj. Map | Adj. Matrix |
| numVertices() | O(1) | O(1) | O(1) | O(1) |
| numEdges() | O(1) | O(1) | O(1) | O(1) |
| vertices() | O(n) | O(n) | O(n) | O(n) |
| edges() | O(m) | O(m) | O(m) | O(m) |
| getEdge(u, v) | O(m) | O(min(du, dv)) | O(1) exp. | O(1) |
| outDegree(v)  inDegree(v) | O(m) | O(1) | O(1) | O(n) |
| outgoingEdges(v)  incomingEdges(v) | O(m) | O(dv) | O(dv) | O(n) |
| insertVertex(x) | O(1) | O(1) | O(1) | O(n2) |
| removeVertex(x) | O(m) | O(dv) | O(dv) | O(n2) |
| insertEdge(u, v, x) | O(1) | O(1) | O(1) exp. | O(1) |
| removeEdge(e) | O(1) | O(1) | O(1) exp. | O(1) |

The adjacency map has the best overall running times for these methods, and we will therefore choose this implementation for our graph structure. We also make use of the Vertex and Edge ADT’s which are trivially implemented as interfaces providing only a getElement() method.

*Stacks:*

Stacks are a widely used data structure that adhere to the last-in, first-out principle. This means that objects can be inserted or pushed into the stack, and the last element pushed to the stack can be removed or popped from the stack. There are numerous applications for these data structures such as the back button on a browser that stores your previously visited sites on a stack, or the undo button in a text editor. The Stack ADT used in this study is the Stack ADT provided by Java. This was chosen over the text books implementation because in extends the Vector ADT which has a contains(e) method that executes a linear search of the stack returning true or false depending on if the element was found. The contains(e) method runs in worst case O(n) time, where n is the number of elements on the stack. All of the other methods, pop(), push(e), peek(), size(), and empty(), run in constant time. This data structure is uniquely suited for backtracking because we have O(1) access to the last element on the stack. It is for these O(1) push(e) and pop() methods that we chose this data structure. By storing the visited vertices in a stack, we can very efficiently push() a visited vertex to the stack or check the last vertex visited with the peek() method, and if we need to backtrack, we simply pop() the last visited vertex from the stack. The methods provided by java.util.Stack and the one java.util.Vector used in the project are detailed below.

* **push(e):** Adds element e to the top of the stack.
* **pop():** Removes and returns the top element from the stack (or null if the stack is empty).
* **peek():** Returns but doesn’t remove the top element from the stack (or null if the stack is empty).
* **empty():** Returns a boolean indicating whether the stack is empty.
* **contains():** Returns a boolean indicating whether or not element e is in the stack.

*Heap Priority Queue:*

A Priority Queue is a data structure whose objects are added in any order but are removed based on a given priority. The object whose priority is the lowest will be removed first. The objects are stored in the priority queue as entries, each having a key and a value. There are many applications of priority queues, ‘whales’ at casinos may getting the best rooms over lower spending guests, frequent flyers getting access to limited seating, all because they have a higher priority. We will need to insert items into this priority queue and then remove the minimum keyed item multiple times in this study, so we will need to implement this with a data structure that allows for us to do this efficiently, enter the heap.

A heap is a binary tree that stores entries at its positions and satisfies the heap-order and complete binary tree property. The heap order property states that in a heap, for every position p other than the root, the key stored at p is greater than or equal to the key stored at p’s parent. And the complete binary tree property states that a heap with height h is complete if and only if the levels 0, 1, …, h – 1 of the tree have the maximum number of nodes possible and the remaining nodes at level h reside in the leftmost possible positions at that level. The result of these properties is that the root will always have the minimum key and the height of the tree will be equal to floor(log n).

When an entry is added to the heap, it is always added to the left most available position on the bottom level, or the left most position on a new level if bottom level is full. After this, the heap-order property may be violated, so an up-heap bubbling is executed. This is done by comparing the new entry p to its parent q. If the key of p is greater than or equal to the key of q, then nothing must be done. Otherwise, the entries are swapped and the process is repeated until the heap-order property is satisfied. Because a heap has a max height of log n, the worst case running time of adding an element to a heap is O(log n). We make frequent use of the insert(key, value) method in our study by inserting the adjacent vertices of the current vertex into the priority queue with the direct distance to the destination vertex (direct distance + edge weight for algorithm 2) as the key and the vertex as the value.

The other method we make frequent use of the removeMin() method. Because of the heap-order property, we know that the element with the smallest key value is stored at the root of the heap. We can’t just remove it or we would have two sub trees. How we implement this method and preserve the heap-order and complete binary tree properties is by replacing the entry at the root that was removed with the entry that is at the right most position at the bottom level of the heap and deleting that node. Then perform a down-heap bubbling to preserve the heap-order property. Using the terminology of the last paragraph, there are two cases we must consider, other than the trivial case of the heap only having a root node.

1. If p has no right child, let c be the left child of p.
2. Otherwise (p has both children), let c be a child of p with minimal key.

If the key of p is less than or equal to the key of c, the property is satisfied. Otherwise, if the key of p is greater than the key of c, then we swap the entries stored at p and c, and continue with c’s children until the property is restored. Since the height of the tree is floor(log n), this runs in O(log n) worst case time. Since we loaded the priority queue with adjacent vertices having the distance to the destination as keys, the vertex we are to travel to next can be accessed in O(log n) time by calling removeMin(). A description of insert(k, v) and removeMin() follows.

* **insert(k, v):** Creates an entry with key k and value v in the priority queue.
* **removeMin():** Removes and returns an entry (k, v) having minimal key (or null if the priority queue is empty.

*Hash Maps:*

A map is an ADT that allows for the efficient storing and retrieving of values based upon a unique search key. It stores these entries as key value pairs. Since the vertices of our graph are unique, this serves as an efficient way to store the vertex labels as keys and their direct distance to the destination as the values. When implementing a map with a hash table, as we do in this study, we realize an O(1) expected get(k) and put(k, v), with worse case being O(n). Other methods that are common in maps are remove(k), size(), isEmpty(), entrySet(), keySet(), and values(). However, we only make use of get(k) and put(k, v), and they are described below. Maps have many applications such as storing student information based on a student ID, domain-name system maps a host name to an IP address, and other use cases that are rather similar.

* **get(k):** Returns the value v associated with key k, if such an entry exists, otherwise returns null.
* **put(k, v):** if the map does not have an entry with key equal to k, then adds entry(k, v) to the map and returns null. Else, replaces the existing value associated with k with v and returns the old value.

*Hash Set:*

The last data structure we will discuss is one we also use quite often but is probably the simplest of all of the data structures we have discussed thus far. A Set is an unordered collection of elements, without duplicates, that typically supports efficient membership tests. It is similar to a map, except that are no associated values. We use this in ADT in our study to by adding to it vertices we no longer want to visit and checking if a vertex is in the set. The methods provided by Java for this are add(e) and contains(e), both which run in O(1) time. It should be noted that the Hash Set ADT, provided by Java, is an unordered set implemented with a hash table. Ordering was not necessary, because we never have to access any of the objects in any order, we just need to add and check for membership. For the above reasons, a Hash Set was chosen to store the vertices we no longer want to visit along our path to the destination vertex. The methods we use are detailed below.

* **add(e):** Adds the element e to the set if it is not already present.
* **contains(e):** Returns a boolean indicating if element e is a member of the set.

Now that we are done discussing the main data structures used in the study, we will turn our attention to the implementation of the heuristic algorithms employed to find the shortest path from a source vertex to a destination vertex.

The purpose of these heuristic algorithms is to find a shortest path from a source vertex to a destination vertex. And while both of these algorithms accomplish that, it should be noted that the ‘shortest’ path found, may not indeed be ‘the’ shortest path. But, since these are heuristic algorithms, optimality and precision can be shunned for getting any answer, and sometimes even getting the right answer. The first algorithm attempts to find the shortest path by selecting the next vertex, from the adjacent vertices to the current vertex, the one that has the closest direct, as the bird flies, distance to the destination vertex. The second algorithm is much like the first, except it chooses the next vertex from the vertices adjacent to the current vertex by selecting the one who has the minimum sum of edge distance plus the direct distance to the destination vertex. Both of these values, the direct distance and the direct distance plus the edge distance, are stored as the keys in our priority queue, ensuring the proper vertex is selected. A high-level overview of each algorithm follows.

**Pseudocode**

**Algorithm:** algorithmOne (g, s, d, m)

**Input:** An undirected AdjacencyMapGraph, g, with weighted edges, a starting point, s, the destination vertex, d, and a map of the vertex labels to their direct distance from the destination vertex.

**Output:** The function returns void but displays the sequence of all vertices along the path taken, the sequence of vertices along the shortest path, and the distance traveled along the edges of the shortest path, to the console.

Create Stack ***path*** // to hold shortest path vertices.

Create Stack ***all*** // to hold all vertices visited.

Create HashSet ***dead*** // to hold vertices we no longer want to visit.

Set ***current*** vertex to ***s***.

*Push* ***current*** to ***path*** and ***all***.

*While* ***current*** vertex is not ***d*** do:

Create HeapPriorityQueue ***pq***. // this will contain adjacent vertices as values

Insert vertices adjacent to current into ***pq***. // and keys are direct distance to ***d***

Set ***next*** vertex to first vertex from ***pq***. // this vertex has shortest direct distance to ***d***

*If* ***next*** vertex is not in ***dead*** and not in ***path,*** then:

Push ***next*** to ***path***.

Push ***next*** to ***all***.

*If* ***next*** has only one adjacent vertex and it is not ***d,*** then:

Add ***next*** to ***dead***.

*If* ***current*** has only one adjacent vertex then:

Add ***current*** to ***dead***.

Add ***next*** to ***dead***.

Set ***current*** to ***next***.

*Continue*.

*Else* do:

Remove ***next*** from ***pat***h.

*If* ***next*** was not the last vertex pushed to ***all*** then:

Push ***next*** to all.

Add ***current*** to ***dead***

Add ***next*** to ***dead***.

Set ***current*** to the last vertex pushed to ***path*** and *continue*;

Calculate the distance traveled along the shortest path.

Display the results.

**Algorithm:** algorithmTwo (g, s, d, m)

**Input:** An undirected AdjacencyMapGraph, g, with weighted edges, a starting point, s, the destination vertex, d, and a map of the vertex labels to their direct distance from the destination vertex.

**Output:** The function returns void but displays the sequence of all vertices along the path taken, the sequence of vertices along the shortest path, and the distance traveled along the edges of the shortest path, to the console.

Create Stack ***path*** // to hold shortest path vertices.

Create Stack ***all*** // to hold all vertices visited.

Create HashSet ***dead*** // to hold vertices we no longer want to visit.

Set ***current*** vertex to ***s***.

*Push* ***current*** to ***path*** and ***all***.

*While* ***current*** vertex is not ***d*** do:

Create HeapPriorityQueue ***pq***. // this will contain adjacent vertices as values

Insert vertices adjacent to current into ***pq***. // keys are direct distance to ***d*** + edge weight

Set ***next*** vertex to first vertex from ***pq***. // this vertex has shortest ***d*** + edge weight

*If* ***next*** vertex is not in ***dead*** and not in ***path,*** then:

Push ***next*** to ***path***.

Push ***next*** to ***all***.

*If* ***next*** has only one adjacent vertex and it is not ***d,*** then:

Add ***next*** to ***dead***.

*If* ***current*** has only one adjacent vertex then:

Add ***current*** to ***dead***.

Add ***next*** to ***dead***.

Set ***current*** to ***next***.

Continue.

*Else* do:

Remove ***next*** from ***pat***h.

*If* ***next*** was not the last vertex pushed to ***all*** then:

Push ***next*** to all.

Add ***current*** to ***dead***

Add ***next*** to ***dead***.

Set ***current*** to the last vertex pushed to ***path*** and continue;

Calculate the distance traveled along the shortest path.

Display the results.

**Example Run**

Given the files graph\_input.txt and direct\_distance.txt, hard coded in class HelperFunctions lines 31 and 89 respectively, construct an AdjacencyMapGraph from graph\_input.txt and a HashMap from direct\_distance.txt. Then the starting vertex is gotten from the user. If the user enters an invalid vertex, the program will loop until a valid vertex is entered. Next, the destination vertex ‘Z’ is set and the first algorithm is executed followed by the second, and the results are displayed to the console. An example input, and output is given below.

*Run One: Run Two:*

Enter a node:

g

Algorithm 1:

Sequence of all nodes: G -> H -> T -> U -> T -> H -> L -> Z

Shortest path: G -> H -> L -> Z

Distance traveled: 359

Algorithm 2:

Sequence of all nodes: G -> H -> T -> U -> T -> H -> L -> Z

Shortest path: G -> H -> L -> Z

Distance traveled: 359

Enter a node:

x

That vertex is not in the graph.

Enter a node:

j

Algorithm 1:

Sequence of all nodes: J -> K -> Z

Shortest path: J -> K -> Z

Distance traveled: 310

Algorithm 2:

Sequence of all nodes: J -> I -> L -> Z

Shortest path: J -> I -> L -> Z

Distance traveled: 278

**Algorithm Analysis**

An overall runtime for these algorithms is out of the scope of this study but we will give it a shot just for fun. We have the main while loop, that in worst case runs through all vertices of the graph, so worst case is O(n) for that loop. Nested in that we have a for loop that checks the edges of the current vertex, it runs in O(du) time. Inside of that we have the insert method of the HeapPriorityQueue that runs in O(log n) time, and it is executed only when the adjacent vertex is not in the HashSet. Every method after that runs in O(1) constant time except the one call to the contains method of the shortest path Stack, which runs in O(n). So, the biggest contributors to the asymptotic runtime of the algorithm is the outer while loop and the for loop that checks the edges of the current vertex, so we will limit our analysis to these two loops. For each iteration of the outer while loop, the for loop executes once for each edge of the current vertex. Therefore, we can construct a total run time as such:

Where n is the number of times the while loop executes, d is the degree of the current vertex, and c a constant. One case when the while loop will execute the maximum number of times is when each vertex is connected to only one other vertex and the graph is in a straight line from starting vertex to the destination vertex, or some crazy setup that would cause maximum backtracking, but we will not investigate that in this analysis. When the graph is in this straight-line configuration, the degree of the current vertex is one at the start, and two at every other node except the destination node, at which time the algorithm will stop. This gives the following function, ignoring the constant term:

This would be O(n2). The next case we will investigate is when the d’s are maximized. This happens when each vertex is connected to every other vertex, and the while loop would execute the most times when the starting vertex is furthest from the destination vertex than the rest of the vertices, with each ‘next’ vertex being the next furthest away, which is actually difficult to visualize. None the less, this would give the following function:

This leads us to a runtime of O(n3), and since the while loop and the for loop execute the most times in this graph configuration, this is the upper bound for the runtime of the algorithms. While this analysis is definitely incomplete, it gives us a good estimation of the running times of the algorithm in a few edge cases that maximize the run time.

**Conclusion**

The algorithms sometimes take the same path and sometimes take different paths. This is to be expected because they use different selection strategies when choosing the next vertex, and because they are heuristic algorithms, they are not required to always find the correct ‘shortest’ path. The data structures chosen to solve this problem serve their purpose well. The AdjacencyMapGraph has the all-around best run times of the other Graph implementations, and provides numerous functions that allow the creation of, access to, iteration through, removal of, and counting of the vertices and edges. The HeapPriorityQueue possesses O(log n) insertion and removal of the minimum element, in our case the closest vertex chosen by either algorithm one or two’s selection strategy. This was better than putting all of the adjacent vertices in a collection and sorting them, because common sorting methods are O(n log n) at best, and then we would still have to retrieve them. The choice of a HeapPriorityQueue for these operations was the best choice. Our choice of a HashMap to store the direct distance to the destination node was also the best choice. It allowed for constant time retrieval of that distance by ‘getting’ the value associated with the vertex label as the key. The HashSet also served us well as we utilized its constant time addition and membership testing. With the amount of times we needed to check for membership in these algorithms, the HashMap played a huge roll in keeping the algorithms efficient.

1. https://en.wikipedia.org/wiki/Heuristic\_(computer\_science) [↑](#footnote-ref-1)
2. Data Structures and Algorithms in Java. 6th edition. Goodrich, Tamassia, Goldwasser. [↑](#footnote-ref-2)
3. Data Structures and Algorithms in Java. 6th edition. Goodrich, Tamassia, Goldwasser. [↑](#footnote-ref-3)
4. Data Structures and Algorithms in Java. 6th edition. Goodrich, Tamassia, Goldwasser. [↑](#footnote-ref-4)