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() 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. The methods provided by java.util.Stack and the one java.util.Vector method 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.

*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.

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. 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:*

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)