Section 5: Dijkstra's Algorithm (Continued) and Heaps

Amy Zhao

February 23, 2024

1 Revisiting Dijkstra's Algorithm

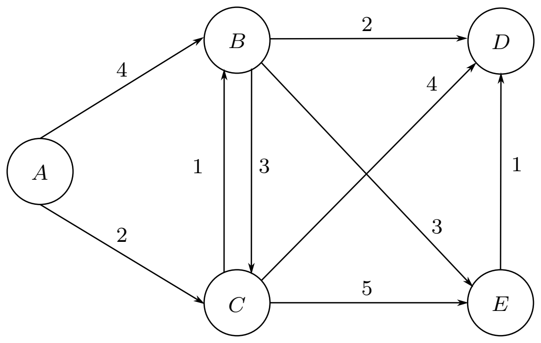
We present Dijkstra's algorithm again which explicitly uses a priority queue to find the shortest paths from a source vertex s. By searching for the next closest node outside a known or explored region, the algorithm minimizes path distances.

1.1 Highlighting Priority Queues

```
Input: Graph $G = (V, E)$ given as an adjacency list, starting vertex $s$
Post : Shortest distances from $s$ to $v \in V$, stored in dist$[v]$

Function Dijkstra($G = (V, E)$, $s$):
    for all $v \in V$ do
        dist$[u] \leftarrow \infty$
        prev$[u] \leftarrow$ null
    end

    dist$[s] \leftarrow 0$
    $Q \leftarrow$ new priority queue storing $V$ with distances as keys

    while $H$ is not empty do
        $u \leftarrow$ ExtractMin($Q$)

        for all edges $(u, v) \in E$ do
            if dist$[v] >$ dist$[u] + \ell_{uv}$ then
                dist$[v] \leftarrow$ dist$[u] + \ell_{uv}$
                prev$[v] = u$
                ChangeKey($Q$, $v$, dist$[v]$)
        end
    end
```

1.2 Worked Example

Here, we provide a full worked example of Dijkstra's algorithm using a priority queue. Consider the directed graph depicted below and the source vertex A.



The priority queue is initialized using distances as keys:

$$Q = \{(A, 0), (B, \infty), (C, \infty), (D, \infty), (E, \infty)\},$$

v	A	B	C	D	E
dist$[v]$	0	∞	∞	∞	∞

where each pair is of the form (vertex, distance). Q is non-empty, so we delete the entry with the minimum key and consider the neighbors. In this example, we dequeue $(A, 0)$ and consider $(A, B) \in E$ and $(A, C) \in E$.

dist$[B] = \infty >$ dist$[A] + 4$, so we update dist(B) to be 4 and decrease the corresponding key in the queue:

$$Q = \{(B, 4), (C, \infty), (D, \infty), (E, \infty)\},$$

v	A	B	C	D	E
dist$[v]$	0	4	∞	∞	∞

We do the same for (A, C): dist$[C] = \infty >$ dist$[A] + 2$, so we update dist$[C]$ to be 2 and modify the priority queue:

$$Q = \{(C, 2), (B, 4), (D, \infty), (E, \infty)\}.$$

v	A	B	C	D	E
dist$[v]$	0	4	2	∞	∞

Notice that the first element in the queue is now vertex C and its shortest distance from A; the priority queue re-balances itself so that the optimal value is first.

We continue to dequeue: $(C, 2)$ is removed, and we consider the neighbors of C. With the edge (C, B), we consider dist$[B]$ and see that dist$[B] = 4 >$ dist$[C] + 1$. dist$[B]$ is updated to be 3, and the queue is modified:

$$Q = \{(B, 3), (D, \infty), (E, \infty)\}.$$

v	A	B	C	D	E
dist$[v]$	0	3	2	∞	∞

We also encounter vertex E through edge (C, E); dist$[E] = \infty >$ dist$[C] + 5$, so we update the distance from A to E and the queue:

$$Q = \{(B, 3), (E, 7), (D, \infty)\},$$

v	A	B	C	D	E
dist$[v]$	0	3	2	∞	7

Vertex C's other neighbor is D, so we look at the edge (C, D). dist$[D] = \infty >$ dist$[C] + 4$, so we update the priority queue and distance array accordingly:

$$Q = \{(B, 3), (D, 6), (E, 7)\}.$$

v	A	B	C	D	E
dist$[v]$	0	3	2	6	7

Vertex C has no more unexplored neighbors, so we move on and dequeue the next unexpanded vertex. We look at the edges which connect vertex B to an unexplored neighbor. (B, D) has edge weight 2 and dist$[D] = 6 >$ dist$[B] + 2$, so we make updates again:

$$Q = \{(D, 5), (E, 7)\}.$$

v	A	B	C	D	E
dist$[v]$	0	3	2	5	7

$\text{dist}[E] = 7 > \text{dist}[B] + 3$, so we continue to update:

$$Q = \{(D, 5), (E, 6)\}.$$

v	A	B	C	D	E
dist$[v]$	0	3	2	5	6

The next vertex in the queue has no outgoing edges, so we don't take any action:

$$Q = \{(D, 5)\}.$$

v	A	B	C	D	E
dist$[v]$	0	3	2	5	7

Finally, the last vertex is taken off the queue. There is an outgoing edge to D, but we also don't take any action because vertex D has already been explored. At this point, the queue is empty, and we have the lengths of the shortest paths with vertex A as the source:

$$Q = \{\}.$$

v	A	B	C	D	E
dist$[v]$	0	3	2	5	7

1.3 Modifying the Input Graph

Last week, we saw some examples of problems where we modified Dijkstra's algorithm itself to compute other solutions related to the shortest path problem. This week, we take a look at modifying the input graph in order to obtain the shortest path using Dijkstra's without modifications.

1.3.1 Example: Tolls

Suppose each vertex v has an associated toll w_v which we must add to the total when computing path length. We need to consider how we can account for these tolls when running Dijkstra's algorithm. The idea is to add edges and vertices such that Dijkstra's will be forced to take the route which accounts for the tolls.

We replace each vertex v with two vertices v_{in} and v_{out}, connected by a directed edge $(v_{\text{in}}, v_{\text{out}})$ with edge length equal to the toll. For each directed edge (u, v) in the original graph, we make an adjustment by replacing it with $(u_{\text{out}}, v_{\text{in}})$. We claim that the shortest path between s_{in} and v_{out} is the shortest path from s to v in the original graph.

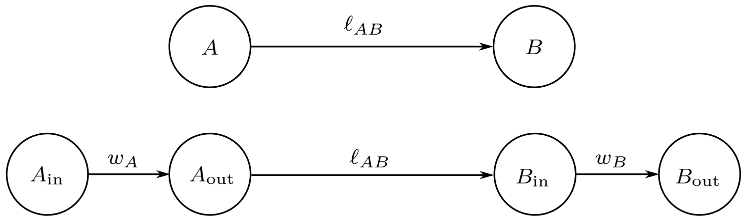


Figure 1.1: Example where the original graph is modified to correctly account for tolls. Vertices A and B are split into two nodes each; each pair is connected with a single directed edge.

The effect of splitting each vertex into an in-node and out-node, connected by a directed edge with a length equal to the toll is equivalent to accounting for the toll of the vertex when computing path lengths. When running Dijkstra's, the algorithm is forced to take the path between v_{in} and v_{out}, which means the shortest distance computation will add the toll value to the total shortest distance sum. It cannot take any other path because there is no edge or path connecting v_{in} to w_{out} for some vertices v and w in the graph.

1.3.2 Example: Complexity

Suppose each path has a complexity cost equal to the number of edges contained within the path. We compute the path length by summing up the edge lengths and adding this complexity cost.

This graph doesn't require too many modifications; we simply add one to each edge length to account for the complexity cost. Notice that we can calculate the path length using:

$$\text{path length} = \left(\sum_{e \in P} \ell_e\right) + |P|,$$

where P represents the set of edges in the shortest path. The summation contains $|P|$ terms, which means we can redistribute as follows:

$$\text{path length} = \sum_{e \in P} (\ell_e + 1).$$

Running Dijkstra's algorithm will correctly compute the shortest path in the original graph.

2 Heaps

Heaps, also known as priority queues, are data structures which keep track of changing objects by accounting for key values. These key values are often used to identify the object with the smallest or largest value, which can be determined quickly as the "top" element as the data structure continuously re-balances itself.

2.1 Example: Heapify and Conquer

Suppose you decide to come up with your own heapify algorithm. You realize that you can impose a heap-like structure into an array by ensuring that the smallest element is in the first position, the next two smallest elements are the next two elements in the array (in any order), the next four smallest elements are in the next four positions of the array (in any order), and so on. To generalize this, positions 2^i through $2^{i+1} - 1$ are occupied by the elements of order statistic 2^i through $2^{i+1} - 1$ in any order.

Assume that $n = 2^k - 1$ for some integer $k \geq 0$. Given a list $A[1, \dots, n]$, we want to create an algorithm which will output the elements in the order as described above; the algorithm should run in $O(n)$ time. It turns out that we can solve this problem using the partitioning algorithm we learned about in divide-and-conquer!

Starting with $i = k$ and decrementing until $i = 2$, we find the 2^{i-1}-th smallest element x_i and partition around x_i so that all elements in the first half are less and all elements in the second half are greater than x_i.

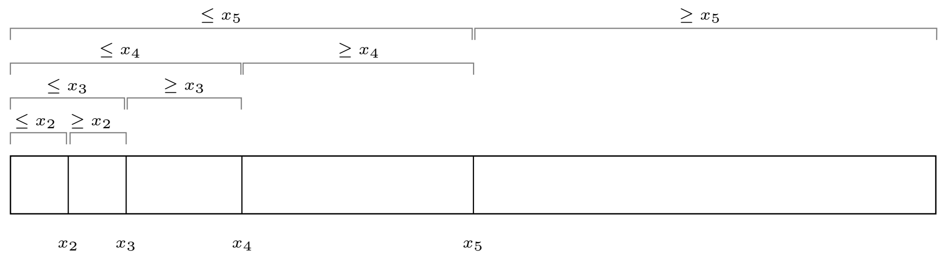


Figure 2.1: Example of the array order when $k = 5$. The algorithm first partitions around x_5 so that the first half of the entire array contains values less than or equal to x_5. The algorithm then identifies x_4 and partitions accordingly. The same is repeated for x_3 and x_2.

After iteration i of the algorithm, the array is ordered such that $A[x] \leq A[y]$ for $x < 2^{i-1} \leq y < 2^i$, which follows simply from the partitioning algorithm. To analyze the runtime, we need to consider the runtime of finding the 2^{i-1}-th smallest element and the runtime of partitioning. Both take $O(2^i)$ time, so the overall runtime is given as follows:

$$O\left(\sum_{i=2}^{k} 2^i\right) \leq O(2^{k+1}) \leq O(n).$$

2.2 Example: Heap Sort

Heaps are also very useful when it comes to sorting. Given an unsorted array, suppose we want to obtain the sorted version. This algorithm converts the array into a max heap and repeatedly deletes the root node of the heap, replacing it with the last node in the heap. It then invokes heapify to maintain the heap invariant.

```
Input: Unsorted array $A$
Post : Sorted version of $A$
Function Heapify($A$, $n$, $i$):
    largest $\leftarrow i$
    $\ell \leftarrow 2i+1$, $r \leftarrow 2i+2$

    if $\ell < n$ and $A$[largest] $< A[\ell]$ then
        largest $\leftarrow \ell$

    if $r < N$ and $A$[largest] $< A[r]$ then
        largest $\leftarrow r$

    if largest $\neq i$ then
        $A[i]$, $A$[largest] $\leftarrow$ $A$[largest], $A[i]$
        Heapify($A$, $n$, largest)

Function HeapSort($A$):
    $n \leftarrow$ length($n$)

    for $i \in$ range($\lfloor \frac{n}{2} \rfloor - 1$, $1$, $-1$) do
        Heapify($A$, $n$, $i$)
    end

    for $i \in$ range($N-1$, $0$, $-1$) do
        $A[i]$, $A[0] \leftarrow A[0]$, $A[i]$
        Heapify($A$, $i$, $0$)
    end
```

Heapsort runs in $O(n \log n)$ time.

2.3 Example: Median of Integer Stream

Suppose we consider a situation where we need to keep track of a running median. That is, we are presented a stream of numbers one by one, and we need to update the median of the numbers we have seen thus far. Using heaps, we can solve this problem with an algorithm which runs faster than recomputing the median at each iteration.

The general idea is to use two heaps, which we will denote H_1 and H_2. We want H_1 to be a max heap and H_2 to be a min heap. As we process integers, we want to keep both heaps either the same length or off by one, and we want to make sure that all of H_1 is less than all of H_2.

When we process a new element x, we make a comparison with the maximum element y of H_1 and the minimum element z of H_2, which we can compute in logarithmic time. If $x < y$, we add x to H_1; if $x > z$, we add x to H_2. In the case where the heaps become off by two in length, we move the extrema element and re-insert it in the other heap.

After we finish processing the stream of integers, the median as calculated as the average of H_1's maximum and H_2's minimum if the two heaps are equal size. If the heap sizes are off by one, the median is the extrema element in the larger heap.