# COMP4019 - Lab Session 5 – Graphs; Heaps

Xavier Carpent & Ian Knight

November 11, 2021

## 1 Shortest Path Algorithms

Below are the pseudocode for the single-source all-destinations Dijkstra and all-sources all-destinations Floyd-Warshall algorithms. Make sure you understand them properly. You may use the graph of Figure 1 to execute the algorithms as a way to practice them.

Assume the following function specifications (the costs $c_*$ depend on the implementation of the underlying function and are to be discussed below):

| Function | Specification | Cost |
|---|---|---|
| $\textsc{emptyArray}(n)$ | returns an empty array of size $n$ | $O(n)$ |
| $\textsc{emptyMatrix}(m, n)$ | returns an empty matrix of size $m \times n$ | $O(mn)$ |
| $\textsc{neighbours}(v)$ | returns the list of neighbours of $v$ in the context of a graph $G$ | $c_n$ |
| $\textsc{PriorityQueue}$ | returns an empty priority queue | $O(1)$ |
| $Q.\textsc{isEmpty}()$ | returns true iff $Q$ is empty | O(1) |
| $Q.\textsc{insert}(x, p)$ | inserts $x$ in $Q$ with priority $p$ | $c_{ins}$ |
| $Q.\textsc{minExtract}()$ | removes and returns the element in $Q$ with the smallest priority | $c_{ext}$ |
| $Q.\textsc{decreaseKey}(x, p)$ | updates the priority of $x$ in $Q$ to $p$ | $c_{dec}$ |

```
function Dijkstra(V, E, s)
    Q ← PriorityQueue()
    dist ← emptyArray(|V|)
    dist[s] ← 0
    for all v ∈ V do
        if v ≠ s then
            dist[v] ← ∞
        Q.insert(v, dist[v])
    end for
    while not Q.isEmpty() do
        v ← Q.minExtract()
        for all u ∈ neighbours(v) do
            dist[u] = min(dist[u], dist[v] + w(v, u))
            Q.decreaseKey(u, dist[u])
        end for
    end while
    return dist
```

```
function FloydWarshall(V, E)
    dist ← emptyMatrix(|V|, |V|)
    for (u, v) ∈ E do
        dist[u][v] ← w(u, v)
    end for
    for v ∈ V do
        dist[v][v] ← 0
    end for
    for k ∈ V do
        for u ∈ V do
            for v ∈ V do
                altdist ← dist[u][k] + dist[k][v]
                dist[u][v] ← min(dist[u][v], altdist)
            end for
        end for
    end for
```

## 2 Complexity of Shortest Path Algorithms

1. What are the costs $c_{ins}$, $c_{ext}$, $c_{dec}$ if you use a *balanced binary search tree* (like Red-Black trees) to implement the priority queue?

2. What is the complexity of $\textsc{Dijkstra}$, and how does it depend on $c_n$?

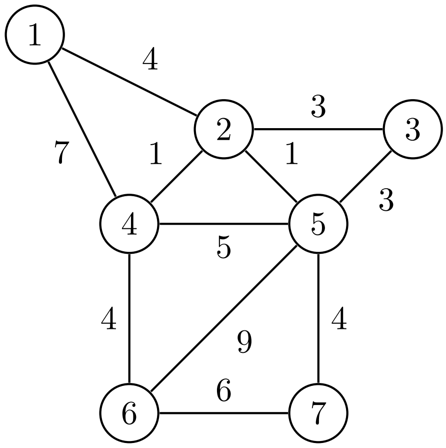


Figure 1: A weighted undirected graph $G$.

3. Now consider that you use an implementation of priority queue for which $c_{ins} = O(1)$, $c_{ext} = O(\log n)$, $c_{dec} = O(1)$ (with $n$ the number of elements in the queue). How does the complexity of Dijkstra change?

4. What could be $c_n$ for a graph represented by (a) an adjacency list or (b) an adjacency matrix?

5. Consider both dense and sparce graphs; comment on the overal complexity of Dijkstra in both cases.

6. What is the complexity of FloydWarshall?

7. How does it compare to the complexity of $|V|$ calls to Dijkstra? Conclude.

# 3 Finding Paths

How would you extend the Dijkstra pseudocode above to also return the paths in addition to the lengths? Aim to come up with a solution that does not impact the complexity.

# 4 Heap Sort

String together the priority queue operations described in the first exercise to implement (in pseudocode or your favorite programming language) a *heap sort*, that is, a list sorting algorithm using a heap.

If a *balanced binary search tree* is used to implement the priority queue, what is the overal complexity? What about using the implementation of priority queue evoked in point 3 from exercise 2? *Hint: you may find it useful to use the Sterling approximation formula (in its* log *form):* $\ln(n!) \approx n \ln n - n + \Theta(\log n)$.