COMP4019 - Solutions Session 6 – Graphs; Heaps

Xavier Carpent & Ian Knight

November 11, 2021

2 Complexity of Shortest Path Algorithms

1. $\Theta(\log n)$ for all.
2. $O(n\,(c_{ext} + c_n) + mc_{dec})$, with $n = |V|$ and $m = |E|$; for BST implementation: $O(n(\log n + c_n) + m \log n)$.
3. $O(n\,(\log n + c_n) + m)$.
4. (a) $c_n = O(degree(v))$, i.e. linear in the *degree* (=number of neighbours) of a node v; we have that $\sum_{v \in V} degree(v) = 2m$, which makes the inner loop execute $O(m)$ times. (b) $c_n = O(n)$ (worse for sparse graphs).
5. General case is $O(n \log n + m)$. Dense graphs (meaning $m = O(n^2)$), overal complexity is $O(n^2)$. Sparse graphs (with $m = O(n)$), overal complexity is $O(n \log n)$.
6. Triple nested loop dominates, $O(n^3)$.
7. For sparse graphs, n applications of Dijkstra has *better* asymptotic complexity $O(n^2 \log n)$. For dense graphs, they both have complexity $O(n^3)$. In both cases, Floyd-Warshall has the added benefit of handling negative weights (provided negative cycles do not exist). In addition to this, due to the overhead in Dijkstra (specifically in the Fibonacci heap version of it), Floyd-Warshall is sometimes a better choice if the shortest path length between all pairs has to be computed.

3 Finding Paths

The idea is to keep track of the *predecessor* of a node for each shortest path, in addition to distances.

```
function DIJKSTRA(V, E, s)
    Q ← PRIORITYQUEUE()
    dist ← EMPTYARRAY(|V|)
    dist[s] ← 0
    prev ← EMPTYARRAY(|V|)
    prev[s] ← None
    for all v ∈ V do
        if v ≠ s then
            dist[v] ← ∞
            prev[v] ← None
        Q.INSERT(v, dist[v])
    end for
    ...
    ...
    while not Q.ISEMPTY() do
        v ← Q.MINEXTRACT()
        for all u ∈ NEIGHBOURS(v) do
            newdist ← dist[v] + w(v, u)
            if newdist < dist[u] then
                dist[u] ← newdist
                Q.DECREASEKEY(u, dist[u])
                prev[u] ← v
        end for
    end while
    return dist, prev
```

To print a path, simply follow the predecessors from the target t to the source s and reverse:

```
path = emptyList()
v ← t
while v ≠ s do
    path.insert(v)
    v ← prev[v]
end while
reverseList(path)
```

4 Heap Sort

```
function HeapSort(L)
    S ← emptyArray(|L|)
    H ← emptyHeap()
    for i ∈ 1..|L| do
        H.insert(L[i])
    end for
    for i ∈ 1..|L| do
        S[i] = H.minExtract()
    end for
    return S
```

Complexity is $O(n(c_{ins} + c_{ext}))$, with $n = |L|$. This is $O(n \log n)$ for both BST and Fibonacci heap. In the former case this is called *tree sort*; the second case is *heap sort*, however the typical incarnation of heap sort is an *in-place* sort, requiring no additional auxillary memory, which is much preferable in practice.

The Sterling approximation formula can be used to simplify the cost of adding n elements in a progressively growing data structure:

$$T(n) = \sum_{i=1}^{n} \Theta(\log i) = \Theta(\log \prod_{i=1}^{n} i) = \Theta(\log n!) \approx \Theta(n \log n).$$