DMA 12g

Carl Dybdahl, Patrick Hartvigsen, Emil Chr. Søderblom

January 16, 2017

Part 1

(a)

We need to show that given a path e_n with $0 \leq n < k$, we can rewrite $-\log(\prod_{0 \leq n < k} w(e_n)) = \sum_{0 \leq n < k} -\log(w(e_n))$.

We will do this by induction on k.

When $k = 0$, we have:

$$\begin{aligned} -\log(\prod_{n \in \emptyset} w(e_n)) &= -\log(1) \\ &= 0 \\ &= \sum_{n \in \emptyset} -\log(w(e_n)) \end{aligned}$$

When $k = j + 1$, we have:

$$\begin{aligned} -\log(\prod_{0 \leq n < j+1} w(e_n)) &= -\log(w(e_j) \cdot \prod_{0 \leq n < j} w(e_n)) \\ &= -\log(w(e_j)) - \log(\prod_{0 \leq n < j} w(e_n)) \\ &= -\log(w(e_j)) + \sum_{0 \leq n < j} -\log(w(e_n)) \\ &= \sum_{0 \leq n < k} -\log(w(e_n)) \end{aligned}$$

(b)

To find the most probable path, we wish to maximize $\prod_{0 \leq n < k} w(e_n)$. This is equivalent to minimizing $-\log \prod_{0 \leq n < k} w(e_n)$, which we have just shown is the same as $\sum_{0 \leq n < k} -\log(w(e_n))$. This is equivalent to finding the shortests path with a weight function $w'(e) = -\log(w(e))$. We use Dijkstra's algorithm for this:

pathfind.fsx

```
type Graph = {
```

```
        size : int;
        edges : Set<int * int>;
        weight : (int * int -> float)
    }

type Search = float * int

let setToMap n s =
    [0 .. n] |> List.map (fun k ->
        (k, s |> Set.filter (fun (a, _) -> a = k) |> Set.map snd)
    ) |> Map.ofList

let pathfind (g : Graph) (a : int) (b : int) =
    let neighbours = setToMap g.size g.edges
    let mutable queue = Set.ofList[(0.0, a)]
    let dists = Array.init g.size (fun _ -> infinity)
    while not queue.IsEmpty do
        let (dist, node) = queue.MinimumElement
        queue <- queue.Remove (dist, node)
        if dists.[node] > dist then
            dists.[node] <- dist
            for neighbour in Set.toSeq neighbours.[node] do
                let cost = - log(g.weight(node, neighbour))
                queue <- queue.Add (dist + cost, neighbour)
    if dists.[b] = infinity then
        None
    else
        let reversed =
            g.edges |> Set.map (fun (x, y) -> (y, x)) |> setToMap g.size
        let mutable path = [b]
        while path.Head <> a do
            let neighs = reversed.[path.Head] |> Set.toList
            let prev = neighs |> List.minBy (fun node -> dists.[node])
            path <- prev :: path
        Some path
```

Part 2

(a)

$\delta_{BFS}(s, v)$ represents the length of the shortest path from s to v, whereas $\delta_{SP}(s, v)$ represents the cost of the lowest-cost path from s to v. However, when $w(e) = 1$, the cost of a path and its length coincides, and as such δ_{BFS} and δ_{SP} will be equal.

(b)

A node is black during BFS if it has been visited. This means that the processing for that node is done.

The set S represents the nodes for which the shortest distance has been determined. Similarly to a node in BFS being black, a node being in S means that the processing for that node is done.

(c)

The loop invariant states that at the start of each iteration of the while loop, `v.d = `δ`(s, v)` for each vertex `v`. When Dijkstra's algorithm visits a node `u`, it has already visited all nodes `w` with a distance $\delta(s, w)$ less than $\delta(s, u)$. This means that any node in V with distance less than $\delta(s, u)$ is already in S, and so the nodes `v` in $V \setminus S$ must have a distance greater than or equal to that of `u` from `s`.