01:198:344 - Homework V

Kev Sharma - kks107, Section 08

April 12, 2021

Partners collaborated with on this assignment: 1) Joel Martinez - Section 06. I am submitting alone. Joel and I discussed problem 5 together.

1.
```
procedure FullyInward(A)
    row ← 1, col ← 1
    candidate ← -1
    while true do
        if row = col then
            ++col
            continue to next iteration of loop
        if col > A.length then                                  ▷ A.length = |V|
            candidate ← row
            break out of loop
        ▷ Here we deduce which vertices aren't candidates from A[row][col] value. ◁
        if A[row][col] = 0 then
            ++col
        else
            row ← col
    ▷ Check whether candidate at whom col surpassed |V| is fully inward:
      A[row][candidate] for all rows in A (s.t. row != candidate) should be 1. ◁
    for row←1, . . . , A.length do
        if row = candidate then
            continue
        if A[row][candidate] = 0 then
            return "no solution"
    return candidate
```

2.

```
procedure CheckBipartite(G)
    s ← G.V[0]
    ▷ s gets first vertex from set V of G. ◁
    dist ← BFS(G,s)
    ▷ Now we know dist(s,v) for all vertices v in G.V ◁
    for all y ∈ G.V do
        if y = s then
            continue to next iteration
        for all x ∈ In(y) do
            if (dist(s,y) mod 2) = (dist(s,x) mod 2) then
                return false
    return true
▷ Runtime = O(|E|) because we visit each vertex in In(x) for all x in V. ◁
```

3.

```
3) Part 1

Iteration | Exploring_vertex | (v in Out(exploring_vertex), d(v))
------------------------------------------------------------------

1              s              [(z,3), (x,7)]

2              z              [(x,7), (y, 8), (t,6)] // d(z) is now dist(s,z) = 3

3              t              [(x,7), (y, 8)]        // d(t) is now dist(s,t) = 6, exploring t relaxed no edges

4              x              [(y,8)]                // d(x) is now dist(s,x) = 7, exploring x relaxed no edges

5              y              [ ]                    // d(y) is now dist(s,y) = 8, exploring y relaxed no edges

loop terminates

```

```
3) Part 2

Let G = (V,E,W)
where
        V = [s,a,b,p]
        E = [(s,a), (s,b), (b,a), (a,p)]
        W = [(s,a,50), (s,b,100), (b,a,-90), (a,p,200)] // the third item is the edge weight

The graph looks as follows:

                        50              200
                 s  -----------> a -----------------> p
                 |               ^
                 |               |
                 |               |-90
                 |               |
                 |---------> b ---
                     100

Note that Dijkstra relaxes each edge at most once.

After Dijkstra(G,s): d(p) = 250, but dist(s,p) = 210

Justification:

0) d(s) = 0, d(v != s) = infinity, s is put in queue.
1) While exploring s, a and b are explored and put in queue -> [(a,50), (b,100)]
        // where queue pairs are (v, d(v))

2) Since min item from queue is a, we explore a next.
3) While exploring a, edge (a,p) is relaxed and the loop completes with the following queue -> [(b,100), (p,250)]

4) Since min item from queue is b, we explore b next.
5) While exploring b, edge (b,a) is relaxed and d(a) is set to 10.
        a is not added to the queue since it was explored already.

6) We complete the Dijkstra after removing p.

Problem: Edge (b,a) was relaxed but vertex a was not added to the queue a second time,
        hence ensuring that all outgoing edges from a are relaxed at most once (rule).

This guarantee that all edges are relaxed at most once, did not allow us to change d(p) to its true value of dist(s,p).
The path s->b->a->p which sets d(p) to its real value of dist(s,p)=10 was not traversed since then we would have relaxed edge (a,p) twice.
The first time we relaxed it was in step 3.


Conclusion: In order to restrict Dijkstra's execution of operation Q.decrease-key(v, d(v)) to atmost |E| times,
                we cannot allow negative edge weights in G.

        If we did allow negative edge weights, this operation Q.decrease-key(v,d(v)) would far exceed |E|
        because some edges would have to be relaxed more than once to ensure that all d(v) = dist(s,v).
```

```
Problem 4
----------

Fancy Data Structure D:

D.insert                 = O(log(n))
D.delete-min()                   = O(log(n))
D.decrease-key(v,k)      = O(1)


Total Runtime = D.insert runtime * num times Dijkstra executes insert +
               D.delete-min() * num times Dijkstra executes delete-min() +
               D.decrease-key(v,k) * num times Dijkstra executes decrease-key(v,k)

               = O(|V|*log(n)) + O(|V|*log(n)) + O(|E|)

               = O(|V|*log(n)) + O(|E|)

        note that n = |V|

Hence Fancy ds Dijkstra yields runtime of O(|V|log(|V|)) + O(|E|)

=====================================================================
Let Fancy dijkstra      = O(|V|log(|V|)) + O(|E|)
And min-heap dijkstra   = O(|E|log(|V|))

Recall  1. |E| is lower bounded by |V|.
        2. |E| is upper bounded by |V|^2.


Case 1: Graph is sparse (extreme)
-----------------------------------------------
then |E| roughly equals |V|
Fancy dijkstra  = O(|V|log(|V|)) + O(|V|)
                = O(|V|log(|V|))

min-heap dijks          = O(|V|log(|V|))

When the graph is extremely sparse, both perform roughly the same.



Case 2: Graph is dense (extreme)
-----------------------------------------------
then |E| roughly equals |V|^2
Fancy Dijkstra  = O(|V|log(|V|)) + O(|V|^2)
                = O(|V|^2)

min-heap dijk   = O(|V|^2 * log(|V|))

When the graph is extremely dense, fancy dijkstra performs slightly better.



Generally:
--------------------
fancy Dijkstra           = O(|V|log(|V|)) + O(|E|)
                 < O(|V|log(|V|)) + O(|E|log(|V|))
                <= O(|E|log(|V|))

Fancy Dijkstra is upper bounded by min-heap dijkstra,
but min-heap dijkstra is not upper bounded by Fancy Dijkstra.
Hence Fancy Dijkstra, for no edge-density, performs worse than min-heap dijkstra.

So if we did have an implementation for Fancy Dijkstra, it would be preferable
to use that implementation over the min-heap dijkstra variant.

This is to be expected given we made one Dijkstra operation faster.
```

4.

5.

```
Proof by Contradiction:
----------------------

Assume that d(s) = 0 and for every edge (x,y) we have: d(x) + w(x,y) >= d(y)
and that there exists a vertex such that d(v) > dist(s,v). (Assumption 1)

Let B = {v exists in V | d(v) > dist(s,v)}
Let child be a vertex in B. (Proposition 1)

Then d(child) > dist(s, child).

Let parent be the vertex that exists in In(child) such that the path from [s to ... to parent to child] forms the minimum path.
That is, dist(s,child) is the distance of the path which 1. starts at s and 2. goes through parent to 3. get to child.

However if d(child) > dist(s,child), then we cannot pick the aforementioned minimum path.
Hence d(child) is the distance of a non-minimum path which
1.starts at s and 2.goes through any vertex in In(child) except for parent to get 3.to child.

Note that any non-minimum path has a distance greater than any minimum path from s to child. (Proposition 2)

Let the value of minimum path k       = d(parent) + w(parent, child)
Let the value of non-minimum path q   > dist(s, child)

By proposition 2, we have k < q.
Equivalently, d(parent) + w(parent, child) < dist(s, child)

Equivalently, substituting Proposition 1 we have
        -> d(parent) + w(parent, child) < dist(s, child) < d(child)
        -> d(parent) + w(parent, child) < d(child) (Final Statement)

Note that *Final Statement* violates the KEY PROPERTY that d(x) + w(x,y) has to be >= d(y).

Specifically, d(parent) + w(parent, child) has to be greater than or equal to d(child) by KEY PROPERTY.
Given our Proposition 1, this KEY PROPERTY is violated.

Hence there is no such vertex child which exists in B.
The cardinality of B, the set of all bad vertices in prompt, is therefore 0.

Therefore, there exists no vertex where d(vertex) > dist(s, vertex).

We have thus proved that d(v) <= dist(s,v) for all vertices v exists in V.





An example:
-----------
Say we had a graph:

              50       20
        s -------> a ---------> c
        |                       ^
        |                       |
        |      100              |70
        -------------->b --------

Then if child = c exists in B, d(child) > dist(s,child)
Hence d(child) > 70. This implies that d(child) is not representative of the path with minimum distance from s to c.

Here d(child) can take on the value 170 using the path s->b->c.
But note that there exists a path s->a->c where the edge (a,c) violates the key property that d(a) + w(a,c) >= d(c).
In particular: d(a) + w(a,c) >= 170 equals 50 + 20 >= 170 which is logically false.

The violation of this key property is not permissible, hence c cannot exist in B.
Since we assumed that c existed in B, but it can't, we have a contradiction of our assumption that there exists a vertex d(v) > dist(s,v).
```