Carlos Mc Gregor c.mcgregormuro@mail.utoronto.ca

CSC373S: Algorithm Design, Analysis & Complexity

Lecture 25

Friday March 10, 2017
based on notes by Denis Pankratov

Assignment 2

Question 3

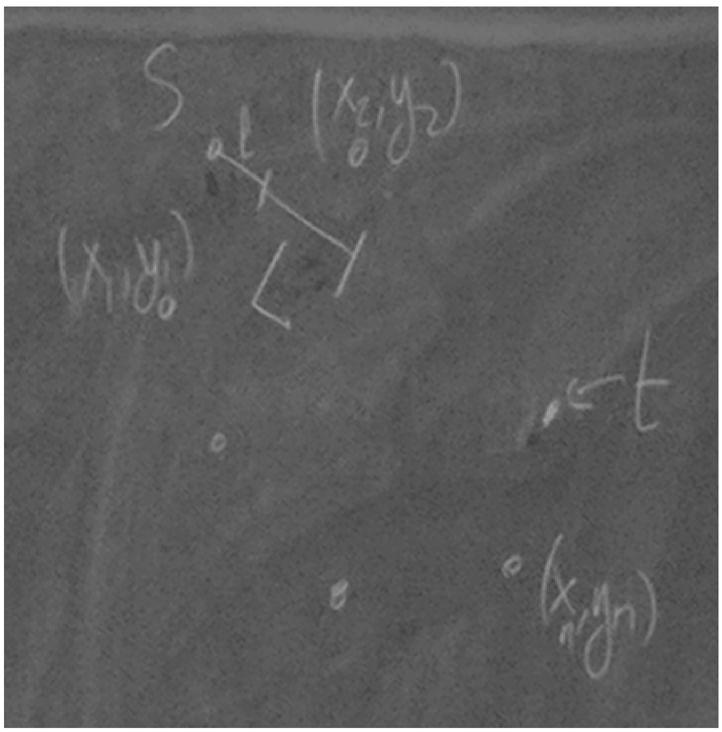

Figure 1: *Visualization of problem*

Approach: Modified Dijkstra (P, n, s, t, l)
$P-$ array of points
$P[i].x, P[i].y$
Distance $2D(P_1, P_2)$ = Euclidean $2D$ distance between points $P_1 \& P_2$

<u>Algorithm:</u>

```
def ModifiedDijkstra(P,n,s,t,l):
    let W be an array of size nxn
```

```
for i = 1 to n:
    for j = 1 to n:
        W[i,j] = max(Distance2d(P[i],P[j]) - 1, 0)
init array L of size n
/* L[i] will contain smallest value of L required to reach i from s
    by edges of length <= L */
for i = 1 to n:
    L[i] = float('inf')
L[S] = 0

Q = PriorityQueue([n]) /* ordered by min value of L[.] */

while Q is not empty:
    V = Q.ExtractMin()
    for u = 1 to n:
        # See Figure 2 (below)
        if max(L[v],W[v,u]) < L[u]:
            L[u] = max(L[v],W[v,u]) /* causes DecreaseKey */

return L[t]
```

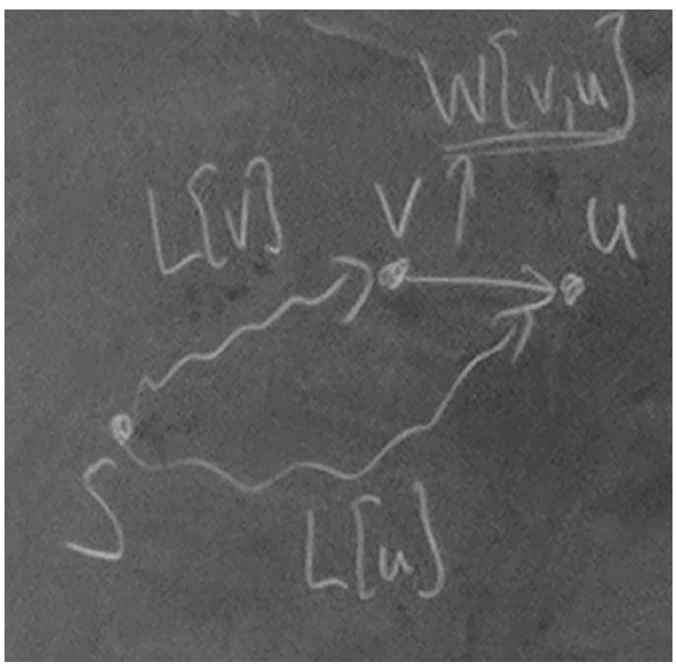

Figure 2: *Visualization of DecreaseKey loop*

Runtime: $n-$ ExtractMin operations
n^2- DecreaseKey operations

If we use binary heaps, we get $\Theta(n^2 \log n)$ runtime.

Array implementation of PQ
$A[i]-$ priority of node i, or NIL if $i \notin PQ$
DecreaseKey takes $\mathcal{O}(1)$
ExtractMin takes $\mathcal{O}(n)$

ModifiedDijkstra runs in $\mathcal{O}(n^2)$ with array input implementation of PQ.

Set of nodes $U-$ processed nodes (nodes taken off the queue)
$T = [n] \setminus U-$ set of *not yet fully processed* nodes

For $u \in U, L[u]$ is computed correctly.
For $u \in T, L[u]$ stores smallest L so that u is reachable from s via edges of length $\leq L$ & using as intermediate nodes only nodes from U.

Question 5

$G = (V, E), c, s, t$ flow network
$c-$ integral
$f-$ max flow, integral
$p : E \to \mathbb{N}$ price

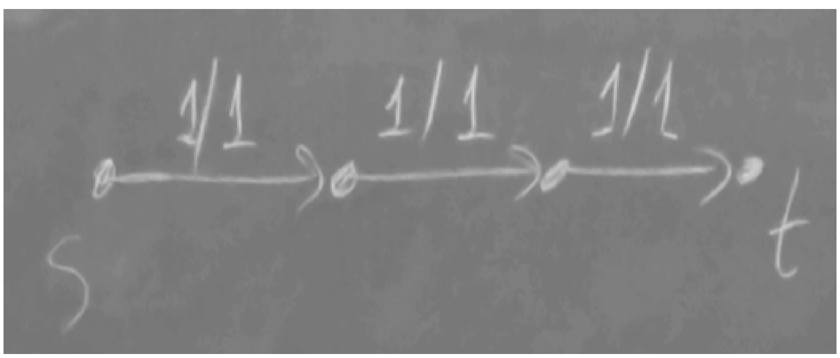

Figure 3: *Visualization of problem*

<u>Algorithm:</u> Considering that $(i, j) \in E^* \Leftrightarrow (i, j) \in E$ or $(j, i) \in E$

```
def ComputeEdges(G, s, t, c, p, f):
  create a weigh function w:E* -> R
  for e in E_f:
    if c_f(e) > 0:
      w(e) = 0

  for e in E:
    if c_f(e) = 0:
      w(e) = p(e)

```

```
# On CLRS, v.pi = prev[v]
Run Dijkstra on (V,E*) w/ weights w from s,
  which returns two arrays prev[−], dist[.]

Ans = []
cur = t

while cur != s:
  if c_f(prev[cur], cur) = 0 and (prev[cur],cur) in E:
    Ans.insert((prev[cur],cur))
  cur = prev[cur]

return Ans
```

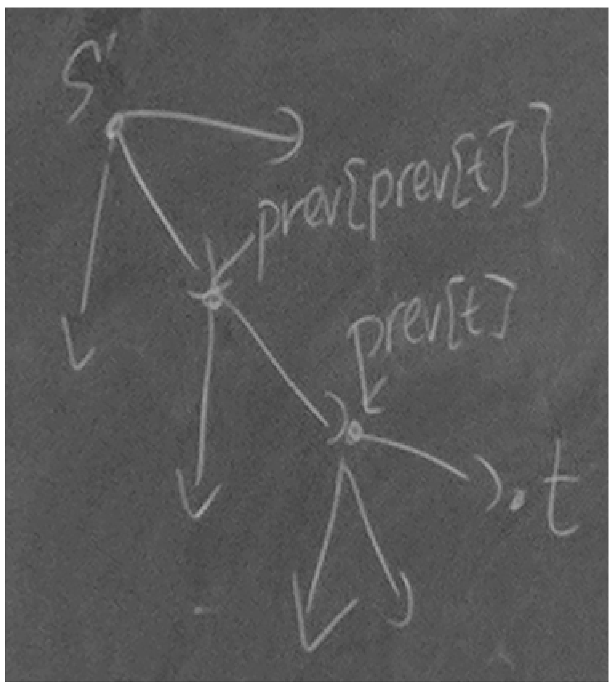

Figure 4: *Visualization of algorithm:* flow is a *function*; flow value is a *real number*

Question 1

(d)

False: Only update edges from a vertex u when u is taken off the queue & u is taken only once off the queue.

(g)

False:

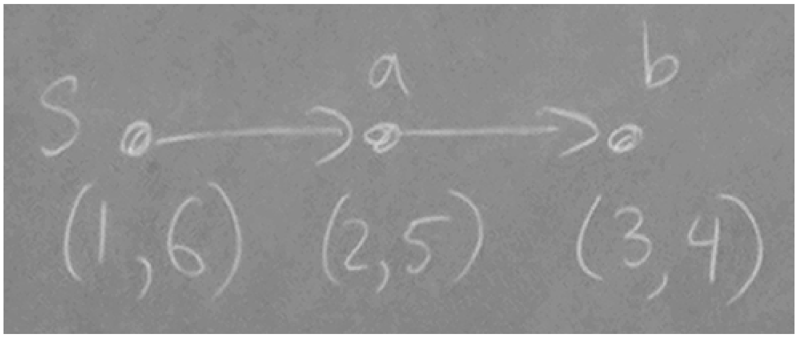

Figure 5: Graph is <u>not</u> strongly connected, so we can only move from *left to right*

Question 4

(e)

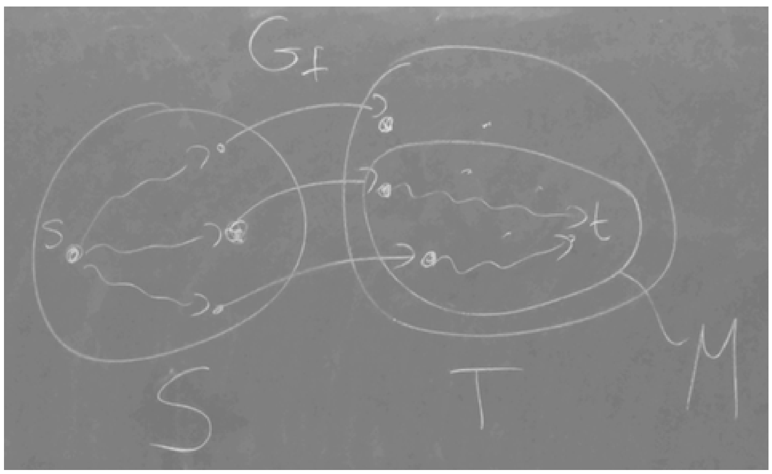

Figure 6: M = set of all nodes in G_f so that t is reachable from those nodes

1. Compute max flow f
2. Compute S = set of nodes reachable from s in $G_f - BFS$
3. Compute G_f^T (transpose of the graph, all edges reversed)
4. Run BFS from t in G_f^t & compute M
5. Collect all edges $e = (u, v) \in E$ so that $u \in S, v \in M$