Algorithmics
Correction Final Exam #4 (P4)

Undergraduate 2^{nd} year S4 – Epita

14 May 2019

Solution 1 **(Restyled Floyd –** ***3 points*****)**

1. The modification is simple. A negative cycle will return a negative cost over the calculated distance from a vertex x to this same vertex x (cycle and negative). It is therefore sufficient to test, when the vertex $x = y$, if the computed distance value is negative. If this is the case, the procedure is disconnected.

2. Again the use is not very complicated. We use the matrix that returns the smaller distances for each pair of vertices (x, y) of the graph. For each vertex x from 1 to n, we compute its eccentricity value by keeping its greatest distance value with the other vertices (the max of its smaller distances). There's nothing for it but to compare the n eccentricities (one for each vertex) and determine the smallest. The vertex to which it belongs is the center of the graph.

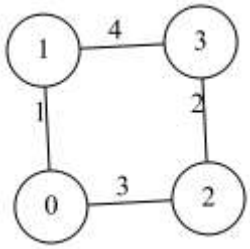
Solution 2 **(MST or not? –** ***2 points*****)**

NO, the spanning subgraph T is not a MST of G!

The stated principle is similar that of Kruskal, but the difference is that the construction is upside down. The graph is split in half each time without worrying about the lowest cost edges.

Let's take the example of the following figure. If the subset $S_1 = \{0, 2\}$ and the subset $S_2 = \{1, 3\}$,we then have the MSTs $T_1 =< S_1, A_1 = \{\{0, 2\}\} >$ and $T2 =< S2, A_2 = \{\{1, 3\}\} >$. if we choose the lowest cost edge $\{0, 1\}$ to connect these two MSTs, we do not get a MST of G since we would have to choose the edge $\{2, 3\}$ instead of the edge $\{1, 3\}$.

Actually, the problem is essentially due to arbitrary splitting of S in two subset of vertices. Also in this example, if the split of S had given $S_1 = \{0, 1\}$ and $S_2 = \{2, 3\}$, there were no problems, the choice of the lowest cost edge (ie $\{0, 2\}$ guaranteed us to obtain a MST T of G.



Solution 3 **(Eat Crepes – *11 points*)**

1. Graph that represents the recipe:

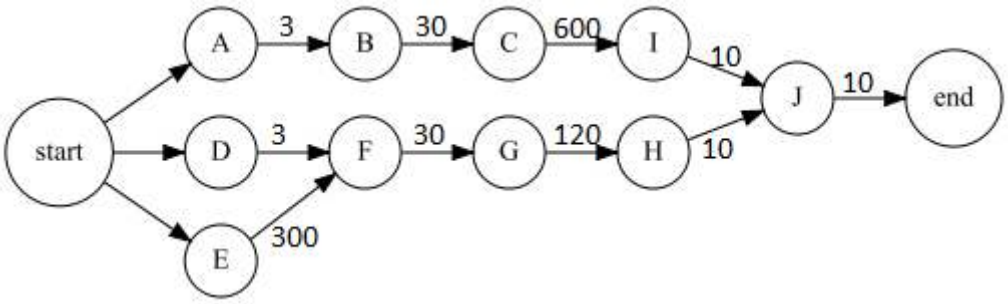


Figure 1: Banana crepe "flambée"

2. **The cook is alone:**

   (a) **Specifications:**
   The function `topologicalSort(G)` returns a topological sort for the acyclic digraph G, where all the vertices are reachable from the vertex 0.

```
def topologicalSort(G):
    # compute half-degrees
    dIn = [0] * G.order
    for s in range(G.order):
        for adj in G.adjlists[s]:
            dIn[adj] += 1

    # 0 is the only vertex without predecessors
    sort = [0]
    first = 0
    while len(sort) < G.order:
        s = sort[first]
        dIn[s] = -1
        for adj in G.adjlists[s]:
            dIn[adj] -= 1
            if dIn[adj] == 0:
                sort.append(adj)
        first += 1
    return sort
```

3. **The cook has found assistants, some tasks can thus be done simultaneously.**

   (a) The longest paths from the first task (*start*) give the early start dates for each task.

   Early start dates for each task of the crepe recipe:

start	A	B	C	D	E	F	G	H	I	J	*end*
0	0	3	33	0	0	300	330	450	633	643	653

   (b) **Minimal duration before we eat our crepe:** 653 secondes (10mn53s).

   (c) To compute the **late start dates**, we must consider the reverse graph and search for the longest paths from the task *end*: the late start date of a task is the difference between the minimum duration of the project and the length of the path obtained.

(d) **Specifications:**
The function `duration(`G`)` computes the minimum duration to complete the project represented by the digraph G.

```
def duration(G):
    """
    return dist: the longest paths in G
    """
    L = topologicalSort(G)
    dist = [0] * G.order
    for i in range(G.order-1):
        x = L[i]
        for y in G.adjlists[x]:
            if dist[x] + G.costs[(x, y)] > dist[y]:
                dist[y] = dist[x] + G.costs[(x, y)]
    return dist[L[-1]]  # the last vertex is the end of the project
```

Solution 4 **(Prim, Quite Simply – *5 points*)**

```
def Prim(G):
    """
    G is connected
    return a MST of G
    """

    T = graph.Graph(G.order, directed = False, costs = True)
    costs = [inf] * G.order
    p = [-1] * G.order
    M = [False] * G.order
    H = heap.Heap(G.order)
    costs[0] = 0
    x = 0
    for _ in range(G.order-1):  # graph is connected
        for y in G.adjlists[x]:
            if not M[y] and G.costs[(x, y)] < costs[y]:
                costs[y] = G.costs[(x, y)]
                p[y] = x
                H.update(y, costs[y])
        (_, x) = H.pop()
        M[x] = True
        T.addedge(x, p[x], costs[x])
    return T
```