graph algorithms sheet

graph traversal: BFS

```
queue q
dictionary prev, dist
set visited
q.enqueue(s)
visited.add(s)
dist[s]=0
while !q.isEmpty() do
    x=q.dequeue()
    for y in N^out(x) do
        if y not in visited then
            q.enqueue(y)
            visited.add(y)
            dist[y]=dist[y]+1
            prev[y]=x
        endif
    endfor
endwhile
accesible = visited
```

O(n+m)

graph traversal: DFS

```
stack stack
dictionary prev
set visited
stack.push(s)
visited.add(s)
prev[s]=null
while !stack.isEmpty do:
    x=stack.pop()
    for y in N^out(x) do
        if y not in visited then
            stack.push(y)
            visited.add(g)
            prev(y)=x
        endif
    endfor
endwhile
accesible = visited
```

O(n+m)

minimum cost walk (dynamic programming)

```
input:
    G-graph
    s-source vertex
    t-target vertex
output:
    minCost: min cost s->t
    path: path that achieves
          the minimum cost
```

- define a table minCost[1,n]: store the min cost from s to each vertex i
- define a table prev[1,n]: store the prev vertex in the optimal path to i
- initialize minCost[s]=0
- initialize prev[s]=0

```
for each vertex i in G\{s} do
    minCost[i]=infinity
end for
```

```
for vertex v in G (starting with s) do
    for neighbour v of u in G do:
        cost = minCost[u]+weight(u,v)
        if cost < minCost[v] then
            minCost[v]=cost
            prev[v]=u
        endif
    endfor
endfor
if minCost[t]=infinity then
    return "no path"
// construct the path
path=[]
current=t
while current is not null do
    append current to path
    current=prev[current]
endwhile
path.reverse()
return minCost[t], path
```

O(n+m)

minimum cost path (bellman-ford)

```
input:
    G: directed graph with costs
    s,t: start, target
output:
    dist: map for each acc.
    vertex to s (cost-wise)
    prev[]
for x in X do:
    dist[x]=infinity
endfor
dist[s]=0
changed=true
while changed do:
    changed=false
    for (x,y) in edges do:
        if dist[y]>dist[x]+
            cost(x,y) then
            dist[y]=dist[x]+cost(x,y)
            prev[y]=x
            changed=true
        endif
    endfor
end while
```

O(n*m)

minimum cost path (floyd-warshall)

based on dyn. programming
the recursion starts like for the matrix multiplication algorithm

```
for i=0,n-1 do
    for j=0,n-1 do
        if i==j then
            w[i,j]=0
        else if (i,j) is edge then
            w[i,j]=cost(i,j)
            p[i,j]=j
        else
            w[i,j]=infinity
        endif
    end for
endfor
```

O(n^3)

write a polynomial time algorithm that, given a directed graph and a pair of vertices s and t finds the number of distinct paths of minimum length from s to t

- we use a modified version of Dijkstra's algorithm the algorithm finds the shortest path from the source 's' to all the other vertices, while keeping track of the number of distinct paths to each vertex

```
def count_shortest_paths(graph,s,t):
    n=len(graph)
    dist=[float('inf')]*n
    paths=[0]*n
    dist[s]=0
    paths[s]=0
    pq=[(0,s)] // priority queue
    while pq:
        distance,u = heappop(pq)
        if distance > dist[u]
            continue
        // if we've already found a shorter path,
        skip the vertex
        for v in graph[u]
        // if the distance to v is shorter via u,
        update the values
            if dist[u]+1 < dist[v]
                dist[v]=dist[u]+1
                paths[v]=paths[u]
                heappush(pq,(dist[v],v))
            elif dist[u]+1 == dist[v]
                paths[v] += paths[u]
    return paths[t]
```

design an algorithm for solving the following problem: given a digraph acyclic (DAG), find a maximum length path in it in O(n+m)

- we use a dynamic prog. approach - we associate, to each vertex x, the length of the longest path ending in x. let's denote it by d[x]. to compute this value, we notice that the longest path to a vertex x is the longest path to a predecessor of x plus the edge (y,x) unless x has no predecessors.
- since d[x] depends on the predecessors of x, it is convenient to compute it in topological sorting order

```
input: G=(X,E), a DAG
output: p a path of max length
x_0,...,x_n-1 = toposort(G)
for x in X do
    d[x]=0
end for
for k=0,n-1 do
    for i=0,n-1 do
        for j=0,n-1 do
            if w[i,j]>w[i,k]+
            +w[k,j] then:
            w[i,j]=w[i,k]+
            +w[k,j]
            p[i,j]=p[i,k]
        endif
    end for
end for
```

```
for i=0,n-1 do
    x=x_i
    for y=N^out(x) do
        if d[y]>d[x]+1 then
            d[y]=d[x]+1
        end if
    end for
end for
maxLen=-1
for x in X do
    if d[x]>maxLen then
        maxLen=d[x]
        final=x
    endif
end for
x=final
p=[]
while d[x]>0 then
    p.append(x)
    for y in N^in(x) do
        if d[y]+1==d[x] then
            x=y
            break
        endif
    end for
end while
p.append(x)
p.reverse()
```

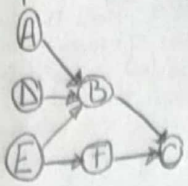
design an algorithm that, given a directed graph, finds a cycle of minimum length through a given vertex, if such a cycle exists

- we use a modified version of the floyd-warshall algorithm. the alg. considers all pairs of vertices and calculates the shortest distance between them while keeping track of the intermediate vertex that leads to the shortest path.

```
def find_min_cycle(graph,vertex):
    n=len(graph)
    dist=[[float('inf')]*n for _ in range(n)]
    path=[[-1]*n for _ in range(n)]
    for i in range(n):
        dist[i][i]=0
    #update the distance matrix using the graph edges
    for u in range(n):
        for v,weight in graph[u]:
            dist[u][v]=weight
            path[u][v]=u
    #floyd-warshall
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if dist[i][j]>dist[i][k]+dist[k][j]:
                    dist[i][j]=dist[i][k]+dist[k][j]
                    path[i][j]=path[k][j]
    #check for a cycle through a given vertex
    cycle=None
    min_cycle_length=float('inf')
    for u in range(n):
        if dist[u][vertex]<float('inf') and
        dist[vertex][u]<float('inf'):
            cycle_length=dist[u][vertex]+
                        +dist[vertex][u]
            if cycle_length<min_cycle_length:
                min_cycle_length=cycle_length
                cycle=reconstruct_cycle(u,vertex,
                path)
    return cycle
def reconstruct_cycle(s,t,path)
    cycle=[t]
    while s!=t
        t=path[s][t]
        cycle.append(cycle)
    return cycle
cycle=find_min_cycle(graph,vertex)
if cycle
    print(cycle)
else
    print("no cycle")
```

1 considering the following activities, determine the earliest and latest schedulings, and the critical activities (show the step by step computations for the topological sorting, then for the starting and ending times)

activity	duration	prerequisites
A	1	–
B	4	A, D, E
C	7	B, F
D	3	–
E	2	–
F	5	E




```
def topological_sort_dfs(self, vertex: int, sorted: list, fully_processed: set, in_process: set):
    in_process.add(vertex)
    for successor in self.successors[vertex]:
        if successor in in_process:
            return False
        elif successor not in fully_processed
            ok = self.topological_sort_dfs(successor, sorted, fully_processed, in_process)
            if not ok
                return False
    in_process.remove(vertex)
    sorted.append(vertex)
    fully_processed.add(vertex)
    return True
```

```
def toposort(self):
    sorted = list()
    fully_processed = set()
    in_process = set()
    for x in self.vertices
        if x not in fully_processed:
            ok = self.topological_sort_dfs
            (x, sorted, fully_processed, in_process)
            if not ok:
                return False
    sorted.reverse()
    return sorted
```

calls	x, y	in_process	fully_processed	sorted	ok
initialization		{}	{}	[]	
(self, E, [], {}, {})	x=E	{}	{}	[]	true
		{E}			
		{}	{E}	[E]	
(self, F, [E], {E}, {})	x=F	{}	{E}	[E]	true
	y=E	{F}			
		{}	{E, F}	[E, F]	
(self, C, [E, F], {E, F}, {})	x=C	{}	{E, F}	[E, F]	
	y=F	{C}			
	y=B				
(self, B, [E, F], {E, F}, {C})	X=B	{C}	{E, F}	[E, F]	
	y=E	{C, B}			
	y=D				
	y=A				
(self, D, [E, F], {E, F}, {C, B})	x=D	{C, B}	{E, F}	[E, F]	true
		{C, B, D}			
		{C, B}	{E, F, D}	[E, F, D]	
(self, A, [E, F, D], {E, F, D}, {C, B})	X=A	{C, B}	{E, F, D}	[E, F, D]	true
		{C, B, A}			
		{C, B}	{E, F, D, A}	[E, F, D, A]	
	X=B	{C}	{E, F, D, A, B}	[E, F, D, A, B]	true
	X=C	{}	{E, F, D, A, B, C}	[E, F, D, A, B, C]	true

topological order = sorted =
= [E, F, D, A, B, C]

```
def get_times(self):
    sorted_vertices = self.toposort()
    self.earliest_start = [float('-inf')]*self.n()
    self.earliest_end = [float('+inf')]*self.n()
    self.latest_start = [float('inf')]*self.n()
    self.latest_end = [float('inf')]*self.n()
    for vertex in sorted_vertices:
        if self.number_of_predecessors(vertex)==0
            self.earliest_start[vertex] = 0
        else:
            for predecessor in self.predecessors(vertex):
                self.earliest_start[vertex] = max
                (self.earliest_start[vertex], self.earliest
                end[predecessor])
        self.earliest_end[vertex] = self.earliest_start[vertex]
        + self.duration[vertex]
    sorted_vertices.reverse()
    for vertex in sorted_vertices:
        if self.number_of_successors(vertex)==0:
            self.latest_end[vertex] = self.earliest_end[sorted_vertices[0]]
        for successor in self.successors(vertex)
            self.latest_end[vertex] = min(self.latest_end[vertex], self.latest_start[successor])
        self.latest_start[vertex] = self.latest_end[vertex] - self.duration[vertex]
    if not sorted_vertices:
        return -1, -1, -1, -1
    return self.earliest_start, self.earliest_end,
    self.latest_start, self.latest_end
    [sorted_vertices[0]]
```

A: 0-1, 2-3
B: 3-7, 3-7
C: 7-8, 7-8
D: 0-3, 0-3
E: 0-2, 0-2
F: 2-7, 2-7
duration: 8
critical activities: B, C, D, E, F

1 in the following graph, find the maximum flow from vertex 1 to 6, start from the given flow and maximize it using Ford-Fulkerson alg. also show the min. cut