A Algorismes

A.1 DFS

```
parent={}
topo=[]
def DFS(Adj):
    node=[]
    for i in range(0, len(Adj)):
        node.append(i)

    for s in node:
        if s not in parent:
            print "From node %d:" %s
            print s
            parent[s]=None
            DFS_recursive(Adj, s)
    print "Recursion order (topological sort for directed acyclic graphs):"
    topo.reverse()
    print topo


def DFS_recursive(Adj, s):
    for v in Adj[s]:
        if v not in parent:
            print v
            parent[v]=s
            DFS_recursive(Adj, v)
    topo.append(s)
```

A.2 BFS

```
def BFS(Adj, s):
    level={s:0}
    parent={s:None}
    i=1
    frontier=[s]
    print s
    while frontier:
```

```
        next=[]
        for u in frontier:
            for v in Adj[u]:
                if v not in  level:
                    level[v]=i
                    parent[v]=u
                    next.append(v)
                    print v
        frontier=next
        i+=1
    print level
```

A.3 Dijkstra

```
def Dijkstra(Adj, s):
    Q={}
    dist={}
    tree={}
    for i in range(0, len(Adj)):
        Q[i]=float("inf")
        dist[i]=float("inf")
    Q[s]=0
    while Q:
        u = min(Q, key=Q.get)
        dist[u] = Q[u]
        for v in Adj[u]:
            if v in Q:
                if Q[v] > Q[u] + Adj[u][v]:
                    Q[v] = Q[u] + Adj[u][v]
                    tree[v] = u
        Q.pop(u)

    return dist, tree


def OrderedDijkstra(Adj, s):
    Q = dict.fromkeys(Adj.keys(), float("inf"))
    dist = dict.fromkeys(Adj.keys(), float("inf"))
    tree = {}
    Q[s] = 0
```

```
    while Q:
        u = min(Q, key=Q.get)
        dist[u] = Q[u]
        for v in Adj[u]:
            if v in Q:
                if Q[v] > Q[u] + Adj[u][v]:
                    Q[v] = Q[u] + Adj[u][v]
                    tree[v] = u
        Q.pop(u)

    return dist, tree
```

A.4 Bellman-Ford

```
def BellmanFord(Adj, s):
    dist={}
    tree={}
    for i in range(0, len(Adj)):
        dist[i]=float("inf")
        tree[i]=None
    dist[s]=0

    for i in range(0, len(Adj)-1):
        for u in range(0, len(Adj)):
            for v in Adj[u]:
                if dist[v] > dist[u] + Adj[u][v]:
                    dist[v] = dist[u] + Adj[u][v]
                    tree[v]=u
    for u in range(0, len(Adj)):
        for v in Adj[u]:
            if dist[v] > dist[u] + Adj[u][v]:
                print "There are negative-weight cycles"
                break
    return dist, tree
```

A.5 Prim

```
def Prim(Adj):
    Q={}
```

```
    tree={}
    for i in range(0,len(Adj)):
        Q[i]=float("inf")
    Q[0]=0
    while Q:
        u = min(Q, key=Q.get)
        for v in Adj[u]:
            if v in Q and Adj[u][v] < Q[v]:
                Q[v] = Adj[u][v]
                tree[v] = u
        Q.pop(u)
    return tree
```

A.6 Kruskal

```
def Kruskal(Adj):
    subtree = UnionFind()
    tree = []
    for e, u, v in sorted((Adj[u][v],u,v) for u in Adj for v in Adj[u]):
        for u in Adj:
            for v in Adj[u]:
                if subtree[u] != subtree[v]:
                    tree.append((u,v))
                    subtree.union(u,v)
    return tree
```

A.7 Floyd-Warshall

```
def FloydWarshall(Adj):
    dist=[[float("inf") for x in range(len(Adj))] for y in range(len(Adj))]
    for i in range(0,len(Adj)):
       dist[i][i] = 0
    for v in range(len(Adj)):
        for u in Adj[v]:
            dist[v][u] = Adj[v][u]
    for x in range(len(Adj)):
        for u in range(len(Adj)):
            for v in range(len(Adj)):
                if dist[u][v] > dist[u][x] + dist[x][v]:
```

```
                    dist[u][v] = dist[u][x] + dist[x][v]
    return dist
```

A.8 Hamilton

```
def Hamilton_recursive(Adj, s, e, path):
    path = path + [s]
    if s == e:
        return path
    for n in Adj[s]:
        if n not in path:
            nou_path = Hamilton_recursive(Adj, n, e, path)
            if nou_path:
                return nou_path
    return None

def Hamilton(Adj, s, e):
    path=[]
    return Hamilton_recursive(Adj, s, e, path)
```

A.9 Euler

```
def Euler(Adj):
    graf = Adj
    senar = [v for v in graf.keys() if len(graf[v])%2 != 0]
    senar.append(graf.keys()[0])
    print senar

    if len(senar)>3:
        return None

    Q = [senar[0]]
    path = []
    while Q:
        v = Q[-1]
        if graf[v]:
            u = graf[v][0]
            Q.append(u)
            del graf[u][graf[u].index(v)]
```

```
            del graf[v][0]
        else:
            path.append(Q.pop())

    return path
```

A.10 Coloració

```
def coloring(Adj):
    graph = sorted(Adj, key=lambda k:len(Adj[k]), reverse=True)
    colors = {}
    usat = False
    actual = 0

    for i in range(0, len(Adj)):
        colors[i]=None
    colors[graph[0]]=0

    while None in colors.values():
        for v in graph:
            if colors[v] == None:
                for k in Adj[v]:
                    if colors[k] == actual:
                        usat = True
                        break

                if usat == False:
                    colors[v] = actual
                usat = False
        actual = actual + 1
    return colors
```

A.11 Metro

```
def metro(Adj, inici, final):
    recorregut=[]

    print "Punt inicial:", inici.decode("ISO-8859-15")
```

```
    print "Punt final:", final.decode("ISO-8859-15")

    dist, tree = OrderedDijkstra(Adj, inici)
    print type(inici)
    print type(final)

    i = final
    while tree[i] != inici:
        recorregut.append(tree[i])
        i = tree[i]

    recorregut.append(inici)
    recorregut.reverse()

    total= dist[final]+(25*(len(recorregut)-2))

    minuts = total/60
    segons = (total%60)*0.60
    print "Temps net del recorregut:", dist[final]
    print "Temps total del recorregut:", int(minuts),"minuts i", int(segons), "sego
    
    print "Recorregut:",
    print "[",
    for i in range(0,len(recorregut)):
        print recorregut[i].decode("ISO-8859-15")+",",

    print final.decode("ISO-8859-15"),"]"
```