BFS

```
BFS

Input:
    G : graph
    s : a vertex
Output:
    accessible : the set of vertices that are accessible from s
    prev : a map that maps each accessible vertex to its predecessor on a path from
s to it
Algorithm:
    Queue q
    Dictionary prev
    Dictionary dist
    Set visited
    q.enqueue(s)
    visited.add(s)
    dist[s] = 0
    while not q.isEmpty() do
        x = q.dequeue()
        for y in Nout(x) do
            if y not in visited then
                q.enqueue(y)
                visited.add(y)
                dist[y] = dist[x] + 1
                prev[y] = x
            end if
        end for
    end while
    accessible = visited
```

DFS

```
DFS - connected components

Input:
    G : directed graph
Output:
    comp : a map that associates, to each vertex, the ID of its strongly connected
component

Subalgorithm DF1(Graph G, vertex x, Set& visited, Stack& processed)
    for y in Nout(x) do
        if y not in visited then
            visited.add(y)
            DF1(y)
        end if
    end for
    processed.push(x)

Algorithm:
    Stack processed
    Set visited

    for s in X do
        if s not in visited then
            visited.add(s)
            DF1(G, s, visited, processed)

    visited.clear()
    Queue q
    int c = 0
    while not processed.isEmpty() do
        s = processed.pop()
        if s not in visited then
            c = c + 1
            comp[s] = c
            q.enqueue(s)
            visited.add(s)
            while not q.isEmpty() do
                x = q.dequeue()
                for y in Nin(x) do
                    if y not in visited then
                        visited.add(y)
                        q.enqueue(y)
                        comp[y] = c
                    end if
                 end for
             end while
         end if
     end while
```

Dijkstra

Dijkstra

```
def dijkstra(g, s):
        prev = {}
        q = PriorityQueue()
        q.add(s, 0)
        d = {}
        d[s] = 0
        visited = set()
        visited.add(s)
        printDijkstraStep(s, None, q, d, prev)
        while not q.isEmpty():
                x = q.pop()
                for y in g.parseNout(x):
                        if y not in visited or d[y] > d[x] + g.cost(x, y):
                                d[y] = d[x] + g.cost(x, y)
                                visited.add(y)
                                q.add(y, d[y])
                                prev[y] = x
                printDijkstraStep(s, x, q, d, prev)


        return (d, prev)
```

Bellman Ford

```
Bellman Ford

def bellman(g, s):
        w = [{s : 0}]
        path = [{s : (0,)}]
        print "k=%s" % 0
        print "w=%s" % w[0]
        print "path = %s" % path[0]
        for k in range(1, len(g.parseX())):
                w.append({})
                path.append({})
                for y in w[k-1]:
                        for x in g.parseNout(y):
                                if ((x not in w[k]) or
                                        (w[k][x]>w[k-1][y]+g.cost(x, y) )):
                                        w[k][x] = w[k-1][y]+g.cost(y, x)
                                        path[k][x] = path[k-1][y]+ (x,)
                print "k=%s" % k
                print "w=%s" % w[k]
                print "path = %s" % path[k]
```

Prim

```
Prim



Input:
    G : directed graph with costs
Output:
    edges : a collection of edges, forming a minimum cost spanning tree
Algorithm:
    PriorityQueue q
    Dictionary prev
    Dictionary dist
    edges = ∅
    choose s in X arbitrarily
    vertices = {s}
    for x in N(x) do
        dist[x] = cost(x, s)
        prev[x] = s
        q.enqueue(x, d[x])              // second argument is priority
    while not q.isEmpty() do
        x = q.dequeue()      // dequeues the element with minimum value of priority
        if x ∉ vertices then
            edges.add({x, prev[x]})
            for y in N(x) do
                if y not in dist.keys() or cost(x,y) < dist[y] then
                    dist[y] = cost(x, y)
                    q.enqueue(y, dist[y])
                    prev[y] = x
                end if
            end for
        end if
    end while
```

Kruskal

Kruskal

```
def kruskal(graph):
    for vertice in graph['vertices']:
        make_set(vertice)

    minimum_spanning_tree = set()
    edges = list(graph['edges'])
    edges.sort()
    for edge in edges:
        weight, vertice1, vertice2 = edge
        if find(vertice1) != find(vertice2):
            union(vertice1, vertice2)
            minimum_spanning_tree.add(edge)
    return minimum_spanning_tree
```

A*

```
A*


Input:
    G : directed graph with costs
    s, t : two vertices
    h : X -> R the estimation of the distance to t
Output:
    dist : a map that associates, to each accessible vertex, the cost of the
minimum
            cost walk from s to it
    prev_: a map that maps each accessible vertex to its predecessor on a path from
s to it
Algorithm:
    PriorityQueue q
    Dictionary prev
    Dictionary dist
    q.enqueue(s, h(s))
    dist[s] = 0
    found = false
    while not q.isEmpty() and not found do
        x = q.dequeue()
        for y in Nout(x) do
            if y not in dist.keys() or dist[x] + cost(x,y) < dist[y] then
                dist[y] = dist[x] + cost(x, y)
                q.enqueue(y, dist[y]+h(y))
                prev[y] = x
            end if
        end for
        if x == t then
            found = true
        endif
    end while
```

Floyd

```
Floyd

let dist be an V x V array of minimum distances initialized to infinity
let next be an V x V array of vertex indices initialized to null

procedure FloydWarshallWithPathReconstruction ()
   for each edge (u,v)
      dist[u][v] ← w(u,v)  // the weight of the edge (u,v)
      next[u][v] ← v
   for k from 1 to |V| // standard Floyd-Warshall implementation
      for i from 1 to |V|
         for j from 1 to |V|
            if dist[i][j] > dist[i][k] + dist[k][j] then
               dist[i][j] ← dist[i][k] + dist[k][j]
               next[i][j] ← next[i][k]

procedure Path(u, v)
   if next[u][v] = null then
       return []
   path = [u]
   while u ≠ v
       u ← next[u][v]
       path.append(u)
   return path
```

DAG

```
DAG

Input:
      G : directed graph
Output:
    sorted : a list of vertices in topological sorting order, or null if G has
cycles
Algorithm:
    sorted = emptyList
    Queue q
    Dictionary count
    for x in X do
        count[x] = indeg(x)
        if count[x] == 0 then
            q.enqueue(x)
        endif
    endfor
    while not q.isEmpty() do
        x = q.dequeue()
        sorted.append(x)
        for y in Nout(x) do
            count[y] = count[y] - 1
            if count[y] == 0 then
                q.enqueue(y)
            endif
        endfor
    endwhile
    if sorted.size() < X.size() then
        sorted = null
    endif
```

DAG DFS

```
Input:
    G : directed graph
Output:
    sorted : a list of vertices in topological sorting order, or null if G has
cycles
Subalgotithm TopoSortDFS(Graph G, Vertex x, List sorted, Set fullyProcessed, Set
inProcess)
    inProcess.add(x)
    for y in Nin(x)
        if y in inProcess then
            return false
        else if y not in fullyProcessed then
            ok = TopoSortDFS(G, y, sorted, fullyProcessed, inProcess)
            if not ok then
                return false
            endif
        endif
    endfor
    inProcess.remove(x)
    sorted.append(x)
    fullyProcessed.add(x)
    return true

Algorithm:
    sorted = emptyList
    fullyProcess = emptySet
    inProcess = emptySet
    for x in X do
        if x not in fullyProcessed then
            ok = TopoSortDFS(G, x, sorted, fullyProcessed, inProcess)
            if not ok then
                sorted = null
                return
            endif
        endif
```

Hamiltonian Cycle

Hamiltonian Cycle

```
def hamiltonianCycle(g):
        '''Returns a Hamiltonian cycle in g, if one exists, as a list of vertices,
        with the first and the last vertex on the list being equal (the length of
        the returned list will be n+1).
        Returns None if no Hamiltonean cycle exists in g.'''

        start_vertex = None


        for x in g.parseVertices():
                start_vertex = x
                break

        sol = []

        dfs(g, [ start_vertex ], sol)

        if( sol == [] ):
                return None

        return sol

def dfs(g, current_path, solution):
        '''
        the dfs function will put the hamiltonean cycle in solution if one exists
        '''
        current_node = current_path[-1]

        if( len( current_path ) == g.nrOfVertices() ):
                if( current_path[ 0 ] in g.parseEdgeOut( current_node ) and solution
== [] ):
                          for it in current_path:
                                solution.append( it )
                          solution.append( current_path[ 0 ] )
                return
```