6. All Graph Algorithms

DATA STRUCTRES AND ALGORITHMS
[17ECSC204]

PRAKASH HEGADE

SCHOOL OF CSE | KLE Tech

1. Depth First Search

```
ALGORITHM DFS (G)
// Implements a depth-first search traversal of a given graph
// Input: Graph G = <V, E>
// Output: Graph G with its vertices marked with consecutive integers in the order
// they have been first encountered by the DFS traversal
Mark each vertex in V with 0 as a mark of being 'unvisited'
count ← 0
for each vertex v in V do
   if v is marked with 0
      dfs(v)

dfs(v)
// Visits recursively all the unvisited vertices connected to vertex v by a path
// and numbers them in the order they are connected
// via global variable count
count ← count + 1; mark v with count
for each vertex w in V adjacent to v do
   if w is marked with 0
      dfs(w)
```

2. Breadth First Search

```
ALGORITHM BFS (G)
// Implements a breadth-first search traversal of a given graph
// Input: Graph G = <V, E>
// Output: Graph G with its vertices marked with consecutive integers in the order
// they have been first encountered by the BFS traversal
Mark each vertex in V with 0 as a mark of being 'unvisited'
count ← 0
for each vertex v in V do
   if v is marked with 0
      bfs(v)

bfs(v)
// Visits all the unvisited vertices connected to vertex v by a path
// and assigns them the numbers in the order they are visited
// via global variable count
count ← count + 1; mark v with count and initialize a queue with v
while the queue is not empty do
   for each vertex w in V adjacent to the front vertex do
      if w is marked with 0
         count ← count + 1; mark w with count
         add w to the queue
   remove the front vertex from the queue
```

3. Dijkstra's Algorithm

ALGORITHM **Dijkstra** (n, cost[ ][ ], src, dest, dist[ ], path[ ])
for i from 0 to n-1 do
   s[i] ← 0
   dist[i] ← cost[src, i]
   path[i] ← src

s[src] = 1

for i from 1 to n-1 **do**
   find u and dist[u] such that dist[u] is minimum and u in V-S
   min ← 99999
   u ← -1
   for j from 0 to n-1 **do**
      if s[j] = 0 and dist[j]<=min
         min← d[j]
         u← j
   if u = -1 **return**
   s[u] ← 1
   if u = destination **return**
   for every v in V-S **do**
      if dist[u] + cost[u, v] < dist[v]
         dist[v] = dist[u] + cost[u, v]
         path[v] = u
return

4. Prim's Algorithm

ALGORITHM Prim(G)
// Prim's algorithms for constructing a minimum spanning tree
// Input: A weighted connected graph G = (V, E)
// Output: E_T, the set of edges composing a minimum spanning tree of G
V_T ← { v_0}
E_T ← 0
for i ← 1 to |V| - 1 do
   find a minimum weight edge e* = (v*, u*) among all the edges (v, u)
   such that v is in V_T and u is in V- V_T
   V_T ← V_T U {u*}
   E_T ← E_T U {e*}
return E_T

5. Kruskal's Algorithm

ALGORITHM Kruskal(G)

// Kruskal's algorithms for constructing a minimum spanning tree
// Input: A weighted connected graph G = (V, E)
// Output: E_T, the set of edges composing a minimum spanning tree of G
Sort E in increasing order of the edge weights $w(e_{i1}) <= w(e_{i2}) <= \ldots\ w(e_{i|E|})$
$E_T \leftarrow 0$; ecounter $\leftarrow 0$
k $\leftarrow$ 0
while ecounter < |V| - 1 do
  k $\leftarrow$ k + 1
  if $E_T \cup \{ e_{i\,k} \}$ is acyclic
    $E_T \leftarrow E_T \cup \{ e_{i\,k} \}$; ecounter $\leftarrow$ ecounter + 1
return E_T

6. Warshall's Algorithm

ALGORITHM Warshall (A[1… n, 1… n])

// Implements Warshal's algorithm for computing the transitive closure
// Input: The adjacency matrix A of a digraph with n vertices
// Output: The transitive closure of the digraph
$R^{(0)} \leftarrow A$
for k $\leftarrow$ 1 to n do
  for i $\leftarrow$ 1 to n do
    for j $\leftarrow$ 1 to n do
      $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ or $(R^{(k-1)}[i, k]$ and $R^{(k-1)}[k, j])$
return $R^{(n)}$

7. Floyd's Algorithm

ALGORITHM Floyd (W[1… n, 1… n])

// Implements Floyd's algorithm for the all-pair shortest-paths problem
// Input: The weight matrix with no negative-length cycles
// Output: The distance matrix of the shortest path's lengths
D $\leftarrow$ W
for k $\leftarrow$ 1 to n do
  for i $\leftarrow$ 1 to n do
    for j $\leftarrow$ 1 to n do
      D[i, j] $\leftarrow$ min{ D[i, j], D[i, k] + D[k, j] }
return D

8. Bellman-Ford Algorithm

```
function BellmanFord(list vertices, list edges, vertex source) :: distance[], predecessor[]

  // This implementation takes in a graph, represented as lists of vertices and edges, and
  // fills two arrays (distance and predecessor) about the shortest path
  // from the source to each vertex

  // Step 1: initialize graph
  for each vertex v in vertices:
     // At the beginning, all vertices have a weight of infinity
     distance[v] := inf
     // And a null predecessor
     predecessor[v] := null

  // The weight is zero at the source
  distance[source] := 0

// Step 2: relax edges repeatedly
  for i from 1 to size(vertices)-1:
     for each edge (u, v) with weight w in edges:
        if distance[u] + w < distance[v]:
           distance[v] := distance[u] + w
           predecessor[v] := u

  // Step 3: check for negative-weight cycles
  for each edge (u, v) with weight w in edges:
     if distance[u] + w < distance[v]:
        error "Graph contains a negative-weight cycle"

  return distance[], predecessor[]
```

Note: The algorithms are referenced from course text books and Wikipedia.

~*~*~*~*~*~*~