Graph Algorithms
Recursive DFS

DFS(v:Integer)
    visit and mark v;
    while there is an unmarked vertex w adjacent to v do
        DFS(w)
    endwhile
end DFS

(Global AdjacencyList is required)
Compute Start Time

DFS(v:Integer)
   Start_Time := Start_Time + 1;
   d[v] := Start_time;
   visit and mark v;
   while there is an unmarked vertex w adjacent to v do
      DFS(w);
   endwhile
end DFS

{Start_Time and AdjacencyList are global}
Compute Finish Time

DFS(v)
    visit and mark v;
    while there is an unmarked vertex w adjacent to v do
        DFS(w)
    endwhile
    Finish_Time := Finish_Time + 1;
    f[v] := Finish_Time;
end DFS

{ AdjacencyList and Finish_Time are global}
Strongly Connected Components: Compute Finish Times

for i:= 1 to n do
    mark[i] := 0;
endfor

FinishTime := 0

for i:= 1 to n do
    if mark[i] = 0 then
        DFS(i);
    endif
endfor

DFS(v)
    mark[v] := 1;
    ptr := AdjacencyList[v];
    while ptr ≠ NIL do
        w:= ptr^.vertex;
        if mark[w] = 0 then
            DFS(w)
        endif
        ptr := ptr^.Link;
    endwhile

FinishTime := FinishTime+1
f[v] := FinishTime
endDFS
Strongly Connected Components: Reverse the Edges

For $i := 1$ to $n$ do
    RevAdj[$i$] := NIL;
endfor
for $i := 1$ to $n$ do
    ptr := AdjList[$i$];
    while ptr $\neq$ NIL do
        w := ptr$^\cdot$vertex;
        ptr2 := new(AdjElem);
        ptr2$^\cdot$vertex := i;
        ptr2$^\cdot$link := RevAdj[w];
        RevAdj[w] := ptr2;
    endwhile
endfor
Strongly Connected Components: Sort into Decreasing Finish Order

for i := 1 to n do
  lookup[n-FinishTime[i]+1] := i;
endfor
Strongly Connected Components: DFS by Decreasing Finish

for i := 1 to n do
  mark[i] := 0;
endfor

SCCNumber := 0;
for i := 1 to n do
  if mark[lookup[i]] = 0 then
    SCCNumber := SCCNumber + 1
    DFS(lookup[i]);
  endif
endfor

DFS(v)
mark[v] := SCCNumber;
ptr := RevAdj[v];
while ptr ≠ NIL do
  w := ptr^.Vertex;
  if mark[w] = 0 then
    DFS(w);
  endif
  ptr := ptr^.Link;
endwhile
endDFS
Strongly Connected Components

DFS(v)
  StartTime := StartTime + 1;
  d[v] := StartTime;
  low[v] := d[v];  rm[v] := False;
  for each w adjacent to v do
    if d[w] = 0 then
      DFS(w);
      low[v] := min(low[v], low[w]);
    else
      if not rm[w] then
        low[v] := min(d[w], low[v]);
      endif
    endif
  endfor;
  if low[v] = d[v] then
    rm[v] := True; Output v;
    while SC nonempty and
      d[Top(SC)] > d[v] do
      output Top(SC);
      rm[Top(SC)] := True;
      Pop SC;
    endwhile;
  else
    Push v onto SC;
  endif
end DFS

for v := 1 to n do d[v] := 0;
StartTime := 0;
for v := 1 to n do
  if d[v] = 0 then DFS(v);
endfor
Biconnected Components

DFS(v)
    StartTime := StartTime + 1;
    d[v] := StartTime;  \{Init to 0\}
    back[v] := d[v];
    for every w adjacent to x do
        if d[w] < d[v] then
            push (v,w) on EdgeStack
        endif
        if d[w] = 0 then
            DFS(w);
            if back[w] \geq d[v] then
                Pop Edgestack until
                (v,w) is popped
            else
                back[v] := min(back[v], d(w));
            endif
        endif
    endfor
end DFS

begin
    for i:= 1 to n do d[i] := 0;
    StartTime := 0;
    DFS(1);
end
Bipartite Matching
Alternating Path
Reverse
Maximum Flow
An Augmenting Path
Compute Residual Network

Add all vertices of \( V(G) \) to \( V(H) \)
For \((u,v)\) in \( E(G) \) do
  if \((u,v)\) is below capacity then
    Add \((u,v)\) to \( E(H) \)
  endif
  if \((u,v)\) has non-zero flow then
    Add \((v,u)\) to \( E(H) \)
  endif
endfor
found := FALSE;
DFS(S,H)
If not found the exit
else AUGMENT
Go Back to first step;
DFS For Augmenting Path

DFS(x,H)
mark x;
if x = L then
    found := TRUE;
    exit;
endif
for y adjacent to x in H do
    if y is unseen then
        Parent[y] = x;
        DFS(y);
    endif;
    if found then
        exit
    endif
endfor
end DFS
vert := L;
Aug := \infty;
while vert <> S do
    if (vert, parent[vert]) is in G then
        if residual_capacity((vert, parent[vert])) < Aug then
            Aug := residual_capacity((vert, parent[vert]));
        endif
    else
        if current_flow((vert, parent[vert]) < Aug then
            Aug := current_flow((vert, parent[vert]));
        endif
    endif
endwhile

/* Augmenting value has now been computed */
vert := L;
while vert <> S do
    if (vert, parent[vert]) is in G Then
        increase flow of (vert, parent[vert]) by Aug;
    else
        decrease flow of (vert, parent[vert]) by Aug;
    endif
    vert = parent[vert];
endwhile

/* Augmentation is now complete */
Max Flow for Bipartite Match
Breadth First Search

BFS(AdjacencyList:ListType, v:Integer)

Q: QUEUE;
Initialize Q to Empty;
Visit and mark v;
Insert v into Q; \{Add to tail of Q\}
while Q is not Empty do
  x := Remove(Q); \{Remove head of Q\}
  for each unmarked vertex w adjacent to x do
    visit and mark w;
    insert w into Q;
  endfor
endwhile
end BFS
Depth First Search

DFS(adjacencyList:ListType,v:Integer)

S: Stack
initialize S to Empty;
visit and mark v;
push v into S;
while S is non-empty do
    for each unmarked vertex w adjacent to TOP(S) do
        visit and mark w;
push w into S;
endfor;
POP(S);
endwhile
end DFS
Connected Components

{Mark vertex \( v \) by assigning number \( > 0 \) to \( \text{mark}[v] \).
{Use component number as mark value.}

\[\begin{align*}
\text{for } i & := 1 \text{ to } n \text{ do } \quad \{ \text{n = number of vertices} \} \\
\text{mark}[i] & := 0; \\
\text{endfor} \\
\text{ComponentNumber} & := 0; \quad \{ \text{Global Variable} \} \\
\text{for } i & := 1 \text{ to } n \text{ do} \\
\text{if } \text{mark}[i] = 0 \text{ then} \\
\text{ComponentNumber} & := \text{ComponentNumber} + 1; \\
\text{DFS}(i) \\
\text{endif} \\
\text{endfor}
\end{align*}\]
Connected Component DFS

DFS(v) {Local declarations omitted.}
    mark[v] := ComponentNumber;
    ptr := AdjacencyList[v];
    while ptr ≠ NIL do
        w := ptr^.vertex;
        if mark[w] = 0 then
            DFS(w)
        endif
        ptr := ptr^.Link;
    endwhile
end DFS
Kruskal’s MST Algorithm

Create a forest of trees numbered 1 to |V|
Sort Edges by increasing weight
For each edge (x,y) do  {in ascending order by weight}
    if vertices x and y are in different trees
        Add (x,y) to the spanning tree
        Renumber all vertices in x’s tree using y’s number
    endif
endfor
Prim/Dijkstra MST

Initialization

Pick any starting vertex $x$
Place $x$ in the MST
For all vertices $y$ adjacent to $x$ do
  Add $y$ to the Fringe_Set;
endfor
For each element $y$ of the Fringe_Set do
  Set weight of $y$ equal to the weight of edge $(x,y)$
  Set Parent of $y$ to $x$
endfor
Prim/Dijkstra Body

While number of vertices in MST is less than |V| do
  Find element y of Fringe_Set with minimum weight
  Add vertex y and edge (y, Parent of y) to MST
  Remove y from Fringe_Set
  For all vertices z adjacent to y do
    if z not in Fringe_Set then
      Put z into Fringe_Set;
      Set weight of z to weight of (y, z)
      Set parent of z to y
    else
      if weight of (y, z) less than weight of z then
        set weight of z to weight of (y, z)
        set parent of z to y
      endif
    endif
  endfor; endwhile
Dijkstra’s Shortest Path

Place the starting vertex A into the SP tree;
For all vertices y adjacent to A
    Add y to the FRINGE_SET;
For each element y in the FRINGE_SET
    Set the weight of y to the weight of the edge xy;
    Set parent[y] to x;
While ending vertex B is not in the SP tree and
    FRINGE_SET is not empty do
    \textbf{Execute Loop Body};
End While
If B is not in the SP Tree
    Print Error Message
Else
    Print contents of SP tree
End If
Dijkstra’s S.P. Body

Find the element $y$ of FRINGE_SET with minimum weight;
Add the vertex $y$, and edge $[y, parent[y]]$ to S.P. tree;
Remove $y$ from FRINGE_SET;
For all vertices $z$ adjacent to $y$ do
  If $z$ is not in FRINGE_SET then
    Add $z$ to FRINGE_SET;
    Set weight($z$) to weight($[y,z]$) + weight($y$);
    Set Parent[$z$] to $y$;
  Else
    if weight($y$) + weight($[y,z]$) < weight($z$) then
      Set weight($z$) to weight($y$) + weight($[y,z]$);
      Set Parent[$z$] to $y$;
    End If
  End If
End For
Dijkstra’s “Other” Algorithm

Initialize-Single-Source(G,s)
S ← φ
Q ← V[G]
While Q ≠ φ do
    u ← EXTRACT-MIN(Q)
    S ← S union {u}
    for each vertex v in Adj[u] do
        RELAX(u,v,w);
    end for
end while
The RELAX Algorithm

RELAX(u,v,w)
    if d[v] > d[u] + w(u,v) then
        d[v] ← d[u] + w(u,v)
        p[v] ← u
    End If
End RELAX
The Bellman-Ford Algorithm

INITIALIZE-SINGLE-SOURCE(G,s)
for i<-1 to |V[G]|-1 do
    for each edge (u,v) of E[G] do
        RELAX(u,v,w)
    end for
end for

for each edge (u,v) in E[G] do
    if d[v] > d[u] + w(u,v) then
        return FALSE;
    end if
end for
return TRUE
Graph Algorithms
Recursive DFS

DFS(v:Integer)
    visit and mark v;
    while there is an unmarked vertex w adjacent to v do
        DFS(w)
    endwhile
end DFS

(Global AdjacencyList is required)
Compute Start Time

DFS(v:Integer)
  Start_Time := Start_Time + 1;
  d[v] := Start_time;
  visit and mark v;
  while there is an unmarked vertex w adjacent to v do
    DFS(w);
  endwhile
end DFS

{Start_Time and AdjacencyList are global}
Compute Finish Time

DFS(v)
    visit and mark v;
    while there is an unmarked vertex w adjacent to v do
        DFS(w)
    endwhile
    Finish_Time := Finish_Time + 1;
    f[v] := Finish_Time;
end DFS

{ AdjacencyList and Finish_Time are global}
Strongly Connected Components: Compute Finish Times

for i:= 1 to n do
    mark[i] := 0;
endfor
FinishTime := 0
for i:= 1 to n do
    if mark[i] = 0 then
        DFS(i);
    endif
endfor

DFS(v)
    mark[v] := 1;
    ptr := AdjacencyList[v];
    while ptr ≠ NIL do
        w:= ptr^.vertex;
        if mark[w] = 0 then
            DFS(w)
        endif
        ptr := ptr^.Link;
    endwhile
    FinishTime := FinishTime+1
    f[v] := FinishTime
endDFS
Strongly Connected Components: Reverse the Edges

For i := 1 to n do
    RevAdj[i] := NIL;
endfor
for i := 1 to n do
    ptr := AdjList[i];
    while ptr ≠ NIL do
        w := ptr^.vertex;
        ptr2 := new(AdjElem);
        ptr2^.vertex := i;
        ptr2^.link := RevAdj[w];
        RevAdj[w] := ptr2;
    endwhile
endfor
Strongly Connected Components: Sort into Decreasing Finish Order

for i := 1 to n do
    lookup[n-FinishTime[i]+1] := i;
endfor
Strongly Connected Components: DFS by Decreasing Finish

for i := 1 to n do
    mark[i] := 0;
endfor

SCCNumber := 0;
for i := 1 to n do
    if mark[lookup[i]] = 0 then
        SCCNumber := SCCNumber + 1
        DFS(lookup[i]);
    endif
endfor

DFS(v)
    mark[v] := SCCNumber;
    ptr := RevAdj[v];
    while ptr ≠ NIL do
        w := ptr^.Vertex;
        if mark[w] = 0 then
            DFS(w);
        endif
        ptr := ptr^.Link;
    endwhile
endDFS
Strongly Connected Components

DFS(v)
   StartTime := StartTime + 1;
   d[v] := StartTime;
   low[v] := d[v];  rm[v] := False;
   for each w adjacent to v do
      if d[w] = 0 then
         DFS(w);
      low[v] := min(low[v],low[w]);
   else
      if not rm[w] then
         low[v] := min(d[w],low[v]);
   end
   endif
endfor;

if low[v] = d[v] then
   rm[v] := True; Output v;
   while SC nonempty and d[Top(SC)] > d[v] do
      output Top(SC);
      rm[Top(SC)] := True;
      Pop SC;
   endwhile;
else
   Push v onto SC;
endif
end DFS

for v := 1 to n do d[v] := 0;
StartTime := 0;
for v := 1 to n do
   if d[v] = 0 then DFS(v);
endfor
Biconnected Components

DFS(v)
    StartTime := StartTime + 1;
    d[v] := StartTime;  {Init to 0}
    back[v] := d[v];
    for every w adjacent to x do
        if d[w] < d[v] then
            push (v,w) on EdgeStack
        endif
        if d[w] = 0 then
            DFS(w);
            if back[w] ≥ d[v] then
                Pop EdgeStack until (v,w) is popped
            endif
        endif
    endif
else
    back[v] := min(back[v],d(w));
endif
endfor
end DFS

begin
    for i:= 1 to n do d[i] := 0;
    StartTime := 0;
    DFS(1);
end
Bipartite Matching
Alternating Path
Reverse
Maximum Flow
An Augmenting Path
Compute Residual Network

Add all vertices of $V(G)$ to $V(H)$

For $(u,v)$ in $E(G)$ do
  if $(u,v)$ is below capacity then
    Add $(u,v)$ to $E(H)$
  endif
  if $(u,v)$ has non-zero flow then
    Add $(v,u)$ to $E(H)$
  endif
endfor

found := FALSE;
DFS(S,H)
If not found the exit
else AUGMENT
Go Back to first step;
DFS For Augmenting Path

DFS(x,H)
mark x;
if x = L then
    found := TRUE;
    exit;
endif
for y adjacent to x in H do
    if y is unseen then
        Parent[y] = x;
        DFS(y);
        endif;
    if found then
        exit
    endif
endfor
end DFS
Augment Part 1

vert := L;
Aug := \infty; 
while vert <> S do 
    if (vert, parent[vert]) is in G then 
        if residual_capacity((vert, parent[vert])) < Aug then 
            Aug := residual_capacity((vert, parent[vert]));
        endif
    else 
        if current_flow((vert, parent[vert]) < Aug then 
            Aug := current_flow((vert, parent[vert]));
        endif
    endif
endwhile

/* Augmenting value has now been computed */
Augment Part 2

vert := L;
while vert <> S do
    if (vert,parent[vert]) is in G Then
        increase flow of (vert,parent[vert]) by Aug;
    else
        decrease flow of (vert,parent[vert]) by Aug;
    endif
    vert = parent[vert];
endwhile

/* Augmentation is now complete */
Max Flow for Bipartite Match
Breadth First Search

BFS(AdjacencyList:ListType , v:Integer)

Q: QUEUE;
Initialize Q to Empty;
Visit and mark v;
Insert v into Q; {Add to tail of Q}
while Q is not Empty do
    x := Remove(Q); {Remove head of Q}
    for each unmarked vertex w adjacent to x do
        visit and mark w;
        insert w into Q;
    endfor
endwhile
end BFS
Depth First Search

DFS(adjacencyList:ListType,v:Integer)

S: Stack
initialize S to Empty;
visit and mark v;
push v into S;
while S is non-empty do
    for each unmarked vertex w adjacent to TOP(S) do
        visit and mark w;
push w into S;
endfor;
POP(S);
endwhile
end DFS
Connected Components

{Mark vertex v by assigning number > 0 to mark[v].}
{Use component number as mark value.}

for i := 1 to n do  {n = number of vertices}
mark[i] := 0;
endfor
ComponentNumber := 0;  {Global Variable}
for i := 1 to n do
  for i := 1 to n do
    if mark[i] = 0 then
      ComponentNumber := ComponentNumber + 1;
      DFS(i)
    endif
  endif
endfor
Connected Component DFS

DFS(v)  {Local declarations omitted.}
mark[v] := ComponentNumber;
ptr := AdjacencyList[v];
while ptr ≠ NIL do
    w := ptr^.vertex;
    if mark[w] = 0 then
        DFS(w)
    endif
    ptr := ptr^.Link;
endwhile
end DFS
Adjacency List Structure
Kruskal’s MST Algorithm

Create a forest of trees numbered 1 to |V|
Sort Edges by increasing weight
For each edge (x,y) do  {in ascending order by weight}
  if vertices x and y are in different trees
    Add (x,y) to the spanning tree
    Renumner all vertices in x’s tree using y’s number
  endif
endfor
Prim/Dijkstra MST
Initialization

Pick any starting vertex \( x \)
Place \( x \) in the MST
For all vertices \( y \) adjacent to \( x \) do
    Add \( y \) to the Fringe_Set;
endfor
For each element \( y \) of the Fringe_Set do
    Set weight of \( y \) equal to the weight of edge \((x,y)\)
    Set Parent of \( y \) to \( x \)
endfor
While number of vertices in MST is less than |V| do
   Find element y of Fringe_Set with minimum weight
   Add vertex y and edge (y,Parent of y) to MST
   Remove y from Fringe_Set
   For all vertices z adjacent to y do
      if z not in Fringe_Set then
         Put z into Fringe_Set;
         Set weight of z to weight of (y,z)
         Set parent of z to y
      else
         if weight of (y,z) less than weight of z then
            set weight of z to weight of (y,z)
            set parent of z to y
         endif
      endif
   endfor; endwhile
Dijkstra’s Shortest Path

Place the starting vertex A into the SP tree;
For all vertices y adjacent to A
    Add y to the FRINGE_SET;
For each element y in the FRINGE_SET
    Set the weight of y to the weight of the edge xy;
    Set parent[y] to x;
While ending vertex B is not in the SP tree and
    FRINGE_SET is not empty do
    \textbf{Execute Loop Body:}
End While
If B is not in the SP Tree
    Print Error Message
Else
    Print contents of SP tree
End If
Dijkstra’s S.P. Body

Find the element y of FRINGE_SET with minimum weight;
Add the vertex y, and edge \([y, \text{parent}[y]]\) to S.P. tree;
Remove y from FRINGE_SET;
For all vertices z adjacent to y do
  If z is not in FRINGE_SET then
    Add z to FRINGE_SET;
    Set weight(z) to weight([y,z]) + weight(y);
    Set Parent[z] to y;
  Else
    if weight(y) + weight([y,z]) < weight(z) then
      Set weight(z) to weight(y) + weight([y,z]);
      Set Parent[z] to y;
    End If
  End If
End For
Dijkstra’s “Other” Algorithm

Initialize-Single-Source(G,s)
S ← ∅
Q ← V[G]
While Q ≠ ∅ do
    u ← EXTRACT-MIN(Q)
    S ← S union {u}
    for each vertex v in Adj[u] do
        RELAX(u,v,w);
    end for
end while
The RELAX Algorithm

RELAX(u,v,w)
    if d[v] > d[u] + w(u,v) then
        d[v] ← d[u] + w(u,v)
        p[v] ← u
    End If
End RELAX
The Bellman-Ford Algorithm

INITIALIZE-SINGLE-SOURCE(G,s)
for i<-1 to |V[G]|-1 do
    for each edge (u,v) of E[G] do
        RELAX(u,v,w)
    end for
end for

for each edge (u,v) in E[G] do
    if d[v] > d[u] + w(u,v) then
        return FALSE;
    end if
end for
return TRUE