21 DFS, BFS and Spanning Trees

Instructor : Ke Wei（柯韋）

A319 ✆ Ext. 6452 ✉ wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP122/19/

Bachelor of Science in Computing, School of Public Administration, Macao Polytechnic Institute

April 17, 2019

Outline

1. **Depth First Search**
2. **Breadth First Search**
3. **Minimum Spanning Trees**
    - Prim's Algorithm
    - Kruskal's Algorithm

Depth First Search

- Depth-first search is a generalization of pre-order traversal.
- Starting from some vertex v, we visit v and then recursively traverse all the vertices adjacent to v.
- We need to be careful to avoid cycles. When we visit a vertex v, we mark it *visited*, and recursively perform depth-first search on all the adjacent vertices that have not been visited.
- Although we must mark the vertex before exploring its adjacent vertices, we may perform the real processing *before* and/or *after* the exploration, according to the application.
- We may use a stack to explicitly express the searching sequence without recursion.
- DFS generates a *spanning tree* if the graph is connected or rooted at v.
- A spanning tree of a graph G is a tree that consists of all the vertices and some of the edges of G.

Depth First Search — Recursion

```
def dfs(v, visited):
    if v.name not in visited:
        visited.add(v.name)

        # pre-order processing

        for e in v.adj_list_values:
            dfs(e.dest, visited)

        # post-order processing
```

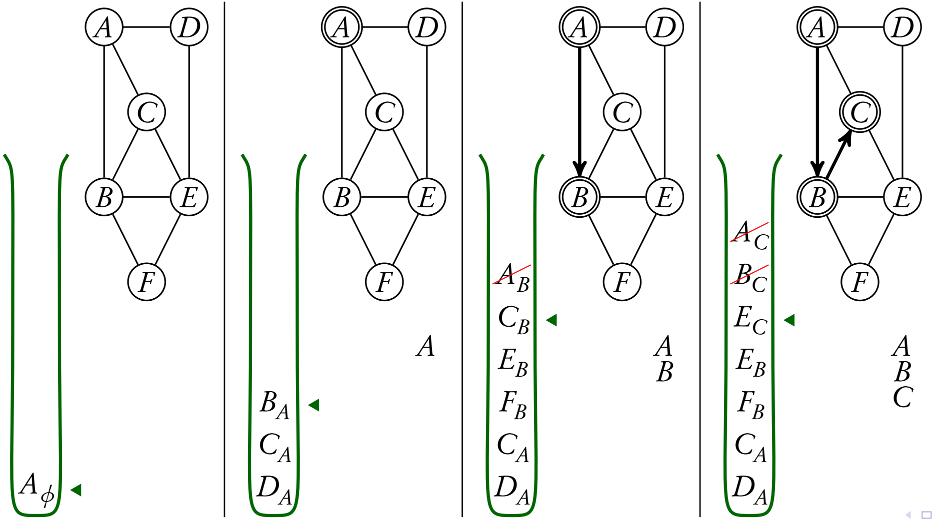
Depth First Search — Stack

```
def dfs_stack(v, visited, st):
    st.push(v)
    while st:
        w = st.pop()
        if w.name not in visited:
            visited.add(w.name)

            # pre-order processing

            for e in reversed(w.adj_list_values):
                st.push(e.dest)
```

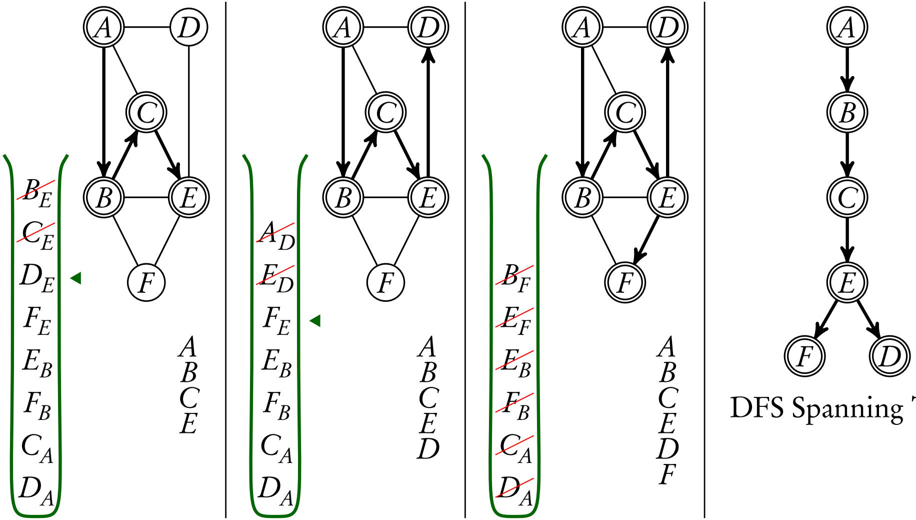
DFS Spanning Trees

```
def dfs_span_stack(v, visited, st, span):
    st.push([v])        # root
    while st:
        p = st.pop()    # path: parent -> vertex
        w = p[-1]
        if w.name not in visited:
            visited.add(w.name)
            span.add_path(u.name for u in p)

            for e in reversed(w.adj_list_values):
                st.push([w, e.dest])
```

DFS — Illustrated



DFS — Illustrated (2)



DFS Spanning Tree

Breadth First Search

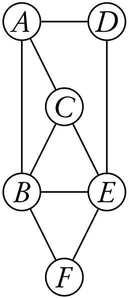
- Breadth-first search is a by-level search strategy.
- Starting from some vertex v, we visit v and then all the vertices adjacent to v, and then their adjacent vertices, and so on.
- We need the same trick, the *visited* marks, as in DFS to avoid cycles.
- A recursive definition of breadth-first search is not possible. We need a FIFO queue to line up the vertices to visit.
  - We first enqueue vertex v.
  - We repeatedly dequeue a new vertex, visit it, enqueue its adjacent vertices.
  - Until all the reachable vertices have been visited (the queue is empty.)
- BFS also generates a spanning tree if the graph is connected or rooted at v.

BFS Spanning Trees

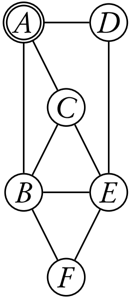
```
def bfs_span_queue(v, visited, q, span):
    q.push_back([v])  # root
    while q:
        p = q.pop()    # path: parent -> vertex
        w = p[-1]
        if w.name not in visited:
            visited.add(w.name)
            span.add_path(u.name for u in p)

            for e in w.adj_list_values:
                q.push_back([w, e.dest])
```

BFS — Illustrated

$\left(A_\phi \right)$

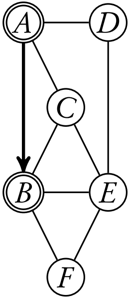


$\left(\begin{matrix} B_A \\ C_A \\ D_A \end{matrix} \right)$



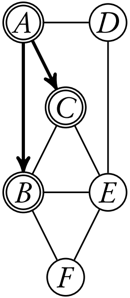
A

$\left(\begin{matrix} C_A \\ D_A \\ A_B \\ C_B \\ E_B \\ F_B \end{matrix} \right)$



A
B

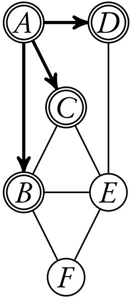
$\left(\begin{matrix} D_A \\ A_B \\ C_B \\ E_B \\ F_B \\ A_C \\ B_C \\ E_C \end{matrix} \right)$



A
B
C

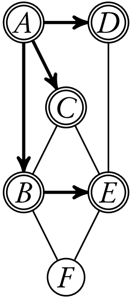
BFS — Illustrated (2)

$\cancel{A_B}$
$\cancel{C_B}$
▶ E_B
F_B
A_C
B_C
E_C
A_D
E_D



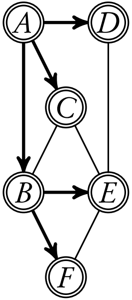
A
B
C
D

▶ F_B
A_C
B_C
E_C
A_D
E_D
B_E
C_E
D_E
F_E

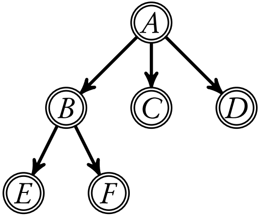


A
B
C
D
E

$\cancel{A_C}$
$\cancel{B_C}$
$\cancel{E_C}$
$\cancel{A_D}$
$\cancel{E_D}$
$\cancel{B_E}$
$\cancel{C_E}$
$\cancel{D_E}$
$\cancel{F_E}$
$\cancel{B_F}$
$\cancel{E_F}$



A
B
C
D
E
F



BFS Spanning Tree

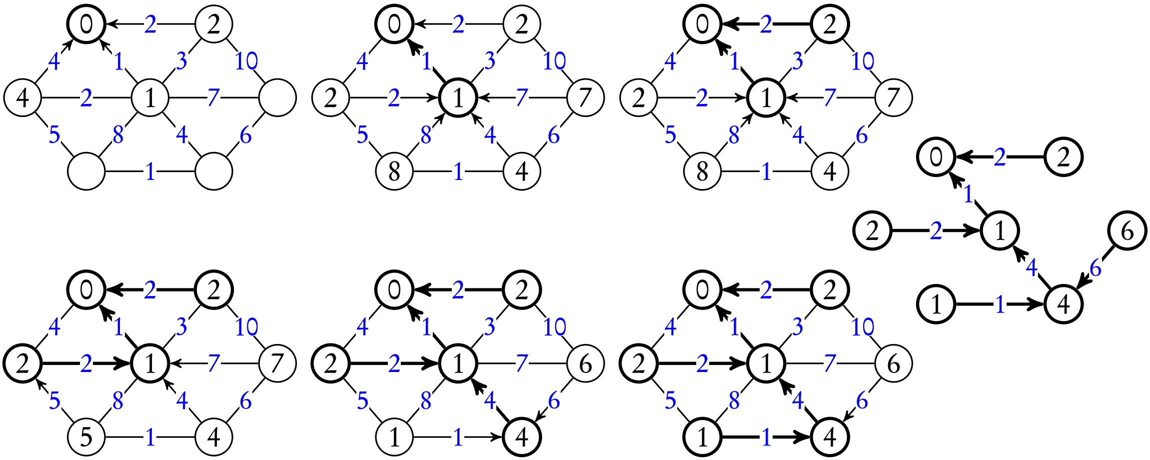
Minimum Spanning Trees

- The problem is to find the minimum spanning tree in an edge-weighted undirected graph.
- The minimum spanning tree of an undirected graph G is a tree consisting of the edges that connects all the vertices of G at the lowest total weight.
- For any spanning tree T, adding an edge e outside T creates a cycle. Removing any edge on the cycle restore the tree property.
- The total weight of the spanning tree is lowered if e is less weighted than the edge removed.
- If we always add the minimum weighted edge to a tree without creating a cycle, then the total weight can not be lowered. Thus, greed works in this problem.
- There are two ways to select the minimum weighted edge.

Prim's Algorithm

- At any point, we divide the vertices into two sets: one is the partially created tree, the other is the rest of the graph.
- We find a new vertex to add to the tree by choosing the edge (u, v) such that the weight of (u, v) is the smallest among all edges where u is in the tree and v is not.
- Similar to the Dijkstra's shortest path algorithm, we set a property $dist(v)$ to each vertex v outside the tree, it tracks the current minimum weighted edge from v to some vertex u inside the tree. We also set $parent(v)$ to u.
- When a new vertex is added to the tree, we only need to update those vertices outside the tree that are adjacent to the newly added vertex.

Prim's Algorithm — Illustrated

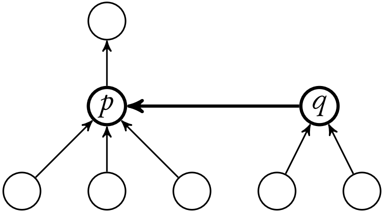


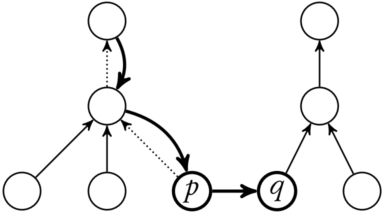
Kruskal's Algorithm

- We continually select the edges in the increasing order of weight, and accept an edge only if it does not cause a cycle.
- The algorithm maintains a forest — a collection of trees. Initially, there are $|V|$ singleton trees. Adding an edge merges two trees into one.
- When the algorithm terminates, there is only one tree left.
- We can store the edges in a heap. It is better than pre-sorting the edges since only a small fraction of edges need to be selected before the algorithm terminates.
- The problem left is to find out whether two vertices belong to the same tree. This is the scope of the *Union/Find* algorithms. At the moment, we may compare the root of the two vertices, although this may cost a lot of time.

Merging Trees Containing Two Specific Vertices

- When one vertex is the root of a tree, we can point its parent to the other vertex.
- When none of the vertices is a root, we must make one of them a root. This can be done by reverting the parent pointers along the path from the vertex to its current root.





Kruskal's Algorithm — Illustrated