Overview of Graph Algorithms

COMP 3711
Breadth First Search (BFS)

• Starting from node $s$, BFS visits all nodes reachable from $s$ in increasing order of their distance (number of edges from $s$)
• Cost $O(V+E)$ ($O(E)$ if graph is connected)
• Many applications: finding connected components, max flows..

BFS($G,s$):
for each vertex $u \in V - \{s\}$
   $u.color \leftarrow \text{white}; u.d \leftarrow \infty; u.p \leftarrow \text{nil}$
$s.color \leftarrow \text{gray}; s.d \leftarrow 0$
initialize empty FIFO queue $Q$
Enqueue($Q,s$)
while $Q \neq \emptyset$ do
   $u \leftarrow \text{Dequeue}(Q)$
   for each $v \in Adj[u]$
      if $v.color = \text{white}$ then
         $v.color \leftarrow \text{gray}; v.d \leftarrow u.d + 1; v.p \leftarrow u$
         Enqueue($Q,v$)
   $u.color \leftarrow \text{black}$
BFS Example

• Apply BFS on the following graph in the order of node id.
FIFO queue Q
Start with Q: 1

Q: 3,4

d=1
pred=1

Q: 4,6,7

d=0
pred=∅

d=1
pred=1

d=2
pred=3

BFS Example (cont)
BFS Example (cont)

Q: 6,7

Q: ∅
BFS Example (cont)

Q: 5

Q: ∅
Depth First Search (BFS)

- Starting from node \( s \), DFS recursively visits all nodes reachable from \( s \)
- Cost \( O(V+E) \) (\( O(E) \) if graph is connected)
- Many applications: cycle detection...

**DFS(\( G \)):**
for each vertex \( u \in V \) do
  \( u.color \leftarrow \text{white} \); \( u.p \leftarrow \text{nil} \)
for each vertex \( u \in V \) do
  if \( u.color=\text{white} \) then
    **DFS-Visit\((u)\)**

**DFS-Visit\((u)\):**
\( u.color \leftarrow \text{gray} \)
for each node \( v \) in adj list of \( u \) do
  if \( v.color=\text{white} \) then
    if \( v.color=\text{white} \) then
      \( v.p \leftarrow u \)
      **DFS-Visit\((v)\)**
    \( u.color \leftarrow \text{black} \)
DFS Example

• Apply DFS on the following graph in the order of node id.
DFS Example (cont)

DFS-visit(1)

1

pred=∅

DFS-visit(3)

1

pred=∅

3

pred=1

DFS-visit(3)

1

pred=∅

3

pred=1

6

pred=3

4

7

2

5
DFS Example (cont)

DFS-visit(6)
DFS-visit(4)
DFS-visit(7)

DFS-visit(2)
DFS-visit(5)
Topological Sort

• Given a DAG, generate a topological ordering: a linear ordering of the vertices such that if \((u,v)\) is in the graph, \(u\) appears before \(v\) in ordering

• Cost \(O(V+E)\)

Initialize \(Q\) to be an empty FIFO queue
for each \(u\) in \(V\) do
  if \(\text{in-degree}(u) = 0\) then
    Enqueue\((u)\);
while \(Q\) is not empty do
  \(u = \text{Dequeue}(Q)\);
  Output \(u\);
  for each \(v\) in Adj-list of \(u\) do
    \(\text{in-degree}(v) = \text{in-degree}(v) - 1\);
    if \(\text{in-degree}(v) = 0\) then
      Enqueue\((v)\);
Minimum Spanning Trees

• Given a connected, undirected, graph $G = (V, E)$, a *spanning tree* is an *acyclic* subset of edges $T \subseteq E$ that connects all the vertices together.

• Assuming $G$ is weighted, we define the *cost* of a spanning tree $T$ to be the sum of edge weights in the spanning tree

$$w(T) = \sum_{(u,v) \in T} w(u,v)$$

• A *minimum spanning tree (MST)* is a spanning tree of minimum weight.
Prim’s Algorithm: pseudo-code

MST-Prim \((G, s)\) \(//\) we start building the MST from vertex \(s\)
For each vertex \(u \in G\)
    set \(u\).key \(\leftarrow\) \(\infty\); \(u\).pred \(\leftarrow\) nil; \(u\).color \(\leftarrow\) white \(//\) key is used for min-heap
\(s\).key=0
Insert the keys of all vertices in a min-heap \(Q\) \(//\) \(Q\) contains non-visited vertices
While \(Q \neq \emptyset\)
    \(u\)=extract-min\((Q)\); \(u\).color \(\leftarrow\) black
    \(//\) initially the top is \(s\).key=0; vertices that are de-heaped are considered visited (i.e., \(u \in S\)), and will not be processed again
for each vertex \(v\) in adjacency list of \(u\)
    if \(v\).color=white and weight\((u,v)\) < \(v\).key
        \(v\).key=weight\((u,v)\)
        \(v\).pred=u \(//\) the predecessor nodes indicate the edges in the MST
decrease-key\((Q,v\).key\) \(//\) log\(V\) cost

- Cost is \(O(E\log V)\)
Prim’s Algorithm: example

Q:  pred:
 s, 0    s, nil
 a, ∞   a, nil
 b, ∞   b, nil
 d, ∞   d, nil

Q:  pred:
 a, 2    a, s
 b, 4    b, s
 d, ∞    d, nil

Q:  pred:
 d, 2    d, a
 b, 3    b, a

Q:  pred:
 b, 1    b, d

Q:  pred:
 b, 1    b, d
Kruskal’s Algorithm: idea

• We sort the edges in increasing order of their weight
• We keep adding edges to the MST provided that the new edge to be added does not create a cycle
  – We identify cycles using a disjoint-set data structure
  – Initially every node is a set
  – When we add an edge \((u,v)\), we perform a union of the sets of \(u\) and \(v\)
    – If \(u\) and \(v\) are already in the same set, there is a cycle; so the edge is not added to the MST
• We finish when we consider all edges
Kruskal’s Algorithm: pseudo-code

MST-Kruskal \((G, s)\)
For each vertex \(u \in G\)
   \text{make-set}(u) // we create a set that initial only contains \(u\)
\(A = \emptyset\)
Sort all edges in increasing order of weight
For each edge \((u,v)\) in the sorted order
   If \text{find-set}(u) \neq \text{find-set}(v)
      \(A = A \cup \{(u,v)\}\)
      \text{set-union}(u,v)

• Cost is \(O(E \log E) = O(E \log V)\) - same a cost of sorting
Kruskal’s Algorithm: example

Sorted edges:
- \((d,b), 1\)
- \((a,s), 2\)
- \((a,d), 2\)
- \((a,b), 3\)
- \((s,b), 4\)

edges:
- \((a,s), 2\)
- \((a,d), 2\)
- \((a,b), 3\)
- \((s,b), 4\)
Shortest Paths

• Given a graph $G = (V, E)$ with a weight function $w$, and a path $p=<u_0, u_1, .., u_k>$, the weight $w(p)$ of the path, is the sum of the weights of the constituent edges.

• The shortest path from $u$ to $v$ is the path connecting the two vertices that has the minimum weight.

• All algorithms, except Dijkstra, allow negative weights, but there can be no negative cycle.

• Two types of algorithms: (1) single source: find the shortest paths from a source to every other node; (2) all pairs shortest paths.
Single Source SPs with Negative Weights: Bellman-Ford Algorithm

• Like all SP algorithms it is based on edge relaxation: relaxing an edge \((u,v)\) means that we use the edge in order to find a SP to from \(s\) to \(v\) that passes through \(u\).
• Since, in the worst case the SP may contain up to \(|V|-1\) edges, it suffices to relax all edges \(|V|-1\) times.

Bellman-SP\((G,s)\)
For each node \(u\), set an upper bound \(d[u]\) of the distance to \(s\)

Initially, \(d[s]=0\), and \(d[u]=\infty \ \forall \ u \neq s\)
For \(i \leftarrow 1\) to \(|V|-1\) \// executed \(|V|-1\) times
  For each edge \(\text{RELAX}(u,v)\)

\(\text{RELAX}(u,v)\)
If \(d[u]+w(u,v) < d[v]\)

\(d[v]=d[u]+w(u,v)\)
Set \(u\) to be the predecessor of \(v\)

• Cost \(O(EV)\), which is \(O(V^3)\) for dense graphs
Single Source SPs in DAGs

• If the graph contains no cycles, we just have to relax each edge exactly once. However, when we relax \((u,v)\) to compute the distance of \(v\), we must make sure that the distance of \(u\) is final.

• To achieve this, we relax edges according to a topological order of the vertices.

DAG-Shortest-Path\((G,s)\)
topologically sort the vertices of \(G\)
for each vertex \(v \in V\)
\[
  v.d \leftarrow \infty; \ v.p \leftarrow \text{nil}; \ s.d \leftarrow 0
\]
for each vertex \(u\) in topological order
  for each vertex \(v\) in \(Adj\) list of \(u\)
    if \(u.d + w(u,v) < v.d\) then
      \[
        v.d \leftarrow u.d + w(u,v)
      \]
      \[
        v.p \leftarrow u
      \]

• Cost \(O(V+E)\) (same as topological sorting)
Single Source SPs with Positive Weights: Dijkstra Algorithm

Idea: we keep a heap with the current distances of nodes to s. The distance of the node at the top of the heap is the actual distance - it cannot decrease any further because there are no negative weights.

SP-Dijkstra(G, s)
For each node u
\[ d[u]=\infty \text{ and } u.p\text{red}=\text{nil} \] // \( d[u] \) is an upper bound of the dist. of \( u \) to \( s \)
\[ d[s]=0 \]
Insert all vertices into a min-heap \( Q \) on \( d[u] \)
While \( Q \) is not empty
\[ u=\text{extract-min}(Q) \] // \( d[u] \) is the actual distance
For each edge \((u,v)\) // \( v \) is in the adjacency list of \( u \)
If \( v \in Q \) and \( d[u]+w(u,v) < d[v] \) // relax \((u,v)\)
\[ d[v]=d[u]+w(u,v) \]
\[ v.p\text{red}=u \]
\[ \text{decrease-key}(Q,d[v]) \] // \( \log V \) cost

- Cost \( O(E\log V) \) - Very similar to Prim's algorithm
Dijkstra’s Algorithm: example

- **Q:** s, 0
  - **pred:** s, nil
- **Q:** a, ∞
  - **pred:** a, nil
- **Q:** b, ∞
  - **pred:** b, nil
- **Q:** d, ∞
  - **pred:** d, nil

1. **Q:** b, 4
   - **pred:** b, s
   - **pred:** d, a

2. **Q:** d, 4
   - **pred:** d, a
All-pairs Shortest Paths: Floyd-Warshall Algorithm

• Goal: determine the length of the shortest path (i.e., distance) between all pairs of vertices in a weighted digraph $G$
  – $G$ may contain negative edges but no negative cycles
• Example:

Output:
- $d(1,2)=d(2,1)=1$
- $d(1,3)=d(3,1)=1$
- $d(1,4)=d(4,1)=2$
- $d(2,3)=d(3,2)=0$
- $d(2,4)=d(4,2)=1$
- $d(3,4)=d(4,3)=1$
Intermediate Vertices

- For convenience, we name the vertices 1, 2, ..., n.
- Let $d^k_{ij}$ be the length of the shortest path from vertex $i$ to $j$ such that all intermediate vertices on the path (if any) are in the set \{1, 2, .., $k$\}, i.e., they are among the first $k$ vertices.
- Example:

\[
\begin{array}{c|c|c|c}
 & k=0 & k=1 & k=2 \\
\hline
\text{Distances for } k=0 & d(1,2)=d(2,1)=3 & d(1,2)=d(2,1)=3 & d(1,2)=d(2,1)=3 \\
& d(1,3)=d(3,1)=1 & d(1,3)=d(3,1)=1 & d(1,3)=d(3,1)=1 \\
& d(1,4)=d(4,1)=\infty & d(1,4)=d(4,1)=\infty & d(1,4)=d(4,1)=4 \\
& d(2,3)=d(3,2)=0 & d(2,3)=d(3,2)=0 & d(2,3)=d(3,2)=0 \\
& d(2,4)=d(4,2)=1 & d(2,4)=d(4,2)=1 & d(2,4)=d(4,2)=1 \\
& d(3,4)=d(4,3)=2 & d(3,4)=d(4,3)=2 & d(3,4)=d(4,3)=1 \\
\end{array}
\]
Optimal Substructure for Floyd-Warshall

• In general, for a shortest path from vertex $i$ to $j$ with intermediate vertices in the set $\{1, 2, \ldots, k\}$, there are two cases:
  • **Case 1:** The shortest path does not contain vertex $k$, in which case: $d^k_{ij} = d^{k-1}_{ij}$
  • **Case 2:** The shortest path contains vertex $k$, in which case: $d^k_{ij} = d^{k-1}_{ik} + d^{k-1}_{kj}$

• Thus, we need to compute all distances $d^1_{ij}, d^2_{ij}, \ldots, d^n_{ij}$ (for all pairs). The final distance between vertices $i$ and $j$ is $d^n_{ij}$
  • $d^0_{ij} = \text{weight of edge } (i, j)$

• We can reduce the space consumption of the algorithm by only keeping the most recent values of $d^k_{ij}$
  • Because $d^{k-1}_{ik} = d^k_{ik}$ (and $d^{k-1}_{kj} = d^k_{kj}$) we do not care about the order by which values are produced
Floyd-Warshall: pseudo-code

Floyd-Warshall(n) // n is the number of vertices
for i = 1 to n do
    for j = 1 to n do
        \(d_{ij} = \text{weight of edge } (i, j)\) // initialization
for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            if \(d_{ik} + d_{kj} < d_{ij}\) then
                \(d_{ij} = d_{ik} + d_{kj}\)

• Cost is \(\Theta(n^3)\)
Maximum Flow

• Given a directed graph with special vertices $s$ and $t$, find a maximum flow from $s$ to $t$.
  – The flow through each edge $(u, v)$ cannot exceed the capacity of $(u, v)$.
  – For each node $v \in V \setminus \{s, t\}$, the total amount of flow entering $v$ must be equal to the total amount of flow leaving $v$.
• Finding the maximum flow is equivalent to finding the min-cut: i.e., the partition of nodes in two disjoint sets $S$ and $T$ such that $s \in S$, $t \in T$ and the sum of capacities of edges connecting a node in $S$ with a node in $T$ are minimized.
• Residual graph.
  – For every edge $(u,v)$ in the original graph, create an opposite edge $(v,u)$. Initially, the residual capacity $c_f(u,v)$ equals the capacity of $(u,v)$, and $c_f(v,u)=0$.
  – Let $f(u,v)$ be the flow passing through edge $(u, v)$: $c_f(u,v)$ decreases by $f(u,v)$ and $c_f(v,u)$ increases by $f(u,v)$.
• The following Ford Fulkerson Algorithm operates in the residual graph.
Ford Fulkerson Algorithm

Start with $f(e) = 0$ for each edge $e$.

Construct Residual Graph $G_f$ for current flow $f(e) = 0$

While there exists some $s$-$t$ path $P$ in $G_f$

Let $c_f(P)$ be the maximum amount of flow that can be pushed through $P$

Push $c_f(P)$ units of flow along the edges $e \in P$

Construct $G_f$ for new flow $f(e)$

Complexity: Assuming that all capacities are integer, and we choose paths at random, the worst case cost is $O(Ef^*)$, where $f^*$ is the maximum flow (pseudo-polynomial complexity).

If we choose, shortest paths (in terms of number of edges), the cost is $O(VE^2)$. Several problems, such as the maximum bi-partite matching, can be solved by maximum flows.