Unit 2: Algorithmic Graph Theory

- Course contents:
  - Introduction to graph theory
  - Basic graph algorithms

- Reading
  - Chapter 3

Algorithms

- **Algorithm**: A well-defined procedure for transforming some input to a desired output.

- **Major concerns**:
  - **Correctness**: Does it halt? Is it correct?
  - **Efficiency**: Time complexity? Space complexity?
    - Worst case? Average case? (Best case?)

- **Better algorithms?**
  - **How**: Faster algorithms? Algorithms with less space requirement?
  - **Optimality**: Prove that an algorithm is best possible/optimal? Establish a lower bound?
Example: Traveling Salesman Problem (TSP)

- **Instance:** A set of points (cities) $P$ together with a distance $d(p, q)$ between any pair $p, q \in P$.
- **Output:** What is the shortest circular route that starts and ends at a given point and visits all the points.

- Correct and efficient algorithms?

Nearest Neighbor Tour

1. pick and visit an initial point $p_0$;
2. $P \leftarrow p_0$;
3. $i \leftarrow 0$;
4. **while** there are unvisited points **do**
   5. visit $p_i$’s closest unvisited point $p_{i+1}$;
   6. $i \leftarrow i + 1$;
   7. return to $p_0$ from $p_i$.

- Simple to implement and very efficient, but **incorrect**!
A Correct, but Inefficient Algorithm

1. \( d \leftarrow \infty \);  
2. for each of the \( n! \) permutations \( \pi_i \) of the \( n \) points  
3. \( \text{if} \ (\text{cost}(\pi_i) \leq d) \ \text{then} \)  
4. \( d \leftarrow \text{cost}(\pi_i); \)  
5. \( T_{\text{min}} \leftarrow \pi_i; \)  
6. return \( T_{\text{min}} \).

- **Correctness:** Tries all possible orderings of the points \( \Rightarrow \) Guarantees to end up with the shortest possible tour.
- **Efficiency:** Tries \( n! \) possible routes!  
  - 120 routes for 5 points, 3,628,800 routes for 10 points, 20 points?  
- **No known efficient, correct algorithm for TSP!**

---

**Example: Sorting**

- **Instance:** A sequence of \( n \) numbers \(<a_1, a_2, \ldots, a_n>\).
- **Output:** A permutation \(<a_1', a_2', \ldots, a_n'>\) such that \( a_1' \leq a_2' \leq \ldots \leq a_n' \).  
  
  Input: \(<8, 6, 9, 7, 5, 2, 3>\)  
  
  Output: \(<2, 3, 5, 6, 7, 8, 9 >\)
- **Correct and efficient algorithms?**
Insertion Sort

InsertionSort(A)
1. for j ← 2 to length[A] do
2. key ← A[j];
3. /* Insert A[j] into the sorted sequence A[1..j-1]. */
4. i ← j - 1;
5. while i > 0 and A[i] > key do
7. i ← i - 1;
8. A[i+1] ← key;

Graph

- **Graph**: A mathematical object representing a set of “points” and “interconnections” between them.
- A graph \( G = (V, E) \) consists of a set \( V \) of vertices (nodes) and a set \( E \) of directed or undirected edges.
  - \( V \) is the vertex set: \( V = \{v_1, v_2, v_3, v_4, v_5, v_6\} \), \(|V|=6\)
  - \( E \) is the edge set: \( E = \{e_1, e_2, e_3, e_4, e_5\} \), \(|E|=5\)
  - An edge has two endpoints, e.g. \( e_1 = (v_1, v_2) \)
  - For simplicity, use \( V \) for \(|V|\) and \( E \) for \(|E|\).
Example Graphs

- Any binary relation is a graph.
  - Network of roads and cities
  - Circuit representation

Terminology

- Degree of a vertex: \( \text{degree}(v_3) = 3 \), \( \text{degree}(v_2) = 2 \)
- Subgraph of a graph:
- Complete (sub)graph: \( V' = \{v_1, v_2, v_3\} \), \( E' = \{e_1, e_2, e_3\} \)
- (Maximal/maximum) clique: maximal/maximum complete subgraph
- Selfloop
- Parallel edges
- Simple graph
- Multigraph
Terminology (cont’d)

• Bipartite graph \( G = (V_1, V_2, E) \)
• Path
• Cycle: a closed path
• Connected vertices
• Connected graph
• Connected components

A bipartite graph

Path \( p = <v_1, v_2, v_3, v_4> \)
Cycle \( C = <v_1, v_2, v_3, v_1> \)

Terminology (cont’d)

• Weighted graph:
  - Edge weighted and/or vertex weighted
• Directed graph: edges have directions
  - Directed path
  - Directed cycle
  - Directed acyclic graph (DAG)
  - In-degree, out-degree
  - Strongly connected vertices
    - Strongly connected components \{v1\}{v2, v3, v4, v5}\)
  - Weekly connected vertices

Weekly connected vertices
Graph Representation: Adjacency List

- **Adjacency list**: An array \( Adj \) of \(|V|\) lists, one for each vertex in \( V \). For each \( u \in V \), \( Adj[u] \) pointers to all the vertices adjacent to \( u \).
- Advantage: \( O(V+E) \) storage, good for **sparse** graph.
- Drawback: Need to traverse list to find an edge.

---

Graph Representations: Adjacency Matrix

- **Adjacency matrix**: A \(|V| \times |V|\) matrix \( A = (a_{ij}) \) such that
  \[
  a_{ij} = \begin{cases} 
  1 & \text{if } (i, j) \in E \\
  0 & \text{otherwise}
  \end{cases}
  \]
- Advantage: \( O(1) \) time to find an edge.
- Drawback: \( O(V^2) \) storage, suitable for **dense** graph.
- How to save space if the graph is undirected?
Explicit Edges and Vertices

struct vertex {
    int vertex_index;
    struct edge *outgoing_edges;
};

struct edge {
    int edge_index;
    struct vertex *from, *to;
    struct edge *next;
};

Tradeoffs between Adjacency List and Matrix

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Winner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faster to find an edge?</td>
<td>matrix</td>
</tr>
<tr>
<td>Faster to find vertex degree?</td>
<td>list</td>
</tr>
<tr>
<td>Faster to traverse the graph?</td>
<td>list $O(V + E)$ vs. matrix $O(V^2)$</td>
</tr>
<tr>
<td>Storage for sparse graph?</td>
<td>list $O(V + E)$ vs. matrix $O(V^2)$</td>
</tr>
<tr>
<td>Storage for dense graph?</td>
<td>matrix (small win)</td>
</tr>
<tr>
<td>Edge insertion or deletion?</td>
<td>matrix $O(1)$</td>
</tr>
<tr>
<td>Weighted-graph implementation?</td>
<td>?</td>
</tr>
<tr>
<td>Better for most applications?</td>
<td>list</td>
</tr>
</tbody>
</table>
**Depth-First Search (DFS)** [Cormen]

**DFS(G)**
1. for each vertex \( u \in V[G] \)
2. \( \text{color}[u] \leftarrow \text{WHITE}; \)
3. \( \pi [u] \leftarrow \text{NIL}; \)
4. \( \text{time} \leftarrow 0; \)
5. for each vertex \( u \in V[G] \)
6. if \( \text{color}[u] = \text{WHITE} \)
7. DFS-Visit(u).

**DFS-Visit(u)**
1. \( \text{color}[u] \leftarrow \text{GRAY}; \)
   /* White vertex \( u \) has just been discovered. */
2. \( d[u] \leftarrow \text{time} \leftarrow \text{time} + 1; \)
3. for each vertex \( v \in \text{Adj}[u] \)
   /* Explore edge \( (u,v) \). */
4. if \( \text{color}[v] = \text{WHITE} \)
5. \( \pi [v] \leftarrow u; \)
6. DFS-Visit(v);
7. \( \text{color}[u] \leftarrow \text{BLACK}; \)
   /* Blacken \( u \); it is finished. */
8. \( f[u] \leftarrow \text{time} \leftarrow \text{time} + 1. \)

**DFS Example** [Cormen]

- **color**[u]: white \( \rightarrow \) gray \( \rightarrow \) black.
- Depth-first forest: \( G_\pi = (V, E_\pi), E_\pi = \{ (\pi[v], v) \in E \mid v \in V, \pi[v] \neq \text{NIL} \} \)
  - \{u \rightarrow v \rightarrow x \rightarrow y\} \{w \rightarrow z\}
DFS Pseudo Code in Text

/* Given is the graph \( G(V, E) \) */

\begin{verbatim}
struct vertex {
  ... int mark;
};

dfs(struct vertex v)
{
  v.mark ← 0;
  "process v";
  for each \( (v, u) \in E \) {
    "process \( (v, u) \)";
    if \( u.mark \)
      dfs(u);
  }
}
\end{verbatim}

main ()
{
  for each \( v \in V \)
    v.mark ← 1;
  for each \( v \in V \)
    if \( v.mark \)
      dfs(v);
}

DFS Application 1: Topological Sort

- A topological sort of a directed acyclic graph (DAG) \( G = (V, E) \) is a linear ordering of \( V \) s.t. \( (u, v) \in E \Rightarrow u \) appears before \( v \).

  **Topological-Sort(G)**
  1. call DFS(G) to compute finishing times \( f[v] \) for each vertex \( v \)
  2. as each vertex is finished, insert it onto the front of a linked list
  3. return the linked list of vertices

- Time complexity: \( O(V+E) \) (adjacent list).

Vertices are arranged from left to right in order of decreasing finishing times.
**DFS Application 2: Hightower’s Maze Router**

- A single escape point on each line segment.
- If a line parallels to the blocked cells, the escape point is placed just past the endpoint of the segment.
- Time and space complexities: $O(L)$, where $L$ is the # of line segments generated.

**Breadth-First Search (BFS) [Cormen]**

1. **BFS(G,s)**
2. 1. for each vertex $u \in V[G]-\{s\}$
3. 2. $color[u] \leftarrow$ WHITE;
4. 3. $d[u] \leftarrow \infty$;
5. 4. $\pi [u] \leftarrow$ NIL;
6. 5. $color[s] \leftarrow$ GRAY;
7. 6. $d[s] \leftarrow 0$;
8. 7. $\pi [s] \leftarrow$ NIL;
9. 8. $Q \leftarrow \{ s \}$;
10. 9. while $Q \neq \emptyset$
11. 10. $u \leftarrow$ head($Q$);
12. 11. for each vertex $v \in Adj[u]$
13. 12. if $color[v] =$ WHITE
14. 13. $color[v] \leftarrow$ GRAY;
15. 14. $d[v] \leftarrow d[u]+1$;
16. 15. $\pi [v] \leftarrow u$;
17. 16. Enqueue($Q,v$);
18. 17. Dequeue($Q$);
19. 18. $color[u] \leftarrow$ BLACK.

- **$color[u]$**:
  - white (undiscovered) $\rightarrow$
  - gray (discovered) $\rightarrow$
  - black (explored: out edges are all discovered)

- **$d[u]$**: distance from source $s$

- **$\pi[u]$**: predecessor of $u$

- Use queue for gray vertices

- Time complexity: $O(V+E)$ (adjacency list).
BFS Example [Cormen]

- Use queue for gray vertices.
  - Each vertex is enqueued and dequeued once: $O(V)$ time.
  - Each edge is considered once: $O(E)$ time.
- Breadth-first tree:
  - $G_\pi = (V_\pi, E_\pi)$, $V_\pi = \{v \in V \mid \pi[v] \neq \text{NIL}\} \cup \{s\}$
    - $\{s, w, r, t, x, v, u, y\}$
  - $E_\pi = \{\pi[v], v \in E \mid v \in V_\pi - \{s\}\}$
    - $\{(s, w), (s, r), (w, t), (w, x), (r, v), (t, u), (x, y)\}$

BFS Pseudo Code in Text

```c
bfs(struct vertex v)
{
    struct fifo *Q;
    struct vertex u, w;
    Q ← ();
    shift_in(Q, v);
    do { w ← shift_out(Q);
        "process w",
        for each (w, u) ∈ E {
            "process (w, u)",
            if (u.mark) {
                u.mark ← 0;
                shift_in(Q, u);
            }
        }
    } while (Q ≠ ()
}
```
BFS Application: Lee’s Maze Router

- Find a path from S to T by “wave propagation.”
- Discuss mainly on single-layer routing
- Strength: Guarantee to find a minimum-length connection between 2 terminals if it exists.
- Weakness: Time & space complexity for an $M \times N$ grid: $O(MN)$ (huge!)

**BFS + DFS Application: Soukup’s Maze Router**

- Depth-first (line) search is first directed toward target T until an obstacle or T is reached.
- Breadth-first (Lee-type) search is used to “bubble” around an obstacle if an obstacle is reached.
- Time and space complexities: $O(MN)$, but 10--50 times faster than Lee's algorithm.
- Find a path between S and T, but may not be the shortest!
Shortest Paths (SP)

• **The Shortest Path (SP) Problem**
  
  - **Given:** A directed graph $G=(V, E)$ with edge weights, and a specific **source node** $s$.
  
  - **Goal:** Find a minimum weight path (or cost) from $s$ to every other node in $V$.

• **Applications:** weights can be distances, times, wiring cost, delay, etc.

• **Special case:** BFS finds shortest paths for the case when all edge weights are 1.

Weighted Directed Graph

• A weighted, directed graph $G = (V, E)$ with the weight function $w: E \rightarrow \mathbb{R}$.
  
  - Weight of path $p = <v_0, v_1, \ldots, v_k>$: $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$.
  
  - **Shortest-path weight** from $u$ to $v$, $\delta(u, v)$:

    $$\delta(u, v) = \begin{cases} \min\{w(p) : u \xrightarrow{p} v\} & \text{if there is a path from } u \text{ to } v, \\ \infty & \text{otherwise.} \end{cases}$$

• **Warning!** negative-weight edges/cycles are a problem.
  
  - Cycle $<e, f, e>$ has weight $-3 < 0 \Rightarrow \delta(s, g) = -\infty$.
  
  - Vertices $h, i, j$ not reachable from $s \Rightarrow \delta(s, h) = \delta(s, i) = \delta(s, j) = \infty$.

• Algorithms apply to the cases for negative-weight edges/cycles??
Optimal Substructure of a Shortest Path

- Subpaths of shortest paths are shortest paths.
  - Let \( p = \langle v_1, v_2, \ldots, v_k \rangle \) be a shortest path from vertex \( v_i \) to vertex \( v_k \), and \( p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle \) be the subpath of \( p \) from vertex \( v_i \) to vertex \( v_j \), \( 1 \leq i \leq j \leq k \). Then, \( p_{ij} \) is a shortest path from \( v_i \) to \( v_j \). (NOTE: reverse is not necessarily true!)

- Suppose that a shortest path \( p \) from a source \( s \) to a vertex \( v \) can be decomposed into \( s \xrightarrow{p'} u \rightarrow v \). Then, \( \delta(s, v) = \delta(s, u) + w(u, v) \).

- For all edges \( (u, v) \in E \), \( \delta(s, v) \leq \delta(s, u) + w(u, v) \).

\[
\begin{align*}
\text{Initialize-Single-Source(G, s)} & \\
1. & \text{for each vertex } v \in V[G] \\
2. & d[v] \leftarrow \infty; \quad \text{ /* upper bound on the weight of a shortest path from } s \text{ to } v \text{ */} \\
3. & \pi[v] \leftarrow \text{NIL}; \quad \text{ /* predecessor of } v \text{ */} \\
4. & d[s] \leftarrow 0;
\end{align*}
\]

\[
\begin{align*}
\text{Relax(u, v, w)} & \\
1. & \text{if } d[v] > d[u] + w(u, v) \\
2. & d[v] \leftarrow d[u] + w(u, v); \\
3. & \pi[v] \leftarrow u;
\end{align*}
\]

- \( d[v] \leq d[u] + w(u, v) \) after calling Relax\((u, v, w)\).
- \( d[v] \geq \delta(s, v) \) during the relaxation steps; once \( d[v] \) achieves its lower bound \( \delta(s, v) \), it never changes.

- Let \( s \xrightarrow{p'} u \rightarrow v \) be a shortest path. If \( d[u] = \delta(s, u) \) prior to the call Relax\((u, v, w)\), then \( d[v] = \delta(s, v) \) after the call.
Dijkstra's Shortest-Path Algorithm

\[
\text{Dijkstra}(G, w, s)
\]
1. Initialize-Single-Source\( (G, s) \);
2. \( S \leftarrow \emptyset \);
3. \( Q \leftarrow V[G] \);
4. while \( Q \neq \emptyset \)
5. \( u \leftarrow \text{Extract-Minimum-Element}(Q) \);
6. \( S \leftarrow S \cup \{u\} \);
7. for each vertex \( v \in \text{Adj}[u] \)
8. \( \text{Relax}(u, v, w) \);

• Idea:
  - search all shortest paths
    - In a smart way (use dynamic-programming, see next lecture)
  - Then choose a shortest path

Example: Dijkstra's Shortest-Path Algorithm

• Find the shortest path from vertex \( s \) to vertex \( v \)
  - \( s \rightarrow x \rightarrow u \rightarrow v \); Weight = 5+3+1
Runtime Analysis of Dijkstra's Algorithm

Dijkstra(G, w, s)
1. Initialize-Single-Source(G, s);
2. S ← ∅;
3. Q ← V[G];
4. while Q ≠ ∅
5. u ← Extract-Minimum-Element(Q);
6. S ← S ∪ {u};
7. for each vertex v ∈ Adj[u]
8. Relax(u, v, w);

• Q is implemented as a linear array: \(O(V^2)\).
  – Line 5: \(O(V)\) for Extract-Minimum-Element, so \(O(V^2)\) with the while loop.
  – Lines 7--8: \(O(E)\) operations, each takes \(O(1)\) time.

• Q is implemented as a binary heap: \(O(E \lg V)\).

• Q is implemented as a Fibonacci heap: \(O(E + V \lg V)\).

Dijkstra’s SP Pseudo Code in Text

```plaintext
struct vertex {
    ...
    int distance;
};

dijkstra(set of struct vertex V, struct vertex v_s, struct vertex v_t) {
    set of struct vertex T;
    struct vertex u, v;
    V ← V \ {v_s};
    T ← {v_s};
    v_s.distance ← 0;
    for each u ∈ V
        if ((v_s, u) ∈ E)
            u.distance ← w((v_s, u))
        else u.distance ← +∞;
    while (v_t ∉ T) {
        u ← “u ∈ V, such that ∀v ∈ V : u.distance ≤ v.distance”;
        T ← T ∪ {u};
        V ← V \ {u};
        for each v “such that (u, v) ∈ E”
            if (v.distance > w((u, v)) + u.distance)
                v.distance ← w((u, v)) + u.distance;
    }
}```
Minimum Spanning Tree (MST)

- Given an undirected graph $G = (V, E)$ with weights on the edges, a **minimum spanning tree (MST)** of $G$ is a subset $T \subseteq E$ such that
  - $T$ has no cycles
  - $T$ contains all vertices in $V$
  - sum of the weights of all edges in $T$ is minimum.
- Number of edges in T is number of vertices minus one
- Applications: circuit interconnection (minimizing tree **radius**), communication network (minimizing tree **diameter**), etc.

Prim's MST Algorithm

```
MST-Prim(G,w,r)
1. Q ← V[G];
2. for each vertex $u \in Q$
3. key[$u$] ← $\infty$;
4. key[r] ← 0;
5. $\pi[r] ←$ NIL;
6. while Q ≠ ∅
7. u ← Extract-Minimum-Element(Q);
8. for each vertex $v \in Adj[u]$
9. if $v \in Q$ and $w(u,v) < key[v]$
10. $\pi[v] ← u$;
11. key[v] ← $w(u,v)$
```

- Starts from a vertex and grows until the tree spans all the vertices.
  - The edges in $A$ always form a single tree.
  - At each step, a safe, minimum-weighted edge connecting a vertex in $A$ to a vertex in $V - A$ is added to the tree.
Example: Prim's MST Algorithm

Time Complexity of Prim's MST Algorithm

MST-Prim(\(G, w, r\))
1. \(Q \leftarrow V[G]\);
2. for each vertex \(u \in Q\)
3. \(key[u] \leftarrow \infty;\)
4. \(key[r] \leftarrow 0;\)
5. \(\pi[r] \leftarrow \text{NIL};\)
6. while \(Q \neq \emptyset\)
7. \(u \leftarrow \text{Extract-Minimum-Element}(Q);\)
8. for each vertex \(v \in \text{Adj}[u]\)
9. if \(v \in Q\) and \(w(u, v) < key[v]\)
10. \(\pi[v] \leftarrow u;\)
11. \(key[v] \leftarrow w(u, v)\)

- Straightforward implementation: \(O(V^2)\) time
  - Lines 1--5: \(O(V)\).
  - Line 7: \(O(V)\) for Extract-Minimum-Element, so \(O(V^2)\) with the while loop.
  - Lines 8--11: \(O(E)\) operations, each takes \(O(lg V)\) time.
- Run in \(O(E \ lg V)\) time if \(Q\) is implemented as a binary heap
- Run in \(O(E + Vlg V)\) time if \(Q\) is implemented as a Fibonacci heap
prim(set of struct vertex V)
{
    set of struct edge E;
    set of struct vertex W;
    struct vertex u;
    u ← “any vertex from V”;
    V ← V \ {u};
    W ← {u};
    F ← ∅;
    for each v ∈ V
        if ((u, v) ∈ E)
            v.distance ← w((u, v));
            v.via_edge ← (u, v);
        else v.distance ← +∞;
    while (V ≠ ∅)
        u ← “u ∈ V, such that ∀v ∈ V : u.distance ≤ v.distance”;
        W ← W ∪ {u};
        V ← V \ {u};
        F ← F ∪ {u.via_edge};
        for each v “such that (u, v) ∈ E”
            if (u.distance > w((u, v)))
                v.distance ← w((u, v));
                v.via_edge ← (u, v);