Graph Algorithms

Graphs are useful for representing a variety of concepts:
- Data Structures
- Relationships
- Families
- Communication Networks
- Road Maps

A Tree Proof

Definition: A free tree is a connected, undirected graph that has no cycles.

Theorem: If \( T \) is a free tree having \( n \) vertices, then \( T \) has exactly \( n - 1 \) edges.

Proof: By induction on \( n \).
- Base Case: \( n = 1 \). \( T \) consists of 1 vertex and 0 edges.
- Inductive Hypothesis: The theorem is true for a tree having \( n - 1 \) vertices.
- Inductive Step:
  - If \( T \) has \( n \) vertices, then \( T \) contains a vertex of degree 1.
  - Remove that vertex and its incident edge to obtain \( T' \), a free tree with \( n - 1 \) vertices.
  - By IH, \( T' \) has \( n - 2 \) edges.
  - Thus, \( T \) has \( n - 1 \) edges.

Graph Traversals

Various problems require a way to traverse a graph – that is, visit each vertex and edge in a systematic way.

Three common traversals:
- Eulerian tours
  Traverse each edge exactly once
- Depth-first search
  Keeps vertices on a stack
- Breadth-first search
  Keeps vertices on a queue
Eulerian Tours

A circuit that contains every edge exactly once.

Example:

```
f
ce
ba
d
```

Tour: b a f c d e.

Example:

```
f
ce
ba
d
g
```

No Eulerian tour. How can you tell for sure?

Eulerian Tours

A circuit that contains every edge exactly once.

Example:

```
f
ce
ba
d
```

Tour: b a f c d e.

Example:

```
f
ce
ba
d
g
```

No Eulerian tour. How can you tell for sure?

Eulerian Tour Proof

- **Theorem:** A connected, undirected graph with \( m \) edges that has no vertices of odd degree has an Eulerian tour.
- **Proof:** By induction on \( m \).
- **Base Case:**
  - Inductive Hypothesis:
    - Inductive Step:
      - Start with an arbitrary vertex and follow a path until you return to the vertex.
      - Remove this circuit. What remains are connected components \( G_1, G_2, ..., G_k \) each with nodes of even degree and \(< m \) edges.
      - By IH, each connected component has an Eulerian tour.
      - Combine the tours to get a tour of the entire graph.

Depth First Search

```java
void DFS(Graph G, int v) { // Depth first search
    PreVisit(G, v); // Take appropriate action
    G.setMark(v, VISITED);
    for (Edge w = each neighbor of v)
        if (G.getMark(G.v2(w)) == UNVISITED)
            DFS(G, G.v2(w));
    PostVisit(G, v); // Take appropriate action
}
```

Initial call: `DFS(G, r)` where \( r \) is the root of the DFS.

Cost: \( O(|V| + |E|) \).

Depth First Search Example

The directions are imposed by the traversal. This is the Depth First Search Tree.
**DFS Tree**

If we number the vertices in the order that they are marked, we get **DFS numbers**.

**Lemma 7.2:** Every edge $e \in E$ is either in the DFS tree $T$, or connects two vertices of $G$, one of which is an ancestor of the other in $T$.

**Proof:** Consider the first time an edge $(v, w)$ is examined, with $v$ the current vertex.
- If $w$ is unmarked, then $(v, w)$ is in $T$.
- If $w$ is marked, then $w$ has a smaller DFS number than $v$ AND $(v, w)$ is an unexamined edge of $w$.
- Thus, $w$ is still on the stack. That is, $w$ is on a path from $v$.

**DFS for Directed Graphs**

- Main problem: A connected graph may not give a single DFS tree.
- Forward edges: $(1, 3)$
- Back edges: $(5, 1)$
- Cross edges: $(6, 1), (8, 7), (9, 5), (9, 8), (4, 2)$
- **Solution:** Maintain a list of unmarked vertices. Whenever one DFS tree is complete, choose an arbitrary unmarked vertex as the root for a new tree.

**Directed Cycles**

**Lemma 7.4:** Let $G$ be a directed graph. $G$ has a directed cycle iff every DFS of $G$ produces a back edge.

**Proof:**
- Suppose a DFS produces a back edge $(v, w)$.
  - $v$ and $w$ are in the same DFS tree, $w$ an ancestor of $v$.
  - $(v, w)$ and the path in the tree from $w$ to $v$ form a directed cycle.
- Suppose $G$ has a directed cycle $C$.
  - Do a DFS on $G$.
  - Let $w$ be the vertex of $C$ with smallest DFS number.
  - Let $(v, w)$ be the edge of $C$ coming into $w$.
  - $v$ is a descendant of $w$ in a DFS tree.
  - Therefore, $(v, w)$ is a back edge.

**Breadth First Search**

- Like DFS, but replace stack with a queue.
- Visit vertex's neighbors before going deeper in tree.
Breadth First Search Algorithm

```java
void BFS(Graph G, int start) {
    Queue Q(G.n());
    Q.enqueue(start);
    G.setMark(start, VISITED);
    while (!Q.isEmpty()) {
        int v = Q.dequeue(); // Take appropriate action
        PreVisit(G, v); // Take appropriate action
        for (Edge w = each neighbor of v)
            if (G.getMark(G.v2(w)) == UNVISITED) {
                G.setMark(G.v2(w), VISITED);
                Q.enqueue(G.v2(w));
            }
        PostVisit(G, v); // Take appropriate action
    }
}
```

Non-tree edges connect vertices at levels differing by 0 or 1.

Topological Sort

Problem: Given a set of jobs, courses, etc. with prerequisite constraints, output the jobs in an order that does not violate any of the prerequisites.

```
J1 J2
J3 J4
J5 J7
J6
```

Topological Sort Algorithm

```java
void topsort(Graph G) { // Top sort: recursive
    for (int i=0; i<G.n(); i++) // Initialize Mark
        G.setMark(i, UNVISITED);
    for (i=0; i<G.n(); i++) // Process vertices
        if (G.getMark(i) == UNVISITED)
            tophelp(G, i); // Call helper
}
```

Prints in reverse order.
Queue-based Topological Sort

```c
void topsort(Graph G) { // Top sort: Queue
    Queue Q(G.n()); int Count[G.n()];
    for (int v=0; v<G.n(); v++) Count[v] = 0;
    for (v=0; v<G.n(); v++) // Process every edge
        for (Edge w each neighbor of v)
            Count[G.v2(w)]++; // Add to v2’s count
    for (v=0; v<G.n(); v++) // Initialize Queue
        if (Count[v] == 0) Q.enqueue(v);
    while (!Q.isEmpty()) { // Process the vertices
        int v = Q.dequeue(); // PreVisit for v
        printout(v);
        for (Edge w = each neighbor of v) {
            Count[G.v2(w)]--; // One less prereq
            if (Count[G.v2(w)]==0) Q.enqueue(G.v2(w));
        }
    }
}
```

Shortest Paths Problems

Input: A graph with weights or costs associated with each edge.

Output: The list of edges forming the shortest path.

Sample problems:
- Find the shortest path between two specified vertices.
- Find the shortest path from vertex S to all other vertices.
- Find the shortest path between all pairs of vertices.

Our algorithms will actually calculate only distances.

Shortest Paths Definitions

d(A, B) is the shortest distance from vertex A to B.

w(A, B) is the weight of the edge connecting A to B.
- If there is no such edge, then w(A, B) = ∞.

Single Source Shortest Paths

Given start vertex s, find the shortest path from s to all other vertices.

Try 1: Visit all vertices in some order, compute shortest paths for all vertices seen so far, then add the shortest path to next vertex x.

Problem: Shortest path to a vertex already processed might go through x.
Solution: Process vertices in order of distance from s.
### Dijkstra's Algorithm Example

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Process A</td>
<td>0</td>
<td>10</td>
<td>3</td>
<td>20</td>
<td>∞</td>
</tr>
<tr>
<td>Process C</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>20</td>
<td>18</td>
</tr>
<tr>
<td>Process B</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>10</td>
<td>18</td>
</tr>
<tr>
<td>Process D</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>10</td>
<td>18</td>
</tr>
<tr>
<td>Process E</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>10</td>
<td>18</td>
</tr>
</tbody>
</table>

---

### Dijkstra's Algorithm: Array (1)

```cpp
void Dijkstra(Graph G, int s) { // Use array
    int D[G.n()];
    for (int i=0; i<G.n(); i++) // Initialize
        D[i] = INFINITY;
    D[s] = 0;
    for (i=0; i<G.n(); i++) { // Process vertices
        int v = minVertex(G, D);
        if (D[v] == INFINITY) return; // Unreachable
        G.setMark(v, VISITED);
        for (Edge w = each neighbor of v)
            if (D[G.v2(w)] > (D[v] + G.weight(w)))
                D[G.v2(w)] = D[v] + G.weight(w);
    }
}
```

---

### Dijkstra's Algorithm: Array (2)

```cpp
// Get mincost vertex
int minVertex(Graph G, int*D) {
    int v; // Initialize v to an unvisited vertex;
    for (int i=0; i<G.n(); i++)
        if (G.getMark(i) == UNVISITED)
            { v = i; break; }
    for (i++; i<G.n(); i++) // Find smallest D val
        if ((G.getMark(i) == UNVISITED) && (D[i]<D[v]))
            v = i;
    return v;
}
```

---

### Dijkstra's Algorithm: Priority Queue (1)

```cpp
class Elem { public: int vertex, dist; };
int key(Elem x) { return x.dist; }
void Dijkstra(Graph G, int s) { // priority queue
    int v; // Initialize v to an unvisited vertex;
    for (int i=0; i<G.n(); i++)
        if (G.getMark(i) == UNVISITED)
            { v = i; break; }
    for (i++; i<G.n(); i++) // Find smallest D val
        if ((G.getMark(i)==UNVISITED) && (D[i]<D[v]))
            v = i;
    Approach 1: Scan the table on each pass for closest vertex.
    Total cost: Θ(|V|^2 + |E|) = Θ(|V|^2).
```
Dijkstra's Algorithm: Priority Queue (2)

```java
for (Edge w = each neighbor of v)
    if (D[G.v2(w)] > (D[v] + G.weight(w))) {
        D[G.v2(w)] = D[v] + G.weight(w);
        temp.dist = D[G.v2(w)];
        temp.vertex = G.v2(w);
        H.insert(temp); // Insert new distance
    }
}
```

- Approach 2: Store unprocessed vertices using a min-heap to implement a priority queue ordered by D value. Must update priority queue for each edge.
- Total cost: \( \Theta((|V| + |E|) \log |V|) \).

All Pairs Shortest Paths

- For every vertex \( u, v \in V \), calculate \( d(u, v) \).
- Could run Dijkstra's Algorithm \(|V|\) times.
- Better is Floyd's Algorithm.
- Define a k-path from \( u \) to \( v \) to be any path whose intermediate vertices all have indices less than \( k \).

Floyd's Algorithm

```java
void Floyd(Graph G) {
    int D[G.n()][G.n()]; // Store distances
    for (int i=0; i<G.n(); i++) // Initialize D
        for (int j=0; j<G.n(); j++)
            D[i][j] = G.weight(i, j);
    for (int k=0; k<G.n(); k++) // Compute k paths
        for (int i=0; i<G.n(); i++)
            for (int j=0; j<G.n(); j++)
                if (D[i][j] > (D[i][k] + D[k][j]))
                    D[i][j] = D[i][k] + D[k][j];
}
```

Minimum Cost Spanning Trees

Minimum Cost Spanning Tree (MST) Problem:
- Input: An undirected, connected graph \( G \).
- Output: The subgraph of \( G \) that
  - has minimum total cost as measured by summing the values for all of the edges in the subset, and
  - keeps the vertices connected.
Key Theorem for MST

Let \( V_1, V_2 \) be an arbitrary, non-trivial partition of \( V \). Let \( (v_1, v_2), v_i \in V_1, v_j \in V_2 \), be the cheapest edge between \( V_1 \) and \( V_2 \). Then \((v_1, v_2)\) is in some MST of \( G \).

**Proof:**
- Let \( T \) be an arbitrary MST of \( G \).
- If \((v_1, v_2)\) is in \( T \), then we are done.
- Otherwise, adding \((v_1, v_2)\) to \( T \) creates a cycle \( C \).
- At least one edge \((u_1, u_2)\) of \( C \) other than \((v_1, v_2)\) must be between \( V_1 \) and \( V_2 \).
- \( c(u_1, u_2) \geq c(v_1, v_2) \).
- Let \( T' = T \cup \{(v_1, v_2)\} - \{(u_1, u_2)\} \).
- Then, \( T' \) is a spanning tree of \( G \) and \( c(T') \leq c(T) \).
- But \( c(T) \) is minimum cost.

Therefore, \( c(T') = c(T) \) and \( T' \) is a MST containing \((v_1, v_2)\).

### Key Theorem Figure

![Key Theorem Figure](image)

### Prim's MST Algorithm (1)

```c
void Prim(Graph G, int s) { // Prim's MST alg
    int D[G.n()]; int V[G.n()]; // Distances
    for (int i=0; i<G.n(); i++) // Initialize
        D[i] = INFINITY;
    D[s] = 0;
    for (i=0; i<G.n(); i++) // Process vertices
        int v = minVertex(G, D);
        G.setMark(v, VISITED);
        if (v != s) AddEdgetoMST(V[v], v);
        if (D[v] == INFINITY) return; //v unreachable
        for (Edge w = each neighbor of v)
            if (D[G.v2(w)] > G.weight(w)) {
                D[G.v2(w)] = G.weight(w); // Update dist
                V[G.v2(w)] = v; // who came from
            }
}
```

### Prim's MST Algorithm (2)

```c
int minVertex(Graph G, int* D) {
    int v; // Initialize v to any unvisited vertex
    for (int i=0; i<G.n(); i++)
        if (G.getMark(i) == UNVISITED) {
            v = i; break;
        }
    for (i=0; i<G.n(); i++) // Find smallest value
        if ((G.getMark(i)==UNVISITED) && (D[i]<D[v]))
            v = i;
    return v;
}
```

This is an example of a **greedy** algorithm.
Alternative Prim's Implementation (1)

Like Dijkstra's algorithm, can implement with priority queue.

```c
void Prim(Graph G, int s) {  
    int v; // The current vertex  
    int D[G.n()]; // Distance array  
    int V[G.n()]; // Who's closest  
    Elem temp;  
    Elem E[G.e()]; // Heap array  
    temp.distance = 0; temp.vertex = s;  
    E[0] = temp; // Initialize heap array  
    heap H(E, 1, G.e()); // Create the heap  
    for (int i=0; i<G.n(); i++) D[i] = INFINITY;  
    D[s] = 0;  
}
```

Kruskal's MST Algorithm (1)

```c
Kruskel(Graph G) { // Kruskal’s MST algorithm  
    Gentree A(G.n()); // Equivalence class array  
    Elem E[G.e()]; // Array of edges for min-heap  
    int edgecnt = 0;  
    for (int i=0; i<G.n(); i++) { // Put edges into E  
        if (G.isEdge(w)) AddEdgetoMST(V[w1], V[w2]);  
        if (D[v] == INFINITY) return; // Unreachable  
        for (Edge w = each neighbor of v)  
            if (D[G.v2(w)] > G.weight(w)) { // Update D  
                D[G.v2(w)] = G.weight(w);  
                V[G.v2(w)] = v; // Who came from  
                temp.distance = D[G.v2(w)];  
                temp.vertex = G.v2(w);  
                H.insert(temp); // Insert dist in heap  
            }  
    }
}
```

How do we compute function \( MSTof(v) \)?
Solution: UNION-FIND algorithm (Section 4.3).
**Kruskal’s Algorithm Example**

Total cost: \( \Theta(|V| + |E| \log |E|) \).

---

**Matching**

- Suppose there are \( n \) workers that we want to work in teams of two. Only certain pairs of workers are willing to work together.
- **Problem**: Form as many compatible non-overlapping teams as possible.
- **Model**: Use \( G \), an undirected graph.
  - Join vertices if the workers will work together.
  - A matching is a set of edges in \( G \) with no vertex in more than one edge (the edges are independent).
  - A maximal matching has free pairs of vertices that can extend the matching.
  - A maximum matching has the greatest possible number of edges.
  - A perfect matching includes every vertex.

---

**Very Dense Graphs (1)**

**Theorem**: Let \( G = (V, E) \) be an undirected graph with \( |V| = 2n \) and every vertex having degree \( \geq n \). Then \( G \) contains a perfect matching.

**Proof**: Suppose that \( G \) does not contain a perfect matching.
- Let \( M \subset E \) be a max matching. \( |M| < n \).
- There must be two unmatched vertices \( v_1, v_2 \) that are not adjacent.
- Every vertex adjacent to \( v_1 \) or \( v_2 \) is matched.
- Let \( M' \subset M \) be the set of edges involved in matching the neighbors of \( v_1 \) and \( v_2 \).
- There are \( 2n \) edges from \( v_1 \) and \( v_2 \) to vertices covered by \( M' \), but \( |M'| < n \).

---

**Very Dense Graphs (2)**

**Proof**: (continued)
- Thus, some edge of \( M' \) is adjacent to 3 edges from \( v_1 \) and \( v_2 \).
- Let \( (u_1, u_2) \) be such an edge.
- Replacing \( (u_1, u_2) \) with \( (v_1, u_2) \) and \( (v_2, u_1) \) results in a larger matching.
- **Theorem proven by contradiction.**

---

Cost is dominated by the edge sort.
Alternative: Use a min heap, quit when only one set left.
“Kth-smallest” implementation.
**The Alternating Path Theorem (1)**

**Theorem:** A matching is maximum iff it has no alternating paths.

**Proof:**
- Clearly, if a matching has alternating paths, then it is not maximum.
- Suppose \( M \) is a non-maximum matching.
- Let \( M' \) be any maximum matching. Then, \( |M'| > |M| \).
- Let \( M \oplus M' \) be the symmetric difference of \( M \) and \( M' \):
  \[
  M \oplus M' = M \cup M' - (M \cap M').
  \]
- \( G' = (V, M \oplus M') \) is a subgraph of \( G \) having maximum degree \( \leq 2 \).

**The Alternating Path Theorem (2)**

**Proof:** (continued)
- Therefore, the connected components of \( G' \) are either even-length cycles or a path with alternating edges.
- Since \( |M'| > |M| \), there must be a component of \( G' \) that is an alternating path having more \( M' \) edges than \( M \) edges.

The first point is the obvious part of the if. If there is an alternating path, simply switch the match and unmatched edges to augment the match.

Symmetric difference: Those in either, but not both.

The max degree is \( \leq 2 \) because a vertex matches one different vertex in \( M \) and \( M' \).
Bipartite Matching

- A bipartite graph $G = (U, V, E)$ consists of two disjoint sets of vertices $U$ and $V$ together with edges $E$ such that every edge has an endpoint in $U$ and an endpoint in $V$.
- Bipartite matching naturally models a number of assignment problems, such as assignment of workers to jobs.
- Alternating paths will work to find a maximum bipartite matching. An alternating path always has one end in $U$ and the other in $V$.
- If we direct unmatched edges from $U$ to $V$ and matched edges from $V$ to $U$, then a directed path from an unmatched vertex in $U$ to an unmatched vertex in $V$ is an alternating path.

Bipartite Matching Example

2, 8, 5, 10 is an alternating path.

1, 6, 3, 7, 4 and 2, 8, 5, 10 are disjoint alternating paths that we can augment independently.

Algorithm for Maximum Bipartite Matching

Construct BFS subgraph from the set of unmatched vertices in $U$ until a level with unmatched vertices in $V$ is found.

- Greedily select a maximal set of disjoint alternating paths.
- Augment along each path independently.
- Repeat until no alternating paths remain.

Time complexity $O((|V| + |E|)\sqrt{|V|})$.

Naive algorithm: Find a maximal matching (greedy algorithm).

For each vertex:
- Do a DFS or other search until an alternating path is found.
- Use the alternating path to improve the match.

$|V|(|V| + |E|)$

Order doesn’t matter. Find a path, remove its vertices, then repeat. Augment along the paths independently since they are disjoint.