CSEP 521 - Applied Algorithms

Graph Algorithms
Broadcasting in a Network
DFS, BFS,
Shortest Path Problems

Graph Algorithms - reading

DFS, BFS -
CLRS: chapter 22 (1st and 2nd editions.)
Skiena: section 4.4
Shortest Path Problems -
CLRS: chapter 25 (1st ed.) or 24 (2nd ed.)
Skiena: section 4.8

Definition

- A graph $G$ is given by the two sets $V$ and $E$.
- $V$ is a set of points (vertices)
- $E$ is a set of lines (edges) connecting pairs of points.

Examples:
- airline flight map.
- communication networks.
- precedence constraints on the scheduling of jobs.
- flow networks.

V=$\{A,B,C,D,E\}$
E=$\{(B,E),(E,D),(D,C),(B,D),(A,E)\}$

Figure 12.1 Three graphs. (a) and (c) are undirected graphs, (b) is a directed graph. The placement of the vertices on the paper is immaterial when we draw graphs; for example, (a) and (c) are in fact depictions of the same graph.

Figure 12.2 A map and its associated undirected graph. Each region is represented by a vertex, and an edge joins each pair of vertices that correspond to bordering regions.
A More Detailed Definition

- An **undirected graph** is a pair \( (V,E) \), where \( V \) is a finite set, and \( E \) is a set of unordered pairs \( (u,v) \), where \( u \) and \( v \) are in \( V \).
- **Terminology:** If \( (u,v) \) is an edge (i.e., in \( E \)): \( u \) and \( v \) are **adjacent**; \( v \) is a **neighbor** of \( u \).
- A **directed graph** is a pair \( (V,E) \), where \( V \) is a finite set, and \( E \) is a set of ordered pairs \( (u,v) \) (both in \( V \)).

More Definitions

- Let \( n = |V|, m = |E| \)
- The **size** of the graph is \( n+m \)
  - Any algorithm that needs to inspect each vertex and edge has running time \( \Omega(n+m) \)
- A **path** in \( G \) is a sequence \( (v_0,v_1,\ldots,v_k) \) of vertices such that \( (v_i, v_{i+1}) \in E \), for all \( 0 \leq i < k \). Its **length** is \( k \) and it is a path from \( v_0 \) to \( v_k \).
- A **cycle** is a path such that \( v_0 = v_k \).
- An undirected graph is **connected** if and only if there is a path between every pair of vertices.
- In any connected graph \( m = \Omega(n) \).

Trees

- An undirected graph is a tree if it is connected and contains no cycles.
- A directed graph is a directed tree if it has a root and its underlying undirected graph is a tree.
- \( r \in V \) is a root if every vertex \( v \in V \) is reachable from \( r \); i.e., there is a directed path which starts in \( r \) and ends in \( v \).

Alternative Definitions of Undirected Trees

- \( G \) is cycles-free, but if any new edge is added to \( G \), a cycle is formed.
- For every pair of vertices \( u,v \), there is a unique, simple path from \( u \) to \( v \).
- \( G \) is connected, but if any edge is deleted from \( G \), the connectivity of \( G \) is interrupted.
- \( G \) is connected and has \( n-1 \) edges.
G is a tree ⇒
G is cycle-free and has n - 1 edges.
⇒ We show, by induction on n, that if G is a tree (cycle-
free and connected), then its number of edges is n-1.
Base: n=1  
Step: Assume that it is true for all n < m, and let G be a
tree with m vertices. Delete from G any edge e. By
definition (3), G is not connected any more, and is
broken into two connected components each of which
is cycle-free and therefore is a tree. By the inductive
hypothesis, each component has one edge less than
the number of vertices. Thus, both have m-2 edges.
Add back e, to get m-1.

More Definitions
• A subgraph of a graph G=(V,E) is a graph
G'=(V',E') such that V'⊆ V and E' ⊆ E ∩ (V'×V).
• A connected component of an undirected
graph G is a maximal connected subgraph of
G.

Enough with the definitions. Let’s do
something.

Applied Algorithm Scenario
Real world problem
Abstractly model the problem
Find abstract algorithm
Adapt to original problem

Broadcasting in a Network
• Network of Routers
  - Organize the routers to efficiently
    broadcast messages to each other.

Incoming message
  • Duplicate and send
to some neighbors.
  • Eventually all routers
    get the message

Goal: Minimize the number of messages.
Spanning Tree in a Graph

![Graph](image1)

- Vertex = router
- Edge = link between routers
- Spanning tree
  - Connects all the vertices
  - No cycles

Spanning Tree Problem

- Input: An undirected graph $G = (V,E)$. $G$ is connected.
- Output: $T$ contained in $E$ such that
  - $(V,T)$ is a connected graph
  - $(V,T)$ has no cycles

Depth First Search Algorithm

- Recursive marking algorithm
- Initially every vertex is unmarked

DFS(i: vertex)
mark i;
for each j adjacent to i do
  if j is unmarked then DFS(j)
end DFS

Example of Depth First Search

![Graph](image2)
Example Step 2

Example Step 3

Example Step 4

Example Step 5
Spanning Tree Algorithm

```
Main
T := empty set;
ST(1);
end{Main}

The addition to DFS

ST(i: vertex)
  mark i;
  for each j adjacent to i do
    if j is unmarked then
      Add {i,j} to T;
      ST(j);
  end{ST}
```

Applied Algorithm Scenario

1. Real world problem
2. Abstractly model the problem
3. Find abstract algorithm
4. Adapt to original problem
5. Evaluate

Note that the edges traversed in the depth first search form a spanning tree.
### Evaluation Step Expanded

- **Algorithm Correct?**
  - yes
  - no
    - New algorithm
    - New model
    - New problem

- **Choose Data Structure**

- **Performance?**
  - unsatisfactory
  - satisfactory
    - New data structure
    - New algorithm
    - New model

- **Implement**

### Correctness of ST Algorithm

- There are no cycles in \( T \)
  - This is an invariant of the algorithm.
  - Each edge added to \( T \) goes from a vertex in \( T \) to a vertex not in \( T \).
- If \( G \) is connected then eventually every vertex is marked.

### Data Structure Step

- **Algorithm Correct?**
  - yes
  - no
    - New algorithm
    - New model
    - New problem

- **Choose Data Structure**

- **Performance?**
  - unsatisfactory
  - satisfactory
    - New data structure
    - New algorithm
    - New model

- **Implement**
Edge List and Adjacency Lists

- List of edges
  \[
  1 5 1 2 2 3 5 7 5 5 \\
  2 1 6 7 3 4 6 4 7 4
  \]

- Adjacency lists
  
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
</table>
  1 | | 2 | 5 | 6 | | | |
  2 | | 3 | 1 | 7 | 1 | | |
  3 | | 2 | 4 | | | | |
  4 | | 3 | 7 | 5 | | | |
  5 | | 6 | 1 | 7 | 4 | | |
  6 | | 1 | 5 | | | | |
  7 | | 4 | 5 | 2 | | | |

Adjacency Matrix

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & 0 & 1 & 0 & 1 & 1 & 0 \\
2 & 1 & 0 & 1 & 0 & 0 & 0 \\
3 & 0 & 1 & 0 & 1 & 0 & 0 \\
4 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
5 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\
6 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
7 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\
\end{array}
\]

Data Structure Choice

- Edge list
  - Simple but does not support depth first search
- Adjacency lists
  - Good for sparse graphs
  - Supports depth first search
- Adjacency matrix
  - Good for dense graphs
  - Supports depth first search

Spanning Tree with Adjacency Lists

```
Main
G is array of adjacency lists;
M[i] := 0 for all i;
T is empty;
Spanning_Tree(1);
end{Main}

ST(i: vertex)
M[i] := 1;
v := G[i];
while (v ≠ null)
  j := v.vertex;
  if (M[j] = 0) then
    add {i,j} to T;
    ST(j);
  end
  v := v.next;
end{ST}
```

Node of linked list:
vertex next
Performance Step

- Algorithm Correct?
  - yes
  - no
  - Choose Data Structure
  - Performance?
    - unsatisfactory
    - satisfactory
    - Implement

Performance of ST Algorithm

- n vertices and m edges
- Connected graph \( m \geq n-1 \)
- Storage complexity \( O(m) \)
- Time complexity \( O(m) \) - for each edge we perform \( O(1) \) operations in each of the two endpoints.

Other Uses of Depth First Search

- Popularized by Hopcroft and Tarjan 1973
- Connected components
- Strongly connected components in directed graphs
- Topological sorting of an acyclic directed graphs
- Maze solving

ST using Breadth First Search 1

- Uses a queue to order search

Queue = 1
Breadth First Search 2

Queue = 2, 6, 5

Breadth First Search 3

Queue = 6, 5, 7, 3

Breadth First Search 4

Queue = 5, 7, 3

Breadth First Search 5

Queue = 7, 3, 4
Breadth First Search 6

Queue = 3,4

Breadth First Search 7

Queue = 4

Breadth First Search 8

Queue =

Spanning Tree using Breadth First Search (BFS)

Initialize T to be empty;
Initialize Q to be empty;
Enqueue(1,Q) and mark 1;
while (Q is not empty) do
  i := Dequeue(Q);
  for each j adjacent to i do
    if j is not marked then
      add {i,j} to T;
      mark j;
      Enqueue(j,Q);
Depth First vs Breadth First

- Depth First
  - Stack or recursion
  - Many applications
- Breadth First
  - Queue (recursion no help)
  - Can be used to find shortest paths from the start vertex
- Both are \( O(|E|) \)

Shortest-path Algorithms

- Scenario: One router creates messages (source). Each message needs to reach other routers (one or more) along the shortest possible path.
- Abstraction: given a vertex \( s \), find the shortest path from \( s \) to any other vertex of \( G \).
- Other shortest path problems:
  - Different edges have different lengths (delay, cost, etc.)
  - All-pair shortest path problem: no specific source.

Using BFS for Shortest-path

- Given a vertex \( s \), find the shortest path from \( s \) to any other vertex of \( G \).

A 'centralized' version of BFS:
1. Label vertex \( s \) with 0.
2. \( i \leftarrow 0 \)
3. Find all unlabeled vertices adjacent to at least one vertex labeled \( i \). If none are found, stop.
4. Label all the vertices found in (3) with \( i + 1 \).
5. \( i \leftarrow i + 1 \) and go to (3).

BFS for Shortest Path (\( i=0 \))

Vertices whose distance from \( s \) is 0 are labeled.
BFS for Shortest Path (i=1)

Vertices whose distance from s is 1 are labeled.

BFS for Shortest Path (i=2)

Vertices whose distance from s is 2 are labeled.

BFS for Shortest Path (i=3)

Vertices whose distance from s is 3 are labeled.

In the next iteration we find out that the whole graph is labeled and we stop.

The BFS Tree

Theorem: Each vertex is labeled by its length from s.
Proof: By induction on the label.

For any $v \neq s$, let $p(v)$ be the vertex that 'discovered' $v$ in BFS.

Then $T = \{(p(v), v)\}$ is a directed spanning tree rooted in $s$, and for each vertex $v$, the path from $s$ to $v$ in $T$ is a shortest path from $s$ to $v$ in $G$.

Note: the 'centralized' version is for simplification only. When implemented, we need the queue as before.
Single-Source Shortest Paths (Dijkstra's algorithm)

- Using BFS, we solve the problem of finding shortest path from s to any vertex v.
- What if edges have associated costs or distances? (BFS assumes edge costs are all 1.)
- Assume each edge (u,v) has non-negative weight c(u,v).
- A weight of a path = total weights of all edges on path.
- Problem: Find, for each vertex v, a shortest (minimum weight) path from s to v.

Dijkstra's Algorithm

Assumption: c(u,v) = ∞ if (u,v) not in E.

1. \( \lambda(s) \leftarrow 0 \) and for all \( v \neq s \), \( \lambda(v) \leftarrow \infty \).
2. \( T \leftarrow V \).
3. Let u be a vertex in T for which \( \lambda(u) \) is minimum.
4. For every edge, if \( v \in T \) and \( \lambda(v) \geq \lambda(u) + c(v,u) \) then \( \lambda(v) \leftarrow \lambda(u) + c(v,u) \).
5. \( T \leftarrow T - \{u\} \), if T is not empty go to step 3.

Idea of Dijkstra's Algorithm:

- Maintain:
  - \( \lambda[0..n-1] \) where \( \lambda(v) \) is the cost of best path from s to v found so far, and
  - \( T \), set of vertices v for which \( \lambda(v) \) is not yet known to be optimal.
- Initially:
  - \( \lambda(s) = 0 \); \( \lambda(v) = \infty \) for all v other than s.
  - \( T = V \).
- In each step:
  - remove that v in T with minimum \( \lambda(v) \)
  - update those w in T s.t. \( (v,w) \) in E and \( \lambda(w) > \lambda(v) + c(v,w) \).

Dijkstra's Algorithm - Example
Dijkstra's Algorithm - Example

<table>
<thead>
<tr>
<th>init</th>
<th>u=s</th>
<th>u=a</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0</td>
<td>0*</td>
</tr>
<tr>
<td>a</td>
<td>∞</td>
<td>3</td>
</tr>
<tr>
<td>b</td>
<td>∞</td>
<td>4</td>
</tr>
<tr>
<td>c</td>
<td>∞</td>
<td>6</td>
</tr>
<tr>
<td>d</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>e</td>
<td>∞</td>
<td>9</td>
</tr>
<tr>
<td>f</td>
<td>∞</td>
<td>8</td>
</tr>
</tbody>
</table>

In class exercise: complete the execution.

* non-T vertices.

Why is this Algorithm Correct?

• **Theorem:** At the termination of the algorithm, \( \lambda(v) \) is the length of the shortest path from \( s \) to \( v \) for each vertex \( v \) of \( G \).

• **Proof:** by induction on \( |V-T| \).

• **Inductive hypothesis:** Let \( |V-T|=k \).
  - \( \forall v \) in \( V-T \), \( \lambda(v) \) is the length of the shortest path from \( s \) to \( v \).
  - the vertices in \( V-T \) are the \( k \) closest vertices to \( s \).
  - \( \forall v \) in \( T \), \( \lambda(v) \) is the length of the shortest path from \( s \) to \( v \) that only goes through vertices in \( V-T \).

The \( \lambda \) values of vertices in \( V-T \) are correct and for each such \( v \), the shortest path from \( s \) to \( v \) only goes through vertices in \( V-T \)

• Induction Step: Suppose true for first \( k \) steps. The SP to the \((k+1)^{st}\) closest vertex, say \( w \), can go through only vertices in \( V-T \), otherwise, there would be a closer vertex. Therefore, when selecting the min, we select the \((k+1)^{st}\) closest vertex to \( s \).

Say \( w \) is added.

New \( \lambda \) value for a vertex \( x \) is min of old \( \lambda \) value and \( \lambda(w) + c(w,x) \)
Dijkstra’s Algorithm - Run Time Analysis

Implementation 1:
- Adjacency lists.
- An array for the λ values.

Complexity:
In each iteration:
1. Finding a vertex u in T with minimal λ.
   In the whole execution: \( n + (n-1) + (n-2) + \ldots + 1 = O(n^2) \)
2. Updating the λ-values of u’s neighbors:
   The total sum of the degrees in \( 2m \rightarrow O(m) \)
All together: \( O(m+n^2) = O(n^2) \) (remember, \( m \leq n(n-1) \))

Priority-Queue Implementations

- Priority-Queue can be implemented such that each of these operations takes \( O(\log n) \) time for sets of size \( n \).

Running time of Dijkstra’s algorithm:
We need to consider insertions, delete Mins, lookups, modifying λ values.

Dijkstra’s Algorithm - Run Time Analysis

- Implementation 2: data structure: priority queue
- Stores set \( S \) (in our case, this is T) such that there is a linear order on key values (in our case the key is the λ value).
- Supports operations:
  - Insert(\( x \)) - insert element with key value \( x \) into set.
  - FindMin() - return value of smallest element in set.
  - DeleteMin() - delete smallest element in set.
- and usually:
  - Lookup(\( x \)), Delete(\( x \))

Running Time of Dijkstra’s Algorithm:

- \( n \) insertions: \( O(n \log n) \) time
- \( n \) deleteMins: \( O(n \log n) \) time
- \( m \) lookups: \( O(m \log n) \) time
- \( m \) λ value mods: \( O(m \log n) \) time
- Running time: \( O((n + m) \log n) \)

- The \( O(n^2) \) is better for dense graphs
Single-Source Shortest Paths (Bellman-Ford’s algorithm)

- each edge \((u,v)\) has a weight \(c(u,v)\).
- \(c(u,v)\) might be negative, but there are no negative cycles.

1. \(\lambda(s) \leftarrow 0\) and for every \(v \neq s\), \(\lambda(v) \leftarrow \infty\).
2. As long as there is an edge \(e\) such that \(\lambda(v) > \lambda(u) + c(e)\) replace \(\lambda(v)\) by \(\lambda(u) + c(e)\).

For our purposes \(\infty\) is not greater than \(\infty + k\), even if \(k\) is negative.

Bellman-Ford algorithm

- How do we implement this algorithm?
- Order the edges: \(e_1, e_2, \ldots, e_{|E|}\).
- Perform step 2 by first checking \(e_1\), then \(e_2\), etc., After the first such sweep, go through additional sweeps, until an entire sweep produces no improvement.

- Running Example:

BF algorithm – correctness and run time analysis

- Theorem: if a shortest path from \(s\) to \(v\) consists of \(k\) edges, then by the end of the \(k^{\text{th}}\) sweep \(v\) will have its final label.
- Proof: induction on \(k\) (not here).
- Since \(k\) is bounded by \(|V|\) (remember, no negative cycles), step 2 is performed at most \(|E| \cdot |V|\) times.
- Each comparison in step 2 can takes \(O(1)\) if the graph is kept in an Adjacency Matrix (with the weights) and an array with the \(\lambda(v)\) values.
- The time complexity of BF is \(O(|E| \cdot |V|)\).