Graphs

A formalism for representing relationships between objects

Graph \( G = (V,E) \)
- \( V \) is the set of vertices: \( \{v_1, v_2, \ldots, v_n\} \)
- \( E \) is the set of edges: \( \{e_1, e_2, \ldots, e_m\} \)
  where each edge connects two vertices \( (v_i, v_j) \)

Examples of Graphs

- The web
  - Vertices are webpages
  - Each edge is a link from one page to another
- Call graph of a program
  - Vertices are subroutines
  - Edges are calls and returns
- Social networks
  - Vertices are people
  - Edges connect friends

Graph Definitions

In directed graphs, edges have a direction:

V is adjacent to u if \((u, v) \in E\)

In undirected graphs, they don’t (are two-way):

V is adjacent to u if \((u, v) \in E\)

Weighted Graphs

Each edge has an associated weight or cost.

Paths and Cycles

A path is a list of vertices \( \{v_1, v_2, \ldots, v_n\} \) such that \((v_i, v_{i+1}) \in E\) for all \(0 \leq i < n\).

A cycle is a path that begins and ends at the same node.
**Path Length and Cost**

- **Path length**: the number of edges in the path
- **Path cost**: the sum of the costs of each edge

*Example:*
- Seattle → Salt Lake City → San Francisco → Dallas
  - length(p) = 5
  - cost(p) = 11.5

**More Definitions: Simple Paths and Cycles**

- A **simple path** repeats no vertices (except that the first can also be the last):
  - p = (Seattle, Salt Lake City, San Francisco, Dallas)
  - p = (Seattle, Salt Lake City, Dallas, San Francisco, Seattle)

- A **cycle** is a path that starts and ends at the same node:
  - p = (Seattle, Salt Lake City, Dallas, San Francisco, Seattle)
  - p = (Seattle, Salt Lake City, Seattle, San Francisco, Seattle)

- A **simple cycle** is a cycle that is also a simple path (in undirected graphs, no edge can be repeated)

**Trees as Graphs**

- Every tree is a graph with some restrictions:
  - the tree is **directed**
  - there are **no cycles** (directed or undirected)
  - there is a **directed path from the root to every node**

**Directed Acyclic Graphs (DAGs)**

- **DAGs** are directed graphs with no (directed) cycles.

**Rep 1: Adjacency Matrix**

A $|V| \times |V|$ array in which an element $(u, v)$ is true if and only if there is an edge from $u$ to $v$

**Rep 2: Adjacency List**

A $|V|$-ary list (array) in which each entry stores a list (linked list) of all adjacent vertices
Some Applications: Moving Around Washington

What’s the *shortest way* to get from Seattle to Pullman?

Edge labels:

Some Applications: Moving Around Washington

What’s the *fastest way* to get from Seattle to Pullman?

Edge labels:

Some Applications: Reliability of Communication

If Wenatchee’s phone exchange *goes down*, can Seattle still talk to Pullman?

Some Applications: Bus Routes in Downtown Seattle

If we’re at 3rd and Pine, how can we get to 1st and University using Metro?

How about 4th and Seneca?

Application: Topological Sort

Given a directed graph, \( G = (V,E) \), output all the vertices in \( V \) such that no vertex is output before any other vertex with an edge to it.

Topological Sort: Take One

1. Label each vertex with its *in-degree* (# of inbound edges)
2. While there are vertices remaining:
   a. Choose a vertex \( v \) of *in-degree zero*; output \( v \)
   b. Reduce the in-degree of all vertices adjacent to \( v \)
   c. Remove \( v \) from the list of vertices

Is the output unique?

Runtime:
void Graph::topsort(){
    Vertex v, w;
    labelEachVertexWithItsIn-degree();
    for (int counter=0; counter < NUM_VERTICES; counter++){
        v = findNewVertexOfDegreeZero();
        v.topologicalNum = counter;
        for each w adjacent to v
            w.indegree--;
    }
}

Topological Sort: Take Two
1. Label each vertex with its in-degree
2. Initialize a queue Q to contain all in-degree zero vertices
3. While Q not empty
   a. v = Q.dequeue; output v
   b. Reduce the in-degree of all vertices adjacent to v
   c. If new in-degree of any such vertex u is zero
      Q.enqueue(u)

Note: could use a stack, list, set, box, … instead of a queue

Runtime:

void Graph::topsort(){
    Queue q(NUM_VERTICES); int counter = 0; Vertex v, w;
    labelEachVertexWithItsIn-degree();
    q.makeEmpty();
    for each vertex v if (v.indegree == 0)
        q.enqueue(v);
    while (!q.isEmpty()){
        v = q.dequeue();
        v.topologicalNum = ++counter;
        for each w adjacent to v
            if (--w.indegree == 0)
                q.enqueue(w);
    }
}

Graph Connectivity
Undirected graphs are connected if there is a path between any two vertices
Directed graphs are strongly connected if there is a path from any one vertex to any other
Directed graphs are weakly connected if there is a path between any two vertices, ignoring direction
A complete graph has an edge between every pair of vertices

Graph Traversals
- Breadth-first search (and depth-first search) work for arbitrary (directed or undirected) graphs - not just mazes!
  - Must mark visited vertices so you do not go into an infinite loop!
- Either can be used to determine connectivity:
  - Is there a path between two given vertices?
  - Is the graph (weakly) connected?
- Which one:
  - Uses a queue?
  - Uses a stack?
  - Always finds the shortest path (for unweighted graphs)?

CSE 326: Data Structures
Graph Traversals
James Fogarty
Autumn 2007
Graph Connectivity

- Undirected graphs are **connected** if there is a path between any two vertices.
- Directed graphs are **strongly connected** if there is a path from any one vertex to any other.
- Directed graphs are **weakly connected** if there is a path between any two vertices, ignoring direction.
- A complete graph has an edge between every pair of vertices.

Graph Traversals

- Breadth-first search (and depth-first search) work for arbitrary (directed or undirected) graphs - not just mazes!
  - Must mark visited vertices. Why?
  - So you do not go into an infinite loop! It’s not a tree.
- Either can be used to determine connectivity:
  - Is there a path between two given vertices?
  - Is the graph (weakly/strongly) connected?
- Which one:
  - Uses a queue?
  - Uses a stack?
  - Always finds the shortest path (for unweighted graphs)?

The Shortest Path Problem

- Given a graph $G$, edge costs $c_{ij}$, and vertices $s$ and $t$ in $G$, find the shortest path from $s$ to $t$.

  - For a path $p = v_0, v_1, v_2, \ldots, v_k$
    - **unweighted length** of path $p = k$ (a.k.a. *length*)
    - **weighted length** of path $p = \sum_{i=0}^{k-1} c_{i,i+1}$ (a.k.a. *cost*)
  - Path length equals path cost when?

Single Source Shortest Paths (SSSP)

- Given a graph $G$, edge costs $c_{ij}$, and vertex $s$, find the shortest paths from $s$ to all vertices in $G$.
  - Is this harder or easier than the previous problem?

All Pairs Shortest Paths (APSP)

- Given a graph $G$ and edge costs $c_{ij}$, find the shortest paths between all pairs of vertices in $G$.
  - Is this harder or easier than SSSP?
  - Could we use SSSP as a subroutine to solve this?

Depth-First Graph Search

Open – Stack
Criteria – Pop

- DFS(Start, Goal_test)
  - push(Start, Open);
  - repeat
    - if (empty(Open)) then return fail;
    - Node = pop(Open);
    - if (Goal_test(Node)) then return Node;
    - for each Child of node do
      - if (Child not already visited) then push(Child, Open);
    - Mark Node as visited;
  - end
Breadth-First Graph Search

Open – Queue
Criteria – Dequeue (FIFO)
- BFS(Start, Goal_test)
  - enqueue(Start, Open);
  - repeat
    - if (empty(Open)) then return fail;
    - Node := dequeue(Open);
    - if (Goal_test(Node)) then return Node;
    - for each Child of node do
      - if (Child not already visited) then enqueue(Child, Open);
    - Mark Node as visited;
  - end

Comparison: DFS versus BFS
- Depth-first search
  - Does not always find shortest paths
  - Must be careful to mark visited vertices, or you could go into an infinite loop if there is a cycle
- Breadth-first search
  - Always finds shortest paths
  - Optimal solutions
  - Marking visited nodes can improve efficiency, but even without doing so search is guaranteed to terminate
  - Is BFS always preferable?

DFS Space Requirements
- Assume:
  - Longest path in graph is length d
  - Highest number of out-edges is k
- DFS stack grows at most to size dk
  - For k=10, d=15, size is 150

BFS Space Requirements
- Assume
  - Distance from start to a goal is d
  - Highest number of out edges is k BFS
- Queue could grow to size kd
  - For k=10, d=15, size is 1,000,000,000,000,000

Conclusion
- For large graphs, DFS is hugely more memory efficient, if we can limit the maximum path length to some fixed d.
  - If we knew the distance from the start to the goal in advance, we can just not add any children to stack after level d
  - But what if we don’t know d in advance?

Iterative-Deepening DFS (I)
- Bounded_DFS(Start, Goal_test, Limit)
  - Start.dist = 0;
  - push(Start, Open);
  - repeat
    - if (empty(Open)) then return fail;
    - Node := pop(Open);
    - if (Goal_test(Node)) then return Node;
    - if (Node.dist >= Limit) then return fail;
    - for each Child of node do
      - if (Child not already i-visited) then
        - Child.dist := Node.dist + 1;
        - push(Child, Open);
        - Mark Node as i-visited;
      - end
    - end
  - end
Iterative-Deepening DFS (II)

- IDFS_Search(Start, Goal_test)
  - i := 1;
  - repeat
    - answer := Bounded_DFS(Start, Goal_test, i);
    - if (answer != fail) then return answer;
    - i := i+1;
  - end

Analysis of IDFS

- Work performed with limit < actual distance to G is wasted – but the wasted work is usually small compared to amount of work done during the last iteration
  \[ \sum_{i=1}^{d} k^i = O(k^d) \]
  Ignore low order terms!
  - Same time complexity as BFS
  - Same space complexity as (bounded) DFS

Saving the Path

- Our pseudocode returns the goal node found, but not the path to it
- How can we remember the path?
  - Add a field to each node, that points to the previous node along the path
  - Follow pointers from goal back to start to recover path

Example

- Seattle
- San Francisco
- Dallas
- Salt Lake City

Example (Unweighted Graph)

- Seattle
- San Francisco
- Dallas
- Salt Lake City
Graph Search, Saving Path

- Search( Start, Goal_test, Criteria)
  - insert(Start, Open);
  - repeat
    - if (empty(Open)) then return fail;
    - select Node from Open using Criteria;
    - if (Goal_test(Node)) then return Node;
    - for each Child of node do
      - if (Child not already visited) then
        - Child.previous := Node;
        - Insert( Child, Open );
      - Mark Node as visited;
    - end
Dijkstra’s Algorithm for Single Source Shortest Path

- Similar to breadth-first search, but uses a heap instead of a queue:
  - Always select (expand) the vertex that has a lowest-cost path to the start vertex
- Correctly handles the case where the lowest-cost (shortest) path to a vertex is not the one with fewest edges

CSE 326: Data Structures

Dijkstra’s Algorithm

Dijkstra, Edsger Wybe

Legendary figure in computer science; was a professor at University of Texas.
Supported teaching introductory computer courses without computers (pencil and paper programming)
Supposedly wouldn’t (until very late in life) read his e-mail; so, his staff had to print out messages and put them in his box.

1972 Turning Award Winner, Programming Languages, semaphores, and …

Dijkstra’s Algorithm: Idea

Adapt BFS to handle weighted graphs
Two kinds of vertices:
- Finished or known vertices
- Unknown vertices
  - Shortest distance has been computed
  - Have tentative distance

Dijkstra’s Algorithm: Pseudocode

At each step:
1) Pick closest unknown vertex
2) Add it to known vertices
3) Update distances

Initialize the cost of each node to $\infty$
Initialize the cost of the source to 0

While there are unknown nodes left in the graph
  Select an unknown node $b$ with the lowest cost
  Mark $b$ as known
  For each node $a$ adjacent to $b$
    $a$’s cost = min($a$’s old cost, $b$’s cost + cost of (b, a))
    $a$’s prev path node = $b$
Important Features

• Once a vertex is made known, the cost of the shortest path to that node is known
• While a vertex is still not known, another shorter path to it might still be found
• The shortest path itself can be found by following the backward pointers stored in node.path

Dijkstra's Algorithm in action

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Visited?</th>
<th>Cost</th>
<th>Found by</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>&lt;=2 A</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>&lt;=1 A</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>&lt;=4 A</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>&lt;=7 C</td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>F</td>
<td>&lt;=2 B</td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>G</td>
<td>&lt;=12 G</td>
<td></td>
<td>G</td>
</tr>
<tr>
<td>H</td>
<td>&lt;=12 H</td>
<td></td>
<td>H</td>
</tr>
</tbody>
</table>
Dijkstra's Algorithm in action

Vertex Visited? Cost Found by
A Y 0 A
B Y 2 A
C Y 1 A
D Y <=4 A
E <=11 C
F Y 4 B
G ?? ??
H <=7 F

Answer

Vertex Visited? Cost Found by
v0 Y 0
V0 Y 2 V3
V1 Y 2 V0
V2 Y 1 V0
V3 Y 2 V3
V4 Y 4 V2
V5 Y 6 V3
V6 Y 8 V3

Your turn

Vertex Visited? Cost Found by
v0 ?? ??
v1 ?? ??
v2 ?? ??
v3 ?? ??
v4 ?? ??
v5 ?? ??
v6 ?? ??
Dijkstra’s Alg: Implementation

Initialize the cost of each node to $\infty$.
Initialize the cost of the source to 0.
While there are unknown nodes left in the graph
   Select the unknown node $b$ with the lowest cost
   Mark $b$ as known
   For each node $a$ adjacent to $b$
      $a$’s cost = min($a$’s old cost, $b$’s cost + cost of ($b$, $a$))
      $a$’s prev path node = $b$ (if we updated $a$’s cost)

What data structures should we use?
Running time?

Dijkstra’s Algorithm: Summary

- Classic algorithm for solving SSSP in weighted graphs without negative weights
- A greedy algorithm (irrevocably makes decisions without considering future consequences)
- Intuition for correctness:
  - shortest path from source vertex to itself is 0
  - cost of going to adjacent nodes is at most edge weights
  - cheapest of these must be shortest path to that node
  - update paths for new node and continue picking cheapest path

Correctness: The Cloud Proof

How does Dijkstra’s decide which vertex to add to the Known set next?

- If path to $V$ is shortest, path to $W$ must be at least as long (or else we would have picked $W$ as the next vertex)
- So the path through $W$ to $V$ cannot be any shorter!

The Known Cloud

The trouble with negative weight cycles

What’s the shortest path from $A$ to $E$?

Problem?
Dijkstra’s vs BFS

At each step:
1) Pick closest unknown vertex
2) Add it to finished vertices
3) Update distances

Dijkstra’s Algorithm

Breadth-first Search

Some Similarities:

1) Single-Source Shortest Path
   • Given a graph \( G = (V, E) \) and a single distinguished vertex \( s \), find the shortest weighted path from \( s \) to every other vertex in \( G \).

2) All-Pairs Shortest Path:
   • Find the shortest paths between all pairs of vertices in the graph.
   • How?

Analysis

• Total running time for Dijkstra’s:
  \( O(|V| \log |V| + |E| \log |V|) \) (heaps)

What if we want to find the shortest path from each point to ALL other points?

Dynamic Programming

Algorithmic technique that systematically records the answers to sub-problems in a table and re-uses those recorded results (rather than re-computing them).

Simple Example: Calculating the Nth Fibonacci number.

\[ \text{Fib}(N) = \text{Fib}(N-1) + \text{Fib}(N-2) \]

Floyd-Warshall

for (int \( k = 1; k <= V; k++ \))
    for (int \( i = 1; i <= V; i++ \))
        for (int \( j = 1; j <= V; j++ \))
            if ( ( \( M[i][k] + M[k][j] \) ) < \( M[i][j] \) )
                \( M[i][j] = M[i][k] + M[k][j] \)

Invariant: After the kth iteration, the matrix includes the shortest paths for all pairs of vertices \((i, j)\) containing only vertices \(1..k\) as intermediate vertices

Initial state of the matrix:

\[
\begin{array}{cccc}
  a & b & c & d & e \\
  a & 0 & 2 & -4 & - \\
  b & - & 0 & -2 & 1 & 3 \\
  c & - & - & 0 & -1 \\
  d & - & - & - & 0 & 4 \\
  e & - & - & - & - & 0 \\
\end{array}
\]

\[ M[i][j] = \min(M[i][j], M[i][k] + M[k][j]) \]
Floyd-Warshall - for All-pairs shortest path

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>-4</td>
</tr>
<tr>
<td>b</td>
<td>-</td>
<td>0</td>
<td>-2</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>d</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>e</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

Final Matrix

CSE 326: Data Structures
Spanning Trees

A Hidden Tree

Spanning Tree in a Graph

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

Undirected Graph

- \( G = (V,E) \)
  - \( V \) is a set of vertices (or nodes)
  - \( E \) is a set of unordered pairs of vertices

\[
V = \{1,2,3,4,5,6,7\}
\]
\[
E = \{(1,2),(1,6),(1,5),(2,7),(2,3),(3,4),(4,7),(4,5),(5,6)\}
\]

2 and 3 are adjacent
2 is incident to edge \( \{2,3\} \)

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
Spanning Tree Algorithm

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}
end{ST}

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

Example of Depth First Search

ST(1)

Example Step 16

ST(1)
{1,2} {2,7} {7,5} {5,4} {4,3} {5,6}

Minimum Spanning Trees

Given an undirected graph \( G=(V,E) \), find a graph \( G'=(V, E') \) such that:
- \( E' \) is a subset of \( E \)
- \( |E'| = |V| - 1 \)
- \( G' \) is connected
- \( \sum_{(u,v) \in E'} c_{uv} \) is minimal

Applications: wiring a house, power grids, Internet connections

Minimum Spanning Tree Problem

- Input: Undirected Graph \( G=(V,E) \) and a cost function \( C \) from \( E \) to the reals. \( C(e) \) is the cost of edge \( e \).
- Output: A spanning tree \( T \) with minimum total cost. That is: \( T \) that minimizes
  \[ C(T) = \sum_{e \in T} C(e) \]

Best Spanning Tree

- Each edge has the probability that it won’t fail
- Find the spanning tree that is least likely to fail
Example of a Spanning Tree

Minimum Spanning Tree Problem

- Input: Undirected Graph $G = (V,E)$ and a cost function $C$ from $E$ to the reals. $C(e)$ is the cost of edge $e$.
- Output: A spanning tree $T$ with minimum total cost. That is: $T$ that minimizes $C(T) = \sum_{e \in T} C(e)$

Reducing Best to Minimum

Let $P(e)$ be the probability that an edge doesn’t fail. Define:

$$C(e) = -\log_{10}(P(e))$$

Minimizing $\sum_{e \in T} C(e)$ is equivalent to maximizing $\prod_{e \in T} P(e)$ because $\prod_{e \in T} P(e) = \prod_{e \in T} 10^{-C(e)} = 10^{-\sum_{e \in T} C(e)}$

Example of Reduction

Find the MST

Find the MST
Two Different Approaches

Prim's Algorithm
Looks familiar!

Kruskals's Algorithm
Completely different!

Prim's Algorithm for MST

A node-based greedy algorithm
Builds MST by greedily adding nodes

1. Select a node to be the “root”
   • mark it as known
   • Update cost of all its neighbors
2. While there are unknown nodes left in the graph
   a. Select an unknown node with the smallest cost
      from some known node
   b. Mark it as known
   c. Add (a, b) to MST
   d. Update cost of all nodes adjacent to b

Prim's Algorithm Analysis

Running time:
Same as Dijkstra's: $O(|E| \log |V|)$

Correctness:
Proof is similar to Dijkstra's

Kruskal's MST Algorithm

Idea: Grow a forest out of edges that do not create a cycle. Pick an edge with the smallest weight.
Kruskal’s Algorithm for MST

An edge-based greedy algorithm
Builds MST by greedily adding edges

1. Initialize with
   • empty MST
   • all vertices marked unconnected
   • all edges unmarked
2. While there are still unmarked edges
   a. Pick the lowest cost edge \((u, v)\) and mark it
   b. If \(u\) and \(v\) are not already connected, add \((u, v)\) to the MST and mark \(u\) and \(v\) as connected to each other

Doesn’t it sound familiar?

Example of Kruskal 1

Data Structures for Kruskal

• Sorted edge list
  \(
  \{7,4\} \{2,1\} \{7,5\} \{5,6\} \{5,4\} \{1,6\} \{2,7\} \{2,3\} \{3,4\} \{1,5\}
  \)

• Disjoint Union / Find
  – Union(a,b) - union the disjoint sets named by a and b
  – Find(a) returns the name of the set containing a

Example of DU/F 1

Example of DU/F 2

Example of DU/F 3
Kraskal’s Algorithm with DU/F

Sort the edges by increasing cost;
Initialize A to be empty;
for each edge (i,j) chosen in increasing order do
  u := Find(i);
  v := Find(j);
  if not(u = v) then
    add (i,j) to A;
    Union(u,v);

Kruskal code

```c
void Graph::kruskal()
{
  int edgesAccepted = 0;
  DisjSet s(NUM_VERTICES);
  while (edgesAccepted < NUM_VERTICES - 1)
  {
    e = smallest weight edge not deleted yet;
    // edge e = (u, v)
    uset = s.find(u);
    vset = s.find(v);
    if (uset != vset){
      edgesAccepted++;
      s.unionSets(uset, vset);
    }
  }
}
```

Find MST using Kruskal’s

• Now find the MST using Prim’s method.
• Under what conditions will these methods give the same result?

Total Cost: