Representations of Graphs

Adjacency List
- Uses space $O(n)$
- Can iterate over all edges in time $O(n)$
- Can answer "Is there an edge from $u$ to $v$?" in $O(1)$ time
- Better for sparse graphs (fewer edges)

Adjacency Matrix
- Uses space $O(n^2)$
- Can iterate over all edges in time $O(n^2)$
- Can answer "Is there an edge from $u$ to $v$?" in $O(1)$ time
- Better for dense graphs (lots of edges)

Shortest Paths in Graphs

• Finding the shortest (min-cost) path in a graph is a problem that occurs often
  – Find the shortest route between Ithaca and West Lafayette, IN
  – Result depends on our notion of cost
    • Least mileage
    • Least time
    • Cheapest
    • Least boring
  – All of these "costs" can be represented as edge weights
• How do we find a shortest path?

Dijkstra’s Algorithm

dijkstra(s) {
    $D[s] = 0$; $D[t] = c(s,t)$, $t \neq s$;
    mark $s$;
    while (some vertices are unmarked) {
        $v =$ unmarked node with smallest $D$;
        mark $v$;
        for (each $w$ adjacent to $v$) {
            $D[w] = \min(D[w], D[v] + c(v,w))$;
        }
    }
}
Proof of Correctness

The following are invariants of the loop:
  • For $u \in X$, $D(u) = d(s,u)$
  • For $u \in X$ and $v \notin X$, $d(s,u) = d(s,v)$
  • For all $u$, $D(u)$ is the length of the shortest path from $s$ to $u$ such that all nodes on the path (except possibly $u$) are in $X$

Implementation:
  • Use a priority queue for the nodes not yet taken – priority is $D(u)$

Shortest Paths for Unweighted Graphs – A Special Case

• Use breadth-first search
• Time is $O(n + m)$ in adj list representation, $O(n^2)$ in adj matrix representation

Undirected Trees

• An undirected graph is a tree if there is exactly one simple path between any pair of vertices

Facts About Trees

• $|E| = |V| - 1$
• connected
• no cycles

In fact, any two of these properties imply the third, and imply that the graph is a tree

Spanning Trees

A spanning tree of a connected undirected graph $(V,E)$ is a subgraph $(V,E')$ that is a tree
**Spanning Trees**

A *spanning tree* of a connected undirected graph \((V,E)\) is a subgraph \((V,E')\) that is a tree

- Same set of vertices \(V\)
- \(E' \subseteq E\)
- \((V,E')\) is a tree

**Finding a Spanning Tree**

A subtractive method

- Start with the whole graph – it is connected
- If there is a cycle, pick an edge on the cycle, throw it out – the graph is still connected (why?)
- Repeat until no more cycles

A subtractive method

- Start with the whole graph – it is connected
- If there is a cycle, pick an edge on the cycle, throw it out – the graph is still connected (why?)
- Repeat until no more cycles

**Finding a Spanning Tree**

An additive method

- Start with no edges – there are no cycles
- If more than one connected component, insert an edge between them – still no cycles (why?)
- Repeat until only one component

An additive method

- Start with no edges – there are no cycles
- If more than one connected component, insert an edge between them – still no cycles (why?)
- Repeat until only one component
Finding a Spanning Tree

An additive method

• Start with no edges – there are no cycles

• If more than one connected component, insert an edge between them – still no cycles (why?)

• Repeat until only one component

Finding a Spanning Tree

An additive method

• Start with no edges – there are no cycles

• If more than one connected component, insert an edge between them – still no cycles (why?)

• Repeat until only one component

Finding a Spanning Tree

An additive method

• Start with no edges – there are no cycles

• If more than one connected component, insert an edge between them – still no cycles (why?)

• Repeat until only one component

Finding a Spanning Tree

An additive method

• Start with no edges – there are no cycles

• If more than one connected component, insert an edge between them – still no cycles (why?)

• Repeat until only one component

Minimum Spanning Trees

• Suppose edges are weighted, and we want a spanning tree of minimum cost (sum of edge weights)

• Useful in network routing & other applications

3 Greedy Algorithms

A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it
3 Greedy Algorithms
A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it.

A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it.
3 Greedy Algorithms

A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it

B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's algorithm
3 Greedy Algorithms

B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's algorithm

C. Start with any vertex, add min weight edge extending that connected component that does not form a cycle

Prim's algorithm (reminiscent of Dijkstra's algorithm)
3 Greedy Algorithms

C. Start with any vertex, add min weight edge extending that connected component that does not form a cycle

Prim's algorithm (reminiscent of Dijkstra's algorithm)

Prim's Algorithm

```c
prim(s) {
    D[s] = 0; mark s; //start vertex
    while (some vertices are unmarked) {
        v = unmarked vertex with smallest D;
        mark v;
        for (each w adj to v) {
            D[w] = min(D[w], c(v,w));
        }
    }
}
```

Greedy Algorithms

- These are examples of Greedy Algorithms
- The Greedy Strategy is an algorithm design technique
  - Like Divide & Conquer
  - Greedy algorithms are used to solve optimization problems
  - The goal is to find the best solution
- Works when the problem has the greedy-choice property
  - A global optimum can be reached by making locally optimum choices

- Example: the Change Making Problem: Given an amount of money, find the smallest number of coins to make that amount
- Solution: Use a Greedy Algorithm
  - Give as many large coins as you can
- This greedy strategy produces the optimum number of coins for the US coin system
- Different money system ⇒ greedy strategy may fail
  - Example: old UK system
Similar Code Structures

while (some vertices are unmarked) {
  v = best of unmarked vertices;
  mark v;
  for (each w adj to v)
    update w;
}

• bfs
  – best: next in queue
  – update: \[ D[w] = D[v] + 1 \]
• dijkstra
  – best: next in PQ
  – update: \[ D[w] = \min D[w], D[v] + c(v,w) \]
• prim
  – best: next in PQ
  – update: \[ D[w] = \min D[w], c(v,w) \]