Greedy algorithms

*Optimization problems* solved through a sequence of choices that are:

- *feasible*
- *locally optimal*
- *irrevocable*

Not all optimization problems can be approached in this manner!
Applications of the Greedy Strategy

**Optimal solutions:**
- change making
- Minimum Spanning Tree (MST)
- Single-source shortest paths
- simple scheduling problems
- Huffman codes

**Approximations:**
- Traveling Salesman Problem (TSP)
- Knapsack problem
- other combinatorial optimization problems
Minimum Spanning Tree (MST)

- *Spanning tree* of a connected graph $G$: a connected acyclic subgraph of $G$ that includes all of $G$’s vertices.

- *Minimum Spanning Tree* of a weighted, connected graph $G$: a spanning tree of $G$ of minimum total weight.

**Example:**

![Graph diagram with vertices labeled 1, 2, 3, 4 and edges with weights 1, 2, 3, 4, 6]
Prim’s MST algorithm

- Start with tree consisting of one vertex

- “grow” tree one vertex/edge at a time to produce MST
  - Construct a series of expanding subtrees $T_1, T_2, \ldots$

- at each stage construct $T_{i+1}$ from $T_i$: add minimum weight edge connecting a vertex in tree ($T_i$) to one not yet in tree
  - choose from “fringe” edges
  - (this is the “greedy” step!)

- algorithm stops when all vertices are included
Examples:
Notes about Prim’s algorithm

- Need to prove that this construction actually yields MST

- Need priority queue for locating lowest cost fringe edge: use min-heap

- Efficiency: For graph with $n$ vertices and $m$ edges:
  
  $$(n - 1 + m) \log n$$

  - number of stages (min-heap deletions)
  - number of edges considered (min-heap insertions)

  $\Theta(m \log n)$
Another Greedy algorithm for MST: Kruskal

- Start with empty forest of trees
- “grow” MST one edge at a time
  - intermediate stages usually have forest of trees (not connected)
- at each stage add minimum weight edge among those not yet used that does not create a cycle
  - edges are initially sorted by increasing weight
  - at each stage the edge may:
    - expand an existing tree
    - combine two existing trees into a single tree
    - create a new tree
  - need efficient way of detecting/avoiding cycles
- algorithm stops when all vertices are included
Examples:

![Graph with labeled nodes and edges]

- Node 1 is connected to node 2 with weight 2.
- Node 2 is connected to node 3 with weight 6, and to node 4 with weight 3.
- Node 3 is connected to node 1 with weight 1.
- Node 4 is connected to node 2 with weight 3.
- Node a is connected to node b with weight 5, and to node d with weight 4.
- Node b is connected to node a with weight 1, and to node d with weight 6.
- Node d is connected to node a with weight 4, and to node c with weight 3, and to node e with weight 7.
- Node c is connected to node d with weight 3.
- Node e is connected to node d with weight 7.
Notes about Kruskal’s algorithm

- Algorithm looks easier than Prim’s but is
  - harder to implement (checking for cycles!)
  - less efficient $\Theta(m \log m)$

- Cycle checking: a cycle exists iff edge connects vertices in the same component.

- Union-find algorithms – see section 9.2
Shortest paths—Dijkstra’s algorithm

- **Single Source Shortest Paths Problem**: Given a weighted graph G, find the shortest paths from a source vertex s to each of the other vertices.

- Dijkstra’s algorithm: Similar to Prim’s MST algorithm, with the following difference:
  - Start with tree consisting of one vertex
  - “grow” tree one vertex/edge at a time to produce MST
    - Construct a series of expanding subtrees $T_1, T_2, \ldots$
  - Keep track of shortest path from source to each of the vertices in $T_i$
  - at each stage construct $T_{i+1}$ from $T_i$: add minimum weight edge connecting a vertex in tree ($T_i$) to one not yet in tree
    - choose from “fringe” edges
    - (this is the “greedy” step!)
  - algorithm stops when all vertices are included

edge $(v,w)$ with lowest $d(s,v) + d(v,w)$
Example:

```
Example:
```

Diagram:

- Nodes: a, b, c, d, e
- Edges and Weights:
  - a to b: 5
  - a to c: 1
  - a to d: 3
  - b to d: 6
  - b to e: 2
  - c to d: 4
  - d to e: 7
Notes on Dijkstra’s algorithm

- Doesn’t work with negative weights
- Applicable to both undirected and directed graphs
- Efficiency: