### About A6, Prelim 2

Prelim 2: Thursday, 15 November.
Visit exams page of course website and read carefully to find out when you take it (5:30 or 7:30) and what to do if you have a conflict.

Time assignments are different from Prelim 1!

### Undirected trees

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

What’s the root? It doesn’t matter! Any vertex can be root.

### Facts about trees

- \#E = \#V – 1
- connected
- no cycles

Any two of these properties imply the third and thus 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.

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

#### Facts about trees

- Same set of vertices \(V\)
- Maximal set of edges that contains no cycle
- Minimal set of edges that connect all vertices

Three equivalent definitions
Spanning trees: examples

Finding a spanning tree: Subtractive method

- Start with the whole graph – it is connected
- While there is a cycle:
  - Pick an edge of a cycle and throw it out
  - the graph is still connected (why?)

One step of the algorithm

Maximal set of edges that contains no cycle

Finding a spanning tree: Additive method

- Start with no edges
- While the graph is not connected:
  - Choose an edge that connects 2 connected components and add it
  - the graph still has no cycle (why?)

Tree edges will be red.
Dashed lines show original edges.
Left tree consists of 5 connected components, each a node

Aside: Test whether an undirected graph has a cycle

/** Visit all nodes reachable along unvisited paths from u.
 * Pre: u is unvisited. */
public static void dfs(int u) {
    Stack s= (u);
    // inv: All nodes to be visited are reachable along an
    // unvisited path from a node in s.
    while (s is not empty) {
        u= s.pop();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}

Aside: Test whether an undirected graph has a cycle

/** Return true if the nodes reachable from u have a cycle. */
public static boolean hasCycle(int u) {
    Stack s= (u);
    // inv: All nodes to be visited are reachable along an
    // unvisited path from a node in s.
    while (s is not empty) {
        u= s.pop();
        if (u has been visited) return true;
        visit u;
        for each edge (u, v) leaving u {
            s.push(v);
        }
    }
    return false;
}
Minimum spanning trees
• Suppose edges are weighted (>0)
• We want a spanning tree of minimum cost (sum of edge weights)
• Some graphs have exactly one minimum spanning tree. Others have several trees with the same minimum cost, each of which is a minimum spanning tree.
• Useful in network routing & other applications. For example, to stream a video.

Greedy algorithm
A greedy algorithm follows the heuristic of making a locally optimal choice at each stage, with the hope of finding a global optimum.

Example. Make change using the fewest number of coins.
Make change for n cents, $n < 100 \text{ (i.e. } < \$1)$
Greedy: At each step, choose the largest possible coin.

If $n \geq 50$ choose a half dollar and reduce $n$ by 50;
If $n \geq 25$ choose a quarter and reduce $n$ by 25;
As long as $n \geq 10$, choose a dime and reduce $n$ by 10;
If $n \geq 5$, choose a nickel and reduce $n$ by 5;
Choose n pennies.

Greediness works here
You’re standing at point x. Your goal is to climb the highest mountain.
Two possible steps: down the hill or up the hill. The greedy step is to walk up hill. That is a local optimum choice, not a global one. Greediness works in this case.

Greediness doesn’t work here
You’re standing at point x, and your goal is to climb the highest mountain.
Two possible steps: down the hill or up the hill. The greedy step is to walk up hill. But that is a local optimum choice, not a global one. Greediness fails in this case.

Greedy algorithm — doesn’t always work!
A greedy algorithm follows the heuristic of making a locally optimal choice at each stage, with the hope of finding a global optimum. Doesn’t always work.

Example. Make change using the fewest number of coins.
Coins have these values: 7, 5, 1
Greedy: At each step, choose the largest possible coin.
Consider making change for 10.
The greedy choice would choose: 7, 1, 1, 1.
But 5, 5 is only 2 coins.

Finding a minimal spanning tree
Suppose edges have >0 weights
Minimal spanning tree: sum of weights is a minimum

We show two greedy algorithms for finding a minimal spanning tree. They are abstract, at a high level.
They are versions of the basic additive method we have already seen: at each step add an edge that does not create a cycle.
Kruskal: add an edge with minimum weight. Can have a forest of trees.
Prim (JPD): add an edge with minimum weight but so that the added edges (and the nodes at their ends) form one tree.
At each step, add an edge (that does not form a cycle) with minimum weight

One of the 4’s

Red edges need not form tree (until end)

At each step, add an edge (that does not form a cycle) with minimum weight

One of the 4’s

We now investigate Prim’s algorithm

Minim set of edges that connect all vertices

MST using Kruskal’s algorithm

There is a forest of trees, each of which is a single node (a leaf).

We do not look more closely at how best to implement Kruskal’s algorithm—which data structures can be used to get a really efficient algorithm.

Leave that for later courses, or you can look them up online yourself.

We now investigate Prim’s algorithm

Prim’s algorithm

Kruskal

Both algorithms find a minimal spanning tree

\begin{align*}
\text{Prim requires that the constructed red tree} \\
\text{always be connected.} \\
\text{Kruskal doesn’t} \\
\text{But: Both algorithms find a minimal spanning tree}
\end{align*}

\begin{align*}
\text{Here, Prim chooses (0, 1)} \\
\text{Kruskal chooses (3, 4)}
\end{align*}

\begin{align*}
\text{Here, Prim chooses (0, 2)} \\
\text{Kruskal chooses (3, 4)}
\end{align*}
Difference between Prim and Kruskal

Prim requires that the constructed red tree always be connected.
Kruskal doesn’t

But: Both algorithms find a minimal spanning tree

If the edge weights are all different, the Prim and Kruskal algorithms construct the same tree.

---

Prim’s (JPD) spanning tree algorithm

Given: graph \((V, E)\) (sets of vertices and edges)
Output: tree \((V_1, E_1)\), where

\[ V_1 = V \]
\[ E_1 \text{ is a subset of } E \]

\((V_1, E_1)\) is a minimal spanning tree – sum of edge weights is minimal

---

Prim’s (JPD) spanning tree algorithm

\( V_1\) = \{an arbitrary node of \(V\); \(E_1\) = \{\};
//inv: \((V_1, E_1)\) is a tree, \(V_1 \leq V, E_1 \leq E\)

while \((V_1\).size() < \(V\).size()) {
Pick an edge \((u, v)\) with:
\[ \text{min weight, } u \text{ in } V_1, \]
\[ v \text{ not in } V_1; \]
Add v to \(V_1\);
Add edge \((u, v)\) to \(E_1\)
}

Consider having a set \(S\) of edges with the property:
If \((u, v)\) an edge with \(u\) in \(V_1\) and \(v\) not in \(V_1\), then \((u, v)\) is in \(S\)

---

Prim’s (JPD) spanning tree algorithm

\( V_1\) = \{an arbitrary node of \(V\); \(E_1\) = \{\};
//inv: \((V_1, E_1)\) is a tree, \(V_1 \leq V, E_1 \leq E\)

while \((V_1\).size() < \(V\).size()) {
Pick an edge \((u, v)\) with:
\[ \text{min weight, } u \text{ in } V_1, \]
\[ v \text{ not in } V_1; \]
Add v to \(V_1\);
Add edge \((u, v)\) to \(E_1\)
}

Consider having a set \(S\) of edges with the property:
If \((u, v)\) an edge with \(u\) in \(V_1\) and \(v\) not in \(V_1\), then \((u, v)\) is in \(S\)

---

Prim’s (JPD) spanning tree algorithm

\( V_1\) = \{an arbitrary node of \(V\); \(E_1\) = \{\};
//inv: \((V_1, E_1)\) is a tree, \(V_1 \leq V, E_1 \leq E\)

while \((V_1\).size() < \(V\).size()) {
Pick an edge \((u, v)\) with:
\[ \text{min weight, } u \text{ in } V_1, \]
\[ v \text{ not in } V_1; \]
Add v to \(V_1\);
Add edge \((u, v)\) to \(E_1\)
}

Consider having a set \(S\) of edges with the property:
If \((u, v)\) an edge with \(u\) in \(V_1\) and \(v\) not in \(V_1\), then \((u, v)\) is in \(S\)

---

Prim’s (JPD) spanning tree algorithm

\( V_1\) = \{an arbitrary node of \(V\); \(E_1\) = \{\};
//inv: \((V_1, E_1)\) is a tree, \(V_1 \leq V, E_1 \leq E\)

\(S\) = set of edges leaving the single node in \(V_1\);
while \((V_1\).size() < \(V\).size()) {
Pick an edge \((u, v)\) with:
\[ \text{min weight, } u \text{ in } V_1, \]
\[ v \text{ not in } V_1; \]
Remove from \(S\) an edge \((u, v)\) with min weight
\[ \text{---} \]
if \(v\) is not in \(V_1\):
Add v to \(V_1\); add \((u, v)\) to \(E_1\);
Add edges leaving \(v\) to \(S\)
}

Consider having a set \(S\) of edges with the property:
If \((u, v)\) an edge with \(u\) in \(V_1\) and \(v\) not in \(V_1\), then \((u, v)\) is in \(S\)
**Prim’s (JPD) spanning tree algorithm**

V1 = {start node}; E1 = {};
S = set of edges leaving the single node in V1;
// inv: (V1, E1) is a tree, V1 ⊆ V, E1 ⊆ E,
// All edges (u, v) in S have u in V1,
// if edge (u, v) has u in V1 and v not in V1, (u, v) is in S
while (V1.size() < V.size()) {
    Remove from S an edge (u, v) with min weight;
    if (v not in V1) {
        add v to V1; add (u,v) to E1;
        add edges leaving v to S
    }
}

**Question:** How should we implement set S?

Implement S as a heap.
Use adjacency lists for edges

**Greedy algorithms**

Suppose the weights are all 1.
Then Dijkstra’s shortest-path algorithm does a breath-first search!

Dijkstra’s and Prim’s algorithms look similar.
The steps taken are similar, but at each step
• Dijkstra’s chooses an edge whose end node has a minimum path length from start node
• Prim’s chooses an edge with minimum length

BUT: Greediness does not always work!
Traveling salesman problem

Given a list of cities and the distances between each pair, what is the shortest route that visits each city exactly once and returns to the origin city?

- The true TSP is very hard (called NP complete)... for this we want the perfect answer in all cases.
- Most TSP algorithms start with a spanning tree, then "evolve" it into a TSP solution. Wikipedia has a lot of information about packages you can download...

But really, how hard can it be? How many paths can there be that visit all of 50 cities?

12,413,915,592,536,072,670,862,289,047,373,375,038,521,486,354,677,760,000,000,000

Graph Algorithms

- **Search**
  - Depth-first search
  - Breadth-first search
- **Shortest paths**
  - Dijkstra's algorithm
- **Minimum spanning trees**
  - Prim's algorithm
  - Kruskal's algorithm