Spanning Trees, greedy algorithms

Lecture 22
CS2110 – Fall 2017
Your space ship is on earth, and you hear a distress signal from a distance Planet X.

Your job:
1. **Rescue stage**: Fly your ship to Planet X as fast as you can!
2. **Return stage**: Fly back to earth. You have to get there before time runs out. But you see gems on planets, and you want to visit as many planets (and pick up gems) on the way back!

Requires use of graph algorithms we have been discussing.

Open-ended design. There is no known algorithm that on any graph will pick up the optimum number of gems. You get to decide how to traverse the graph, always getting back in time.
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

Three equivalent definitions

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

- Same set of vertices \(V\)
- Minimal set of edges that connect all vertices
Spanning trees: examples

http://mathworld.wolfram.com/SpanningTree.html
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?)

Maximal set of edges that contains no cycle

nondeterministic algorithm

One step of the algorithm
Aside: How can you find a cycle in an undirected graph?
/** Visit all nodes reachable along unvisited paths from u.  
   * Pre: u is unvisited. */

public static void dfs(int u) {
    Stack s = (u);
    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: How to tell whether an undirected graph has a cycle?

We modify iterative dfs to calculate whether the graph has a cycle.
Aside: How can you find a cycle in an undirected graph?

/** Return true if the nodes reachable from u have a cycle. */

public static boolean hasCycle(int u) {
    Stack s = (u);
    while (s is not empty) {
        u = s.pop();
        if (u has been visited) {
            return true;
        } else {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
    return false;
}
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?)

Maximal set of edges that contains no cycle

nondeterministic algorithm

One step of the algorithm
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: How do you find connected components?

We modify iterative dfs to construct the nodes in a component.
Aside: How do you find connected components?

/** Visit all nodes reachable ⋅⋅ from u. Pre: u is unvisited. . */

```java
public static void dfs(int u) {
    Stack s = new Stack();
    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);
        }
    }
}
```

We modify iterative dfs to construct the nodes in a component.
Aside: How do you find connected components?

/** Return the set of nodes in u’s connected component. */
public static Set<int> getComponent(int u) {
    Stack s = (u);
    Set C = ();
    while (s is not empty) {
        u = s.pop();
        if (u has not been visited) {
            visit u; C.add(u);
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
    return C;
}
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

Minimal set of edges that connect all vertices
nondeterministic algorithm
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 (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.
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.
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.
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
**MST using Kruskal’s algorithm**

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

- Edge with weight 2
- Edge with weight 3

One of the 4’s

Red edges need not form tree (until end)
Kruskal

Start with the all the nodes and no edges, so there is a forest of trees, each of which is a single node (a leaf).

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

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.
MST using “Prim’s algorithm”
(should be called “JPD algorithm”)


Help: IPA for Czech

Vojtěch Jarník (Czech pronunciation: [ˈvojtsɛx ˈjarɲik];
**Prim’s algorithm**

At each step, add an edge (that does not form a cycle) with minimum weight, but keep added edge connected to the start (red) node.

- Edge with weight 3
- Edge with weight 5

One of the 4’s

The 2

Minimal set of edges that connect all vertices
**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

Here, Prim chooses (0, 1)
Kruskal chooses (3, 4)

Here, Prim chooses (0, 2)
Kruskal chooses (3, 4)
**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

Here, Prim chooses (0, 1)
Kruskal chooses (3, 4)

Here, Prim chooses (0, 2)
Kruskal chooses (3, 4)
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 (V1, E1), where

\[ V1 = V \]

\[ E1 \text{ is a subset of } E \]

(V1, E1) is a minimal spanning tree – sum of edge weights is minimal
Prim’s (JPD) spanning tree algorithm

\[ V_1 = \{ \text{an arbitrary node of } V \}; \quad E_1 = \{ \}; \]

//inv: \((V_1, E_1)\) is a tree, \(V_1 \leq V, E_1 \leq E\)

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

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

while \((V_1 \cdot \text{size}()) < (V \cdot \text{size}())\) {
    Pick an edge \((u, v)\) with:
    min weight, \(u\) in \(V_1\), \(v\) 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

$V_1 = \{\text{an arbitrary node of } V\}; \quad E_1 = \{\};$

//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:
        min weight, $u$ in $V_1$, $v$ 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 $S$

Note: the edge with weight 6 is not in in $S$ – this avoids cycles
Prim’s (JPD) spanning tree algorithm

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

\[ S = \text{set of edges leaving the single node in } V_1; \]

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

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

\[ V_1 = \{ \text{start node} \}; \quad E_1 = \{ \}; \]
\[ S = \text{set of edges leaving the single node in } V_1; \]

//inv: \((V_1, E_1)\) is a tree, \(V_1 \leq V, E_1 \leq E,\)
//        All edges \((u, v)\) in \(S\) have \(u\) in \(V_1,\)
//        if edge \((u, v)\) has \(u\) in \(V_1\) and \(v\) not in \(V_1, (u, v)\) is in \(S\)

while \((V_1 . \text{size()} < V . \text{size()}\)) {
    Remove from \(S\) an edge \((u, v)\) with min weight;
    if \((v \text{ not in } V_1)\) {
        add \(v\) to \(V_1\); add \((u,v)\) to \(E_1\);
        add edges leaving \(v\) to \(S\)
    }
}

Question: How should we implement set \(S\)?
Prim’s (JPD) spanning tree algorithm

\[ V_1 = \{\text{start node}\}; \quad E_1 = \{\}\;; \]
\[ S = \text{set of edges leaving the single node in } V_1; \]

// inv: \((V_1, E_1)\) is a tree, \(V_1 \leq V, E_1 \leq E,\)
// All edges \((u, v)\) in \(S\) have \(u\) in \(V_1,\)
// if edge \((u, v)\) has \(u\) in \(V_1\) and \(v\) not in \(V_1,\) \((u, v)\) is in \(S\)

while (\(V_1.\text{size}() < V.\text{size}()\)) {
    Remove from \(S\) a min-weight edge \((u, v)\); \quad \#V \log \#E
    if (\(v\) not in \(V_1\)) {
        add \(v\) to \(V_1\); add \((u,v)\) to \(E_1\); \quad \#E \log \#E
        add edges leaving \(v\) to \(S\)
    }
}

Implement \(S\) as a heap.
Use adjacency lists for edges

Thought: Could we use for \(S\) a set of nodes instead of edges? Yes. We don’t go into that here
Application of minimum spanning tree

Maze generation using Prim’s algorithm

The generation of a maze using Prim's algorithm on a randomly weighted grid graph that is 30x20 in size.

In this undirected graph, all edge weights are 1. Which of the following visit the nodes in the same order as Prim(1)?

- Always break ties by choosing the lower-numbered node first.
- In tree traversals, use node 1 as the tree’s root.

- Dijkstra(1)
- BFS(1)
- DFS(1)
- Preorder tree traversal
- Postorder tree traversal
- Level order tree traversal
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
Breadth-first search, Shortest-path, Prim

**Greedy algorithm**: An algorithm that uses the heuristic of making the locally optimal choice at each stage with the hope of finding the global optimum.

Dijkstra’s shortest-path algorithm makes a locally optimal choice: choosing the node in the Frontier with minimum L value and moving it to the Settled set. And, it is proven that it is not just a hope but a fact that it leads to the global optimum.

Similarly, Prim’s and Kruskal’s locally optimum choices of adding a minimum-weight edge have been proven to yield the global optimum: a minimum spanning tree.

**BUT**: Greediness does not always work!
while (a vertex is unmarked) {
    v = best unmarked vertex
    mark v;
    for (each w adj to v)
        update D[w];
}

• Breadth-first-search (bfs)
  – best: next in queue
  – update: D[w] = D[v]+1
• Dijkstra’s algorithm
  – best: next in priority queue
  – update: D[w] = min(D[w], D[v] + c(v,w))
• Prim’s algorithm
  – best: next in priority queue
  – update: D[w] = min(D[w], c(v,w))

c(v,w) is the v→w edge weight
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