EE360C: Algorithms

Greedy Algorithms
Part 3/4

Summer 2019

Department of Electrical and Computer Engineering
University of Texas at Austin

Minimum Spanning Trees

Minimum Spanning Trees

Given a connected undirected graph, $G = (V, E)$ where each edge $(u, v) \in E$ is associated with a weight $w(u, v)$ specifying the cost of the edge, find an acyclic subset $T \subseteq E$ that connects all of the vertices in V and whose total weight:

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

is minimized.

This is a common problem:

- interconnect a set of pins in electronic circuitry using the least amount of wire
- find the least latency paths in a network

Minimum Spanning Tress (cont.)

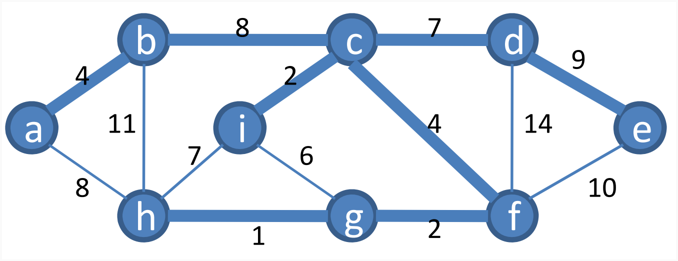
The resulting tree is a *spanning tree* since it touches all of the nodes in G

- we want the **minimum-weight-spanning tree** (or just **minimum spanning tree**)

We look at two algorithms to compute the minimum spanning tree. Both are greedy algorithms

- *greedy* refers to the fact that they make the “best” choice at this moment
- our two minimum spanning tree algorithms *do* end up minimizing the total weight in the spanning tree

An Example MST



Growing a Minimum Spanning Tree

First, let's develop a generic algorithm for computing the minimum spanning tree (MST) by gradually growing the tree

Given a connected, undirected graph, $G(V, E)$ and a weight function $w : E \to \mathbf{R}$, find the minimum spanning tree of G.

Our approach is to grow a set A of edges that will eventually constitute our minimum spanning tree. What should our loop invariant be?

- Prior to each iteration, A is a subset of *some* minimum spanning tree

Growing a Minimum Spanning Tree (cont.)

Let a *safe edge* be any edge that can be added to A while maintaining the invariant.

At each step, we select any safe edge (u, v) and add it to A so that A becomes $A \cup \{(u, v)\}$

```
GENERIC-MST($G, w$)
1  $A \leftarrow \emptyset$
2  **while** $A$ does not form a spanning tree
3      **do** find an edge $(u, v)$ that is safe for $A$
4          $A \leftarrow A \cup \{(u, v)\}$
5  **return** $A$
```

Generic MST Loop Invariant

Remember our loop invariant:

- Prior to each iteration, A is a subset of some minimum spanning tree

Can you prove it true for this generically stated algorithm?

- **Initialization**: initially A is the empty set; it's a subset of the minimum spanning tree
- **Maintenance**: because the edge that we add is selected as a safe edge, it's guaranteed to maintain the invariant when it's added to A
- **Termination**: upon termination, A is a subset of a minimum spanning tree by the loop invariant, and A is a spanning tree by the negation of the guard on the while loop, so A must be a minimum spanning tree upon termination

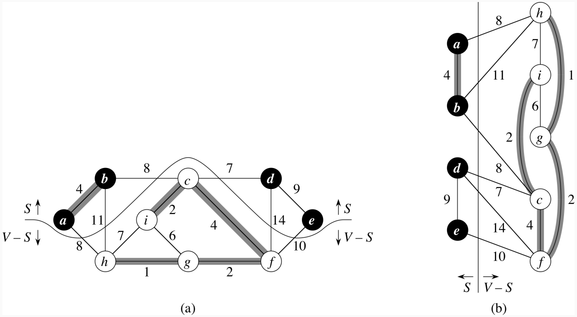
The question is: how do we find a safe edge?

Definitions

A **cut** $(S, V - S)$ of an undirected graph $G = (V, E)$ is a partition of V

- an edge (u, v) **crosses** a cut if one of u and v is in S and the other is in $V - S$
- a cut **respects** a set A of edges if no edges in A **cross** the cut
- an edge is a **light edge** crossing a cut if its weight is the minimum of all of the edges that cross the cut

Definitions Visualized

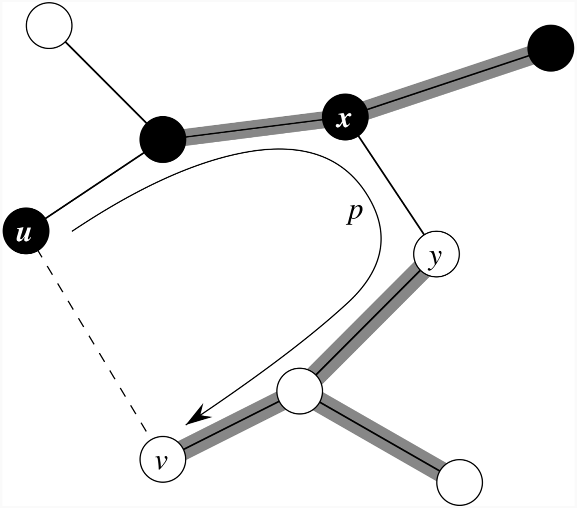


Finding Safe Edges

Theorem

Let $G = (V, E)$ be a connected undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, let $(S, V - S)$ be any cut of G that respects A, and let (u, v) be a light edge crossing $(S, V - S)$. Then (u, v) is a safe edge for A.

Proof Visualized



Proof of the Safe Edge Theorem

- Suppose that T is some minimum spanning tree that includes A but does not contain (u, v).
- We can construct an equivalent minimum spanning tree T' that includes $A \cup \{(u, v)\}$, demonstrating that (u, v) is safe for A.
- (u, v) forms a cycle with the edges on the path from u to v in T. u and v are on opposite sides of the cut $(S, V - S)$; because T is a spanning tree, there must be at least one edge in T that crosses the cut. Call it (x, y).
- $(x, y) \notin A$ because the cut respects A.
- Removing (x, y) and replacing it with (u, v) forms a new spanning tree $T' = T - \{(x, y)\} + \{(u, v)\}$. But is it a *minimum* spanning tree?
- (u, v) is a light edge crossing the cut $(S, V - S)$, so $w(u, v) \leq w(x, y)$.
- But T was a minimum spanning tree, so it must be that $w(u, v) = w(x, y)$, and $w(T') = w(T)$.

The Generic Algorithm Again

What are we really doing in GENERIC-MST?

- at any point, $G_A = (V, A)$ is a forest, and each connected component in G_A is a tree
- any safe edge for A connects distinct components from G_A since $A \cup \{(u, v)\}$ must be acyclic

Corollary

Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, and let $C = (V_C, E_C)$ be a connected component (tree) in the forest $G_A = (V, A)$. If (u, v) is a light edge connecting C to some other component in G_A, then (u, v) is safe for A.

Kruskal's Algorithm

Kruskal's algorithm builds directly on GENERIC-MST. At each step, the algorithm looks at all of the edges connecting any two trees in the forest G_A and chooses the smallest one.

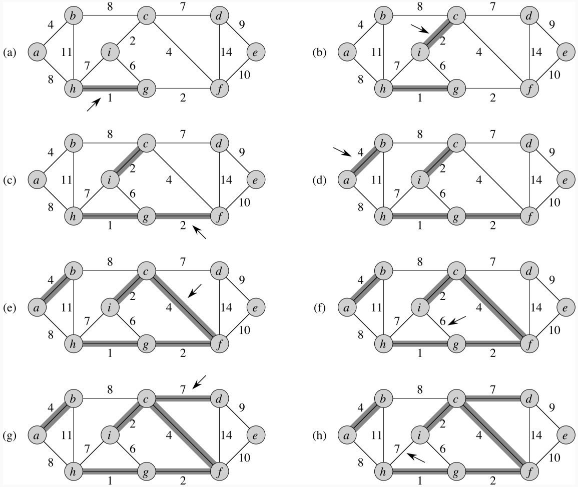
- it's a greedy algorithm because at each step, it just takes the smallest of all of the possibilities

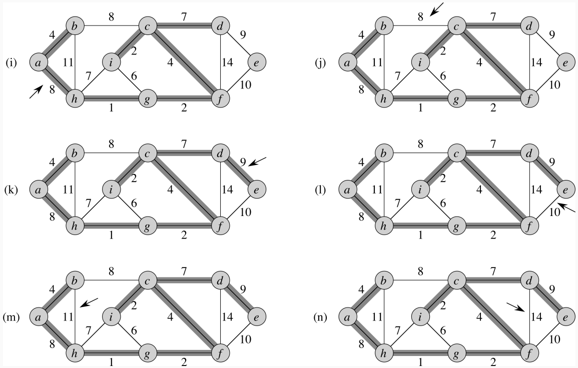
We use the algorithm for computing the connected components of a graph.

- we use a *disjoint-set* data structure to maintain several disjoint sets of elements (initially, each vertex is in its own set)
- FIND-SET(u) returns a representative element from the set that contains u.
- we can use FIND-SET to determine whether two vertices u and v belong to the same tree

Kruskal's Algorithm (cont.)

```
MST-KRUSKAL($G, w$)
1  $A \leftarrow \emptyset$
2  for each vertex $v \in G.V$
3      do MAKE-SET($v$)
4  sort the edges in $G.E$ into nondecreasing order by weight $w$
5  for each edge $(u, v) \in G.E$, in nondecreasing order by weight
6      do if FIND-SET($u$) $\neq$ FIND-SET($v$)
7          then $A \leftarrow A \cup \{(u, v)\}$
8              UNION($u, v$)
9  return $A$
```


(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
a b c d e f g h i
4 8 7 9 11 2 14 4 7 6 10 8 1 2


(i)
(j)
(k)
(l)
(m)
(n)
a
b
c
d
e
f
g
h
i
8
7
4
9
11
2
14
7
6
4
8
10
1
2

Kruskal's Algorithm Running Time

What's the running time?

- initializing A is $O(1)$
- initializing the disjoint sets in lines 2-3 is $O(V)$
- sorting the edges in line 4 is $O(E \lg E)$
- the for loop runs for each edge, and does some amount of work that can be proven to be less than $O(\lg E)$ for each edge; the running time for the loop is therefore $O(E \lg E)$

So the overall running time is: $O(E \lg E)$

Prim's Algorithm

Prim's algorithm differs in that it chooses an arbitrary vertex and grows the minimum spanning tree from there

A is always a single tree.

Prim's algorithm is greedy since it always picks the smallest edge that could grow the tree.

The key challenge is storing and recalling the edges to make it easy to find the right edge to add to *A*.

Prim's Algorithm (cont.)

We construct a min-priority queue Q that holds all of the vertices that are not yet in the minimum spanning tree under construction

- the priority of each vertex v in Q is the minimum weight of any edge connecting v to any vertex in the tree
- we also store $v.\pi$, the parent on the other end of this minimum weight edge

Prim's algorithm does not have to explicitly store A, it's just:

$$A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$$

where r is the provided root vertex

Prim's Algorithm (cont.)

```
MST-PRIM($G, w, r$)
 1  **for** each $u \in G.V$
 2        **do** $u$.key $\leftarrow \infty$
 3           $u.\pi \leftarrow$NIL
 4  $r$.key $\leftarrow 0$
 5  $Q \leftarrow G.V$
 6  **while** $Q \neq \emptyset$
 7        **do** $u \leftarrow$ EXTRACT-MIN($Q$)
 8           **for** each $v \in$ Adj[$u$]
 9                **do if** $v \in Q$ and $w(u, v) < v$.key
10                       **then** $v.\pi \leftarrow u$
11                            $v$.key $\leftarrow w(u, v)$
```

Prim’s Algorithm Running Time

What’s the running time?

- intialization in lines 1-5 is $O(V)$ assuming we use a min-heap as our priority queue
- the **while** loop is executed $|V|$ times, and each EXTRACT-MIN call takes $O(\lg V)$ time, giving us a total here of $O(V \lg V)$
- the **for** loop is executed $O(E)$ times total, and the implicit call to DECREASE-KEY takes $O(\lg V)$ time

So the total is $O(V \lg V + E \lg V) = O(E \lg V)$, (because for connected graphs $|E| \geq |V| - 1$) which is asympotically the same as Kruskal’s since $|E| < |V|^2$.

In Class Exercise

Let G be a weighted undirected graph, where the edge weights are distinct. An edge is *dangerous* if it is the longest edge in some cycle, and an edge is *useful* if it does not belong to any cycle in G.

(a) Prove that any MST of G contains every useful edge.

(b) Prove that any MST of G contains no dangerous edge.

(a) Suppose not. Suppose there is an MST T of G that does not contain some useful edge $e = (u, v)$. T must contain some path from u to v. This path must not contain e. But then in the original graph, there is a cycle defined by this path (which goes from u to v) when it is combined with e. This contradicts the fact that e is a useful edge.

(b) Suppose not. Suppose there is an MST T of G that contains an edge e, which is part of some cycle. There is some edge f in the same cycle that is not in T (since a tree has no cycles). f must have a lower weight than e (because e is a dangerous edge). Create a new tree T' that contains f but not e. T' is a minimum spanning tree with a lower cost than T, which is a contradiction.

Notes on the Greedy Strategy

The Greedy Strategy in General

As a general rule, we are pretty direct in applying the greedy method:

1. Cast the optimization problem into one in which we make a choice and are left with only a single subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice (i.e., that the greedy choice is *safe*)
3. Demonstrate that combining a solution to the remaining subproblem with the greedy choice yields an optimal solution to the original problem

Identifying Greedy Problems

In general, problems that can be solved by a greedy approach exhibit:

- optimal substructure — if an optimal solution to a problem contains within it optimal solutions to subproblems
- greedy-choice property — a globally optimal solution can be arrived at by making a locally optimal choice

The Knapsack Problem

0-1 Knapsack

A thief robbing a store finds n items; the i^{th} item is worth v_i dollars and weighs w_i pounds (both integers). The thief wants to take as valuable a load as possible but can carry only W pounds. Which items should he take?

Fractional Knapsack

The problem is the same as the 0-1 Knapsack problem, but the thief can take fractions of items; he doesn't have to take all or none of a particular item

Fractional Knapsack Problem

Does the problem display the optimal substructure property?

Yes. Take w of item j; then we're left with finding an optimal solution for weight $W - w$ given the remaining $n - 1$ items plus $w_j - w$ of item j

Solution

- calculate v_i / w_i for each item (the value per pound)
- sort the items by value per pound
- take items from the front of the list until the knapsack is full

0-1 Knapsack Problem

Does the problem display the optimal substructure property?

Yes. If we take item j, we're left with finding an optimal solution for weight $W - w_j$ given items $\{1, 2, \ldots, j-1, j+1, \ldots n\}$.

Does the problem satisfy the greedy-choice property?

No. Consider a knapsack that can hold 50 pounds and a set of three possible items: one that weighs 10 pounds and is worth \$60, one that weighs 20 pounds and is worth \$100, and one that weighs 30 pounds and is worth \$120.

0-1 Knapsack Problem (cont.)

What happened?

By choosing greedily, we didn't fill up the knapsack entirely, resulting in a lower effective value per pound of the knapsack.

What we should have done was compare the value of the subproblem that included the greedy choice with the value of the subproblem that *didn't*.

Questions