CS5112: Algorithms and Data Structures for Applications

Minimum spanning trees

Ramin Zabih

Sources: Wikipedia; Kevin Wayne, [Kleinberg/Tardos]

Cornell University

CORNELL TECH

Administrivia

- Prelim (midterm) date: Wednesday October 16
  - In class, closed book
  - Review session in class on October 7
- Web site is: https://cornelltech.github.io/CS5112-F19/
- HW1 is out, due by 10/7
  - Working in groups is important!
- Q3 will be out Thursday, due in 24 hours
- Lectures will be recorded “Real Soon Now”

CORNELL TECH

Lecture Outline

- RB tree example video here
- Graphs, DAG's, trees
- Minimum spanning tree (MST)
- Three simple MST algorithms
- Proofs of correctness
- Implementation notes

Trees, DAGs, graphs

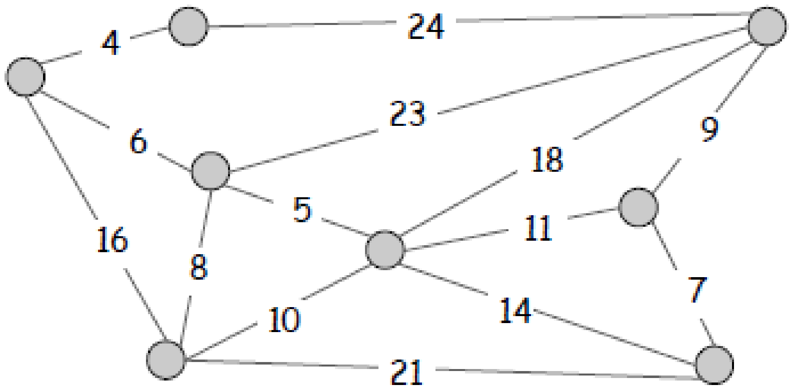
- Most general case: directed graph
- Undirected graph: directed edges in both directions
- DAG: Directed acyclic graph
  - Most ‘tree like’ graph
- Tree

From graphs to trees

- Often useful to find a tree inside a graph
  - Subset of the nodes and edges
  - Almost always cover all nodes, but just some edges
    - Why do we need to omit some edges?
- Example: graph traversal via DFS/BFS
- Sometimes you want to find a tree that is optimal
  - Most definitions of optimal are intractable
    - I.e., require exhaustive search

Spanning trees

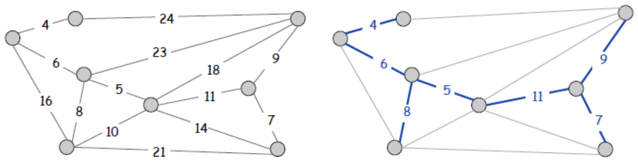
- Definition: a set of edges that spans the graph



- Cayley's theorem: there are n^{n-2} spanning trees
  - Looks very hard to find the 'best' one!

Minimum spanning tree (MST)

- Simplifying assumption: all edge costs distinct
- MST: spanning tree with smallest cost
- Digression: Steiner tree problem in graphs
  - Given an undirected graph with non-negative edge weights and a subset of vertices, usually referred to as terminals
  - Find a tree of minimum weight that contains all terminals (but may include additional vertices)



CORNELL
TECH

MST applications

- Obvious: network connectivity
  - Old fashioned ‘emergency notification’ network
- Non-obvious:
  - Traveling salesman problem
    - Very cool application
  - Image segmentation
    - See: Greg’s lecture on Union-Find

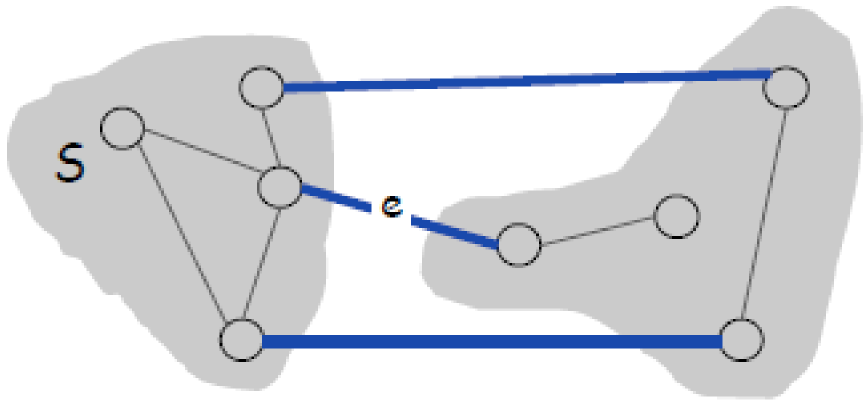
Basic MST algorithms

- Prim: start with a root note and greedily grow a tree outwards, always adding cheapest edge at tree fringe
- Kruskal: start with an empty tree, insert edges cheapest first, unless the edge would create a cycle
  - Nice Kruskal visualization
- Reverse-delete: start with full graph, delete edges most expensive first, unless the edge would disconnect the tree

CORNELL TECH

Cut property

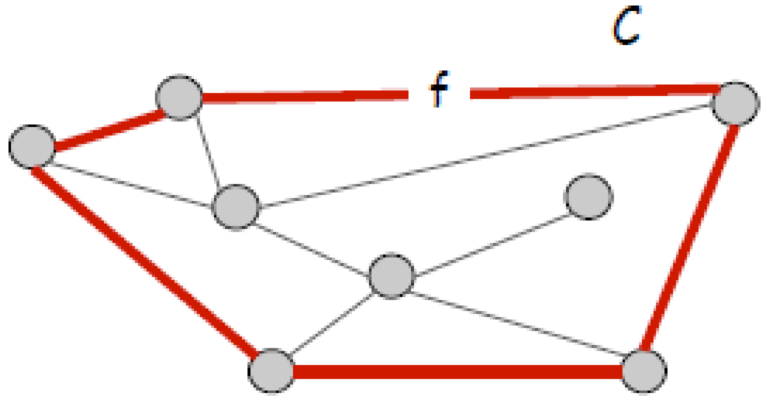
- Let S be any subset of nodes and let e be the min cost edge with exactly one endpoint in S.



- Then the MST must contain e. Why?
  - We will see that this is not trivial

Cycle property

- Let C be any cycle and f the max cost edge belonging to C.

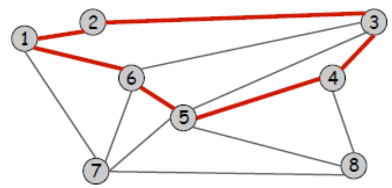


- Then the MST must not contain f. Why?

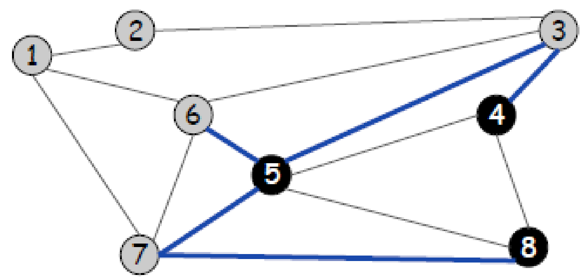
CORNELL TECH

Cycles, cuts, cutsets

- Cycle: set of edges returning to a node.

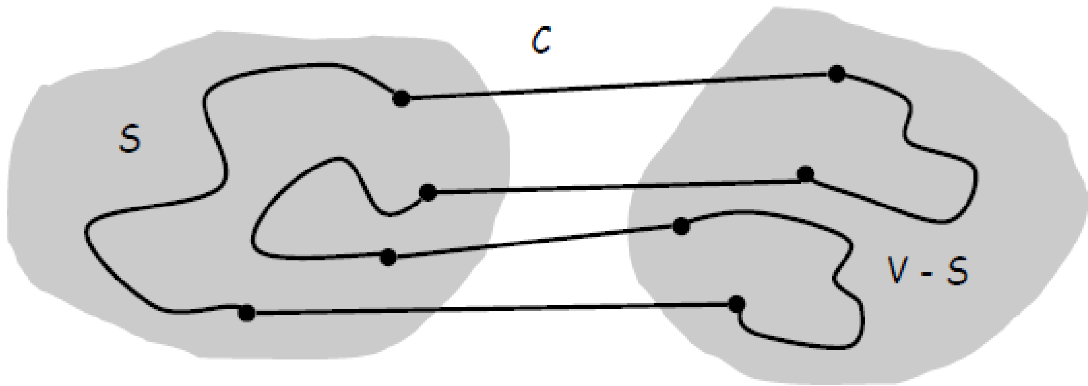


- Cut: subset of nodes. Cutset is the edges on the fringe, i.e. with exactly one side in the cut.
  - Cut = $\{4,5,8\}$
  - Cutset = $\{5-6, 5-7, 3-4, 3-5, 7-8\}$



Cycle-cutset intersection

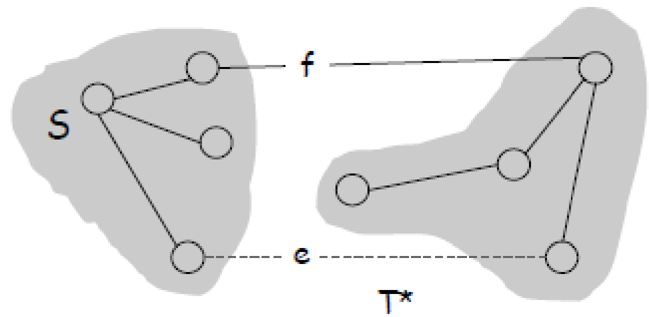
- A cycle and a cutset intersect at an even number of edges
  - Why?
  - Cycle must leave and enter the cut same number of times



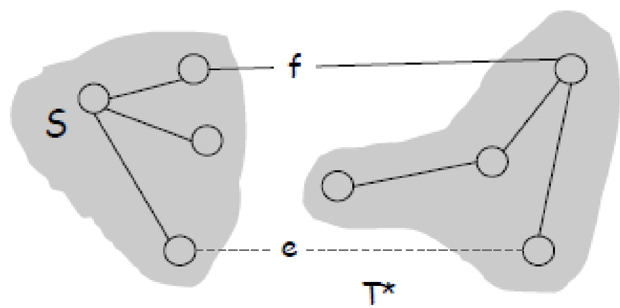
  - Corollary: can't have a lone edge in both cycle and cutset

Proof sketches (exchange arguments)

- Cut property: cheapest e in MST



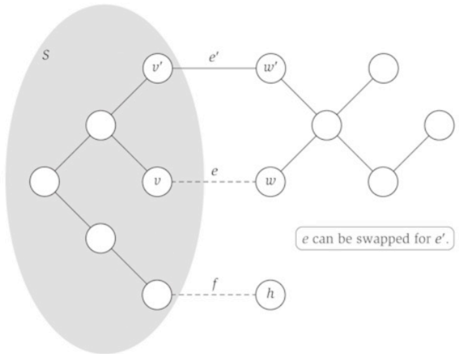
- Cycle property: most expensive f not in MST



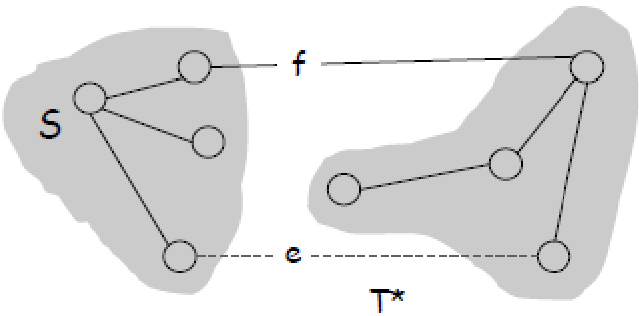
CORNELL TECH

Exchange argument is not trivial

- Consider cut property proof

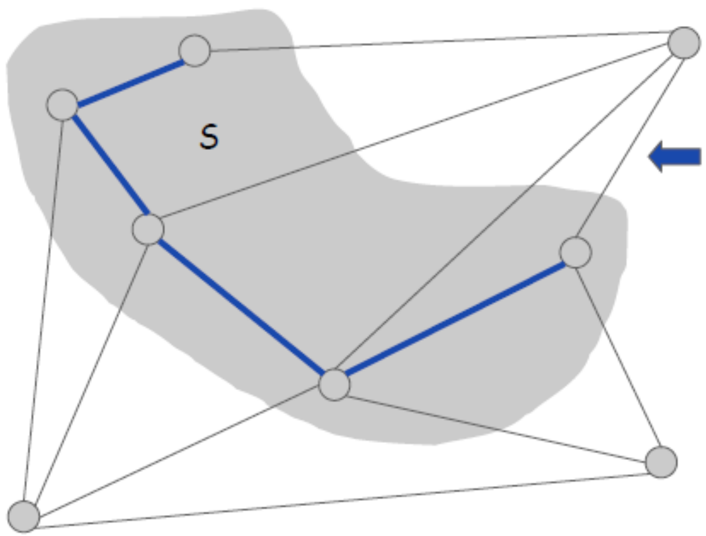


- Need cycle-cutset intersection



CORNELL TECH

Correctness proof for Prim



- Apply cut property at each step

Prim implementation

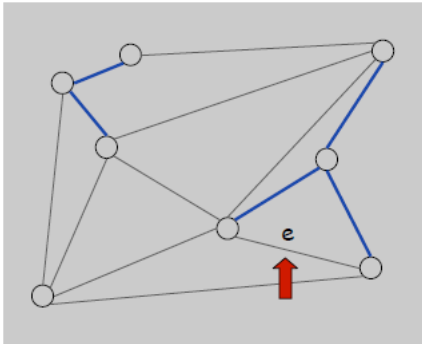
- For each unexplored node v maintain cost of cheapest way of adding it to explored set

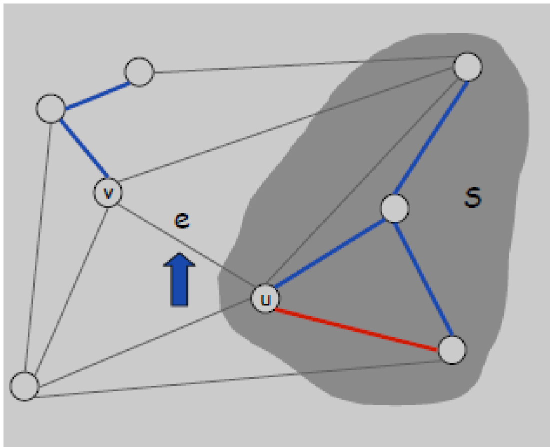
```
Prim(G, c) {
   foreach (v ∈ V) a[v] ← ∞
   Initialize an empty priority queue Q
   foreach (v ∈ V) insert v onto Q
   Initialize set of explored nodes S ← φ

   while (Q is not empty) {
      u ← delete min element from Q
      S ← S ∪ { u }
      foreach (edge e = (u, v) incident to u)
          if ((v ∉ S) and (c_e < a[v]))
             decrease priority a[v] to c_e
}
```

- Speed is $O(n^2)$ naively, $O(m \log n)$ with priority queue

Correctness proof for Kruskal

- Two cases for new edge e
  - Creates a cycle: discard using cycle property
  - Otherwise: insert $e = (u - v)$ into MST
  - Why? Cut property
    - S is connected component with u





CORNELL TECH

Kruskal implementation

- Maintain set of edges for each connected component in MST
- Use union-find to merge

```
Kruskal(G, c) {
   Sort edges weights so that c1 ≤ c2 ≤ ... ≤ cm.
   T ← φ

   foreach (u ∈ V) make a set containing singleton u

   for i = 1 to m          are u and v in different connected components?
      (u,v) = ei
      if (u and v are in different sets) {
         T ← T ∪ {ei}
         merge the sets containing u and v
      }                    merge two components
   return T
}
```

CORNELL TECH

Kruskal vs Prim

- Both are $O(m \log n)$
  - Though a very painful datastructure can improve Prim
- In general, Kruskal is better for sparse graphs and Prim for dense graphs
- Practical impact of reference locality
  - Interaction with CPU memory architecture