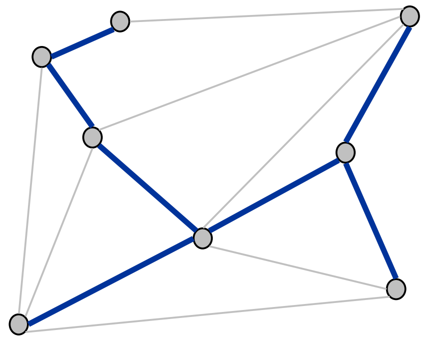
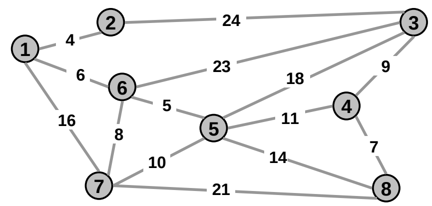
Minimum Spanning Tree



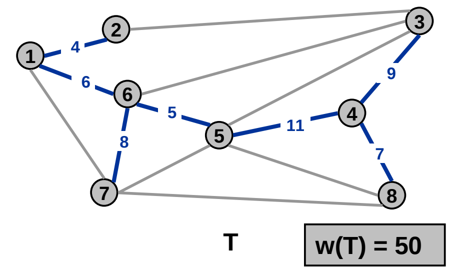
Some of these lecture slides are adapted from material in:
- *Algorithms in C*, R. Sedgewick.

Minimum Spanning Tree

Minimum spanning tree (MST). Given connected graph G with positive edge weights, find a min weight set of edges that connects all of the vertices.



G



T

Cayley's Theorem (1889). There are V^{V-2} spanning trees on the complete graph on V vertices.

- Can't solve MST by brute force.

Applications

MST is fundamental problem with diverse applications.

- Designing physical networks.
  - telephone, electrical, hydraulic, TV cable, computer, road
- Cluster analysis.
  - delete long edges leaves connected components
  - finding clusters of quasars and Seyfert galaxies
  - analyzing fungal spore spatial patterns
- Approximate solutions to NP-hard problems.
  - metric TSP, Steiner tree
- Indirect applications.
  - max bottleneck paths
  - describing arrangements of nuclei in skin cells for cancer research
  - learning salient features for real-time face verification
  - modeling locality of particle interactions in turbulent fluid flow
  - reducing data storage in sequencing amino acids in a protein

Optimal Message Passing

Optimal message passing.

- Distribute message to N agents.
- Each agent can communicate with some of the other agents, but their communication is (independently) detected with probability p_{ij}.
- Group leader wants to transmit message to all agents so as to minimize the total probability that message is detected.

Objective.

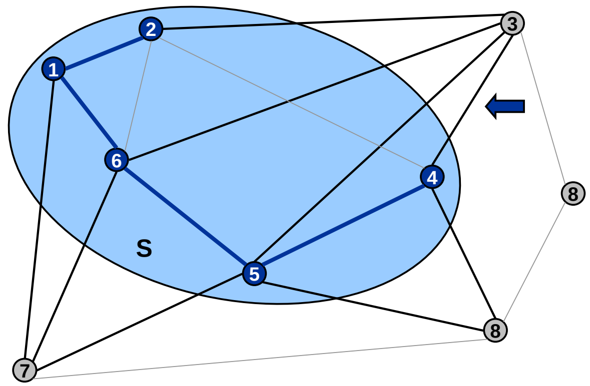
- Find tree T that minimizes: $1 - \prod_{(i,j) \in T} (1 - p_{ij})$
- Or equivalently, that maximizes: $\prod_{(i,j) \in T} (1 - p_{ij})$
- Or equivalently, that maximizes: $\sum_{(i,j) \in T} \log(1 - p_{ij})$
  - MST with weights = - log (1 - p_{ij}) weights p_{ij} also work!

Prim's Algorithm

Prim's algorithm. (Jarník 1930, Dijkstra 1957, Prim 1959)

- Initialize F = ϕ, S = {s} for some arbitrary vertex s.
- Repeat until S has V vertices:
  - let f be smallest edge with exactly one endpoint in S
  - add other endpoint to S
  - add edge f to F

S	F
1	-
2	1-2
6	1-6
5	6-5
4	5-4

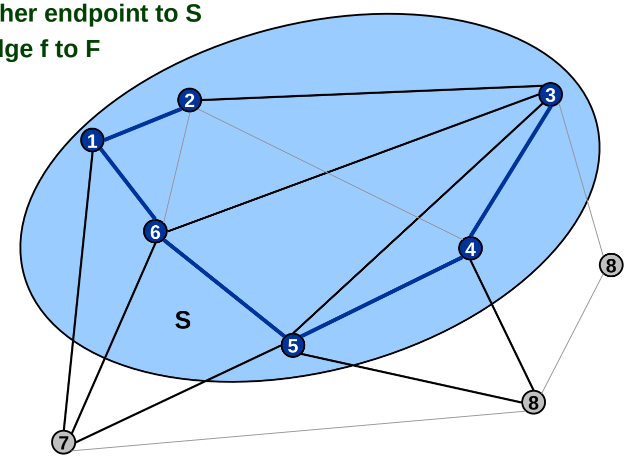


Prim's Algorithm

Prim's algorithm. (Jarník 1930, Dijkstra 1957, Prim 1959)

- Initialize F = ϕ, S = {s} for some arbitrary vertex s.
- Repeat until S has V vertices:
  - let f be smallest edge with exactly one endpoint in S
  - add other endpoint to S
  - add edge f to F

S	F
1	-
2	1-2
6	1-6
5	6-5
4	5-4
3	4-3



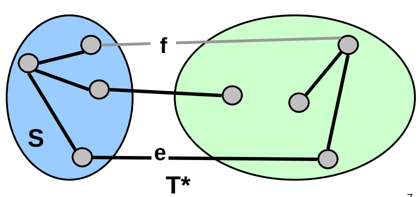
Prim's Algorithm: Proof of Correctness

Theorem. Upon termination of Prim's algorithm, F is a MST.

Proof. (by induction on number of iterations)

Invariant: There exists a MST T* containing all of the edges in F.

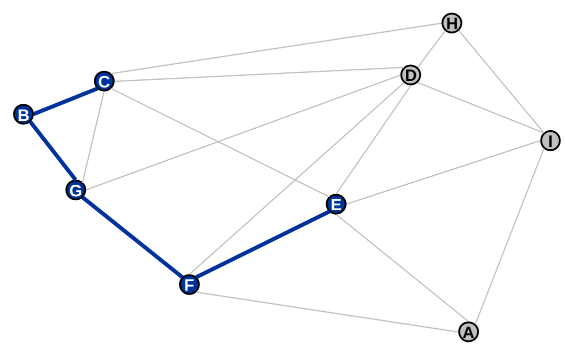
- Base case: F = ϕ $\Rightarrow$ every MST satisfies invariant.
- Induction step: true at beginning of iteration i.
  - at beginning of iteration i, let S be vertex subset and let f be the edge that Prim's algorithm chooses
  - if f $\in$ T*, T* still satisfies invariant
  - o/w, consider cycle C formed by adding f to T*
  - let e $\in$ C be another arc with exactly one endpoint in S
  - $c_f \leq c_e$ since algorithm chooses f instead of e
  - e $\notin$ F by definition of S
  - T* $\cup$ { f } - { e } satisfies invariant



Prim's Algorithm: Classic Implementation

Use adjacency matrix.

- S = set of vertices in current tree.
- For each vertex not in S, maintain vertex in S to which it is closest.
- Choose next vertex to add to S using min `dist[w]`.
- Just before adding new vertex v to S:
  - for each neighbor `w` of v, if `w` is closer to v than to a vertex in S, update `dist[w]`

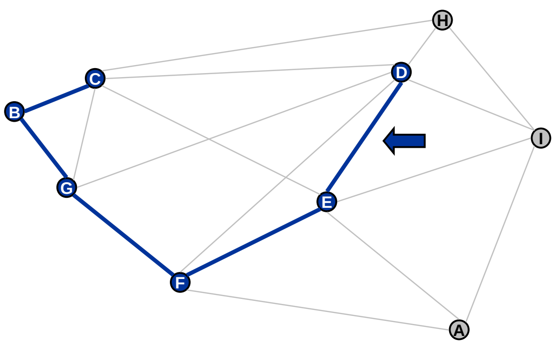


Vertex	Nearest	Dist
A	E	15
B	-	-
C	-	-
D	E	9
E	-	-
F	-	-
G	-	-
H	C	23
I	E	11

Prim's Algorithm: Classic Implementation

Use adjacency matrix.

- S = set of vertices in current tree.
- For each vertex not in S, maintain vertex in S to which it is closest.
- Choose next vertex to add to S using min `dist[w]`.
- Just before adding new vertex v to S:
  - for each neighbor `w` of v, if `w` is closer to v than to a vertex in S, update `dist[w]`



Vertex	Nearest	Dist
A	E	15
B	-	-
C	-	-
D	-	-
E	-	-
F	-	-
G	-	-
H	D	4
I	D	6

Prim's Algorithm: Classic Implementation

Running time.

- V - 1 iterations since each iteration adds 1 vertex.

Each iteration consists of:

- Choose next vertex to add to S by minimum `dist[w]` value.
  - O(V) time.
- For each neighbor w of v, if w is closer to v than to a vertex in S, update `dist[w]`.
  - O(V) time.

$O(V^2)$ overall.

Prim's Algorithm: Priority Queue Implementation

Prim's Algorithm pseudocode

```
Q ← PQinit()
for each vertex v in graph G
   key(v) ← ∞
   pred(v) ← nil
   PQinsert(v, Q)

key(s) ← 0
while (!PQisempty(Q))
   v = PQdelmin(Q)
   for each edge v-w s.t. w is in Q
      if key(w) > c(v,w)
         PQdeckey(w, c(v,w), Q)
         pred(w) ← v
```


Prim's Algorithm: Priority Queue Implementation

Analysis of Prim's algorithm.

- `PQinsert()`: V vertices.
- `PQisempty()`: V vertices.
- `PQdelmin()`: V vertices.
- `PQdeckey()`: E edges.

	Priority Queues		
Operation	Array	Binary heap	Fibonacci heap*
insert	N	log N	1
delete-min	N	log N	log N
decrease-key	1	log N	1
is-empty	1	1	1
Prim	V^2	E log V	E + V log V

PFS vs. Classic Prim

Which algorithm is faster?

- Classic Prim: $O(V^2)$.
- Prim with binary heap: $O(E \log V)$.

Answer depends on whether graph is SPARSE or DENSE.

- 2,000 vertices, 1 million edges
  - Heap: 2-3 times SLOWER
- 100,000 vertices, 1 million edges
  - Heap: 500 times FASTER
- 1 million vertices, 2 million edges
  - Heap: 10,000 times FASTER.

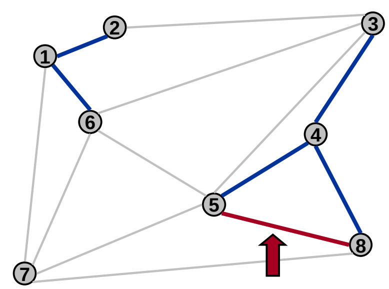
Bottom line.

- Classic Prim is optimal for dense graphs.
- Heap implementation far better for sparse graphs.

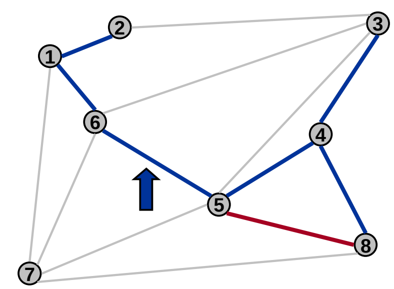
Kruskal's Algorithm

Kruskal's algorithm (1956).

- Initialize F = ϕ.
- Consider arcs in ascending order of weight.
- If adding edge to forest F does not create a cycle, then add it. Otherwise, discard it.



Case 1: {5, 8}

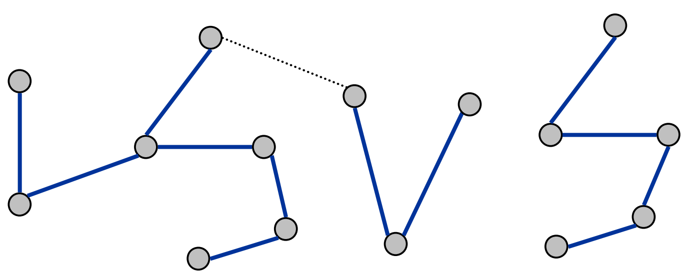


Case 2: {5, 6}

Kruskal's Algorithm: Implementation

How to check if adding an edge to F would create a cycle?

- Naïve solution: use depth first search.
- Clever solution: use union-find data structure from Lecture 1.
  - each tree in forest corresponds to a set
  - to see if adding edge between v and w creates a cycle, check if v and w are already in same component
  - when adding v-w to forest F, merge sets containing v and w



Kruskal's Algorithm: C Implementation

Kruskal's Algorithm

```
// Fill up mst[] with list of edges in MST of graph G
void GRAPHmstE(Graph G, Edge mst[]) {
   int i, k, v, w;
   Edge a[MAXE];              // list of all edges in G
   int E = GRAPHedges(a, G);  // # edges in G
   sort(a, 0, E-1);           // sort edges by weight

   UFinit(G->V);
   for (i = k = 0; i < E && k < G->V-1; i++) {
      v = a[i].v;
      w = a[i].w;
      // if edge a[i] doesn't create a cycle, add to tree
      if (!UFfind(v, w)) {
         UFunion(v, w);
         mst[k++] = a[i];
      }
   }
}
```

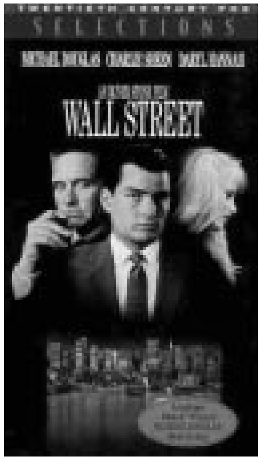
Kruskal's Algorithm: Proof of Correctness

Theorem. Upon termination of Kruskal's algorithm, F is a MST.

Proof. Identical to proof of correctness for Prim's algorithm except that you let S be the set of nodes in component of F containing v.

Corollary. "Greed is good. Greed is right. Greed works. Greed clarifies, cuts through, and captures the essence of the evolutionary spirit."

Gordon Gecko
(Michael Douglas)



Kruskal's Algorithm: Running Time

Kruskal analysis. O(E log V) time.

- `Sort()`: O(E log E) = O(E log V).
- `UFinit()`: V singleton sets.
- `UFfind()`: at most once per edge.
- `UFunion()`: exactly V – 1 times.

If edges already sorted. O(E log* V) time.

- Any sequence of M union-find operations on N elements takes O(M log* N) time.
- In this universe, log* N $\leq$ 6.

Advanced MST Algorithms

Deterministic comparison based algorithms.

- O(E log V) Prim, Kruskal, Boruvka.
- O(E log log V). Cheriton-Tarjan (1976), Yao (1975).
- O(E log*V). Fredman-Tarjan (1987).
- O(E log (log*V)). Gabow-Galil-Spencer-Tarjan (1986).
- O(E α (E, V)). Chazelle (2000).
- O(E). Holy grail.

Worth noting.

- O(E) randomized. Karger-Klein-Tarjan (1995).
- O(E) verification. Dixon-Rauch-Tarjan (1992).