CSE373: Data Structures & Algorithms

Lecture 17: Dijkstra's Algorithm

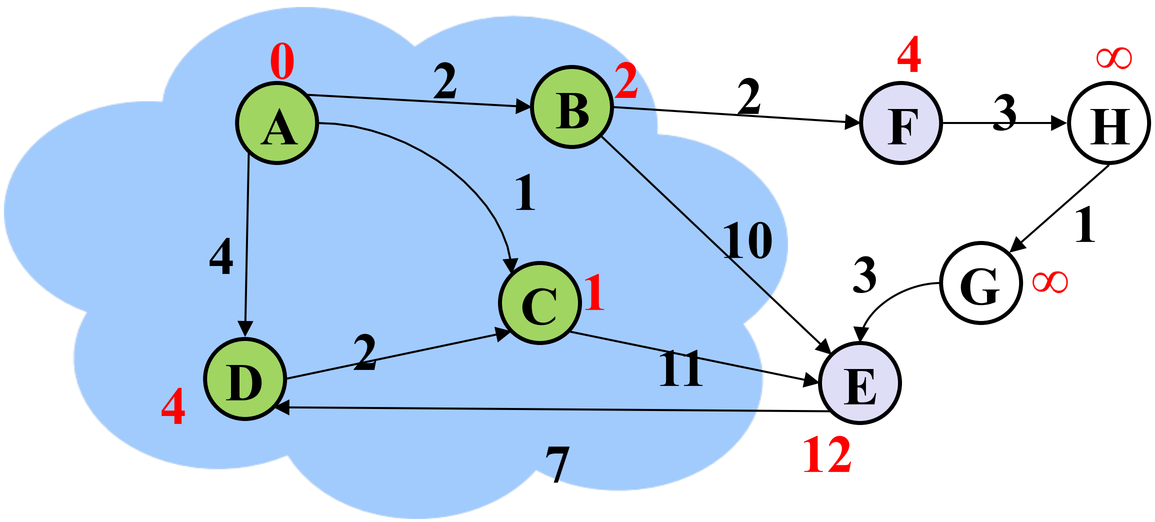
Lauren Milne

Summer 2015

Announcements

- Homework 4 due tonight
- Homework 5 out today

Dijkstra's Algorithm: Lowest cost paths



- Initially, start node has cost 0 and all other nodes have cost ∞
- At each step:
  - Pick closest unknown vertex **v**
  - Add it to the "cloud" of known vertices
  - Update distances for nodes with edges from **v**
- That's it!

The Algorithm

1. For each node **v**, set **v.cost** = ∞ and **v.known = false**
2. Set **source.cost = 0**
3. While there are unknown nodes in the graph
   a) Select the unknown node **v** with lowest cost
   b) Mark **v** as known
   c) For each edge **(v,u)** with weight **w**,

```
c1 = v.cost + w // cost of best path through v to u
c2 = u.cost   // cost of best path to u previously known
if(c1 < c2){ // if the path through v is better
  u.cost = c1
  u.path = v // for computing actual paths
}
```

A Greedy Algorithm

- Dijkstra's algorithm is an example of a *greedy algorithm*:
  - At each step, always does what seems best at that step
    - A locally optimal step, not necessarily globally optimal
  - Once a vertex is known, it is not revisited
    - Turns out to be globally optimal (for this problem)

Where are we?

- Had a problem: Compute shortest paths in a weighted graph with no negative weights
- Learned an algorithm: Dijkstra's algorithm
- What should we do after learning an algorithm?
  - Prove it is correct
    - Did this last time, not doing it again
  - Analyze its efficiency
    - Will do better by using a data structure we learned earlier!

Efficiency, first approach

Use pseudocode to determine asymptotic run-time

- Notice each edge is processed only once

```
dijkstra(Graph G, Node start) {
  for each node: x.cost=infinity, x.known=false
  start.cost = 0
  while(not all nodes are known) {
    b = find unknown node with smallest cost
    b.known = true
    for each edge (b,a) in G
     if(!a.known)
       if(b.cost + weight((b,a)) < a.cost){
         a.cost = b.cost + weight((b,a))
         a.path = b
       }
}
```

Efficiency, first approach

Use pseudocode to determine asymptotic run-time

- Notice each edge is processed only once

```
dijkstra(Graph G, Node start) {
  for each node: x.cost=infinity, x.known=false
  start.cost = 0
  while(not all nodes are known) {
    b = find unknown node with smallest cost
    b.known = true
    for each edge (b,a) in G
     if(!a.known)
       if(b.cost + weight((b,a)) < a.cost){
         a.cost = b.cost + weight((b,a))
         a.path = b
       }
}
```

$O(|V|)$

$O(|V|^2)$

$O(|E|)$

$O(|V|^2)$

Improving asymptotic running time

- So far: $O(|V|^2)$
- We had a similar "problem" with topological sort being $O(|V|^2)$ due to each iteration looking for the node to process next
  - We solved it with a queue of zero-degree nodes
  - But here we need the lowest-cost node and costs can change as we process edges
- Solution?
  - A priority queue holding all unknown nodes, sorted by cost
  - But must support **`decreaseKey`** operation
    - Must maintain a reference from each node to its current position in the priority queue
    - Conceptually simple, but can be a pain to code up

Efficiency, second approach

Use pseudocode to determine asymptotic run-time

```
dijkstra(Graph G, Node start) {
  for each node: x.cost=infinity, x.known=false
  start.cost = 0
  build-heap with all nodes
  while(heap is not empty) {
    b = deleteMin()
    b.known = true
    for each edge (b,a) in G
     if(!a.known)
      if(b.cost + weight((b,a)) < a.cost){
        decreaseKey(a,"new cost – old cost")
        a.path = b
      }
}
```

Efficiency, second approach

Use pseudocode to determine asymptotic run-time

```
dijkstra(Graph G, Node start) {
  for each node: x.cost=infinity, x.known=false
  start.cost = 0
  build-heap with all nodes
  while(heap is not empty) {
    b = deleteMin()
    b.known = true
    for each edge (b,a) in G
     if(!a.known)
      if(b.cost + weight((b,a)) < a.cost){
        decreaseKey(a,"new cost – old cost")
        a.path = b
      }
}
```

O(|V|)

O(|V|log|V|)

O(|E|log|V|)

O(|V|log|V|+|E|log|V|)

Dense vs. sparse again

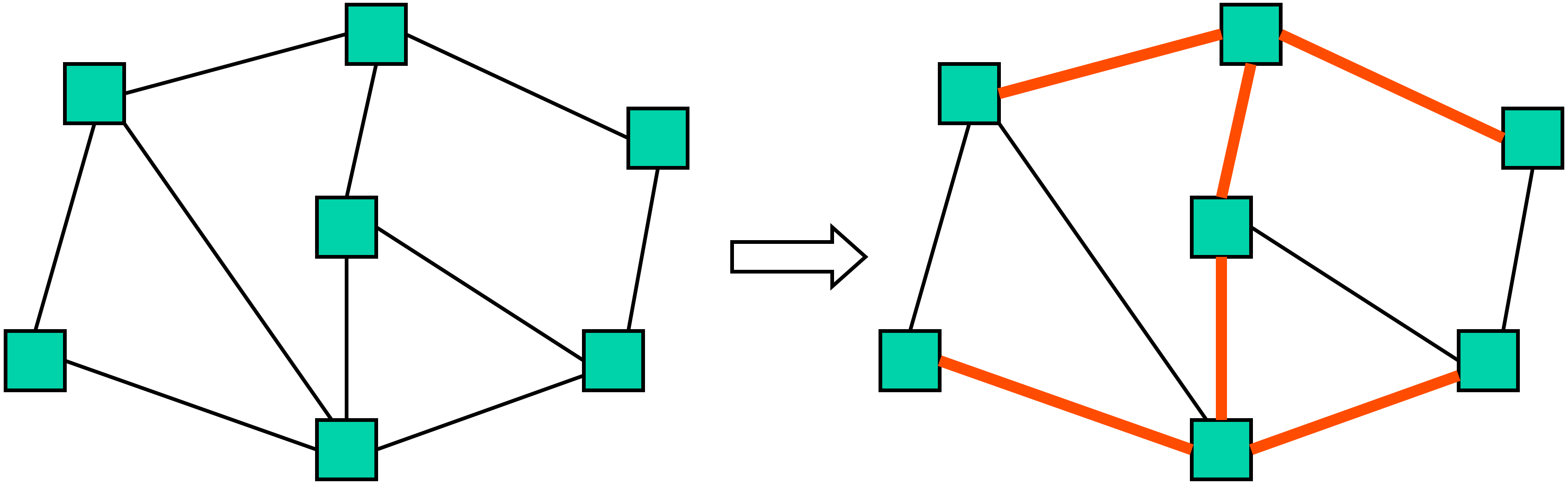
- First approach: $O(|V|^2)$
- Second approach: $O(|V|\log|V|+|E|\log|V|)$
- So which is better?
  - Sparse: $O(|V|\log|V|+|E|\log|V|)$ (if $|E| > |V|$, then $O(|E|\log|V|)$)
  - Dense: $O(|V|^2)$
- But, remember these are worst-case and asymptotic
  - Priority queue might have slightly worse constant factors
  - On the other hand, for “normal graphs”, we might call **`decreaseKey`** rarely (or not percolate far), making $|E|\log|V|$ more like $|E|$

Done with Dijkstra's

- You will implement Dijkstra's algorithm in homework 5 ☺
- Onward….. Spanning trees!

Spanning Trees

- A simple problem: Given a *connected* undirected graph **G**=(**V**,**E**), find a minimal subset of edges such that **G** is still connected
  - A graph **G2**=(**V**,**E2**) such that **G2** is connected and removing any edge from **E2** makes **G2** disconnected



Observations

1. Any solution to this problem is a tree
   - Recall a tree does not need a root; just means acyclic
   - For any cycle, could remove an edge and still be connected

2. Solution not unique unless original graph was already a tree

3. Problem ill-defined if original graph not connected
   - So $|E| \geq |V|-1$

4. A tree with $|V|$ nodes has $|V|-1$ edges
   - So every solution to the spanning tree problem has $|V|-1$ edges

Motivation

A spanning tree connects all the nodes with as few edges as possible

- Example: A "phone tree" so everybody gets the message and no unnecessary calls get made
  - Bad example since would prefer a balanced tree

In most compelling uses, we have a *weighted* undirected graph and we want a tree of least total cost

- Example: Electrical wiring for a house or clock wires on a chip
- Example: A road network if you cared about asphalt cost rather than travel time

This is the minimum spanning tree problem

- Will do that next, after intuition from the simpler case

Two Approaches

Different algorithmic approaches to the spanning-tree problem:

1. Do a graph traversal (e.g., depth-first search, but any traversal will do), keeping track of edges that form a tree

2. Iterate through edges; add to output any edge that does not create a cycle

Spanning tree via DFS

```
spanning_tree(Graph G) {
  for each node i
      i.marked = false
  for some node i: f(i)
}
f(Node i) {
  i.marked = true
  for each j adjacent to i:
    if(!j.marked) {
      add(i,j) to output
      f(j) // DFS
    }
}
```

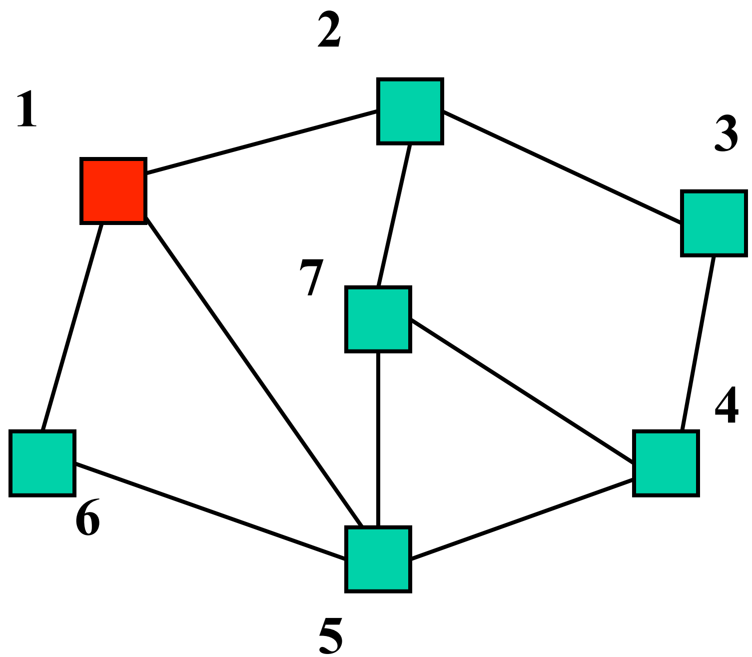
Correctness: DFS reaches each node. We add one edge to connect it to the already visited nodes. Order affects result, not correctness.

Time: $O(|\mathbf{E}|)$

Example

Stack

f(1)



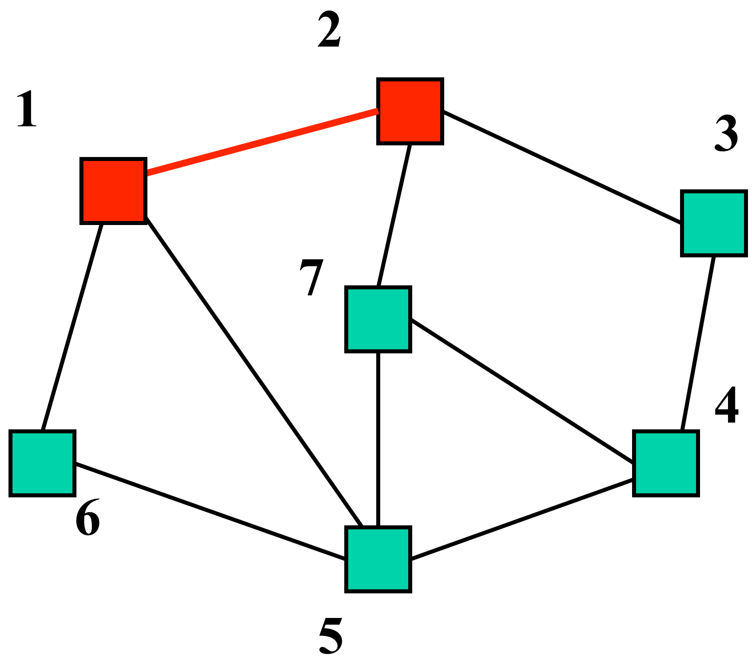
Output:

Example

Stack

f(1)

f(2)



Output: (1,2)

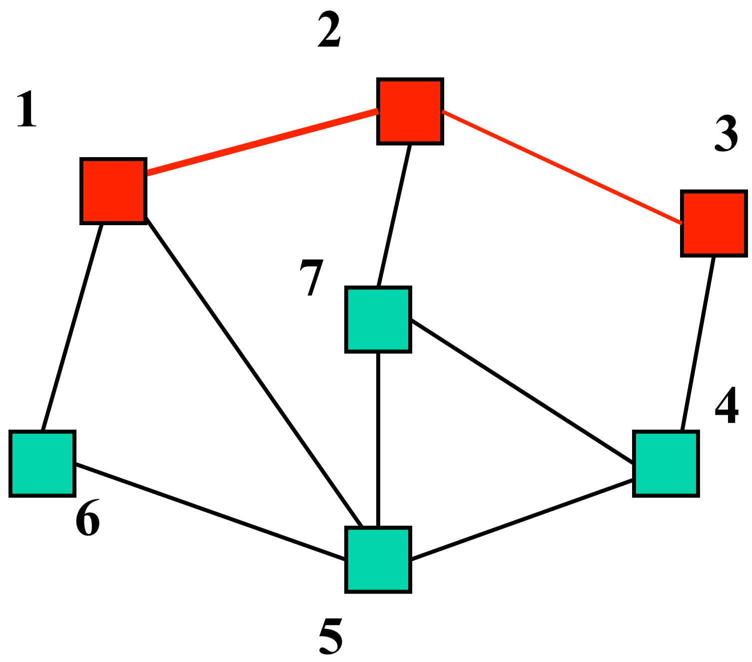
Example

Stack

f(1)

f(2)

f(3)



Output: (1,2), (2,3)

Example

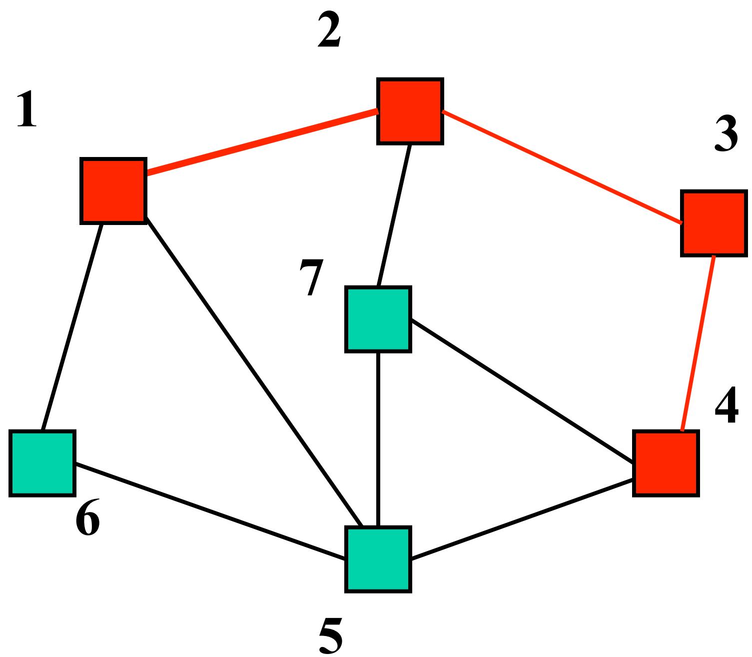
Stack

f(1)

f(2)

f(3)

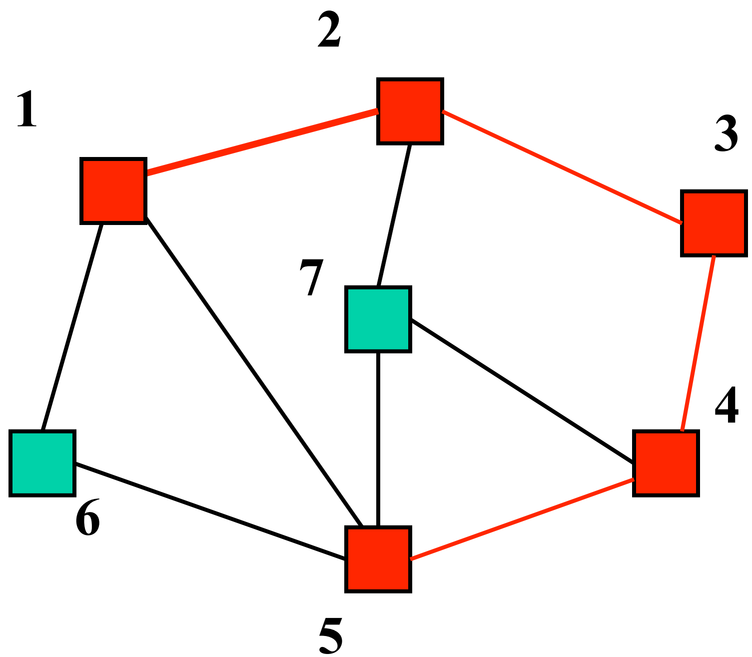
f(4)



Output: (1,2), (2,3), (3,4)

Example

Stack
f(1)
f(2)
f(3)
f(4)
f(5)



Output: (1,2), (2,3), (3,4), (4,5)

Example

Stack

f(1)

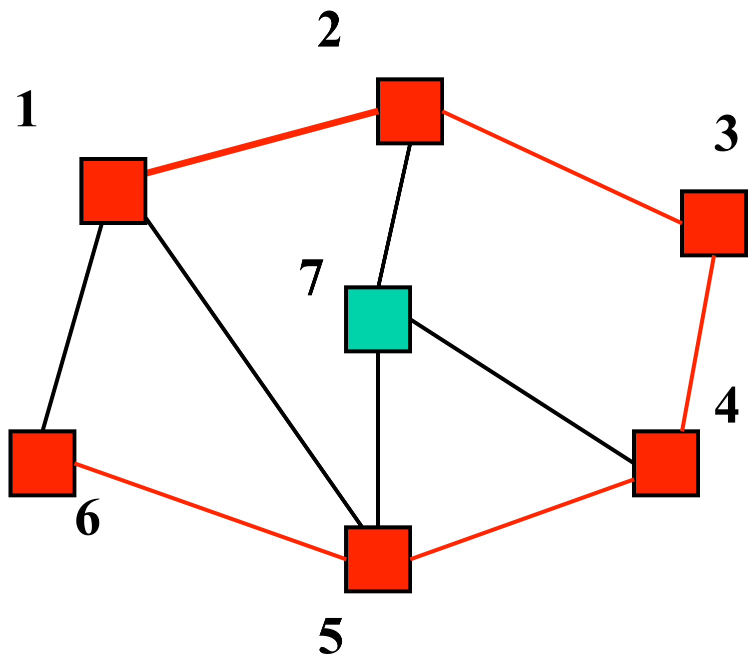
f(2)

f(3)

f(4)

f(5)

f(6)



Output: (1,2), (2,3), (3,4), (4,5), (5,6)

Example

Stack

f(1)

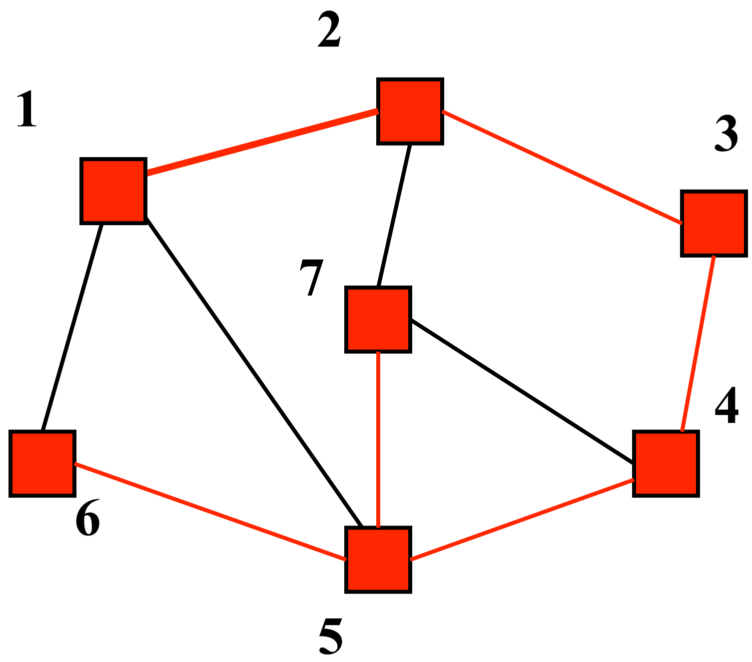
f(2)

f(3)

f(4)

f(5)

f(6), f(7)



Output: (1,2), (2,3), (3,4), (4,5), (5,6), (5,7)

Example

Stack

f(1)

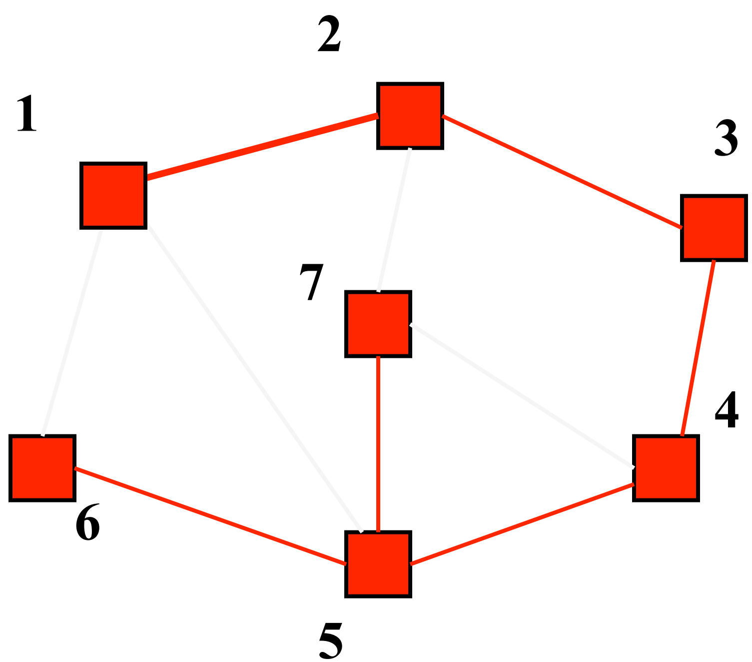
f(2)

f(3)

f(4)

f(5)

f(6), f(7)



Output: (1,2), (2,3), (3,4), (4,5), (5,6), (5,7)

Second Approach

Iterate through edges; output any edge that does not create a cycle

Correctness (hand-wavy):

- Goal is to build an acyclic connected graph
- When we add an edge, it adds a vertex to the tree
  - Else it would have created a cycle
- The graph is connected, so we reach all vertices

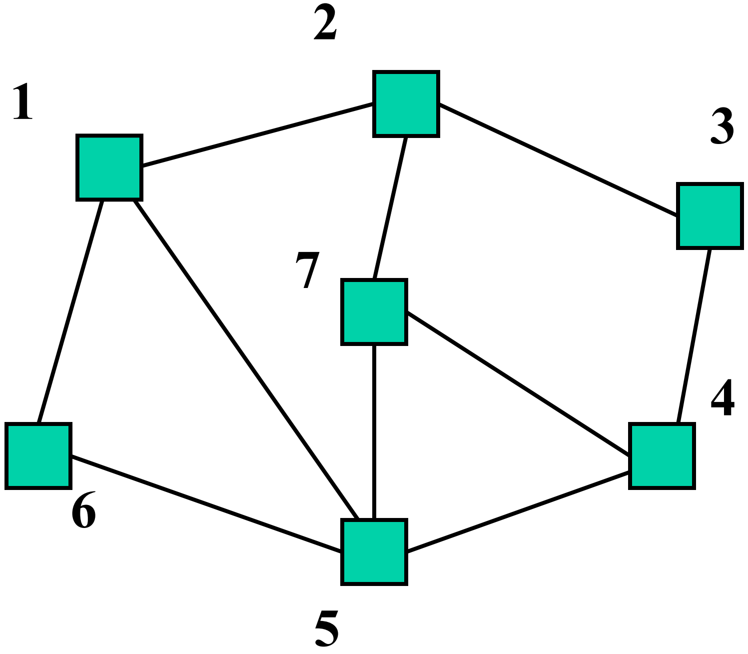
Efficiency:

- Depends on how quickly you can detect cycles
- Reconsider after the example

Example

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7),(1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

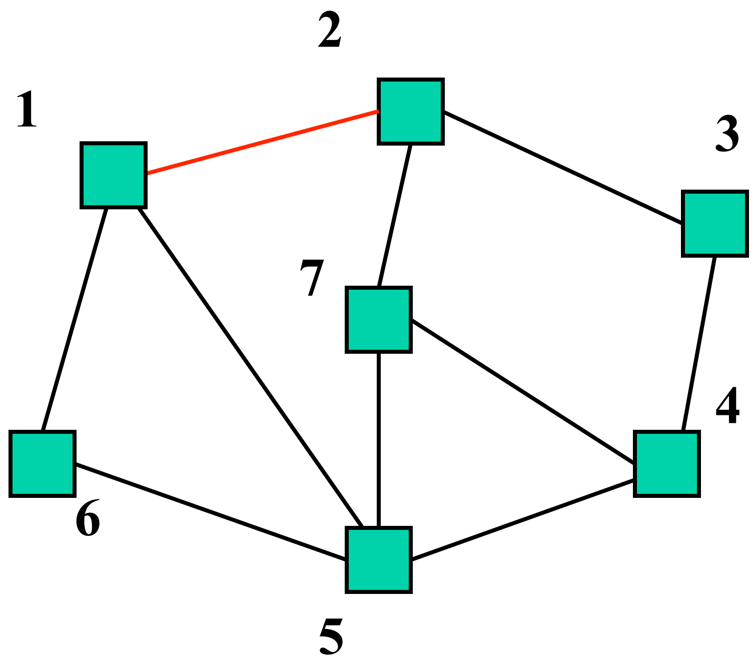


Output:

Example

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7),(1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

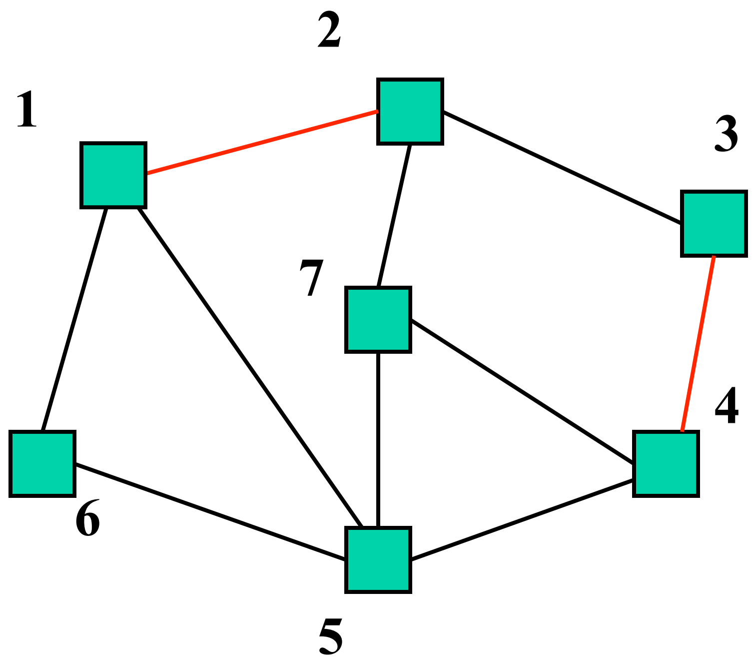


Output: (1,2)

Example

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7),(1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

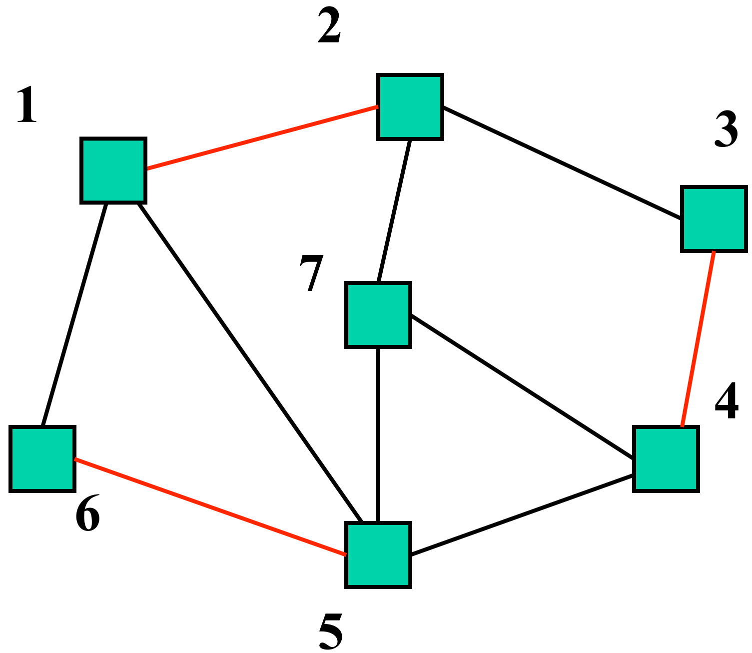


Output: (1,2), (3,4)

Example

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7),(1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

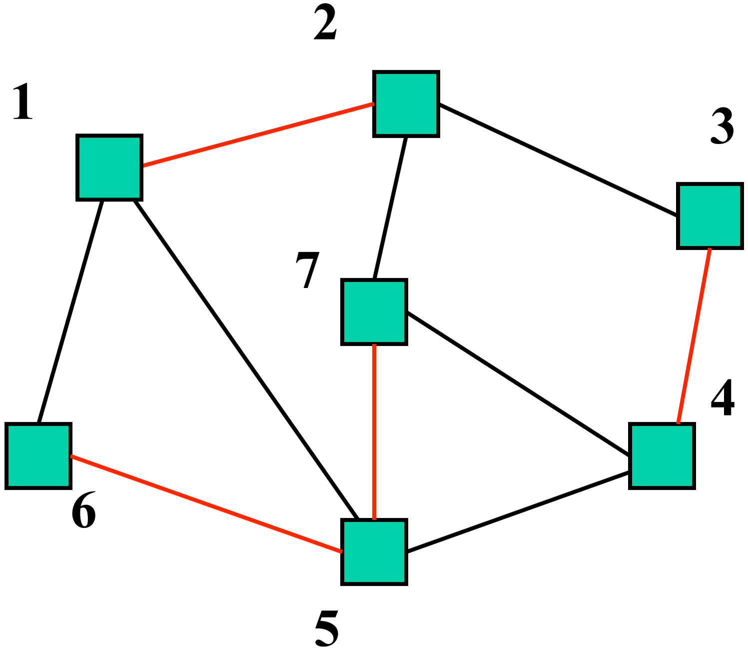


Output: (1,2), (3,4), (5,6),

Example

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7),(1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

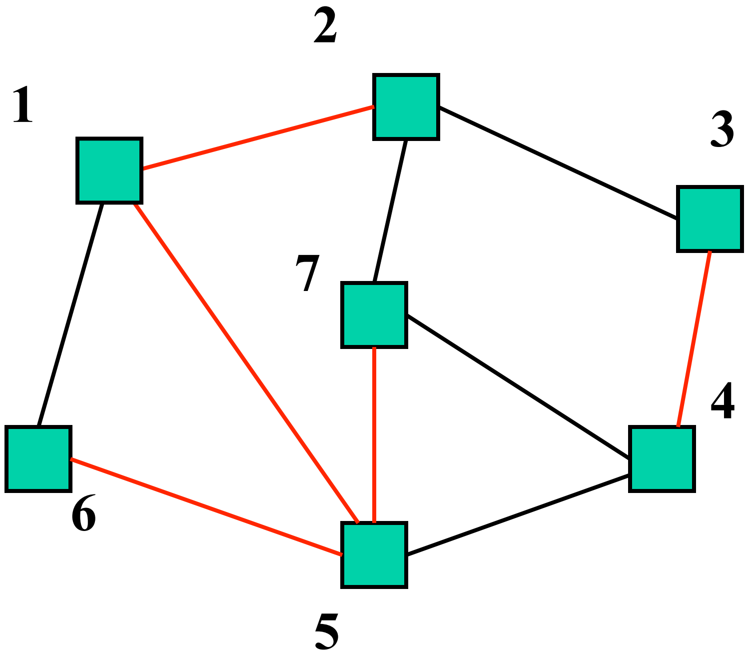


Output: (1,2), (3,4), (5,6), (5,7)

Example

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

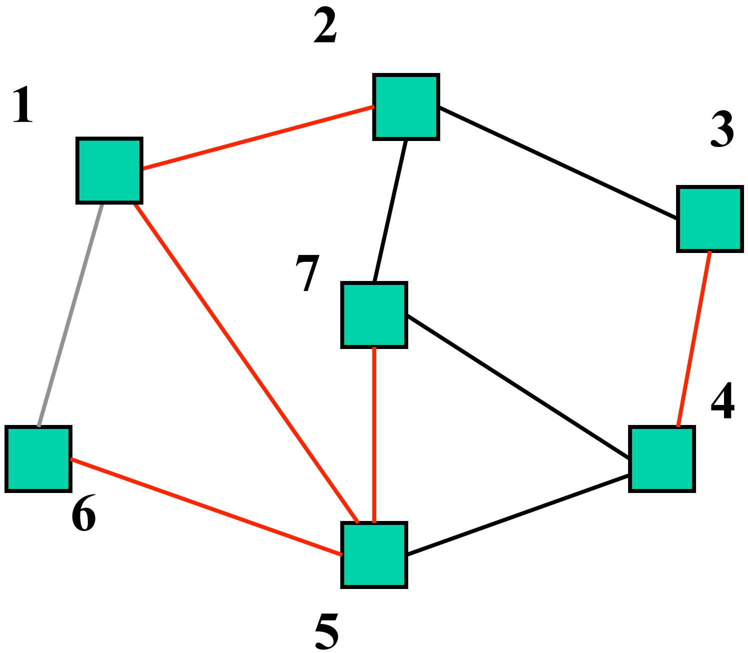


Output: (1,2), (3,4), (5,6), (5,7), (1,5)

Example

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

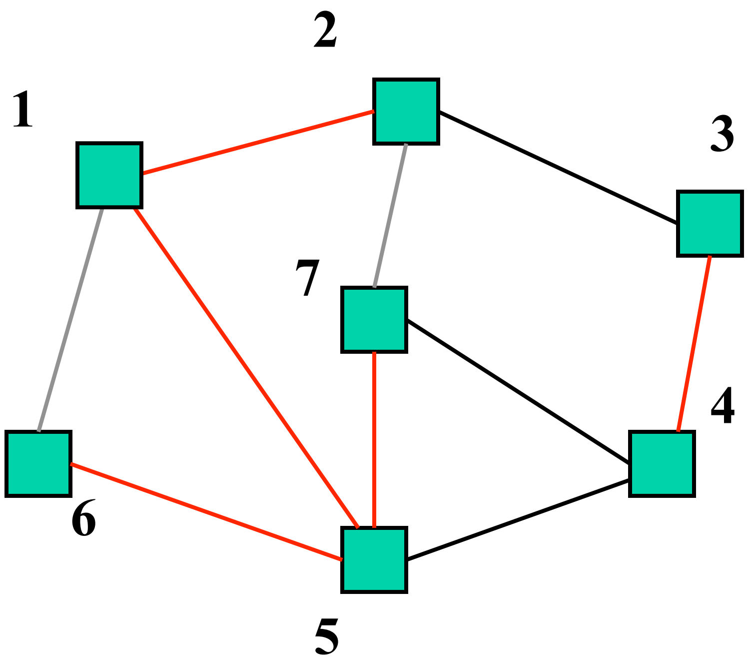


Output: (1,2), (3,4), (5,6), (5,7), (1,5)

Example

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

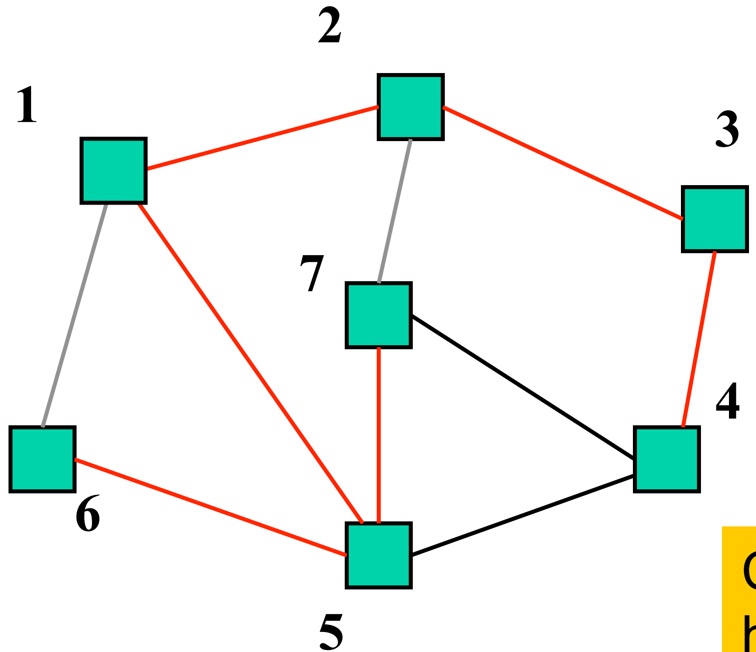


Output: (1,2), (3,4), (5,6), (5,7), (1,5)

Example

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)



Can stop once we have **|V|-1** edges

Output: (1,2), (3,4), (5,6), (5,7), (1,5), (2,3)

Cycle Detection

- To decide if an edge could form a cycle is *O*(**|V|**) because we may need to traverse all edges already in the output
- So overall algorithm would be *O*(**|V||E|**)
- But there is a faster way we know
- Use union-find!
  - Initially, each item is in its own 1-element set
  - Union sets when we add an edge that connects them
  - Stop when we have one set

Using Disjoint-Set

Can use a disjoint-set implementation in our spanning-tree algorithm to detect cycles:

Invariant: **`u`** and **`v`** are connected in output-so-far

iff

`u` and **`v`** in the same set

- Initially, each node is in its own set
- When processing edge **`(u,v)`**:
  - If **`find(u)`** equals **`find(v)`**, then do not add the edge
  - Else add the edge and **`union(find(u),find(v))`**
  - $O(|\mathbf{E}|)$ operations that are almost $O(1)$ amortized

Summary So Far

The spanning-tree problem

- Add nodes to partial tree approach is $O(|\mathbf{E}|)$
- Add acyclic edges approach is *almost* $O(|\mathbf{E}|)$
  - Using union-find "as a black box"

But really want to solve the minimum-spanning-tree problem

- Given a weighted undirected graph, give a spanning tree of minimum weight
- Same two approaches will work with minor modifications
- Both will be $O(|E| \log |V|)$

Minimum Spanning Tree Algorithms

Algorithm #1

Shortest-path is to Dijkstra's Algorithm

as

Minimum Spanning Tree is to Prim's Algorithm

(Both based on expanding cloud of known vertices, basically using a priority queue instead of a DFS stack)

Algorithm #2

Kruskal's Algorithm for Minimum Spanning Tree

is

Exactly our 2nd approach to spanning tree

but process edges in cost order