SPANNING TREES, INTRO. TO THREADS

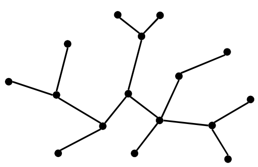
Lecture 23
CS2110 – Fall 2013

A lecture with two distinct parts

- Part I: Finishing our discussion of graphs
  - Today: Spanning trees
  - Definitions, algorithms (Prim's, Kruskal's)
  - Travelling salesman problem
- Part II: Introduction to the idea of threads
  - Why do we need them?
  - What is a thread?

Undirected Trees

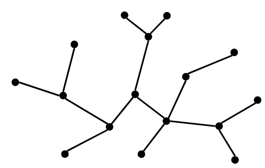
- An undirected graph is a *tree* if there is exactly one simple path between any pair of vertices



Facts About Trees

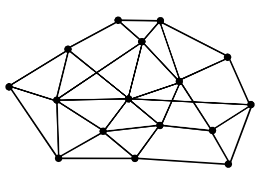
- |E| = |V| – 1
- connected
- no cycles

In fact, any two of these properties imply the third, and imply that the graph is a tree



Spanning Trees

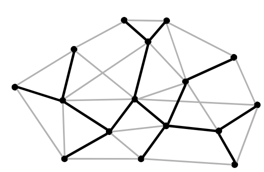
A *spanning tree* of a connected undirected graph (V,E) is a subgraph (V,E') that is a tree



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

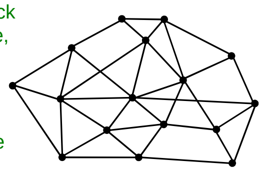


Finding a Spanning Tree

7

A subtractive method

- Start with the whole graph – it is connected
- If there is a cycle, pick an edge on the cycle, throw it out – the graph is still connected (why?)
- Repeat until no more cycles

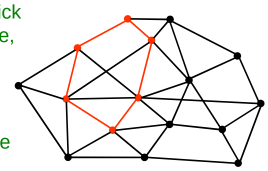


Finding a Spanning Tree

8

A subtractive method

- Start with the whole graph – it is connected
- If there is a cycle, pick an edge on the cycle, throw it out – the graph is still connected (why?)
- Repeat until no more cycles

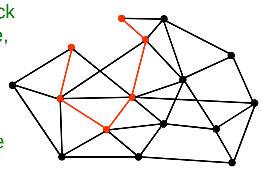


Finding a Spanning Tree

9

A subtractive method

- Start with the whole graph – it is connected
- If there is a cycle, pick an edge on the cycle, throw it out – the graph is still connected (why?)
- Repeat until no more cycles

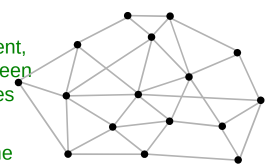


Finding a Spanning Tree

10

An additive method

- Start with no edges – there are no cycles
- If more than one connected component, insert an edge between them – still no cycles (why?)
- Repeat until only one component

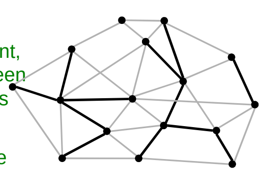


Finding a Spanning Tree

11

An additive method

- Start with no edges – there are no cycles
- If more than one connected component, insert an edge between them – still no cycles (why?)
- Repeat until only one component

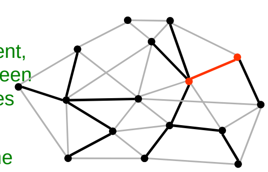


Finding a Spanning Tree

12

An additive method

- Start with no edges – there are no cycles
- If more than one connected component, insert an edge between them – still no cycles (why?)
- Repeat until only one component

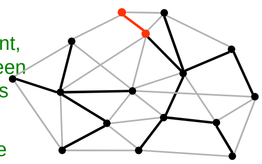


Finding a Spanning Tree

13

An additive method

- Start with no edges – there are no cycles
- If more than one connected component, insert an edge between them – still no cycles (why?)
- Repeat until only one component

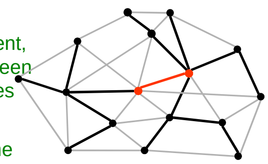


Finding a Spanning Tree

14

An additive method

- Start with no edges – there are no cycles
- If more than one connected component, insert an edge between them – still no cycles (why?)
- Repeat until only one component

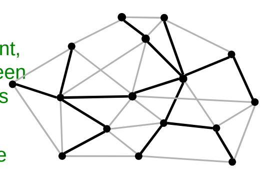


Finding a Spanning Tree

15

An additive method

- Start with no edges – there are no cycles
- If more than one connected component, insert an edge between them – still no cycles (why?)
- Repeat until only one component



Minimum Spanning Trees

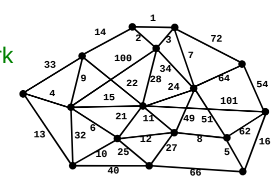
16

- Suppose edges are weighted, and we want a spanning tree of *minimum cost* (sum of edge weights)
- Some graphs have exactly one minimum spanning tree. Others have multiple trees with the same cost, any of which is a minimum spanning tree

Minimum Spanning Trees

17

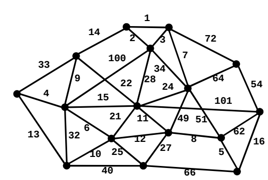
- Suppose edges are weighted, and we want a spanning tree of *minimum cost* (sum of edge weights)
- Useful in network routing & other applications
- For example, to stream a video



3 Greedy Algorithms

18

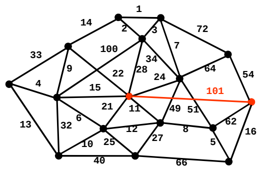
A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it



3 Greedy Algorithms

19

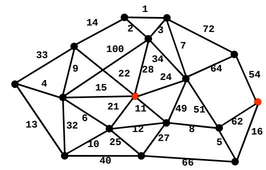
A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it



3 Greedy Algorithms

20

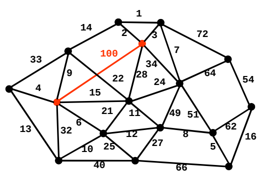
A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it



3 Greedy Algorithms

21

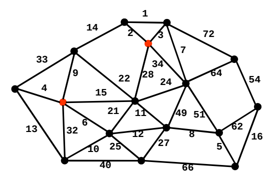
A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it



3 Greedy Algorithms

22

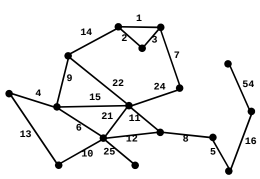
A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it



3 Greedy Algorithms

23

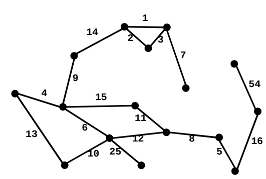
A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it



3 Greedy Algorithms

24

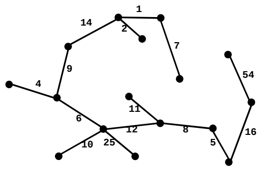
A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it



3 Greedy Algorithms

25

A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it

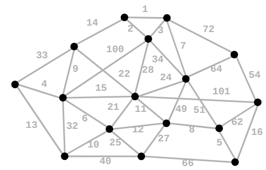


3 Greedy Algorithms

26

B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's algorithm

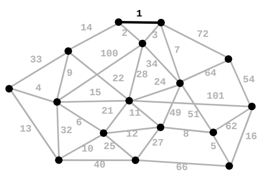


3 Greedy Algorithms

27

B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's algorithm

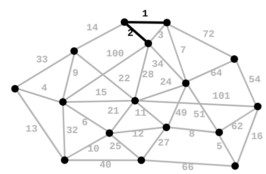


3 Greedy Algorithms

28

B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's algorithm

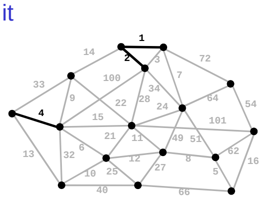


3 Greedy Algorithms

29

B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's algorithm

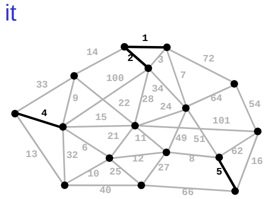


3 Greedy Algorithms

30

B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's algorithm

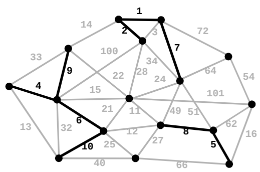


3 Greedy Algorithms

31

B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's algorithm

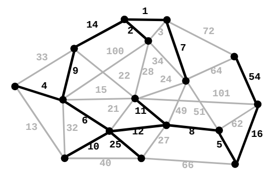


3 Greedy Algorithms

32

B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's algorithm

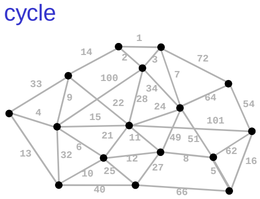


3 Greedy Algorithms

33

C. Start with any vertex, add min weight edge extending that connected component that does not form a cycle

Prim's algorithm
(reminiscent of Dijkstra's algorithm)

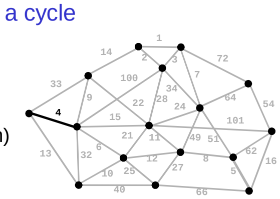


3 Greedy Algorithms

34

C. Start with any vertex, add min weight edge extending that connected component that does not form a cycle

Prim's algorithm
(reminiscent of Dijkstra's algorithm)

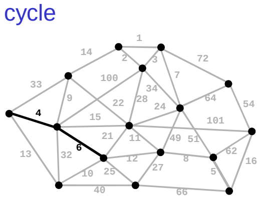


3 Greedy Algorithms

35

C. Start with any vertex, add min weight edge extending that connected component that does not form a cycle

Prim's algorithm
(reminiscent of Dijkstra's algorithm)

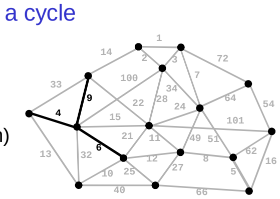


3 Greedy Algorithms

36

C. Start with any vertex, add min weight edge extending that connected component that does not form a cycle

Prim's algorithm
(reminiscent of Dijkstra's algorithm)

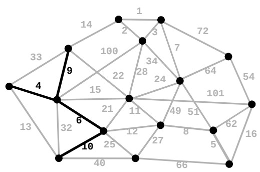


3 Greedy Algorithms

37

C. Start with any vertex, add min weight edge extending that connected component that does not form a cycle

Prim's algorithm
(reminiscent of Dijkstra's algorithm)

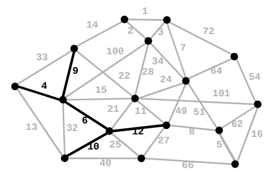


3 Greedy Algorithms

38

C. Start with any vertex, add min weight edge extending that connected component that does not form a cycle

Prim's algorithm
(reminiscent of Dijkstra's algorithm)

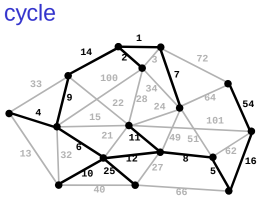


3 Greedy Algorithms

39

C. Start with any vertex, add min weight edge extending that connected component that does not form a cycle

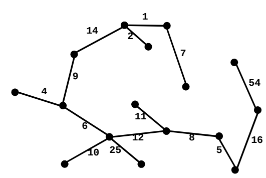
Prim's algorithm
(reminiscent of Dijkstra's algorithm)



3 Greedy Algorithms

40

- When edge weights are all distinct, or if there is exactly one minimum spanning tree, the 3 algorithms all find the identical tree



Prim's Algorithm

41

```
prim(s) {
   D[s] = 0; mark s; //start vertex
   while (some vertices are unmarked) {
      v = unmarked vertex with smallest D;
      mark v;
      for (each w adj to v) {
         D[w] = min(D[w], c(v,w));
      }
   }
}
```

- $O(n^2)$ for adj matrix
  - While-loop is executed n times
  - For-loop takes O(n) time

- O(m + n log n) for adj list
  - Use a PQ
  - Regular PQ produces time O(n + m log m)
  - Can improve to O(m + n log n) using a fancier heap

Greedy Algorithms

42

- These are examples of Greedy Algorithms
- The Greedy Strategy is an algorithm design technique
  - Like Divide & Conquer
- Greedy algorithms are used to solve optimization problems
  - The goal is to find the *best* solution
- Works when the problem has the greedy-choice property
  - A global optimum can be reached by making locally optimum choices

- Example: the Change Making Problem: Given an amount of money, find the smallest number of coins to make that amount
- Solution: Use a Greedy Algorithm
  - Give as many large coins as you can
- This greedy strategy produces the optimum number of coins for the US coin system
- Different money system ⇒greedy strategy may fail
  - Example: old UK system

Similar Code Structures

43

```
while (some vertices are
      unmarked) {
   v = best of unmarked
       vertices;
   mark v;
   for (each w adj to v)
        update 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))

here c(v,w) is the v→w edge weight

Traveling Salesman Problem

44

- 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?
  - Basically what we want the butterfly to do in A6! But we don't mind if the butterfly revisits a city (Tile), or doesn't use the very shortest possible path.
  - The true TSP is very hard (NP complete)… for this we want the *perfect* answer in all cases, and can't revisit.
  - 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…

45

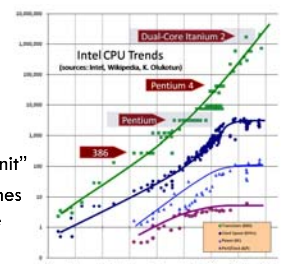
THREADS: WHO NEEDS 'EM?

Introduction to the concept…

The Multicore Trend

46

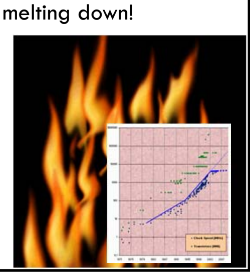
- Moore's Law: Computer speeds and memory densities nearly double each year
- But we no longer are getting this speed purely by running a faster CPU clock
  - CPU = "central processor unit"
  - CPU clock roughly determines instructions / second for the computer



Issue: A fast computer runs hot

47

- Power dissipation rises as the *square of the CPU clock rate*
- Chips were heading towards melting down!
- Multicore: with four CPUs (cores) on one chip, even if we run each at half speed we get more overall performance!



How a computer works

48

- Your program translates to machine instructions
- CPU has a pointer into the code: Program Counter
  - To execute an instruction, it fetches what the PC points to, decodes it, fetches the arguments, and performs the required action (such as add two numbers, then store at some location)
  - We call this a "thread of execution" or a "context of execution"
- One CPU == 1 thread, right? Well, not really….

Each program has its own thread!

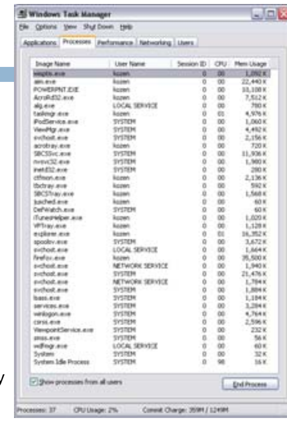
49

- Earliest days: shared one CPU among many programs by just having it run a few instructions each, "round robin"
  - Program A gets to run 10,000 instructions
  - Then pause A, "context switch" to B, run 10,000 of B
  - Then pause B, context switch to C, run 10,000 for C…
- This makes one CPU seem like N (slower) CPUs
- With the new trend toward multicore we can have a lot of threads all concurrently active

Keeping those cores busy

50

- The operating system provides support for multiple "processes"
- In reality there there may be fewer processors than processes
- Processes are an illusion – at the hardware level, lots of multitasking
  - memory subsystem
  - video controller
  - buses
  - instruction prefetching
- Virtualization can even let one machine create the illusion of many machines (they share disks, etc)



How is a Thread defined?

51

- *A separate "execution" <u>that runs within a single program</u> and can perform a computational task independently and concurrently with other threads*
- Many applications do their work in just a single thread: the one that called main() at startup
  - But there may still be extra threads...
  - ... Garbage collection runs in a "background" thread
  - GUIs have a separate thread that listens for events and "dispatches" upcalls
- Today: learn to create new threads of our own

What <u>*is*</u> a Thread in Java?

52

- A thread is a kind of object that "independently computes"
  - Has an associated stack and local variables (context)
  - Needs to be created, like any object
  - Then "started". This causes some method (like main()) to be invoked. It runs side by side with other thread in the same program and they see the same global data
- The actual execution could occur on distinct CPU cores, but Java could also simulate multiple cores. You can't really tell which approach Java is using

Concurrency

53

- *Concurrency* refers to a single program in which several threads are running simultaneously
  - Special problems arise: These threads literally access the same shared memory regions at the same time!
  - They are at risk of interfering with each other, e.g. if one thread is modifying a complex structure like a heap while another is trying to read it
- In cs2110 we focus on simple ways to use this model without bugs introduced by interference