CSE373: Data Structures & Algorithms

Lecture 18: Network Flow, NP-Completeness, and More

Aaron Bauer

Winter 2014

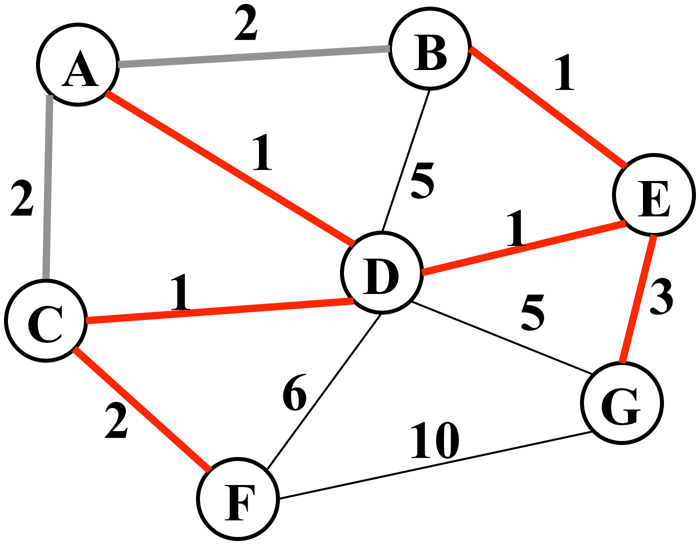
Pseudocode for Kruskal's

1. Sort edges by weight (better: put in min-heap)
2. Each node in its own set
3. While output size **< |V|-1**
   - Consider next smallest edge **`(u,v)`**
   - if **`find(u)`** and **`find(v)`** indicate **`u`** and **`v`** are in different sets
     - output **`(u,v)`**
     - **`union(find(u),find(v))`**

Recall invariant:

`u` and **`v`** in same set if and only if connected in output-so-far

Example



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

6: (D,F)

10: (F,G)

Output: (A,D), (C,D), (B,E), (D,E), (C,F), (E,G)

Note: At each step, the union/find sets are the trees in the forest

Correctness

Kruskal's algorithm is clever, simple, and efficient

- But does it generate a minimum spanning tree?
- How can we prove it?

First: it generates a spanning tree

- Intuition: Graph started connected and we added every edge that did not create a cycle
- Proof by contradiction: Suppose `u` and `v` are disconnected in Kruskal's result. Then there's a path from `u` to `v` in the initial graph with an edge we could add without creating a cycle. But Kruskal would have added that edge. Contradiction.

Second: There is no spanning tree with lower total cost…

The inductive proof set-up

Let **F** (stands for "forest") be the set of edges Kruskal's has added at some point during its execution.

Claim: **F** is a subset of *one or more* MSTs for the graph

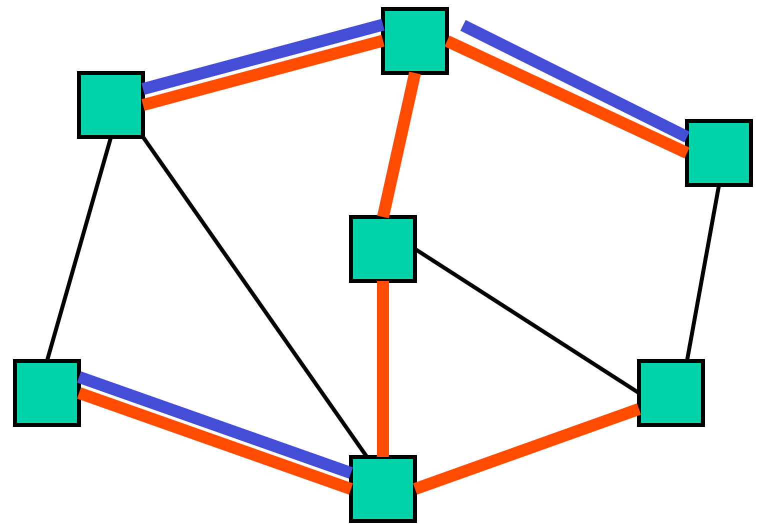
– Therefore, once **|F|=|V|-1**, we have an MST

Proof: By induction on **|F|**

Base case: **|F|=0**: The empty set is a subset of all MSTs

Inductive case: **|F|=k+1**: By induction, before adding the $(k+1)^{th}$ edge (call it **e**), there was some MST **T** such that **F-{e}** $\subseteq$ **T** …

*Staying a subset of **some** MST*



Claim: **F** is a subset of *one or more* MSTs for the graph

So far: **F-{e}** ⊆ **T**:

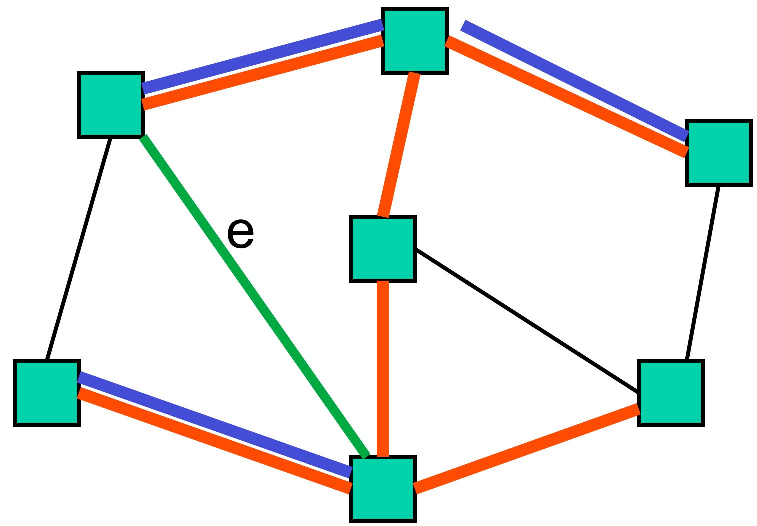
Two disjoint cases:

- If **{e}** ⊆ **T**: Then **F** ⊆ **T** and we're done
- Else **e** forms a cycle with some simple path (call it **p**) in **T**
  - Must be since **T** is a spanning tree

Staying a subset of ***some*** *MST*

Claim: **F** is a subset of *one or more* MSTs for the graph

So far: **F-{e}** $\subseteq$ **T** and
e forms a cycle with **p** $\subseteq$ **T**



- There must be an edge **e2** on **p** such that **e2** is not in **F**
  - Else Kruskal would not have added **e**
- Claim: **e2.weight == e.weight**

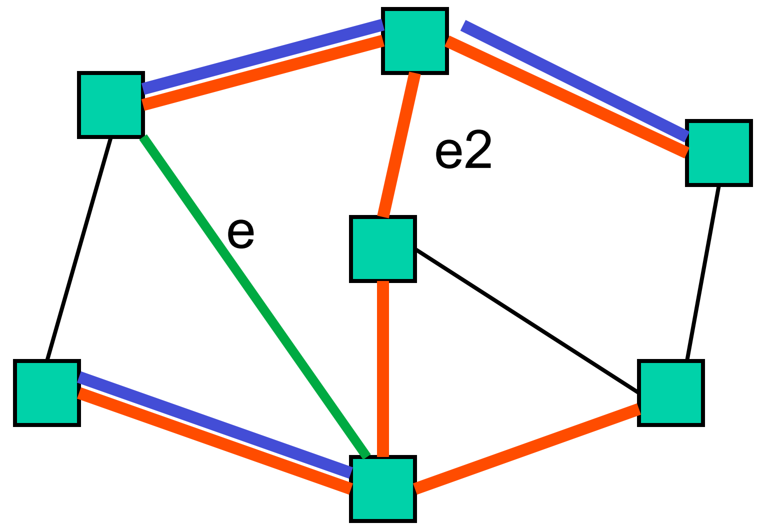
*Staying a subset of **some** MST*

Claim: **F** is a subset of *one or more* MSTs for the graph

So far: **F-{e}** ⊆ **T**

e forms a cycle with **p** ⊆ **T**

e2 on **p** is not in **F**



- Claim: **e2.weight == e.weight**
  - If **e2.weight > e.weight**, then **T** is not an MST because **T-{e2}+{e}** is a spanning tree with lower cost: contradiction
  - If **e2.weight < e.weight**, then Kruskal would have already considered **e2**. It would have added it since **T** has no cycles and **F-{e}** ⊆ **T.** But **e2** is not in **F**: contradiction

*Staying a subset of **some** MST*

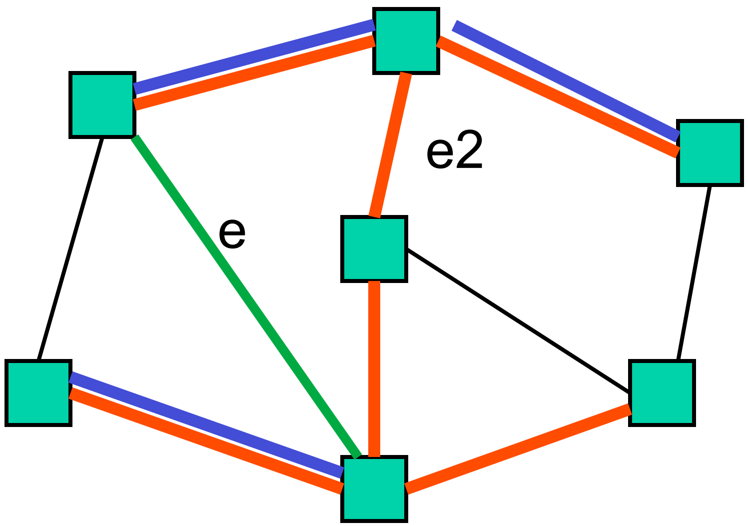
Claim: **F** is a subset of *one or more* MSTs for the graph

So far: **F-{e}** ⊆ **T**

e forms a cycle with **p** ⊆ **T**

e2 on **p** is not in **F**

e2.weight == e.weight



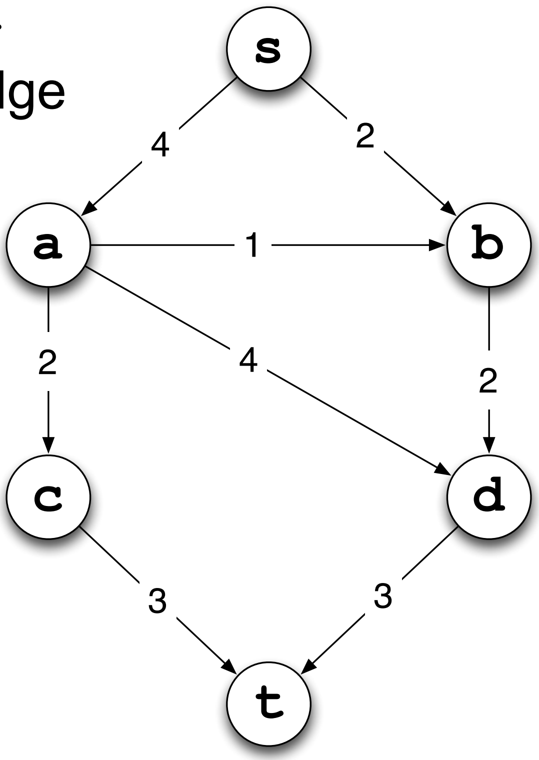
- Claim: **T-{e2}+{e}** is an MST
  - It is a spanning tree because **p-{e2}+{e}** connects the same nodes as **p**
  - It is minimal because its cost equals cost of **T**, an MST
- Since **F** ⊆ **T-{e2}+{e}**, **F** is a subset of one or more MSTs

Done

Network Flow

- A directed graph G= (V,E) with capacities on the edges
  - c(u,v) is the capacity of edge (u,v)
  - Capacities could represent amount of water, traffic, etc.
- "Flow" passes through the graph from **`s`** to **`t`**
  - The maximum that can pass along an edge is its capacity
  - Flow must be conserved (same amount must leave a node that enters it)
- The Maximum Flow Problem

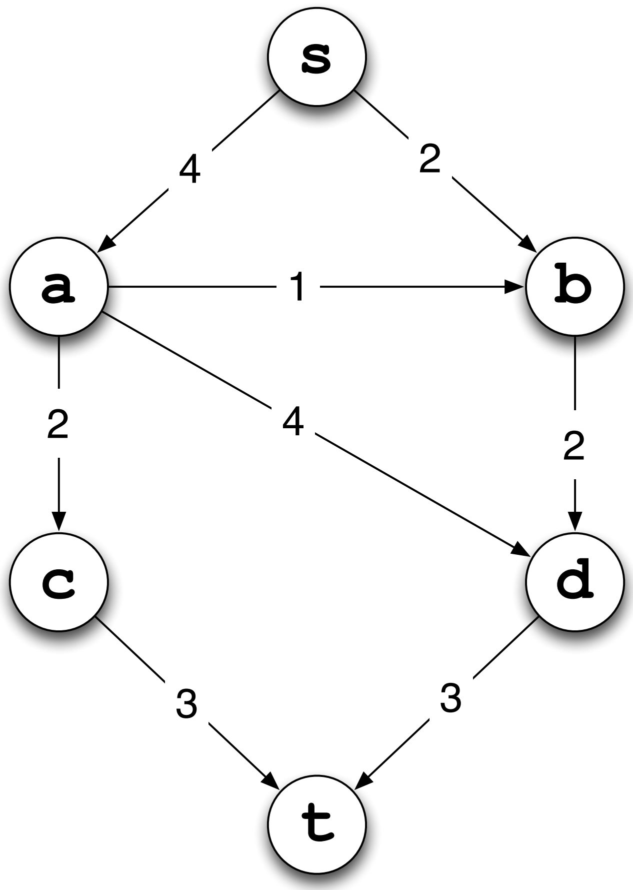
  Determine the maximum flow that can pass from **`s`** to **`t`**



Motivation

- Many networks have “flow” going across them
  - Water
  - Electricity
  - Transportation
  - …
- Energy and Nutrients flow between organisms
- Related problems:
  - Multi-commodity flow
  - Minimum cost flow
  - Circulation

Will Greedy Work?

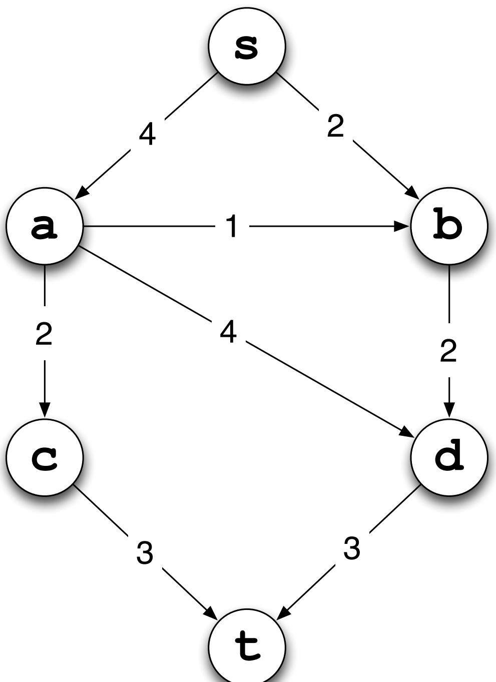


No!

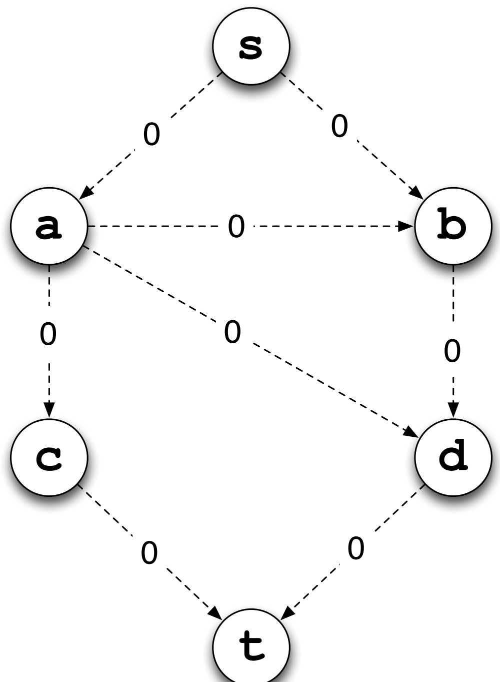
Ford-Fulkerson: Idea

- Repeatedly identify paths from `s` to `t`
  - Called augmenting paths
- Send as much flow as possible down the path
- Stop when there are no more paths to be found
- Amount of flow entering `t` is the maximum flow
- We will need to construct two additional graphs F and R
  - F will represent the current flow (initially 0)
  - R (called the residual graph) will show, for each edge, how much more flow can be added
    - Calculated by subtracting current flow from capacity
    - Edges called residual edges

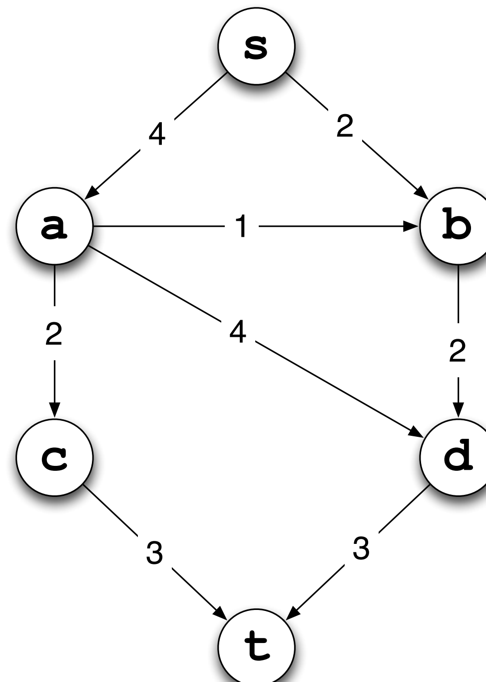
Setup



G

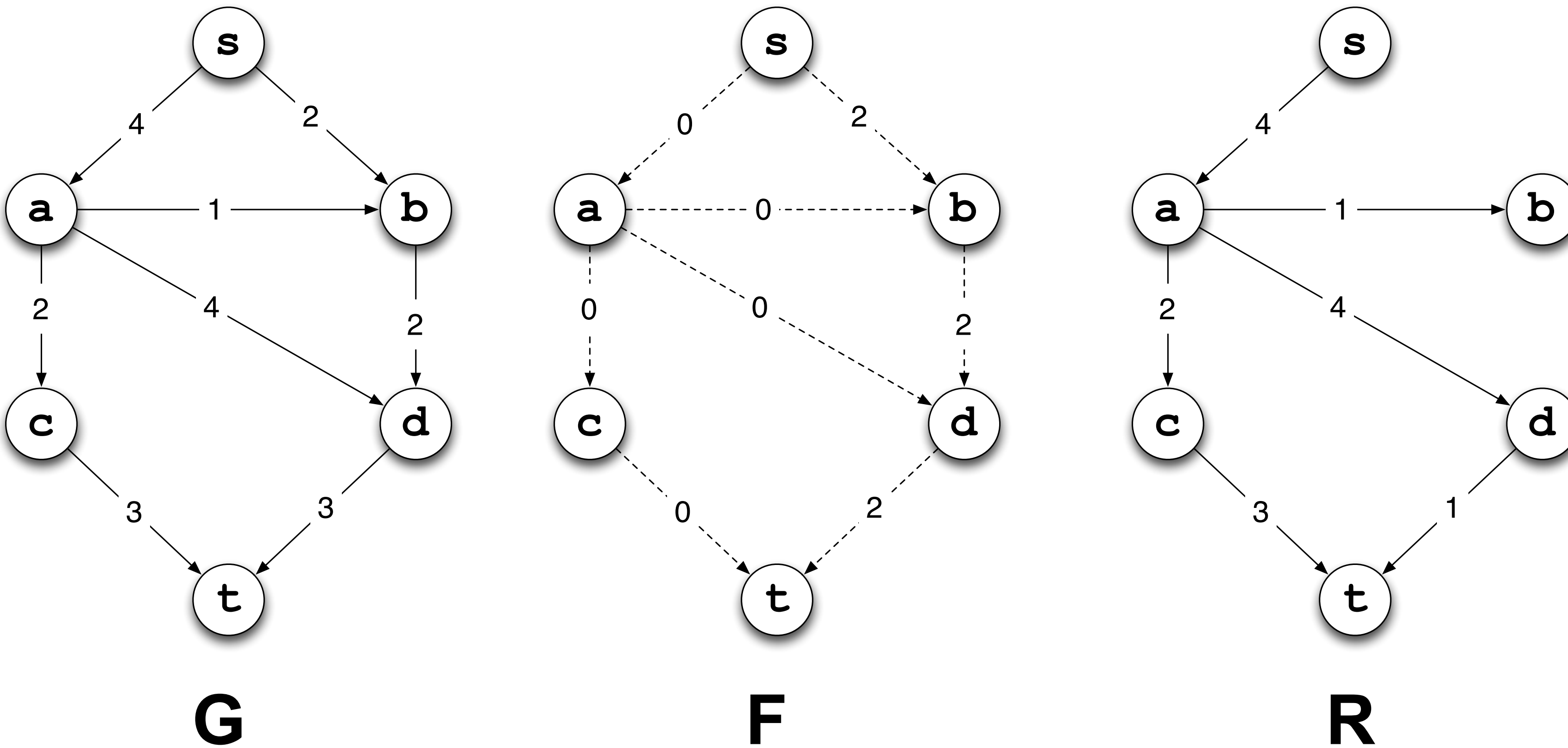


F

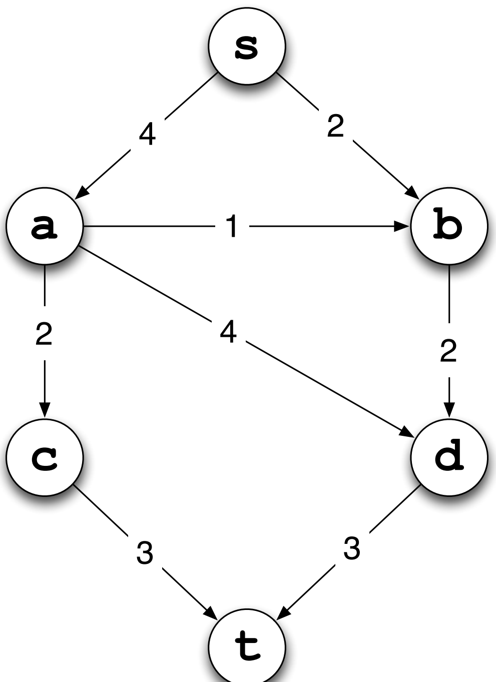


R

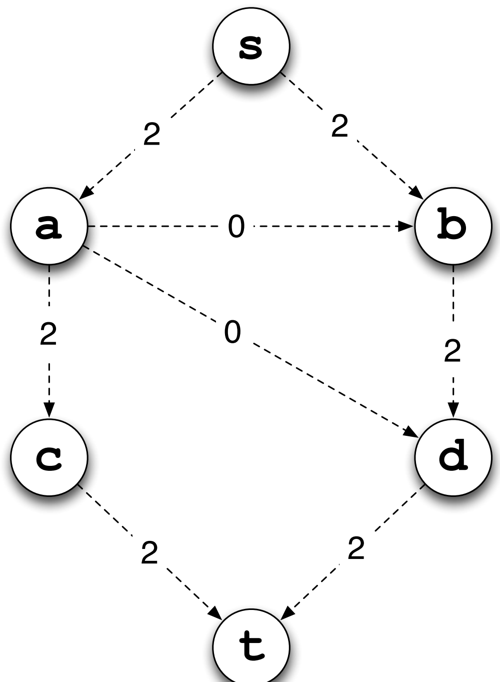
Example 1



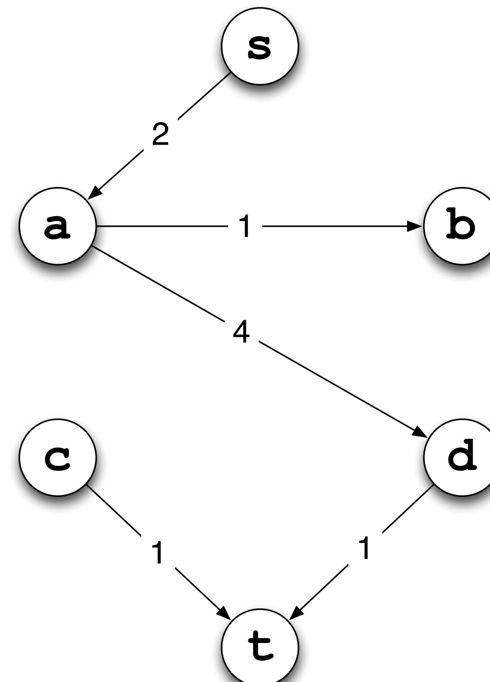
Example 1



G

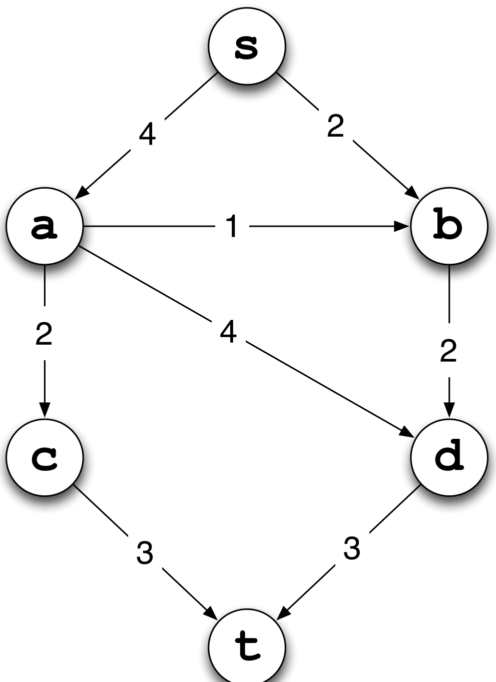


F

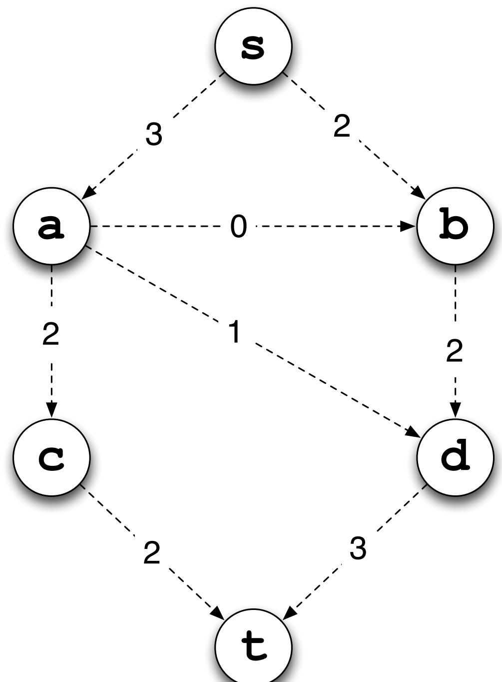


R

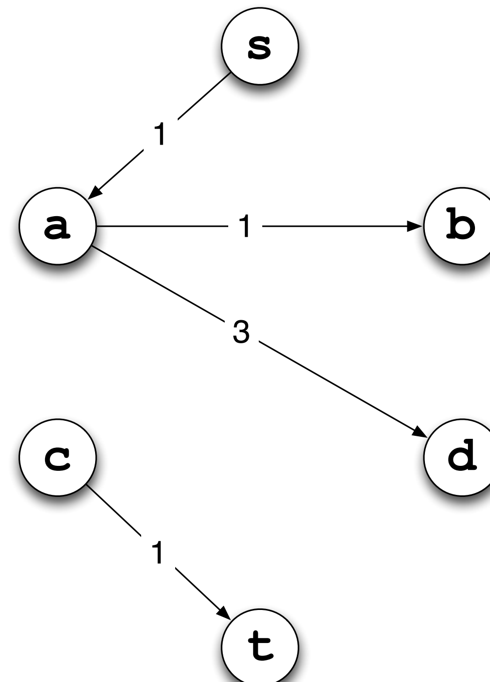
Example 1



G

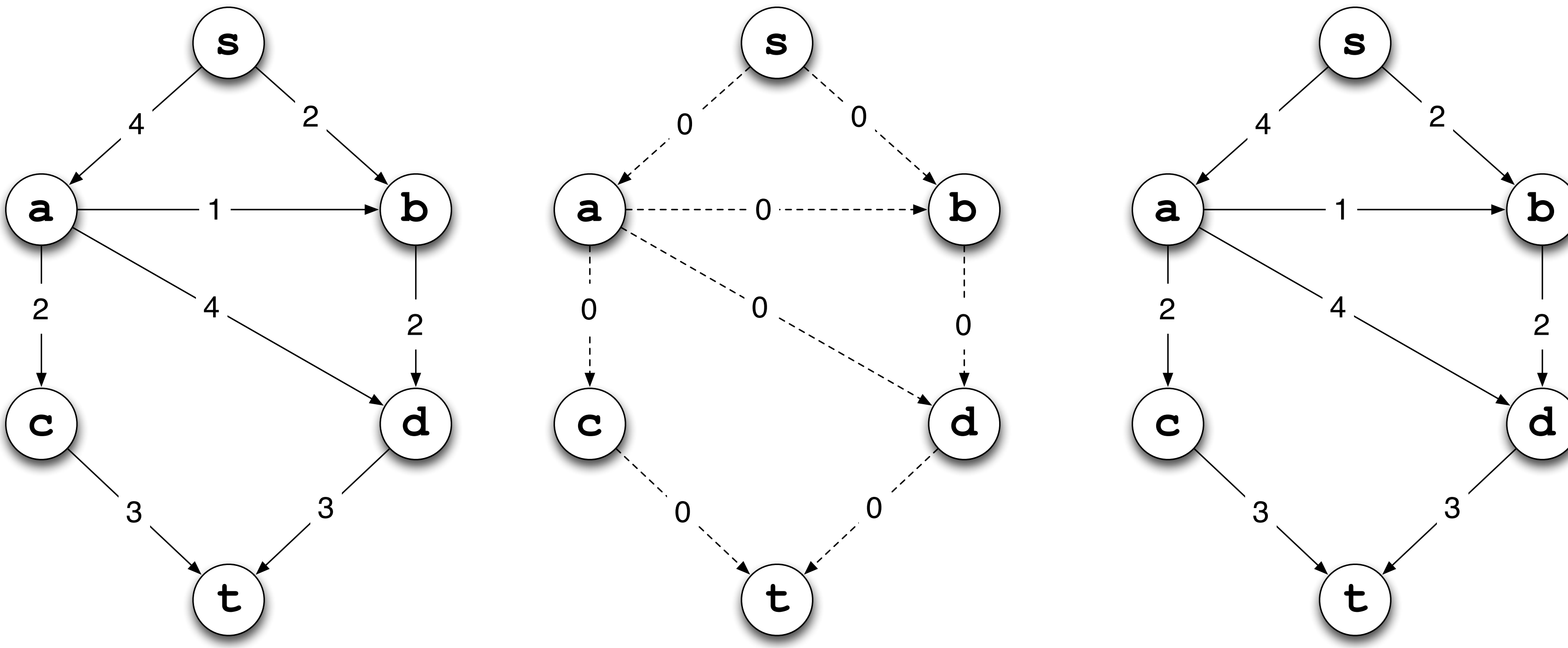


F



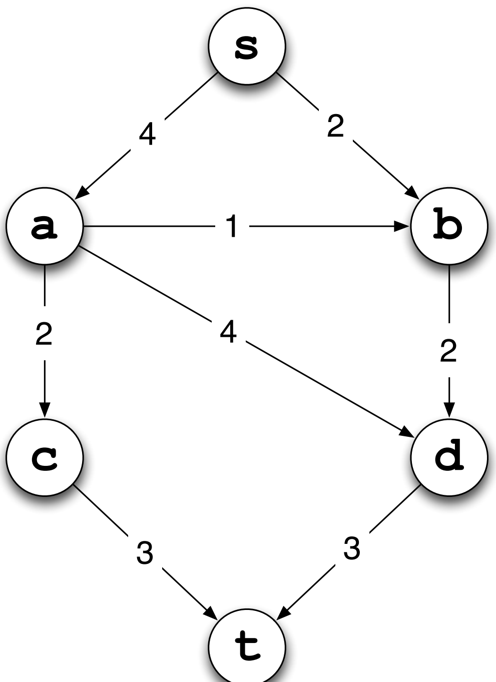
R

Example 2

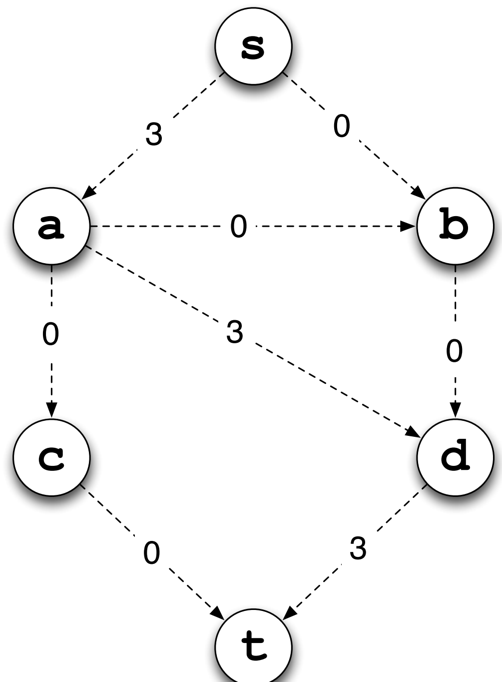


G **F** **R**

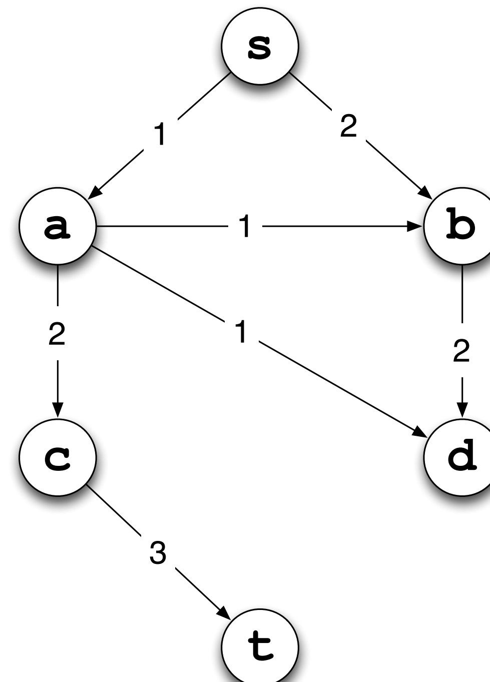
Example 2



G

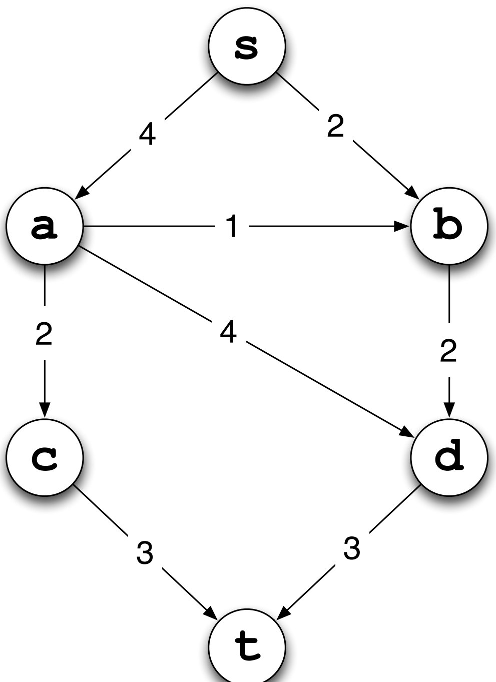


F

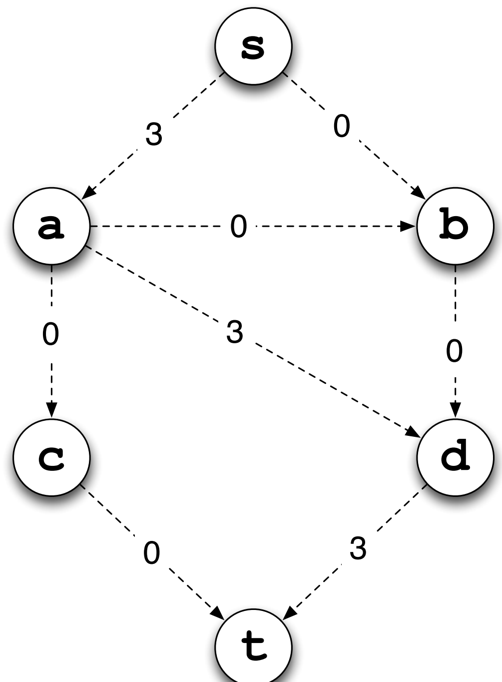


R

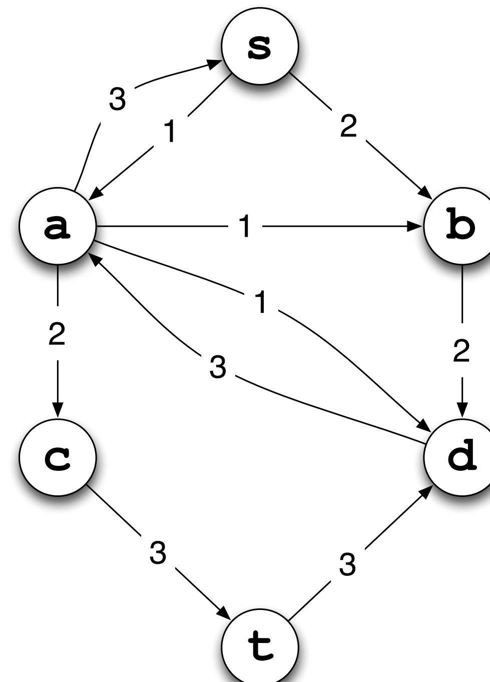
Let the Algorithm Change Its Mind



G

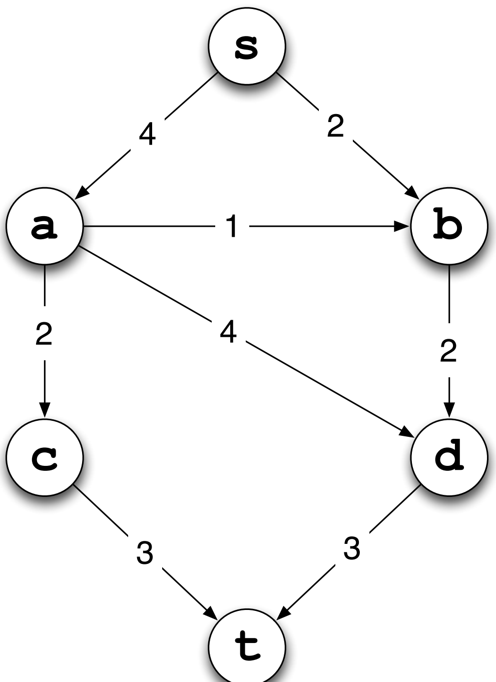


F

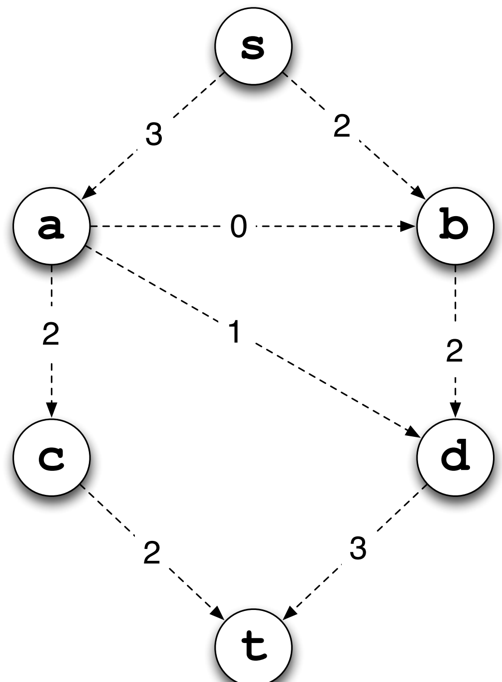


R

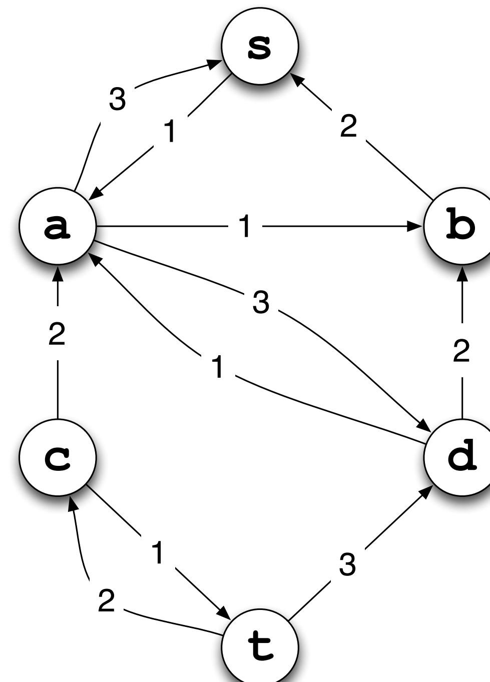
Let the Algorithm Change Its Mind



G



F



R

Correctness

- Termination
  - As long as the edge capacities are integers the algorithm will terminate
  - Each augmenting path increases the flow by at least 1
- Since we continue until the residual graph has no s-t paths remaining, max flow is guaranteed to be found

Complexity

- An augmenting path can be found in O(|E|) by the unweighted shortest path algorithm
- Each augmenting path increases the flow by at least 1
- Hence, in the worst case, for a max flow of *f*, the worst-case asymptotic running time is O(f*|E|)
  - A variation on Dijkstra's algorithm to choose the largest capacity augmenting path can improve this

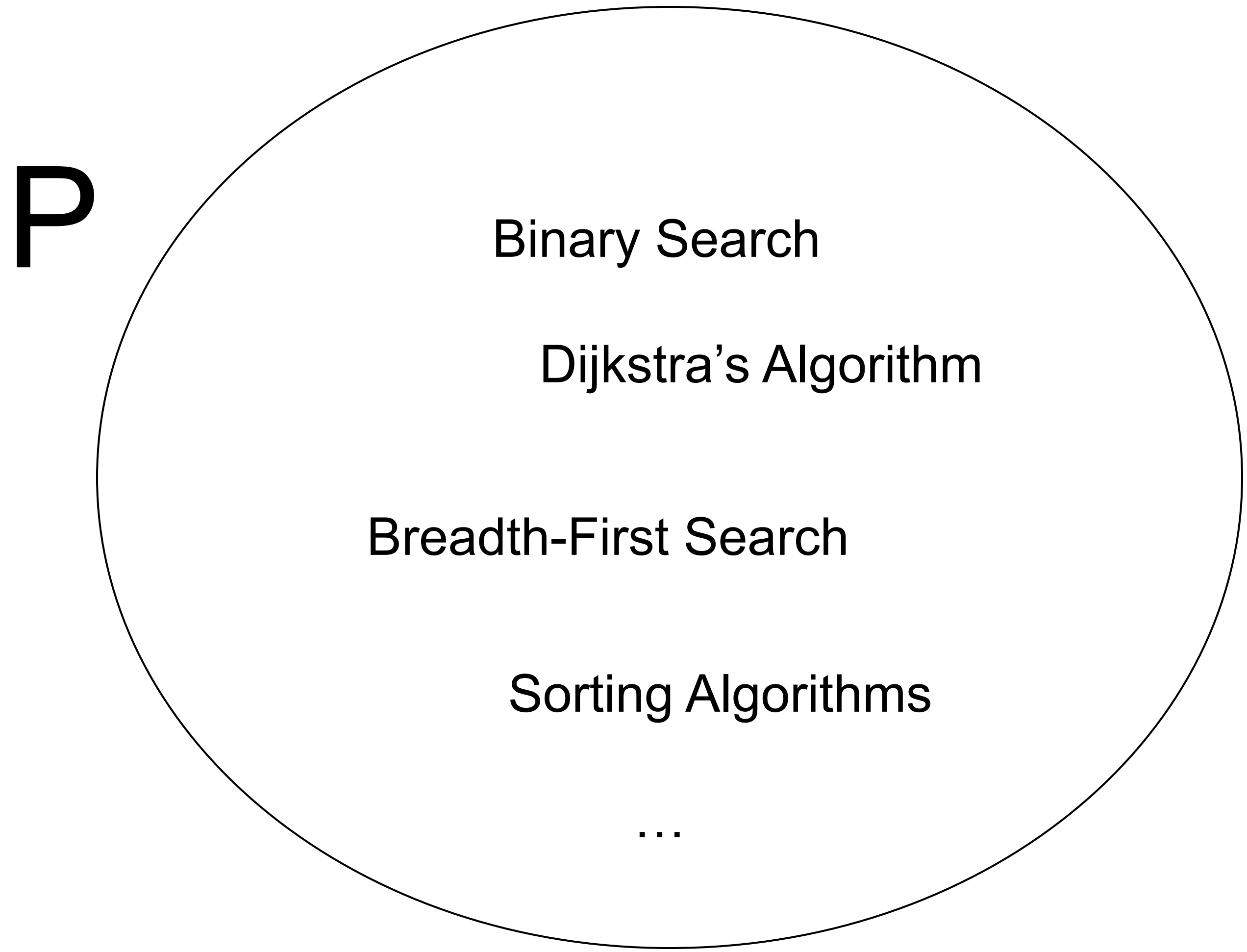
Timing

- Prefer timing a sequence of instructions
- Prefer large and spread out values of *n*
- Beware of initial timings
- When timing sequence
  - For *O(log n)* operations
    a sequence of *m* take *O(m*log n)*
  - Divide by *m* to get per-instruction time

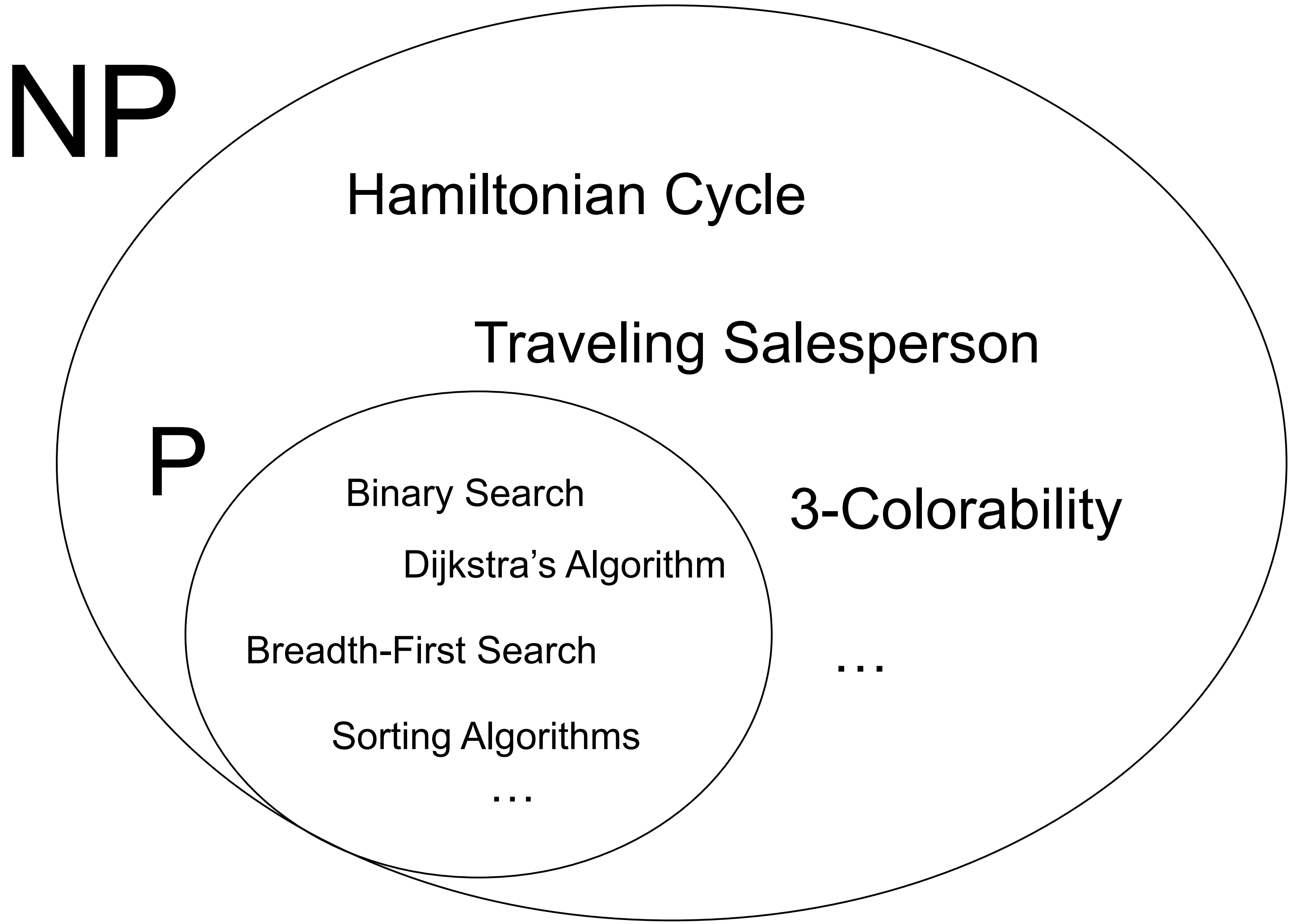
Some problems are harder than others

- Euler circuit (path touching every edge once)
  - linear time
- Hamiltonian cycle (simple cycle containing every vertex)
  - no known linear time algorithm
- Single-source unweighted shortest path
  - BFS solves it in linear time
- Single-source unweighted **longest** path
  - no known linear algorithm
- In fact, no known polynomial algorithms for variants
  - best known algorithms are exponential in worst case
  - belong to a class of problems called NP-complete

Polynomial Time



Nondeterministic Polynomial Time

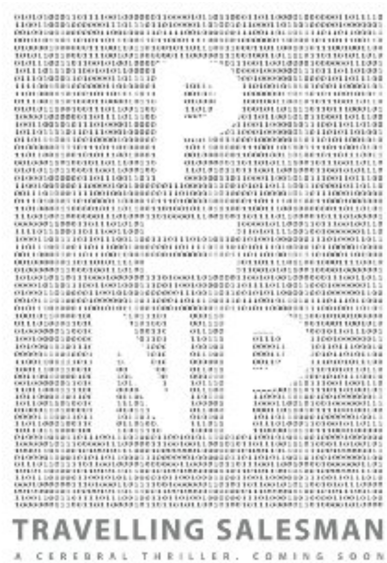


What does NP mean?

- Any problem "in NP" can be solved in polynomial time by a nondeterministic algorithm
  - A deterministic algorithm must choose one path when presented with a choice
  - A nondeterministic algorithm can choose multiple paths
- Any problem "in NP" is one whose solution is *verifiable* in polynomial time
  - If the solution to a problem is fast to verify, we can nondeterministically try all possible solutions quickly
- A problem is NP-complete if it's as hard to solve as any other problem in NP

P vs NP

- It’s currently unknown whether there exist polynomial time algorithms for NP-complete problems
  - That is, does P = NP?
  - People generally believe P ≠ NP, but no proof yet
- One of the major open questions in computer science
- Important enough to make its way into popular culture
  - Travelling Salesman (2012 film)
  - Episode of Elementary (CBS)



Some problems are impossible

- Why doesn't the Java compiler have an infinite loop checker?
  - It would be very useful
  - Industry would definitely pay for it
- Let's say we create such a program and call it H
  - H takes a program P and some input x
  - H(P,x) returns true if P(x) returns true
  - H(P,x) returns false if P(x) does not return true
- Now we create a program D that uses H as a subroutine
  - D takes a program P and returns the opposite of H(P,P)
  - D(P) returns true if P(P) does not return true
  - D(P) returns false if P(P) returns true

Halting Problem

- What happens if we run D on itself?
  - D(D) returns true if D(D) does not return true
  - D(D) returns false if D(D) returns true
  - Contradiction!
- It turns out a program such as H is not possible :(
- Known as the Halting Problem
  - One example of an undecidable problem
- Classic part of CS theory
  - Originally proved by Alan Turing

Algorithm Design Techniques

- Greedy
  - Shortest path, minimum spanning tree, …
- Divide and Conquer
  - Divide the problem into smaller subproblems, solve them, and combine into the overall solution
  - Often done recursively
  - We'll see examples when we get to sorting
- Dynamic Programming
  - Brute force through all possible solutions, storing solutions to subproblems to avoid repeat computation
- Backtracking
  - A clever form of exhaustive search

Dynamic Programming: Idea

- Divide problem into many subproblems
- An individual subproblem may occur many times
  - Store the result in a table to enable reuse
  - Technique called memoization
- Dijkstra's does this!
  - Breaks the problem of finding all shortest paths into subproblems of finding paths to increasingly distant nodes
  - It finds the shortest path to some intermediate node **`v`**
  - Stores this path for use in computing other shortest paths
- If the number of subproblems grows exponentially, a recursive solution may have an exponential running time
  - We can use dynamic programming to help with this

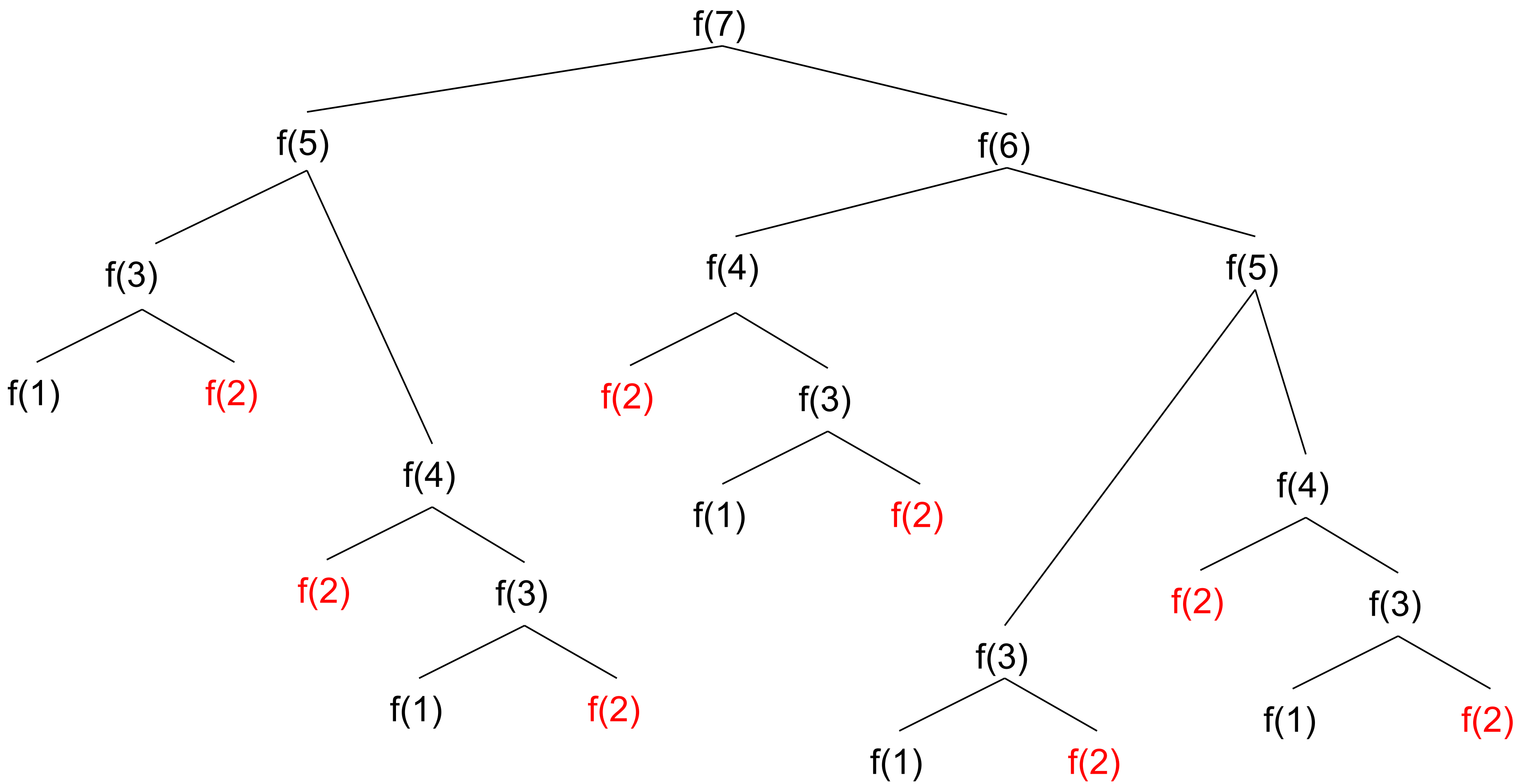
Fibonacci Sequence: Recursive

- Fibonacci sequence
  - 1, 1, 2, 3, 5, 8, 13, …
- Recursive solution:

```
fib(int n) {
  if (n == 1 || n == 2) {
    return 1
  }
  return fib(n - 2) + fib(n - 1)
}
```

- Exponential running time!
  - A lot of repeated computation

Repeated computation



Fibonacci Sequence: memoized

```
fib(int n) {
  Map results = new Map()
  results.put(1, 1)
  results.put(2, 1)
  return fibHelper(n, results)
}
fibHelper(int n, Map results) {
  if (!results.contains(n)) {
    results.put(n, fibHelper(n-2)+fibHelper(n-1))
  }
  return results.get(n)
}
```

Now each call of **`fib(x)`** only gets computed once for each x!

Spellcheck

- When your spellchecker suggests a word, how does it know what word to suggest?
  - May involve statistics about word frequency, context, etc.
  - Almost certainly includes edit distance
- Edit distance is the number of “edits” it takes to turn a word w1 into a word w2
  - Edits are insertions, deletions, and substitutions

Randomized Algorithms

- Randomized algorithms (or data structures) rely on some source of randomness
  - Usually a random number generator (RNG)
- True randomness is impossible on a computer
  - We will make do with pseudorandom numbers
- Suppose we only need to flip a coin
  - Can we use the lowest it on the system clock?
  - Does not work well for a sequence of numbers
- Simple method: linear congruential generator
  - Generate a pseudorandom sequence $x_1, x_2, \ldots$ with

$$x_{i+1} = Ax_i \bmod M$$

Linear Congruential Generator

$$x_{i+1} = Ax_i \bmod M$$

- Very sensitive to the choice of A and M
  - Also need to choose x_0 (“the seed”)
- For M = 11, A = 7, and x_0 = 1, we get

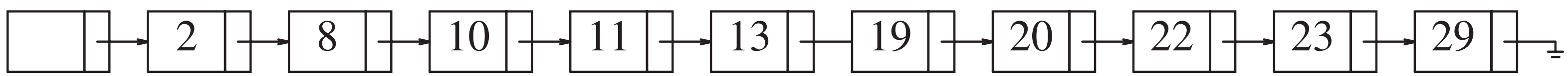
  7,5,2,3,10,4,6,9,8,1,7,5,2,...
- Sequence has a period of M – 1
- Choice of M and A should work to maximize the period
- The Java library’s Random uses a slight variation

$$x_{i+1} = (Ax_i + C) \bmod 2^B$$

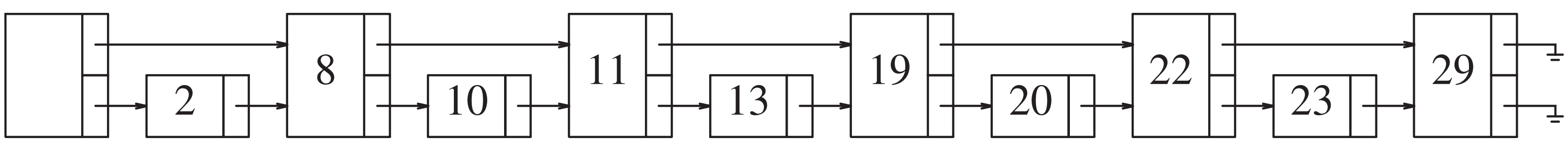
- Using A = 25,214,903,917, C = 13, and B = 48
  - Returns only the high 32 bits

Making sorted linked list better

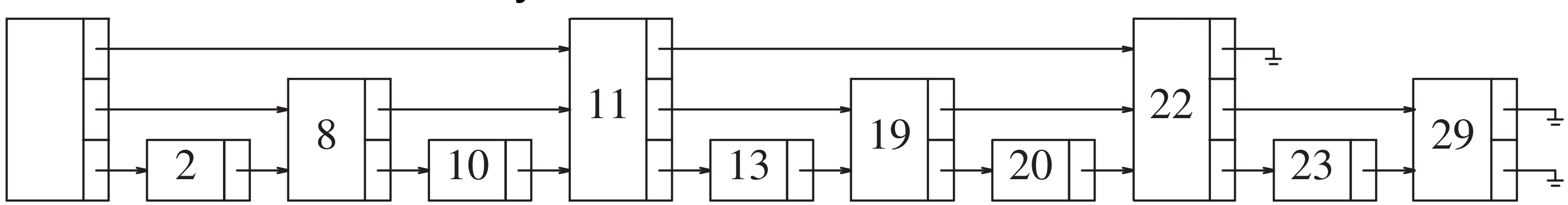
- We can search a sorted array in *O(log n)* using binary search
- But no such luck for a sorted linked list



- We could, however, add additional links
  - Every other node links to the node two ahead of it

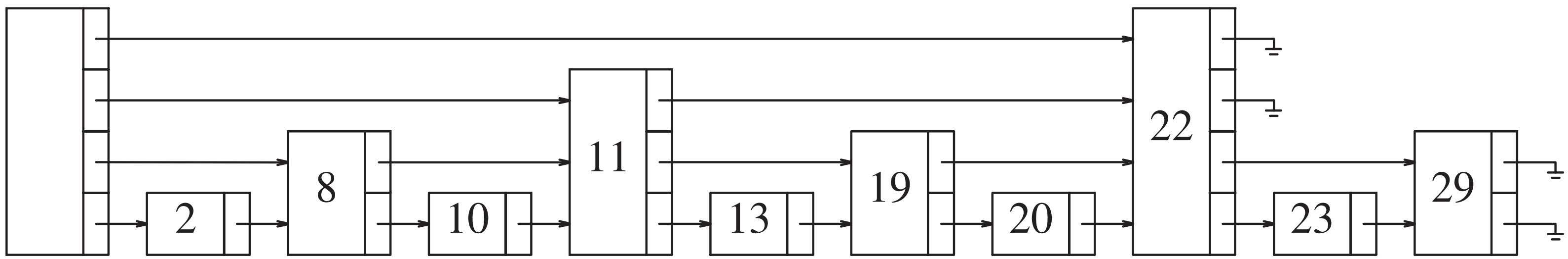


  - Go further: every fourth node links to the node four ahead



To the Logical Conclusion

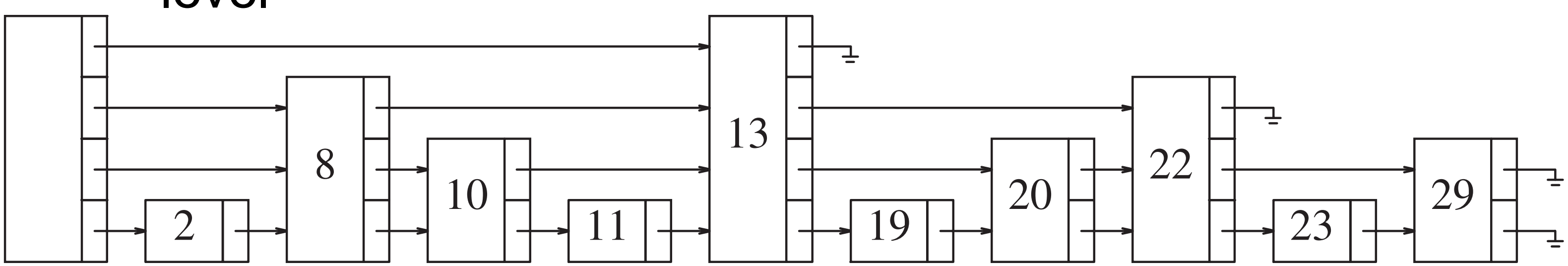
- Take this idea to the logical conclusion
  - Every 2^i th node links to the node 2^i ahead of it



  - Number of links doubles, but now only *log n* nodes are visited in a search!
  - Problem: insert may require completely redoing links
- Define a *level k node* as a node with *k* links
  - We require that the *i*th link in any level *k* node links to the next node with at least *i* levels

Skip List

- Now what does insert look like?
  - Note that in the list with links to nodes 2^i ahead, about 1/2 the nodes are level 1, about a quarter are level 2, …
  - In general, about $1/2^i$ are level *i*
- When we insert, we'll choose the level of the new node randomly according to this probability
  - Flip a coin until it comes up heads, the number of flips is the level



- Operations have expected worst-case running time of *O(log n)*

Backtracking

- Minimax