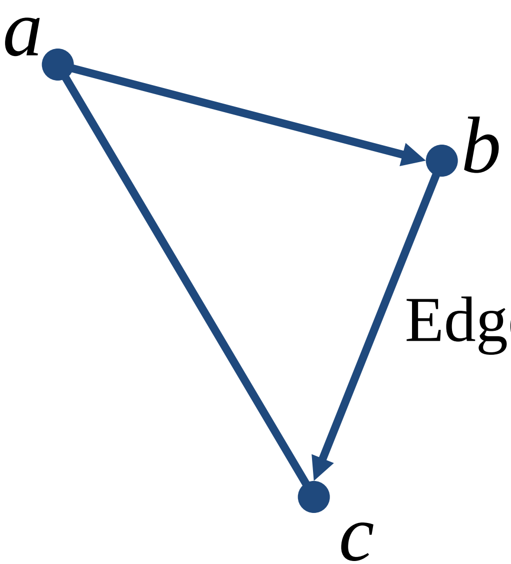
ADVANCED DATA STRUCTURES AND ALGORITHMS

Optimization Problems-Graph Search Algorithms

- Generic Search
- Breadth First Search
- Dijkstra's Shortest Paths Algorithm
- Depth First Search
- Linear Order

Graph



Node ~ city or computer

Edge ~ road or network

Undirected or Directed

A surprisingly large number of problems
in computer science can be expressed
as a graph theory problem.

Generic Search-Graph Search

Specification: Reachability-from-single-source *s*

- <preCond>:
  The input is a graph *G*
  (either directed or undirected)
  and a source node *s*.
- <postCond>:
  Output all the nodes *u* that are
  reachable by a path in *G* from *s*.

Graph Search

Basic Steps:

- Suppose you know that *u* is reachable from *s* & there is an edge from *u* to *v*
- You know that *v* is reachable from *s*
- Build up a set of reachable nodes.

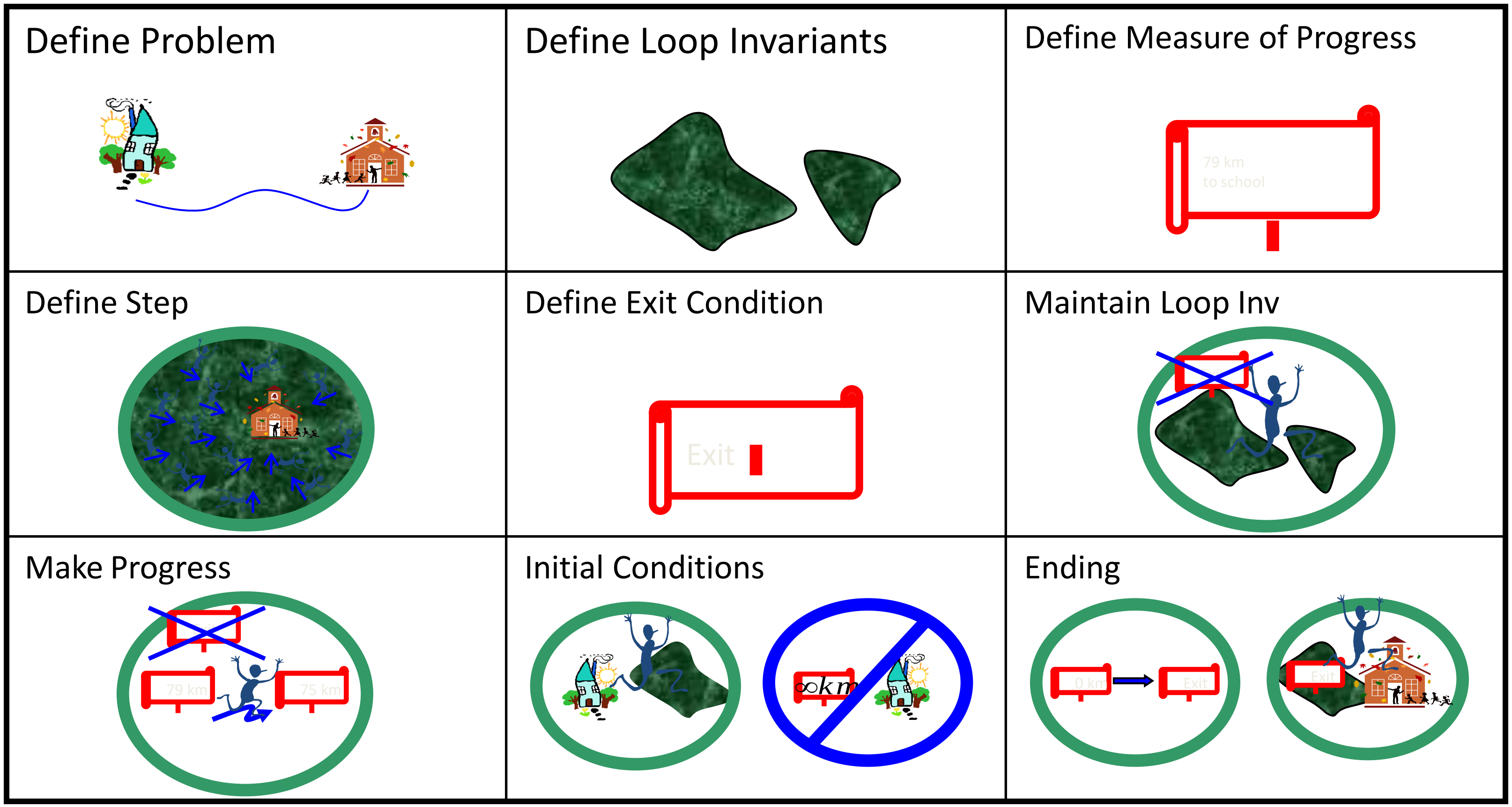
algorithm $Search\,(G, s)$

$\langle \boldsymbol{pre-cond} \rangle$**:** G is a (directed or undirected) graph and s is one of its nodes.

$\langle \boldsymbol{post-cond} \rangle$**:** The output consists of all the nodes u that are reachable by a path in G from s.

```
begin
    foundHandled = ∅
    foundNotHandled = {s}
    loop
        ⟨loop−invariant⟩: See above.

        exit when foundNotHandled = ∅
        let u be some node from foundNotHandled
        for each v connected to u
            if v has not previously been found then
                add v to foundNotHandled
            end if
        end for
        move u from foundNotHandled to foundHandled
    end loop
    return foundHandled
end algorithm
```

Graph Search



Breadth First Search(BFS)

<preCond> &<postCond>

Algorithm

BFS

What order are the nodes found?

So far, the nodes have been found in order of length from *s*.

<postCond>: Finds a shortest path from *s* to each node *v* and its length.

To prove path is shortest:

Prove there is a path of this length. ← Give a path (witness)

Prove there are no shorter paths.

BFS

Basic Steps:

- The shortest path to *u* has length *d* & there is an edge from *u* to *v*
- There is a path to *v* with length *d*+1.

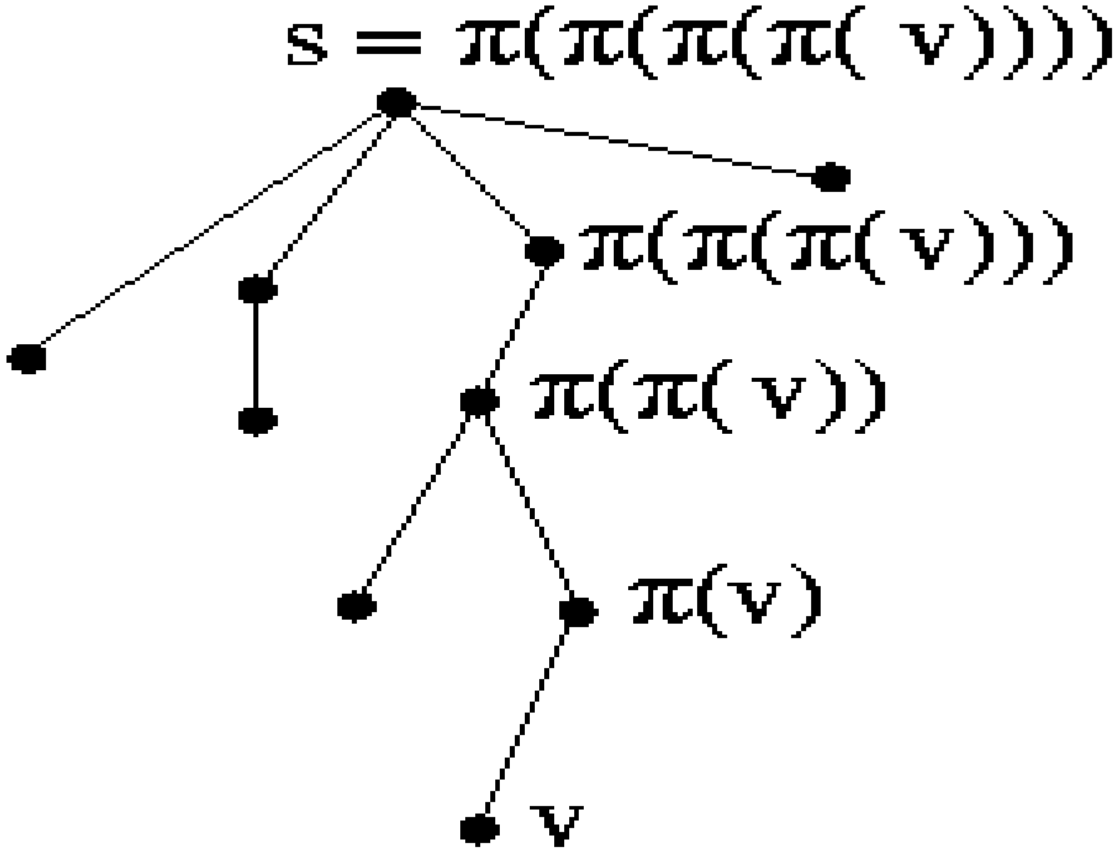
BFS

- What order are the nodes found?
- So far, the nodes have been found in order of length from *s*.
- <postCond>:
- Finds a shortest path from *s* to each node *v* and its length.
- Prove there are no shorter paths.
- When we find *v*, we know
  there isn't a shorter path to it because ?
- Otherwise, we would have found it already.

BFS

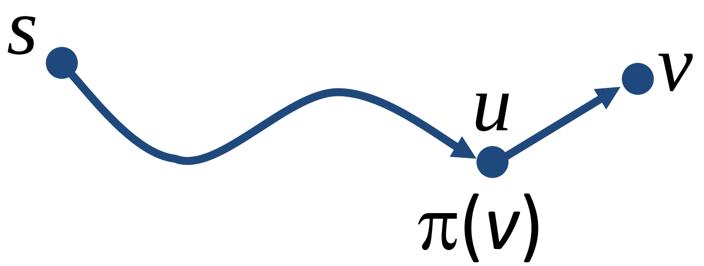
Data structure for storing tree:

- For each node *v*, store $\pi(v)$ to be parent of *v*.



BFS

Basic Steps:



Path to u & there is an edge from u to v

Parent of v is $\pi(v) = u$.

algorithm $ShortestPath\,(G, s)$

$\langle \boldsymbol{pre-cond} \rangle$: G is a (directed or undirected) graph and s is one of its nodes.

$\langle \boldsymbol{post-cond} \rangle$: π specifies a shortest path from s to each node of G and d specifies their lengths.

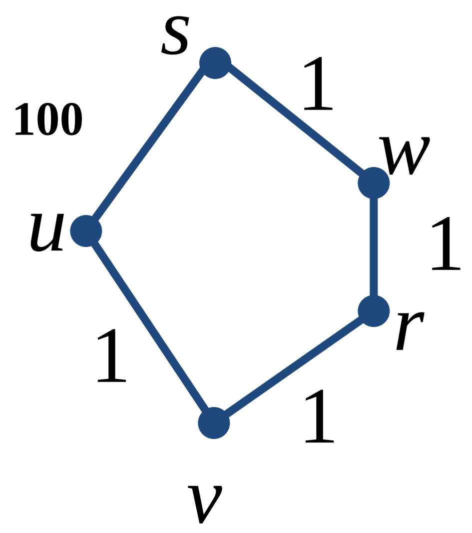
```
begin
    foundHandled = ∅
    foundNotHandled = {s}
    d(s) = 0, π(s) = ε
    loop
        ⟨loop−invariant⟩: See above.

        exit when foundNotHandled = ∅
        let u be the node in the front of the queue foundNotHandled
        for each v connected to u
            if v has not previously been found then
                add v to foundNotHandled
                d(v) = d(u) + 1
                π(v) = u
            end if
        end for
        move u from foundNotHandled to foundHandled
    end loop
    (for unfound v, d(v) = ∞)
    return ⟨d, π⟩
end algorithm
```

Dijkstra's Shortest-Weighted Paths

- Specification: Dijkstra's Shortest-Weighted Paths
- Reachability-from-single-source *s*
  - <preCond>:
    The input is a graph *G*
    (either directed or undirected)
    with positive edge weights
    and a source node *s*.
  - <postCond>:
    Finds a shortest weighted path from *s*
    to each node *v* and its length.

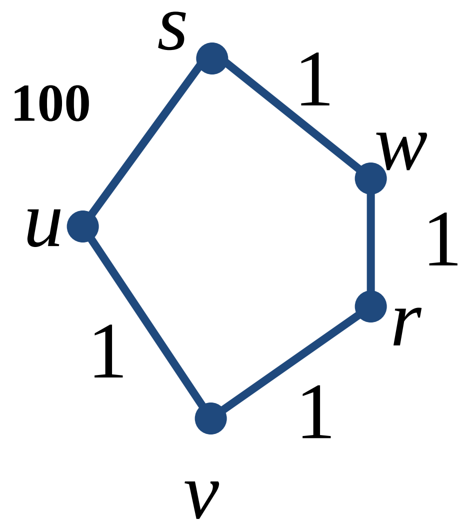
Dijkstra's Shortest-Weighted Paths

- Length of shortest path from *s* to *u*?



BFS

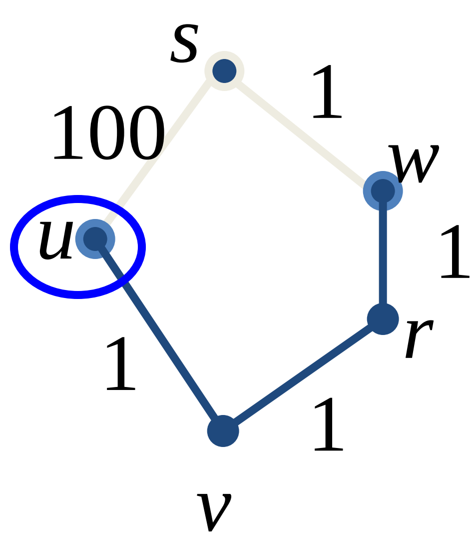
- So far, the nodes have been found in order of length from s.
- Is the same true for Dijkstra's Algorithm?



Which node is found first?

BFS

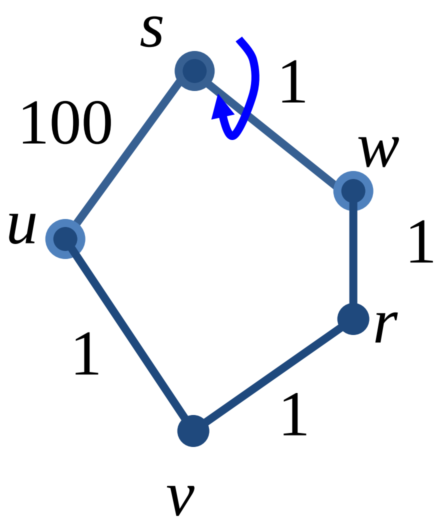
- So far, the nodes have been found in order of length from *s*.
- Is the same true for Dijkstra's Algorithm?



- Which node is found first?
- It has the longest path from *s*.

Dijkstra's

- So far, the nodes have been found in order of length from *s*. handled
- In what order do we handle the foundNotHandled nodes?



Handle node that “seems” to be closest to *s*.

Dijkstra's

- So far, the nodes have been handled in order of length from *s*.

<postCond>:

- Finds a shortest weighted path
  from *s* to each node *v* and its length.
- To prove path is shortest:
- Prove there is a path of this length.
- Prove there are no shorter paths.
- Give a path (witness)

Dijkstra's

- So far, the nodes have been handled in order of length from *s*.
- <postCond>:
- Finds a shortest weighted path
  from *s* to each node *v* and its length.
- To prove path is shortest:
- Prove there is a path of this length.
- Prove there are no shorter paths.
- When we handle *v*, we know
  there isn't a shorter path to it because?

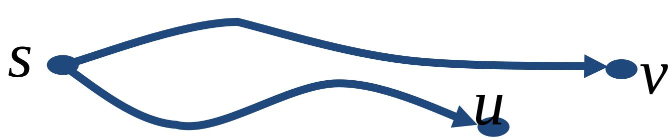
Dijkstra's

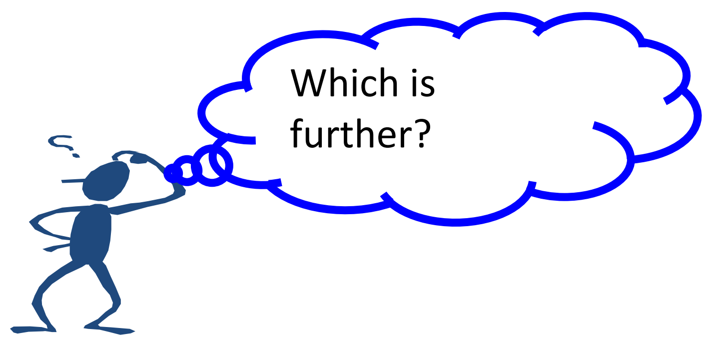
Basic Steps:

Handle node that "seems" to be closest to *s*.

Need to keep approximate shortest distances.

- Path that we have "seen so far" will be called handled paths.
- Let *d*(*v*) the length of the shortest such path to *v*.





Dijkstra's

Basic Steps:

Updating $d(u)$.

The shortest of handled paths to v has length $d(v)$

- The shortest of handled paths to u has length $d(u)$ & there is an edge from u to v
- The shortest known path to v has length

min($d(v)$, $d(u)+w_{<u,v>}$).

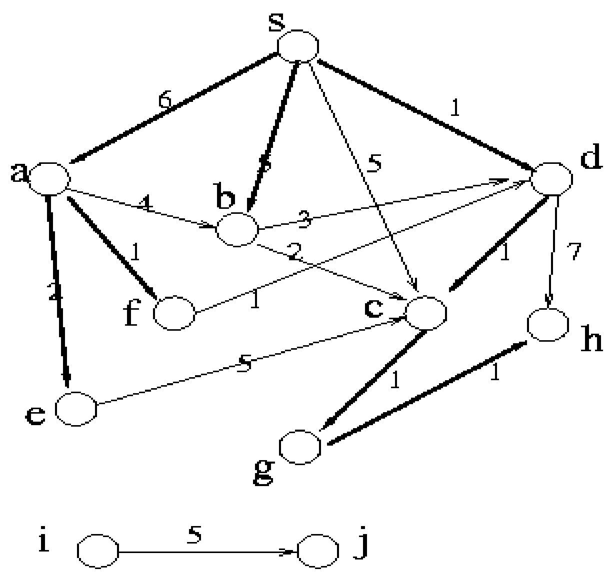
algorithm $ShortestWeightedPath(G, s)$

$\langle \boldsymbol{pre-cond} \rangle$: G is a weighted (directed or undirected) graph and s is one of its nodes.

$\langle \boldsymbol{post-cond} \rangle$: π specifies a shortest weighted path from s to each node of G and d specifies their lengths.

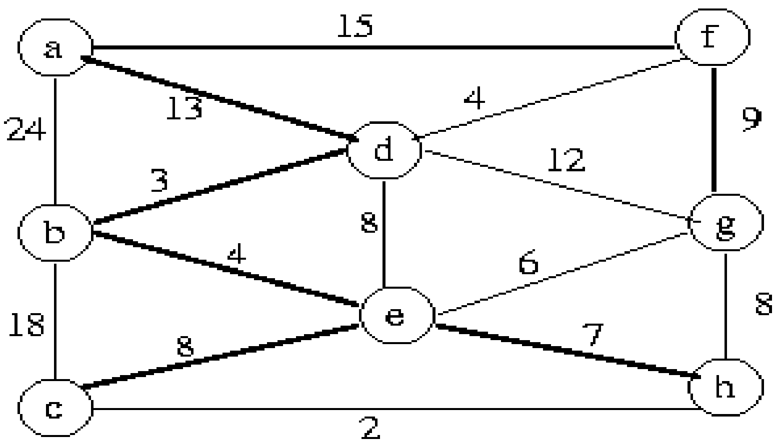
```
begin
    d(s) = 0, π(s) = ε
    for other v, d(v) = ∞ and π(v) = nil
    handled = ∅
    notHandled = priority queue containing all nodes. Priorities given by d(v).
    loop
        ⟨loop-invariant⟩: See above.

        exit when notHandled = ∅
        let u be a node from notHandled with smallest d(u)
        for each v connected to u
            foundPathLength = d(u) + w⟨u,v⟩
            if d(v) > foundPathLength then
                d(v) = foundPathLength
                (update the notHandled priority queue)
                π(v) = u
            end if
        end for
        move u from notHandled to handled
    end loop
    return ⟨d, π⟩
end algorithm
```

Dijkstra's



Handled	s	a	b	c	d	e	f	g	h	i	j
	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
s		6	5	5	1						
d				2					8		
c								3			
g									4		
h			5								
b		6									
a						8	7				
f						8					
e											
i										∞	
j											∞

Dijkstra's



Handled	a	b	c	d	e	f	g	h
	0	∞	∞	∞	∞	∞	∞	∞
a		24		13		15		
d		16			21	15	25	
f		16					24	
b			34		20			
e			28				24	27
g								27
h			28					
c								

Depth First Search

- Breadth first search makes a lot of sense for dating in general actually.
- It suggests dating a bunch of people casually before getting serious rather than having a series of five year relationships.

algorithm $DepthFirstSearch(G, s)$

$\langle \boldsymbol{pre-cond} \rangle$**:** G is a (directed or undirected) graph and s is one of its nodes.

$\langle \boldsymbol{post-cond} \rangle$**:** The output is a depth-first search tree of G rooted at s.

```
begin
    foundHandled = ∅
    foundNotHandled = {⟨s,0⟩}
    loop
        ⟨loop−invariant⟩: See above.

        exit when foundNotHandled = ∅
        pop ⟨u,i⟩ off the stack foundNotHandled
        if u has an (i+1)^st edge ⟨u,v⟩
            push ⟨u,i+1⟩ onto foundNotHandled
            if v has not previously been found then
                π(v) = u
                ⟨u,v⟩ is a tree edge
                push ⟨v,0⟩ onto foundNotHandled
            else if v has been found but not completely handled then
                ⟨u,v⟩ is a back edge
            else (v has been completely handled)
                ⟨u,v⟩ is a forward or cross edge
            end if
        else
            move u to foundHandled
        end if
    end loop
    return foundHandled
end algorithm
```

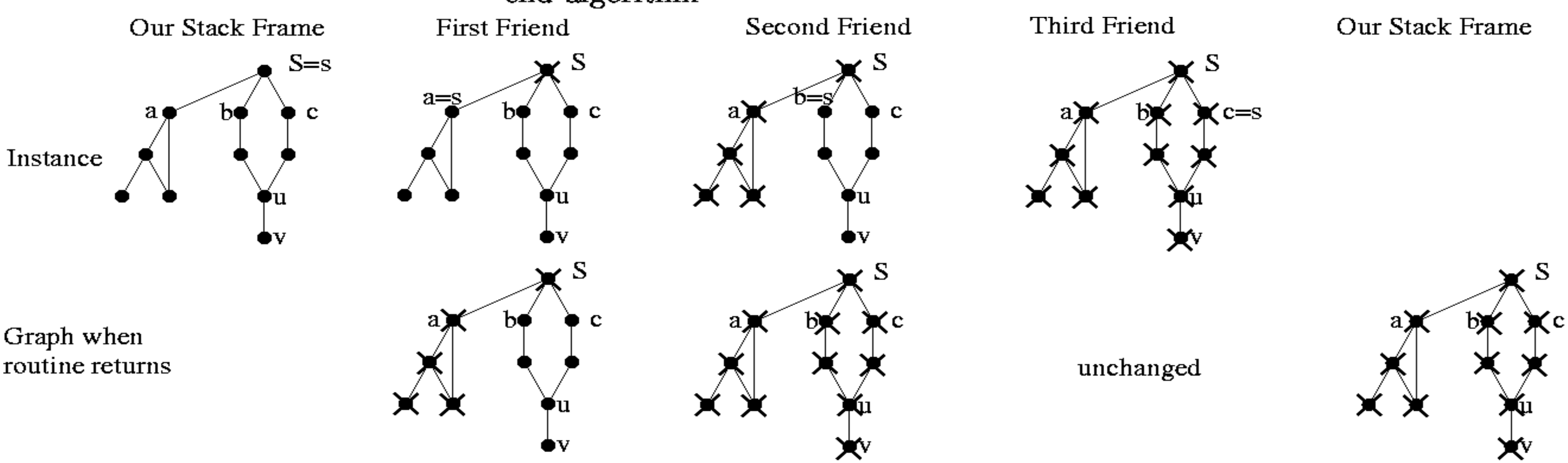
Recursive Depth First Search

algorithm $DepthFirstSearch\,(s)$

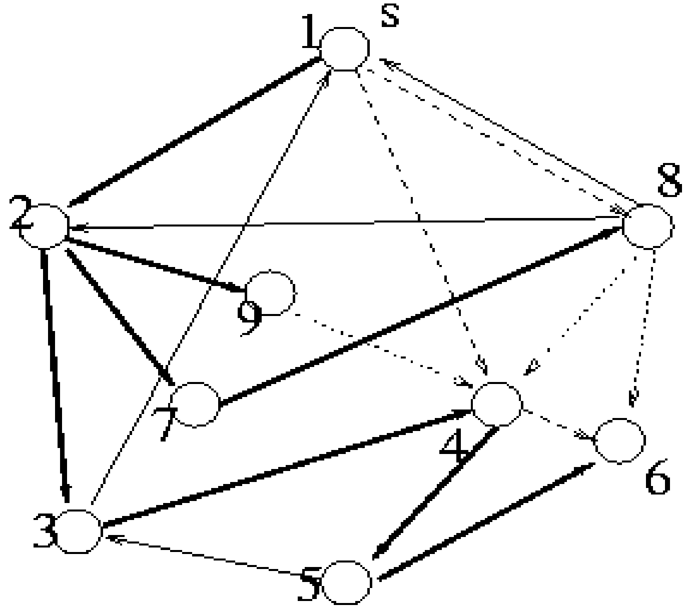
$\langle \textbf{\textit{pre}}-\textbf{\textit{cond}} \rangle$: An input instance consists of a (directed or undirected) graph G with some of its nodes marked $found$ and a source node s.

$\langle \textbf{\textit{post}}-\textbf{\textit{cond}} \rangle$: The output is the same graph G except all nodes v reachable from s without passing through a previously found node are now also marked as being found.

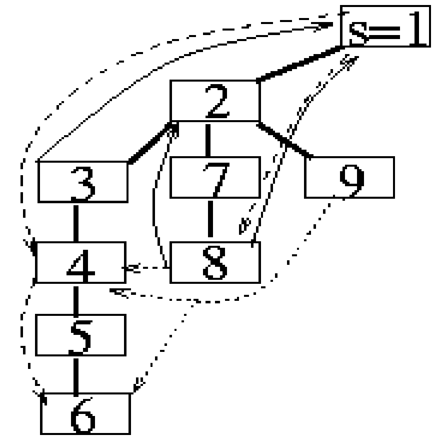
begin
    if s is marked as found then
        do nothing
    else
        mark s as found
        for each v connected to s
            $DepthFirstSearch\,(v)$
        end for
    end if
end algorithm



Graph



Recursive Stack Frames



Iterative Alg

Stack	Handled
{s=1}	
{1,2,3,4,5,6}	
{1,2}	6,5,4,3
{1,2,7,8}	6,5,4,3
{1,2}	6,5,4,3,8,7
{1,2,9}	6,5,4,3,8,7
	6.5.4.3.8.7.9.2.1

Types of Edges

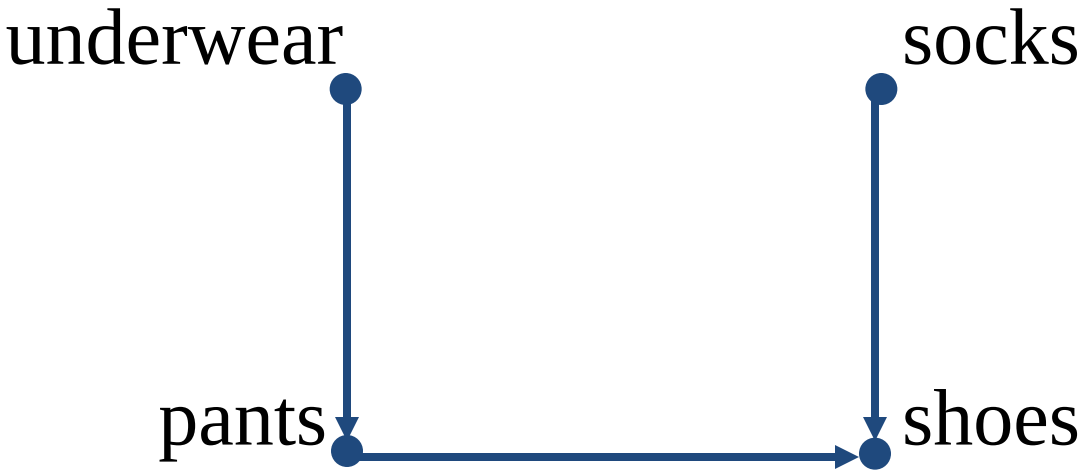
Tree edges

Back edges

Forward edges

Cross edges

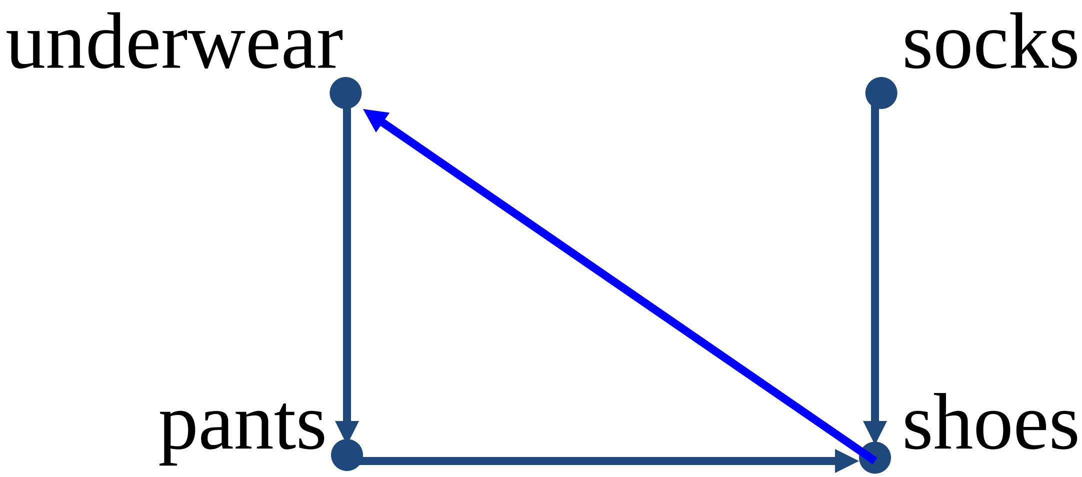
Linear Order of a Partial Order



underwear
pants
socks
shoes

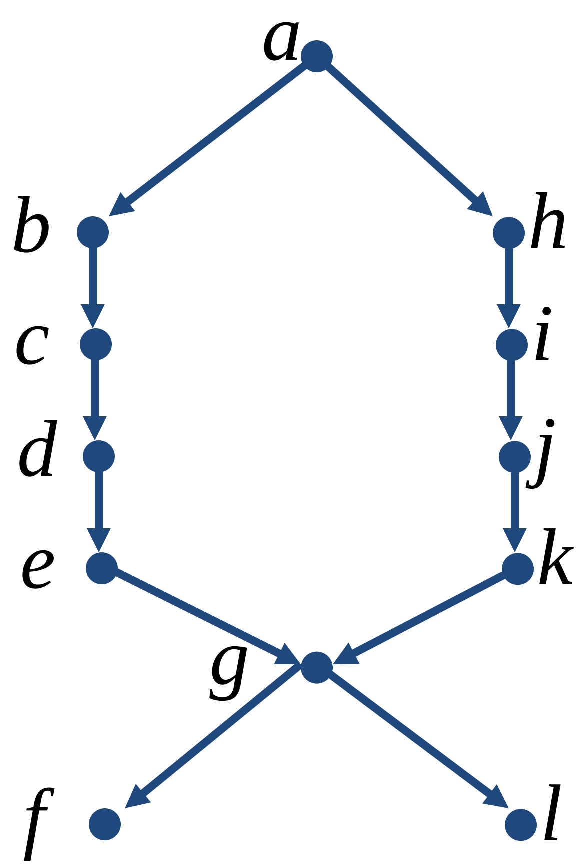
socks
underwear
pants
shoes

Linear Order



?

Linear Order

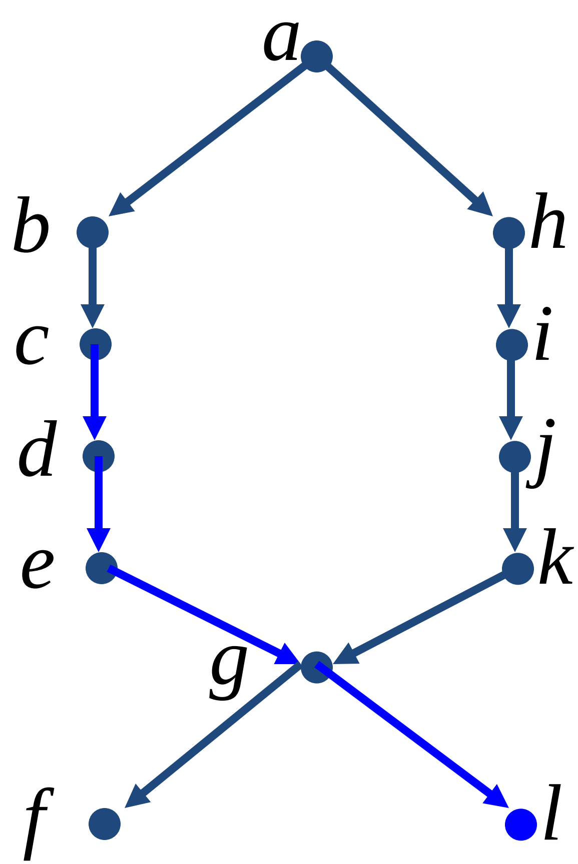


<preCond>:
A Directed Acyclic Graph(DAG)
<postCond>:
Find one valid linear order

Algorithm:
- Find a sink ?
- Put it last in order.
- Delete & Repeat

….. *l*

Linear Order



<preCond>:
A Directed Acyclic Graph(DAG)
<postCond>:
Find one validlinear order

Algorithm:
- Find a sink.
- Put it last in order.
- Delete & Repeat

$\Theta(n)$
$\Theta(n^2)$

….. l

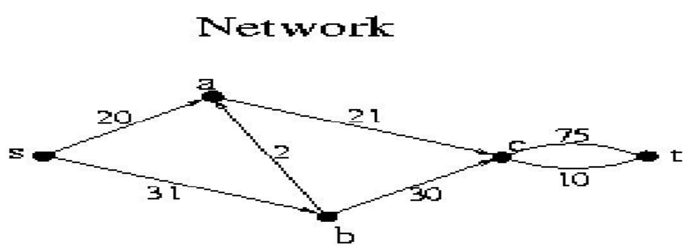
Network Flow & Linear Programming

Optimization Problems

- Ingredients:
  - Instances: The possible inputs to the problem.
  - Solutions for Instance: Each instance has an exponentially large set of solutions.
  - Cost of Solution: Each solution has an easy to compute cost or value.
- Specification
  - Preconditions: The input is one instance.
  - Postconditions: An valid solution with optimal cost. (minimum or maximum)

Network Flow

- Instance:
  - A Network is a directed graph G
  - Edges represent pipes that carry flow
  - Each edge $<u,v>$ has a maximum capacity $c_{<u,v>}$
  - A source node s out of which flow leaves
  - A sink node t into which flow arrives
  - Goal: Max Flow



Network Flow

- For some edges/pipes, it is not clear which direction the flow should go
  in order to maximize the flow from *s* to *t*.
- Hence we allow flow in both directions.

•Solution:

- The amount of flow $F_{<u,v>}$ through each edge.
- Flow $F_{<u,v>}$ can't exceed capacity $c_{<u,v>}$.
- No leaks, no extra flow.

  For each node v: flow in = flow out

  $$\sum_u F_{<u,v>} = \sum_w F_{<v,w>}$$

- Value of Solution:
  - Flow from s into the network
  - minus flow from the network back into s.
  - rate(F) = $\sum_u F_{<s,u>}$
- Goal: Max Flow

$- \sum_v F_{<v,s>}$

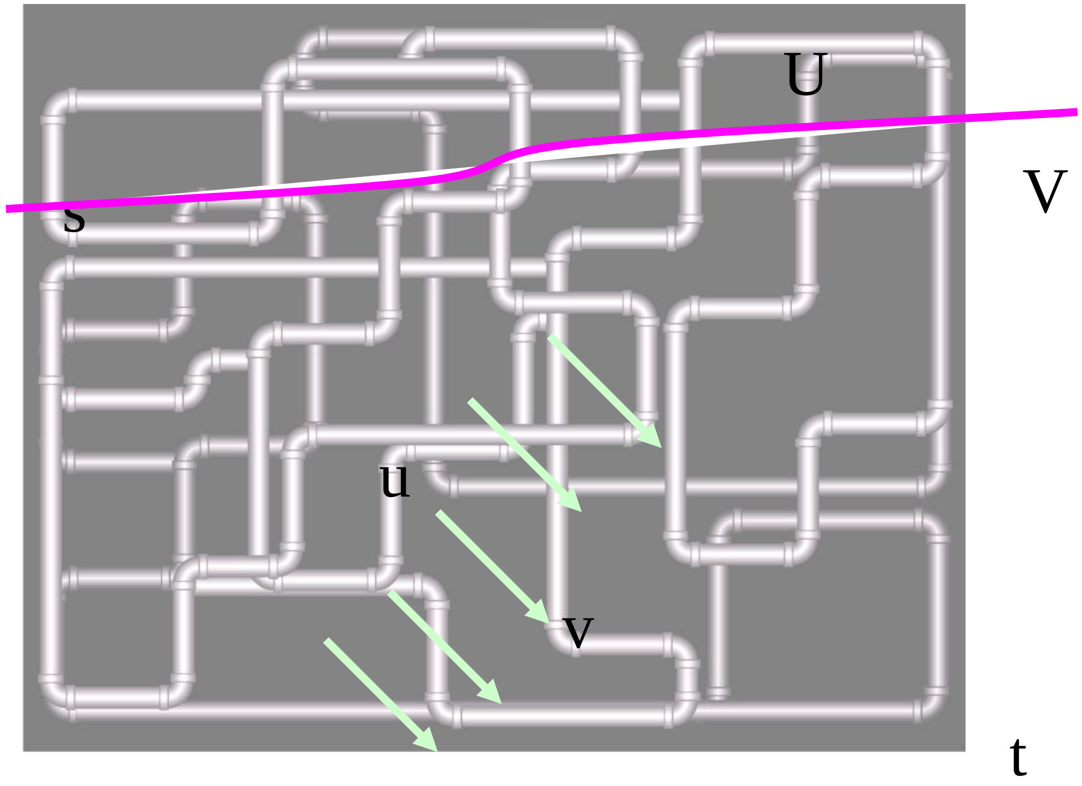
Min Cut

- Value Solution C=<U,V>:

cap(C) = how much can flow from U to V

$$= \sum_{u \in U, v \in V} c_{<u,v>}$$

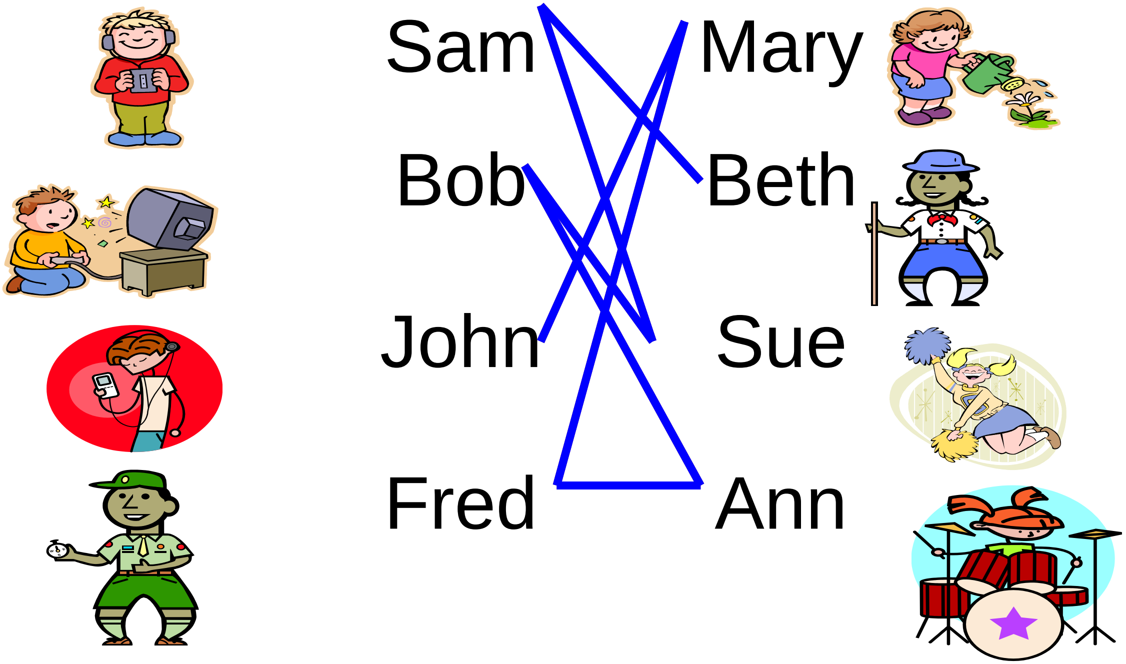
Goal: Min Cut



Max Flow = Min Cut

- Theorem:
  - For all Networks Max_F rate(F) = Min_C cap(C)
- Prove: $\forall$ F,C   rate(F) $\le$ cap(C)
- Prove: $\forall$ flow F, alg either
  - finds a better flow F
  - or finds cut C such that rate(F) = cap(C)
- Algotiyhm stops with an F and C for which rate(F) = cap(C)
  - F witnesses that the optimal flow can't be less
  - C witnesses that it can't be more.

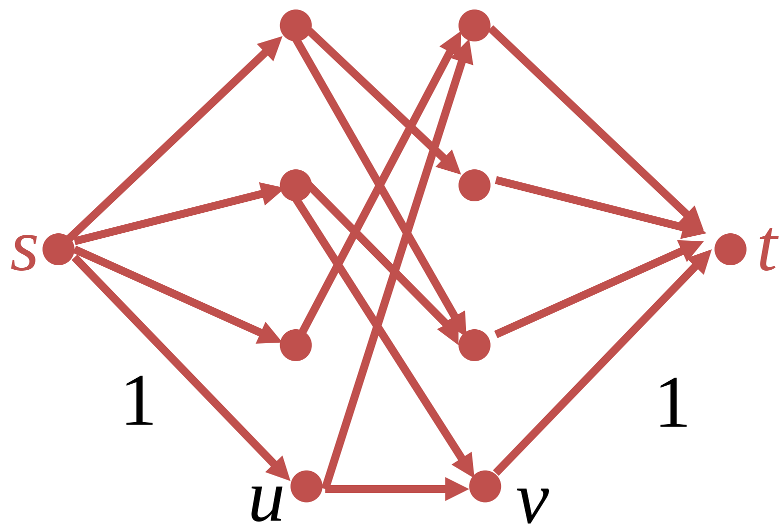
An Application: Matching



3 matches
Can we do better?
4 matches

Who likes whom?
Who should be matched with whom?
so as many as possible matched
and nobody matched twice?

An Application: Matching



$c_{<s,u>} = 1$

- Total flow out of u = flow into $u \leq 1$
- Boy u matched to at most one girl.

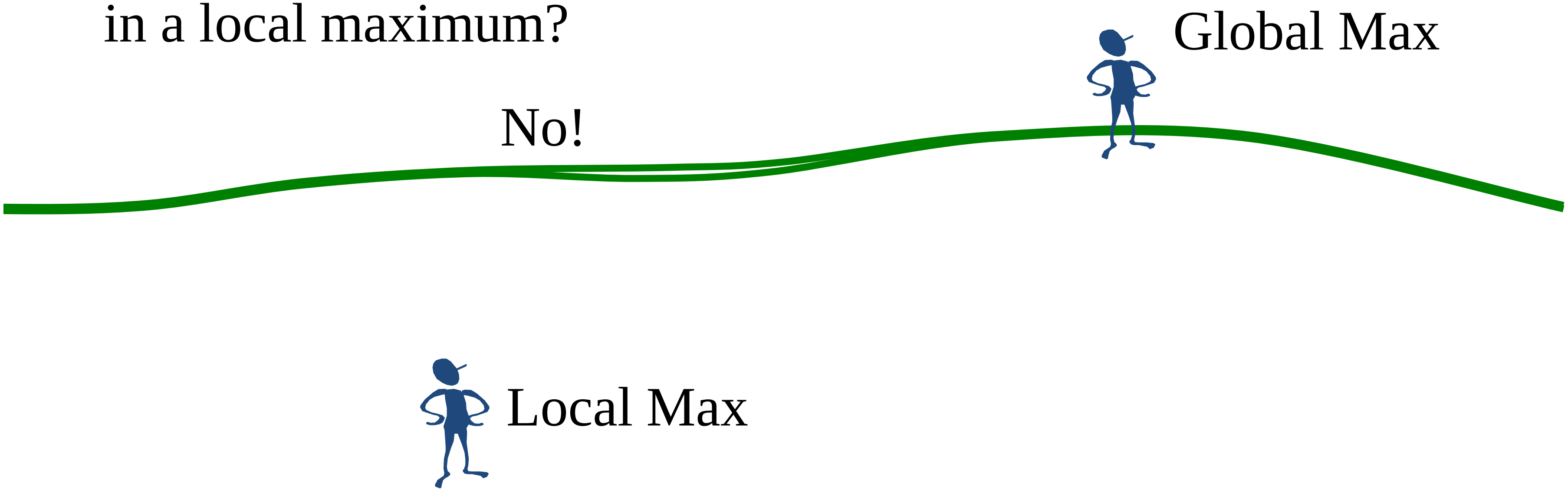
$c_{<v,t>} = 1$

- Total flow into v = flow out of $v \leq 1$
- Girl v matched to at most one boy.

Hill Climbing

Problems:

Can our Network Flow
Algorithm get stuck
in a local maximum?



Hill Climbing

Problems:

Running time?

If you take small step,
could be exponential time.

Hill Climbing

Problems:
Running time?

- If each iteration you take the biggest step possible,
  - Algorithm is poly time
    - in number of nodes
    - and number of bits in capacities.
- If each iteration you take path with the fewest edges
  - Algorithm is poly time
    - in number of nodes

Taking the biggest step possible

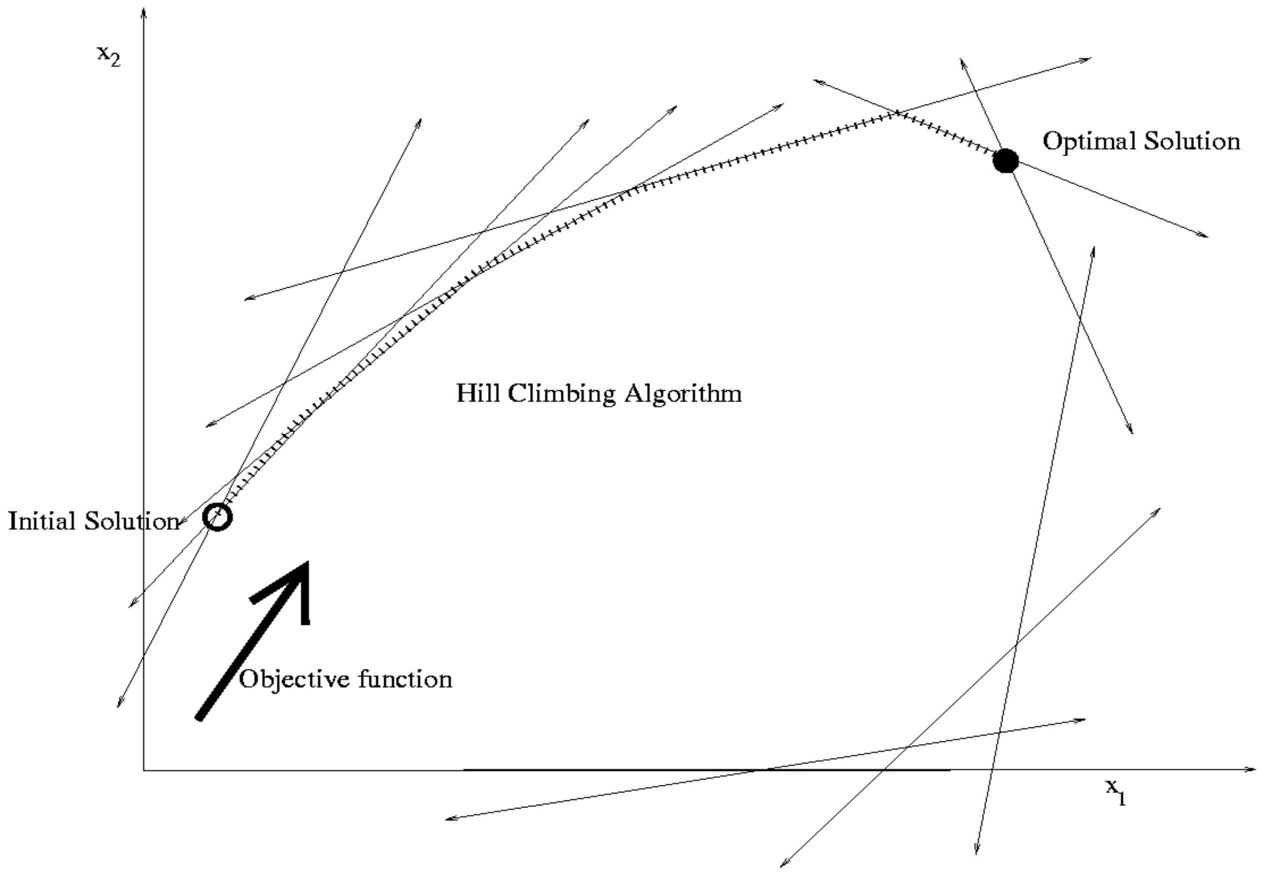
algorithm $LargestShortestWeight\,(G, s, t)$

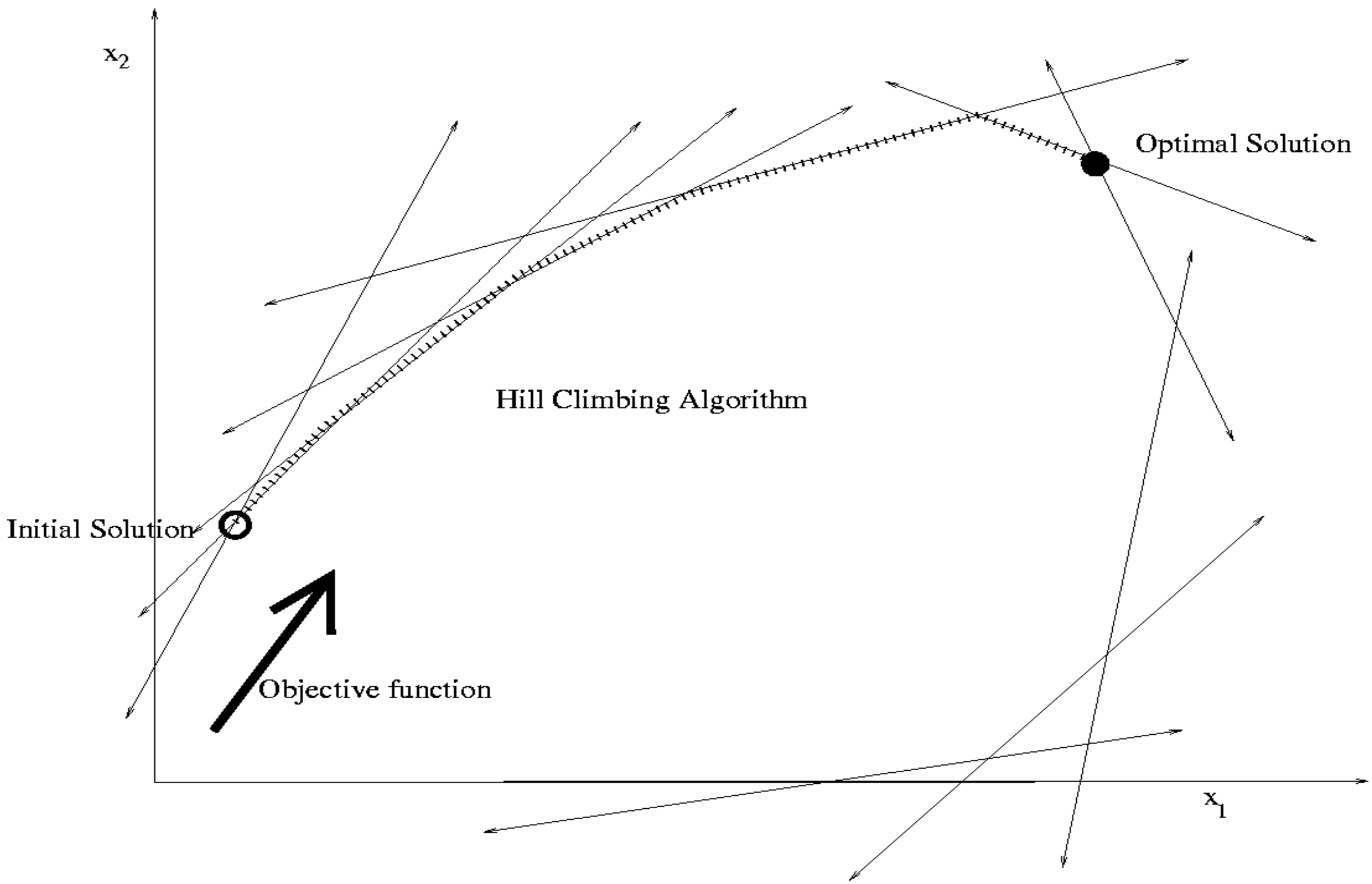
$\langle \boldsymbol{pre-cond} \rangle$: G is a weighted directed graph. s is the source node. t is the sink.

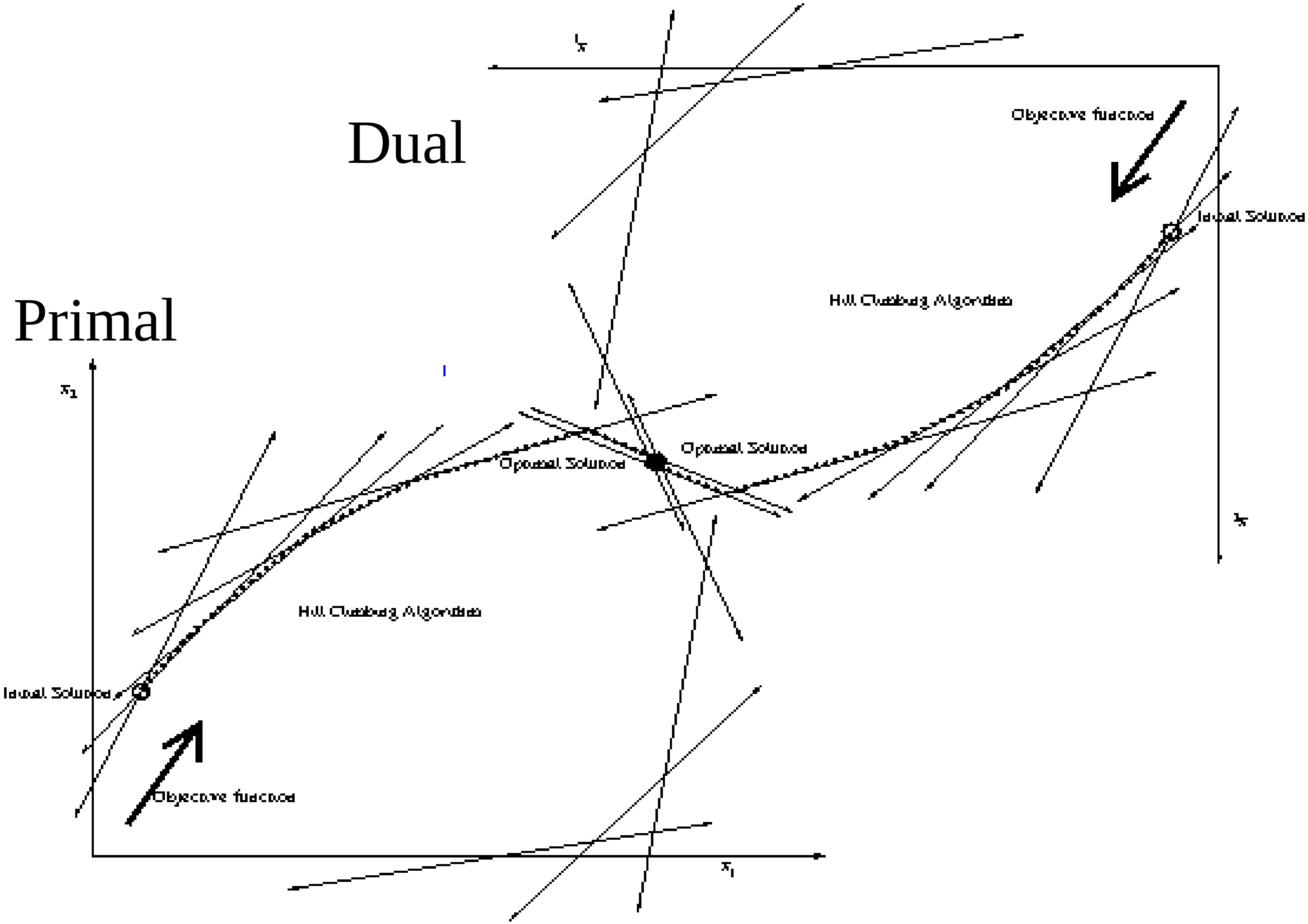
$\langle \boldsymbol{post-cond} \rangle$: P specifies a path from s to t whose smallest edge weight is as large as possible.

```
begin
      Sort the edges by weight from largest to smallest
      G' = graph with no edges
      mark s reachable
      loop
            ⟨loop-invariant⟩: Every node reachable from s in G' is
                  marked reachable.
            exit when t is reachable
            ⟨u, v⟩ = the next largest weighted edge in G
            Add ⟨u, v⟩ to G'
            if( u is marked reachable and v is not ) then
                  Do a depth first search from v marking all reachable nodes
                  not marked before.
            end if
      end loop
      P = path from s to t in G'
      return( P)
end algorithm
```

Linear Programming




x_2
Optimal Solution
Hill Climbing Algorithm
Initial Solution
Objective function
x_1


Dual
Primal
Objective function
Initial Solution
Hill Climbing Algorithm
Optimal Solution
Optimal Solution
Hill Climbing Algorithm
Initial Solution
Objective function

Linear Programming

Linear Program:

- An *optimization* problem whose *constraints* and *cost function* are linear functions
- **Goal:** Find a solution which optimizes the cost.

E.g.

Maximize Cost Function :

$21x_1 - 6x_2 - 100x_3 - 100x_4$

Constraint Functions:

$5x_1 + 2x_2 + 31x_3 - 20x_4 \leq 21$

$1x_1 - 4x_2 + 3x_3 + 10x_1 \ {}^{3}\ 56$

$6x_1 + 60x_2 - 31x_3 - 15x_4 \leq 200$

…..

Primal-Dual Hill Climbing

Mars settlement has hilly landscape
and many layers of roofs.
Primal Problem:
•Exponential # of locations to stand.
•Find a highest one.
Dual problem:
•Exponential # of roofs.
•Find a lowest one.
Prove:
•Every roof is above every location to stand.

$$\forall R \;\; \forall L \; height(R) \geq height(L)$$
$$\Rightarrow height(R_{min}) \geq height(L_{max})$$

• Is there a gap?

- Prove:
- For every location to stand either:
    - the alg takes a step up or
    - the alg gives a reason that explains why not by giving a ceiling of equal height.
  - i.e. $\forall L\ [\exists L'\ height(L') \geq height(L)$ or $\exists R\ \ height(R) = height(L)]$
- But $\forall R\ \forall L\ height(R) \geq height(L)$

Recursive Backtracking

- The brute force algorithm for an optimization problem is to simply compute the cost or value of each of the exponential number of possible solutions and return the best.
- A key problem with this algorithm is that it takes exponential time.
- Another (not obviously trivial) problem is how to write code that enumerates over all possible solutions.
- Often the easiest way to do this is recursive backtracking.

An Algorithm as a Sequence of Decisions:

- An algorithm for finding an optimal solution for your instance must make a sequence of small decisions about the solution
- "Do we include the first object in the solution or not?"
- "Do we include the second?"
- "The third?" . . . , or "At the first fork in the road, do we go left or right?"

- “At the second fork which direction do we go?” “At the third?” . . . .
- As one stack frame in the recursive algorithm, our task is to deal only with the first of these decisions.
- A recursive friend will deal with the rest

Searching for the Best Animal

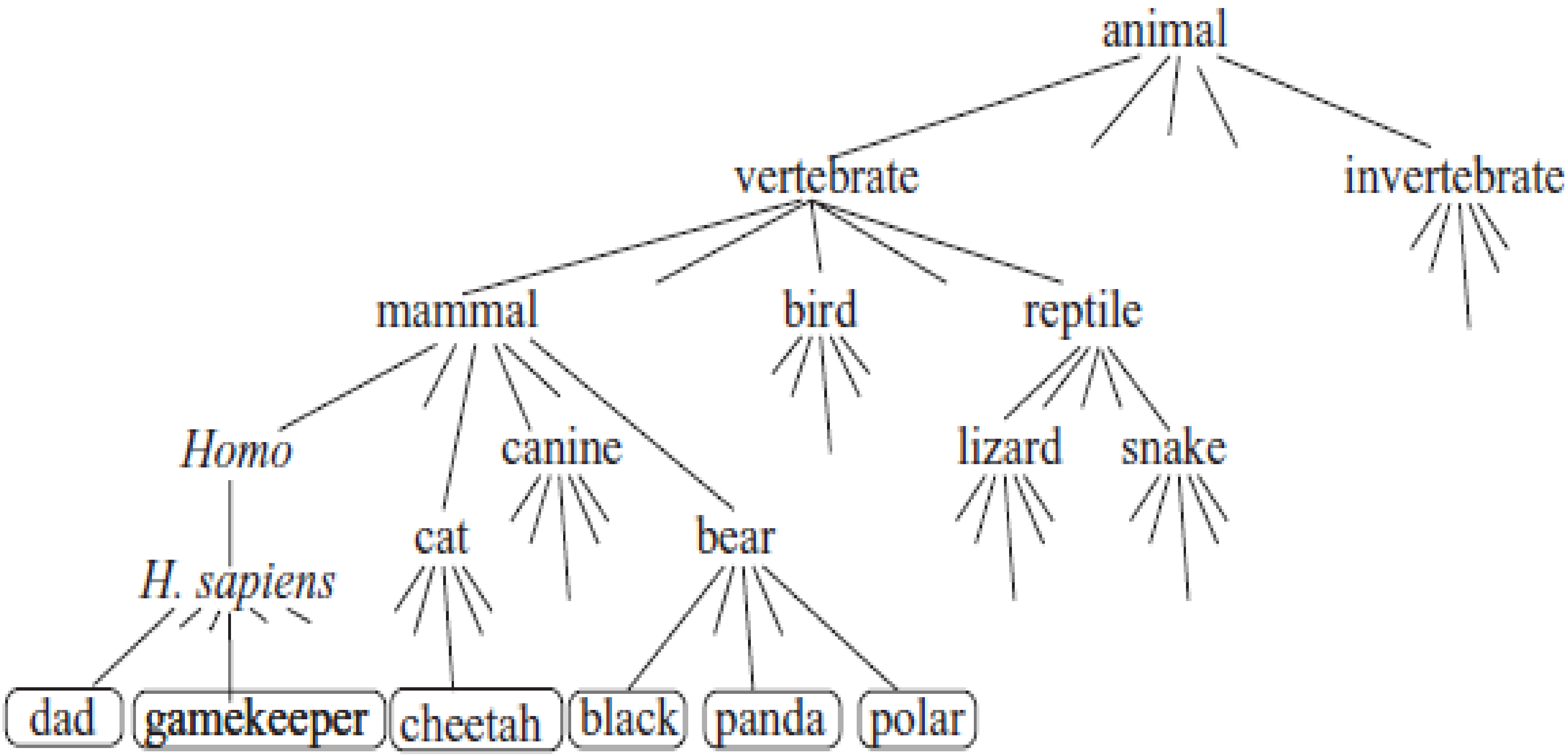
- Searching through a large set of objects, say for the best animal at the zoo.
- we break the search into smaller searches, each of which we delegate to a friend.
- We might ask one friend for the best vertebrate and another for the best invertebrate.
- We will take the better of these best as our answer.
- This algorithm is recursive.

- The friend with the vertebrate task asks a friend to find the best mammal, another for the best bird, and another for the best reptile.

A Classification Tree of Solutions:

- This algorithm unwinds into the tree of stack frames that directly mirrors the taxonomy tree that classifies animals.
- Each solution is identified with a leaf.

Classification Tree of Animals



The Little Bird Abstraction:

- A little bird abstraction to help focus on two of the most difficult and creative parts of designing a recursive backtracking algorithm.

A Flock of Stupid Birds vs.wise Little Bird:

A Flock of Stupid Birds:

- whether the optimal solution is a mammal, a bird, or a reptile has K different answers
- Giving her the benefit of doubt, we ask a friend to give us the optimal solution from among those that are consistent with this answer.

- At least one of these birds must have been telling us the truth.

Wise Little Bird:

- If little bird answers correctly, designing an algorithm would be a lot easier
- Ask the little bird “Is the best animal a bird, a mammal, a reptile, or a fish?”
- Little Bird tells us a mammal.
- Just ask our friend for the best mammal.
- Trusting the little bird and the friend, we give this as the best animal.

Developing a Recursive Backtracking Algorithm

Objectives:

- Understand backtracking algorithms and use them to solve problems
- Use recursive functions to implement backtracking algorithms
- How the choice of data structures can affect the efficiency of a program?

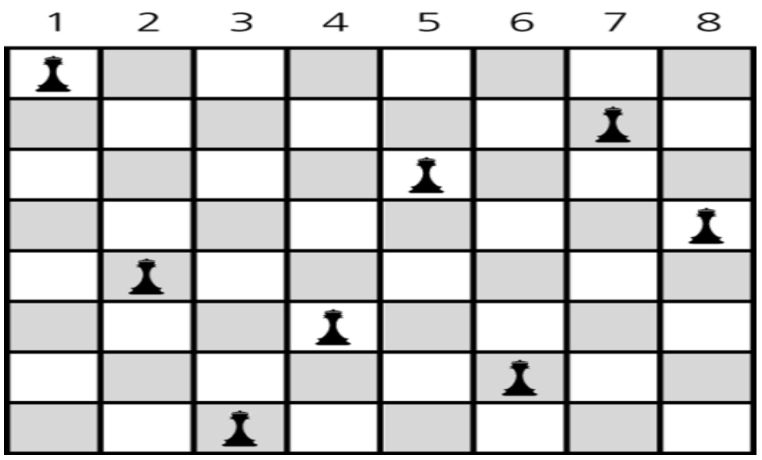
Backtracking

- Backtracking
  - A strategy for guessing at a solution and backing up when an impasse is reached
- Recursion and backtracking can be combined to solve problems
- Eight-Queens Problem
  - Place eight queens on the chessboard so that no queen can attack any other queen

The Eight Queens Problem

- One strategy: guess at a solution
  - There are 4,426,165,368 ways to arrange 8 queens on a chessboard of 64 squares
- An observation that eliminates many arrangements from consideration
  - No queen can reside in a row or a column that contains another queen
    - Now: only 40,320 (8!) arrangements of queens to be checked for attacks along diagonals

- Providing organization for the guessing strategy
  - Place queens one column at a time
  - If you reach an impasse, backtrack to the previous column

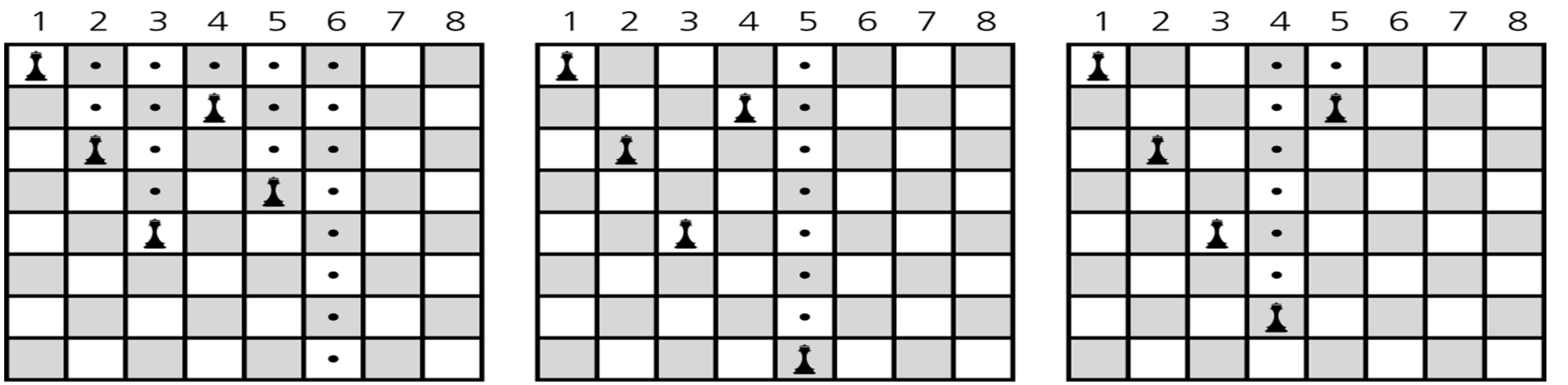


A solution to the Eight Queens problem

The Eight Queens Problem

- A recursive algorithm that places a queen in a column
  - Base case
    - If there are no more columns to consider
      - You are finished
  - Recursive step
    - If you successfully place a queen in the current column
      - Consider the next column
    - If you cannot place a queen in the current column
      - You need to backtrack

The Eight Queens Problem



(a) (b) (c)

a)Five queens that cannot attack each other, but that can attack all of column 6;

b)Backtracking to column 5 to try another square for the queen;

c)Backtracking to column 4 to try another square for the queen and then considering column 5 again

Pruning Branches

- The typical reasons why an entire branch of the solution classification tree can be pruned off.

Invalid Solutions:

- It happens partway down the tree the algorithm has already received enough information about the solution
- Then it determine that it contains a conflict or defect making any such solution invalid.
- The algorithm can stop recursing at this point and backtrack.
- This effectively prunes off the entire subtree of solutions rooted at this node in the tree.

No Highly Valued Solutions:

- The algorithm arrives at the root of a subtree, it might realize that no solutions within this subtree are rated sufficiently high to be optimal
- Perhaps because the algorithm has already found a solution probably better than all of these.
- Again, the algorithm can prune this entire subtree from its search.

Greedy Algorithms:

- Greedy algorithms are effectively recursive backtracking algorithms with extreme pruning.
- Whenever the algorithm has a choice as to which little bird’s answer to take
- Then it looks best according to some greedy criterion.

Modifying Solutions:

- Modifying any possible solution that is not consistent with the latest choice into onethat has at least as good value and is consistent with this choice.

Satisfiability

- A famous optimization problem is called satisfiability.
- The recursive backtracking algorithm is referred to as the Davis–Putnam algorithm.
- An example of an algorithm whose running time is exponential for worst case inputs

Satisfiability Problem

Instances:

- An instance (input) consists of a set of constraints on the assignment to the binary variables $x1, x2, . . . , xn$.
- A typical constraint might be $x1$ *or* $x3$ *or* $x8$, equivalently that either $x1$ is true, $x3$ is false, or $x8$ is true.

Solutions:

- Each of the $2n$ assignments is a possible solution.
- An assignment is valid for the given instance if it satisfies all of the constraints.

Measure of Success:

- An assignment is assigned the value one if it satisfies all of the constraints, and the value zero otherwise.

Goal:

- Given the constraints, the goal is to find a satisfying assignment.

Code:

```
algorithm DavisPutnam (c)
⟨pre-cond⟩: c is a set of constraints on the assignment to x⃗.
⟨post-cond⟩: If possible, optSol is a satisfying assignment and optCost is also c
Otherwise optCost is zero.

begin
    if( c has no constraints or no variables ) then
        % c is trivially satisfiable.
        return ⟨∅, 1⟩
    else if( c has both a constraint forcing a variable x_i to 0
        and one forcing the same variable to 1) then
        % c is trivially not satisfiable.
        return ⟨∅, 0⟩
    else
        for any variable forced by a constraint to some value
            substitute this value into c.
        let x_i be the variable that appears the most often in c
        % Loop over the possible bird answers.
        for k = 0 to 1 (unless a satisfying solution has been found)
            % Get help from friend.
            let c' be the constraints c with k substituted in for x_i
            ⟨optSubSol, optSubCost⟩ = DavisPutnam (c')
            optSol_k = ⟨forced values, x_i = k, optSubSol⟩
            optCost_k = optSubCost
        end for
        % Take the best bird answer.
        k_max = a k that maximizes optCost_k
        optSol = optSol_{k_max}
        optCost = optCost_{k_max}
        return ⟨optSol, optCost⟩
    end if
end algorithm
```

Running Time:

- If no pruning is done, then the running time is $(2n)$, as all $2n$ assignments are tried.
- Considerable pruning needs to occur to make the algorithm polynomial-time.
- Certainly in the worst case, the running time is $2(n)$.