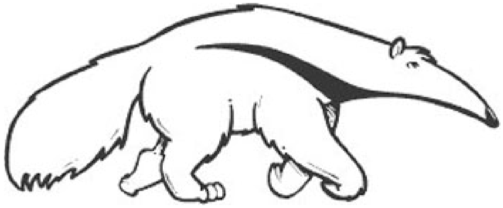
Midterm Review

CS171, Fall Quarter, 2018

Introduction to Artificial Intelligence

Prof. Richard Lathrop



Read Beforehand: **R&N All Assigned Reading Chaps. 1-4, 7-9, 13-14**

BREN:ICS
INFORMATION AND COMPUTER SCIENCES

UNIVERSITY of CALIFORNIA IRVINE

Review Agents
Chapter 2.1-2.3

- Agent definition (2.1)

- Rational Agent definition (2.2)
  - Performance measure

- Task evironment definition (2.3)
  - PEAS acronym
  - Properties of task environments

Agents

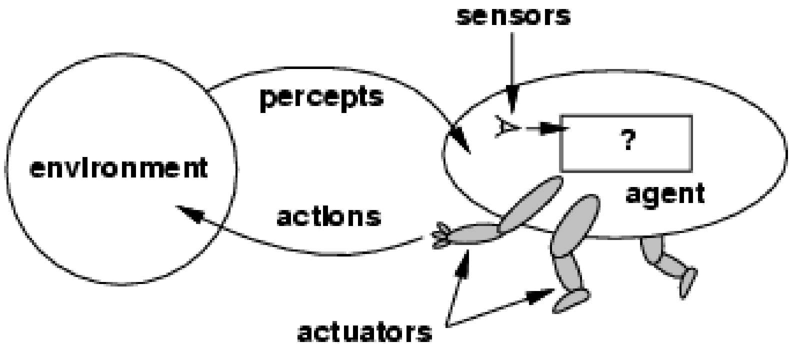
- An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators

- Human agent:
  - Sensors: eyes, ears, …
  - Actuators: hands, legs, mouth…

- Robotic agent
  - Sensors: cameras, range finders, …
  - Actuators: motors

www.themindrobber.co.uk (c)2005 dalek@themindrobber.co.uk

Agents and environments



- Percept: agent’s perceptual inputs at an instant
- The agent function maps from percept sequences to actions: [f: $\mathcal{P}^* \rightarrow \mathcal{A}$]
- The agent program runs on the physical architecture to produce f
- agent = architecture + program

Rational agents

- Rational Agent: For each possible percept sequence, a rational agent should select an action that is *expected* to maximize its performance measure, based on the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

- Performance measure: An objective criterion for success of an agent's behavior (“cost”, “reward”, “utility”)

- E.g., performance measure of a vacuum-cleaner agent could be amount of dirt cleaned up, amount of time taken, amount of electricity consumed, amount of noise generated, etc.

Task Environment

- Before we design an intelligent agent, we must specify its “task environment”:

PEAS:

Performance measure

Environment

Actuators

Sensors

Environment types

- **Fully observable (vs. partially observable):** An agent's sensors give it access to the complete state of the environment at each point in time.

- **Deterministic (vs. stochastic):** The next state of the environment is completely determined by the current state and the action executed by the agent. (If the environment is deterministic except for the actions of other agents, then the environment is strategic)

- **Episodic (vs. sequential):** An agent’s action is divided into atomic episodes. Decisions do not depend on previous decisions/actions.

- **Known (vs. unknown):** An environment is considered to be "known" if the agent understands the laws that govern the environment's behavior.

Environment types

- **Static (vs. dynamic):** The environment is unchanged while an agent is deliberating. (The environment is **semidynamic** if the environment itself does not change with the passage of time but the agent's performance score does)

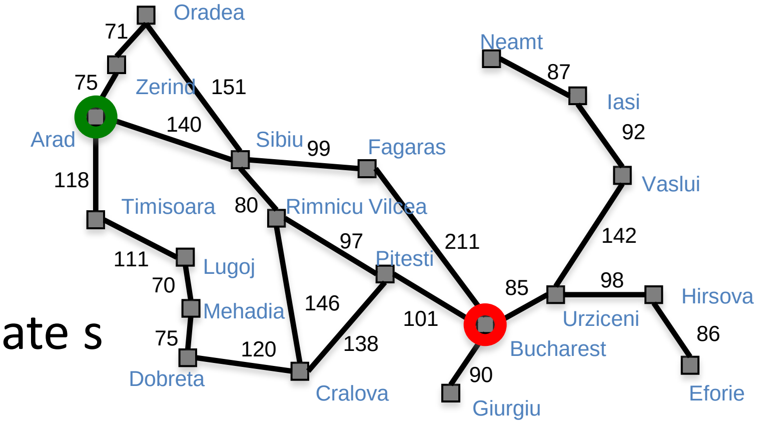
- **Discrete (vs. continuous):** A limited number of distinct, clearly defined percepts and actions.
  - How do we **represent** or **abstract** or **model** the world?

- **Single agent (vs. multi-agent):** An agent operating by itself in an environment. Does the other agent interfere with my performance measure?

Review State Space Search Chapter 3

- Problem Formulation (3.1, 3.3)
- Blind (Uninformed) Search (3.4)
  - Depth-First, Breadth-First, Iterative Deepening
  - Uniform-Cost, Bidirectional (if applicable)
  - Time? Space? Complete? Optimal?
- Heuristic Search (3.5)
  - A*, Greedy-Best-First

State-Space Problem Formulation



A **problem** is defined by five items:

(1) initial state e.g., "at Arad“

(2) actions *Actions(s)* = set of actions avail. in state s

(3) transition model Results(s,a) = state that results from action a in state s

Alt: successor function *S(x)* = set of action–state pairs

- e.g., *S(Arad) = {<Arad → Zerind, Zerind>, … }*

(4) goal test, (or goal state)

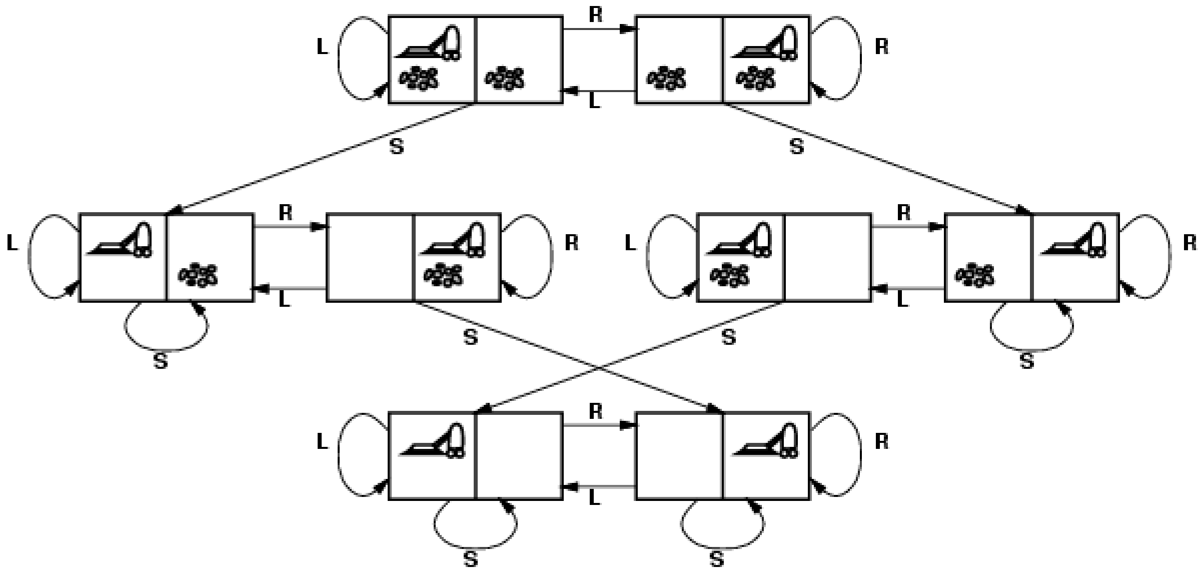
e.g., *x* = "at Bucharest”, *Checkmate(x)*

(5) path cost (additive)

- e.g., sum of distances, number of actions executed, etc.
- *c(x,a,y)* is the step cost, assumed to be ≥ 0 (and often, assumed to be $\geq \varepsilon > 0$)

A **solution** is a sequence of actions leading from the initial state to a goal state

Vacuum world state space graph

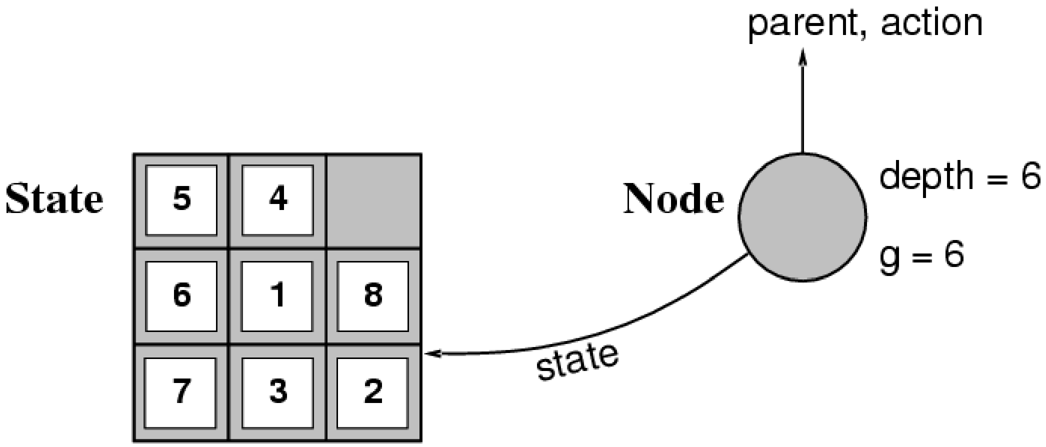


- states? discrete: dirt and robot locations
- initial state? any
- actions? *Left*, *Right*, *Suck*
- transition model? as shown on graph
- goal test? no dirt at all locations
- path cost? 1 per action

Implementation: states vs. nodes

- A state is a (representation of) a physical configuration

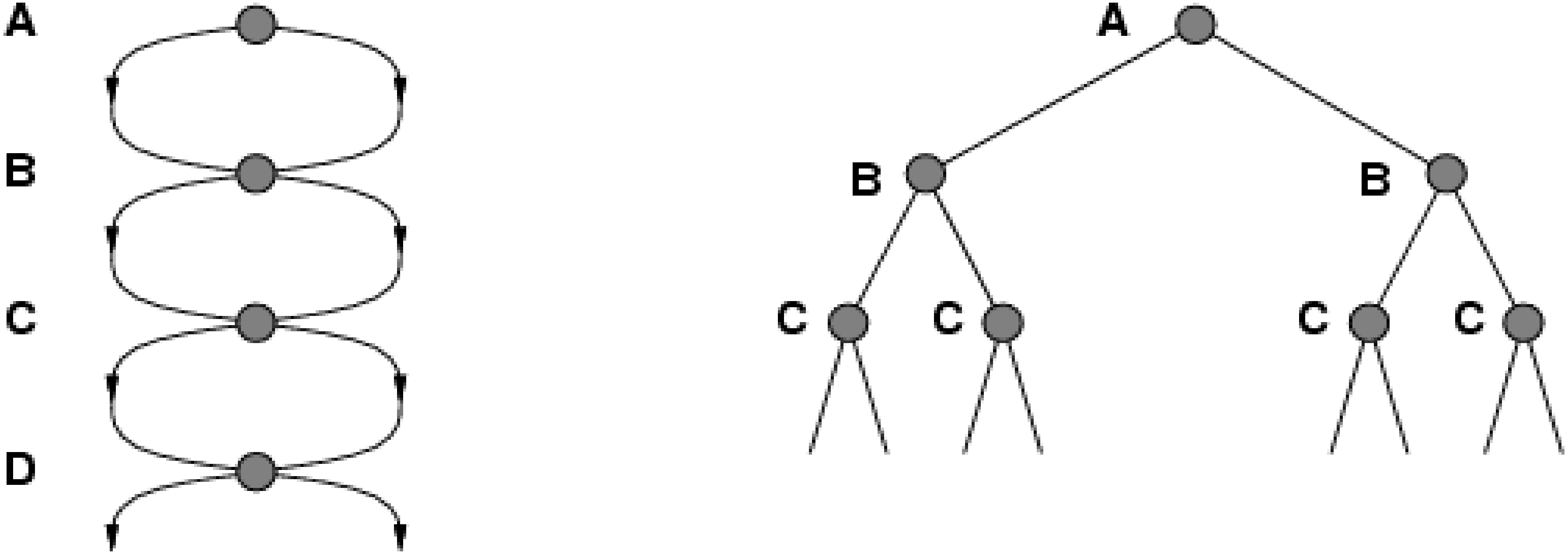
- A node is a data structure constituting part of a search tree
- A node contains info such as:
  - state, parent node, action, path cost *g(x)*, depth, etc.



- The `Expand` function creates new nodes, filling in the various fields using the `Actions(S)` and `Result(S,A)`functions associated with the problem.

Tree search vs. Graph search Review Fig. 3.7, p. 77

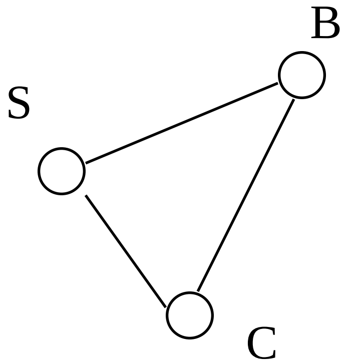
- Failure to detect repeated states can turn a linear problem into an exponential one!
- Test is often implemented as a hash table.



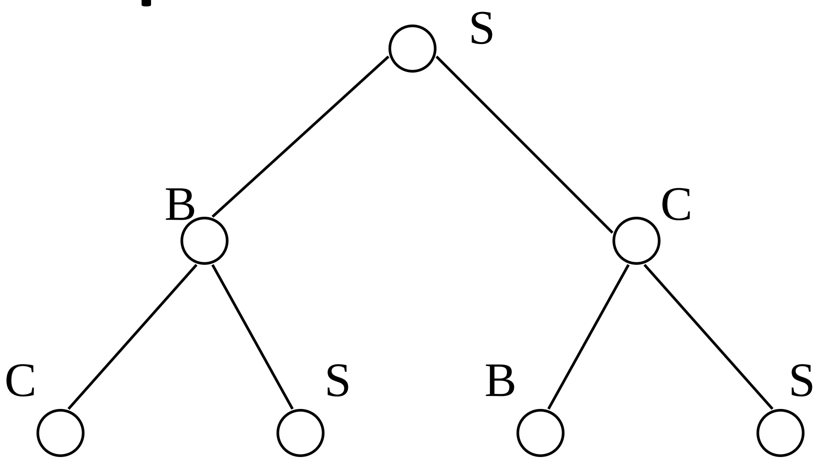
Tree search vs. Graph search
Review Fig. 3.7, p. 77

- What R&N call Tree Search vs. Graph Search
  - (And we follow R&N **exactly** in this class)
  - Has **NOTHING** to do with searching trees vs. graphs
- **Tree Search** = do **NOT** remember visited nodes
  - Exponentially slower search, but memory efficient
- **Graph Search** = **DO** remember visited nodes
  - Exponentially faster search, but memory blow-up
- **CLASSIC Comp Sci TIME-SPACE TRADE-OFF**

Solutions to Repeated States



State Space



Example of a Search Tree

- Graph search ← faster, but memory inefficient
  - never generate a state generated before
    - must keep track of all possible states (uses a lot of memory)
    - e.g., 8-puzzle problem, we have 9! = 362,880 states
    - approximation for DFS/DLS: only avoid states in its (limited) memory: avoid infinite loops by checking path back to root.
  - "visited?" test usually implemented as a hash table

Checking for identical nodes (1)

Check if a node is already in fringe-frontier

- It is “easy” to check if a node is already in the fringe/frontier (recall fringe = frontier = open = queue)
  - Keep a hash table holding all fringe/frontier nodes
    - Hash size is same O(.) as priority queue, so hash does not increase overall space O(.)
    - Hash time is O(1), so hash does not increase overall time O(.)
  - When a node is expanded, remove it from hash table (it is no longer in the fringe/frontier)
  - For each resulting child of the expanded node:
    - If child is not in hash table, add it to queue (fringe) and hash table
    - Else if an old lower- or equal-cost node is in hash, discard the new higher- or equal-cost child
    - Else remove and discard the old higher-cost node from queue and hash, and add the new lower-cost child to queue and hash

Always do this for tree or graph search in BFS, UCS, GBFS, and A*

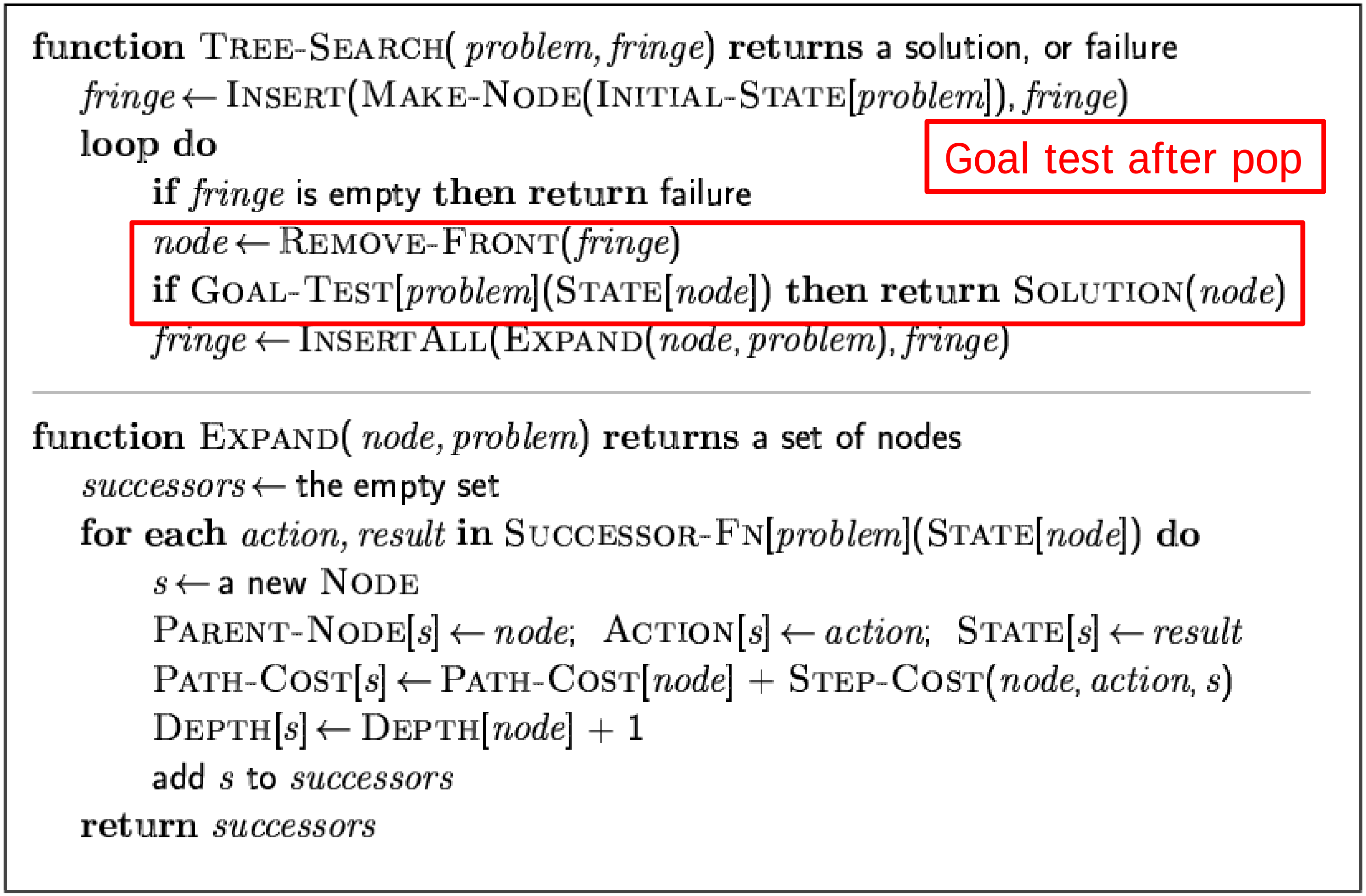
Checking for identical nodes (2)

Check if a node is in explored/expanded

- It is memory-intensive [$O(b^d)$ or $O(b^m)$]to check if a node is in explored/expanded (recall explored = expanded = closed)
  - Keep a hash table holding all explored/expanded nodes (hash table may be HUGE!!)
- When a node is expanded, add it to hash (explored)
- For each resulting child of the expanded node:
  - If child is not in hash table or in fringe/frontier, then add it to the queue (fringe/frontier) and process normally (BFS normal processing differs from UCS normal processing, but the ideas behind checking a node for being in explored/expanded are the same).
  - Else discard any redundant node.

Always do this for graph search

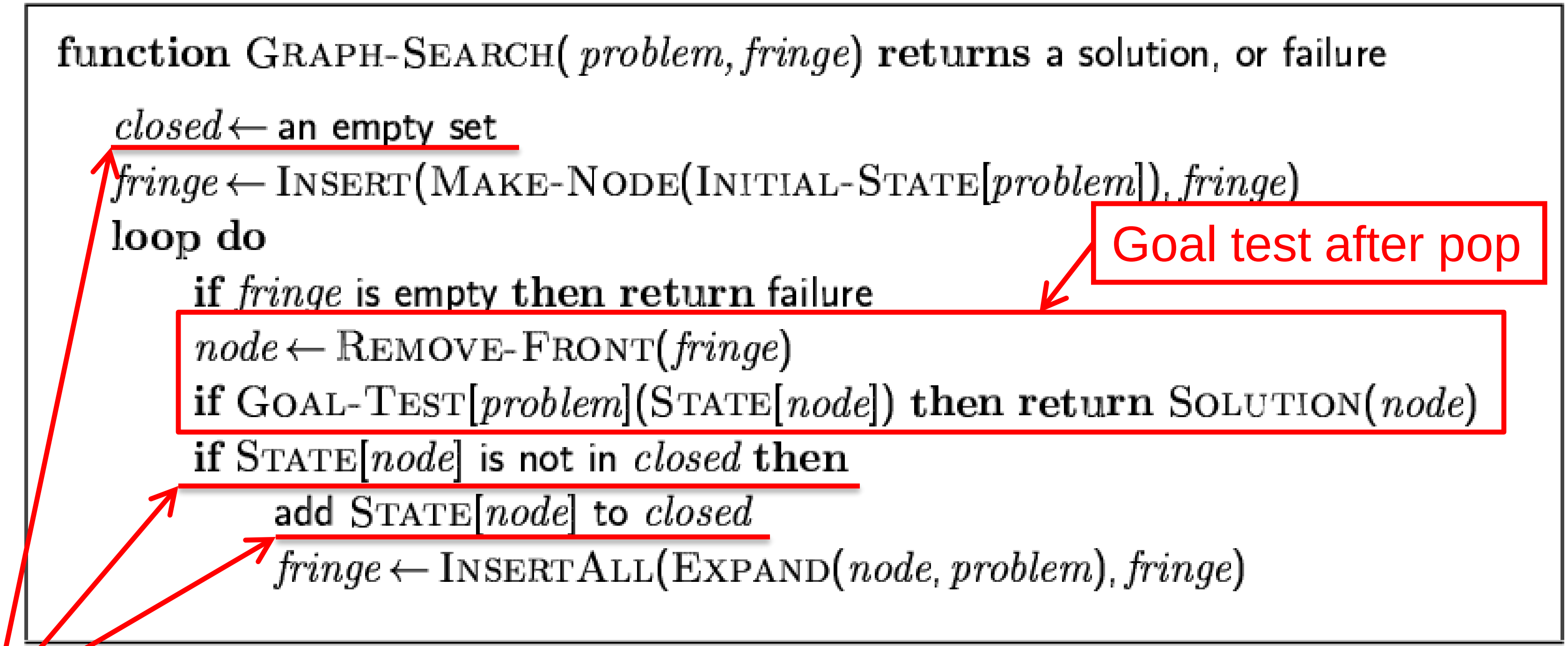
General tree search
Do not remember visited nodes



```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        fringe ← INSERTALL(EXPAND(node, problem), fringe)

function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;   ACTION[s] ← action;   STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

Goal test after pop

General graph search (R&N Fig. 3.7) Do remember visited nodes



```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

Goal test after pop

These three statements change tree search to graph search.

When to do Goal-Test? (Summary)

- For BFS, the goal test is done when the child node is generated.
  - Not an optimal search in the general case.
- For DLS, IDS, and DFS as in Fig. 3.17, goal test is done in the recursive call.
  - Result is that children are generated then iterated over. For each child DLS, is called recursively, goal-test is done first in the callee, and the process repeats.
  - More efficient search goal-tests children as generated. We follow your text.
- For DFS as in Fig. 3.7, goal test is done when node is popped.
  - Search behavior depends on how the LIFO queue is implemented.
- For UCS and A*(next lecture), goal test when node removed from queue.
  - This avoids finding a short expensive path before a long cheap path.
- Bidirectional search can use either BFS or UCS.
  - Goal-test is search fringe intersection, see additional complications below
- For GBFS (next lecture) the behavior is the same either way
  - h(goal)=0 so any goal will be at the front of the queue anyway.

Breadth-first graph search

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier)   /* chooses the shallowest node in frontier */
      add node.STATE to explored
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          if child.STATE is not in explored or frontier then
              if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
              frontier ← INSERT(child, frontier)
```

Goal test before push

Figure 3.11 Breadth-first search on a graph.

Properties of breadth-first search

- Complete? Yes, it always reaches a goal (if *b* is finite)
- Time? $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$

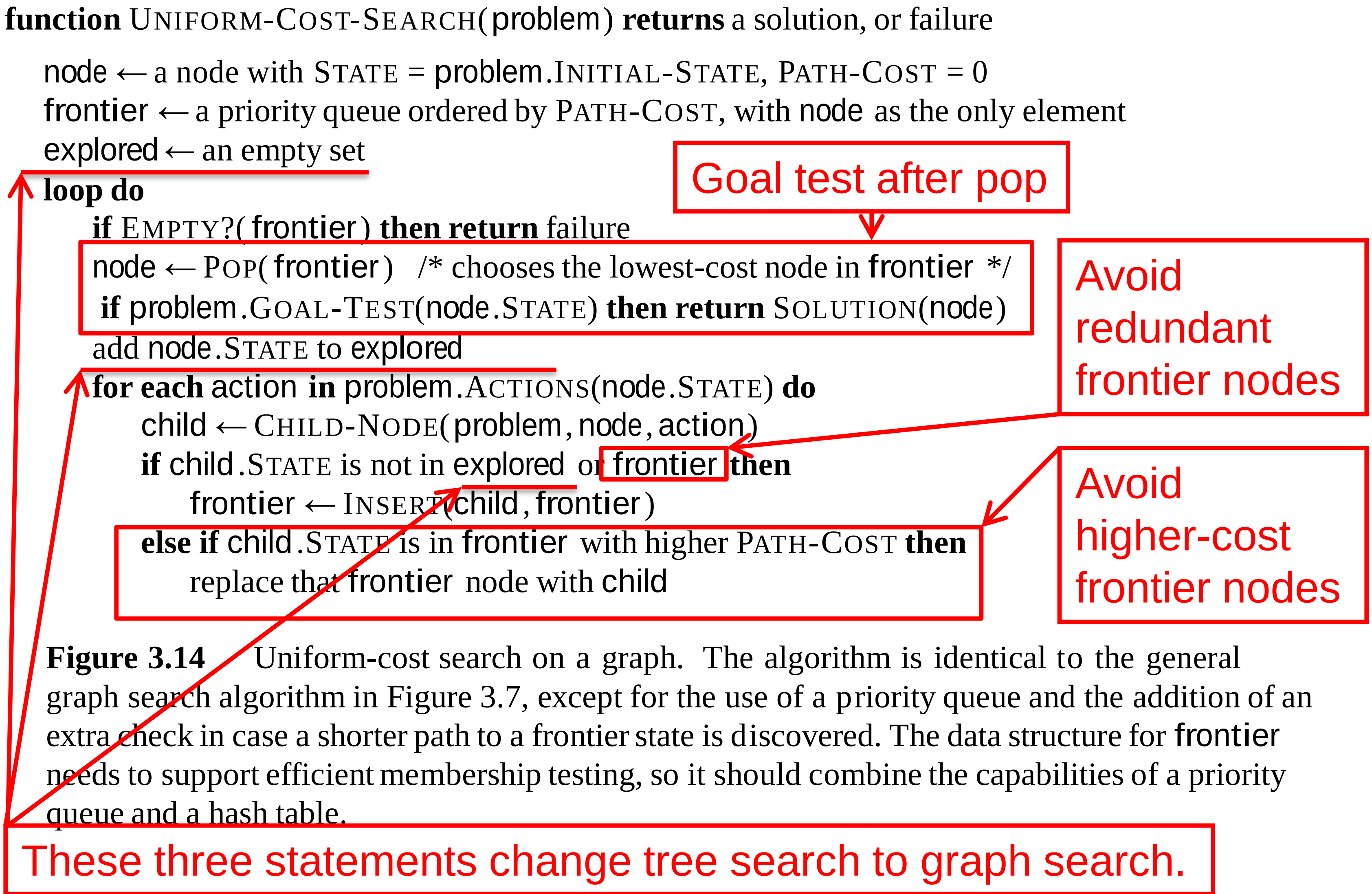
  (this is the number of nodes we generate)
- Space? $O(b^d)$

  (keeps every node in memory, either in frontier or on a path to frontier).
- Optimal? No, for general cost functions.

  Yes, if cost is a non-decreasing function only of depth.
  - With $f(d) \geq f(d-1)$, e.g., step-cost = constant:
    - All optimal goal nodes occur on the same level
    - Optimal goals are always shallower than non-optimal goals
    - An optimal goal will be found before any non-optimal goal

- Usually Space is the bigger problem (more than time)

Uniform cost search (R&N Fig. 3.14) [A* is identical except queue sort = f(n)]



```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
   node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
   frontier ← a priority queue ordered by PATH-COST, with node as the only element
   explored ← an empty set
   loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
       if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
      add node.STATE to explored
      for each action in problem.ACTIONS(node.STATE) do
         child ← CHILD-NODE(problem, node, action)
         if child.STATE is not in explored or frontier then
            frontier ← INSERT(child, frontier)
         else if child.STATE is in frontier with higher PATH-COST then
            replace that frontier node with child
```

Goal test after pop

Avoid redundant frontier nodes

Avoid higher-cost frontier nodes

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for frontier needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

These three statements change tree search to graph search.

Uniform-cost search

Implementation: *Frontier* = queue ordered by path cost.
Equivalent to breadth-first if all step costs all equal.

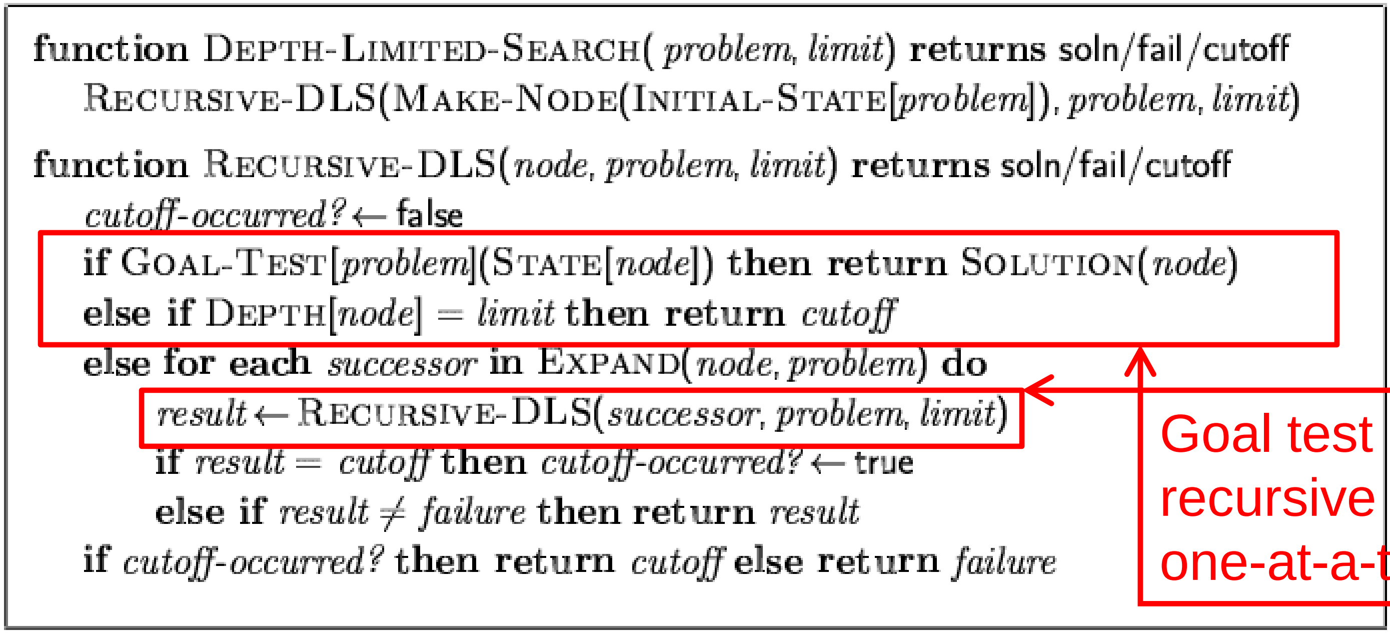
- Complete? Yes, if b is finite and step cost ≥ ε > 0.
  (otherwise it can get stuck in infinite regression)

- Time? # of nodes with path cost ≤ cost of optimal solution.
  $O(b^{\lfloor 1+C^*/\varepsilon \rfloor}) \approx O(b^{d+1})$

- Space? # of nodes with path cost ≤ cost of optimal solution.
  $O(b^{\lfloor 1+C^*/\varepsilon \rfloor}) \approx O(b^{d+1})$.

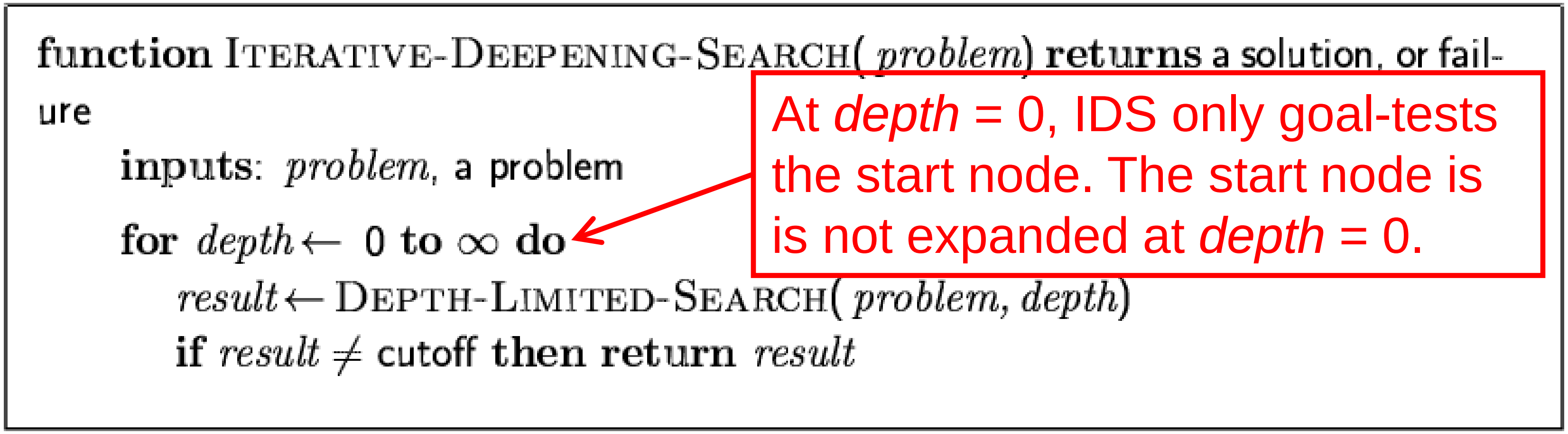
- Optimal? Yes, for step cost ≥ ε > 0.

Depth-limited search & IDS (R&N Fig. 3.17-18)



```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
    RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred? ← false
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    else if DEPTH[node] = limit then return cutoff
    else for each successor in EXPAND(node, problem) do
        result ← RECURSIVE-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred? ← true
        else if result ≠ failure then return result
    if cutoff-occurred? then return cutoff else return failure
```

Goal test in recursive call, one-at-a-time



```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or failure
    inputs: problem, a problem

    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
```

At *depth* = 0, IDS only goal-tests the start node. The start node is is not expanded at *depth* = 0.

Properties of iterative deepening search

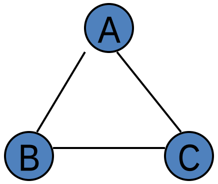
- Complete? Yes
- Time? O(b^d)
- Space? *O(bd)*
- Optimal? No, for general cost functions.
  Yes, if cost is a non-decreasing function only of depth.

Generally the preferred uninformed search strategy.

Depth-First Search (R&N Section 3.4.3)

- Your textbook is ambiguous about DFS.
  - The second paragraph of R&N 3.4.3 states that DFS is an instance of Fig. 3.7 using a LIFO queue. Search behavior may differ depending on how the LIFO queue is implemented (as separate pushes, or one concatenation).
  - The third paragraph of R&N 3.4.3 says that an alternative implementation of DFS is a recursive algorithm that calls itself on each of its children, as in the Depth-Limited Search of Fig. 3.17 (above).
- **For quizzes and exams, we will follow Fig. 3.17.**

Properties of depth-first search



- Complete? No: fails in loops/infinite-depth spaces
  - Can modify to avoid loops/repeated states along path
    - check if current nodes occurred before on path to root
  - Can use graph search (remember all nodes ever seen)
    - problem with graph search: space is exponential, not linear
  - Still fails in infinite-depth spaces (may miss goal entirely)
- Time? $O(b^m)$ with m =maximum depth of space
  - Terrible if m is much larger than d
  - If solutions are dense, may be much faster than BFS
- Space? $O(bm)$, i.e., linear space!
  - Remember a single path + expanded unexplored nodes
- Optimal? No: It may find a non-optimal goal first

Bidirectional Search

- Idea
  - simultaneously search forward from S and backwards from G
  - stop when both “meet in the middle”
  - need to keep track of the intersection of 2 open sets of nodes

- What does searching backwards from G mean
  - need a way to specify the predecessors of G
    - this can be difficult,
    - e.g., predecessors of checkmate in chess?
  - what if there are multiple goal states?
  - what if there is only a goal test, no explicit list?

- Complexity
  - time complexity is best: $O(2\ b^{(d/2)}) = O(b^{(d/2)})$
  - memory complexity is the same as time complexity

Bi-Directional Search

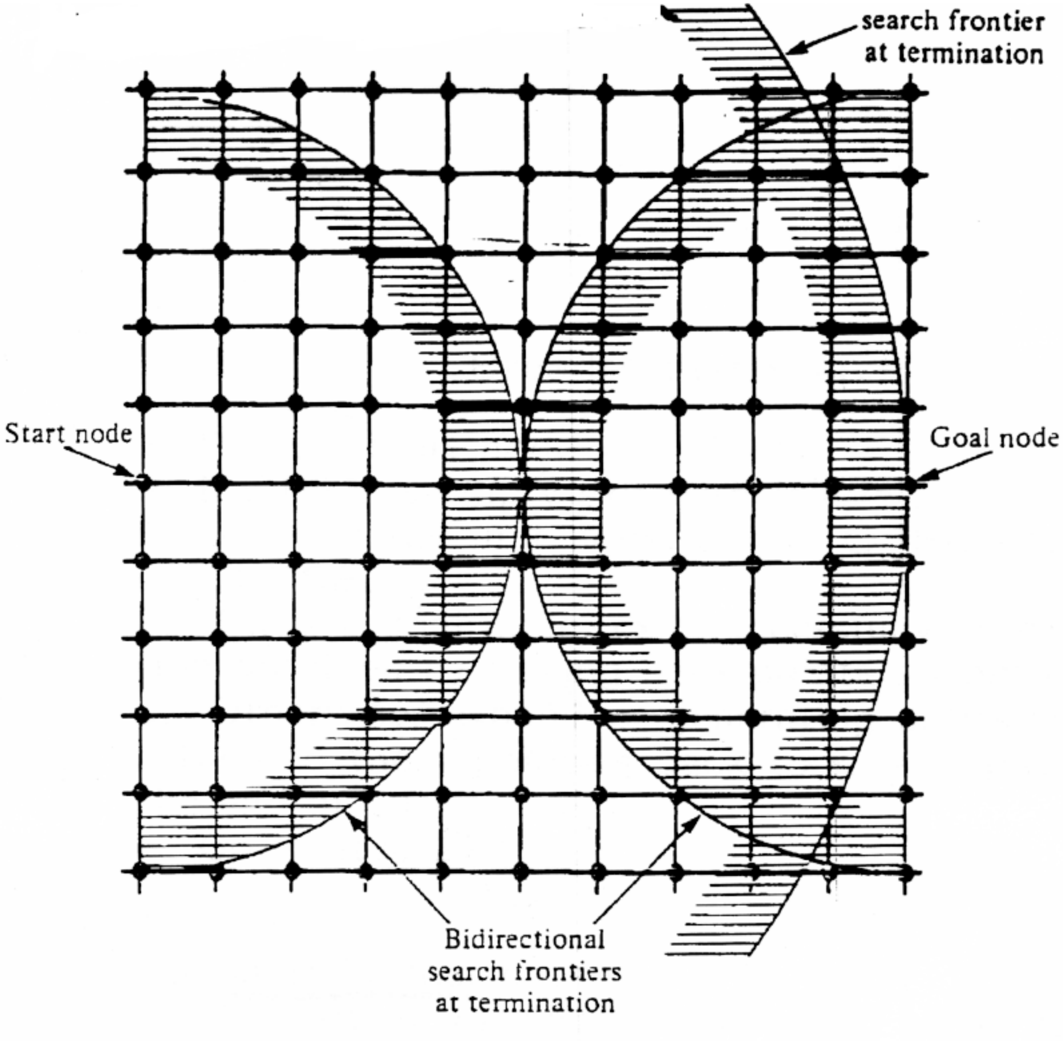


Fig. 2.10 Bidirectional and unidirectional breadth-first searches.

Blind Search Strategies (3.4)

- Depth-first: Add successors to front of queue
- Breadth-first: Add successors to back of queue
- Uniform-cost: Sort queue by path cost g(n)
- Depth-limited: Depth-first, cut off at limit *l*
- Iterated-deepening: Depth-limited, increasing *l*
- Bidirectional: Breadth-first from goal, too.

- **Review “Example hand-simulated search”**
  - Lecture on “Uninformed Search”

Search strategy evaluation

- A search **strategy** is defined by **the order of node expansion**

- Strategies are evaluated along the following dimensions:
  - **completeness**: does it always find a solution if one exists?
  - **time complexity**: number of nodes generated
  - **space complexity**: maximum number of nodes in memory
  - **optimality**: does it always find a least-cost solution?

- Time and space complexity are measured in terms of
  - ***b****:* maximum branching factor of the search tree
  - ***d:*** depth of the least-cost solution
  - ***m*****:** maximum depth of the state space (may be ∞)
  - (UCS: **C*:** true cost to optimal goal; **$\varepsilon > 0$:** minimum step cost)

Summary of algorithms Fig. 3.21, p. 91

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening DLS	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{\lfloor 1+C^*/\varepsilon \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{\lfloor 1+C^*/\varepsilon \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Generally the preferred uninformed search strategy

There are a number of footnotes, caveats, and assumptions.
See Fig. 3.21, p. 91.

[a] complete if b is finite

[b] complete if step costs $\geq \varepsilon > 0$

[c] optimal if step costs are all identical
(also if path cost non-decreasing function of depth only)

[d] if both directions use breadth-first search
(also if both directions use uniform-cost search with step costs $\geq \varepsilon > 0$)

Summary

- Generate the search space by applying actions to the initial state and all further resulting states.
- Problem: initial state, actions, transition model, goal test, step/path cost
- Solution: sequence of actions to goal
- Tree-search (don't remember visited nodes) vs. Graph-search (do remember them)
- Search strategy evaluation: b, d, m (UCS: C*, ε)
  - Complete? Time? Space? Optimal?

Heuristic function (3.5)

- Heuristic:
  - Definition: a commonsense rule (or set of rules) intended to increase the probability of solving some problem
  - “using rules of thumb to find answers”

- Heuristic function h(n)
  - Estimate of (optimal) cost from n to goal
  - Defined using only the *state* of node *n*
  - h(n) = 0 if n is a goal node
  - Example: straight line distance from n to Bucharest
    - Note that this is not the true state-space distance
    - It is an estimate – actual state-space distance can be higher

  - Provides problem-specific knowledge to the search algorithm

Relationship of search algorithms

- Notation:
  - *g(n)* = known cost so far to reach *n*
  - *h(n)* = estimated optimal cost from *n* to goal
  - *h*(n)* = true optimal cost from *n* to goal (unknown to agent)
  - *f(n) = g(n)+h(n)* = estimated optimal total cost through *n*

- Uniform cost search: sort frontier by *g(n)*
- Greedy best-first search: sort frontier by *h(n)*
- A* search: sort frontier by *f(n) = g(n) + h(n)*
  - Optimal for admissible / consistent heuristics
  - Generally the preferred heuristic search framework
  - Memory-efficient versions of A* are available: RBFS, SMA*

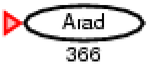
Greedy best-first search

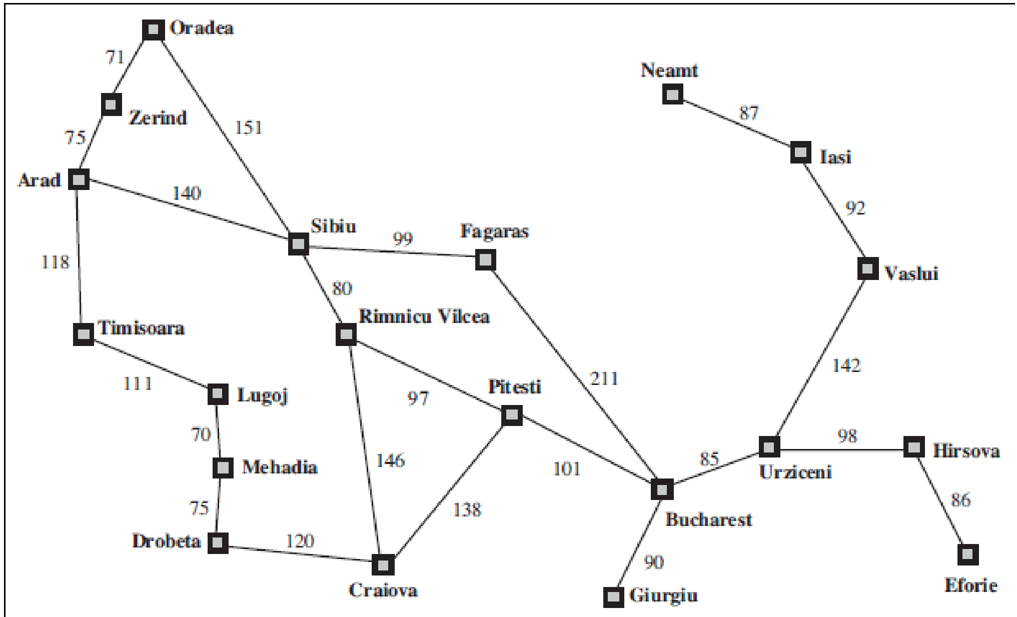
- *h(n)* = estimate of cost from *n* to *goal*
  - e.g., *h(n)* = straight-line distance from *n* to Bucharest

- Greedy best-first search expands the node that appears to be closest to goal.
  - Sort queue by *h(n)*

- Not an optimal search strategy
  - May perform well in practice

Greedy best-first search example

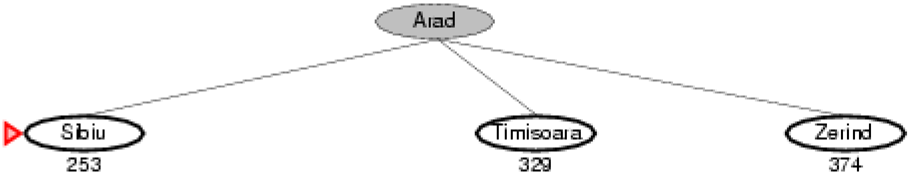


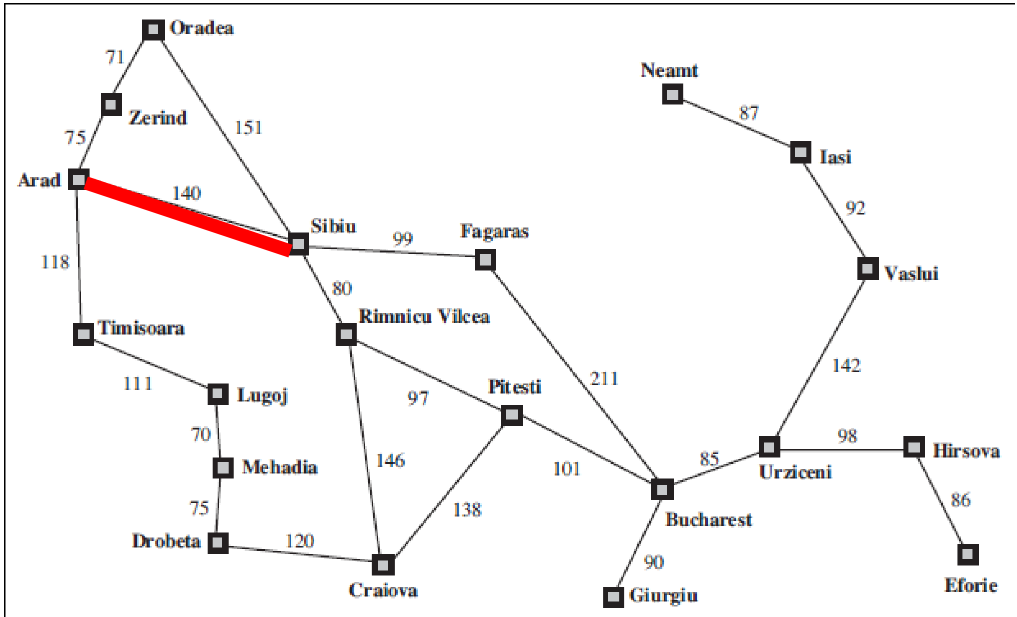


Values of h_{SLD}—straight-line distances to Bucharest.

Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best-first search example

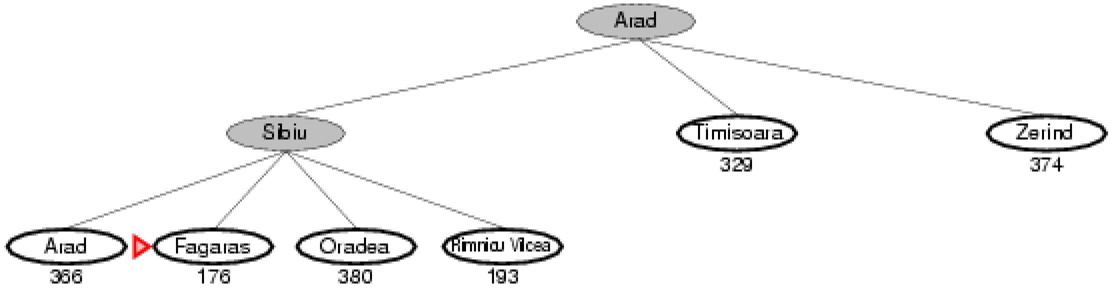


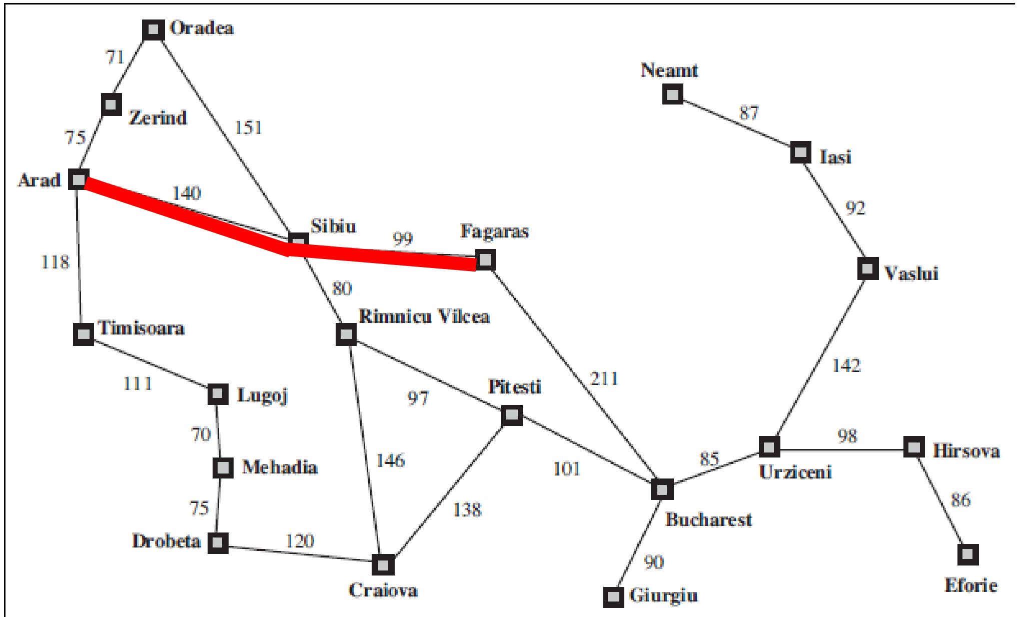


Values of h_{SLD}—straight-line distances to Bucharest.

Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best-first search example

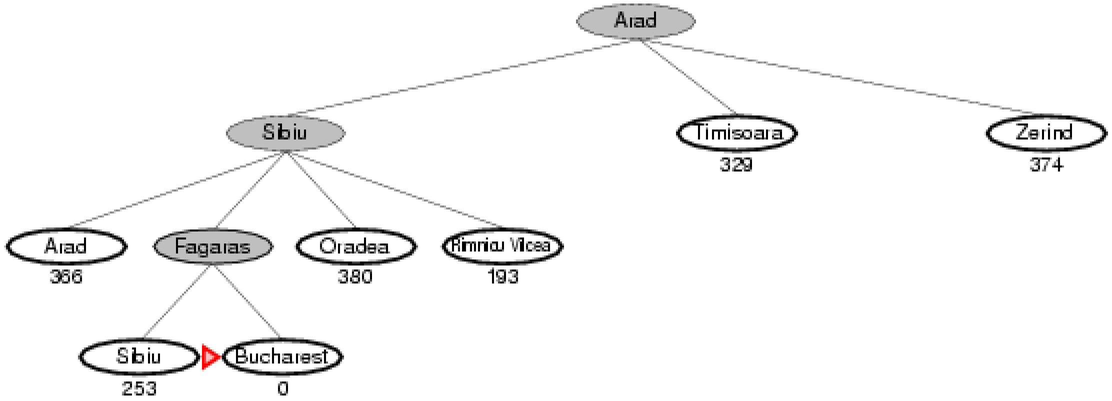


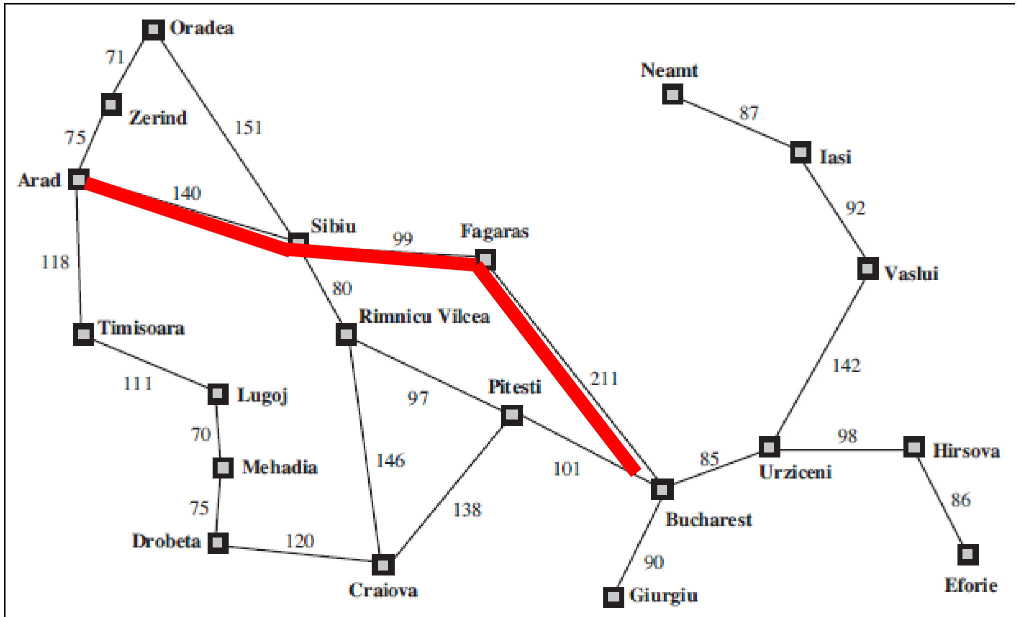


Values of h_{SLD}—straight-line distances to Bucharest.

Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best-first search example

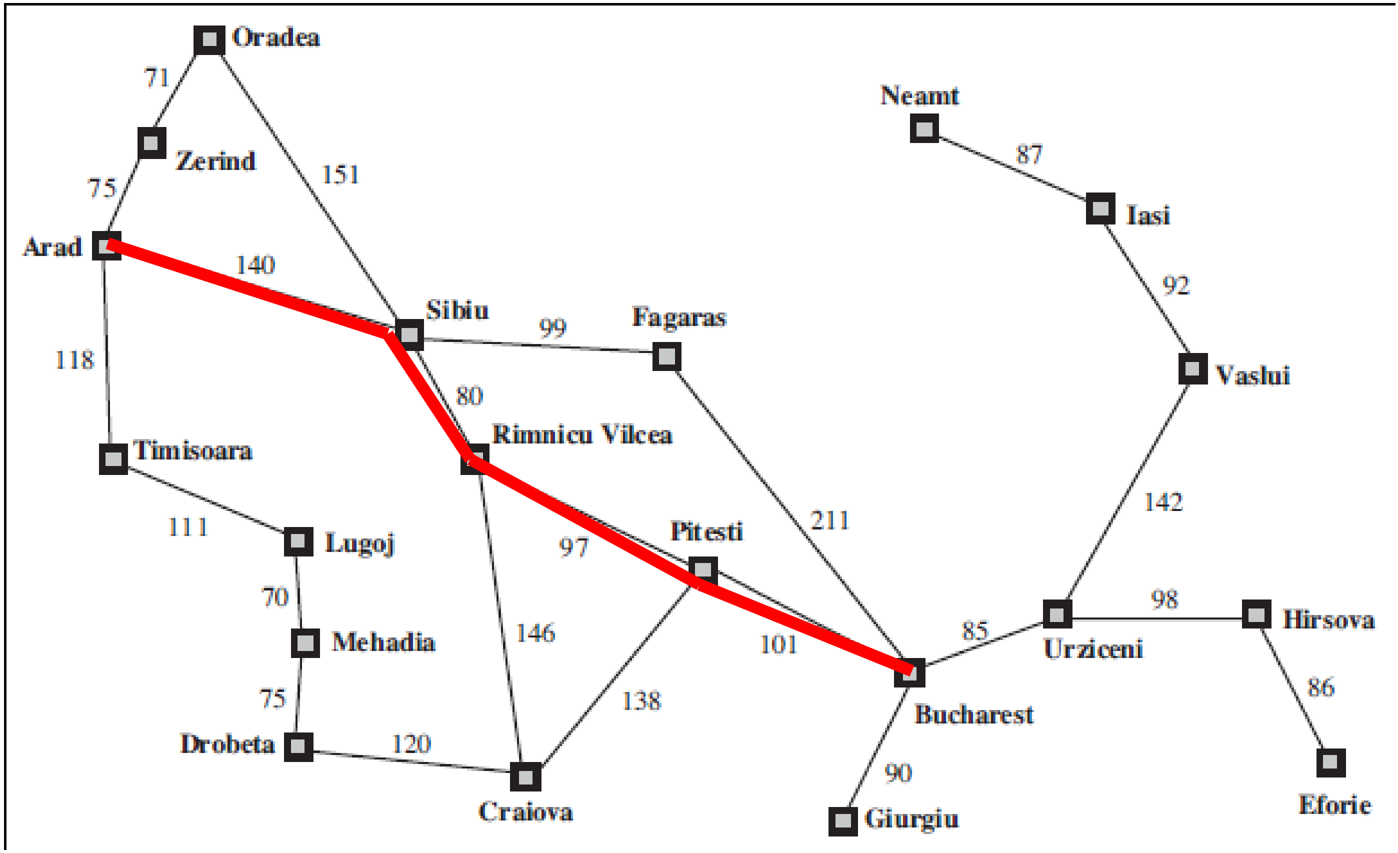




Values of h_{SLD}—straight-line distances to Bucharest.

Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Optimal Path



Values of h_{SLD}— straight-line distances to Bucharest.

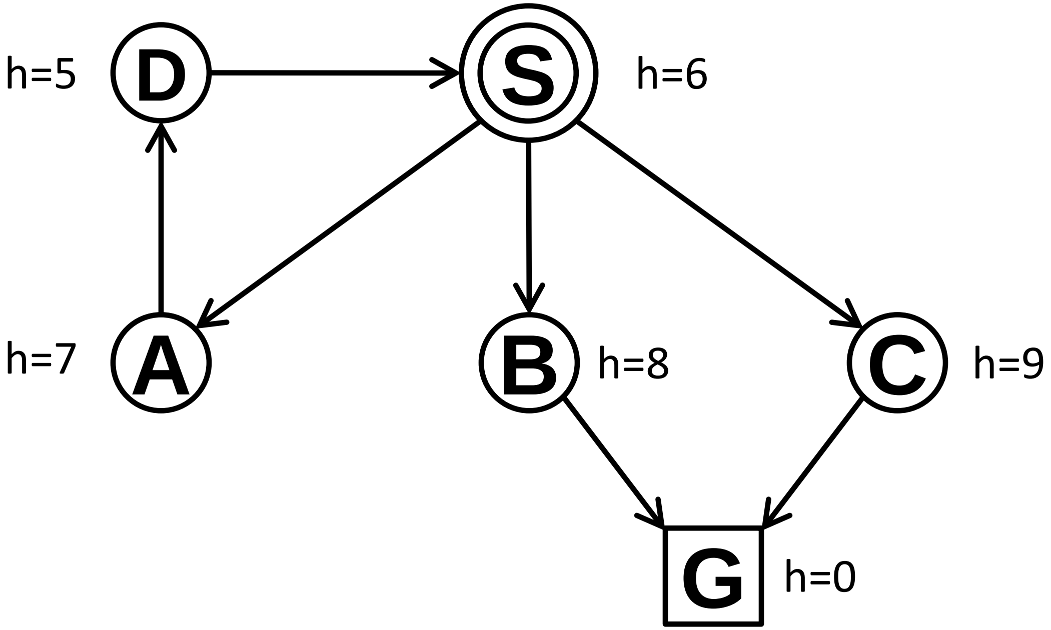
Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy Best-first Search

With tree search, will become stuck in this loop

Order of node expansion: S A D S A D S A D. . . .

Path found: none Cost of path found: none .



Properties of greedy best-first search

- Complete?
  - Tree version can get stuck in loops.
  - Graph version is complete in finite spaces.
- Time? *$O(b^m)$*
  - A good heuristic can give **dramatic** improvement
- Space? *$O(b^m)$*
  - Graph search keeps all nodes in memory
  - A good heuristic can give **dramatic** improvement
- Optimal? No
  - E.g., Arad → Sibiu → Rimnicu Vilcea → Pitesti → Bucharest is shorter!

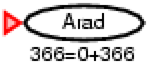
A^* search

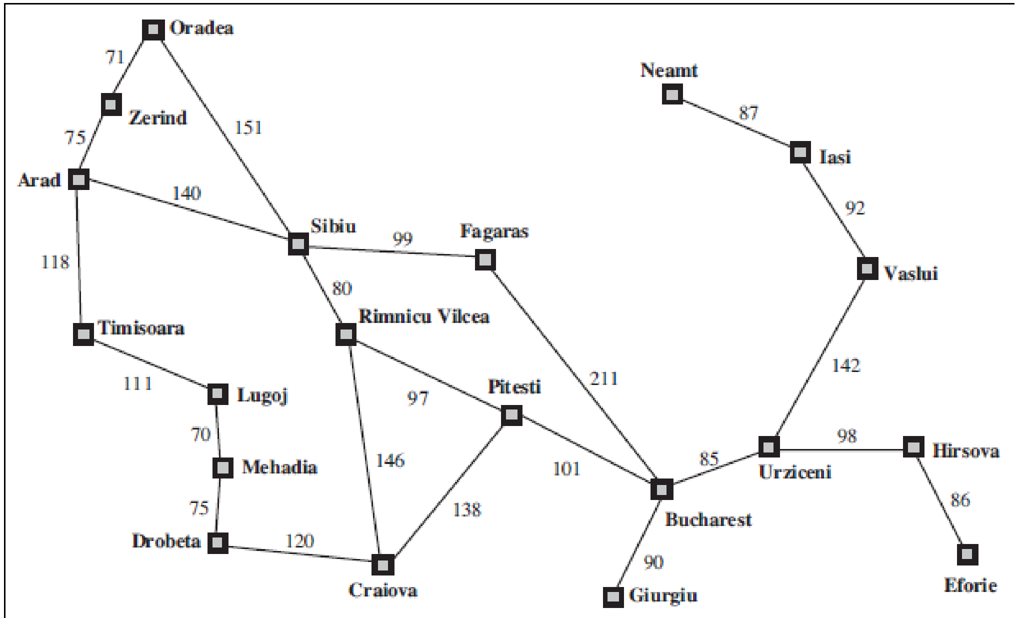
- Idea: avoid paths that are already expensive
  - Generally the preferred simple heuristic search
  - Optimal if heuristic is:

    admissible (tree search)/consistent (graph search)
- Evaluation function *f(n) = g(n) + h(n)*
  - g(n) = known path cost so far to node n.
  - h(n) = estimate of (optimal) cost to goal from node n.
  - f(n) = g(n)+h(n)

    = estimate of total cost to goal through node n.
- *Priority queue sort function = f(n)*

A* tree search example





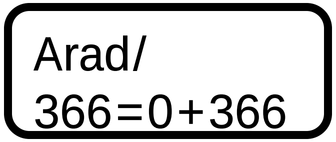
Values of h_{SLD}—straight-line distances to Bucharest.

Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* tree search example: Simulated queue. City/f=g+h

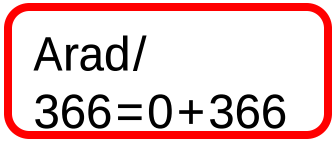
- Next:
- Children:
- Expanded:
- Frontier: Arad/366=0+366

A^* tree search example: Simulated queue. City/f=g+h



Arad/
366=0+366

A^* tree search example: Simulated queue. City/f=g+h

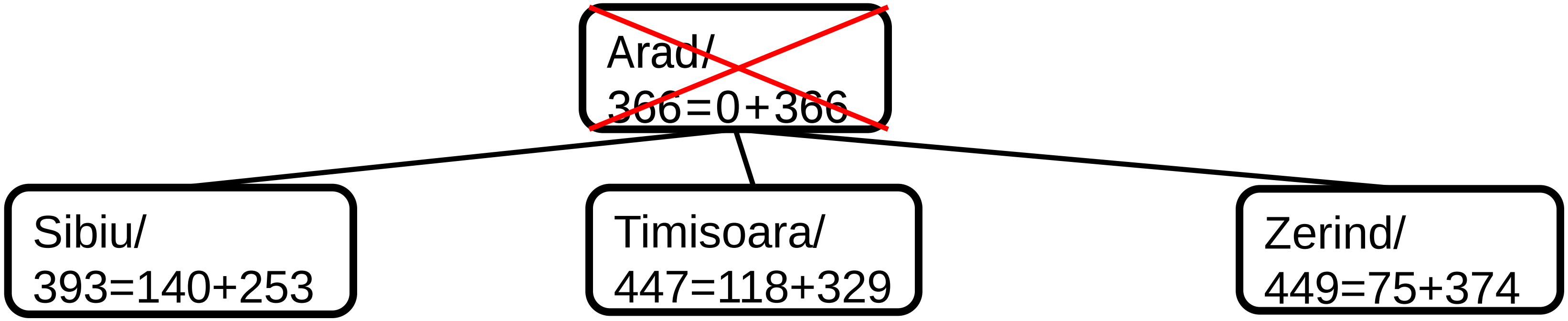


Arad/
366=0+366

A^* tree search example: Simulated queue. City/f=g+h

- Next: Arad/366=0+366
- Children: Sibiu/393=140+253, Timisoara/447=118+329, Zerind/449=75+374
- Expanded: Arad/366=0+366
- Frontier: ~~Arad/366=0+366~~, Sibiu/393=140+253, Timisoara/447=118+329, Zerind/449=75+374

A^* tree search example: Simulated queue. City/f=g+h



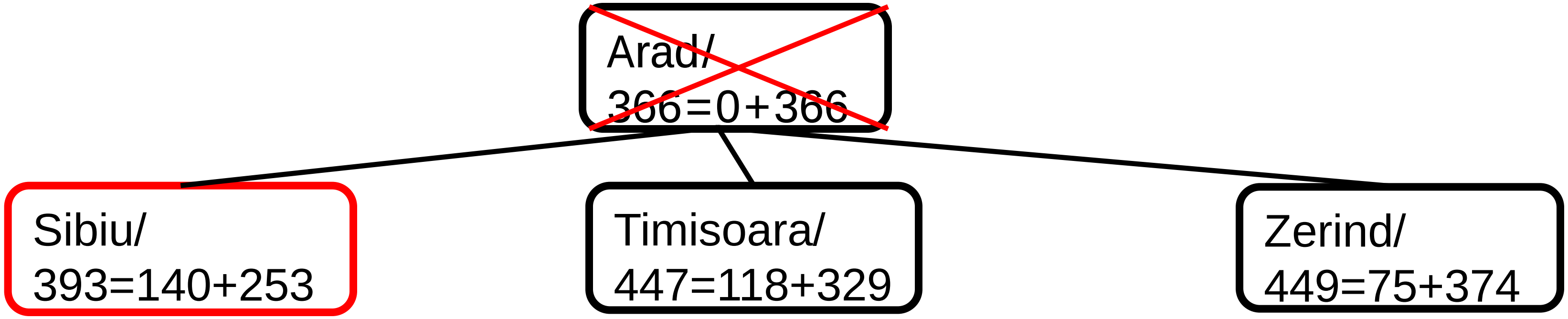
~~Arad/~~
~~366=0+366~~

Sibiu/
393=140+253

Timisoara/
447=118+329

Zerind/
449=75+374

A* tree search example:
Simulated queue. City/f=g+h



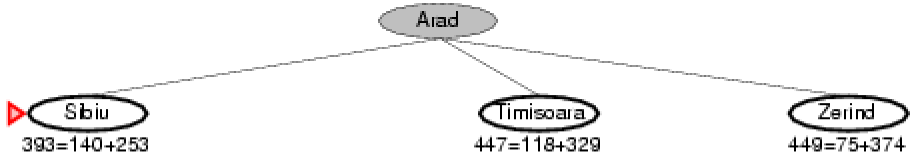
~~Arad/~~
~~366=0+366~~

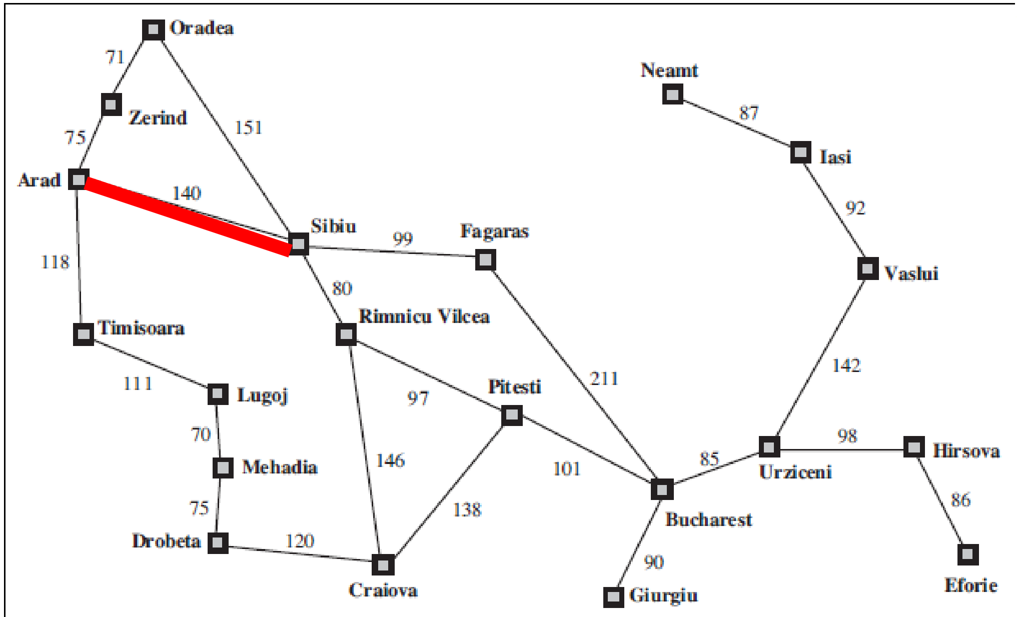
Sibiu/
393=140+253

Timisoara/
447=118+329

Zerind/
449=75+374

A* tree search example





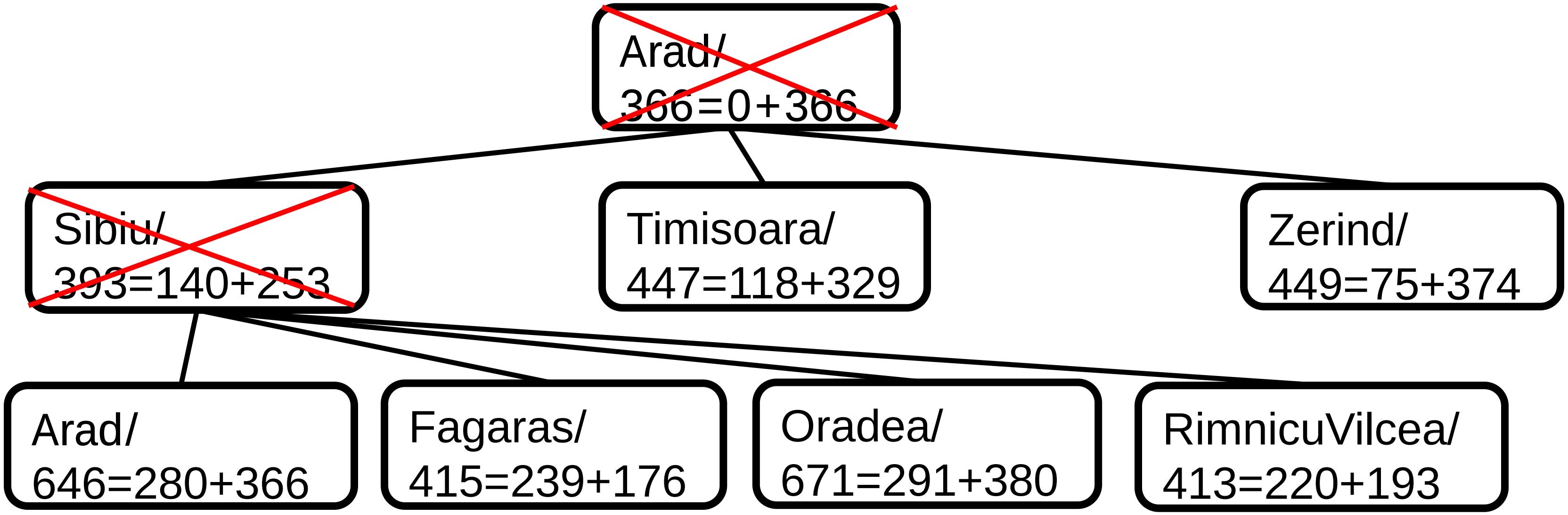
Values of h_{SLD}—straight-line distances to Bucharest.

Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A^* tree search example: Simulated queue. City/f=g+h

- Next: Sibiu/393=140+253
- Children: Arad/646=280+366, Fagaras/415=239+176, Oradea/671=291+380, RimnicuVilcea/413=220+193
- Expanded: Arad/366=0+366, Sibiu/393=140+253
- Frontier: ~~Arad/366=0+366, Sibiu/393=140+253,~~ Timisoara/447=118+329, Zerind/449=75+374, Arad/646=280+366, Fagaras/415=239+176, Oradea/671=291+380, RimnicuVilcea/413=220+193

A^* tree search example: Simulated queue. City/f=g+h



~~Arad/ 366=0+366~~

~~Sibiu/ 393=140+253~~

Timisoara/ 447=118+329

Zerind/ 449=75+374

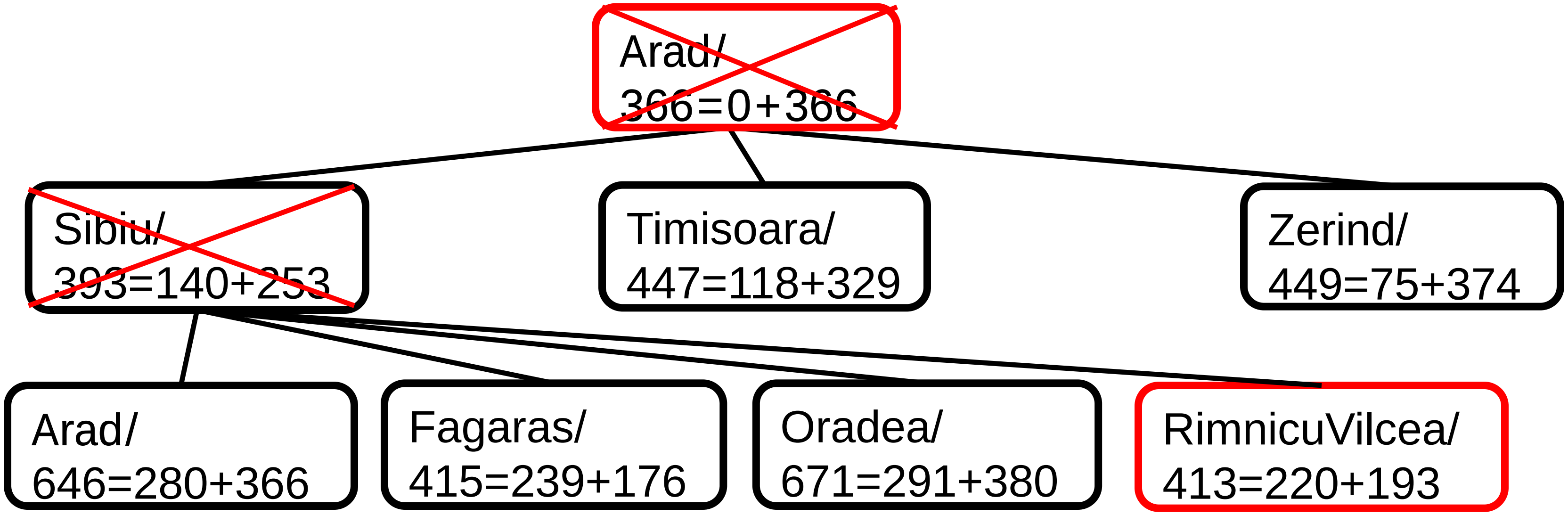
Arad/ 646=280+366

Fagaras/ 415=239+176

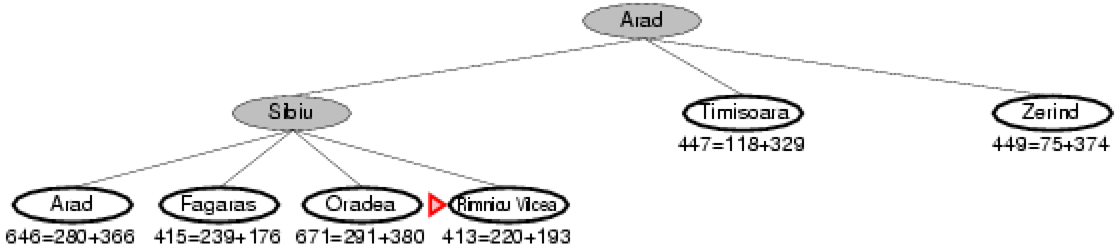
Oradea/ 671=291+380

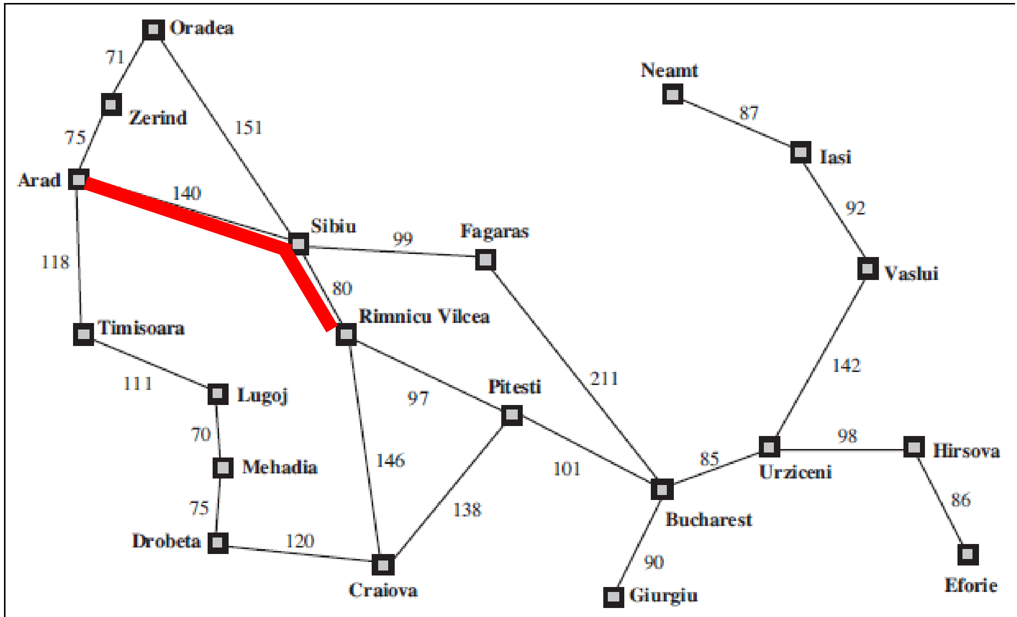
RimnicuVilcea/ 413=220+193

A* tree search example: Simulated queue. City/f=g+h



A^* tree search example





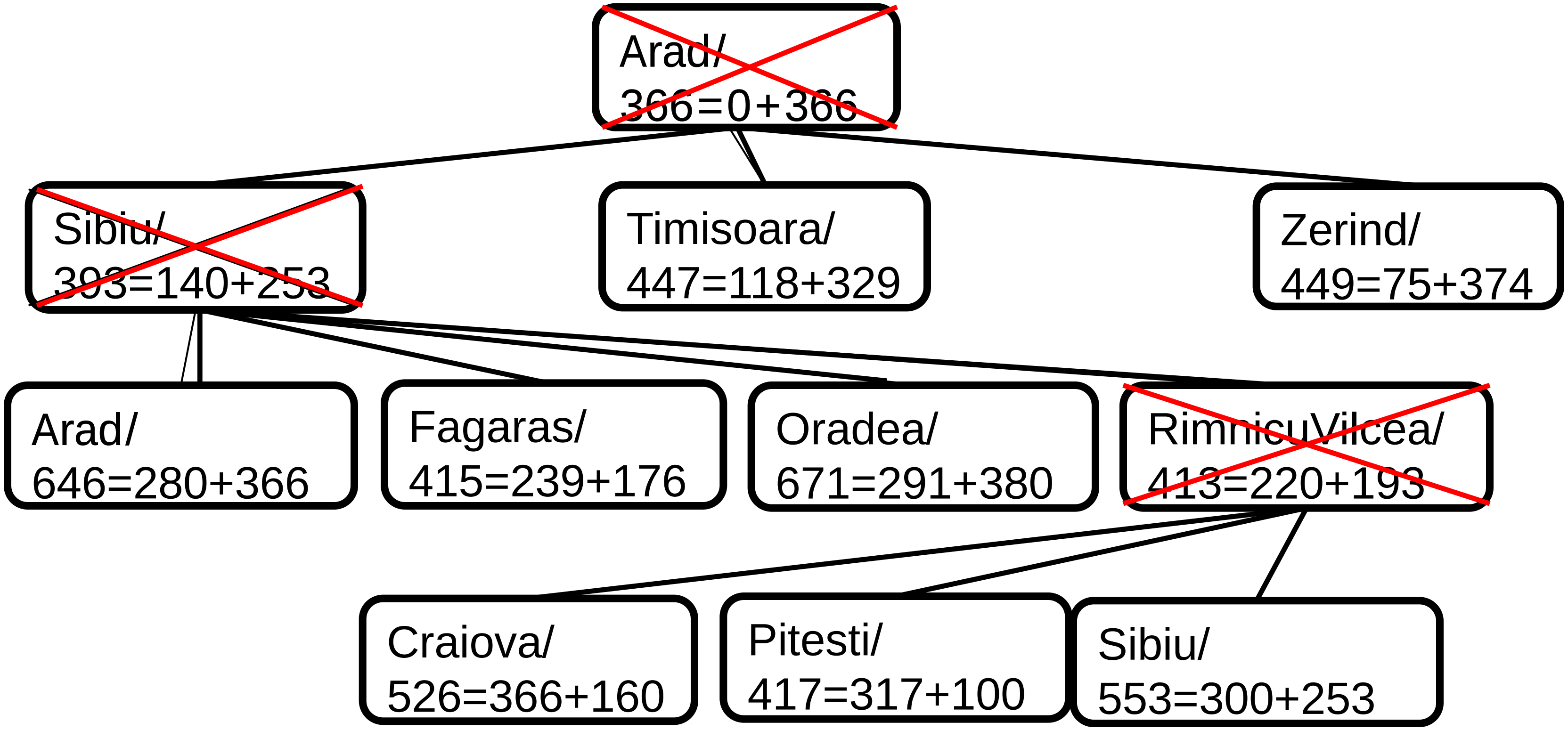
Values of h_{SLD}—straight-line distances to Bucharest.

Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

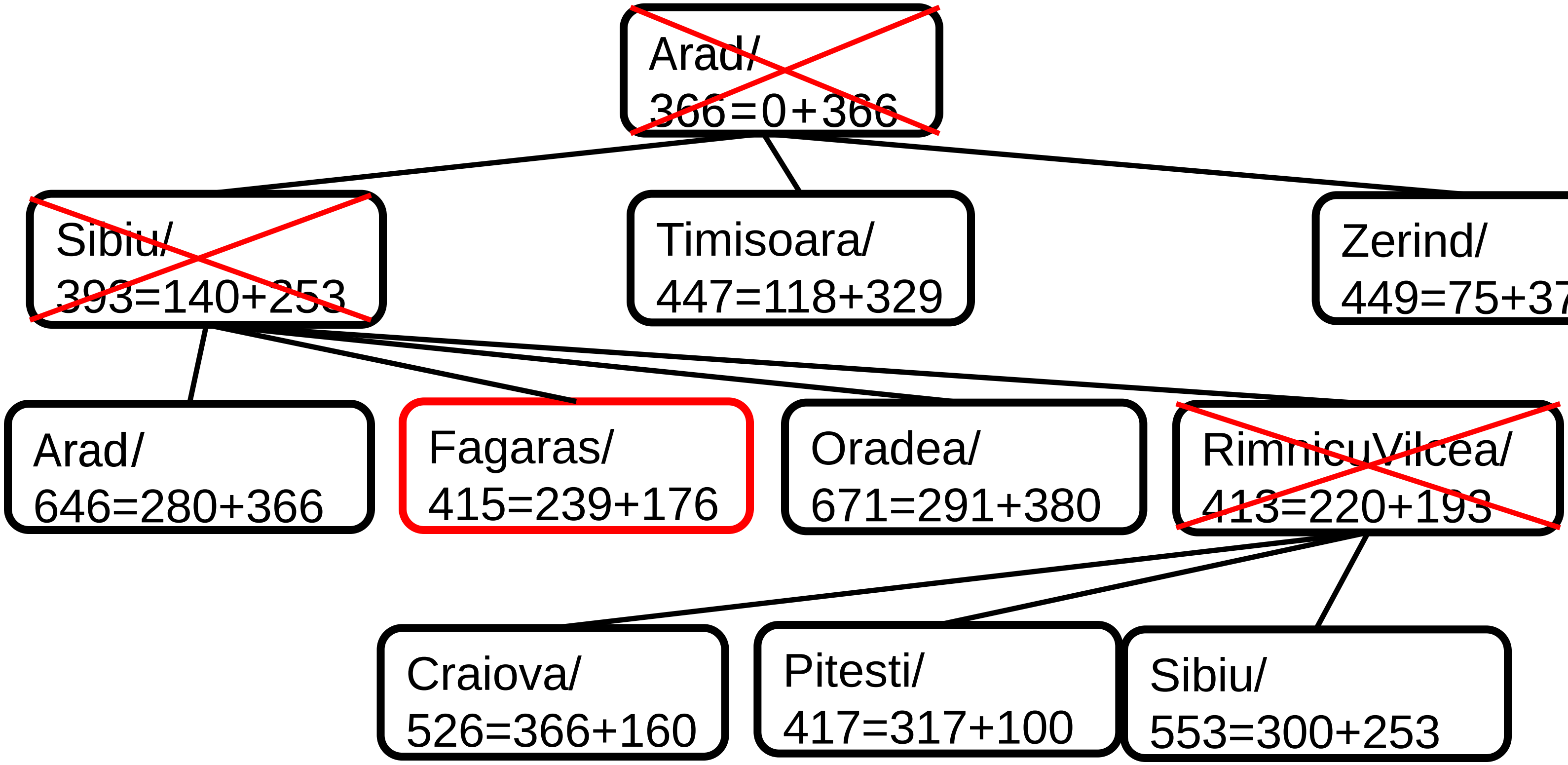
A^* tree search example: Simulated queue. City/f=g+h

- Next: RimnicuVilcea/413=220+193
- Children: Craiova/526=366+160, Pitesti/417=317+100, Sibiu/553=300+253
- Expanded: Arad/366=0+366, Sibiu/393=140+253, RimnicuVilcea/413=220+193
- Frontier: ~~Arad/366=0+366, Sibiu/393=140+253~~, Timisoara/447=118+329, Zerind/449=75+374, Arad/646=280+366, Fagaras/415=239+176, Oradea/671=291+380, ~~RimnicuVilcea/413=220+193~~, Craiova/526=366+160, Pitesti/417=317+100, Sibiu/553=300+253

A* tree search example: Simulated queue. City/f=g+h



A* search example: Simulated queue. City/f=g+h



~~Arad/ 366=0+366~~

~~Sibiu/ 393=140+253~~

Timisoara/ 447=118+329

Zerind/ 449=75+374

Arad/ 646=280+366

Fagaras/ 415=239+176

Oradea/ 671=291+380

~~RimnicuVilcea/ 413=220+193~~

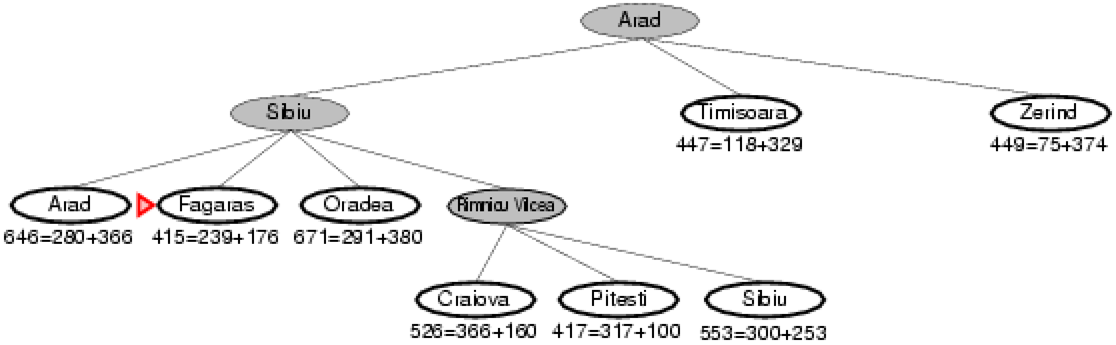
Craiova/ 526=366+160

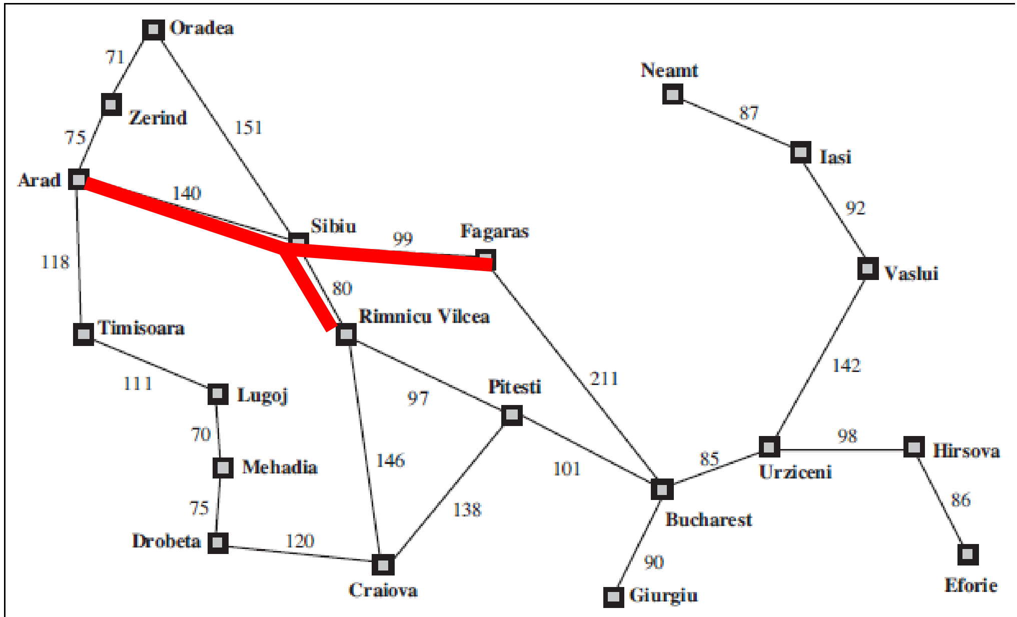
Pitesti/ 417=317+100

Sibiu/ 553=300+253

A* tree search example

Note: The search below did not “back track.” Rather, both arms are being pursued in parallel on the queue.





Values of h_{SLD}—straight-line distances to Bucharest.

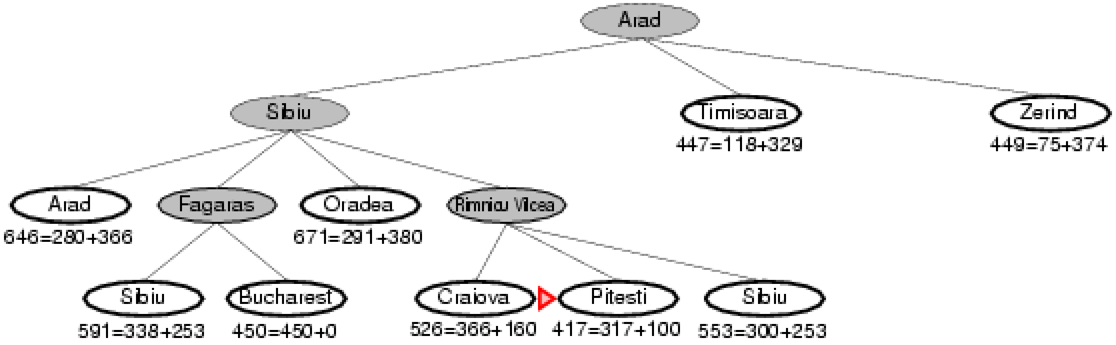
Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

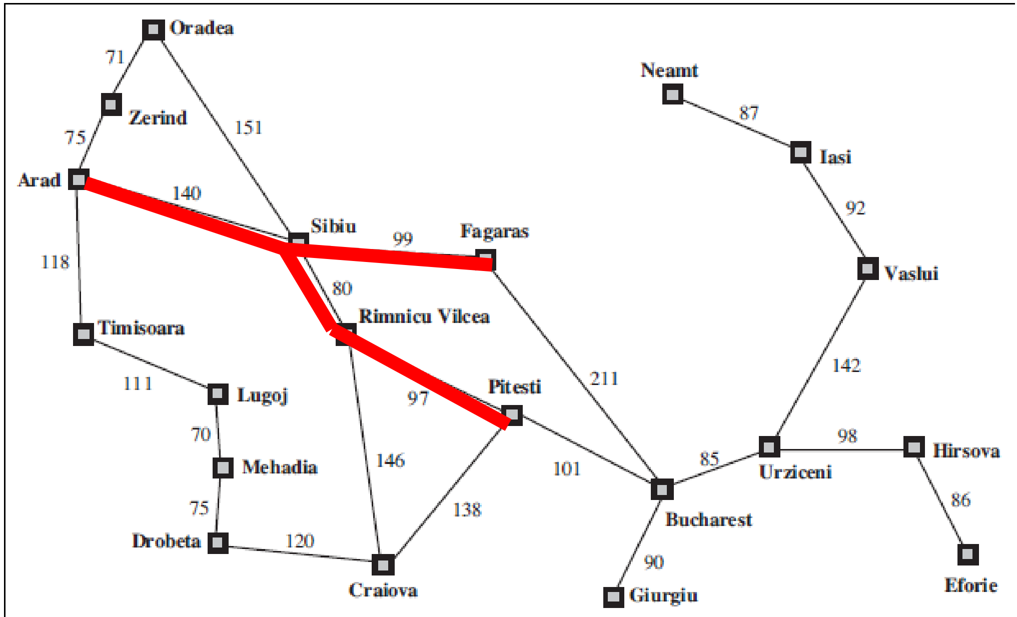
A^* tree search example: Simulated queue. City/f=g+h

- Next: Fagaras/415=239+176
- Children: Bucharest/450=450+0, Sibiu/591=338+253
- Expanded: Arad/366=0+366, Sibiu/393=140+253, RimnicuVilcea/413=220+193, Fagaras/415=239+176
- Frontier: ~~Arad/366=0+366, Sibiu/393=140+253~~, Timisoara/447=118+329, Zerind/449=75+374, Arad/646=280+366, ~~Fagaras/415=239+176~~, Oradea/671=291+380, ~~RimnicuVilcea/413=220+193~~, Craiova/526=366+160, Pitesti/417=317+100, Sibiu/553=300+253, Bucharest/450=450+0, Sibiu/591=338+253

A* tree search example

Note: The search below did not “back track.” Rather, both arms are being pursued in parallel on the queue.





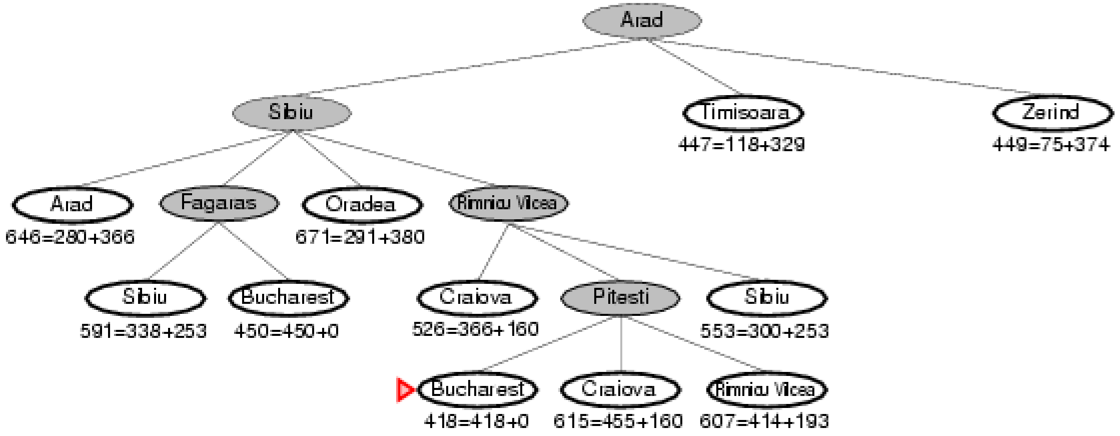
Values of h_{SLD}—straight-line distances to Bucharest.

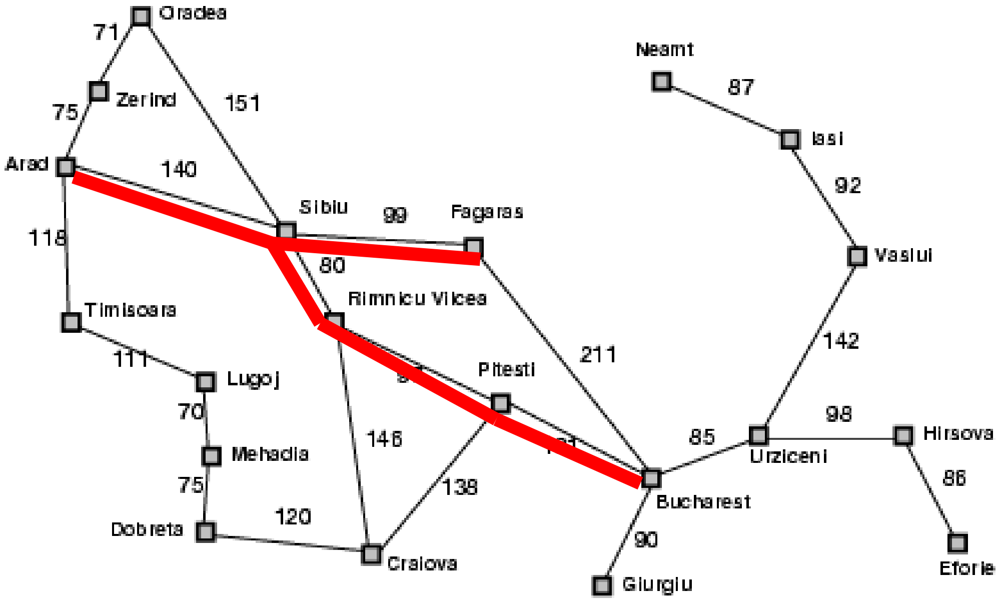
Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A^* tree search example: Simulated queue. City/f=g+h

- Next: Pitesti/417=317+100
- Children: Bucharest/418=418+0, Craiova/615=455+160, RimnicuVilcea/607=414+193
- Expanded: Arad/366=0+366, Sibiu/393=140+253, RimnicuVilcea/413=220+193, Fagaras/415=239+176, Pitesti/417=317+100
- Frontier: ~~Arad/366=0+366, Sibiu/393=140+253,~~ Timisoara/447=118+329, Zerind/449=75+374, Arad/646=280+366, ~~Fagaras/415=239+176~~, Oradea/671=291+380, ~~RimnicuVilcea/413=220+193,~~ Craiova/526=366+160, ~~Pitesti/417=317+100~~, Sibiu/553=300+253, Bucharest/450=450+0, Sibiu/591=338+253, Bucharest/418=418+0, Craiova/615=455+160, RimnicuVilcea/607=414+193

A* tree search example





Straight–line distance to Bucharest

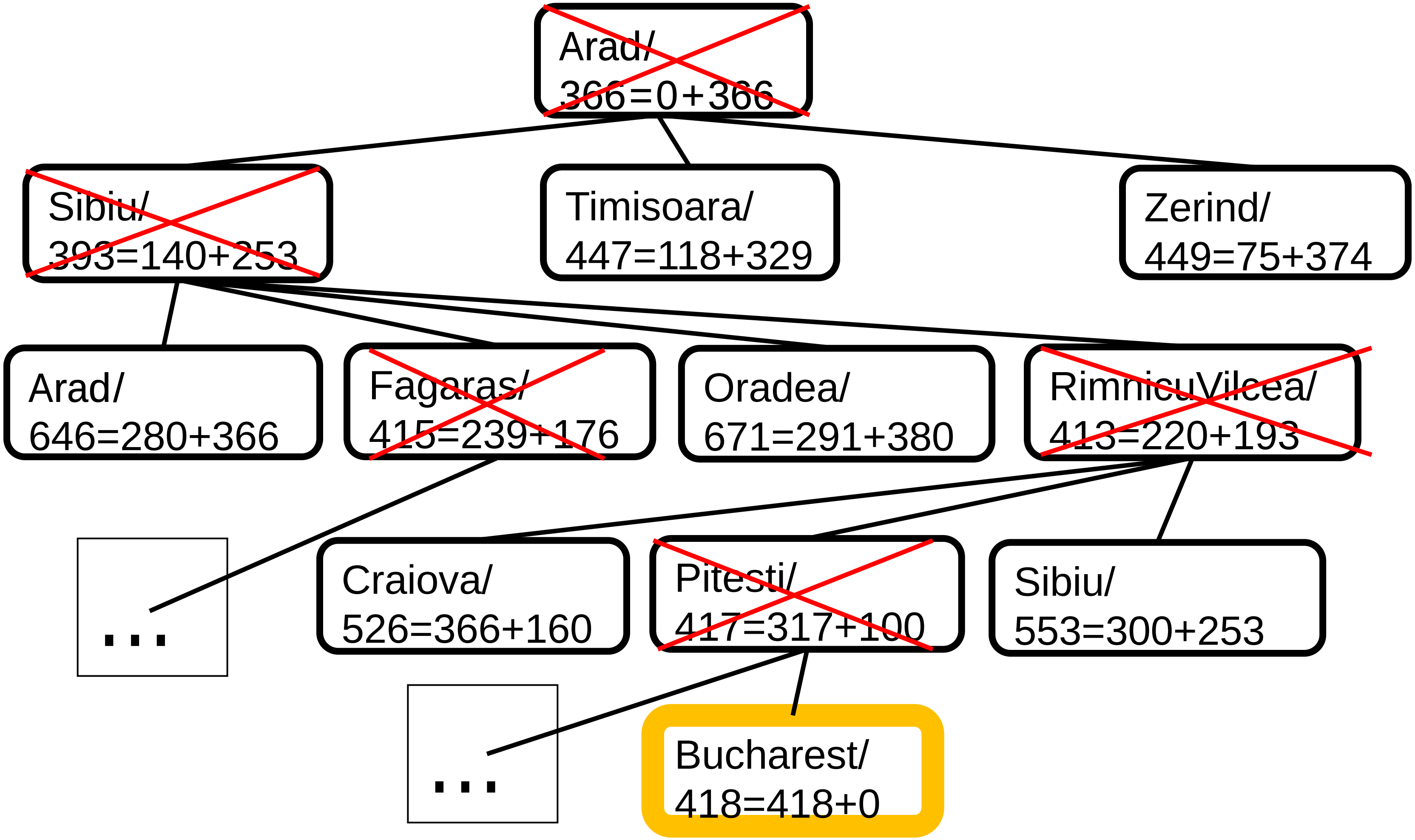
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* tree search example: Simulated queue. City/f=g+h

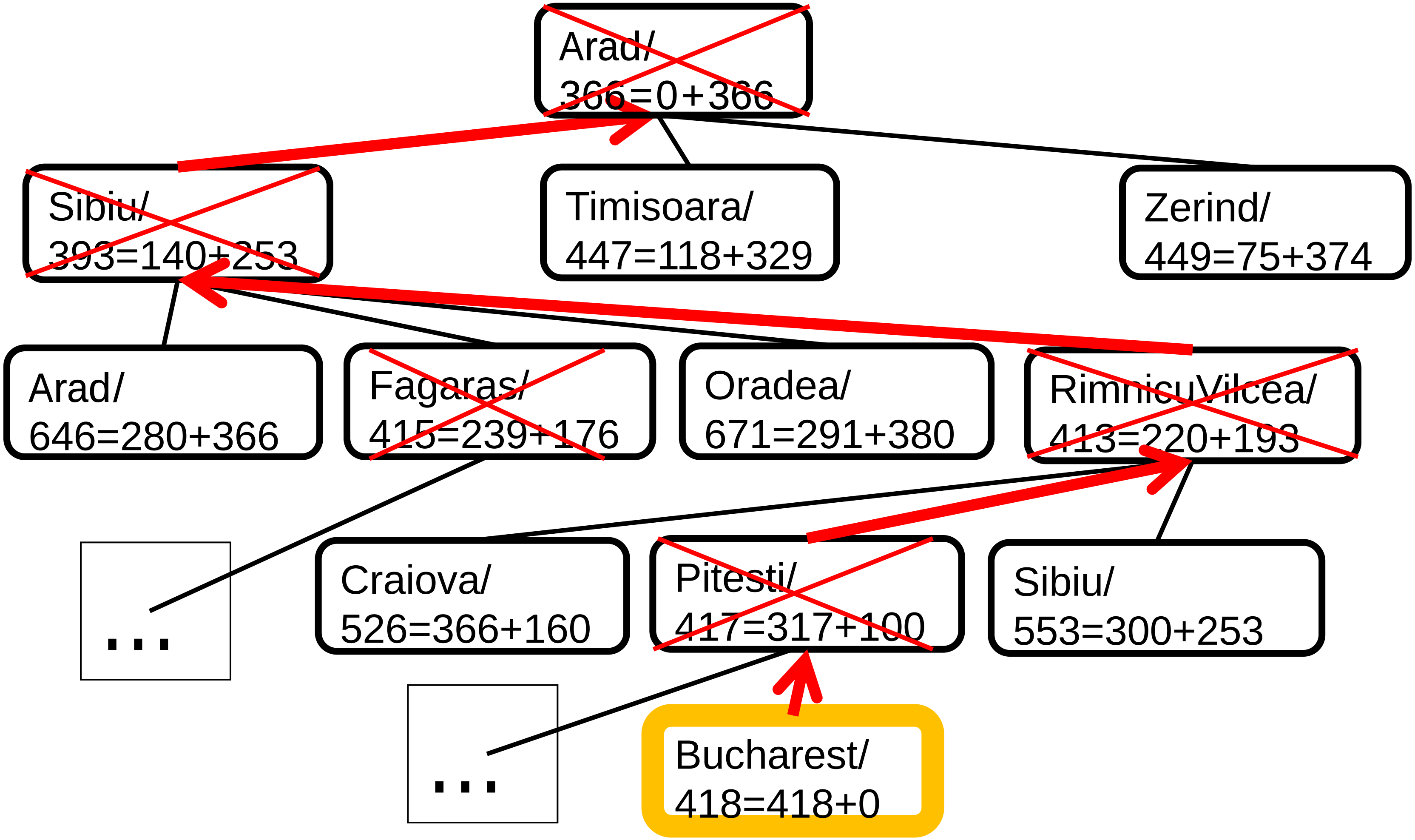
- Next: Bucharest/418=418+0
- Children: **None; goal test succeeds.**
- Expanded: Arad/366=0+366, Sibiu/393=140+253, RimnicuVilcea/413=220+193, Fagaras/415=239+176, Pitesti/417=317+100, Bucharest/418=418+0
- Frontier: ~~Arad/366=0+366, Sibiu/393=140+253,~~ Timisoara/447=118+329, Zerind/449=75+374, Arad/646=280+366, ~~Fagaras/415=239+176,~~ Oradea/671=291+380, ~~RimnicuVilcea/413=220+193,~~ Craiova/526=366+160, ~~Pitesti/417=317+100,~~ Sibiu/553=300+253, Bucharest/450=450+0, Sibiu/591=338+253, ~~Bucharest/418=418+0,~~ Craiova/615=455+160, RimnicuVilcea/607=414+193

Note that the short expensive path stays on the queue. The long cheap path is found and returned.

A* tree search example: Simulated queue. City/f=g+h



A^* tree search example: Simulated queue. City/f=g+h



Properties of A*

- Complete? Yes

  (unless there are infinitely many nodes with $f \leq f(G)$;

  can’t happen if step-cost ≥ ε > 0)
- Time/Space? Exponential $O(b^d)$

  except if: $| h(n) - h^*(n) | \leq O(\log h^*(n))$
- Optimal?

  (with: Tree-Search, admissible heuristic;
  Graph-Search, consistent heuristic)
- *Optimally Efficient?*

  (no optimal algorithm with same heuristic is guaranteed to expand fewer nodes)

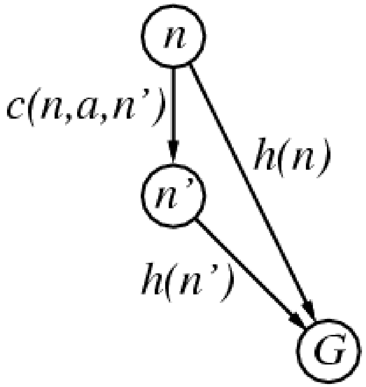
Admissible heuristics

- A heuristic $h(n)$ is admissible if for every node n, $h(n) \leq h^{*}(n)$, where $h^{*}(n)$ is the true cost to reach the goal state from n.
- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic
- Example: $h_{SLD}(n)$ (never overestimates the actual road distance)
- Theorem: If $h(n)$ is admissible, A^{*} using TREE-SEARCH is optimal

Consistent heuristics (consistent => admissible)

- A heuristic is consistent if for every node *n*, every successor *n'* of *n* generated by any action *a*,

$$\boxed{h(n) \le c(n,a,n') + h(n')}$$



- If *h* is consistent, we have

$f(n') = g(n') + h(n')$ (by def.)
$= g(n) + c(n,a,n') + h(n')$ $(g(n')=g(n)+c(n.a.n'))$
$\ge g(n) + h(n) = f(n)$ (consistency)
$f(n') \ge f(n)$

- i.e., *f(n)* is non-decreasing along any path.

It's the triangle inequality !

- Theorem:

  If *h(n)* is consistent, A* using GRAPH-SEARCH is optimal

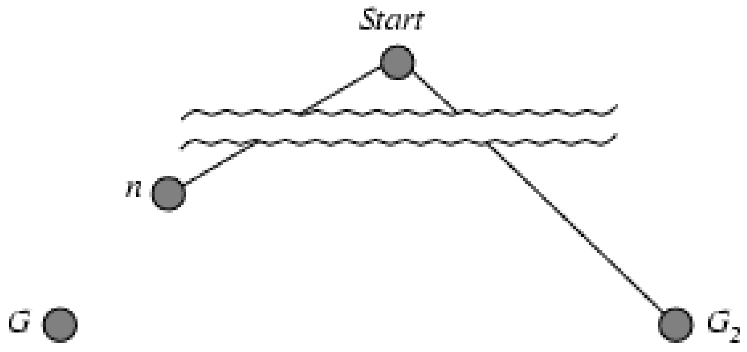
keeps all checked nodes in memory to avoid repeated states

Optimality of A* (proof)

Tree Search, where *h(n)* is admissible

- Suppose some suboptimal goal G_2 has been generated and is in the frontier. Let *n* be an unexpanded node in the frontier such that *n* is on a shortest path to an optimal goal *G*.

We want to prove:
f(n) < f(G2)
(then A* will expand n before G2)



- $f(G_2) = g(G_2)$ since $h(G_2) = 0$
- $f(G) = g(G)$ since $h(G) = 0$
- $g(G_2) > g(G)$ since G_2 is suboptimal
- $f(G_2) > f(G)$ from above, with h=0
- $h(n) \leq h^*(n)$ since h is admissible (*under*-estimate)
- $g(n) + h(n) \leq g(n) + h^*(n)$ from above
- $f(n) \leq f(G)$ since $g(n)+h(n)=f(n)$ & $g(n)+h^*(n)=f(G)$
- $f(n) < f(G2)$ from above

R&N pp. 95-98 proves the optimality of A* graph search with a consistent heuristic

Dominance

- IF $h_2(n) \geq h_1(n)$ for all *n*
  THEN h_2 dominates h_1
  - h_2 is almost always better for search than h_1
  - h_2 guarantees to expand no more nodes than does h_1
  - h_2 almost always expands fewer nodes than does h_1
  - Not useful unless both h_1 & h_2 are admissible/consistent

- Typical 8-puzzle search costs
  (average number of nodes expanded):
  - *d=12* IDS = 3,644,035 nodes
    $A^*(h_1)$ = 227 nodes
    $A^*(h_2)$ = 73 nodes
  - *d=24* IDS = too many nodes
    $A^*(h_1)$ = 39,135 nodes
    $A^*(h_2)$ = 1,641 nodes

Review Local Search
Chapter 4.1-4.2, 4.6; Optional 4.3-4.5

- Problem Formulation (4.1)
- Hill-climbing Search (4.1.1)
- Simulated annealing search (4.1.2)
- Local beam search (4.1.3)
- Genetic algorithms (4.1.4)

Local search algorithms

- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution
  - Local search: widely used for *very big* problems
  - Returns good but *not optimal* solutions
  - *Usually very slow,* but can yield good solutions if you wait

- State space = set of "complete" configurations
- Find a complete configuration satisfying constraints
  - Examples: n-Queens, VLSI layout, airline flight schedules

- Local search algorithms
  - Keep a single "current" state, or small set of states
  - Iteratively try to improve it / them
  - Very memory efficient
    - keeps only one or a few states
    - You control how much memory you use

Random restart wrapper

- We’ll use stochastic local search methods
  - Return different solution for each trial & initial state
- Almost every trial hits difficulties (see sequel)
  - Most trials will not yield a good result (sad!)
- Using many random restarts improves your chances
  - Many “shots at goal” may finally get a good one
- Restart a random initial state, *many times*
  - Report the best result found across *many* trials

Random restart wrapper

```
best_found ← RandomState()   // initialize to something

// now do repeated local search
loop do
   if (tired of doing it)
      then return best_found
   else
      result ← LocalSearch( RandomState() )
      if ( Cost(result) < Cost(best_found) )
        // keep best result found so far

         then best_found ← result
```

You, as algorithm designer, write the functions named in red.

Typically, **“tired of doing it”** means that some resource limit has been exceeded, e.g., number of iterations, wall clock time, CPU time, etc. It may also mean that result improvements are small and infrequent, e.g., less than 0.1% result improvement in the last week of run time.

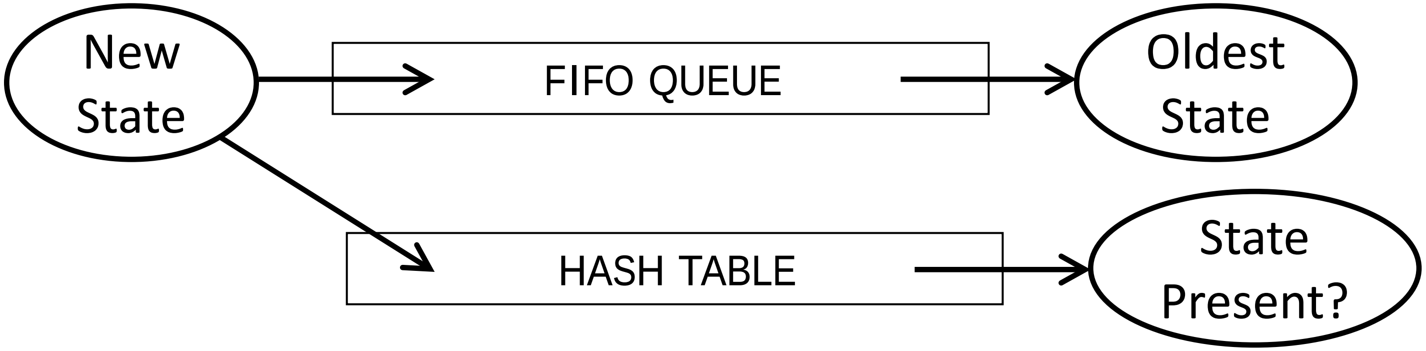
Tabu search wrapper

- Add recently visited states to a tabu-list
  - Temporarily excluded from being visited again
  - Forces solver away from explored regions
  - Less likely to get stuck in local minima (hope, in principle)

- Implemented as a hash table + FIFO queue
  - Unit time cost per step; constant memory cost
  - You control how much memory is used

- RandomRestart(TabuSearch (LocalSearch()))

Tabu search wrapper **(<u>inside</u> random restart!)**



```
best_found ← current_state ← RandomState()   // initialize
loop do         // now do local search
    if (tired of doing it) then return best_found else
        neighbor ← MakeNeighbor( current_state )
        if ( neighbor is in hash_table ) then discard neighbor
        else push neighbor onto fifo, pop oldest_state
                remove oldest_state from hash_table, insert neighbor
                    current_state ← neighbor;
                if ( Cost(current_state ) < Cost(best_found) )
                    then best_found ← current_state
```

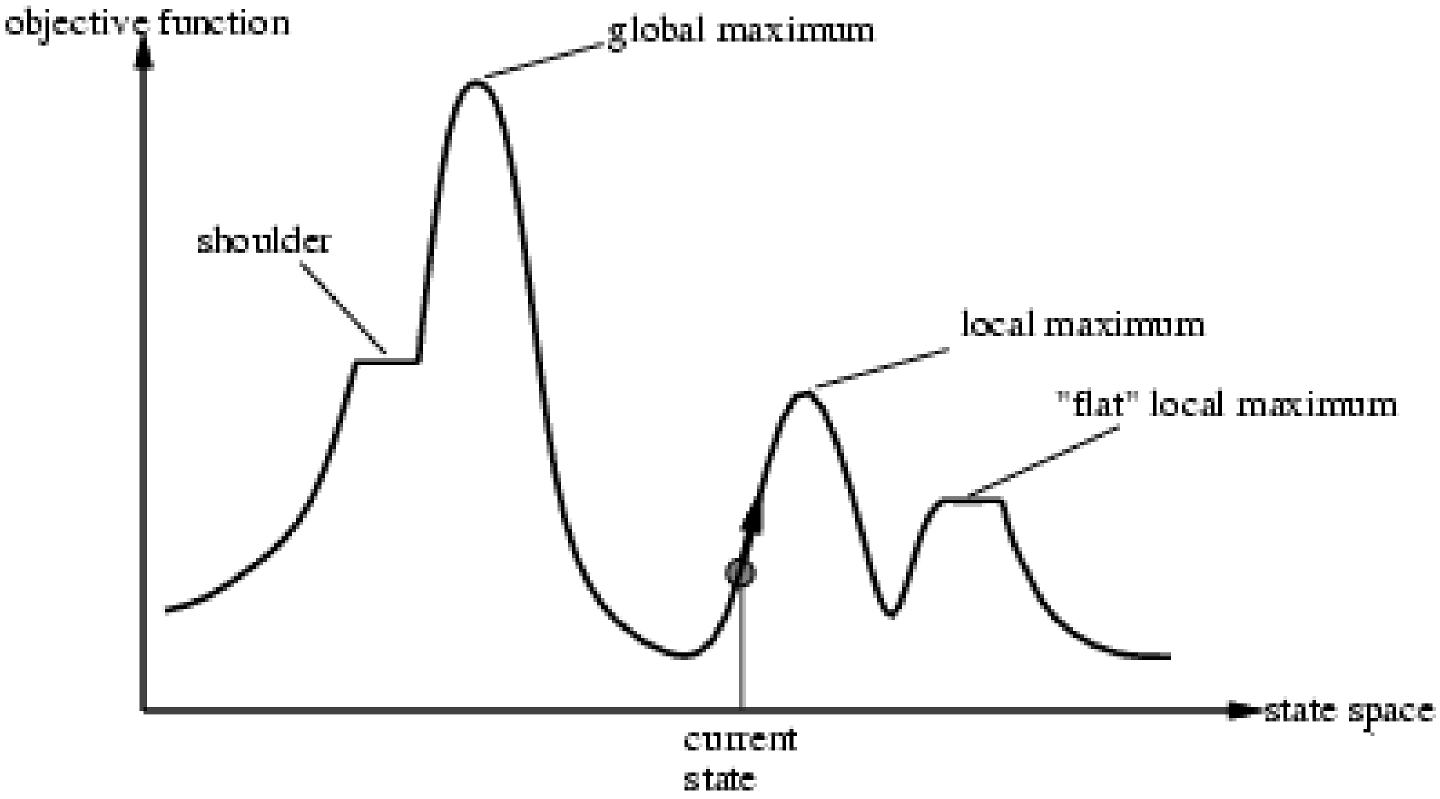
Local search algorithms

- Hill-climbing search
    - Gradient descent in continuous state spaces
    - Can use, e.g., Newton’s method to find roots
- Simulated annealing search
- Local beam search
- Genetic algorithms
- Linear Programming (for specialized problems)

Local Search Difficulties

These difficulties apply to ALL local search algorithms, and become MUCH more difficult as the search space increases to high dimensionality.

- Problems: depending on state, can get stuck in local maxima
  - Many other problems also endanger your success!!



Local Search Difficulties

These difficulties apply to ALL local search algorithms, and become MUCH more difficult as the search space increases to high dimensionality.

- Ridge problem: Every neighbor appears to be downhill
  - But the search space has an uphill!! (worse in high dimensions)

Ridge:
Fold a piece of paper and hold it tilted up at an unfavorable angle to every possible search space step. Every step leads downhill; but the ridge leads uphill.

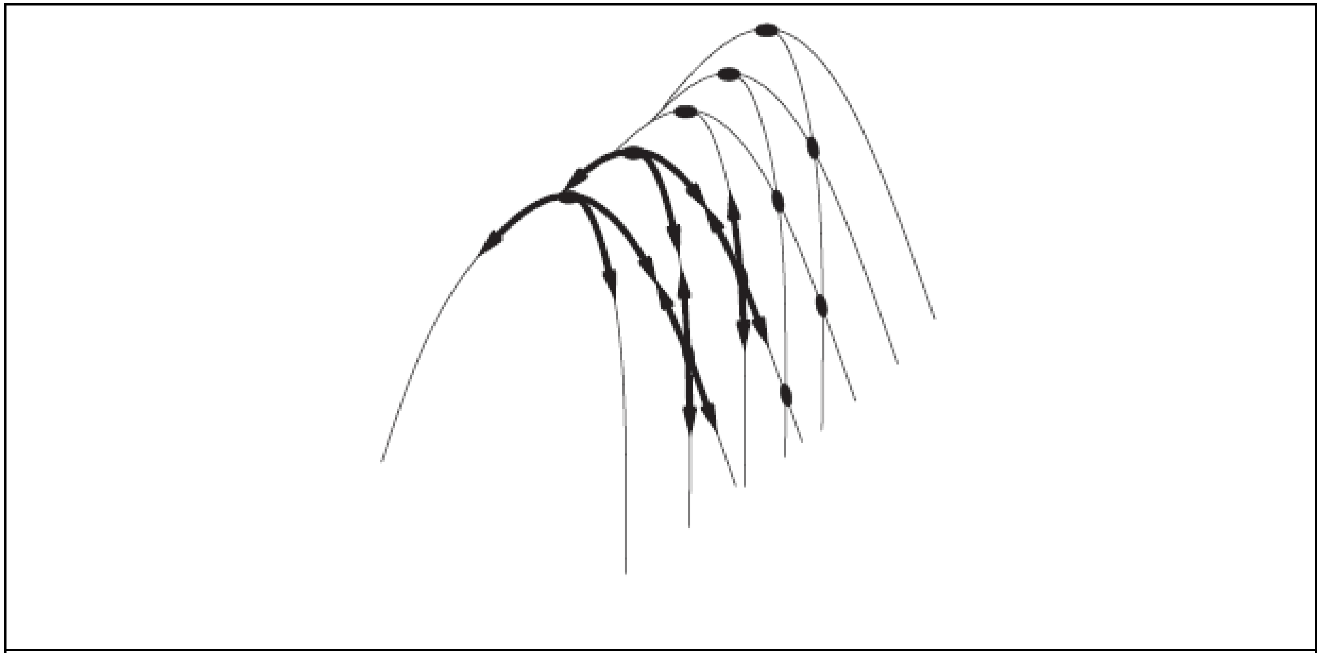
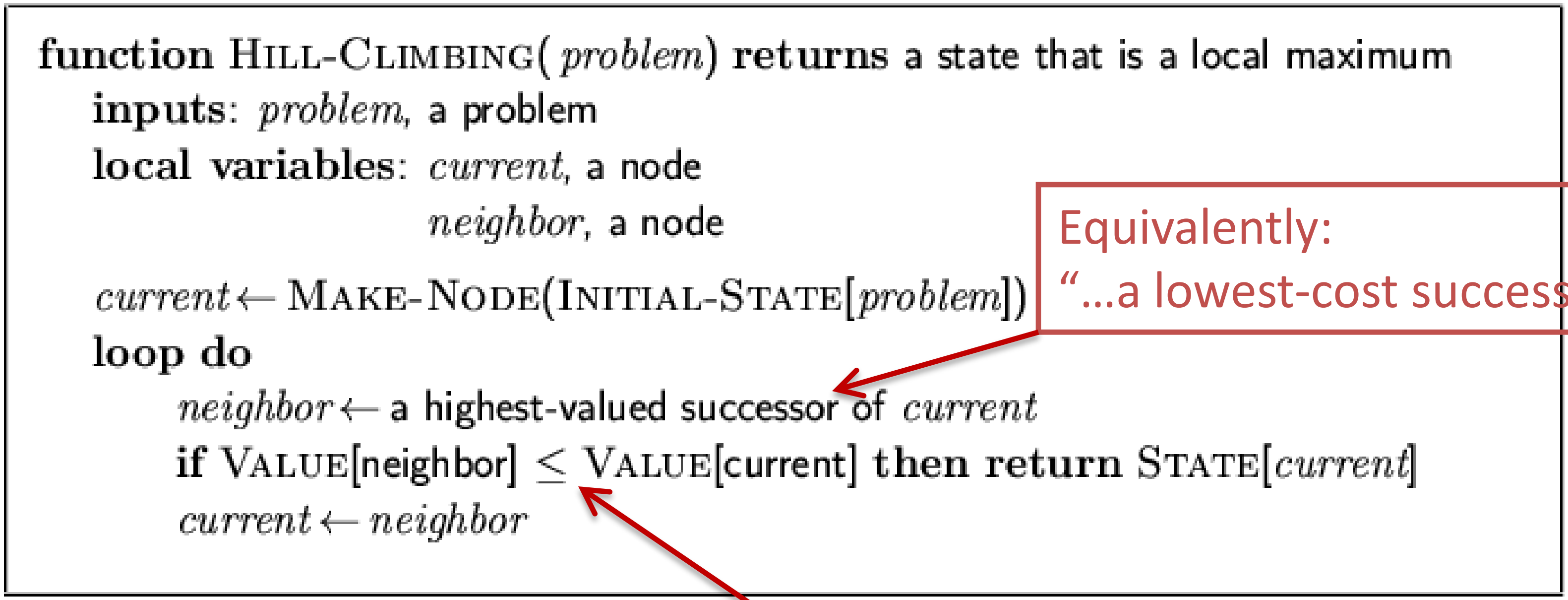


Figure 4.4 **FILES: figures/ridge.eps (Tue Nov 3 16:23:29 2009).** Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill.

Hill-climbing search

You must shift effortlessly between maximizing value and minimizing cost

"…like trying to find the top of Mount Everest in a thick fog while suffering from amnesia"



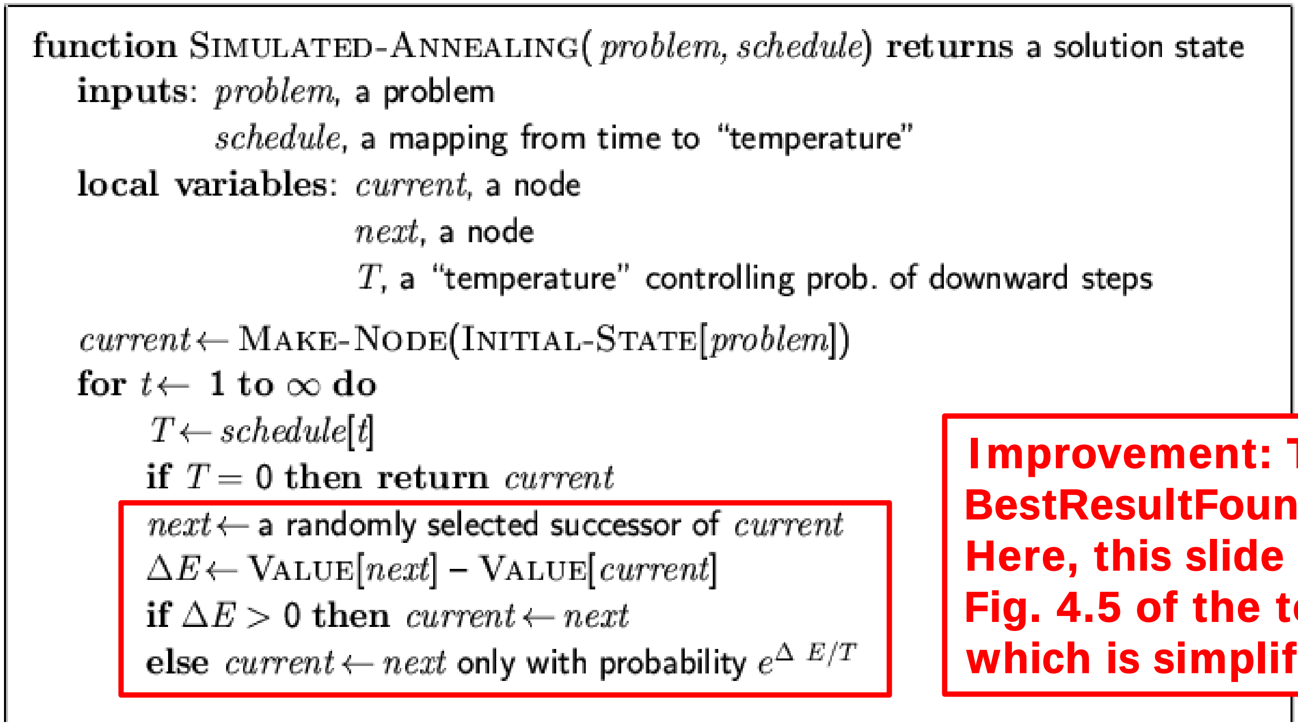
```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest-valued successor of current
        if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
        current ← neighbor
```

Equivalently:
"…a lowest-cost successor…"

Equivalently: "if **COST**[neighbor] ≥ **COST**[current] then …"

Simulated annealing (Physics!)

- Idea: escape local maxima by allowing some "bad" moves but gradually decrease their frequency



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     $T$, a "temperature" controlling prob. of downward steps

    current ← MAKE-NODE(INITIAL-STATE[problem])
    for $t$ ← 1 to $\infty$ do
        $T$ ← schedule[$t$]
        if $T = 0$ then return current
        next ← a randomly selected successor of current
        $\Delta E$ ← VALUE[next] – VALUE[current]
        if $\Delta E > 0$ then current ← next
        else current ← next only with probability $e^{\Delta E/T}$
```

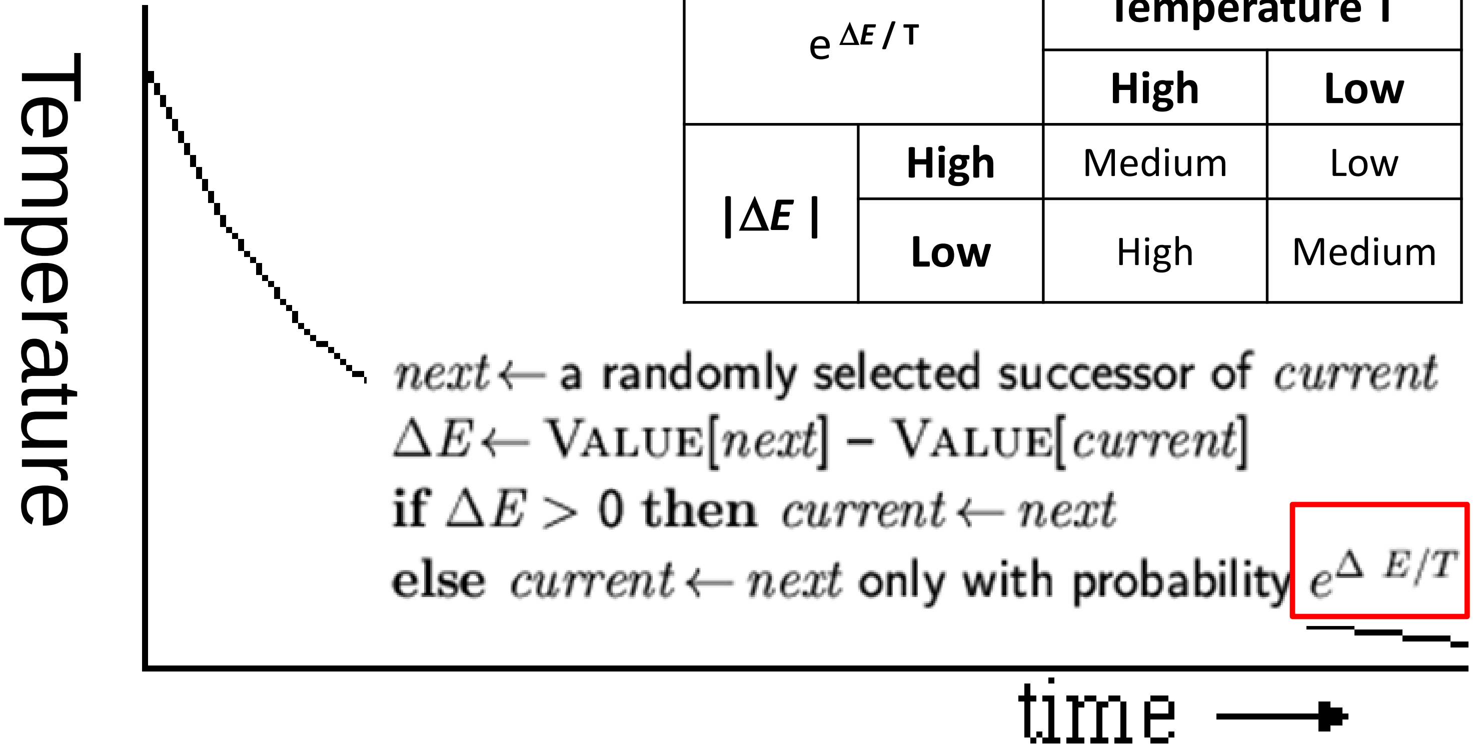
Improvement: Track the BestResultFoundSoFar. Here, this slide follows Fig. 4.5 of the textbook, which is simplified.

Probability(accept worse successor)

- Decreases as temperature T decreases
- Increases as |Δ E| decreases
- Sometimes, step size also decreases with T

(accept very bad moves early on; later, mainly accept “not very much worse”)

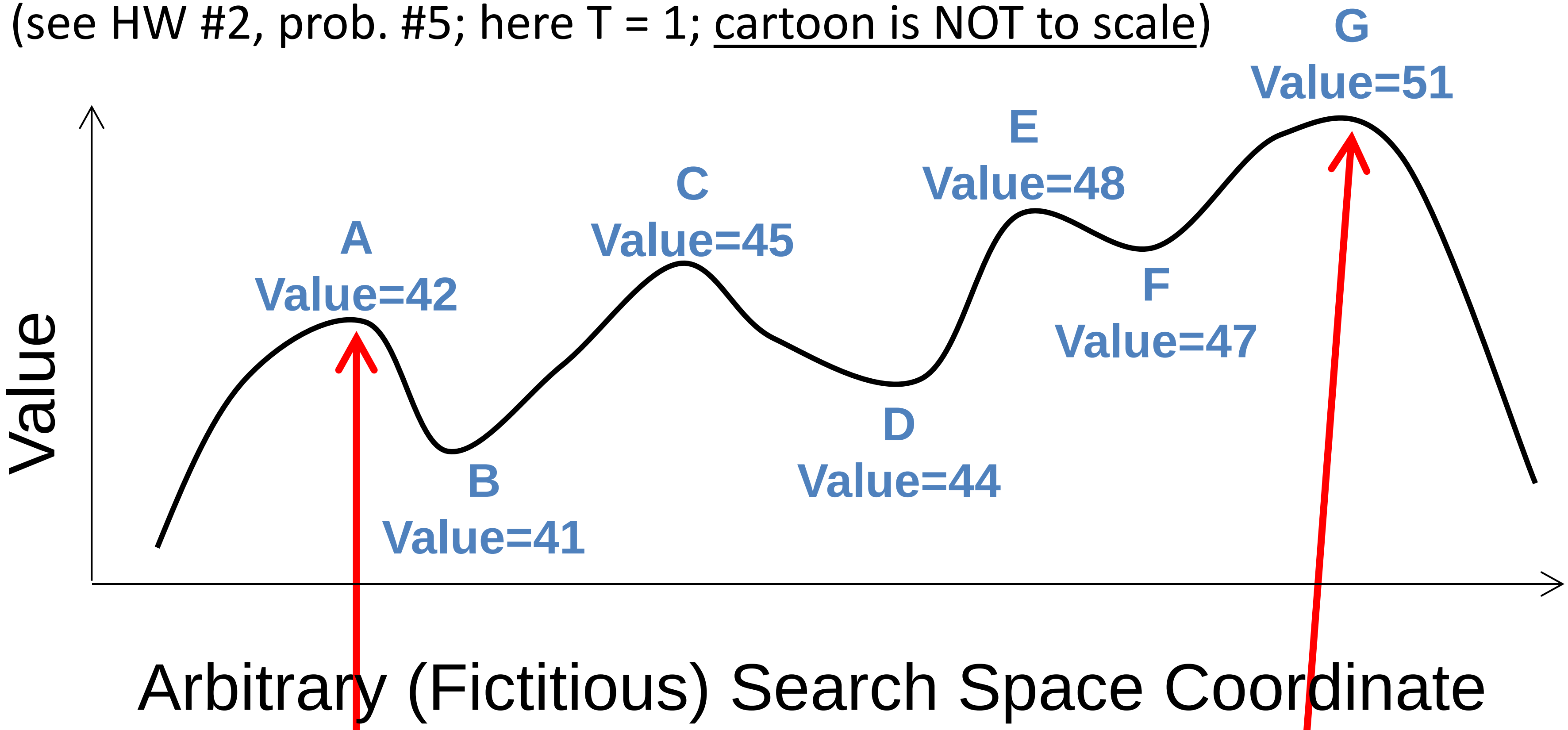
$e^{\Delta E / T}$		**Temperature T**	
		High	**Low**
$\lvert \Delta E \rvert$	**High**	Medium	Low
	Low	High	Medium



```
$next \leftarrow$ a randomly selected successor of $current$
$\Delta E \leftarrow \text{VALUE}[next] - \text{VALUE}[current]$
if $\Delta E > 0$ then $current \leftarrow next$
else $current \leftarrow next$ only with probability $\boxed{e^{\Delta\ E/T}}$
```

Goal: “ratchet up” a bumpy slope

(see HW #2, prob. #5; here T = 1; cartoon is NOT to scale)

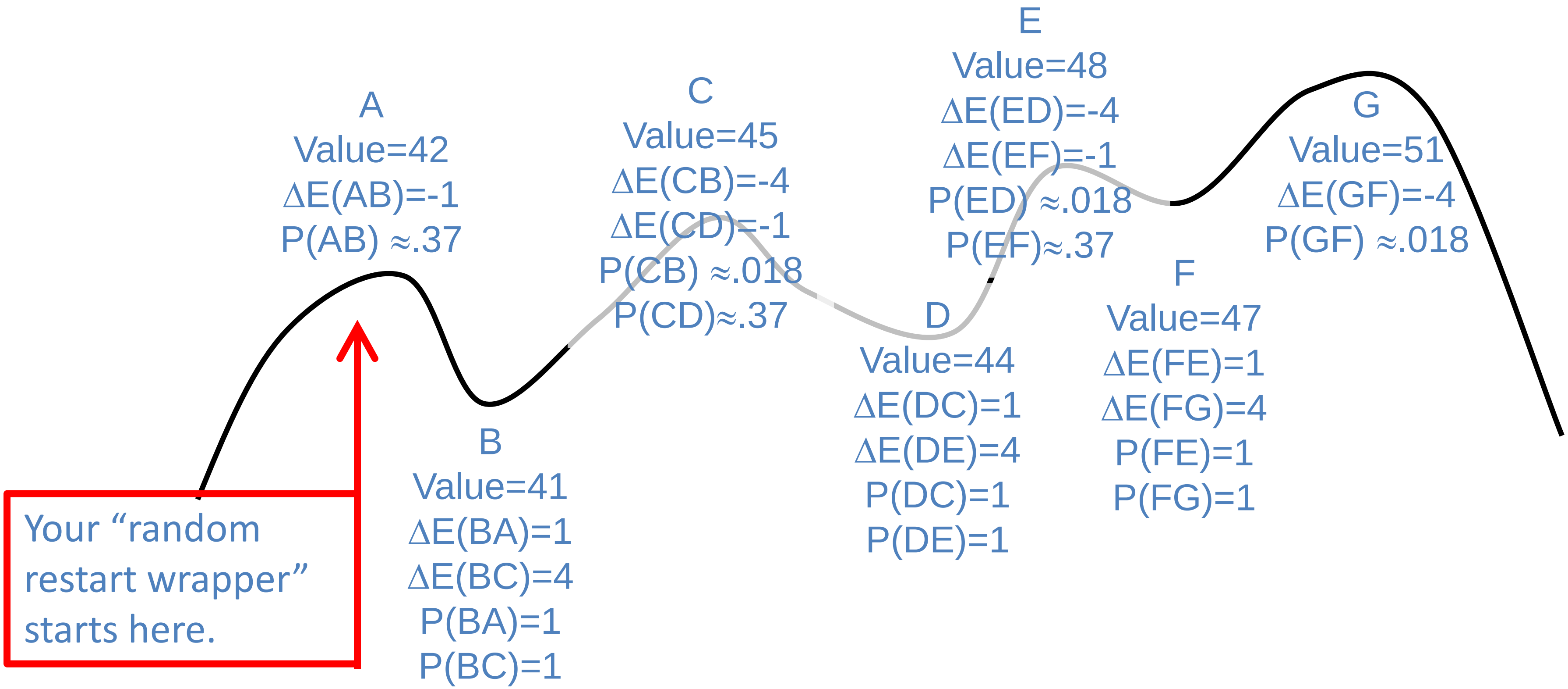


Your “random restart wrapper” starts here.

You want to get here. HOW??

This is an illustrative *cartoon*…

Goal: “ratchet up” a jagged slope



A
Value=42
ΔE(AB)=-1
P(AB) ≈.37

B
Value=41
ΔE(BA)=1
ΔE(BC)=4
P(BA)=1
P(BC)=1

C
Value=45
ΔE(CB)=-4
ΔE(CD)=-1
P(CB) ≈.018
P(CD)≈.37

D
Value=44
ΔE(DC)=1
ΔE(DE)=4
P(DC)=1
P(DE)=1

E
Value=48
ΔE(ED)=-4
ΔE(EF)=-1
P(ED) ≈.018
P(EF)≈.37

F
Value=47
ΔE(FE)=1
ΔE(FG)=4
P(FE)=1
P(FG)=1

G
Value=51
ΔE(GF)=-4
P(GF) ≈.018

Your “random restart wrapper” starts here.

x	-1	-4
e^x	≈.37	≈.018

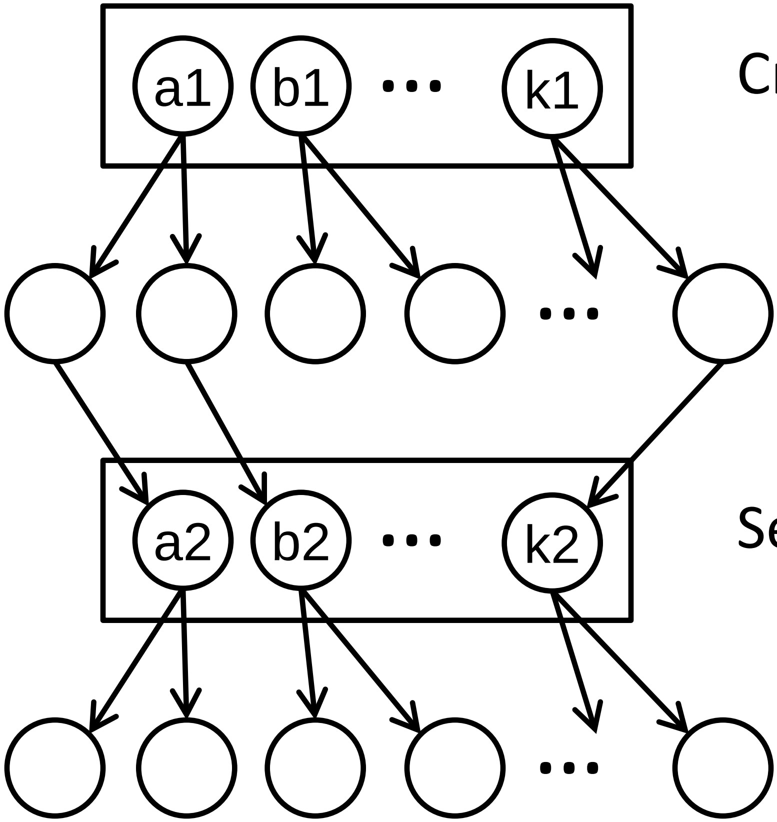
From A you will accept a move to B with P(AB) ≈.37.
From B you are equally likely to go to A or to C.
From C you are ≈20X more likely to go to D than to B.
From D you are equally likely to go to C or to E.
From E you are ≈20X more likely to go to F than to D.
From F you are equally likely to go to E or to G.
Remember best point you ever found (G or neighbor?).

This is an illustrative *cartoon*…

Local beam search

- Keep track of *k* states rather than just one
- Start with *k* randomly generated states
- At each iteration, all the successors of all *k* states are generated
- If any one is a goal state, stop; else select the *k* best successors from the complete list and repeat.
- Concentrates search effort in areas believed to be fruitful
  - May lose diversity as search progresses, resulting in wasted effort

Local beam search



Create *k* random initial states

Generate their children

Select the *k* best children

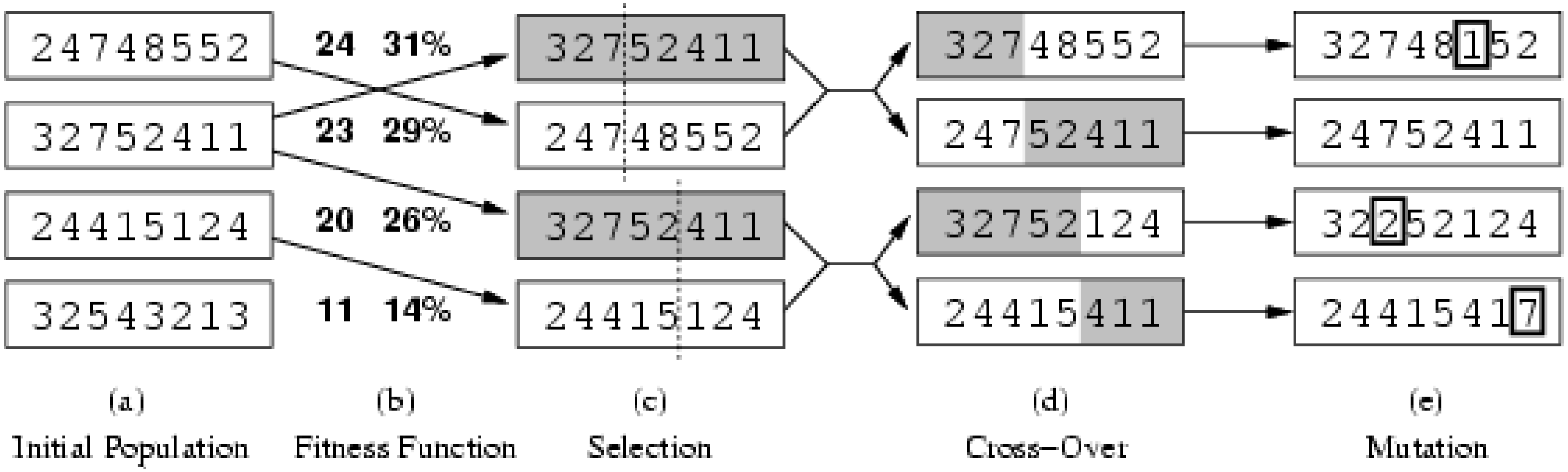
Repeat indefinitely…

Is it better than simply running *k* searches?
Maybe…??

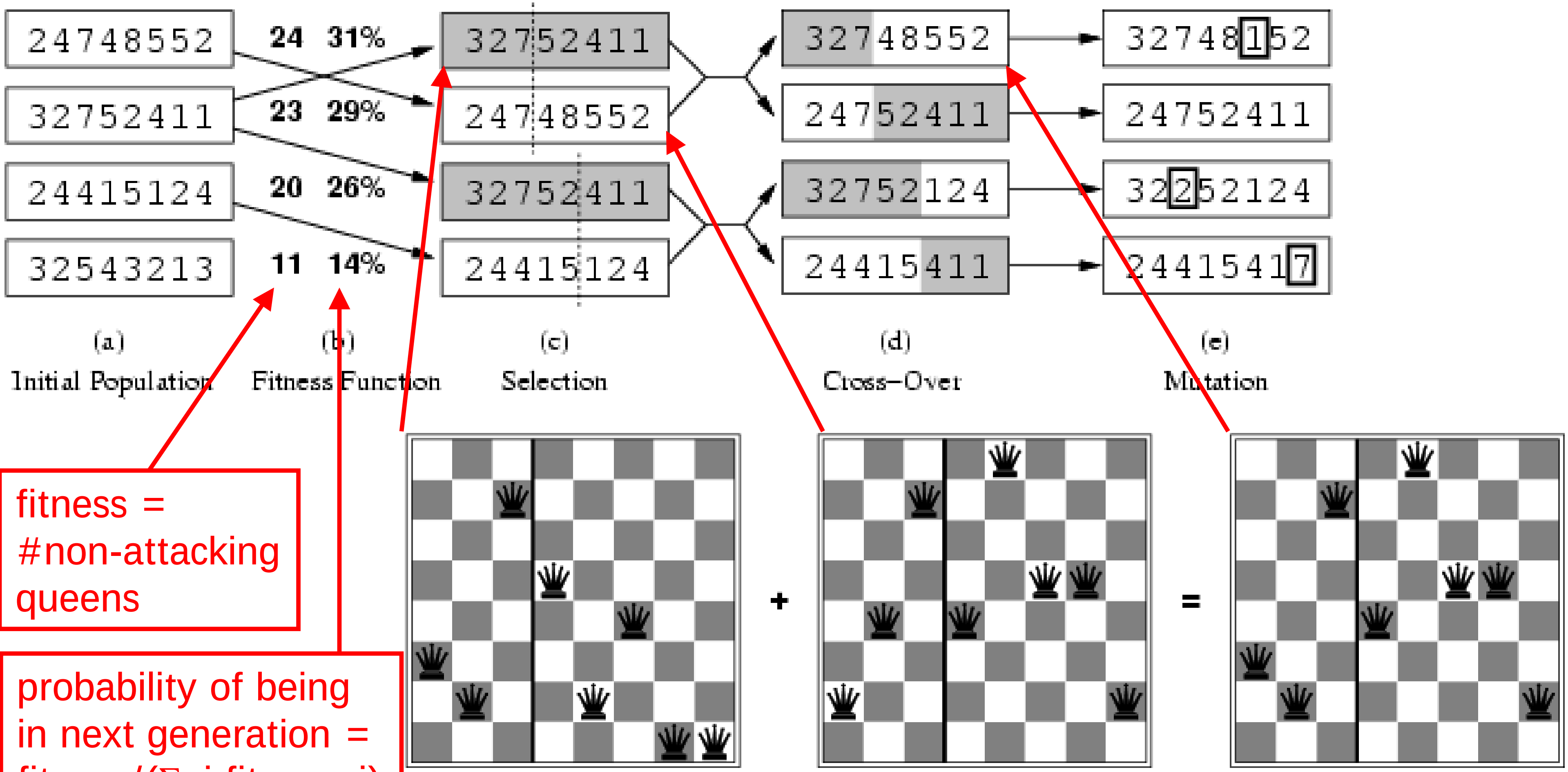
Genetic algorithms (**Darwin!!**)

- A state = a string over a finite alphabet (an **individual**)
  - A successor state is generated by combining two parent states
- Start with *k* randomly generated states (a **population**)
- **Fitness** function (= our heuristic objective function).
  - Higher fitness values for better states.
- **Select** individuals for next generation based on fitness
  - P(individual in next gen.) = individual fitness/total population fitness
- **Crossover** fit parents to yield next generation (**offspring**)
- **Mutate** the offspring randomly with some low probability

Genetic algorithms



- Fitness function (value): number of non-attacking pairs of queens (min = 0, max = 8 × 7/2 = 28)
- 24/(24+23+20+11) = 31%
- 23/(24+23+20+11) = 29%; etc.



fitness = #non-attacking queens

probability of being in next generation = fitness/(Σ_i fitness_i)

How to convert a fitness value into a probability of being in the next generation.

- Fitness function: #non-attacking queen pairs
  - min = 0, max = 8 × 7/2 = 28
- Σ_i fitness_i = 24+23+20+11 = 78
- P(child_1 in next gen.) = fitness_1/(Σ_i fitness_i) = 24/78 = 31%
- P(child_2 in next gen.) = fitness_2/(Σ_i fitness_i) = 23/78 = 29%; etc

Review Propositional Logic
Chapter 7.1-7.5; Optional 7.6-7.8

- Definitions:
  - Syntax, Semantics, Sentences, Propositions, Entails, Follows, Derives, Inference, Sound, Complete, Model, Satisfiable, Valid (or Tautology)
- Syntactic & Semantic Transformations:
  - E.g., $(A \Rightarrow B) \Leftrightarrow (\neg A \vee B)$
  - E.g., $(KB \models \alpha) \equiv (\models (KB \Rightarrow \alpha)$
- Truth Tables:
  - Negation, Conjunction, Disjunction, Implication, Equivalence (Biconditional)
- Inference:
  - By Resolution (CNF)
  - By Backward & Forward Chaining (Horn Clauses)
  - By Model Enumeration (Truth Tables)

Recap propositional logic: Syntax

- Propositional logic is the simplest logic – illustrates basic ideas

- The proposition symbols P_1, P_2 etc are sentences

  - If S is a sentence, $\neg S$ is a sentence (negation)
  - If S_1 and S_2 are sentences, $S_1 \wedge S_2$ is a sentence (conjunction)
  - If S_1 and S_2 are sentences, $S_1 \vee S_2$ is a sentence (disjunction)
  - If S_1 and S_2 are sentences, $S_1 \Rightarrow S_2$ is a sentence (implication)
  - If S_1 and S_2 are sentences, $S_1 \Leftrightarrow S_2$ is a sentence (biconditional)

Recap propositional logic: Semantics

Each model/world specifies true or false for each proposition symbol

E.g.,	$P_{1,2}$	$P_{2,2}$	$P_{3,1}$
	false	true	false

With these symbols, 8 possible models can be enumerated automatically.

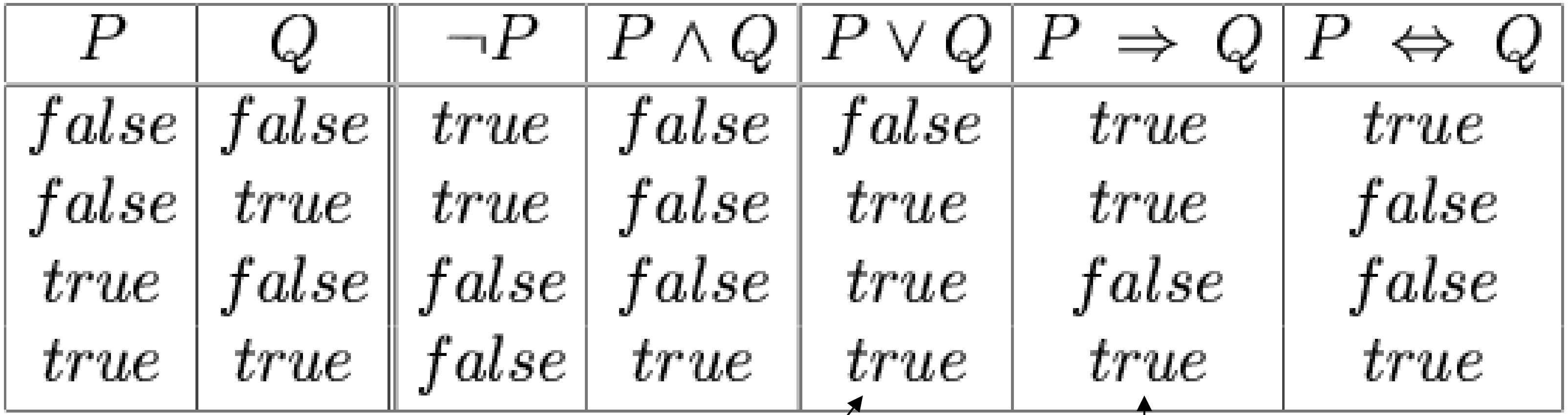
Rules for evaluating truth with respect to a model *m*:

$\neg S$	is true iff	S is false	
$S_1 \wedge S_2$	is true iff	S_1 is true and	S_2 is true
$S_1 \vee S_2$	is true iff	$S_1$is true or	S_2 is true
$S_1 \Rightarrow S_2$	is true iff	S_1 is false or	S_2 is true
(i.e.,	is false iff	S_1 is true and	S_2 is false)
$S_1 \Leftrightarrow S_2$	is true iff	$S_1 \Rightarrow S_2$ is true and $S_2 \Rightarrow S_1$ is true	

Simple recursive process evaluates an arbitrary sentence, e.g.,

$$\neg P_{1,2} \wedge (P_{2,2} \vee P_{3,1}) = \mathit{true} \wedge (\mathit{true} \vee \mathit{false}) = \mathit{true} \wedge \mathit{true} = \mathit{true}$$

Recap propositional logic:
Truth tables for connectives



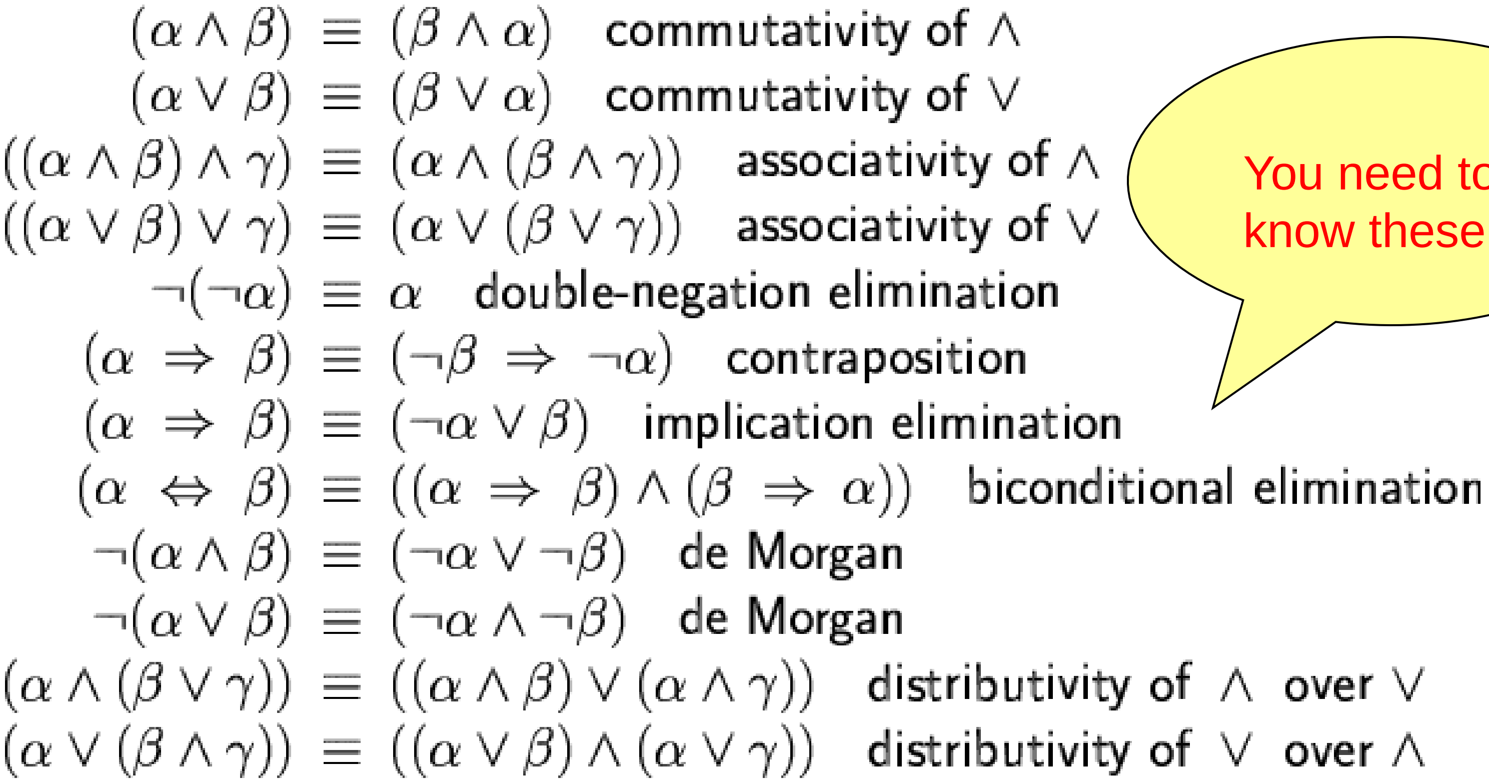
P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	*false*	*true*	*false*	*false*	*true*	*true*
false	*true*	*true*	*false*	*true*	*true*	*false*
true	*false*	*false*	*false*	*true*	*false*	*false*
true	*true*	*false*	*true*	*true*	*true*	*true*

OR: P or Q is true or both are true.
XOR: P or Q is true but not both.

Implication is always true
when the premises are False!

Recap propositional logic:
Logical equivalence and rewrite rules

- To manipulate logical sentences we need some rewrite rules.
- Two sentences are logically equivalent iff they are true in same models: $\alpha \equiv$ ß iff $\alpha \models \beta$ and $\beta \models \alpha$



$(\alpha \land \beta)$	$\equiv$	$(\beta \land \alpha)$	commutativity of $\land$
$(\alpha \lor \beta)$	$\equiv$	$(\beta \lor \alpha)$	commutativity of $\lor$
$((\alpha \land \beta) \land \gamma)$	$\equiv$	$(\alpha \land (\beta \land \gamma))$	associativity of $\land$
$((\alpha \lor \beta) \lor \gamma)$	$\equiv$	$(\alpha \lor (\beta \lor \gamma))$	associativity of $\lor$
$\neg(\neg\alpha)$	$\equiv$	α	double-negation elimination
$(\alpha \Rightarrow \beta)$	$\equiv$	$(\neg\beta \Rightarrow \neg\alpha)$	contraposition
$(\alpha \Rightarrow \beta)$	$\equiv$	$(\neg\alpha \lor \beta)$	implication elimination
$(\alpha \Leftrightarrow \beta)$	$\equiv$	$((\alpha \Rightarrow \beta) \land (\beta \Rightarrow \alpha))$	biconditional elimination
$\neg(\alpha \land \beta)$	$\equiv$	$(\neg\alpha \lor \neg\beta)$	de Morgan
$\neg(\alpha \lor \beta)$	$\equiv$	$(\neg\alpha \land \neg\beta)$	de Morgan
$(\alpha \land (\beta \lor \gamma))$	$\equiv$	$((\alpha \land \beta) \lor (\alpha \land \gamma))$	distributivity of $\land$ over $\lor$
$(\alpha \lor (\beta \land \gamma))$	$\equiv$	$((\alpha \lor \beta) \land (\alpha \lor \gamma))$	distributivity of $\lor$ over $\land$

You need to know these !

Recap propositional logic: Entailment

- Entailment means that one thing follows from another:

$$KB \models \alpha$$

- Knowledge base *KB* entails sentence α if and only if α is true in all worlds where *KB* is true
  - E.g., the KB containing “the Giants won and the Reds won” entails “The Giants won”.
  - E.g., x+y = 4 entails 4 = x+y
  - E.g., “Mary is Sue’s sister and Amy is Sue’s daughter” entails “Mary is Amy’s aunt.”

Review: Models (and in FOL, Interpretations)

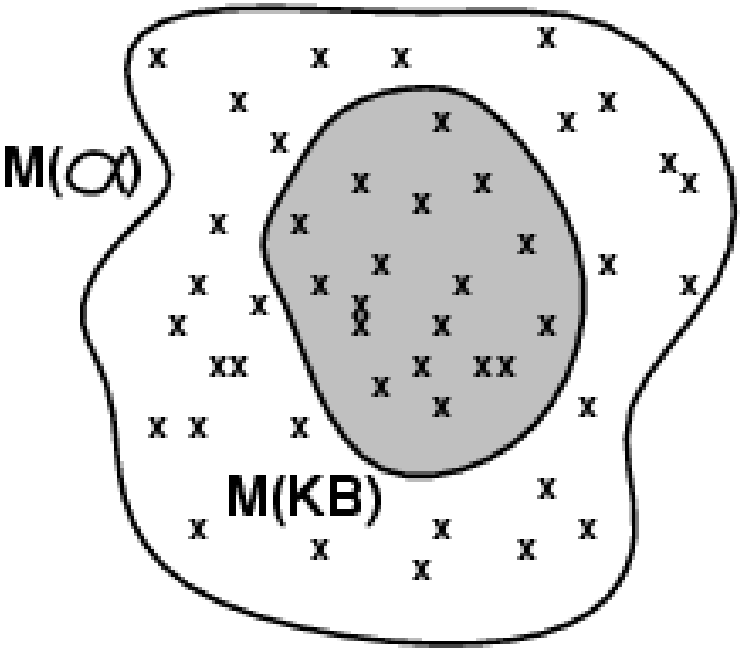
- Models are formal worlds in which truth can be evaluated

- We say *m* is a model of a sentence α if α is true in *m*

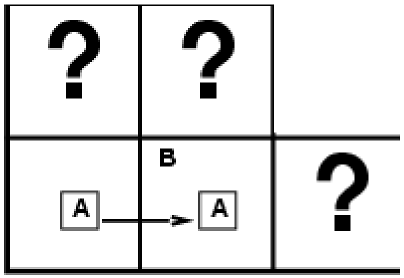
- *M(α)* is the set of all models of α

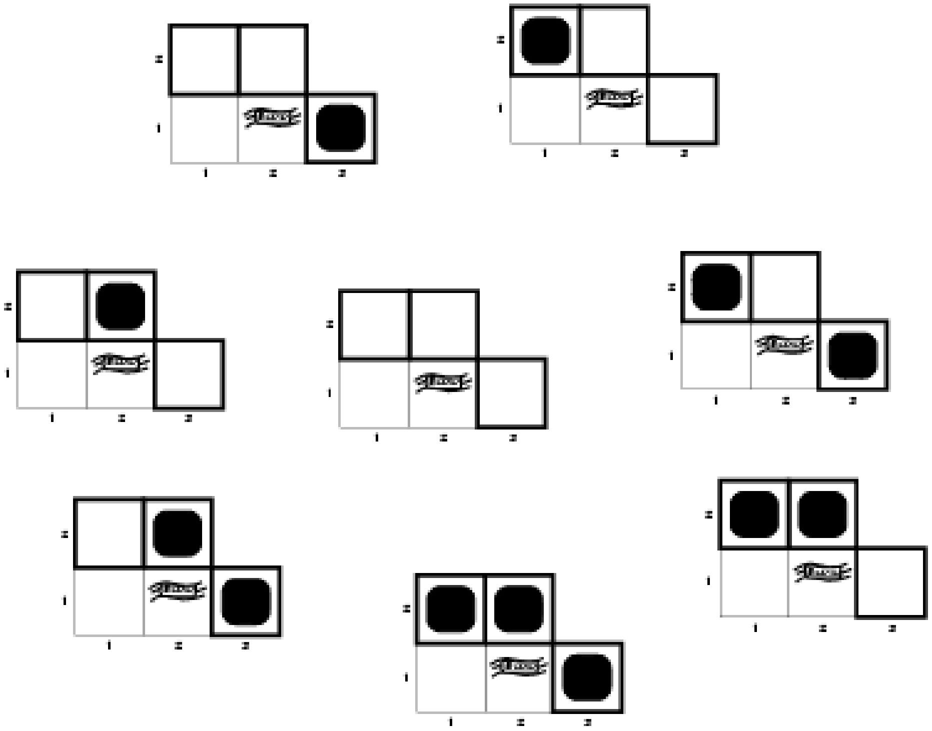
- Then KB ╞ α iff *M(KB)* ⊆ *M*(α)
  - E.g. *KB,* = "Mary is Sue's sister and Amy is Sue's daughter."
  - α = "Mary is Amy's aunt."

- Think of KB and α as constraints, and of models m as possible states.
- M(KB) are the solutions to KB and M(α) the solutions to α.
- Then, KB ╞ α, i.e., ╞ (KB ⇒ a) , when all solutions to KB are also solutions to α.



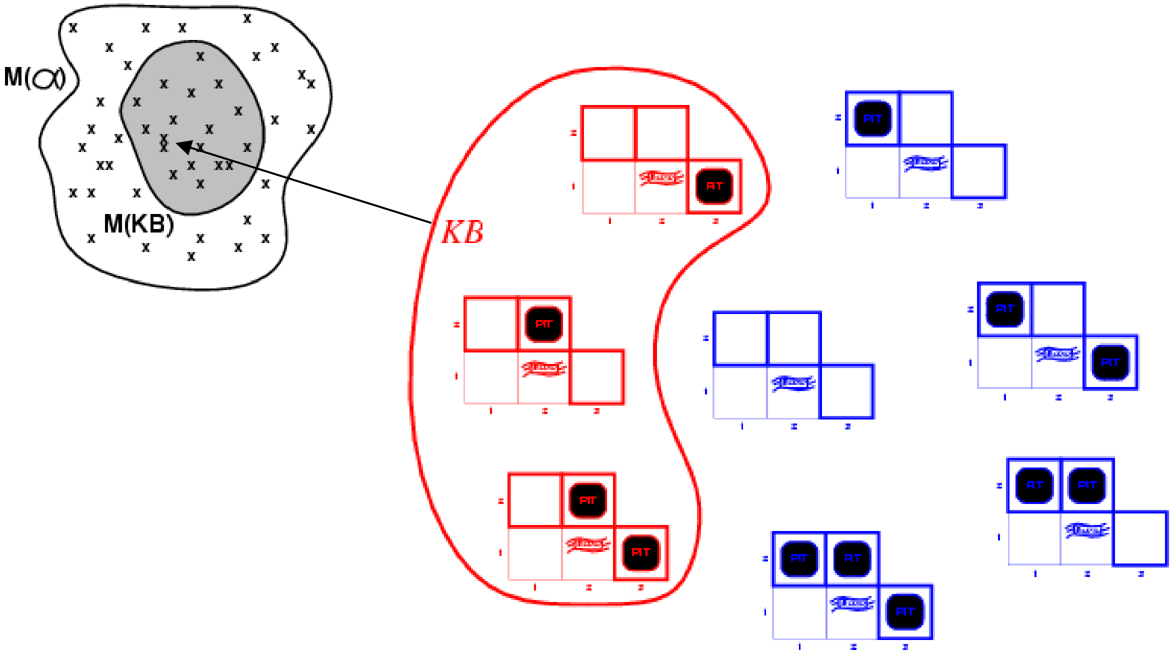
Wumpus models





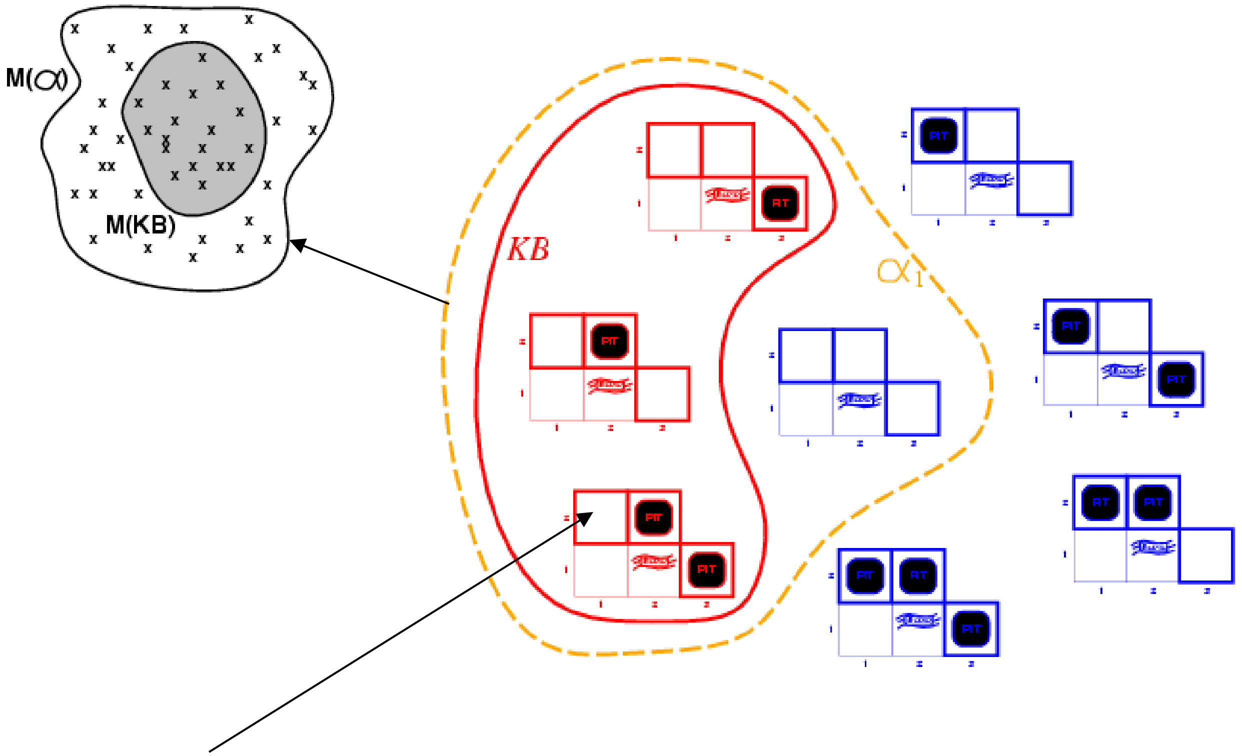
All possible models in this reduced Wumpus world. What can we infer?

Review: Wumpus models



- *KB* = all possible wumpus-worlds consistent with the observations and the "physics" of the Wumpus world.

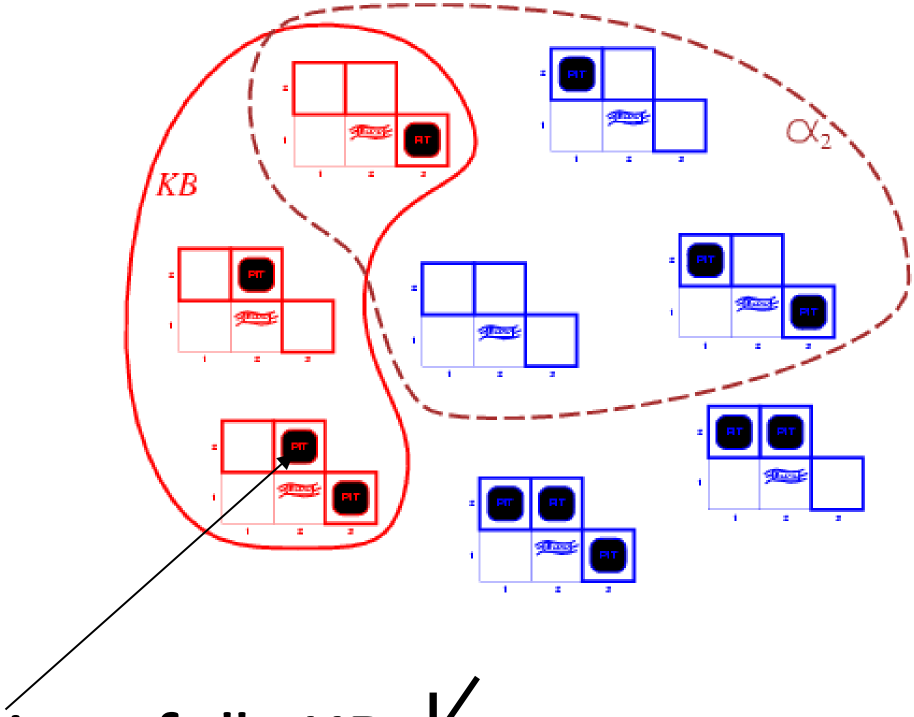
Review: Wumpus models



α_1 = "[1,2] is safe", *KB* $\models \alpha_1$, proved by model checking.

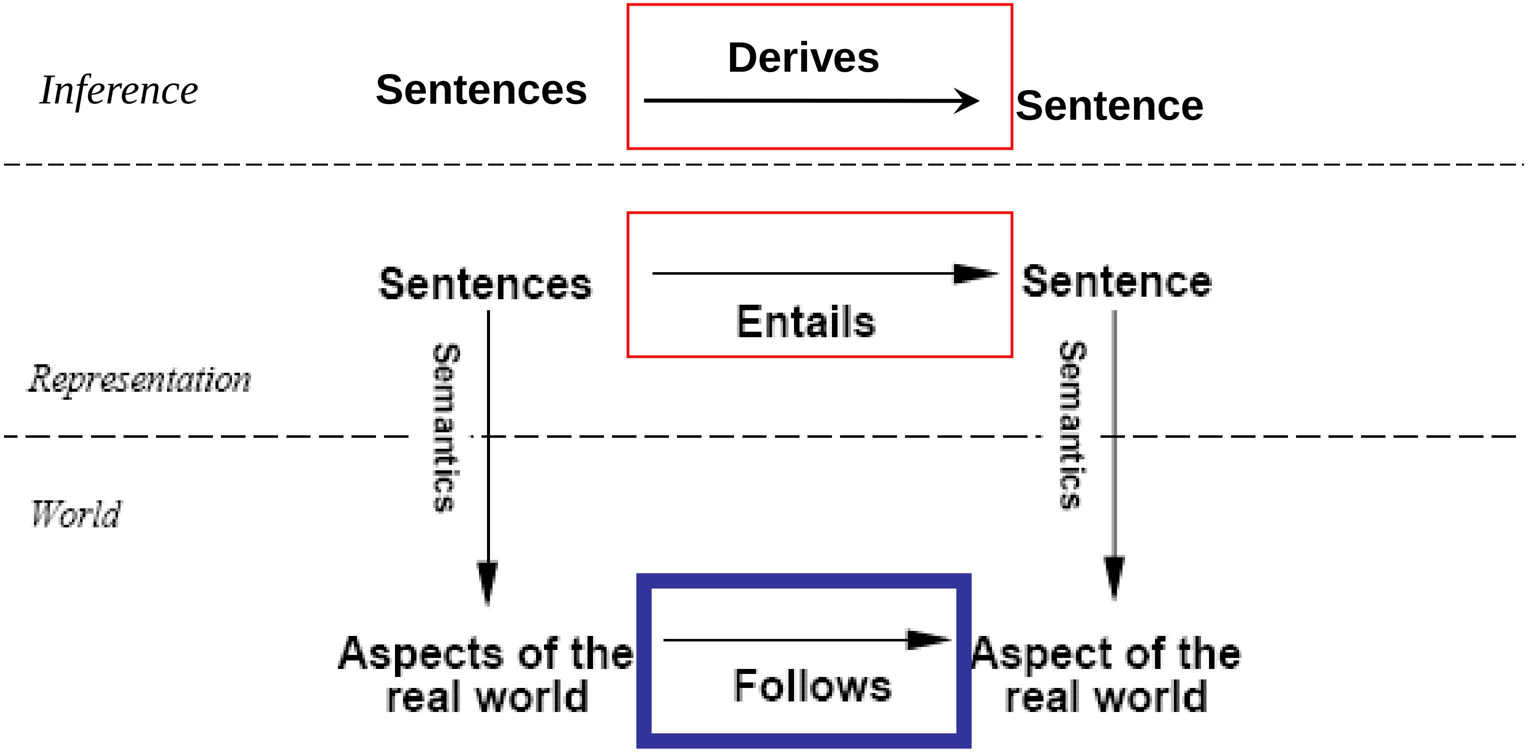
Every model that makes KB true also makes α_1 true.

Wumpus models



α_2 = "[2,2] is safe", $KB \not\models \alpha_2$

Review: Schematic for Follows, Entails, and Derives



If KB is true in the real world,

then any sentence α ***entailed*** *by KB*

and any sentence α ***derived*** *from KB*

by a sound inference procedure

is also true in the real world.

Recap propositional logic: Validity and satisfiability

A sentence is valid if it is true in all models,

e.g., *True*, $A \vee \neg A$, $A \Rightarrow A$, $(A \wedge (A \Rightarrow B)) \Rightarrow B$

Validity is connected to inference via the Deduction Theorem:

$KB \models \alpha$ if and only if $(KB \Rightarrow \alpha)$ is valid

A sentence is satisfiable if it is true in some model

e.g., $A \vee B$, C

A sentence is unsatisfiable if it is false in all models

e.g., $A \wedge \neg A$

Satisfiability is connected to inference via the following:

$KB \models A$ if and only if $(KB \wedge \neg A)$ is unsatisfiable

(there is no model for which KB is true and A is false)

Inference Procedures

- *KB* $\vdash_i$ A means that sentence A can be derived from *KB* by procedure *i*
- Soundness: *i* is sound if whenever *KB* $\vdash_i \alpha$, it is also true that *KB* $\models \alpha$
  - *(no wrong inferences, but maybe not all inferences)*
- Completeness: *i* is complete if whenever *KB* $\models \alpha$, it is also true that *KB* $\vdash_i \alpha$
  - *(all inferences can be made, but maybe some wrong extra ones as well)*
- Entailment can be used for inference (Model checking)
  - enumerate all possible models and check whether α is true.
  - For *n* symbols, time complexity is $O(2^n)$...
- Inference can be done directly on the sentences
  - Forward chaining, backward chaining, resolution (see FOPC, later)

Inference by Resolution

- KB is represented in CNF
  - KB = AND of all the sentences in KB
  - KB sentence = clause = OR of literals
  - Literal = propositional symbol or its negation

- Find two clauses in KB, one of which contains a literal and the other its negation
  - Cancel the literal and its negation
  - Bundle everything else into a new clause
  - Add the new clause to KB
  - Repeat

Example: Conversion to CNF

Example: $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$

1. Eliminate $\Leftrightarrow$ by replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.

   $= (B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$

2. Eliminate $\Rightarrow$ by replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$ and simplify.

   $= (\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1})$

3. Move $\neg$ inwards using de Morgan's rules and simplify.

   $$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta), \neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta)$$

   $= (\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1})$

4. Apply distributive law ($\wedge$ over $\vee$) and simplify.

   $= (\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1})$

Example: Conversion to CNF

Example: $B_{1,1} \Leftrightarrow (P_{1,2} \lor P_{2,1})$

From the previous slide we had:

$= (\neg B_{1,1} \lor P_{1,2} \lor P_{2,1}) \land (\neg P_{1,2} \lor B_{1,1}) \land (\neg P_{2,1} \lor B_{1,1})$

5. KB is the conjunction of all of its sentences (all are true), so write each clause (disjunct) as a sentence in KB:

KB =

…

$(\neg B_{1,1} \lor P_{1,2} \lor P_{2,1}) \longrightarrow (\neg B_{1,1}\quad P_{1,2}\quad P_{2,1})$

$(\neg P_{1,2} \lor B_{1,1}) \longrightarrow (\neg P_{1,2}\quad B_{1,1})$

$(\neg P_{2,1} \lor B_{1,1}) \longrightarrow (\neg P_{2,1}\quad B_{1,1})$

(same)

…

Often, Won’t Write “$\lor$” or “$\land$”

(we know they are there)

Resolution = Efficient Implication

Recall that (A => B) = ((NOT A) OR B)
and so:

(Y OR X) = ((NOT X) => Y)

((NOT Y) OR Z) = (Y => Z)

which yields:

((Y OR X) AND ((NOT Y) OR Z)) = ((NOT X) => Z) = (X OR Z)

```
(OR   A  B  C  D)        ->Same ->      (NOT (OR  B  C  D))  =>  A
(OR  ¬A  E  F  G)        ->Same ->      A  =>  (OR  E  F  G)
-----------------------------           ----------------------------------------------------
(OR  B  C  D  E  F  G)                  (NOT (OR  B  C  D))  => (OR  E  F  G)
                                        ----------------------------------------------------
                                        (OR  B  C  D  E  F  G)
```

Recall: All clauses in KB are conjoined by an implicit AND (= CNF representation).

Resolution Examples

- Resolution: inference rule for CNF: sound and complete! *

$(A \vee B \vee C)$

$(\neg A)$

– – – – – – – – – – – – –

$\therefore (B \vee C)$

“If A or B or C is true, but not A, then B or C must be true.”

$(A \vee B \vee C)$

$(\neg A \vee D \vee E)$

– – – – – – – – – – – –

$\therefore (B \vee C \vee D \vee E)$

“If A is false then B or C must be true, or if A is true then D or E must be true, hence since A is either true or false, B or C or D or E must be true.”

$(A \vee B)$

$(\neg A \vee B)$

– – – – – – – – –

$\therefore (B \vee B) \equiv B$ ←

“If A or B is true, and not A or B is true, then B must be true.”

Simplification
is done always.

* Resolution is “refutation complete” in that it can prove the truth of any entailed sentence by refutation.

More Resolution Examples

- (P Q ¬R S) with (P ¬Q W X) yields (P ¬R S W X)
  - Order of literals within clauses does not matter.
- (P Q ¬R S) with (¬P) yields (Q ¬R S)
- (¬R) with (R) yields ( ) or FALSE
- (P Q ¬R S) with (P R ¬S W X) yields (P Q ¬R R W X) or (P Q S ¬S W X) or TRUE
- (P ¬Q R ¬S) with (P ¬Q R ¬S) yields None possible
- (P ¬Q ¬S W) with (P R ¬S X) yields None possible
- ((¬ A) (¬ B) (¬ C) (¬ D)) with ((¬ C) D) yields ((¬ A) (¬ B) (¬ C))
- ((¬ A) (¬ B) (¬ C)) with ((¬ A) C) yields ((¬ A) (¬ B))
- ((¬ A) (¬ B)) with (B) yields (¬ A)
- (A C) with (A (¬ C)) yields (A)
- (¬ A) with (A) yields ( ) or FALSE

Only Resolve ONE Literal Pair!

If more than one pair, result always = TRUE.

Useless!! Always simplifies to TRUE!!

No!

(OR A B C D)

(OR ¬A ¬B F G)

(OR C D F G)

No! This is wrong!

Yes! (but = TRUE)

(OR A B C D)

(OR ¬A ¬B F G)

(OR B ¬B C D F G)

Yes! (but = TRUE)

No!

(OR A B C D)

(OR ¬A ¬B ¬C)

(OR D)

No! This is wrong!

Yes! (but = TRUE)

(OR A B C D)

(OR ¬A ¬B ¬C)

(OR A ¬A B ¬B D)

Yes! (but = TRUE)

Resolution Algorithm

- The resolution algorithm tries to prove: $KB \models \alpha$ *equivalent to* $KB \wedge \neg\alpha$ *unsatisfiable*

- Generate all new sentences from KB and the (negated) query.
- One of two things can happen:

1. We find $P \wedge \neg P$ which is unsatisfiable. I.e. we can entail the query.

2. We find no contradiction: there is a model that satisfies the sentence $KB \wedge \neg\alpha$ (non-trivial) and hence we cannot entail the query.

Resolution example

Resulting Knowledge Base stated in CNF

- "Laws of Physics" in the Wumpus World:

  $(\neg B_{1,1} \quad P_{1,2} \quad P_{2,1})$
  $(\neg P_{1,2} \quad B_{1,1})$
  $(\neg P_{2,1} \quad B_{1,1})$

- Particular facts about a specific instance:

  $(\neg\, B_{1,1})$

- Negated goal or query sentence:

  $(P_{1,2})$

Resolution example

A Resolution proof ending in ( )

- Knowledge Base at start of proof:

  $(\neg B_{1,1} \quad P_{1,2} \quad P_{2,1})$
  $(\neg P_{1,2} \quad B_{1,1})$
  $(\neg P_{2,1} \quad B_{1,1})$
  $(\neg\ B_{1,1})$
  $(P_{1,2})$

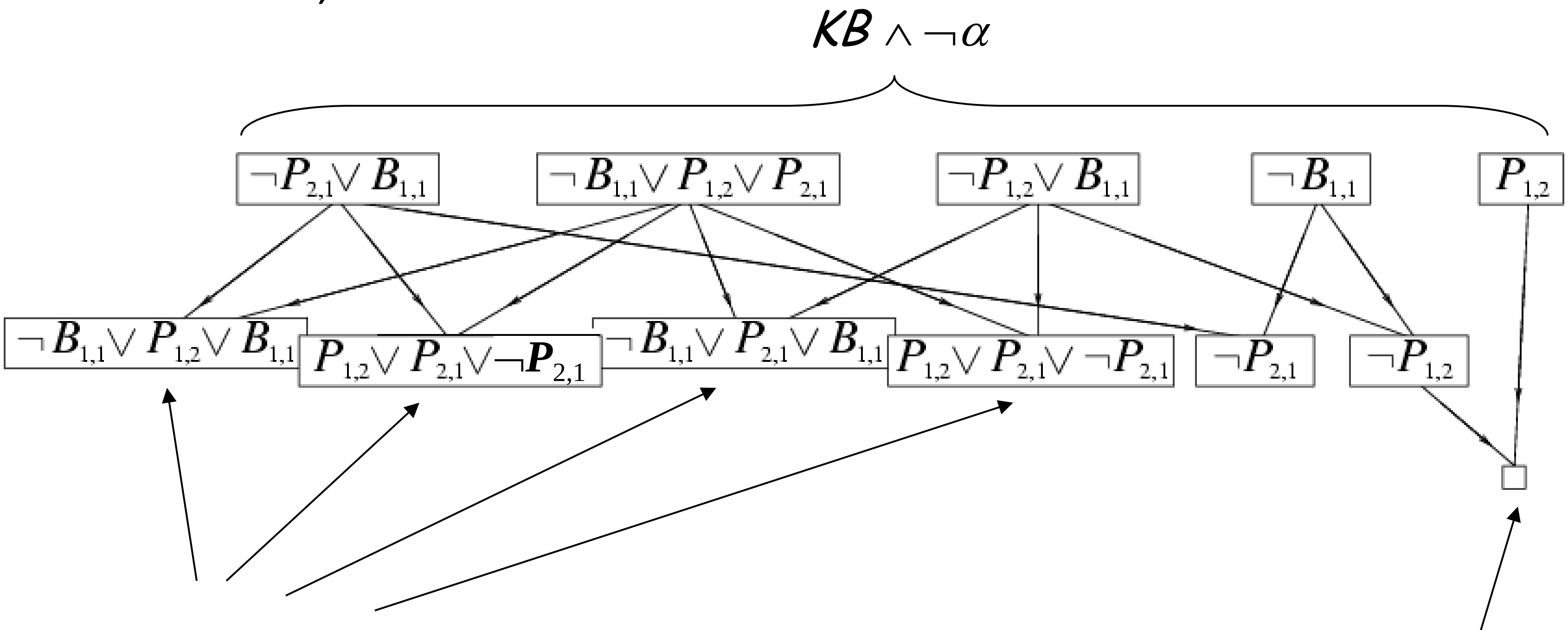
A resolution proof ending in ( ):

- Resolve $(\neg P_{1,2} \quad B_{1,1})$ and $(\neg\ B_{1,1})$ to give $(\neg P_{1,2})$
- Resolve $(\neg P_{1,2})$ and $(P_{1,2})$ to give $(\ )$

- Consequently, the goal or query sentence is entailed by KB.
- Of course, there are many other proofs, which are OK iff correct.

Resolution example

- $KB = (B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})) \wedge \neg B_{1,1}$
- $\alpha = \neg P_{1,2}$



$KB \wedge \neg\alpha$

$\neg P_{2,1} \vee B_{1,1}$ | $\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}$ | $\neg P_{1,2} \vee B_{1,1}$ | $\neg B_{1,1}$ | $P_{1,2}$

$\neg B_{1,1} \vee P_{1,2} \vee B_{1,1}$ | $P_{1,2} \vee P_{2,1} \vee \neg P_{2,1}$ | $\neg B_{1,1} \vee P_{2,1} \vee B_{1,1}$ | $P_{1,2} \vee P_{2,1} \vee \neg P_{2,1}$ | $\neg P_{2,1}$ | $\neg P_{1,2}$

□

True!

A sentence in KB is not “used up” when it is used in a resolution step. It is true, remains true, and is still in KB.

False in all worlds

Detailed Resolution Proof Example

- **In words:** *If the unicorn is mythical, then it is immortal, but if it is not mythical, then it is a mortal mammal. If the unicorn is either immortal or a mammal, then it is horned. The unicorn is magical if it is horned.*

  Prove that the unicorn is both magical and horned.

  ((NOT Y) (NOT R)) **(M Y)** **(R Y)** **(H (NOT M))**

  (H R) **((NOT H) G)** **((NOT G) (NOT H))**

- **Fourth, produce a resolution proof ending in ( ):**
- Resolve (¬H ¬G) and (¬H G) to give (¬H)
- Resolve (¬Y ¬R) and (Y M) to give (¬R M)
- Resolve (¬R M) and (R H) to give (M H)
- Resolve (M H) and (¬M H) to give (H)
- Resolve (¬H) and (H) to give ( )

- Of course, there are many other proofs, which are OK iff correct.

Propositional Logic --- Summary

- Logical agents apply inference to a knowledge base to derive new information and make decisions

- Basic concepts of logic:
  - syntax: formal structure of sentences
  - semantics: truth of sentences wrt models
  - entailment: necessary truth of one sentence given another
  - inference: deriving sentences from other sentences
  - soundness: derivations produce only entailed sentences
  - completeness: derivations can produce all entailed sentences
  - valid: sentence is true in every model (a tautology)

- Logical equivalences allow syntactic manipulations

- Propositional logic lacks expressive power
  - Can only state specific facts about the world.
  - Cannot express general rules about the world
    (use First Order Predicate Logic instead)

Review First-Order Logic

Chapter 8.1-8.5, 9.1-9.2, 9.5.1-9.5.5

- Syntax & Semantics
  - Predicate symbols, function symbols, constant symbols, variables, quantifiers.
  - Models, symbols, and interpretations
- De Morgan’s rules for quantifiers
- Nested quantifiers
  - Difference between “$\forall x \exists y\ P(x, y)$” and “$\exists x \forall y\ P(x, y)$”
- Translate simple English sentences to FOPC and back
  - $\forall x \exists y\ Likes(x, y) \Leftrightarrow$ “Everyone has someone that they like.”
  - $\exists x \forall y\ Likes(x, y) \Leftrightarrow$ “There is someone who likes every person.”
- Unification and the Most General Unifier
- Inference in FOL
  - By Resolution (CNF)
  - By Backward & Forward Chaining (Horn Clauses)
- Knowledge engineering in FOL

Syntax of FOL: Basic elements

Constants	KingJohn, 2, UCI,...
Predicates	Brother, >,...
Functions	Sqrt, LeftLegOf,...
Variables	x, y, a, b,...
Quantifiers	∀, ∃
Connectives	¬, ∧, ∨, ⇒, ⇔ (standard)
Equality	= (but causes difficulties….)

Syntax of FOL: Basic syntax elements are symbols

- **Constant** Symbols (correspond to English nouns)
  - Stand for objects in the world.
    - E.g., KingJohn, 2, UCI, ...

- **Predicate** Symbols (correspond to English verbs)
  - Stand for relations (maps a tuple of objects to a **truth-value**)
    - E.g., Brother(Richard, John), greater_than(3,2), ...
  - P(x, y) is usually read as “x is P of y.”
    - E.g., Mother(Ann, Sue) is usually “Ann is Mother of Sue.”

- **Function** Symbols (correspond to English nouns)
  - Stand for functions (maps a tuple of objects to an **object**)
    - E.g., Sqrt(3), LeftLegOf(John), ...

- **Model** (world) = set of domain objects, relations, functions
- **Interpretation** maps symbols onto the model (world)
  - Very many interpretations are possible for each KB and world!
  - The KB is to rule out those inconsistent with our knowledge.

Syntax of FOL: Terms

- **Term** = logical expression that **refers to an object**

- **There are two kinds of terms:**

  - **Constant Symbols** stand for (or name) objects:
    - E.g., KingJohn, 2, UCI, Wumpus, ...

  - **Function Symbols** map tuples of objects to an object:
    - E.g., LeftLeg(KingJohn), Mother(Mary), Sqrt(x)
    - This is nothing but a complicated kind of name
      - No “subroutine” call, no “return value”

Syntax of FOL: Atomic Sentences

- **Atomic Sentences** state facts (logical truth values).
  - An **atomic sentence** is a Predicate symbol, optionally followed by a parenthesized list of any argument terms
  - E.g., *Married(Father(Richard), Mother(John))*
  - An **atomic sentence** asserts that some relationship (some predicate) holds among the objects that are its arguments.

- An **Atomic Sentence is true** in a given model if the relation referred to by the predicate symbol holds among the objects (terms) referred to by the arguments.

Syntax of FOL: Connectives & Complex Sentences

- **Complex Sentences** are formed in the same way, using the same logical connectives, as in propositional logic

- The **Logical Connectives**:
  - $\Leftrightarrow$ biconditional
  - $\Rightarrow$ implication
  - $\wedge$ and
  - $\vee$ or
  - $\neg$ negation

- **Semantics** for these logical connectives are the same as we already know from propositional logic.

Syntax of FOL: Variables

- **Variables** range over objects in the world.
- A **variable** is like a **term** because it represents an object.
- A **variable** may be used wherever a **term** may be used.
  - **Variables** may be arguments to functions and predicates.
- (A **term with NO variables** is called a **ground term**.)
- (A **variable not bound by a quantifier** is called **free**.)
  - All variables we will use are bound by a quantifier.

Syntax of FOL: Logical Quantifiers

- There are two **Logical Quantifiers:**
  - **Universal:** $\forall$ x P(x) means "For all x, P(x)."
    - The "upside-down A" reminds you of "ALL."
    - Some texts put a comma after the variable: $\forall$ x, P(x)
  - **Existential:** $\exists$ x P(x) means "There exists x such that, P(x)."
    - The "backward E" reminds you of "EXISTS."
    - Some texts put a comma after the variable: $\exists$ x, P(x)

- You can ALWAYS convert one quantifier to the other.
  - $\forall x\ P(x) \equiv \neg\exists x\ \neg P(x)$
  - $\exists x\ P(x) \equiv \neg\forall x\ \neg P(x)$
  - **RULES:** $\forall \equiv \neg\exists\neg$ and $\exists \equiv \neg\forall\neg$

- **RULES:** To move negation "in" across a quantifier,
  Change the quantifier to "the other quantifier"
  and negate the predicate on "the other side."
  - $\neg\forall x\ P(x) \equiv \neg\ \neg\exists x\ \neg P(x) \equiv \exists x\ \neg P(x)$
  - $\neg\exists x\ P(x) \equiv \neg\ \neg\forall x\ \neg P(x) \equiv \forall x\ \neg P(x)$

Universal Quantification ∀

- ∀ x means "for all x it is true that…"

- Allows us to make statements about all objects that have certain properties

- Can now state general rules:

  ∀ x King(x) => Person(x) "All kings are persons."
  ∀ x Person(x) => HasHead(x) "Every person has a head."
  ∀ i Integer(i) => Integer(plus(i,1)) "If i is an integer then i+1 is an integer."

- **Note: ∀ x King(x) ∧ Person(x) is not correct!**

  This would imply that all objects x are Kings and are People (!)

  ∀ x King(x) => Person(x) is the correct way to say this

- **Note that => is the natural connective to use with ∀ .**

Existential Quantification ∃

- ∃ x means "there exists an x such that…."
  - There is in the world at least one such object x

- Allows us to make statements about some object without naming it, or even knowing what that object is:

  ∃ x King(x) "Some object is a king."
  ∃ x Lives_in(John, Castle(x)) "John lives in somebody's castle."
  ∃ i Integer(i) ∧ Greater(i,0) "Some integer is greater than zero."

- **Note: ∃ i Integer(i) ⇒ Greater(i,0) is not correct!**

  It is vacuously true if anything in the world were not an integer (!)

  ∃ i Integer(i) ∧ Greater(i,0) is the correct way to say this

- **Note that ∧ is the natural connective to use with ∃ .**

Combining Quantifiers --- Order (Scope)

The order of “unlike” quantifiers is important.

Like nested variable scopes in a programming language.

Like nested ANDs and ORs in a logical sentence.

$\forall x \exists y$ Loves(x,y)

- For everyone (“all x”) there is someone (“exists y”) whom they love.
- There might be a different y for each x (y is inside the scope of x)

$\exists y \forall x$ Loves(x,y)

- There is someone (“exists y”) whom everyone loves (“all x”).
- Every x loves the same y (x is inside the scope of y)

Clearer with parentheses: $\exists y (\forall x$ Loves(x,y))

The order of “like” quantifiers does not matter.

Like nested ANDs and ANDs in a logical sentence

$\forall x \forall y P(x, y) \equiv \forall y \forall x P(x, y)$

$\exists x \exists y P(x, y) \equiv \exists y \exists x P(x, y)$

De Morgan’s Law for Quantifiers

De Morgan’s Rule	**Generalized De Morgan’s Rule**
$P \wedge Q \equiv \neg (\neg P \vee \neg Q)$	$\forall x\, P(x) \equiv \neg \exists x \neg P(x)$
$P \vee Q \equiv \neg (\neg P \wedge \neg Q)$	$\exists x\, P(x) \equiv \neg \forall x \neg P(x)$
$\neg (P \wedge Q) \equiv (\neg P \vee \neg Q)$	$\neg \forall x\, P(x) \equiv \exists x \neg P(x)$
$\neg (P \vee Q) \equiv (\neg P \wedge \neg Q)$	$\neg \exists x\, P(x) \equiv \forall x \neg P(x)$

AND/OR Rule is simple: if you bring a negation inside a disjunction or a conjunction, always switch between them ($\neg$ OR → AND $\neg$; $\neg$ AND → OR $\neg$).

QUANTIFIER Rule is similar: if you bring a negation inside a universal or existential, always switch between them ($\neg \exists \rightarrow \forall \neg$; $\neg \forall \rightarrow \exists \neg$).

Fun with sentences

Brothers are siblings

$$\forall\, x, y \;\; Brother(x, y) \;\Rightarrow\; Sibling(x, y).$$

"Sibling" is symmetric

$$\forall\, x, y \;\; Sibling(x, y) \;\Leftrightarrow\; Sibling(y, x).$$

One's mother is one's female parent

$$\forall\, x, y \;\; Mother(x, y) \;\Leftrightarrow\; (Female(x) \land Parent(x, y)).$$

A first cousin is a child of a parent's sibling

$$\forall\, x, y \;\; FirstCousin(x, y) \;\Leftrightarrow\; \exists\, p, ps \;\; Parent(p, x) \land Sibling(ps, p) \land Parent(ps, y)$$

More fun with sentences

- **“All persons are mortal.”**
- [Use: Person(x), Mortal (x)]

More fun with sentences

- **“All persons are mortal.”**

  [Use: Person(x), Mortal (x)]

- $\forall x \; Person(x) \Rightarrow Mortal(x)$

- **Equivalent Forms:**
- $\forall x \; \neg Person(x) \vee Mortal(x)$

- **Common Mistakes:**
- $\forall x \; Person(x) \wedge Mortal(x)$

More fun with sentences

- **“Fifi has a sister who is a cat.”**
- [Use: Sister(Fifi, x), Cat(x)]
-

More fun with sentences

- **“Fifi has a sister who is a cat.”**
- [Use: Sister(Fifi, x), Cat(x)]
- $\exists$x Sister(Fifi, x) $\wedge$ Cat(x)
- **Common Mistakes:**
- $\exists$x Sister(Fifi, x) $\Rightarrow$ Cat(x)

More fun with sentences

- **“For every food, there is a person who eats that food.”**

  [Use: Food(x), Person(y), Eats(y, x)]

More fun with sentences

- **"For every food, there is a person who eats that food."**

  [Use: Food(x), Person(y), Eats(y, x)]

- $\forall x\ \exists y\ \text{Food}(x) \Rightarrow [\ \text{Person}(y) \wedge \text{Eats}(y, x)\ ]$

- **Equivalent Forms:**
- $\forall x\ \text{Food}(x) \Rightarrow \exists y\ [\ \text{Person}(y) \wedge \text{Eats}(y, x)\ ]$
- $\forall x\ \exists y\ \neg\text{Food}(x) \vee [\ \text{Person}(y) \wedge \text{Eats}(y, x)\ ]$
- $\forall x\ \exists y\ [\ \neg\text{Food}(x) \vee \text{Person}(y)\ ] \wedge [\neg\ \text{Food}(x) \vee \text{Eats}(y, x)\ ]$
- $\forall x\ \exists y\ [\ \text{Food}(x) \Rightarrow \text{Person}(y)\ ] \wedge [\ \text{Food}(x) \Rightarrow \text{Eats}(y, x)\ ]$

- **Common Mistakes:**
- $\forall x\ \exists y\ [\ \text{Food}(x) \wedge \text{Person}(y)\ ] \Rightarrow \text{Eats}(y, x)$
- $\forall x\ \exists y\ \text{Food}(x) \wedge \text{Person}(y) \wedge \text{Eats}(y, x)$

More fun with sentences

- **“Every person eats every food.”**

  [Use: Person (x), Food (y), Eats(x, y)]

More fun with sentences

- **"Every person eats every food."**
  [Use: Person (x), Food (y), Eats(x, y)]
- $\forall x\ \forall y\ [\ \text{Person}(x) \wedge \text{Food}(y)\ ] \Rightarrow \text{Eats}(x, y)$
- **Equivalent Forms:**
- $\forall x\ \forall y\ \neg\text{Person}(x) \vee \neg\text{Food}(y) \vee \text{Eats}(x, y)$
- $\forall x\ \forall y\ \text{Person}(x) \Rightarrow [\ \text{Food}(y) \Rightarrow \text{Eats}(x, y)\ ]$
- $\forall x\ \forall y\ \text{Person}(x) \Rightarrow [\ \neg\text{Food}(y) \vee \text{Eats}(x, y)\ ]$
- $\forall x\ \forall y\ \neg\text{Person}(x) \vee [\ \text{Food}(y) \Rightarrow \text{Eats}(x, y)\ ]$
- **Common Mistakes:**
- $\forall x\ \forall y\ \text{Person}(x) \Rightarrow [\text{Food}(y) \wedge \text{Eats}(x, y)\ ]$
- $\forall x\ \forall y\ \text{Person}(x) \wedge \text{Food}(y) \wedge \text{Eats}(x, y)$

More fun with sentences

- **“All greedy kings are evil.”**

  [Use: King(x), Greedy(x), Evil(x)]

More fun with sentences

- **“All greedy kings are evil.”**
  [Use: King(x), Greedy(x), Evil(x)]
- $\forall x\ [\ \text{Greedy}(x) \wedge \text{King}(x)\ ] \Rightarrow \text{Evil}(x)$
- **Equivalent Forms:**
- $\forall x\ \neg\text{Greedy}(x) \vee \neg\text{King}(x) \vee \text{Evil}(x)$
- $\forall x\ \text{Greedy}(x) \Rightarrow [\ \text{King}(x) \Rightarrow \text{Evil}(x)\ ]$
- **Common Mistakes:**
- $\forall x\ \text{Greedy}(x) \wedge \text{King}(x) \wedge \text{Evil}(x)$

More fun with sentences

- **“Everyone has a favorite food.”**

  [Use: Person(x), Food(y), Favorite(y, x)]

More fun with sentences

- **“Everyone has a favorite food.”**

  [Use: Person(x), Food(y), Favorite(y, x)]

- **Equivalent Forms:**
- $\forall x\ \exists y\ \text{Person}(x) \Rightarrow [\ \text{Food}(y) \wedge \text{Favorite}(y, x)\ ]$
- $\forall x\ \text{Person}(x) \Rightarrow \exists y\ [\ \text{Food}(y) \wedge \text{Favorite}(y, x)\ ]$
- $\forall x\ \exists y\ \neg\text{Person}(x) \vee [\ \text{Food}(y) \wedge \text{Favorite}(y, x)\ ]$
- $\forall x\ \exists y\ [\ \neg\text{Person}(x) \vee \text{Food}(y)\ ] \wedge [\ \neg\text{Person}(x)$

  $\vee\ \text{Favorite}(y, x)\ ]$
- $\forall x\ \exists y\ [\text{Person}(x) \Rightarrow \text{Food}(y)\ ] \wedge [\ \text{Person}(x) \Rightarrow \text{Favorite}(y, x)\ ]$

- **Common Mistakes:**
- $\forall x\ \exists y\ [\ \text{Person}(x) \wedge \text{Food}(y)\ ] \Rightarrow \text{Favorite}(y, x)$
- $\forall x\ \exists y\ \text{Person}(x) \wedge \text{Food}(y) \wedge \text{Favorite}(y, x)$

More fun with sentences

- **“There is someone at UCI who is smart.”**

  [Use: Person(x), At(x, UCI), Smart(x)]

More fun with sentences

- **“There is someone at UCI who is smart.”**

  [Use: Person(x), At(x, UCI), Smart(x)]

- $\exists x$ Person(x) $\wedge$ At(x, UCI) $\wedge$ Smart(x)

- **Common Mistakes:**
- $\exists x$ [Person(x) $\wedge$ At(x, UCI)] $\Rightarrow$ Smart(x)

More fun with sentences

- **“Everyone at UCI is smart.”**

  [Use: Person(x), At(x, UCI), Smart(x)]

More fun with sentences

- **“Everyone at UCI is smart.”**
  [Use: Person(x), At(x, UCI), Smart(x)]
- $\forall x\ [\text{Person}(x) \wedge \text{At}(x, \text{UCI})\ ] \Rightarrow \text{Smart}(x)$
- **Equivalent Forms:**
- $\forall x\ \neg[\text{Person}(x) \wedge \text{At}(x, \text{UCI})\ ] \vee \text{Smart}(x)$
- $\forall x\ \neg\text{Person}(x) \vee \neg\text{At}(x, \text{UCI}) \vee \text{Smart}(x)$
- **Common Mistakes:**
- $\forall x\ \text{Person}(x) \wedge \text{At}(x, \text{UCI}) \wedge \text{Smart}(x)$
- $\forall x\ \text{Person}(x) \Rightarrow [\text{At}(x, \text{UCI}) \wedge \text{Smart}(x)\ ]$
-

More fun with sentences

- **“Every person eats some food.”**

  [Use: Person (x), Food (y), Eats(x, y)]

More fun with sentences

- **“Every person eats some food.”**

  [Use: Person (x), Food (y), Eats(x, y)]

- $\forall x\ \exists y\ \text{Person}(x) \Rightarrow [\ \text{Food}(y) \wedge \text{Eats}(x, y)\ ]$
-
- **Equivalent Forms:**
- $\forall x\ \text{Person}(x) \Rightarrow \exists y\ [\ \text{Food}(y) \wedge \text{Eats}(x, y)\ ]$
- $\forall x\ \exists y\ \neg\text{Person}(x) \vee [\ \text{Food}(y) \wedge \text{Eats}(x, y)\ ]$
- $\forall x\ \exists y\ [\ \neg\text{Person}(x) \vee \text{Food}(y)\ ] \wedge [\ \neg\text{Person}(x) \vee \text{Eats}(x, y)\ ]$

- **Common Mistakes:**
- $\forall x\ \exists y\ [\ \text{Person}(x) \wedge \text{Food}(y)\ ] \Rightarrow \text{Eats}(x, y)$
- $\forall x\ \exists y\ \text{Person}(x) \wedge \text{Food}(y) \wedge \text{Eats}(x, y)$
-

More fun with sentences

- **“Some person eats some food.”**

  [Use: Person (x), Food (y), Eats(x, y)]

More fun with sentences

- **“Some person eats some food.”**

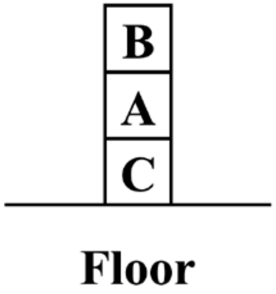
  [Use: Person (x), Food (y), Eats(x, y)]

- $\exists x\ \exists y\ \text{Person}(x) \wedge \text{Food}(y) \wedge \text{Eats}(x, y)$

- **Common Mistakes:**
- $\exists x\ \exists y\ [\ \text{Person}(x) \wedge \text{Food}(y)\ ] \Rightarrow \text{Eats}(x, y)$

Semantics: Interpretation

- An interpretation of a sentence is an assignment that maps
  - Object constants to objects in the worlds,
  - n-ary function symbols to n-ary functions in the world,
  - n-ary relation symbols to n-ary relations in the world
- Given an interpretation, an atomic sentence has the value “true” if it denotes a relation that holds for those individuals denoted in the terms. Otherwise it has the value “false”
  - Example: Block world:
    - A, B, C, floor, On, Clear
  - World:
  - On(A,B) is false, Clear(B) is true, On(C,Floor) is true…
    - Under an interpretation that maps symbol A to block A, symbol B to block B, symbol C to block C, symbol Floor to the floor



Semantics: Models and Definitions

•An interpretation and possible world **satisfies** a wff (sentence) if the wff has the value “true” under that interpretation in that possible world.

•Model: A domain and an interpretation that satisfies a wff is a **model** of that wff

•Validity: Any wff that has the value “true” in all possible worlds and under all interpretations is **valid.**

•Any wff that does not have a model under any interpretation is inconsistent or **unsatisfiable.**

•Any wff that is true in at least one possible world under at least one interpretation is **satisfiable**.

•If a wff w has a value true under all the models of a set of sentences KB then KB logically **entails** w.

Conversion to CNF

- Everyone who loves all animals is loved by someone:

$$\forall \mathrm{x}\ [\forall \mathrm{y}\ \mathit{Animal}(y) \Rightarrow \mathit{Loves}(x,y)] \Rightarrow [\exists \mathrm{y}\ \mathit{Loves}(y,x)]$$

1. Eliminate biconditionals and implications

$$\forall \mathrm{x}\ [\neg\forall \mathrm{y}\ \neg\mathit{Animal}(y) \vee \mathit{Loves}(x,y)] \vee [\exists \mathrm{y}\ \mathit{Loves}(y,x)]$$

2. Move $\neg$ inwards:

$$\neg\forall \mathrm{x}\ \mathrm{p} \equiv \exists \mathrm{x}\ \neg \mathrm{p},\quad \neg\ \exists \mathrm{x}\ \mathrm{p} \equiv \forall \mathrm{x}\ \neg \mathrm{p}$$

$$\forall \mathrm{x}\ [\exists \mathrm{y}\ \neg(\neg\mathit{Animal}(y) \vee \mathit{Loves}(x,y))] \vee [\exists \mathrm{y}\ \mathit{Loves}(y,x)]$$
$$\forall \mathrm{x}\ [\exists \mathrm{y}\ \neg\neg\mathit{Animal}(y) \wedge \neg\mathit{Loves}(x,y)] \vee [\exists \mathrm{y}\ \mathit{Loves}(y,x)]$$
$$\forall \mathrm{x}\ [\exists \mathrm{y}\ \mathit{Animal}(y) \wedge \neg\mathit{Loves}(x,y)] \vee [\exists \mathrm{y}\ \mathit{Loves}(y,x)]$$

Conversion to CNF contd.

3. Standardize variables: each quantifier should use a different one

$\forall x\ [\exists y\ \mathit{Animal}(y) \land \neg \mathit{Loves}(x,y)] \lor [\exists z\ \mathit{Loves}(z,x)]$

4. Skolemize: a more general form of existential instantiation.

Each existential variable is replaced by a Skolem function of the enclosing universally quantified variables:

$\forall x\ [\mathit{Animal}(F(x)) \land \neg \mathit{Loves}(x,F(x))] \lor \mathit{Loves}(G(x),x)$

5. Drop universal quantifiers:

$[\mathit{Animal}(F(x)) \land \neg \mathit{Loves}(x,F(x))] \lor \mathit{Loves}(G(x),x)$

6. Distribute $\lor$ over $\land$:

$[\mathit{Animal}(F(x)) \lor \mathit{Loves}(G(x),x)] \land [\neg \mathit{Loves}(x,F(x)) \lor \mathit{Loves}(G(x),x)]$

Unification

- Recall: Subst(θ, p) = result of substituting θ into sentence p

- Unify algorithm: takes 2 sentences p and q and returns a unifier if one exists

  Unify(p,q) = θ   where Subst(θ, p) = Subst(θ, q)

  where θ is a list of variable/substitution pairs
  that will make p and q syntactically identical

- Example:
  p = Knows(John,x)
  q = Knows(John, Jane)

  Unify(p,q) = {x/Jane}

Unification examples

- simple example: query = Knows(John,x), i.e., who does John know?

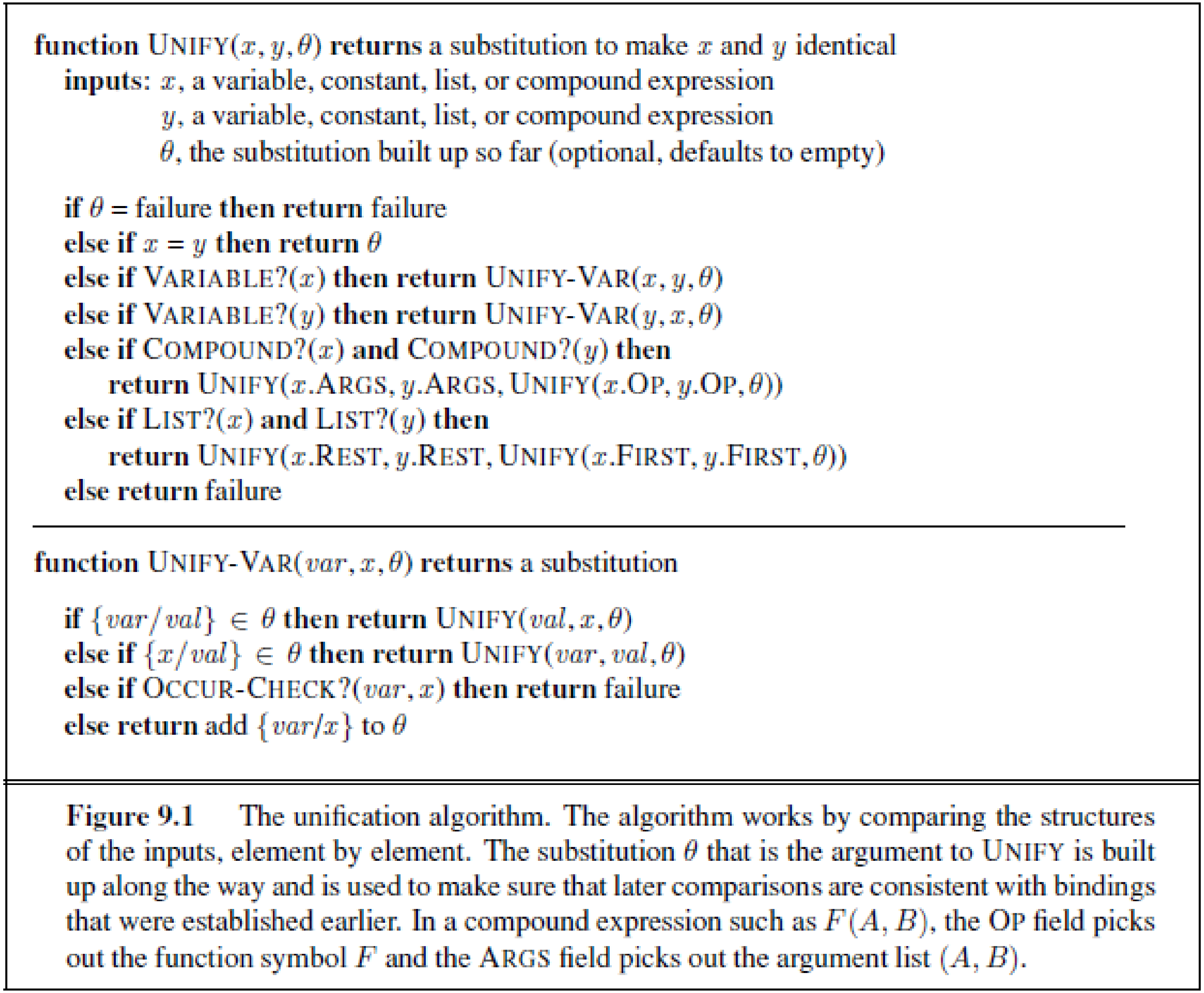
p	q	θ
Knows(John,x)	Knows(John,Jane)	{x/Jane}
Knows(John,x)	Knows(y,OJ)	{x/OJ,y/John}
Knows(John,x)	Knows(y,Mother(y))	{y/John,x/Mother(John)}
Knows(John,x)	Knows(x,OJ)	{fail}

- Last unification fails: only because x can’t take values John and OJ at the same time
  - But we know that if John knows x, and everyone (x) knows OJ, we should be able to infer that John knows OJ
- Problem is due to use of same variable x in both sentences
- Simple solution: Standardizing apart eliminates overlap of variables, e.g., Knows(z,OJ)

Unification examples

- UNIFY(Knows(John, x), Knows(John, Jane))   { x / Jane }
- UNIFY(Knows(John, x), Knows(y, Jane))   { x / Jane, y / John }
- UNIFY(Knows(y, x), Knows(John, Jane))   { x / Jane, y / John }
- UNIFY(Knows(John, x), Knows(y, Father (y)))   { y / John, x / Father (John) }
- UNIFY(Knows(John, F(x)), Knows(y, F(F(z))))   { y / John, x / F (z) }
- UNIFY(Knows(John, F(x)), Knows(y, G(z)))   None
- UNIFY(Knows(John, F(x)), Knows(y, F(G(y))))   { y / John, x / G (John) }

Unification Algorithm



```
function UNIFY($x, y, \theta$) returns a substitution to make $x$ and $y$ identical
   inputs: $x$, a variable, constant, list, or compound expression
           $y$, a variable, constant, list, or compound expression
           $\theta$, the substitution built up so far (optional, defaults to empty)

   if $\theta$ = failure then return failure
   else if $x = y$ then return $\theta$
   else if VARIABLE?($x$) then return UNIFY-VAR($x, y, \theta$)
   else if VARIABLE?($y$) then return UNIFY-VAR($y, x, \theta$)
   else if COMPOUND?($x$) and COMPOUND?($y$) then
      return UNIFY($x$.ARGS, $y$.ARGS, UNIFY($x$.OP, $y$.OP, $\theta$))
   else if LIST?($x$) and LIST?($y$) then
      return UNIFY($x$.REST, $y$.REST, UNIFY($x$.FIRST, $y$.FIRST, $\theta$))
   else return failure
```

```
function UNIFY-VAR($var, x, \theta$) returns a substitution

   if $\{var/val\} \in \theta$ then return UNIFY($val, x, \theta$)
   else if $\{x/val\} \in \theta$ then return UNIFY($var, val, \theta$)
   else if OCCUR-CHECK?($var, x$) then return failure
   else return add $\{var/x\}$ to $\theta$
```

Figure 9.1 The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution θ that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier. In a compound expression such as $F(A, B)$, the OP field picks out the function symbol F and the ARGS field picks out the argument list (A, B).

Example knowledge base

- The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

- Prove that Col. West is a criminal

Example knowledge base (Horn clauses)

... it is a crime for an American to sell weapons to hostile nations:

$American(x) \land Weapon(y) \land Sells(x,y,z) \land Hostile(z) \Rightarrow Criminal(x)$

Nono … has some missiles, i.e., $\exists x\ Owns(Nono,x) \land Missile(x)$:

$Owns(Nono,M_1) \land Missile(M_1)$

… all of its missiles were sold to it by Colonel West

$Missile(x) \land Owns(Nono,x) \Rightarrow Sells(West,x,Nono)$

Missiles are weapons:

$Missile(x) \Rightarrow Weapon(x)$

An enemy of America counts as "hostile“:

$Enemy(x,America) \Rightarrow Hostile(x)$

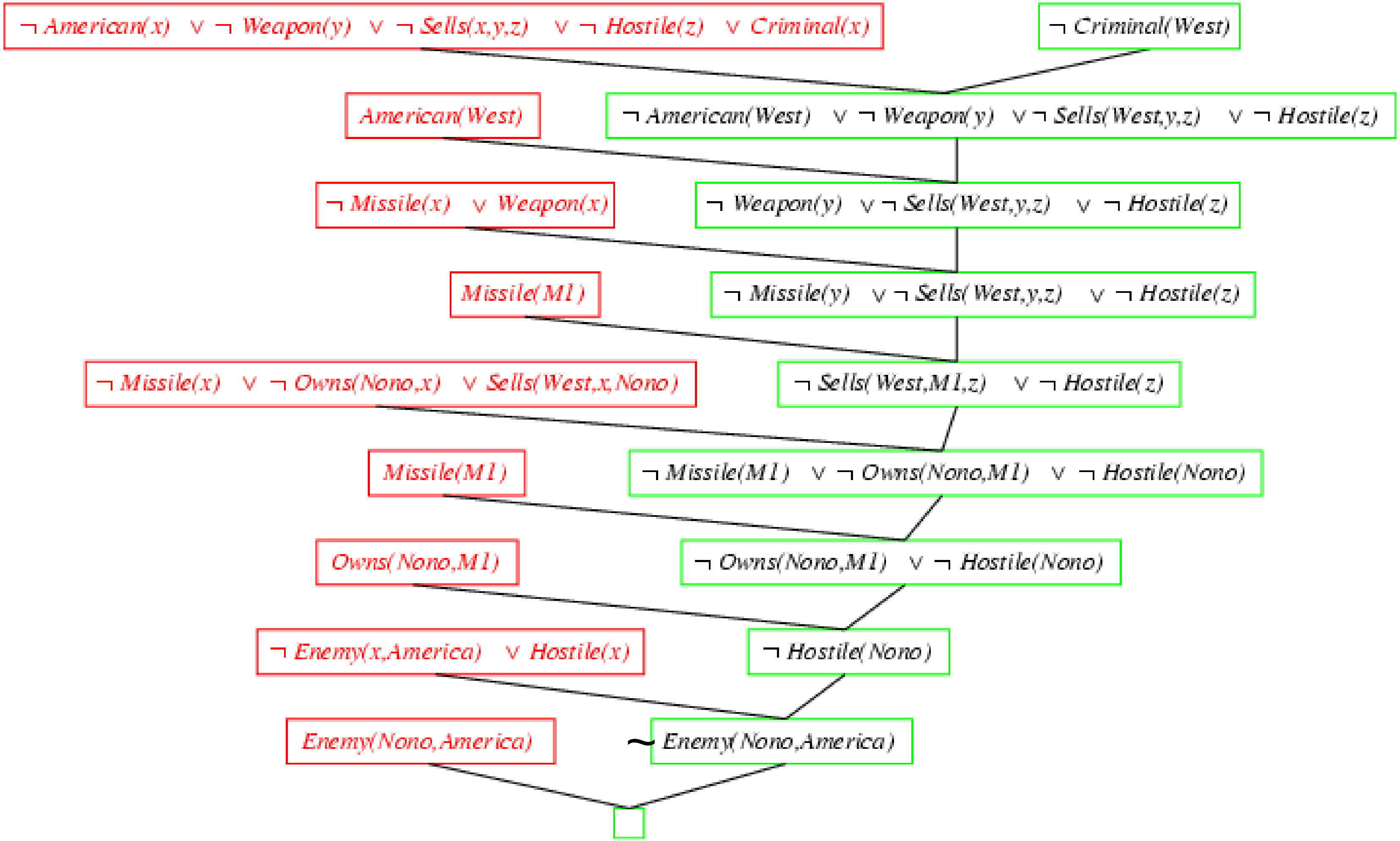
West, who is American …

$American(West)$

The country Nono, an enemy of America …

$Enemy(Nono,America)$

Resolution proof:



¬ American(x) ∨ ¬ Weapon(y) ∨ ¬ Sells(x,y,z) ∨ ¬ Hostile(z) ∨ Criminal(x)

¬ Criminal(West)

American(West)

¬ American(West) ∨ ¬ Weapon(y) ∨ ¬ Sells(West,y,z) ∨ ¬ Hostile(z)

¬ Missile(x) ∨ Weapon(x)

¬ Weapon(y) ∨ ¬ Sells(West,y,z) ∨ ¬ Hostile(z)

Missile(M1)

¬ Missile(y) ∨ ¬ Sells(West,y,z) ∨ ¬ Hostile(z)

¬ Missile(x) ∨ ¬ Owns(Nono,x) ∨ Sells(West,x,Nono)

¬ Sells(West,M1,z) ∨ ¬ Hostile(z)

Missile(M1)

¬ Missile(M1) ∨ ¬ Owns(Nono,M1) ∨ ¬ Hostile(Nono)

Owns(Nono,M1)

¬ Owns(Nono,M1) ∨ ¬ Hostile(Nono)

¬ Enemy(x,America) ∨ Hostile(x)

¬ Hostile(Nono)

Enemy(Nono,America)

~ Enemy(Nono,America)

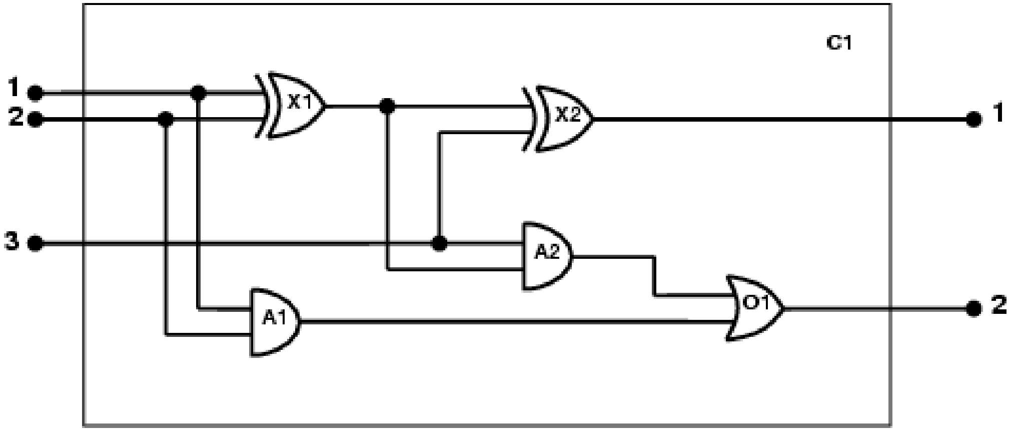
□

Knowledge engineering in FOL

1. Identify the task
2. Assemble the relevant knowledge
3. Decide on a vocabulary of predicates, functions, and constants
4. Encode general knowledge about the domain
5. Encode a description of the specific problem instance
6. Pose queries to the inference procedure and get answers
7. Debug the knowledge base

The electronic circuits domain

One-bit full adder



Possible queries:

- does the circuit function properly?
- what gates are connected to the first input terminal?
- what would happen if one of the gates is broken?

and so on

The electronic circuits domain

1. Identify the task
   - Does the circuit actually add properly?

2. Assemble the relevant knowledge
   - Composed of wires and gates; Types of gates (AND, OR, XOR, NOT)
   -
   - Irrelevant: size, shape, color, cost of gates
   -

3. Decide on a vocabulary
   - Alternatives:
   -

     Type(X_1) = XOR (function)
     Type(X_1, XOR) (binary predicate)
     XOR(X_1)
       (unary predicate)

The electronic circuits domain

4. Encode general knowledge of the domain
   - $\forall t_1,t_2$ Connected(t_1, t_2) $\Rightarrow$ Signal(t_1) = Signal(t_2)
   - $\forall$t Signal(t) = 1 $\vee$ Signal(t) = 0
   - 1 ≠ 0
   - $\forall t_1,t_2$ Connected(t_1, t_2) $\Rightarrow$ Connected(t_2, t_1)
   - $\forall$g Type(g) = OR $\Rightarrow$ Signal(Out(1,g)) = 1 $\Leftrightarrow$ $\exists$n Signal(In(n,g)) = 1
   - $\forall$g Type(g) = AND $\Rightarrow$ Signal(Out(1,g)) = 0 $\Leftrightarrow$ $\exists$n Signal(In(n,g)) = 0
   - $\forall$g Type(g) = XOR $\Rightarrow$ Signal(Out(1,g)) = 1 $\Leftrightarrow$ Signal(In(1,g)) ≠ Signal(In(2,g))
   - $\forall$g Type(g) = NOT $\Rightarrow$ Signal(Out(1,g)) ≠ Signal(In(1,g))

The electronic circuits domain

5. Encode the specific problem instance

Type(X_1) = XOR
Type(A_1) = AND
Type(O_1) = OR

Type(X_2) = XOR
Type(A_2) = AND

Connected(Out(1,X_1),In(1,X_2))
Connected(Out(1,X_1),In(2,A_2))
Connected(Out(1,A_2),In(1,O_1))
Connected(Out(1,A_1),In(2,O_1))
Connected(Out(1,X_2),Out(1,C_1))
Connected(Out(1,O_1),Out(2,C_1))

Connected(In(1,C_1),In(1,X_1))
Connected(In(1,C_1),In(1,A_1))
Connected(In(2,C_1),In(2,X_1))
Connected(In(2,C_1),In(2,A_1))
Connected(In(3,C_1),In(2,X_2))
Connected(In(3,C_1),In(1,A_2))

The electronic circuits domain

6. Pose queries to the inference procedure:

What are the possible sets of values of all the terminals for the adder circuit?

$$\exists i_1,i_2,i_3,o_1,o_2\ \mathrm{Signal}(\mathrm{In}(1,C_1)) = i_1 \land \mathrm{Signal}(\mathrm{In}(2,C_1)) = i_2 \land \mathrm{Signal}(\mathrm{In}(3,C_1)) = i_3 \land \mathrm{Signal}(\mathrm{Out}(1,C_1)) = o_1 \land \mathrm{Signal}(\mathrm{Out}(2,C_1)) = o_2$$

7. Debug the knowledge base

May have omitted assertions like $1 \neq 0$

Review Probability
Chapter 13

- Basic probability notation/definitions:
  - Probability model, unconditional/prior and conditional/posterior probabilities, random variable, (joint) probability distribution, probability density function (pdf), marginal probability, (conditional) independence, normalization, etc.
- Basic probability formulas:
  - E.g., Probability axioms, sum rule, product rule, Bayes' rule.
- How to use Bayes' rule:
  - Naïve Bayes model (naïve Bayes classifier)

Syntax

- Basic element: random variable
- Similar to propositional logic: possible worlds defined by assignment of values to random variables.

- Booleanrandom variables

  e.g., *Cavity (= do I have a cavity?)*
- Discreterandom variables

  e.g., *Weather is one of <sunny,rainy,cloudy,snow>*
- Domain values must be exhaustive and mutually exclusive

- Elementary proposition is an assignment of a value to a random variable:

  e.g., *Weather = sunny; Cavity = false(abbreviated as ¬cavity)*

- Complex propositions formed from elementary propositions and standard logical connectives :

  e.g., *Weather = sunny* $\vee$ *Cavity = false*

Probability

- P(a) is the probability of proposition “a”
  - e.g., P(it will rain in London tomorrow)
  - The proposition a is actually true or false in the real-world

- **<u>Probability Axioms</u>**:
  - $0 \leq P(a) \leq 1$
  - $P(NOT(a)) = 1 - P(a)$ => $\Sigma_A P(A) = 1$
  - $P(true) = 1$
  - $P(false) = 0$
  - $P(A \text{ OR } B) = P(A) + P(B) - P(A \text{ AND } B)$

- Any agent that holds degrees of beliefs that contradict these axioms will act irrationally in some cases

- **Rational agents <u>cannot</u> violate probability theory.**
  - Acting otherwise results in irrational behavior.

Conditional Probability

- P(a|b) is the conditional probability of proposition a, conditioned on knowing that b is true,
  - E.g., P(rain in London tomorrow | raining in London today)
  - P(a|b) is a “posterior” or conditional probability
  - The updated probability that a is true, now that we know b
  - $P(a|b) = P(a \wedge b) / P(b)$
  - Syntax: P(a | b) is the probability of a given that b is true
    - a and b can be any propositional sentences
    - e.g., p(John wins OR Mary wins | Bob wins AND Jack loses)

- P(a|b) obeys the same rules as probabilities,
  - E.g., P(a | b) + P(NOT(a) | b) = 1
  - All probabilities in effect are conditional probabilities
    - E.g., P(a) = P(a | our background knowledge)

Concepts of Probability

- **Unconditional Probability**
  - **P(a)**, the probability of “a” being true, or **P(a=True)**
  - Does not depend on anything else to be true (**unconditional**)
  - Represents the probability prior to further information that may adjust it (**prior**)

- **Conditional Probability**
  - **P(a|b)**, the probability of “a” being true, given that “b” is true
  - Relies on “b” = true (**conditional**)
  - Represents the prior probability adjusted based upon new information “b” (**posterior**)
  - Can be generalized to more than 2 random variables:
    - e.g. P(a|b, c, d)

- **Joint Probability**
  - **P(a, b) = P(a ∧ b)**, the probability of “a” and “b” both being true
  - Can be generalized to more than 2 random variables:
    - e.g. P(a, b, c, d)

Basic Probability Relationships

- $\mathbf{P(A) + P(\neg A) = 1}$
  - Implies that $P(\neg A) = 1 - P(A)$
- $\mathbf{P(A, B) = P(A \wedge B) = P(A) + P(B) - P(A \vee B)}$
  - Implies that $P(A \vee B) = P(A) + P(B) - P(A \wedge B)$
- $\mathbf{P(A \mid B) = P(A, B) / P(B)}$
  - Conditional probability; “Probability of A **given** B”
- $\mathbf{P(A, B) = P(A \mid B)\, P(B)}$
  - Product Rule (Factoring); applies to any number of variables
  - $P(a, b, c, \ldots z) = P(a \mid b, c, \ldots z)\; P(b \mid c, \ldots z)\; P(c | \ldots z) \ldots P(z)$
- $\mathbf{P(A) = \Sigma_{B,C}\, P(A, B, C) = \Sigma_{b \in B, c \in C}\, P(A, b, c)}$
  - Sum Rule (Marginal Probabilities); for any number of variables
  - $P(A, D) = \Sigma_B \; \Sigma_C \; P(A, B, C, D) = \Sigma_{b \in \mathbf{B}} \; \Sigma_{c \in \mathbf{C}} \; P(A, b, c, D)$
- $\mathbf{P(B \mid A) = P(A \mid B)\, P(B) / P(A)}$
  - Bayes’ Rule; for any number of variables

You need to know these !

Summary of Probability Rules

- **Product Rule**:
  - **P(a, b) = P(a|b) P(b) = P(b|a) P(a)**
  - Probability of “a” and “b” occurring is the same as probability of “a” occurring given “b” is true, times the probability of “b” occurring.
    - e.g., *P(rain, cloudy) = P(rain | cloudy) * P(cloudy)*

- **Sum Rule**: (AKA **Law of Total Probability**)
  - **P(a) = Σ_b P(a, b) = Σ_b P(a|b) P(b),** where B is any random variable
  - Probability of “a” occurring is the same as the sum of all joint probabilities including the event, provided the joint probabilities represent all possible events.
  - Can be used to “marginalize” out other variables from probabilities, resulting in prior probabilities also being called marginal probabilities.
    - e.g., *P(rain) = $\Sigma_{Windspeed}$ P(rain, Windspeed)*
      where Windspeed = {0-10mph, 10-20mph, 20-30mph, etc.}

- **Bayes’ Rule**:
  - **P(b|a) = P(a|b) P(b) / P(a)**
  - Acquired from rearranging the product rule.
  - Allows conversion between conditionals, from P(a|b) to P(b|a).
    - e.g., b = disease, a = symptoms
      More natural to encode knowledge as P(a|b) than as P(b|a).

Full Joint Distribution

- We can fully specify a probability space by constructing a **full joint distribution**:
  - A full joint distribution contains a probability for every possible combination of variable values.
  - E.g., P(J=f, M=t, A=t, B=t, E=f)
- From a full joint distribution, the product rule, sum rule, and Bayes' rule can create any desired joint and conditional probabilities.

Computing with Probabilities: Law of Total Probability

Law of Total Probability (aka “summing out” or marginalization)

$$P(a) = \Sigma_b \; P(a, b)$$
$$= \Sigma_b \; P(a \mid b) \, P(b)$$

where B is any random variable

Why is this useful?

Given a joint distribution (e.g., P(a,b,c,d)) we can obtain any “marginal” probability (e.g., P(b)) by summing out the other variables, e.g.,

$$P(b) = \Sigma_a \, \Sigma_c \, \Sigma_d \; P(a, b, c, d)$$

We can compute any conditional probability given a joint distribution, e.g.,

$$P(c \mid b) = \Sigma_a \, \Sigma_d \, P(a, c, d \mid b)$$
$$= \Sigma_a \, \Sigma_d \, P(a, c, d, b) / P(b)$$

where P(b) can be computed as above

Computing with Probabilities: The Chain Rule or Factoring

We can always write

$P(a, b, c, \ldots z) = P(a \mid b, c, \ldots. z)\, P(b, c, \ldots z)$

(by definition of joint probability)

Repeatedly applying this idea, we can write

$P(a, b, c, \ldots z) = P(a \mid b, c, \ldots. z)\, P(b \mid c, .. z)\, P(c \mid .. z) .. P(z)$

This factorization holds for any ordering of the variables

This is the chain rule for probabilities

Independence

- Formal Definition:
  - 2 random variables A and B are independent iff:

    P(a, b) = P(a) P(b), for all values a, b

- Informal Definition:
  - 2 random variables A and B are independent iff:

    P(a | b) = P(a) OR P(b | a) = P(b), for all values a, b
  - P(a | b) = P(a) tells us that knowing b provides no change in our probability for a, and thus b contains no information about a.

- Also known as marginal independence, as all other variables have been marginalized out.

- In practice true independence is very rare:
  - “butterfly in China” effect
  - Conditional independence is much more common and useful

Conditional Independence

- Formal Definition:
  - 2 random variables A and B are conditionally independent given C iff:

    $P(a, b|c) = P(a|c) P(b|c)$, for all values a, b, c

- Informal Definition:
  - 2 random variables A and B are conditionally independent given C iff:

    $P(a|b, c) = P(a|c)$ OR $P(b|a, c) = P(b|c)$, for all values a, b, c
  - $P(a|b, c) = P(a|c)$ tells us that learning about b, given that we already know c, provides no change in our probability for a, and thus b contains no information about a beyond what c provides.

- Naïve Bayes Model:
  - Often a single variable can directly influence a number of other variables, all of which are conditionally independent, given the single variable.
  - E.g., k different symptom variables X_1, X_2, … X_k, and C = disease, reducing to:

    $P(X_1, X_2,\dots. X_K \mid C) = P(C) \prod P(X_i \mid C)$

Examples of Conditional Independence

- **H=Heat, S=Smoke, F=Fire**
  - P(H, S | F) = P(H | F) P(S | F)
  - P(S | F, S) = P(S | F)
  - If we know there is/is not a fire, observing heat tells us no more information about smoke
- **F=Fever, R=RedSpots, M=Measles**
  - P(F, R | M) = P(F | M) P(R | M)
  - P(R | M, F) = P(R | M)
  - If we know we do/don't have measles, observing fever tells us no more information about red spots
- **C=SharpClaws, F=SharpFangs, S=Species**
  - P(C, F | S) = P(C | S) P(F | S)
  - P(F | S, C) = P(F | S)
  - If we know the species, observing sharp claws tells us no more information about sharp fangs

Review Bayesian Networks
Chapter 14.1-5

- **Basic concepts and vocabulary of Bayesian networks.**
  - Nodes represent random variables.
  - Directed arcs represent (informally) direct influences.
  - Conditional probability tables, P(Xi | Parents(Xi)).

- **Given a Bayesian network:**
  - Write down the full joint distribution it represents.

- **Given a full joint distribution in factored form:**
  - Draw the Bayesian network that represents it.

- **Given a variable ordering and background assertions of conditional independence among the variables:**
  - Write down the factored form of the full joint distribution, as simplified by the conditional independence assertions.
- **Use the network to find answers to probability questions about it.**

Bayesian Networks

- Represent dependence/independence via a directed graph
  - Nodes = random variables
  - Edges = direct dependence
- Structure of the graph ⬄ Conditional independence

- Recall the chain rule of repeated conditioning:

$$P(X_1, X_2, X_3 ..., X_N) = P(X_1|X_2, X_3 ..., X_N)P(X_2|X_3, ..., X_N) \cdots P(X_N)$$

$$P(X_1, X_2, X_3 ..., X_N) = \prod_{i=1}^{n} P(X_i|parents(X_i))$$

The full joint distribution

The graph-structured approximation

- Requires that graph is acyclic (no directed cycles)
- 2 components to a Bayesian network
  - The graph structure (conditional independence assumptions)
  - The numerical probabilities (of each variable given its parents)

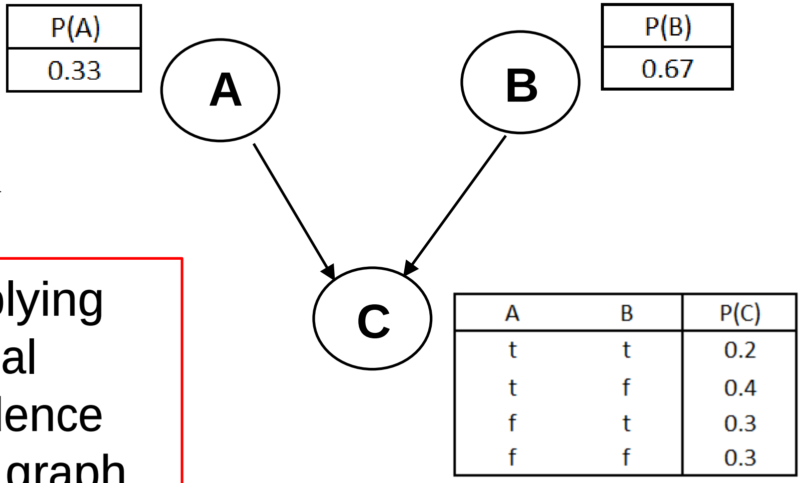
Bayesian Network

- A Bayesian network specifies a joint distribution in a structured form:

Full factorization

$$\begin{aligned} p(A,B,C) &= p(C|A,B)p(A|B)p(B) \\ &= p(C|A,B)p(A)p(B) \end{aligned}$$

After applying conditional independence from the graph



P(A)
0.33

P(B)
0.67

A	B	P(C)
t	t	0.2
t	f	0.4
f	t	0.3
f	f	0.3

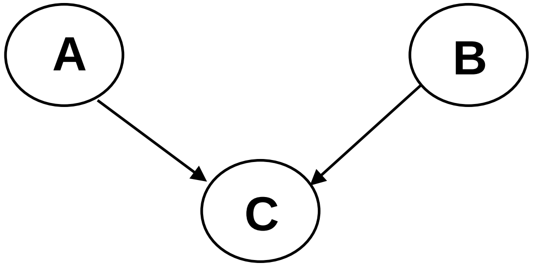
- Dependence/independence represented via a directed graph:

− Node	= random variable
− Directed Edge	= conditional dependence
− Absence of Edge	= conditional independence

- Allows concise view of joint distribution relationships:
  - Graph nodes and edges show conditional relationships between variables.
  - Tables provide probability data.

Examples of 3-way Bayesian Networks

Independent Causes
A Earthquake
B Burglary
C Alarm



Independent Causes:
$p(A,B,C) = p(C|A,B)p(A)p(B)$

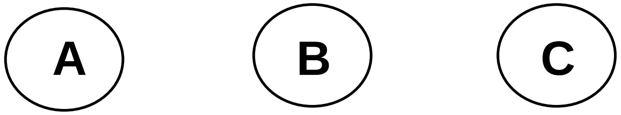
"Explaining away" effect:
Given C, observing A makes B less likely
e.g., earthquake/burglary/alarm example

A and B are (marginally) independent
but become dependent once C is known

You heard alarm, and observe Earthquake
…. It explains away burglary

Nodes: Random Variables
A, B, C
Edges: P(Xi | Parents) → Directed edge from parent nodes to Xi
A → C
B → C

Examples of 3-way Bayesian Networks



Marginal Independence:
p(A,B,C) = p(A) p(B) p(C)

Nodes: Random Variables
A, B, C
Edges: P(Xi | Parents) → Directed edge from parent nodes to Xi
No Edge!

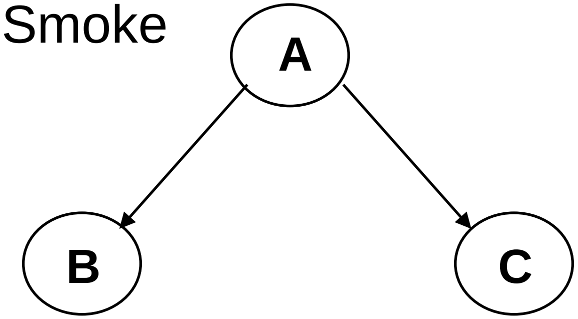
Extended example of 3-way Bayesian Networks

Common Cause

A : Fire

B: Heat

C: Smoke



Conditionally independent effects:
p(A,B,C) = p(B|A)p(C|A)p(A)

B and C are conditionally independent Given A

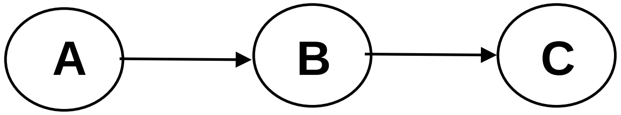
“Where there’s Smoke, there’s Fire.”

If we see Smoke, we can infer Fire.

If we see Smoke, observing Heat tells us very little additional information.

Examples of 3-way Bayesian Networks

Markov Dependence
A Rain on Mon
B Ran on Tue
C Rain on Wed



Markov dependence:
$p(A,B,C) = p(C|B)\ p(B|A)p(A)$

A affects B and B affects C
Given B, A and C are independent

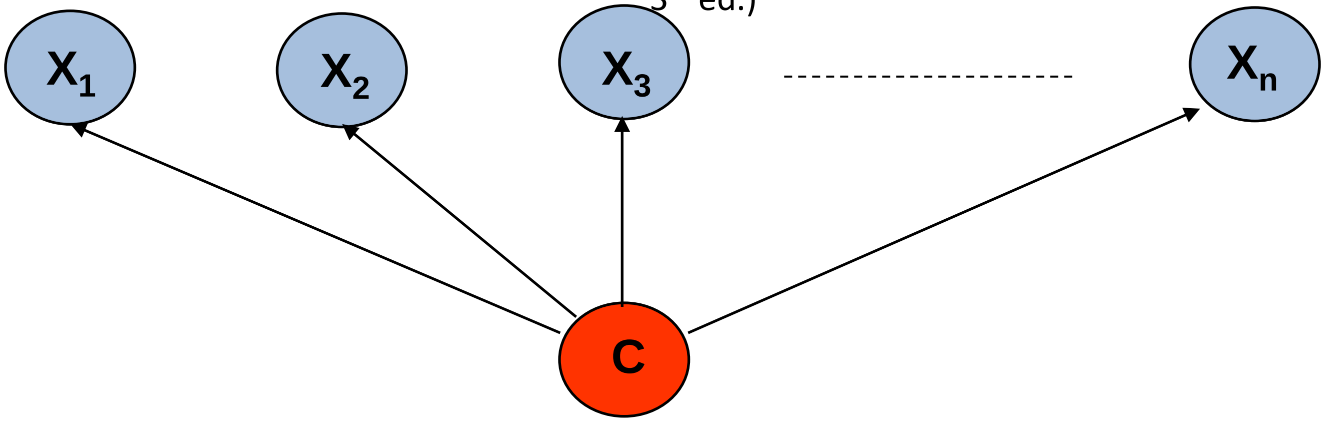
e.g.
If it rains today, it will rain tomorrow with 90%

On Wed morning…
If you know it rained yesterday,
it doesn’t matter whether it rained on Mon

Nodes: Random Variables
A, B, C
Edges: P(Xi | Parents) → Directed edge from parent nodes to Xi
A → B
B → C

Naïve Bayes Model

(section 20.2.2 R&N 3rd ed.)



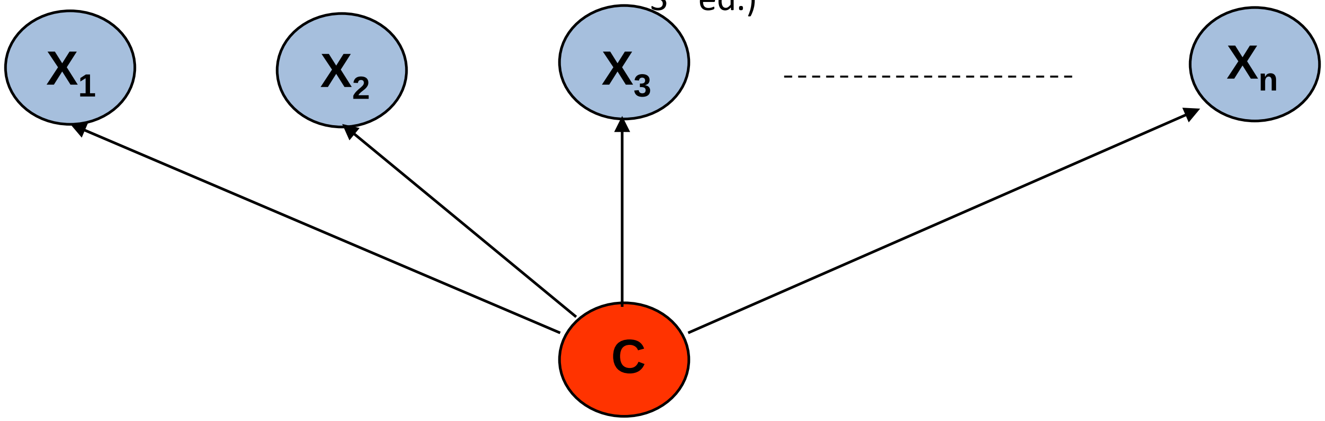
Basic Idea: We want to estimate P(C | X_1,…X_n), but it's hard to think about computing the probability of a class from input attributes of an example.

Solution: Use Bayes' Rule to turn P(C | X_1,…X_n) into a proportionally equivalent expression that involves only P(C) and P(X_1,…X_n | C). Then assume that feature values are conditionally independent given class, which allows us to turn P(X_1,…X_n | C) into Π_i P(X_i | C).

We estimate P(C) easily from the frequency with which each class appears within our training data, and we estimate P(X_i | C) easily from the frequency with which each X_i appears in each class C within our training data.

Naïve Bayes Model

(section 20.2.2 R&N 3rd ed.)



Bayes Rule: $P(C \mid X_1,\ldots X_n)$ is proportional to $P(C)\ \Pi_i\ P(X_i \mid C)$
[note: denominator $P(X_1,\ldots X_n)$ is constant for all classes, may be ignored.]

Features Xi are conditionally independent given the class variable C

- choose the class value c_i with the highest $P(c_i \mid x_1,\ldots, x_n)$
- simple to implement, often works very well
- e.g., spam email classification: X's = counts of words in emails

Conditional probabilities $P(X_i \mid C)$ can easily be estimated from labeled date

- Problem: Need to avoid zeroes, e.g., from limited training data
- Solutions: Pseudo-counts, beta[a,b] distribution, etc.

Naïve Bayes Model (2)

$$P(C \mid X_1,\ldots X_n) = \alpha\ P(C)\ \Pi_i\ P(X_i \mid C)$$

Probabilities $P(C)$ and $P(X_i \mid C)$ can easily be estimated from labeled data

$P(C = c_j) \approx$ #(Examples with class label $C = c_j$) / #(Examples)

$P(X_i = x_{ik} \mid C = c_j)$
$\approx$ #(Examples with attribute value $X_i = x_{ik}$ and class label $C = c_j$)
/ #(Examples with class label $C = c_j$)

Usually easiest to work with logs

$$\log [\, P(C \mid X_1,\ldots X_n) \,] = \log \alpha + \log P(C) + \Sigma \log P(X_i \mid C)$$

DANGER: What if ZERO examples with value $X_i = x_{ik}$ and class label $C = c_j$?
An unseen example with value $X_i = x_{ik}$ will NEVER predict class label $C = c_j$!

Practical solutions: Pseudocounts, e.g., add 1 to every #() , etc.
Theoretical solutions: Bayesian inference, beta distribution, etc.

Bigger Example

- Consider the following 5 binary variables:
  - B = a burglary occurs at your house
  - E = an earthquake occurs at your house
  - A = the alarm goes off
  - J = John calls to report the alarm
  - M = Mary calls to report the alarm
- Sample Query: What is P(B|M, J) ?
- Using full joint distribution to answer this question requires
  - 2^5 - 1= 31 parameters
- Can we use prior domain knowledge to come up with a Bayesian network that requires fewer probabilities?

Constructing a Bayesian Network: Step 1

- Order the variables in terms of influence (may be a partial order)

  e.g., {E, B} -> {A} -> {J, M}

  Generally, order variables to reflect the assumed causal relationships.

- Now, apply the chain rule, and simplify based on assumptions

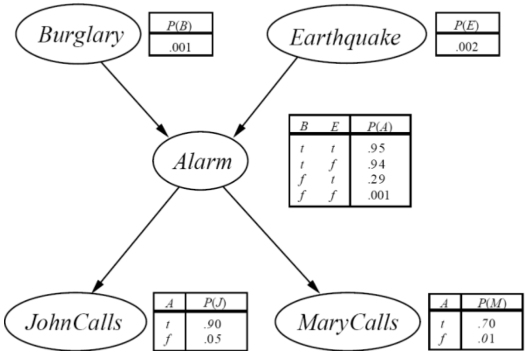
- $P(J, M, A, E, B) = P(J, M \mid A, E, B)\, P(A \mid E, B)\, P(E, B)$

  $\approx P(J, M \mid A) \quad P(A \mid E, B)\, P(E)\, P(B)$

  $\approx P(J \mid A)\, P(M \mid A)\, P(A \mid E, B)\, P(E)\, P(B)$

These conditional independence assumptions are reflected in the graph structure of the Bayesian network

Constructing this Bayesian Network: Step 2



P(B)
.001

P(E)
.002

B	E	P(A)
t	t	.95
t	f	.94
f	t	.29
f	f	.001

A	P(J)
t	.90
f	.05

A	P(M)
t	.70
f	.01

- P(J, M, A, E, B) =
  P(J | A) P(M | A) P(A | E, B) P(E) P(B)

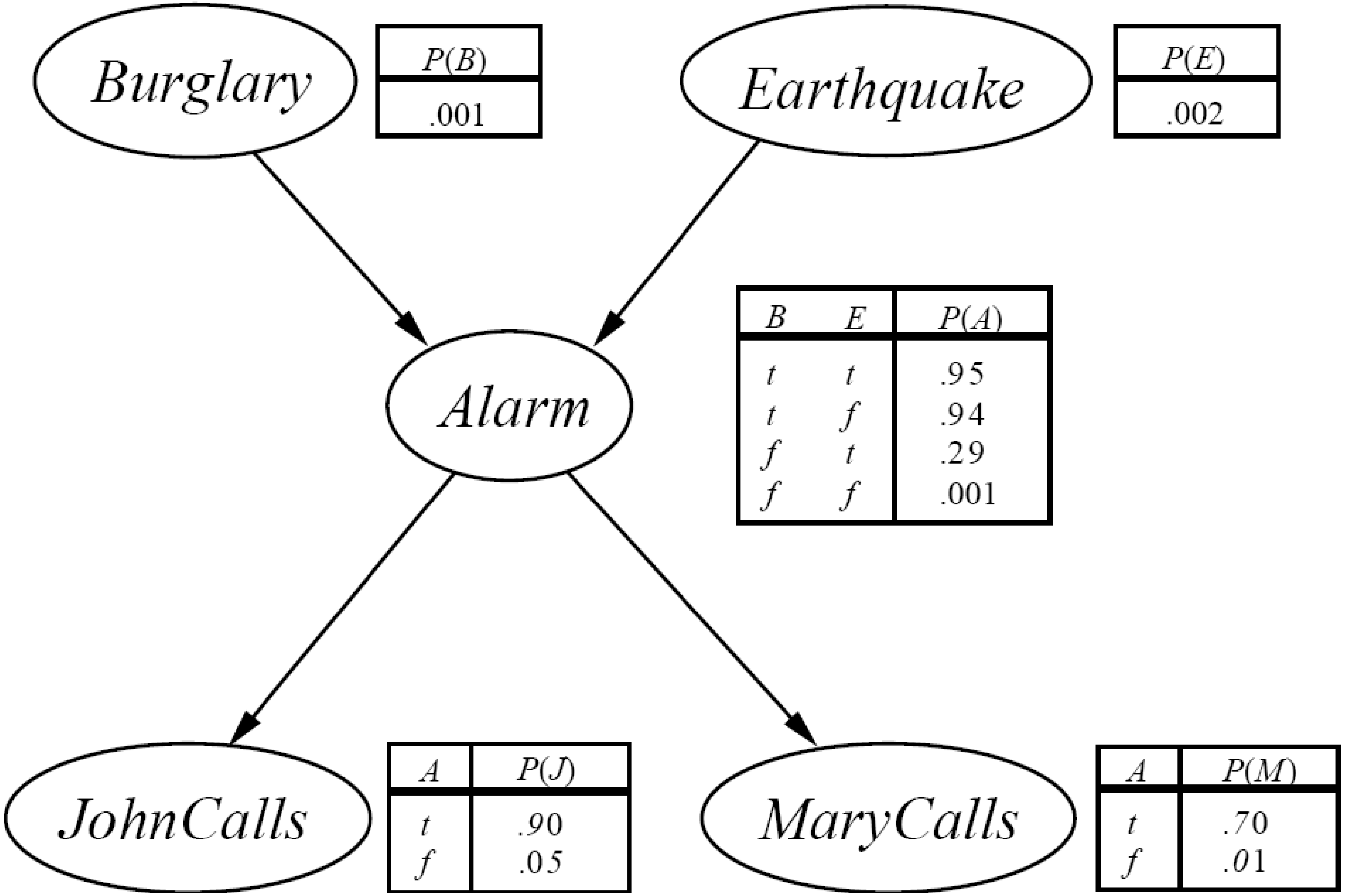
  Parents in the graph ⇔ conditioning variables (RHS)

- There are 3 conditional probability tables (CPDs) to be determined: P(J | A), P(M | A), P(A | E, B)
  - Requiring 2 + 2 + 4 = 8 probabilities

- And 2 marginal probabilities P(E), P(B) -> 2 more probabilities

- Where do these probabilities come from?
  - Expert knowledge
  - From data (relative frequency estimates)
  - Or a combination of both - see discussion in Section 20.1 and 20.2 (optional)

The Resulting Bayesian Network



P(B)
.001

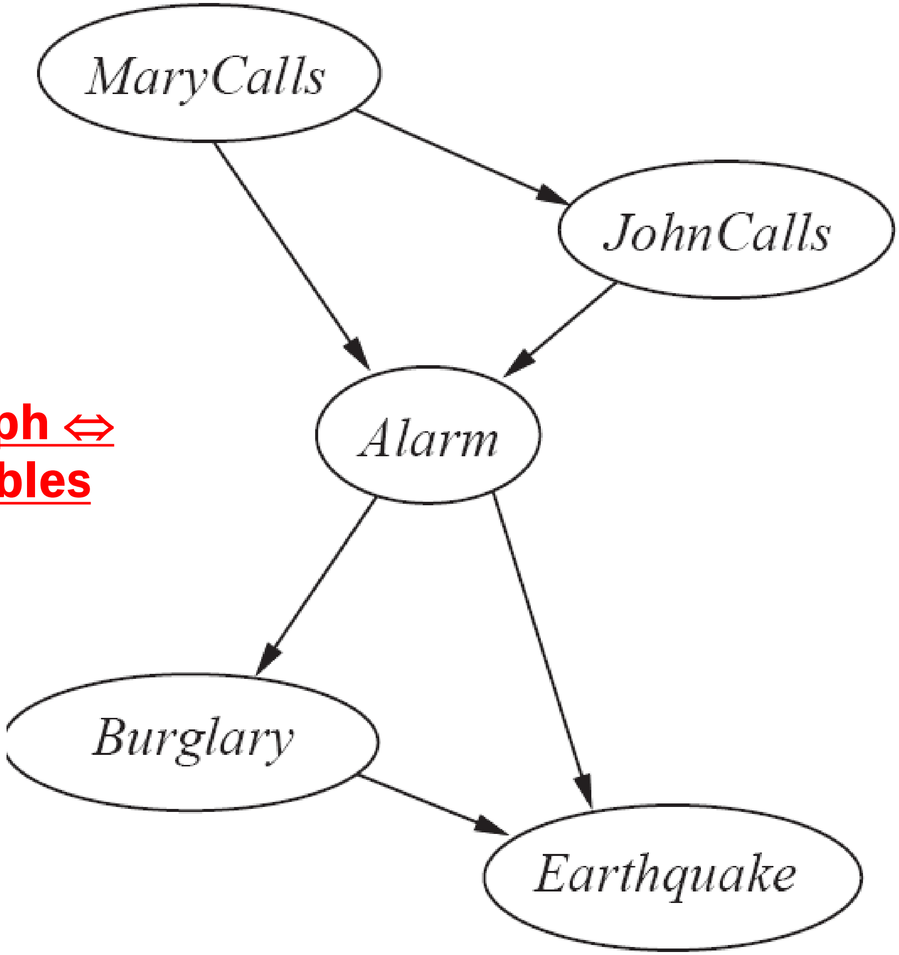
P(E)
.002

B	E	P(A)
t	t	.95
t	f	.94
f	t	.29
f	f	.001

A	P(J)
t	.90
f	.05

A	P(M)
t	.70
f	.01

The Bayesian Network From a Different Variable Ordering



Parents in the graph ⇔ conditioning variables (RHS)

P(J, M, A, E, B) = P(E | A, B) P(B | A) P(A | M, J) P(J | M) P(M)

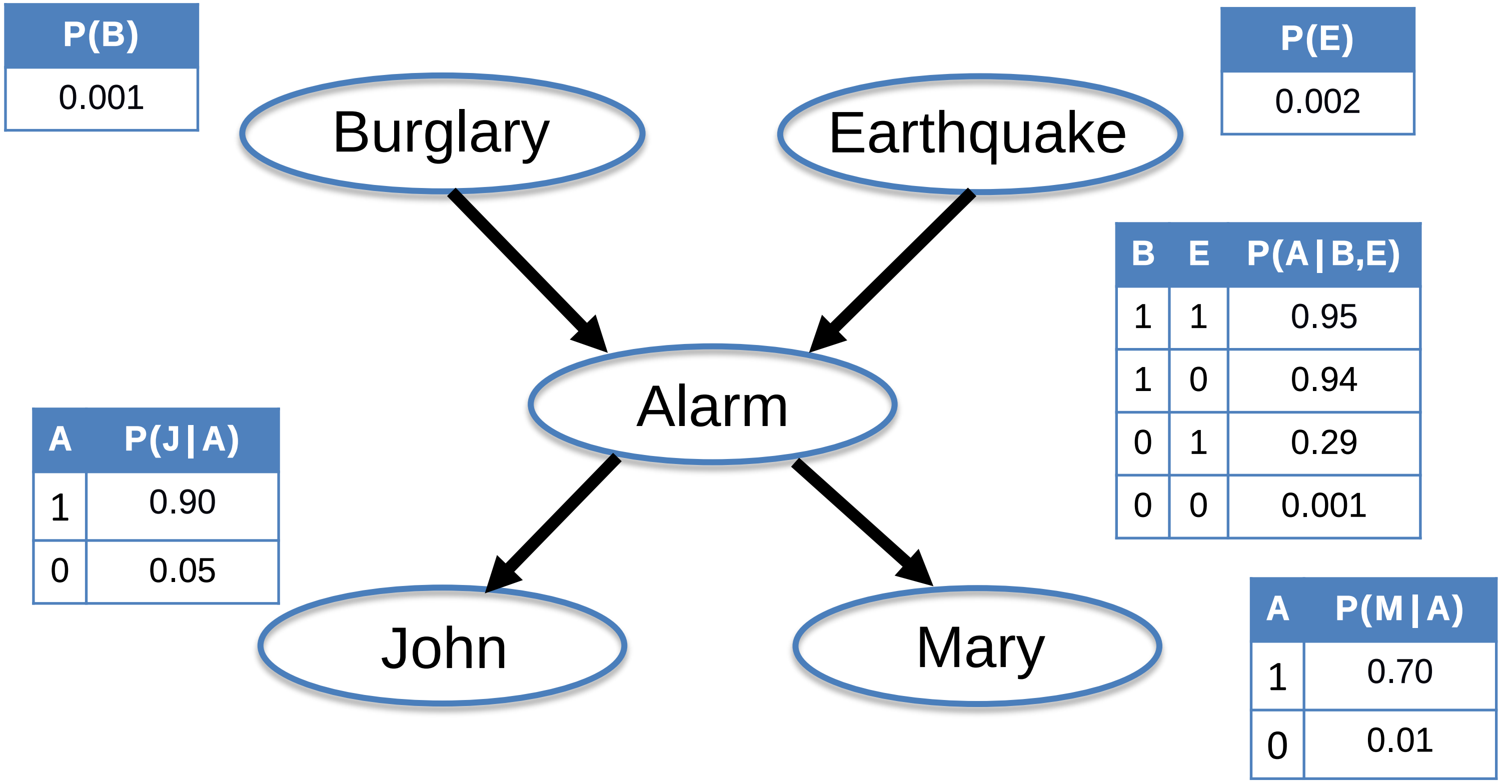
Generally, order variables so that resulting graph reflects assumed causal relationships.

Example of Answering a Simple Query

- What is $P(\neg j, m, a, \neg e, b) = P(J = \text{false} \wedge M=\text{true} \wedge A=\text{true} \wedge E=\text{false} \wedge B=\text{true})$

$P(J, M, A, E, B) \approx P(J \mid A)\, P(M \mid A)\, P(A \mid E, B)\, P(E)\, P(B)$; by conditional independence

$$P(\neg j, m, a, \neg e, b) \approx P(\neg j \mid a)\, P(m \mid a)\, P(a \mid \neg e, b)\, P(\neg e)\, P(b)$$
$$= 0.10 \times 0.70 \times 0.94 \times 0.998 \times 0.001 \approx .0000657$$



P(B)
0.001

P(E)
0.002

B	E	P(A|B,E)
1	1	0.95
1	0	0.94
0	1	0.29
0	0	0.001

A	P(J|A)
1	0.90
0	0.05

A	P(M|A)
1	0.70
0	0.01

Inference in Bayesian Networks

- **X** *= { X1, X2, …, Xk }* = **query variables** of interest
- **E** *= { E1, …, El }* = **evidence variables** that are observed
- **Y** = *{ Y1, …, Ym }* = **hidden variables** (nonevidence, nonquery)

- **What is the posterior distribution of X, given E?**
  - $\mathbf{P}(\,\mathbf{X} \mid \mathbf{e}\,) = \alpha \, \Sigma_{\mathbf{y}} \; \mathbf{P}(\,\mathbf{X}, \mathbf{y}, \mathbf{e}\,)$

    Normalizing constant $\alpha = \Sigma_{\mathbf{x}} \; \Sigma_{\mathbf{y}} \; \mathbf{P}(\,\mathbf{X}, \mathbf{y}, \mathbf{e}\,)$

- **What is the most likely assignment of values to X, given E?**
  - $\mathbf{argmax}_{\mathbf{x}} \, P(\,\mathbf{x} \mid \mathbf{e}\,) = \mathbf{argmax}_{\mathbf{x}} \; \Sigma_{\mathbf{y}} \; P(\,\mathbf{x}, \mathbf{y}, \mathbf{e}\,)$

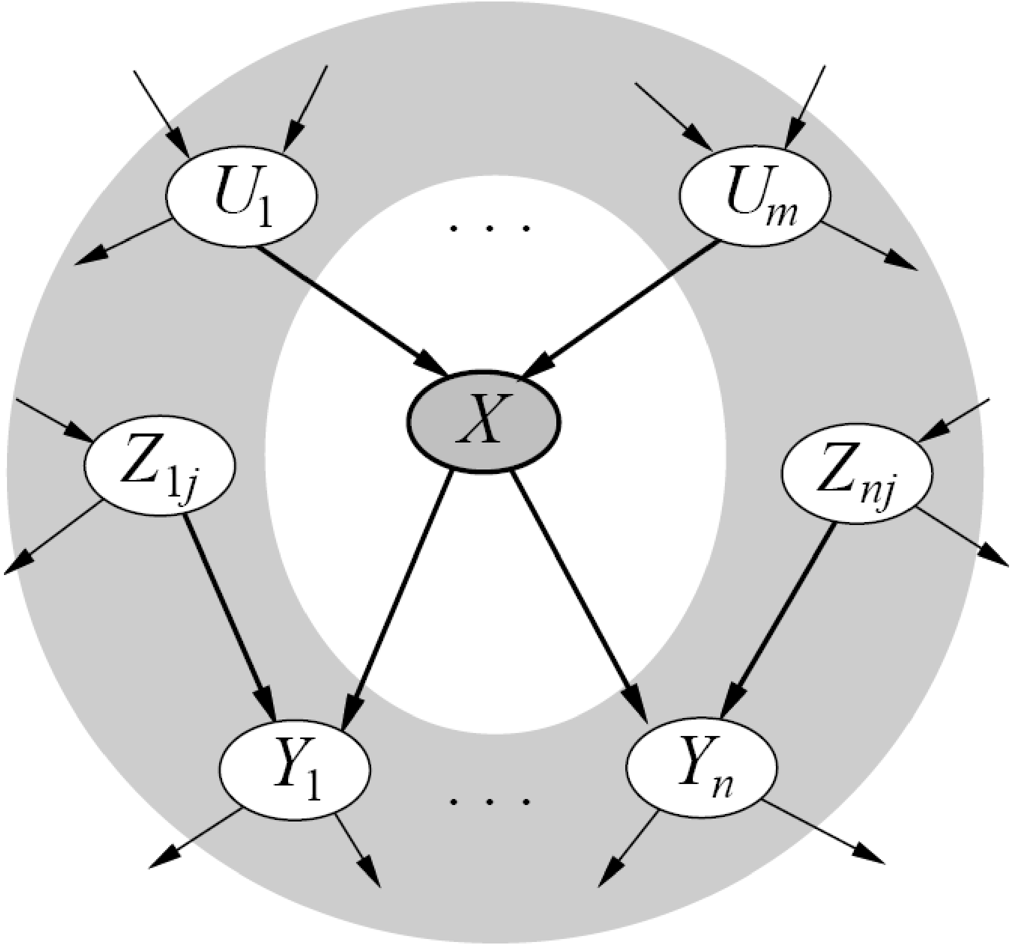
Given a graph, can we "read off" conditional independencies?

The "Markov Blanket" of X (the gray area in the figure)

X is conditionally independent of everything else, GIVEN the values of:

* X's parents
* X's children
* X's children's parents

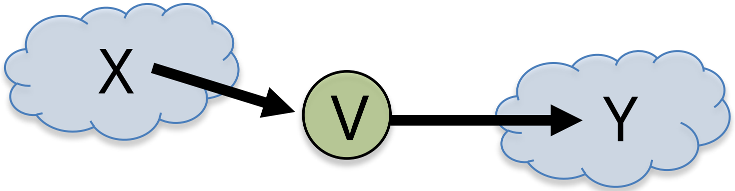
X is conditionally independent of its non-descendants, GIVEN the values of its parents.



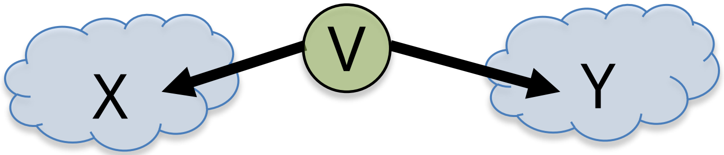
D-Separation

- Prove sets X,Y independent given Z?
- Check all *undirected* paths from X to Y
- A path is “inactive” if it passes through:

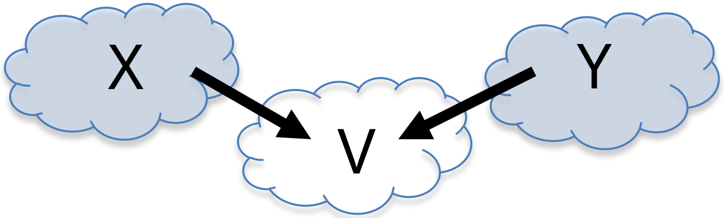
(1) A “chain” with an observed variable



(2) A “split” with an observed variable



(3) A “vee” with **only unobserved** variables below it



- If all paths are inactive, conditionally independent!

Summary

- Bayesian networks represent a joint distribution using a graph

- The graph encodes a set of conditional independence assumptions

- Answering queries (or inference or reasoning) in a Bayesian network amounts to computation of appropriate conditional probabilities

- Probabilistic inference is intractable in the general case
  - Can be done in linear time for certain classes of Bayesian networks (polytrees: at most one directed path between any two nodes)
  - Usually faster and easier than manipulating the full joint distribution