Mid-term Review
Chapters 2-5, 13, 14

- Review Agents (2.1-2.3)
- Review State Space Search
  - Problem Formulation (3.1, 3.3)
  - Blind (Uninformed) Search (3.4)
  - Heuristic Search (3.5)
  - Local Search (4.1, 4.2)
- Review Adversarial (Game) Search (5.1-5.4)
- Review Probability & Bayesian Networks (13, 14.1-14.5)

- Please review your quizzes and old CS-171 tests
  - At least one question from a prior quiz or old CS-171 test will appear on the mid-term (and all other tests)

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

Agents

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

  Human agent:

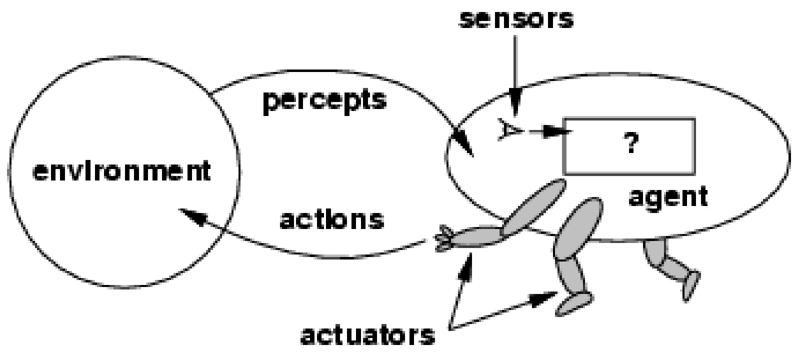
  eyes, ears, and other organs for sensors;

  hands, legs, mouth, and other body parts for actuators

- Robotic agent:

  cameras and infrared range finders for sensors; various motors for actuators

Agents and environments



- Percept: agent's perceptual inputs at an instant
- The agent function maps from percept sequences to actions: [f: $\mathcal{P}^\star \rightarrow \mathcal{A}$]
- 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)

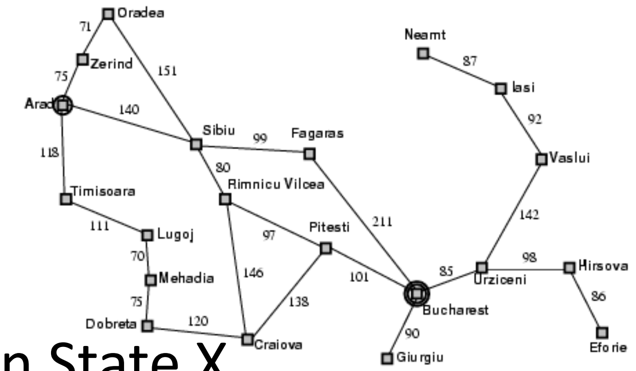
- **Discrete (vs. continuous):** A limited number of distinct, clearly defined percepts and actions.
  - How do we **represent** or **abstract** or **model** the world?

- **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 Chapters 3-4

- 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
- Local Search (4.1, 4.2)
  - Hill-climbing, Simulated Annealing, Genetic Algorithms
  - Gradient descent

Problem Formulation



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

initial state, e.g., "at Arad“

actions

– Actions(X) = set of actions available in State X

transition model

– Result(S,A) = state resulting from doing action A in state S

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

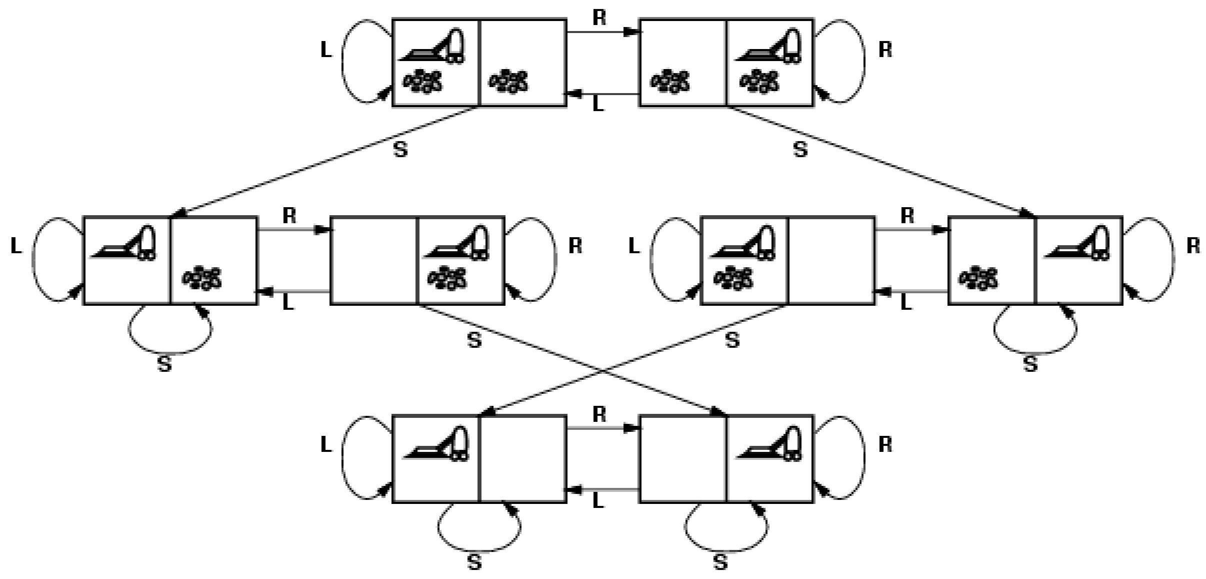
path cost (additive, i.e., the sum of the step costs)

– *c(x,a,y)* = **step cost** of action *a* in state *x* to reach state *y*

  – assumed to be ≥ 0

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

Vacuum world state space graph

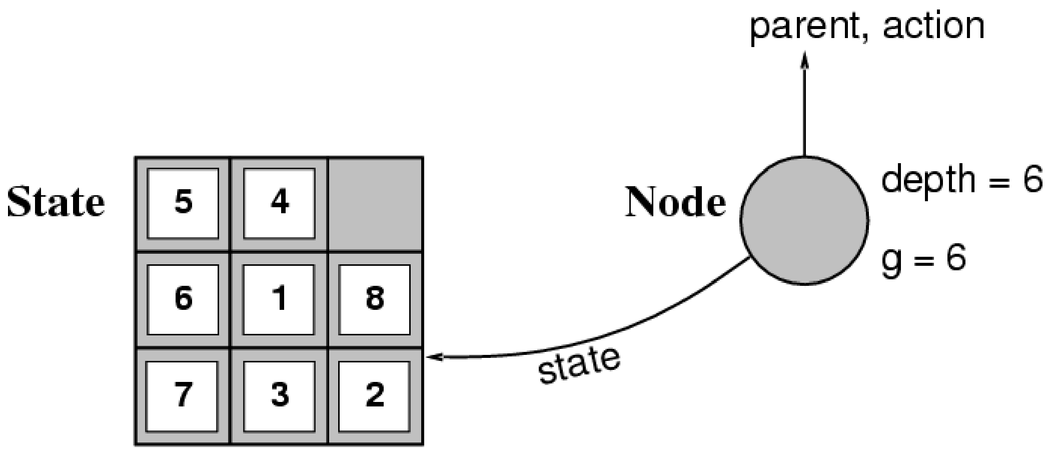


- states? discrete: dirt and robot locations
- initial state? any
- actions? *Left*, *Right*, *Suck*
- transition model? as shown on graph
- goal test? no dirt at all locations
- path cost? 1 per action

Implementation: states vs. nodes

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

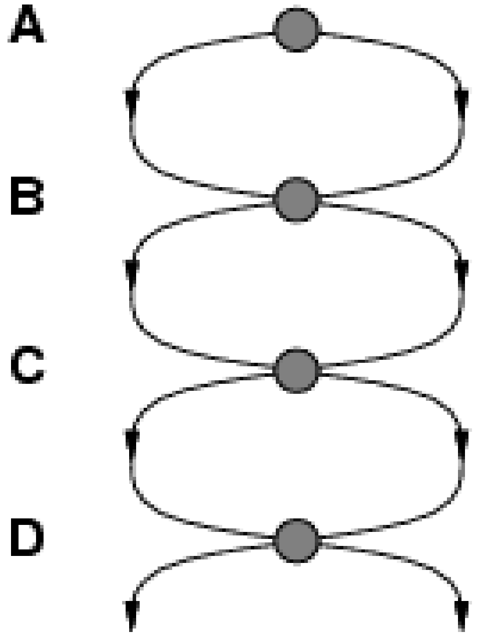
- A node is a data structure constituting part of a search tree
- A node contains info such as:
  - state, parent node, action, path cost *g(x)*, depth, etc.

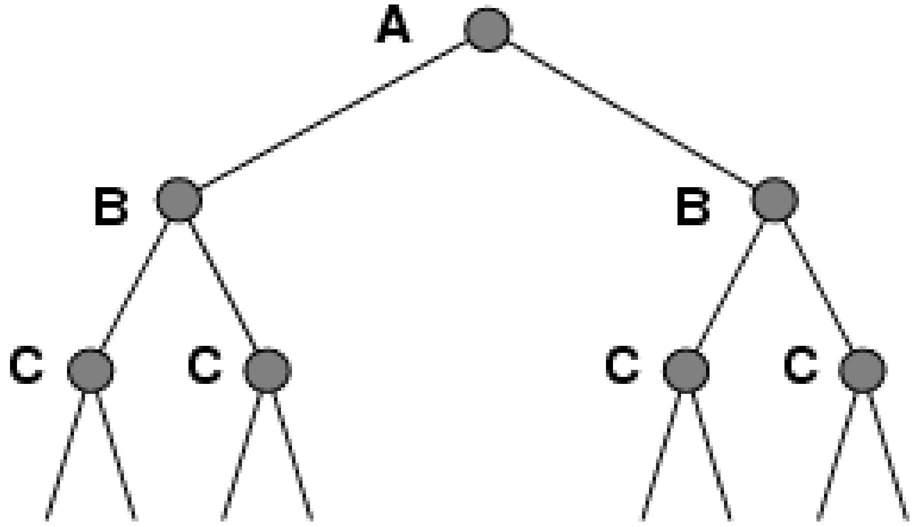


- The `Expand` function creates new nodes, filling in the various fields using the `Actions(S)` and `Result(S,A)` functions associated with the problem.

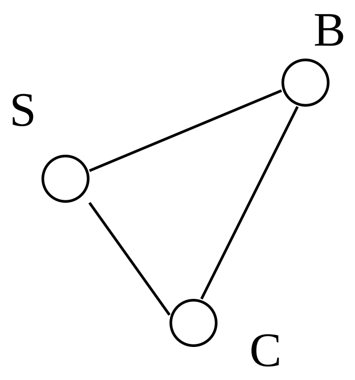
Tree search vs. Graph search Review Fig. 3.7, p. 77

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

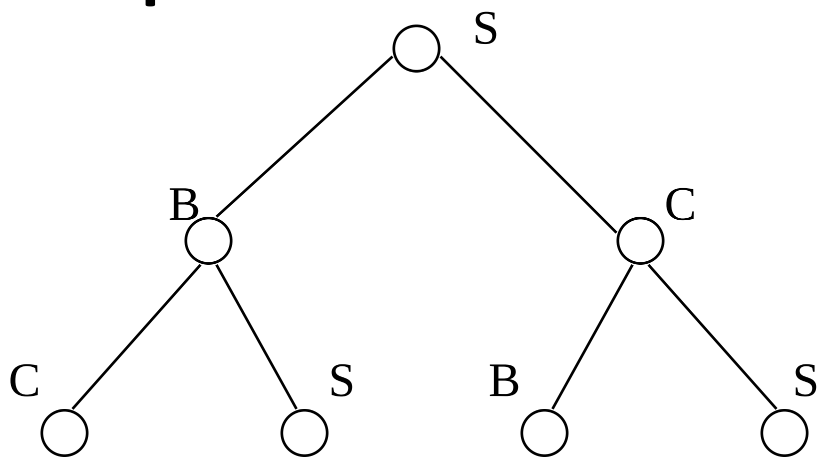




Solutions to Repeated States



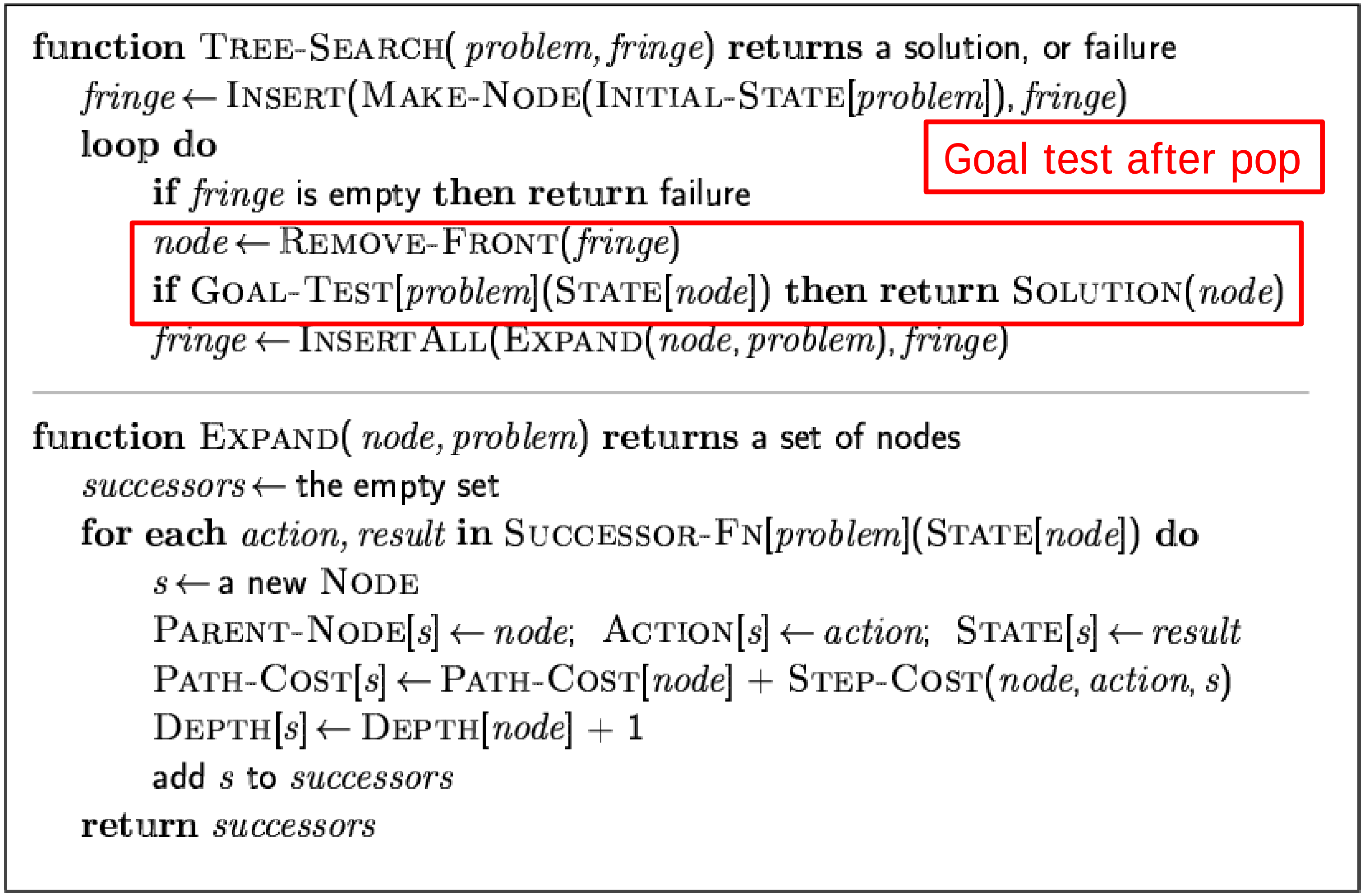
State Space



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 <u>hash table</u>

General tree search
Do not remember visited nodes

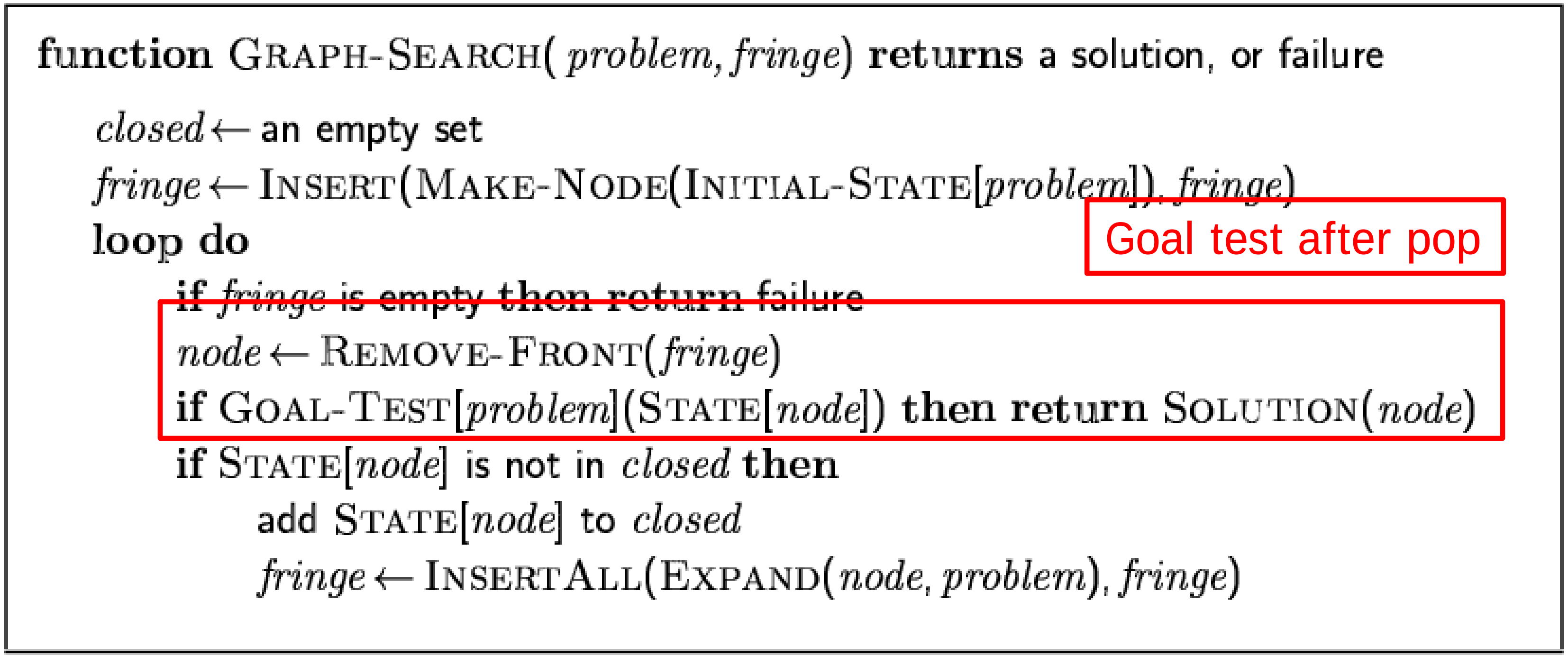


Goal test after pop

```
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
```

General graph search
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

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.

Uniform cost graph search: sort by *g*

A* is identical but uses *f=g+h*

Greedy best-first is identical but uses *h*

```
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

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.

Depth-limited search & IDS

```
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
```

Iterate over successors,
call recursively on each.
Goal test at head of call.

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

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

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.

- **<u>Review "Example hand-simulated search"</u>**
  - Slides 29-38, 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 <u>*state*</u> 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

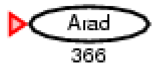
Greedy best-first search

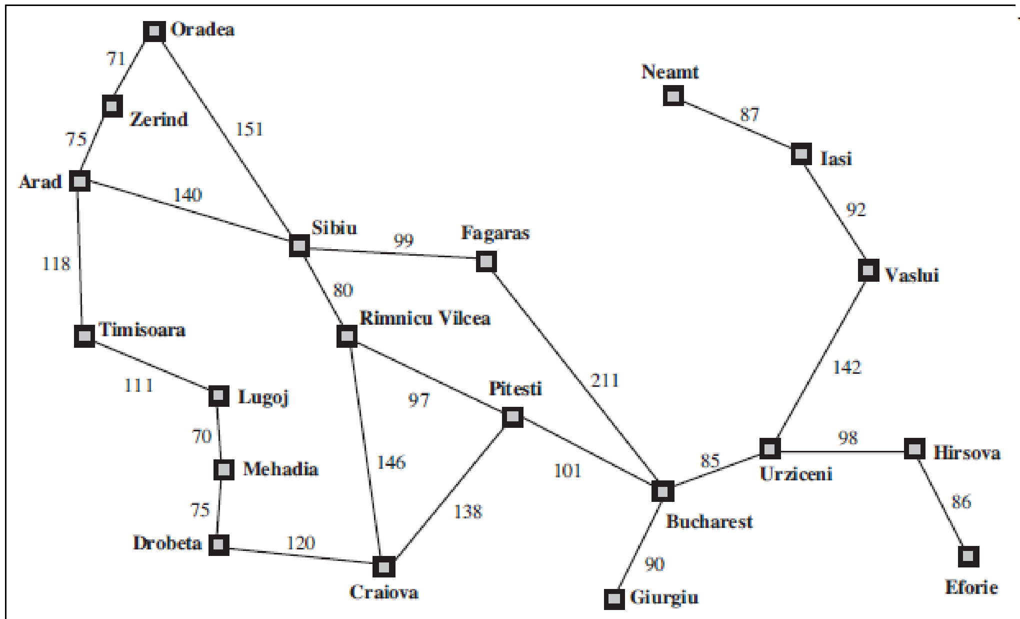
- *h(n)* = estimate of cost from *n* to *goal*
  - e.g., *h(n)* = straight-line distance from *n* to Bucharest

- 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

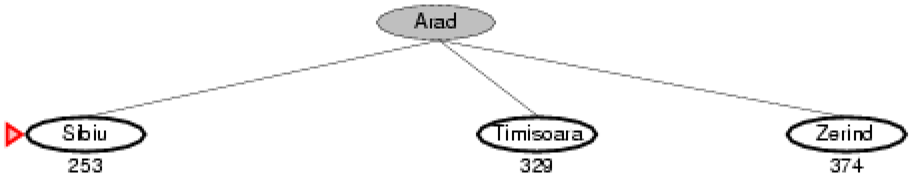


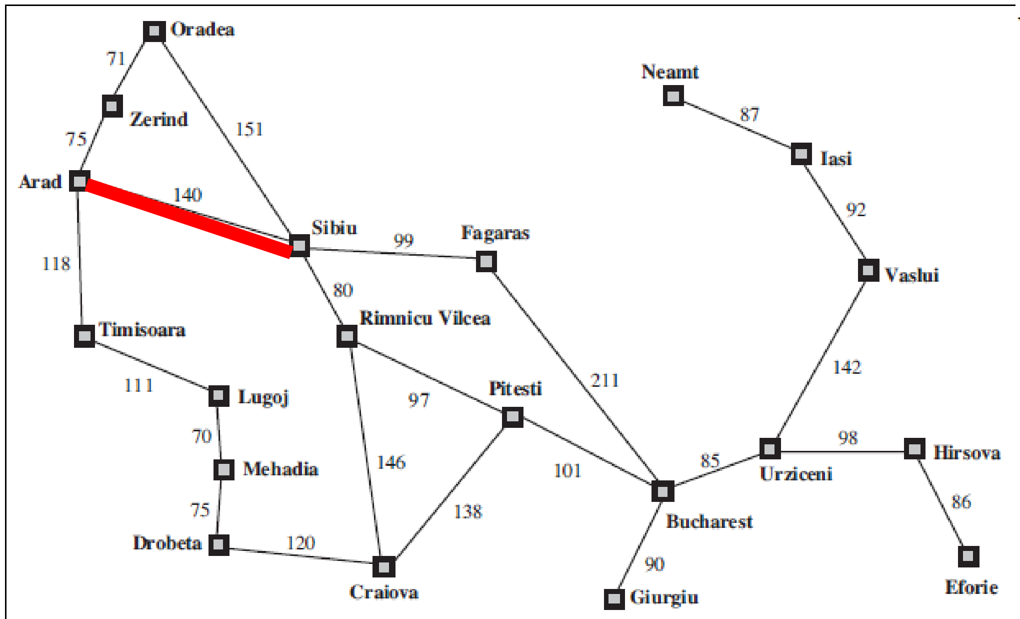


Values of h_{SLD}—straight-line distances to Bucharest.

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

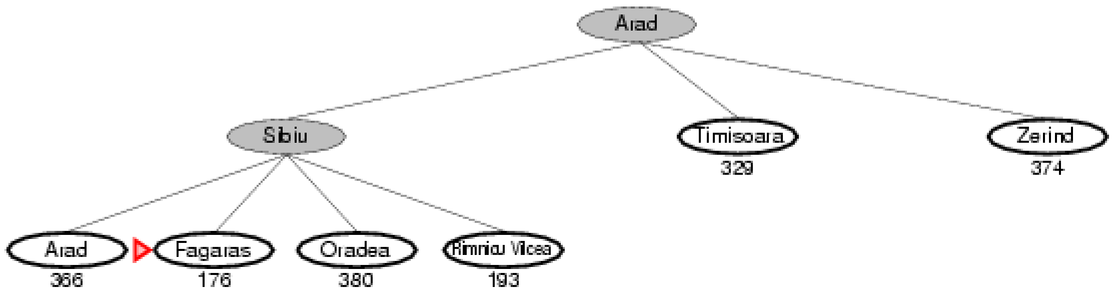


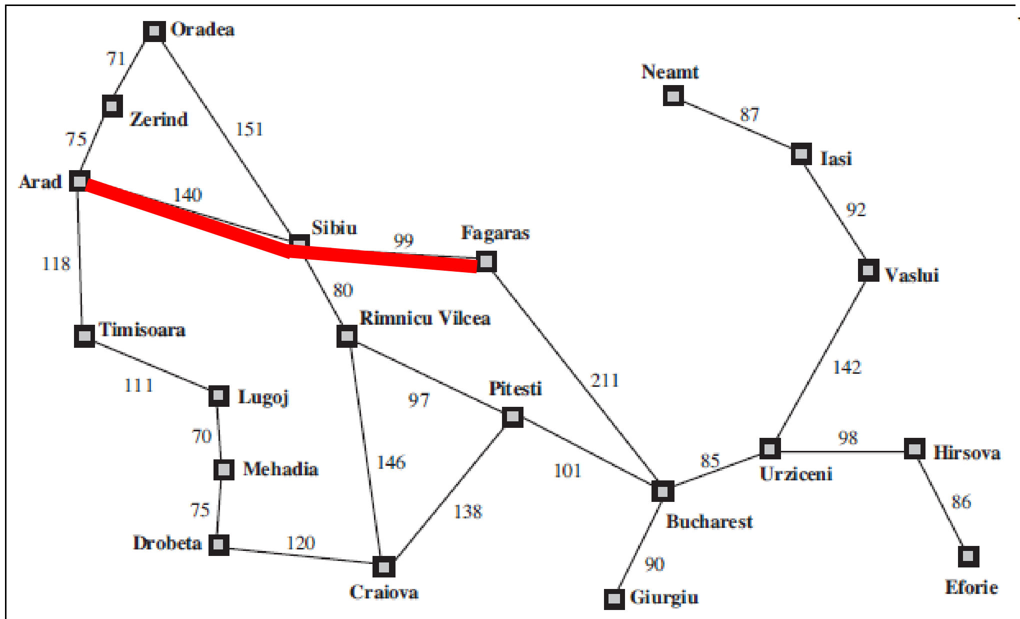


Values of h_{SLD}—straight-line distances to Bucharest.

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

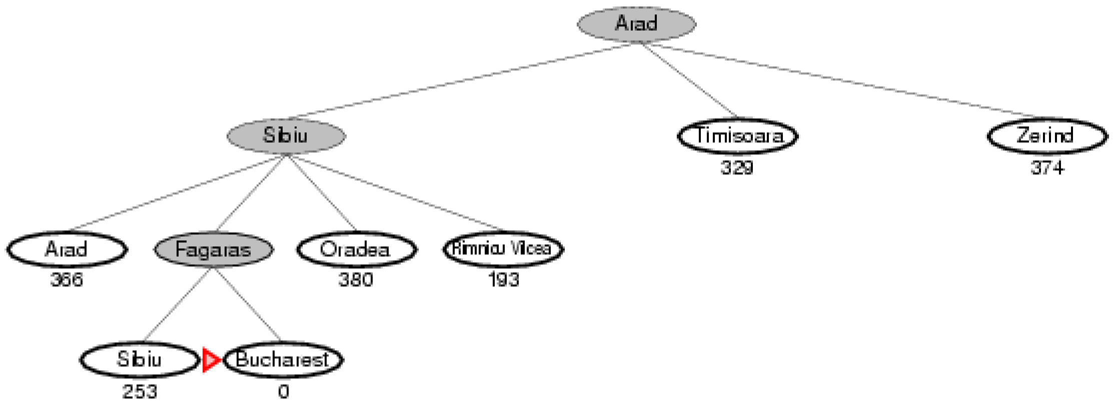


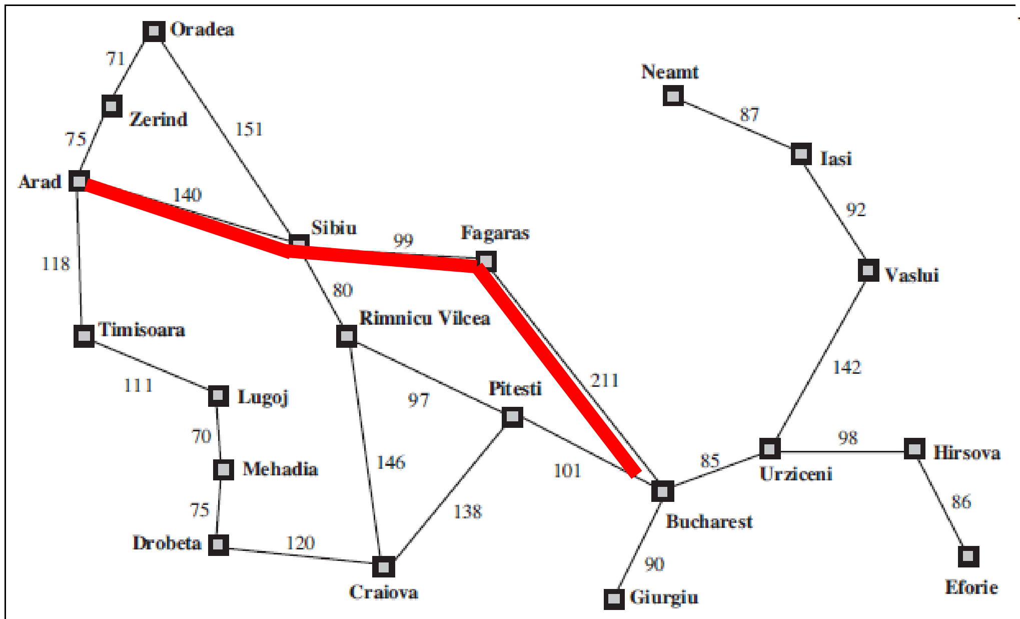


Values of h_{SLD}—straight-line distances to Bucharest.

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

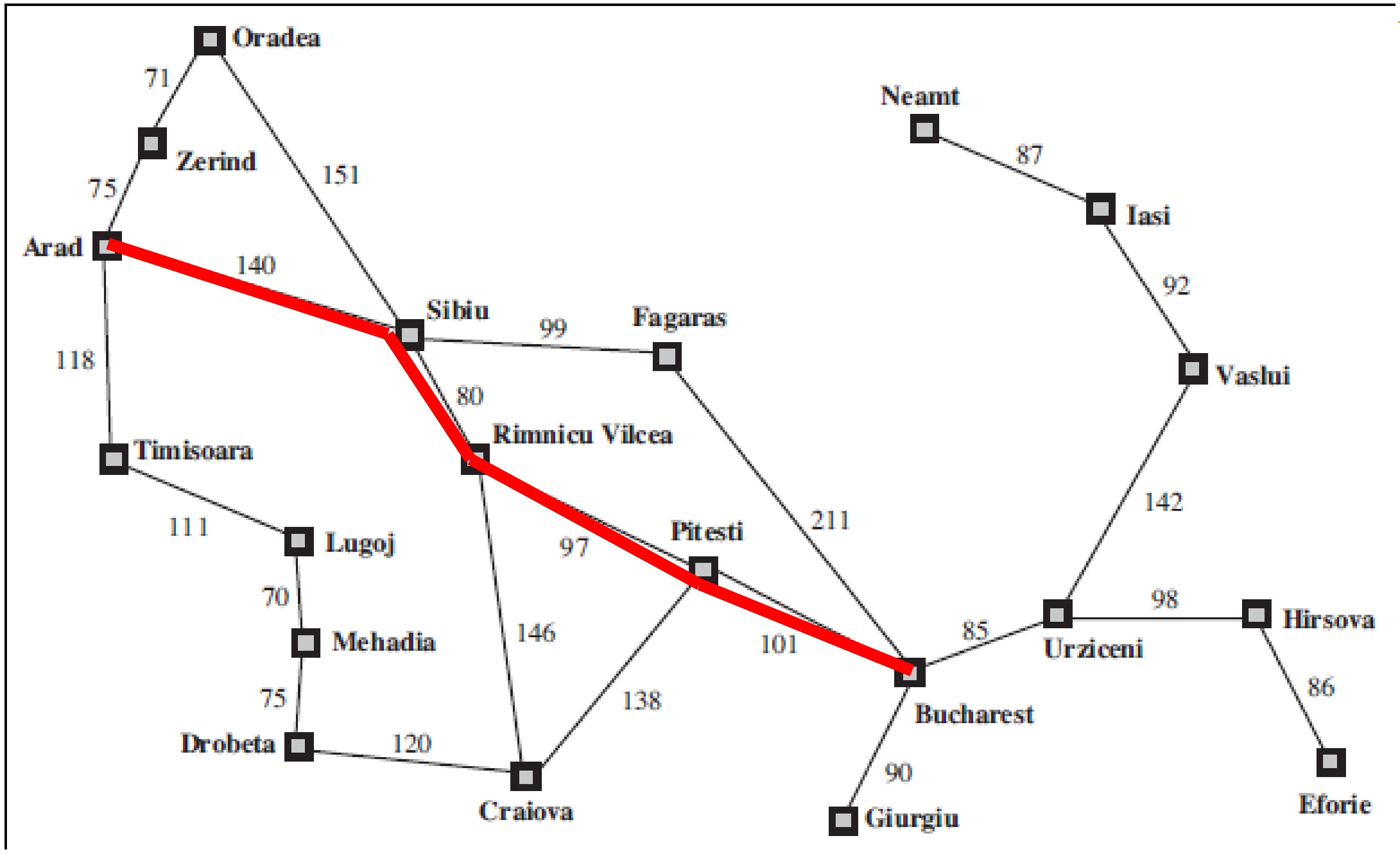




Values of h_{SLD}—straight-line distances to Bucharest.

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.

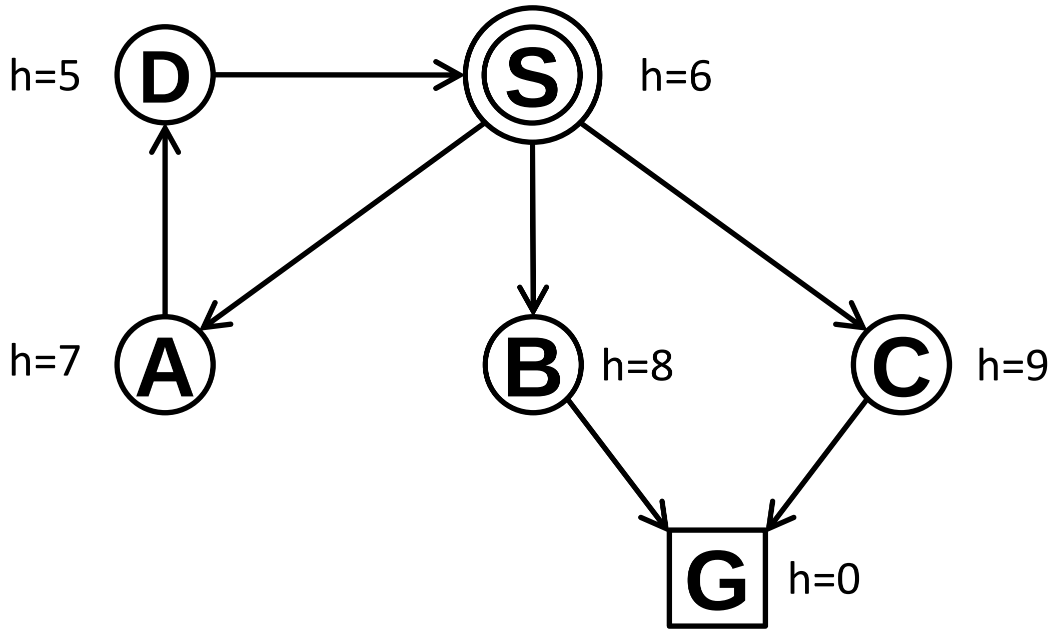
Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

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? $O(1)$ tree search, $O(b^m)$ graph search
  - 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!

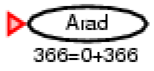
A* search

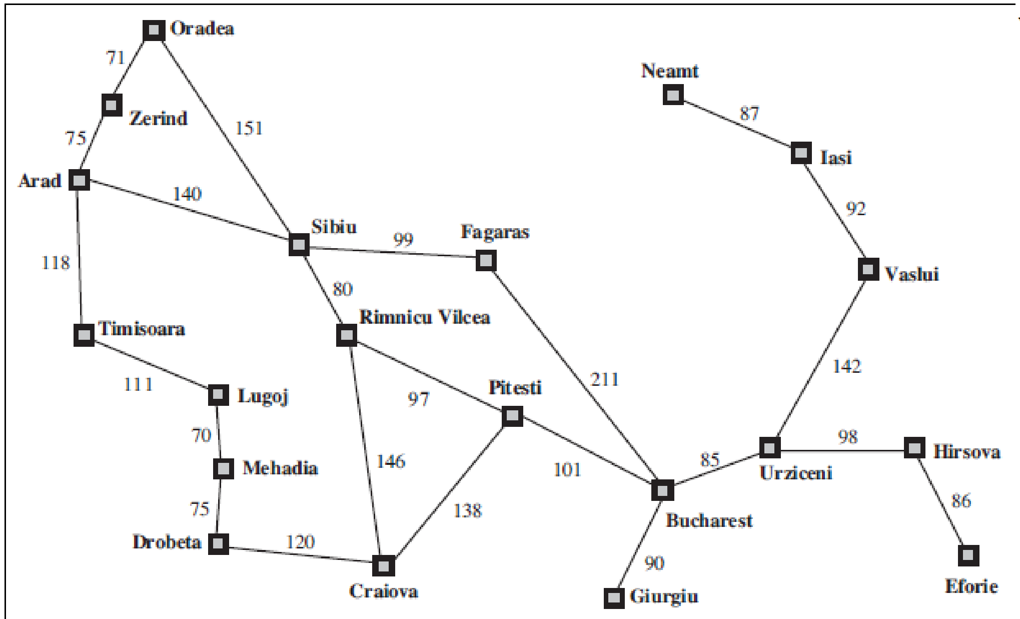
- Idea: avoid paths that are already expensive
  - Generally the preferred simple heuristic search
  - Optimal if heuristic is:

    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





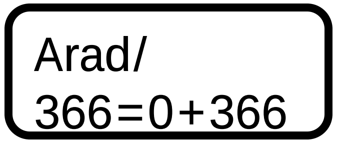
Values of h_{SLD}—straight-line distances to Bucharest.

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: $\boxed{\text{Arad/366=0+366}}$

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



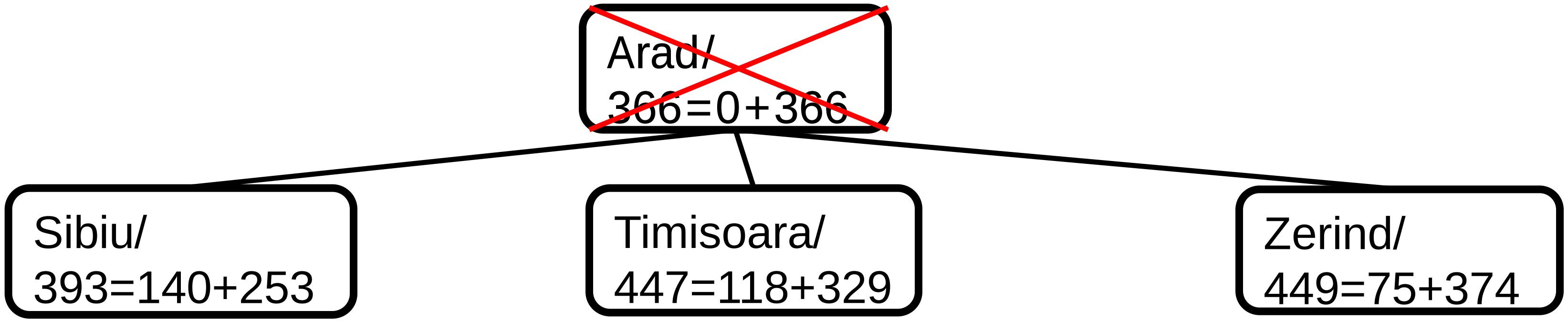
A^* tree search example: Simulated queue. City/f=g+h

Arad/
366=0+366

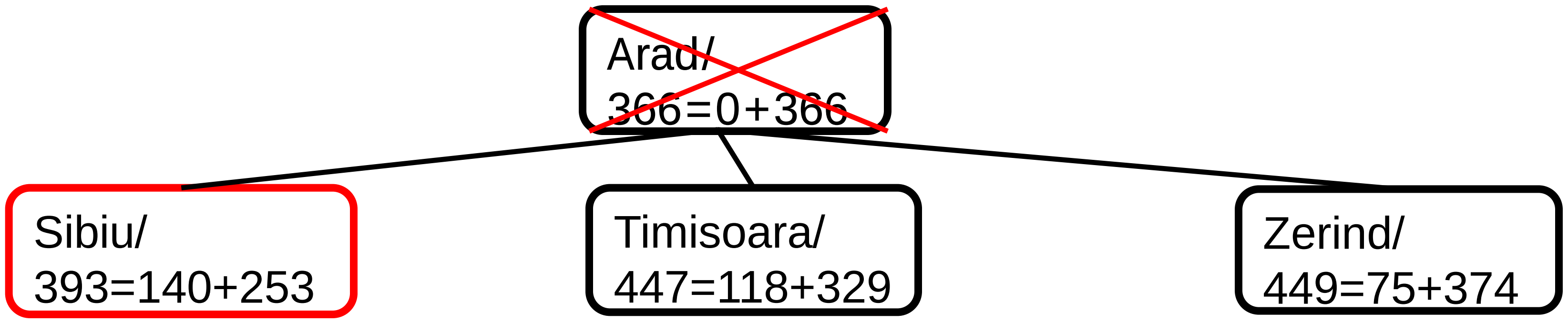
A^* tree search example: Simulated queue. City/f=g+h

- 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~~, $\boxed{\text{Sibiu/393=140+253}}$, Timisoara/447=118+329, Zerind/449=75+374

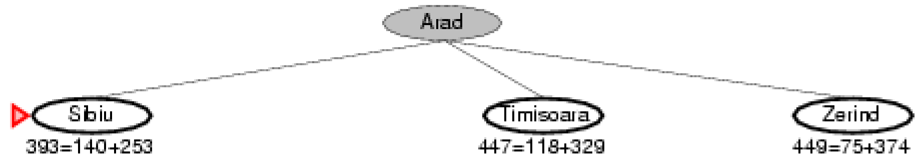
A* tree search example: Simulated queue. City/f=g+h

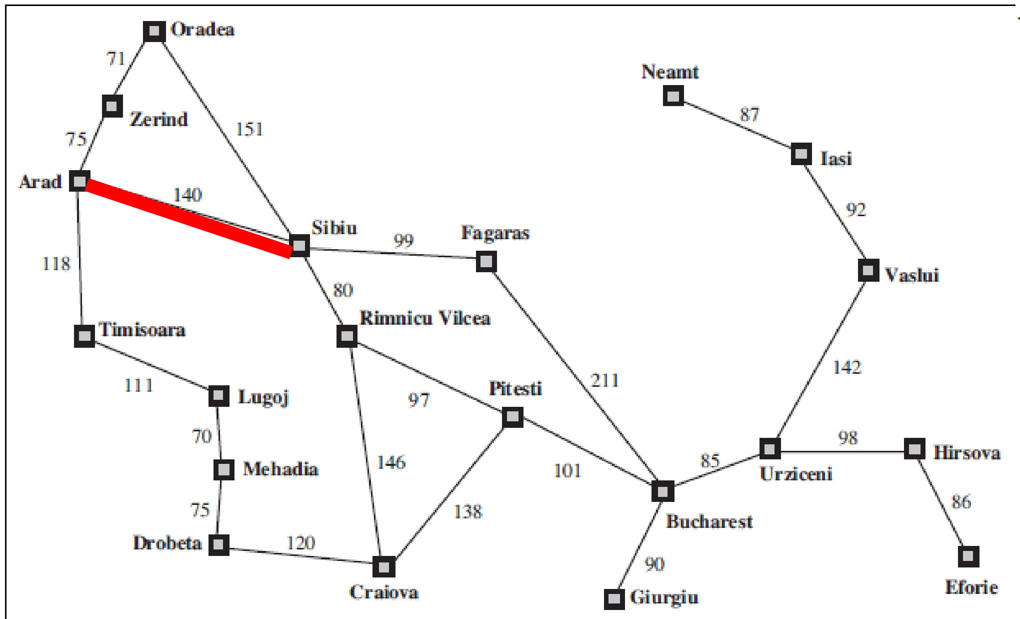


A* tree search example: Simulated queue. City/f=g+h



A* tree search example





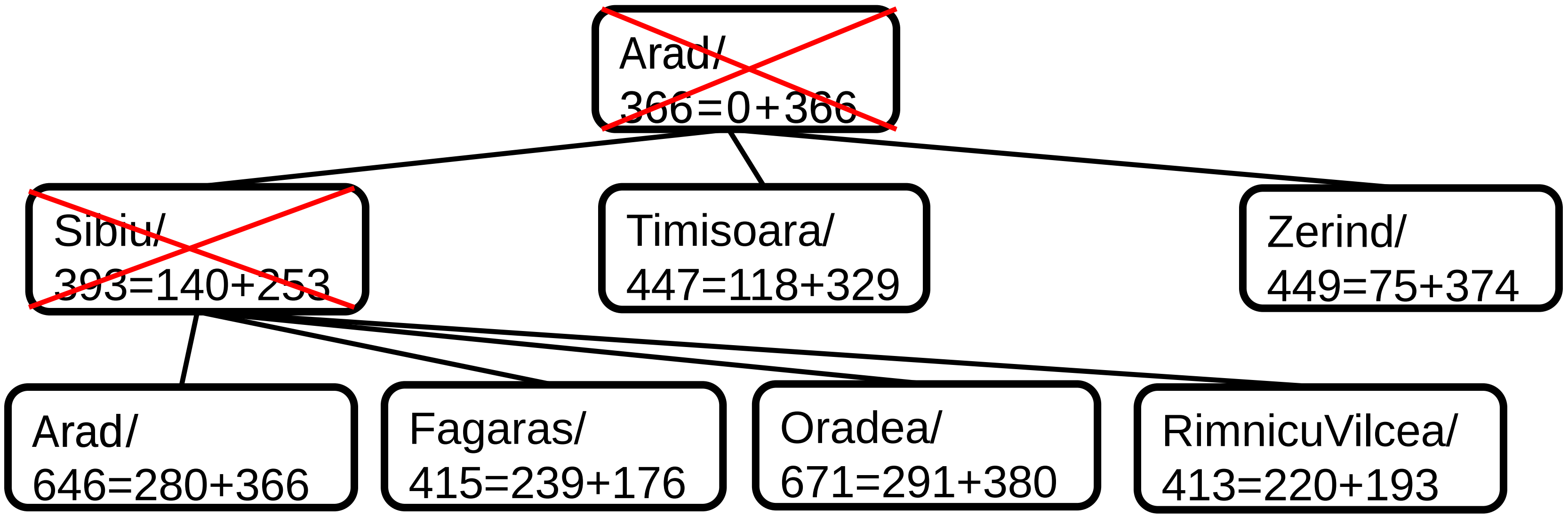
Values of h_{SLD}—straight-line distances to Bucharest.

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

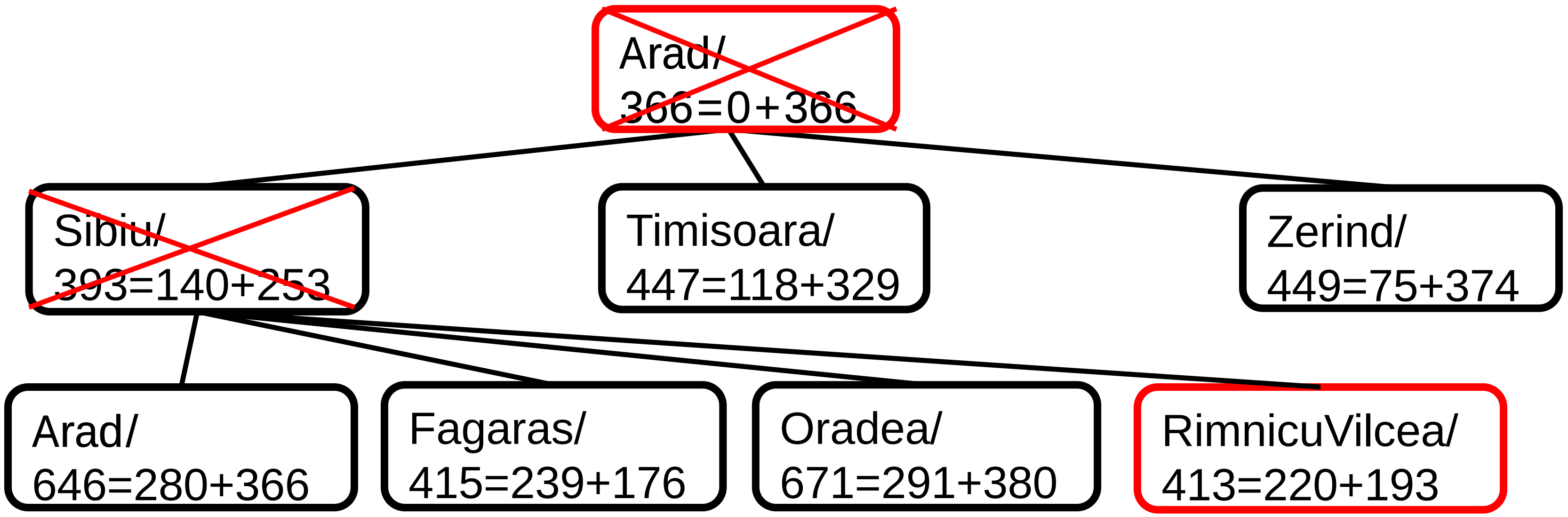
A* tree search example: Simulated queue. City/f=g+h

- Next: 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, $\boxed{\text{RimnicuVilcea/413=220+193}}$

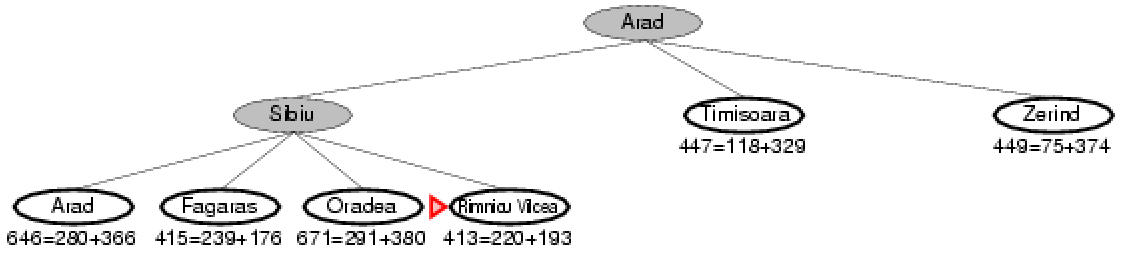
A* tree search example: Simulated queue. City/f=g+h

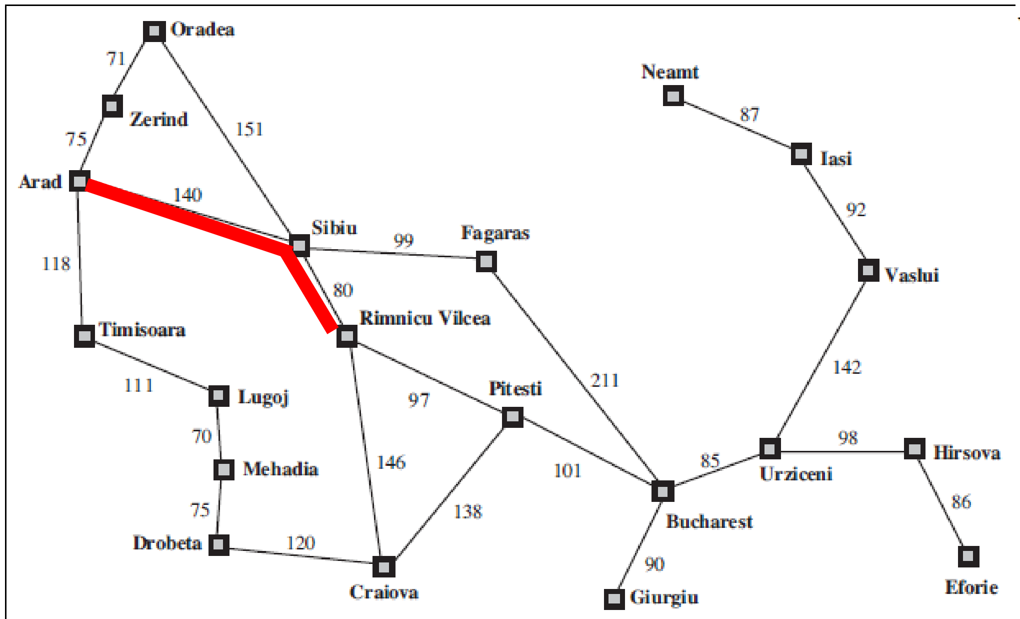


A* tree search example: Simulated queue. City/f=g+h



A* tree search example





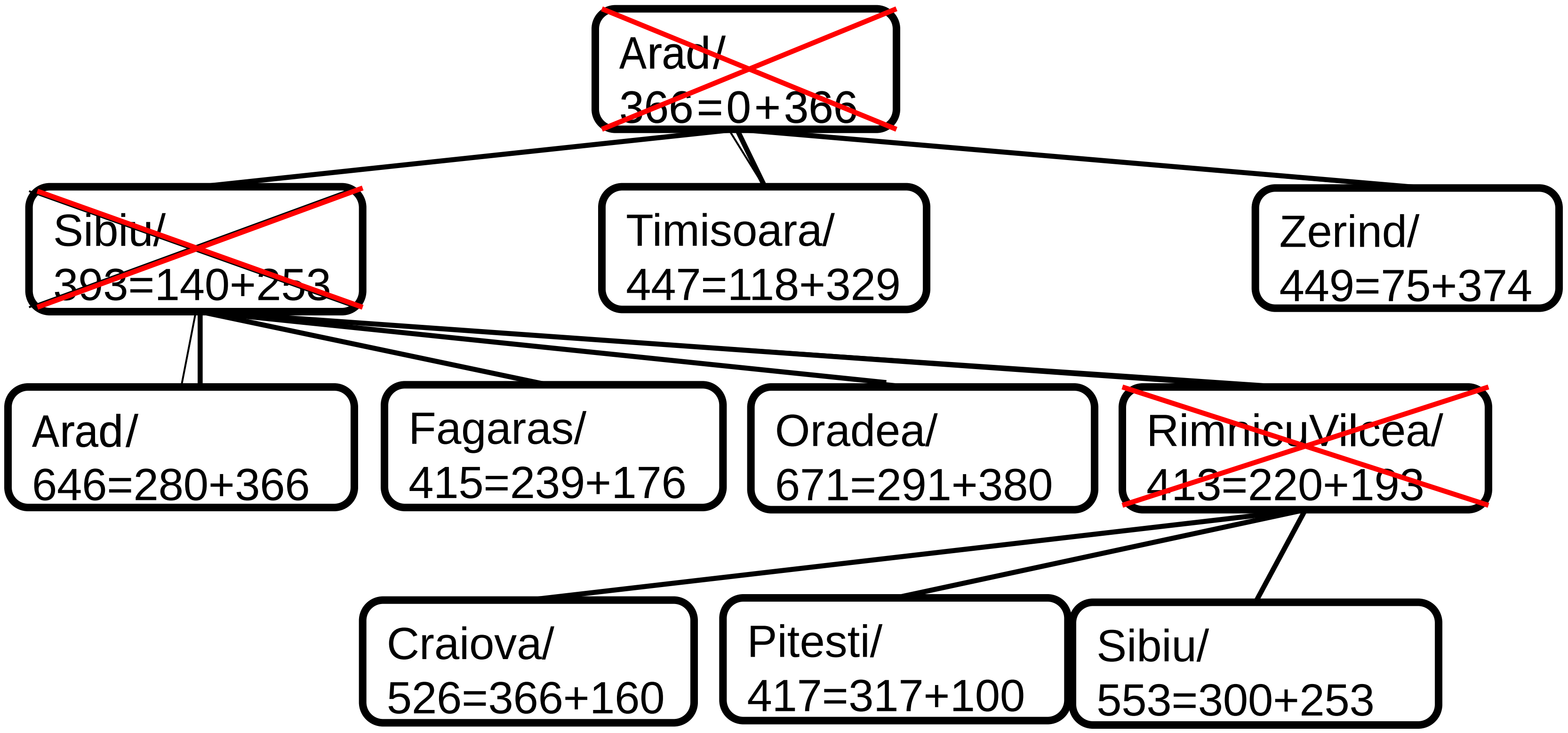
Values of h_{SLD}—straight-line distances to Bucharest.

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

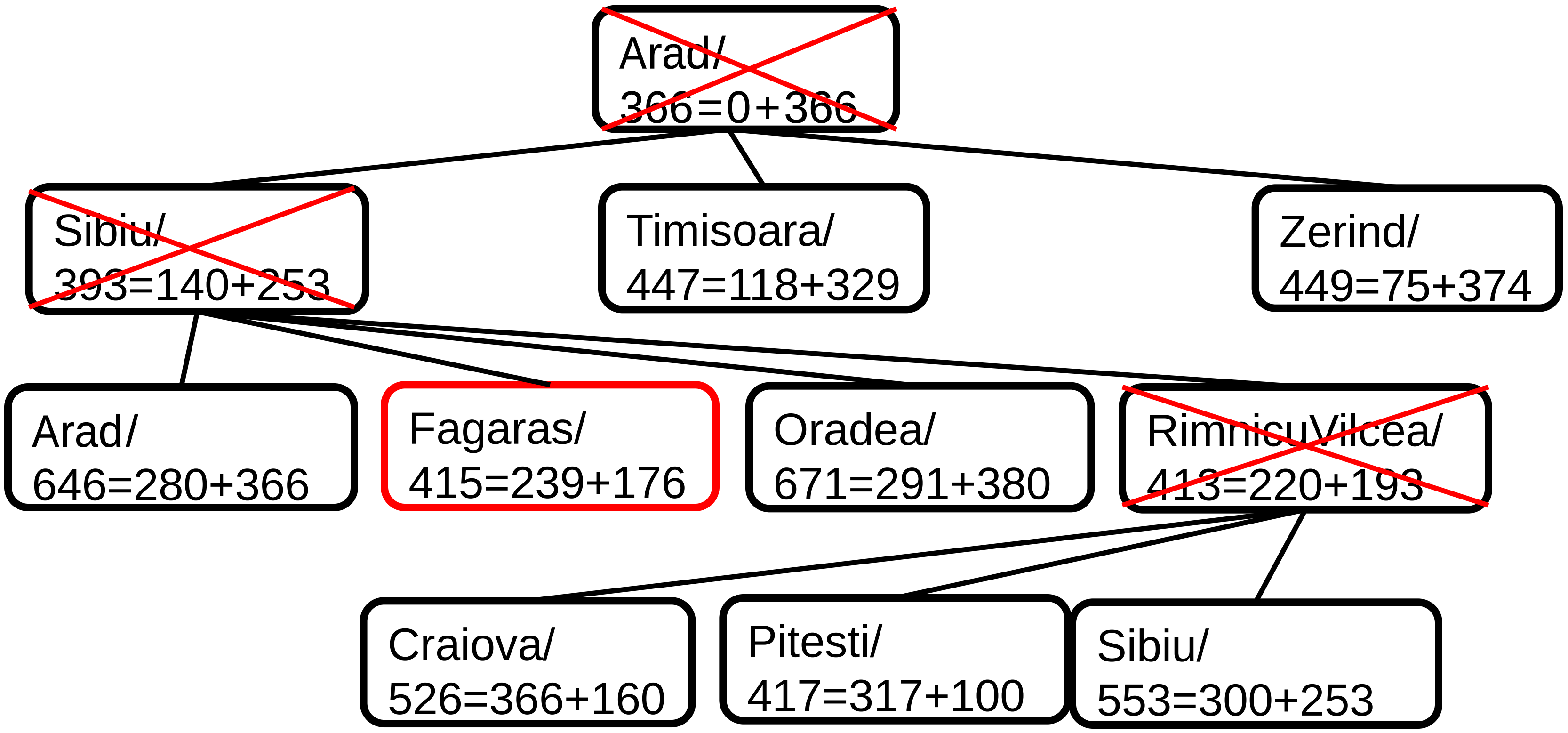
A* tree search example: Simulated queue. City/f=g+h

- 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, $\boxed{\text{Fagaras/415=239+176,}}$ 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

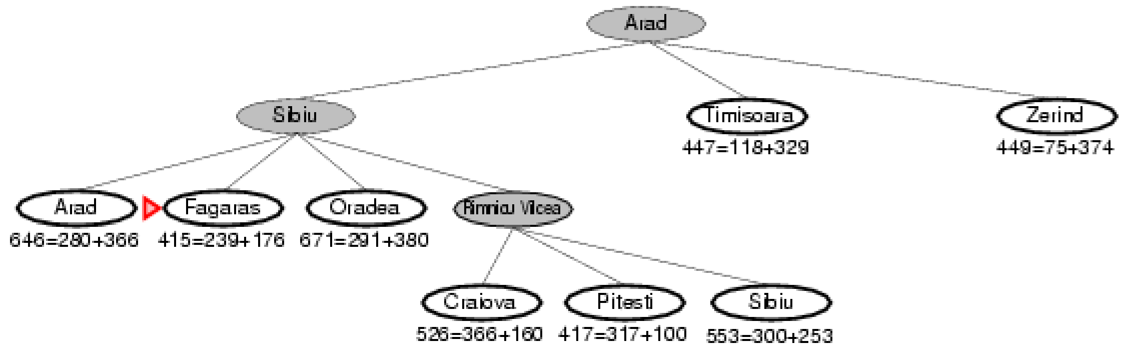


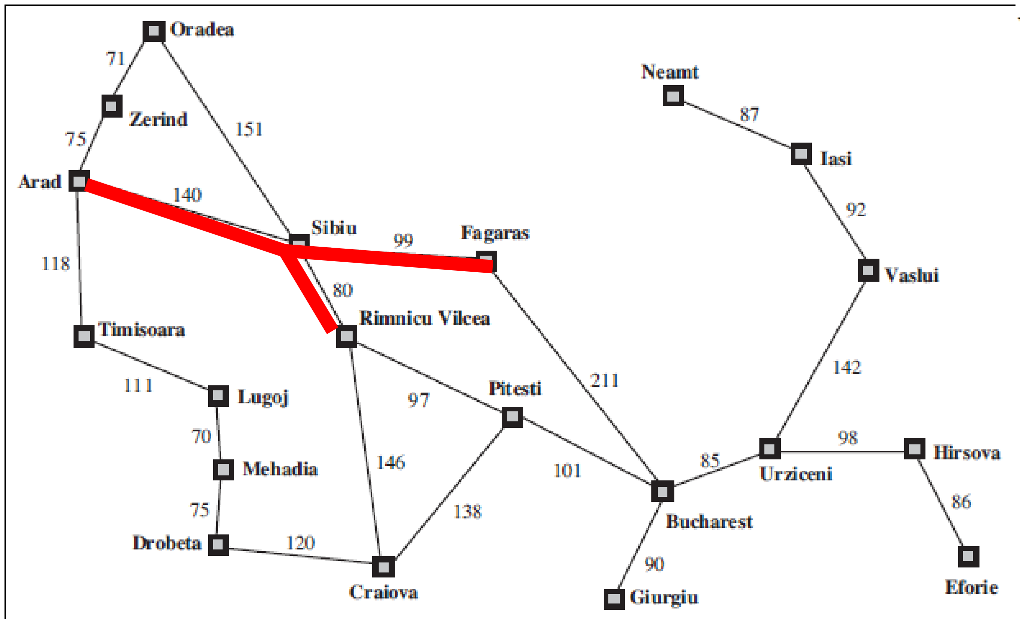
A^* search example: Simulated queue. City/f=g+h



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.

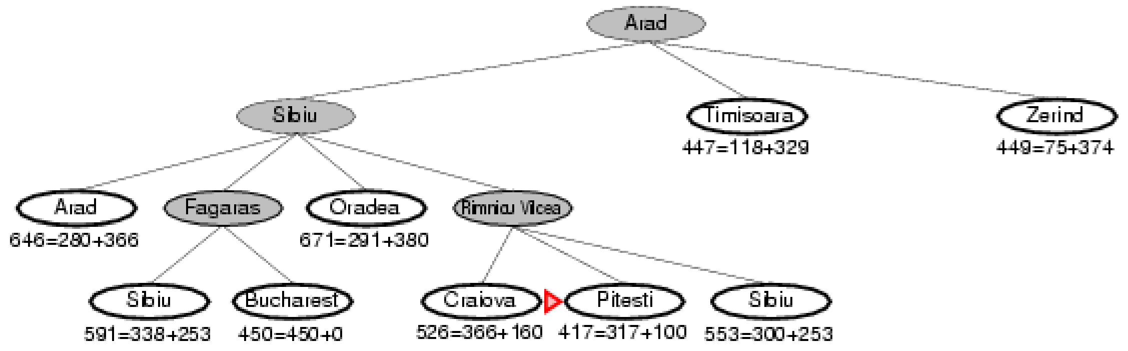
Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

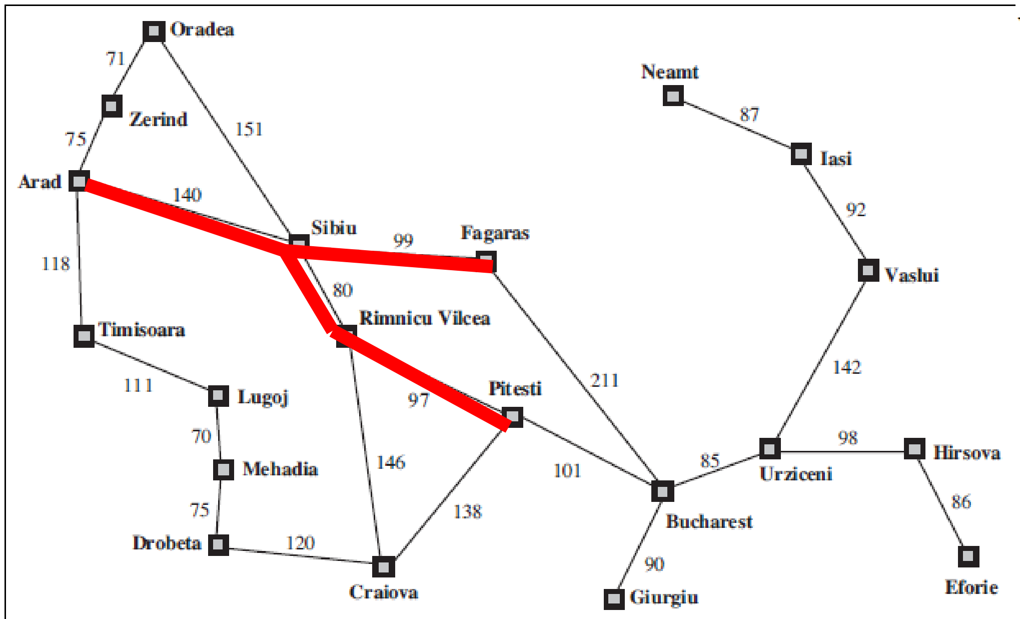
A* tree search example: Simulated queue. City/f=g+h

- 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, $\boxed{\text{Pitesti/417=317+100}}$, 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.





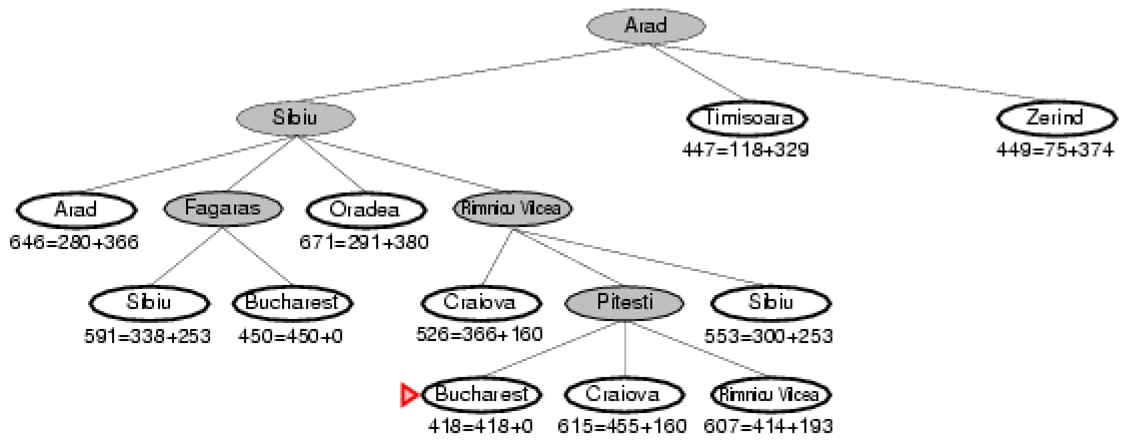
Values of h_{SLD}—straight-line distances to Bucharest.

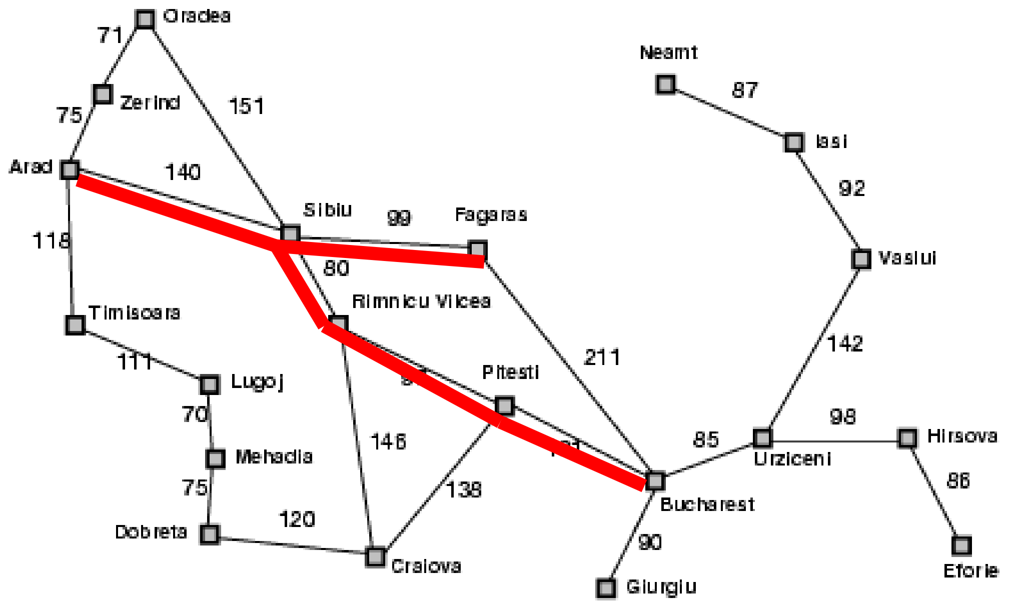
Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

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, $\boxed{\text{Bucharest/418=418+0}}$, Craiova/615=455+160, RimnicuVilcea/607=414+193

A* tree search example





Straight–line distance to Bucharest

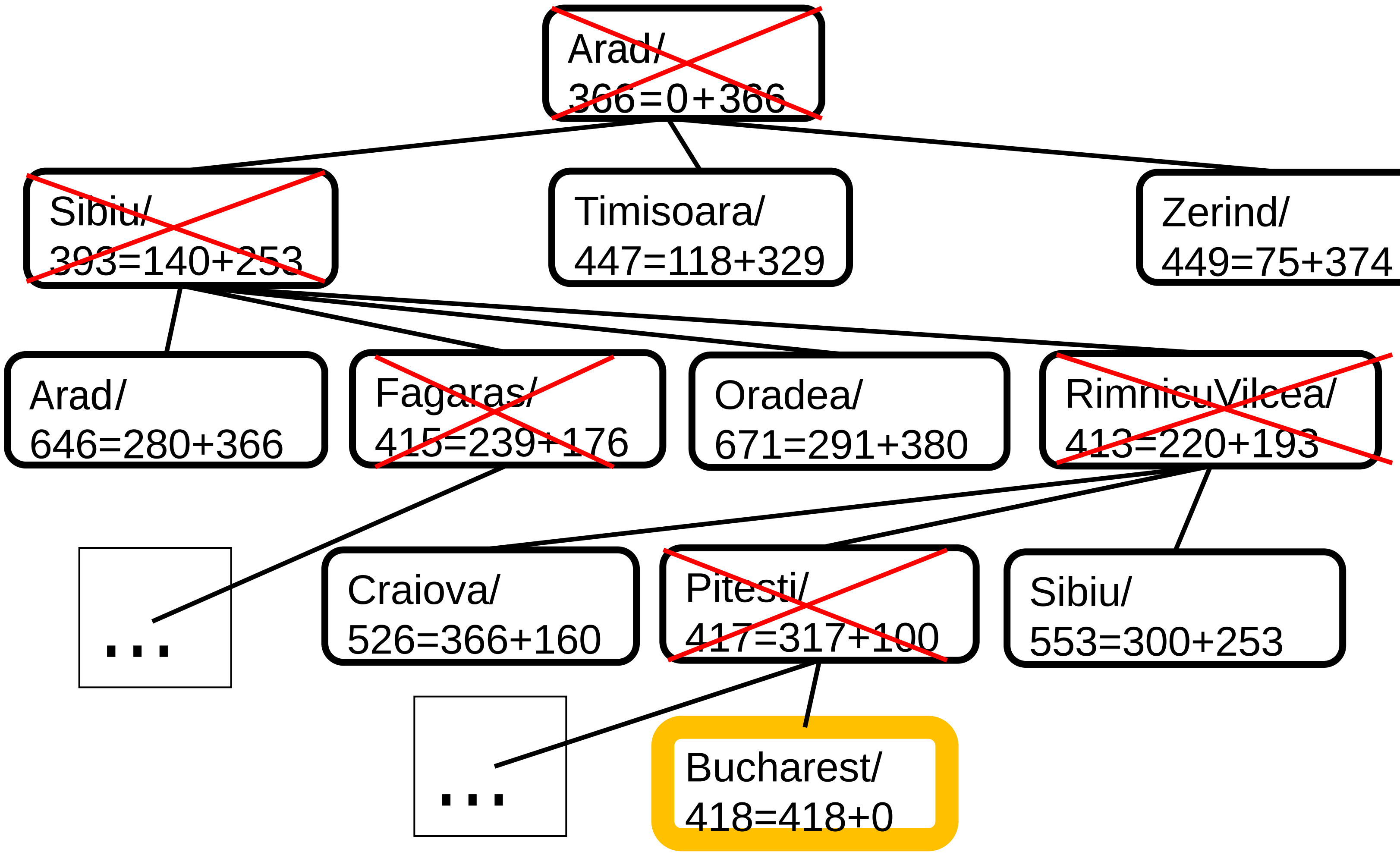
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

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

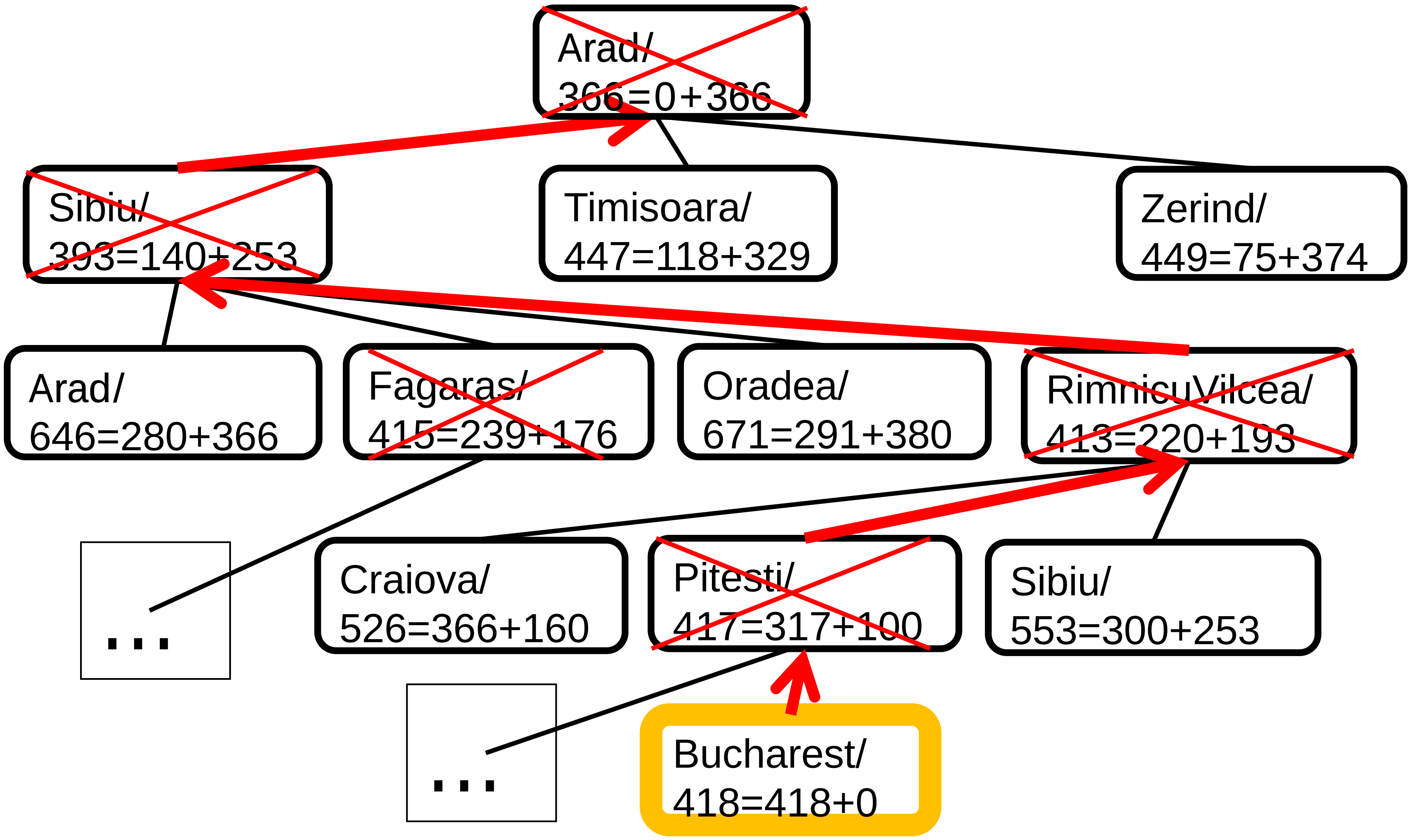
- Next: Bucharest/418=418+0
- Children: **None; goal test succeeds.**
- Expanded: Arad/366=0+366, Sibiu/393=140+253, RimnicuVilcea/413=220+193, Fagaras/415=239+176, Pitesti/417=317+100, Bucharest/418=418+0
- Frontier: ~~Arad/366=0+366, Sibiu/393=140+253~~, Timisoara/447=118+329, Zerind/449=75+374, Arad/646=280+366, ~~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 $\geq \varepsilon > 0$)
- 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)

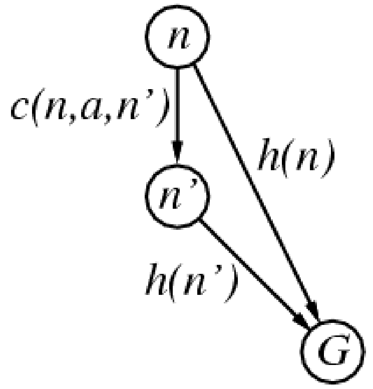
Admissible heuristics

- 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'))
≥ g(n) + h(n) = f(n) (consistency)
f(n') ≥ f(n)

It's the triangle inequality !

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

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

keeps all checked nodes in memory to avoid repeated states

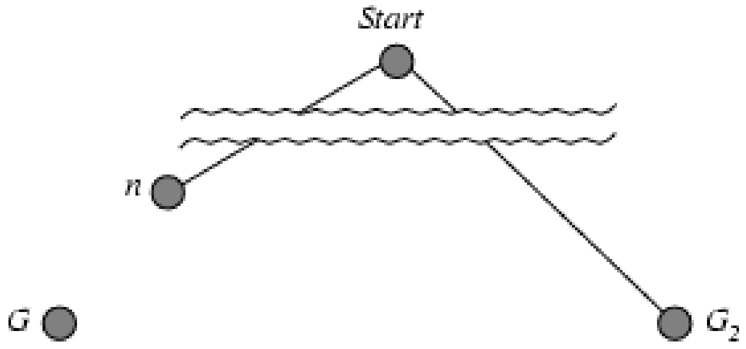
Optimality of A* (proof)

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
- $h(n) \le h^*(n)$ since h is admissible (*under*-estimate)
- $g(n) + h(n) \le g(n) + h^*(n)$ from above
- $f(n) \le f(G)$ since $g(n)+h(n)=f(n)$ & $g(n)+h^*(n)=f(G)$
- $f(n) < f(G2)$ from above



Dominance

- IF $h_2(n) \geq h_1(n)$ for all n
  THEN h_2 <u>dominates</u> h_1
  - h_2 is <u>almost always better</u> for search than h_1
  - h_2 <u>guarantees</u> to expand no more nodes than does h_1
  - h_2 <u>almost always</u> 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

Local search algorithms (4.1, 4.2)

- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution

- State space = set of "complete" configurations
- Find configuration satisfying constraints, e.g., n-queens
- In such cases, we can use local search algorithms
- keep a single "current" state, try to improve it.
- Very memory efficient (only remember current state)

Random Restart Wrapper

- These are stochastic local search methods
    - Different solution for each trial and initial state
- Almost every trial hits difficulties (see below)
    - Most trials will not yield a good result (sadly)
- Many random restarts improve your chances
    - Many “shots at goal” may, finally, get a good one
- Restart a random initial state; <u>many times</u>
    - Report the best result found; <u>across many trials</u>

Random Restart Wrapper

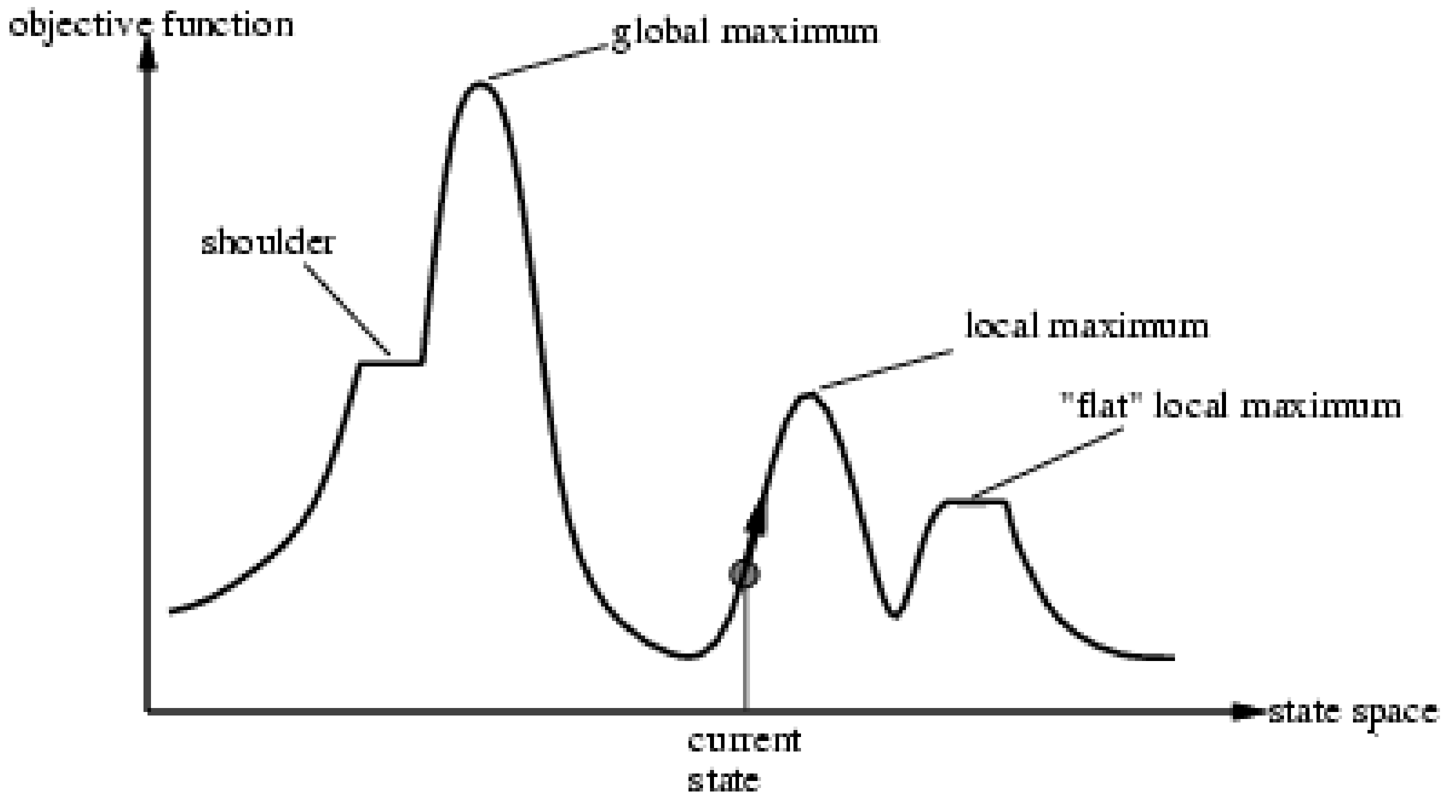
```
BestResultFoundSoFar <- infinitely bad;
UNTIL ( you are tired of doing it ) DO {
    Result <- ( Local search from random initial state );
    IF ( Result is better than BestResultFoundSoFar )
        THEN ( Set BestResultFoundSoFar to Result );
    }
RETURN BestResultFoundSoFar;
```

Typically, “you are tired of doing it” means that some resource limit is 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.

Local Search Difficulties

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

- 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 dimensionality of the search space increases to high dimensions.

- 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.

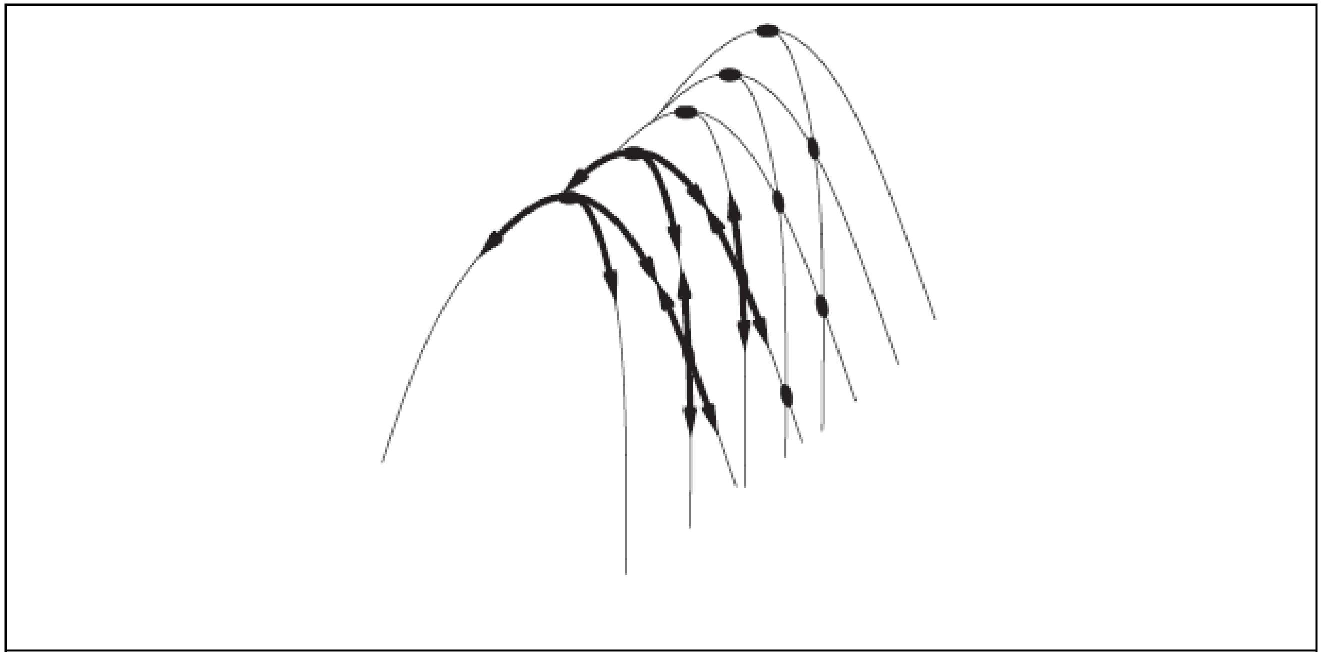
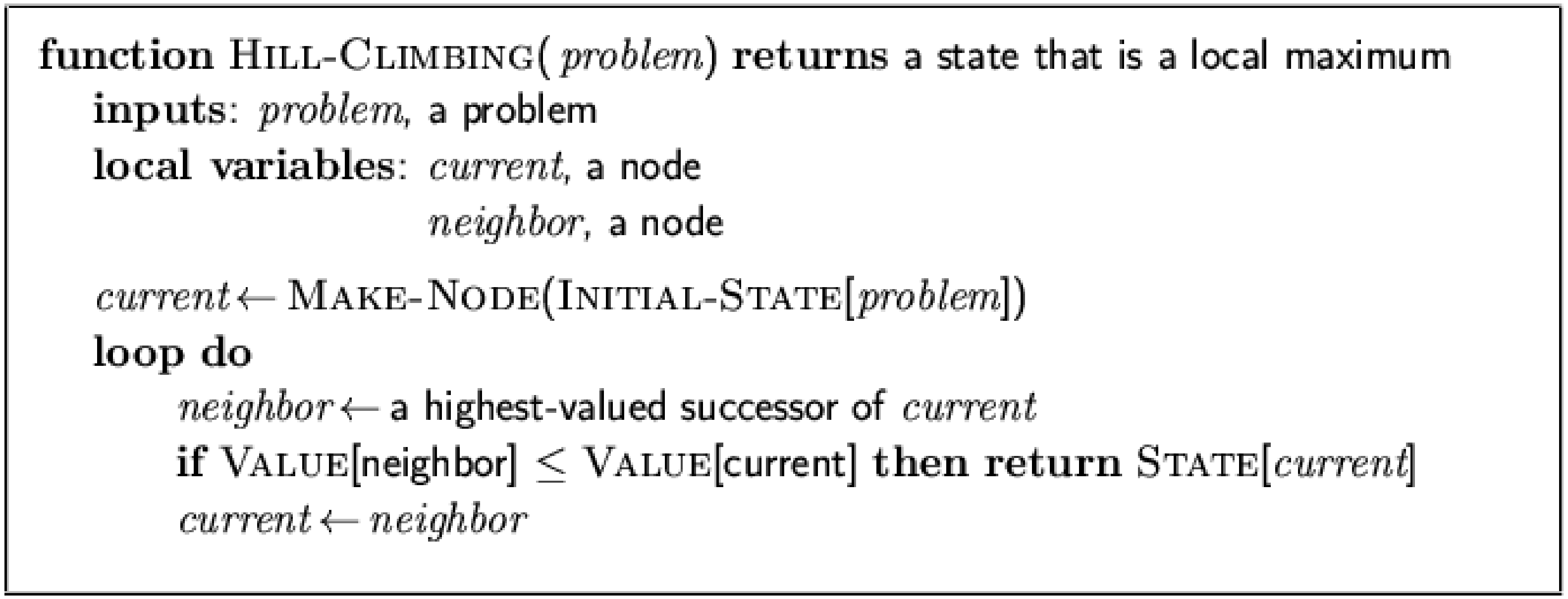


Figure 4.4 **FILES: figures/ridge.eps (Tue Nov 3 16:23:29 2009).** Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other.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

Hill-climbing search

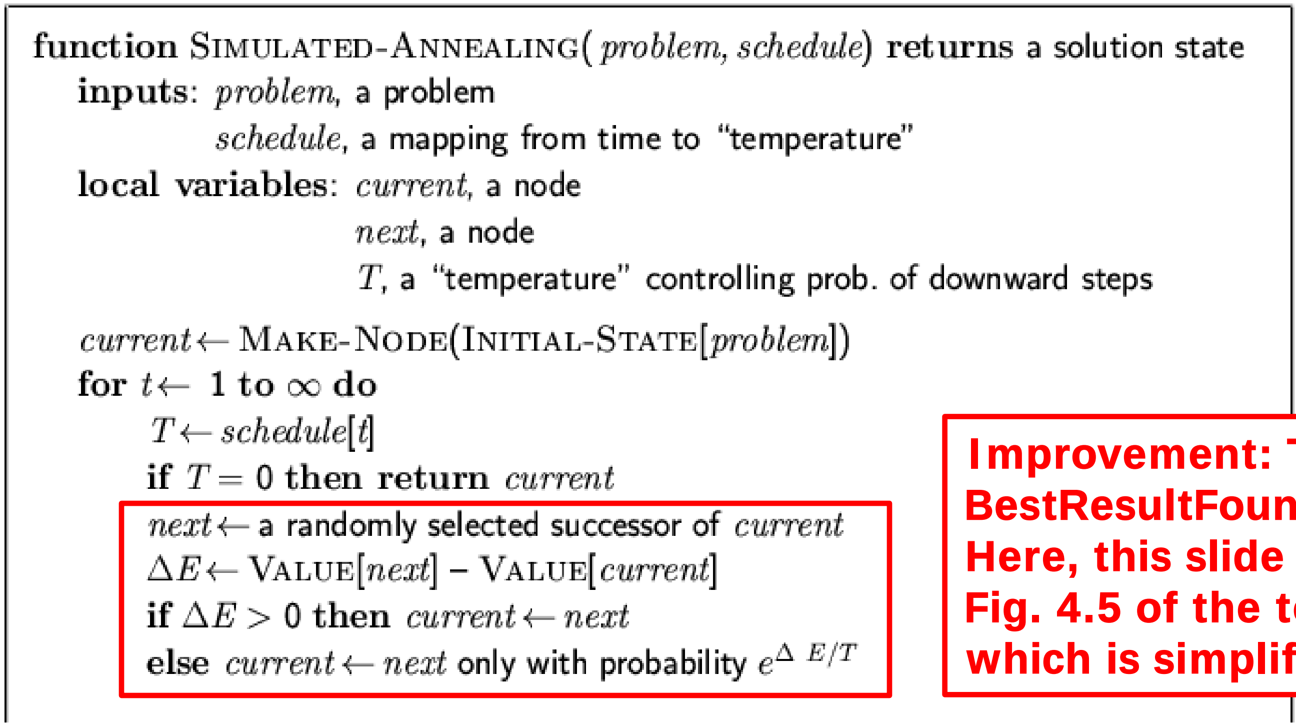
- "Like climbing Everest in thick fog with amnesia"
- 

```
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
```

Simulated annealing search

- 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 ∞ 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.

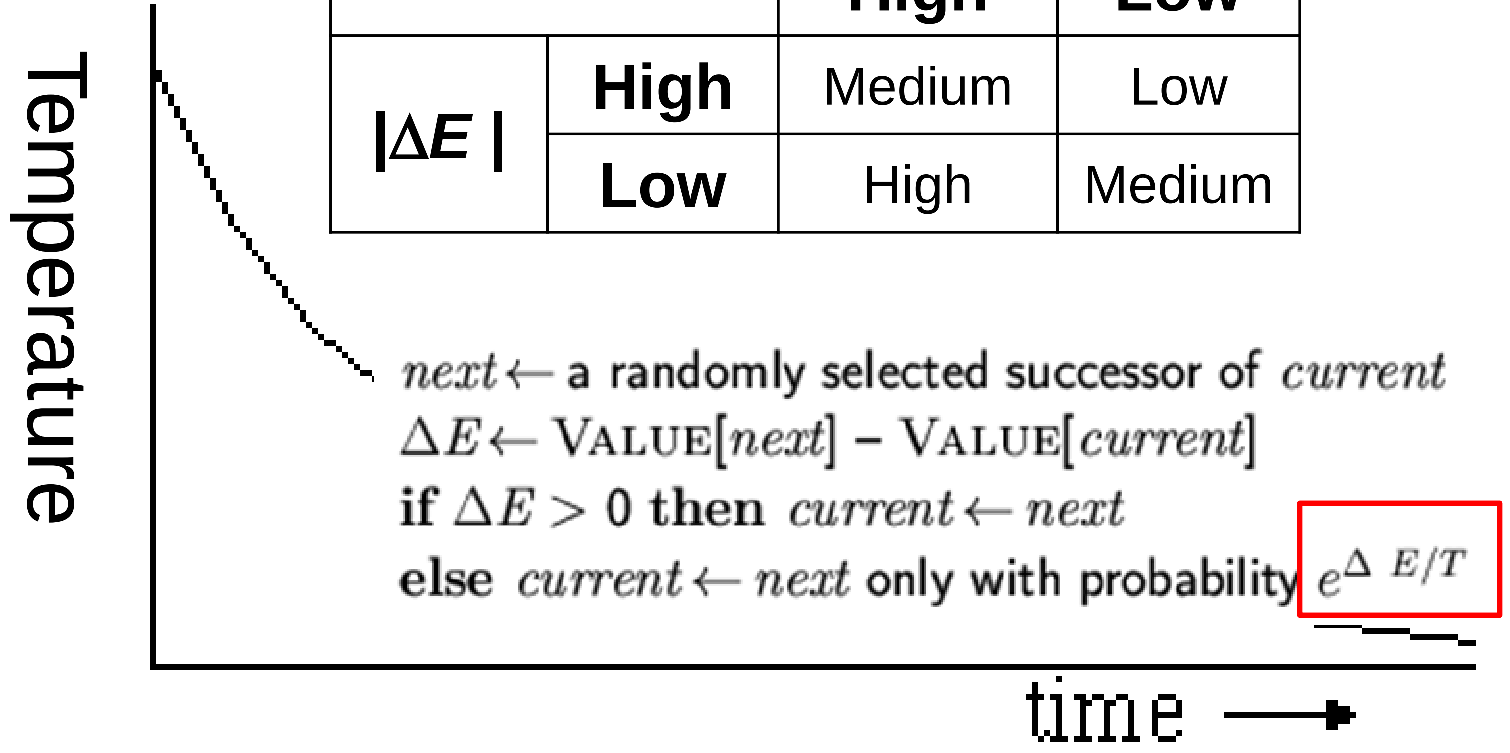
P(accepting a worse successor)

Decreases as Temperature T decreases

Increases as | Δ *E* | decreases

(Sometimes step size also decreases with T)

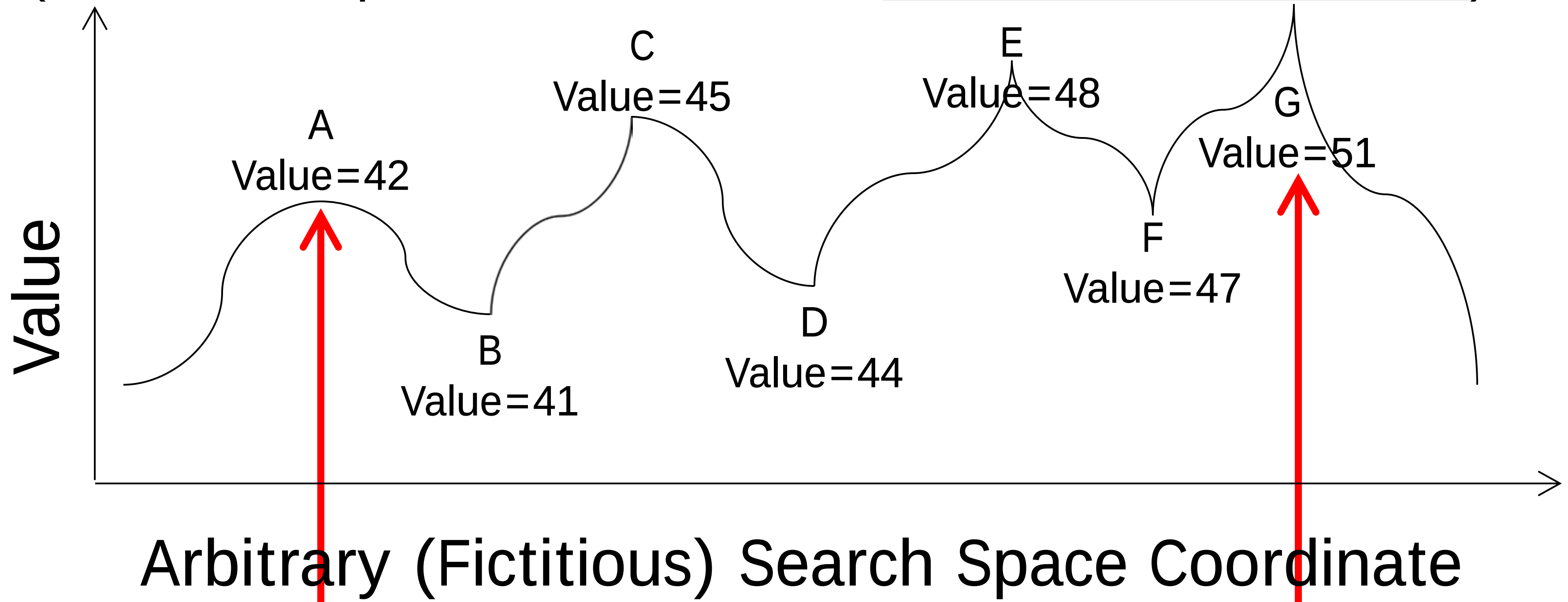
e^(Δ***E* / T**)		**Temperature T**	
		High	**Low**
|Δ*E* |	**High**	Medium	Low
	Low	High	Medium



```
$next \leftarrow$ a randomly selected successor of $current$
$\Delta E \leftarrow$ VALUE[$next$] – 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 jagged slope

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



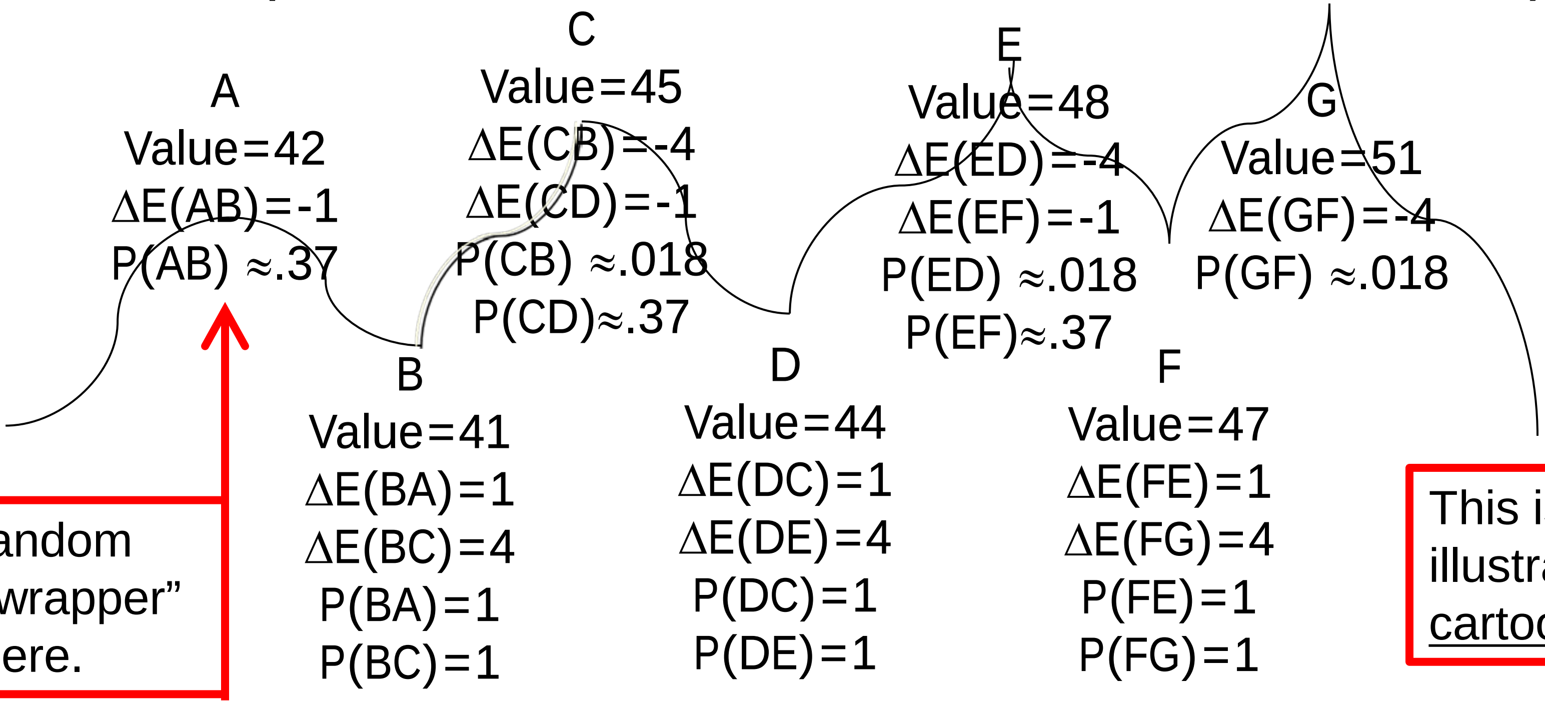
Your “random restart wrapper” starts here.

You want to get here. HOW??

This is an illustrative <u>cartoon.</u>

Goal: “Ratchet” up a jagged slope

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



Your “random restart wrapper” starts here.

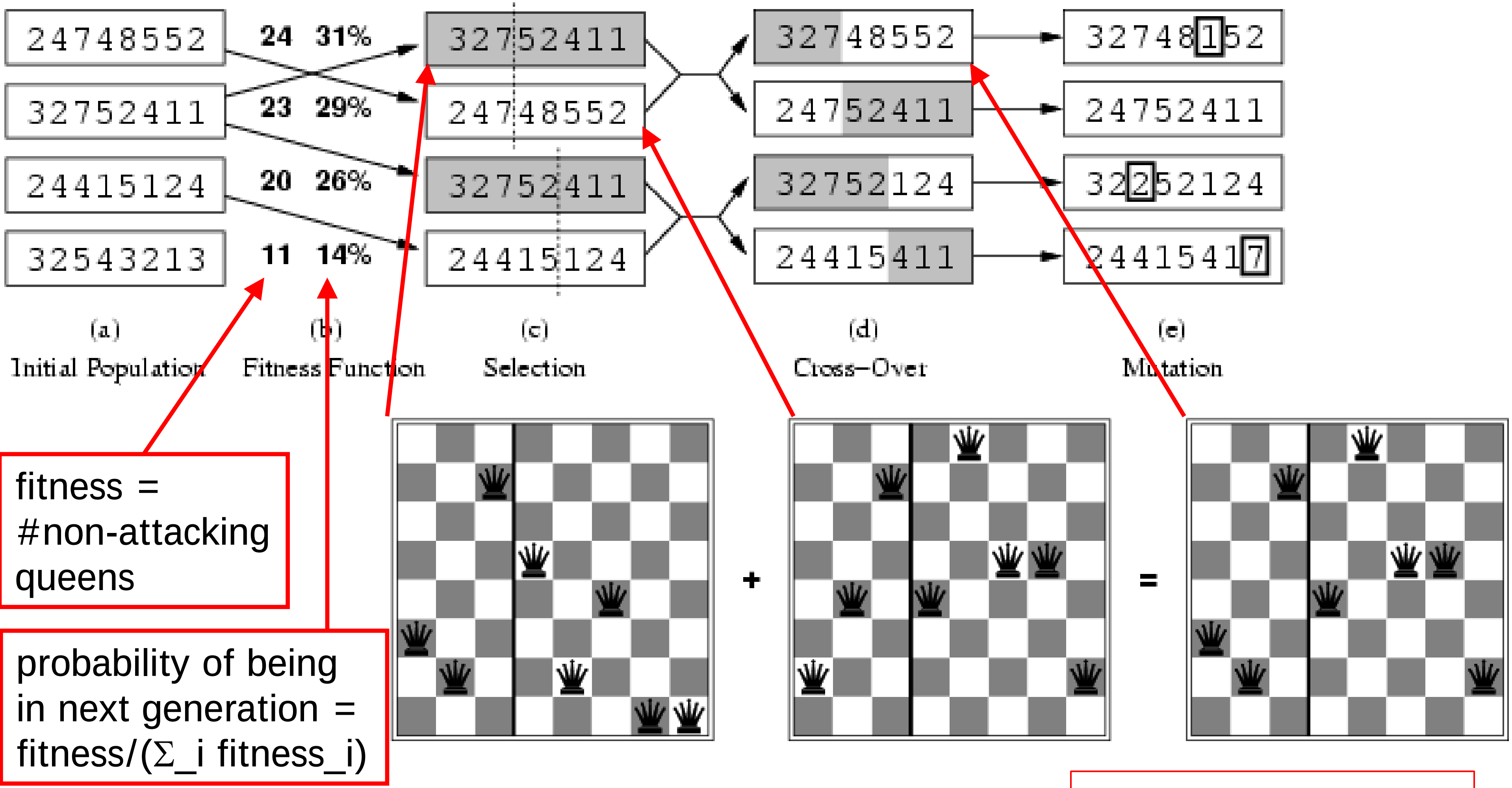
This is an illustrative cartoon.

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?).

Genetic algorithms (Darwin!!)

- A state = a string over a finite alphabet (an individual)
- 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/Σ population fitness
- Crossover fit parents to yield next generation (off-spring)
- Mutate the offspring randomly with some low probability



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 Adversarial (Game) Search Chapter 5.1-5.4

- Minimax Search with Perfect Decisions (5.2)
  – Impractical in most cases, but theoretical basis for analysis
- Minimax Search with Cut-off (5.4)
  – Replace terminal leaf utility by heuristic evaluation function
- Alpha-Beta Pruning (5.3)
  – The fact of the adversary leads to an advantage in search!
- Practical Considerations (5.4)
  – Redundant path elimination, look-up tables, etc.

Games as Search

- Two players: MAX and MIN
- MAX moves first and they take turns until the game is over
  - Winner gets reward, loser gets penalty.
  - “Zero sum” means the sum of the reward and the penalty is a constant.

- Formal definition as a search problem:
  - **Initial state:** Set-up specified by the rules, e.g., initial board configuration of chess.
  - **Player(s):** Defines which player has the move in a state.
  - **Actions(s):** Returns the set of legal moves in a state.
  - **Result(s,a):** Transition model defines the result of a move.
  - (**2nd ed.: Successor function:** list of (move,state) pairs specifying legal moves.)
  - **Terminal-Test(s):** Is the game finished? True if finished, false otherwise.
  - **Utility function(s,p):** Gives numerical value of terminal state s for player p.
    - E.g., win (+1), lose (-1), and draw (0) in tic-tac-toe.
    - E.g., win (+1), lose (0), and draw (1/2) in chess.

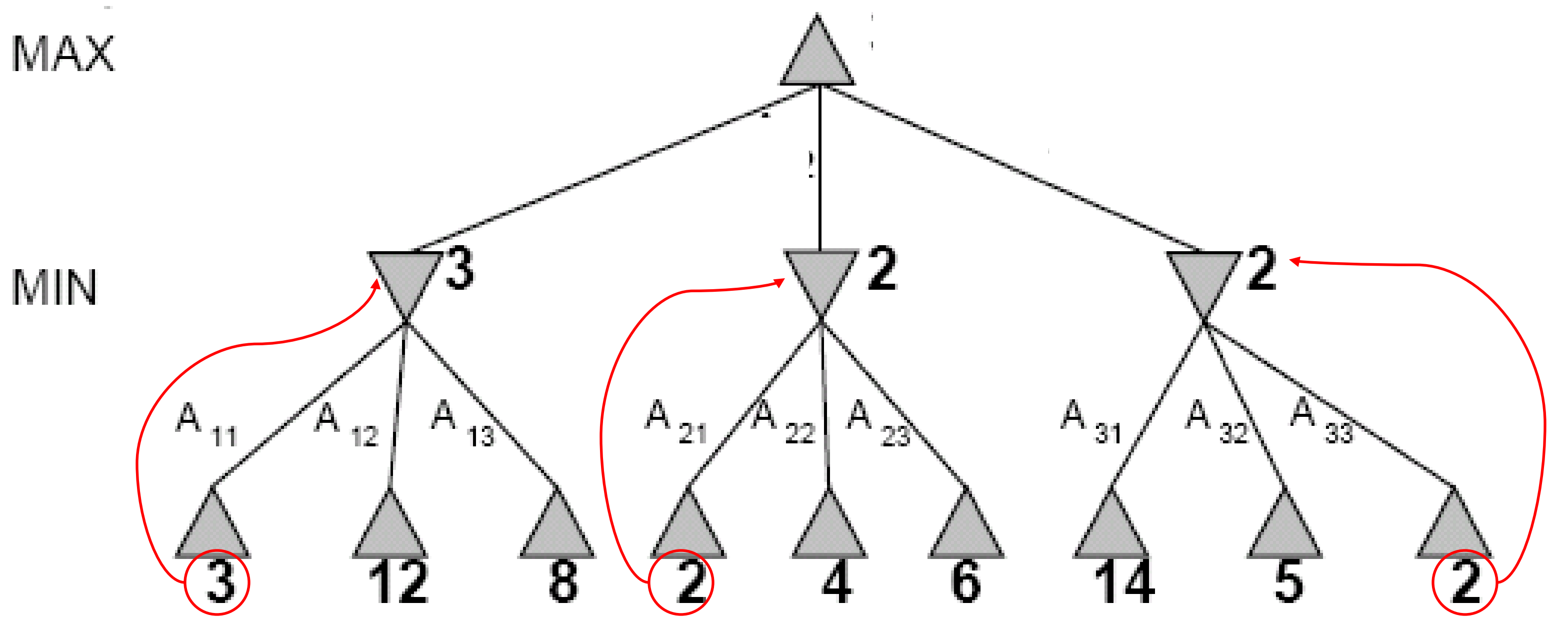
- MAX uses search tree to determine “best” next move.

An optimal procedure: The Min-Max method

Will find the optimal strategy and best next move for Max:

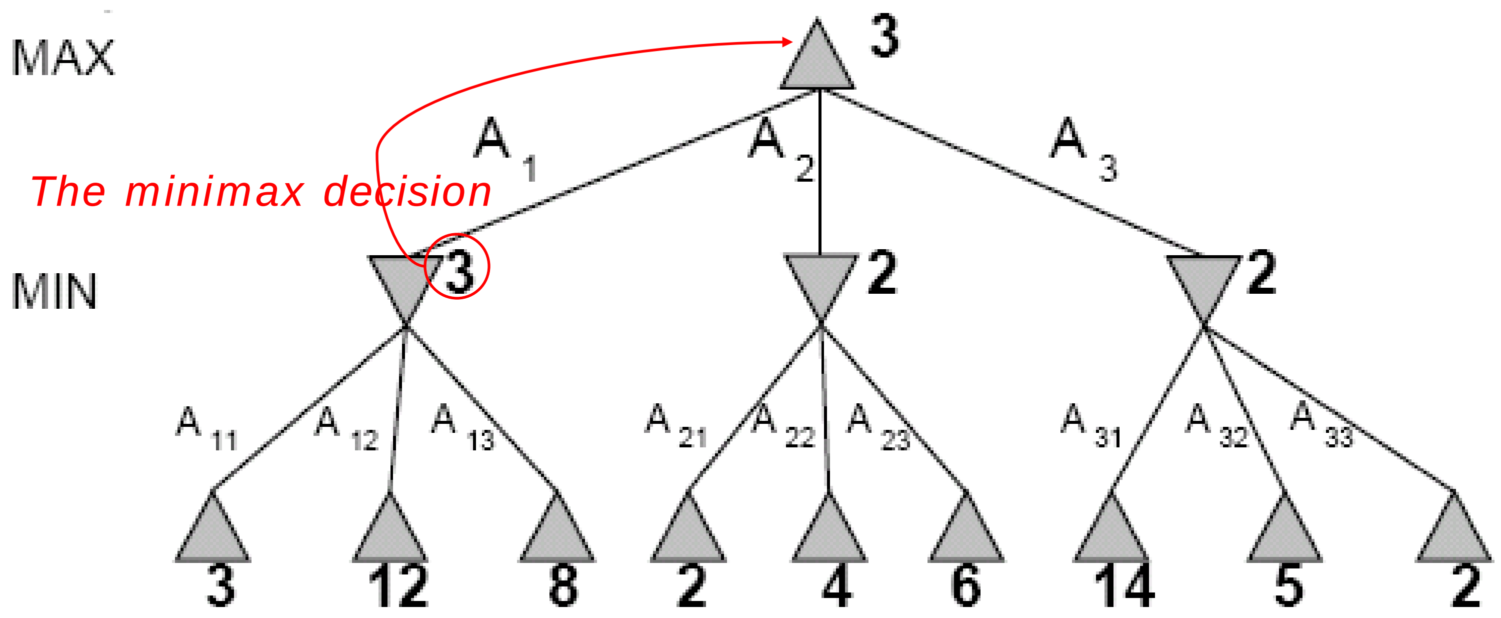
- 1. Generate the whole game tree, down to the leaves.
- 2. Apply utility (payoff) function to each leaf.
- 3. Back-up values from leaves through branch nodes:
  - a Max node computes the Max of its child values
  - a Min node computes the Min of its child values
- 4. At root: choose move leading to the child of highest value.

Two-Ply Game Tree



Two-Ply Game Tree

Minimax maximizes the utility of the worst-case outcome for Max



Pseudocode for Minimax Algorithm

```
function MINIMAX-DECISION(state) returns an action
   inputs: state, current state in game
return arg max$_{a \in \text{ACTIONS}(state)}$ MIN-VALUE(Result(state,a))
```

```
function MAX-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow -\infty$
   for a in ACTIONS(state) do
      $v \leftarrow$ MAX(v,MIN-VALUE(Result(state,a)))
   return v
```

```
function MIN-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow +\infty$
   for a in ACTIONS(state) do
      $v \leftarrow$ MIN(v,MAX-VALUE(Result(state,a)))
   return v
```

Properties of minimax

- **Complete?**
  - Yes (if tree is finite).

- **Optimal?**
  - Yes (against an optimal opponent).
  - Can it be beaten by an opponent playing sub-optimally?
    - No. (Why not?)

- **Time complexity?**
  - O(b^m)

- **Space complexity?**
  - O(bm) (depth-first search, generate all actions at once)
  - O(m) (backtracking search, generate actions one at a time)

Cutting off search

MinimaxCutoff is identical to MinimaxValue except

1. Terminal? is replaced by Cutoff?
2. Utility is replaced by Eval

Does it work in practice?

$$b^m = 10^6, \quad b = 35 \quad \Rightarrow \quad m = 4$$

4-ply lookahead is a hopeless chess player!

4-ply $\approx$ human novice
8-ply $\approx$ typical PC, human master
12-ply $\approx$ Deep Blue, Kasparov

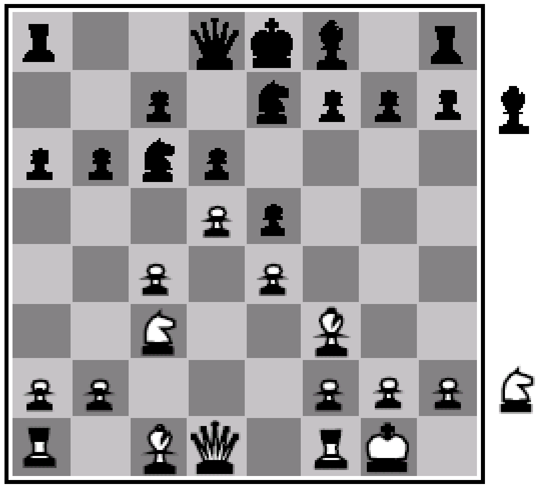
Static (Heuristic) Evaluation Functions

- An Evaluation Function:
  - Estimates how good the current board configuration is for a player.
  - Typically, evaluate how good it is for the player, how good it is for the opponent, then subtract the opponent's score from the player's.
  - Othello: Number of white pieces - Number of black pieces
  - Chess: Value of all white pieces - Value of all black pieces

- Typical values from -infinity (loss) to +infinity (win) or [-1, +1].

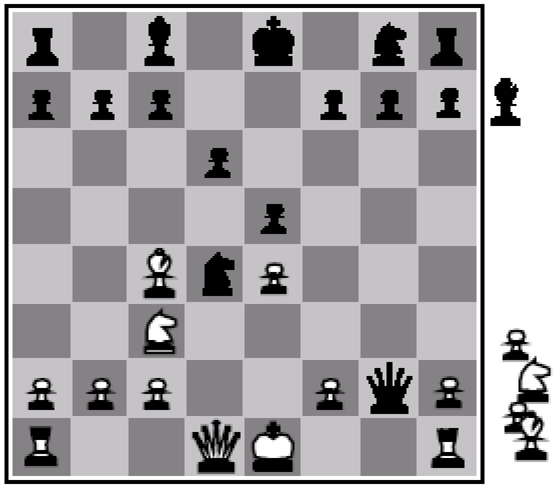
- If the board evaluation is X for a player, it's -X for the opponent
  - "Zero-sum game"

Evaluation functions



Black to move

White slightly better



White to move

Black winning

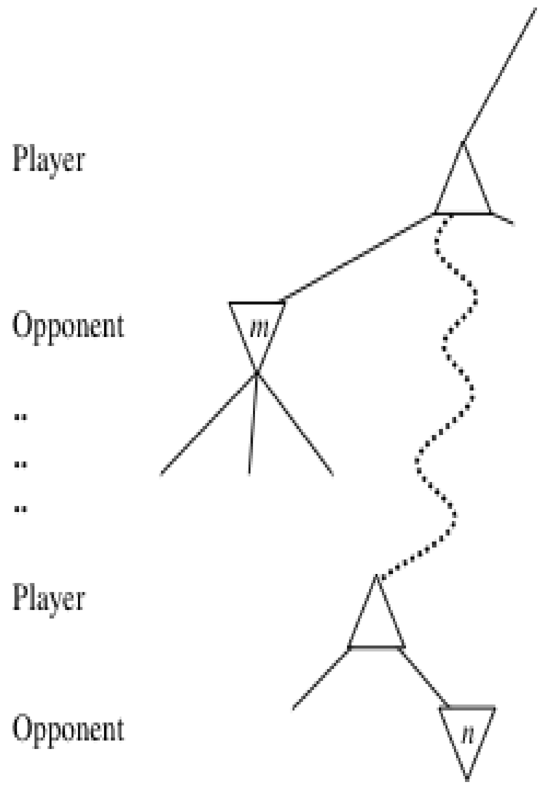
For chess, typically *linear* weighted sum of features

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

e.g., $w_1 = 9$ with
$f_1(s) =$ (number of white queens) – (number of black queens), etc.

General alpha-beta pruning

- Consider a node *n* in the tree ---
- If player has a better choice at:
  - Parent node of n
  - Or any choice point further up
- Then *n* will never be reached in play.
- Hence, when that much is known about *n*, it can be pruned.



Alpha-beta Algorithm

- Depth first search
  - only considers nodes along a single path from root at any time

α = highest-value choice found at any choice point of path for MAX
(initially, $\alpha = -\text{infinity}$)

β = lowest-value choice found at any choice point of path for MIN
(initially, $\beta = +\text{infinity}$)

- Pass current values of α and β down to child nodes during search.
- Update values of α and β during search:
  - MAX updates α at MAX nodes
  - MIN updates β at MIN nodes
- Prune remaining branches at a node when $\alpha \geq \beta$

Pseudocode for Alpha-Beta Algorithm

```
function ALPHA-BETA-SEARCH(state) returns an action
  inputs: state, current state in game
  $v \leftarrow$ MAX-VALUE(state, $-\infty$ , $+\infty$)
  return the action in ACTIONS(state) with value $v$
```

```
function MAX-VALUE(state, $\alpha$ , $\beta$) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow -\infty$
  for a in ACTIONS(state) do
     $v \leftarrow$ MAX($v$, MIN-VALUE(Result(s,a), $\alpha$ , $\beta$))
     if $v \geq \beta$ then return $v$
     $\alpha \leftarrow$ MAX($\alpha$ ,$v$)
  return $v$
```

(MIN-VALUE is defined analogously)

When to Prune?

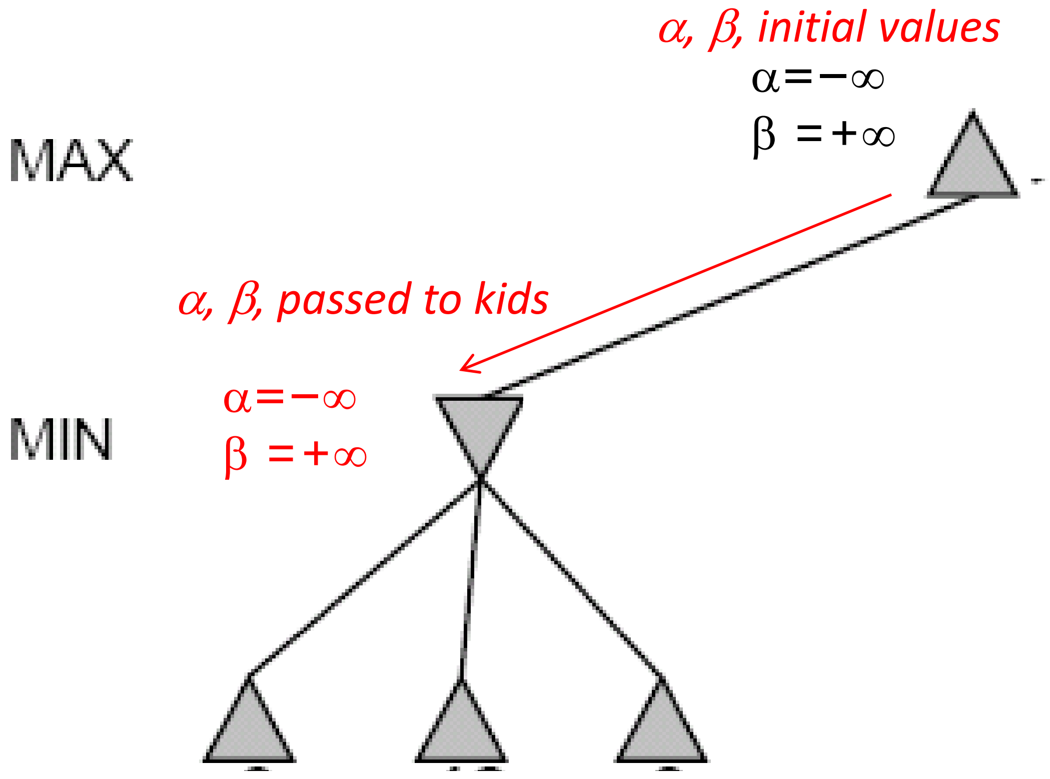
- **Prune whenever $\alpha \geq \beta$.**
  - Prune below a Max node whose alpha value becomes greater than or equal to the beta value of its ancestors.
    - **Max nodes update alpha** based on children's returned values.
  - Prune below a Min node whose beta value becomes less than or equal to the alpha value of its ancestors.
    - **Min nodes update beta** based on children's returned values.

α/β Pruning vs. Returned Node Value

- Some students are confused about the use of α/β pruning vs. the returned value of a node
- <u>α/β are used **ONLY FOR PRUNING**</u>
  - α/β have no effect on anything other than pruning
  - IF (α >= β) THEN prune & return current node value
- <u>Returned node value = “best” child seen so far</u>
  - Maximum child value seen so far for MAX nodes
  - Minimum child value seen so far for MIN nodes
  - If you prune, return to parent <u>“best” child so far</u>
- <u>Returned node value is received by parent</u>

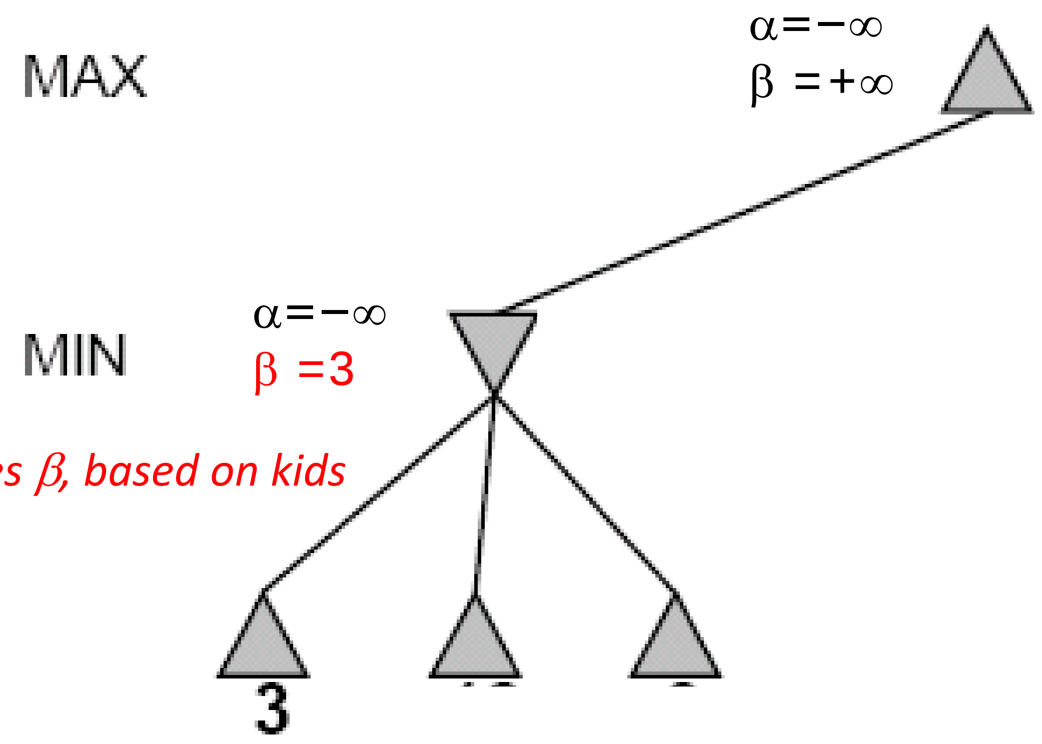
Alpha-Beta Example Revisited

Do DF-search until first leaf



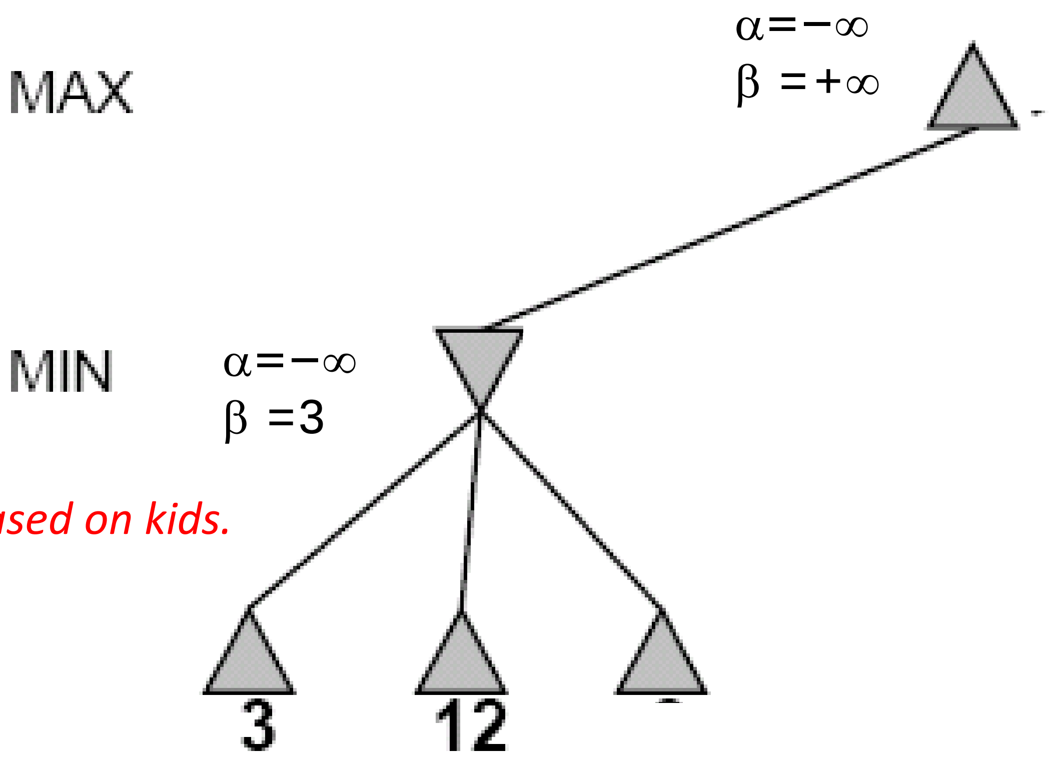
<u>Review Detailed Example of Alpha-Beta Pruning in lecture slides.</u>

Alpha-Beta Example (continued)



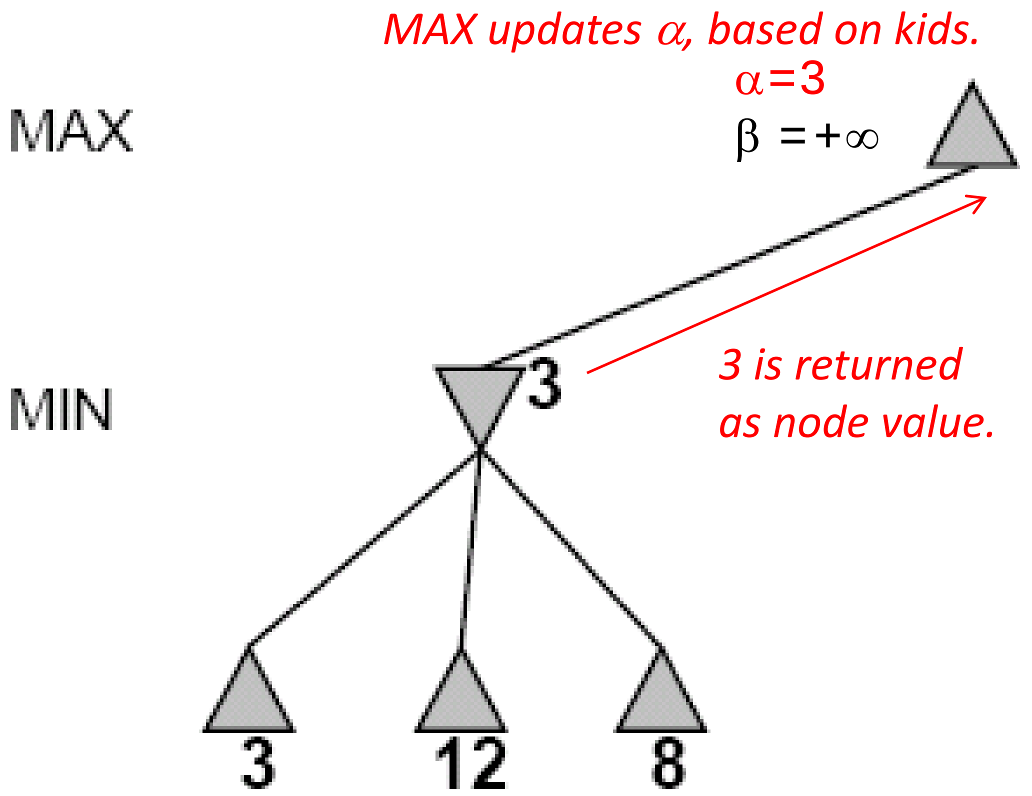
MIN updates β, based on kids

Alpha-Beta Example (continued)



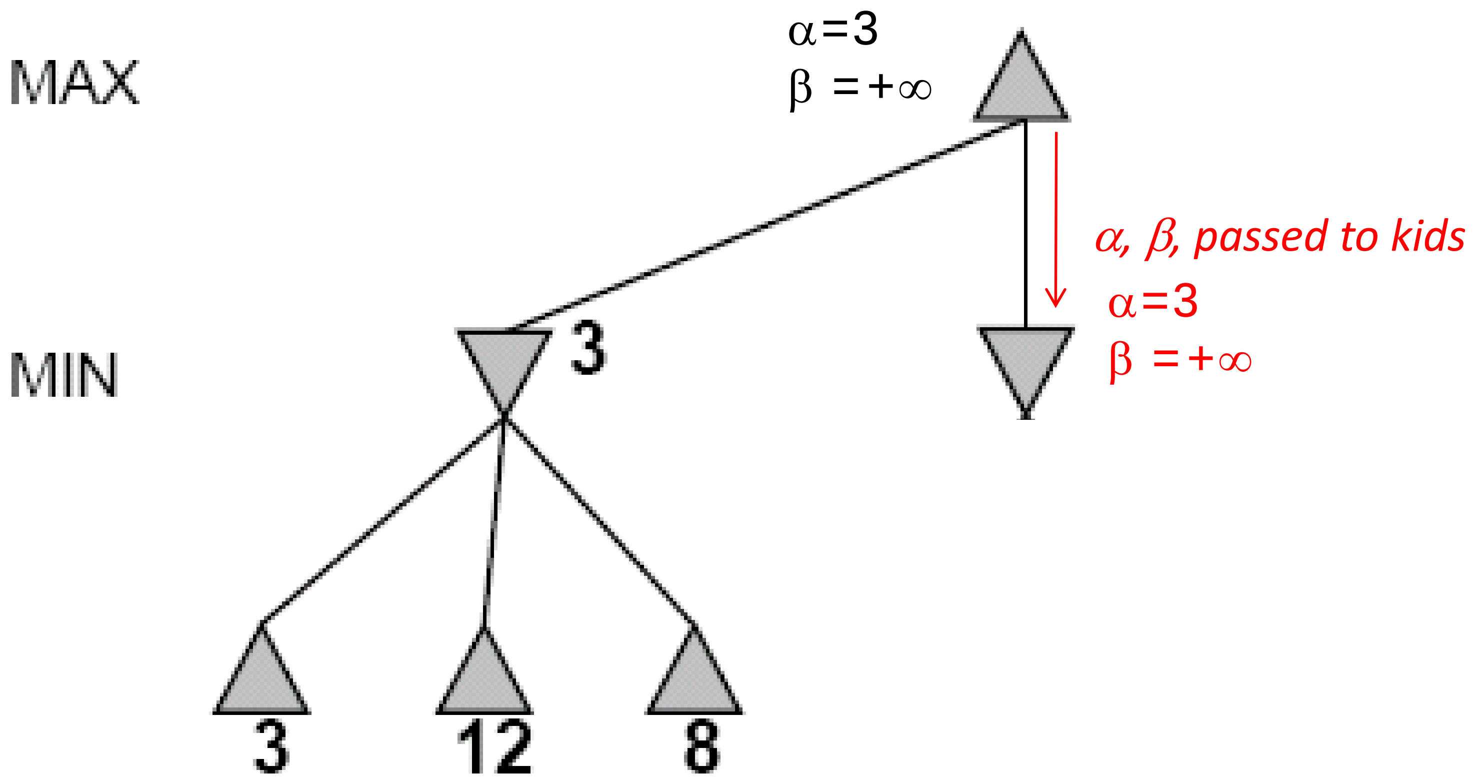
MIN updates β, based on kids.
No change.

Alpha-Beta Example (continued)

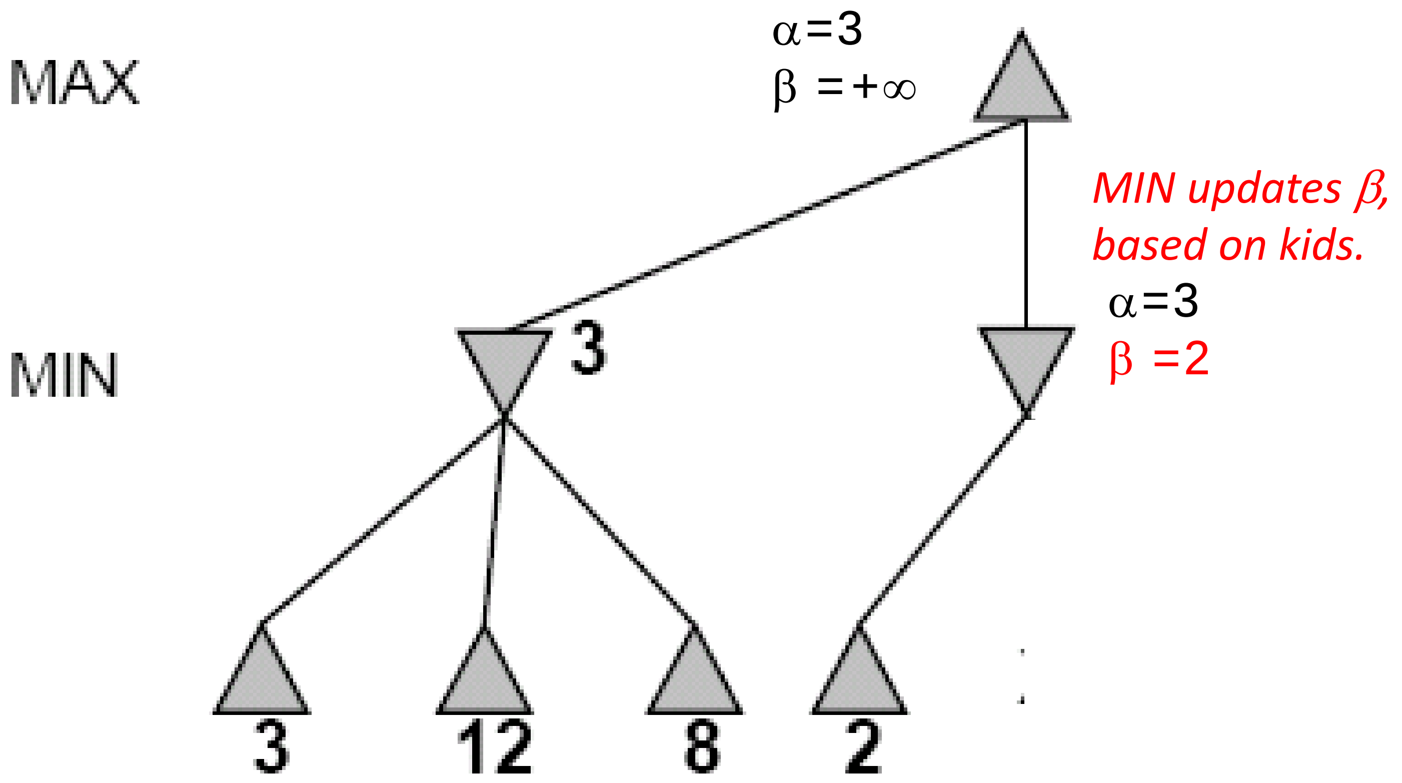


MAX updates α, based on kids.
α=3
β =+∞
MAX
MIN
3
3 is returned as node value.
3
12
8

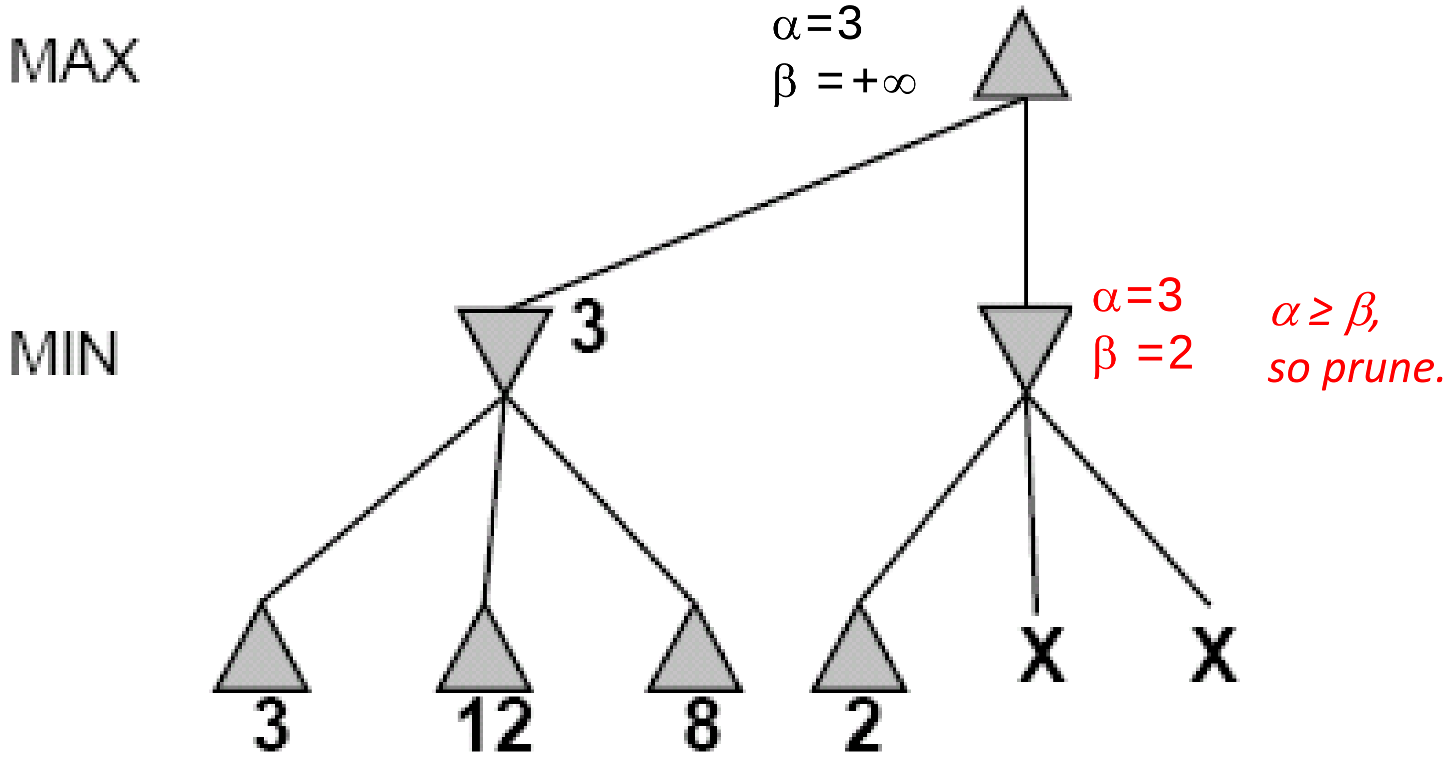
Alpha-Beta Example (continued)



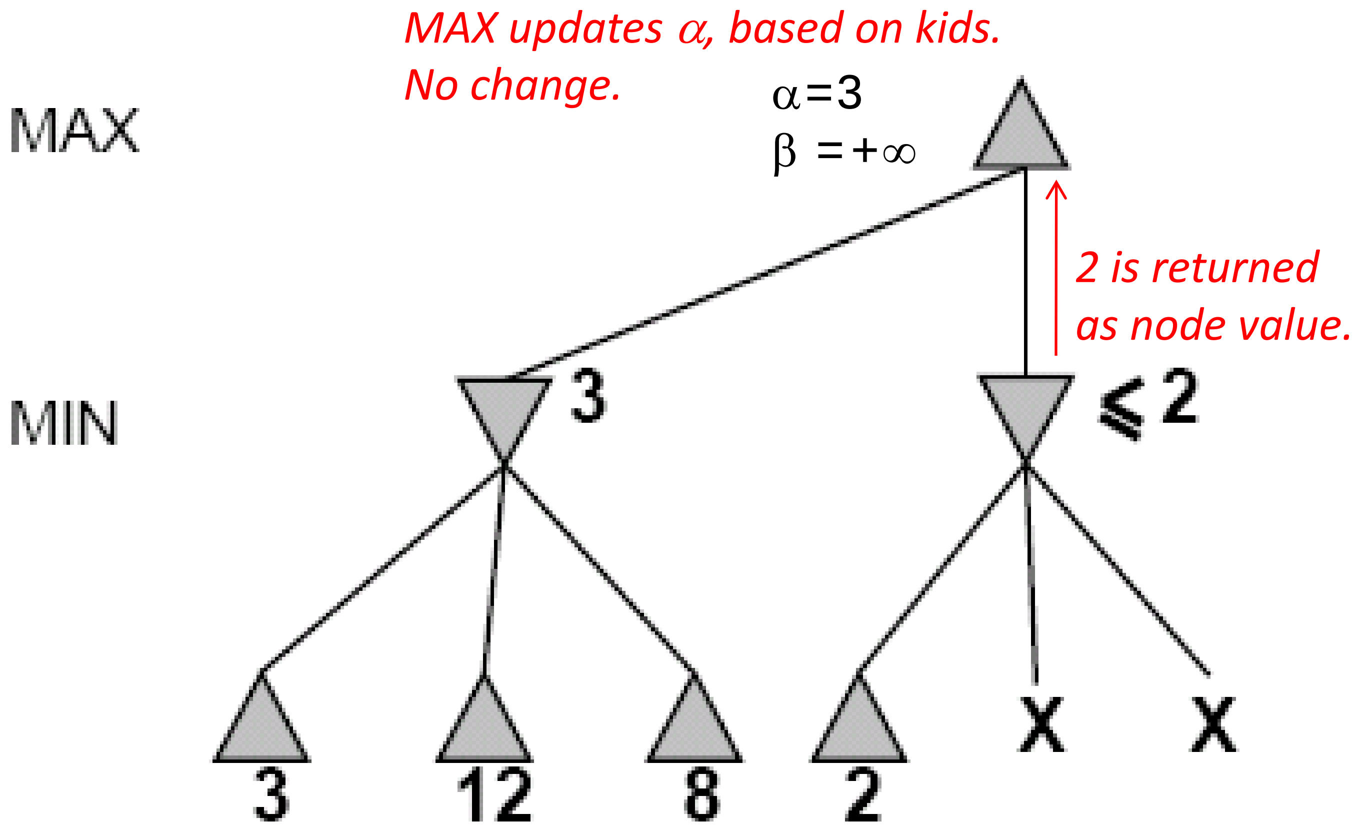
Alpha-Beta Example (continued)



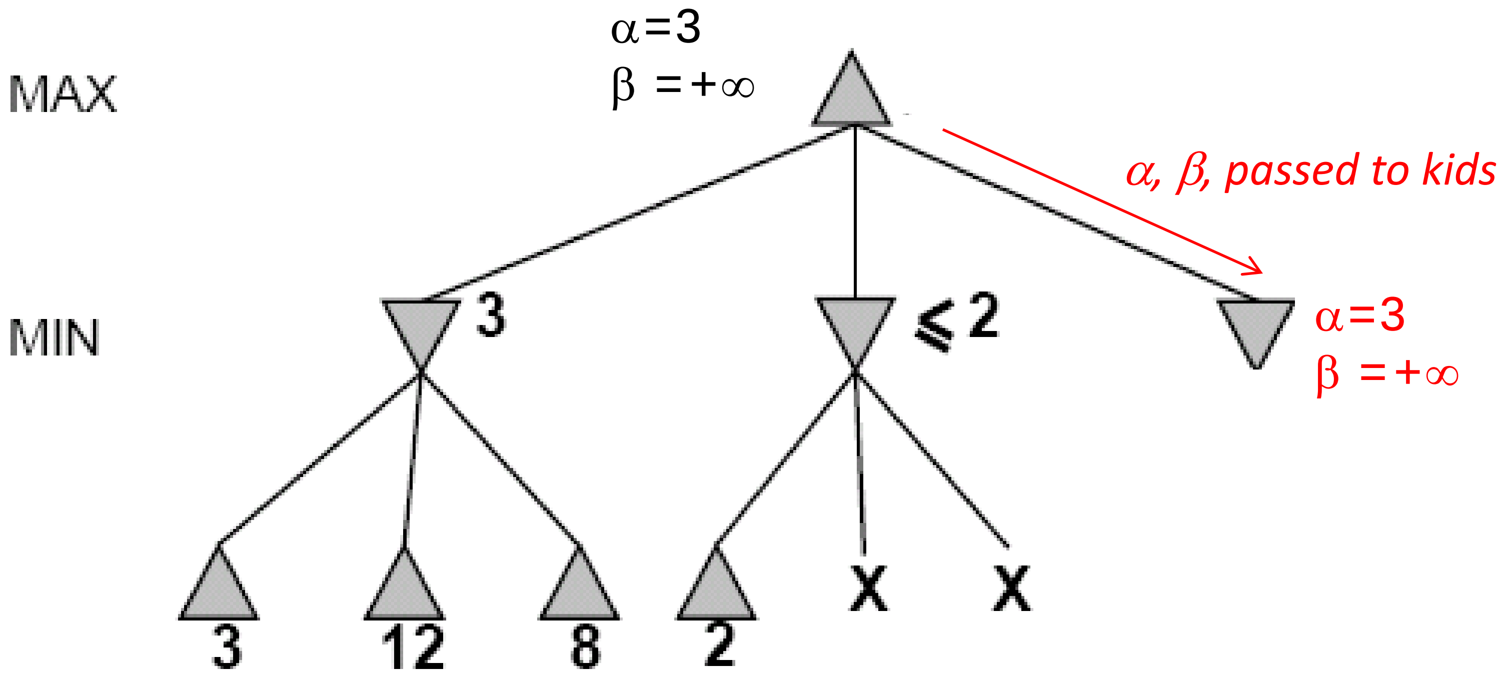
Alpha-Beta Example (continued)



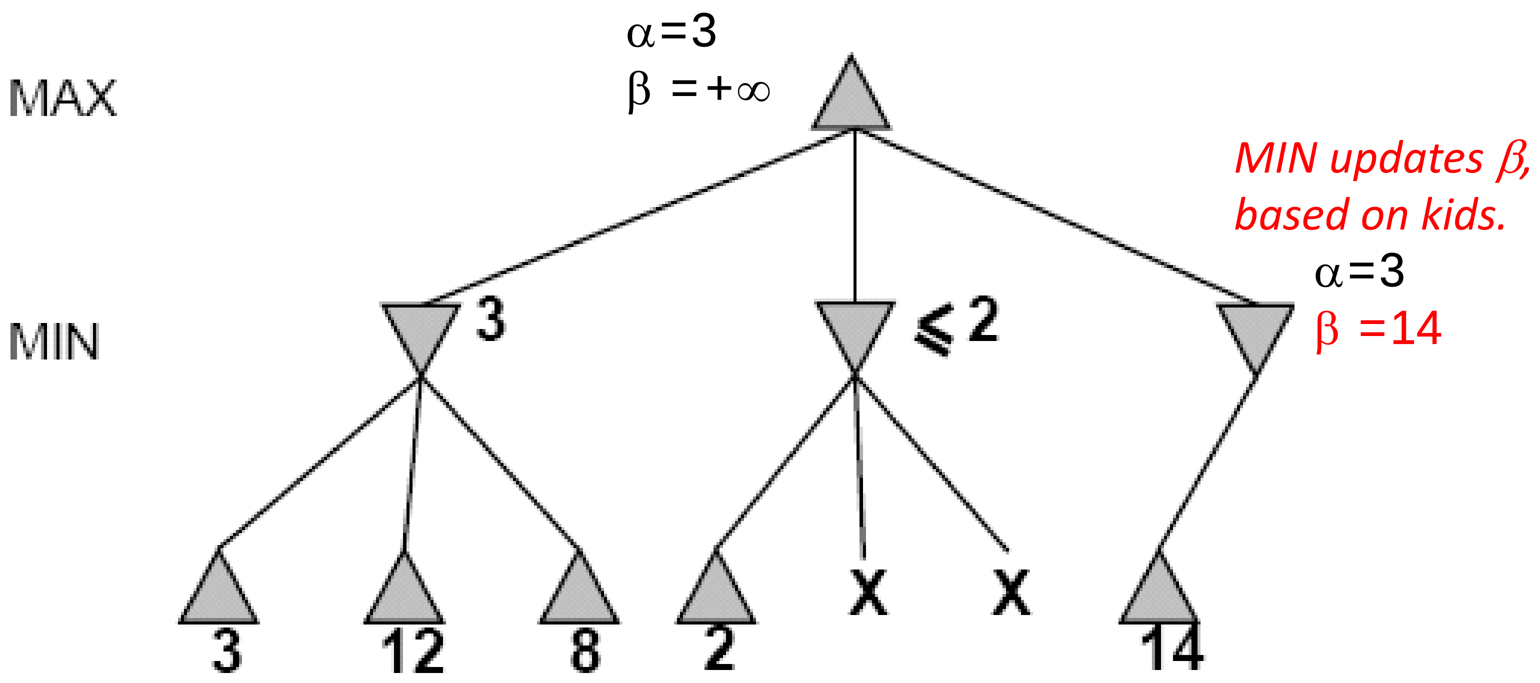
Alpha-Beta Example (continued)



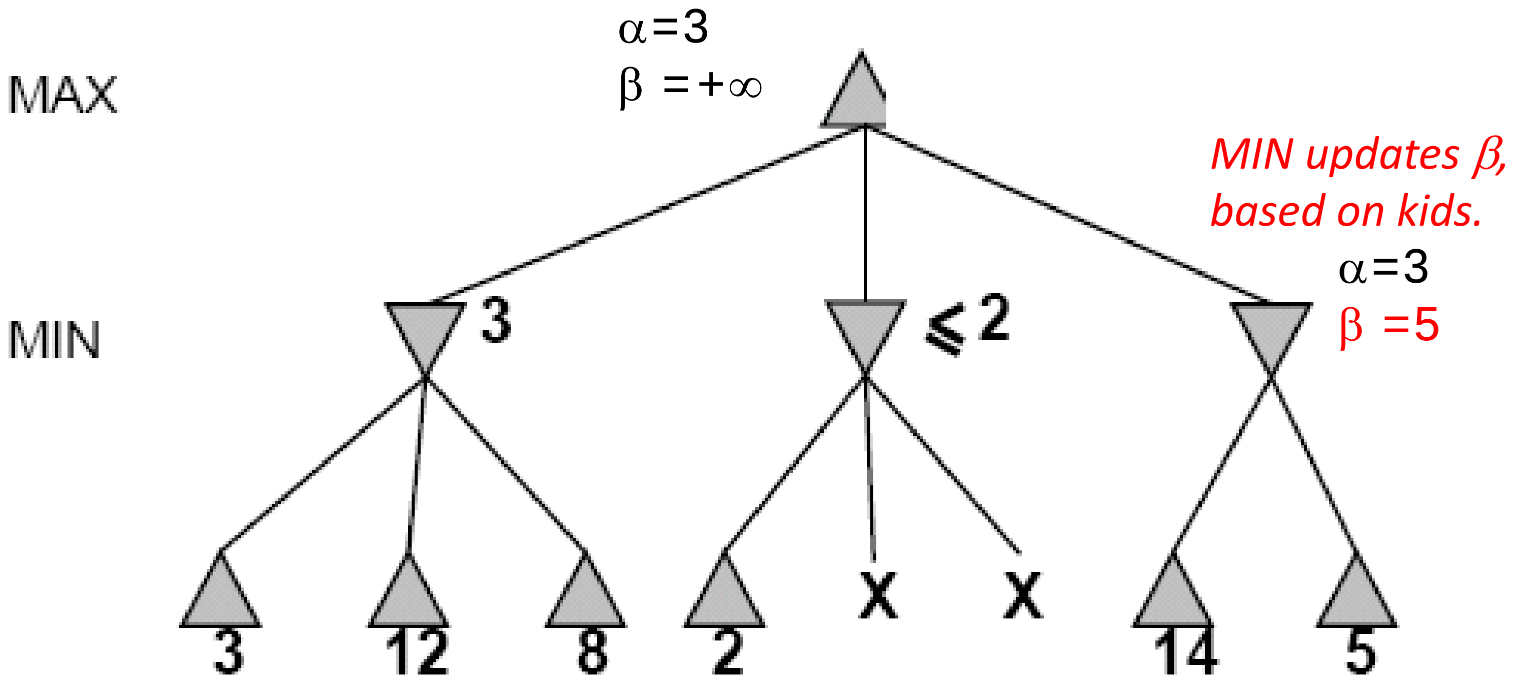
Alpha-Beta Example (continued)



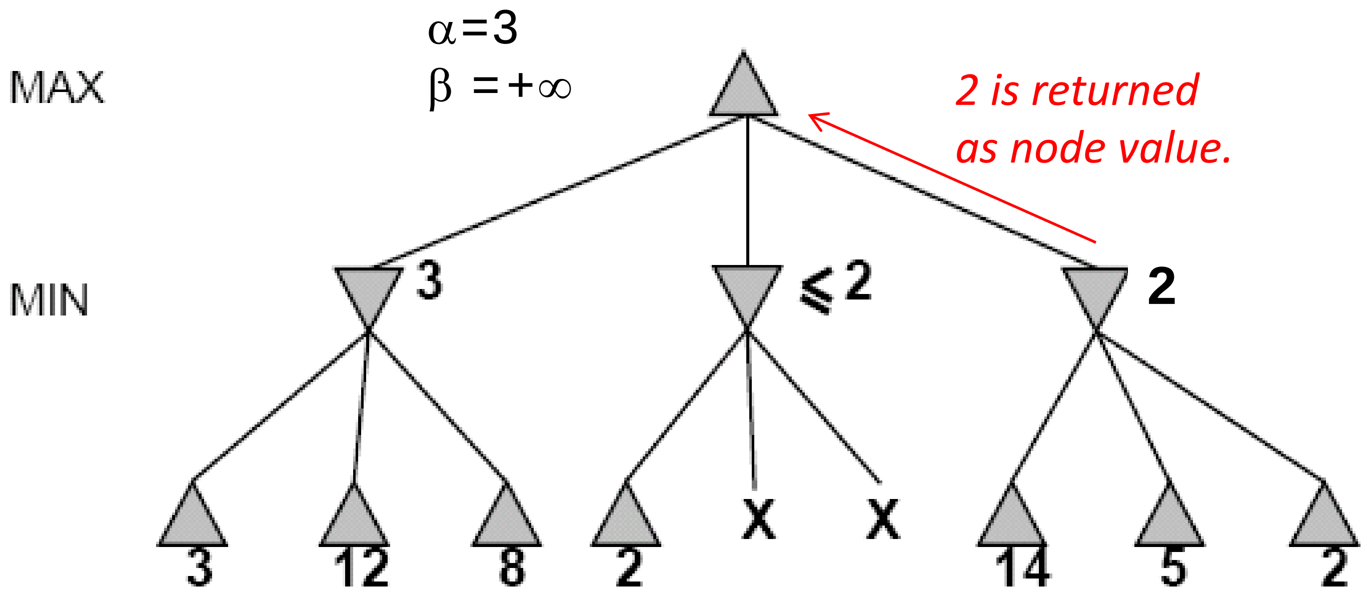
Alpha-Beta Example (continued)



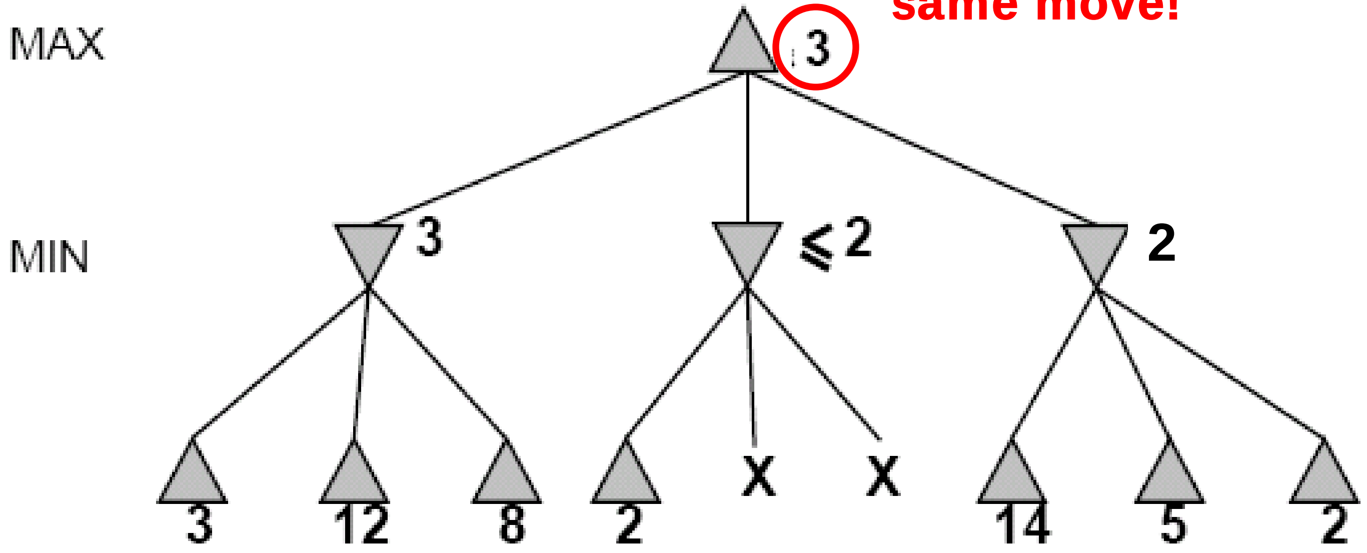
Alpha-Beta Example (continued)



Alpha-Beta Example (continued)



Alpha-Beta Example (continued)



Max calculates the same node value, and makes the same move!

Review Detailed Example of Alpha-Beta Pruning in lecture slides.

Review Probability (Chapter 13) You will be expected to know

- Basic probability notation/definitions:
  - Probability model, unconditional/prior and conditional/posterior probabilities, factored representation (= variable/value pairs), random variable, (joint) probability distribution, probability density function (pdf), marginal probability, (conditional) independence, normalization, etc.
- Basic probability formulae:
  - 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

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

(by definition of joint probability)

Repeatedly applying this idea, we can write

P(a, b, c, … z) = P(a | b, c, …. z) P(b | c,.. z) P(c| .. z)..P(z)

This factorization holds for any ordering of the variables

This is the chain rule for probabilities

Independence

- <u>Formal Definition</u>:
  - 2 random variables A and B are independent iff:

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

- <u>Informal Definition</u>:
  - 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 $X_1, X_2, \ldots X_k$, and C = disease, reducing to:

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

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)

- **You will be expected to know:**

- **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.
  - Inference by Variable Elimination

- **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.

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:

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

$$\underset{\text{The full joint distribution}}{P(X_1, X_2, X_3..., X_N)} = \underset{\text{The graph-structured approximation}}{\prod_{i=1}^{n} P(X_i|parents(X_i))}$$

- 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)

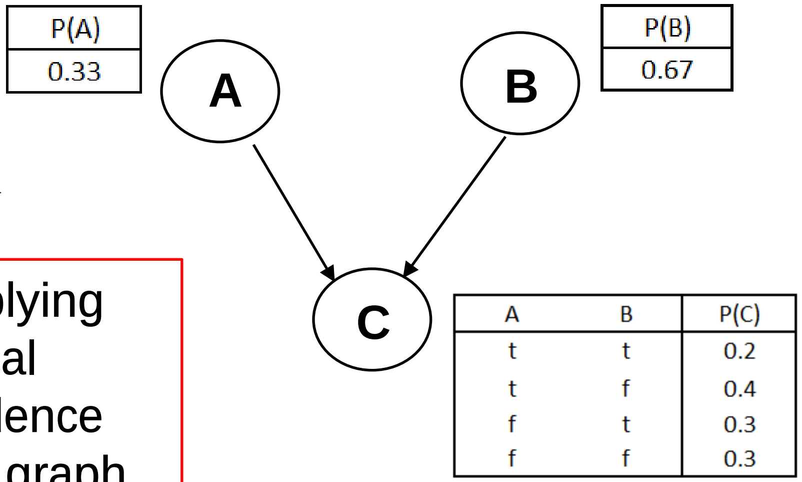
Bayesian Network

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

Full factorization

$$p(A,B,C) = p(C|A,B)p(A|B)p(B)$$
$$= p(C|A,B)p(A)p(B)$$

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

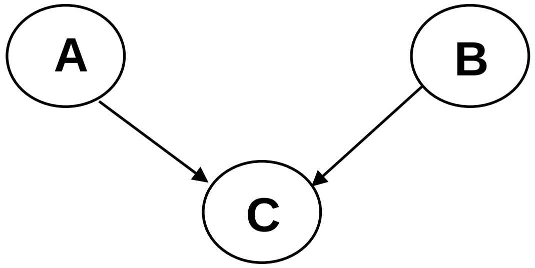
- Dependence/independence represented via a directed graph:

− 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:
$p(A,B,C) = p(C|A,B)p(A)p(B)$

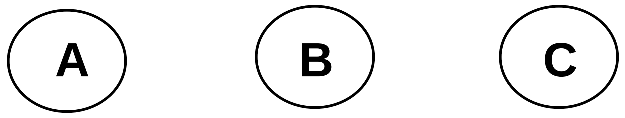
"Explaining away" effect:
Given C, observing A makes B less likely
e.g., earthquake/burglary/alarm example

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

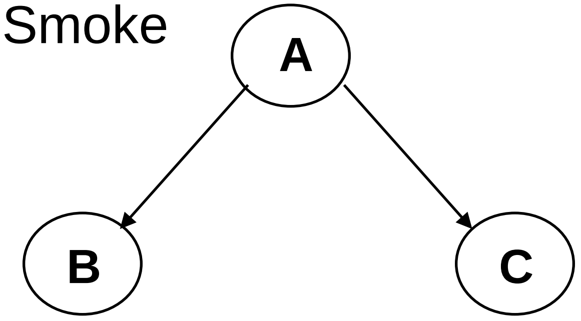


Marginal Independence:
p(A,B,C) = p(A) p(B) p(C)

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


$$\begin{tikzcd} & A \arrow[dl] \arrow[dr] & \\ B & & C \end{tikzcd}$$

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

B and C are conditionally independent Given A

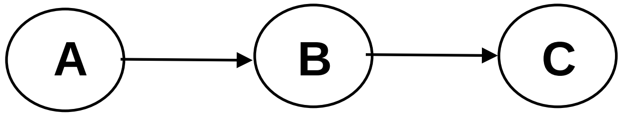
"Where there's Smoke, there's Fire."

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


$$\begin{tikzcd} A \arrow[r] & B \arrow[r] & C \end{tikzcd}$$

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

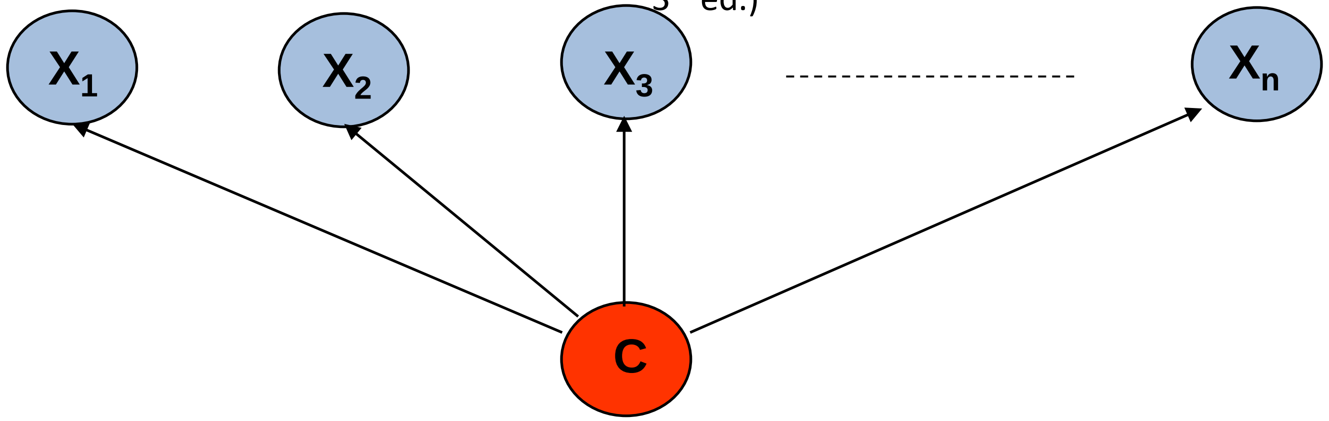
e.g.
If it rains today, it will rain tomorrow with 90%

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.)



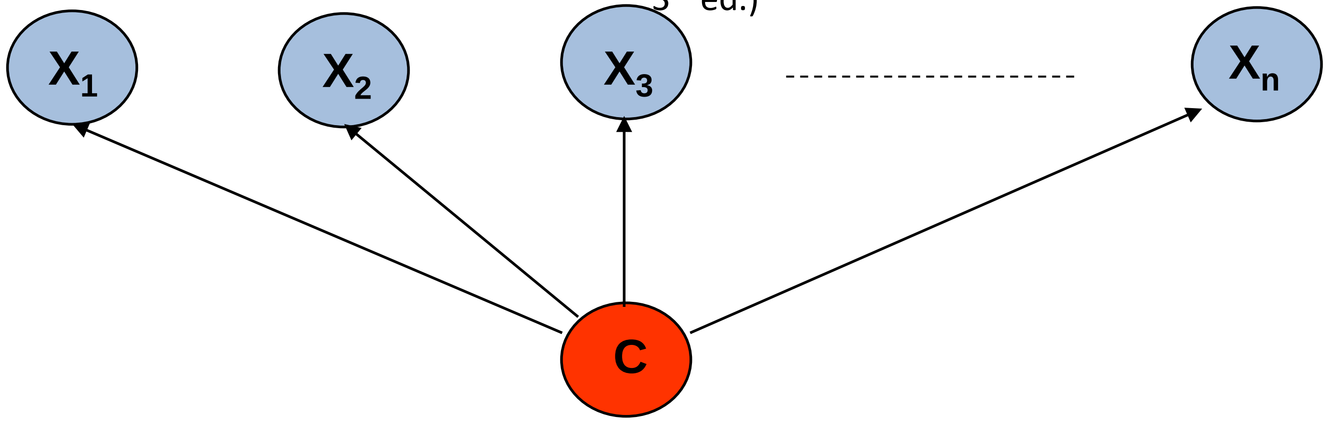
Basic Idea: We want to estimate $P(C \mid X_1,\ldots X_n)$, but it's hard to think about computing the probability of a class from input attributes of an example.

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

We estimate $P(C)$ easily from the frequency with which each class appears within our training data, and we estimate $P(X_i \mid C)$ 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}

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

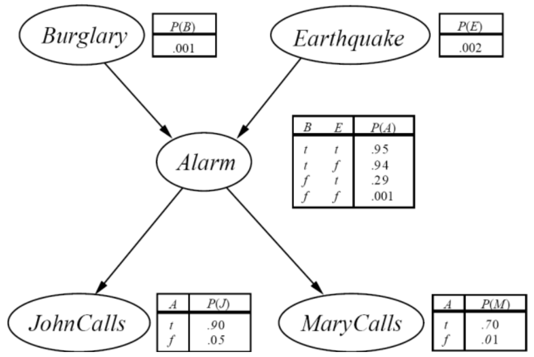
$$\approx P(J, M \mid A) \qquad P(A \mid E, B)\, P(E)\, P(B)$$

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

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

Constructing this Bayesian Network: Step 2

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

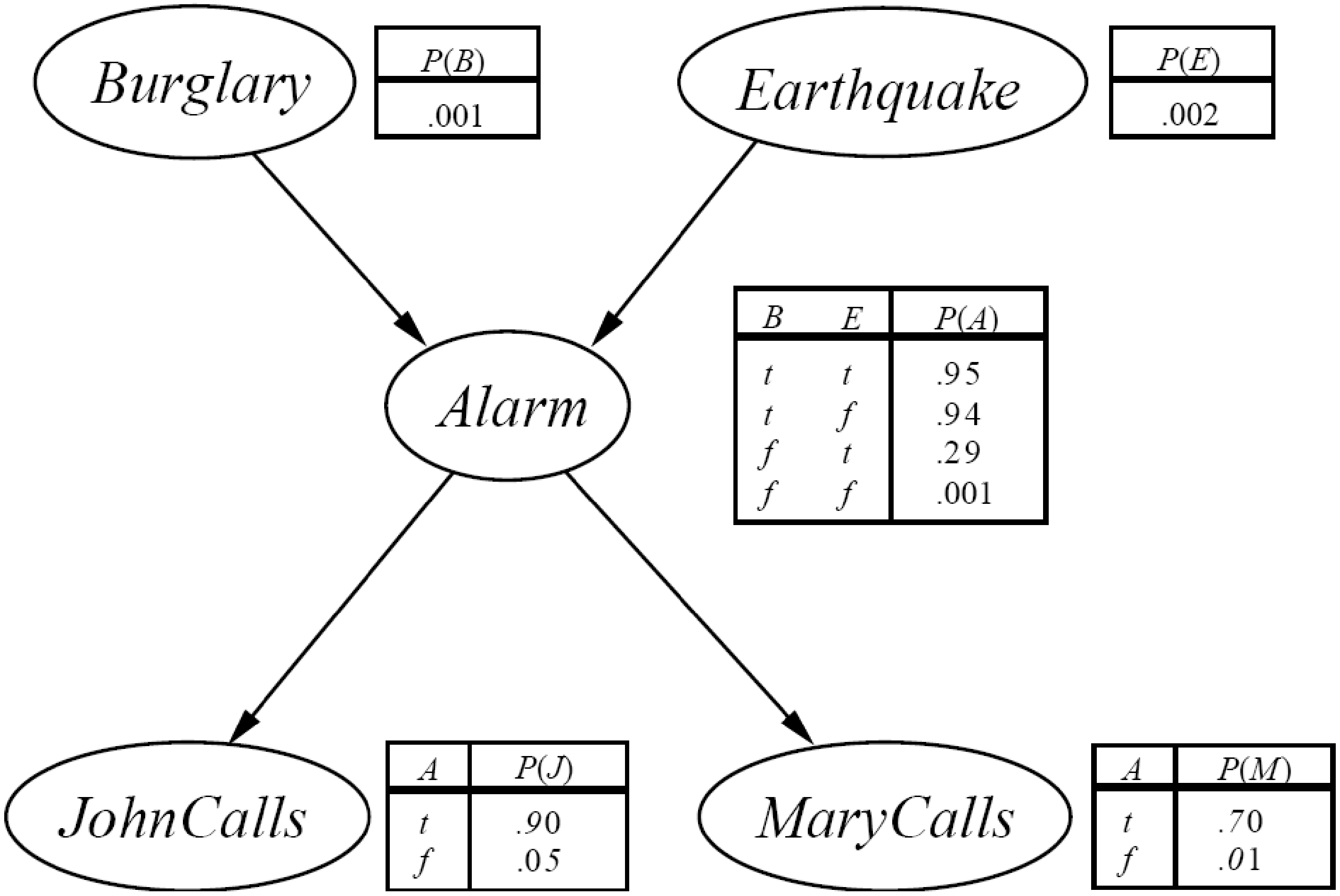


- There are 3 conditional probability tables (CPDs) to be determined:
  P(J | A), P(M | A), P(A | E, B)
  - Requiring 2 + 2 + 4 = 8 probabilities

- 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

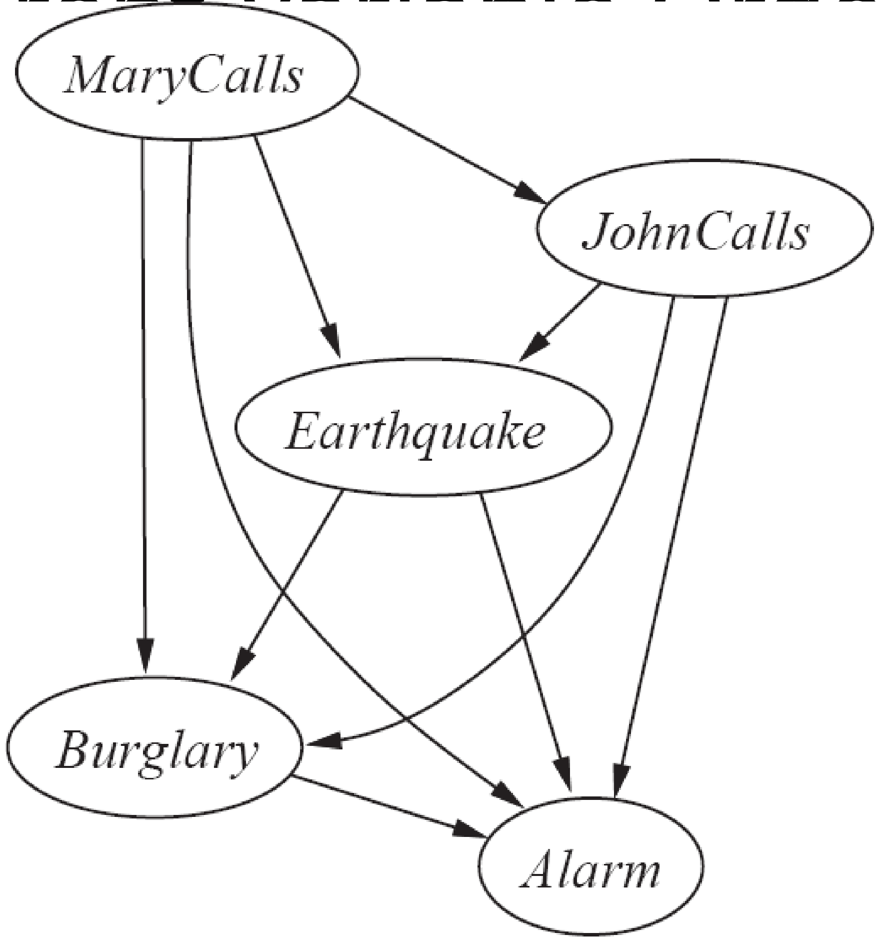
P(E)
.002

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

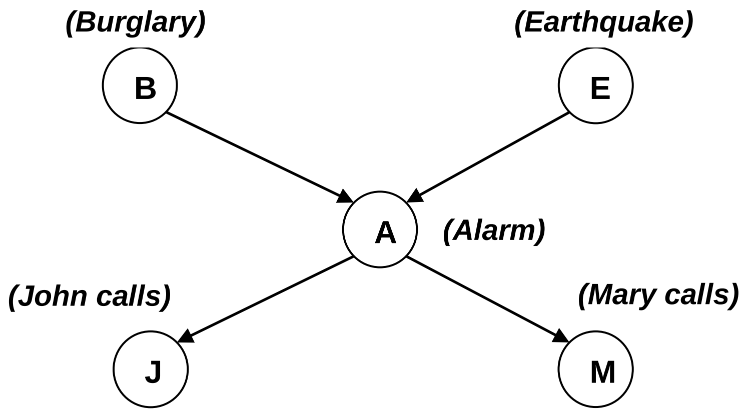


(b)

Computing Probabilities from a Bayesian Network

Shown below is the Bayesian network for the Burglar Alarm problem, i.e.,

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



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

Suppose we wish to compute P(J=f $\wedge$ M=t $\wedge$ A=t $\wedge$ B=t $\wedge$ E=f):

P(J=f $\wedge$ M=t $\wedge$ A=t $\wedge$ B=t $\wedge$ E=f)

= P(J=f | A=t) * P(M=t | A=t) * P(A=t | B=t $\wedge$ E=f) * P(B=t) * P(E=f)

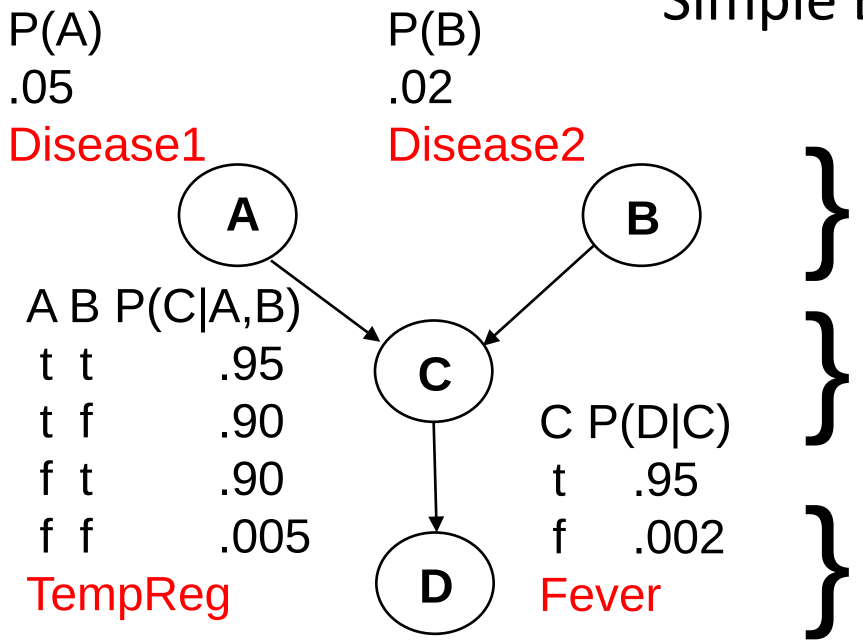
= .10 * .70 * .94 * .001 * .998

Note: P(E=f) = [1 – P(E=t)] = [1 – .002)] = .998

P(J=f | A=t) = [1 – P(J=t | A=t)] = .10

Inference in Bayesian Networks

Simple Example



A	B	P(C|A,B)
t	t	.95
t	f	.90
f	t	.90
f	f	.005

C	P(D|C)
t	.95
f	.002

Query Variables A, B

Hidden Variable C

Evidence Variable D

(A=True, B=False | D=True) : Probability of getting Disease1 when we observe Fever

Note: Not an anatomically correct model of how diseases cause fever!

Suppose that two different diseases influence some imaginary internal body temperature regulator, which in turn influences whether fever is present.

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?**
  - $\mathbf{P}(\mathbf{X} \mid \mathbf{e}) = \alpha \sum_{\mathbf{y}} \mathbf{P}(\mathbf{X}, \mathbf{y}, \mathbf{e})$

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

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

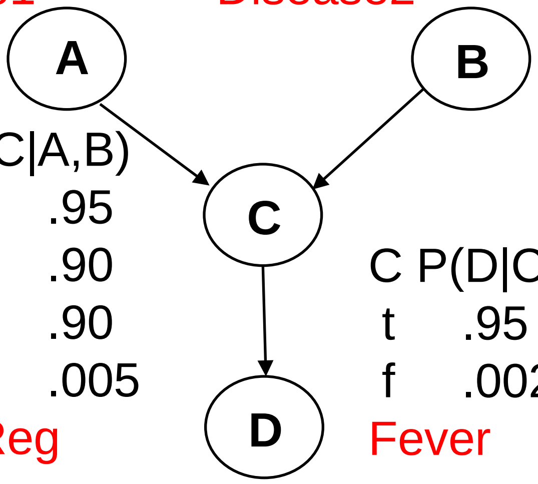
Inference by Variable Elimination

P(A)
.05

Disease1

P(B)
.02

Disease2



A	B	P(C|A,B)
t	t	.95
t	f	.90
f	t	.90
f	f	.005

TempReg

C	P(D|C)
t	.95
f	.002

Fever

What is the posterior conditional distribution of our query variables, given that fever was observed?

$$P(A,B|d) = \alpha\, \Sigma_c\ P(A,B,c,d)$$
$$= \alpha\, \Sigma_c\ P(A)P(B)P(c|A,B)P(d|c)$$
$$= \alpha\, P(A)P(B)\, \Sigma_c\ P(c|A,B)P(d|c)$$

$P(a,b|d) = \alpha\, P(a)P(b)\, \Sigma_c\ P(c|a,b)P(d|c) = \alpha\, P(a)P(b)\{ P(c|a,b)P(d|c)+P(\neg c|a,b)P(d|\neg c) \}$
$= \alpha\ .05 \text{x} .02 \text{x} \{.95 \text{x} .95+.05 \text{x} .002\} \approx \alpha\ .000903 \approx .014$

$P(\neg a,b|d) = \alpha\, P(\neg a)P(b)\, \Sigma_c\ P(c|\neg a,b)P(d|c) = \alpha\, P(\neg a)P(b)\{ P(c|\neg a,b)P(d|c)+P(\neg c|\neg a,b)P(d|\neg c) \}$
$= \alpha\ .95 \text{x} .02 \text{x} \{.90 \text{x} .95+.10 \text{x} .002\} \approx \alpha\ .0162 \approx .248$

$P(a,\neg b|d) = \alpha\, P(a)P(\neg b)\, \Sigma_c\ P(c|a,\neg b)P(d|c) = \alpha\, P(a)P(\neg b)\{ P(c|a,\neg b)P(d|c)+P(\neg c|a,\neg b)P(d|\neg c) \}$
$= \alpha\ .05 \text{x} .98 \text{x} \{.90 \text{x} .95+.10 \text{x} .002\} \approx \alpha\ .0419 \approx .642$

$P(\neg a,\neg b|d) = \alpha\, P(\neg a)P(\neg b)\, \Sigma_c\ P(c|\neg a,\neg b)P(d|c) = \alpha\, P(\neg a)P(\neg b)\{ P(c|\neg a,\neg b)P(d|c)+P(\neg c|\neg a,\neg b)P(d|\neg c) \}$
$= \alpha\ .95 \text{x} .98 \text{x} \{.005 \text{x} .95+.995 \text{x} .002\} \approx \alpha\ .00627 \approx .096$

$\alpha \approx 1 / (.000903+.0162+.0419+.00627) \approx 1 / .06527 \approx 15.32$ **[Note: α = normalization constant, p. 493]**

Mid-term Review
Chapters 2-5, 13, 14

- Review Agents (2.1-2.3)
- Review State Space Search
  - Problem Formulation (3.1, 3.3)
  - Blind (Uninformed) Search (3.4)
  - Heuristic Search (3.5)
  - Local Search (4.1, 4.2)
- Review Adversarial (Game) Search (5.1-5.4)
- Review Probability & Bayesian Networks (13, 14.1-14.5)

- Please review your quizzes and old CS-171 tests
  - At least one question from a prior quiz or old CS-171 test will appear on the mid-term (and all other tests)