Artificial Intelligence

CS482, CS682, MW 1 – 2:15, SEM 201, MS 227

Prerequisites: 302, 365

Instructor: Sushil Louis, sushil@cse.unr.edu, http://www.cse.unr.edu/~sushil

Questions

- Rational agents and performance metrics
  - Suppose that the performance measure is concerned with just the first T time steps of the environment and ignores everything thereafter. Show that a rational agent's action may depend not just on the state of the environment but also on the time step it has reached

Questions (True or False)

- An agent that senses only partial information about the state cannot be perfectly rational
- There exist task environments in which no pure reflex agent can behave rationally
- There exists a task environment in which every agent is rational
- The input to an agent program is the same as the input to the agent function
- Every agent function is implementable by some program/machine combination
- Suppose an agent selects its action uniformly at random from the set of possible actions. There exists a deterministic task environment in which this agent is rational

True or False

- It is possible for a given agent to be perfectly rational in two distinct task environments
- Every agent is rational in an unobservable environment
- A perfectly rational poker-playing agent never loses

Types of task environments

Task Env	Observable	Agents	Deterministic	Episodic	Static	Discrete
Soccer						
Exploring the subsurface oceans of Titan						
Shopping for used AI books on the net						
Playing a tennis match						
Practicing tennis against a wall						
Performing a high jump						
Knitting a sweater						
Bidding on an item at anauction						

Types of task environments

Task Env	Observable	Agents	Deterministic	Episodic	Static	Discrete
Soccer	Partial	Multi	Stochastic	Sequential	Dynamic	Continuous
Exploring the subsurface oceans of Titan	Partial	Single?	Stochastic	Sequential	Dynamic	Continuous
Shopping for used AI books on the net	Partial	Single ?	Deterministic	Sequential	Static	Discrete
Playing a tennis match	Fully	Multi	Stochastic	Episodic/Seq	Dynamic	Continuous
Practicing tennis against a wall	Fully	Single	Stochastic	Episodic/seq	Dynamic	Continuous
Performing a high jump	Fully	Single	Stochastic	Sequential	Static	Continuous
Knitting a sweater	Fully	Single	Deterministic	Sequential	Static	Continuous
Bidding on an item at an auction	Fully	Multi	Stochastic/ Strategic	Sequential	Static	Discrete

Quotes

MURPHY'S LAWS

1. Nothing is as easy as it looks.
2. Everything takes longer than you think.
3. Anything that can go wrong will go wrong.
4. If there is a possibility of several things going wrong, the one that will cause the most damage will be the one to go wrong. Corollary: If there is a worse time for something to go wrong, it will happen then.
5. If anything simply cannot go wrong, it will anyway.
6. If you perceive that there are four possible ways in which a procedure can go wrong, and circumvent these, then a fifth way, unprepared for, will promptly develop.
7. Every solution breeds new problems.

The Murphy Philosophy

Smile . . . tomorrow will be worse.

Arthur C. Clarke

- Any sufficiently advanced technology is indistinguishable from magic.

Outline

- Problem solving agents
- Problem types
- Problem formulation
- Example Problems
- Basic Search Algorithms

Problem Solving Agents

- Restricted form of general agent

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
   persistent: seq, an action sequence, initially empty
               state, some description of the current world state
               goal, a goal, initially null
               problem, a problem formulation

   state ← UPDATE-STATE(state, percept)
   if seq is empty then
       goal ← FORMULATE-GOAL(state)
       problem ← FORMULATE-PROBLEM(state, goal)
       seq ← SEARCH(problem)
       if seq = failure then return a null action
   action ← FIRST(seq)
   seq ← REST(seq)
   return action
```

Figure 3.1 A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

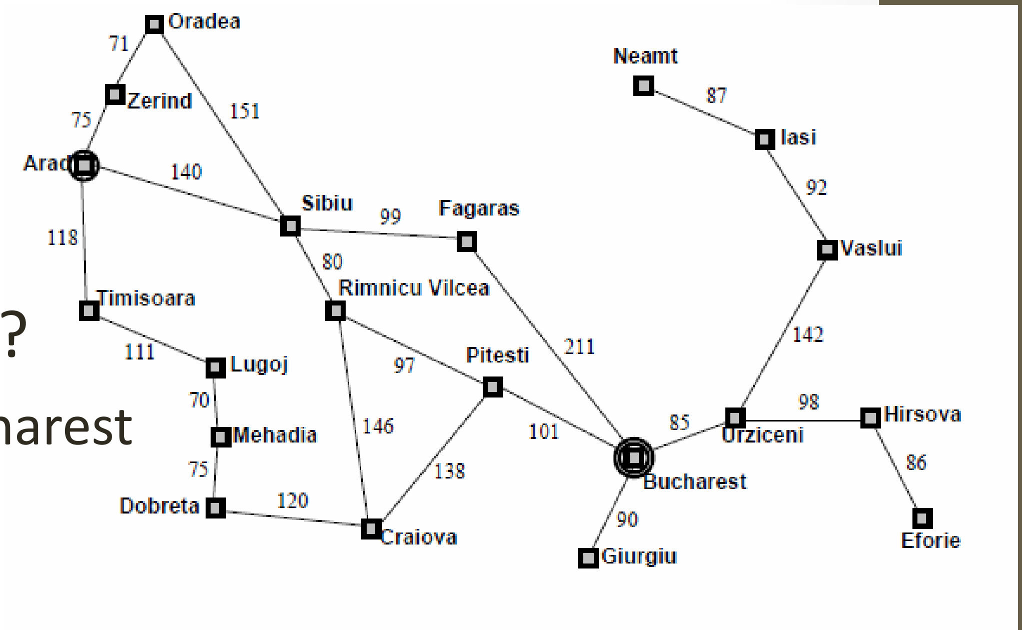
This is **offline** problem solving. Search for solution, then execute. During execution we are not using subsequent percepts

Problem solving agent example

- Consider a holiday in Romantic Romania
  - You are an agent, holiday touring in Arad, Romania
  - What are your performance measures?
    - Improve suntan, look at the sights, check out Transylvania, enjoy the nightlife, become one of the undead, avoid hangovers, …
    - The action sequence to do this is long and complicated and you need to read guidebooks, books, talk to people, make tradeoffs
    - Very complex, let us simplify
  - You have a non-refundable ticket to get home from Bucharest tomorrow
  - Now you have a goal: Get to Bucharest in time to catch your flight tomorrow

Romantic Romania

- Goal: Get to Bucharest
- Formulate Problem:
  - States: Cities
  - Actions: Drive to city
- What level of abstraction?
  - Turn wheel or Drive to Bucharest
  - What is a state?
  - What is an action?



- Goal: Set of states, specifically: {Bucharest}
- Solution: Sequence of actions that results in a goal state

What type of task environment?

Task Env	Observable	Agents	Deterministic	Episodic	Static	Discrete
Romantic Romania						

Task Env	Observable	Agents	Deterministic	Episodic	Static	Discrete
Romantic Romania	Yes	Single	Yes	Sequential	Static	Discrete

Problem solution

- A fixed sequence of actions
- Agent **searches** for a sequence of actions that will lead to a goal state
- So we :
  - **Formulate** the problem,
  - **Search** for a solution,
  - **Execute** the action sequence

- Execution phase does NOT consider percepts in this simple example. In control theory: **Open-Loop** system

Back to Romanian problem formulation

- **Initial State**, S_0
  - In(Arad)
- **Actions**
  - Actions(S) returns set of actions possible in state S
  - {Go(Sibiu), Go(Timisoara), Go(Zerind)}
- **Transition Model**: What does an action do?
  - Result (In(Arad), Go(Zerind)) = In(Zerind)
- **State space** is a **directed graph**
- A **Path** in the state space is a sequence of states connected by a sequence of actions
- **Goal State(s)** → In(Bucharest)
- **Path COST** function
  - Some agents are better than others → lower cost
  - Path costs are non-negative (>= 0)

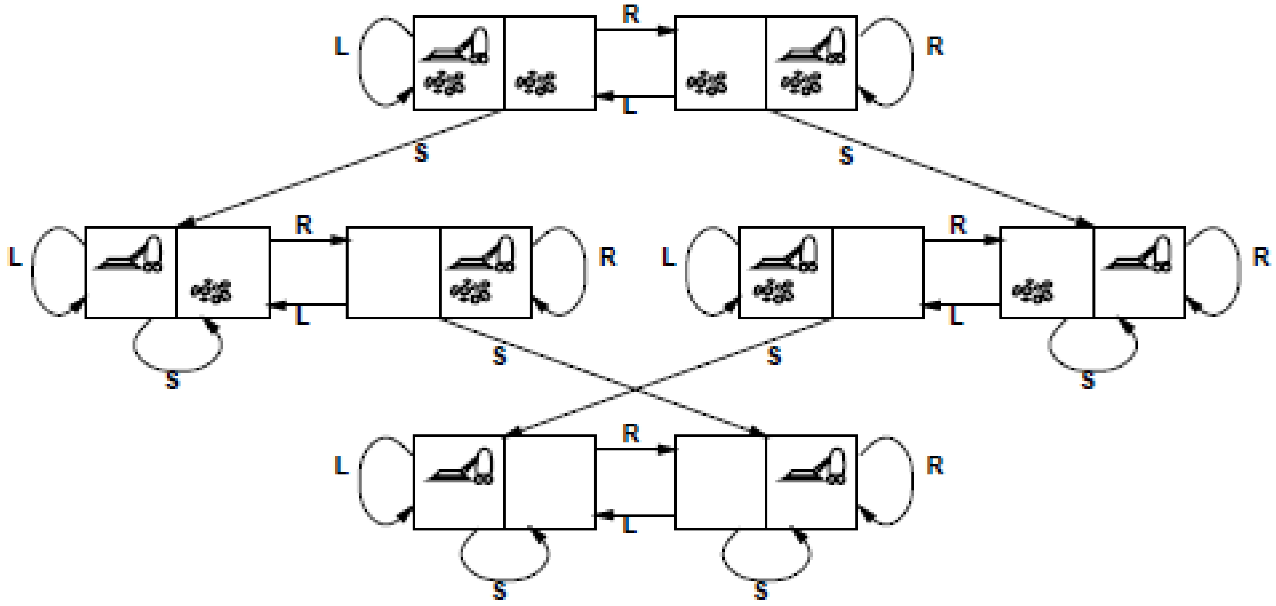
State Space of our problem

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

Abstraction

- The real world is absurdly complex so state space must be abstracted for problem solving
- In(Arad) means somewhere in Arad but where
- Result(In (Arad), Go(Zerind)) = In (Zerind). Yay but how do you find the highway out and what side do you drive on and where's the gas station, and …..
  - In a more expressive, less abstract representation of the world, In(Arad) must correspond to some real location in Arad (Hotel Phoenix perhaps)
- Similarly a solution, a sequence of actions, must correspond to real actions in the less abstract real-world. A Solution Path must correspond to a real path
- Our abstraction should make the original problem easier while at the same time enabling a correspondence with a more expressive representation

Vacuum world. States and transitions

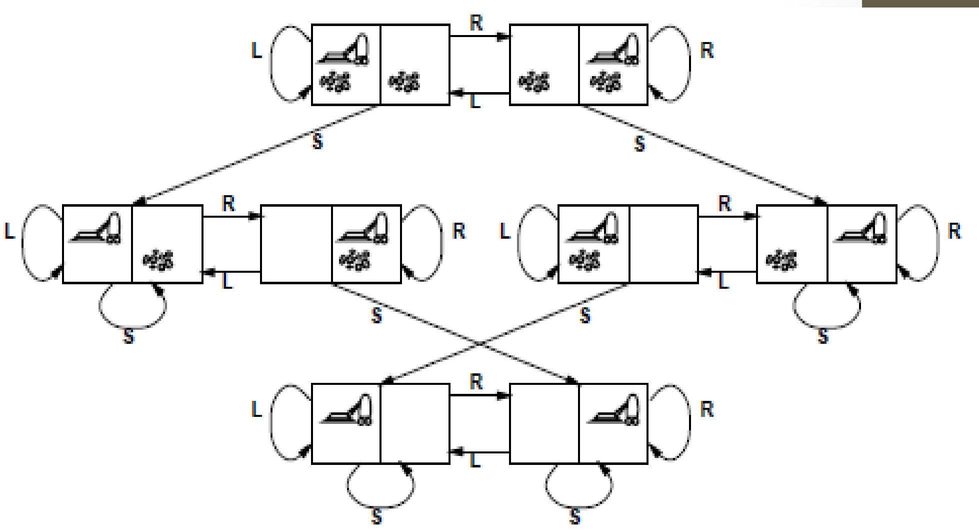


Vacuum world

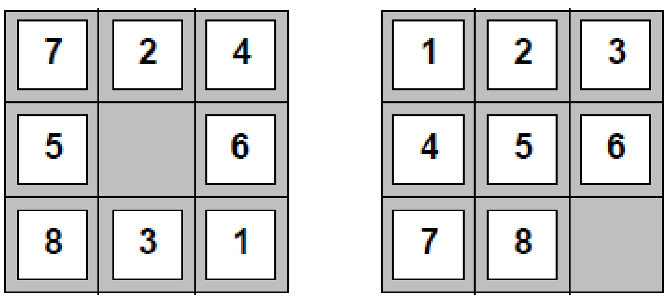
- States
  - ?
- Actions
  - ?
- Transition model (see figure)
- Goal test
  - ?
- Path cost
  - ?

Vacuum world

- States
  - Dirt location (0, 1), Robot location (0, 1)
  - Initial state can be any state →
- Actions
  - Left, Right, Suck, NoOp
- Transition model →
- Goal test
  - No Dirt. All squares are clean
- Path cost
  - 1 per action, 0 for NoOp



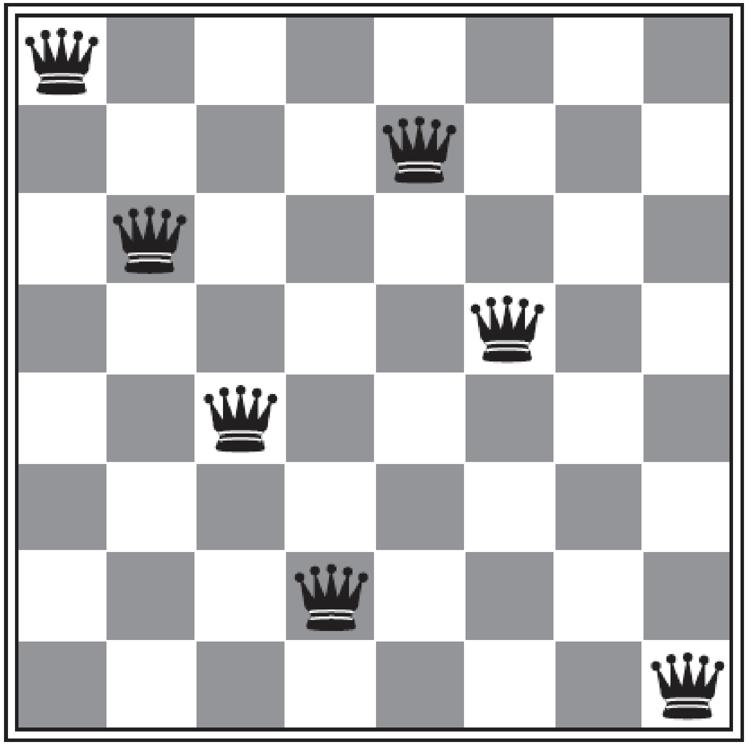
8 puzzle



Start State Goal State

- States
  - Location of every tile and blank
- Initial state
  - Any state
- Actions
  - Movement of blank
    - Up, down, left, right
- Transition model
  - New state after blank move
- Goal Test
  - Test if configuration matches figure
- Path cost
  - 1 per blank move

8 Queens



- States
  - ?
- Initial State
  - ?
- Actions
  - ?
- Transition model
  - ?
- Goal Test
  - ?
- Path cost
  - ?

Real world problems

- Route finding
- TSP
- VLSI
- Robot Navigation
- Automatic assembly sequencing

Solving Romania

function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set

Figure 3.7 An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.

Romanian problem formulation

- **Initial State**, S_0
  - In(Arad)
- **Actions**
  - Actions(S) returns set of actions possible in state S
  - {Go(Sibiu), Go(Timisoara), Go(Zerind)}
- **Transition Model**: What does an action do?
  - Result (In(Arad), Go(Zerind)) = In(Zerind)
- **State space** is a **directed graph**
- A **Path** in the state space is a sequence of states connected by a sequence of actions
- **Goal State(s)** → In(Bucharest)
- **Path COST** function
  - Some agents are better than others → lower cost
  - Path costs are non-negative (>= 0)

State Space of our problem

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

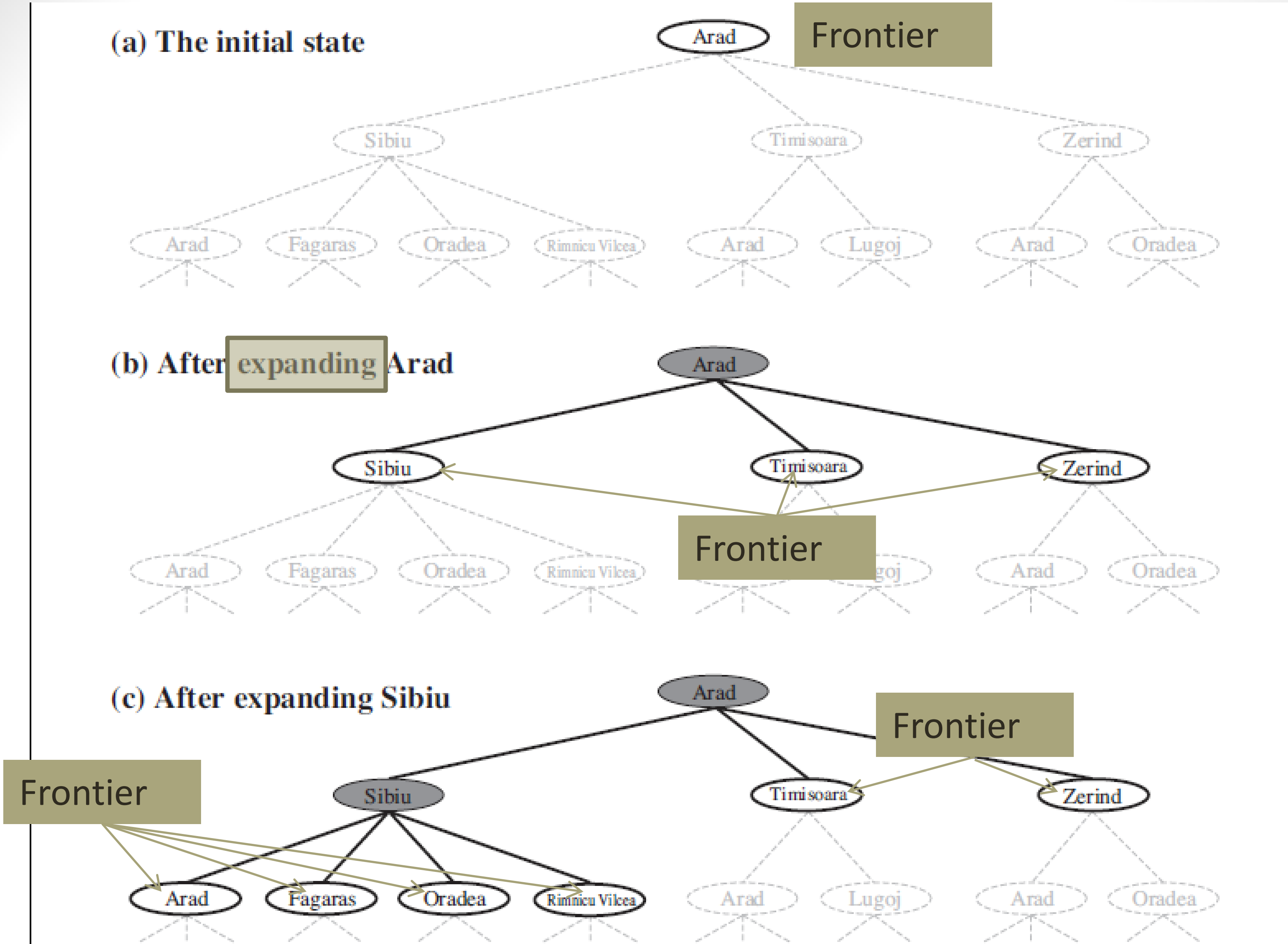


Figure 3.6 **FILES: figures/search-map.eps (Tue Nov 3 16:23:38 2009).** Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

Arad is loopy

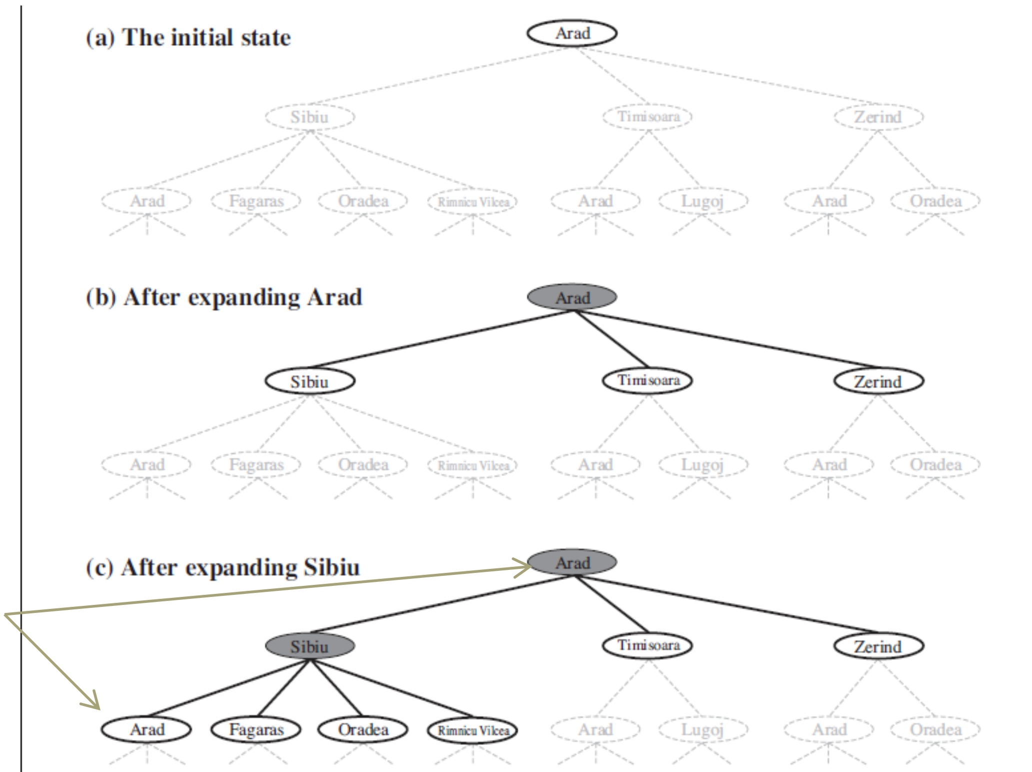


Figure 3.6 **FILES: figures/search-map.eps (Tue Nov 3 16:23:38 2009).** Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

Why should we ignore loopy (redundant) paths?

1. DynProg
2. PathCost

Should we always ignore redundant paths?

Graph search avoids redundant paths

And, very importantly, getting rid of redundant paths reduces the number of tree nodes from pow(b, d) to approximately 2 d^2 !!!!!
b = branching factor
d = tree depth

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set

Figure 3.7 An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.

Graph search makes a state tree

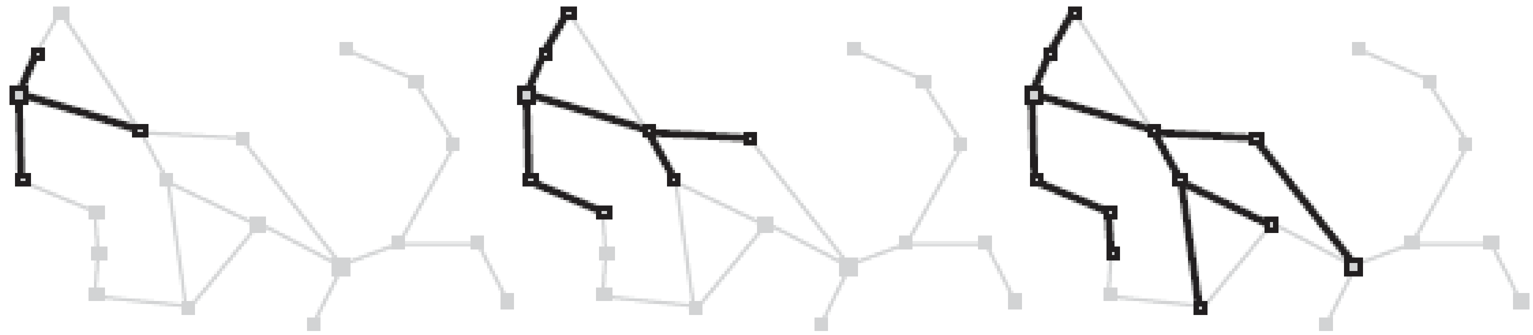
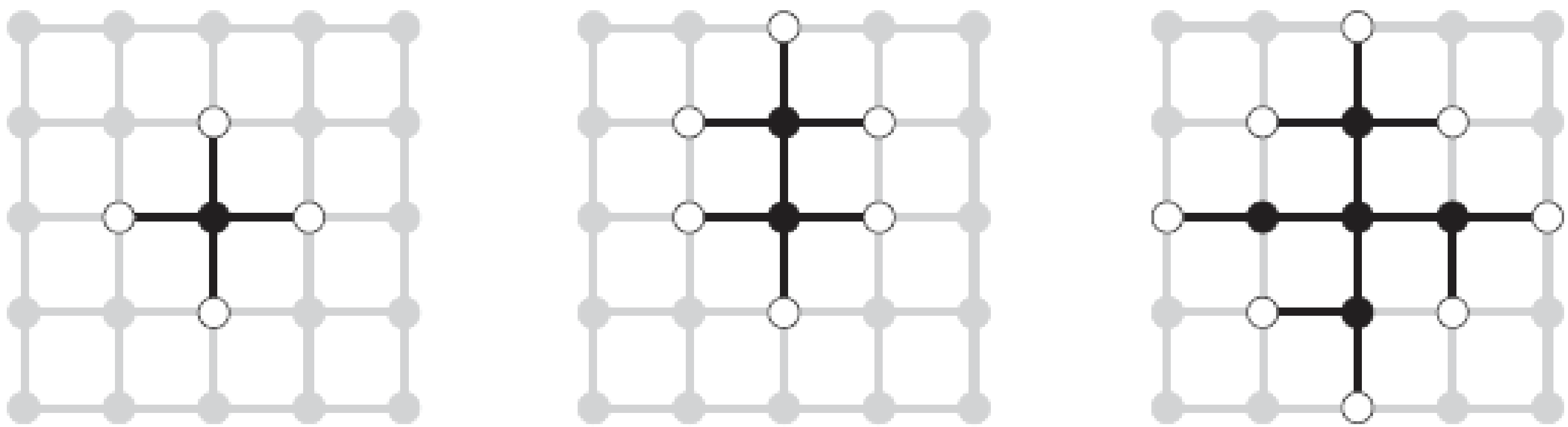


Figure 3.8 **FILES: figures/romania-graph-search.eps (Tue Nov 3 13:48:17 2009).** A sequence of search trees generated by a graph search on the Romania problem of Figure 3.2. At each stage, we have extended each path by one step. Notice that at the third stage, the northernmost city (Oradea) has become a dead end: both of its successors are already explored via other paths.

Graph search frontier separates explored and unexplored states



(a) (b) (c)

Implementing graph search

- Node != problem state (states do not have parent, action, path-cost, …)
  - Parent
  - Action
  - State
  - Path-cost
- function ChildNode(*problem*, *parent*, *action*) returns Node
  - return a Node with
    - State = *problem.*Result(parent.State, action)
    - Parent = *parent*
    - Action = *action*
    - Path-cost = *parent.*Path-cost + *problem.*Step-cost(*parent.*State, *action)*
- If node contains goal state, then you have to construct the solution – a path – by following the parent chain to the root

Implementing graph search

- Frontier:
  - Queue
    - FIFO
    - LIFO
    - Priority
      - Path-Cost?
- Explored-Set:
  - Hash table

Ready for Search

- Different search strategies are defined by the order in which we choose nodes from the frontier to expand
  - Lifo, fifo, …
- We compare search strategies along the following dimensions
  - Completeness: Does it always find a solution if one exists?
  - Time Complexity: Number of nodes expanded/generated
  - Space Complexity: Max number of nodes in Memory
  - Optimality: Does it always find least-cost solution
- Time and space complexity are measured in terms of
  - b → maximum branching factor of search tree
  - d → depth of least cost solution
  - m → maximum depth of the tree (may be infinite!)

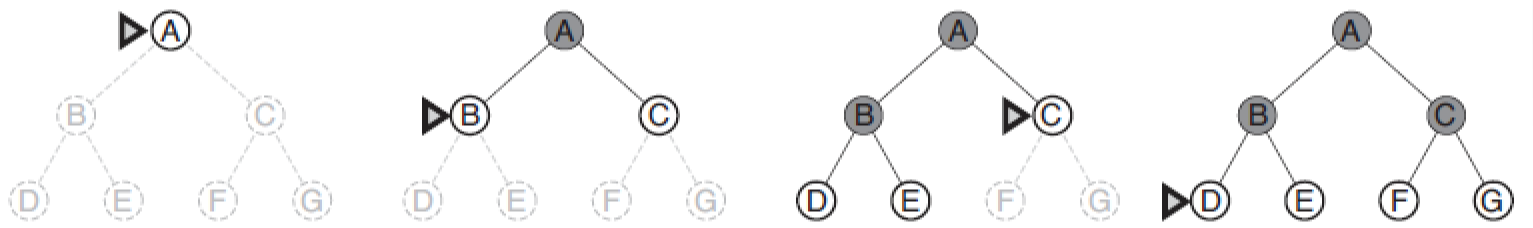
Uninformed Search

- Breadth-first
- Uniform-cost
- Depth-first
- Depth-limited
- Iterative deepening

Breadth-first search – FIFO Q

```
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)
```

BFS



- **Complete**: Yes – shallowest goal node
- **Time** == Number of nodes expanded – assume b constant
  - O(b^d) if you check for goal state upon generation of node or
  - O(b^(d+1)) if you check when you pick node for expansion
- **Space** == Space for nodes = number of nodes in explored set + number of nodes in frontier
  - O(b^(d-1)) in explored + O(b^d) in frontier
  - Uh-oh! Can generate nodes at the rate of 100MB/sec so 24 hours means 8640GB
  - Look at figure 3.13 in the book
  - With b = 10, d = 16, and 1M nodes/sec, 350 Years and 10 exabytes of storage needed
- **Optimality**: Optimal if path cost is non-decreasing function of depth

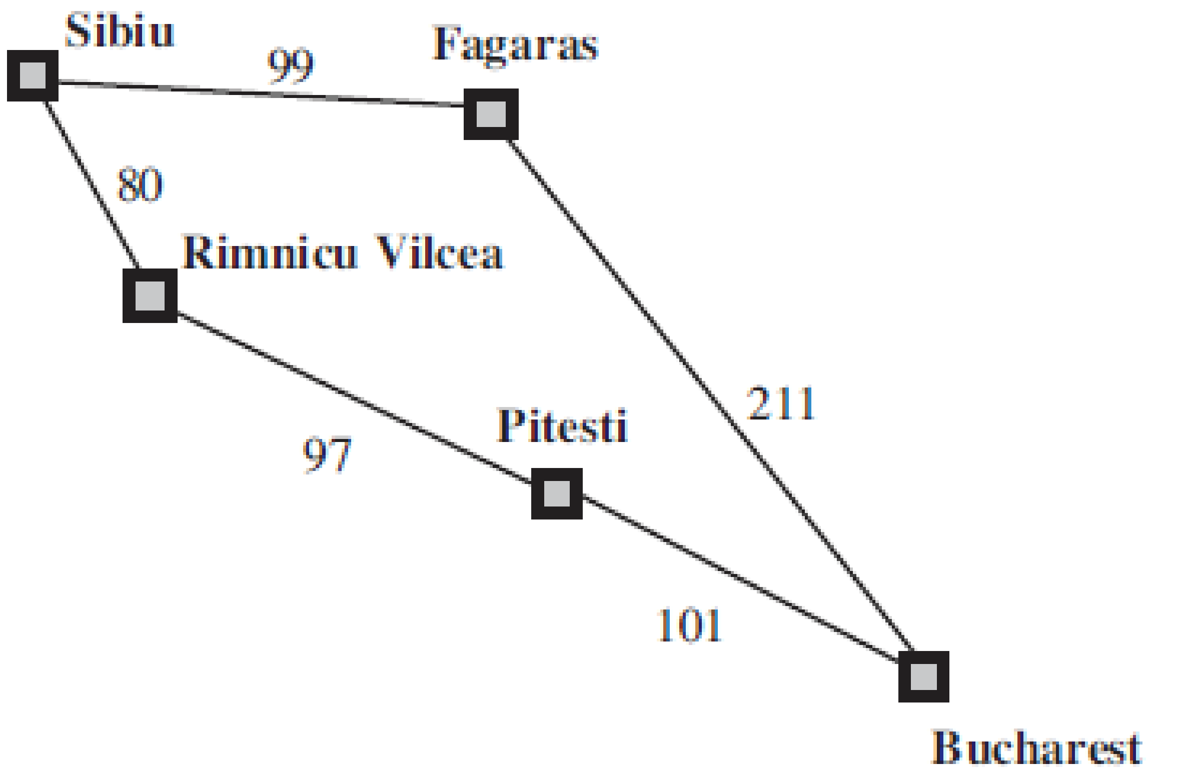
Uniform-cost search

- Expand node with lowest path-cost
- Goal test on expansion
- Replace frontier node if you find better path to same node.State

```
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
```

Uniform cost search

- Draw the Uniform-cost search tree for getting from Sibiu to Bucharest

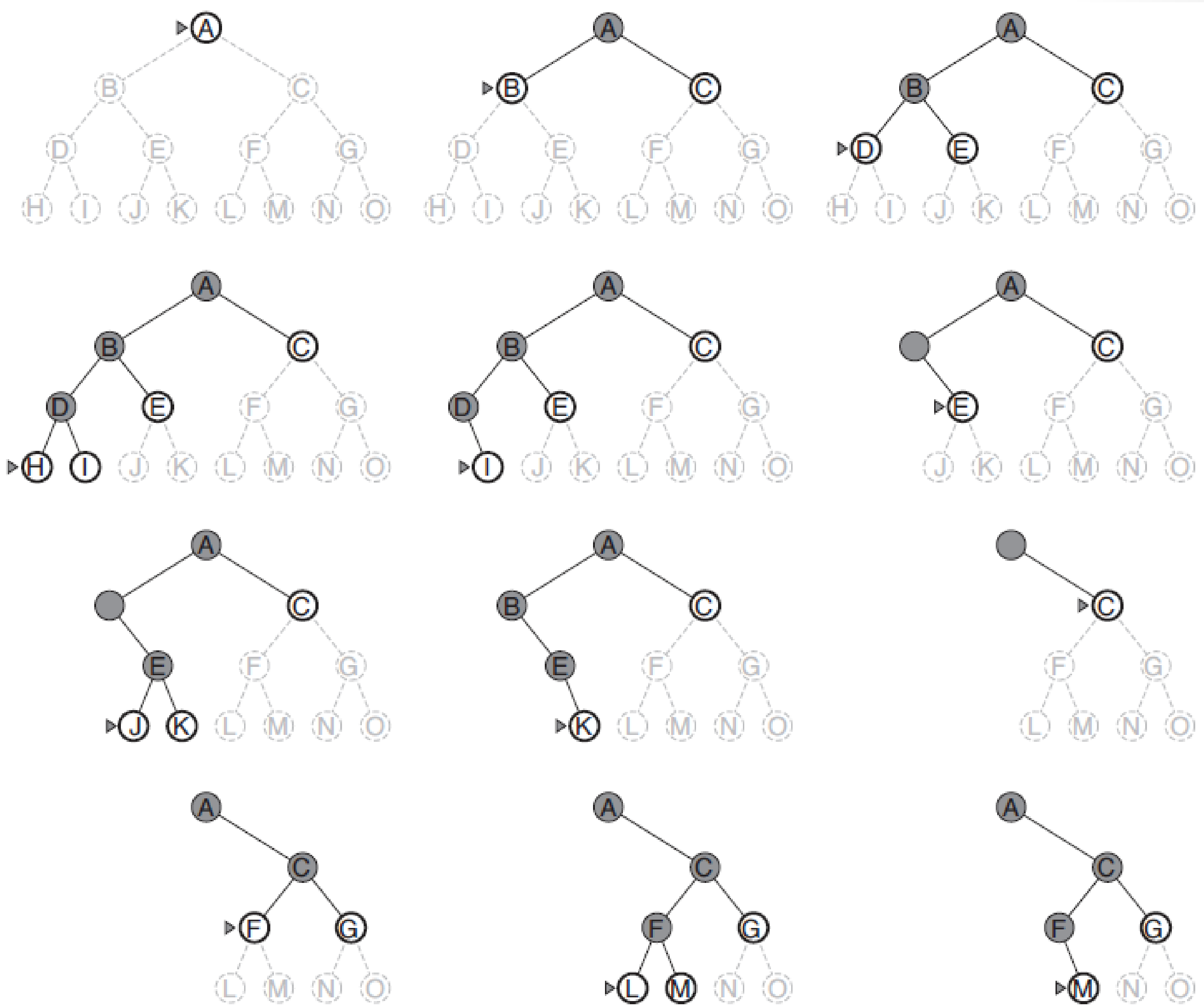


Uniform cost search

- Complete if every step cost is > 0
- Optimal
- Time/Space – Strictly more than BFS

Depth-first search

- LIFO Q



DFS

- Often easy to implement recursively
- Completeness:
  - Graph search version is complete in finite spaces
  - Tree search version can be infinitely loopy
- Not-optimal
- Time: If d is depth of shallowest optimal solution, and m is max depth of tree, DFS may generate O(b^m) >> O(b^d)
- Space: O(bm) ! Not bad and we can go lower to O(m) with some fancy housekeeping (backtracking search)
- Some kind of DFS used a lot in AI because space requirements are low
- What kinds?

Depth-limited search

- DFS with depth limit, l (el)
  - If l < d you will never find solution (incomplete)
  - If l > d non-optimal
  - DFS = DLS with l = infinity
- Romanian problem depth is 20 == number of states
  - Actually 9! The diameter of the state space (max steps between any pair of states)

DLS (or DFS)

- Remove limit to make DFS

function DEPTH-LIMITED-SEARCH(*problem*, *limit*) **returns** a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(*problem*.INITIAL-STATE), *problem*, *limit*)

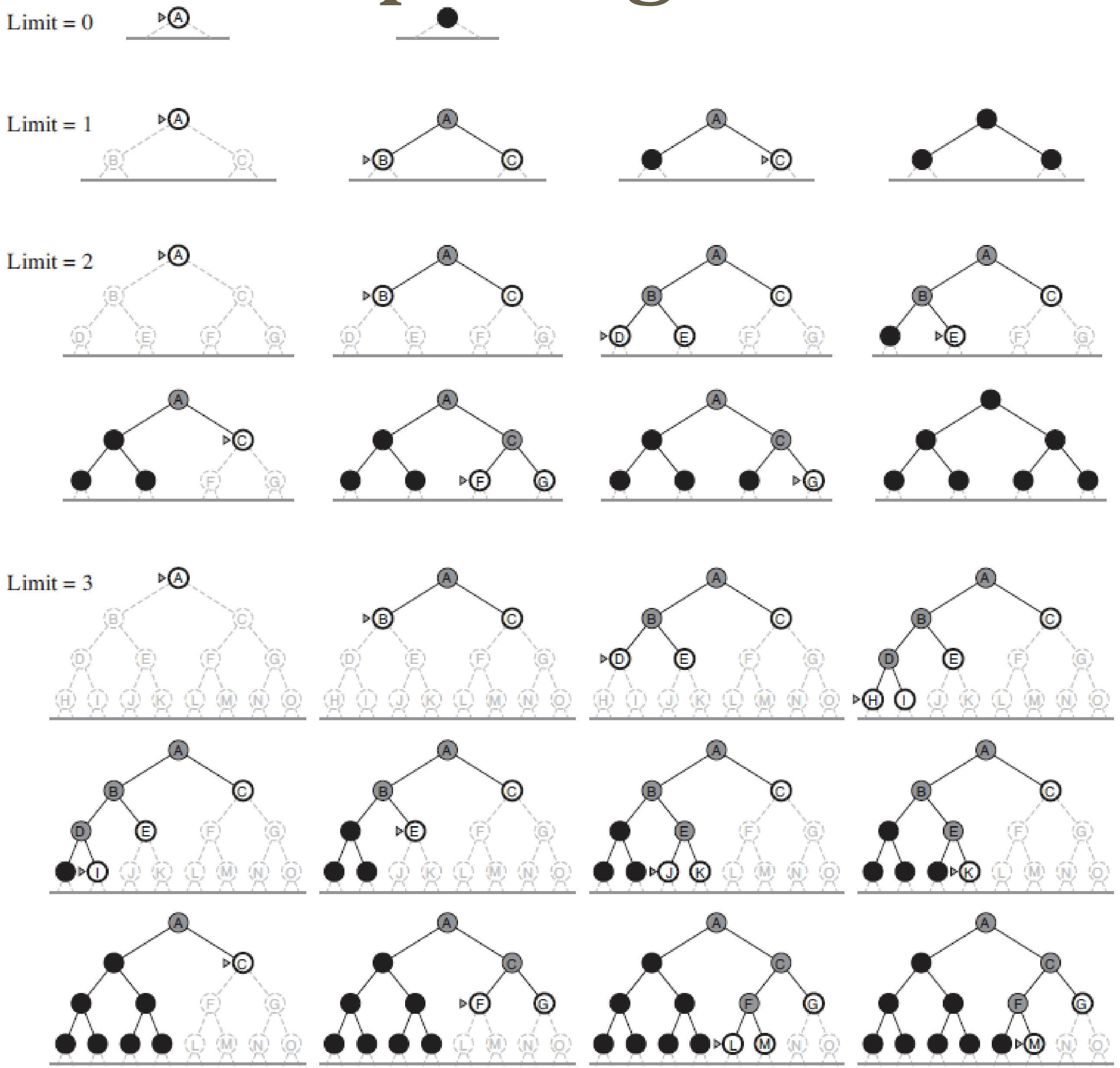
function RECURSIVE-DLS(*node*, *problem*, *limit*) **returns** a solution, or failure/cutoff
  if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)
  else if *limit* = 0 **then return** *cutoff*
  else
    cutoff_occurred? ← false
    for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**
      child ← CHILD-NODE(*problem*, *node*, *action*)
      result ← RECURSIVE-DLS(*child*, *problem*, *limit* − 1)
      if *result* = *cutoff* **then** *cutoff_occurred?* ← true
      else if *result* ≠ *failure* **then return** *result*
    if *cutoff_occurred?* **then return** *cutoff* **else return** *failure*

Iterative deepening DFS

- DLS but keep increasing limit
- Why?
  - Space efficient like DFS and
  - complete and optimal like BFS
  - Not much extra work since the number of nodes at depth d is b^d
    - And number of interior nodes = b^d -1
    - Most nodes are leaves
- Numerical comparison for b = 10 and d = 5, solution at far right leaf:
- N(IDS) = 50 + 400 + 3; 000 + 20; 000 + 100; 000 = 123; 450
- N(BFS) = 10 + 100 + 1; 000 + 10; 000 + 100; 000 + 999; 990 = 1; 111; 100

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure
  for *depth* = 0 **to** ∞ **do**
    result ← DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    if *result* $\neq$ cutoff **then return** *result*

Iterative deepening

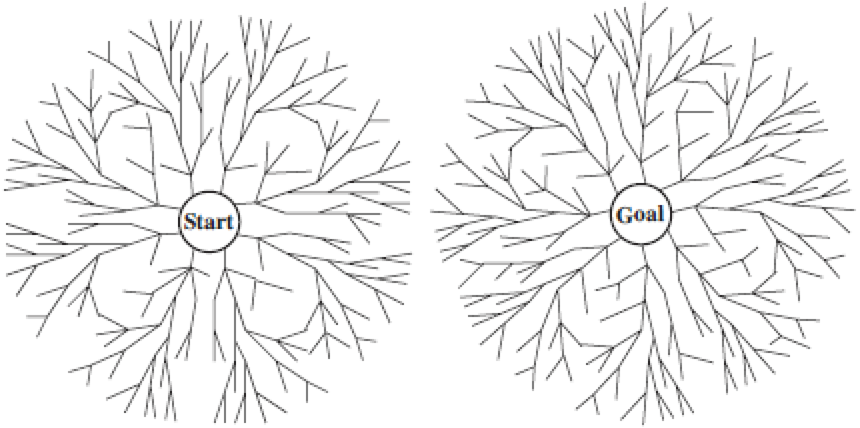


Iterative lengthening

- Check textbook

Bidirectional Search

- b^(d/2) + b^(d/2) << b^d
- Search “forwards” from start and “backwords” from goal
- Check for frontier intersection
- One search must be BFS for good check on frontier intersection
- How do you search backwards for
  - Romania
  - Vacuum cleaner
  - 8-queens



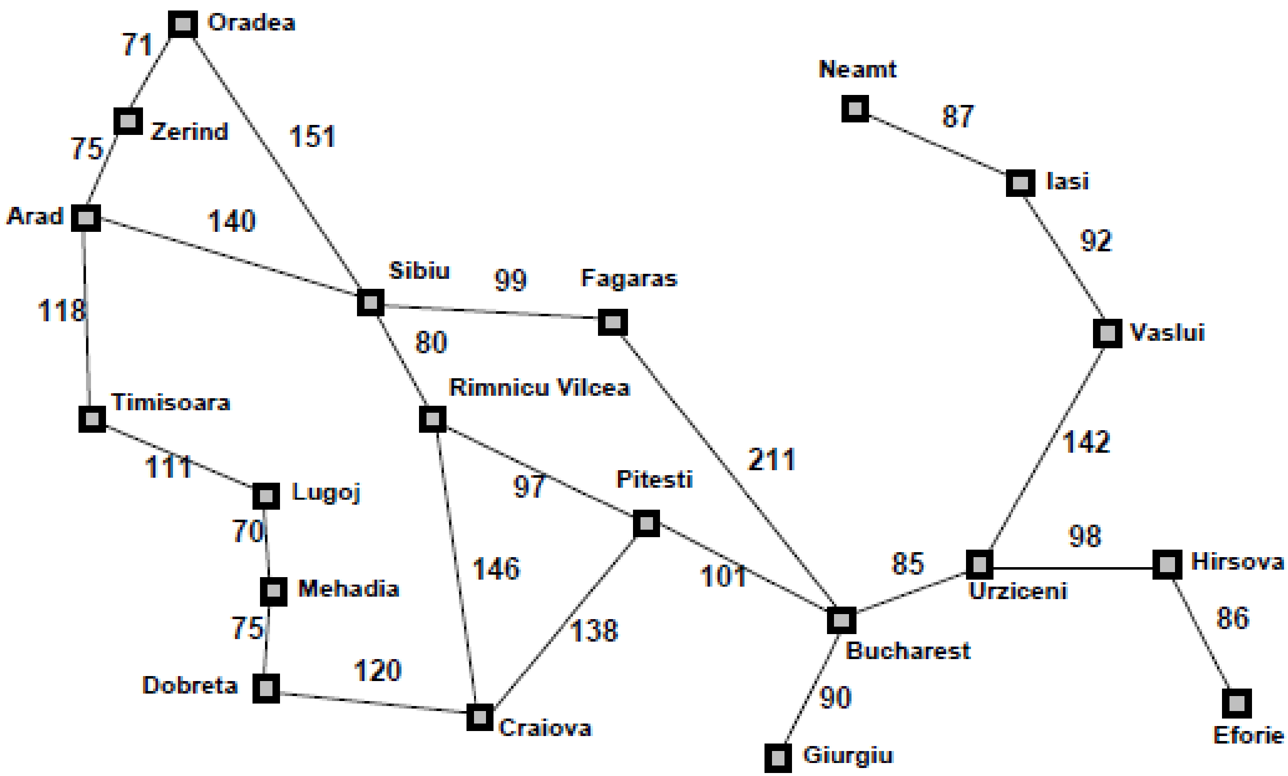
Comparison of uninformed search

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*

Informed Search

- **Best First Search**
  - A*
  - Heuristics
- Basic idea
  - Order nodes for expansion using a specific search strategy
    - Remember uniform cost search?
      - Nodes ordered by path length = path cost and we expand least cost
      - This function was called g(n)
  - Order nodes, n, using an evaluation function f(n)
  - Most evaluation functions include a heuristic h(n)
    - For example: **Estimated** cost of the cheapest path from the state at node n to a goal state
    - Heuristics provide domain information to guide informed search

Romania with straight line distance heuristic



Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

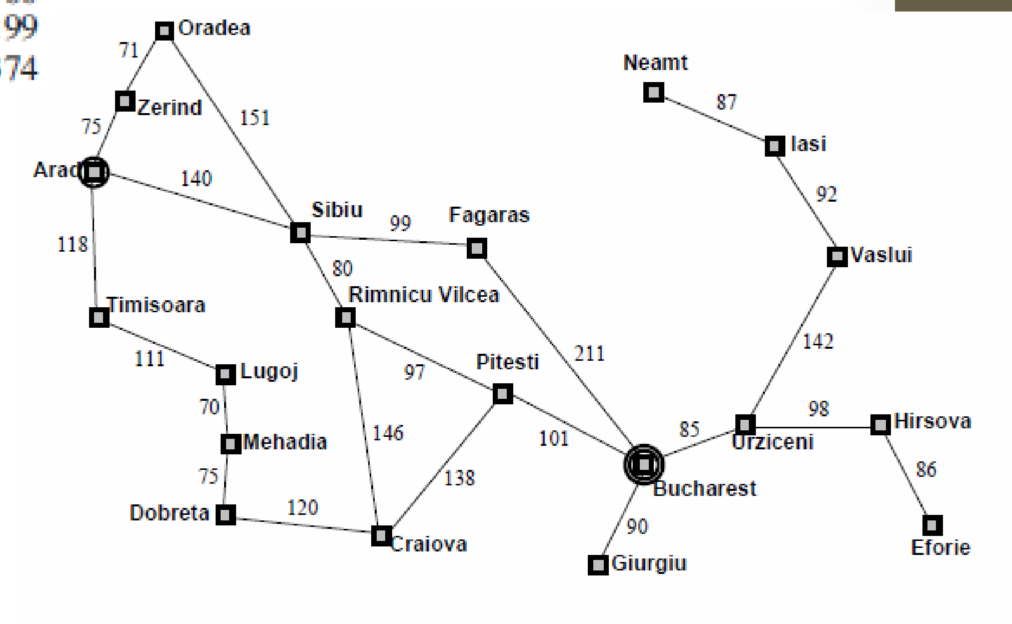
h(n) = straight line distance to Bucharest

Greedy search

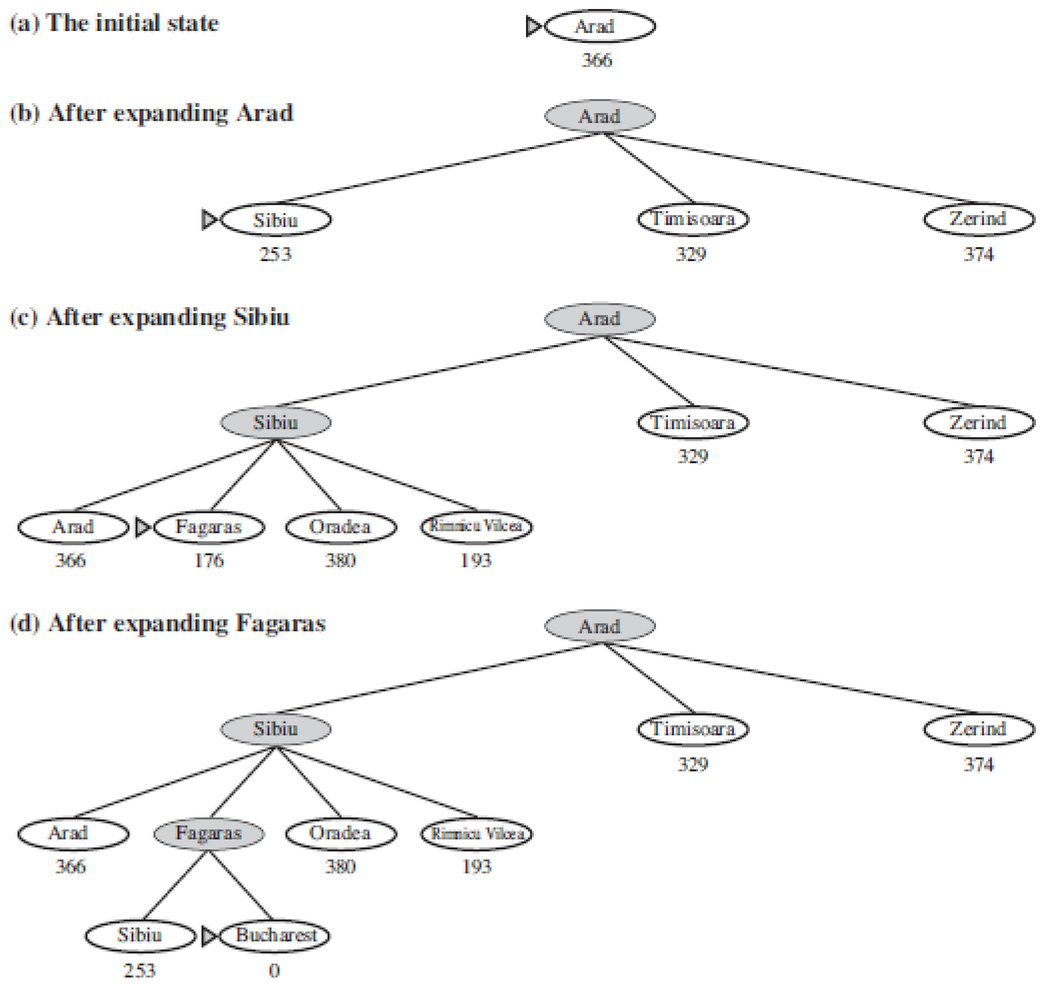
- F(n) = h(n) = straight line distance to goal
- Draw the search tree and list nodes in order of expansion (5 minutes)

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

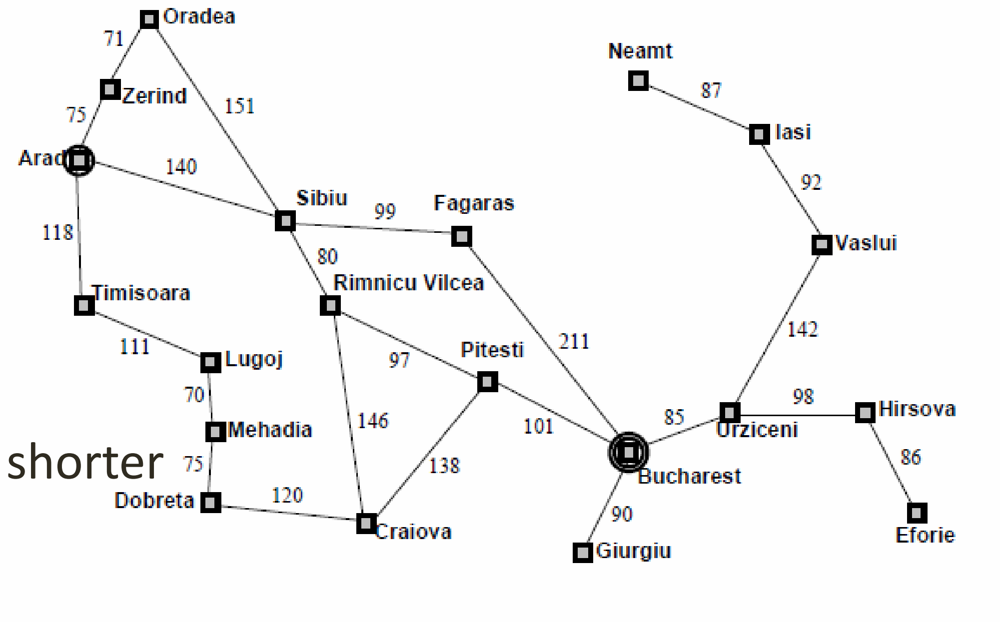
Time?
Space?
Complete?
Optimal?



Greedy search



Greedy analayis



- Optimal?
  - Path through Rimniu Velcea is shorter
- Complete?
  - Consider Iasi to Fagaras
  - Tree search no, but graph search with no repeated states version → yes
    - In finite spaces
- Time and Space
  - Worst case b^m where m is the maximum depth of the search space
  - Good heuristic can reduce complexity

A^*

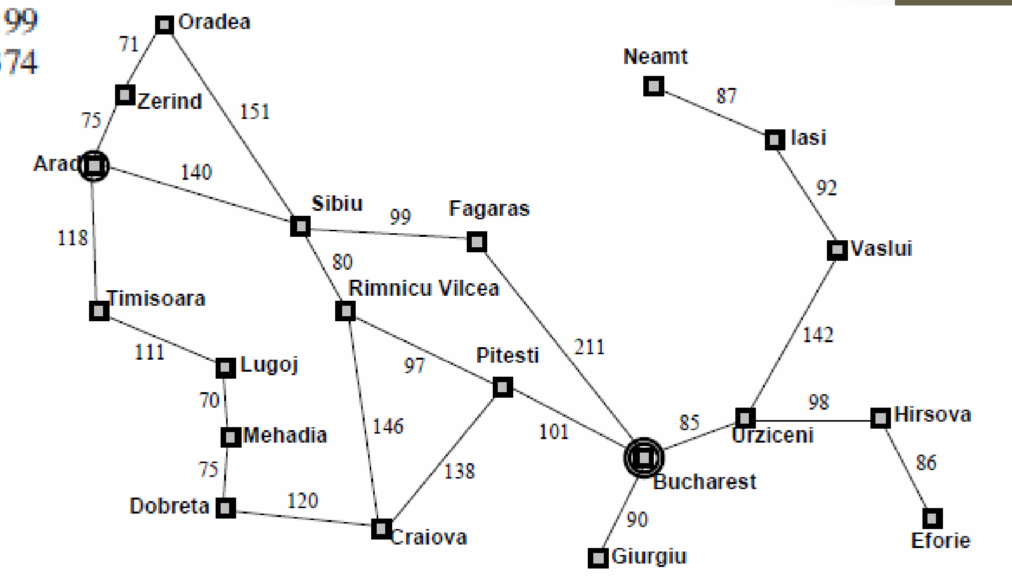
- f(n) = g(n) + h(n)
- = cost to state + estimated cost to goal
- = estimated cost of cheapest solution through *n*

A^*

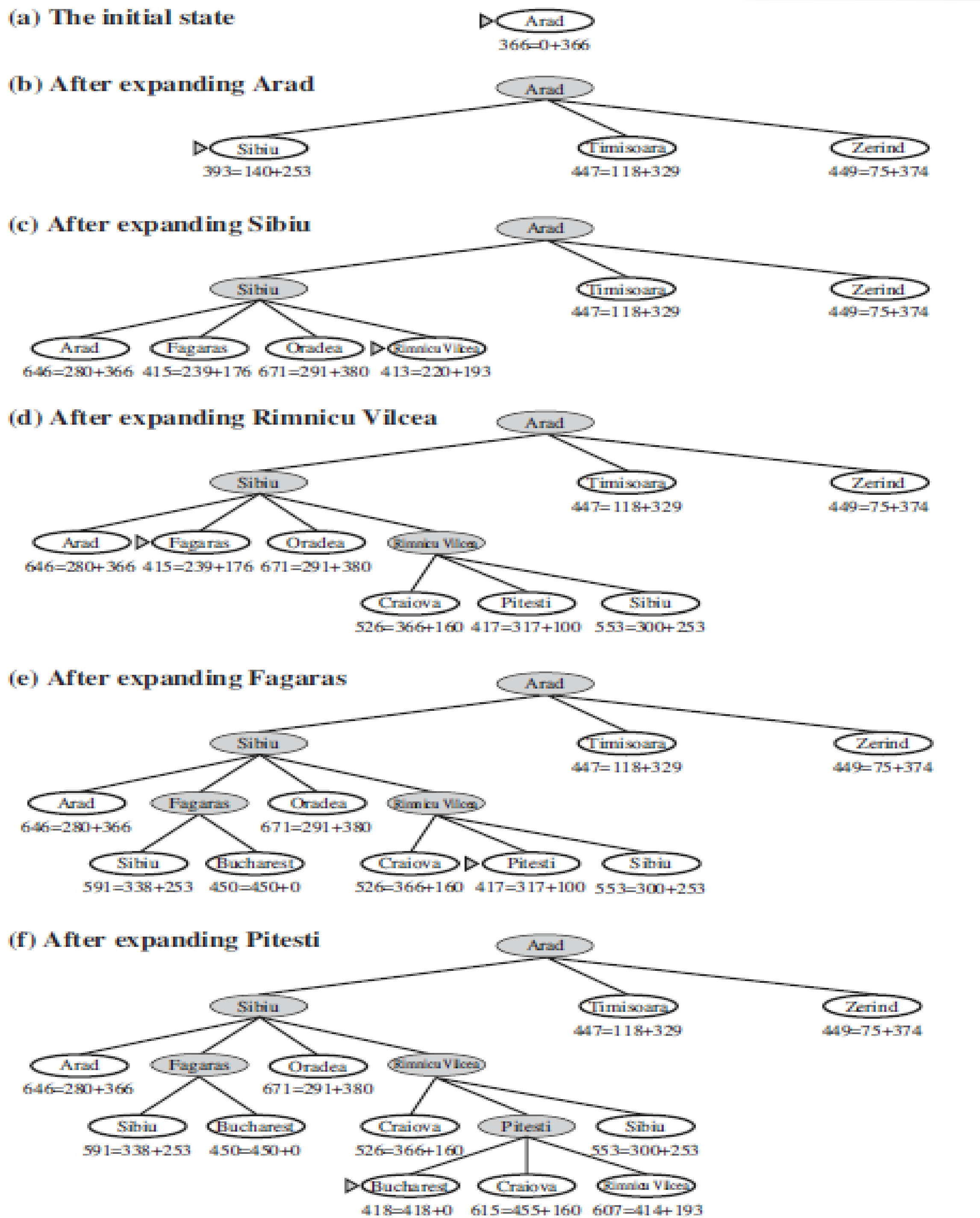
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Draw the search tree and list the nodes and their associated cities in order of expansion for going from Arad to Bucharest

5 minutes



A*

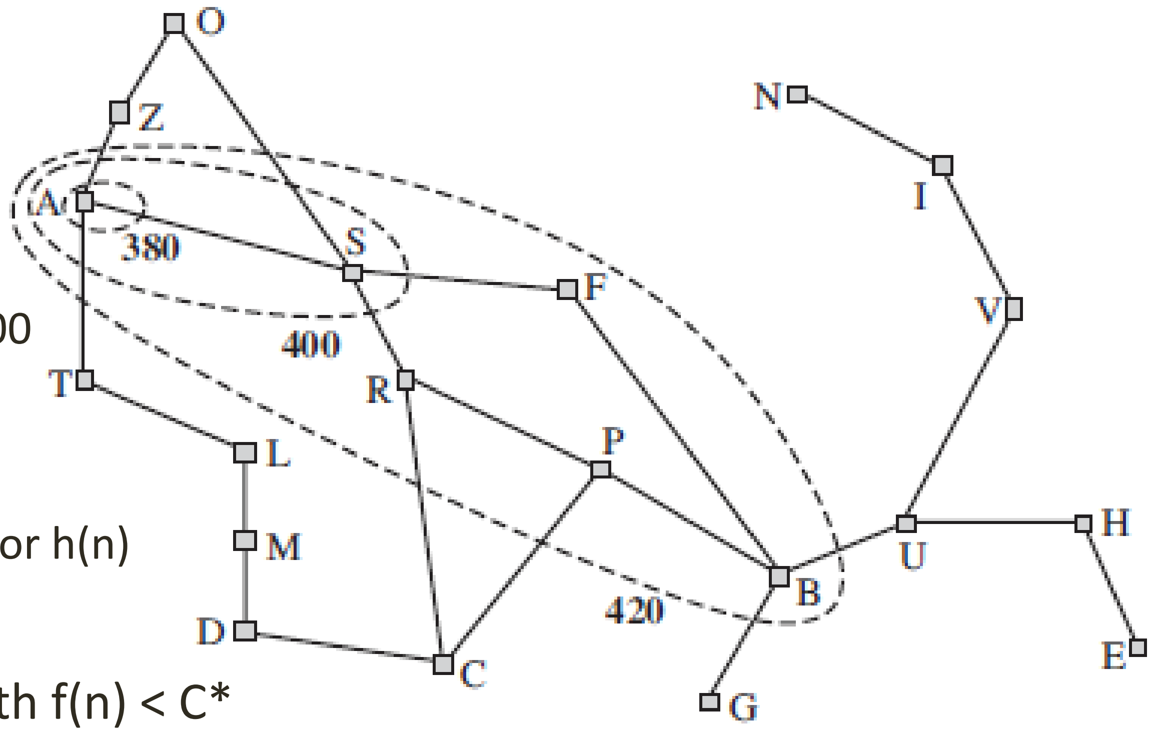

(a) The initial state
Arad
366=0+366
(b) After expanding Arad
Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374
(c) After expanding Sibiu
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193
(d) After expanding Rimnicu Vilcea
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253
(e) After expanding Fagaras
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253
(f) After expanding Pitesti
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
Sibiu
553=300+253
Bucharest
418=418+0
Craiova
615=455+160
Rimnicu Vilcea
607=414+193

A^*

- f(n) = g(n) + h(n)
- = cost to state + estimated cost to goal
- = estimated cost of cheapest solution through *n*
- Seem reasonable?
  - If heuristic is *admissible*, A^* is optimal and complete for Tree search
    - Admissible heuristics underestimate cost to goal
  - If heuristic is *consistent*, A^* is optimal and complete for graph search
    - Consistent heuristics follow the triangle inequality
    - If n' is successor of n, then $h(n) \leq c(n, a, n') + h(n')$
    - Is less than cost of going from n to n' + estimated cost from n' to goal
      - Otherwise you should have expanded n' before n and you need a different heuristic
  - f costs are always non-decreasing along any path

A^* contours

- Non decreasing f implies
  - We can draw contours
  - Inside the 400 contour
    - All nodes have f(n) ≤ 400
  - Contour shape
    - Circular if h(n) = 0
    - Elliptical towards goal for h(n)
- If C* is optimal path cost
  - A* expands **all** nodes with f(n) < C*
  - A* may expand some nodes with f(n) = C* before getting to a goal state
  - If b is finite and all step costs > e, then A* is complete since
    - There will only be a finite number of nodes with f(n) < C*



Search

- Problem solving by searching for a solution in a space of possible solutions
- Uninformed versus Informed search
- Atomic representation of state
- Solutions are fixed sequences of actions