Search

- One of the most fundamental techniques in AI
    - Underlying sub-module in many AI systems
- Can solve many problems that humans are not good at.
- Can achieve super-human performance on some problems (Chess, go)
- Very useful as a general algorithmic technique for solving problems (both in AI and in other areas)

How do we plan our holiday?

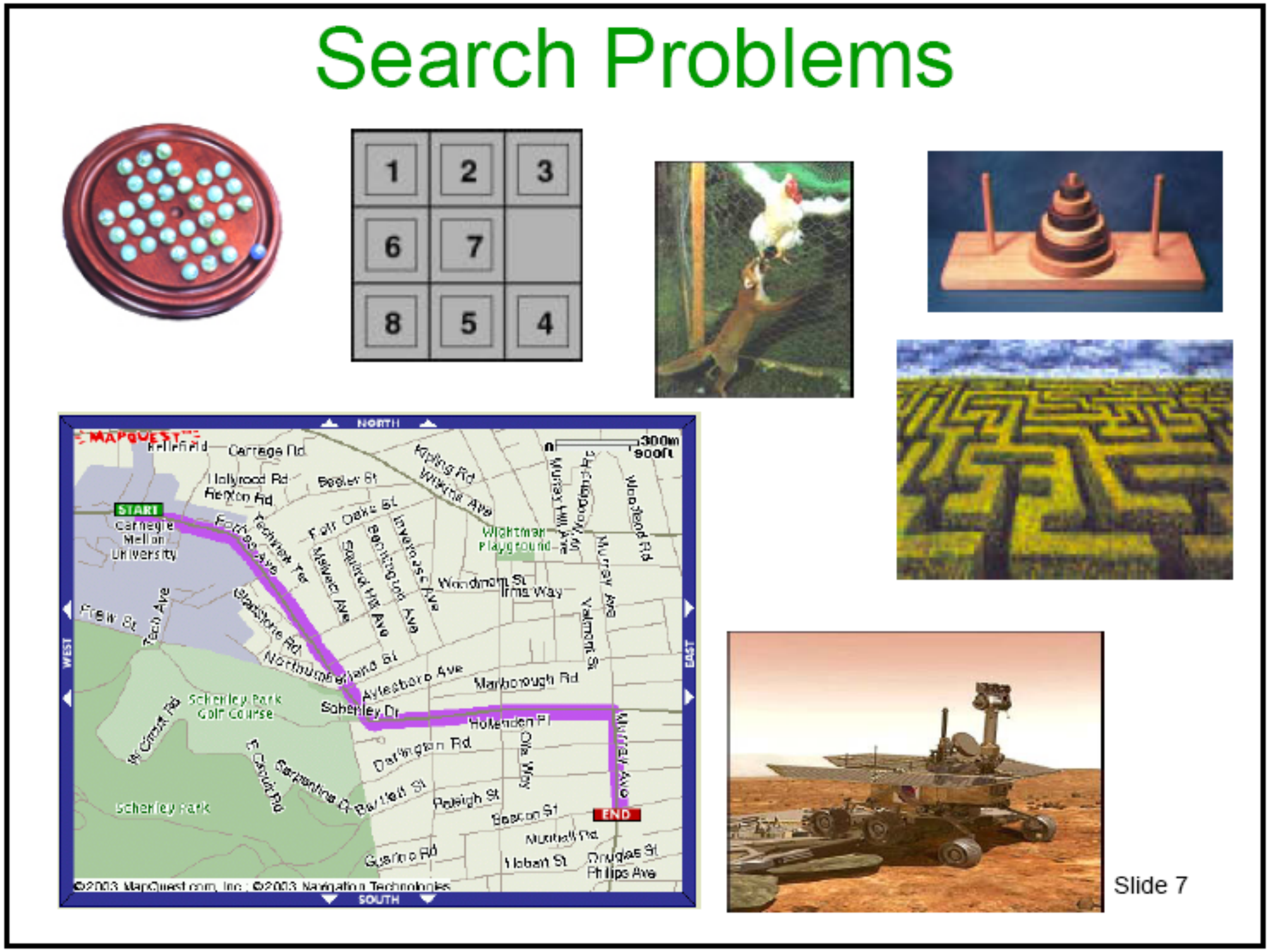
- We must take into account various preferences and constraints to develop a schedule.
- An important technique in developing such a schedule is "hypothetical" reasoning.

- Example: On holiday in England
  - Currently in Edinburgh
  - Flight leaves tomorrow from London
    - Need plan to get to your plane
    - If I take a 6 am train where will I be at 2 pm? Will I be still able to get to the airport on time?

How do we plan our holiday?

- This kind of hypothetical reasoning involves asking
  - what state will I be in after taking certain actions, or after certain sequences of events?
- From this we can reason about particular sequences of actions one should execute to achieve a desirable state.
- Search is a computational method for capturing a particular version of this kind of reasoning.

Many problems can be solved by search:



Search Problems

1 2 3
6 7
8 5 4
MAPQUEST
NORTH
SOUTH
WEST
EAST
START
Carnegie Mellon University
END
Forbes Ave
Northumberland St
Schenley Dr
Murray Ave
Schenley Park
Schenley Park Golf Course
Wightman Playground
©2003 MapQuest.com, Inc.; ©2003 Navigation Technologies

Slide 7

Many problems can be solved by search:

Deepblue 1997

beats Kasparov world
champion Chess player

AlphaGo 2016

beats Lee Sedol 9th
dan Go player

2017 beats Ke Jie
World #1 ranked player

Why Search?

- Successful
  - Success in game playing programs based on search.
  - Many other AI problems can be successfully solved by search.
- Practical
  - Many problems don't have specific algorithms for solving them. Casting as search problems is often the easiest way of solving them.
  - Search can also be useful in approximation (e.g., local search in optimization problems).
  - Problem specific heuristics provides search with a way of exploiting extra knowledge.
- Some critical aspects of intelligent behaviour, e.g., planning, can be naturally cast as search.

Limitations of Search

- There are many difficult questions that are not resolved by search. In particular, the whole question of how does an intelligent system formulate the problem it wants to solve as a search problem is not addressed by search.

- Search only provides a method for solving the problem once we have it correctly formulated.

Search

Search

- Formulating a problem as search problem (representation)
- Heuristic Search

- Readings
  - Introduction: Chapter 3.1 – 3.3
  - Uninformed Search: Chapter 3.4
  - Heuristic Search: Chapters 3.5, 3.6

Representing a problem: The Formalism

To model a problem as a search problem we need the following components:

1. **STATE SPACE:** A state is a representation of a configuration of the problem domain. The **state space** is a set of states included in our model of the problem.
2. **ACTIONS or STATE SPACE Transitions:** Actions these model the actions of the problem domain. In our model the domain actions are modeled as allowed transitions between state.

Representing a problem: The Formalism

3. **INITIAL or START STATE and GOAL:** Identify the initial state that represents the starting conditions, and the goal state or goal condition one wants to achieve.
4. **Heuristics:** Formulate various heuristics to help guide the search process.

Note that this representation of the problem abstracts from the real problem (i.e., omits some details)—it is a model of the problem suitable for use in a computer.

Typically we only model relevant parts of the real world states, and the actions/initial and goal state/heuristics operate on that model in a way that reflects the real problem.

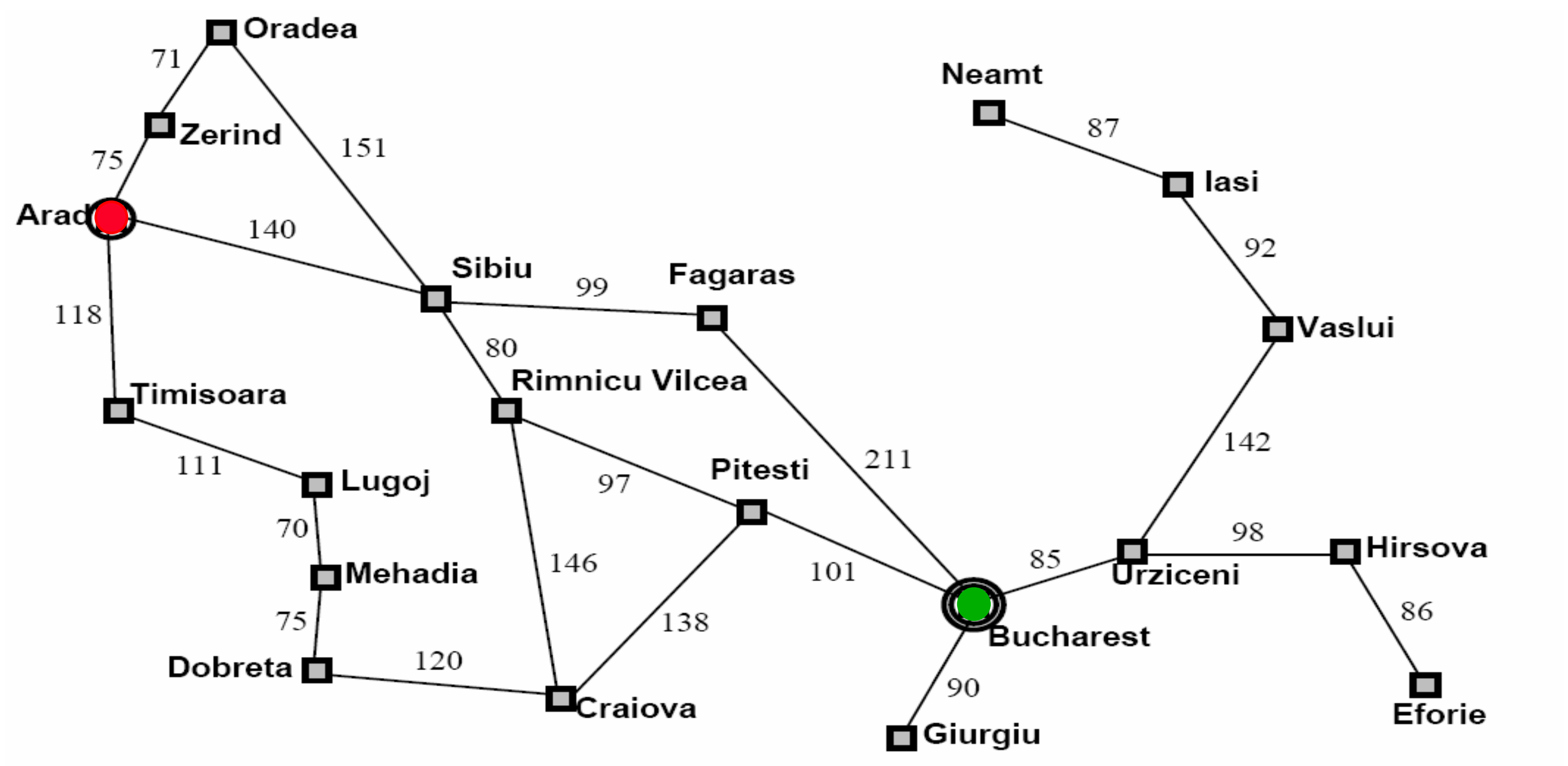
The Formalism

Once the problem has been formulated as a state space search, various algorithms can be utilized to solve the problem.

- A solution to the problem will be a sequence of actions/moves that can transform your current state into a state where your desired condition holds.

Example 1: Romania Travel.

Currently in Arad, need to get to Bucharest by tomorrow to catch a flight. What is a reasonable State Space?



Example 1. Romania Travel.

- State space.
  - States: The set of cites we can be in {A, B, C, ..., Z}. E.g., the state A represents being in Arad.
    - Our abstraction: we are ignoring the low level details of driving, states where you are on the road between cities, etc.
  - Actions: drive between neighboring cities. This changes the state we are in (the city we are in).
  - Initial state: in Arad.
  - Desired condition (Goal): be in a state where you are in Bucharest. (How many states satisfy this condition?)
- Solution will be the route, the sequence of cities to travel through to get to Bucharest.

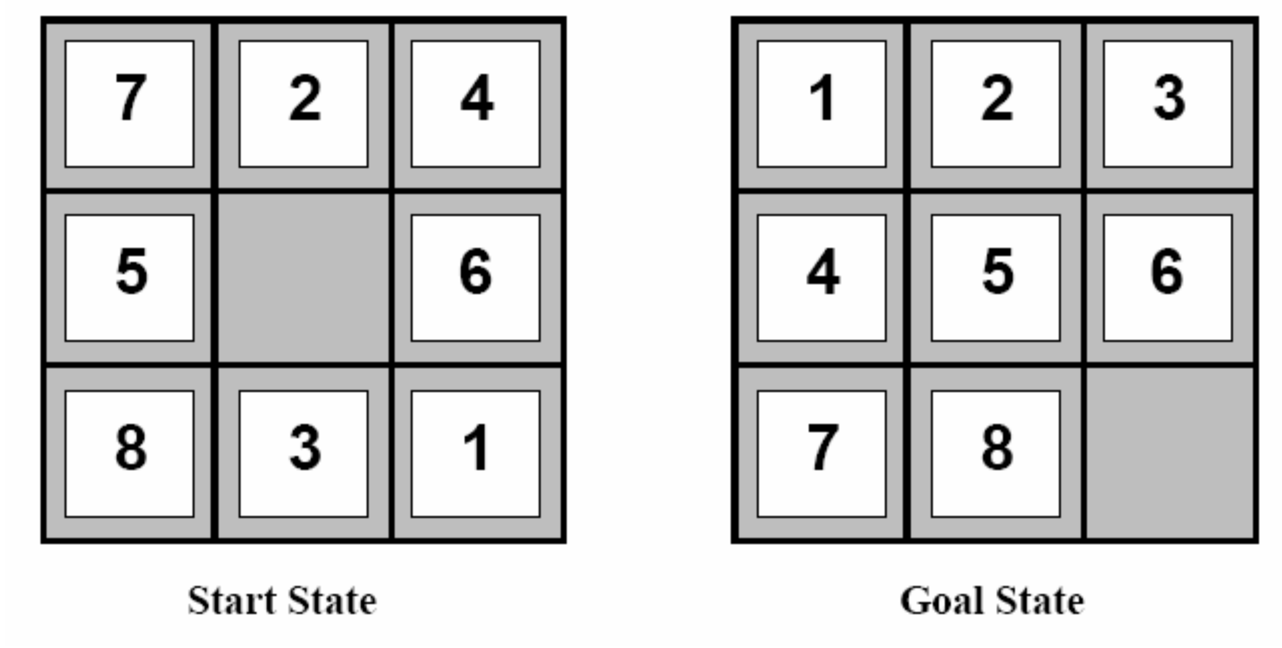
Example 2 Water Jugs.

- Water Jugs
  - We have a 3 gallon (liter) jug and a 4 gallon jug. We can fill either jug to the top from a tap, we can empty either jug, or we can pour one jug into the other (at least until the other jug is full).
  - States: each state is a pair of numbers (gal3, gal4)
    gal3 = the number of gallons in the 3 gallon jug
    gal4 = the number of gallons in the 4 gallon jug.
  - Actions: Empty-3-Gallon, Empty-4-Gallon, Fill-3-Gallon, Fill-4-Gallon, Pour-3-into-4, Pour 4-into-3.
  - Initial state: Various initial states are possible, e.g., (0,0)
  - Desired condition (Goal): Various, e.g., (0,2) or (*, 3) where * means we don't care.

Example 2 Water Jugs.

- Water Jugs
  - If we start off with gal3 and gal4 as integer, can only reach integer values.
  - Some values, e.g., (1,2) are not reachable from some initial state, e.g., (0,0).
  - Some actions are no-ops. They do not change the state, e.g.,
    - (0,0) → Empty-3-Gallon → (0,0)

Example 3. The 8-Puzzle



Rule: Can slide a tile into the blank spot.

Alternative view: move the blank spot around.

Example 3. The 8-Puzzle

- State space.
  - States: The different configurations of the tiles. How many different states? We can represent these states in a matrix or a 9 element vector.
  - Actions: Moving the blank up, down, left, right. Can every action be performed in every state?
  - Initial state: Various initial states are possible, e.g., the state shown on previous slide.
  - Desired condition (Goal): be in a state where the tiles are all in the positions shown on the previous slide.
- Solution will be a sequence of moves of the blank that transform the initial state to a goal state.

Question: we could have as actions the movements of the individual tiles. Would this be a better representation?

Example 3. The 8-Puzzle

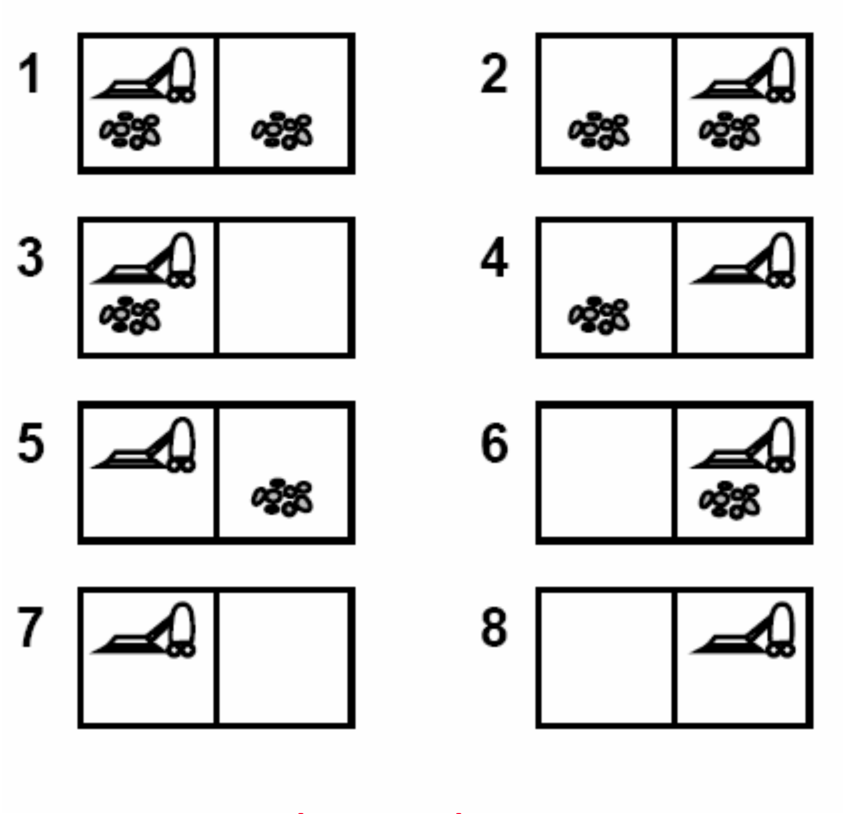
- Although there are 9! different configurations of the tiles (362,880) in fact the state space is divided into two disjoint parts.
- Only when the blank is in the middle are all four actions possible.
- Our goal condition is satisfied by only a single state. But one could easily have a goal condition like
    - The 8 is in the upper left hand corner.
        - How many different states satisfy this goal?

Example 4: Vacuum World

- In the previous examples, a state in the search space represented a particular state of the world.
- However, states need not map directly to world configurations. Instead, a state could map to knowledge states.
- A knowledge state is a set of world states (set of ground states)—every world state that you believe to be possible.
- If you know the exact state of the world your knowledge state is set containing only that world state.
- The facts you know are those facts that are true in every world state contained in your knowledge state.

Example 4. Vacuum World

- We have a vacuum cleaner and two rooms.
- Each room may or may not be dirty.
- The vacuum cleaner can move left or right *(the action has no effect if there is no room to the right/left).*
- The vacuum cleaner can suck; this cleans the room (*even if the room was already clean).*

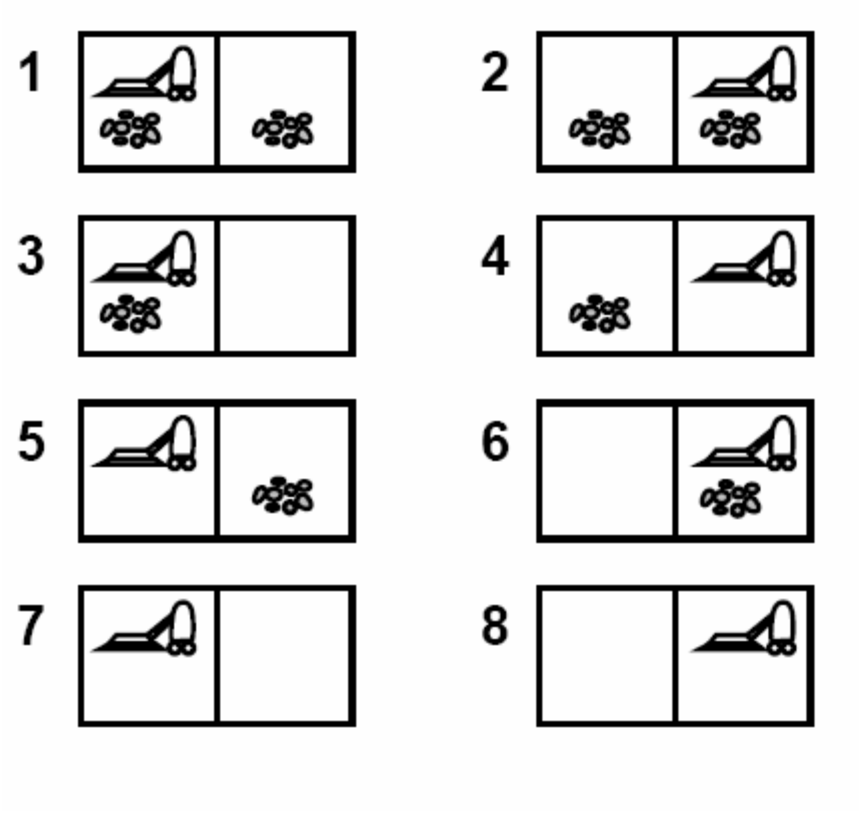


Physical states

Example 4. Vacuum World

Knowledge-level State Space

- Each knowledge state consists of a set of possible world states. The agent knows that it is in one of these world states, but doesn't know which.

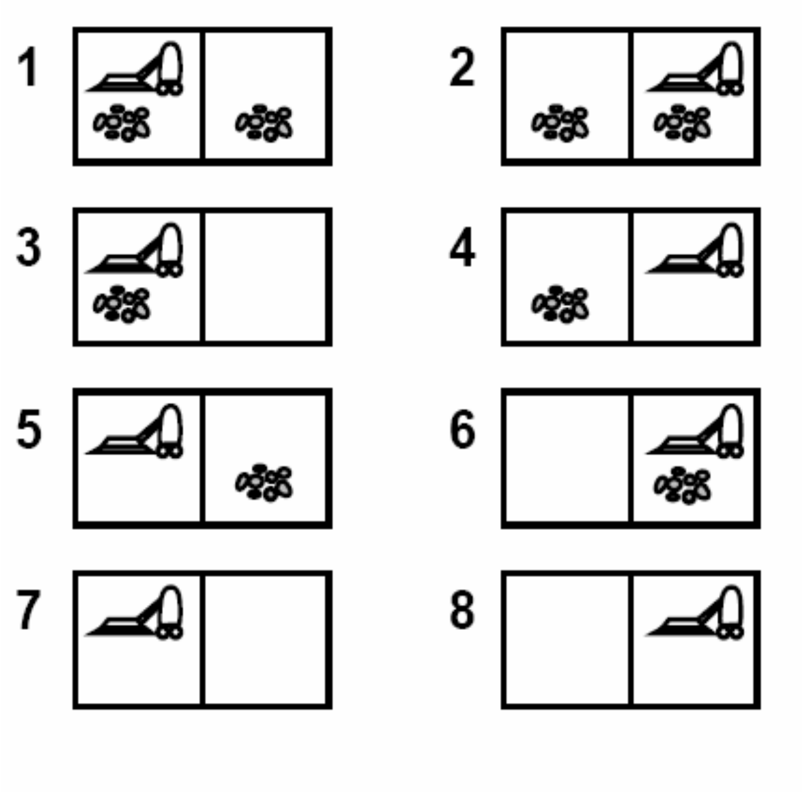


Goal is to have all rooms clean.

Example 4. Vacuum World

Knowledge-level State Space

- Complete knowledge of the world: agent knows exactly which physical state it is in. Then the states in the agent's state space consist of single physical states.
- Start in {5}:
  <right, suck>

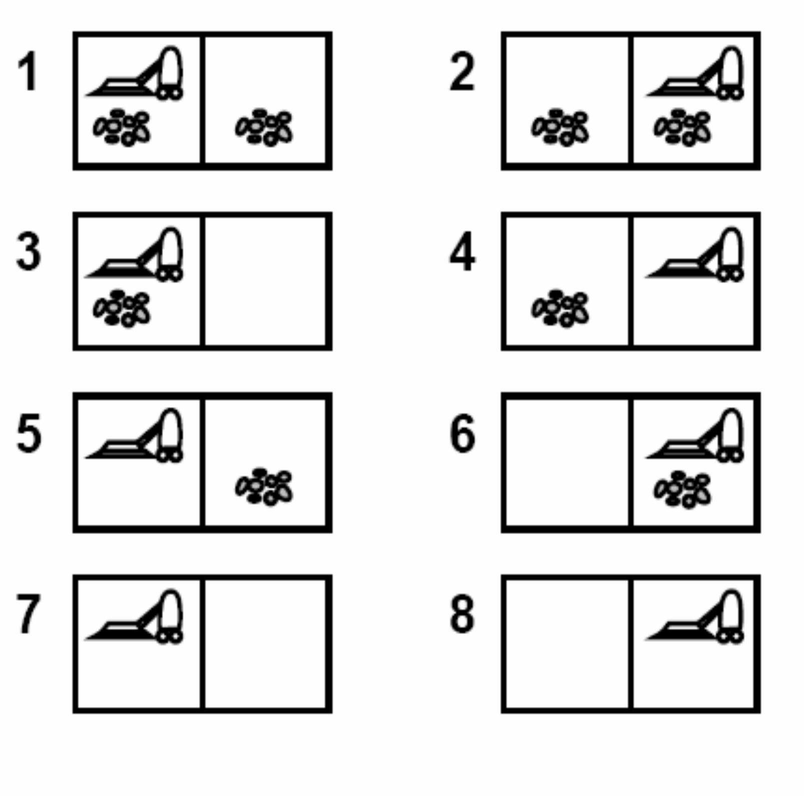


Goal is to have all rooms clean.

Example 4. Vacuum World

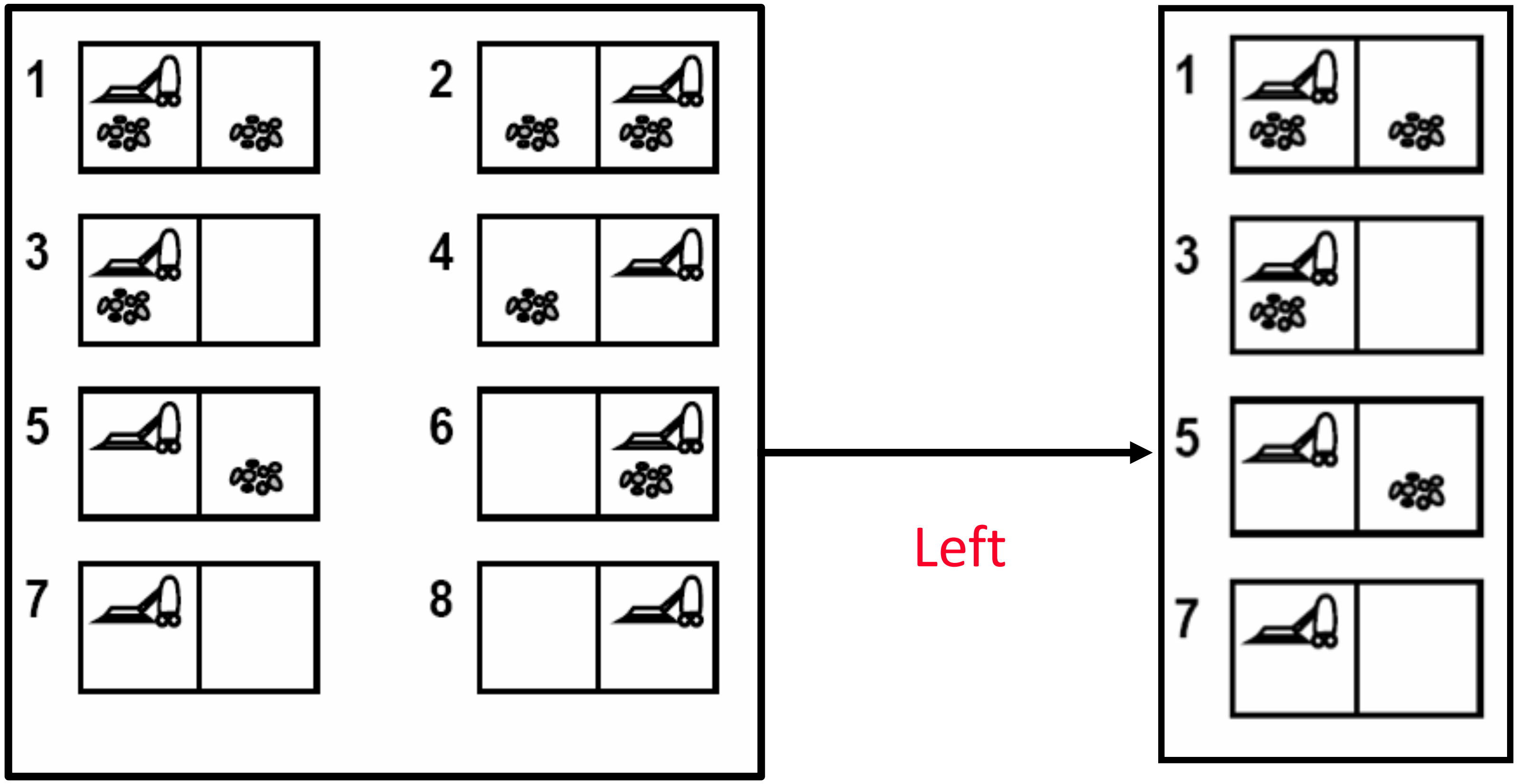
Knowledge-level State Space

- knowledge states:
  Agent's knowledge states consist of *sets of world states.*
- E.g. starting in {1,2,3,4,5,6,7,8}, the agent doesn't have any knowledge of where it is.
- Nevertheless, the action sequence
  <right, suck, left, suck>
  achieves the goal.



Goal is to have all rooms clean.

Example 4. Vacuum World

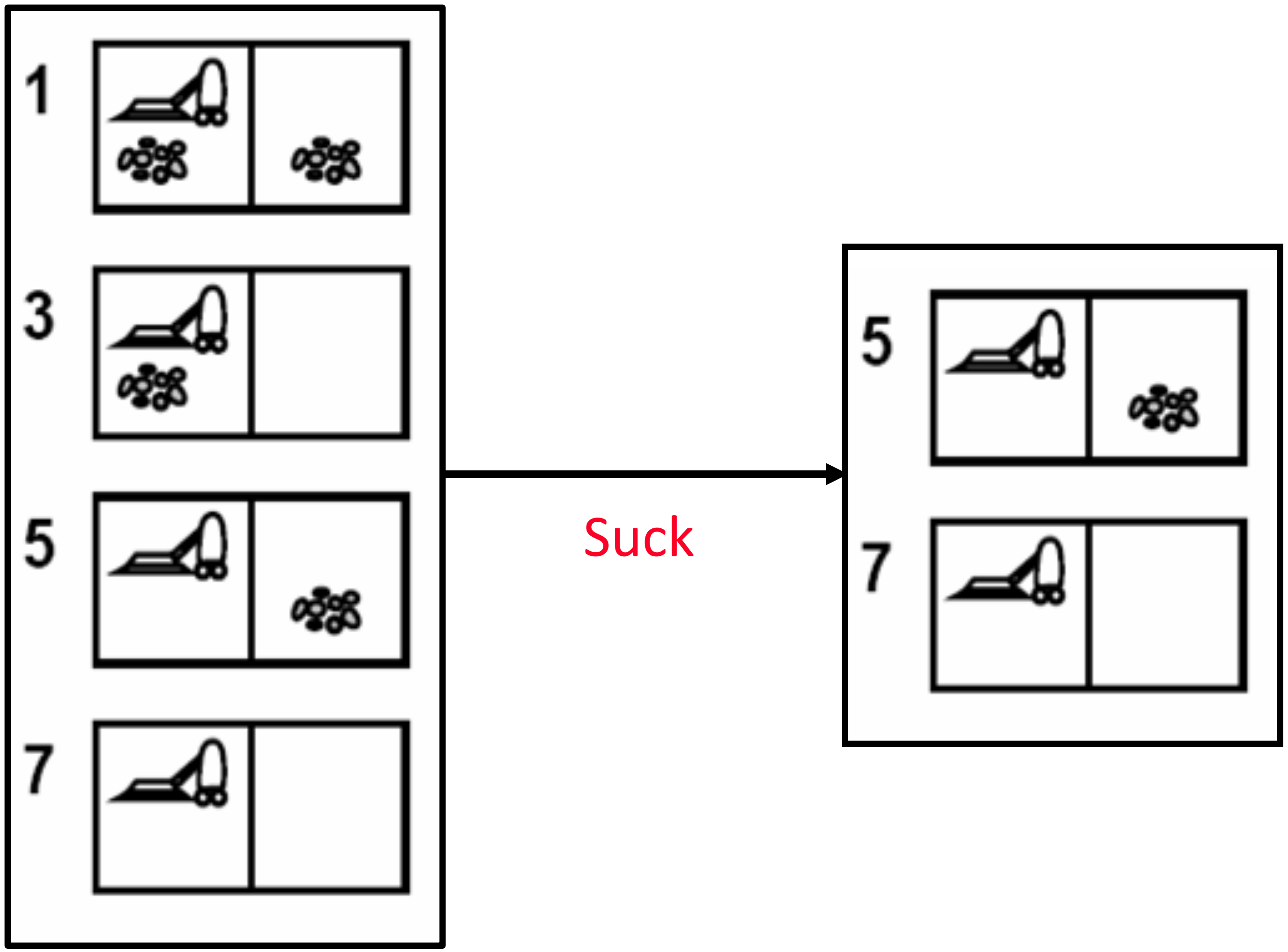


Initial state.

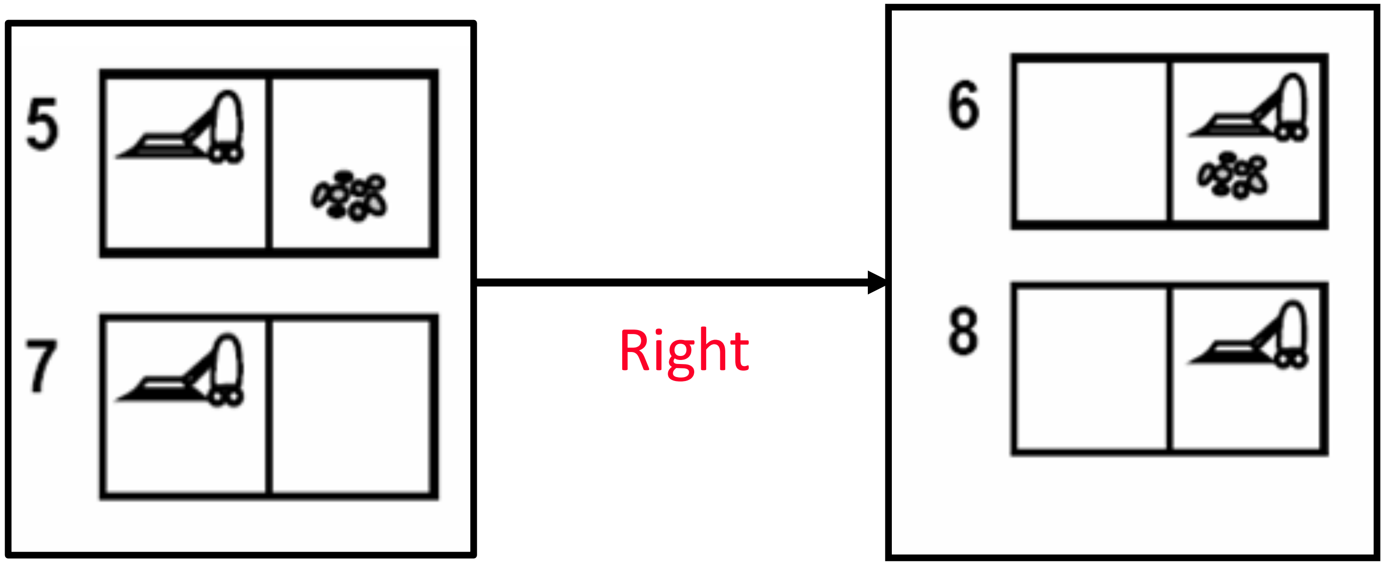
{1,2,3,4,5,6,7,8}

What does the agent know in this knowledge state?

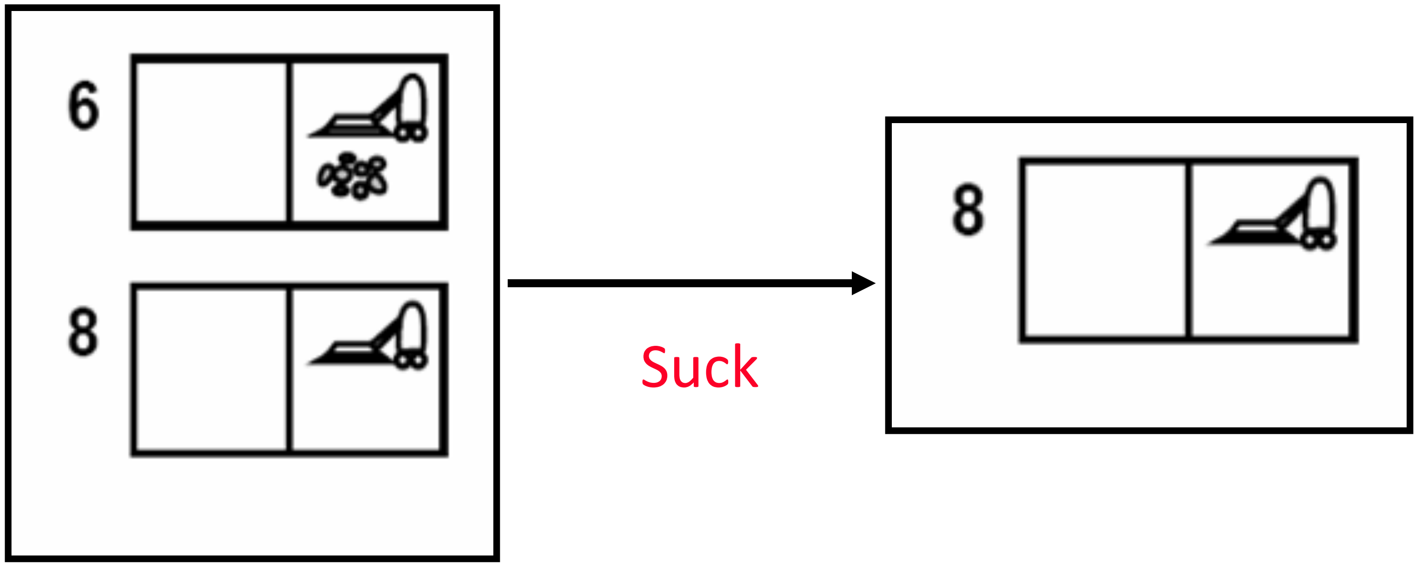
Example 4. Vacuum World



Example 4. Vacuum World



Example 4. Vacuum World

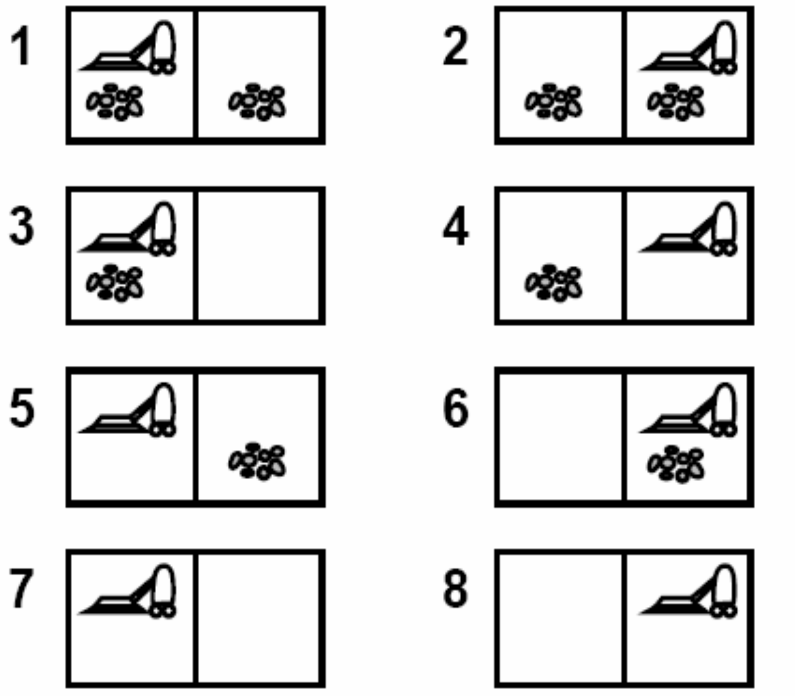


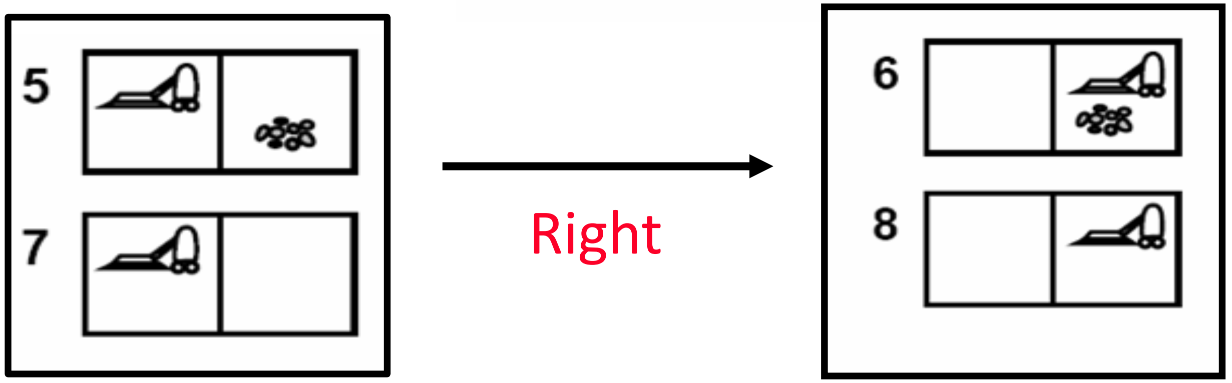
Example 4. Vacuum World

- State space.
  - States:
    - ground states G = {1, 2, 3, 4, 5, 6, 7, 8}—the configuration shown on previous slide.
    - states = {s | s is a non-empty subset of G}. There are 2^8 -1. Each state s is the set of world configurations the agent believes are possible. The agent does not know which ground state in s is the true world configuration.

Example 4. Vacuum World

- Actions: Left, Right, Suck
  - How would you specify these actions?
  - Left(s) = (s ∩ { 1, 3, 5, 7}) ∪ { g – 1 | g ∈ s ∩ {2, 4, 6, 8}
  - Right(s) = (s ∩ { 2, 4, 6, 8}) ∪ { g + 1 | g ∈ s ∩ {1, 3, 5, 7}
  - Suck(s) = exercise
- e.g., Right({5,7}) = {6,8}





Example 4. Vacuum World

- Initial state: Various initial states are possible, e.g., the state of not knowing anything about your room is {1,2,3,4,5,6,7,8}.
- Desired condition (Goal): $\{ s \mid s \subseteq \{7, 8\} \}$ note this is a set of states (the goal need not be a single state)

- Solution will be a sequence of Left, Right, Suck moves that transform the initial state to a goal state.

- This example shows that the **state space** formalism can deal with domains in which our knowledge of the world state is incomplete.

More complex situations

- Actions can lead to multiple states, e.g., flip a coin to obtain heads OR tails. Or we don't know for sure what the initial state is (prize is behind door 1, 2, or 3). Now we might want to consider how likely different states and action outcomes are.
- This leads to probabilistic models of the search space and different algorithms for solving the problem.
- Later we will see some techniques for reasoning under uncertainty.

More complex situations

- The agent might be equipped with sensing actions.
  - These actions change the agent’s knowledge state—they don’t change the state of the world.
- With sensing we can search for contingent solutions: solutions that contain branches depending on the outcome of sensing actions.

  <right, if dirt then suck, left>
- Searching for contingent plans needs different algorithms.

Algorithms for Search

- AI search algorithms work with implicitly defined state spaces.
- There are typically an exponential number of states: impossible to explicitly represent them all.
- The space of possible configurations of a Go board is about 3^{361} (standard 19X19 board).
- There are even more actions than state.

Algorithms for Search

- In AI search we find solutions by constructing only those states we need to. In the worst case we will need to construct an exponential number of states—and the search might require too much computation.
- But often we can solve hard problems (like Go) while only examining a small fraction of the states.
- Hence the actions are given as compact successor state functions that when given a state S return the set of states S can be transformed to by the available actions.
    - This means that the state must contain enough information to allow the successor state function to perform its computation.

Algorithms for Search

Inputs:

- a specified initial state I
- a successor function S(x) yields the set of **all** states action pairs (y,a) such that y can be reached from x by applying an action a. The successor function returns all states reachable by a single action from x.
- a goal test a function that can be applied to a state and returns true if the state satisfies the goal condition.
- An action cost function C(x,a,y) which determines the cost of moving from state x to state y using action a. (C(x,a,y) = ∞ if a does not yield y from x). Note that different actions might generate the same transition x → y.

Algorithms for Search

Output:

- a sequence of actions that transforms the initial state to a state satisfying the goal test.
  - Or just the sequence of states that arise from these actions (depends on what kind of information is most useful)

- The sequence might be, optimal in cost for some algorithms, optimal in length for some algorithms, come with no optimality guarantees from other algorithms.
  - That is, no other sequence transforms the initial state to a goal satisfying state with lower cost (or lesser length).

Algorithms for Search

Obtaining the action sequence.

- The set of successors of a state x might arise from different actions, e.g.,
  - x → a → y
  - x → b → z
- Successor function S(x) yields a set of states that can be reached from x via **any** action.
  - S(x) = {<y,a>, <z,b>}
    y via action a, z via action b.
  - S(x) = {<y,a>, <y,b>}
    y via action a, also y via alternative action b.

Search Algorithms

- The search space consists of **states** and actions that move between states.
- A **path** in the search space is a **sequence** of states connected by actions, $<s_0, s_1, s_2, \ldots, s_k>$,
  for every s_i and its successor s_{i+1} there must exist an action a_i that transitions s_i to s_{i+1}.
  - Alternately a path can be specified by
    (a) an initial state s_0, and
    (b) a sequence of actions that are applied in turn starting from s_0.

Search Algorithms

- The search algorithms perform search by examining alternate paths of the search space. The objects used in the algorithm are called **nodes**—each node **n** is a path in the search space:
    - In practice the path might be stored as a pointer from a node data structure to its parent node. Following those pointers to the initial state yields the path.
  - n.state() is a function that returns the terminal state of the path n
- e.g., n = <A,B,C>, n.state = C

- All paths examined by our search algorithms will start at the initial state I.
  - In the example the initial state must be A.

Algorithm for Search

- We maintain a set of nodes called the OPEN set (or frontier).
  - These nodes are paths in the search space that all start at the initial state.
- Initially we set OPEN = {<Start State>}
  - The path (node) that starts and terminates at the start state.
- At each step we select a node n from OPEN.
  - We check if **n.state()** satisfies the goal,
  - If not we add all extensions of n to OPEN
    - for all successor states y $\in$ S(n.state)
      create a new node n_y = <n,y> – extend the path n.path include y

  - e.g., if n = <a, b, c, d> then n.state() = d. Let S(d) = {e, f} then we get two new extensions of n that will be added to OPEN
    1. n_e = <a, b, c, d, e>
    2. n_f = <a, b, c, d, f>

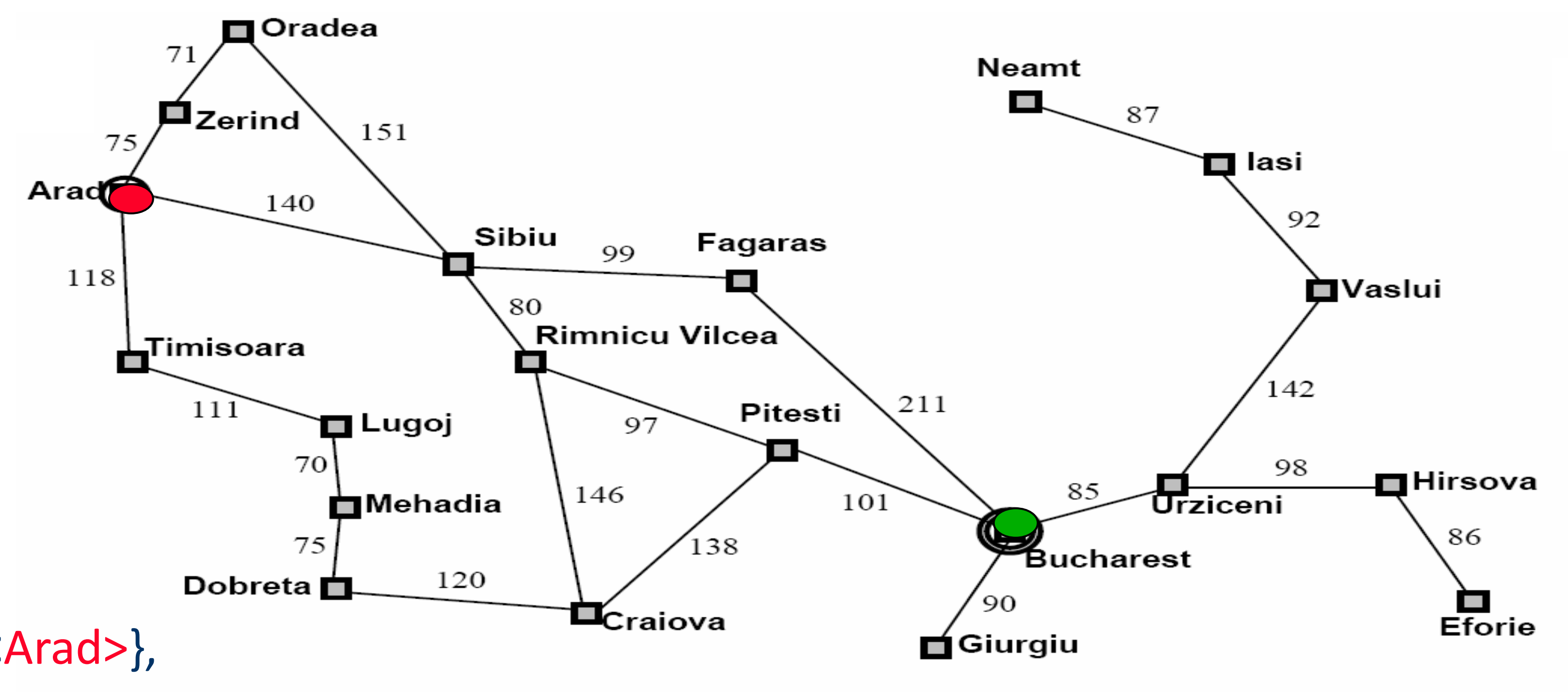
Algorithm for Search

```
Search(open, successors, goal? ):
    open.insert(<start>)
    while not open.empty():
        n = open.extract()   #remove node from OPEN
        if (goal?(n.state())):
            return n                #n is solution
        for succ in S(n.state()): #S is the successor function
            open.insert({<n,succ>, )
                                    #open could grow or shrink
    return false
```

When does OPEN get smaller in size?

Algorithm for Search

- When a node n is extracted from open, we say that the algorithm expands n.
- The number of states we actually construct (i.e., the total number of states returned by the successor function S() summed over all calls to S()), we hope is low compared to the total number of states.
- The number of states expanded depends on the order of nodes we extract from open.



{<Arad>},

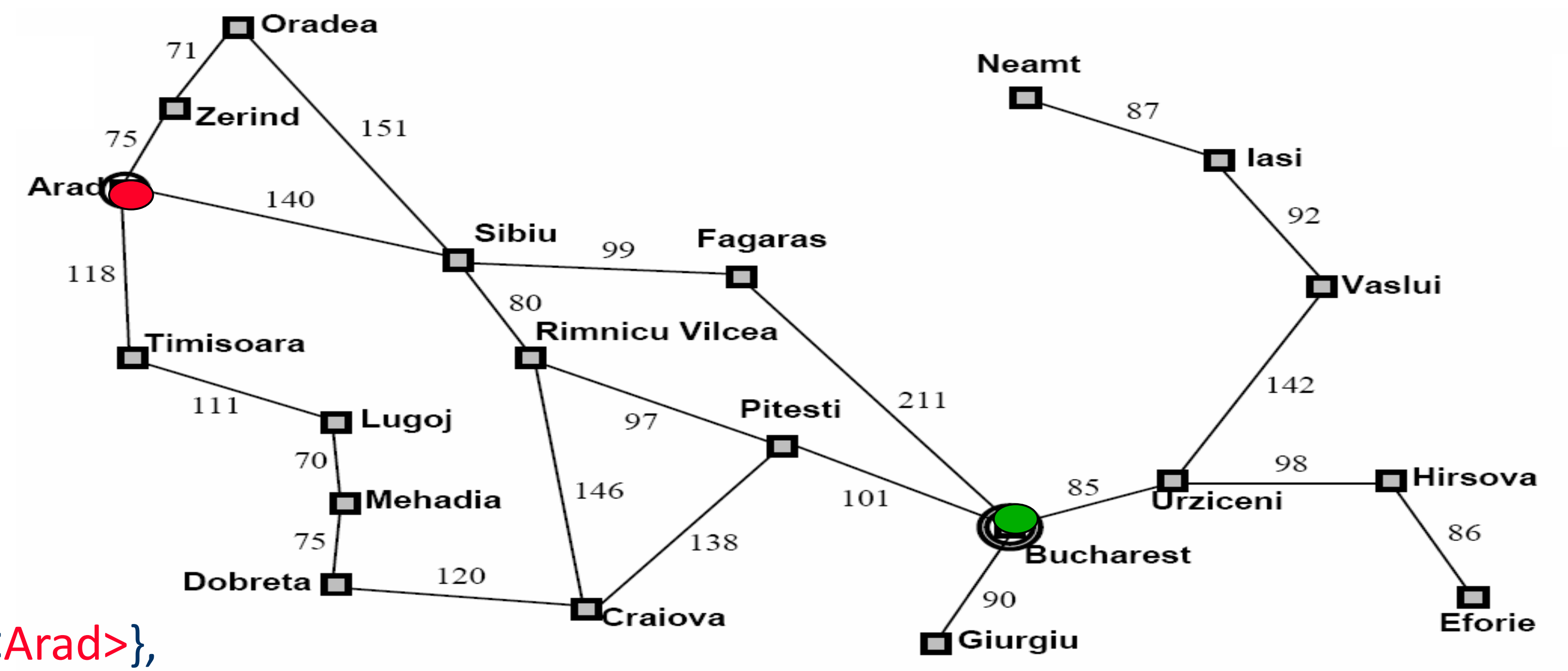
{**<A,Z>, <A,T>, <A, S>**},

{<A,Z>, <A,T>, **<A,S,A>, <A,S,O>, <A,S,F>, <A,S,R>**}

{<A,Z>, <A,T>, <A,S,A>, <A,S,O>, <A,S,R>, **<A,S,F,S>, <A,S,F,B>**}

Solution: Arad -> Sibiu -> Fagaras -> Bucharest

Cost: 140 + 99 + 211 = 450



{<Arad>},

{**<A,Z>, <A,T>, <A, S>**},

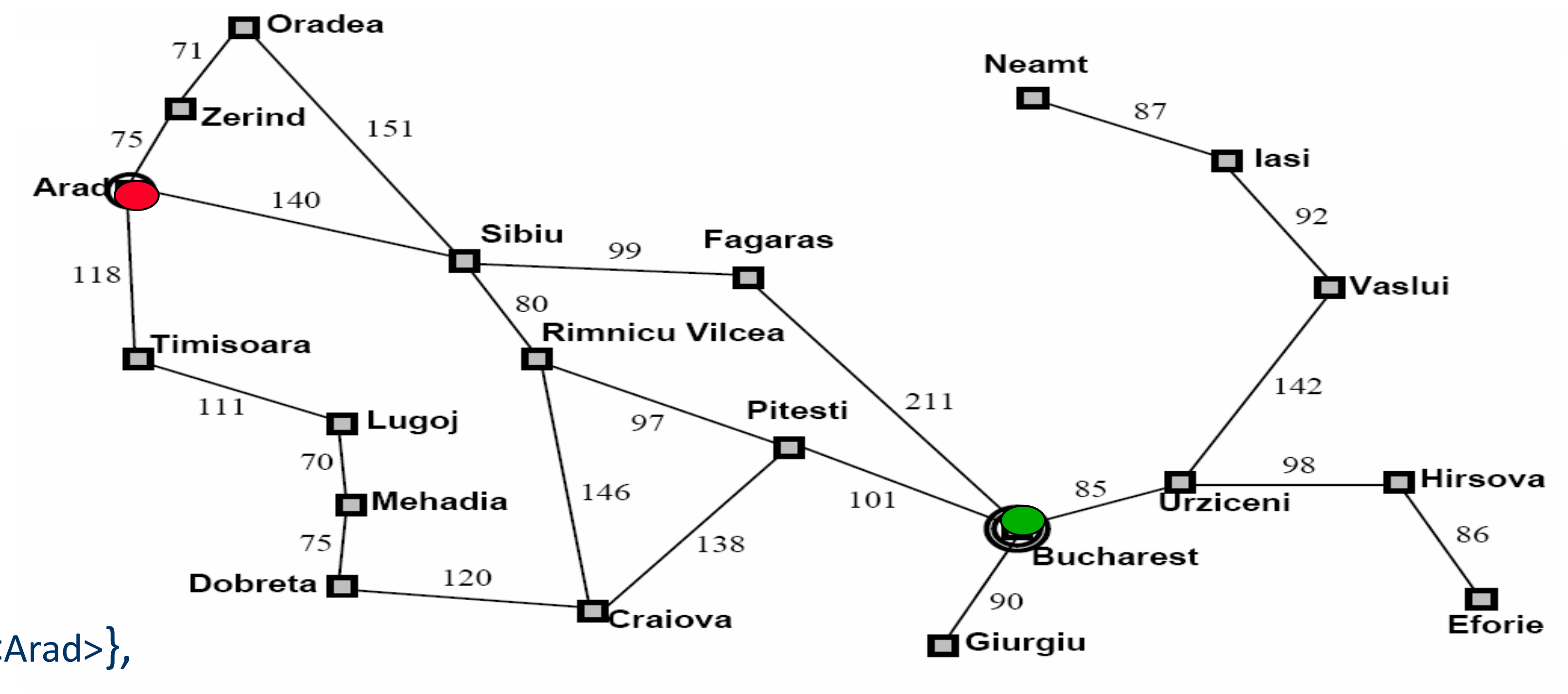
{<A,Z>, <A,T>, **<A,S,A>, <A,S,O>, <A,S,F>, <A,S,R>**}

{<A,Z>, <A,T>, <A,S,A>, <A,S,O>, <A,S,F>, **<A,S,R,S>, <A,S,R,C>, <A,S,R,P>**}

{<A,Z>, <A,T>, <A,S,A>, <A,S,O>, <A,S,F>, <A,S,R,S>, <A,S,R,C>, **<A,S,R,P,R>**, **<A,S,R,P,C>**, <A,S,R,P,B>}

Solution: Arad -> Sibiu -> Rimnicu Vilcea -> Pitesti -> Bucharest

Cost: 140 + 80 + 97 + 101 = 418



{<Arad>},

{<A,Z>, <A,T>, <A, S>},

{<A,Z>, <A,T>, **<A,S,A>,** <A,S,O>, <A,S,F>, <A,S,R>}

{<A,Z>, <A,T>, **<A,S,A>**, <A,S,O>, <A,S,R>, **<A,S,F,S>**, <A,S,F,B>}

…..

cycles can cause non-termination!

… we deal with this issue later

Selection Rule

The order paths are selected from OPEN has a critical effect on the operation of the search:

- Whether or not a solution is found
- The cost of the solution found.
- The time and space required by the search.

How to select the next path from OPEN?

All search techniques keep OPEN as an ordered set and repeatedly execute:

- If OPEN is empty, terminate with failure.
- Remove the **first** path (search node) from OPEN (OPEN is ordered!)
- If the path leads to a goal state, terminate with success.
- Extend the path (i.e. generate the successor states of the final state of the path) and put the new paths in OPEN.

- The question of which path to select next from OPEN is now equivalent to the question of how do we order the paths on OPEN?

Critical Properties of Search

- Completeness: will the search always find a solution if a solution exists?
- Optimality: will the search always find the least cost solution? (when actions have costs)
- Time complexity: what is the maximum number of nodes (paths) than can be expanded or generated?
- Space complexity: what is the maximum number of nodes (paths) that have to be stored in memory?

Uninformed Search Strategies

- These are strategies that adopt a fixed rule for selecting the next state to be expanded.
- The rule does not change, particular properties of the search problem being solved are ignored.
- Uninformed search techniques:
  - Breadth-First, Uniform-Cost, Depth-First, Depth-Limited, and Iterative-Deepening search

Uninformed Search

- You would have seen breadth-first search and depth-first search in CSC263/265.
  - Graph nodes = state space states
  - Graph edges = state space actions
- In that course however, it is assumed that the graph we are searching is explicitly represented as an adjacency list (or adjacency matrix).
  - This won’t work when there an exponential number of and edges.
- Similarly uniform cost search is like Dijkstra’s algorithm, but without an explicitly represented graph.
- All of these algorithms are simple instantiations of our implicit graph search.

Breadth-First Search

Breadth-First Search

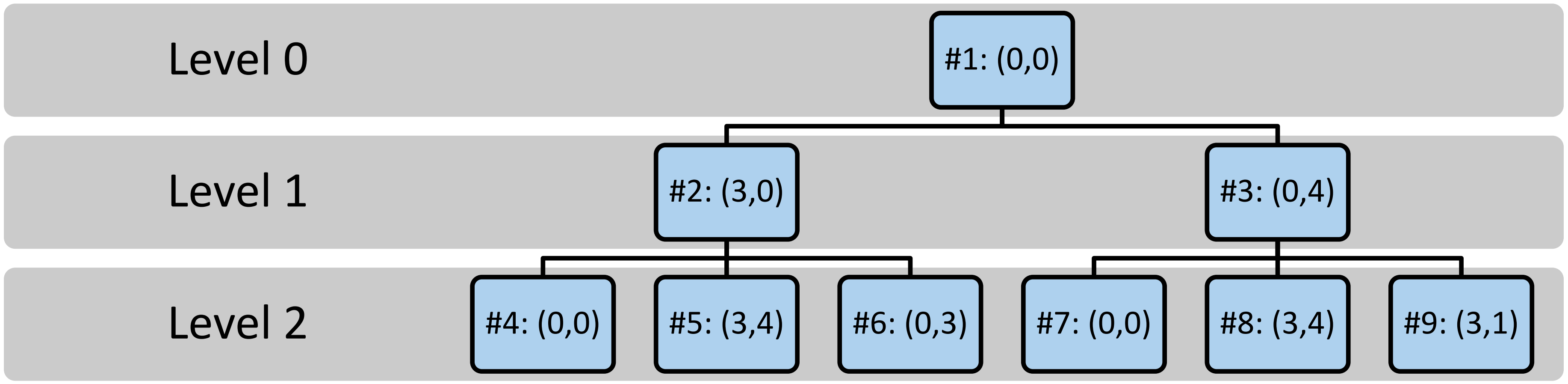
- Place the new paths that extend the current path at the end of OPEN. Extract first element of OPEN

  WaterJugs. Start = (0,0), Goal = (*,2)

Red = Expanded next. Green = newly added

1. OPEN = {<(0,0)>}
2. OPEN = {<(0,0),(3,0)>, <(0,0),(0,4)>}
3. OPEN = {<(0,0),(0,4)>, <(0,0),(3,0),(0,0)>,
   <(0,0) ,(3,0),(3,4)>, <(0,0),(3,0),(0,3)>}
4. OPEN = {<(0,0),(3,0),(0,0)>, <(0,0),(3,0),(3,4)>,
   <(0,0),(3,0),(0,3)>, <(0,0),(0,4),(0,0)>,
   <(0,0),(0,4),(3,4)>, <(0,0),(0,4),(3,1)>}

Breadth-First Search



- Above we indicate only the state that each nodes terminates at. The path represented by each node is the path from the root to that node.
- Breadth-First explores the search space level by level.

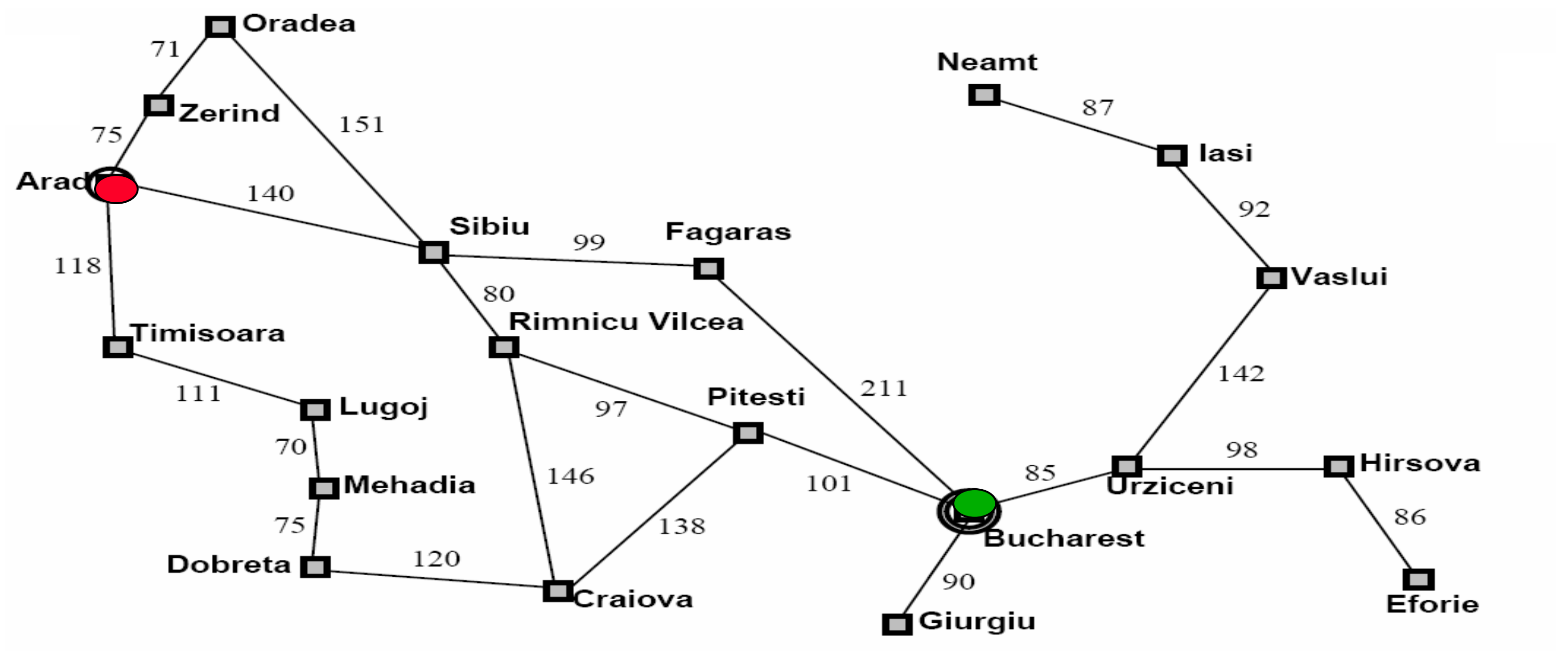
Breadth-First Properties

Completeness?

- The length of the path removed from OPEN is non-decreasing.
  - we replace each expanded node **n** with an extension of **n**.
  - **All** shorter paths are expanded prior to any longer path.
- Hence, eventually we must examine all paths of length d, and thus find a solution if one exists.

Optimality?

- By the above will find shortest length solution
  - least cost solution?
  - Not necessarily: shortest solution not always cheapest solution if actions have varying costs



Beadth first Solution: Arad -> Sibiu -> Fagaras -> Bucharest
Cost: 140 + 99 + 211 = 450

Lowest cost Solution: Arad -> Sibiu -> Rimnicu Vilcea -> Pitesti -> Bucharest
Cost: 140 + 80 + 97 + 101 = 418

Breadth-First Properties

Measuring time and space complexity **(Note our search algorithm operates slightly differently from the Breadth-First Algorithm specified in the recommended text. Hence our analysis is different.)**

- let b be the maximum number of successors of any node (**maximal branching factor**).
- let d be the depth of the shortest solution.
  - Root at depth 0 is a path of length 1
  - So length of path = d+1

Time Complexity?

$$1 + b + b^2 + b^3 + \dots + b^{d-1} + b^d + b(b^d - 1) = O(b^{d+1})$$

Breadth-First Properties

Space Complexity?

- $O(b^{d+1})$: If goal node is last node at level d, all of the successors of the other nodes will be on OPEN when the goal node is expanded. This is $b(b^d – 1)$ nodes at level d+1

Breadth-First Properties

Space complexity is a real problem.

- E.g., let b = 10, and say 100,000 nodes can be expanded per second and each node requires 100 bytes of storage:

Depth	Nodes	Time	Memory
1	1	0.01 millisec.	100 bytes
6	10^6	10 sec.	100 MB
8	10^8	17 min.	10 GB
9	10^9	3 hrs.	100 GB

- Typically run out of space before we run out of time in most applications.

Uniform-Cost Search

Uniform-Cost Search

- Keep OPEN ordered by increasing cost of the path.
- Always expand the least cost path.
- Identical to Breadth first if each action has the same cost.

Uniform-Cost Properties

Completeness?

- If each transition has costs $\geq \varepsilon > 0$.
- The previous argument used for breadth first search holds: the cost of the path represented by each node n chosen to be expanded must be non-decreasing.

Optimality?

- Finds optimal solution if each transition has cost $\geq \varepsilon > 0$.
  - Explores paths in the search space in increasing order of cost. So must find minimum cost path to a goal before finding any higher costs paths leading to a goal

Uniform-Cost Search. Proof of Optimality

Let us prove Optimality more formally.

Uniform-Cost Search. Proof of Optimality

Lemma 1.

Let c(n) be the cost of node n on OPEN (cost of the path). If n2 is expanded IMMEDIATELY after n1 then $c(n1) \leq c(n2)$.

Proof: there are 2 cases:

a. n2 was on OPEN when n1 was expanded:

We must have $c(n1) \leq c(n2)$ otherwise n2 would have been selected for expansion rather than n1

b. n2 was added to OPEN when n1 was expanded

Now $c(n1) < c(n2)$ since the path represented by n2 extends the path represented by n1 and thus n2 has cost at least ε more than n1

Uniform-Cost Search. Proof of Optimality

Lemma 2.

When node n is expanded every path in the search space with cost strictly less than c(n) has already been expanded.

Proof:

- Assume nk = <Start, s1, …, sk> is a path with cost less than c(n). We show that it must have been expanded before node n.
- Define all prefixes of path nk.
  n0 = <Start>,
  n1 = <Start, s1>,
  n2 = <Start, s1, s2>,
  …,
  ni = <Start, s1, …, si>,
  …,
  nk = <Start, s1, …, sk>.

Uniform-Cost Search. Proof of Optimality

- Let ni be the last node in this sequence that has already been expanded by search. We show by **contradiction** that i = k so that nk has been expanded.
- Suppose i < k, then ni has been expanded but ni+1 has not. **Note** that i > 0 since <Start> is the first node expanded by the search.
- ni+1 must still be on OPEN.
  When ni was expanded all of its successors paths are added to OPEN and ni+1 is one of its successor paths.
- c(ni+1) ≤ c(nk) < c(n)
  a) c(ni+1) is a subpath of nk
  b) we have assumed that c(nk) is < c(n).
- But then uniform-cost would have expanded ni+1 before n. **Contradiction.** So i = k and every node ni including nk must already be expanded, i.e., this lower cost path has already been expanded.

Uniform-Cost Search. Proof of Optimality

Lemma 3.

The first time uniform-cost expands a node n terminating at state S, it has found the minimal cost path to S (it might later find other paths to S but none of them can be cheaper).

Proof:

- All cheaper paths have already been expanded, none of them terminated at S.
- All paths expanded after n will be at least as expensive, so no cheaper path to S can be found later.

So, when a path to a goal state is expanded the path must be optimal (lowest cost).

Uniform Cost Search Visualizing the Proof

- Paths in the State Space ordered by cost

Path	Cost
<S>	0
<S, a>	1.0
<S,b>	1.0
<S,a,c>	1.5
<S,d>	2.0
…	…

Uniform Cost Search Visualizing the Proof

- Uniform Cost Search expands paths in non-decreasing order of cost. So it can only go down this list.
  LEMMA 1

Path	Cost
<S>	0
<S, a>	1.0
<S,a,b>	1.5
<S,a,b,c>	3.0
<S,a,b,d,e>	4.0
…	…

Uniform Cost Search Visualizing the Proof

It does not miss any paths on this list. LEMMA 2

Path	Cost
<S>	0
<S, a>	1.0
<S,a,b>	1.5
<S,a,b,c>	3.0
<S,a,b,d>	4.0
…	…

- If <S,a,b,d> is expanded next, <S,a,b,c>, <S,a,b> …, must all have already been expanded as all paths with lower cost than <S,a,b,d> must already be expanded

Uniform Cost Search Visualizing the Proof

Thus working its way down such a list of paths the first path achieving the goal that UCS finds must be the cheapest path reaching the goal.

Uniform-Cost Properties

Time and Space Complexity?

- O($b^{C^*/\varepsilon}$) where C* is the cost of the optimal solution.
- There may be many paths with cost ≤ C*

  Paths with cost lower than C* can be as long as C*/ε (why no longer?), there can be as many as $b^{C^*/\varepsilon}$ paths with C*/ε and we have to explore them all before finding an optimal cost path.

Depth-First Search

Depth-First Search

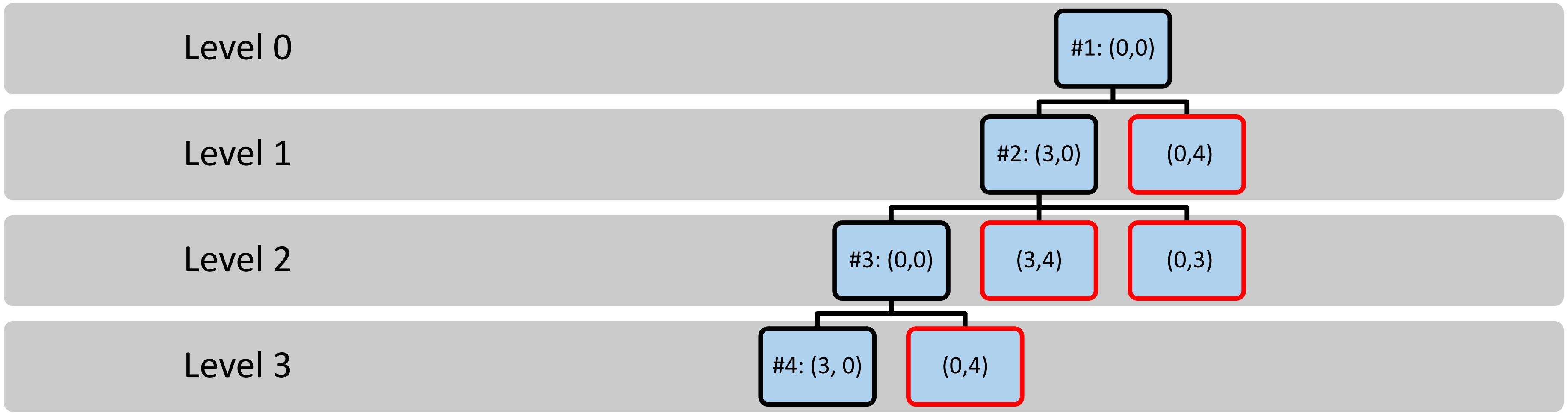
- Place the new paths that extend the current path at the front of OPEN.

  WaterJugs. Start = (0,0), Goal = (*,2)

Green = Newly Added. Red expanded next

1. OPEN = {<(0,0)>}
2. OPEN = {**<(0,0), (3,0)>, <(0,0), (0,4)>**}
3. OPEN = {**<(0,0),(3,0),(0,0)>, <(0,0),(3,0),(3,4)>, <(0,0),(3,0),(0,3)>**, <(0,4),(0,0)>}
4. OPEN = {**<(0,0),(3,0),(0,0),(3,0)>, <(0,0),(3,0),(0,0),(0,4)>** <(0,0), (3,0), (3,4)>, <(0,0),(3,0),(0,3)>, <(0,0),(0,4)>}

Depth-First Search



- Red nodes are backtrack points (these nodes remain on open).

Depth-First Properties

Completeness?

- Infinite paths? Cause incompleteness!

- Prune paths with cycles (duplicate states)

We get completeness if state space is finite

Optimality?

No!

Depth-First Properties

Time Complexity?

- $O(b^m)$ where m is the length of the longest path in the state space.
- Very bad if m is much larger than d (shortest path to a goal state), but if there are many solution paths it can be much faster than breadth first. (Can by good luck bump into a solution quickly).

Depth-First Properties

- Depth-First Backtrack Points = unexplored siblings of nodes along current path.
  - These are the nodes that remain on open after we extract a node to expand.

Space Complexity?

- O(bm), linear space!
  - Only explore a single path at a time.
  - OPEN only contains the deepest node on the current path along with the backtrack points.
    - How many backtrack points are their for a depth k node?
- A significant advantage of DFS

Depth-Limited Search

Depth Limited Search

- Breadth first has space problems. Depth first can run off down a very long (or infinite) path.

Depth limited search

- Perform depth first search but only to a pre-specified depth limit D.
  - THE ROOT is at DEPTH 0. ROOT is a path of length 1.
- No node representing a path of length more than D+1 is placed on OPEN.
- We "truncate" the search by looking only at paths of length D+1 or less.

- Now infinite length paths are not a problem.

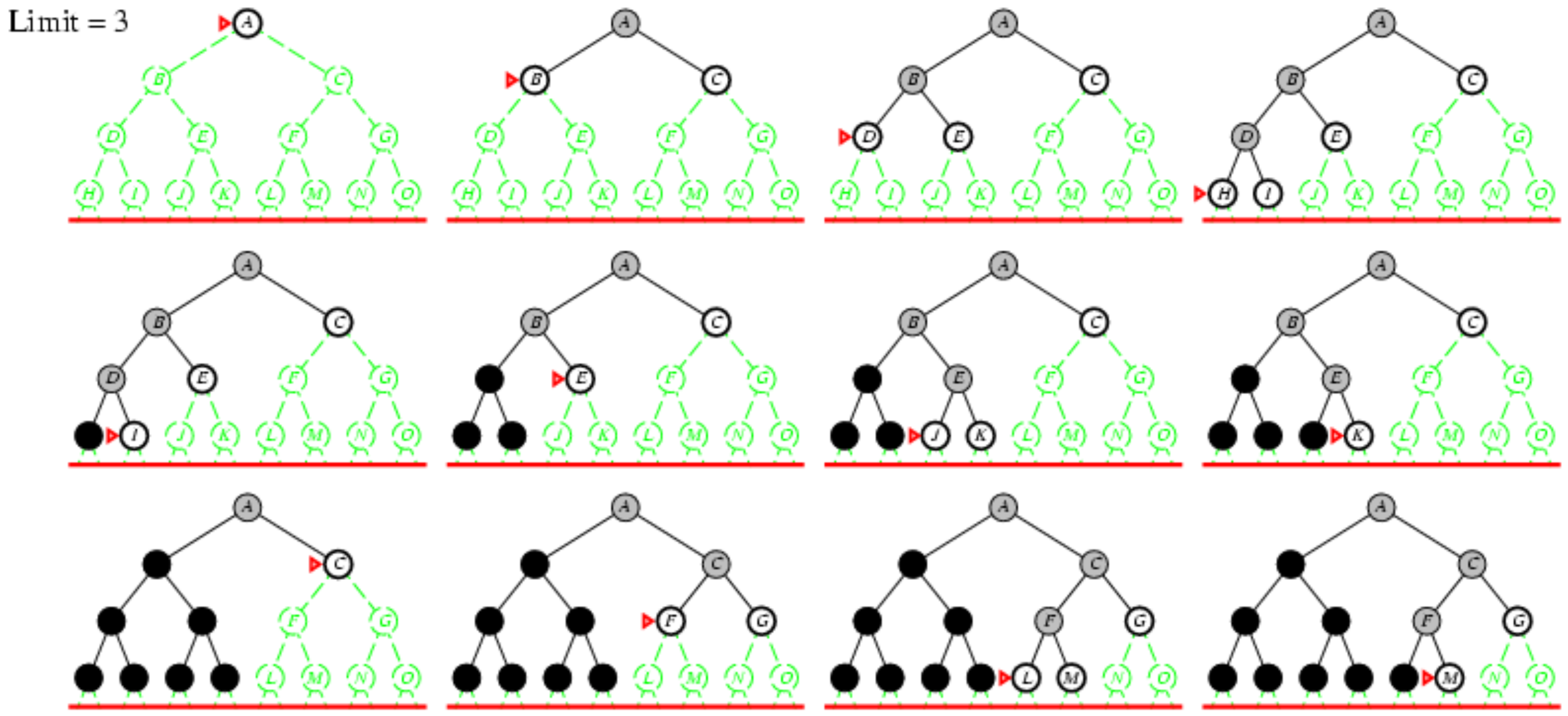
- But will only find a solution if a solution of DEPTH $\leq$ D exists.

Depth Limited Search

```
DL_Search(open, successors, goal?, maxd):
    open.insert(<start>)          #OPEN MUST BE A STACK FOR DFS
    cutoff = false
    while not open.empty():
        n = open.extract()         #remove node from OPEN
        state = n.state()
        if (goal?(state)):
            return (n,cutoff)      #n is solution
        if depth(n) < maxd:        #Only successors if depth(n) < maxd
            for succ in sucessors(state):
                open.insert(<n,succ>)
        else:
                cutoff= true.      #some node was not
                                   #expanded because of depth
                                   #limit.
    return  (false, cutoff)
```

CSC384, University of Toronto

Depth Limited Search Example



Iterative Deepening Search

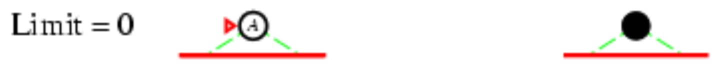
Iterative Deepening Search

- Solve the problems of depth-first and breadth-first by extending depth limited search
- Starting at depth limit L = 0, we iteratively increase the depth limit, performing a depth limited search for each depth limit.
- Stop if a solution is found, or if the depth limited search failed without cutting off any nodes because of the depth limit.
  - If no nodes were cut off, the search examined all paths in the state space and found no solution → no solution exists.

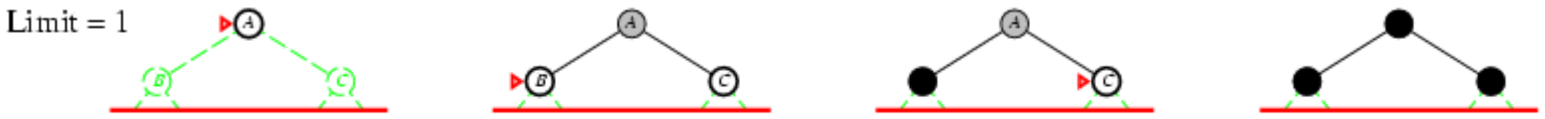
Iterative Deeping Search

```
ID_Search(open, successors, goal?):
    maxd = 0
    while true:
        (n, cutoff) = DL_Search(open, successors, goal?, maxd)
        if n:
            return n
        elif not cutoff:          #no nodes at deeper levels exit
            return fail
        else:
            maxd = maxd + 1
```

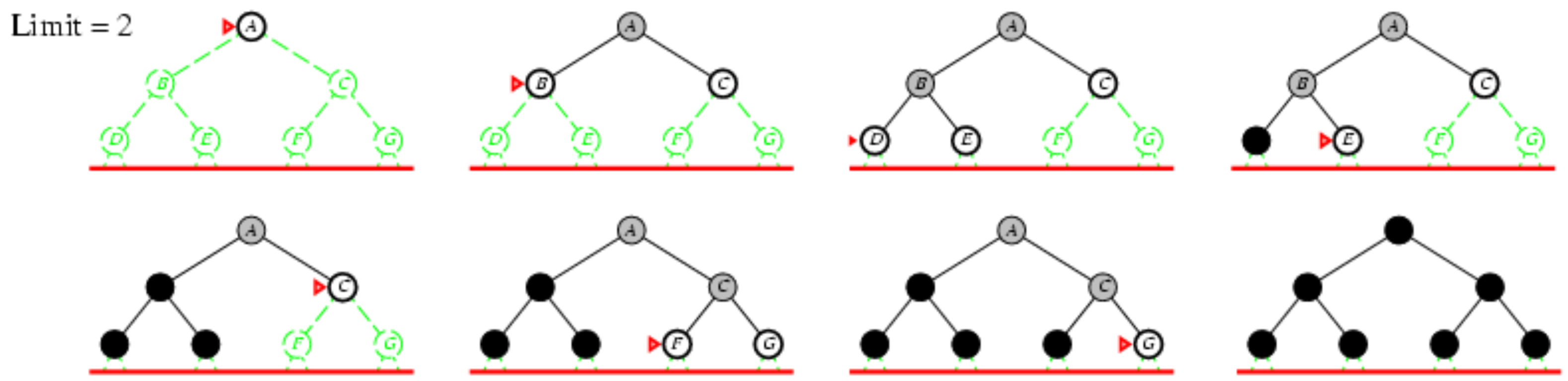
Iterative Deepening Search Example



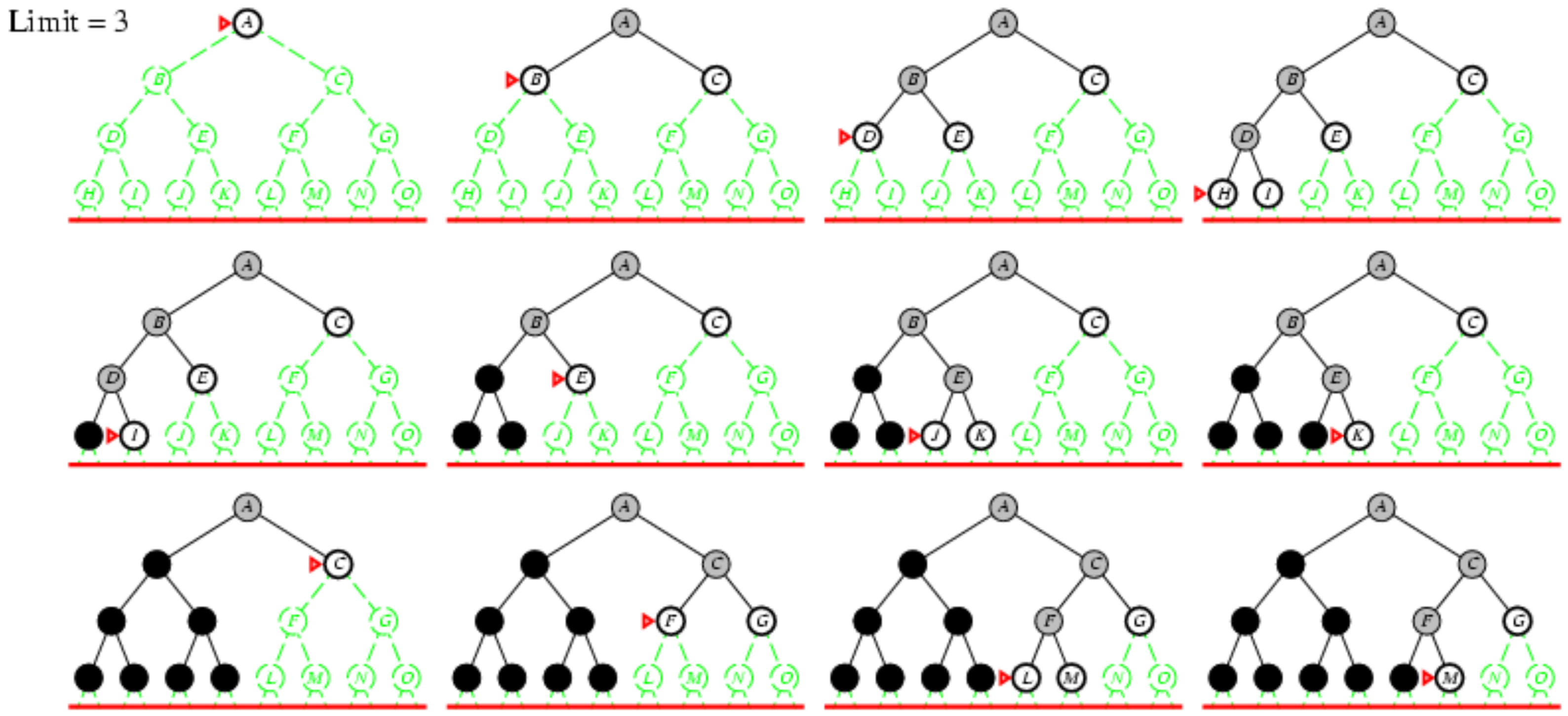
Iterative Deepening Search Example



Iterative Deepening Search Example



Iterative Deepening Search Example



Iterative Deepening Search Properties

Completeness?

- Yes if a minimal depth solution of depth d exists.
  - What happens when the depth limit L=d?
  - What happens when the depth limit L<d?

Time Complexity?

Iterative Deepening Search Properties

Time Complexity

- $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- E.g. b=4, d=10
    - $(11)*4^0 + 10*4^1 + 9*4^2 + \ldots + 4^{10} = 1{,}864{,}131$
    - $4^{10} = 1{,}048{,}576$
    - Most nodes lie on bottom layer.

BFS can explore more states than IDS!

- For IDS, the time complexity is
  - $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

- For BFS, the time complexity is
  - $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$

E.g. b=4, d=10

- For IDS
  - $(11)*4^0 + 10*4^1 + 9*4^2 + \ldots + 4^{10} =$ 1,864,131 (states generated)
- For BFS
  - $1 + 4 + 4^2 + \ldots + 4^{10} + 4(4^{10} - 1) =$ 5,592,401 (states generated)
  - In fact IDS can be more efficient than breadth first search: nodes at limit are not expanded. BFS must expand all nodes until it expands a goal node. So a the bottom layer it will add many nodes to OPEN before finding the goal node.

Iterative Deepening Search Properties

Space Complexity

- O(bd) Still linear!

Optimal?

- Will find shortest length solution which is optimal if costs are uniform.
- If costs are not uniform, we can use a “cost” bound instead.
  - Only expand paths of cost less than the cost bound.
  - Keep track of the minimum cost unexpanded path in each depth first iteration, increase the cost bound to this on the next iteration.
  - This can be more expensive. Need as many iterations of the search as there are distinct path costs.

Path/Cycle Checking

Path Checking

If nk is the path $<s_0,s_1,\ldots,s_k>$ and we expand s_k to obtain child successor state c, we have

$<s_0,s_1,\ldots,s_k,c>$

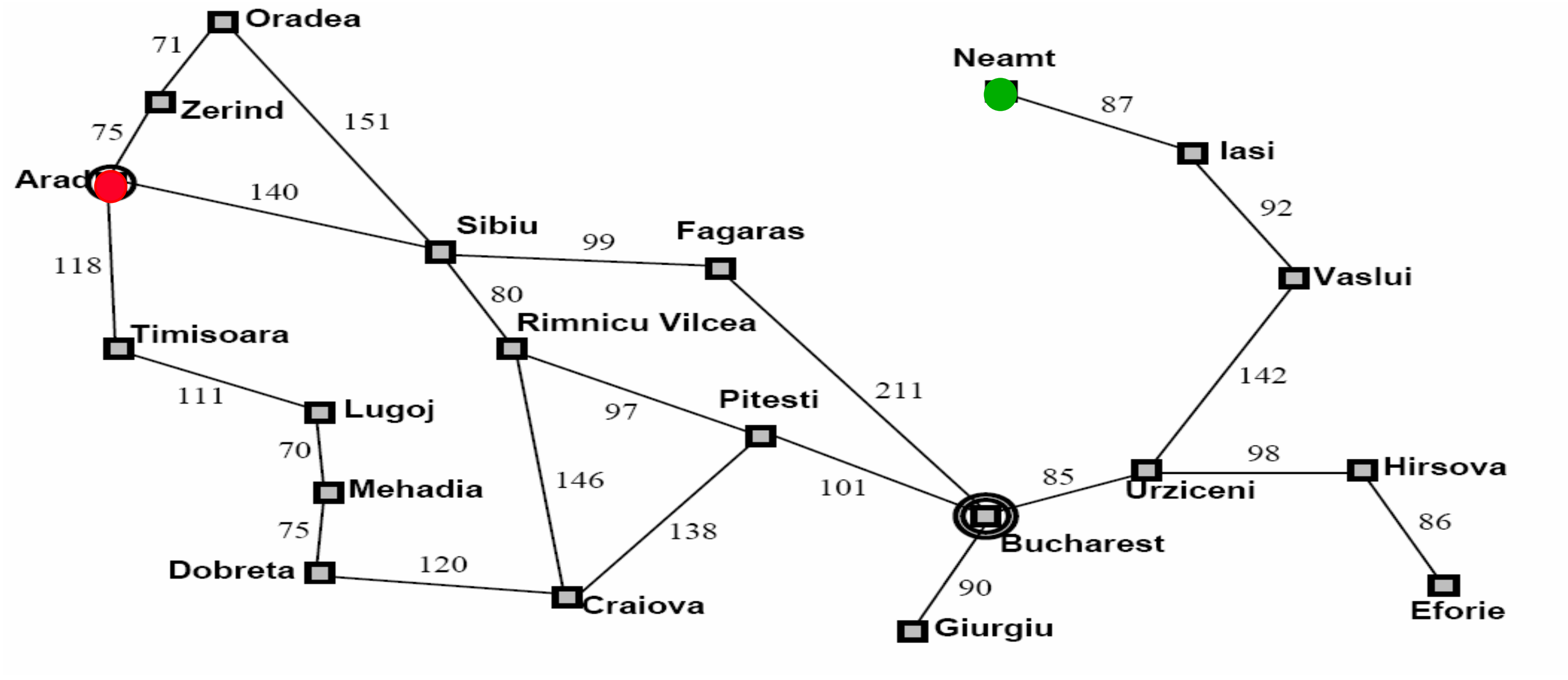
As the path to "c".

We write such paths as <n,c> where n is the prefix and c is the final state in the path.

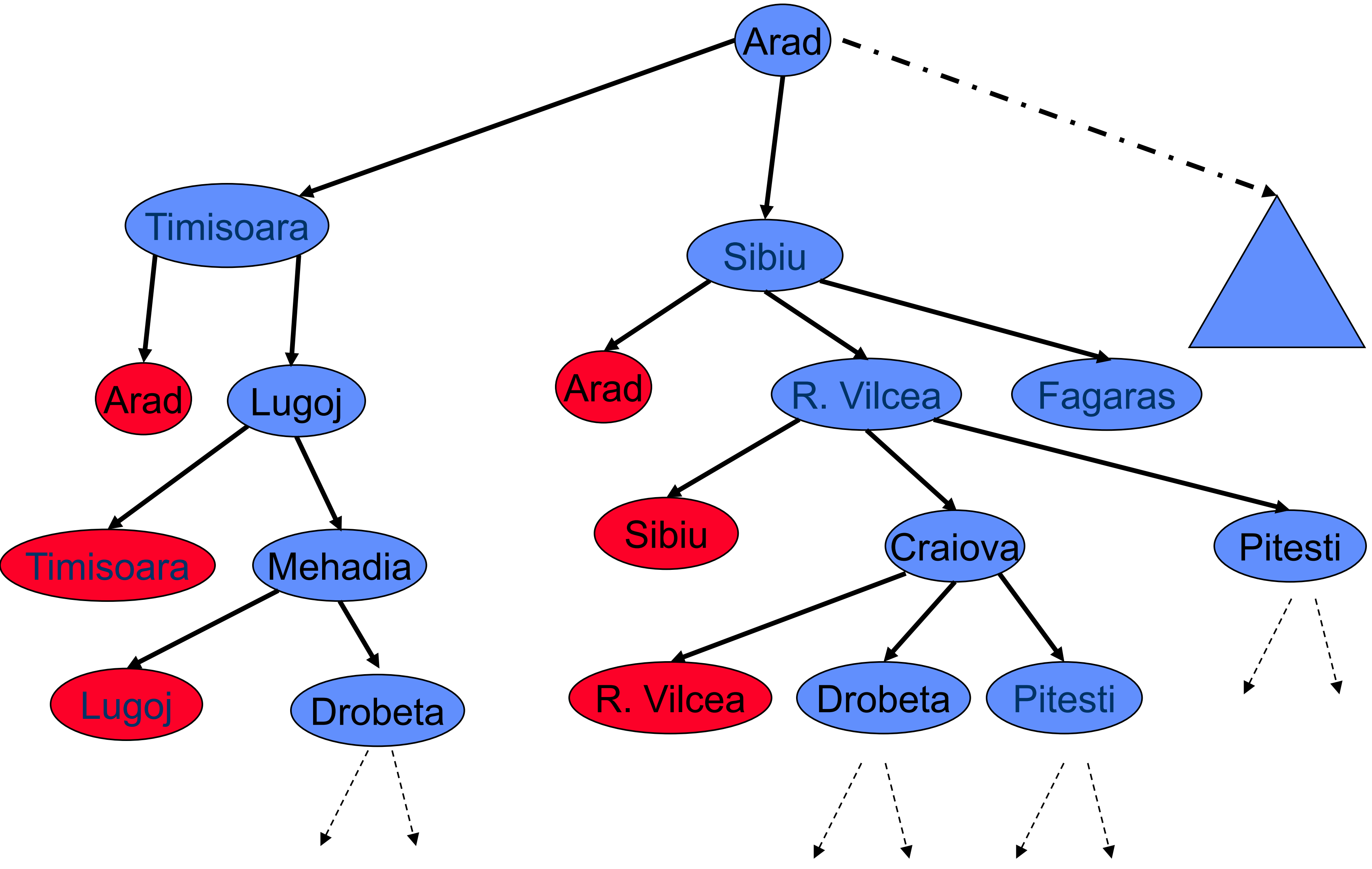
Path checking:

- Ensure that the state c is not equal to the state reached by any ancestor of c along this path. $c \notin \{s_0,s_1,\ldots,s_k\}$
- Paths are checked in isolation!

Example: Arad to Neamt



Path Checking Example



Search with Path Checking

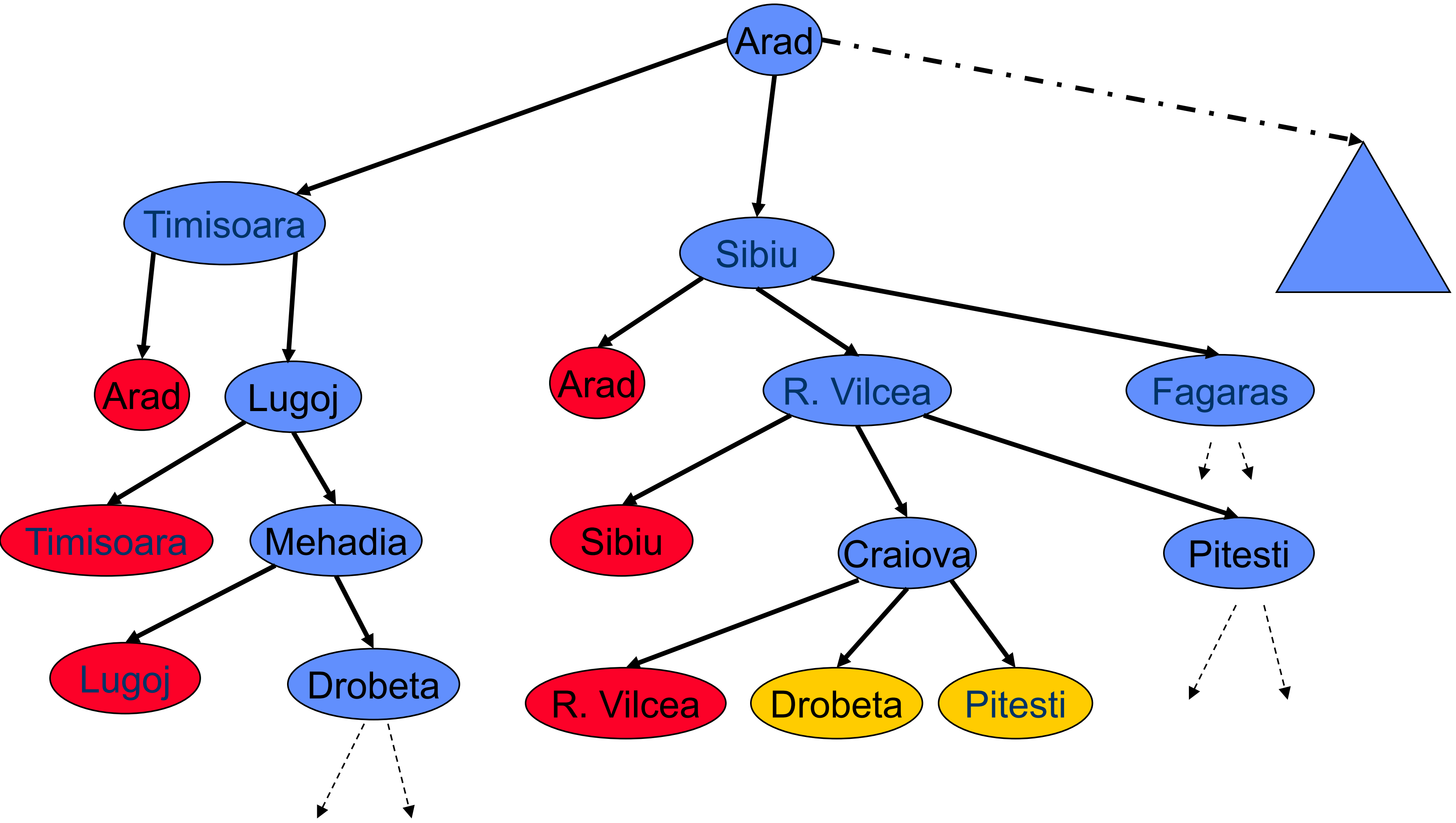
```
Search(open, successors, goal? ):
    open.insert(<start>)
    while not open.empty():
        n = open.extract()        #remove node from OPEN
        state = n.state()
        if (goal?(state)):
            return n                  #n is solution
        for succ in sucessors(state):
            if not succ in <n>:     #Don't put cyclic paths on OPEN
                open.insert(<n,succ>)
    return false
```

Cycle Checking

Cycle Checking

- Keep track of all states added to OPEN during the search (i.e., end state of every path added to OPEN)
- When we expand n_k to obtain successor state c
  - Ensure that c is not equal to **any** previously seen state.
  - If it is we do not add the path <nk,c> to OPEN.
- This is called cycle checking, or multiple path checking.
- What happens when we utilize this technique with depth-first search?
  - **What happens to space complexity?**

Cycle Checking Example (BFS)



Cycle Checking

- Higher space complexity (equal to the space complexity of breadth-first search).
- There is an additional issue when we are looking for an optimal solution
  - If we reject a node <n,c> because we have previously seen state c via a different path it could be that <n,c> is a shorter path to c that we had previously seen.
  - Solution is to also keep track of the minimum cost path to each seen state.

Cycle Checking

- Keep track of each state as well as minimum known cost of a path to that state.
- If we find a longer path to a previously seen state, we don’t add it to OPEN
- If we find a shorter path to a previously seen state, we add it to OPEN and
  - Remove other more expensive paths to the same state OR
  - Lazily remove these more expensive paths if and only if we decide to expand them.

Cycle Checking—ensuring optimality

```
Search(open, successors, goal? ):
    open.insert(<start>)
    seen = {start : 0}              #seen is dict storing min cost
    while not open.empty():
        n = open.extract()
        state = n.state()
        if cost(n) <= seen[state]:
            #Expand if n is cheapest known path to state
            if (goal?(state)):
                return n
            for succ in sucessors(state):
                if not succ in seen or cost(<n,succ>) < seen[succ]:
                    #Add this path to succ to open if this is  or the
                    #the first cheapest path to succ found.
                    open.insert(<n,succ>)
                    seen[succ] = cost(<n,succ>)
    return false
```

Heuristic Search (Informed Search)

Heuristic Search

- In **uninformed search**, we don't try to evaluate which of the nodes on OPEN are most promising. We never "look-ahead" to the goal.

  E.g., in uniform cost search we always expand the cheapest path. We don't consider the cost of getting from the end of the current path to the goal

- Often we have some other knowledge about the merit of nodes, e.g., going the wrong direction in Romania.

Heuristic Search

Merit of an OPEN node: different notions of merit.

- If we are concerned about the cost of the solution, we might want to consider how costly it is to get to the goal from the end state of that node.
- If we are concerned about minimizing computation in search we might want to consider how easy it is to find the goal from the end state of that node.
- We will focus on the "cost of solution" notion of merit.

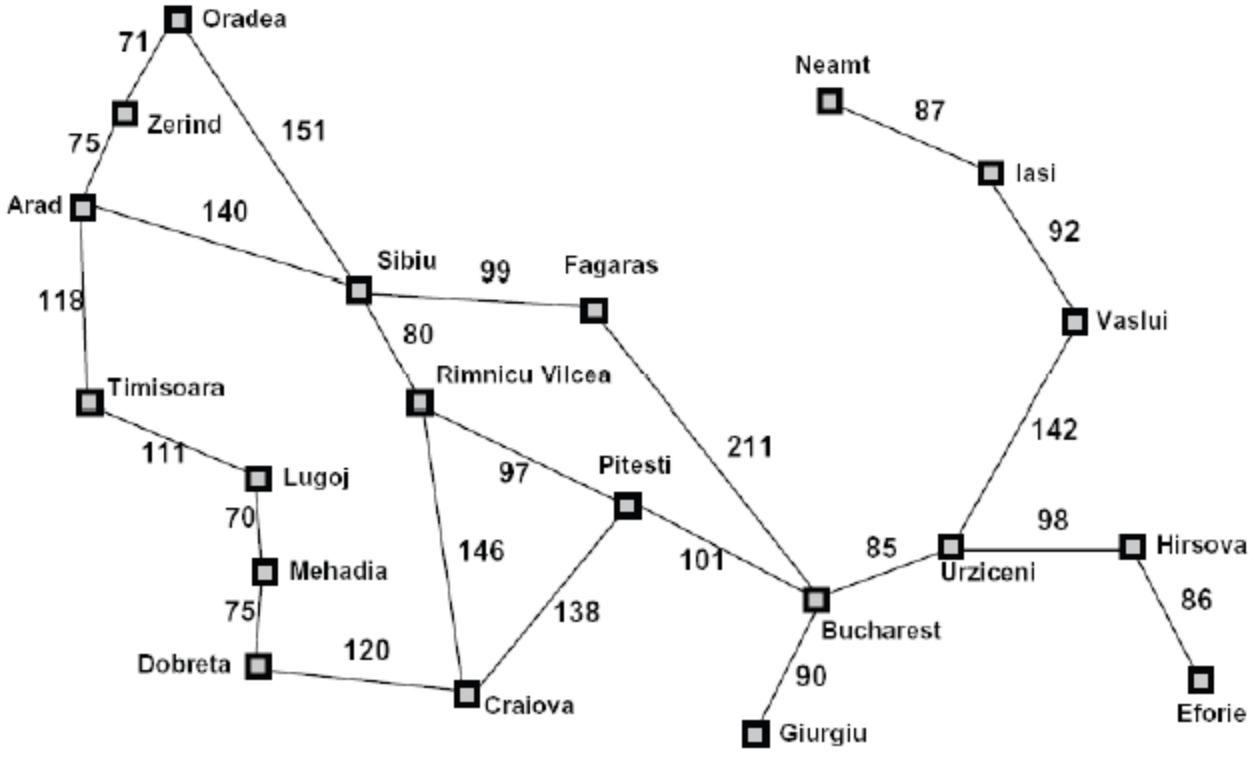
Heuristic Search

- The idea is to develop a domain specific heuristic function h(n).

- h(n) guesses the cost of getting to the goal from node n (i.e., from the final state of the path represented by n).
- h(n) is a function only of n.state()!
  If n1.state() = n2.state() then h(n1) = h(n2)

- There are different ways of guessing this cost in different domains.
  - heuristics are domain specific.
  - Alpha Go exploited machine learning techniques to compute the heuristic estimate of a state (go board configuration)

Example: Euclidean distance



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

Planning a path from Arad to Bucharest, we can utilize the straight line distance from each city to our goal. This lets us plan our trip by picking cities at each time point that minimize the distance to our goal.

Heuristic Search

- If $h(n_1) < h(n_2)$ this means that we guess that it is cheaper to get to the goal from n_1 than from n_2.

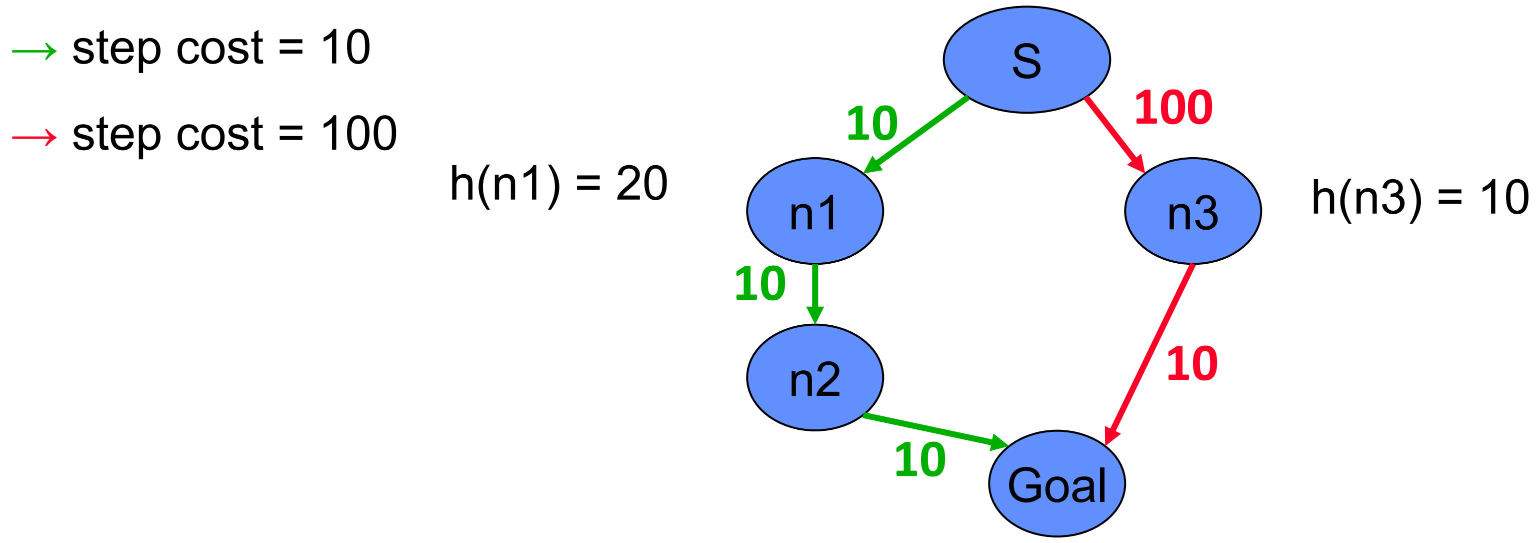
- We require that
  - $h(n) = 0$ for every node n whose final state satisfies the goal.

    Zero cost of achieving the goal from node that already satisfies the goal.

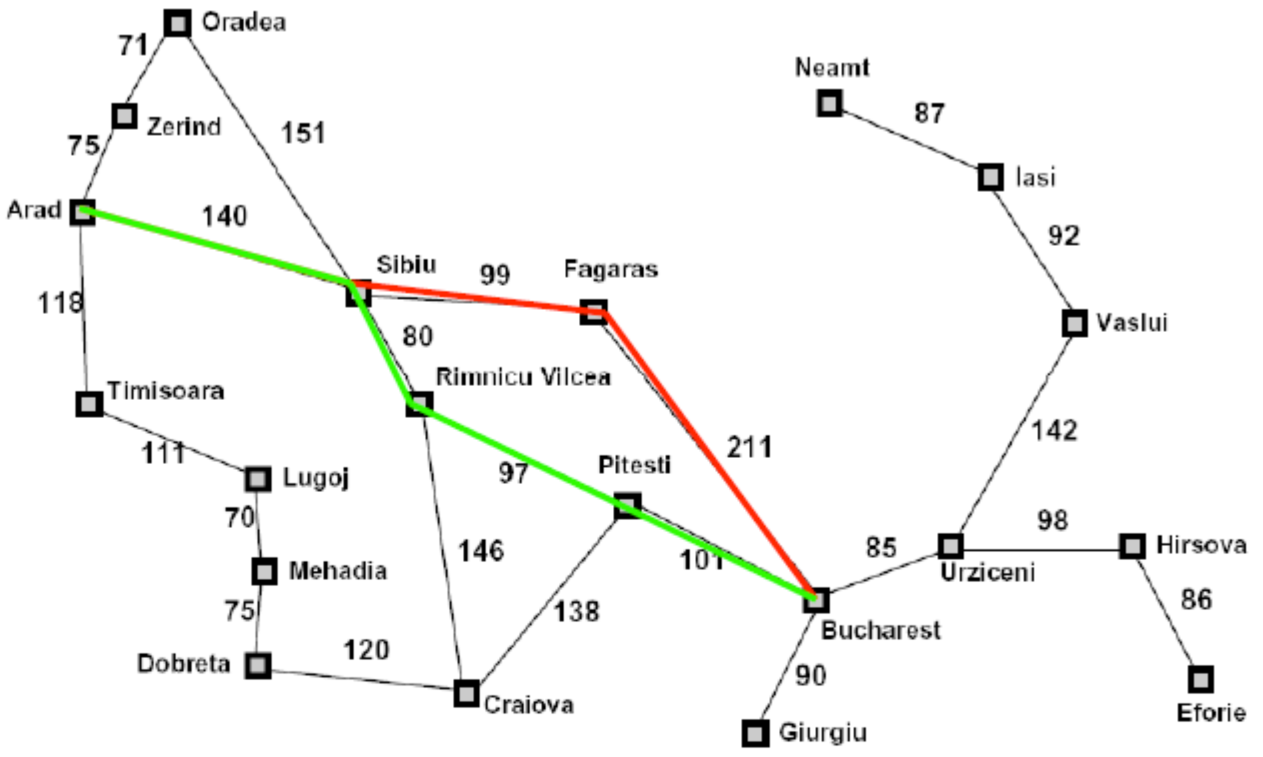
Using only h(n): Greedy best-first search

- We use h(n) to rank the nodes on OPEN
  - Always expand node with lowest h-value.
- We are greedily trying to achieve a low cost solution.

- However, this method **ignores the cost of getting to n**, so it can be lead astray exploring nodes that cost a lot but seem to be close to the goal:



Greedy best-first search example



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

Greedy best-first search

- Greedy search can be very efficient in practice at finding solutions...require developing a good heuristic.
- Greedy search is incomplete—if it fails to find a solution this does not mean a solution does not exist.
- The solution returned by a greedy search can be very far from optimal—not easy to control greedy search so that it finds close to optimal solutions.

    One method is to add some randomness to the heuristic and compute many different solutions taking the best.

A* search

- Take into account the cost of getting to the node as well as the heurstic estimate of the cost of getting to the goal from the node.

- Define an evaluation function f(n)

  f(n) = g(n) + h(n)

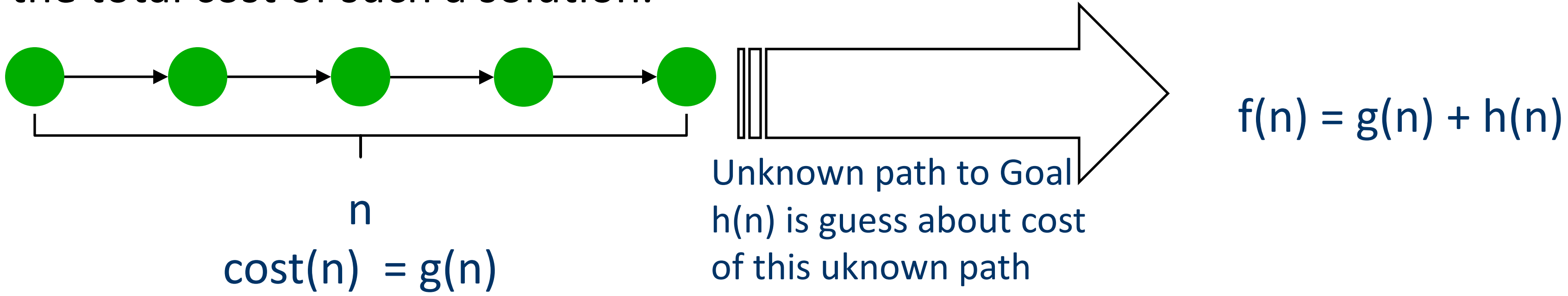
  - g(n) is the cost of the path represented by node n
  - h(n) is the heuristic estimate of the cost of achieving the goal from n.

A* search

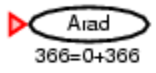
- h(n) is a function only of n.state()—the end state reached by path n. So if n1.state() = n2.state() then h(n1) = h(n2)
- g(n) is a function of the path. It is the sum of the costs of the actions on the path n.

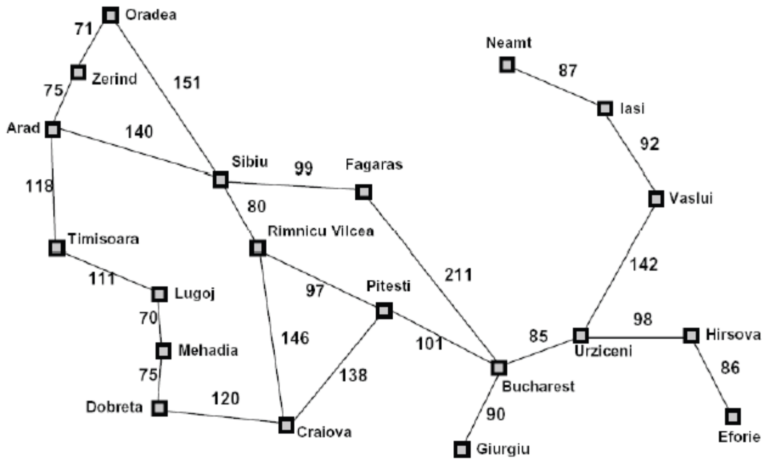
- Always expand the node with lowest f-value on OPEN.

- The f-value, f(n) is an estimate of the cost of getting to the goal via the node (path) n.
  - I.e., we first follow the path n then we try to get to the goal. f(n) estimates the total cost of such a solution.



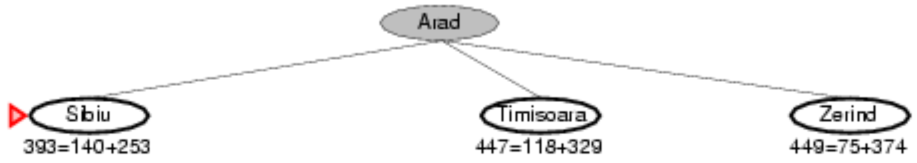
A* example

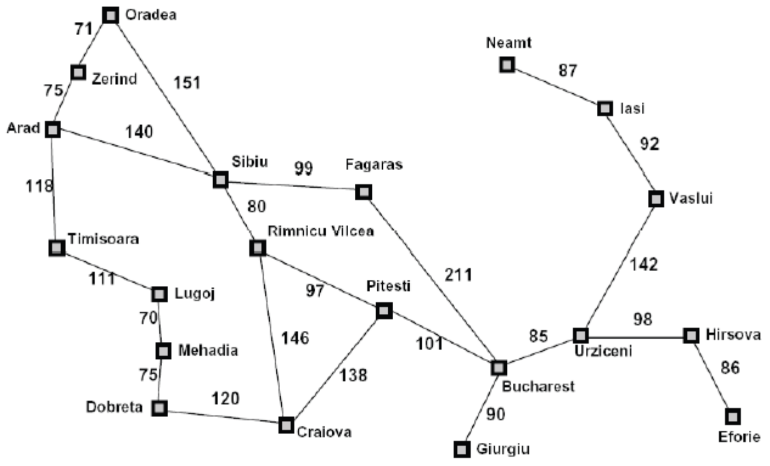




Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* example

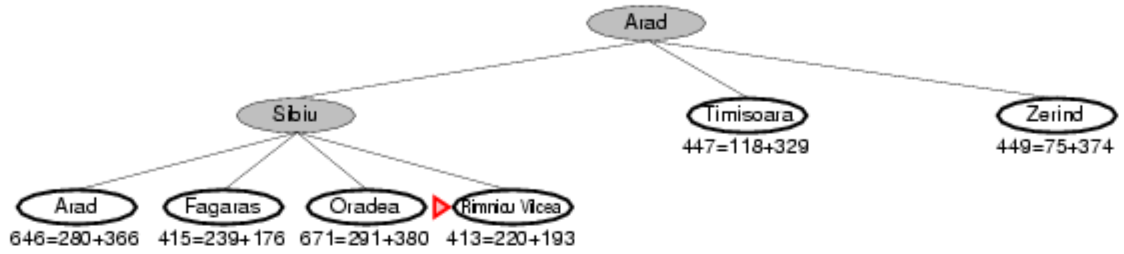


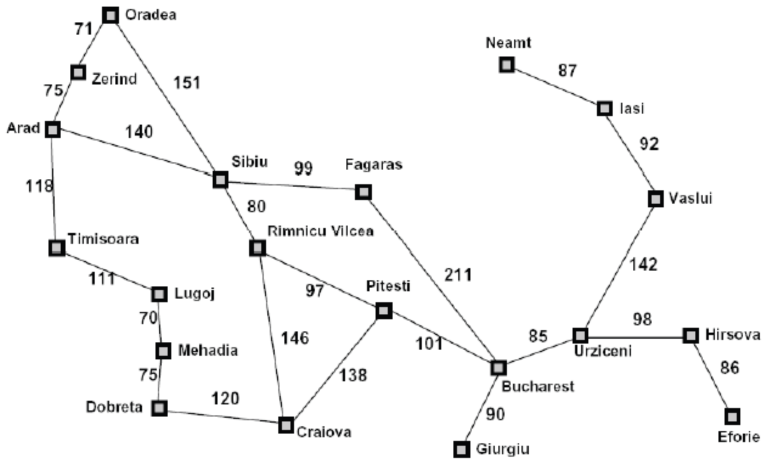


Straight−line distance to Bucharest

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

A^* example

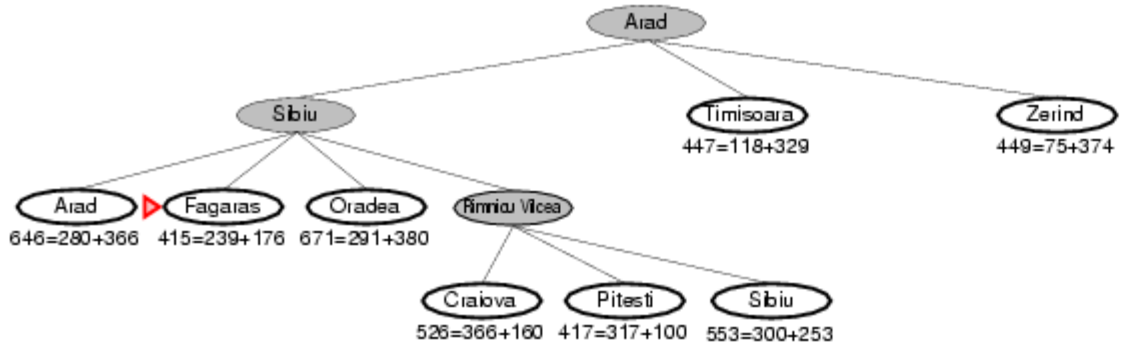


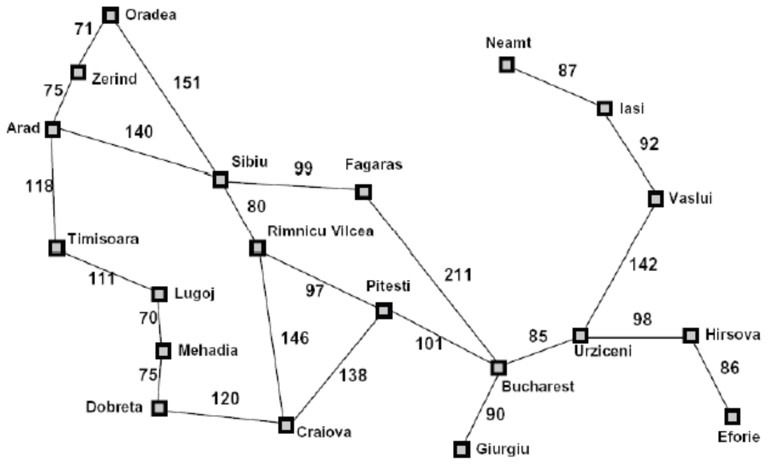


Straight−line distance
to Bucharest

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

A* example

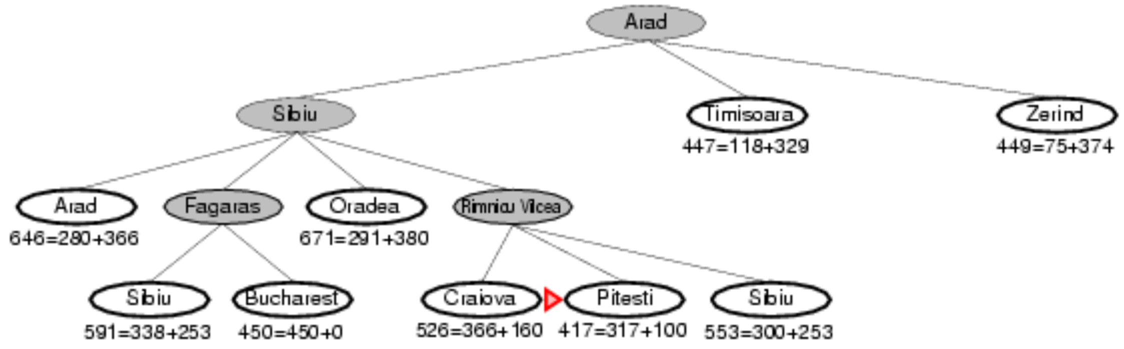


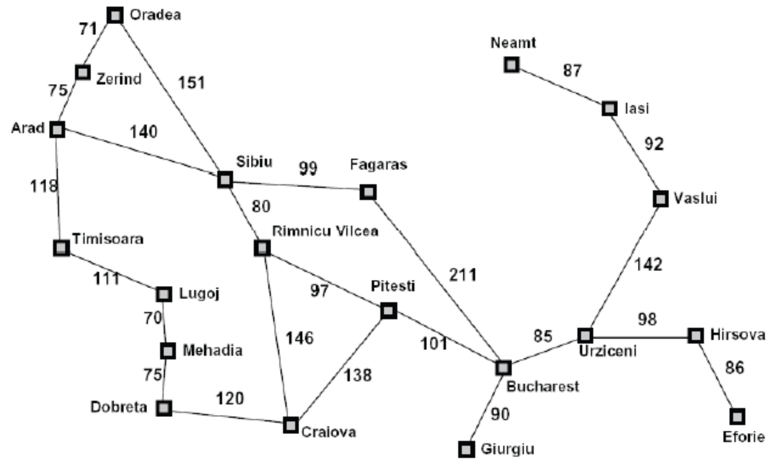


Straight−line distance to Bucharest

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

A* example

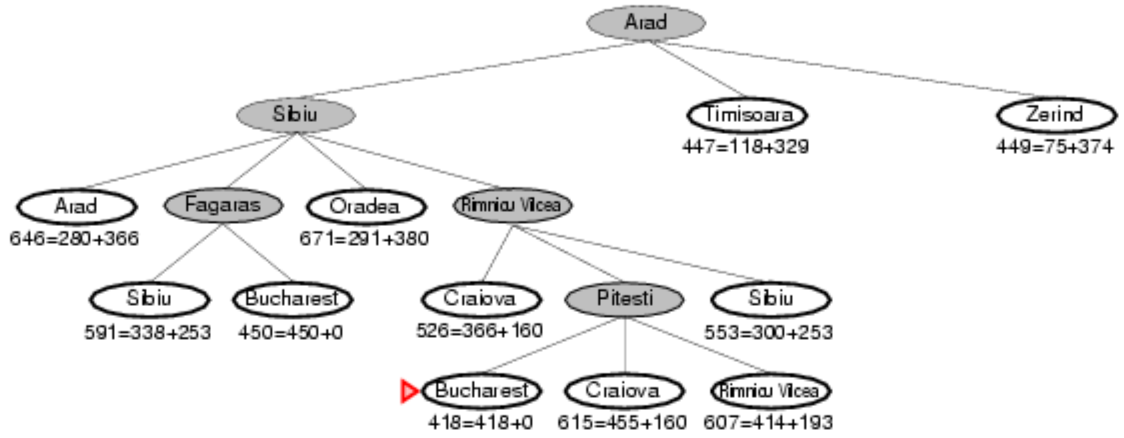


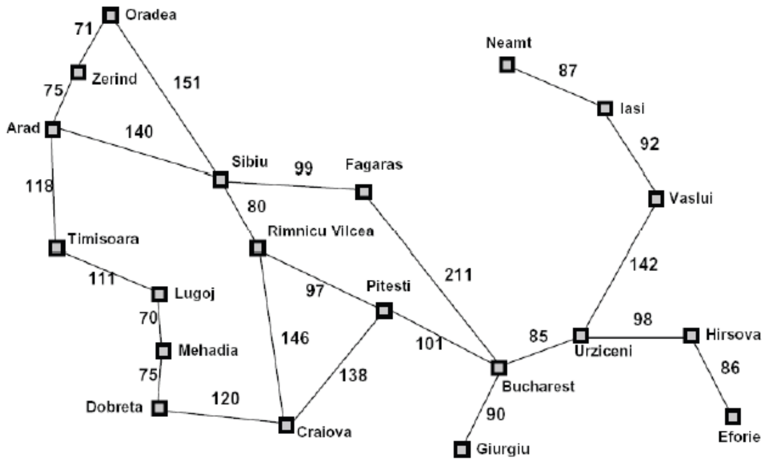


Straight−line distance to Bucharest

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

A* example





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

A* search Breaking Ties

- Suppose we have multiple nodes with the lowest f-value. Is it important which of these we choose?
- **YES!** When we have admissible heuristics (the notion of admissibility is coming next) this can give an exponential speed-up.

A* search Breaking Ties

- Given that h(n) is an estimate of “how far away the goal is”, and g(n) is the cost already incurred for traversing the path n.
  - It makes intuitive sense to break ties by largest g-value (equivalently largest h-value)
  - That is, if f(n1) = f(n2)

    $f(n1) = g(n1) + h(n1) = g(n2) + h(n2) = f(n2)$

    so $g(n1) > g(n2) \Leftrightarrow h(n1) < h(n2)$
  - So intuitively we would prefer to expand n1 as it is closest to the goal, and is estimated to have equal total cost.
- Breaking ties in this way can reduce A*’s search by an exponential factor! (more on this in tutorial).

Formal Properties of A*

- We want to analyze the behavior of the resultant search.
  - Completeness, time and space, optimality?
- We start with a simple result concerning paths in the search space that will be useful in proving other results.

Reminder—Paths in the search space

- Remember that a path is a sequence of states $<s_0, ..., s_k>$ generated by a sequence of actions $<a_0, ..., a_{k-1}>$. Where s_{i+i} is the result of applying action a_i to state s_i.

- The cost of the path is the sum of the costs of the actions a_i.
- In our discussion we will mostly leave the sequence of actions implicit—it exists and it determines the cost of the path, but we only need to talk about the states.

Paths in the Search Space.

Proposition 1. For every path n = <s_0, ..., s_k> in the search space, at any time in the running of A* either:

a. <s_0,...,s_k> has been expanded by A* OR
b. Some prefix <s_0, ..., s_i> of n is on OPEN.

Proof: By induction over the number of node expansions A* has done.

Base Case (zero node expansions): A* always starts off with <s_0> on OPEN (where s_0 is the initial state). <s_0> is a prefix of **<s_0, ..., s_k>** so (b) is true.

Paths in the Search Space.

Induction. Assume true after k node expansions, prove true after the k+1 node expansion.

By induction after the k-th node expansion either (a) or (b) is true. If (a) then n has already been expanded, if (b) then a prefix of n is on open after the k-th expansion.

If (a) is true (n has been expanded) it clearly stays true after k+1 node expansions.

If (b) is true then let n_i= **<s_0, ..., s_i>** be the prefix of an optimal path that is on OPEN after k node expansions. Two possibilities:

1. A* expands some node x ≠ n_i as the k+1 node expansion. In this case n_i a prefix of n is still on OPEN.
2. A* expands n_i as the k+1 node expansion. Then either
   a. n_i = n and now (a) is true-–n has been expanded OR
   b. One of the successors of n_i is n_{i+1} = **<s_0, ..., s_{i+1}>**. So now n_{i+1} is on OPEN, and again (b) is true, i.e., a prefix of n remains on OPEN.

Some Properties of A* depend on conditions on h(n)

- We want to analyze the behavior of the resultant search.
  - Completeness, time and space, optimality?

- We can obtain some results, but others depend on the properties of h(n).

- First, what happens if there is no path from the initial state to a state satisfying the goal (i.e., there is no solution)?

No Solution

- A* as well as all of the uninformed search algorithms, depth first (DFS), breadth first (BFS), uniform cost (UCS), and iterative deepening (IDS) have the same behaviour when there is no solution:
  - If there are an **infinite** number of different states these algorithms never terminate.
  - If there are a finite number of different states **and** we do cycle checking (either path checking of full cycle checking) they we will eventually terminate correctly declaring that there is no solution. (Note costs are always $\geq \epsilon > 0$)

Solutions

- Now consider the case where there are solutions

Completeness of A*

Theorem 1. A* will always find a solution if one exists (i.e., it is complete) as long as

- Every state has a finite number of successors (the branching factor is finite)—**note state space might still be infinite.**
- Every action has finite cost ≥ ε > 0.
- h(n) is finite for every path n that can be extended to reach a goal state.

Proof: A* If a solution path **n** exists, then by **Proposition 1** at all times either

(a) **n** been expanded by **A*** OR

(b) a prefix **n** is on OPEN

If (a) then **A*** has halted and returned **n** as the solution when it expands it and the theorem holds.

Completeness of A*

If (b) then let the prefix on OPEN be n_i

> n_i must have a finite f-value. It has a finite cost (g-value) and its h-value is finite.

As **A*** continues to run, the f-value of the nodes on OPEN eventually increase.
At every iteration we remove a node of lowest f-value and replace it with its successors that have larger g-values. So the g-values of the nodes on OPEN must eventually increase without bound as **A*** continues to run, and thus so must the f-values.

Note. the h-value of the successors might be lower, so the new successors might have lower f-value. But as we continue eventually the g-value increases so much that the f-values of all nodes on OPEN must increase.

Completeness of **A***

So, eventually either
(a) **A*** terminates because it found a solution OR
(b) n_i becomes the node on OPEN with lowest f-value

If (a) then the theorem holds—**A*** finds a solution.

If (b) then n_i is expanded and either

1) n_i = **n** and **A*** returns **n** as a solution and the theorem holds OR
2) n_i is replaced by its successors one of which is a longer prefix of **n,** n_{i+1}

Completeness of A*

Applying the same argument to n_{i+1} we see that if **A*** continues to run without finding a solution it will eventually expand every prefix of the path **n** and it must terminate when it expands **n.**

Completeness of A*

Note. A* will eventually find a solution even in infinite search spaces (as long as the branching factor is finite), even if we don’t use cycle checking, and no matter what function h we use (as long it is always assigns a finite value)

But we want to go beyond completeness and develop conditions that ensure that **A*** finds **optimal solutions**.

One condition that ensures optimality is when the heuristic **h** is **admissible.**

Conditions on h(n): Admissible

- We assume that the cost of any action a is ≥ ε > 0: the cost of any transition is greater than zero and can't be arbitrarily small.

Let **h*(n)** be the **cost of an optimal path** from n to a goal node. Then an admissible heuristic satisfies the condition

$$h(n) \leq h^*(n)$$

- an admissible heuristic under-estimates the true cost to reach the goal, i.e., it is optimistic

- Hence $h(g) = 0$, for any goal node g since $h^*(g) = 0$
- Also define $h^*(n) = \infty$ if there is no path from n to a goal node

Intuition behind admissibility

$h(n) \leq h^*(n)$ means that the search won't miss any promising paths.

- If it really is cheap to get to a goal via n (i.e., both $g(n)$ and $h^*(n)$ are low), then $f(n) = g(n) + h(n)$ will also be low, and the search won't ignore n in favor of more expensive options.

- This can be formalized to show that admissibility implies optimality.

Admissible Heuristics → Optimal solutions

- Now we prove formally that if h(n) is admissible, then A* (ordering OPEN by f(n) = g(n) + h(n)) always finds an optimal solution.
- As before we must assume that
  - No state has an infinite number of successors, i.e., the branching factor b is finite. (But the state space can be infinite)
  - The cost of any action is $\geq \varepsilon > 0$.

Admissible Heuristics ➔ Optimal solutions

- If there exists a solution, then there must exist an optimal (perhaps many) solution.
- Start with a few observations about any optimal path.
- Let **n =** $<s_0, s_1, \ldots., s_n>$ be an optimal solution, i.e., a path from the initial state s_0 to a state s_n that satisfies the goal such that n has cost ≤ cost of any other path to a goal satisfying state.
- Let

$n_0 = <s_0>$
$n_1 = <s_0, s_1>$
…
$n_i = <s_0, s_1, \ldots, s_i>$
…
$n_k = <s_0, s_1, \ldots, s_k>$

That is, the prefixes of various lengths of this optimal path

Admissible Heuristics → Optimal solutions

- Observe that
  - $n_i = <s_0, s_1, ..., s_i>$ is an optimal path from state s_0 to state s_i
  - $<s_{i+1}, ..., s_n>$ is an optimal path from state s_{i+1} to any goal node. That is, there is no cheaper way of getting from s_{i+1} to a goal node than $<s_{i+1}, ..., s_n>$
  - Every sub-path $<s_i, ..., s_j>$ is an optimal path from s_i to s_j.

If not we could replace the sub-path with a cheaper path and obtain a cheaper path from s_0 to a goal node which would contradict that **n =** $<s_0, s_1, ..., s_n>$ is optimal.

Admissible Heuristics → Optimal solutions

Proposition 2: A* with an admissible heuristic never expands a node with f-value greater than the cost of an optimal solution.

Proof. Let **C*** be the cost of an optimal solution.
(i.e., = cost(<s_0, s_1, …., s_n>).

A* always expands a node on OPEN that has lowest f-value.

Let n = <s_0, s_1, ..., s_n> be an optimal solution (so s_n satisfies the goal condition)

By **Proposition 1**, we know that since **A*** terminates if it expands **n,** at every stage OPEN always contains a prefix of **n =** <s_0, s_1, ..., s_n>

So A* never expands a node with f-value greater than the f-value of that prefix.

Now we observe that when the heuristic is admissible for any prefix of **n** say n_i we have

$$C^* \geq f(n_i)$$

So **A*** never expands a node with f-value > $f(n_i)$, i.e., with f-value > **C*** and the proposition is proved.

Admissible Heuristics ➔ Optimal solutions

$C^* = \text{cost}(<s_0, s_1, ..., s_n>)$

$= \text{cost}(<s_0,...,s_i>) + \text{cost}(<s_i, ..., s_n>)$

$= g(n_i) + h^*(n_i)$ (1)

$\geq g(n_i) + h(n_i) = f(n_i)$ (2)

(1) $g(n_i)$ is the $\text{cost}(n_i) = \text{cost}(<s_i, ..., s_i>)$

$h^*(n_i)$ is the cost of an optimal path from s_i to any goal node, which must be equal to $\text{cost}(<s_i, ..., s_n>)$ since $<s_0, ..., s_n>$ is optimal

(2) $h^*(n_i) \geq h(n_i)$ since h is admissible.

Admissible Heuristics → Optimal solutions

Theorem 2. A* with an admissible heuristic always find an optimal solution if a solution exists.

Proof:

- If a solution exists then by **Theorem 1**, **A*** will terminate by expanding a solution path **n**.
- By **Proposition 2,** f(**n**) ≤ **C*** (the cost of an optimal solution).
- Since **n** is a solution h(**n**) = 0 so f(**n**) = g(**n**) = cost(**n**).
- **C*** ≤ cost(**n**) = f(**n**) (no solution can have lower cost than the optimal).
- So cost(**n**) = **C*,** i.e., **A*** returns an optimal solution.

Consistency (aka monotonicity)

- A stronger condition than $h(n) \leq h^*(n)$.

- A monotone/consistent heuristic satisfies the triangle inequality: for all nodes n1, n2 and for all actions a

  $$h(n1) \leq C(n1,a,n2) + h(n2)$$

  Where C(n1, a, n2) means the cost of getting from the final state of n1 to the final state of n2 via action a.
- Note that there might be more than one transition (action) between n1 and n2, the inequality must hold for all of them.
- Monotonicity implies admissibility.
  - (forall n1, n2, a) $h(n1) \leq C(n1,a,n2) + h(n2) \rightarrow$ (forall n) $h(n) \leq h^*(n)$

Consistency $\rightarrow$ Admissible

- **Assume consistency:** $h(n) \leq c(n,a,n2) + h(n2)$

  Prove admissible: $h(n) \leq h^*(n)$

Proof:

If no path exists from n to a goal then $h^*(n) = \infty$ and $h(n) \leq h^*(n)$

Else let $<s_n,s_{n+1},\ldots,g>$ be an OPTIMAL path from n to a goal state g. Note the cost of this path is $h^*(n)$, and each subpath $n_i = <s_{n+i},\ldots,g>$ has cost equal to $h^*(n_i)$.

Prove $h(n) \leq h^*(n)$ by induction on the length of this optimal path.

Base Case: n = g

By our conditions on h, h(n) and h(n)* are equal to zero so $h(n) \leq h(n)^*$

Induction Hypothesis: $h(n_1) \leq h^*(n_1)$

$h(n) \leq c(n,a_1,n_1) + h(n_1) \leq c(n,a_1,n_1) + h^*(n_1) = h^*(n)$

Consequences of monotonicity

1. The sequence of f-values of the nodes expanded by A* is non-decreasing. That is, if n_2 is expanded **after** n_1 by A* then f(n1) ≤ f(n2). (Not necessarily true for just an admissible heuristic).
2. With a monotone heuristic, the first time A* expands a path **n =** $<s_0, ..., s_n>$ that reaches the state s_n, **n** must be a minimum cost path to s_n
   - This means that cycle checking need not keep track of the minimum cost of getting to a state found so far. It can simply prune all future paths to a state if that state has already been reached by an expanded path.
   - Again this is not necessarily true for just an admissible heuristic. So with admissible heuristics that are not monotone (or not known to be monotone) we have to keep track of the cost of getting to states as well as the states visited during cycle checking.

A*

Complete.

- Yes, as we saw in **Theorem 1.**

Time and Space complexity.

- When h(n) = 0, for all n h is monotone.
  - A* becomes uniform-cost search!
- It can be shown that when h(n) > 0 for some n and still admissible, the number of nodes expanded can be no larger than uniform-cost.
- Hence the same bounds as uniform-cost apply. (These are worst case bounds). Still exponential unless we have a very good *h*!
- In real world problems, we sometimes run out of time and memory.

Space Problems with A*

- A* has the same potential space problems as BFS or UCS

- IDA* - Iterative Deepening A* is similar to Iterative Deepening Search addresses space issues, but it can require too many depth-first iterations.

IDA* - Iterative Deepening A*

Objective: reduce memory requirements for A*

- Like iterative deepening, but now the "cutoff" is the f-value (g+h) rather than the depth
- At each iteration, the cutoff value is the smallest f-value of any node that exceeded the cutoff on the previous iteration
- Avoids overhead associated with keeping a sorted queue of nodes, and the open list occupies only linear space.
- Two new parameters:
  - curBound (any node with a larger f-value is discarded)
  - smallestNotExplored (the smallest f-value for discarded nodes in a round) when OPEN becomes empty, the search starts a new round with this bound.
    - Easier to expand all nodes with f-value EQUAL to the f-limit. This way we can compute "smallestNotExplored" more easily.

Constructing Heuristics

Building Heuristics: Relaxed Problem

- One useful technique is to consider an easier problem, and let h(n) be the cost of reaching the goal in the easier problem.

- 8-Puzzle moves.
  - Can move a tile from square A to B if
    - A is adjacent (left, right, above, below) to B
    - and B is blank

- Can relax some of these conditions
  1. can move from A to B if A is adjacent to B (ignore whether or not position is blank)
  2. can move from A to B if B is blank (ignore adjacency)
  3. can move from A to B (ignore both conditions).

Building Heuristics: Relaxed Problem

- **#3** *"can move from A to B (ignore both conditions)".*
  leads to the misplaced tiles heuristic.
  - To solve the puzzle, we need to move each tile into its final position.
  - Number of moves = number of misplaced tiles.
  - Clearly h(n) = number of misplaced tiles ≤ the h*(n) the cost of an optimal sequence of moves from n.

- **#1** *"can move from A to B if A is adjacent to B (ignore whether or not position is blank)"*
  leads to the Manhattan distance heuristic.
  - To solve the puzzle we need to slide each tile into its final position.
  - We can move vertically or horizontally.
  - Number of moves = sum over all of the tiles of the number of vertical and horizontal slides we need to move that tile into place.
  - Again h(n) = sum of the Manhattan distances ≤ h*(n)
    - in a real solution we need to move each tile at least that far and we can only move one tile at a time.

Building Heuristics: Relaxed Problem

Comparison of IDS and A* (average total nodes expanded):

Depth	IDS	A*(Misplaced) **h1**	A*(Manhattan) **h2**
10	47,127	93	39
14	3,473,941	539	113
24	---	39,135	1,641

Let h1=Misplaced,   h2=Manhattan

- Does h2 **always** expand fewer nodes than h1?
    - Yes! Note that **h2 dominates h1**, i.e. for all n: h1(n)≤h2(n). From this you can prove h2 is better than h1 (once both are admissible).
    - Therefore, among several admissible heuristic the one with highest value is better (will cause fewer nodes to be expanded).

Building Heuristics: Relaxed Problem

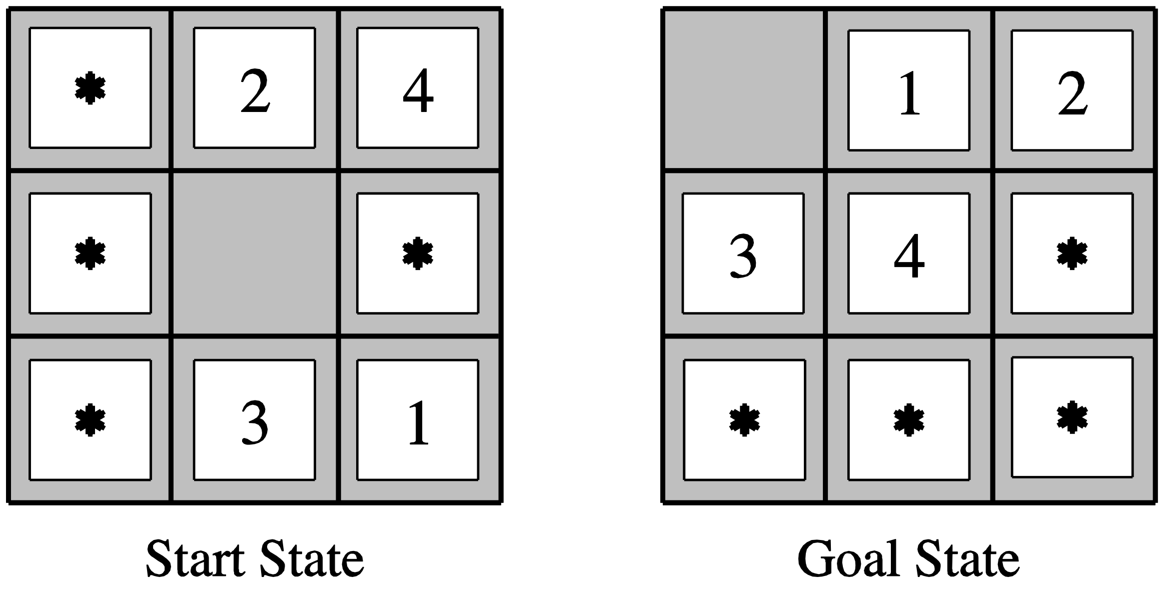
The **optimal** cost to nodes in the relaxed problem is an admissible heuristic for the original problem!

Proof Idea: the optimal solution in the original problem is a solution for relaxed problem, therefore it must be at least as expensive as the optimal solution in the relaxed problem.

So admissible heuristics can sometimes be constructed by finding a relaxation whose optimal solution can be easily computed.

Building Heuristics: Pattern databases

- Try to generate admissible heuristics by solving a subproblem and storing the exact solution cost for that subproblem
- See Chapter 3.6.3 if you are interested.



Start State
Goal State