CSC384 Introduction to Artificial Intelligence

Search

- One of the most basic techniques in AI
  - Underlying sub-module in most AI systems
- Can solve many problems that humans are not good at (achieving super-human performance)
- Very useful as a general algorithmic technique for solving many non-AI problems.

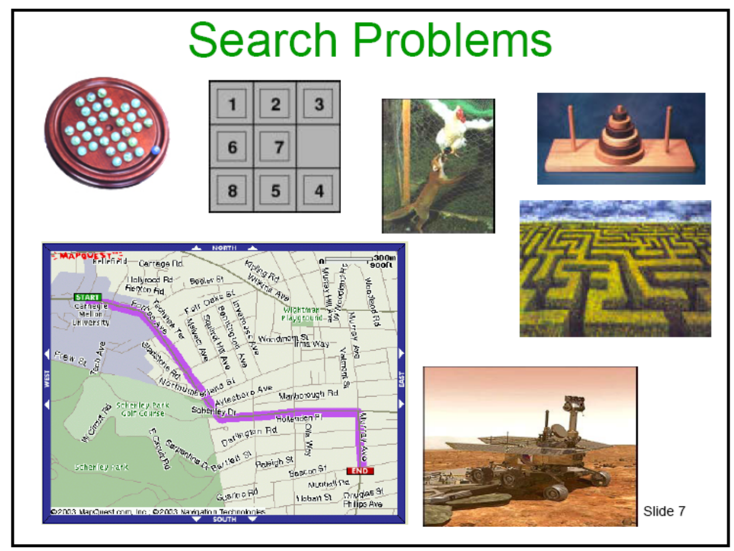
How do we plan our holiday?

- 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 events or actions one should try to bring about 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:



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 shows how to solve the problem once we have it correctly formulated.

Search

Search

- Formulating a problem as search problem (representation)
- Heuristic Search
- Game-Tree-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 formulate a problem as a search problem we need the following components:

1. **STATE SPACE:** Formulate a state space over which we perform search. The state space is a way or representing in a computer the states of the real problem.
2. **ACTIONS or STATE SPACE Transitions:** Formulate actions that allow one to move between different states. The actions reflect the actions one can take in the real problem but operate on the state space instead.

Representing a problem: The Formalism

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

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

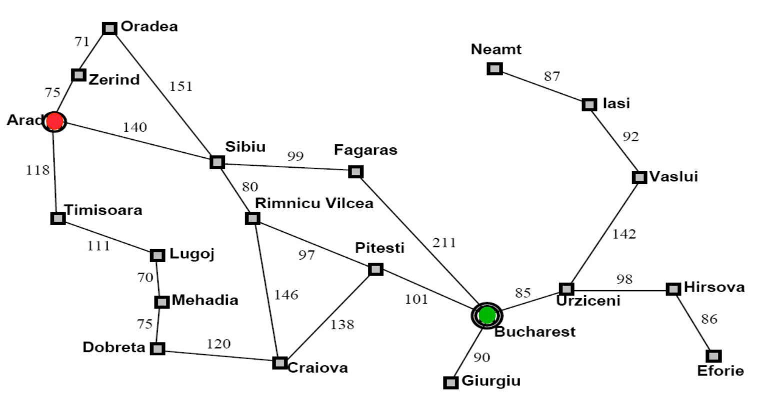
The Formalism

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 the State Space?



Example 1.

- State space.
  - States: the various cities you could be located in.
    - 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.
  - 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.

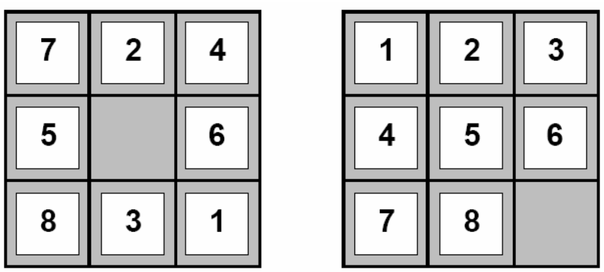
Example 2.

- Water Jugs
  - 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: pairs of numbers (gal3, gal4) where gal3 is the number of gallons in the 3 gallon jug, and gal4 is 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, e.g., (0,0)
  - Desired condition (Goal): Various, e.g., (0,2) or (*, 3) where * means we don't care.

Example 2.

- 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



Start State Goal State

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?
  - Actions: Moving the blank up, down, left, right. Can every action be performed in every state?
  - Initial state: e.g., 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.

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?

More complex situations

- Perhaps actions 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.

Algorithms for Search

Inputs:

- a specified initial state (a specific world state)
- a successor function S(x) = {set of states that can be reached from state x via a single action}.
- 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 move of x → y.

Algorithms for Search

Output:

- a sequence of states leading from the initial state to a state satisfying the goal test.
- The sequence might be, optimal in cost for some algorithms, optimal in length for some algorithms, come with no optimality guarantees from other algorithms.

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** single action.
  - Rather than just return a set of states, we annotate these states by the action used to obtain them:
    - S(x) = {<y,a>, <z,b>}
      y via action a, z via action b.
    - S(x) = {<y,a>, <y,b>}
      y via alternative action b.

Template 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.
- The search algorithms perform search by examining alternate paths of the search space. The objects used in the algorithm are called **nodes**—each node contains a path.

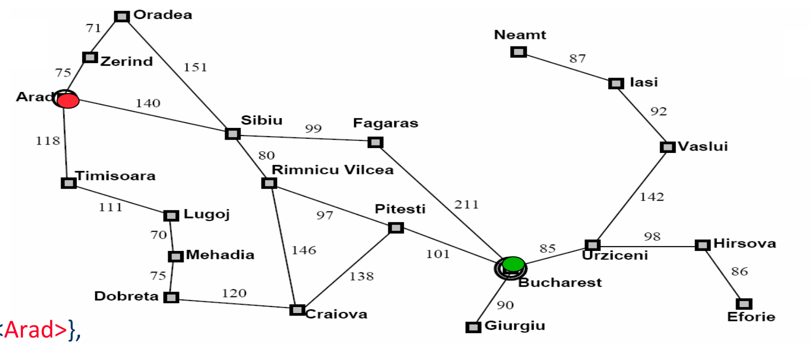
Template Algorithm for Search

- We maintain a set of Frontier nodes also called the OPEN set.
  - 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. Let x be the state n terminates at. We check if x satisfies the goal, if not we add all extensions of n to OPEN (by finding all states in S(x)).

Template Algorithm for Search

```
Search(OPEN, Successors, Goal? )
   While(Open not EMPTY) {
        n = select node from OPEN
        Curr = terminal state of n
        If (Goal?(Curr)) return n.
        OPEN = (OPEN– {n}) U_{s ∈ Successors(Curr)} <n,s>
          /* Note OPEN could grow or shrink */
   }
   return FAIL
```

When does OPEN get smaller in size?



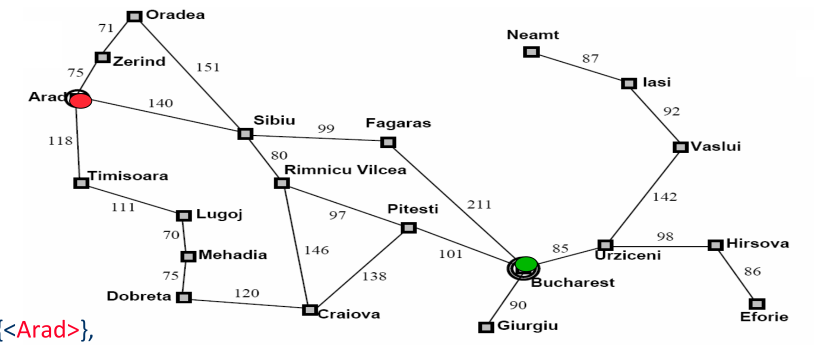
{<Arad>},

{**<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>},

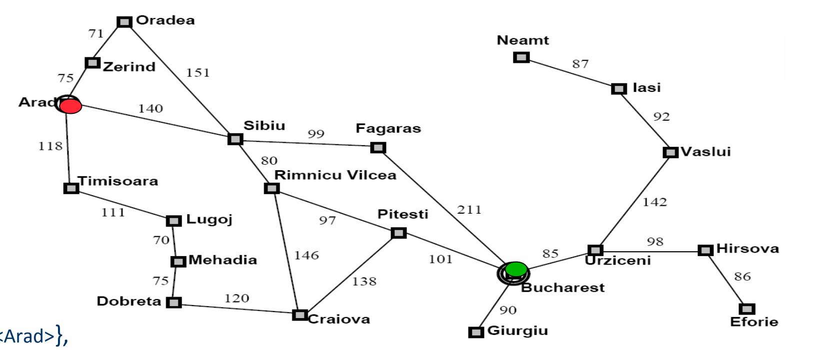
{**<A,Z>, <A,T>, <A, S>**},

{<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 (e.g., a priority queue) and repeatedly execute:

- If OPEN is empty, terminate with failure.
- Get the **next** path from OPEN.
- If the path leads to a goal state, terminate with success.
- Extend the path (i.e. generate the successor states of the terminal state of the path) and put the new paths in OPEN.

- 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 irrespective of the search problem being solved.
- These strategies do not take into account any domain specific information about the particular search problem.
- Uninformed search techniques:
  - Breadth-First, Uniform-Cost, Depth-First, Depth-Limited, and Iterative-Deepening search

Breadth-First Search

Breadth-First Search

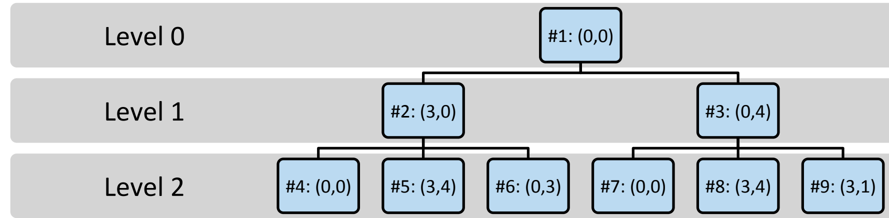
- Place the new paths that extend the current path at the end of OPEN.

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

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.

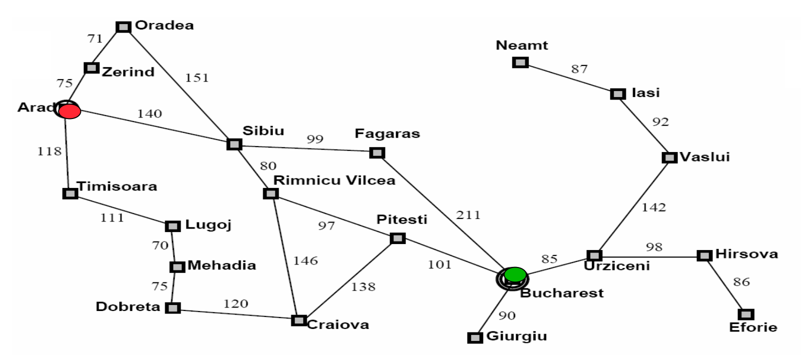
Breadth-First Properties

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 before 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.

- 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 d = length of path - 1

Time Complexity?

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

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 $b(b^d - 1)$

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 ≥ ε > 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 ≥ ε > 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.

Uniform-Cost Search. Proof of Optimality

Let us prove Optimality more formally. We will reuse this argument later on when we examine Heuristic Search

Uniform-Cost Search. Proof of Optimality

Lemma 1.

Let c(n) be the cost of node n on OPEN (cost of the path represented by n). If n2 is expanded IMMEDIATELY after n1 then
c(n1) ≤ c(n2).

Proof: there are 2 cases:

a. n2 was on OPEN when n1 was expanded:
We must have c(n1) ≤ 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 cost at least ε more.

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:

- Let nk = <Start, s1, …, sk> be a path with cost less than c(n). Let n0 = <Start>, n1 = <Start, s1>, n2 = <Start, s1, s2>, …, ni = <Start, s1, …, si>, …, nk = <Start, s1, …, sk>. Let ni be *the last node in this sequence that has already been expanded by search.*
- So, ni+1 must still be on OPEN: it was added to open when ni was expanded. Also c(ni+1) ≤ c(nk) < c(n): c(ni+1) is a subpath of nk we have assumed that c(nk) is < c(n).
- But then uniform-cost would have expanded ni+1 not n.
- So 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 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*: there can be as many as b^d paths of length d in the worst case.

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

Depth-First Search

Depth-First Search

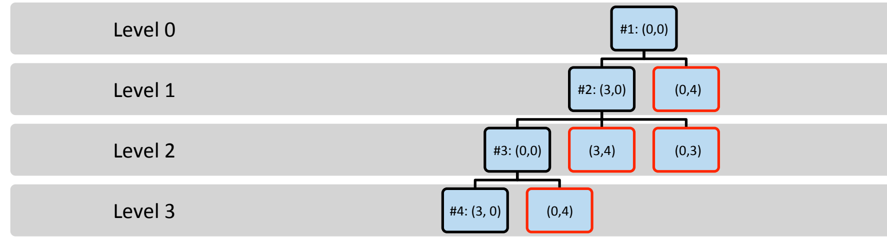
- Place the new paths that extend the current path at the front of OPEN.

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

Green = Newly Added.

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.
- 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 ≤ D exists.

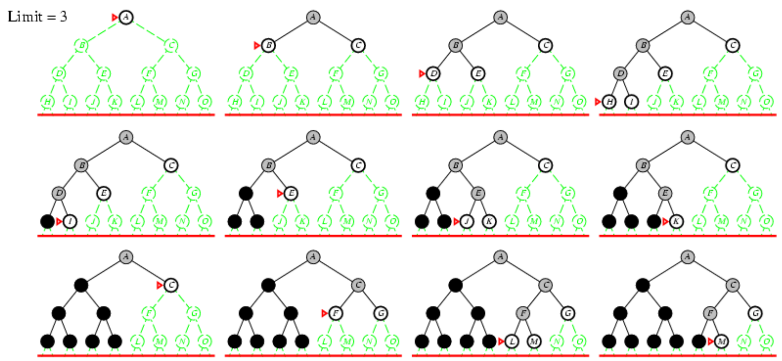
Depth Limited Search

```
DLS(OPEN, Successors, Goal?) /* Call with OPEN = {<START>} */
   WHILE(OPEN not EMPTY) {
       n= select first node from OPEN
       Curr = terminal state of n
       If(Goal?(Curr)) return n

       If Depth(n) < D   //Don't add successors if Depth(n) = D
              OPEN = (OPEN– {n}) U_{s ∈ Successors(Curr)} <n,s>
       Else
              OPEN = OPEN – {n}
              CutOffOccured = TRUE.
   }
   return FAIL
```

We will use CutOffOccured later.

Depth Limited Search Example



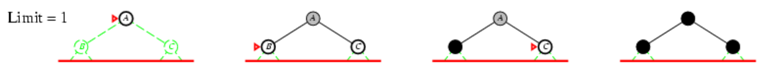
Iterative Deepening Search

Iterative Deepening Search

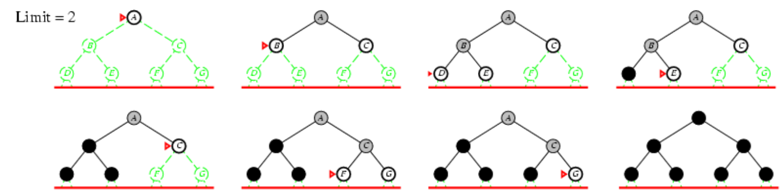
- Solve the problems of depth-first and breadth-first by extending depth limited search
- Starting at depth limit L = 0, we iteratively increase the depth limit, performing a depth limited search for each depth limit.
- Stop if a solution is found, or if the depth limited search failed without cutting 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 Deepening Search Example

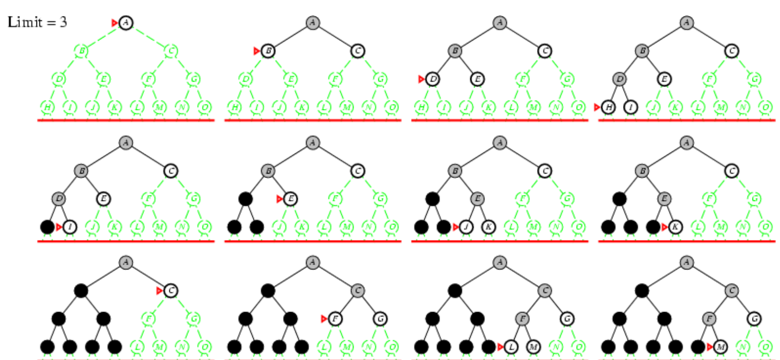
Iterative Deepening Search Example



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 represents the path $<s_0,s_1,\ldots,s_k>$ and we expand s_k to obtain child c, we have

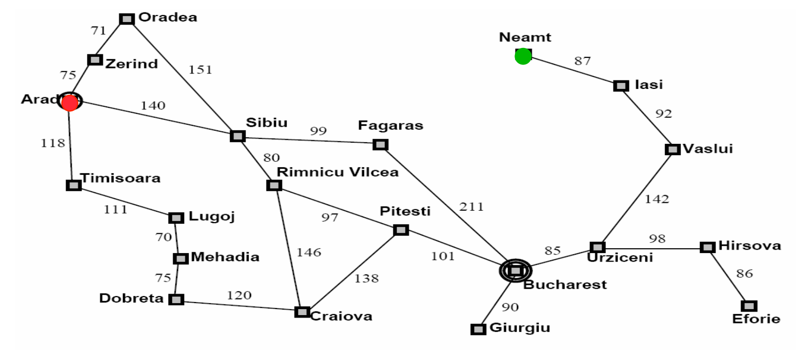
$<S,s_1,\ldots,s_k,c>$

As the path to "c".

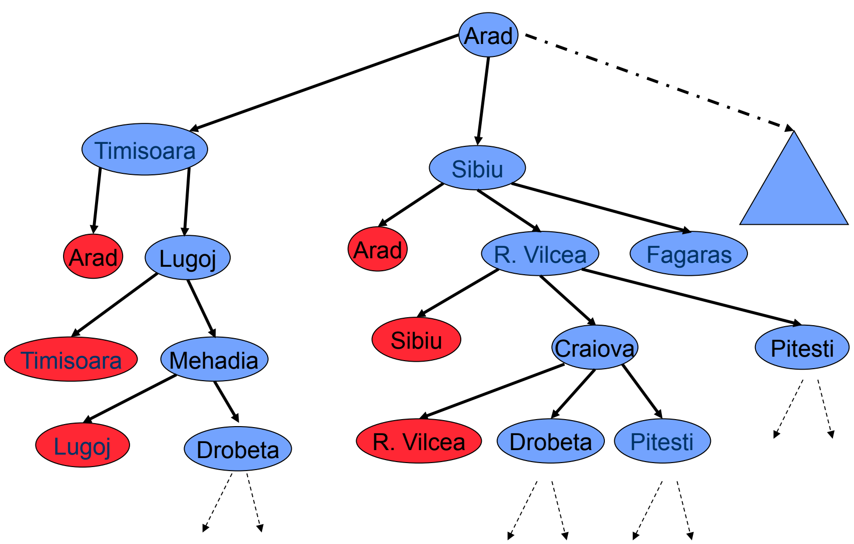
Path checking:

- Ensure that the state c is not equal to the state reached by any ancestor of c along this path.
- Paths are checked in isolation!

Example: Arad to Neamt



Path Checking Example

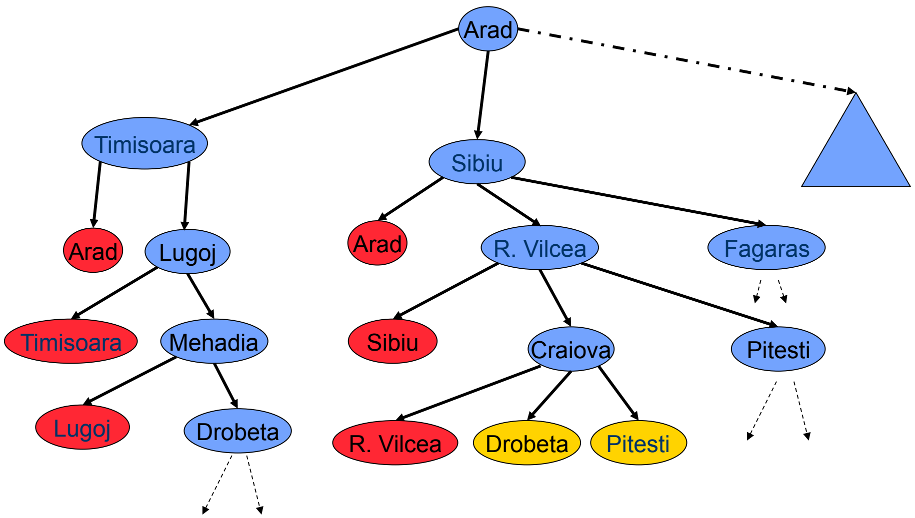


Cycle Checking

Cycle Checking

- Keep track of all states previously expanded during the search.
- When we expand n_k to obtain child c
  - Ensure that c is not equal to **any** previously expanded state.
- 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
  - With uniform-cost search, we still find an optimal solution
    - The first time uniform-cost expands a state it has found the minimal cost path to it.
  - This means that the nodes rejected subsequently by cycle checking can't have better paths.
  - We will see later that we don't always have this property when we do heuristic search.

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 to the goal from the end of the current path.

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

Heuristic Search

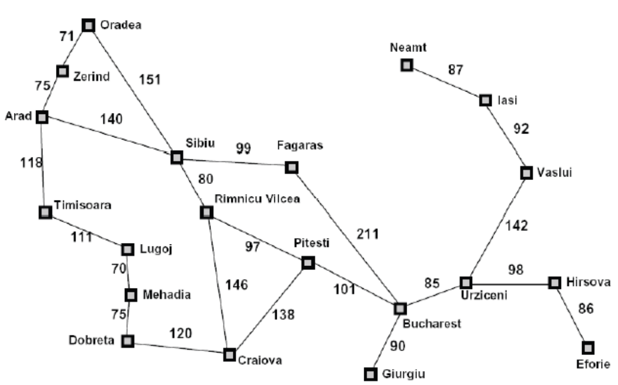
Merit of an OPEN node: different notions of merit.

- 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 terminal 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 terminal 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 terminal state of the path represented by n).
- There are different ways of guessing this cost in different domains.
  - heuristics are domain specific.

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.

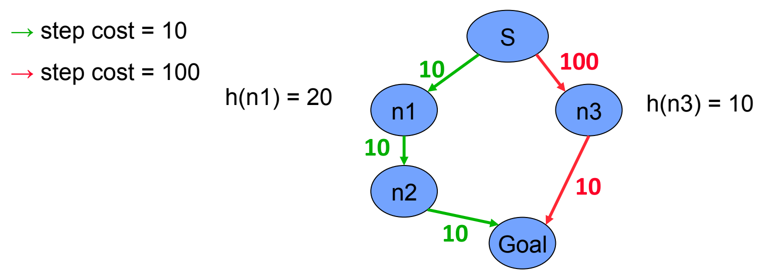
Heuristic Search

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

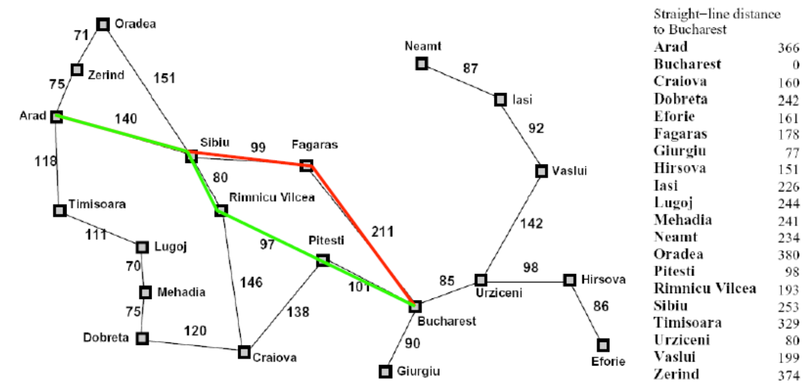
Using only h(n): Greedy best-first search

- 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



A* search

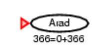
- Take into account the cost of getting to the node as well as our estimate of the cost of getting to the goal from the node.

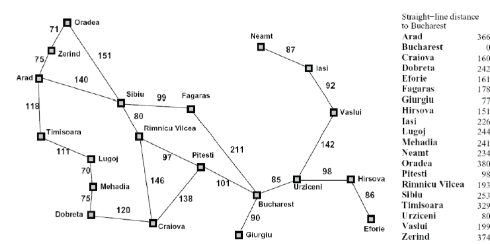
- Define an evaluation function f(n)
  f(n) = g(n) + h(n)
  - g(n) is the cost of the path represented by node n
  - h(n) is the heuristic estimate of the cost of achieving the goal from n.

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

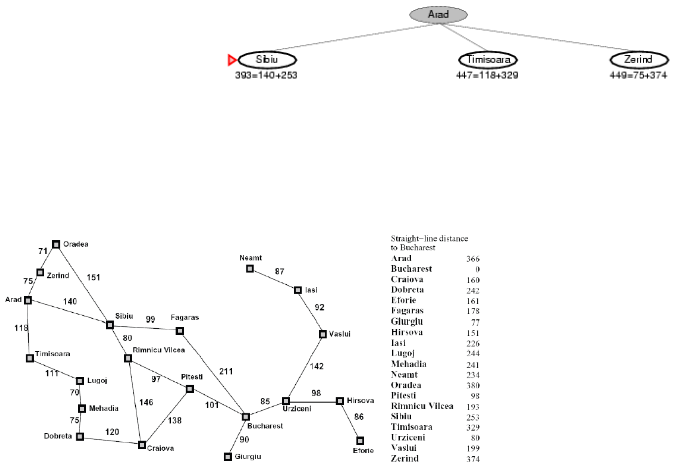
- The f-value, f(n) is an estimate of the cost of getting to the goal via the node (path) n.
  - I.e., we first follow the path n then we try to get to the goal. f(n) estimates the total cost of such a solution.

A* example

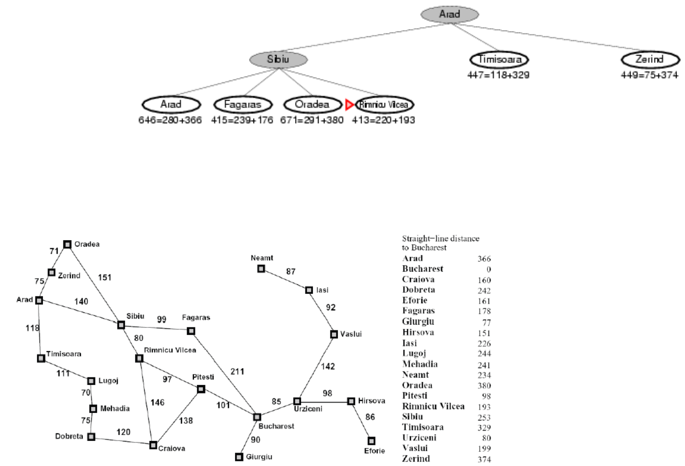




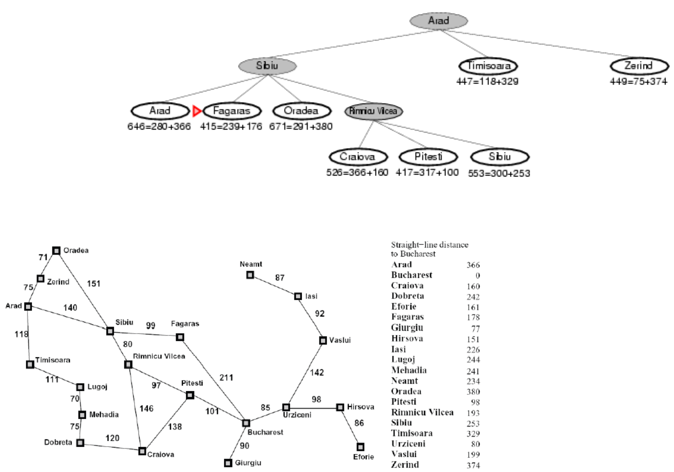
A* example



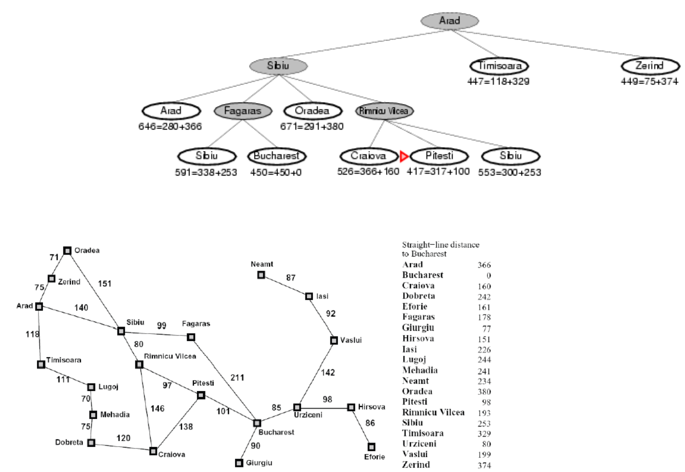
A* example



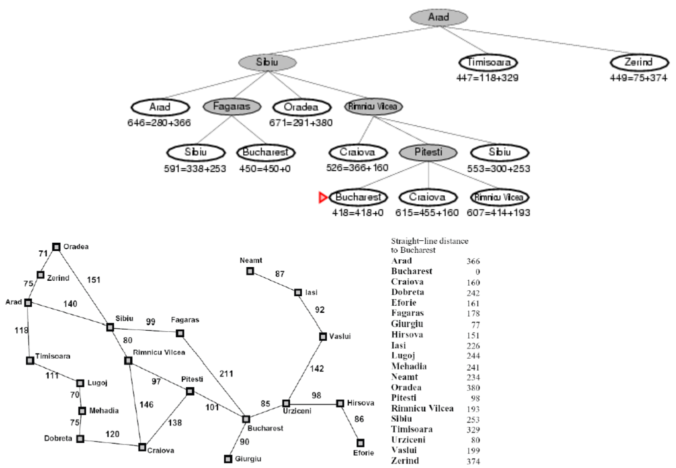
A* example



A* example



A* example



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

- We want to analyze the behavior of the resultant search.
  - Completeness, time and space, optimality?
- To obtain such results we must put some further conditions on the heuristic function h(n) and the search space.

Conditions on h(n): Admissible

- We **always** assume that c(s1, a, s2) ≥ ε > 0 for any two states s1 and s2 and any action a: 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 (∞ if there is no path). Then an admissible heuristic satisfies the condition

h(n) ≤ h*(n)

- an admissible heuristic never over-estimates the cost to reach the goal, i.e., it is optimistic
- Hence h(g) = 0, for any goal node g
- Also h*(n) = ∞ if there is no path from n to a goal node

Consistency (aka monotonicity)

- A stronger condition than h(n) ≤ h*(n).
- A monotone/consistent heuristic satisfies the triangle inequality: for all nodes n1, n2 and for all actions a

h(n1) ≤ C(n1,a,n2) + h(n2)

Where C(n1, a, n2) means the cost of getting from the terminal state of n1 to the terminal 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) ≤ C(n1,a,n2) + h(n2) ➔ (forall n) h(n) ≤ h*(n)

Consistency $\rightarrow$ Admissible

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

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

Proof:

Let $n \rightarrow n1 \rightarrow \ldots \rightarrow n^*$ be an OPTIMAL path from n to a goal (with actions a1, a2). Note the cost of this path is $h^*(n)$, and each subpath $(ni \rightarrow \ldots \rightarrow n^*)$ has cost equal to $h^*(ni)$.

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

Otherwise prove $h(n) \le h^*(n)$ by induction on the length of this optimal path.

Base Case: n = n*

By our conditions on h, $h(n) = 0 \le h(n)^* = 0$

Induction Hypothesis: $h(n1) \le h^*(n1)$

$h(n) \le c(n,a1,n1) + h(n1) \le c(n,a1,n1) + h^*(n1) = h^*(n)$

Intuition behind admissibility

$h(n) \le h^*(n)$ 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.
- Monotonicity gives some additional properties when it comes to cycle checking.

Consequences of monotonicity

1. The f-values of nodes along a path must be non-decreasing.

Let <Start$\rightarrow$ s1$\rightarrow$ s2 $\rightarrow$ … $\rightarrow$ sk> be a path. Let ni be the subpath <Start $\rightarrow$ s1 $\rightarrow$… $\rightarrow$ si>:

We claim that: $f(ni) \le f(ni+1)$

Proof

$$\begin{aligned} f(ni) &= c(Start \rightarrow \ldots \rightarrow ni) + h(ni) \\ &\le c(Start \rightarrow \ldots \rightarrow ni) + c(ni \rightarrow ni+1) + h(ni+1) \\ &\le c(Start \rightarrow \ldots \rightarrow ni \rightarrow ni+1) + h(ni+1) \\ &\le g(ni+1) + h(ni+1) = f(ni+1) \end{aligned}$$

Consequences of monotonicity

2. If n2 is expanded immediately after n1, then $f(n1) \le f(n2)$

(the f-value of expanded nodes is **monotonic** non-decreasing)

Proof:

- If n2 was on OPEN when n1 was expanded, then $f(n1) \le f(n2)$ otherwise we would have expanded n2.
- If n2 was added to OPEN after n1's expansion, then n2 extends n1's path. That is, the path represented by n1 is a prefix of the path represented by n2. By property (1) we have $f(n1) \le f(n2)$ as the f-values along a path are non-decreasing.

Consequences of monotonicity

3. Corollary: the sequence of f-values of the nodes expanded by A* is non-decreasing. I.e, If n2 is expanded **after** (not necessarily immediately after) n1, then f(n1) ≤ f(n2)

(the f-value of expanded nodes is **monotonic** non-decreasing)

Proof:

- If n2 was on OPEN when n1 was expanded, then f(n1) ≤ f(n2) otherwise we would have expanded n2.
- If n2 was added to OPEN after n1's expansion, then let n be an ancestor of n2 that was present when n1 was being expanded (this could be n1 itself). We have f(n1) ≤ f(n) since A* chose n1 while n was present on OPEN. Also, since n is along the path to n2, by property (1) we have f(n)≤f(n2). So, we have f(n1) ≤ f(n2).

Consequences of monotonicity

4. When n is expanded every path with lower f-value has already been expanded.

- **Proof:** Assume by contradiction that there exists a path <Start, n0, n1, ni-1, ni, ni+1, …, nk> with f(nk) < f(n) and ni is its last expanded node.
  - ni+1 must be on OPEN while n is expanded, so

  a) by (1) f(ni+1) ≤ f(nk) since they lie along the same path.

  b) since f(nk) < f(n) so we have f(ni+1) < f(n)

  c) by (2) f(n) ≤ f(ni+1) because n is expanded before ni+1.

  - Contradiction from b&c!

Consequences of monotonicity

5. With a monotone heuristic, the first time A* expands a state, it has found the minimum cost path to that state.

Proof:

- Let **PATH1 = <Start, s0, s1, …, sk, s>** be **the first** path to a state s found. We have f(path1) = c(PATH1) + h(s).
- Let **PATH2 = <Start, t0, t1, …, tj, s>** be another path to s found later. we have f(path2) = c(PATH2) + h(s).
  - Note h(s) is dependent only on the state s (terminal state of the path) it does not depend on how we got to s.
- By property (3), f(path1) ≤ f(path2)
- hence: c(PATH1) ≤ c(PATH2)

Consequences of monotonicity

Complete.

- Yes, consider a least cost path to a goal node
  - SolutionPath = <Start→ n1→ …→ G> with cost c(SolutionPath). Since h(G) = 0, this means that f(SolutionPath) = cost(SolutionPath)
  - Since each action has a cost ≥ ε > 0, there are only a finite number of paths that have f-value < c(SolutionPath). None of these paths lead to a goal node since SolutionPath is a least cost path to the goal.
  - So eventually SolutionPath, or some equal cost path to a goal must be expanded.

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. IDA* can sometimes be used to address memory issues, but IDA* isn't very good when many cycles are present.

Consequences of monotonicity

Optimality

- Yes, by (5) the first path to a goal node must be optimal.

> 5. With a monotone heuristic, the first time A* expands a state, it has found the minimum cost path to that state.

Cycle Checking

- We can use a simple implementation of cycle checking (multiple path checking)---just reject all search nodes visiting a state already visited by a previously expanded node. By property (5) we need keep only the first path to a state, rejecting all subsequent paths.

Admissibility without monotonicity

When "h" is admissible but not monotonic.

- Time and Space complexity remain the same. Completeness holds.
- Optimality still holds (without cycle checking), but need a different argument: don't know that paths are explored in order of cost.

Admissibility without monotonicity

What about Cycle Checking?

- No longer guaranteed we have found an optimal path to a node ***the first time*** we visit it.
- So, cycle checking might not preserve optimality.
  - To fix this: for previously visited nodes, must remember cost of previous path. If new path is cheaper must explore again.

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 and similarly addresses space issues.

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 bigger 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 faster than h1 (once both are admissible).
  - Therefore, among several admissible heuristic the one with highest value is the fastest.

Building Heuristics: Relaxed Problem

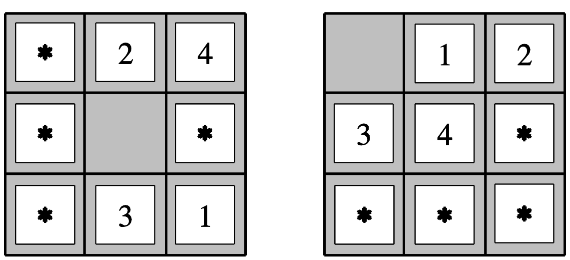
The **optimal** cost to nodes in the relaxed problem is an admissible heuristic for the original problem!

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