Problem Solving Agents

Aditya Challa

August 18, 2024

BITS Pilani
Pilani Campus

Course: Artificial and Computational Intelligence
Lecture No. 3 & 4

RECAP

- Four things they mean when people say AI
  - Acting Humanly - Turing Test - Can a machine exhibit intelligent behavior indistinguishable from a human?
  - Thinking Humanly - Cognitive Science - How do humans think and can machines think like humans?
  - Thinking Rationally - Logic - Can machines think logically?
  - Acting Rationally - Rational Agents - Can machines act rationally?
- There are issues with all of these ideas!
- We focus on the Acting Rationally part - Rational Agents

RECAP

- Other important points:
  - Philosophy, Mathematics, Economics, Neuroscience, Psychology, Control Theory, Computer Science, Linguistics - All contribute to AI. So, its a very interdisciplinary field.
  - The history of AI is a series of overexpectations and underdeliveries. But the field has matured a lot in the last 10 years to a *point of no return*!.

RECAP

- The broad lessons from history:
  - Economics trumps always. Anything which is not feasible and does not reduce costs might not last. See `https://www.goldmansachs.com/intelligence/pages/gs-research/gen-ai-too-much-spend-too-little-benefit/report.pdf`
  - Humans have an amazing ability to adapt and accept - ChapGPT was a phenomenon in 2021, but after 3 years there are open source models Llama3.1 which can do the same thing.
  - Efficiency is important - Bayesian methods can theoretically solve every problem out there, but it's highly inefficient.
  - What you measure is what you get - Evaluation metrics and designing them are probably the most important value addition humans still make.

RECAP

- In this course we are interested in "acting rationally". So, we try to model rational agents. Before that we need to setup the *environment* -

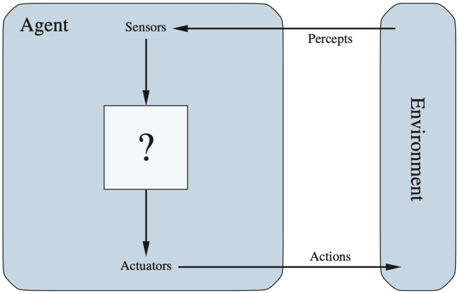


Figure 2.1 Agents interact with environments through sensors and actuators.

RECAP

- So our model for rational agent
  - Takes as input a sequence of sensory reading which we will call *percept sequence*
  - Returns as output some actuator movements, which the environment uses.

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

RECAP

- The broad aim is going to be to model these rational agents.
  - Question: Do you think it is possible to have a rational agent irrespective of the environment?
  - So, we will try and have some restrictions on the environment as well.
  - We looked a lot of examples – the running example is Autonomous driving.

RECAP

• We saw some common architectures (rather design patterns) to model these rational agents:

- A simple table driven action - The agent looks-up the sequence from a long table and returns the action.
- Simple reflex agents - Use markov assumption (essentially forgets percept history)
- Model reflex agents - Builds an internal model of the world
- Goal based agents - Uses goals to dictate the actions, they too build an internal model of the world.
- Utility based agents - Uses "utility" functions to dictate the actions. They try to buils an internal model of performance.
- Learning Agents - They keep on improving the internal models.

RECAP

- We also discussed the concept of representations.
  - Atomic Representations are just vectors without any meaning within it.
  - Factored Representations are representations which you can think of as a python dictionary – indicators on a dashboard.
  - Structured Representations are essentially a fixed format where each element has a fixed meaning. We shall look at these in first order logic.
- Local vs Distributed Representations
  - Local Representations separate out the concepts – Each concept is at a memory location.
  - Distributed Representations are where concepts are interleaved with memory locations. Example:Word2Vec.

Problem Solving Agents

- So, let's dive in to the first actual model of rational agent.
  - They use atomic representations.
  - The environment is – episodic, single agent, fully observable, deterministic, static, discrete. (can you recall all the terms??)
  - *informed* agents know how far it is from the goal - (think optimization problems??)
  - *uninformed* agents do not know how far they are from the goal.

Problem Solving Agents

- Single vs Sequence of Actions:
  - If the agent knows which *single* action will give it what it wants, then the problem is pretty simple – So, we won't consider this.
  - Instead, we are interested in – If the agent knows it's final goal and the environment, can it identify a sequence of actions so that it would reach it's goal?
  - Side Note: Starting simple with large constraints and slowly relaxing the constraints is a very good approach to science in general. (think "spherical cow" joke!). This is what we will also do.

Problem Solving Agents

- The aim is to reach *Bucharest* from *Arad*

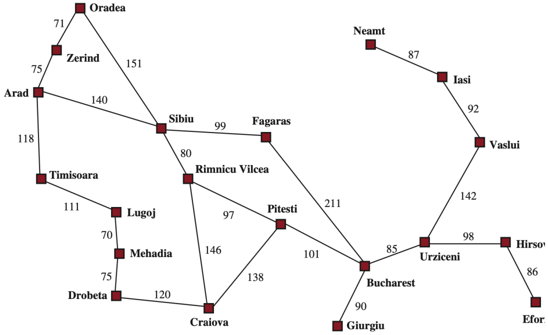


Figure 3.1 A simplified road map of part of Romania, with road distances in miles.

Problem Solving Agents

• Let us go in the reverse and check what a “solution” (behaviour of rational agent looks like) – It would have 4 steps:

- Goal Formulation - Fix a goal of reaching a specific state (ex. reaching bucharest)
- Problem Formulation - Have an internal description of states and actions and how it will affect the environment.
- Search for a solution - Get a solution with respect to the current formulation before actually taking any actions.
- Execution - Actually perform the actions.

• So, basically we (rational agent) have to formulate the problem and solve it internally.

Problem Solving Agents

- Search Problem Formulation: (Abstracting several problems!!)
  - State-Space : An abstraction of all possible states one can be in. - For example all possible cities in romania.
  - Initial State : The state the agent starts in.
  - Goal States : Where do you want to reach.
  - Actions : ACTION(s) denotes the possible actions we can take from s. – For example , from Arad we can go to Sibiu, Timisora etc.
  - Transition Model: If you take a action *a* at a specific state *s* what will you get – In simple cases (travelling) it is obvious. But might be more of an issue in few contexts (chess)
  - Action-Cost : ACTION-COST(s,a,s') – What is the cost of - if we are at s, take action a and end-up in state s'. We sometimes use $c(s,a,s')$ to denote this.

Problem Solving Agents

- What is a solution:
  - A solution is nothing but a sequence of actions which will take me from the present state to the goal state (Ex: Arad $\rightarrow$ Bucharest). This is also referred to as path.
  - An optimal solution is the path which has the least cost assigned to it??
- We represent the state space as a graph where the verticies are states, edges are all actions we can take from a state.

Problem Solving Agents

- Abstraction and Practical Significance
  - So, we abstracted the problem of going from Arad $\rightarrow$ Bucharest as a graph of cities and cost to be distances etc.
  - Why did we not take mile markers instead of cities as states? Why not districts?
  - A often ignored aspect is – *Level of abstraction* – How do we formulate the problem at hand? (will see a few examples)
- Hallmarks of good abstraction:
  - If we have a solution at lower abstraction (example mile markers), we should be able to have a solution at higher abstraction (example cities).– This is called valid abstraction.
  - Each action should be easier than the original problem. Can you think of a useless abstraction for the travelling problem? – (a sequence of steps instead of a single step???)

Problem Solving Agents

- Another Example: Vaccum World:

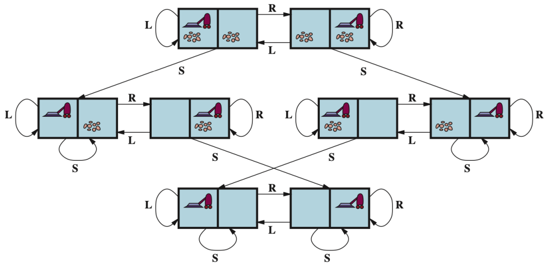


Figure 3.2 The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: L = *Left*, R = *Right*, S = *Suck*.

- Identify states, initial state, actions, transition model, goal state, action costs???
- Here there are only 2 positions. What if there are *n* positions? How many states will we get?

Problem Solving Agents

- A more complex example:

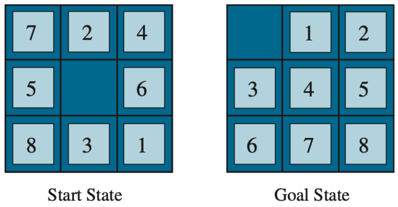


Figure 3.3 A typical instance of the 8-puzzle.

- Identify states, initial state, actions, transition model, goal state, action costs???
- In general we will not be able to write down all the states completely.

Problem Solving Agents

- Real world problems:
  - Navigation, Web-Traffic routing
  - Travelling Salesman Problem
  - VLSI Layout
  - Robot Navigation (vaccum cleaners!!)
  - And a lot more...

Search Algorithms

• To recall, we have a graph with – states, actions, costs, start-state, goal-state – and the aim is to identify a lowest cost path between start-state and final state.
• We use the idea of *search tree* to solve this problem – A search tree is nothing but a hierarchy starting from initial state which covers all possible actions.
• The main advantage is – search trees are usually finite while the underlying graph can be infinite! (Example: grid in R^2??)

Search Algorithms

- Example (Arad $\rightarrow$ Bucharest)

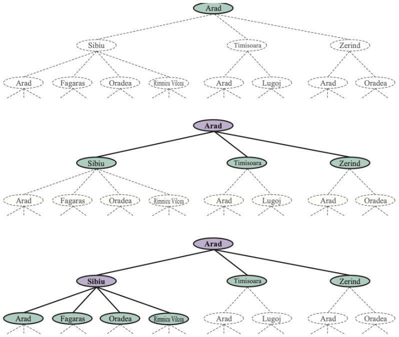


Figure 3.4 Three partial search trees for finding a route from Arad to Bucharest. Nodes that have been *expanded* are lavender with bold letters; nodes on the frontier that have been *generated* but not yet expanded are in green; the set of states corresponding to these two types of nodes are said to have been *reached*. Nodes that could be generated next are shown in faint dashed lines. Notice in the bottom tree there is a cycle from Arad to Sibiu to Arad; that can't be an optimal path, so search should not continue from there.

Search Algorithms

- Getting a search tree.
    - We start with a node.
    - We *Generate* various nodes, called *Child/Successor Nodes* based on all the actions available.
    - We repeat this procedure *recursively* for all child nodes.
- Suppose you repeat the above procedure 3 times, the *frontier* is the set of nodes which are the leaves, you can reach the nodes within a frontier – *Reached Nodes*

Search Algorithms

- Example:

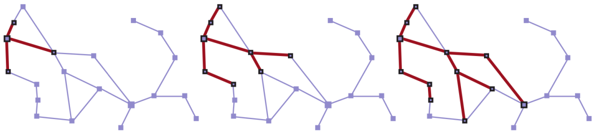


Figure 3.5 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.1. At each stage, we have expanded every node on the frontier, extending every path with all applicable actions that don't result in a state that has already been reached. Notice that at the third stage, the topmost city (Oradea) has two successors, both of which have already been reached by other paths, so no paths are extended from Oradea.

Search Algorithms

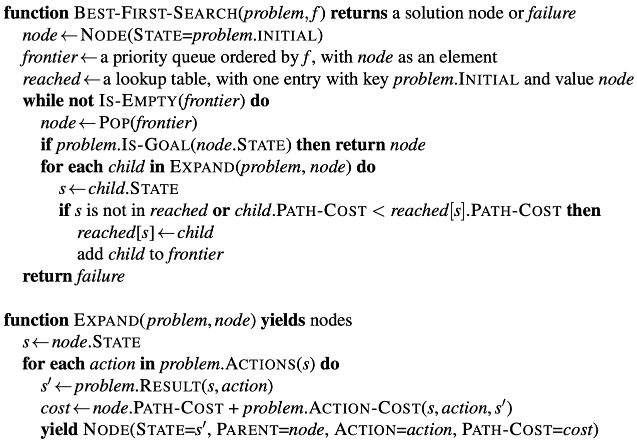
- Which nodes at the frontier to generate further?
  - This is probably the main crux of the problem!
  - No simple solution (we shall shortly see this!)
- So, what we do is – Assume an evaluation function $f(n)$ which tells us the best possible node!

Search Algorithms

- SIDE NOTE: Observe the big picture here!
  - We started with rational agent within an environment.
  - Simplified it to a graph.
  - Further simplified it to a search tree,
  - Now simplified this to a function on a node!
- We have been slowly converting the problem at hand to something we can handle!!

Search Algorithms

- Assuming $f(n)$ we have Best-First-Search



```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s′ ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s′)
    yield NODE(STATE=s′, PARENT=node, ACTION=action, PATH-COST=cost)
```

Figure 3.7 The best-first search algorithm, and the function for expanding a node. The data structures used here are described in Section 3.3.2. See Appendix B for **yield**.

Search Algorithms

- NODE Data Structure:
  - node.STATE - The state to which the node corresponds.
  - node.PARENT - From which node did we reach present node?
  - node.ACTION - Which action from node.PARENT returned the present node.
  - node.PATH-COST - Cost of the path from initial node to present node.

Search Algorithms

- Frontier Data Structure – Need the following operations:
  - IS-EMPTY – tell of all the nodes are done!
  - POP – Returns the best node w.r.t $f(n)$ and removes it.
  - TOP – Returns the best node w.r.t $f(n)$ but does not remove it.
  - ADD – Adds a new node to the frontier.
- the QUEUE data structure is ideal to code this!

Search Algorithms

- Some issues to consider:
  - The graphs can have loops. This can lead to a lot of inefficiency. Need to handle this using –
    - Remember the previous states
    - Frame the problem such that it can only go one way!
    - Checking for cycles (follow the parent!)
  - Which algorithm to choose?? – There are usually different design choices. For example - queue can be (a) Priority Queue (b) LIFO (c) FIFO. Which is better?
    - Completeness – Can the algorithm find the solution if there is one and report failure if not??
    - Cost optimality – Can the algorithm find optimal cost solution
    - Complexity - Space and Time

Search Algorithms

- Recall there are 2 kinds of search strategies:
  - *Uninformed* search does not how close you are to the goal
  - *Informed* search has “some” information on how close you are to the goal.
- Note: we are still in the problem formulation and solution finding stage within the agent. The agent itself will *not* move (execution) in the real world.

Uninformed Search

- There are several algorithms here, but the following are prominent:
  - Breadth First Search
  - Dijkshitra's algorithm
  - Depth-first search
  - Depth limited and Iterative Deepening search.
  - Bidirectional Search

Uninformed Search

- Breadth-First Search
  - Good in the case all actions have the same cost.
  - We take $f(n)$ to be the depth of the node in the search tree.
  - Use FIFO queue – (look this up?)
  - The first time we reach the state is the optimal path – So, the first time we reach the goal is when we can stop! (early stopping).

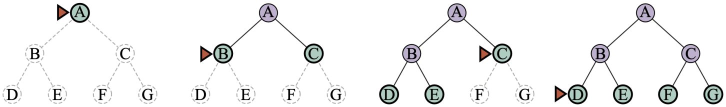
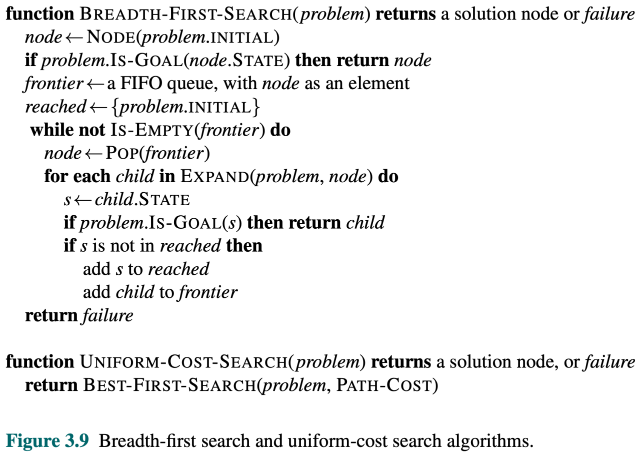


Figure 3.8 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by the triangular marker.

Uninformed Search

- Breadth-First Search



```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure

function UNIFORM-COST-SEARCH(problem) returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem, PATH-COST)
```

Figure 3.9 Breadth-first search and uniform-cost search algorithms.

Uninformed Search

- Breadth-First Search
  - Complete – Can return if there is no answer.
  - Optimal – Will find the optimal path if it exists.
  - Time Complexity - $\mathcal{O}(b^d)$ – b is the branching factor and d is the depth. (WHY??)
  - Space Complexity - $\mathcal{O}(b^d)$
  - An issue if the costs of all actions are not the same.

Uninformed Search

- Dijkshitra Algorithm (Uniform Cost Search)
    - If all actions are not of same cost, then $f(n)$ is denotes the path cost till n.
    - Use a priority queue instead.

Uninformed Search

- Example - Dijkshitra Algorithm

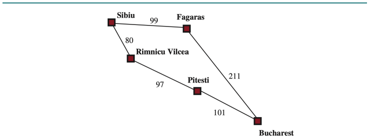


Figure 3.10 Part of the Romania state space, selected to illustrate uniform-cost search.

- Add Sibiu and expand.
- Add Rimnicu (80) and Fagaras (99)
- expand Rimnicu to add Pitesti (177)
- expand Fagaras to add Bucharest (310)
- Expand Pitesti to *update* Bucharest (278).

- We would only know the shortest path to a node when we *expand* it, not when we add it!

Uninformed Search

- Dijkshitra Algorithm (Uniform Cost Search)
  - This is complete and cost-optimal.
  - If C^* is the cost of optimal path, and ϵ the lowest cost of a action, both time and space complexity is $\mathcal{O}(b^{1+\lfloor C^*/\epsilon \rfloor})$

Uninformed Search

- Depth First Search
  - It is not cost-optimal – does not find the shortest path!
  - It is also incomplete – Can get stuck in an infinite path (if the state space is not finite!)
  - However, it is efficient if the graph is also a tree!

Uninformed Search

- Depth First Search

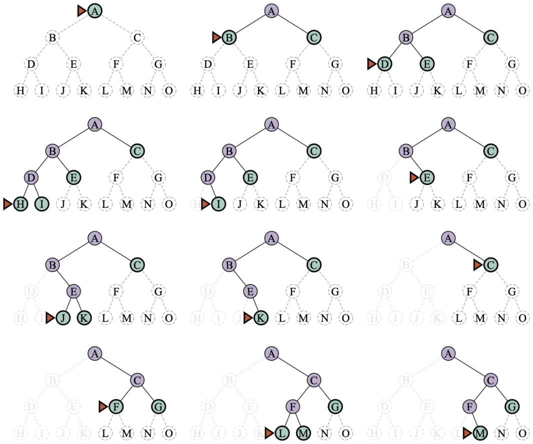


Figure 3.11 A dozen steps (left to right, top to bottom) in the progress of a depth-first search on a binary tree from start state A to goal M. The frontier is in green, with a triangle marking the node to be expanded next. Previously expanded nodes are lavender, and potential future nodes have faint dashed lines. Expanded nodes with no descendants in the frontier (very faint lines) can be discarded.

Uninformed Search

- Variants of DFS
  - Depth Limited Search : Stops the search at depth l. In general difficult to find l
  - Iterative Deepening solves this problem by considering $l = 0, 1, \cdots$

Uninformed Search

- Bidirectional Search
  - Does the search by going initial state $\rightarrow$ goal and goal $\rightarrow$ initial state simultatenously.

Informed Search strategies

• Uninformed search is okay, but they are not always efficient. Especially if you have some information on where you want to reach and how far your are from your goal!

- We assume we have access to $h(n)$ an estimated optimal cost to reach the goal state from the node n.
- This is called a heuristic function.
- This information usually comes from domain - for instance straight line distance in our example.

Informed Search strategies

- Greedy best-first search
  - It's basically the best first search where $f(n) = h(n)$.
  - Its complete on finite space, does not necessarily find optimal cost.
  - Time and Space complexity is $\mathcal{O}(|V|)$

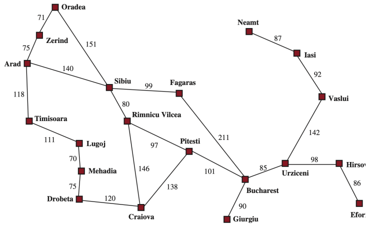
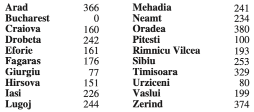


Figure 3.1 A simplified road map of part of Romania, with road distances in miles.



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

Figure 3.16 Values of h_{SLD}—straight-line distances to Bucharest.

Informed Search strategies

- Greedy Best First Search

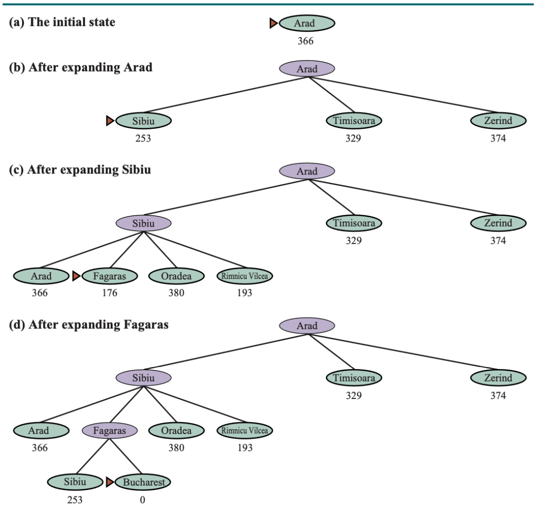
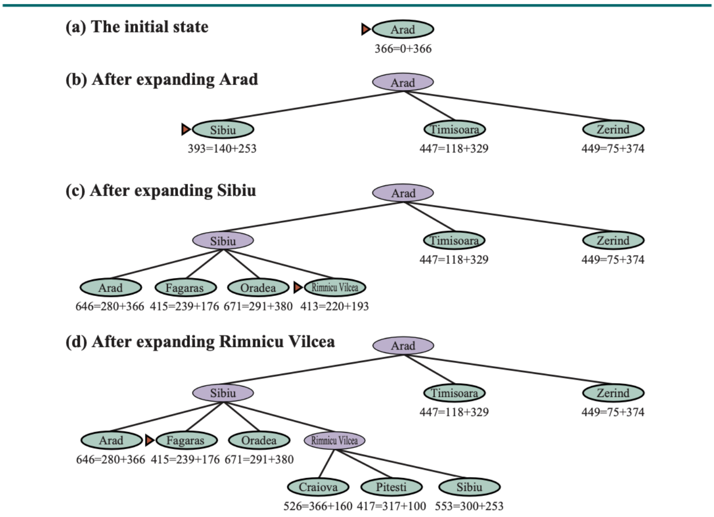


Figure 3.17 Stages in a greedy best-first tree-like search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

Informed Search strategies

- A^* search.
  - It uses the evaluation function $f(n) = g(n) + h(n)$, where $g(n)$ the cost till node n from initial state.
  - Complete – i.e it can find a solution
  - A^* is cost optimal if the heuristic is "admissible"– i.e if it does not *over-estimate the cost to reach the goal.*

Informed Search strategies



Informed Search strategies

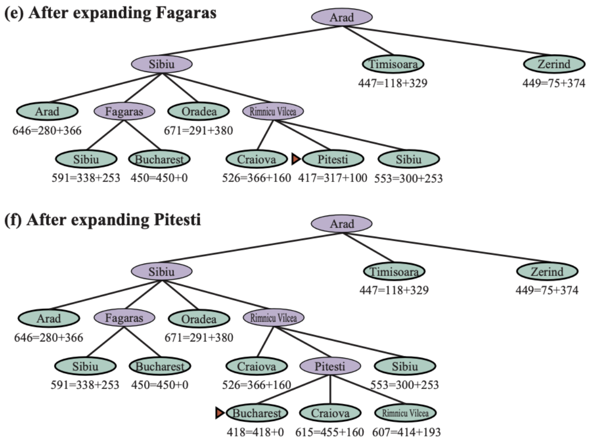


Figure 3.18 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.16.

Informed Search strategies

- Proof that A^* is optimal if h is admissible
  - Let optimal cost C^*, but algorithm returns $C > C^*$.
  - Let n be the node where the optimal path differs from the one that we find.
  - $f(n) > C^*$, since we did not find the optimal path.
  - But,

$$\begin{aligned} f(n) = g(n) + h(n) &= g^*(n) + h(n) \leq g^*(n) + h^*(n) \\ &\leq C^* \end{aligned}$$

  - Hence we have a contradiction.

Informed Search strategies

- Consistency of a heuristic
  - A heuristic is called consistent if

$$h(n) \leq c(n, a, n') + h(n')$$

    where $n \rightarrow n'$.
  - This is a version of triangle inequality (same thing which euclidean distances follow.)
  - Important: Every consistent heuristic is admissible!

Informed Search strategies

- Intuition behind why A^* works really well – Search Contours

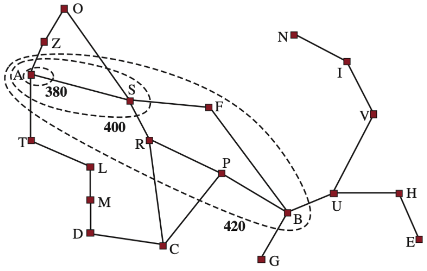


Figure 3.20 Map of Romania showing contours at $f = 380$, $f = 400$, and $f = 420$, with Arad as the start state. Nodes inside a given contour have $f = g + h$ costs less than or equal to the contour value.

- Contours for $g(n) + h(n)$ deviate towards the optimal state. So, we tend to do better.
- However, with only $g(n)$, we get concentric contours!

Informed Search strategies

- When is $f = g + h$ strictly monotonic?
  - Say you go from node $n \to n'$, then,

$$g(n) + h(n) \to g(n') + c(n, a, n') + h(n')$$

  - So, if it is consistent (i.e triange inequality) then we have that the costs are strictly monotonic.

Informed Search strategies

- Intuitively, how many nodes will I *expand* before reaching the optimum?

– If C^* is the optimum,

  - if $f(n) < C^*$ these nodes will be **surely expanded**
  - if $f(n) = C^*$ then these nodes *might* be expanded!
  - if $f(n) > C^*$ then these nodes will not be reached!

Informed Search strategies

- The “pruning” Intuition
  - One way to look at A^* search is that is prunes those branches which *would not* lead to an optimal solution.
  - A^* with a consistent heuristic is *optimally* efficient – which means it expands all states with $f(n) < C^*$.

Informed Search strategies

- Does it mean – A^* is perfect? – Unfortunately NO!
  - The number of nodes with $f(n) < C^*$ can still be almost all the nodes!
  - Example - Vaccum World with N squares has 2^N states which have $f(n) < C^*$.

Informed Search strategies

- Weighted A^* search
  - By considering *inadmissible* heuristics, we might be able to do better. We multiply the heuristic with some constant W

$$f(n) = g(n) + Wh(n)$$

  - This gives us a way to "interpolate" the extremes

$$\begin{aligned}
A^* \text{ search } g(n) + h(n) &\qquad W = 1 \\
\text{Uniform Cost search } g(n) &\qquad W = 0 \\
\text{Best first search } h(n) &\qquad W = \infty \\
\text{Weighted } A^* \text{ search } g(n) + Wh(n) &\qquad 1 < W < \infty
\end{aligned}$$

Informed Search strategies

- Tricks to improve A^* search
  - Prune the frontier by removing the nodes which would not lead to an optimal solution.
  - **Beam Search:** Always keep only the top-k nodes.

Designing Heuristic Functions

• So, having a “good” heuristic helps a lot in improving our search for optimal path. But how do we design such a heuristic?

- Before that we need *a way to evaluate a heuristic*
- Tricks to construct heuristics:
    - Relaxed Problems
    - Combining Admissible Heuristic
    - Pattern Databases
    - Landmarks
    - Learning???

Designing Heuristic Functions

- Evaluating a heuristic - Effective Branching Factor and Effective Depth
  - Recall number of nodes we visit

$$N + 1 = 1 + b + b^2 + \cdots b^d$$

  - Ideally we want b and d to be as small as possible.
  - For example, if we get a solution at depth 5, and have visited 52 nodes, then solving the above fpr b will give 1.92.
  - In practice, we take a few examples of small problems run simultations collect the b and d values.

Designing Heuristic Functions

- Example : 8-puzzle

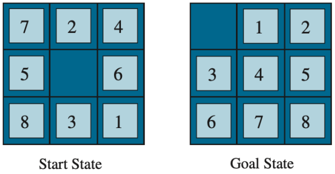


Figure 3.25 A typical instance of the 8-puzzle. The shortest solution is 26 actions long.

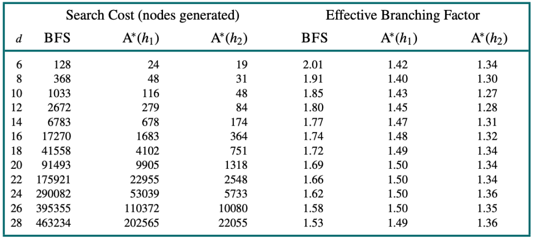
- Two possible heuristics
  - h_1 denotes the number of misplaced tiles. In this example this is 8.
  - h_2 Manhattan Distance

$$\sum_i |x_i - x_j| + |y_i - y_j|$$

In this example this is 18

Designing Heuristic Functions

- How do these heuristics compare??



	Search Cost (nodes generated)			Effective Branching Factor		
d	BFS	$A^*(h_1)$	$A^*(h_2)$	BFS	$A^*(h_1)$	$A^*(h_2)$
6	128	24	19	2.01	1.42	1.34
8	368	48	31	1.91	1.40	1.30
10	1033	116	48	1.85	1.43	1.27
12	2672	279	84	1.80	1.45	1.28
14	6783	678	174	1.77	1.47	1.31
16	17270	1683	364	1.74	1.48	1.32
18	41558	4102	751	1.72	1.49	1.34
20	91493	9905	1318	1.69	1.50	1.34
22	175921	22955	2548	1.66	1.50	1.34
24	290082	53039	5733	1.62	1.50	1.36
26	395355	110372	10080	1.58	1.50	1.35
28	463234	202565	22055	1.53	1.49	1.36

Figure 3.26 Comparison of the search costs and effective branching factors for 8-puzzle problems using breadth-first search, A^* with h_1 (misplaced tiles), and A^* with h_2 (Manhattan distance). Data are averaged over 100 puzzles for each solution length d from 6 to 28.

- Which is better?

Designing Heuristic Functions

- The concept of *dominating* heuristics-
  - Recall we said a heuristic is "admissible" is it does not overestimate the cost. And showed that this will reach the optimal solution.
  - But $h(n) = 0$ is always admissible! – Does it make it good?
  - What we want is it should not overestimate, and it should not underestimate either!
  - We say h_2 dominates h_1 if $h_2(n) \geq h_1(n)$ for all n, and if both are admissible, then we can be sure that h_2 will search less number of nodes than h_1.

Designing Heuristic Functions

- Generating heuristics from Relaxed problems
  - Recall that we model our problem as a graph.
  - Now, basically add "higher order edges" to your graph and search for a solution here.
  - This solution is an *admissible* heuristic!

Designing Heuristic Functions

- Example: 8-puzzle

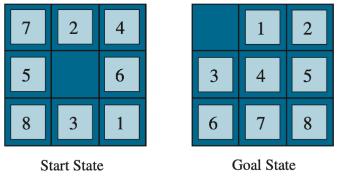


Figure 3.25 A typical instance of the 8-puzzle. The shortest solution is 26 actions long.

- Two possible heuristics
  - h_1 denotes the number of misplaced tiles – This is obtained by allowing the agent to simply remove the tile and place it at the right position.
  - h_2 manhattan distance – This is obtained by allowing the agent to have *more than one tile* at the same position! (Note: Technically this also increases the number if states in the graph!)

Designing Heuristic Functions

- Combining admissible heuristics
  - Suppose $h_1, h_2, \cdots h_n$ are some admissible heuristics, then

$$h(n) = \max\{h_1, h_2, \cdots h_n\} \tag{1}$$

    is also admissible – WHY??

  - Recall the "dominating" argument.
  - Here $h(n)$ dominates all h_i, but is still admissible because it would not overestimate the cost to the goal state.

Designing Heuristic Functions

- Pattern Databases
  - One way to add edges in the previous idea (relaxed problem) is by considering a subproblems.
  - So, we store the solutions (and costs) for each subproblem and use them to estimate the heuristic.
  - In practice, we generate this database by *moving back* from the goal state!
  - Hence the name **Pattern Databases**

Designing Heuristic Functions

Example: 8-Puzzle

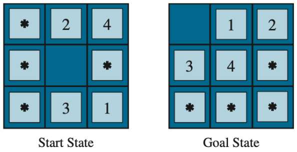


Figure 3.27 A subproblem of the 8-puzzle instance given in Figure 3.25. The task is to get tiles 1, 2, 3, 4, and the blank into their correct positions, without worrying about what happens to the other tiles.

- For example we can consider a subproblem by only asking one to place the $1, 2, 3, 4$ tiles in it's right position.

Designing Heuristic Functions

- Using Landmarks
  - Instead of saving all possible solutions (optimal paths) between vertices, we can choose a few *Landmark* points. Then, use

$$h_L(n) = \min_{L \in \text{landmarks}} C^*(n, L) + C^*(L, \text{goal}) \tag{1}$$

  - Unfortunately this might overestimate the optimal cost. (WHY?? - Hint: You might not have reach L to reach the goal!)
  - A small change **differential heuristic** corrects this by considering:

$$h_{DH}(n) = \max_L |C^*(n, L) - C^*(goal, L)|$$

  This is an optimal path to L via the goal.

Designing Heuristic Functions

- Learning to search??
  - Recall *markov* idea from previous class? – You can have a *meta state space* where each state is a tree in A^* search.
  - If we can find a way to remember these meta-states, you can train the model (think neural network) to use the tree and predict the action.

Designing Heuristic Functions

- Learning Heuristic?
  - Another way is to learn the heuristic itself by repeated experiments, collecting the data and learning.
  - Reinforcement Learning is a popular way to learn the these heuristics. We shall see this at the end of this course!