Branch and bound

Chief **References consulted : Prof Mausam's,IIT DELHI lectures**

Notion of a state

In simple words, all relevant information about the environment at a particular instant can be used to define a state.

How can we represent a state?

Technically, an ID can be given to each distinct state.

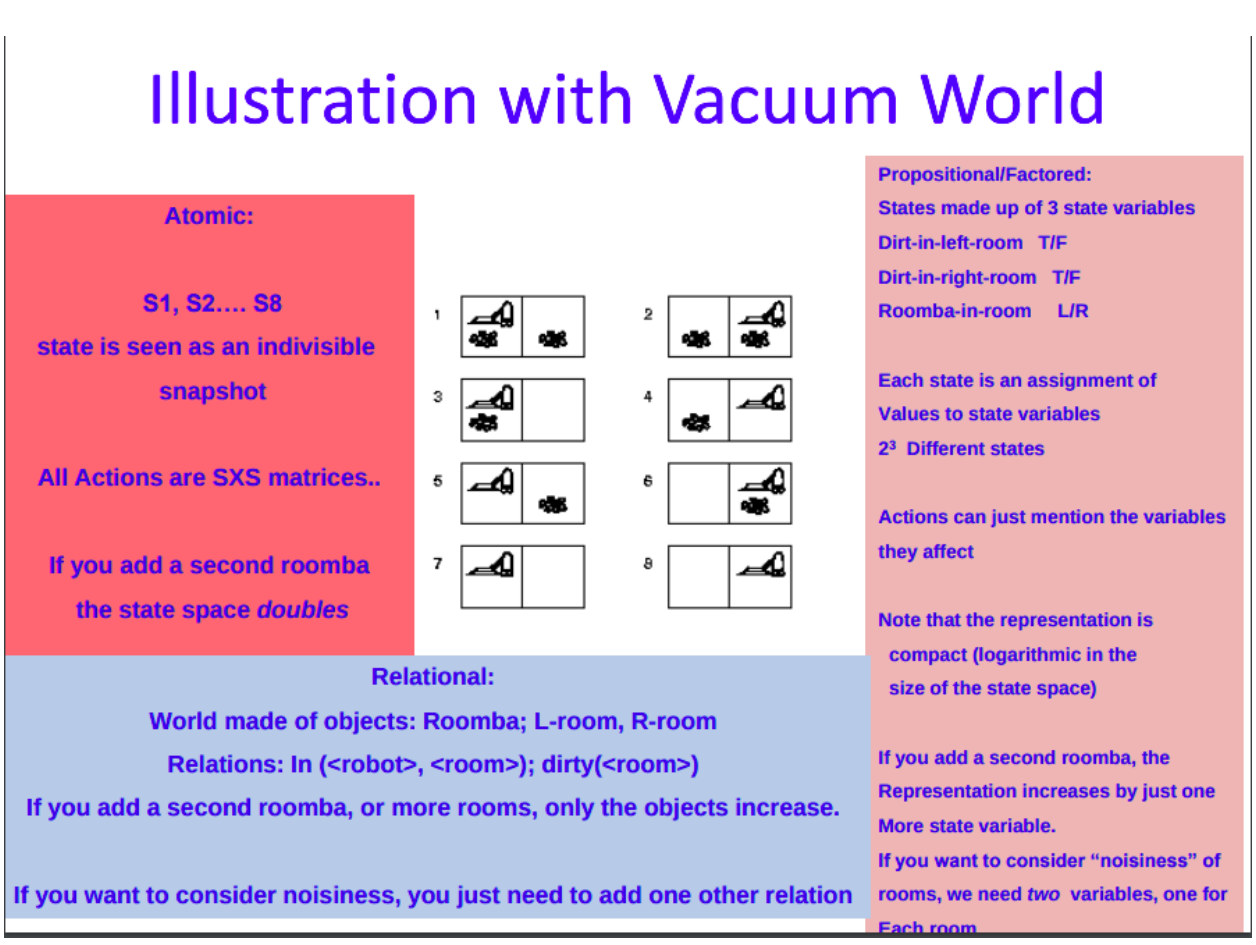
Let us consider the scenario where we are interested in the seats occupied by two students: Joe and Donald.

Scenario A: Joe sits at seat 46 and Donald sits at seat 45.

Scenario B: Joe sits at seat 46 and Donald sits at seat 200.

Now, states A and B can be given respective ids of say, state A : id=10 and state B : id=11.

An alternate way to represent the states would be in the form of tuples of state variables, which the state is comprised of. So, scenario A can be represented as **<46,45>,** and scenario B can be represented as **<46,200>**. Representing the states in this way and thinking in terms of state variables is easier to visualize as local moves often involve going from one state S to another state S' where **S and S'** have the same value of some state variables and different values for the remaining state variables.



Understanding the difference between modelling a problem and solving it.

[credit: content covered in Prof Mausam's lectures]

Imagine the **states** as the **nodes of a graph**.

Defining a goal: Involves describing a situation we want to achieve, a set of properties that we want to hold, etc. This requires defining a "goal test" so that we know what it means to have achieved/satisfied our goal.

Imagine a **black box A** which takes a **state S as input** and gives a **boolean output** [Y/N] indicating whether the current state is a GOAL state(desired state) or not. Often there can be many goal states as many states may be acceptable.

An edge A → B can be thought of as a move from state A to state B. Each edge also has additional properties as follows:

- The **type of move** which was made (in the case that a state is being represented as a tuple of values of decision variables, a move type can be defined as per which particular variables undergo a change in value while moving from source node to destination node). This can be additionally thought of as the type of **operator** applied to a state A to move to state B.
- The **cost associated** with making the move.

Modeling the problem would be to

1. Find a way to represent the states in the problem.
2. Deciding which nodes (states) are reachable from other states.
3. The cost associated with such a move.

Solving the problem would involve the algorithm used to **make the decision of which state S' to select as the next state from a list of possible neighbours of current state S**.

Search strategies judging criterias

A search strategy is defined by picking the order of node expansion. Strategies can be evaluated along the following dimensions:

- **Completeness:** Does it always find a solution if at least one exists?
- **Time complexity:** number of nodes generated (worst case)
- **Space complexity:** maximum number of nodes in memory (worst case)
- **Optimality:** Does it always find a least-cost solution (solution achieved by traversing the minimum possible sum of edge weights)?
- **Systematicity:** Does it visit every node at most once?

Time and space complexity are measured in terms of:

- b **:** maximum branching factor of the search tree (approx. how many actions can one take at a given node to move to the next node)
- d : depth of the shallowest goal node
- m : maximum depth of any path in the state space (may potentially be ∞)

Search strategies

- **Uninformed search**: All non-goal nodes in frontier look equally good.
- **Informed search:** Some non-goal nodes can be ranked above others.

UNINFORMED SEARCH STRATEGIES

An uninformed (a.k.a. blind, brute-force) search algorithm **generates the search tree without using any domain-specific knowledge**. These search strategies use only the information available in the problem definition. Such strategies **do not take into account the location of the goal**. They ignore where they are going until they find a goal and report success.

- **Depth First Search** (expand deepest node yet first) **:**

1. This strategy maintains a **stack** of nodes it is currently exploring.
2. **Completeness:** No, as m may be infinite.
3. **Time Complexity**: $O(b^m)$
4. **Space Complexity:** $O(bm)$

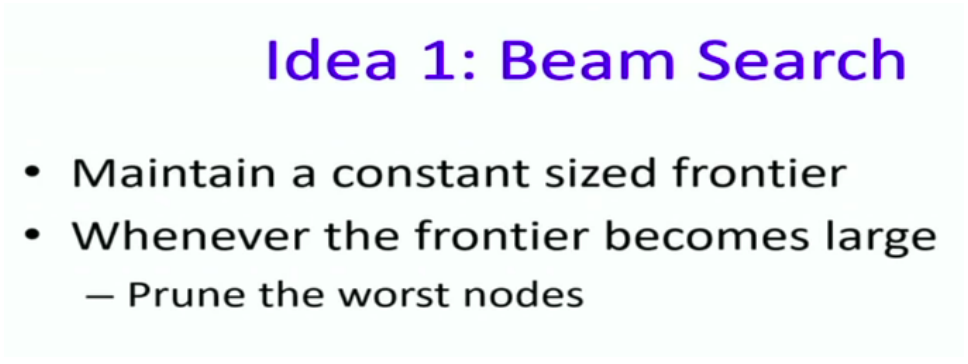
https://www.youtube.com/watch?v=dtoFAvtVE4U

Breadth First Search (expand shortest node first):

1. This strategy maintains a queue of nodes it is currently exploring.
2. Least cost and shortest may not be the same.
3. **Completeness:** Yes (b is finite).
4. **Time Complexity**: $O(b^d)$
5. **Space Complexity**:$O(b^d)$
6. **Optimal** when **step cost is 1**.

- **Uniform Cost Search** (expand cheapest node first)**:**
  1. This strategy maintains a **queue** of nodes it is currently exploring.
  2. **Completeness:** Yes (b is finite).
  3. **Time Complexity:** Let C^* be the cost of the optimal solution, and ε be at least each step to get closer to the goal node. Then the number of steps is = C^*/ε. Hence, the worst-case time complexity of Uniform-cost search is $O(b^{C^*/\varepsilon})$.
  4. **Space Complexity:** The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is $O(b^{C^*/\varepsilon})$.
  5. **Optimal**
- Space → DFS wins
- Time → BFS wins as 'd' may be less than 'm'

https://www.youtube.com/watch?v=z6lUnb9ktkE



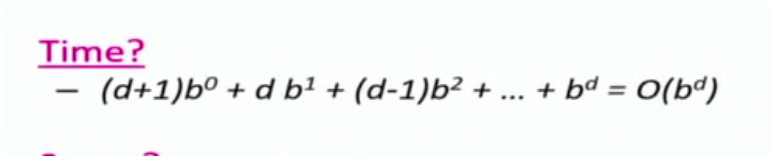
- It is possible that whatever node was taking us to the goal got pruned, hence beam=not optimal
- Optimal=NO
- Complete=NO

Iterative deepening search

- Do a DFS, stop in the middle and repeat.
- Is it complete? YES, I don't need to expand after reaching depth 'd' (which we assume is finite)

- Time here is pretty terrible as if we suppose that the target is at depth d', then a node at level 0 will be reached (n+1) times, a node at node '1' will be checked 'n' times, and so on.

Notice the **ARITHMETIC-GEOMETRIC PROGRESSION**



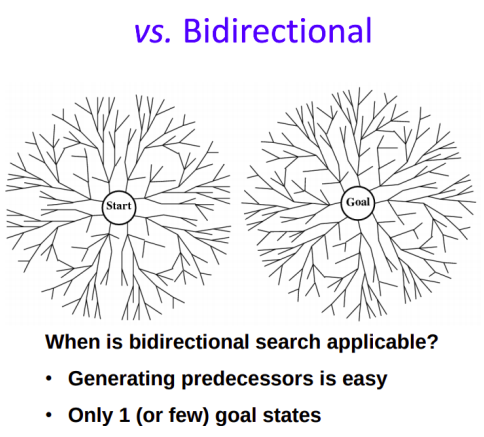
Space : $O(bd)$

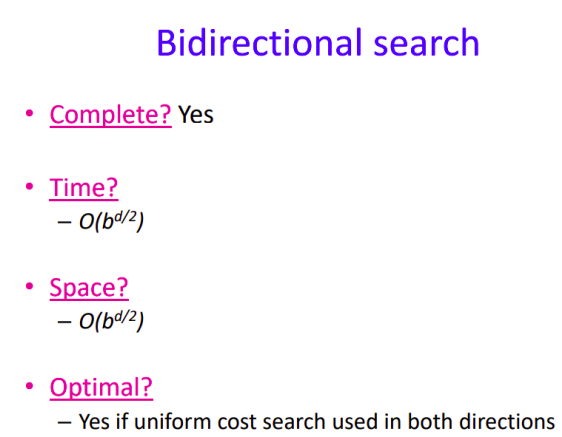
Optimality : Not optimal (optimal only in the special case when all costs are equal)

Systematic : It does not require to touch all nodes atleast once.

Bidirectional search

- When is bidirectional search applicable ??
- **Initial inputs:** Starting state + edges in forward direction
- For such a search, we also need an expansion function for backward direction (a way to traverse edges in a reverse manner)
- **When should we use it ?**





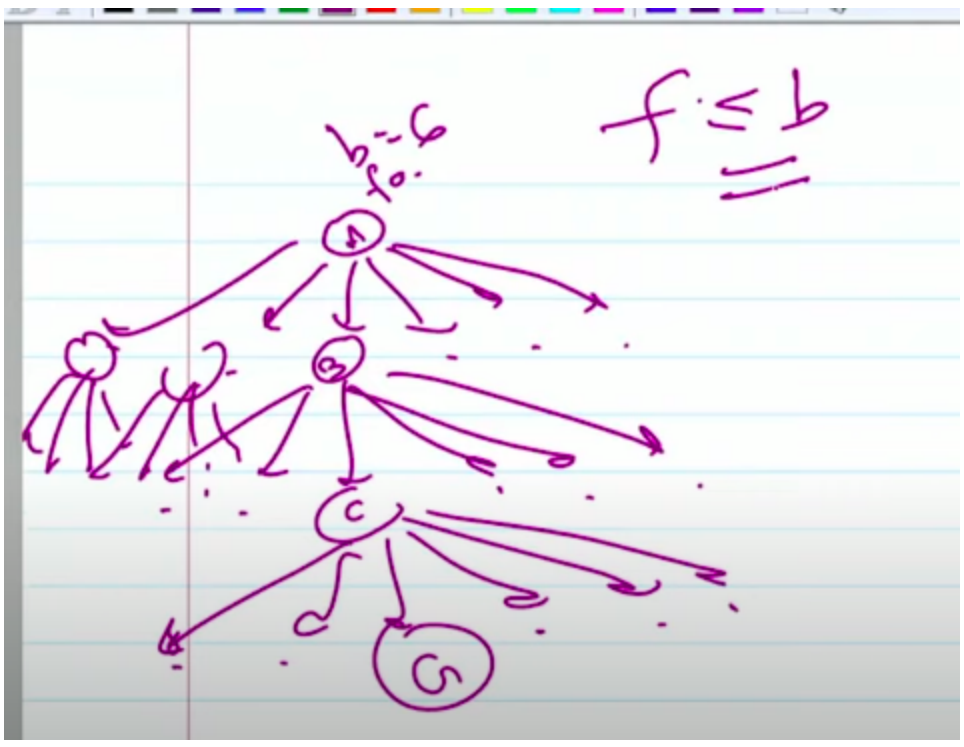
- **Complete?** Yes as an expansion from both ends, the forward and backward pointers must meet somewhere. If done alternatively, they will meet at a distance of d/2.
- **Time** → instead of one search of depth 'd', we are doing 2 searches of depth d/2 and so time = $O(b^{[d/2]}) + O(b^{[d/2]}) = O(b^{[d/2]})$
- **Optimal** : No in general. Yes if instead of BFS, Uniform cost search is done in both directions.

My concerns

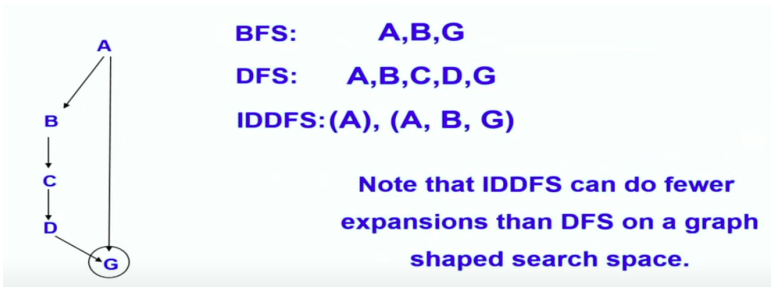
If goal state is known, why is there even a need to search stuff.

My view

Apparently, a huge role in whether to choose forward search or backward search is of the degree of in-edges and out-edges.



As can be seen in this diagram, the out-factor (forward)~6 but in-factor(backward)=1 as each node seems to have only one parent. In such a case, it seems better to move backward as we would only explore four nodes in the process rather than exploring ~ approx 6^4 nodes.



Repetition of nodes in case of bidirectional edges is a huge problem

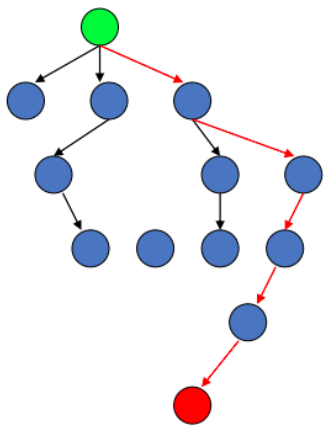
INFORMED SEARCH

Uninformed search algorithms have the con of searching in all directions. They are not guided towards the goal in any specific way. Hence, it is desirable to add this guidance factor to the searches we perform.

"Intuition, like the rays of the sun, acts only in an inflexibly straight line; it can guess right only on condition of never diverting its gaze; the freaks of chance disturb it."

The objective is to be smart about what paths we must try.

Idea: be **smart** about what paths to try.



The motive is to capture our intuition in terms of a particular neighbour A seems to be better (intuitively speaking) and is more likely to reach an acceptable solution than another neighbour B.

In such a case, a node N is selected as a successor to the current node **based on an evaluation function that estimates cost to goal.**

📌 fringe=open list=frontier

📌 closed-list=explored-list

General tree search paradigm

General Tree Search Paradigm

```
function tree-search(root-node)
  fringe ← successors(root-node)
  while ( notempty(fringe) )
        {node ← remove-first(fringe) //lowest f value
           state ← state(node)
           if goal-test(state) return solution(node)
           fringe ← insert-all(successors(node),fringe) }
  return failure
end tree-search
```

General graph search paradigm

Differs from tree search paradigm in the notion that in a tree, we are sure that there are no repeated vertices. But in a graph search algo, there can be repeated vertices and we must make sure that we do not check already explored vertices by keeping a EXPLORED_LIST.

BEST FIRST SEARCH

General Graph Search Paradigm

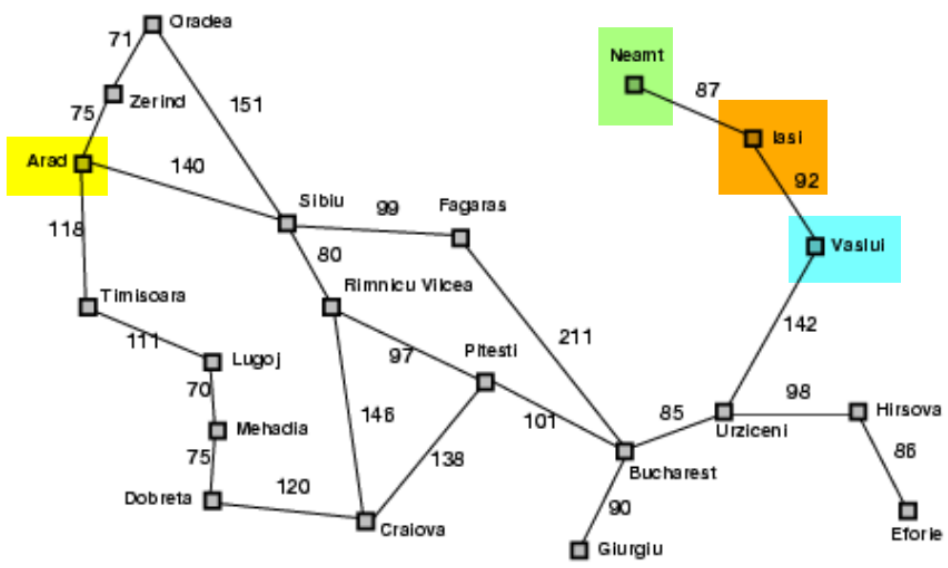
```
function tree-search(root-node)
  fringe ← successors(root-node)
  explored ← empty
  while ( notempty(fringe) )
        {node ← remove-first(fringe)
           state ← state(node)
           if goal-test(state) return solution(node)
           explored ← insert(node,explored)
           fringe ← insert-all(successors(node),fringe, if node not in explored)
        }
  return failure
end tree-search
```

Any search strategy can be identified by observing the order in which it expands its nodes. DFS→ depth, BFS→ all nodes at a given distance in ascending order etc

Likewise, Best First search → expands in order of lowest f value where f is the evaluation $f(n)$ for node n.
Always choose the node from fringe(open list) that has the lowest f value.

Intuitively, expand the most desirable unexpanded note.

> Summarizing, DFS→ stack, BFS→ queue, Best first search → priority queue



1. Let us say our initial state = City Aradh
2. Goal state = Reaching City Bucharest
3. Before any further matter on best-first search, one can notice that we can express the uninformed search strategies as special cases of Best First Search :
    1. **BFS→ f = depth**
    2. **Uniform cost search →** f **= length of path to go from origin to node**

Let us define 3 functions:

1. $f(n)$ **= evaluation function to choose which neighbour (chosen from the fringe) must be expanded next**
2. $g(n)$ **= sum of edge costs traversed to reach a node** n **from origin (starting state)**
3. $h(n)$**= guess (estimate) of the sum of edge costs which we would need to traverse from current node** n **to the closest goal**

Now, in the map of Romania, we can choose the straight-line Euclidean distance to the target city as our $h(n)$ where 'n' is the node being judged. So, if we want to minimize this, $f(n) = h(n)$

But let's say we want to go from ORANGE TO YELLOW and the implementation is based on the tree search version (we are not keeping a note of the already explored nodes) , then such a defined heuristic will get stuck in a cycle, as :

When current state = ORANGE, our algorithm will choose GREEN

When current state = GREEN, our algorithm will choose ORANGE

And hence, the algorithm will keep moving in a cycle. Hence, choosing minimization of Euclidean distance as $h(n)$ is not a COMPLETE strategy.

Intuitively, we can see why this is happening.

The overall cost of reaching would be g(n)+h(n). But our evaluation function f(n) = h(n) is only judging based on the second half of the journey ie h(n). So, while moving between the 2 nodes in a cyclic manner, g(GREEN) keeps on increasing as 1*87, 3*87, 5*87, and so on. But, our chosen heuristic, in this case, seems to ignore this fact and hence, the inefficiency.

FOR THE TREE BASED IMPLEMENTATION

Properties of greedy best-first search

- Complete?
- No – can get stuck in loops, e.g., Iasi → Neamt → Iasi → Neamt →
- Time?
- $O(b^m)$, but a good heuristic can give dramatic improvement
- Space?
- $O(b^m)$ -- keeps all nodes in memory
- Optimal?
- No

As a result, we consider the A* algorithm which seems to keep track of both the parts of the journey.

A*- a Special Best-First Search

- Goal: find a minimum sum-cost path
- Notation:
  - c(n,n') - cost of arc (n,n')
  - g(n) = cost of current path from start to node n in the search tree.
  - h(n) = estimate of the cheapest cost of a path from n to a goal.
  - evaluation function: f = g+h
- f(n) estimates the cheapest cost solution path that goes through n.
  - h*(n) is the true cheapest cost from n to a goal.
  - g*(n) is the true shortest path from the start s, to n.
  - C* is the cost of optimal solution.
- If the heuristic function, h always underestimates the true cost (h(n) is smaller than h*(n)), then A* is guaranteed to find an optimal solution.

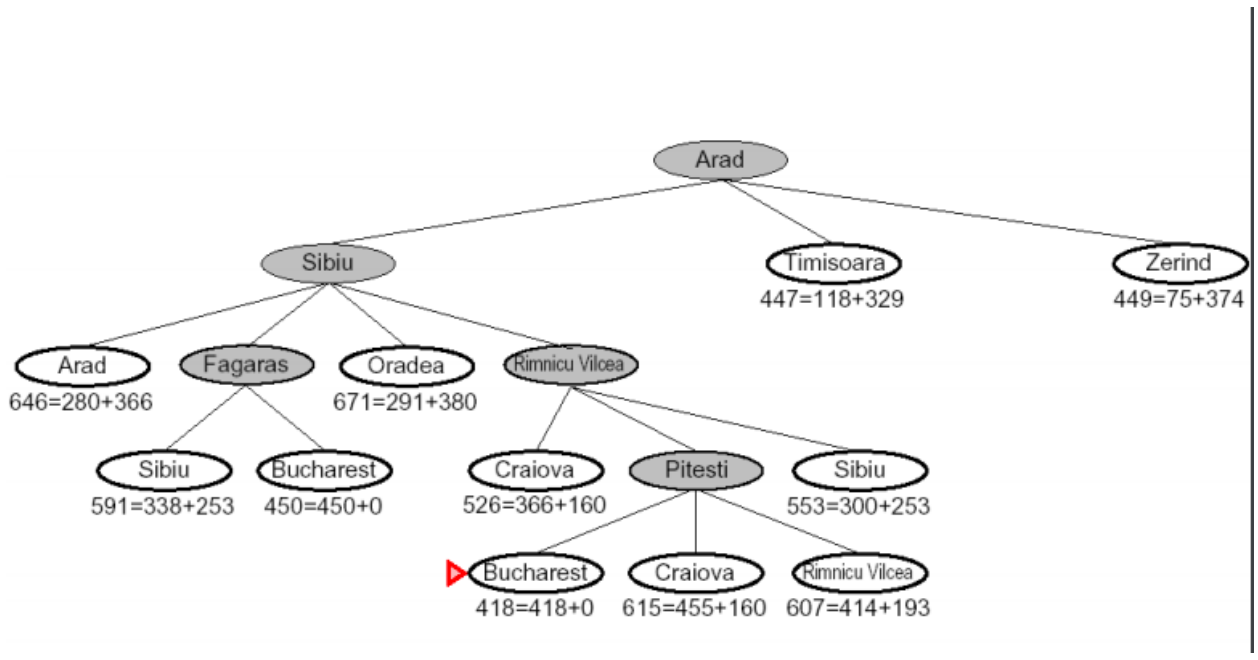
A * algorithm

A* is basically an extended BFS that prioritizes the shortest routes that can be reached first and then the other routes. It is optimal and complete. So, for the above map, it easy to calculate g(n) since we are traversing along the edges to reach node n from the origin. h(n) is estimated as the euclidean distance from node 'n' to the target goal node. In such a case, f(n) can be written as

$$f(n) = \alpha\, g(n) + \beta\, h(n) \quad where\ \alpha, \beta > 0$$

For simplicity, let's assume that $\alpha = \beta = 1$.

Now attempting to go from Arad to Bucharest,



A thing to observe is that we reach Bucharest from Fagaras by incurring a cost of 450 but at that point of time, we would still have Pitesti as unexpanded but with $g(Pitesti)$ lesser than 450. So, that still makes it possible for a $g(Pitesti)$ + $h(Pitesti)$ to have a lesser cost than 450 and so, it would be imperative to check for all such cities unless we are able to conclude that for a city C, $g(city) + h(city)$ cannot be less than 450.

Whether such an algorithm is optimal or not would depend on the effectiveness of the selected heuristic function.

Admissible heuristics

Definition of being admissible : A heuristic $h(n)$ is **admissible** if for every node n,

$$" \leq ",$$

where $h^*(n)$ is the true cost to reach the goal state from n.

From definition, we can conclude that an admissible heuristic **never overestimates** the cost to reach the goal, i.e., it is optimistic.

An example of an admissible heuristic would be the one we used above to calculate Arad's distance to Bucharest. In the above-used heuristic,

$h^*(n)$ **: the true cost to reach the goal state from** n**.**

$h(n)$ **: straight line (Euclidean distance) distance from node** n **to goal node.**

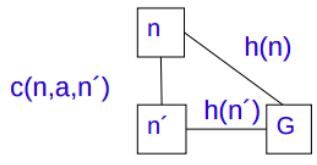
Since the straight line distance between 2 points is always less than the distance along any other path, the inequality $h(n) \leq h^*(n)$ holds for all pair of coordinates.

Theorem: If $h(n)$ is admissible, A^* using **TREE-SEARCH** version is optimal [in the case that GRAPH-SEARCH version is used, admissibility is a necessary but not a sufficient condition for optimality].

Consistent heuristics

Consistent Heuristics

- h(n) is consistent if
  - for every node n
  - for every successor n´ due to legal action a
  - h(n) <= c(n,a,n´) + h(n´)



- Every consistent heuristic is also admissible.
- Theorem: If *h(n)* is consistent, A* using GRAPH-SEARCH is optimal

- Tree search version is optimal when heuristic is admissible

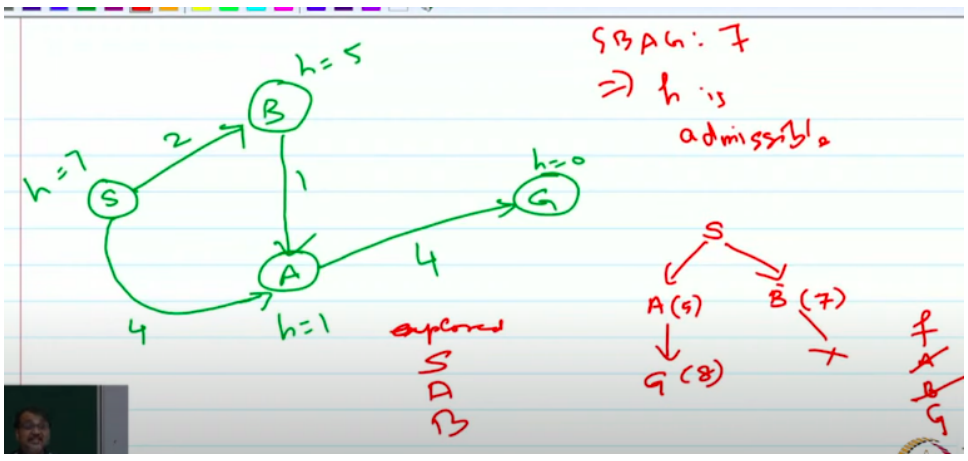
- Graph search version is optimal when heuristic is consistent
- Consistency subsumes admissibility, so if one can conclude that consistency $\rightarrow$ admissibility

Some intuitions:

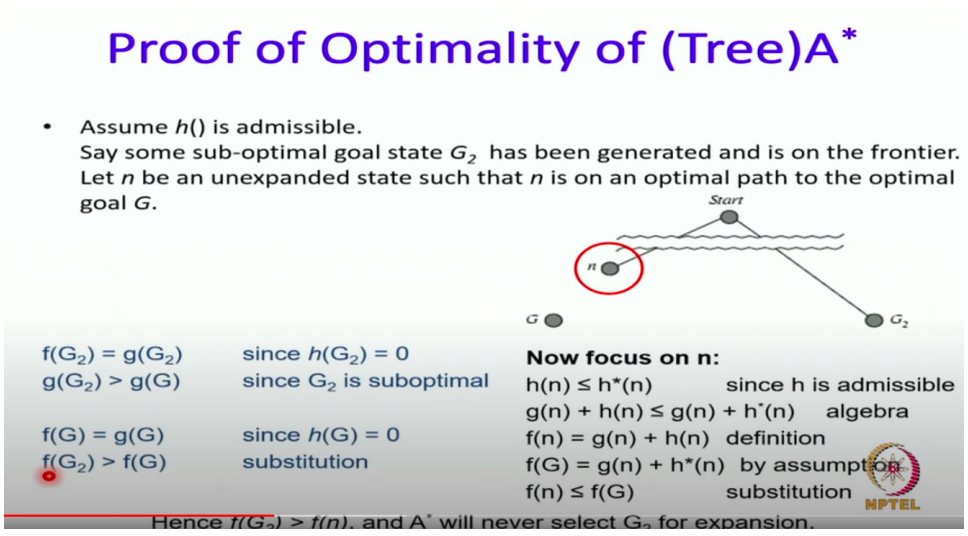
Scenario: Admissible heuristics on graph search version of A^*

Example that in graph search version of A^*, admissibility does not imply optimality.

Essentially, SBAG (cost=7) is the most optimal path, But on applying A^* graph version here, we get path SAG (cost=8) as the proposed final solution, which is not optimal. Intuitively, this is happening as A's heuristic function dupes into thinking the second phase of its journey (predicted cost=1 instead of actual cost-4) makes it an overall better option than B, and hence, we make the wrong choice. Now, since A was put into the EXPLORED_LIST, when B tries to explore a new path having the edge B→A, the search does not consider this as A has already been put into the EXPLORED_LIST.



Proof of optimality of TREE version of A^*



https://www.youtube.com/watch?v=huJEgJ82360