SEARCH STRATEGIES

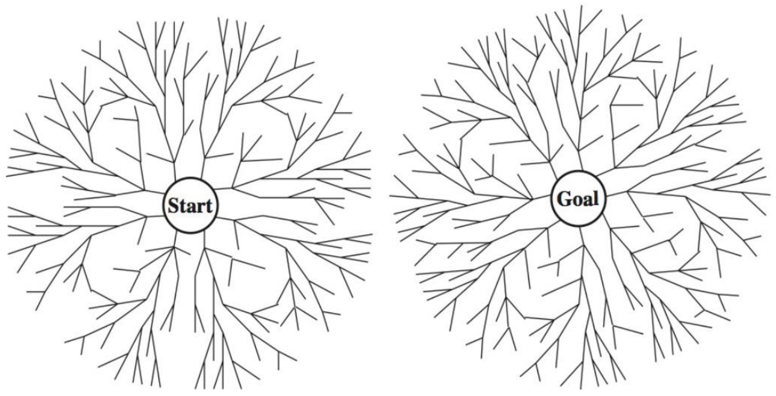
Recall all the search algorithm we consider have the same basic structure:

1. Start with a priority queue consisting of just the initial state
2. Choose a state from the queue of states which have been generated but not yet expanded
3. Check if the selected state is a GOAL STATE. If it is, STOP (solution has been found)
4. Otherwise, expand the chosen state by applying all possible transitions and generating all its children
5. If the queue is empty, stop (no solution exists)
6. Otherwise, go back to step 2

BEST-FIRST SEARCH

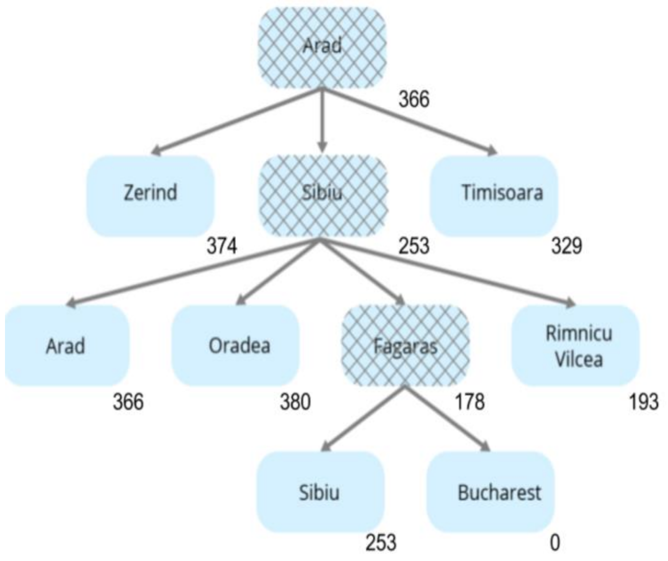
- You should recall that BFS and DFS treat all node the same way
    - BFS adds all new nodes to the back of the queue
    - DFS adds all new nodes to the front of the queue
- BEST-FIRST SEARCH is a general term for more sophisticated(complicated) algorithms which use an evaluation function to try to guess which would be the best node to expand next. Normally, there is a function f(n) which assigns a number to every previously generated node n, the node n chosen to be expanded next is the one with the smallest value of f(n).
    - F(n) is the way you choose the best node, e.g. choose the shortest path, etc.
    - ** we don't know for sure that this node is the best one, we are just choosing what seems to be the best node, based on the information currently available.
- We have already seen one example of best-first search – namely, UNIFORM COST SEARCH:
    - UCS, f(n) = the cost g(n) of the path from the start node to n.
    - For GREEDY SEARCH, f(n) = an estimate h(n) of the cost if getting to goal from node n.
    - For A*SEARCH, f(n) = g(n) + h(n). (which is a combination of UNIFORM-COST SEARCH AND GREEDY SEARCH)

BIDIRECTIONAL SEARCH



- Bidirectional Search searches from BOTH the initial state moving forward and from the goal moving backwards and stops when two searches meet in the middle.
- However, we need an efficient way to check if a new node already appears in the order half of the search. The complexity analysis assumes this can be done in constant time, using a HASH TABLE.
- If we assume a branching factor b in both directions and that there is a solution at depth = d, then bidirectional search finds a solution in $O\left(2 * b^{\frac{d}{2}}\right) = O(b^{\frac{d}{2}})$ time steps.
- Issues with Bidirectional search. As every search there are strengths and weaknesses. Here are some of the issues with bidirectional search:
    - Searching backwards means generating predecessors starting from the goal, which may be difficult or impossible in some domains
    - There can be several goals, for example, in chess there are many checkmate positions
        - The Romania problem is not ideal for illustrating bidirectional search.
    - Space complexity: $O(b^{\frac{d}{2}})$ because the nodes of at least one half must be kept in memory.

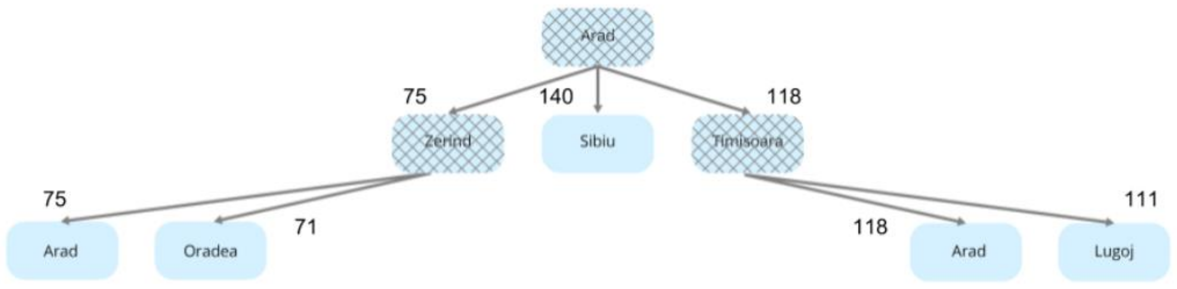
①GREEDY BEST-FIRST SEARCH



- Greedy search is a Best-First search that selects the next node for expansion using the heuristic function for its evaluation. $f(n) = h(n)$
- We assume that $h(n) = 0$ if and only if n is a goal state.
    - i.e. greedy search minimizes the estimated cost to the goal; it expands whichever node n is estimated to be closest to the goal.
        - The Romania problem in the graph above, we're given the distance between the current node to the goal state, Bucharest, every time we choose the generated node that has the shortest distance to Bucharest, and expand it, till the city to the goal state becomes 0, we stop.
- **COMPLETE**? NO, it can get stuck in loops, however, if we modify the algorithm to avoid repeated states along a path, it becomes complete in finite spaces.
- **TIME complexity**: $O(b^m)$ where m is the maximum depth of the state space.
- **SPACE complexity**: $O(b^m)$ (retains all nodes in memory)
- **OPTIMAL**? NO.
- In the worst case, greedy search suffers the same drawbacks as depth-first search, however, with a good heuristic, it can often reduce the time and memory costs substantially, compare to breadth-first search.

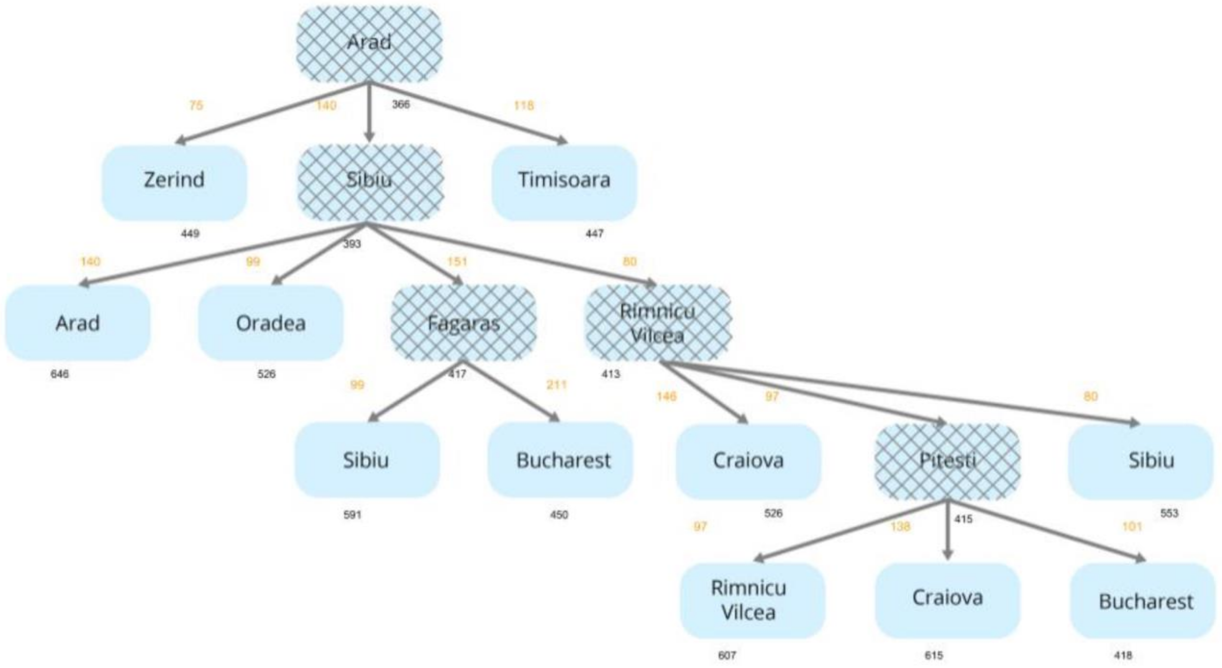
!!②A*SEARCH

- RECALL: UNIFORM-COST SEARCH
  For uniform cost search, we always expand the node whose total path length from the start node is the shortest. This reduces to breadth first search when all actions have the same cost. UCS finds the cheapest goal provided path cost is monotonically increasing along each path.
  IMPLEMENTATION: QUEUEING FUNCTION = insert nodes in order of increasing path cost.

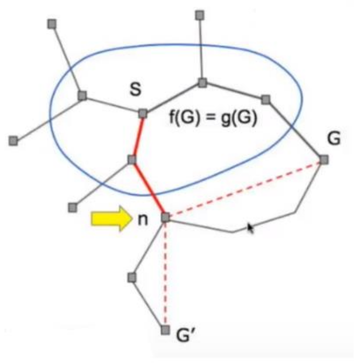


- A*SEARCH is a combination of GREEDY SEACH and UNIFORM COST SEARCH.
- A*SEARCH uses evaluation function $f(n) = g(n) + h(n)$

- $g(n) = the\ cost\ from\ initial\ node\ to\ node\ n$
    - From uniform cost search: it is optimal and complete but not efficient
- $h(n) = the\ estimated\ cost\ of\ cheapest\ path\ from\ n\ to\ goal$
    - from greedy search, it is efficient, but not optimal nor complete
- $f(n) = the\ estimated\ total\ cost\ of\ cheapest\ solution\ through\ node\ n$
- IDEA: preserve(keep) the efficiency of greedy search but avoid to expanding paths that are already expensive



- The black numbers are the (total distance of current path) + (the distance from the current node to goal state)
    - Total distance of current path = (e.g.) distance(Arad, Sibiu) + distance(Sibiu, Rimnicu Vilcea), etc.
    - The distance from the current node to goal state = e.g. if we are in Sibiu, we only add one extra number which is the distance from Sibiu to Bucharest, and it changes when we moves to the next node.
- **COMPLETE**? YES, unless there are infinitely many nodes with $f \leq cost$ of solution.
- **TIME complexity**: exponential in (relative error in hx length of solution)
- **SPACE complexity**: keeps all nodes in memory
- **OPTIMAL**? YES, (assuming h() is admissible)
- Proof that A*SEARCH is OPTIMAL:



- Assume G is selected for expansion.
  Then f(G) = g(G) + h(G)
  = g(G) (when G is the goal h(G) equals to zero)
- Consider another path from S to G or G`. let n be the last unexpanded node on this alternative path from S to G or G`. then f(G) ≤ f(n), g(G) ≤ g(n) + h(n). so the original path to G is the shortest.

ITERATIVE DEEPENING A*SEARCH

- Iterative deepening A* is a low-memory variant of A* which performs a series of depth-first searches, but cuts off each search when the sum $f() = g() + h()$ exceeds some predefined threshold.
- The threshold is steadily increased with each successive search.
- IDA* is asymptotically as efficient as A* for domains where the number of states grows exponentially.