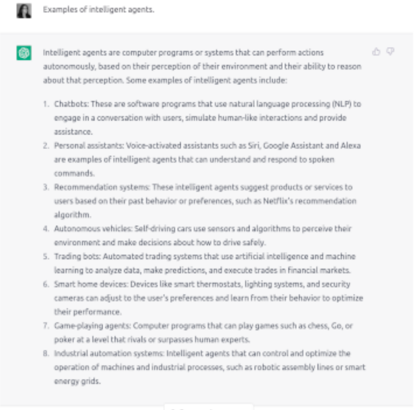
Search Algorithms

M. Mitrović Dankulov

April 20, 2023

Outline

Examples of intelligent agents



Types of agents

- Simple Reflex Agent
- Model-based reflex agent
- Goal-based agents
- Utility-based agent
- Learning agent

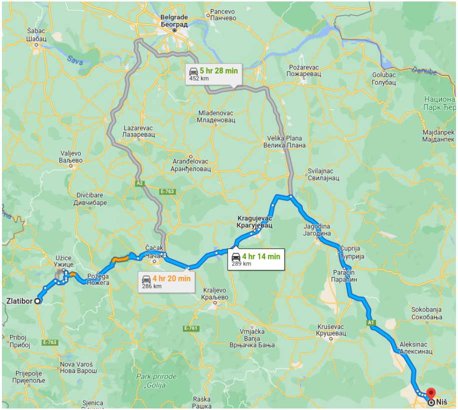
Problem-solving agents

- A special type of goal-based agent
- Goal-based Agents expand the capabilities of the model-based agent by having the "goal" information
- Their choice of action depends on the goal
- These agents may have to consider a long sequence of possible actions before deciding whether the goal is achieved or not; considerations of different scenario are called searching and planning, which makes an agent proactive

Example: tourist in Serbia

- Tourist is Zlatibor mountain; she can have different goals but she has a flight from Niš tomorrow morning; **Goal formulation** - she needs to be in Niš in time
- **Problem formulation** - given a goal what actions and states to consider; our agent needs to drive to Niš
- If the agent does not have a map the environment is **unknown**; our agent has a map - she needs to find a path from Zlatibor to Niš and then carries out her journey

Example: tourist in Serbia



Agent needs to examine future actions and decided on the best choice of path

- Environment is observable, discrete, known, deterministic - the solution to any problem is a fixed sequence of actions
- **Search** - process s of looking for a sequence of actions that reaches the goal
- **Search algorithm** - takes a problem as input and returns **solution**

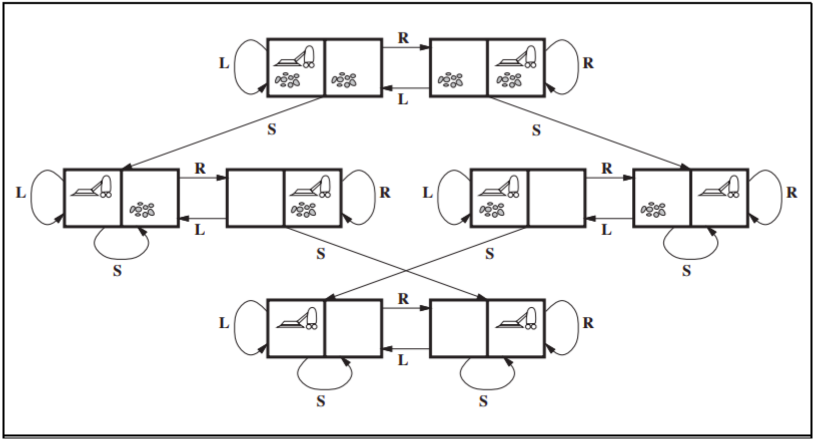
Problem elements

- **Initial state** - the state that agent starts in; Zlatibor for our agent
- List of possible **actions**; *s* state of the agent - ACTIONS(s) set of actions that can be executed in *s*;
- **Transition model** - description what action does; **state space** - all states reachable from the initial state; all states create a **graph** - nodes are states and links are actions
- **Goal state** - determines if the given state is a goal state
- **Path cost** - sets a numeric cost of each path

Formulating a problem

- **State abstraction** - simplification of the problem; states can be more complex than the need of the problem
- **Action abstraction** - simplification of actions
- **Level of abstraction** - usage of highroads or driving though mountains; abstraction is useful if carrying out each of the actions in the solution is easier than the original problem;
- **Toy problem** - we omit as much as details as possible but keep the problem and solution; toy problem vs. real-world problems

World vacuum cleaner - state graph



Route-finding problem - airplane route

- States - airport position and time
- Initial state - user's location
- Action - take a flight from current location
- Goal test - is current location goal location
- Path cost - monetary cost, time cost, number of stops, etc.

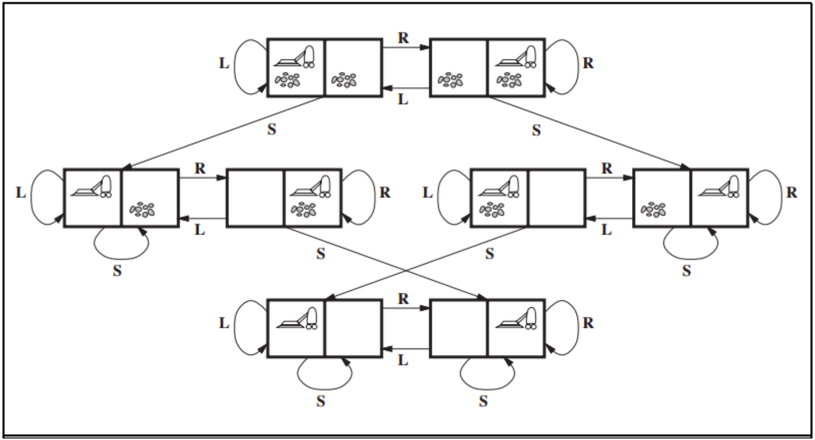
Real-world problems - other examples

- Touring problem - visit every city at least once; we need to known each city that agent visited
- Traveling salesperson problem - touring problem where we visit each city only once
- VLSI layout problem - position of millions components on the limited space; packing problem; cell layout and channel routing
- Robot navigation - route-finding in continuous space
- Automatic assembly sequencing - finding order in which to assemble an object

Search algorithm - definition

- We have defined problem - solution is an sequence of actions that solves a problem
- Action sequences form a **search tree** - edges are actions and nodes are states
- First we check if initial state is a goal state; than we **expand** from the initial state; expanding state - apply all legal action to current state and **generate** new set of states; we go from **parent node** to child node
- Essence of search - following one option and put other aside for latter

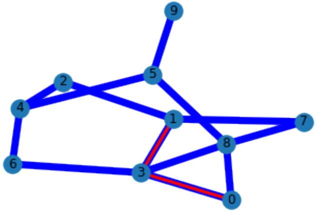
Example: vacuum cleaner

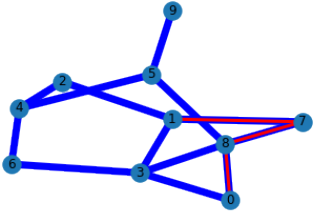


leaf nodes; repeated state; loopy path; redundant path;

Paths in networks

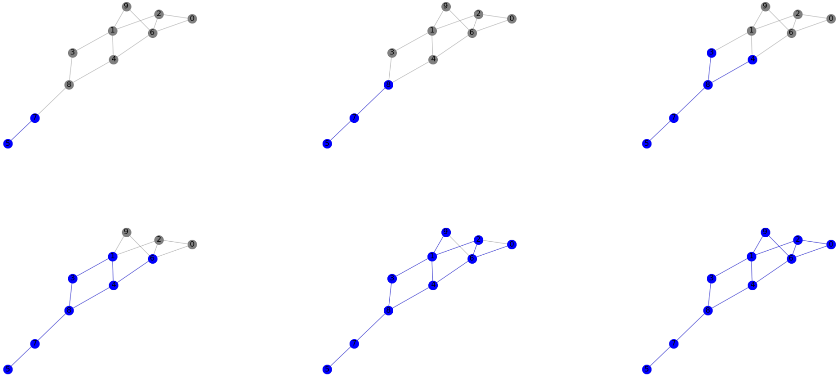
Redundant paths are common





Explored set or **closed list** - list of every expanded state

Graph-search algorithms



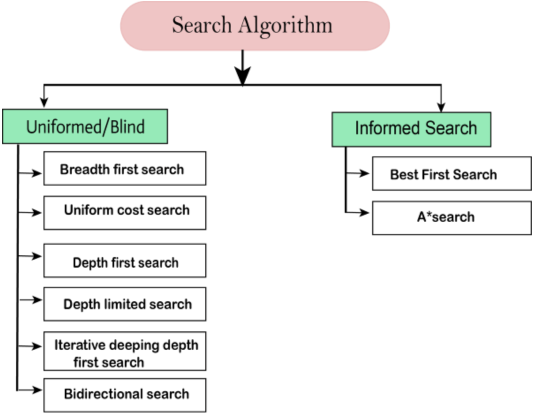
Search algorithm - infrastructure

- We need data structure to keep track of search tree
- For each node n of the tree:
  - $n.State$ - the state in the state space to which the node corresponds
  - $n.Parent$ - the node in the search tree that generated this node
  - $n.Action$ - the action that was applied to the parent to generate the node
  - $n.Path - Cost$ - $g(n)$ the cost of the path from the initial state to the node, indicated by parent pointers
- Nodes is different from state - we can have more nodes for the same state
- **Queue** - EMPTY?, POP and INSERT; FIFO queue, LIFO queue, priority queue

Measuring problem-solving performance

- Completeness - algorithm guarantees to return a solution if at least any solution exists for any random input
- Optimally - algorithm guarantees the best solution (lowest path cost) among all other solutions
- Time complexity - measures time for an algorithm to complete its task
- Space complexity - measures the maximum storage space required at any point during the search

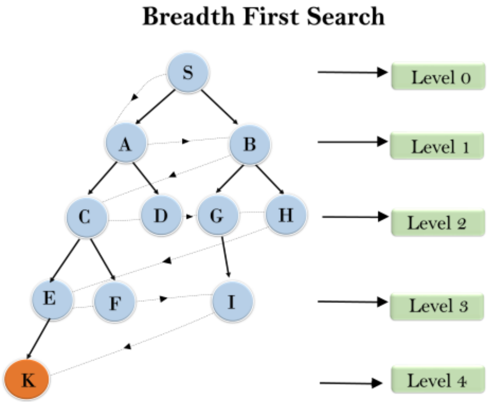
Types of search algorithms



Breadth-first search

- Starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level
- An example of a general-graph search algorithm; FIFO queue
- Advantages: provides a solution if exists; in the case with more than one solution provides the one with the least number of steps
- Disadvantages: requires a lot of memory; needs a lot of time if the goal is far away from the initial state

Breadth-first search

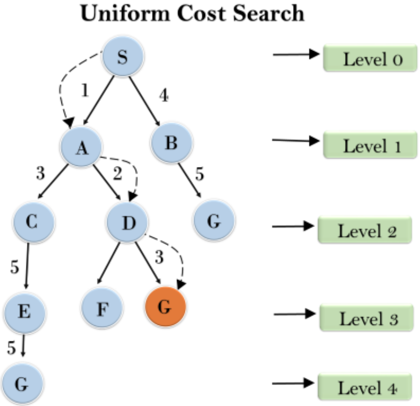


Time and space complexity $O(b) = O(b^d)$; complete if solution exists; optimal for non decreasing cost function

Uniform-cost search algorithm

- Used for weighted tree or graph; different costs for each edge
- Find path to goal state with lowest cumulative cost; priority queue
- Advantages: optimal because at every state the path with the least cost is chosen
- Disadvantages: does not care about the number of steps involve in searching and only concerned about path cost; it can be stacked in the infinite loop

Uniform-cost search algorithm

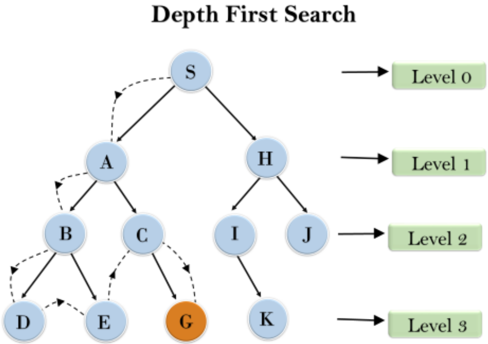


Time and space complexity: $O(n) = O(b^{1+[\frac{C^*}{\epsilon}]})$; complete; optimal - always chooses path of the least cost

Depth-first search

- Starts from the root node and follows each path to its greatest depth node before moving to the next path
- LIFO queue; similar to breadth-first search (BFS)
- Advantages: less memory than BSF; less time to reach the goal node
- Disadvantages: the solution can re-occur, no guarantee for the solution; it can go to infinite loop

Depth-first search

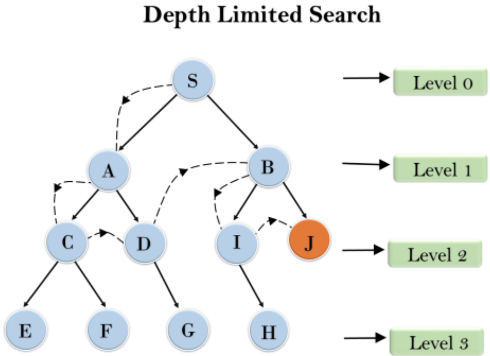


Time complexity: $O(n) = O(n^m)$; m - maximal depth of any node; for finite state space it is complete; non optimal; space complexity $O(b) = O(bm)$

Depth-limited search algorithm

- There is a set depth limit; even if there is a successor node after the depth limit it will disregard it
- Two types of failure: standard failure value and cutoff failure value
- Advantages: memory efficient
- Disadvantages: incompleteness; not optimal for problems with more than one solution

Depth-limited search algorithm

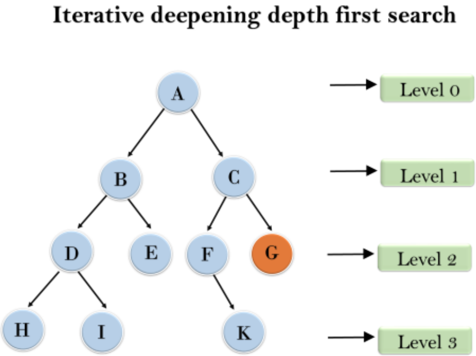


Time complexity: $O(b) = O(b^l)$; Space complexity: $O(b) = O(b * l)$; complete if solution is above depth limit; optimal

Iterative deepening depth-first search

- A combination of DFS and BFS; finds out the best depth limit and does it by gradually increasing the limit until a goal is found
- Performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found
- Advantages: fast search and memory efficient
- Disadvantages: it repeats all the work of the previous phase

Iterative deepening depth-first search

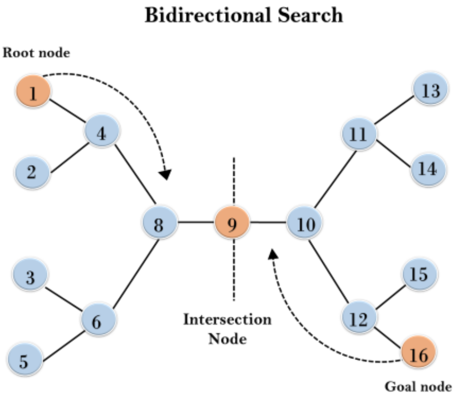


Time complexity: $O(b) = O(b^d)$; Space complexity: $O(b) = O(b * d)$; complete if the branching factor is complete; optimal for non decreasing cost function

Bidirectional search algorithm

- Two simultaneous searches: forward-search from initial state and backward-search from the goal state
- Two smaller graphs; the search stops when they intersect
- It can use all described techniques
- Advantages: fast and memory efficient
- Disadvantages: complex implementation; we need to know the goal state in advance

Bidirectional search algorithm



Time and space complexity: $O(b) = O(b^d)$; complete if we use BFS for both searches; optimal

Informed search algorithms

- Informed search algorithm contains an array of knowledge such as how far we are from the goal, path cost, how to reach to goal node, etc.
- It helps agent to explore less to the search space and find more efficiently the goal node
- More suitable for large spaces; uses heuristics thus it is called heuristic search

Heuristic function

- Heuristic function is used in informed search to find the most promising path
- It produces estimation of how close is the agent from the goal based on its current state
- Heuristic method does not necessarily gives the best solution, but it guarantees to find good solution in the reasonable time
- $h(n)$ - it calculates the cost of an optimal path between the pair of states; it is always positive
- Admissibility of the heuristic function is given as $h(n) \leq h^*(n)$

Pure heuristic search

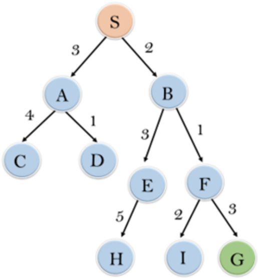
- The simplest form of heuristic search algorithms; expands nodes based on their heuristic value $h(n)$; two lists: OPEN and CLOSED list
- On each iteration, each node n with the lowest heuristic value is expanded and generates all its successors and n is placed to the closed list
- Algorithm stops when we reach the goal state
- Two types of algorithms: Greedy search algorithm and A* search algorithm

Greedy search algorithm

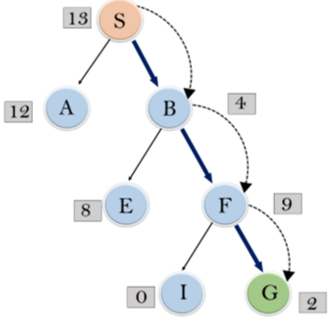
Always selects the path which appears best at that moment

- Step 1: place the starting node into the OPEN list
- Step 2: if the OPEN list is empty, Stop and return failure
- Step 3: remove the node n, from the OPEN list which has the lowest value of h(n), and places it in the CLOSED list
- Step 4: expand the node n, and generate the successors of node n
- Step 5: check each successor of node n, and find whether any node is a goal node or not; if any successor node is goal node, then return success and terminate the search, else proceed to Step 6
- Step 6: for each successor node, algorithm checks for evaluation function f(n), and then check if the node has been in either OPEN or CLOSED list; if the node has not been in any of the lists, then adds it to the OPEN list
- Step 7: Return to Step 2

Greedy search algorithm - example



node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0



Time and space complexity: $O(bm)$, m - maximum depth of the search space; offten incomplete even in the finite space; not-optimal

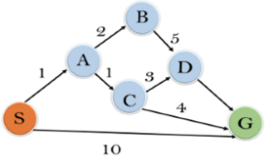
A* search algorithm

- Fitness number $f(n) = g(n) + h(n)$
- Advantages: the best algorithm than other search algorithms; optimal and complete; can solve very complex problems
- Disadvantages: does not always produce the shortest path as it mostly based on heuristics and approximation; has some complexity issues; memory demanding

A^* search algorithm

- Step1: place the starting node in the OPEN list
- Step 2: check if the OPEN list is empty or not, if the list is empty then return failure and stops
- Step 3: select the node from the OPEN list which has the smallest value of evaluation function $(g + h)$, if node n is goal node then return success and stop, otherwise
- Step 4: expand node n and generate all of its successors, and put n into the closed list; for each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list
- Step 5: else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value
- Step 6: Return to Step 2

A* search algorithm - example



State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0