The Document

João Morais

October 16, 2019

Contents

1 Mobile Communications: Cellular & Radio Access Networks

Currently, 70 to 2600 MHz is the used band mainly due to propagation characteristics - a notion that we'll reinforce is that the higher the frequency, the higher the attenuation - and due to the size of the antennas we can achieve. They are human scale, from 3m to 10cm.

- In the VHF/UHF band, propagation is characterised by being:
  - almost independent of polarisation and soil electromagnetic properties;
  - essentially done via direct and reflected rays;
  - influenced by the presence of obstacles;
  - almost insensitive to refraction by atmosphere;
  - basically limited by the radio-horizon;
  - basically independent of rain, gases and others.

2 Artificial Intelligence / Machine Learning

2.1 Artificial Intelligence

There are some goal states and one initial state. The objective is to find the goal state that is closer (with the shortest path to the root/ with the least search cost). Is given as the solution the steps necessary to perform that path.

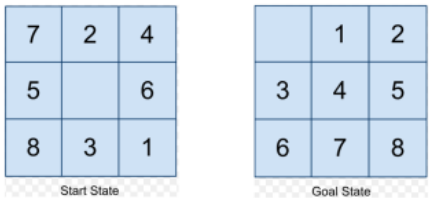
To keep the formulation as general as possible, abstractions are required, keeping in mind that they will need a correspondence when applied to a real problem.

Example of the application to a problem: a vacuum cleaner that needs to clean every square. In this case, only 2 squares are presented, no localization sensors are present, only "rubbish" sensors that tell if the current square is dirty or not.

Therefore:

- States : $< r, d_1, d_2 >$ where r is the robot position, d_1 and d_2 are binary, representing the existence of dirt in each of the rooms.
- Operators/Actions : L (go left), R (go right) or S (suck dirt)
- Goal Test : d_1 = False and d_2 = False. $((d_1 \text{ nor } d_2) == 1)$
- Initial State : $< r, d_1, d_2 >=< 1, T, T >$ is at square 1, and squares 1 and 2 are dirty
- Step Cost : the description of each action cost. In this case, 1 for each one.

8-puzzle



States: specify in a 3x3 matrix the location of the 8 numbered tiles plus the blank one

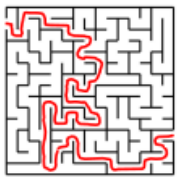
Initial state: any puzzle configuration

Operators: move "blank" to the left, right, up and down

Goal test: checks whether the state matches the goal configuration

Path cost: each step costs 1 (so path cost = number of steps)

Mazes



States: maze configuration plus current location

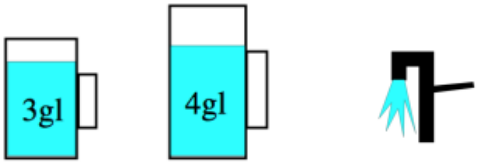
Initial state: any maze location

Operators: move to an adjacent and linked position

Goal test: current location = maze exit

Path cost: number of steps

Water jugs



States: amount of water in each jug (e.g., <1,4>)

Initial state: <0,0>

Operators: fill J4; fill J3;
empty J4; empty J3;
pour water from J4 into J3 until J3 is full or J4 is empty;
pour water from J3 into J4 until J4 is full or J3 is empty;

Goal test: <0,2>

Path cost: amount of water used

Hanoi towers

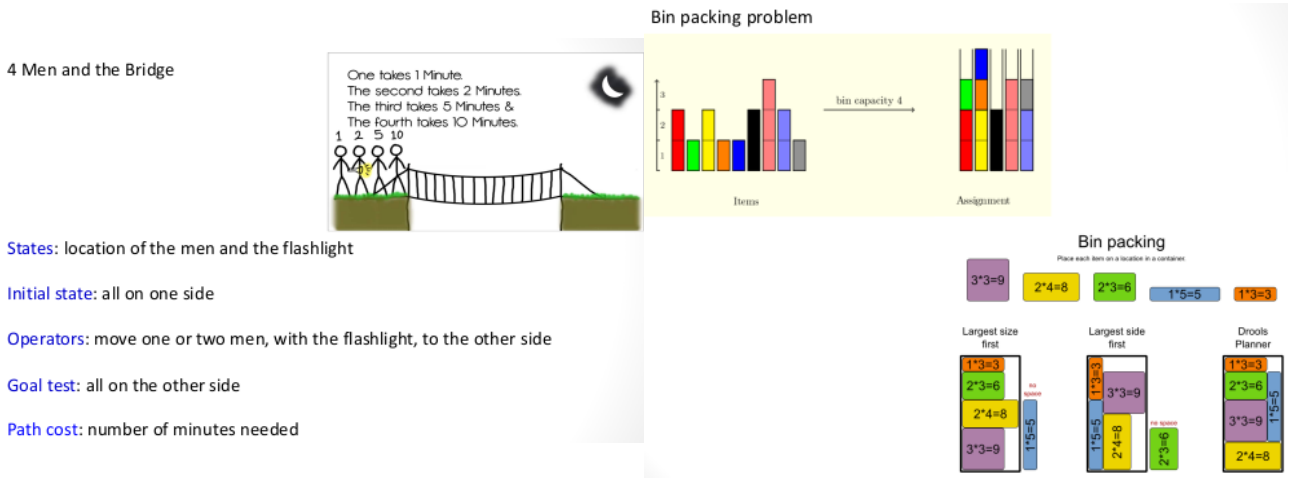
States: location of the disks in the three poles

Initial state: all disks correctly arrange in the left pole

Operators: move a free disk to another pole, which is empty or all disks in it are bigger

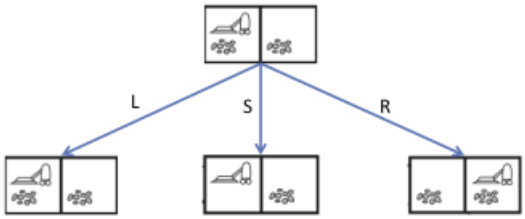
Goal test: all disks correctly arrange in the right pole

Path cost: number of disk moves

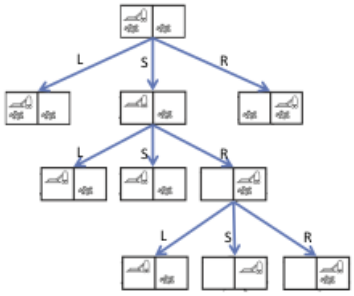


As you can see, every problem is very well defined in terms of the start, the end, the possible moves and what makes a solution better than other. Therefore, the conditions to put the computer thinking how to get from the possible steps to the best solution are assembled. One way the computer should be able to get to the solution is by exploring every move combination and check the state it ended up with.

Given the initial state, use the operators to generate successor states.



Then a choice of which is more "profitable" or more likely to lead to a goal state needs to be made and this cycle continues until the arrival at a goal state.



In this case, the solution would be $\{S, R, S\}$.

However, before diving directly in algorithms, is important to note the characteristics of the Environment and of the Agent. Based on these we'll be able to perform much better choosing the search algorithm that is best for us.

2.1.1 Environment

Observability: how much can the agent know about the environment.

An environment is fully observable if the agent can see everything. Or partial observable: - part of the state is occulted and you simply can't

Deterministic: the effect of the actions are predictable If I move one queen from one place to the other, I can predict the effects, what is attacking what, etc... Instead of Deterministic, it can be stochastic, where the outcomes are functions of probability functions. Therefore, you can't be 100% sure of the outcome. Therefore, you can use probability to make choices...

Neither the environment nor the agent performance change while the agent is deliberating. –¿ Static.

In a dynamic environment, the agent performance can change with time.

Semi-dynamic means that the world is static, but the performance is changing. ex: turn game where the game doesn't change while you are thinking but the more you take, the less point you get.

Continuous or discrete

Sequential or episodic. Episodic means that one episode doesn't influence the next one. It's very related with causality. In episodic environments, there's no influence in consequent problems.

E.g. one game of chess is sequential, but different games are episodic.

The outcomes for all actions are given. –¿ Known environment.

In a case where the less time you spend thinking before answering,

Knowing these is very important because it allows us to best choose the shelf of methods from which we take our algorithms from. Some are best from some things and some can only be used in certain situations as well.

Internally to the agent, you have a way of representing of the world. (Internal representation)

Example when you don't have an internal representation: random vacuum cleaning robots don't know anything about the environment, they just rotate randomly when they see an object.

- Fully observable vs. partially observable
  (partially observable because of noise, inaccurate or faulty sensors, or hidden parts)
- Single agent vs. multiagent (cooperative vs. competitive)
  (in multiagent environments, communication may be a key issue)
- Deterministic vs. stochastic
  (an environment is uncertain if it is not fully observable or not deterministic)
  (nondeterministic environment is when actions have different possible outcomes but no probabilities associated)
- Episodic vs. sequential
- Static vs. dynamic
  (if the environment doesn't change with the time but agent's performance does, it is **semidynamic**)
- Discrete vs. continuous (applies to states, time, percepts, and actions)
- Know vs. unknown
  (depends on the agent's knowledge about how the world evolves – the "laws of physics" of the environment)

	observable	deterministic	episodic	static	discrete
Chess	Y	Y	N	Y	Y
Poker	N	N	N	Y	Y
Taxi	N	N	N	N	N
Image analysis	Y	Y	Y	Semi	N

2.1.2 Agent

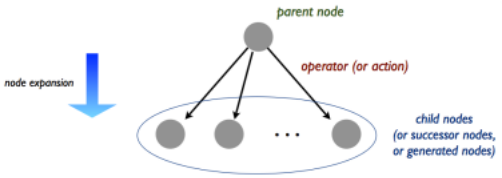
Model-based: typically, when you have partial observability.

Goal-based: when you aim for a goal. You have states and actions and can search for the goal.

Utility-based: are very similar to goal-based but are a bit more, not all goals are the same. There a preference between goals. Utility Theory handles how to translate preferences to numbers.

2.1.3 Search Problems

In essence, is necessary some **Search Terminology** to refer to certain things:



Successor function: given a node, returns the set of child nodes

Open list (or frontier or fringe): set of nodes not yet expanded

Closed (explored) list: set of nodes already expanded

Leaf node: a node without successors

State space can be:

- Tree-based – no repeated nodes in search
- Graph-based – directed cycle graph

General algorithm shape, starting with $open\_list = \{initial_node\}$, iterate over:

1. Select a node from the open_list;
2. Check if it's a goal node (in case it satisfies the goal test). If yes, return solution by backing up to the root;
3. If not, remote it from the open_list, expand it with the successor function and insert the nodes that come from there to the open_list.

Different selection criteria leads to a variety of search methods.

If it's tree based, then no nodes will be repeated and it's not necessary to have a list of the already visited nodes. In a graph however, is needed to have a list of these, or else a cycle is possible.

To evaluate the algorithm, many parameters may be enumerated:

Completeness: guarantee that a solution is found if there is one

Optimality: the solution found minimizes path cost over all possible solutions

Time complexity: how long does it take to find a solution (usually measured in terms of the number of nodes generated)

Space complexity: how much memory is needed to find a solution (usually measured in terms of the maximum number of nodes stored in memory)

Branching factor (b): maximum number of successors of any node

Depth (d) of the shallowest goal node

m = maximum length of any path in state space (may be infinite)

g(n): the cost of going from the root node to node n (path cost function)

Mentioning types of search strategies:

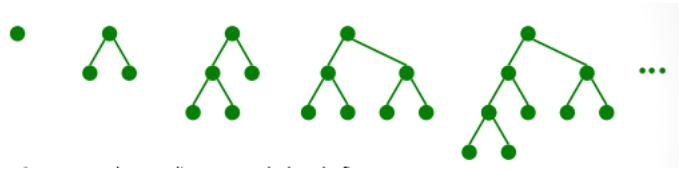
Uninformed (or blind) search: does not use additional (domain-dependent) information about states beyond that provided in the problem definition

Informed (or heuristic) search: uses problem-specific knowledge about the domain to "guide" the search towards more promising paths

2.1.4 Uniformed Search Strategies

A list of **Uniformed search strategies** we'll have a deeper look to:

- **Breadth-first Search** - Select earliest expanded node first - uses a FIFO queue (First In, First out). This leads to opening every node at the safe depth first before moving the deeper nodes.



Because order of depth is followed and every node checked, this search strategy is Complete and Optimal (if the path cost increases - or at least doesn't decreases - with depth). Being **d** the depth of the solution and **b** the branching factor(max number of successors of a node.) then in the worst case, the total number of nodes generated is: $1 + b + b^2 + b^3 + \cdots + b^d$

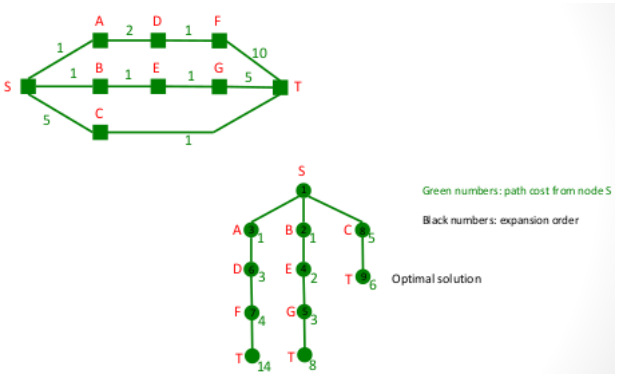
Time Complexity - O(b^d) Space Complexity - O(sum no of nodes) = O(b^d)

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

Note that it makes a difference if you test the node before or after expanding it. If it's tested before, there's no need of expanding the node. If the test is made only after the expansion, the complexities grows to O(b^{d+1}).

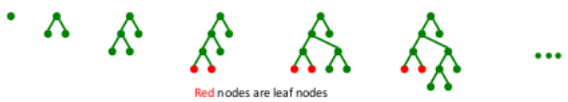
- **Uniform-cost Search** - expands the node that has the smallest cost from the root to it.



Note that each action generates one possible state from the state where the robot was previously.

This search strategy is only complete and optimal if the step costs are strictly positive. Else, it can give several steps that cost nothing to places far away from the solution.

- **Depth-first Search** - Exactly as the name suggests, goes until the deepest node first, and only then looks at the other nodes at the first level. Therefore, it can be very inefficient on a large or infinite tree. Open the last node added to the list (LIFO- Last In, First Out).



- **Backtrack Search** - a variation of depth-first, but expands one node at a time, only stores in memory that only node and the expansion is made by modifying the node, while backtrack is nothing more than undoing the modification..

  It's not complete nor it is optimal... but saves a lot of memory.

- **Depth-limited search** - another variation of depth-first where limiting the search tree to a depth L, is possible to contain the inefficiency. It's complete if the depth of the solution if smaller than L but still not optimal.

  Time complexity: O(b^L) Space complexity: O($b \times L$)

- **Iterative deepening depth-first search** - A variation of the previous one. In this one, the idea will be to run depth-limited search for an increasing L. Run for L=1, L=2, ...

This way, it is complete and it's optimal (if the path cost is a non-decreasing function of depth).

Time complexity: $O(b^d)$ Space complexity: $O(b \times d)$

Node expansion:

Breadth-first

$$1+b+b^2+b^3+\cdots+b^{d-1}+b^d$$

Iterative deepening

$$(d+1)1+(d)b+(d-1)b^2+(d-2)b^3+\cdots+(2)b^{d-1}+(1)b^d$$

Example: b = 10 and d = 5

breadth-first = 111 111

iterative deepening = 123 456 (+ 11%)

- **Bidirectional Search** - Search both from initial node and from the gold node. Note however that can only be used when a goal node is known and when the parent nodes can be computed given its child (through the sets of available actions). It's complete (if breath-first in both directions) and Optimal, if the step costs are equal.

  Time and space complexity: $O(b^{d/2})$

Example: $b = 10$ and $d = 5$

$b^d = 1\,111\,111$

$2\,b^{d/2} = 2\,222$

A summary of the above analysis:

Criterion	Breadth-first	Uniform-cost	Depth-first	Depth-limited	Iterative deepening	bidirectional
Complete?	✓	✓[1]	×	×	✓	✓[3]
Optimal?	✓[2]	✓	×	×	✓[2]	✓[2,3]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(b^m)$	$O(b^L)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(b.m)$	$O(b.L)$	$O(b.d)$	$O(b^{d/2})$

[1] for strictly positive step costs
[2] for path costs a non-decreasing function of depth
[3] for breadth-first in both directions

A **General Search Algorithm** can be formulated in the following way:

function General-search (***problem***, ***strategy***) **returns** a solution or failure
insert the root node into the open list
(the root node contains the initial state of problem)
loop do
if there are no candidate nodes for expansion **then return** failure
choose a node for expansion according to strategy (using strategy function)
if the node contains a goal state (using goal checking function) **then**
return the corresponding solution
else
for each operator in the list of operators (or successor function)
create a child node (for the new child state)
update child node path cost (using g-function)
add the resulting node to the open list [*unless… see graph search versions*]
end

domain (problem) independent

problem argument should include at least:

- initial state (using a specific state representation)
- successor function: new state = succ (current state, operator)
- path cost function (g-function)

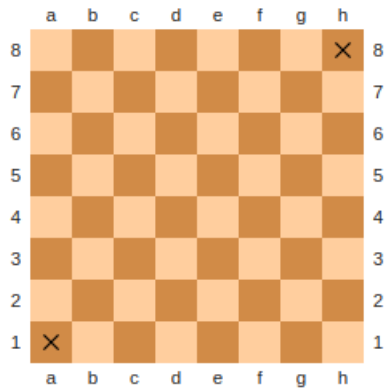
domain (problem) dependent

strategy argument (e.g., FIFO, LIFO, priority queue (by path cost), etc.)

algorithm selection

Finally, there are problem dependent and problem independent details. The search strategies are problem independent, but how we define the actions, states, ect… is problem dependent. And the way we see and think about the problem can be highly related to the way we represent it.

An incredibly well formulated example is the Mutilated Chess Board problem. If we take just the squares of the corners of a chess board, can we still fill the whole board with dominos (each domino takes 2 squares). It becomes slightly harder! If you keep removing squares it gets increasingly harder to figure out by head.



However, if you represent the chess board as the remaining black and white pieces and notice that a domino piece must always cover one black and one white squares, then by taking those two corners then 30 black and 32 white squares will be remaining making it impossible to fill with dominos. As a matter of fact, accordingly with Gomory's Theorem is also possible to say that if 2 square of opposite colours are removed then is always possible to fill the board with dominos! More in: Mutilated chessboard problem

The bottom line is that the representation (problem dependent details) matters.

With this is mind, consider the following example:

Initial state with: bedroom(3), living room(2), kitchen(3), hall(2), truck(0)

a) State: $< pos, n_{bedroom}, n_{living}, n_{kitchen}, n_{hall}, n_{truck} >$

b) Initial State: $< truck, 3, 2, 3, 2, 0 >$

c) m for move, p for push $< m_N, m_S, m_E, m_W, p_N, p_W, p_E, p_W >$. However, is only possible to push in directions where there are rooms (as for walking) and when there are boxes in the current room we are located in.

d) A goal condition could be $n_{truck} = 10$. Another could be the sum of all rooms to be 0. But the first one is more elegant and also seems more general... We don't want to throw boxes out of the window.

Sometimes, just finding the solution is enough. Some times, there's a best solution.

The heuristic (the only difference between informed and uninformed search) is a function of a state that gives us the appreciation we have for that state - how good that state is. If we want the best solution, we must have the heuristic function is key. If you just want a solution, that function can be much simpler or even nonexistent.

2.2 Supervised Learning

Extrema Conditions and Hessian Matrix

Videos: 87 - Warm up to the second partial derivative test to... 89 - Second partial derivative test intuition

This sub (...) sub section aims to explain why the conditions imposed on the Hessian matrix - the second derivative matrix - correspond to imposing in 2D what we know already.

In practice, in 2D, to find a extrema we need to guarantee that the first derivative is zero, which in N dimensions is correspondent to guaranteeing that the gradient is zero in that point, because the gradient is nothing more than all partial derivatives in that point. Therefore, setting $\nabla f = 0$ means that, in that point, the function is not increasing or decreasing in any of the N directions. So the first derivative makes sense.

However, when we go to the second derivative, the meanings get a bit more complicated.

In 2D, if the second derivative was 0, it was probably (not certainly) a saddle point, if it was > 0 or < 0 it was, respectively, a local minimum or maximum.

The conditions we should impose in 3D is to have a positive (for finding a minimum) or negative (for finding a maximum) definite Hessian Matrix. While second derivatives in order to just one variable is possible to attribute a sense to it, why do the cross derivatives play a role as well? And, why do they mean really?

Well, first the explanation on why it is needed: there are functions that across multiple dimensions still show that it is an extrema but then there's an inflexion along directions that are not along the axis. So, checking the axis is not enough. Why checking the cross partial derivatives makes it enough?

The Second Derivative Test

$$f_{xx}(x_o, y_o) f_{yy}(x_o, y_o) - f_{xy}(x_o, y_o)^2 \gtrless 0$$

If it is greater than 0, we have a maximum or a minimum and have to check the value of f_{xx} or f_{yy} to be sure. If it is less than 0, we have a saddle point. If it equals 0, then we don't know if it is a saddle point, but it is not a min or max therefore, at least for now, we certainly don't care.

Cross or Mixed partial derivatives can be switched? Yes if the function is C^2. (Boring to prove theorem called: Schwarz' Theorem)

Therefore, we just need to compute one of the cross derivatives.

Also this works because the second derivative test is nothing more than the determinant of the Hessian matrix. The determinant is the product of every eigenvalue of that matrix, therefore it can only be positive if they are both positive or both negative, in which cases there is, respectively, a minimum or a maximum.

But why do eigenvalues tell us this? Because they tell us how the eigenvectors are scaled! And the eigenvectors of such matrix will be the greatest and the least curvatures. Therefore, they either have the same signal / are scaled the same amount, or

Some other links that helped with this:

- Differential Geometry
- Criterior for critical points - Maximum, Minimum or Saddle?
- David Butler - Facts about Eigenvalues

2.2.1 Neural Networks - BackPropagation

Two of the most useful websites to check while trying to demonstrate the backpropagation algorithm:

- all backpropagation derivatives
- Error (deltas) derivation for backpropagation in neural networks
- 3Blue1Brown Neural Networks - Specially the last one, on backpropagation.

Why kernels are important? They facilitate a lot the mapping of features to higher dimensions!

Why kernels?

2.3 Unsupervised Learning

2.3.1 Lagrange Multipliers

Perfect Explanation why it works:

Quora - Intuition on Langrange Multipliers

And a video showing exactly how it's done:

Khan Academy - Lagrange Multipliers Example

2.4 Reinforcement Learning and Decision Making

An agent to make the wisest decision in its situation has to have knowledge on what state he is in, how the environment will evolve, how it will be like if he performs a certain action (taking into account stochastic environments) and a utility function to know how much that state contributes to its happiness.