T. Y. B. SC. IT

SEMESTER - V (CBCS)

ARTIFICIAL INTELLIGENCE

SUBJECT CODE : USIT504

Programme Co-ordinator : **Prof. Mandar Bhanushe**
Head, Faculty of Science & Technology,
IDOL, University of Mumbai - 400 098.

Course Co-ordinator : **Gouri S. Sawant**
Assistant Professor B.Sc.IT, IDOL,
University of Mumbai- 400098.

Editor : **Dr. Rajeshri Shinkar**
SIES (NERUL) college of Arts, Science & Commerce.

Course Writers : **Mr. Rupesh Ganesh Bhoir**
Infosys Ltd.

: **Dr. Juita Tushar Raut**
Assistant Professor, Sonopant Dandekar College,
Palghar(W) Pin Code. 401 404.

: **Ms. Vijaya Yogesh Rane**
Assistant Professor, AVM's Karmaveer Bhaurao Patil Degree College, Thane.

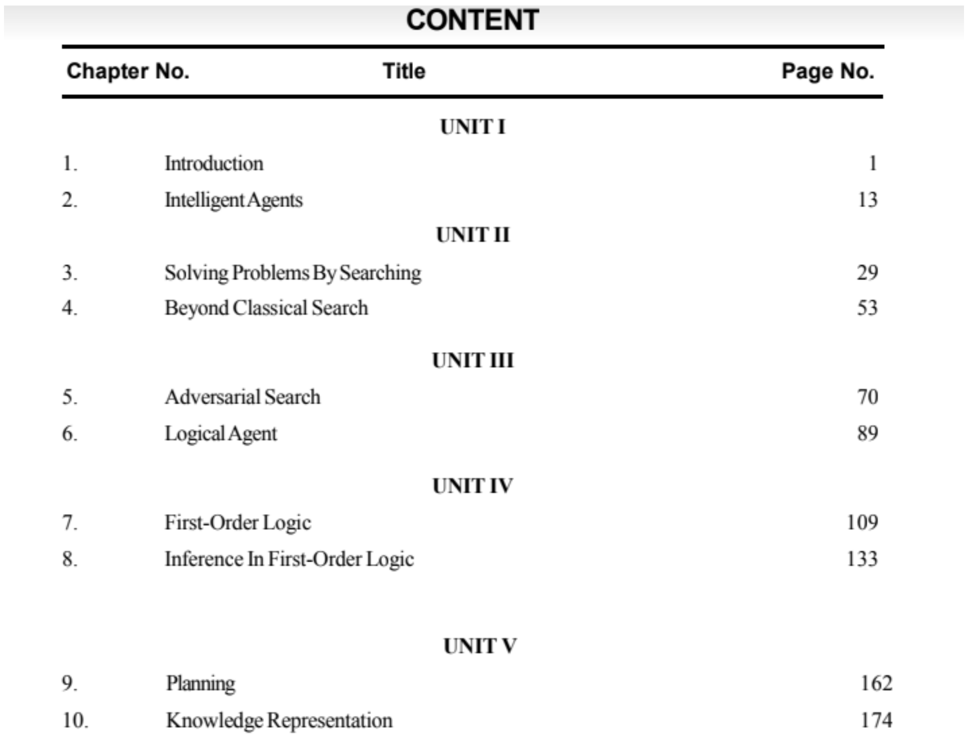
: **Ms. Anam Ansari**
Assistant Professor, B.N.N. College, Bhiwandi.

: **Ms. Jayalalita B Iyer**
Lecturer, N.E.S Ratnam College of Arts, science and commerce.

May 2022, Print I

Published by
Director
Institute of Distance and Open learning ,
University of Mumbai,Vidyanagari, Mumbai - 400 098.

DTP COMPOSED AND PRINTED BY
Mumbai University Press
Vidyanagari, Santacruz (E), Mumbai - 400098.



CONTENT

UNIT II

4

BEYOND CLASSICAL SEARCH

Unit Structure

4.2.1 Objective
4.2.2 Introduction
4.2.3 Local search algorithms
4.2.4 searching with non-deterministic action
4.2.5 searching with partial observations
4.2.6 online search agents and unknown environments
4.2.7 Summary

4.2.1 OBJECTIVES

After this chapter, you should be able to:

- Understand the local search algorithms and optimization problems.
- Understand searching techniques of non-deterministic action.
- Familiar with partial observation.
- Understand online search agents and unknown environments.

4.2.2 INTRODUCTION

This chapter covers algorithms that perform purely local search in the state space, evaluating and modifying one or more current states rather than systematically exploring paths from an initial state. These algorithms are suitable for problems in which all that matters is the solution state, not the path cost to reach it. The family of local search algorithms includes methods inspired by statistical physics (simulated annealing) and evolutionary biology (genetic algorithms).

The search algorithms we have seen so far, more often concentrate on path through which the goal is reached. But the problem does not demand the path of the solution and it expects only the final configuration of the solution then we have different types of problem to solve.

Local search algorithm operates using single path instead of multiple paths. They generally move to neighbours of the current state. Local algorithms use very little and constant amount of memory. Such kind of algorithms have ability to figure out reasonable solution for infinite state spaces.

They are useful for solving pure optimization problems. For search where path does not matter, Local search algorithms can be used, which operates by using a single current state and generally move only to neighbours of the state.

4.2.3 LOCAL SEARCH ALGORITHMS

- Algorithms that perform purely local search in the state space, evaluating and modifying one or more current states rather than systematically exploring paths from an initial state.
- These algorithms are suitable for problems in which all that matters is the solution state, not the path cost to reach it.
- The family of local search algorithms includes methods inspired by statistical physics **(simulated annealing)** and evolutionary biology **(genetic algorithms).**

- This systematicity is achieved by keeping one or more paths in memory and by recording which alternatives have been explored at each point along the path.
- When a goal is found, the path to that goal also constitutes a solution to the problem.
- In many problems, however, the path to the goal is irrelevant.
- Local search algorithms operate using CURRENT NODE a single current node (rather than multiple paths) and generally move only to neighbors of that node.
- Local Search Algorithm use very little memory—usually a constant amount; and they can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.
- It is also useful for solving pure optimization problems, in which the aim is to find the best state according to an objective function.
- **Hill Climbing** is a technique to solve certain optimization problems.
- In these techniques, we start with a sub-optimal solution and the solution is improved repeatedly until some condition is maximized.
- The idea of starting with a sub-optimal solution is compared to walking up the hill, and finally maximizing some condition is compared to reaching the top of the hill.

Algorithm:

1) Evaluate the initial state. If it is goal state then return and quit.
2) Loop until a solution is found or there are no new operators left.
3) Select & apply new operator.
4) Evaluate new state

If it is goal state, then quit.

If it is better than current state then makes it new current state.

If it is not better than current then go to step 2.

4.2.3.1 Hill Climbing search:

It is so called because of the way the nodes are selected for expansion. At each point search path, successor node that appears to lead most quickly to the top of the hill is selected for exploration. It terminates when it reaches a “peak” where no neighbour has a higher value. This algorithm doesn’t maintain a search tree so the data structure for the current node need only record the state and the value of the objective function. Hill Climbing Algorithm is sometimes called as a greedy local search because it grabs a good neighbour state without thinking ahead about where to go next. Unfortunately, hill climbing suffers from following problems:

This is shown in Figure 4.2.1

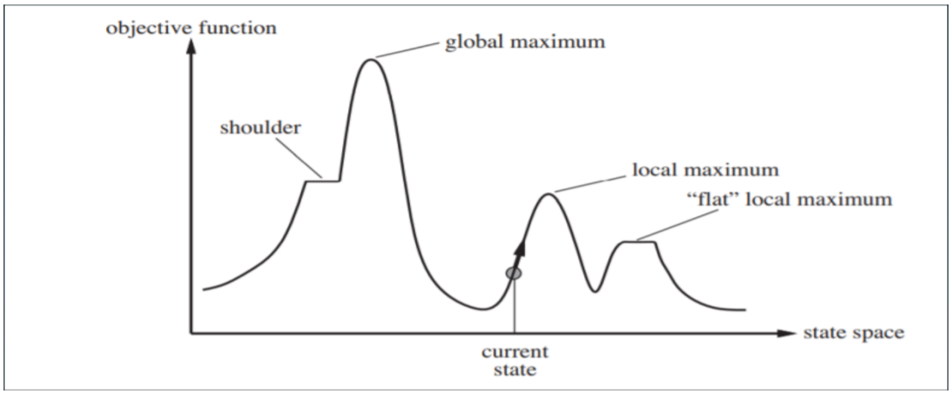


Fig. 4.2.1 Hill Climbing

- **Local maxima:**

Local maximum is a state which is better than its neighbour states, but there is also another state which is higher than it. They are also called as foot-hills. It is shown in Fig. 4.2.2

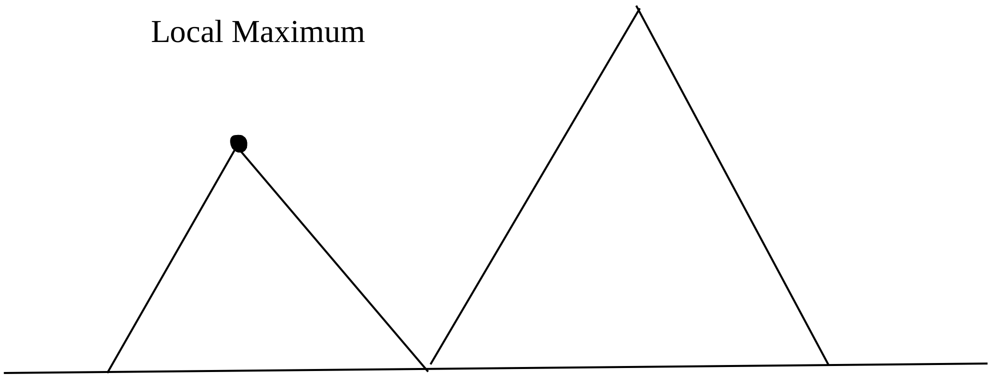


Fig. 4.2.2 Local maximum or Foot Hill

Solution to this problem:

Backtracking technique can be a solution of the local maximum in state space landscape. Create a list of the promising path so that the algorithm can backtrack the search space and explore other paths as well.

- **Plateau:**

It is a flat area of the search space in which a whole set of neighbouring states (nodes) have the same heuristic value. A plateau is shown in Fig. 4.2.3.

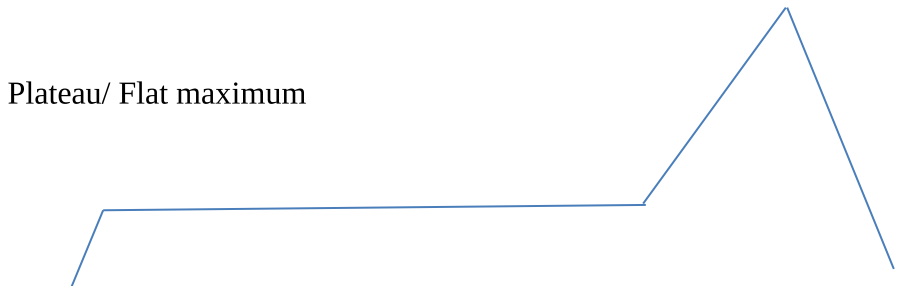


Fig 4.2.3 Plateau

Solution to this problem:

The solution for the plateau is to take big steps or very little steps while searching, to solve the problem. Randomly select a state which is far away from the current state so it is possible that the algorithm could find non-plateau region.

Another solution is to apply small steps several times in the same direction.

- **Ridge:**

It's an area which is higher than surrounding states but it cannot be reached in a single move. It is an area of the search space which is higher than the surrounding areas and that itself has a slope. Ridge is shown in Fig. 4.2.4.

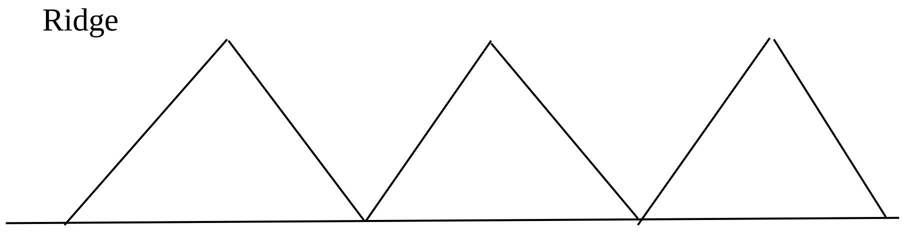


Fig. 4.2.4 Ridge

Solution to this problem:

Trying different paths at the same time is a solution. With the use of bidirectional search, or by moving in different directions, we can improve this problem.

Advantages of Hill Climbing:

It can be used in continuous as well as discrete domains.

Disadvantages of Hill Climbing:

It is not an efficient method.

It is not suitable for those problems whose value of heuristic function drops off suddenly when solution may be in sight.

It is local method as it looks at the immediate solution and decides about the next step to be taken rather than exploring all consequences before taking a move.

4.2.3.2 Simulated Annealing:

A hill-climbing algorithm which never makes a move towards a lower value guaranteed to be incomplete because it can get stuck on a local maximum. And if algorithm applies a random walk, by moving a successor, then it may complete but not efficient. Simulated Annealing is an algorithm which yields both efficiency and completeness. In mechanical term Annealing is a process of hardening a metal or glass to a high temperature then cooling gradually, so this allows the metal to reach a low-energy crystalline state.

The same process is used in simulated annealing in which the algorithm picks a random move, instead of picking the best move. If the random move improves the state, then it follows the same path. Otherwise, the algorithm follows the path which has a probability of less than 1 or it moves downhill and chooses another path.

Algorithm:

1. Evaluate the initial state. If it is also goal state, then return it and quit. Otherwise, continue with the initial state as the current state.
2. Initialize BEST-SO-FAR to the current state.
3. Initialize T according to the annealing schedule.
4. Loop until a solution is found or until there are no new operators left to be applied in the current state.
   a) Select an operator that has not yet been applied to the current state and apply it to produce a new state.

   b) Evaluate the new state. Compute

   Δ E = (value of current)- (value of new state)

   - If the new state is a goal state, then return it and quit.
   - If it is not goal state but is better than the current state, then make it the current state. Also set BEST-SO-FAR to this new state.
   - If it is not better than current state, then make it the current state with probability p’ as defined above. This step is usually implemented by invoking a random number generator to produce a number in the range [0, 1]. If that number is less than p’ then the move is accepted. Otherwise, do nothing.

c) Revise T as necessary according to the annealing schedule.

5. Return BEST-SO-FAR, as the answer.

To implement this revised algorithm, it is necessary to select an annealing schedule, which has three components. The first is the initial value to be used for temperature. The second is the criteria that will be used to decide when the temperature of the system should be reduced. The third is a amount by which the temperature will be reduced each time it is changed. There may also be a fourth component of the schedule, namely, when to quit. Simulated annealing is often used to solve problems in which the number of moves from a given sate is very large. For such problems, it may not make sense to try all possible moves.

4.2.3.3 Local beam search:

In this algorithm, it holds k number of states at any given time. At the start, these states are generated randomly. The successors of these k states are computed with the help of objective function. If any of these successors is the maximum value of the objective function, then the algorithm stops. Otherwise, the (initial k states and k number of successors of the states = 2k) states are placed in a pool. The pool is then sorted numerically. The highest k states are selected as new initial states. This process continues until a maximum value is reached. function Beam Search (problem, k), returns a solution state.

```
Start with k randomly generated states

Loop
    Generate all successors of all k states
    If any of the states =solution, then return the state
    Else select the k best successors
End
```

4.2.3.4 Genetic algorithms:

Genetic Algorithms (GAs) are adaptive heuristic search algorithms that belong to the larger part of evolutionary algorithms. Genetic algorithms are based on the ideas of natural selection and genetics. These are intelligent exploitation of random search provided with historical data to direct the search into the region of better performance in solution space. They are commonly used to generate high-quality solutions for optimization problems and search problems.

Genetic algorithms simulate the process of natural selection which means those species who can adapt to changes in their environment are able to survive and reproduce and go to next generation. In simple words, they simulate “survival of the fittest” among individual of consecutive generation for solving a problem. Each generation consist of a population of individuals and each individual represents a point in search space and possible solution. Each individual is represented as a string of character/integer/float/bits. This string is analogous to the Chromosome.

Steps of genetic algorithms:

Initial Population:

The process begins with a set of individuals which is called a Population. Each individual is a solution to the problem you want to solve. An individual is characterized by a set of parameters (variables) known as Genes. Genes are joined into a string to form a Chromosome (solution). In a genetic algorithm, the set of genes of an individual is represented using a string, in terms of an alphabet. Usually, binary values are used (string of 1s and 0s). We say that we encode the genes in a chromosome.

Fitness Function:

The fitness function determines how fit an individual is (the ability of an individual to compete with other individuals). It gives a fitness score to each individual. The probability that an individual will be selected for reproduction is based on its fitness score.

Selection:

The idea of selection phase is to select the fittest individuals and let them pass their genes to the next generation. Two pairs of individuals (parents) are selected based on their fitness scores. Individuals with high fitness have more chance to be selected for reproduction.

Crossover:

It is the most significant phase in a genetic algorithm. For each pair of parents to be mated, a crossover point is chosen at random from within the genes. Offspring are created by exchanging the genes of parents among themselves until the crossover point is reached.

Mutation:

In certain new offspring formed, some of their genes can be subjected to a mutation with a low random probability. This implies that some of the bits in the bit string can be flipped.

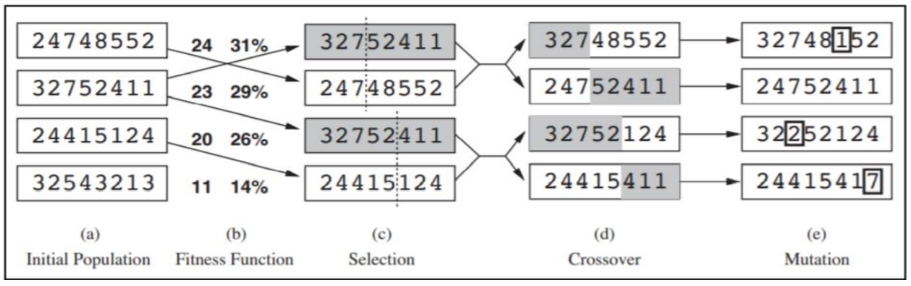


Fig. 4.2.5 The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e)

4.2.4 SEARCHING WITH NON-DETERMINISTIC ACTION

When the environment is either partially observable or nondeterministic (or both), precepts become useful. When the environment is nondeterministic, precepts tell the agent which of the possible outcomes of its actions has actually occurred. In both cases, the future precepts cannot be determined in advance and the agent's future actions will depend on those future precepts. So, the solution to a problem is not a sequence but a contingency plan (also known as a strategy) that specifies what to do depending on what precepts are received.

4.2.4.1 The erratic vacuum world:

There are three actions- Left, right and Suck. And the goal is to clean up all the dirt.

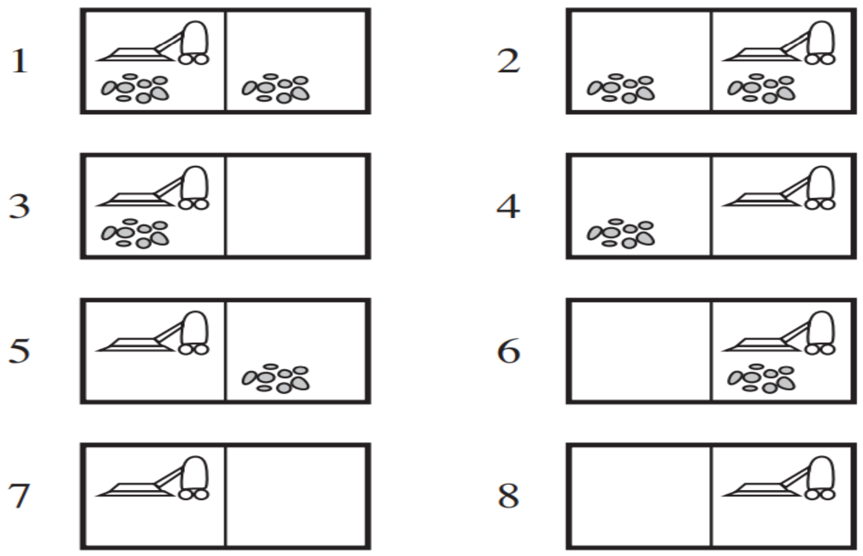


Fig. 4.2.6 The eight possible states of the vacuum world; states 7 and 8 are goal states.

In the erratic vacuum world, the Suck action works as follows:

- When applied to a dirty square the action cleans the square and sometimes cleans up dirt in an adjacent square, too.
- When applied to a clean square the action sometimes deposits dirt on the carpet.
- a contingency plan such as the following:

[Suck, if State = 5 then [Right, Suck] else [ ]]

If the environment is observable, deterministic, and completely known, then the problem is trivially solvable by any of the algorithms.

For example, if the initial state is 1, then the action sequence [Suck, Right, Suck] will reach a goal state, 8. Thus, solutions for nondeterministic problems can contain nested if–then–else statements;

4.2.4.2 AND- OR search trees:

A solution to a problem can be obtained by decomposing it into smaller sub-problems. Each of this sub-problem can then be solved to get its solution. These sub-solutions can then be recombined to get a solution as a whole. OR graph finds a single path.

AND arc may point to any number of successor nodes, all of which must be solved in order for an arc to point a solution. This is actually called as an AND-OR graph or AND/OR tree.

```
function AND-OR-GRAPH-SEARCH(problem) returns a conditional plan, or failure
  OR-SEARCH(problem.INITIAL-STATE, problem, [ ])
________________________________________________________________________
function OR-SEARCH(state, problem, path) returns a conditional plan, or failure
  if problem.GOAL-TEST(state) then return the empty plan
  if state is on path then return failure
  for each action in problem.ACTIONS(state) do
     plan ← AND-SEARCH(RESULTS(state, action), problem, [state | path])
     if plan ≠ failure then return [action | plan]
  return failure
________________________________________________________________________
function AND-SEARCH(states, problem, path) returns a conditional plan, or failure
  for each $s_i$ in states do
     $plan_i$ ← OR-SEARCH($s_i$, problem, path)
     if $plan_i$ = failure then return failure
  return [if $s_1$ then $plan_1$ else if $s_2$ then $plan_2$ else . . . if $s_{n-1}$ then $plan_{n-1}$ else $plan_n$]
```

Fig. 4.2.7 An algorithm for searching AND–OR graphs generated by nondeterministic environments.

Figure 4.2.8 gives a recursive, depth-first algorithm for AND–OR graph search. One key aspect of the algorithm is the way in which it deals with cycles, which often arise in nondeterministic problems (e.g., if an action sometimes has no effect or if an unintended effect can be corrected). If the current state is identical to a state on the path from the root, then it returns with failure. This doesn't mean that there is no solution from the current state; it simply means that if there is a noncyclic solution, it must be reachable from the earlier incarnation of the current state, so the new incarnation can be discarded. With this check, we ensure that the algorithm terminates in every finite state space, because every path must reach a goal, a dead end, or a repeated state.

Example.

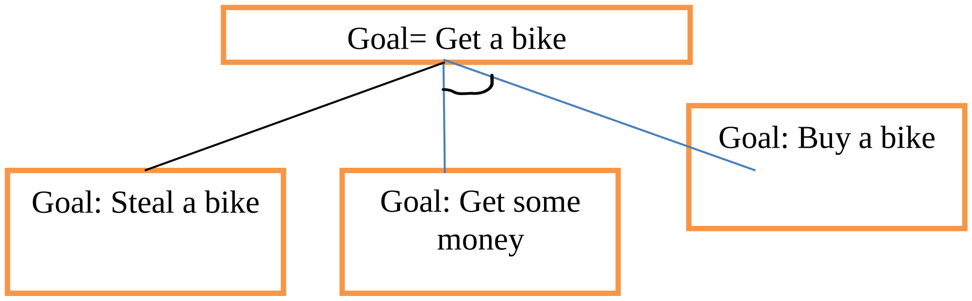


Fig. 4.2.8 AND/OR graph example

4.2.5 SEARCHING WITH PARTIAL OBSERVATIONS:

when an environment is partially observable, an agent can be in one of several possible states. An action leads to one of several possible outcomes.

To solve these problems, an agent maintains a belief state that represent the agent's current belief about the possible physical state it might be in, given the sequence of actions and percepts up to that point.

Agents percepts cannot pin down the exact state the agent is in

Let Agents have **Belief** states

- Search for a sequence of belief states that leads to a goal
- Search for a plan that leads to a goal

Sensor-less vacuum world

Assume belief states are the same but no location or dust sensors

Initial state = {1, 2, 3, 4, 5, 6, 7, 8}

Action: Right

Result = {2, 4, 6, 8}

Right, Suck

Result = {4, 8}

Right, Suck, Left, Suck

Result = {7} guaranteed!

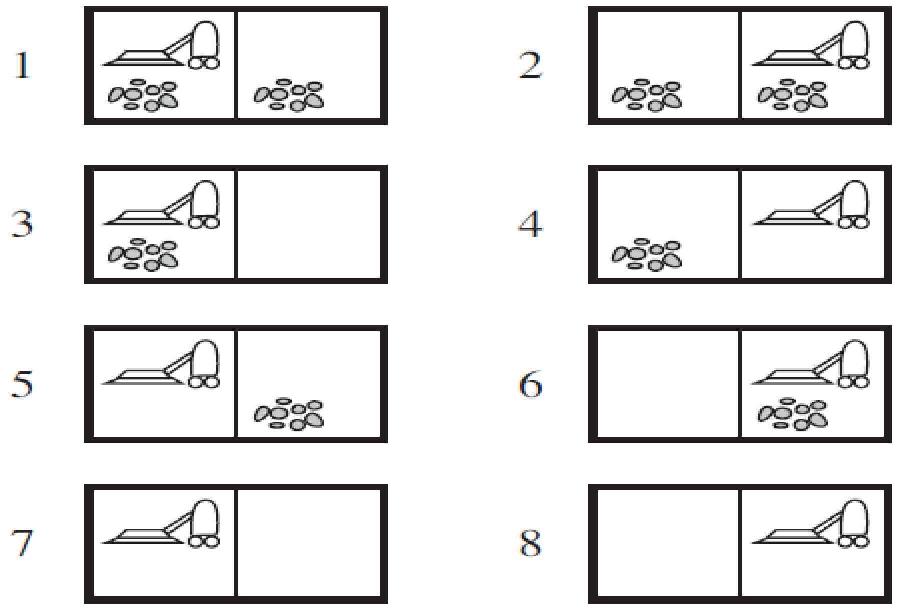


Fig. 4.2.9 Sensor less vacuum world

4.2.5.1 Searching with no observation:

It is instructive to see how the belief-state search problem is constructed. Suppose the underlying physical problem P is defined by ACTIONSP, RESULTP, GOAL-TESTP, and STEP-COSTP.

Then we can define the corresponding sensor less problem as follows:

- **Belief states:** The entire belief-state space contains every possible set of physical states. If P has N states, then the sensor less problem has up to 2N states, although many may be unreachable from the initial state.
- **Initial state:** Typically, the set of all states in P, although in some cases the agent will have more knowledge than this.
- **Actions:** This is slightly tricky. Suppose the agent is in belief state b = {s1, s2}, but ACTIONS P(s1) ≠ ACTIONS (s2); then the agent is unsure of which actions are legal.
- **Transition model:** The agent doesn't know which state in the belief state is the right one; so as far as it knows, it might get to any of the states resulting from applying the action to one of the physical states in the belief state.
- **Goal test:** The agent wants a plan that is sure to work, which means that a belief state satisfies the goal only if all the physical states in it satisfy GOAL-TEST P. The agent may accidentally achieve the goal earlier, but it won't know that it has done so.
- **Path cost:** This is also tricky. If the same action can have different costs in different states, then the cost of taking an action in a given belief state could be one of several values.

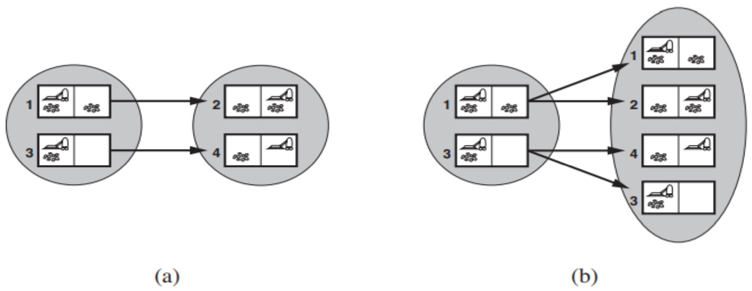


Fig. 4.2.10 (a) Predicting the next belief state for the sensorless vacuum world with a deterministic action, right. (b) Prediction for the same belief state and action in the slippery version of the sensorless vacuum world.

4.2.5.2 Searching with observations:

For a general partially observable problem, we have to specify how the environment generates percepts for the agent. For example, we might define the local-sensing vacuum world to be one in which the agent has a position sensor and a local dirt sensor but has no sensor capable of detecting dirt in other squares. The formal problem specification includes a PERCEPT(s) function that returns the percept received in a given state. (If sensing is nondeterministic, then we use a PERCEPTS function that returns a set of possible percepts.) For example, in the local-sensing vacuum world, the PERCEPT in state 1 is [A, Dirty]. Fully observable problems are a special case in which PERCEPT(s) = s for every state s, while sensorless problems are a special case in which PERCEPT(s) = null.

- The ACTIONS, STEP-COST, and GOAL-TEST are constructed from the underlying physical problem just as for sensorless problems, but the transition model is a bit more complicated.
- The prediction stage is the same as for sensorless problems: given the action a in belief state b, the predicted belief state is ˆb = PREDICT (b, a).
- The observation prediction stage determines the set of percepts o that could be observed in the predicted belief state: POSSIBLE-PERCEPTS (ˆb) = {o : o = PERCEPT(s) and s ∈ ˆb} .
- The update stage determines, for each possible percept, the belief state that would result from the percept. The new belief state bo is just the set of states in ˆb that could have produced the percept: bo = UPDATE (ˆb, o) = {s : o = PERCEPT(s) and s ∈ ˆb}.
- Putting these three stages together, we obtain the possible belief states resulting from a given action and the subsequent possible percepts: RESULTS (b, a) = {bo : bo = UPDATE(PREDICT(b, a), o) and o ∈ POSSIBLE-PERCEPTS(PREDICT(b, a))} .

Fig.4.2.10 Two example of transitions in local-sensing vacuum worlds. (a) In the deterministic world, Right is applied in the initial belief state, resulting in a new belief state with two possible physical states; for those states, the possible percepts are [B, Dirty] and [B, Clean], leading to two belief states, each of which is a singleton. (b) In the slippery world, Right is applied in the initial belief state, giving a new belief state with four physical states; for those states, the possible percepts are [A, Dirty], [B, Dirty], and [B, Clean], leading to three belief states as shown.

4.2.5.3 Solving partially observable problems:

The preceding section showed how to derive the RESULTS function for a nondeterministic belief-state problem from an underlying physical problem and the PERCEPT function.

To solve these problems, an agent maintains a belief state that represent the agent's current belief about the possible physical state it might be in, given the sequence of actions and percepts up to that point.

Agent’s percepts cannot pin down the exact state the agent is in

Let Agents have **Belief** states

- Search for a sequence of belief states that leads to a goal
- Search for a plan that leads to a goal

For such formulation, the AND-OR search algorithm can be applied directly to derive a solution. Figure 4.2.11 shows part of the search tree for the local-sensing vacuum world, assuming an initial percept [A, Dirty]. The solution is the conditional plan [Suck, Right, if B state = {6} then Suck else [ ]] .

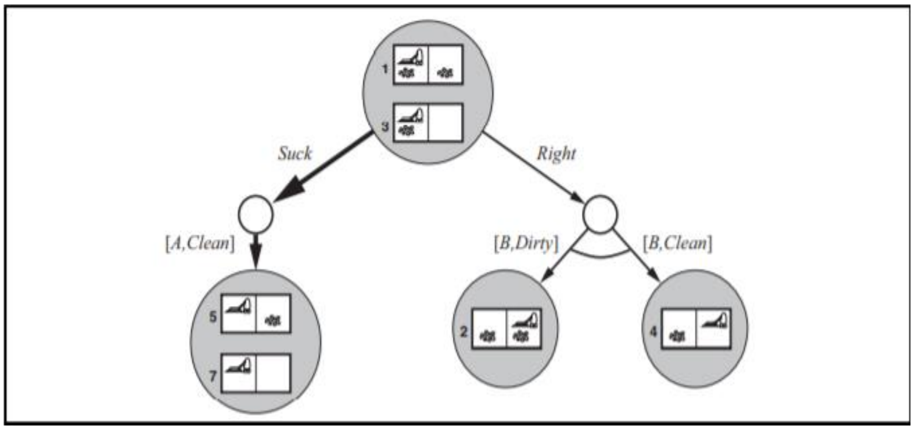


Fig. 4.2.11 AND -OR search tree

4.4.4 An agent for partially observable environments:

The design of a problem-solving agent for partially observable environments is quite similar to the simple problem-solving agent. the agent formulates a problem, calls a search algorithm (such as AND-OR-GRAPH-SEARCH) to solve it, and executes the solution. There are two main differences. First, the solution to a problem will be a conditional plan rather than a sequence; if the first step is an if–then–else expression, the agent will need to test the condition in the if-part and execute the then-part or the else-part accordingly. Second, the agent will need to maintain its belief state as it performs actions and receives percepts.

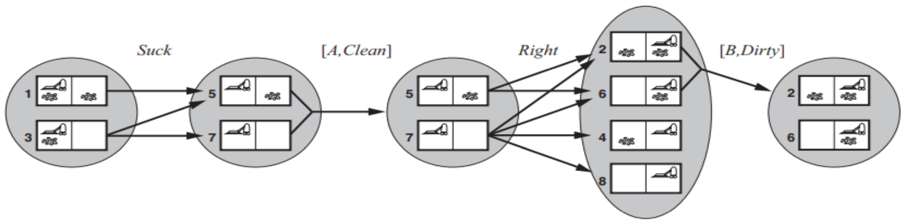


Fig. 4.2.12 Two prediction–update cycles of belief-state maintenance in the kindergarten vacuum world with local sensing.

4.2.6 ONLINE SEARCH AGENTS AND UNKNOWN ENVIRONMENTS

So far, we have concentrated on agents that use offline search algorithms. They compute a complete solution before setting foot in the real world and then execute the solution. In contrast, an online search agent interleaves computation and action: first it takes an action, then it observes the environment and computes the next action. Online search is a good idea in dynamic or semi dynamic domains—domains where there is a penalty for sitting around and computing too long.

Online search is also helpful in nondeterministic domains because it allows the agent to focus its computational efforts on the contingencies that actually arise rather than those that might happen but probably won't. Of course, there is a trade-off: the more an agent plans ahead, the less often it will find itself up the creek without a paddle.

Online search is a necessary idea for unknown environments, where the agent does not know what states exist or what its actions do. In this state of ignorance, the agent faces an exploration problem and must use its actions as experiments in order to learn enough to make deliberation worthwhile.

The canonical example of online search is a robot that is placed in a new building and must explore it to build a map that it can use for getting from A to B.

4.2.6.1 Online search problem:

An online search problem must be solved by an agent executing actions, rather than by pure computation. online search agents can build a map and find a goal if one exists.

We assume a deterministic and fully observable environment but we stipulate that the agent knows only the following:

1. ACTIONS(s)
2. The step-cost function
3. GOAL-TEST(s).

For example, in the maze problem shown in Figure, the agent does not know that going Up from (1,1) leads to (1,2); nor, having done that, does it know that going Down will take it back to (1,1).

- Necessary in unknown environments
- Robot localization in an unknown environment (no map)
- Does not know about obstacles, where the goal is, that UP from (1,1) goes to (1, 2)
- Once in (1, 2) does not know that down will go to (1, 1)
- Some knowledge might be available
- If location of goal is known, might use Manhattan distance heuristic
- Competitive Ratio = Cost of shortest path without exploration/Cost of actual agent path
- Irreversible actions can lead to dead ends and CR can become infinite.

- Hill- Climbing is already an online search algorithm but stops at local optimal.

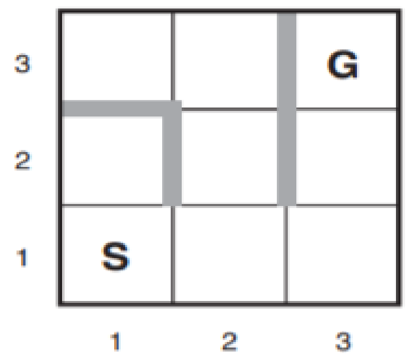


Fig. 4.2.13 A simple maze problem

4.2.6.2 Online search agents:

After each action, an online agent receives a percept telling it what state it has reached; from this information, it can augment its map of the environment. An online algorithm, on the other hand, can discover successors only for a node that it physically occupies.

To avoid traveling all the way across the tree to expand the next node, it seems better to expand nodes in a local order. Depth-first search has exactly this property because (except when backtracking) the next node expanded is a child of the previous node expanded.

An online depth-first search agent is shown in Figure 4.2.14

```
function ONLINE-DFS-AGENT($s'$) returns an action
  inputs: $s'$, a percept that identifies the current state
  persistent: $result$, a table indexed by state and action, initially empty
              $untried$, a table that lists, for each state, the actions not yet tried
              $unbacktracked$, a table that lists, for each state, the backtracks not yet tried
              $s$, $a$, the previous state and action, initially null

  if GOAL-TEST($s'$) then return $stop$
  if $s'$ is a new state (not in $untried$) then $untried[s'] \leftarrow$ ACTIONS($s'$)
  if $s$ is not null then
     $result[s, a] \leftarrow s'$
     add $s$ to the front of $unbacktracked[s']$
  if $untried[s']$ is empty then
     if $unbacktracked[s']$ is empty then return $stop$
     else $a \leftarrow$ an action $b$ such that $result[s', b]$ = POP($unbacktracked[s']$)
  else $a \leftarrow$ POP($untried[s']$)
  $s \leftarrow s'$
  return $a$
```

Fig. 4.2.14 An online search agent that uses depth-first exploration. The agent is applicable only in state spaces in which every action can be "undone" by some other action

It is fairly easy to see that the agent will, in the worst case, end up traversing every link in the state space exactly twice. For exploration, this is optimal; for finding a goal, on the other hand, the agent's competitive ratio could be arbitrarily bad if it goes off on a long excursion when there is a goal right next to the initial state.

Because of its method of backtracking, ONLINE-DFS-AGENT works only in state spaces where the actions are reversible. There are slightly more complex algorithms that work in general state spaces, but no such algorithm has a bounded competitive ratio.

4.2.6.3 Online local search:

Like depth-first search, hill-climbing search has the property of locality in its node expansions. In fact, because it keeps just one current state in memory, hill-climbing search is already an online search algorithm! Unfortunately, it is not very useful in its simplest form because it leaves the agent sitting at local maxima with nowhere to go. Moreover, random restarts cannot be used, because the agent cannot transport itself to a new state.

Instead of random restarts, one might consider using a random walk to explore the environment. A random walk simply selects at random one of the available actions from the current state; preference can be given to actions that have not yet been tried. It is easy to prove that a random walk will eventually find a goal or complete its exploration, provided that the space is finite. On the other hand, the process can be very slow.

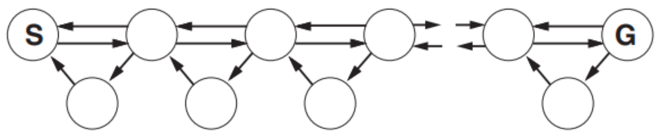


Fig. 4.2.15 An environment in which a random walk will take exponentially many steps to find the goal

4.2.7 SUMMARY

This chapter has examined search algorithms for problems beyond the “classical” case of finding the shortest path to a goal in an observable, deterministic, discrete environment. Local search methods such as hill climbing operate on complete-state formulations, keeping only a small number of nodes in memory. Several stochastic algorithms have been developed, including simulated annealing, which returns optimal solutions when given an appropriate cooling schedule.

A genetic algorithm is a stochastic hill-climbing search in which a large population of states is maintained. New states are generated by mutation and by crossover, which combines pairs of states from the population.

In nondeterministic environments, agents can apply AND–OR search to generate contingent plans that reach the goal regardless of which outcomes occur during execution.

When the environment is partially observable, the belief state represents the set of possible states that the agent might be in. Standard search algorithms can be applied directly to belief-state space to solve sensor less problems, and belief-state AND–OR search can solve general partially observable problems.

4.2.8 UNIT END QUESTIONS

1 What do you mean by local maxima with respect to search technique?

2 Explain Hill Climbing Algorithm.

3 Explain AO* Algorithm.

4 Explain a searching with nondeterministic action.

5 Explain searching with partial observations.

6 Write a note on simulated annealing.

7 Explain a local search algorithm.

8 Write a note on online search agents.

9 Write a note on unknown environments.

10 Consider the sensorless version of the erratic vacuum world. Draw the belief-state space reachable from the initial belief state {1, 2, 3, 4, 5, 6, 7, 8}, and explain why the problem is unsolvable.

4.2.9 BIBLIOGRAPHY

- Deva Rahul, Artificial Intelligence a Rational Approach, Shroff, 2014.
- Khemani Deepak, A First course in Artificial Intelligence, McGraw Hill, 2013.
- Chopra Rajiv, Artificial Intelligence a Practical Approach, S. Chand, 2012.
- Russell Stuart, and Peter Norvig Artificial Intelligence a Modern Approach, Pearson, 2010.
- Rich Elaine, et al. Artificial intelligence, Tata McGraw Hill, 2009.

*****