Lecture 24: Classical Methods of AI

17 April 2024 11:07

In the early times (60's and 70's) AI was primarily used for problem solving

Eg: Solving a puzzle

These problems usually looks like :

Given State $\xrightarrow{\text{sequence of decisions}}$ Desired State

Most of these problems are single agent centered.

Sequence of decisions
- Knowledge based
  - Memory based
  - rule based
- Search

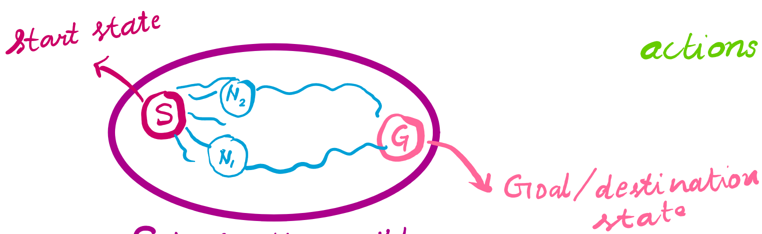
Knowledge: exploit additional knowledge from experiences.

Memory: experiences are stored so that next time a problem instance occurs, start from nearest state to solve

Rule: domain experts formulate a rule from experiences.

Search: The first approach by any intelligent agent, when there is no past experiences are available

State Space Search:



Set of all possible states

Set of states = $\{s_1, s_2, \ldots s_m\}$

actions(s) : actions at a state s

neighbours(s) = $\{ s' \mid \exists$ action 'a' which takes s to s'$\}$

isGoal(s) = bool value to check if the state is Goal

Ex: River crossing puzzle:

Farmer
Wolf } can't be kept together.
{ Goat
{ Cabbages

" F ... takes the goat to the

Ex: River crossing puzzle:

farmer
wolf } can't be kept together.
{Goat }
{ Cabbages

boat for conveyance

"Farmer takes the goat to the other side" is an action

The farmer has to take the wolf, goat, cabbage across the river. But WG and GC can't be together.

Initially: FWGC ||

After the above action: WC || FG

Similarly we can define other actions and states

We can create a tree of possible actions and seach on it to reach the goal

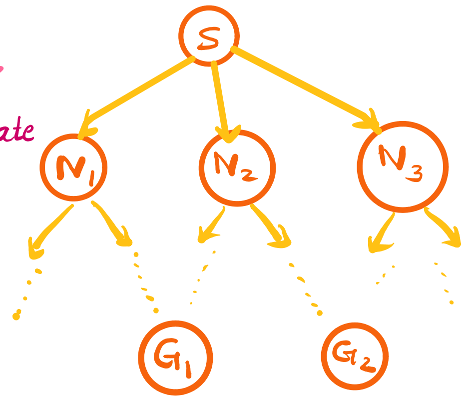
Any search method needs:

i) Abstraction: states, actions, neighbours, goal, cost.

ii) Algorithm development.

General Anatomy of a search algorithm

we will maintain a data structure for Visited states, we will have current state



For each iteration in our search algorithm,

For state c ∈ neighbours (N):
    if isGoal (c):
        do something

    else:
        do something else.

Search Algorithms

i) Uninformed search

ii) Informed search.

Bredth-first search

Finds the shortest path to the goal, but maynot be the least cost path.

PSEUDO-CODE FOR BFS

```
BFS(graph, start):
    create an empty queue
    enqueue start into the queue
    create an empty visited set
    add start to visited set
while the queue is not empty:
        dequeue a vertex, v, from the queue
        for each neighbor, u, of v:
            if u is not visited:
                enqueue u into the queue
                add u to visited set
```

From <https://www.codingdrills.com/tutorial/introduction-to-graph-algorithms/bfs-pseudocode>

Depth first Search:

Depth-First Search (DFS) is a fundamental graph algorithm that explores a graph by traversing as far as possible along each branch before backtracking. It is widely used in various applications like maze solving, topological sorting, and finding strongly connected components.

From <https://www.codingdrills.com/tutorial/introduction-to-graph-algorithms/dfs-pseudocode>

PSEUDO-CODE FOR DFS

```
DFS-A(G,s)
   for all v in V[G] do
      visited[v] := false
   end for
   S := EmptyStack
   Push(S,s)
   while not Empty(S) do
      u := Pop(S)
      if not visited[u] then
         visted[u] := true
         for all w in Adj[u] do
            if not visited[w] then
               Push(S,w)
            end if
         end for
      end if
   end while
```

DFS finds the goal quickly, but the path is neither shortest nor least cost.

Uniform cost Search:

Finds the least cost path to the goal.

The Uniform Cost Search Algorithm is a search algorithm to find the minimum cumulative cost of the path from the source node to the destination node. It is an uninformed algorithm i.e. it doesn't have prior information about the path or nodes and that is why it is a brute-force approach.

We use a boolean array visited and a priority queue to find the minimum cost. The node with minimum cost has the highest priority. It uses blind search as there is no prior information on the nodes.
The algorithm for the above algorithm is given as below:

- Create a priority queue, a boolean array visited of the size of the number of nodes, and a min_cost variable initialized with maximum value. Add the source node to the queue and mark it visited.
- Pop the element with the highest priority from the queue. If the removed node is the destination node, check the min_cost variable, if the value of the min_cost variable is greater than the current cost then update the variable.
- If the given node is not the destination node then add all the unvisited nodes to the priority queue adjacent to the current node.

Refer the following website for worked out example:
From <https://www.scaler.com/topics/uniform-cost-search/>

Informed Search

A* search:

Wikipedia page:
https://en.wikipedia.org/wiki/A*_search_algorithm

For each node we have an estimate of that state

A* score for a path = cost of path + estimate of end node.

Refer this for detailed worked out examples:
1. https://www.geeksforgeeks.org/a-search-algorithm/
2. https://www.codecademy.com/resources/docs/ai/search-algorithms/a-star-search

Estimat is smaller than actual, then always least cost path.

If the estimate is zero, same as Dijikstra's algorithm

Estimates were perfect, the least cost is quickly

If we over estimate, we get a suboptimal solution.