Artificial intelligence

state space search

criteria for comparing search algorithms:

1) <u>completeness</u> : does the algorithm always find a sol^n when there exists one?
2) <u>Time complexity</u> : runtime of algo before sol^n is found.
3) <u>Space complexity</u> : No. of nodes in the <u>OPEN List</u> is seen.
4) <u>Quality</u> of sol^n :

<u>Comparison of BFS and DFS</u> :

(I) <u>COMPLETENESS</u>

for finite state space : BFS, DFS both are complete.

for infinite state space : BFS complete, DFS incomplete.

(II) <u>TIME COMPLEXITY</u> ('b' is branching factor, 'd' is depth/level at which goal node is at)

BFS

If goal is present at leftmost node,

nodes searched $= 1 + b + b^2 + \dots b^{d-1}$

$= \dfrac{b^d - 1}{b - 1}$

If goal is present at rightmost node,

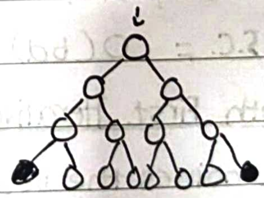
nodes searched $= \dfrac{b^d - 1}{b - 1} + b^d$

$= \dfrac{b^{d+1} - 1}{b - 1}$

$\therefore$ avg. $T = \dfrac{\dfrac{b^d - 1}{b-1} + \dfrac{b^{d+1} - 1}{b-1}}{2} \approx \dfrac{b^d}{2}$

(if b is large)

DFS



for leftmost case, no. of nodes searched $= d+1$

for rightmost case, no. of nodes searched $=$ all nodes $= \dfrac{b^{d+1} - 1}{b - 1}$

$\therefore$ avg $T = \dfrac{d+1 + \dfrac{b^{d+1} - 1}{b-1}}{2} \approx \dfrac{b^d}{2}$

(b is large).

$\therefore$ Time complexity is similar, of the order (b^d).

(III) SPACE COMPLEXITY

BFS : exponential

DFS : At each level, DFS leaves $(b-1)$ nodes in OPEN.

so, when it enters 'd' it has almost

$(b-1)(d-1)+b$ nodes in the OPEN LIST.

$\therefore$ space complexity is linear wrt 'd'. $\Rightarrow O(bd)$

(IV) Quality of solⁿ

BFS : finds the shortest solⁿ, $\therefore$ it is optimal

DFS : finds the 1st available solⁿ, which may not generally be the shortest

$\therefore$ It may be non-optimal.

Depth Bound DFS (DBDFS)

Everything is similar to DFS except the following:

→ we only search upto a predetermined depth.

→ Advantage : limits the memory.

→ Disadv. : may report failure without finding solⁿ.

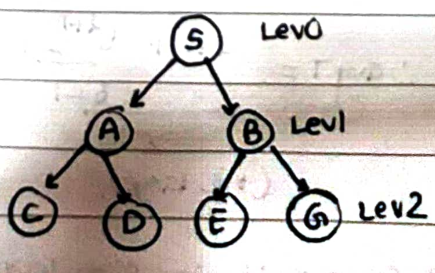
→ incomplete.

→ S.C. = $O(bd)$ (d = max allowed depth).

Depth First Iterative Deepening (DFID)

→ combines the best features of BFS (completeness, optimality) and DFS (linear space).

→ Idea is to increase the depth of the tree by 1 in every other iteration of doing DFS, therefore limiting the search space to a given level



1st iteration : $d=0$, [S]

2nd iteration : $d=1$, [S→A→B]

3rd iteration : $d=2$, [S→A→C→D→B→E→G]

Nodes covered at a depth d $= 1 + b + b^2 + \ldots b^d = \dfrac{b^{d+1} - 1}{b-1} \approx b^d$

$\therefore$ Time complexity $\approx O(b^d)$

Since it performs DBDFS in every iteration, space comp. $\approx O(bd)$

There is an additional time cost that DFID incurrs. In each cycle, it explores a new level of leaf nodes, alongside all the internal nodes that are above this level and had already been searched in the previous iterations. This happens because to perform DFID, we have to generate the tree again upto the new level in each iteration. How significant is this cost is measured in terms of the ratio of the number of internal nodes (the extra cost) to the number of leaf nodes (the new cost) $= \dfrac{\text{internal nodes}}{\text{leaf nodes}}$

For a binary tree, this ratio is highest and is equal to

$$\frac{b^0 + b + b^2 + \ldots b^{d-1}}{b^d} = \frac{\frac{b^d - 1}{b-1}}{b^d} = \frac{b^d - 1}{b^d(b-1)} \approx \frac{1}{b-1}$$

(for large d)

Heuristic Search

\# A heuristic func. matches the current state of the problem with goal state against some numerical value to estimate how far-off are we currently from the goal state. We choose the next state towards our goal according to the heuristic values of possible candidates. The state with the best (min or max, depends on type of problem) heuristic value $H(n)$ is chosen.

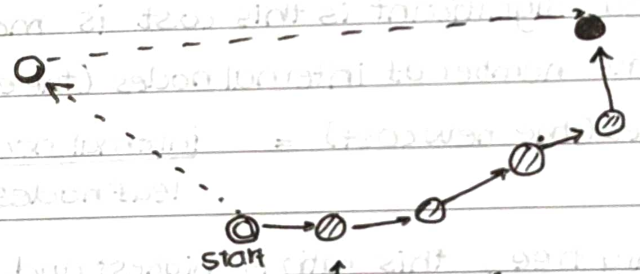
Best First search (BFS)

→ the OPEN list is taken as a priority queue with the elements of queue as the heuristic values of the next possible states from the current state.

→ we also maintain a CLOSED list to record already seen states.

→ COMPLETENESS : complete for finite domain.

→ QUALITY OF SOLN. : provides sub-optimal sol^n because it only looks at the heuristic value of the 2 states and not consider how expensive it was to reach either of them, provided they have the same heuristics. Incorporating cost cond. on heuristic will give optimal sol^{n}s.



BFS chooses this path. (via 5 moves)

→ Space complexity : if $h(n)$ is accurate, linear, else exponential

→ Time complexity : same as space.

To predict the performance, and thus accuracy, we define effective branching factor as a measure of how many extra nodes a search algo inspects.

$$\underset{\text{(e.b.f.)}}{\text{effective branching factor}} = \frac{\text{total nodes expanded}}{\text{nodes in the solution}}$$

$$\text{penetrance} = 1/\text{e.b.f.}$$

Hill Climbing Algo :

This algo is an example of a greedy algo that makes locally best choices and stops when no local better option exists. It cannot backtrack to find a better sol^n.

Steps : (I) Evaluate current state

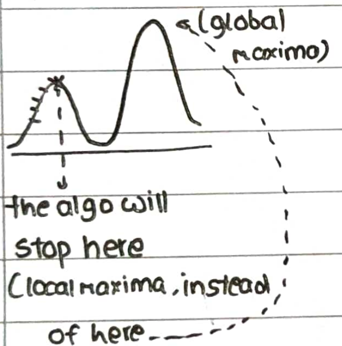
(II) Find heuristic values of local states that are neighbours.

(III) If a better heuristic value than that of the current state is found, then make it the current state, and move on.

(IV) Proceed until no better heuristic value can be found or goal state is achieved.

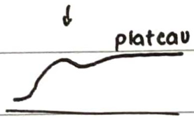
Problems :

Local Maxima



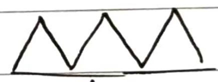
the algo will stop here (local maxima, instead of here

Plateau



when neighbours also have the same heuristic value.

Ridge



all successors have same value.

COMPLETENESS : depends on quality of H.F.

QUALITY : No guarantee can be given

SPACE : constant

TIME : proportional to the length of steepest ascent from start node.

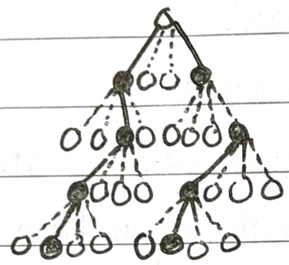
$\therefore$ linear.

Beam Search :

At various levels in a search tree, a few choices of the next state may almost look equal and the function may not be able to discriminate b/w them. In such cases, it may help to keep more than one nodes in the search tree at each level. The number of nodes kept is known as the beam width 'b'.

At each stage of expansion, all b nodes are expanded ; and from the successors, the best b are retained.

The search tree for b=2 is shown as:



at each stage, the best 2 nodes are retained and expanded upon.

The rest part is the same as the best first search.