1.3.4.6 Bidirectional Search

The idea behind bidirectional search is to **run two simultaneous** searches one forward from the initial state and the other backward from the goal, **stopping when** the two searches meet in the middle (Figure 1.37)

The motivation is that $b^{d/2} + b^{d/2}$ much less than ,or in the figure ,the area of the two small circles is less than the area of one big circle centered on the start and reaching to the goal.

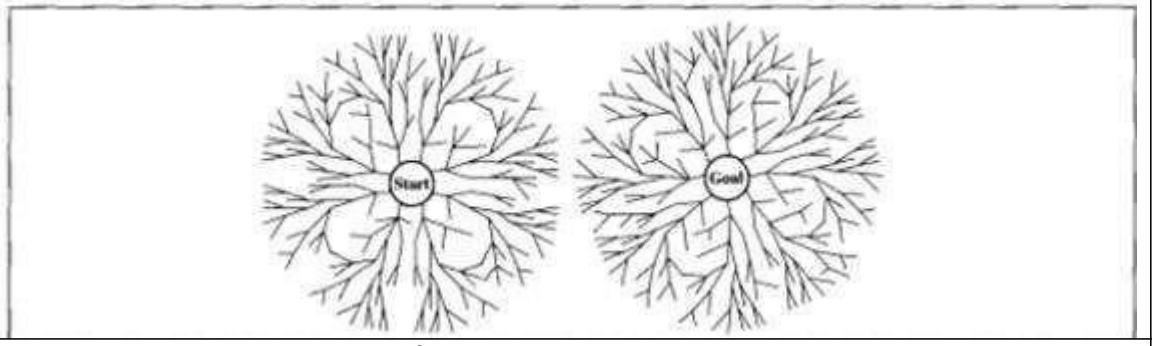


Figure 1.37 A schematic view of a bidirectional search that is about to succeed, when a Branch from the Start node meets a Branch from the goal node.

1.3.4.7 Comparing Uninformed Search Strategies

Figure 1.38 compares search strategies in terms of the four evaluation criteria .

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figure 1.38 Evaluation of search strategies, b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs >= E for positive E; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

1.3.5 Avoiding Repeated States

In searching, time is wasted by expanding states that have already been encountered and expanded before. For some problems, repeated states are unavoidable. The search trees for these problems are infinite. If we prune some of the repeated states, we can cut the search tree down to finite size. Considering search tree up to a fixed depth, eliminating repeated states yields an exponential reduction in search cost.

Repeated states can cause a solvable problem to become unsolvable if the algorithm does not detect them.

Repeated states can be the source of great inefficiency: identical sub trees will be explored many times!

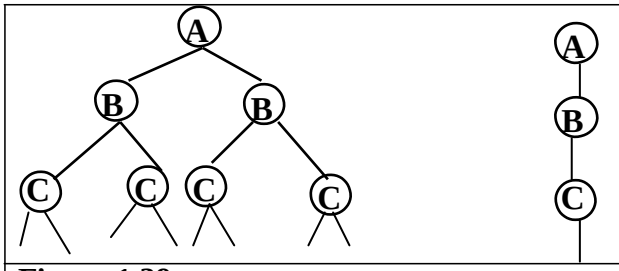


Figure 1.39

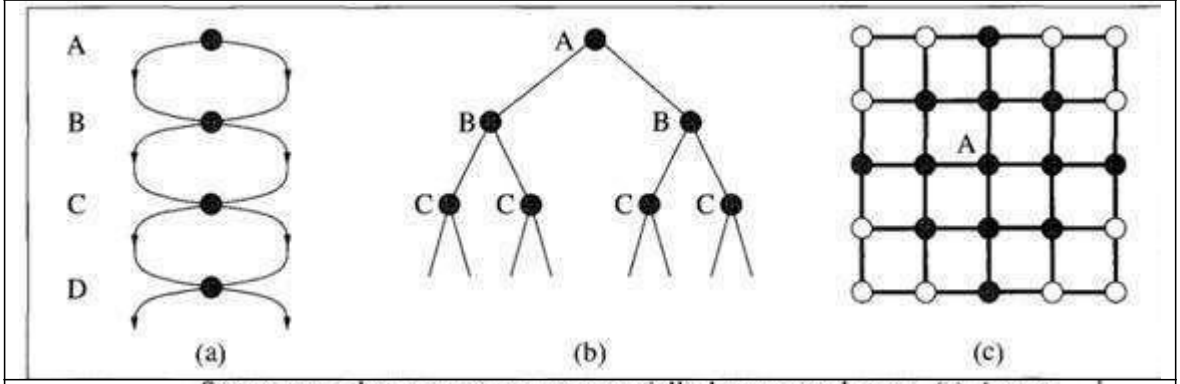
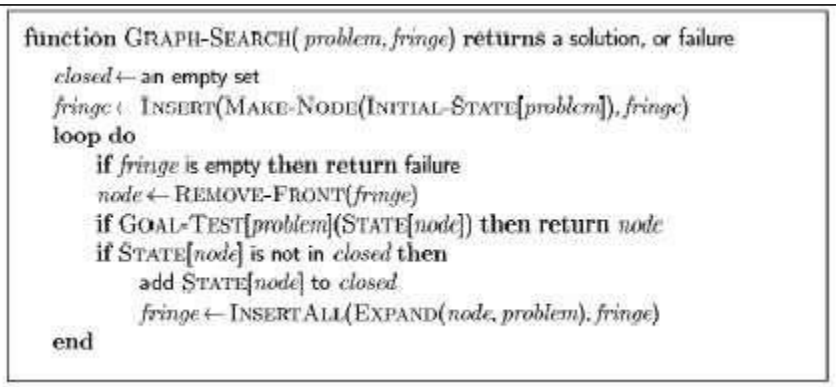


Figure 1.40 State spaces that generate an exponentially larger search tree. (a) A state space in which there are two possible actions leading from A to B, two from B to C, and so on. The state space contains $d + 1$ states, where d is the maximum depth. (b) The corresponding search tree, which has 2^d branches corresponding to the 2^d paths through the space. (c) A rectangular grid space. States within 2 steps of the initial state (A) are shown in gray.



```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
    end
```

Figure 1.41 The General graph search algorithm. The set closed can be implemented with a hash table to allow efficient checking for repeated states.

- Do not return to the previous state.
- Do not create paths with cycles.
- Do not generate the same state twice.
  - Store states in a hash table.

- Check for repeated states.

  - Using more memory in order to check repeated state
    - *Algorithms that forget their history are doomed to repeat it.*
    - Maintain Close-List beside Open-List(fringe)

Strategies for avoiding repeated states

We can modify the general TREE-SEARCH algorithm to include the data structure called the **closed list** ,which stores every expanded node. The fringe of unexpanded nodes is called the **open list.**

If the current node matches a node on the closed list, it is discarded instead of being expanded.

The new algorithm is called GRAPH-SEARCH and much more efficient than TREE-SEARCH. The worst case time and space requirements may be much smaller than $O(b^d)$.

1.3.6 SEARCHING WITH PARTIAL INFORMATION

- Different types of incompleteness lead to three distinct problem types:
  - **Sensorless problems** (conformant): If the agent has no sensors at all
  - **Contingency problem**: if the environment if partially observable or if action are uncertain (adversarial- Involving opposition, often hostile)
  - **Exploration problems**: When the states and actions of the environment are unknown.

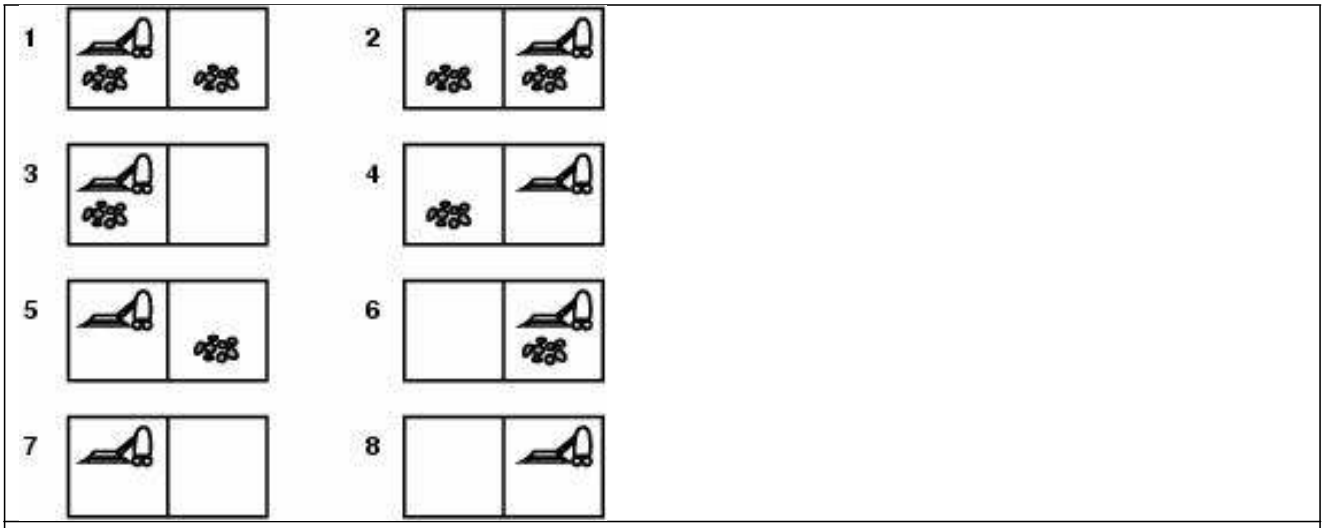
What will be the final state of vacuum problem if it is sensor less? expalin

- No sensor
- Initial State(1,2,3,4,5,6,7,8)
- After action [Right] the state (2,4,6,8)
- After action [Suck] the state (4, 8)
- After action [Left] the state (3,7)
- After action [Suck] the state (7)

- Answer : [Right,Suck,Left,Suck] coerce the world into state 7 without any sensor
- Belief State: Such state that agent belief to be there

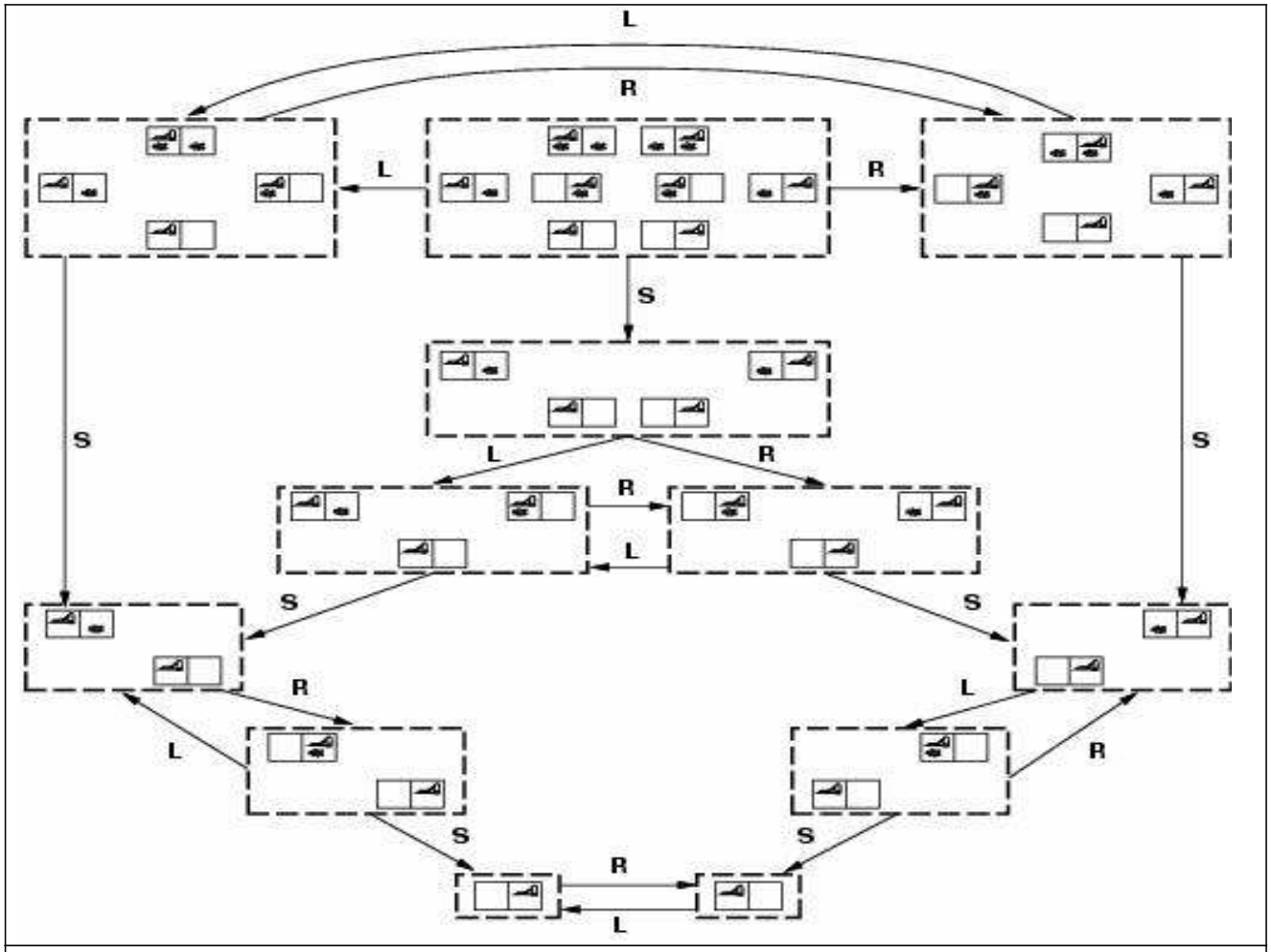
Partial knowledge of states and actions:

- *sensorless or conformant problem*
  - Agent may have no idea where it is; solution (if any) is a sequence.
- *contingency problem*
  - Percepts provide *new* information about current state; solution is a tree or policy; often interleave search and execution.
  - If uncertainty is caused by actions of another agent: *adversarial problem*
- *exploration problem*
  - When states and actions of the environment are unknown.



Figure

Initial State(1,2,3,4,5,6,7,8)



Figure

Contingency, start in {1,3}.
Murphy's law, Suck *can* dirty a clean carpet.
Local sensing: dirt, location only.

- Percept = [L,Dirty] ={1,3}
- [Suck] = {5,7}
- [Right] ={6,8}
- [Suck] in {6}={8} (Success)
- BUT [Suck] in {8} = failure

Solution??

- Belief-state: no fixed action sequence guarantees solution

Relax requirement:

- [*Suck*, *Right*, if *[R,dirty]* then *Suck*]
- Select actions based on contingencies arising during execution.

Time and space complexity are always considered with respect to some measure of the problem difficulty. In theoretical computer science ,the typical measure is the size of the state space.
In AI,where the graph is represented implicitly by the initial state and successor function,the complexity is expressed in terms of three quantities:

b,the **branching factor** or maximum number of successors of any node;
d,the **depth** of the **shallowest goal node**; and
m,the **maximum length** of any path in the state space.

Search-cost - typically depends upon the time complexity but can also include the term for memory usage.
Total–cost – It combines the search-cost and the path cost of the solution found.