A goal-based agent aims at solving problems by performing actions that lead to desirable states. Let us first consider the uninformed situation in which the agent is not given any information about the problem other than its definition. In blind search only the structure of the problem to be solved is formulated. The agent aims at reaching a goal state. The world is static, discrete, and deterministic.

The possible states of the world define the state space \( \Sigma \) of the search problem.

In the beginning the world is in the initial state \( s_1 \in \Sigma \). The agent aims at reaching one of the goal states \( G \subseteq \Sigma \).

Quite often one uses a successor function \( S: \Sigma \rightarrow \mathcal{P}(\Sigma) \) to define the agent's possible actions. Each state \( s \in \Sigma \) has a set of legal successor states \( S(s) \) that can be reached in one step.

Paths have a non-negative cost, most often the sum of costs of steps in the path.
• The definitions above naturally determine a directed and weighted graph
• The simplest problem in searching is to find out whether any goal state can be reached from the initial state \( S_1 \)
• The actual solution to the problem, however, is a path from the initial state to a goal state
• Usually one takes also the costs of paths into account and tries to find a cost-optimal path to a goal state
• Many tasks of a goal-based agent are easy to formulate directly in this representation

For example, the states of the world of 8-puzzle (a sliding-block puzzle) are all \( 9!/2 = 181 440 \) reachable configurations of the tiles
• Initial state is one of the possible states
• The goal state is the one given on the right above
• Possible values of the successor function are moving the blank to left, right, up, or down
• Each move costs one unit and the path cost is the total number of moves
• Donald Knuth’s (1964) illustration of how infinite state spaces can arise
• Conjecture: Starting with the number 4, a sequence of factorial, square root, and floor operations will reach any desired positive integer

\[
\sqrt{4!} = 5
\]

- **States**: Positive numbers
- **Initial state** \( s_0 \): 4
- **Actions**: Apply factorial, square root, or floor operation
- **Transition model**: As given by the mathematical definitions of the operations
- **Goal test**: State is the desired positive integer
Search Tree

- When the search for a goal state begins from the initial state and proceeds by steps determined by the successor function, we can view the progress of search in a tree structure.
- When the root of the search tree, which corresponds to the initial state, is expanded, we apply the successor function to it and generate new search nodes to the tree — as children of the root — corresponding to the successor states.
- The search continues by expanding other nodes in the tree respectively.
- The search strategy (search algorithm) determines in which order the nodes of the tree are expanded.
• The node is a data structure with five components:
  – The state to which the node corresponds,
  – Link to the parent node,
  – The action that was applied to the parent to generate this node, and
  – The cost of the path from the initial state to the node

• Global parameters of a search tree include:
  – \( b \) (average or maximum) branching factor,
  – \( d \) the depth of the shallowest goal, and
  – \( m \) the maximum length of any path in the state space

### 3.4 Uninformed Search Strategies

• The search algorithms are implemented as special cases of normal tree traversal
• The time complexity of search is usually measured by the number of nodes generated to the tree
• Space complexity, on the other hand, measures the number of nodes that are maintained in the memory at the same time
• A search algorithm is **complete** if it is guaranteed to find a solution (reach a goal state starting from the initial state) when there is one
• The solution returned by a complete algorithm is not necessarily **optimal**: several goal states with different costs may be reachable from the initial state
3.4.1 Breadth-first search

- When the nodes of the search tree are traversed in level-order, the tree gets searched in breadth-first order.
- All nodes at a given depth are expanded before any nodes at the next level are expanded.
- Suppose that the solution is at depth $d$.
- In the worst case we expand all but the last node at level $d$.
- Every node that is generated must remain in memory, because it may belong to the solution path.
- Let $b$ be the branching factor of the search.
- Thus the worst-case time and space complexities are $b + b^2 + b^3 + \cdots + b^d = O(b^d)$. 

![Diagram of a breadth-first search tree](image)
In general, the weakness of breadth-first search is its exponential (in depth) time and space usage.

The need to maintain all explored nodes causes problems.

For example, when the tree has branching factor $10$, nodes are generated $1$ million per second and each node requires $1$ KB of storage, then:

<table>
<thead>
<tr>
<th>Depth</th>
<th>Nodes</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>110</td>
<td>.11 μsec</td>
<td>107 KB ($10^3$)</td>
</tr>
<tr>
<td>4</td>
<td>11110</td>
<td>11 μsec</td>
<td>10.6 MB ($10^6$)</td>
</tr>
<tr>
<td>6</td>
<td>$10^6$</td>
<td>1.1 sec</td>
<td>1 GB ($10^9$)</td>
</tr>
<tr>
<td>8</td>
<td>$10^8$</td>
<td>2 min</td>
<td>103 GB</td>
</tr>
<tr>
<td>10</td>
<td>$10^{10}$</td>
<td>3 h</td>
<td>10 teraB ($10^{12}$)</td>
</tr>
<tr>
<td>12</td>
<td>$10^{12}$</td>
<td>13 days</td>
<td>1 petaB ($10^{15}$)</td>
</tr>
<tr>
<td>14</td>
<td>$10^{14}$</td>
<td>3.5 years</td>
<td>99 petaB</td>
</tr>
<tr>
<td>16</td>
<td>$10^{16}$</td>
<td>350 years</td>
<td>10 exaB ($10^{18}$)</td>
</tr>
</tbody>
</table>

Breadth-first search is optimal when all step costs are equal, because it always expands the shallowest unexpanded node.

On the other hand, to this special case of uniform-cost search, we could also apply the greedy algorithm, which always expands the node with the lowest path cost.

If the cost of every step is strictly positive the solution returned by the greedy algorithm is guaranteed to be optimal.

The space complexity of the greedy algorithm is still high.

When all step costs are equal, the greedy algorithm is identical to breadth-first search.
3.4.3 Depth-first search

- When the nodes of a search tree are expanded in preorder, the tree gets searched in depth-first order.
- The deepest node in the current fringe of the search tree becomes expanded.
- When one branch from the root to a leaf has been explored, the search backs up to the next deepest node that still has unexplored successors.
- Depth-first search has very modest memory requirements:
  - It needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path.
  - Depth-first search requires storage of only $b m$ nodes.
Using the same assumptions as in the previous example, we find that depth-first search would require \( 156 \times 10^{16} \) instead of \( 10^{16} \) at depth 16 (7 trillion times less). If the search tree is infinite, depth-first search is not complete. The only goal node may always be in the branch of the tree that is examined the last. In the worst case also depth-first search takes an exponential time: \( O(b^m) \). At its worst \( m \gg d \), the time taken by depth-first search may be much more than that of breadth-first search. Moreover, we cannot guarantee the optimality of the solution that it comes up with.

### 3.4.4 Depth-limited search

- We can avoid examining unbounded branches by limiting the search to depth \( \ell \).
  - The nodes at level \( \ell \) are treated as if they have no successors.
  - Depth-first search can be viewed as a special case with \( \ell = m \).
  - When \( d \leq \ell \) the search algorithm is complete, but in general one cannot guarantee finding a solution.
  - Obviously, the algorithm does not guarantee finding an optimal solution.
  - The time complexity is now \( O(b^\ell) \) and the space complexity is \( O(b\ell) \).
3.4.5 Iterative deepening search

- We can combine the good properties of limited-depth search and general depth-first search by letting the value of the parameter $\ell$ grow gradually.
- E.g., $\ell = 0, 1, 2, \ldots$ until a goal node is found.
- In fact, thus we gain a combination of the benefits of breadth-first and depth-first search.
- The space complexity is controlled by the fact that the search algorithm is depth-first search.
- On the other hand, gradual growth of the parameter $\ell$ guarantees that the method is complete.
- It is also optimal when the cost of a path is non-decreasing function of the depth of the node.

3.4.6 Bidirectional search

- If node predecessors are easy to compute — e.g., $\text{Pred}(n) = S(n)$ — then search can proceed simultaneously forward from the initial state and backward from the goal.
- The process stops when the two searches meet.
- The motivation for this idea is that $2b^{d/2} \ll b^d$.
- If the searches proceeding to both directions are implemented as breadth-first searches, the method is complete and leads to an optimal result.
- If there is a single goal state, the backward search is straightforward, but having several goal states may require creating new temporary goal states.
An important complication of the search process is the possibility to expand states that have already been expanded before.

Thus, a finite state space may yield an infinite search tree.

A solvable problem may become practically unsolvable.

Detecting already expanded states usually requires storing all states that have been encountered.

One needs to do that even in depth-first search.

On the other hand, pruning may lead to missing an optimal path.