Solving Problems by Searching

Abdeslam Boularias

Friday, September 9, 2016
Solving Problems by Searching (Planning agents)

How can an agent find a sequence of actions that achieves its goals when no single action will do?

Route-finding problem using a simplified road map of Romania
Solving Problems by Searching: Examples

Route planning

Robot navigation

(Copyright Wikimedia Commons)

Start State Goal State

7 2 4
5 6
8 3 1

⇒

3 4 5
6 7 8

8 queens puzzle

Sliding-block puzzle
A well-defined problem is described by

- **States**: set of all possible situations (configurations, positions). Example: locations on a map.
- **Initial state**: starting state. Example: \( In(Busch\ campus) \)
- **Goal states**: destinations, final positions. Example: \( In(New\ Brunswick) \). The goal state can be implicitly defined by test, such as in the *eight queens puzzle*.
- **Actions**: set of everything an agent can do to change its current state. Examples: \( Go(Livingston\ campus) \)
- **Transition model**: the effect of each action. Example:
  \[
  \text{RESULT}(In(Busch\ campus), Go(Livingston\ campus)) = In(Livingston\ campus)
  \]
- **Path cost**: function that assigns a numeric cost to each path. A path is a sequence of states connected by a sequence of actions. Example: *length in kilometers*. 
We make the following assumptions

- The cost of a path is the sum of the costs of the individual actions along the path. The **step cost** of action $a$ in state $s$ to reach state $s'$ is denoted by $c(s, a, s')$.
- The environment is known and fully observable: the agent knows exactly where it is and how its actions will change its state.
- Actions are deterministic: each action has exactly one outcome.

Under these assumptions, the solution to any problem is a fixed sequence of actions. Under these assumptions, the agent can ignore its observations, why?

This is known as open-loop control (in contrast with closed-loop control where actions are chosen according to observations).
Toy example: the vacuum world

The state space for the vacuum world.
Links denote actions: $L = \text{Left}$, $R = \text{Right}$, $S = \text{Suck}$. 
Toy problem : the vaccum world

- **States**: (agent location, dirt location). There are two locations, each of which may or may not contain dirt. Thus, there are $2 \times 2^2 = 8$ possible world states. In an environment with $n$ locations, there are $n2^n$ states.

- **Initial state**: any state can be a starting state.

- **Goal states**: all the squares are clean (two goal states).

- **Actions**: Suck, move left, move right.

- **Transition model**: Actions have their expected effects, except that moving left in the left square or right in the right square has no effect. Also, sucking in an already clean square has no effect.

- **Path cost**: Each action has a cost of 1 (or a value related to time or energy).
Toy example: the sliding-block puzzle

Instance of the 8-puzzle
Toy problem: the sliding-block puzzle

- **States**: location of each of the eight tiles and the blank. There are $9!$ states.
- **Initial state**: Any state.
- **Goal state**: Any chosen configuration (only half of the states are accessible from a given initial state).
- **Actions**: Move the blank space to *left*, *right*, *up* or *down*.
- **Transition model**: deterministic, depends on the state and action.
- **Path cost**: Each action has a cost of 1. We try to minimize the number of steps needed to solve the puzzle.
Toy example: the eight queens puzzle

Attempt to solve the eight queens puzzle
Toy problem: the eight queens puzzle

- **States**: Placements of up to eight queens on a chessboard. There are $P(64, 8)$ states.
  
  $P(64, 8) = 64 \times 63 \times 62 \times 61 \times 60 \times 59 \times 58 \times 57 \approx 1.8 \times 10^{14}$.

- **Initial state**: Empty board.

- **Goal state**: 8 queens on the board and none is attacked.

- **Actions**: Put a queen in any empty square.

- **Transition model**: adds a queen to the specified square.

- **Path cost**: Each action has a cost of 0. Each path to the goal will have exactly 8 steps, so we do not care about minimizing the number of steps.
Toy problem: the eight queens puzzle

- **Actions**: Put a queen in an empty square that is not being attacked, and that is on the leftmost column that is empty. The first queen is placed anywhere on the first column.

- **States**: Placements of up to eight queens on a chessboard such that no queen is attacking another. This more clever representation reduces the number of possible states from $1.8 \times 10^{14}$ to 2057.
Toy problem: Donald Knuth’s conjecture (1964)

One can start at 3 and reach any integer by iterating factorial, sqrt, and floor.

Example: \[ \left\lfloor \sqrt{\sqrt{(3!)!}} \right\rfloor = 5 \]

- **States**: set of natural numbers \( \mathbb{N} \)
- **Initial state**: number 3.
- **Goal state**: any given natural number.
- **Actions**: floor, square root, and factorial operations.
- **Transition model**: result of the operation.
- **Path cost**: Each action has a cost of 1.
Search tree

- A solution is an action sequence (path).
- Search algorithms work by considering various possible action sequences.
- Starting from the initial state, we consider all the different actions that we can execute.
- Each action leads to a new state.
- For each new state, we consider all the possible actions and the resulting states.
- This process forms a tree of states, the initial state is the root of the tree.
- The goal state (or states) is somewhere in the tree.
- The process of finding the goal state is called tree searching.
- The different searching methods are called tree search algorithms.
Search tree

(a) The initial state

(b) After expanding Arad

(c) After expanding Sibiu

(d) After expanding Fagaras

Search tree for finding a route from Arad to Bucharest
Tree searching

- A node is said to be **expanded** when its children are added to the search.

- The set of nodes that have been added to the search but have not yet been expanded (the leaves) is called the **open list**, the **fringe**, or the **frontier**.

```plaintext
function TREE-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        expand the chosen node, adding the resulting nodes to the frontier
    end loop
end TREE-SEARCH
```

Generic algorithm for tree searching

Search algorithms vary according to how they choose which state to expand next (search strategy).
Searching for solutions

- Notice the repeated states in the search tree in the example.
- Loopy paths can lead to infinite search trees.
- The set of expanded nodes is called the **closed** (or **explored**) list.
- Graph searching algorithms avoid redundant paths by keeping track of the explored nodes.

```plaintext
function GRAPH-SEARCH(F) returns a solution, or failure
initialize the frontier using the initial state of F
initialize the explored set to be empty
loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
    only if not in the frontier or explored set
```

Generic algorithm for graph searching
Implementing a search tree algorithm

In a search tree, each node $n$ is represented by a data structure that contains the following fields:

- **$n$.State**: the state that the node corresponds to. Example: $n$.State = In(Busch campus)
- **$n$.Parent**: the parent of node $n$.
- **$n$.Action**: the action that lead to node $n$ (from $n$.Parent).
- **$n$.Path-cost**: the cost path cost from the root to node $n$.
- **$n$.Depth**: the path cost from the root to the current node.
Data structure used for nodes in a search tree

Node

STATE

PARENT

ACTION = Right

PATH-COST = 6

Node

STATE

PARENT

ACTION = Right

PATH-COST = 6
Implementing a search tree algorithm

The nodes are kept in a queue. The operations on a queue are as follows:

- **EMPTY?(queue)**: returns true if the queue is empty.
- **POP(queue)**: removes the first element and returns it.
- **INSERT(element,queue)**: inserts an element and returns the resulting queue.
- **INITIALIZE(element)**: returns a queue that contains element.

Queues are characterized by the order in which they store the inserted nodes.

- **FIFO queue**: first-in, first-out
- **LIFO queue**: last-in, first-out
- **Priority queue**: pops the element with the highest priority according to some function

Different queue structures give different search algorithms.
Search Algorithm Comparison Criteria And Complexity Parameters

There are four criteria comparing the various search tree algorithms:

- **Completeness**: Is the algorithm guaranteed to find a solution when there is one?
- **Optimality**: Does the strategy find the optimal solution?
- **Time complexity**: How long does it take to find a solution?
- **Space complexity**: How much memory is needed to perform the search?

The following parameters are used to calculate and compare the performance of an algorithm:

- **b**: the maximum branching factor (number of children per node)
- **d**: the depth of the shallowest goal state
- **m**: maximum length of path in the tree
Search Algorithms

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening depth-first search
- Bidirectional search
Breadth-First Search (BFS)

Breadth-First Search expands all the nodes first before expanding the nodes at the next level.

Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

Breadth-first search always has the shallowest path to every node on the frontier.
Breadth-First Search (BFS)

```plaintext
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure

    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)

    frontier ← a FIFO queue with node as the only element
    explored ← an empty set

    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier) /* chooses the shallowest node in frontier */
        add node.STATE to explored

        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
```

Figure 3.11 Breadth-first search on a graph.

Technically, breadth-first search is optimal if the path cost is a nondecreasing function of the depth of the node. The most common such scenario is that all actions have the same cost.

So far, the news about breadth-first search has been good. The news about time and space is not so good. Imagine searching a uniform tree where every state has \( b \) successors. The root of the search tree generates \( b \) nodes at the first level, each of which generates \( b \) more nodes, for a total of \( b^2 \) at the second level. Each of these generates \( b \) more nodes, yielding \( b^3 \) nodes at the third level, and so on. Now suppose that the solution is at depth \( d \).

In the worst case, it is the last node generated at that level. Then the total number of nodes generated is

\[
O(b^d)
\]

If the algorithm were to apply the goal test to nodes when selected for expansion, rather than when generated, the whole layer of nodes at depth \( d \) would be expanded before the goal was detected and the time complexity would be \( O(b^d + 1) \).

As for space complexity: for any kind of graph search, which stores every expanded node in the expanded set, the space complexity is always within a factor of \( b \) of the time complexity. For breadth-first graph search in particular, every node generated remains in memory. There will be \( O(b^d - 1) \) nodes in the expanded set and \( O(b^d) \) nodes in the frontier.

Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.
Properties of Breadth-First Search

- **Frontier queue**: First In First Out (FIFO)
- **Completeness**: Complete if branching factor $b$ is finite
- **Optimality**: Optimal only if all the costs of the edges are equal (shallowest path in the tree is not always the shortest in the problem).
- **Time complexity**: $O\left(\sum_{i=0}^{d-1} b^i\right) = O(b^d)$
- **Space complexity**: $O\left(\sum_{i=0}^{d-1} b^i\right) = O(b^d)$
Scalability of Breadth-First Search

Exponential complexity $O(b^d)$ is scary!

<table>
<thead>
<tr>
<th>Depth</th>
<th>Nodes</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>110</td>
<td>.11 milliseconds</td>
<td>107 kilobytes</td>
</tr>
<tr>
<td>4</td>
<td>11,110</td>
<td>11 milliseconds</td>
<td>10.6 megabytes</td>
</tr>
<tr>
<td>6</td>
<td>$10^6$</td>
<td>1.1 seconds</td>
<td>1 gigabyte</td>
</tr>
<tr>
<td>8</td>
<td>$10^8$</td>
<td>2 minutes</td>
<td>103 gigabytes</td>
</tr>
<tr>
<td>10</td>
<td>$10^{10}$</td>
<td>3 hours</td>
<td>10 terabytes</td>
</tr>
<tr>
<td>12</td>
<td>$10^{12}$</td>
<td>13 days</td>
<td>1 petabyte</td>
</tr>
<tr>
<td>14</td>
<td>$10^{14}$</td>
<td>3.5 years</td>
<td>99 petabytes</td>
</tr>
<tr>
<td>16</td>
<td>$10^{16}$</td>
<td>350 years</td>
<td>10 exabytes</td>
</tr>
</tbody>
</table>

Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node. Bad news: It doesn’t get much better with faster computers.
Uniform-cost Search

- Breadth-First Search is optimal only when all step costs are equal.
- To solve this problem, Uniform-cost Search expands the node \( n \) with the lowest path cost \( g(n) \), instead of the shallowest node.
- The queue of the frontier is ordered by path cost.
- In BFS all the nodes in the queue have the same path cost, the goal test is performed when a node is generated.
- In Uniform-cost Search, the nodes in the queue have different path costs, the goal test is performed when a node is expanded.
Uniform-cost Search

Part of Romania, selected to illustrate uniform-cost search

Nodes are expanded in the following order:
- Sibiu,
- Rimnicu,
- Fagaras (goal node Bucharest generated here, but this is not the optimal path yet),
- Pitesti,
- Bucharest (optimal path found).
Uniform-cost Search

Uniform Cost Search (UCS) calculates the path cost of all the nodes in the fringe and expands the one that has the minimum cost. Figure 10 describes the operation of the algorithm. After the initial node S is expanded, three nodes are generated: A, B and C. Among them node A has the smallest path cost and is selected for expansion, generating node G, a goal node. Then node B is the node with the minimum path cost and is expanded first, also resulting in a goal node. The goal node produced by B is the optimal solution and the node that will be visited first by UCS.

There are two parameters used to express time and space complexity for UCS:

- $C^*$: optimal cost
- $\epsilon$: smallest edge cost, $\epsilon > 0$

Figure 10: Uniform-First Search.

These are the properties of Uniform Cost Search:

- Complete and Optimal
- Time Complexity $O\left(b \lfloor \frac{C^*}{\epsilon} \rfloor + 1\right)$
- Space Complexity $O\left(b \lfloor \frac{C^*}{\epsilon} \rfloor + 1\right)$

It is complete if $b$ is finite and also if the smallest edge cost is $\epsilon > 0$. In the latter case the algorithm will not search infinite paths. At some point along every path the path cost will be greater than other alternatives along the tree due to the minimum edge cost $\epsilon$. Thus, the algorithm is complete in this case. It is optimal because it expands optimal. Moreover, its time complexity is worse than BFS. But, it has the best space complexity compared to the alternatives.
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure

node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
frontier ← a priority queue ordered by PATH-COST, with node as the only element
explored ← an empty set

loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier) /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
            frontier ← INSERT(child, frontier)
        else if child.STATE is in frontier with higher PATH-COST then
            replace that frontier node with child

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for frontier needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

Figure 3.15 Part of the Romania state space, selected to illustrate uniform-cost search. May be on a suboptimal path. The second difference is that a test is added in case a better path is found to a node currently on the frontier. Both of these modifications come into play in the example shown in Figure 3.15, where the problem is to get from Sibiu to Bucharest. The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99, respectively. The least-cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost 80 + 97 = 177. The least-cost node is now Fagaras, so it is expanded, adding Bucharest with cost 99 + 211 = 310. Now a goal node has been generated, but uniform-cost search keeps going, choosing Pitesti for expansion and adding a second path.
Dijkstra's algorithm (a slightly different variant of uniform-cost search)

Dijkstra uses a list of unvisited vertices (initially contains all vertices), and a table that keeps track the shortest distance to each vertex so far (initialized to infinity) instead of the open and closed lists.

```python
1  function Dijkstra(Graph, source):
2      create vertex set Q
3
4      for each vertex v in Graph:           // Initialization
5          dist[v] ← INFINITY              // Unknown distance from source to v
6          prev[v] ← UNDEFINED            // Previous node in optimal path from source
7          add v to Q                      // All nodes initially in Q (unvisited nodes)
8
9      dist[source] ← 0                     // Distance from source to source
10
11     while Q is not empty:
12        u ← vertex in Q with min dist[u]  // Source node will be selected first
13        remove u from Q
14
15        for each neighbor v of u:         // where v is still in Q.
16            alt ← dist[u] + length(u, v)
17            if alt < dist[v]:             // A shorter path to v has been found
18                dist[v] ← alt
19                prev[v] ← u
20
21     return dist[], prev[]
```

Inconvenient: the graph could be too large to store in memory!
Properties of Uniform-cost Search

- **Frontier queue**: Ordered by path cost
- **Completeness**: Complete if the branching factor is finite
- **Optimality**: Optimal. Whenever uniform-cost search selects a node \( n \) for expansion, the optimal path to that node has been found.
- **Time complexity**: \( O(b^{1+\lceil C^*/\epsilon \rceil}) \), where \( C^* \) is the path cost of the optimal path and \( \epsilon \) is the minimum step cost.
- **Space complexity**: \( O(b^{1+\lceil C^*/\epsilon \rceil}) \)

\( O(b^{1+\lceil C^*/\epsilon \rceil}) \) can be much worse than \( O(b^d) \) (the complexity of BFS), because Uniform-cost Search can explore a large number of small steps before trying longer steps that may lead immediately to the goal.
Properties of Uniform-cost Search

- Another worst-case bound on the time and space complexities can be obtained by considering the maximum number of operations on vertices (states) in the graph.
- Let $V$ be the set of vertices in the graph, and let $|V|$ denote the total number of vertices.
- Each vertex is expanded only once at most, then there are $|V|$ expansions at most.
- Every time a vertex is expanded, at most $|V|$ children are pushed to the open list after updating their distances, because a vertex cannot be connected to more than $|V|$.
- Therefore, there are $|V||V|$ operations at most.

The time and space complexity of uniform-cost search is $O(\min\{|V|^2, b^{1+\lceil C^*/\epsilon \rceil}\})$
Depth-First Search (DFS)

- Depth-first search always expands the deepest node in the current frontier of the search tree.
- The frontier is stored in a LIFO (Last In First Out) queue.
Depth-First Search (DFS)

(a) The initial state

(b) After expanding Arad

(c) After expanding Sibiu
Properties of Depth-First Search (DFS)

- DFS can run into an infinite loop when repeated states are allowed.
- In infinite state spaces, DFS fails if an infinite non-goal path is encountered. Example: Knuth’s problem.
- DFS is non-optimal, it returns the first path that contains the goal.
- Time complexity: $O(b^m)$ where $m$ is the maximum depth of any node. This is much worse than BFS complexity $O(b^d)$ where $d$ is the depth of the shallowest node.
- Space complexity: $O(bm)$, and only $O(m)$ using backtracking (where we do not need to generate all the successors of each node).
Depth-limited Search

- To avoid failure of DFS in infinite state spaces, we supply depth-first search with a predetermined depth limit $l$.
- This introduces a source of incompleteness if $l < d$.
- Time complexity: $O(b^l)$
- Space complexity: $O(bl)$
- Depth limit $l$ is chosen depending on the problem. Example: we know that in the map of Romania there are 20 cities, therefore $l = 19$ makes sense. Moreover, any city can be reached from any other city in at most 9 steps. This number is known as the diameter of the state space. Therefore, $l = 9$ is a better choice.
Recursive Depth-limited Search

```plaintext
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
    return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    else if limit = 0 then return cutoff
    else
        cutoff_occurred? ← false
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            result ← RECURSIVE-DLS(child, problem, limit - 1)
            if result = cutoff then cutoff_occurred? ← true
            else if result ≠ failure then return result
        if cutoff_occurred? then return cutoff else return failure
```

Figure 3.17 A recursive implementation of depth-limited tree search.

map carefully, we would discover that any city can be reached from any other city in at most 9 steps. This number, known as the diameter of the state space, gives us a better depth limit, DIAMETER which leads to a more efficient depth-limited search. For most problems, however, we will not know a good depth limit until we have solved the problem.

3.4.5 Iterative deepening depth-first search

Iterative deepening search (or iterative deepening depth-first search) is a general strategy, often used in combination with depth-first tree search, that finds the best depth limit. It does this by gradually increasing the limit—first 0, then 1, then 2, and so on—until a goal is found. This will occur when the depth limit reaches \( d \), the depth of the shallowest goal node. The algorithm is shown in Figure 3.18. Iterative deepening combines the benefits of depth-first and breadth-first search. Like depth-first search, its memory requirements are modest: \( O(b^d) \) to be precise. Like breadth-first search, it is complete when the branching factor is finite and optimal when the path cost is a nondecreasing function of the depth of the node. Figure 3.19 shows four iterations of ITERATIVE-DEEPENING-SEARCH on a binary search tree, where the solution is found on the fourth iteration.

Iterative deepening search may seem wasteful because states are generated multiple times. It turns out this is not too costly. The reason is that in a search tree with the same (or nearly the same) branching factor at each level, most of the nodes are in the bottom level, so it does not matter much that the upper levels are generated multiple times. In an iterative deepening search, the nodes on the bottom level (depth \( d \)) are generated once, those on the
Iterative Deepening Depth-First Search

- To solve the problem of choosing the right depth $l$, one can start with $l = 0$ and iteratively increase it during the search.
- Iterative Deepening DFS runs repeatedly through the tree, increasing the depth limit with each iteration until it reaches the depth of the shallowest goal.
- Iterative Deepening DFS has the space complexity of DFS and the completeness properties of BFS.
Iterative Deepening Depth-First Search

Limit = 0

Limit = 1

Limit = 2

Limit = 3

Limit = 0

Limit = 1

Limit = 2

Limit = 3

Limit = 0

Limit = 1

Limit = 2

Limit = 3
Iterative Deepening Depth-First Search

Properties:

- Complete: by iteratively increasing the limit, at some point the algorithm reaches a goal node and the algorithm never searches an infinite path.
- Time complexity: $O(b^d)$, the number of nodes using depth limit $d$ is equal to the number of nodes using all depths $< d$.
- Space complexity: $O(bd)$, the same as in DFS.
- Optimality: Yes, if all the edges have the same cost.
Bidirectional Search

- Runs two simultaneous breadth-first searches: one forward from the initial state, and one backward from the goal.
- Stop when the two meet in the middle.
Bidirectional Search Properties

- Complete: like in BFS.
- Time complexity: $O(2b^{d/2}) = O(b^{d/2})$
- Space complexity: $O(2b^{d/2}) = O(b^{d/2})$
- Optimality: Yes, if all the edges have the same cost, like in BFS.