CHAPTER 3
Solving Problems by Searching

This material was originated by personnel performing work related to Computer Science 533. Part or all of this information may constitute sensitive student confidential and/or proprietary material.

Copyright, 2000 © Cheryl Arany, Roi Barak, Martin Dang & Arthur Kt
We have kept the breakdown of the sub-chapters to that of the AI course book for easy of reference.

The purpose of this document is:

- To help the student understand the basic concepts of Search
- To familiarize with terminology
- To present an example of different search examples.
3.1 Problem-Solving Agents
3.2 Formulating Problems
3.3 Example Problems
3.4 Searching for Solutions
3.5 Search Strategies
3.6 Avoiding Repeated States
3.8 Summary
3.1 Problem-Solving Agents

- Problem-solving agents decide what to do by finding sequences of actions that lead to desirable states.
- Goal formulation is the first step in problem solving.
- In addition to formulating a goal, the agent may need to decide on other factors that affect the achieve ability of the goal.
3.1 Problem-solving agents (cont.)

- A goal is a set of world states.
- Actions can be viewed as causing transitions between world states.
- What sorts of actions and states does the agent need to consider to get it to its goal state?
- Problem formulation is the process of deciding what actions and states to consider and follows goal formulation.
3.1 Problem-solving agents (cont.)

• What if there is no additional information available?
• In some cases, the agent will not know which of its possible actions is best because it does not know enough about the state that results from taking each action.
• The best it can do is choose one of the actions at random.
3.1 Problem-solving agents (cont.)

- An agent with several immediate options of unknown value can decide what to do by examining different possible sequences of actions that lead to states of known value then choosing the best one.

- This process is called a search.
3.1 Problem-solving agents (cont.)

- The search algorithm takes a problem as input and returns a solution in the form of an action sequence.
- Once a solution is found, the actions it recommends can be carried out.
- This is called the *execution phase*. 
3.1 Problem-solving agents (cont.)

- A simple formulate, search, execute design is as follows:
  - after formulating a goal and a problem to solve, the agent calls a search procedure to solve it.
  - it then uses the solution to guide its actions, doing whatever the solution recommends.
  - Once the solution has been executed, the agent will find a new goal.

```python
function SIMPLE-PROBLEM-SOLVING-AGENT(p) returns an action
    inputs: p, a perpect
    static: s, an action sequence, initially empty
            state, some description of the current world state
            g, a goal, initially null
            problem, a problem formulation

    state ← UPDATE-STATE(state, p)
    if s is empty then
        g ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, g)
        s ← SEARCH(problem)
    action ← RECOMMENDATION(s, state)
    s ← REMAINDER(s, state)
    return action
```

Figure 3.1 A simple problem-solving agent.
3.2 Formulating Problems

There are four different types of problems:

1. single-state problems
   - Suppose the agent’s sensors give it enough information to tell exactly which state it is in and it knows exactly what each of its actions does. Then it can calculate exactly which state it will be in after any sequence of actions.

2. multiple-state problems
   - This is the case when the world is not fully accessible. The agent must reason about the sets of states that it might get to rather than single states.
3.2 Formulating Problems (cont.)

3 contingency problems

- This occurs when the agent must calculate a whole tree of actions rather than a single action sequence. Each branch of the tree deals with a possible contingency that might arise.

- Many problems in the real, physical world are contingency problems because exact prediction is impossible. These types of problems require very complex algorithms.

- They also follow a different agent design in which the agent can act before it has found a guaranteed plan.
4 exploration problems

- In this type of problem, the agent learns a ‘map’ of the environment by experimenting, gradually discovering what its actions do and what sorts of states exist. This occurs when the agent has no information about the effects of its actions.

- If it survives, this ‘map’ can be used to solve subsequent problems.
3.2.1 Well-defined problems & solutions

• A problem is a collection of information that the agent will use to decide what to do.

• The basic elements of a problem definition are the states and actions. These are more formally stated as follows:
  – the initial state that the agent knows itself to be in
  – the set of possible actions available to the agent. The term operator will be used to denote the description of an action in terms of which state will be reached by carrying out the action

• Together, these define the state space of the problem: the set of all states reachable from the initial state by any sequence of actions.
3.2.1 Well-defined problems & solutions (cont.)

- The next element of a problem is the following:
  - The goal test, which the agent can apply to a single state description to determine if it is a goal state.

- Finally, there may be cases where one solution is preferable to another even though they both reach the goal.

- A path in the state space is any sequence of actions leading from one state to another.
  - A path cost function is a function that assigns cost to a path. We will consider the cost of a path as the sum of the costs of the individual actions along the path.
3.2.1 Well-defined problems & solutions (cont.)

• We can define a datatype to represent problems:
  – **datatype**: PROBLEM
  – **components**: INITIAL-STATE, OPERATORS, GOAL-TEST, PATH-COST-FUNCTION
• Instances of this datatype will be the input to the search algorithms.
• The output of a search algorithm is a solution, that is, a path from the initial state to a state that satisfies the goal test.
To deal with multiple-state problems, only minor modifications are needed.

An initial state set is required, a set of operators specifying for each action the set of states reached from any given state and a goal test and path cost as before.

A path now connects sets of states and a solution is now a path that leads to a set of states all of which are goal states.

The state space is replaced by the state set space.
3.2.2 Measuring problem-solving performance

• The effectiveness of a search can be measured in at least three ways.
  1. Does it find a solution at all?
  2. Is it a good solution, low path cost?
  3. What is the search cost associated with the time and memory required to find a solution?

• The total cost of the search is the sum of the path cost and the search cost.
3.2.2 Measuring problem-solving performance

- The agent must somehow decide what resources to devote to search and what resources to devote to execution.
- For large, complicated state spaces, there is a trade-off -- the agent can search for a very long time and get an optimal solution, or a short time and get a solution with a larger path cost.
3.2.3 Choosing states and actions

- What should go into the description of states and operators?
- Irrelevant details should be removed, this is called abstraction.
- You must ensure that you retain the validity of the problem when using abstraction.
3.3 Example Problems

- The problems are usually characterized into two types
  - Toy Problems
    - may require little or no ingenuity
    - games
  - Real World Problems
    - complex to solve
3.3.1 Toy Problems

Some examples of toy problems are:

- The 8-puzzle
- The 8-queens problem
- Cryptarithmetic
- The Vacuum World
- River-Crossing Puzzles
3.3.1.1 The 8-puzzle

The 8-puzzle is a small single board player game:

- Tiles are numbered 1 through 8 and one blank space on a 3 x 3 board.
- A 15-puzzle, using a 4 x 4 board, is commonly sold as a child's puzzle.
- Possible moves of the puzzle are made by sliding an adjacent tile into the position occupied by the blank space, this will exchanging the positions of the tile and blank space.
- Only tiles that are horizontally or vertically adjacent (not diagonally adjacent) may be moved into the blank space.
3.3.1.1 The 8-puzzle

- **Goal test:** have the tiles in ascending order.
- **Path cost:** each move is a cost of one.
- **States:** the location of the tiles + blank in the n x n matrix.
- **Operators:** blank moves left, right, up or down.
3.3.1.2 The 8-queens problem

- In this problem, we need to place eight queens on the chess board so that they do not check each other. This problem is probably as old as the chess game itself, and thus its origin is not known, but it is known that Gauss studied this problem.
The 8-queens problem (cont.)

- **Goal test**: eight queens on board, none attacked
- **Path cost**: zero.
- **States**: any arrangement of zero to eight queens on board.
- **Operators**: add a queen to any square
The 8-queens Solution

- If we want to find a single solution, it is not hard. If we want to find all possible solutions, the problem becomes increasingly difficult and the backtrack method is the only known method. For 8-queen, we have 96 solutions. If we exclude symmetry, there are 12 solutions.
3.3.1.3 Cryptarithmetic

- In 1924 Henry Dudeney published a popular number puzzle of the type known as a cryptarithm, in which letters are replaced with numbers.

- Dudeney's puzzle reads: SEND + MORE = MONEY.

- Cryptarithmetic are solved by deducing numerical values from the mathematical relationships indicated by the letter arrangements (i.e.) . S=9, E=5, N=6, M=1, O=0,….

- The only solution to Dudeney's problem: 9567 + 1085 = 10,652.

Goal test: puzzle contains only digits and represents a correct sum.

Path cost: zero. All solutions equally valid

States: a cryptarithmetic puzzle with some letters replaced by digits.

Operators: replace all occurrences of a letter with a digit not already appearing in the puzzle.
3.3.1.4 The Vacuum World

- Lets examine the Vacuum World.
- Assume, the agent knows where located and the dirt places
  - **Goal test:** no dirt left in any square
  - **Path cost:** each action costs one
  - **States:** one of the eight states
  - **Operators:** move left, move right, suck
3.3.1.4 The Vacuum World (cont.)
3.3.1.5 River-Crossing Puzzles

- A sequential-movement puzzle, first described by Alcuin in one of his 9th-century texts.
- The puzzle presents a farmer who has to transport a goat, a wolf, and some cabbages across a river in a boat that will only hold the farmer and one of the cargo items. In this scenario, the cabbages will be eaten by the goat, and the goat will be eaten by the wolf, if left together unattended.
- Solutions to river-crossing puzzles usually involve multiple trips with certain items brought back and forth between the riverbanks.
- Contributed by: Jerry Slocum B.S., M.S.
  - "Puzzle," Microsoft® Encarta® 97 Encyclopedia. © 1993-1996 Microsoft Corporation. All rights reserved.
3.3.1.5 River-Crossing Puzzles (cont.)

- **Goal test:** reached state \((0,0,0,0)\).
- **Path cost:** number of crossings made.
- **States:** a state is composed of four numbers representing the number of goats, wolfs, cabbages, boat trips. At the start state there is \((1,1,1,0)\).
- **Operators:** from each state the possible operators are: move wolf, move cabbages, move goat. In total, there are 3 operators.
3.3.2 Real World Problems

- Route finding
- Travelling salesman problems
- VLSI layout
- Robot navigation
- Assembly sequencing
3.3.2.1 Route finding

- Used in
  - computer networks
  - automated travel advisory systems
  - airline travel planning systems
    - path cost
    - money
    - seat quality
    - time of day
    - type of airplane
3.3.2.2 Travelling Salesman Problem (TSP)

- A salesman must visit N cities. Each city is visited exactly once and finishing the city started from. There is usually an integer cost $c(a,b)$ to travel from city $a$ to city $b$. However, the total tour cost must be minimum, where the total cost is the sum of the individual cost of each city visited in the tour.
- It’s an NP Complete problem
  - no one has found any really efficient way of solving them for large $n$.
- Closely related to the hamiltonian-cycle problem.
Travelling Salesman interactive

The goal of this problem is to find the shortest route for a set of cities. Each city must be visited once. The route must be a round trip. Click the 'New Game' button to start and then click on any city. After completing your round trip, click on the 'Solution' button to view the computer's route.
The goal of this problem is to find the shortest route for a set of cities. Each city must be visited once. The route must be a round trip. Click the 'New Game' button to start and then click on any city. After completing your round trip, click on the 'Solution' button to view the computer's route.
3.3.2.3 VLSI layout

- The decision of placement of silicon chips on bread boards is very complex.
- This includes
  - cell layout
  - channel routing
- The goal is to place the chips without overlap.
- Finding the best way to route the wires between the chips becomes a search problem.
3.4 Searching for Solutions

- Generating action sequences
- Data structures for search trees
3.4.1 Generating action sequences

• What do we know?
  – define a problem and recognize a solution

• Finding a solution is done by a search in the state space

• Maintain and extend a partial solution sequence
3.4.1 Generating action sequences (cont.)

- Start at initial state and check whether it’s a goal state
- Apply the operator (generate a new set of states-expending)
- The choice depends on the search strategy
3.4.1 Generating action sequences (cont.)

- A search tree is built over the state space
- A node corresponds to a state
- The initial state corresponds to the root of the tree
- Leaf nodes are expended
- There is a difference between state space and search tree
3.4.2 Data structures for search trees

- Data structure for a node
  - corresponding state of the node
  - parent node
  - the operator that was applied to generate the node
  - the depth of the tree at that node
  - path cost from the root to the node
3.4.2 Data structures for search trees (cont.)

• There is a distinction between a node and a state
• We need to represent the nodes that are about expended
• This set of nodes is called a fringe or frontier
  • MAKE-QUEUE(Elements) creates a queue with the given elements
  • Empty?(Queue) returns true only if there are no more elements in the queue.
  • REMOVE-QUEUE(Queue) removes the element at the front of the queue and return it
  • QUEUING-FN(Elements,Queue) inserts a set of elements into the queue. Different varieties of the queuing function produce different varieties of the search algorithm.
3.4.2 Data structures for search trees (cont.)

function GENERAL-SEARCH( problem, QUEUING-FN) returns a solution, or failure

    nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))

    loop do
        if nodes is empty then return failure
        node ← REMOVE-FRONT(nodes)
        if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
        nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
    end

Figure 3.10  The general search algorithm. (Note that QUEUING-FN is a variable whose value will be a function.)
3.5 Search Strategies

• We have to find the right search strategy, but how do we determine correctness?

• We evaluate it using 4 criteria:
  • **Completeness** … is an answer guaranteed?
  • **Time complexity** … can we go out for lunch while we search?
  • **Space Complexity** … how much memory is required?
  • **Optimality** … if we find a solution, is it an optimal one?
Two Categories of Search

Uniformed Search
- We can distinguish the goal state from the non-goal state, but the path and cost to find the goal is unknown.
- Also called known as blind search.

Informed Search
- We know something about the nature of our path that might increase the effectiveness of our search.
- Generally better than Uninformed search but will not be covered until later.
Six Uninformed Strategies

- We will cover six uninformed strategies and evaluate them according to the four criteria we mentioned earlier.
  - Breadth First Search
  - Uniform Cost Search
  - Depth First Search
  - Depth Limited Search
  - Iterative Deepening Search
  - Bi-directional Search
3.5.1 Breadth First Search (BFS)

- All nodes are expanded from the root, and then every successor is expanded and so on until we find the goal state.

Figure 3.11  Breadth-first search trees after 0, 1, 2, and 3 node expansions.
3.5.1 Breadth First Search (cont.)

- As you can see, BFS is:
  - Very systematic
  - Guaranteed to find a solution
- What does this mean? From the four criteria, it means:
  - BFS is complete. If there exists an answer, it will be found.
  - BFS is optimal. The path from the start state to goal state will be shallow.
3.5.1 Breadth First Search (cont.)

- What about time complexity and space complexity?
  - If we look at how BFS expands from the root we see that it first expands on a set number of nodes, say $b$.
  - On the second level it becomes $b^2$.
  - On the third level it becomes $b^3$.
  - And so on until it reaches $b^d$ for some depth $d$.

$$1 + b + b^2 + b^3 + \ldots + b^d \text{ which is } O(b^d)$$

- Since all leaf nodes need to be stored in memory, space complexity is the same as time complexity.
3.5.1 Breadth First Search (cont.)

- So in general, BFS tends to be a better choice of searching for smaller instances.

<table>
<thead>
<tr>
<th>Depth</th>
<th>Nodes</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1 millisecond</td>
<td>100 bytes</td>
</tr>
<tr>
<td>2</td>
<td>111</td>
<td>.1 seconds</td>
<td>11 kilobytes</td>
</tr>
<tr>
<td>4</td>
<td>11,111</td>
<td>11 seconds</td>
<td>1 megabyte</td>
</tr>
<tr>
<td>6</td>
<td>$10^6$</td>
<td>18 minutes</td>
<td>111 megabytes</td>
</tr>
<tr>
<td>8</td>
<td>$10^8$</td>
<td>31 hours</td>
<td>11 gigabytes</td>
</tr>
<tr>
<td>10</td>
<td>$10^{10}$</td>
<td>128 days</td>
<td>1 terabyte</td>
</tr>
<tr>
<td>12</td>
<td>$10^{12}$</td>
<td>35 years</td>
<td>111 terabytes</td>
</tr>
<tr>
<td>14</td>
<td>$10^{14}$</td>
<td>3500 years</td>
<td>11,111 terabytes</td>
</tr>
</tbody>
</table>

**Figure 3.12** Time and memory requirements for breadth-first search. The figures shown assume branching factor $b = 10$; 1000 nodes/second; 100 bytes/node.
3.5.2 Uniform Cost Search (UCS)

- Uniform Cost Search is a modification of BFS.
- BFS returns a solution, but it may not be optimal -- remember what we mentioned earlier?
3.5.2 Uniform Cost Search (cont.)

- UCS basically takes into account the cost of moving from one node to the next.
- This is probably best demonstrated with an example.
3.5.2 Uniform Cost Search (cont.)

Figure 3.13 A route-finding problem. (a) The state space, showing the cost for each operator. (b) Progression of the search. Each node is labelled with $g(n)$. At the next step, the goal node with $g = 10$ will be selected.
3.5.2 Uniform Cost Search (cont.)

- Evaluating UCS with the four criteria:
  - Complete? Yes. Same as BFS.
  - Optimal? Yes. More so than BFS, because the shortest path is guaranteed.
  - Time complexity? Still $O(b^d)$
  - Space complexity? Still $O(b^d)$
3.5.3 Depth First Search

- DFS expands one node to the deepest level until a dead end is reached and then traces back and expands on shallower nodes.
- This should come as nothing new.
3.5.3 Depth First Search (cont.)

Evaluation of DFS by four criteria:

😊 Good:
- Since we don’t expand all nodes at a level, space complexity is modest. For branching factor $b$ and depth $m$, we require $bm$ number of nodes to be stored in memory. This is much better than $b^d$.

- In some cases, DFS can be faster than BFS. However, the worse case is still $O(b^m)$.

😢 Bad:
- If you have deep search trees (or infinite – which is quite possible), DFS may end up running off to infinity and may not be able to recover.

- Thus DFS is neither optimal or complete.
3.5.4 Depth Limited Search (DLS)

- Modifies DFS to avoid its pitfalls.
  
  Say that within a given area, we had to find the shortest path to visit 10 cities. If we start at one of the cities, then there are at least nine other cities to visit. So nine is the limit we impose.

- Since we impose a limit, little changes from DFS with the exception that we will avoid searching an infinite path. DLS is now complete if the limit we impose is greater than or equal to the depth of our solution.
3.5.4 Depth Limited Search (cont.)

- So how is it’s time complexity and space complexity?
- DLS is $O(b^l)$ where $l$ is the limit we impose.
- Space complexity is $bl$.
- As you can see, space and time are almost identical to DFS.
- What about optimality? Nope.
3.5.5 Iterative Deepening Search

- In DLS, we chose an arbitrary limit. This was an obvious limit. Often it is the case that finding a “good” limit is not so obvious.

Suppose there are 20 cities in a given area. Now, starting at one city, find the shortest distance to some city G. We know that if we visit 20 or more cities then we’ve overlapped. Suppose, though, we learn that given any of the 20 cities, it takes at most nine steps to reach any other city. Would nine not be the better choice for the limit?
3.5.5 Iterative Deepening Search (IDS)

- Often we don’t know this good depth until we get there, but DLS determines this for us.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution sequence
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        if DEPTH-LIMITED-SEARCH(problem, depth) succeeds then return its result
    end
    return failure
```

Figure 3.15 The iterative deepening search algorithm.
3.5.5 Iterative Deepening Search (IDS)

Evaluating IDS:

- We look at the bottom most nodes once, while the level above twice, and the level above that thrice and so on until the root.
- Combining this factor with BFS, we get:

\[(d+1)1 + (d)b + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d\]

- Like DLS, IDS is still \(O(b^d)\) while space complexity is \(bd\).
3.5.6 Bi-directional search

- Idea is to simultaneously search from the start node out and from the goal outward. The search ends somewhere in the middle when the two touch each other.
- Time complexity: $O(2b^{d/2}) = O(b^{d/2})$
- Space complexity is the same because for the two searches to meet at some point, all nodes need to be stored in memory.
- It is complete and optimal.
3.5.6 Bi-directional search (cont.)

Problems:

- Do we know where the goal is?
- What if there is more than one possible goal state? We may be able to apply a multiple state search but this sounds a lot easier said than done. Example: How many checkmate states are there in chess?
- We may utilize many different methods of search but which one is choice?
## Making Comparisons

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breadth-First</th>
<th>Uniform-Cost</th>
<th>Depth-First</th>
<th>Depth-Limited</th>
<th>Iterative Deepening</th>
<th>Bidirectional (if applicable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>(b^d)</td>
<td>(b^d)</td>
<td>(b^m)</td>
<td>(b^l)</td>
<td>(b^d)</td>
<td>(b^{d/2})</td>
</tr>
<tr>
<td>Space</td>
<td>(b^d)</td>
<td>(b^d)</td>
<td>(bm)</td>
<td>(bl)</td>
<td>(bd)</td>
<td>(b^{d/2})</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Complete?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes, if (l \geq d)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Figure 3.18**  Evaluation of search strategies. \(b\) is the branching factor; \(d\) is the depth of solution; \(m\) is the maximum depth of the search tree; \(l\) is the depth limit.
3.6 Avoiding Repeated States

- Sometimes it is possible to expand to states/nodes that have already been visited. This may become costly and inefficient.
- There are three ways to deal with this problem and they are...
3 Ways to Deal with Repeat States

• Do not return to the state you just came from.
• Do not create paths with cycles in them.
• Do not generate any state that was ever generated before. Requires every state generated before to be kept in memory resulting in a huge space complexity $O(b^d)$.

Note: For the 3rd option, we can make use of hash tables to make look-up more efficient.
3.8 Summary

- problem-solving-agents
  - decide what to do by finding sequence of actions that lead to desirable states (this is known as searching)
- before searching it must express clearly a goal and use the goal to formulate a problem
- problems and solutions
- searching strategies
Resources and References Used

• "Puzzle," Microsoft(R) Encarta(R) 97 Encyclopedia. (c) 1993-1996 Microsoft Corporation.

• Travelling Salesman problem
  – applet by http://www.cacr.caltech.edu/~manu/tsp.html

• The 8-Queens problem
  – images by http://bridges.canterbury.ac.nz/features/eight.html