Search — Problem Solving as Search

Artificial Intelligence I — CS475/CS505
(Review and Updated)

August 21, 2008

1 Search

Search is an important part of our problem solving process. Practically, we search for a solution every time we try to solve a problem. Search is often needed when we do not have a step-by-step algorithm but we know what is a solution. Examples:

- *Travesing salesman* Given n cities, distance between every pair of two cities. A salesman needs to visit these cities, each at least one. Find for him a shortest route through the cities.
- *Knap-sack problem* Given n items, the weight and value of each item, and a knap-sack and its capacity. Find the most valuable way to pack the items into the knap-sack.
- *Navigation path* Find a path connecting the two points on a map for a robot.
- *SLD derivation* Given a goal $?g$, find a SLD derivation for g.

Definition 1.1 *Search is an enumeration of a set of potential partial solutions to a problem so that they can be checked to see if they truly are solutions, or could lead to a solution.*

To carry out a search, we need:

- A definition of a potential solution.
- A method of generating the potential solutions (hopefully in a clever way).
- A way to check whether a potential solution is a solution.

Example 1.1 *1. Search can be used to solve the n-queens problem with*

- *A* potential solution *in this problem is a $n \times n$-chess board with n-queens on it.*
- *Placing the queen one-by-one on the board is a method for generating the potential solutions (this is obviously not so clever!)*
- *If no two queens on the board attack each other, then the current potential solution is indeed a solution.*

2. Search can be used to solve the 8-puzzle problem with

- *A* potential solution *in this problem is a possible configuration of the 3×3-board with 8 digits and an empty cell.*
- *Exchanging the places of the empty cell and one of its neighbors allows us to generate potential solutions.*
- *If the current configuration of the board satisfies the final configuration then it is a solution.*

A search problem can be formally given by a tuple $\langle \Sigma, S_0, succ, G \rangle$ where

- Σ is the set of *states*, which represents the set of *all* potential solutions.
- $succ \subseteq \Sigma \times \Sigma$ is a binary relation over the set of states, which denotes a relationship between the potential solutions. Intuitively, $(s, u) \in succ$ means that the potential solution u can be constructed from s.
- S_0 is a set of states describing the set of *initial* states from which the search starts.
- G is a set of states desribing the set of *goal* states where the search can stop.

As we can see from the example, it is possible that S_0 (resp. G) contains more than one states. (see why?)

2 Graph Searching

Graph is used to present general mechanism of searching. To solve a problem using search, we translate it into a graph searching problem and use the graph searching algorithms to solve it.

Definition 2.1 *A graph consists of a set N of* nodes *and a set A of ordered pairs of nodes, called* arcs *(or* edges*).*

Two possible ways to represent a problem as a graph:

- *State-space graph*: each node represents a state of the world and an acr represents changing from one state to another.
- *Problem-space graph*: each node represents a problem to be solved and an arc represents alternate decomposition of the problems.

Example:

- *State-space graph*: finding path for robot – each node is a location. The state of the world is the location of the robot.
- *Problem-space graph*: SLD resolution – each node is a goal. Connection from one node to the other represents that the second one is obtained from the other through a SLD resolution. (**Note**: you might not know about SLD resolution yet; skip this one then.)

Node n_2 is a **neighbor** (or a **successor**) of n_1 if there is an arc from n_1 to n_2. That is, if $\langle n_1, n_2 \rangle \in A$. An arc may be labeled.

A **path** is a sequence of nodes $\langle n_0, n_1, ..., n_k \rangle$ such that $\langle n_{i+1}, n_i \rangle \in A$.

A **cycle** is a nonempty path such that the end node is the same as the start node. A graph with out cycle is called **directed acyclic graph** or DAG.

Given a set of start nodes and goal nodes, a solution is a path from a start node to a goal node.

The *forward branching factor* of a node is the number of arcs going out from the node, and he *backward branching factor* of a node is the number of arcs going into form the node.

3 A Generic Searching Algorithm

Given a graph, the set of start nodes, and the set of goal nodes. A path between a start node and a goal node is a solution. Searching algorithms provide us a way to find a solution.

Idea: Incrementally explore paths from start nodes. Maintaining a **frontier** or **fringe** of paths from the start nodes that have been explored.

The algorithm:

```
Input: a graph,
       a set of start nodes,
       Boolean procedure goal(n) that tests if n is a goal node.

frontier := {<s> : s is a start node};

while frontier is not empty:
       select and remove path <n0, . . . , nk> from frontier;
       if goal(nk)
            return <n0, . . . , nk>;
       for every neighbor n of nk
            add <n0, . . . , nk, n> to frontier;
end while
```

4 Uninformed (or Blind) Search Strategies

So far, we do not pay attention to the detail of how to select the next node when expand the frontier. The algorithm does not specify how they should be implemented.

Definition. A *search strategy* specifies which node should be selected at each iteration and how the frontier should be expanded.

Definition. A *blind search strategy* is a search strategy that does not take into account where the goal is.

- **Depth-First Search**: Completing the search of one path before exploring the other. Treats the frontier as a stack.
- **Breadth-First Search**: Always takes the path with fewest arcs to expand. Treats the frontier as a queue.

Space and Complexity of the Blind Search Strategies: Important factors in deciding which strategy to use.

Depth-First	Breadth-First
Might not find the solution	Guarantee to find a solution if one exists if branching factor is finite
Linear in size of the path being explored	Exponential time and space in size of the path being explored
Search is unconstrained until solution is found	Search is unconstrained by the goal

5 Variations of Breadth/Depth first search

Different variations for BFS and DFS have been proposed. Each has certain advantages but also has some disadvantages.

- **Lowest-Cost-First Search** also called **Uniform-Cost Search**: Need to have a function $c(n)$ that returns the cost of reaching a node n. Very often, this function is defined by associating some cost to the arcs of the graph and the cost between two nodes n and n' is defined by the sumation of all the costs of the arcs along the path. This strategy requires the expansion of the lowest cost path first. Treats the frontier as a priority queue. Lowest-cost strategy is similar to breath-first. It coincides with BFS if the cost of every arc is the same.
- **Depth-limited search**: Use DFS and limit the depth that will be exlored. The advantage of this approach is that it does not require exponential space. The disadvantage of this approach is that we often do not have enough information to select a depth that can be used as the effective limit.

- **Iterative deepening DFS** also called **IDS**: Use depth-limited search for depth = 0, 1, ..., until find the solution. Guarantees completeness and minimal solution. The disadvantage was the amount of nodes that need to be recomouted.
- **Bidirectional search**: a great idea but it is difficult to implement due to the fact that real-world problems do not provide an easy way for computing of the inverse function.

6 Comparing between Search Strategies

Criterion	BFS	Uniform Cost	DFS	Depth-Limited	Iterative Deepening	Bidirectional
Complete?	Yes^a	Yes^{a,b}	No	No	Yes^a	Yes^{a,d}
Time	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimal	Yes^c	Yes	No	No	Yes^c	Yes^{c,d}

In the above table:

- b is the forward branching factor.
- d is the depth of the shalowest solution.
- m is the maximum depth of the search tree.
- l is the depth limit.
- C^* is the cost of the optimal solution.
- Superscript meaning: ^{a} complete if b is finite; ^{b} complete if step costs $\geq \epsilon$ for positive ϵ; ^{c} optimal if step costs are all identical; ^{d} if both directions use breadth-first search.
- The book has $O(b^m)$ as the time needs for DFS. I suspect that they are using depth-limited search in that situation already.

7 Heuristic Search – Informed Search Strategies

Idea: Taking into account the goal information and (if available) knowledge about the goal. At any iteration, the "most promising" node – one, that probably leads to the goal – is selected to expand the frontier. Represent as a *heuristic function*, h, from the set of nodes into non-negative real numbers, i.e., for each node n, $h(n) \geq 0$.

$h(n)$ is *underestimate* if it is less than or equal the actual cost of the lowest-cost path from n to the goal.

Example: For the robot delivery, the straight-line distance between the node and the goal is a good heuristic function, which is underestimate.

Example: For the SLD search graph, the number of atoms in the query is a heuristic function.

- **Best-First Search**: Always select the element that appears to be the closest to the goal, i.e., lowest $h(n)$. Frontier is treated as priority queue.
- **Heuristic Depth-First Search**: Like depth-first, but use $h(n)$ in deciding what branch of the search tree to explore.

- **A* Search**: Selecting the next node based on the actual cost and the estimate cost. If the actual cost to the node n is $g(n)$ and the estimate cost from n to the goal is $h(n)$, the value $f(n) = g(n) + h(n)$ will be used in selecting the node to expand the frontier. This method is implemented by a priority queue based on $f(n)$.

Best-First	Depth-First	A*
Might not find the solution	Might not find the solution	Guarantee to find an optimal solution if one exists and branching factor is finite
Exponential time and space in size of the path being explored	Linear in size of the path being explored	Exponential time and space in size of the path being explored

8 Refirements to Search Strategies

Idea: Deals with cycles in the graph

- **Cycle checking**: Before inserting new paths into the frontier, check for their occurrence in the path. If the path selected to expand is $\langle n_0, \ldots, n_k \rangle$ and m is a neighbor of n_k we add to the frontier the path $\langle n_0, \ldots, n_k, m \rangle$ if m does not occur in $\langle n_0, \ldots, n_k \rangle$. This is
  - easy to implement in depth-first (one extra bit); set when visits; reset when backtracks;
  - need more time in exponential space strategies.
- **Multi-path Prunnning**: Before inserting new paths into the frontier, check for the occurrence of new neighbors n the frontier. Need to be done carefully if shortest/lowest cost path need to be found. In A*, *monotone restriction* is sufficient to guarantee that the shortest path to a node is the first path found to the node.

  monotone restriction: $|h(n') - h(n)| \leq d(n', n)$ where $d(n', n)$ is the actual cost from n' to n.

  Subsumes cycle checking. Preferred in strategies where the visited nodes are explicitely stored (breath-first); not prefered in depth-first searchs since the requirement of space required.
- **Iterative deepening**: Instead of storing the frontier, recompute it. Use depth-first to explore paths of 1, 2, 3, ... arcs until solution is found. When the search fails *unnaturally*, i.e., the depth bound is reached; in that case, restart with the new depth bound.
  - Linear space in size of the path being explored.
  - Little overhead in recomputing the frontier.

  Iterative deepening A*: Instead of using the number of arcs as the bound, use $f(n)$. Initially, $f(s)$ is used (s is the start node with minimal h-value). When the search fails unnaturally, the next bound is the minimal f-value that exceeded the previous bound.
- **Direction of Search**: forward (from start to goal), backward (from goal to start), bidirectional (both directions until meet). The main problem in bidirectional search is to ensure that the frontiers will meet (e.g. breath-first in one direction and depth-first in the other).
  - **Island-driven Search**: Limit the places where backward and forward search will meet (designated *islands* on the graph). Allows a decomposition of the problem in group of smaller problems. To find a path between s and g, identify the set of islands $i_0, \ldots, i_k$ and then find the path from s to i_0, from i_j to i_{j+1}, and finally from i_k to g.
  - **Searching in a Hierarchy of Abstractiosn**: Find solution at different level of abstraction. Details are added to the solution in refinement steps.

- **Dynamic Programming**: construct the perfect heuristic function that allows depth-first heuristic to find a solution without backtracking. The heuristic function represents the *exact costs* of a minimal path from each node to the goal. This will allow us to specify *which arcs to take* in each step, which is called a **policy**. Define

$$dist(n) = \begin{cases} 0 & if \ \ is\_goal(n) \\ min_{\langle n,m\rangle \in A}(|\langle n,m\rangle| + dist(m)) & otherwise \end{cases}$$

  where $dist(n)$ is the actual cost to the goal from n.

  $dist(n)$ can be computed backward from the goal to each node. It can then be stored and used in selecting the next node to visit.

  - $dist(n)$ depends on the goal;
  - $dist(n)$ can be pre-computed only when the graph is finite;
  - when $dist(n)$ is available, only linear space/time is needed to reach the goal;

9 Local Search Algorithms

Most of the previous algorithms need to store the complete search tree in the memory and return the path from the start node to the goal node as solution. In several problems, it is not very *important* to know *how to get to the goal*. For example,

- In the map coloring problem, we are interested in figuring out whether there is a way to color the map. It is not so important to know in which order the states/countries are colored;
- In the $n \times n$ queens problem, we are interested in the final configuration of the board, in which we know the location of each queen. It does not matter whether the queen on the first column is placed on the board in the last step or in the first step.
- There are several practical problems with this property: floor-planning, job-shop scheduling, etc.

For problems with the above characterization, local search algorithms can be useful. The key idea of local search algorithms is

- it keeps only one complete-state representation of the problem in the memory;
- it considers only *neighbors of the current state* in deciding which will be the next state (In this sense, it does not complete the *expand* phase of the previously studied algorithms);
- it uses an evaluation function to evaluate possible successors; the decision for which state should be consider next is made based using this function. This function is often referred to as *heuristic function*.

We discuss the above using the two examples:

- **Complete-state representation**:

  For the map coloring example, a local search algorithm will store the incomplete colored map (perhaps as a set of pairs of the form (S, C) where S is a state a C is a color or *nil*; the set must contain – for each state – one pair).

  For the n queens problem, a possible complete state representation consists of the location of the n-queens.

- **Neighbor**:

  For the map coloring example, a neighbor of a state can be any of the possible coloring which has one more state with a new color provided that this state is not colored before and shares border with some already colored states.

  For the n queens problem, a neighbor of a state is another configuration in which one of the queen moves to a new location (in the same row).

- **Evaluation function**:

  For the map coloring example, the evaluation function can be one that returns the number of states which have not been colored and share some border with some already colored states.

  For the n queens problem, the evaluation function can be one that returns the number of pairs of attacking queens.

There are a number of local search algorithms:

- **Hill-climbing algorithm** (also called **greedy local search**): the basic idea of this algorithm is to consider the *best* successor among all possible neighbors. The algorithm begins with the generation of a random initial state. It then goes into a loop which executes (a) evaluate the neighbors; (b) select a neighber for the next move (if the goal has not won yet. For example, if the current state S has the value $h(S) = 3$, then the next state S' has to have the value $h(S') > 3$. Please see the book for the example on the n queens problem.

  This algorithm is *incomplete* for different reasons: it can (i) stuck at a local maxima; (ii) stuck at a ridge; or (iii) stuck at a shoulder.

- **Random-restart hill climbing**: try to avoid the local maxima by randomly restart the search. This is done as follows: (a) timeout the algorithm after a certain limit; (b) regenerate the initial state. This method guarantess that the solution will be found if exists with the probability 1. This algorithm avoids the possible of the algorithm getting stucked at a local maxima.

- **Random sideways moves**: in this algorithm, not only the best neighbor but also a neighbor with the same value as the current state will be considered. This helps to avoid shoulders.

- **Simulated annealing**: this algorithm tries to avoid the local maxima and the shoulders by accepting some bad moves, with different probablility in the long run.

NOTE': All algorithms can be found in the book or on the slide.