Planning as Search – Review

CS 579

January 28, 2004

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 solutions.*

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.

2 Graph Searching

Use 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.*

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.

Node n_2 is a **neighbor** 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 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.
- **Lowest-Cost-First Search**: Need to have a function $c(n)$ that returns the cost of reaching a node n. This strategy requires the expansion of the lowest cost path first. Treats the frontier as a priority 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

Lowest-cost strategy: similar to breath-first.

5 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

6 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 if n_k we add to the frontier the path $\langle n_0, \ldots, n_k, m \rangle$ if m does not occur in $\langle n_0, \ldots, n_k \rangle$.
  - 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;

7 Planning as Search

A planning domain is specified by a set of fluents and a set of actions. Often, the set of fluents is implicitely given by the set of fluents used in representing the actions. We will assume that the actions will be given in ADL/STRIPS like notation as follow: $Action(a, Pre_a, Add_a, Delete_a)$.

Given a planning problem $P = \langle Act, I, G\rangle$ where

- Act – a set of actions
- I – a state
- G – a conjunction of fluent literals

A **fluent** is a time-dependent property. A **fluent literal** is either a fluent f or its negation $\neg f$. A **state** is a set of fluents. A set of fluents S is satisfied by a state s if $S \subseteq s$. For a set of fluent literals X, X^+ and X^- denote the set of positive and negative fluent literals belonging to X, respectively. X is satisfied by a state s if $X^+ \subset s$ and $s \cap X^- = \emptyset$.

P can be represented as a search problem for a state satisfying the goal, starting from the initial state, as follows.

- A potential solution is a state
- Actions change the state (computing potential solutions)
- Checking if the goal is satisfied is exactly the same as checking if the conjunction of goal is satissfied by a state.

The graph representing the search problem with

- the set of nodes is the set of states
- the set of arcs consists of pairs $\langle s_1, s_2\rangle$ such that there exists an action $a \in Act$ such that $Pre(a)$ is satisfied in s_1 and $s_2 = s_1 \cup Add_a \setminus Delete_a$. For simplicity, we will use $Progress(a, s)$ to denote $s \cup Add_a \setminus Delete_a$. If Pre_a is not satisfied by s then $Progress(s, a)$ is said to be undefined.

Example 7.1 *Let* $Act = \{a, b, c\}$ *where*
$Action(a, \{f, \neg g\}, \{g\}, \emptyset)$
$Action(b, \{f, \neg g\}, \emptyset, \{h, f\})$
$Action(c, \{h\}, \{g\}, \{f, h\})$

The set of fluents in this domain is $\{f, g, h\}$.

For action a, $Pre_a = \{f, \neg g\}$, $Add_a = \{g\}$, *and* $Delete_a = \emptyset$.

For action b, $Pre_b = \{f, \neg g\}$, $Add_b = \emptyset$, *and* $Delete_b = \{h, f\}$.

For action c, $Pre_c = \{h\}$, $Add_c = \{g\}$, *and* $Delete_c = \{h, f\}$.

Given a state $s = \{f\}$ *then* Pre_a *(and so is* Pre_b*) is satisfied by* s *but* Pre_c *is not.*

We can compute:

- $Progress(a, s) = \{f, g\}$.
- $Progress(b, s) = \emptyset$.
- $Progress(c, s)$ *is undefined.*