State space search

A lot of AI is about SEARCH.

Search can be applied to many problems:

- soln to a puzzle
- shortest path on a map
- proof of a theorem
- sequence of moves to win a game
- sequence of transformations to solve a symbolic integration problem
- sequence of moves to achieve a desired state
  (monkey & bananas, SHRDLU)

The "state space" is the set of states of the problem we can get to by applying *operators* to a state of the problem to get a new state.

The state space can be HUGE! (Combinatorial explosion)

Theorem Proving: Infinite!
Chess: 10^{120} (in an average length game)
Checkers: 10^{40}
Eight puzzle: 181,440

Right representation helps: mutilated checkerboard.

Control strategy helps choose which operators to apply:

Small # of operators: general, but bushy tree
Large #: perhaps overly specific, but less bushy trees

State space search

The search problem: How to control the search?

Given:

- an initial state
- a set of operators (actions the agent can take)
- a set of goal states
- (optional) a *path cost* function

Find:

a sequence of operators that leads from

the initial state to a goal state

The *search space* is the *implicit* tree (or graph)

defined by the initial state and the operators

The *search tree* (or graph) is the *explicit* tree

generated during the search by the control strategy

NOTE: *Each node* of this graph is a complete description of the state of the problem.

State Space Search

Measuring effectiveness:

1. Does the method find a solution at all?

2. Is it a good solution (low **path cost**)

3. What is the **search cost**?

The **total cost** is the sum of the search cost and the path cost (which may not be commensurate!)

State Space Search

State space search is an example of a *weak method*.

A *weak method* is:

1. a problem-independent *framework* for solving problems

2. It may have "stubs" for incorporating domain knowledge

However: weak methods usually cannot overcome the

combinatorial explosion.

Representation

It is important when representing a problem
to *abstract away* from the details.

E.g., for a search problem such as:

how to get to San Francisco by the shortest path
given a map:

Don’t need to represent *all* the aspects
of the trip!

E.g., the states may be:

In(San Diego)
In(Los Angeles)

and actions: go(X,Y)

and goal: In(San Francisco)

(doesn’t matter

what we had for breakfast,
what kind of car we’re driving,
how many passengers we have,
what the weather is like...

most of the time!)

Toy Problems versus Real-world problems

Most of the problems we will work on will be *toy problems*:

- Puzzle-solving (8, 15-puzzle)
- 8-queens
- Cryptarithmetic

But in fact, these techniques are used to solve *Real problems*:

- Route-finding in airline travel planners
- Travelling Salesperson Problem
  (planning drill movements for automatic circuit board drills)
- VLSI layout (cell layout and channel routing)
- Automatic manufacturing (assembly sequencing)

The search tree

- Starting from the start state, generating successors is called *expanding* the node.

- We attach these into the *search tree*

- Then we have to decide *which node to expand next*

  How we make this decision is the essence of the *search strategy*

General Tree search algorithm
(OBVIOUSLY NOT IN PURE LISP!!!)

(ignores bookkeeping for graphs and keeping track of the path):

```
/* Initialize */

1. OPEN = {Start_node}; CLOSED = NIL;

While TRUE do

    2. If empty(OPEN) then return(*FAIL*);

    /* Expand next node */

    3. CURRENT = car(OPEN);

    4. CLOSED = cons(CURRENT, CLOSED);

    5. If Goal(CURRENT) Then Return(PATH)

    /* apply operators to get a list of successors */

    6. SUCCESSORS = Expand(CURRENT);

    /* Heuristics go here, in how things are inserted */

    7. OPEN = Insert(SUCCESSORS, OPEN);
od
```

Search strategies

These will (almost) all vary depending on how things are inserted in the OPEN list.

Divide them based on *informedness*

Uninformed: Make no use of domain knowledge

(also known as *blind search*

Breadth-first search, Depth first search, ...

Informed: Make use of heuristics

Hill climbing (gradient descent)

Best-first search: "Algorithm A" and A*

IDA*

Evaluating Search strategies

Four criteria:

1. Completeness: Are they *guaranteed* to find a solution?
2. Time complexity: How long do they take in search time?
3. Space complexity: Storage required
4. Optimality: When there are several solutions, does it find the best one?

Six Blind Search algorithms

1. Breadth First Search

2. Uniform cost search

3. Depth first search

4. Depth-limited search

5. Iterative deepening

6. Bi-directional search

Breadth First search

always puts new nodes on the *end* of the OPEN list

Is it complete?

time & space complexity?

optimal?

Uniform cost search

Just like BFS, but uses the *path cost* of a node to order it on the OPEN list

For example, in the "find-a-route" problem, BFS will return the path through the fewest cities.

UCS will return the shortest path.

Nodes are ordered on OPEN in terms of $g(n)$ - the cost in the graph so far.

Depth first search

always puts new nodes on the *front* of the OPEN list

Is it complete?

time & space complexity?

optimal?

Depth-limited DF search

always puts new nodes on the *front* of the OPEN list

But stops after some *depth limit* is reached.

Is it complete?

time & space complexity?

optimal?

Iterative Deepening Search

Simple idea: Use Depth-bounded DFS, but keep increasing the depth bound until the answer is found.

Is it complete?

time & space complexity?

optimal?

Iterative Deepening

The time complexity is much less than you might expect!

Recall: The number of expansions to a depth of d with branching factor b is:

$$1 + b + b^2 + \cdots + b^{d-2} + b^{d-1} + b^d$$

For b=10 and d=5 this is 111,111

For each expansion to depth k, the root has been expanded $k + 1$ times, the next level k times, etc., giving:

$$(d+1)1 + (d)b + (d-1)b^2 + \cdots + 3b^{d-2} + 2b^{d-1} + b^d$$

For b=10 and d=5 this is 123,456

Bi-directional Search

Idea: Search forwards from initial state

and...backwards from goal!

Check if they link up!

Why is this good?

What could be costly?

Should search be the same kind from both ends?

Is it complete?

time & space complexity?

optimal?

Summary

Criterion	Time	Space	Optimal?	Complete?
Breadth First	b^d	b^d	**Yes**	**Yes**
Uniform Cost	b^d	b^d	**Yes**	**Yes**
Depth First	b^d	bm	**No**	**No**
Depth-limited(l)	b^l	bl	**No**	**If** $l \geq d$
Iter. Deepening	b^d	bd	**Yes**	**Yes**
Bidirectional	$b^{d/2}$	$b^{d/2}$	**Yes**	**Yes**

b = branching factor, d = depth of solution
m = max depth of search tree, l = depth limit

Non-redundant generators

Is the state space graph a tree or a graph with loops?

A *loop* means

you can visit the same state over and over

Non-redundant generators (3 levels of difficulty)

1. Never generate the state you just came from

2. Do not create paths with cycles (check if a node is already on the current path

3. Check if a node has ever been generated before. (Tricky for find a route problems)

Note:

Tradeoff between checking (may be expensive)

and not checking (generate a bigger search tree)