5 Automated problem solving

Last time: represented problem as a CSP

This time: represent problem as a state space search problem

5.1 State space search problem

Given:

- initial state s_0
- set of possible actions: $a_1, \ldots, a_k$ where $a_i : s \mapsto s'$
- goal test

Goal: find a sequence of actions that transforms initial state into a goal state.

(Thus, the search space is an implicit graph generated by objects and operators on objects.)

5.2 Generalizes CSPs

- state = partial assignment
- action = assign value to an unassigned variable
- initial state = empty assignment
- goal test = conjunction of constraints

But, now solutions may not have a bounded length.

5.3 Examples

Water jugs problem 1

Given a 3 liter jug and a 4 liter jug, where jugs can be filled with water, emptied, or water can be poured from one jug into another until either the source jug is empty or the destination jug is full. Consider a problem where the jugs are initially empty and the goal is to achieve 2 gallons in the 4 gallon jug.

Solution: Systematically expand a search tree for the problem to find the solution as follows

00

30 04

...

34 ~~00~~ 03

~~04~~ ~~20~~ 33 ~~30~~ ~~04~~ ~~00~~

24

20

02

Water jugs problem 2

Given three jugs of capacity 2, 5, and 7 gallons respectively. Initially the 7 gallon jug is full and the other jugs are emtpy. The goal is to achieve 1 gallon of water in some jug, without spilling any water, and using no external sources of water.

Solution: (Ignoring repeated states, 007 is crossed out for the sake of illustration.)

007

205 052

025 ~~007~~ 250

223

043

241

In general, the *search graph* for this problem is given as follows

Missionaries and cannibals problem

3 missionaries and 3 cannibals want to cross a river using a boat that holds 2 people. Cannibals can never outnumber missionaries, and an empty boat

Other examples:

- towers of Hanoi
- Rubik's cube

cannot cross the river.

- 8 puzzle

Initial state: MMMCCCB|

5.4 Automating problem solving search: Graph search

Goal state: |BMMMCCC

Automated problem solving search is graph search:

Action: take 1 or 2 people across the river

states = vertices

actions = labeled edges

Solution:

General graph search strategies

5.5 Depth-First Search(DFS)

Algorithm 1 Depth-First Search(DFS)

```
1: list ← {s_0}
2: while list is not empty do
3:    s ← head(list)
4:    list ← rest(list)
5:    if s is a goal then
6:       return s
7:    else
8:       newstates ← apply actions to s
9:       list ← prepend(newstates, list)
10:   end if
11: end while
12: return fail
```

Problem: - graph may be infinite, or have cycles

5.6 Breadth-first search (BFS)

Same as DFS, except:

```
9: list ← append(list, newstates)
```

5.7 DFS versus BFS

BFS:

- guaranteed to find solution (if one exists)
- not space efficient $|b|^{\text{solution depth}}$, where b is the branching factor

DFS:

- space efficient
- not guaranteed

How to be space efficient and guaranteed?

5.8 Iterative deepening search (IDS)

Space efficient BFS

Algorithm 2 Iterative deepening search (IDS)

```
for depth bound = 1, 2, ... do
  list ← {s_0}
  while list is not empty do
    s ← head(list)
    list ← rest(list)
    if s is a goal then
      return s
    else if depth(s) < depth bound then
      newstates ← apply actions to s
      list ← prepend(newstates, list)
    end if
  end while
end for
```

- Same space as DFS
- Guaranteed to find solution like BFS

- Almost the same time as BFS:

$$\text{BFS running time} = b^d \quad \text{where } b \text{ is the branching factor (i.e., the number of actions per state)}$$

$$\text{IDS running time} = 1 + b + b^2 + \ldots + b^d = \frac{b^{d+1} - 1}{b - 1} \approx b^d$$

5.9 Speedups

Pruning:

- May determine that goal constraints are already violated
- Do not revisit states!
  Put states in a hash table; check if visited already

Heuristics:

- Use heuristic function $\hat{h}(s)$ that estimates distance from s to goal

5.10 Best first search

Same as DFS, except that lines 3 and 4 are replaced by:

3: s $\leftarrow$ extract_best_$\hat{h}$_value(list)

- Allows big speedups
  e.g., if $\hat{h}$ is perfect, walk straight to the goal
- Problem: can be space inefficient

Note For DFS the appropriate data structure for the list is a stack, for BFS it is a FIFO queue, and for best first search it is a heap (i.e., a priority queue).

5.11 Harder problem: Finding *shortest* solution

Constrained optimization task

Definition: Algorithm that is guaranteed to find shortest solutions is called *admissible*

BFS, IDS	admissible
DFS, best first	not admissible

How to make best first admissible?

Let:

$g(s)$ = shortest distance from s_0 to s

$\hat{g}(s)$ = shortest distance from s_0 to s
(that we know at certain moment during algorithm execution)

$h(s)$ = shortest distance from s to a goal

$\hat{h}(s)$ = our heuristic function, which approximates h

$d(s)$ = shortest distance from s_0 to a goal through s
i.e., $d(s) = g(s) + h(s)$

$\hat{d}(s)$ = our approximation of d
i.e., $\hat{d}(s) = \hat{g}(s) + \hat{h}(s)$

Definition An *admissible heuristic* is a heuristic function $\hat{h}(s)$ that underestimates $h(s)$. That is, $\hat{h}(s) \leq h(s)$, and it does not lie about the goal, i.e., $\hat{h}(s) = 0$ iff s is goal, $\hat{h}(s) > 0$ otherwise.

5.12 Admissible best first search (A*)

Uses an admissible heuristic, and it is same as best first search, except:

3: s ← extract_min_$\hat{d}$_value(list)

5.13 Proof that A* finds shortest path

Proof
Let k be the length of the optimal solution, and let $s_1^*, \ldots, s_k^*$ be an optimal solution path (where $s_1^* = s_0$).
Assume the algorithm finds a non-optimal solution $s_1, s_2, \ldots, s_\ell$ for $\ell > k$. (Note that $\hat{g}(s_j) = j$ for all $1 \leq j \leq \ell$, or else the algorithm would have found a shorter path.)
Now consider the time when $s_{\ell-1}$ is on the list. In this case we must have

$$\begin{aligned} \hat{d}(s_{\ell-1}) &= \hat{g}(s_{\ell-1}) + \hat{h}(s_{\ell-1}) \\ &\geq \ell - 1 + 1 = \ell. \end{aligned}$$

Claim 1 The algorithm must expand s_1^* before $s_{\ell-1}$.
Pf: Expanding s_0 immediately puts s_1^* on the list, therefore

$$\begin{aligned} \hat{d}(s_1^*) &= \hat{g}(s_1^*) + \hat{h}(s_1^*) \\ &= 1 + \hat{h}(s_1^*) \\ &\leq 1 + k - 1 \;=\; k \;<\; \ell. \end{aligned}$$

Claim 2 For any s_{i-1}^*, if s_{i-1}^* is expanded with $\hat{g}(s_{i-1}^*) = i - 1$, then s_i^* must be expanded before $s_{\ell-1}$.
Pf: Expanding s_{i-1}^* puts s_i^* on the list with $\hat{g}(s_i^*) = i$, therefore

$$\begin{aligned} \hat{d}(s_i^*) &= \hat{g}(s_i^*) + \hat{h}(s_i^*) \\ &\leq i + k - i \;=\; k \;<\; \ell. \end{aligned}$$

This implies that $s_1^*, s_2^*, ...$ must all be expanded before $s_{\ell-1}$. ∎

5.14 Generalized A*

The distance does not have to be expressed as the number of states expanded. One can associate a weight $w(a)$ (distance or cost) with each action a and define the distance as the sum of those weights along a path. (Note $w(a) = 1$ for the previous version of A*.) The problem is to find a shortest path in terms of this distance. The basic algorithm is the same, and the proof of optimality is almost the same, with a couple of details changed.

5.15 Iterative deepening A*

Problem: A^* is space inefficient

IDA*: Iterative deepening on $\hat{d}$ bound.

- Guaranteed
- Admissible
- Space efficient
- Time efficient? (depends on $\hat{h}$)

Algorithm 3 Iterative deepening A* search (IDA*)

```
d̂_limit ← d̂(s_0)
while d̂_limit < ∞ do
   next_d̂_limit ← ∞
   list ← {s_0}
   while list is not empty do
      s ← head(list)
      list ← rest(list)
      if d̂(s) > d̂_limit then
         next_d̂_limit ← min(next_d̂_limit, d̂(s))
      else
         if s is a goal then
            return s
         end if
         newstates ← apply actions to s
         list ← prepend(newstates, list)
      end if
   end while
   d̂_limit ← next_d̂_limit
end while
return fail
```

5.16 Incomplete search

- beam search
- genetic algorithm

5.17 Readings

Russell and Norvig 2nd Ed.: Sect 3.2-3.7, 4.1-4.2
Dean, Allen, Aloimonos: Sect 4.1–4.3