Overview

- Search problems: definition
- Example: 8-puzzle
- General search
- Evaluation of search strategies
- Strategies: breadth-first, uniform-cost, depth-first

Search Problems: Definition

\[ \text{Search} = \langle \text{initial state, operators, goal states} \rangle \]

- Initial State: description of the current situation as given in a problem
- Operators: functions from any state to a set of successor (or neighbor) states
- Goal: subset of states, or test rule

Variants of Search Problems

\[ \text{Search} = \langle \text{state space, initial state, operators, goal states} \rangle \]

- State space: set of all possible states reachable from the current initial state through repeated application of the operators (i.e. path).

\[ \text{Search} = \langle \text{initial state, operators, goal states, path cost} \rangle \]

- Path cost: find the best solution, not just a solution. Cost can be many different things.

Types of Search

- Uninformed: systematic strategies (Chapter 3)
- Informed: Use domain knowledge to narrow search (Chapter 4)
- Game playing as search: minimax, state pruning, probabilistic games (Chapter 5).
Search State

State as Data Structure

- examples: variable assignment, properties, order in list, bitmap, graph (vertex and edges)
- captures all possible ways world could be
- typically static, discrete (symbolic), but does not have to be

Choosing a Good Representation

- concise (keep only the relevant features)
- explicit (easy to compute when needed)
- embeds constraints

Goals: Subset of states or test rules

Specification:

- set of states: enumerate the eligible states
- partial description: e.g. a certain variable has value over x
- constraints: or set of constraints. Hard to enumerate all states matching the constraints, or very hard to come up with a solution at all (i.e. you can only verify it; P vs. NP).

Other considerations:

- space, time, quality (exact vs. approximate tradeoffs)

Operators

Function from state to subset of states

- drive to neighboring city
- place piece on chess board
- add person to meeting schedule
- slide tile in 8-puzzle

Characteristics

- often requires instantiation (fill in variables)
- encode constraints (only certain operations are allowed)
- generally discrete: continuous parameters \( \rightarrow \) infinite branching

An Example: 8-Puzzle

- State: location of 8 number tiles and one blank tile
- Operators: blank moves left, right, up, or down
- Goal test: state matches the configuration on the right (see above)
- Path cost: each step cost 1, i.e. path length, or search tree depth

Generalization: 15-puzzle, ..., \((N^2 - 1)\)-puzzle
8-Puzzle: Example

Possible state representations in LISP (0 is the blank):
- \((0 \ 2 \ 3 \ 1 \ 8 \ 4 \ 7 \ 6 \ 5)\)
- \(((0 \ 2 \ 3) \ (1 \ 8 \ 4) \ (7 \ 6 \ 5))\)
- \(((0 \ 1 \ 7) \ (2 \ 8 \ 6) \ (3 \ 4 \ 5))\)
- or use the make-array, aref functions.

How easy to: (1) compare, (2) operate on, and (3) store (i.e. size).

Goal Test

As simple as a single LISP call:

```lisp
(defun *goal-state* '(1 2 3 8 0 4 7 6 5))
*(GOAL-STATE)*

(equal *goal-state* '(1 2 3 8 0 4 7 6 5)) T
```

General Search Algorithm

Pseudo-code:

```lisp
(defun General-Search (problem, Que-Fn)
  node-list := initial-state
  loop begin
    // fail if node-list is empty
    if Empty(node-list) then return FAIL
    // pick a node from node-list
    node := Get-First-Node(node-list)
    // if picked node is a goal node, success!
    if (node == goal) then return as SOLUTION
    // otherwise, expand node and enqueue
    node-list := Que-Fn(node-list, Expand(node))
  loop end
```

(8-Puzzle: Search Tree)

Goal Test

As simple as a single LISP call:

```lisp
(defun *goal-state* '(1 2 3 8 0 4 7 6 5))
*(GOAL-STATE)*

(equal *goal-state* '(1 2 3 8 0 4 7 6 5)) T
```
Evaluation of Search Strategies

- time-complexity: how many nodes expanded so far?
- space-complexity: how many nodes must be stored in node-list at any given time?
- completeness: if solution exists, guaranteed to be found?
- optimality: guaranteed to find the best solution?

Breadth First Search

1. node visit order (goal test): 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
2. queuing function: enqueue at end (add expanded node at the end of the list)

BFS: Expand Order

1. [1] : initial state
2. [2][3] : dequeue 1 and enqueue 2 and 3
3. [3][4][5] : dequeue 2 and enqueue 4 and 5
4. [4][5][6][7] : all depth 3 nodes
5. 8. [8][9][10][11][12][13][14][15] : all depth 4 nodes

Evolution of the queue (bold = expanded and added children):
1. [1] : initial state
2. [2][3] : dequeue 1 and enqueue 2 and 3
3. [3][4][5] : dequeue 2 and enqueue 4 and 5
4. [4][5][6][7] : all depth 3 nodes
5. 8. [8][9][10][11][12][13][14][15] : all depth 4 nodes

BFS: Evaluation

branching factor $b$, depth of solutions $d$:
1. complete: it will find the solution if it exists
2. time: $1 + b + b^2 + \ldots + b^d$
3. space: $b^k$ where $k$ is the current depth
4. space is more problem than time in most cases (p 75, figure 3.12).
5. time is also a major problem nonetheless
Uniform Cost

BFS with expansion of lowest-cost nodes: path cost is $g(n) = \text{Depth}(node)$.

- BFS: $g(n) = \text{Depth}(node)$

Depth First Search

- node visit order (goal test): 1 2 4 8 9 5 10 11 3 6 12 13 7 14 15
- queuing function: enqueue at left (stack push; add expanded node at the beginning of the list)

DFS: Expand Order

Evolution of the queue (bold = expanded and added children):
1. [1] : initial state
2. [2][3] : pop 1 and push expanded in the front
3. [4][5][3] : pop 2 and push expanded in the front
4. [8][9][5][3] : pop 4 and push expanded in the front

DFS: Evaluation

branching factor $b$, depth of solutions $d$, max depth $m$:

- incomplete: may wander down the wrong path
- time: $b^m$ nodes expanded (worst case)
- space: $bm$ (just along the current path)
- good when there are many shallow goals
- bad for deep or infinite depth state space
Implementation

- Use of stack or queue: explicit storage of expanded nodes
- Recursion: implicit storage in the recursive call stack

Key Points

- Description of a search problem: initial state, goals, operators, etc.
- Considerations in designing a representation for a state
- Evaluation criteria
- BFS, UCS, DFS: time and space complexity, completeness
- Differences and similarities between BFS and UCS
- When to use one vs. another
- Node visit orders for each strategy
- Tracking the stack or queue at any moment

Next Time

- More uninformed search
- Comparison of search strategies.
- Repeated states: duplicate detection
- Constraint satisfaction search
- Informed search intro (time permitting)