Search problem: example 1

Example:

Start State

<table>
<thead>
<tr>
<th>7</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Goal State

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

States:

- integer locations of tiles (ignore intermediate positions)

Actions:

- move blank left, right, up, down (ignore unjamming etc.)

Goal test:

= goal state (given)

Path cost:

1 per move

[Note: optimal solution of n-Puzzle family is NP-hard]
Search problem: example 2

On holiday in Romania; currently in Arad. Flight leaves tomorrow from Bucharest

Formulate goal:
be in Bucharest

Formulate problem:
states: various cities
actions: drive between cities

Find solution:
sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest
Search problems

A search problem is defined by 5 components:

- **initial state**

- **possible actions** (and state associated actions)

- **transition model**
  taking an action will cause a state change

- **goal test**
  judge if the goal state is found

- **path cost**
  constitute the cost of a solution
Problems

Example:

Romania
Giurgiu
Urziceni
Hirsova
Eforie
Neamt
Oradea
Zerind
Arad
Timisoara
Lugoj
Mehadia
Dobreta
Craiova
Sibiu
Fagaras
Pitesti
Vaslui
Iasi
Rimnicu Vilcea
Bucharest

71
75
118
70
75
120
111
80
99
881
151
140
151
70
99
97
101
120
146
138
85
87
90
92
98
142
86
87
90
92
98
142
86

Chapter
3

state
transition
cost

initial
state
actions

goal
Problems

initial state e.g., “at Arad”

successor function $S(x) = \text{set of action–state pairs}$
  e.g., $S(\text{Arad}) = \{ \langle \text{Arad} \rightarrow \text{Zerind}, \text{Zerind} \rangle, \ldots \}$

goal test, can be
  explicit, e.g., $x = \text{“at Bucharest”}$
  implicit, e.g., $\text{NoDirt}(x)$

path cost (additive)
  e.g., sum of distances, number of actions executed, etc.
  $c(x, a, y)$ is the step cost, assumed to be $\geq 0$

A solution is a sequence of actions
leading from the initial state to a goal state
Problems

we assume

observable states (a seen state is accurate)
   in partial observable case, states are not accurate

discrete states
   there are also continuous state spaces

deterministic transition
   there could be stochastic transitions
Example: vacuum world

**states**: integer dirt and robot locations (ignore dirt *amounts* etc.)

**actions**: *Left*, *Right*, *Suck*, *NoOp*

**goal test**: no dirt

**path cost**: 1 per action (0 for *NoOp*)
Example: 8-puzzle

**States??**: integer locations of tiles (ignore intermediate positions)

**Actions??**: move blank left, right, up, down (ignore unjamming etc.)

**Goal test??**: = goal state (given)

**Path cost??**: 1 per move

[Note: optimal solution of \(n\)-Puzzle family is NP-hard]
Agent that searches

can simple reflex agents do the search?

Agent

Sensors

What the world is like now

Condition–action rules

What action I should do now

Actuators

Environment
Agent that searches

can reflex agents with state do the search?

Agent

Environment

State

How the world evolves

What the world is like now

What my actions do

Condition–action rules

What action I should do now

Sensors

Actuators

Chapter 2

Example:

Romania

Giurgiu

Urziceni

Hirsova

Eforie

Neamt

Oradea

Zerind

Timisoara

Arad

Sibiu

Fagaras

Rimnicu Vilcea

Pitesti

Vaslui

Iasi

Rimnicu Vilcea

Bucharest

Oradea

Zerind

Timisoara

Arad

Sibiu

Fagaras

Rimnicu Vilcea

Pitesti

Vaslui

Iasi

Rimnicu Vilcea

Bucharest

Oradea

Zerind

Timisoara

Arad

Sibiu

Fagaras

Rimnicu Vilcea

Pitesti

Vaslui

Iasi

Rimnicu Vilcea

Bucharest
consider goal-based agents
Agent that searches

consider goal-based agents

possible movements

predefined goal

State

How the world evolves

What the world is like now

What my actions do

What it will be like if I do action A

Goals

What action I should do now

Actuators

Sensors

Environment

Chapter 2

Example: The 8-puzzle

Start State

Goal State

states: ???

actions: ???

goal test: ???

path cost: ???

[Note: optimal solution of n-Puzzle family is NP-hard]

Chapter 3

Example: Romania

possible movements

predefined goal
Agent that searches

transition model by world rules

possible movements

predefined goal
Extra knowledge

time complexity: number of key operations
space complexity: number of key bits stored

the big O representation:

\[ O(1) \quad O(\ln n) \quad O(n) \quad O(n^2) \]

\[ O(2^n) \quad O(n^n) \]

NP-hardness and NP-completeness

[from wikipedia: “Big O”]
Search Algorithms on Graphs
binary tree: each node has at most two branches

search tree: a tree data structure for search
A **state** is a (representation of) a physical configuration.

A **node** is a data structure constituting part of a search tree, which includes parent, children, depth, path cost $g(x)$.

States do not have parents, children, depth, or path cost!

The **Expand** function creates new nodes, filling in the various fields and using the **SuccessorFn** of the problem to create the corresponding states.
Agent that searches

- **What the world is like now**
- **What my actions do**
- **What it will be like if I do action A**
- **What action I should do now**
- **How the world evolves**
- **Goals**

The search does NOT change the world! Only actions change the world.

Evolves in a tree structure: use tree search to find the goal.
Tree search

1. start from the initial state (root)
2. expand the current state

essence of search: following up one option now and putting the others aside

function Tree-Search( problem, strategy) returns a solution, or failure
initialize the search tree using the initial state of problem
loop do
  if there are no candidates for expansion then return failure
  choose a leaf node for expansion according to strategy
  if the node contains a goal state then return the corresponding solution
  else expand the node and add the resulting nodes to the search tree
end

all search algorithms share this tree search structure
they vary primarily according to how they choose which state to expand --- the so-called search strategy
Storage data structure

**Stack**

- **Insert:** Push
- **Remove:** Pop

**Queue**

- **Insert:** Enqueue (Back)
- **Remove:** Dequeue (Front)

(images from https://stackoverflow.com/questions/10974922/what-is-the-basic-difference-between-stack-and-queue)
General tree search

function Tree-Search(problem, fringe) returns a solution, or failure
    fringe ← Insert(Make-Node(Initial-State[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← Remove-Front(fringe)
        if Goal-Test(problem, State(node)) then return node
        fringe ← InsertAll(Expand(node, problem), fringe)
    end loop

function Expand(node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in Successor-Fn(problem, State[node]) do
        s ← a new Node
        Parent-Node[s] ← node; Action[s] ← action; State[s] ← result
        Path-Cost[s] ← Path-Cost[node] + Step-Cost(node, action, s)
        Depth[s] ← Depth[node] + 1
        add s to successors
    end for
    return successors

note the time of goal-test: expanding time not generating time
Example
Example
Example
function **Tree-Search** (problem, fringe) returns a solution, or failure

fringe ← Insert(Make-Node(Initial-State[problem]), fringe)

loop do
  if fringe is empty then return failure
  node ← Remove-Front(fringe)
  if Goal-Test(problem, State[node]) then return node
  fringe ← InsertAll(Expand(node, problem), fringe)

function **Graph-Search** (problem, fringe) returns a solution, or failure

**closed** ← an empty set

fringe ← Insert(Make-Node(Initial-State[problem]), fringe)

loop do
  if fringe is empty then return failure
  node ← Remove-Front(fringe)
  if Goal-Test(problem, State[node]) then return node
  if State[node] is not in **closed** then
    add State[node] to **closed**
    fringe ← InsertAll(Expand(node, problem), fringe)
  end

end
Figure 3.8 shows the sequence of search trees generated by a graph search on the Roman problem of Figure 3.2. At each stage, we have extended each path by one step. Notice that at the third stage, the northernmost city (Oradea) has become a dead end: both of its successors are already explored via other paths.

Figure 3.9 illustrates the separation property of GraphSearch, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been expanded. In (c), the remaining successors of the root have been expanded in clockwise order.

The initial state to an unexplored state has to pass through a state in the frontier. (If this seems completely obvious, try Exercise 3.13 now.) This property is illustrated in Figure 3.9. As every step moves a state from the frontier into the explored region while moving some states from the unexplored region into the frontier, we see that the algorithm is systematically examining the states in the state space, one by one, until it finds a solution.
Search strategies

A strategy is defined by picking the order of node expansion

Strategies are evaluated along the following dimensions:
- **completeness**—does it always find a solution if one exists?
- **time complexity**—number of nodes generated/expanded
- **space complexity**—maximum number of nodes in memory
- **optimality**—does it always find a least-cost solution?

Time and space complexity are measured in terms of
- \( b \)—maximum branching factor of the search tree
- \( d \)—depth of the least-cost solution
- \( m \)—maximum depth of the state space (may be \( \infty \))
Uninformed search strategies use only the information available in the problem definition.

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search
Breadth-first search

Expand shallowest unexpanded node

**Implementation:**

*fringe* is a FIFO queue, i.e., new successors go at end

![Tree Diagram]

- A
- B
- C
- D
- E
- F
- G

Chapter 33
Breadth-first search

Expand shallowest unexpanded node

**Implementation:**

*fringe* is a FIFO queue, i.e., new successors go at end
Breadth-first search

Expand shallowest unexpanded node

**Implementation:**

*fringe* is a FIFO queue, i.e., new successors go at end
Properties

**Complete**? Yes (if $b$ is finite)

**Time**? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in $d$

**Space**? $O(b^{d+1})$ (keeps every node in memory)

**Optimal**? Yes (if cost = 1 per step); not optimal in general

**Space** is the big problem; can easily generate nodes at 100MB/sec so 24hrs = 8640GB.
Depth-first search

Expand deepest unexpanded node

Implementation:

-fringe = LIFO queue, i.e., put successors at front

Diagram:

- Tree structure with nodes A, B, C, D, E, F, G, H, I, J, K, L, M, N, O.
- Starting at node A, explore paths downwards and to the left.
- When encountering a leaf node, backtrack to the previous node.

Diagram:

- Nodes are organized in a hierarchical structure, with A at the root and B, C, D, E, F, G as its children.
- Successors are added to the fringe in LIFO order, ensuring the deepest node is expanded first.
Depth-first search

Expand deepest unexpanded node

**Implementation:**

fringe = LIFO queue, i.e., put successors at front
Expand deepest unexpanded node

**Implementation:**

\( fringe = \text{LIFO queue, i.e., put successors at front} \)
Depth-first search

Implementation:

\( fringe = \text{LIFO queue, i.e., put successors at front} \)
Depth-first search

Expand deepest unexpanded node

Implementation:

fringe $= \text{LIFO queue, i.e., put successors at front}$
Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front
Depth-first search

Expand deepest unexpanded node

**Implementation:**

fringe = LIFO queue, i.e., put successors at front
Depth-first search

Expand deepest unexpanded node

**Implementation:**

fringe = LIFO queue, i.e., put successors at front
Depth-first search

Expand deepest unexpanded node

Implementation: \( fringe = \text{LIFO queue, i.e., put successors at front} \)
Properties

**Complete??** No: fails in infinite-depth spaces, spaces with loops
  Modify to avoid repeated states along path
  ⇒ complete in finite spaces with repeated states avoid

**Time??** $O(b^m)$: terrible if $m$ is much larger than $d$
  but if solutions are dense, may be much faster than breadth-first

**Space??** $O(bm)$, i.e., linear space!

**Optimal??** No
Uniform-cost search

Breadth-first search: First In First Out queue
Depth-first search: Last In First Out queue (stack)
Uniform-cost search: Priority queue (least cost out)

function UNIFORM-COST-SEARCH(problem)
returns a solution, or failure
node ← an initial state
frontier ← a priority queue ordered by PATH-COST, with node as the only element
explored ← an empty set
loop do
if EMPTY?(frontier) then return failure
node ← POP(frontier) /* chooses the lowest-cost node in frontier */
if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
add node.STATE to explored
for each action in problem.ACTIONS(node.STATE) do
child ← CHILD-NODE(problem, node, action)
if child.STATE is not in explored or frontier then
frontier ← INSERT(child, frontier)
else if child.STATE is in frontier with higher PATH-COST then
replace that frontier node with child

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for frontier needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

Figure 3.15 Part of the Romania state space, selected to illustrate uniform-cost search. The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99, respectively. The least-cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost $80 + 97 = 177$. The least-cost node is now Fagaras, so it is expanded, adding Bucharest with cost $99 + 211 = 310$. No nodes are being generated, but uniform-cost search keeps going, choosing Pitesti for expansion and adding a second path part of the map.
Uniform-cost search

Breadth-first search: First In First Out queue
Depth-first search: Last In First Out queue (stack)
Uniform-cost search: Priority queue (least cost out)

part of the map

Sibiu

Fagaras

Rimnicu Vilcea

Pitesti

Bucharest

cost=99

cost=80

99

80

97

211

101
Uniform-cost search

Breadth-first search: First In First Out queue
Depth-first search: Last In First Out queue (stack)
Uniform-cost search: Priority queue (least cost out)

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for frontier needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

Figure 3.15 Part of the Romania state space, selected to illustrate uniform-cost search. May be on a suboptimal path. The second difference is that a test is added in case a better path is found to a node currently on the frontier. Both of these modifications come into play in the example shown in Figure 3.15, where the problem is to get from Sibiu to Bucharest. The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99, respectively. The least-cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost $80 + 97 = 177$. The least-cost node is now Fagaras, so it is expanded, adding Bucharest with cost $99 + 211 = 310$. No node has been generated, but uniform-cost search keeps going, choosing Pitesti for expansion and adding a second path part of the map.
Uniform-cost search

Breadth-first search: First In First Out queue
Depth-first search: Last In First Out queue (stack)
Uniform-cost search: Priority queue (least cost out)

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for frontier needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

Figure 3.15 Part of the Romania state space, selected to illustrate uniform-cost search. Both of these modifications come into play in the example shown in Figure 3.15, where the problem is to get from Sibiu to Bucharest. The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99, respectively. The least-cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost $80 + 97 = 177$. The least-cost node is now Fagaras, so it is expanded, adding Bucharest with cost $99 + 211 = 310$. No wagons are being generated, but uniform-cost search keeps going, choosing Pitesti for expansion and adding a second path part of the map.
Uniform-cost search

Breadth-first search: First In First Out queue
Depth-first search: Last In First Out queue (stack)
Uniform-cost search: Priority queue (least cost out)

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for frontier needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

Figure 3.15 Part of the Romania state space, selected to illustrate uniform-cost search. Both of these modifications come into play in the example shown in Figure 3.15, where the problem is to get from Sibiu to Bucharest. The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99, respectively. The least-cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost $80 + 97 = 177$. The least-cost node is now Fagaras, so it is expanded, adding Bucharest with cost $99 + 211 = 310$. No nodes have been generated, but uniform-cost search keeps going, choosing Pitesti for expansion and adding a second path.
Uniform-cost search

Breadth-first search: First In First Out queue
Depth-first search: Last In First Out queue (stack)
Uniform-cost search: Priority queue (least cost out)

Equivalent to breadth-first if step costs all equal

part of the map

best path from Sibiu to Bucharest
**Properties**

**Complete**?? Yes, if step cost $\geq \epsilon$

**Time**?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$
where $C^*$ is the cost of the optimal solution

**Space**?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$

**Optimal**?? Yes—

**Question:** why it is optimal?
Breadth-first v.s. depth-first

Breadth-first: faster, larger memory
Depth-first: less memory, slower

Question: how to better balance time and space?
Depth-limited search

limit the maximum depth of the depth-first search
i.e., nodes at depth \( l \) have no successors

```plaintext
function Depth-Limited-Search(problem, limit) returns soln/fail/cutoff
    Recursive-DLS(Make-Node(Initial-State[problem]), problem, limit)

function Recursive-DLS(node, problem, limit) returns soln/fail/cutoff

    cutoff-occurred? ← false
    if Goal-Test(problem, State[node]) then return node
    else if Depth[node] = limit then return cutoff
    else for each successor in Expand(node, problem) do
        result ← Recursive-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred? ← true
        else if result ≠ failure then return result
        if cutoff-occurred? then return cutoff else return failure
```
Iterative deepening search

try depth-limited search with increasing limit
restart the search at each time

function Iterative-Deepening-Search( problem) returns a solution
inputs: problem, a problem
for depth ← 0 to ∞ do
    result ← Depth-Limited-Search( problem, depth)
    if result ≠ cutoff then return result
end
Example

Limit = 0

Limit = 1

Limit = 3

wasteful searching the beginning nodes many times?
Properties

Complete?? Yes

Time?? \((d + 1)b^0 + db^1 + (d - 1)b^2 + \ldots + b^d = O(b^d)\) small space as depth-first search

Space?? \(O(bd)\)

Optimal?? Yes, if step cost = 1

Can be modified to explore uniform-cost tree

Numerical comparison for \(b = 10\) and \(d = 5\), solution at far right leaf:

\[
N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450 \\
N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100
\]

IDS does better because other nodes at depth \(d\) are not expanded

BFS can be modified to apply goal test when a node is \textit{generated}
## Summary

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breadth-First</th>
<th>Uniform-Cost</th>
<th>Depth-First</th>
<th>Depth-Limited</th>
<th>Iterative Deepening</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>Yes*</td>
<td>Yes*</td>
<td>No</td>
<td>Yes, if $l \geq d$</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>$b^{d+1}$</td>
<td>$b^{C^*/\epsilon}$</td>
<td>$b^m$</td>
<td>$b^l$</td>
<td>$b^d$</td>
</tr>
<tr>
<td>Space</td>
<td>$b^{d+1}$</td>
<td>$b^{C^*/\epsilon}$</td>
<td>$b^m$</td>
<td>$b^l$</td>
<td>$bd$</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes*</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes*</td>
</tr>
</tbody>
</table>
作业截止日期：9月28日23:59