CS 561: Artificial Intelligence

Instructor: Sofus A. Macskassy, macskass@usc.edu
TAs: Nadeesha Ranashinghe (nadeeshr@usc.edu)
     William Yeoh (wyeoh@usc.edu)
     Harris Chiu (chiciu@usc.edu)

Lectures: MW 5:00-6:20pm, OHE 122 / DEN
Office hours: By appointment
Class page: http://www.rcf.usc.edu/~macskass/CS561-Spring2010/

This class will use http://www.uscden.net/ and class webpage
- Up to date information
- Lecture notes
- Relevant dates, links, etc.

Course material:
Practical issues – coding projects

- Programming projects should be done in C++ or C
- No java.
Last time: Summary

- Definition of AI?
- Turing Test?
- Intelligent Agents:
  - Anything that can be *viewed as* perceiving its *environment* through *sensors* and *acting* upon that environment through its *effectors* to maximize progress towards its *goals*.
  - PAGE (Percepts, Actions, Goals, Environment)
  - Described as a Perception (sequence) to Action Mapping: $f : P^* \rightarrow A$
  - Using look-up-table, closed form, etc.

- **Agent Types**: Reflex, state-based, goal-based, utility-based

- **Rational Action**: The action that maximizes the expected value of the performance measure given the percept sequence to date
Outline: Problem solving and search
(Uninformed Search – AIMA Ch. 3)

- Introduction to Problem Solving
- Complexity

- Uninformed search
  - Problem formulation
  - Search strategies: depth-first, breadth-first

- Informed search
  - Search strategies: best-first, A*
  - Heuristic functions
Example: Measuring problem!

**Problem:** Using these three buckets, measure 7 liters of water.
Example: Measuring problem!

• (one possible) Solution:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

start

\[
\begin{array}{c}
3 / \\
5 / \\
9 / \\
\end{array}
\]

a
b
c
Example: Measuring problem!

- Another Solution:

```
  0  0  0  start
```

![Diagram showing three bars](image)

- Bar a: 3
- Bar b: 5
- Bar c: 9
Which solution do we prefer?

- **Solution 1:**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

  **start**

  **goal**

- **Solution 2:**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

  **start**

  **goal**

CS561 - Lectures 3-4 - Macskassy - Spring 2010
Problem-Solving Agent

function SIMPLE-PROBLEM-SOLVING-AGENT( percept) returns an action

static: seq, an action sequence, initially empty
state, some description of the current world state
goal, a goal, initially null
problem, a problem formulation

state \leftarrow UPDATE-STATE(state, percept)  // What is the current state?

if seq is empty then

goal \leftarrow FORMULATE-GOAL(state)  // From LA to San Diego (given curr. state)

problem \leftarrow FORMULATE-PROBLEM(state, goal)  // e.g., Gas usage

seq \leftarrow SEARCH(problem)

action \leftarrow RECOMMENDATION(seq, state)

seq \leftarrow REMAINDER(seq, state)  // If fails to reach goal, update

return action

Note: This is offline problem-solving. Online problem-solving involves acting w/o complete knowledge of the problem and environment
Example: Buckets

Measure 7 liters of water using a 3-liter, a 5-liter, and a 9-liter buckets.

- **Formulate goal:** Have 7 liters of water in 9-liter bucket

- **Formulate problem:**
  - States: amount of water in the buckets
  - Operators: Fill bucket from source, empty bucket

- **Find solution:** sequence of operators that bring you from current state to the goal state
## Environment types

<table>
<thead>
<tr>
<th>Environment</th>
<th>Accessible</th>
<th>Deterministic</th>
<th>Episodic</th>
<th>Static</th>
<th>Discrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Virtual Reality</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes/no</td>
<td>No</td>
<td>Yes/no</td>
</tr>
<tr>
<td>Office Environment</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Mars</td>
<td>No</td>
<td>Semi</td>
<td>No</td>
<td>Semi</td>
<td>No</td>
</tr>
</tbody>
</table>

The environment types largely determine the agent design.
Problem types

- **Single-state problem**: deterministic, accessible
  
  Agent knows everything about world, thus can calculate optimal action sequence to reach goal state.

- **Multiple-state problem**: deterministic, inaccessible
  
  Agent must reason about sequences of actions and states assumed while working towards goal state.

- **Contingency problem**: nondeterministic, inaccessible
  
  - Must use sensors during execution
  - Solution is a tree or policy
  - Often interleave search and execution

- **Exploration problem**: unknown state space
  
  Discover and learn about environment while taking actions.
Problem types

- **Single-state problem:** deterministic, accessible
  - Agent knows everything about world (the exact state),
  - Can calculate optimal action sequence to reach goal state.
  - E.g., playing chess. Any action will result in an exact state
Problem types

- **Multiple-state problem:** deterministic, inaccessible
  - Agent does not know the exact state (could be in any of the possible states)
    - May not have sensor at all
  - Assume states while working towards goal state.
  - Example: walking in a dark room
    - If you are at the door, going straight will lead you to the kitchen
    - If you are at the kitchen, turning left leads you to the bedroom
    - ...

CS561 - Lectures 3-4 - Macskassy - Spring 2010
Problem types

• **Contingency problem:** nondeterministic, inaccessible
  
  ◦ Must use sensors during execution
  ◦ Solution is a tree or policy
  ◦ Often interleave search and execution
  
  ◦ E.g., a new skater in an arena
    • Sliding problem.
    • Many skaters around
Problem types

- **Exploration problem:** unknown state space

  *Discover and learn about environment while taking actions.*

  - *E.g., Maze*
Example: Vacuum world

Simplified world: 2 locations, each may or not contain dirt, each may or not contain vacuuming agent.
Goal of agent: clean up the dirt.

Single-state, start in #5. Solution?? [Right; Suck]

Conformant, start in \{1; 2; 3; 4; 5; 6; 7; 8\} e.g., Right goes to \{2; 4; 6; 8\}. Solution?? [Right; Suck; Left; Suck]

Contingency, start in #5
Murphy's Law: Suck can dirty a clean carpet
Local sensing: dirt, location only.
Solution?? [Right; if dirt then Suck]
Example: Romania

- In Romania, on vacation. Currently in Arad.
- Flight leaves tomorrow from Bucharest.

- **Formulate goal:**  
  - be in Bucharest

- **Formulate problem:**  
  - states: various cities
  - operators: drive between cities

- **Find solution:**  
  - sequence of cities, such that total driving distance is minimized.
Example: Traveling from Arad To Bucharest
Problem formulation

A problem is defined by four items:

- **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
Selecting a state space

- Real world is absurdly complex; some abstraction is necessary to allow us to reason on it...

- Selecting the correct abstraction and resulting state space is a difficult problem!

- Abstract states $\leftrightarrow$ real-world states

- Abstract operators $\leftrightarrow$ sequences or real-world actions (e.g., going from city i to city j costs $L_{ij} \leftrightarrow$ actually drive from city i to j)

- Abstract solution $\leftrightarrow$ set of real actions to take in the real world such as to solve problem
Example: 8-puzzle

- State:
- Operators:
- Goal test:
- Path cost:
Example: 8-puzzle

- State: integer location of tiles (ignore intermediate locations)
- Operators: moving blank left, right, up, down (ignore jamming)
- Goal test: does state match goal state?
- Path cost: 1 per move

start state

```
5 4
6 1 8
7 3 2
```

goal state

```
1 2 3
8 4
7 6 5
```
Example: 8-puzzle

Why search algorithms?
- 8-puzzle has 362,880 states
- 15-puzzle has $10^{12}$ states
- 24-puzzle has $10^{25}$ states

So, we need a principled way to look for a solution in these huge search spaces...
Back to Vacuum World

states??
actions??
goal test??
path cost??
**Back to 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*)
Real-life example: VLSI Layout

- Given schematic diagram comprising components (chips, resistors, capacitors, etc) and interconnections (wires), find optimal way to place components on a printed circuit board, under the constraint that only a small number of wire layers are available (and wires on a given layer cannot cross!)

- “optimal way”??
  - minimize surface area
  - minimize number of signal layers
  - minimize number of vias (connections from one layer to another)
  - minimize length of some signal lines (e.g., clock line)
  - distribute heat throughout board
  - etc.
Tree search algorithms

Basic idea:

offline, systematic exploration of simulated state-space by generating successors of explored states (expanding)

**Function** General-Search(*problem, strategy*) returns a *solution*, or failure
initialize the search tree using the initial state 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 resulting nodes to the search tree

**end**
Last time: Problem-Solving

- **Problem solving:**
  - Goal formulation
  - Problem formulation (states, operators)
  - Search for solution

- **Problem formulation:**
  - Initial state
  - Operators
  - Goal test
  - Path cost

- **Problem types:**
  - single state: accessible and deterministic environment
  - multiple state: inaccessible and deterministic environment
  - contingency: inaccessible and nondeterministic environment
  - exploration: unknown state-space
Last time: Finding a solution

**Solution:** is ???

**Basic idea:** offline, systematic exploration of simulated state-space by generating successors of explored states (expanding)

```plaintext
Function General-Search(problem, strategy) returns a solution, or failure
initialize the search tree using the initial state 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 resulting nodes to the search tree
end
```
**Solution:** is a sequence of operators that bring you from current state to the goal state.

**Basic idea:** offline, systematic exploration of simulated state-space by generating successors of explored states (expanding).

**Function** General-Search(*problem, strategy*) returns a *solution*, or failure

*initialize the search tree using the initial state 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 resulting nodes to the search tree

**end**

**Strategy:** The search strategy is determined by ???
**Solution:** is a sequence of operators that bring you from current state to the goal state

**Basic idea:** offline, systematic exploration of simulated state-space by generating successors of explored states (expanding)

**Function** General-Search(*problem, strategy*) returns a *solution*, or failure
  
  initialize the search tree using the initial state 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 resulting nodes to the search tree

  **end**

**Strategy:** The search strategy is determined by the *order* in which the nodes are expanded.
Example: Traveling from Arad To Bucharest
Search example
Search example
Search example
A **state** is a (representation of) a physical configuration

A **node** is a data structure constituting part of a search tree

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.
Implementation: General 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)

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
return successors
Implementation: General Search

The operations on `fringe` are queuing functions which inserts and removes elements into the `fringe` and determines the order of node expansion. Varieties of the queuing functions produce varieties of the search algorithm.

```plaintext
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 Expand(node, problem) returns a set of nodes
    successors ← the empty set
    for each state s ∈ problem.get-successors(node) do
        result ← the result of applying action; State[s] ← result
        fringe ← Insert(Leaf-State(problem, action, node, s), fringe)
    return successors
```
Evaluation of search strategies

- A search strategy is defined by picking the order of node expansion.

- Search algorithms are commonly evaluated according to the following four criteria:
  - **Completeness**: does it always find a solution if one exists?
  - **Time complexity**: how long does it take as function of number of nodes?
  - **Space complexity**: how much memory does it require?
  - **Optimality**: does it guarantee the least-cost solution?

- Time and space complexity are measured in terms of:
  - $b$ – max branching factor of the search tree
  - $d$ – depth of the least-cost solution
  - $m$ – max depth of the search tree (may be infinity)
Uninformed search strategies

Use only information available in the problem formulation

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

Expand shallowest unexpanded node

**Implementation:**

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

- Completeness:
- Time complexity:
- Space complexity:
- Optimality:

Search algorithms are commonly evaluated according to the following four criteria:

- **Completeness**: does it always find a solution if one exists?
- **Time complexity**: how long does it take as function of num. of nodes?
- **Space complexity**: how much memory does it require?
- **Optimality**: does it guarantee the least-cost solution?

Time and space complexity are measured in terms of:

- $b$ – max branching factor of the search tree
- $d$ – depth of the least-cost solution
- $m$ – max depth of the search tree (may be infinity)
Properties of breadth-first search

- Completeness: Yes (if $b$ is finite)
- Time complexity: $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$
- Space complexity: $O(b^{d+1})$ (keeps every node in memory)
- Optimality: Yes, if cost=1 per step; not in general

Search algorithms are commonly evaluated according to the following four criteria:
- Completeness: does it always find a solution if one exists?
- Time complexity: how long does it take as function of num. of nodes?
- Space complexity: how much memory does it require?
- Optimality: does it guarantee the least-cost solution?

Time and space complexity are measured in terms of:
- $b$ – max branching factor of the search tree
- $d$ – depth of the least-cost solution
- $m$ – max depth of the search tree (may be infinity)
Time complexity of breadth-first

- If a goal node is found on depth $d$ of the tree, all nodes up till that depth are created and examined (note: and the children of nodes at depth $d$ are created and enqueued, but not yet examined).

Thus: $O(b^d)$
Space complexity of breadth-first

- Largest number of nodes in QUEUE is reached on the level $d+1$ just beyond the goal node.

- QUEUE contains all red nodes. (Thus: 4).
- In General: $b^{d+1} - b \sim b^d$
Uniform-cost search

Expand least-cost unexpanded node

**Implementation:**

- **fringe** = queue ordered by path cost, lowest first

Equivalent to breadth-first if step costs all equal
Romania with step costs in km

Straight-line distance to Bucharest
- Arad: 366 km
- Bucharest: 0 km
- Craiova: 160 km
- Dobroști: 242 km
- Eforie: 161 km
- Făgăraș: 178 km
- Giurgiu: 77 km
- Hirsova: 151 km
- Iași: 226 km
- Lugoj: 244 km
- Mehadia: 241 km
- Neamț: 234 km
- Oradea: 380 km
- Pitești: 98 km
- Rimnicu Vâlcea: 193 km
- Sibiu: 253 km
- Timișoara: 329 km
- Urziceni: 80 km
- Vaslui: 199 km
- Zerind: 374 km
Uniform-cost search
Uniform-cost search
Uniform-cost search
Properties of uniform-cost search

Expand least-cost unexpanded node

**Implementation:**

*fringe* = queue ordered by path cost, lowest first

Equivalent to breadth-first if step costs all equal

**Complete??** Yes, if step cost ≥ ε

**Time??** # of nodes with $g ≤ \text{cost of optimal solution}$, $O(b^{[C^*/\epsilon]})$

where $C^*$ is the cost of the optimal solution

**Space??** # of nodes with $g ≤ \text{cost of optimal solution}$, $O(b^{[C^*/\epsilon]})$

**Optimal??** Yes—nodes expanded in increasing order of $g(n)$

**Note:** $g(n)$ is the path cost to node n
Note: Loop Detection

- A search may fail or be sub-optimal if:
  - no loop detection: then algorithm runs into infinite cycles (A -> B -> A -> B -> ...)
  - not queuing-up a node that has a state which we have already visited: may yield suboptimal solution
  - simply avoiding to go back to our parent: looks promising, but we have not proven that it works

Solution? do not enqueue a node if its state matches the state of any of its parents (assuming path costs > 0).

Indeed, if path costs > 0, it will always cost us more to consider a node with that state again than it had already cost us the first time.

Is that enough??
Example Illustrating Uninformed Search Strategies

From: http://www.csee.umbc.edu/471/current/notes/uninformed-search/
Breadth-First Search Solution

From: http://www.csee.umbc.edu/471/current/notes/uninformed-search/

Breadth-First Search

return GENERAL-SEARCH(problem, ENQUEUE-AT-END)

exp. node nodes list

(S)
S  {ABC}
A  {BCDEG}
B  {CDEGG'}
C  {DEGG'G''}
D  {EGGG''}
E  {GG'G''}
G  {G'G''}

Solution path found is S A G  <-- this G also has cost 10
Number of nodes expanded (including goal node) = 7
Uniform-Cost Search Solution

From: http://www.csee.umbc.edu/471/current/notes/uninformed-search/

Uniform-Cost Search

GENERAL-SEARCH(problem, ENQUEUE-BY-PATH-COST)

exp. node  nodes list

  ( S )
S   ( A(1) B(5) C(8) )
A   ( D(4) B(5) C(8) E(8) G(10) )  (NB, we don't return G)
D   ( B(5) C(8) E(8) G(10) )
B   ( C(8) E(8) G(9) G(10) )
C   ( E(8) G(9) G(10) G(13) )
E   ( G(9) G(10) G(13) )
G   ( )

Solution path found is S B G  <-- this G has cost 9, not 10
Number of nodes expanded (including goal node) = 7
Queueing in Uniform-Cost Search

In the previous example, it is wasteful (but not incorrect) to queue-up three nodes with G state, if our goal if to find the least-cost solution: Although they represent different paths, we know for sure that the one with smallest path cost (9 in the example) will yield a solution with smaller total path cost than the others.

So we can refine the queueing function by:
- queue-up node if
  1) its state does not match the state of any parent and
  2) path cost smaller than path cost of any unexpanded node with same state in the queue (and in this case, replace old node with same state by our new node)

Is that it??
Example

<table>
<thead>
<tr>
<th>#</th>
<th>State</th>
<th>Depth</th>
<th>Cost</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

Black = open queue
Grey = closed queue

Insert expanded nodes
Such as to keep open queue sorted
Example

Node 2 has 2 successors: one with state B and one with state S.

We have node #1 in closed with state S; but its path cost 0 is smaller than the path cost obtained by expanding from A to S. So we do not queue-up the successor of node 2 that has state S.
Node 4 has a successor with state C and Cost smaller than node #3 in open that Also had state C; so we update open To reflect the shortest path.
### Example

<table>
<thead>
<tr>
<th>#</th>
<th>State</th>
<th>Depth</th>
<th>Cost</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>C</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>D</td>
<td>4</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>G</td>
<td>3</td>
<td>102</td>
<td>4</td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>#</th>
<th>State</th>
<th>Depth</th>
<th>Cost</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>C</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>D</td>
<td>4</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>E</td>
<td>5</td>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>G</td>
<td>3</td>
<td>102</td>
<td>4</td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>#</th>
<th>State</th>
<th>Depth</th>
<th>Cost</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>C</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>D</td>
<td>4</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>E</td>
<td>5</td>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>6</td>
<td>18</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>G</td>
<td>3</td>
<td>102</td>
<td>4</td>
</tr>
</tbody>
</table>
The node with state G and cost 102 has been removed from the open queue and placed into the closed queue, as a cheaper node with state G and code 23 was pushed into the open queue.
Example

<table>
<thead>
<tr>
<th>#</th>
<th>State</th>
<th>Depth</th>
<th>Cost</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>C</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>D</td>
<td>4</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>E</td>
<td>5</td>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>6</td>
<td>18</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>G</td>
<td>7</td>
<td>23</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>G</td>
<td>3</td>
<td>102</td>
<td>4</td>
</tr>
</tbody>
</table>

Goal reached
Depth-first search

Expand deepest unexpanded node

**Implementation:**

*fringe* = LIFO queue, i.e., put successors at front
Romania with step costs in km

Straight-line distance to Bucharest

- **Arad**: 366
- **Bucharest**: 0
- **Craiova**: 160
- **Dobrota**: 242
- **Eforie**: 161
- **Fagaras**: 178
- **Giurgiu**: 77
- **Hirsova**: 151
- **Iasi**: 226
- **Lugoj**: 244
- **Mehadia**: 241
- **Neamt**: 234
- **Oradea**: 380
- **Pitesti**: 98
- **Rimnicu Vilcea**: 193
- **Sibiu**: 253
- **Timisoara**: 329
- **Urziceni**: 80
- **Vaslui**: 199
- **Zerind**: 374
Depth-first search
Depth-first search
Depth-first search

I.e., depth-first search can perform infinite cyclic excursions
Need a finite, non-cyclic search space (or repeated-state checking)
Depth-first search

Expand deepest unexpanded node

**Implementation:**

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

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

- **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
Time complexity of depth-first

- In the worst case:
  - the (only) goal node may be on the right-most branch,

\[
\text{Time complexity } = b^m + b^{m-1} + \ldots + 1 = \frac{b^{m+1} - 1}{b - 1}
\]

- Thus: \( O(b^m) \)
Space complexity of depth-first

- Largest number of nodes in QUEUE is reached in bottom left-most node.
- Example: $m = 3$, $b = 3$:
  - QUEUE contains all red nodes = 7.
  - In General: $((b-1) * m) + b$
  - Order: $O(m*b)$
Avoiding repeated states

In increasing order of effectiveness and computational overhead:

- **do not return to state we come from**, i.e., expand function will skip possible successors that are in same state as node’s parent.

- **do not create paths with cycles**, i.e., expand function will skip possible successors that are in same state as any of node’s ancestors.

- **do not generate any state that was ever generated before**, by keeping track (in memory) of every state generated, unless the cost of reaching that state is lower than last time we reached it.
**Depth-limited search**

Is a depth-first search with depth limit $l$

**Recursive implementation:**

```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
```

CS561 - Lectures 3-4 - Macskassy - Spring 2010
Iterative deepening search

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

Combines the best of breadth-first and depth-first search strategies.
Iterative deepening search $l=0$
Iterative deepening search $l=1$

Limit = 1

Diagram showing iterative deepening search with levels.
Iterative deepening search $l=2$

Limit = 2
Iterative deepening search $l=3$

Limit = 3

- Iteration 1: Start at A, then B, C, D, E, F, G
- Iteration 2: Add H, I, J, K, L, M, N, O to the search space
- Iteration 3: No further iterations needed as all nodes have been explored
Iterative deepening complexity

- Iterative deepening search may seem wasteful because so many states are expanded multiple times.

- In practice, however, the overhead of these multiple expansions is small, because most of the nodes are towards leaves (bottom) of the search tree:

  \[ \text{thus, the nodes that are evaluated several times (towards top of tree) are in relatively small number.} \]

- In iterative deepening, nodes at bottom level are expanded once, level above twice, etc. up to root (expanded \( d+1 \) times) so total number of expansions is:

  \[(d+1)b^0 + (d)b^1 + (d-1)b^2 + \ldots + 3b^{(d-2)} + 2b^{(d-1)} + 1b^d = O(b^d)\]

- In general, iterative deepening is preferred to depth-first or breadth-first when search space large and depth of solution not known.
Properties of iterative deepening search

**Complete??** Yes

**Time??** \((d + 1)b^0 + db^1 + (d - 1)b^2 + \ldots + b^d = O(b^d)\)

**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 **generated**
Bidirectional search

- Both search forward from initial state, and \textbf{backwards from goal}.
- Stop when the two searches meet in the middle.

\underline{Problem}: how do we search backwards from goal??

\begin{itemize}
  \item predecessor of node $n = \text{all nodes that have } n \text{ as successor}$
  \item this may not always be easy to compute!
  \item if several goal states, apply predecessor function to them just as we applied successor (only works well if goals are explicitly known; may be difficult if goals only characterized implicitly).
\end{itemize}
Bidirectional search

- **Problem:** how do we search backwards from goal?? (cont.)
  - ...
  - for bidirectional search to work well, there must be an efficient way to check whether a given node belongs to the other search tree.
  - select a given search algorithm for each half.
Bidirectional search

1. QUEUE1 ← path only containing the root;
   QUEUE2 ← path only containing the goal;

2. WHILE both QUEUEs are not empty
   AND QUEUE1 and QUEUE2 do NOT share a state
   DO
       remove their first paths;
       create their new paths (to all children);
       reject their new paths with loops;
       add their new paths to back;

3. IF QUEUE1 and QUEUE2 share a state
   THEN success;
   ELSE failure;
Bidirectional search

- **Completeness:** Yes,
- **Time complexity:** $2 \times O(b^{d/2}) = O(b^{d/2})$
- **Space complexity:** $O(b^{d/2})$
- **Optimality:** Yes

- To avoid one by one comparison, we need a hash table of size $O(b^{d/2})$
  - *If hash table is used, the cost of comparison is $O(1)$*
Bidirectional Search

Initial State

Final State

$d / 2$

$d$
Bidirectional search

- Bidirectional search merits:
  - Big difference for problems with branching factor $b$ in both directions
    - A solution of length $d$ will be found in $O(2b^{d/2}) = O(b^{d/2})$
    - For $b = 10$ and $d = 6$, only 2,222 nodes are needed instead of 1,111,111 for breadth-first search
Bidirectional search

- Bidirectional search issues
  - *Predecessors* of a node need to be generated
    - Difficult when operators are not reversible
  - What to do if there is no *explicit list of goal* states?
  - For each node: *check if it appeared in the other search*
    - Needs a hash table of $O(b^{d/2})$
  - What is the *best search strategy* for the two searches?
Comparing uninformed search strategies

<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>
<th>Bidirectional (if applicable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>$b^{d+1}$</td>
<td>$b^{\lceil C/\epsilon \rceil}$</td>
<td>$b^m$</td>
<td>$b^l$</td>
<td>$b^d$</td>
<td>$b^{(d/2)}$</td>
</tr>
<tr>
<td>Space</td>
<td>$b^{d+1}$</td>
<td>$b^{\lceil C/\epsilon \rceil}$</td>
<td>$bm$</td>
<td>$bl$</td>
<td>$bd$</td>
<td>$b^{(d/2)}$</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Complete?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes if $l \geq d$</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- $b$ – max branching factor of the search tree
- $d$ – depth of the least-cost solution
- $m$ – max depth of the state-space (may be infinity)
- $l$ – depth cutoff
Repeated States

- Failure to detect repeated states can turn a linear problem into an exponential one!
function \text{GRAPH-SEARCH}(\text{problem, fringe}) \text{ returns} \ a \ solution, \ or \ failure

\text{closed} \leftarrow \text{an} \ \text{empty} \ \text{set}
\text{fringe} \leftarrow \text{INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)}

\text{loop do}

\text{if} \ \text{fringe} \ \text{is} \ \text{empty} \ \text{then} \ \text{return} \ \text{failure}
\text{node} \leftarrow \text{REMOVE-FRONT(fringe)}
\text{if} \ \text{GOAL-TEST(problem, STATE[node])} \ \text{then} \ \text{return} \ \text{node}
\text{if} \ \text{STATE[node]} \ \text{is} \ \text{not} \ \text{in} \ \text{closed} \ \text{then}
\quad \text{add} \ \text{STATE[node]} \ \text{to} \ \text{closed}
\quad \text{fringe} \leftarrow \text{INSERTALL(EXPAND(node, problem), fringe)}

\text{end}
Summary

- Problem formulation usually requires **abstracting away real-world details** to define a **state space** that can be explored using computer algorithms.

- Once problem is formulated in abstract form, **complexity analysis** helps us picking out best algorithm to solve problem.

- Variety of uninformed search strategies; difference lies in method used to **pick node that will be further expanded**.

- **Iterative deepening** search only uses linear space and not much more time than other uniformed search strategies.

- **Graph search** can be exponentially more efficient than tree search.