Problems and Search

Chapter 2
Outline

• State space search
• Search strategies
• Problem characteristics
• Design of search programs
State Space Search

Problem solving = Searching for a goal state
State Space Search: Playing Chess

• Each **position** can be described by an 8-by-8 array.

• **Initial position** is the game opening position.

• **Goal position** is any position in which the opponent does not have a legal move and his or her king is under attack.

• **Legal moves** can be described by a set of rules:
  – Left sides are matched against the current state.
  – Right sides describe the new resulting state.
State Space Search: Playing Chess

- **State space** is a set of legal positions.
- Starting at the initial state.
- Using the set of rules to move from one state to another.
- Attempting to end up in a goal state.
State Space Search: Water Jug Problem

“You are given two jugs, a 4-litre one and a 3-litre one. Neither has any measuring markers on it. There is a pump that can be used to fill the jugs with water. How can you get exactly 2 litres of water into 4-litre jug.”
State Space Search: Water Jug Problem

• State: \((x, y)\)

  \[ x = 0, 1, 2, 3, \text{ or } 4 \quad y = 0, 1, 2, 3 \]

• Start state: \((0, 0)\).

• Goal state: \((2, n)\) for any \(n\).

• Attempting to end up in a goal state.
State Space Search: Water Jug Problem

1. \((x, y)\) → \((4, y)\)  
   if \(x < 4\)

2. \((x, y)\) → \((x, 3)\)  
   if \(y < 3\)

3. \((x, y)\) → \((x - d, y)\)  
   if \(x > 0\)

4. \((x, y)\) → \((x, y - d)\)  
   if \(y > 0\)
State Space Search: Water Jug Problem

5. \((x, y) \rightarrow (0, y)\)
   if \(x > 0\)

6. \((x, y) \rightarrow (x, 0)\)
   if \(y > 0\)

7. \((x, y) \rightarrow (4, y - (4 - x))\)
   if \(x + y \geq 4, y > 0\)

8. \((x, y) \rightarrow (x - (3 - y), 3)\)
   if \(x + y \geq 3, x > 0\)
State Space Search: Water Jug Problem

9. \((x, y)\) \rightarrow (x + y, 0) 
   if \(x + y \leq 4\), \(y > 0\)

10. \((x, y)\) \rightarrow (0, x + y) 
    if \(x + y \leq 3\), \(x > 0\)

11. \((0, 2)\) \rightarrow (2, 0)

12. \((2, y)\) \rightarrow (0, y)
State Space Search: Water Jug Problem

1. current state = \((0, 0)\)

2. Loop until reaching the goal state \((2, 0)\)
   
   - Apply a rule whose left side matches the current state
   - Set the new current state to be the resulting state
State Space Search: Water Jug Problem

The role of the condition in the left side of a rule
⇒ restrict the application of the rule
⇒ more efficient

1. (x, y) \rightarrow (4, y)
   if x < 4

2. (x, y) \rightarrow (x, 3)
   if y < 3
State Space Search: Water Jug Problem

**Special-purpose** rules to capture special-case knowledge that can be used at some stage in solving a problem

11. \((0, 2)\) \(\rightarrow\) \((2, 0)\)

12. \((2, y)\) \(\rightarrow\) \((0, y)\)
State Space Search: Summary

1. Define a state space that contains all the possible configurations of the relevant objects.

2. Specify the initial states.

3. Specify the goal states.

4. Specify a set of rules:
   - What are unstated assumptions?
   - How general should the rules be?
   - How much knowledge for solutions should be in the rules?
Search Strategies

Requirements of a good search strategy:

1. It causes **motion**
   Otherwise, it will never lead to a solution.

2. It is **systematic**
   Otherwise, it may use more steps than necessary.

3. It is **efficient**
   Find a good, but not necessarily the best, answer.
Search Strategies

1. **Uninformed search** (blind search)
   
   Having no information about the number of steps from the current state to the goal.

2. **Informed search** (heuristic search)

   More efficient than uninformed search.
Search Strategies

- (0, 0)
  - (4, 0)
    - (4, 3)
    - (0, 0)
    - (1, 3)
  - (0, 3)
    - (4, 3)
    - (0, 0)
    - (3, 0)
Search Strategies: Blind Search

• **Breadth-first search**
  Expand all the nodes of one level first.

• **Depth-first search**
  Expand one of the nodes at the deepest level.
## Search Strategies: Blind Search

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breadth-First</th>
<th>Depth-First</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Space</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Optimal?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complete?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ b: \text{branching factor} \quad d: \text{solution depth} \quad m: \text{maximum depth} \]
## Search Strategies: Blind Search

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breadth-First</th>
<th>Depth-First</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>$b^d$</td>
<td>$b^m$</td>
</tr>
<tr>
<td>Space</td>
<td>$b^d$</td>
<td>$b^m$</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Complete?</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

$b$: branching factor  
$d$: solution depth  
$m$: maximum depth
Search Strategies: Heuristic Search

- **Heuristic**: involving or serving as an aid to learning, discovery, or problem-solving by experimental and especially trial-and-error methods. (Merriam-Webster’s dictionary)

- Heuristic technique improves the efficiency of a search process, possibly by **sacrificing** claims of **completeness** or **optimality**.
Search Strategies: Heuristic Search

• Heuristic is for combinatorial explosion.

• Optimal solutions are rarely needed.
Search Strategies: Heuristic Search

The Travelling Salesman Problem

“A salesman has a list of cities, each of which he must visit exactly once. There are direct roads between each pair of cities on the list. Find the route the salesman should follow for the shortest possible round trip that both starts and finishes at any one of the cities.”
Search Strategies: Heuristic Search

**Nearest neighbour heuristic:**

1. Select a starting city.
2. Select the one closest to the current city.
3. Repeat step 2 until all cities have been visited.
Search Strategies: Heuristic Search

Nearest neighbour heuristic:

1. Select a starting city.
2. Select the one closest to the current city.
3. Repeat step 2 until all cities have been visited.

$O(n^2)$ vs. $O(n!)$
Search Strategies: Heuristic Search

• **Heuristic function:**

  state descriptions $\rightarrow$ measures of desirability
Problem Characteristics

To choose an appropriate method for a particular problem:
• Is the problem decomposable?
• Can solution steps be ignored or undone?
• Is the universe predictable?
• Is a good solution absolute or relative?
• Is the solution a state or a path?
• What is the role of knowledge?
• Does the task require human-interaction?
Is the problem decomposable?

- Can the problem be broken down to smaller problems to be solved independently?

- Decomposable problem can be solved easily.
Is the problem decomposable?

\[ \int (x^2 + 3x + \sin^2 x \cdot \cos^2 x) \, dx \]

\[ \int x^2 \, dx \quad \int 3x \, dx \quad \int \sin^2 x \cdot \cos^2 x \, dx \]

\[ \int (1 - \cos^2 x) \cdot \cos^2 x \, dx \]

\[ \int \cos^2 x \, dx \quad - \int \cos^4 x \, dx \]
Is the problem decomposable?

Blocks World

CLEAR(x) → ON(x, Table)

CLEAR(x) and CLEAR(y) → ON(x, y)
Is the problem decomposable?

ON(B, C) and ON(A, B)

ON(B, C)  ON(A, B)

CLEAR(A)  ON(A, B)

A  B  C

A  B
Can solution steps be ignored or undone?

Theorem Proving

A lemma that has been proved can be ignored for next steps.

Ignorable!
Can solution steps be ignored or undone?

**The 8-Puzzle**

Moves can be undone and backtracked.

**Recoverable!**
Can solution steps be ignored or undone?

Playing Chess

Moves cannot be retracted.

Irrecoverable!
Can solution steps be ignored or undone?

• **Ignorable problems** can be solved using a simple control structure that never backtracks.

• **Recoverable problems** can be solved using backtracking.

• **Irrecoverable problems** can be solved by recoverable style methods via planning.
Is the universe predictable?

The 8-Puzzle

Every time we make a move, we know exactly what will happen.

Certain outcome!
Is the universe predictable?

**Playing Bridge**

We cannot know exactly where all the cards are or what the other players will do on their turns.

**Uncertain outcome!**
Is the universe predictable?

• For certain-outcome problems, planning can be used to generate a sequence of operators that is guaranteed to lead to a solution.

• For uncertain-outcome problems, a sequence of generated operators can only have a good probability of leading to a solution.

Plan revision is made as the plan is carried out and the necessary feedback is provided.
Is a good solution absolute or relative?

1. Marcus was a man.
2. Marcus was a Pompeian.
3. Marcus was born in 40 A.D.
4. All men are mortal.
5. All Pompeians died when the volcano erupted in 79 A.D.
6. No mortal lives longer than 150 years.
7. It is now 2004 A.D.
Is a good solution absolute or relative?

1. Marcus was a man.
2. Marcus was a Pompeian.
3. Marcus was born in 40 A.D.
4. All men are mortal.
5. All Pompeians died when the volcano erupted in 79 A.D.
6. No mortal lives longer than 150 years.
7. It is now 2004 A.D.

Is Marcus alive?
Is a good solution absolute or relative?

1. Marcus was a man.
2. Marcus was a Pompeian.
3. Marcus was born in 40 A.D.
4. All men are mortal.
5. All Pompeians died when the volcano erupted in 79 A.D.
6. No mortal lives longer than 150 years.
7. It is now 2004 A.D.

Is Marcus alive?

Different reasoning paths lead to the answer. It does not matter which path we follow.
Is a good solution absolute or relative?

**The Travelling Salesman Problem**

We have to try all paths to find the shortest one.
Is a good solution absolute or relative?

• Any-path problems can be solved using heuristics that suggest good paths to explore.

• For best-path problems, much more exhaustive search will be performed.
Is the solution a state or a path?

Finding a consistent interpretation

“The bank president ate a dish of pasta salad with the fork”.

– “bank” refers to a financial situation or to a side of a river?
– “dish” or “pasta salad” was eaten?
– Does “pasta salad” contain pasta, as “dog food” does not contain “dog”?
– Which part of the sentence does “with the fork” modify?

What if “with vegetables” is there?
Is the solution a state or a path?

**The Water Jug Problem**

The path that leads to the goal must be reported.
Is the solution a state or a path?

• A path-solution problem can be reformulated as a state-solution problem by describing a state as a partial path to a solution.

• The question is whether that is natural or not.
What is the role of knowledge

**Playing Chess**
Knowledge is important only to constrain the search for a solution.

**Reading Newspaper**
Knowledge is required even to be able to recognize a solution.
Does the task require human-interaction?

• **Solitary problem**, in which there is no intermediate communication and no demand for an explanation of the reasoning process.

• **Conversational problem**, in which intermediate communication is to provide either additional assistance to the computer or additional information to the user.
Problem Classification

• There is a variety of problem-solving methods, but there is no one single way of solving all problems.

• Not all new problems should be considered as totally new. Solutions of similar problems can be exploited.
Homework

Exercises 1-7 (Chapter 2 – AI Rich & Knight)
Heuristic Search

Chapter 3
Outline

• Generate-and-test
• Hill climbing
• Best-first search
• Problem reduction
• Constraint satisfaction
• Means-ends analysis
Generate-and-Test

Algorithm
1. Generate a possible solution.
2. Test to see if this is actually a solution.
3. Quit if a solution has been found. Otherwise, return to step 1.
Generate-and-Test

• Acceptable for simple problems.

• Inefficient for problems with large space.
Generate-and-Test

• **Exhaustive** generate-and-test.

• **Heuristic** generate-and-test: not consider paths that seem unlikely to lead to a solution.

• **Plan** generate-test:
  – Create a list of candidates.
  – Apply generate-and-test to that list.
Generate-and-Test

Example: coloured blocks
“Arrange four 6-sided cubes in a row, with each side of each cube painted one of four colours, such that on all four sides of the row one block face of each colour is showing.”
Generate-and-Test

Example: coloured blocks

Heuristic: if there are more red faces than other colours
then, when placing a block with several red faces, use few of them as possible as outside faces.
Hill Climbing

• Searching for a goal state = Climbing to the top of a hill
Hill Climbing

• Generate-and-test + **direction to move.**

• **Heuristic function** to estimate how close a given state is to a goal state.
Simple Hill Climbing

Algorithm
1. Evaluate the initial state.

2. Loop until a solution is found or there are no new operators left to be applied:
   - Select and apply a new operator
   - Evaluate the new state:
     - goal \(\rightarrow\) quit
     - better than current state \(\rightarrow\) new current state
Simple Hill Climbing

• Evaluation function as a way to inject *task-specific knowledge* into the control process.
Simple Hill Climbing

Example: coloured blocks

Heuristic function: the sum of the number of different colours on each of the four sides (solution = 16).
Steepest-Ascent Hill Climbing (Gradient Search)

• Considers all the moves from the current state.

• Selects the best one as the next state.
Steepest-Ascent Hill Climbing
(Gradient Search)

Algorithm

1. Evaluate the initial state.

2. Loop until a solution is found or a complete iteration produces no change to current state:
   - SUCC = a state such that any possible successor of the current state will be better than SUCC (the worst state).
   - For each operator that applies to the current state, evaluate the new state:
     goal → quit
     better than SUCC → set SUCC to this state
   - SUCC is better than the current state → set the current state to SUCC.
Hill Climbing: Disadvantages

**Local maximum**

A state that is better than all of its neighbours, but not better than some other states far away.
Hill Climbing: Disadvantages

Plateau
A flat area of the search space in which all neighbouring states have the same value.
Hill Climbing: Disadvantages

Ridge
The orientation of the high region, compared to the set of available moves, makes it impossible to climb up. However, two moves executed serially may increase the height.
Hill Climbing: Disadvantages

Ways Out

• **Backtrack** to some earlier node and try going in a different direction.
• Make a **big jump** to try to get in a new section.
• Moving in **several directions** at once.
Hill Climbing: Disadvantages

• Hill climbing is a **local method**: Decides what to do next by looking only at the “immediate” consequences of its choices.

• **Global information** might be encoded in heuristic functions.
Hill Climbing: Disadvantages

Start

A
D
C
B

Goal

D
C
B
A

Blocks World
Hill Climbing: Disadvantages

Blocks World

Start 0
A
D
C
B

Goal 4
D
C
B
A

Local heuristic:
+1 for each block that is resting on the thing it is supposed to be resting on.
−1 for each block that is resting on a wrong thing.
Hill Climbing: Disadvantages

Diagram:

0
A
D
C
B

2
D
C
B
A
Hill Climbing: Disadvantages

```
A  0  B
D  0  C
C  0  B
B
```

```
D  2  A
C  
B  
A
```

```
C  0  D
B  A  D
A  
B
```
Hill Climbing: Disadvantages

Start

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>D</td>
<td>B</td>
</tr>
</tbody>
</table>

Goal

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>C</td>
<td>A</td>
</tr>
</tbody>
</table>

Blocks World

Global heuristic:
For each block that has the correct support structure: +1 to every block in the support structure.
For each block that has a wrong support structure: −1 to every block in the support structure.
Hill Climbing: Disadvantages

-6
A
D
C
B

-3
D
C
B
A

-2
C
D
B
A

-1
C
B
A
D
Hill Climbing: Conclusion

• Can be very inefficient in a large, rough problem space.

• Global heuristic may have to pay for computational complexity.

• Often useful when combined with other methods, getting it started right in the right general neighbourhood.
Simulated Annealing

• A variation of hill climbing in which, at the beginning of the process, some downhill moves may be made.

• To do enough exploration of the whole space early on, so that the final solution is relatively insensitive to the starting state.

• Lowering the chances of getting caught at a local maximum, or plateau, or a ridge.
Simulated Annealing

Physical Annealing

• Physical substances are melted and then **gradually cooled** until some solid state is reached.

• The goal is to produce a **minimal-energy** state.

• **Annealing schedule**: if the temperature is lowered sufficiently slowly, then the goal will be attained.

• Nevertheless, there is some probability for a transition to a higher energy state: $e^{-\Delta E/kT}$. 
Simulated Annealing

Algorithm
1. Evaluate the initial state.

2. Loop until a solution is found or there are no new operators left to be applied:
   – Set $T$ according to an annealing schedule
   – Selects and applies a new operator
   – Evaluate the new state:
     goal $\rightarrow$ quit
     $\Delta E = \text{Val(current state)} - \text{Val(new state)}$
     $\Delta E < 0 \rightarrow$ new current state
Best-First Search

• **Depth-first search**: not all competing branches having to be expanded.

• **Breadth-first search**: not getting trapped on dead-end paths.

⇒ Combining the two is to **follow a single path at a time**, but **switch paths** whenever some competing path look more promising than the current one.
Best-First Search

```
A
   B  C  D
  3  5  1

A
   B  C
  3  5

A
   G  H  E  F
  6  5  4  6

A
   B  C  D
  6  5  4  6
```

```
A
   B  C  D
  3  5  1

A
   B  C
  3  5

A
   G  H  E  F
  6  5  4  6

A
   B  C  D
  6  5  4  6
```
Best-First Search

• **OPEN**: nodes that have been generated, but have not examined.

  This is organized as a *priority queue*.

• **CLOSED**: nodes that have already been examined.

  Whenever a new node is generated, check whether it has been *generated before*.
Best-First Search

Algorithm

1. OPEN = \{initial state\}.

2. Loop until a goal is found or there are no nodes left in OPEN:
   - Pick the best node in OPEN
   - Generate its successors
   - For each successor:
     - new \(\rightarrow\) evaluate it, add it to OPEN, record its parent
     - generated before \(\rightarrow\) change parent, update successors
Best-First Search

- **Greedy search:**
  \[ h(n) = \text{estimated cost of the cheapest path from node } n \text{ to a goal state.} \]
Best-First Search

- **Uniform-cost search:**
  \[ g(n) = \text{cost of the cheapest path from the initial state to node } n. \]
Best-First Search

• **Greedy search:**
  
  \[ h(n) = \text{estimated cost of the cheapest path from node } n \text{ to a goal state.} \]

  Neither optimal nor complete
Best-First Search

- **Greedy search:**
  \[ h(n) = \text{estimated cost of the cheapest path from node } n \text{ to a goal state.} \]

  Neither optimal nor complete

- **Uniform-cost search:**
  \[ g(n) = \text{cost of the cheapest path from the initial state to node } n. \]

  Optimal and complete, but very inefficient
Best-First Search

- Algorithm A* (Hart et al., 1968):

\[ f(n) = g(n) + h(n) \]

\[ h(n) = \text{cost of the cheapest path from node } n \text{ to a goal state.} \]

\[ g(n) = \text{cost of the cheapest path from the initial state to node } n. \]
Best-First Search

- **Algorithm A***:

\[ f^*(n) = g^*(n) + h^*(n) \]

- **\( h^*(n) \)** (heuristic factor) = estimate of \( h(n) \).

- **\( g^*(n) \)** (depth factor) = approximation of \( g(n) \) found by \( A^* \) so far.
Problem Reduction

Goal: Acquire TV set
- Goal: Steal TV set
- Goal: Earn some money
- Goal: Buy TV set

AND-OR Graphs

Problem Reduction: AO*
Problem Reduction: AO*
Constraint Satisfaction

- Many AI problems can be viewed as problems of constraint satisfaction.

Cryptarithmetic puzzle:

\[
\begin{align*}
\text{SEND} \\
+ \text{MORE} \\
\hline
\text{MONEY}
\end{align*}
\]
Constraint Satisfaction

- As compared with a straightforward search procedure, viewing a problem as one of constraint satisfaction can reduce substantially the amount of search.
Constraint Satisfaction

• Operates in a space of constraint sets.

• Initial state contains the original constraints given in the problem.

• A goal state is any state that has been constrained “enough”.
Constraint Satisfaction

Two-step process:

1. Constraints are discovered and propagated as far as possible.

2. If there is still not a solution, then search begins, adding new constraints.
Initial state:

- No two letters have the same value.
- The sum of the digits must be as shown.

\[
\begin{align*}
M &= 1 \\
S &= 8 \text{ or } 9 \\
O &= 0 \\
N &= E + 1 \\
C2 &= 1 \\
N + R &> 8 \\
E &\neq 9
\end{align*}
\]

\[
\begin{align*}
E &= 2 \\
N &= 3 \\
R &= 8 \text{ or } 9 \\
2 + D &= Y \text{ or } 2 + D = 10 + Y
\end{align*}
\]

\[
\begin{align*}
C1 &= 0 \\
2 + D &= Y \\
N + R &= 10 + E \\
R &= 9 \\
S &= 8
\end{align*}
\]

\[
\begin{align*}
C1 &= 1 \\
2 + D &= 10 + Y \\
D &= 8 + Y \\
D &= 8 \text{ or } 9 \\
D &= 8 \\
Y &= 0 \\
D &= 9 \\
Y &= 1
\end{align*}
\]
Constraint Satisfaction

**Two kinds of rules:**

1. Rules that define valid constraint propagation.

2. Rules that suggest guesses when necessary.
Homework

**Exercises**  1-14 (Chapter 3 – AI Rich & Knight)

**Reading**  Algorithm A*

(http://en.wikipedia.org/wiki/A%2A_algorithm)
Game Playing

Chapter 8
Outline

• Overview
• Minimax search
• Adding alpha-beta cutoffs
• Additional refinements
• Iterative deepening
• Specific games
Overview

Old beliefs

Games provided a structured task in which it was very easy to measure success or failure.

Games did not obviously require large amounts of knowledge, thought to be solvable by straightforward search.
Overview

Chess

The average branching factor is around 35.

In an average game, each player might make 50 moves.

One would have to examine $35^{100}$ positions.
Overview

• Improve the *generate procedure* so that only good moves are generated.

• Improve the *test procedure* so that the best moves will be recognized and explored first.
Overview

• Improve the *generate procedure* so that only good moves are generated.
  
  plausible-moves vs. legal-moves

• Improve the *test procedure* so that the best moves will be recognized and explored first.
  
  less moves to be evaluated
Overview

• It is not usually possible to search until a goal state is found.

• It has to evaluate individual board positions by estimating how likely they are to lead to a win.

  **Static evaluation function**

• Credit assignment problem (Minsky, 1963).
Overview

- Good plausible-move generator.
- Good static evaluation function.
Minimax Search

- Depth-first and depth-limited search.
- At the player choice, maximize the static evaluation of the next position.
- At the opponent choice, minimize the static evaluation of the next position.
Minimax Search

Two-ply search

Maximizing ply
Player

Minimizing ply
Opponent
Minimax Search

Player(Position, Depth):
for each $S \in$ SUCCESSORS(Position) do

RESULT = Opponent($S$, Depth + 1)

NEW-VALUE = VALUE(RESULT)

if NEW-VALUE > MAX-SCORE, then

MAX-SCORE = NEW-VALUE
BEST-PATH = PATH(RESULT) + S

return

VALUE = MAX-SCORE
PATH = BEST-PATH
Minimax Search

\[ \text{Opponent}(\text{Position}, \text{Depth}): \]
\[ \text{for each } S \in \text{SUCCESSORS}(\text{Position}) \text{ do} \]
\[ \quad \text{RESULT} = \text{Player}(S, \text{Depth} + 1) \]
\[ \quad \text{NEW-VALUE} = \text{VALUE(RESULT)} \]
\[ \quad \text{if NEW-VALUE} < \text{MIN-SCORE}, \text{ then} \]
\[ \quad \quad \text{MIN-SCORE} = \text{NEW-VALUE} \]
\[ \quad \quad \text{BEST-PATH} = \text{PATH(RESULT)} + S \]
\[ \text{return} \]
\[ \quad \text{VALUE} = \text{MIN-SCORE} \]
\[ \quad \text{PATH} = \text{BEST-PATH} \]
Minimax Search

\textbf{Any-Player}(Position, Depth):
for each \( S \in \text{SUCCESSORS}(\text{Position}) \) do

\[ \text{RESULT} = \text{Any-Player}(S, \text{Depth} + 1) \]

\[ \text{NEW-VALUE} = - \text{VALUE}(\text{RESULT}) \]

if \( \text{NEW-VALUE} > \text{BEST-SCORE} \), then

\[ \text{BEST-SCORE} = \text{NEW-VALUE} \]
\[ \text{BEST-PATH} = \text{PATH}(\text{RESULT}) + S \]

return

\[ \text{VALUE} = \text{BEST-SCORE} \]
\[ \text{PATH} = \text{BEST-PATH} \]
Minimax Search

MINIMAX(Position, Depth, Player):

• MOVE-GEN(Position, Player).
• STATIC(Position, Player).
• DEEP-ENOUGH(Position, Depth)
Minimax Search

1. if DEEP-ENOUGH(Position, Depth), then return:

   VALUE = STATIC(Position, Player)
   PATH = nil

2. SUCCESSORS = MOVE-GEN(Position, Player)

3. if SUCCESSORS is empty, then do as in Step 1
Minimax Search

4. if SUCCESSORS is not empty:
   RESULT-SUCC = MINIMAX(SUCC, Depth+1, Opp(Player))
   NEW-VALUE = - VALUE(RESULT-SUCC)
   if NEW-VALUE > BEST-SCORE, then:
     BEST-SCORE = NEW-VALUE
     BEST-PATH = PATH(RESULT-SUCC) + SUCC

5. Return:
   VALUE = BEST-SCORE
   PATH = BEST-PATH
Adding Alpha-Beta Cutoffs

- **Depth-first and depth-limited search.**  
  
  branch-and-bound

- At the player choice, maximize the static evaluation of the next position.  
  
  > $\alpha$ threshold

- At the opponent choice, minimize the static evaluation of the next position.  
  
  < $\beta$ threshold
Adding Alpha-Beta Cutoffs

Maximizing ply
Player

Minimizing ply
Opponent

Alpha cutoffs
Adding Alpha-Beta Cutoffs

- Alpha and Beta cutoffs
- Maximizing ply
- Player
- Minimizing ply
- Opponent
- Maximizing ply
- Player
- Minimizing ply
- Opponent

Diagram:

A
  / \  > 3
B C
 /   |
D E F G H
 /   |   |
0 5 5 7
I J M N
 /   |
K L
0
Adding Alpha-Beta Cutoffs

Opponent

Player

\( \alpha \in \beta \)

\( \alpha \in \beta \)

\( \alpha \in \beta \)

Alpha and Beta cutoffs

Opponent

Player
Player(Position, Depth, α, β):
for each $S \in \text{SUCCESSORS}(\text{Position})$ do

RESULT = Opponent(S, Depth + 1, α, β)

NEW-VALUE = VALUE(RESULT)

if NEW-VALUE > α, then

$\alpha = \text{NEW-VALUE}$
BEST-PATH = PATH(RESULT) + S

if $\alpha \geq \beta$ then return

VALUE = $\alpha$
PATH = BEST-PATH

return

VALUE = $\alpha$
PATH = BEST-PATH
**Opponent**(Position, Depth, α, β):

for each $S \in$ SUCCESSORS(Position) do

RESULT = Player(S, Depth + 1, α, β)

NEW-VALUE = VALUE(RESULT)

if NEW-VALUE < β, then

$\beta = $ NEW-VALUE

BEST-PATH = PATH(RESULT) + S

if $\beta \leq \alpha$ then return

VALUE = β
PATH = BEST-PATH

return

VALUE = β
PATH = BEST-PATH
Any-Player(Position, Depth, \( \alpha \), \( \beta \)):
for each \( S \in \text{SUCCESORS}(\text{Position}) \) do

\[
\text{RESULT} = \text{Any-Player}(S, \text{Depth} + 1, -\beta, -\alpha)
\]

\[
\text{NEW-VALUE} = -\text{VALUE}(\text{RESULT})
\]

if \( \text{NEW-VALUE} > \alpha \), then

\[
\alpha = \text{NEW-VALUE}
\]

\[
\text{BEST-PATH} = \text{PATH}(\text{RESULT}) + S
\]

if \( \alpha \geq \beta \) then return

\[
\text{VALUE} = \alpha
\]

\[
\text{PATH} = \text{BEST-PATH}
\]

return

\[
\text{VALUE} = \alpha
\]

\[
\text{PATH} = \text{BEST-PATH}
\]
Adding Alpha-Beta Cutoffs

**MINIMAX-A-B** (Position, Depth, Player, UseTd, PassTd):

- **UseTd**: checked for cutoffs.
- **PassTd**: current best value
Adding Alpha-Beta Cutoffs

1. if DEEP-ENOUGH(Position, Depth), then return:
   
   VALUE = STATIC(Position, Player)
   PATH = nil

2. SUCCESSORS = MOVE-GEN(Position, Player)

3. if SUCCESSORS is empty, then do as in Step 1
Adding Alpha-Beta Cutoffs

4. if SUCCESSORS is not empty:
   \[
   \text{RESULT-SUCC} = \text{MINIMAX-A-B(SUCC, Depth + 1, Opp(Player), } \neg \text{PassTd, } \neg \text{UseTd)}
   \]

   \[
   \text{NEW-VALUE} = -\text{VALUE(RESULT-SUCC)}
   \]

   if NEW-VALUE > PassTd, then:
   \[
   \text{PassTd} = \text{NEW-VALUE}
   \]

   \[
   \text{BEST-PATH} = \text{PATH(RESULT-SUCC)} + \text{SUCC}
   \]

   if PassTd ≥ UseTd, then return:
   \[
   \text{VALUE} = \text{PassTd}
   \]

   \[
   \text{PATH} = \text{BEST-PATH}
   \]

5. Return:
   \[
   \text{VALUE} = \text{PassTd}
   \]

   \[
   \text{PATH} = \text{BEST-PATH}
   \]
Additional Refinements

- Futility cutoffs
- Waiting for quiescence
- Secondary search
- Using book moves
- Not assuming opponent’s optimal move
Iterative Deepening

Iteration 1

Iteration 2

Iteration 3
Iterative Deepening

• Search can be aborted at any time and the best move of the previous iteration is chosen.

• Previous iterations can provide invaluable move-ordering constraints.
Iterative Deepening

- Can be adapted for single-agent search.
- Can be used to combine the best aspects of depth-first search and breadth-first search.
Iterative Deepening

Depth-First Iterative Deepening (DFID)

1. Set SEARCH-DEPTH = 1

2. Conduct depth-first search to a depth of SEARCH-DEPTH. If a solution path is found, then return it.

3. Increment SEARCH-DEPTH by 1 and go to step 2.
Iterative Deepening

**Iterative-Deepening-A* (IDA*)**

1. Set $\text{THRESHOLD} = \text{heuristic evaluation of the start state}$

2. Conduct depth-first search, pruning any branch when its total cost exceeds $\text{THRESHOLD}$. If a solution path is found, then return it.

3. Increment $\text{THRESHOLD}$ by the minimum
Iterative Deepening

- Is the process wasteful?
Homework

Presentations
– Specific games: Chess – State of the Art

Exercises 1-7, 9 (Chapter 12)
Solving problems by searching

Chapter 3
Outline

- Problem-solving agents
- Problem types
- Problem formulation
- Example problems
- Basic search algorithms
Problem-solving agents

```plaintext
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 ← UPDATE-STATE(state, percept)
  if seq is empty then do
    goal ← FORMULATE-GOAL(state)
    problem ← FORMULATE-PROBLEM(state, goal)
    seq ← SEARCH(problem)
    action ← FIRST(seq)
    seq ← REST(seq)
  return action
```
Example: Romania

- 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
  -
Example: Romania
Problem types

• Deterministic, fully observable → single-state problem
  – Agent knows exactly which state it will be in; solution is a sequence

• Non-observable → sensorless problem (conformant problem)
  – Agent may have no idea where it is; solution is a sequence

• Nondeterministic and/or partially observable → contingency problem
  – percepts provide new information about current state
  – often interleave search, execution

• Unknown state space → exploration problem
Example: vacuum world

- Single-state, start in #5.

Solution?
Example: vacuum world

- **Single-state**, start in #5. 
  **Solution?** [Right, Suck]
- 
- **Sensorless**, start in 
  \( \{1,2,3,4,5,6,7,8\} \) e.g., 
  **Right goes to** \( \{2,4,6,8\} \)  
  **Solution?**
Example: vacuum world

• Sensorless, 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
  – Nondeterministic: Suck may dirty a clean carpet
  – Partially observable: location, dirt at current location
  – Percept: [L, Clean], i.e., start in #5 or #7
  Solution?
Example: vacuum world

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

- **Contingency**
  - Nondeterministic: *Suck* may dirty a clean carpet
  - Partially observable: location, dirt at current location.
  - Percept: \[L, \text{Clean}\], i.e., start in #5 or #7
    **Solution?** \[\text{[Right, if dirt then Suck]}\]
Single-state problem formulation

A problem is defined by four items:

1. **Initial state** e.g., "at Arad"
2. **Actions or successor function** $S(x) = \text{set of action–state pairs}$
   - e.g., $S(Arad) = \{<Arad \rightarrow Zerind, Zerind>, \ldots \}$
3. **Goal test**, can be
   - **Explicit**, e.g., $x = \text{"at Bucharest"}$
   - **Implicit**, e.g., $\text{Checkmate}(x)$
4. **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
  \[ \rightarrow \text{state space must be abstracted} \] for problem solving

- (Abstract) state = set of real states

- (Abstract) action = complex combination of real actions
  - e.g., "Arad \rightarrow Zerind" represents a complex set of possible routes, detours, rest stops, etc.

- For guaranteed realizability, any real state "in Arad“ must get to some real state "in Zerind"

- (Abstract) solution =
  - set of real paths that are solutions in the real world

- Each abstract action should be "easier" than the original problem
Vacuum world state space graph

- states?
- actions?
- goal test?
- path cost?
Vacuum world state space graph

- **states?** integer dirt and robot location
- **actions?** Left, Right, Suck
- **goal test?** no dirt at all locations
- **path cost?** 1 per action
Example: The 8-puzzle

- states?
- actions?
- goal test?
- path cost?
Example: The 8-puzzle

- **states**? locations of tiles
- **actions**? move blank left, right, up, down
- **goal test**? = goal state (given)
- **path cost**? 1 per move

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

- **states?**: real-valued coordinates of robot joint angles
  parts of the object to be assembled
  -
  - **actions?**: continuous motions of robot joints
    -
    - **goal test?**: complete assembly
      -
      - **path cost?**: time to execute
Tree search algorithms

• Basic idea:
  – offline, simulated exploration of state space by generating successors of already-explored states (a.k.a. expanding states)

```plaintext
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
```
Tree search example
Tree search example
Tree search example
Implementation: 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 SOLUTION(node)
        fringe ← INSERT-ALL(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
Implementation: states vs. nodes

- A **state** is a (representation of) a physical configuration
- A **node** is a data structure constituting part of a search tree includes state, parent node, action, path cost $g(x)$, depth

- The **Expand** function creates new nodes, filling in the various fields and using the **SuccessorFn** of the problem to create the corresponding states.
Search strategies

• A search 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
  – 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

- **Uninformed** search strategies use only the information available in the problem definition
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
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
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 of breadth-first search

- **Complete?** Yes (if $b$ is finite)
- **Time?** $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d-1) = O(b^{d+1})$
- **Space?** $O(b^{d+1})$ (keeps every node in memory)
- **Optimal?** Yes (if cost = 1 per step)
- **Space** is the bigger problem (more than time)
Uniform-cost search

- Expand least-cost unexpanded node

- **Implementation:**
  - fringe = queue ordered by path cost

- Equivalent to breadth-first if step costs all equal

- **Complete?** Yes, if step cost ≥ \( \varepsilon \)

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

- **Space?** # of nodes with \( g \leq \text{cost of optimal solution} \), \( O(b^{\lceil C^*/\varepsilon \rceil}) \)
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 = 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 = LIFO queue, i.e., put successors at front
Depth-first search

- Expand deepest unexpanded node

- **Implementation:**
  - fringe method
  - use LIFO queue
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
Properties of depth-first search

- **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!
Depth-limited search

= depth-first search with depth limit \( l \),
i.e., nodes at depth \( l \) have no successors

```
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 SOLUTION(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

function Iterative-Deepening-Search( problem) returns a solution, or failure

inputs: problem, a problem

for depth ← 0 to ∞ do
    result ← Depth-Limited-Search( problem, depth)
    if result ≠ cutoff then return result
Iterative deepening search / =0
Iterative deepening search \( l = 1 \)
Iterative deepening search $l = 2$
Iterative deepening search \( l = 3 \)
Iterative deepening search

- Number of nodes generated in a depth-limited search to depth $d$ with branching factor $b$:
  \[ N_{DLS} = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d \]

- Number of nodes generated in an iterative deepening search to depth $d$ with branching factor $b$:
  \[ N_{IDS} = (d+1)b^0 + d b^1 + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d \]

- For $b = 10, d = 5$,
  - $N_{DLS} = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111$
  - $N_{IDS} = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456$

- Overhead = \( \frac{123,456 - 111,111}{111,111} = 11\% \)
Properties of iterative deepening search

- **Complete?** Yes
- **Time?** $(d+1)b^0 + d b^1 + (d-1)b^2 + ... + b^d = O(b^d)$
- **Space?** $O(bd)$
- **Optimal?** Yes, if step cost = 1
# Summary of algorithms

<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>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>$O(b^{d+1})$</td>
<td>$O(b^{C*/\varepsilon})$</td>
<td>$O(b^m)$</td>
<td>$O(b^l)$</td>
<td>$O(b^d)$</td>
</tr>
<tr>
<td>Space</td>
<td>$O(b^{d+1})$</td>
<td>$O(b^{C*/\varepsilon})$</td>
<td>$O(bm)$</td>
<td>$O(bl)$</td>
<td>$O(bd)$</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Repeated states

- Failure to detect repeated states can turn a linear problem into an exponential one!
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 Solution(node)
    if State[node] is not in closed then
        add State[node] to closed
        fringe ← InsertAll(Expand(node, problem), fringe)
Summary

• Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored
  •

• Variety of uninformed search strategies
  •

• Iterative deepening search uses only linear space and not much more time than other uninformed algorithms
  •