Best-First Search

- **Idea:** use a function $f$ for each node $n$ to estimate of "desirability"
- **Strategy:** Always expand unexpanded node $n$ with least $f(n)$
- **Implementation:** fringe is a priority queue sorted in decreasing order of $f(n)$
- **Special cases:**
  - Greedy search: $f(n) = h(n)$
  - A* search: $f(n) = g(n) + h(n)$

where $g(n)$ is the real cost from the initial node to $n$ and $h(n)$ is an estimate of the cost from $n$ to a goal.

Optimality/Completeness of A* Search

If the problem is solvable, A* always finds an optimal solution when

- the standard assumptions are satisfied,
- the heuristic function is admissible.

A* is optimally efficient for any heuristic function $h$: No other optimal strategy expands fewer nodes than A*.

Artificial Intelligence

Informed Search and Exploration

**Readings:** Chapter 4 of Russell & Norvig.

Properties of Heuristic Function

- **Admissible:** $h(n) \leq h^*(n)$ for any node, where $h^*(n)$ is the true cost from $n$ to the nearest goal.
- **Consistent:** $h(n) + c(n, a, n') \leq h(n')$, where $n' = \text{RESULT}(A, N)(n)$.
How to Obtain Admissible Heuristics

- Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem
- Example: the 8-puzzle
  - $h_1(n) =$ number of misplaced tiles
    - How: If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then $h_1(n)$ gives the shortest solution
  - $h_2(n) =$ total Manhattan distance (i.e., number of squares from desired location of each tile)
    - How: If the rules are relaxed so that a tile can move to any adjacent square, then $h_2(n)$ gives the shortest solution.
- Key point: the optimal cost of a relaxed problem is no greater than the optimal cost of the real problem.

Complexity of A* Search

- Worst-case time complexity: still exponential ($O(b^d)$) unless the error in $h$ is bounded by the logarithm of the actual path cost.
  - That is, unless
    $$|h(n) - h^*(n)| \leq O(\log h^*(n))$$
  - Where $h^*(n) =$ actual cost from $n$ to goal.
- Worst-Case Space Complexity: $O(b^m)$ as in greedy best-first.
- A* generally runs out of memory before running out of time.
- Improvements: IDA*, SMA*.

Local Search Algorithms

- In many optimization problems, such as $n$–queens, path is irrelevant; the goal state itself is the solution
- Then state space = set of “complete” configurations; find optimal configuration, e.g., TSP or, find configuration satisfying constraints, e.g., timetable
- In such cases, can use local search (or iterative improvement) algorithms; keep a single “current” state, try to improve it.
- It uses constant space, suitable for online as well as offline search.

Relaxed Problems

- Well-known example: travelling salesperson problem (TSP)
  - Find the shortest tour visiting all cities exactly once

Minimum spanning tree can be computed in $O(n^2)$ and is a lower bound on the shortest (open) tour.
Local Search Example: \( n \)-queens
- Put \( n \) queens on an \( n \times n \) board with no two queens on the same row, column, or diagonal
- Move a queen to reduce number of conflicts

Hill-Climbing
- Problem: depending on initial state, can get stuck on local maxima

Hill-Climbing (or Gradient Descent)
- “Like climbing Everest in thick fog with amnesia”

```
function Hill-Climbing(problem) return state
    node: current, neighbor;
    current := Make-Node(Initial-State(problem));
    loop do
        neighbor := highest-value-successor(current)
        if (Value(neighbor) < Value(current))
            then return State(current)
        else current := neighbor
    end loop
end function
```

The returned state is a local maximum state.

Local Search Example: TSP
- TSP: Travelling Salesperson Problem
- Start with any complete tour, perform pairwise exchanges

Hill-Climbing
- Problem: depending on initial state, can get stuck on local maxima

Hill-Climbing (or Gradient Descent)
- “Like climbing Everest in thick fog with amnesia”

```
function Hill-Climbing(problem) return state
    node: current, neighbor;
    current := Make-Node(Initial-State(problem));
    loop do
        neighbor := highest-value-successor(current)
        if (Value(neighbor) < Value(current))
            then return State(current)
        else current := neighbor
    end loop
end function
```

The returned state is a local maximum state.
**Simulated Annealing Algorithm**

```plaintext
function Simulated-Annealing(problem, schedule)
    return state
node: current, neighbor; integer: t, T;
current := Make-Node(Initial-State(problem));
for t := 1 to MAX-ITERATION do
    T := schedule[t];
    if (T == 0) return State(current);
    neighbor := random-successor(current);
    if (Value(neighbor) < Value(current))
        then v := Value(neighbor) - Value(current);
        current := neighbor with Prob(exp(v/T))
    else current := neighbor;
end for
end function
```

**Simulated Annealing**

- Idea: escape local maxima by allowing some “bad” moves but gradually decrease their size and frequency.
- Realize it by a random assignment like
  
  \[
  \text{current} := \text{neighbor with } \text{Prob}(\exp(v/T))
  \]

  where \( v < 0 \) and \( |v| \) is the “size”; \( T > 0 \) is the “frequency”.
- Because \( v < 0 \) and \( T > 0 \), \( 0 < e^{v/T} < 1 \).
- How to implement such a random assignment:
  - Generate a random real number \( x \) in \([0..1]\).
  - If \( x \geq e^{v/T} \), then do the assignment; otherwise, do nothing.

**Exercise 3.1**

Define in your own words the following items: state, state space, search tree, search node, action, successor function, and branching factor.

**Properties of Simulated Annealing**

- The smaller \( |v| \), the greater \( e^{v/T} \); the greater \( T \), the greater \( e^{v/T} \).
- At fixed “temperature” \( T \), state occupation probability reaches Boltzman distribution

  \[
  p(x) = \alpha e^{\frac{E(x)}{kT}}
  \]

  \( T \) decreased slowly enough ⇒ always reach best state

- Is this necessarily an interesting guarantee??
- Devised by Metropolis et al., 1953, for physical process modelling
- Widely used in VLSI layout, airline scheduling, etc.
Exercise 3.1

- State: A condition of (a problem or an agent) being in a stage or form
- State space: A collection of all states
- Action: An act which causes a state to change to another state, called successor.
- Successor function: returns all the successors of a state
- Branching factor: The maximum number of successors of any state
- Search tree: A graphical representation of the order of successors explored from the initial state.
- Search node: A node in the search tree which contains the information of a state and its location in the tree.

Exercise 3.3

Suppose that \( \text{LEGAL-ACTION}(s) \) denotes the set of actions that are legal in state \( s \), and \( \text{RESULT}(a, s) \) denotes the state that results from performing a legal action \( a \) in state \( s \). Define \( \text{SUCCESSOR-FN} \) in terms of \( \text{LEGAL-ACTIONS} \) and \( \text{RESULT} \), and \textit{vice versa}.

Exercise 3.4

Show that the 8-puzzle states are divided into two disjoint sets, such that no state in one set can be transformed into a state in the other set by any number of moves. Devise a procedure that will tell you which class a given state is in, and explain why this is a good thing to have for generating random states.

Exercise 3.3

Suppose that \( \text{LEGAL-ACTION}(s) \) denotes the set of actions that are legal in state \( s \), and \( \text{RESULT}(a, s) \) denotes the state that results from performing a legal action \( a \) in state \( s \). Define \( \text{SUCCESSOR-FN} \) in terms of \( \text{LEGAL-ACTIONS} \) and \( \text{RESULT} \), and \textit{vice versa}.

\[
\text{SUCCESSOR-FN}(s) = \{ \text{RESULT}(a, s) \mid a \in \text{LEGAL-ACTIONS}(s) \}
\]

\[
\text{LEGAL-ACTIONS}(s) = \{ a \mid \text{RESULT}(a, s) \in \text{SUCCESSOR-FN}(s) \}
\]
Exercise 3.4

- Define the ordering on the squares. An “inversion” is a pair of tiles such that the tile with bigger number precedes the tile with smaller number in the given order.
- Compute the numbers of inversions of starting state and goal states.
- If these numbers are both even or both odd, then a solution is possible; otherwise, it’s impossible.


Exercise 3.6

- Does a finite state space always lead to a finite search tree? How about a finite state space that is a tree? Can you be more precise about what types of state spaces always lead to finite search trees?

> initial state
> goal test
> successor function
> cost function

Exercise 3.7b

A 3-foot-tall monkey is in a room where some bananas are suspended from the 8-foot ceiling. Hw would like to get the bananas. The room contains two stackable, movable, climbing 3-foot-high crates.

- initial state
- goal test
- successor function
- cost function

Exercise 3.6

- Does a finite state space always lead to a finite search tree? No. Without checking duplicates in a path, a loop may occur.
- How about a finite state space that is a tree? Yes.
- Can you be more precise about what types of state spaces always lead to finite search trees? Acyclic graphs of successor relations will always lead to finite search trees. Finite state space with duplicate checking in a path will always lead to finite search trees.
Exercise 3.7b

**initial state:** Initially, four objects, \(m\) (monkey), \(b\) (bananas), \(c_1\) (crate1), \(c_2\) (crate2), are at four different locations \(l_1, l_2, l_3,\) and \(l_4,\) respectively. We use \((l_1, l_2, l_3, l_4)\) as the initial state.

**goal test:** To get bananas, crate2 must be at location \(l_2;\) crate1 must be on top of crate2 \((c_2);\) monkey must be on top of crate1 \((c_1).\) That is, the goal state is \((c_1, l_2, c_2, l_2).\) Switching crate1 and crate2, we have another goal state: \((c_2, l_2, l_2, c_1).\)

**successor function:** There are many.
1. \(s((l_1, l_2, l_3, l_4)) = \{(l_2, l_2, l_3, l_4), (l_3, l_2, l_3, l_4), (l_4, l_2, l_3, l_4)\}\)
2. \(s((l_3, l_2, l_3, l_4)) = \{(l_2, l_2, l_2, l_4), \ldots\}\)
3. \(s((l_2, l_2, l_2, l_4)) = \{(l_4, l_2, l_2, l_4), \ldots\}\)
4. \(s((l_4, l_2, l_2, l_4)) = \{(l_2, l_2, l_2, l_2), \ldots\}\)
5. \(s((l_2, l_2, l_2, l_2)) = \{(l_2, l_2, c_2, l_2), \ldots\}\)
6. \(\ldots\)

**cost function:** 1 for each action.