## Informed/Heuristic Search

### Algorithms

- **Best-First Search** ......................................... 2
- **General Best-First Search Algorithm** ..................... 3
- **A* Search Algorithm** ......................................... 4
- **Iterative Deepening A* Search Algorithm** ................ 5
- **IDA*'s Contour Procedure** .................................. 6

### Analysis

- **Properties of A* Search** ...................................... 7
- **Optimality Proof** ................................................ 8
- **Efficiency of A*** .................................................. 9
- **Performance of Heuristic Functions** .......................... 10
- **Book Experiment Avoiding Reverse Moves** ................. 11

### Local Search

- **Local Search** ..................................................... 12
- **Local Optima Example** ......................................... 13
- **Local Search Algorithm** ......................................... 14
- **Examples of Local Search Algorithms** ................. 15

---

## Algorithms

### Best-First Search

- Simple search algorithms such as IDS do not consider the goodness of states.
- Informed/heuristic search prefers to visit states that appear to be better.
- Best-first search visits states according to an evaluation function.
- An evaluation function $f$ gives lower numbers to (seemingly) better states.
- The evaluation function for A* search is the cost $g$ from the initial state plus a heuristic function $h$.
- A heuristic function estimates the cost from a given state to the closest goal state.

### General Best-First Search Algorithm

```plaintext
function Best-FS(s_initial, EXPAND, GOAL, f)
    q ← New-Priority-Queue()
    Insert(s_initial, q, f(s_initial))
    while q is not empty
        s_current ← Extract-Min(q)
        if GOAL(s_current) then return solution
        for each s_successor in EXPAND(s_current)
            do Insert(s_successor, q, f(s_successor))
    return failure
```

---

CS 3793 Artificial Intelligence

Heuristic Search – 2

Heuristic Search – 3
**A* Search Algorithm**

```python
function A*(s_initial, Expand, Goal, g, h)
    q ← NEW-PRIORITY-QUEUE()
    INSERT(s_initial, q, h(s_initial))
    while q is not empty do
        s_current ← EXTRACT-MIN(q)
        if GOAL(s_current) then return solution
        for each s_successor in EXPAND(s_current) do
            INSERT(s_successor, q, g(s_successor) + h(s_successor))
    return failure
```

---

**Iterative Deepening A* Search Algorithm**

```python
function IDA*(s_initial, Expand, Goal, g, h)
    limit ← HEURISTIC(s_initial)
    loop
        do solution, limit ← CONTOUR(s_initial, limit)
            if solution ≠ null then return solution
            if limit = ∞ then return failure
```

---

**Properties of A* Search**

- □ n: variable standing for a state
- □ g(n): the cost from the initial state to n.
- □ h(n): the estimate from n to a goal state.
- □ f(n) = g(n) + h(n).
- □ Each action costs at least 1 unit.
- □ Number of actions are finite.
- □ h is admissible (h is never an overestimate).
- □ Under above conditions, A* finds optimal path.
- □ If above conditions, h has at most \( \epsilon \) error, and the search space is a uniform tree with one goal state, then A* searches at most \( \epsilon/2 \) from solution path.
Optimality Proof

- Let \( f^* \) be optimal path cost.
- Because \( h \) never overestimates, then all states \( n \) on optimal path have \( f(n) \leq f^* \).
- Any nonoptimal goal state \( n' \) has \( f(n') > f^* \).
- Because of priority queue, \( A^* \) will visit states on optimal path before any nonoptimal goal state.
- Other conditions prevent infinite search in a flat region of the state space.

Efficiency of \( A^* \)

- Assume tree-structured state space (\( b = \) branching factor, \( d = \) goal depth), single goal state, each edge costs 1, and maximum error of \( \epsilon \).
- Any state \( n \) more than \( \epsilon/2 \) off of solution path has \( f(n) = g(n) + h(n) > f^* \).
- All states \( n \) on solution path have \( f(n) = g(n) + h(n) \leq f^* \).
- \( A^* \) and IDA* visit \( O(db^{\epsilon/2}) \) states.
- \( A^* \) uses \( O(db^{\epsilon/2}) \) memory. IDA* uses \( O(db) \).

Performance of Heuristic Functions

Consider these 8-puzzle heuristic functions:

- \( h_1 \): number of tiles in goal position.
- \( h_2 \): Manhattan distance from tiles to goals.
- Both never overestimate and \( h_1 \leq h_2 \)

Characterize by effective branching factor

- Let \( N \) states be visited and solution depth be \( d \).
- Solve for \( x \) in \( N = \sum_{x=0}^{d} x^i \)

Efficiency of A*

- Assume tree-structured state space (\( b = \) branching factor, \( d = \) goal depth), single goal state, each edge costs 1, and maximum error of \( \epsilon \).
- Any state \( n \) more than \( \epsilon/2 \) off of solution path has \( f(n) = g(n) + h(n) > f^* \).
- All states \( n \) on solution path have \( f(n) = g(n) + h(n) \leq f^* \).
- \( A^* \) and IDA* visit \( O(db^{\epsilon/2}) \) states.
- \( A^* \) uses \( O(db^{\epsilon/2}) \) memory. IDA* uses \( O(db) \).

Performance of Heuristic Functions

Consider these 8-puzzle heuristic functions:

- \( h_1 \): number of tiles in goal position.
- \( h_2 \): Manhattan distance from tiles to goals.
- Both never overestimate and \( h_1 \leq h_2 \)

Characterize by effective branching factor

- Let \( N \) states be visited and solution depth be \( d \).
- Solve for \( x \) in \( N = \sum_{x=0}^{d} x^i \)
Local Search Algorithm

```
function LOCAL-SEARCH(s_initial, EXPAND, GOAL, SELECT)
    s_current ← s_initial
    loop
        do if GOAL(s_current) then return solution
        s_current ← SELECT(EXPAND(s_current))
```

Examples of Local Search Algorithms

- **Hill-Climbing, Gradient Descent:** Select state improving an evaluation function.
- **Random-restart hill-climbing:** Repeat hill climbing from random initial states.
- **Stochastic Hill Climbing/Simulated Annealing:** Hill-climbing with randomized selection.
- **Beam Search/Genetic Algorithms:** Maintain a set of “current states.” GA crossover generates new states from pairs of states.
- **Tabu Search:** Like hill-climbing, but avoid recently visited states or recently used operators.