# ROB311 Quiz 1

Hanhee Lee

April 13, 2025

## Contents

# 1 Search Algorithms

**Summary:**

| **Alg.** | Halting | Sound | Complete | Optimal | Time | Space |
|---|---|---|---|---|---|---|
| Choose `REMOVE(·)` so algo. exhibits the characteristics:<br>• **Halting:** Terminates after finitely many nodes explored \| **Sound:** Returned (possibly NULL) soln. is correct<br>• **Complete:** Halting & sound when a non-NULL soln. exists \| **Opt.:** Returns an opt. soln. when mult. exist<br>• **Time:** Minimizes nodes **explored**/expanded/exported \| **Space:** Minimizes nodes simultaneously open | | | | | | |
| Choose `REMOVE(·)` so algo. exhibits the characteristics for as many path trees as possible.<br>• $b$ ($b < \infty$): Branching factor (the maximum number of children a node can have)<br>• $d$: Depth (the length of the longest path), $l^*$: Length of the shortest solution<br>• $c^*$: Cost of the cheapest solution, $\epsilon$: Cost of the cheapest edge | | | | | | |
| **Uninformed Search Algorithms** | | | | | | |
| **BFS** | $d < \infty$ \| non-`NULL` soln. | always | always | constant cst | $b^{l^*}$ | $b^{l^*+1}$ |
| • Explores the least-recently expanded open node first. | | | | | | |
| **DFS** | $d < \infty$ | always | $d < \infty$ | never | $b^d$ | $bd$ |
| • Explores the most-recently expanded open node first. | | | | | | |
| **IDDFS** | always | always | always | constant cst | $b^{l^*}$ | $bl^*$ |
| • Same as DFS but with iterative deepening. | | | | | | |
| **CFS** | $d < \infty$ \| non-`NULL` soln. | yes | $\epsilon > 0$ | $\epsilon > 0$ | $b^{c^*/\epsilon}$ | $b^{c^*/\epsilon+1}$ |
| • Explores the cheapest open node first. | | | | | | |
| **Informed Search Algorithms** | | | | | | |
| **HFS** | $d < \infty$ | never | never | never | - | - |
| • Explores the node with the smallest hur-value first, $\text{ecst}(p) = \text{hur}(p)$ | | | | | | |
| **A*** | hur admissible, $\epsilon > 0$ | always | hur admissible, $\epsilon > 0$ | hur admissible, $\epsilon > 0$ | $O\left(b^{c^*/\epsilon}\right)$ | $O\left(b^{c^*/\epsilon+1}\right)$ |
| • Explores the node with the smallest ecst-value first, $\text{ecst}(p) = \text{cst}(p) + \text{hur}(p)$ | | | | | | |
| **IIA*** | always | always | always | always | $b^{l^*}$ | $bl^*$ |
| • Same as A* but with iterative inflating on ecst. | | | | | | |
| **WA*** | - | - | - | - | - | - |
| • Same as A* but $\text{ecst}(s) = w\text{cst}(s) + (1-w)\text{hur}(s)$ w/ $w \in [0,1]$<br>• $w = 0$: HFS, $w = 0.5$: A*, $w = 1$: CFS, iteratively increasing $w$ from 0 to 1: anytime version of $WA^*$ | | | | | | |

## 1.1 Modifications to Search Algorithms:

**Summary**:

| **Modifications** |
|---|
| **Depth-Limiting**<br>• Enforce a depth limit, $d_{\max}$, to any search algorithm. |
| **Iterative-Deepening**<br>• Iteratively increase the depth-limit to any search algorithm w/ depth-limiting. |
| **Cost-Limiting**<br>• Enforce a cost limit of $c_{\max}$ to any search algorithm. |
| **Iterative Inflating**<br>• Iteratively increase the cost limit, $c_{\max}$, to any search algorithm w/ cost-limiting. |
| **Intra-Path Cycle Checking**<br>• Do not expand a path if it is cyclic. |
| **Inter-Path Cycle Checking**<br>• Do not expand a path if its destination is that of an explored path. |

### 1.2 Setup

**Definition:** In a search problem, it is assumed that:
- There is only one agent (us).
- For each state, $s \in S$, we have a discrete set of actions, $\mathcal{A}(s)$.
- The transition resulting from a move, $(s, a)$, is deterministic; the resulting state is $tr(s, a)$.
- $cst(s, a, tr(s, a))$ is our cost for the transition, $(s, a, tr(s, a))$.
- We want to realize a path that minimizes our cost.

A search problem may have no solutions, in which case, we define the solution as `NULL`.

**Warning:** A NULL solution is not the same as $p = \langle\rangle$ (an empty solution w/ $s^{(0)} \in \mathcal{G}$).

### 1.3 Search Graphs

**Definition:** In a search graph (a graph representing a search problem):
- $S$ is defined by the vertices.
- $\mathcal{G}$ is a subset of the vertices.
- $s^{(0)}$ is some vertex.
- $tr(\cdot, \cdot)$ and $\mathcal{T}$ are defined by the edges.
- $cst(\cdot, \cdot, \cdot)$ is defined by the edge weights.

### 1.4 Path Trees

**Definition:** A search algorithm explores a tree of possible paths.
- In such a tree, each node represents the path from the root to itself.
  - The node may also include other info (such as the path's origin, cost, etc).

### 1.5 Search Algorithms

**Algorithm:** All search algorithms follow the template below:

```
𝒪 ← {(⟨⟩, 0)}                                  ▷ initialize a set of open nodes
SEARCH(𝒪)
```

- $\langle\rangle$: Empty path, 0: Cost of empty path.

```
procedure SEARCH(𝒪)
    if 𝒪 = ∅ then
        return NULL                ▷ the search algorithm failed to find a path to a goal
    n ← REMOVE(𝒪)                                              ▷ "explore" a node n
    if DST(n) ∈ 𝒢 then
        return n                          ▷ the search algorithm found a path to a goal
    for n′ ∈ CHL(n) do                           ▷ "expand" n and "export" its children
        𝒪 ← 𝒪 ∪ {n′}
    SEARCH(𝒪)
```

- Explore: Remove a node from the open set.
- Expand: Generate the children of the node.
- Export: Add the children to the open set.

**Warning:** The key difference is in the order that $\text{Remove}(\cdot)$ removes nodes.

## 1.6 Modifications to Search Algorithms

### 1.6.1 Depth-Limiting

**Algorithm:**

```
procedure SEARCHDL(𝒪, d_max):
    if 𝒪 = ∅ then
        return NULL                    ▷ the search algorithm failed to find a path to a goal
    n ← REMOVE(𝒪)                                                  ▷ "explore" a node, n
    if dst(n) ∈ 𝒢 then
        return n                              ▷ the search algorithm found a path to a goal
    for n′ ∈ chl(n) do                              ▷ "expand" n and "export" its children
        if len(n′) ≤ d_max then                               ▷ unless the child is too long
            𝒪 ← 𝒪 ∪ {n′}
    SEARCHDL(𝒪, d_max)
```

### 1.6.2 Iterative Deepening

**Algorithm:**

```
procedure SEARCHID():
    n ← NULL
    d_max = 0
    ▷ while a solution has not been found, reset the open set, run the search algorithm, then increase the
    depth-limit
    while n = NULL do
        𝒪 ← {(⟨⟩, 0)}
        n ← SEARCHDL(𝒪, d_max)
        d_max ← d_max + 1
    return n
```

**Warning:** Increasing $d_{\text{max}}$ can be done in different ways.

### 1.6.3 Cost-Limiting

**Algorithm:**

```
procedure SEARCHCL(𝒪, c_max):
    if 𝒪 = ∅ then
        return NULL                    ▷ the search algorithm failed to find a path to a goal
    n ← REMOVE(𝒪)                                                  ▷ "explore" a node, n
    if dst(n) ∈ 𝒢 then
        return n                              ▷ the search algorithm found a path to a goal
    for n′ ∈ chl(n) do                              ▷ "expand" n and "export" its children
        if cst(n′) ≤ c_max then                          ▷ unless the child is too expensive
            𝒪 ← 𝒪 ∪ {n′}
    SEARCHCL(𝒪, c_max)
```

### 1.6.4 Iterative-Inflating

**Algorithm:**

```
procedure SEARCHII():
    n ← NULL
    c_max = 0
    ▷ while a solution has not been found, reset the open set, run the search algorithm, then increase the cost-limit
    while n = NULL do
        O ← {(⟨⟩, 0)}
        n ← SEARCHCL(O, c_max)
        c_max ← c_max + ϵ
    return n
```

**Warning:** Increasing $c_{\max}$ can be done in different ways.

### 1.6.5 Intra-Path Cycle Checking

**Algorithm:**

```
procedure SEARCH(O):
    if O = ∅ then
        return NULL
    n ← REMOVE(O)
    if dst(n) ∈ G then
        return n
    for n′ ∈ chl(n) do                        ▷ "expand" n and "export" its children
        if not CYCLIC(n′) then                ▷ unless the child is cyclic
            O ← O ∪ {n′}
    SEARCH(O)
```

- Optimately of an algorithm is preserved provided $\epsilon > 0$.

### 1.6.6 Inter-Path Cycle Checking

**Algorithm:**

```
procedure SEARCH(O, C):
    if O = ∅ then
        return NULL
    n ← REMOVE(O)
    C ← C ∪ {n}                               ▷ add n to the closed set
    if dst(n) ∈ G then
        return n
    for n′ ∈ chl(n) do                        ▷ "expand" n and "export" its children
        if n′ ∉ C then                        ▷ unless the child's destination is closed
            O ← O ∪ {n′}
    SEARCH(O, C)
```

and then call the algorithm as follows:

```
O ← {(⟨⟩, 0)}
C ← {}                                        ▷ initialize a set of closed vertices
SEARCH(O, C)
```

## 1.7 Informed Search Algorithms

### 1.7.1 Estimated Cost

**Definition**: $\text{ecst}(\cdot)$: estimate the total cost to a goal given a path, $p$, based on:

- $\text{cst}(p)$: Cost of path $p$
- $\text{hur}: S \to \mathbb{R}_+$: Estimate of the extra cost needed to get to a goal from $\text{dst}(p)$
  - $\text{hur}(s)$ estimates the cost to get to $\mathcal{G}$ from $s$ and $\text{hur}(p)$ means $\text{hur}(\text{dst}(p))$.
  - $\text{hur}^*(s)$: The true cost to get to $\mathcal{G}$ from $s$.

### 1.7.2 Admissible

**Motivation**: We want to find a heuristic that under estimates (i.e. make paths look better than they are) the costs, rather than over estimate (i.e. make paths look worse than they are).

- Least useful heuristic: $\text{hur}(s) = 0$ for all $s \in \mathcal{S}$ or any other constant.
- Most useful heuristic: $\text{hur}(s) = \text{hur}^*(s)$ for all $s \in \mathcal{S}$.

**Definition**: A heuristic, $\text{hur}(\cdot)$, is said to be **admissible** if

$$\text{hur}(s) \leq \text{hur}^*(s)$$

for all $s \in \mathcal{S}$ and

$$\text{hur}(s) = 0$$

for all $s \in \mathcal{G}$.

**Warning**: Never over-estimates the overall cost, but may still estimate the cost of individual transition.

### 1.7.3 Consistent

**Definition**: A heuristic, $\text{hur}(\cdot)$, is said to be **consistent** if

$$\underbrace{\text{hur}(s) - \text{hur}(\text{tr}(s,a))}_{\text{estimated cost of the transition } (s,a,\text{tr}(s,a))} \leq \underbrace{\text{cst}(s,a,\text{tr}(s,a))}_{\text{true cost of the transition, } (s,a,\text{tr}(s,a))}$$

for all $s \in \mathcal{S}$, and $a \in \mathcal{A}(s)$, and

$$\text{hur}(s) = 0$$

for all $s \in \mathcal{G}$.

**Warning**: Never over-estimates the cost of individual transitions (and hence the overall cost).

**Theorem**: If a heuristic, $\text{hur}(\cdot)$, is consistent, then it is also admissible.

### 1.7.4 Domination

**Definition**: If $\text{hur}_1$ and $\text{hur}_2$ are admissible, then:

- $\text{hur}_1$ **strongly dominates** $\text{hur}_2$ if for all $s \in \mathcal{S} \setminus \mathcal{G}$:
$$\text{hur}_1(s) > \text{hur}_2(s)$$
- $\text{hur}_1$ **weakly dominates** $\text{hur}_2$ if for all $s \in \mathcal{S}$:
$$\text{hur}_1(s) \geq \text{hur}_2(s)$$
and for some $s \in \mathcal{S}$:
$$\text{hur}_1(s) > \text{hur}_2(s)$$

**Notes**: Want the heuristic that dominates but is also admissible.

### 1.7.5 Designing Heuristics via Problem Relaxation

**Definition**: Let $\text{hur}^*_{\text{ori}}$ be the perfect heuristic for a search problem, and $\text{cst}^*_{\text{rel}}$ be the optimal cost for a relaxed version of the problem. Then
$$\text{cst}^*_{\text{rel}}(s) \leq \text{hur}^*_{\text{ori}}(s) \text{ for all } s \in \mathcal{S}.$$

### 1.7.6 Combining Heuristics

**Definition**: If $\{\text{hur}_k(\cdot)\}_k$ are admissible (or consistent), then $\max_k \{\text{hur}_k\}(\cdot)$ is also admissible (or consistent).

**Definition**: If $\text{hur}_{\max} \equiv \max\{\text{hur}_1, \text{hur}_2\}$, then if $\text{hur}_k$ is consistent:
$$\text{hur}_k(s) - \text{hur}_k(\text{tr}(s,a)) \leq \text{cst}(s,a,\text{tr}(s,a))$$
$$\text{hur}_{\max}(s) = \text{hur}_{\max}(\text{tr}(s,a)) - \text{cst}^*(s,a,\text{tr}(s,a))$$

### 1.7.7 Anytime Search Algorithms

**Definition**: An **anytime algorithm** finds a solution quickly (even if it is sub-optimal), and then iteratively improves it (if time permits).

## 1.8 Canonical Examples

### 1.8.1 How to setup a search problem?

**Process:**

1. Given a search graph, we need to define the following:
   - $\mathcal{S}$: set of vertices
   - $\mathcal{G}$: goal states (subset of $\mathcal{S}$)
   - $s^{(0)}$: initial state
   - $\mathcal{T}$: set of edges (defined by $\text{tr}(\cdot,\cdot)$)
     - $\text{tr}(\cdot,\cdot)$: transition function
   - $\text{cst}(\cdot,\cdot,\cdot)$: cost function (defined by edge weights)

**Example**:

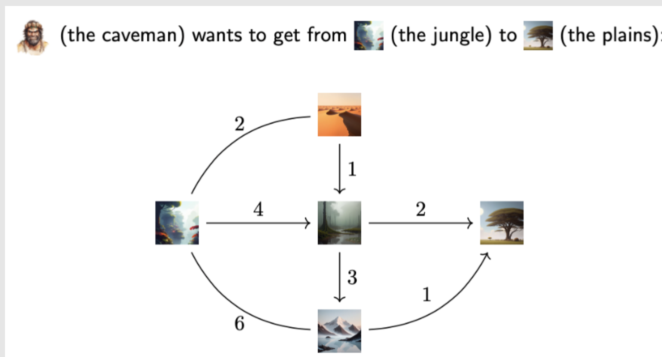



Figure 1

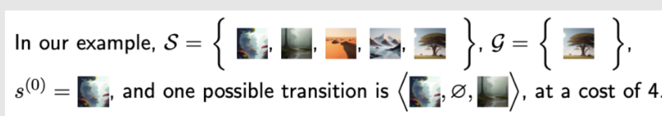



Figure 2

**Example**:

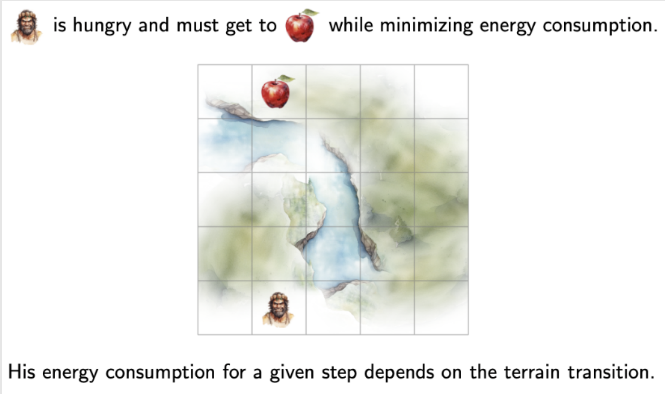



Figure 3

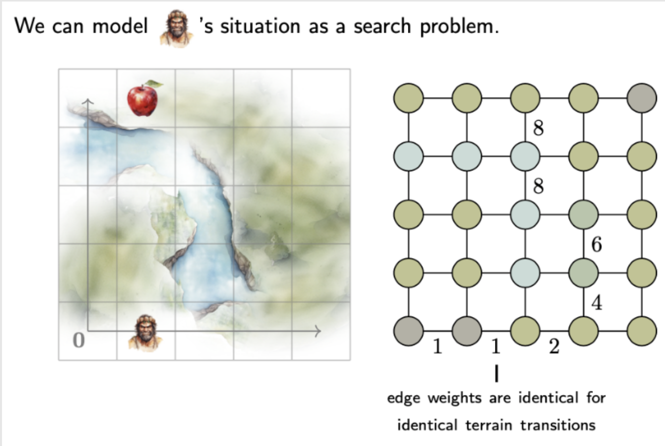



Figure 4

- $\mathcal{S} = \{0, \ldots, 4\}^2$
- $\mathcal{G} = \left\{ \begin{bmatrix} 1 \\ 4 \end{bmatrix} \right\}$
- $s^{(0)} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$

### 1.8.2 How to setup a path tree?

**Process:**

1. Start at $s^{(0)}$
2. Choose a path until you reach a goal state.
3. Repeat until you have found all paths (probably infinite).

**Example:**

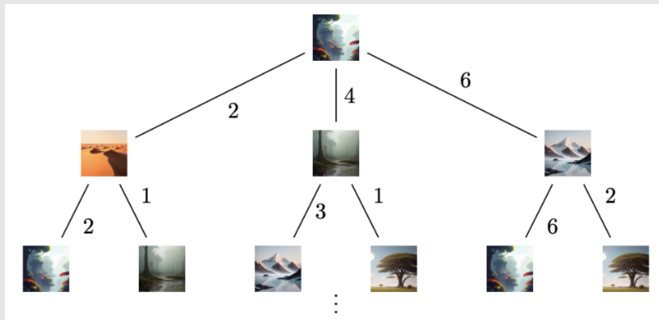



Figure 5

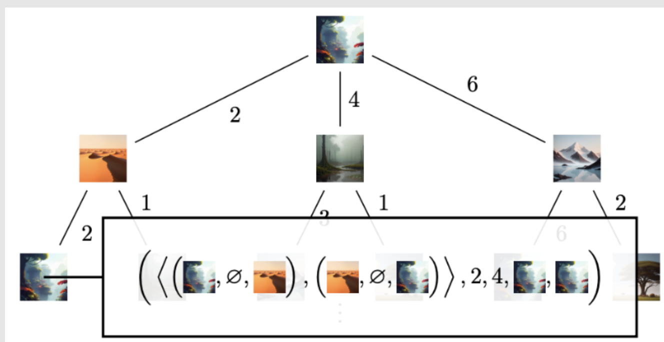



Figure 6

### 1.8.3 When to use each algorithm?

Process:

1. Do we have a heuristic?
   - **Yes:** Use informed search algorithms.
   - **No:** Use uninformed search algorithms.
2. Are path costs non-uniform?
   - **Yes:** Eliminate BFS.
   - **No:** Eliminate CFS, $A*$
3. 
4. Is the search space finite or infinite?
   - **Finite:** Use any algorithm.
   - **Infinite:** Use BFS, IDDFS, CFS, or A*.
5. Do we need to guarantee finding a solution (completeness)?
   - **Yes:** Use BFS, IDDFS, IIA*, CFS (if $\epsilon > 0$).
   - **No:** Use DFS, HFS, WA*
6. Find properties needed for the problem and match them to the characteristics of the algorithm.
7. Choose the algorithm that best matches the properties.
   - **BFS:** Need shortest path in an unweighted graph.
   - **DFS:** Explore a deep path quickly, and completeness is not needed.
   - **IDDFS:** Want completeness of BFS but with the complexity of DFS.
   - **CFS:** Need the least-cost path in a weighted graph.
   - **HFS:**
   - **A*:**
   - **IIA*:**
   - **WA*:**

### 1.8.4 Heuristic Availability

Process:

1. **No Heuristic:**
   - **BFS**, **DFS**, **IDDFS**, **CFS**.
2. **Yes, Heuristic Provided:**
   - **HFS**, **A***, **IIA***, **WA***.

### 1.8.5 Halting

Process:

1. **Guaranteed Halting (Under Finite Branching or Positive Costs):**
   - **BFS**, **IDDFS**, **CFS** ($\epsilon > 0$), **A*** ($\epsilon > 0$, admissible), **IIA***.
2. **No Guaranteed Halting (May Loop in Some Cases):**
   - **DFS**, **HFS**.

### 1.8.6 Completeness

Process:

1. **Complete Under Certain Conditions:**
   - **BFS** (finite depth), **IDDFS**, **CFS** ($\epsilon > 0$), **A*** (admissible heuristic), **IIA***.
2. **Not Guaranteed Complete:**
   - **DFS** (can miss solutions in infinite-depth spaces), **HFS** (can get stuck if heuristic is misleading).

#### 1.8.7 Optimality

Process:

1. **Optimal (Under Specific Assumptions):**
   - **BFS** (optimal in shallowest depth for uniform costs).
   - **CFS** (optimal if all edges have strictly positive cost).
   - **A**$^*$, **IIA**$^*$ (optimal if the heuristic is admissible and edge costs are $> 0$).
2. **Not Guaranteed Optimal:**
   - **DFS**, **HFS**, **WA**$^*$ (unless $w = 0.5$ with an admissible heuristic, but even then it may require careful tuning).

#### 1.8.8 Complexity

Process:

1. **Memory-Intensive But Faster to Find Solutions:**
   - **BFS**, **CFS**, **A**$^*$ (exponential growth in open list).
2. **More Memory-Efficient:**
   - **DFS** (linear in depth), **IDDFS**, **IIA**$^*$.

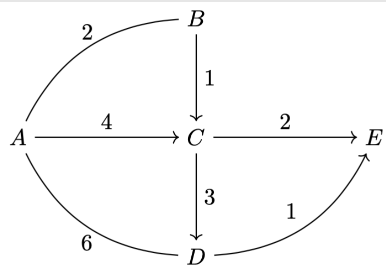
#### 1.8.9 Summary of Algorithm Selection

Process:

1. **No Heuristic, Must Halt, Complete, and Possibly Optimal (Uniform Cost):**
   - **BFS** (optimal in shallowest depth), **IDDFS** (similar but uses less space).
2. **No Heuristic, Low-Cost Path, Strictly Positive Edge Costs:**
   - **CFS** (cheapest-first).
3. **Heuristic Available, Need Completeness & Optimality, Positive Edge Costs:**
   - **A**$^*$ (admissible heuristic), **IIA**$^*$ (iterative improvement).
4. **Heuristic Available, Faster Non-Optimal Solution:**
   - **HFS**, **WA**$^*$ (anytime approach with varying weight).

### 1.8.10 Tracing Search Algorithms

**Example**:




$$\text{hur}\,(s) = \begin{cases} 3 & s = A \\ 2 & s = B \\ 1 & s = C \\ 1 & s = D \\ 0 & s = E \end{cases}$$

Figure 7

**Process: BFS**

1. Start at $s_0$ as **current node**
2. Expand all neighboring nodes of the **current node** and add them to the open set (queue).
3. Remove the **current node** from the open set and add it to the path.
4. Choose the least-recently expanded node from the open set as the **current node**.
5. Repeat steps 2 and 4 until the goal state is reached or the open set is empty.

**Example: BFS**

| **Path** | **Open Set** |
|---|---|
| | $\{A\}$ |
| $A$ | $\{AB, AC, AD\}$ |
| $AB$ | $\{AC, AD, ABA, ABC\}$ |
| $AC$ | $\{AD, ABA, ABC, ACD, ACE\}$ |
| $AD$ | $\{ABA, ABC, ACD, ACE, ADA, ADE\}$ |
| $ABA$ | $\{ABC, ACD, ACE, ADA, ADE, ABAB, ABAC, ABAD\}$ |
| $ABC$ | $\{ACD, ACE, ADA, ADE, ABAB, ABAC, ABAD, ABCD, ABCE\}$ |
| $ACD$ | $\{ACE, ADA, ADE, ABAB, ABAC, ABAD, ABCD, ABCE, ACDA, ACDE\}$ |
| $ACE$ | $\{ADA, ADE, ABAB, ABAC, ABAD, ABCD, ABCE, ACDA, ACDE\}$ |

**Intra:**

| **Path** | **Open Set** |
|---|---|
| | $\{A\}$ |
| $A$ | $\{AB, AC, AD\}$ |
| $AB$ | $\{AC, AD, ABC, \cancel{ABA}\}$ |
| $AC$ | $\{AD, ABC, ACD, ACE\}$ |
| $AD$ | $\{ABC, ACD, ACE, ADE, \cancel{ADA}\}$ |
| $ABC$ | $\{ACD, ACE, ADE, ABCD, ABCE\}$ |
| $ACD$ | $\{ACE, ADE, ABCD, ABCE, ACDE, \cancel{ACBA}\}$ |
| $ACE$ | $\{ADE, ABCD, ABCE, ACDE\}$ |

**Inter:**

| **Path** | **Open Set** | **Closed Set** |
|---|---|---|
| - | $\{A\}$ | - |
| $A$ | $\{AB, AC, AD\}$ | $\{A\}$ |
| $AB$ | $\{AC, AD, ABC, \cancel{ABA}\}$ | $\{A, B\}$ |
| $AC$ | $\{AD, ABC, ACD, ACE\}$ | $\{A, B, C\}$ |
| $AD$ | $\{ABC, ACD, ACE, ADE, \cancel{ADA}\}$ | $\{A, B, C, D\}$ |
| $ABC$ | $\{ACD, ACE, ADE, ABCE, \cancel{ABCD}\}$ | $\{A, B, C, D\}$ |
| $ACD$ | $\{ACE, ADE, ABCE, ACDE, \cancel{ACDA}\}$ | $\{A, B, C, D\}$ |
| $ACE$ | $\{ADE, ABCE, ACDE\}$ | $\{A, B, C, D, E\}$ |

**Process: DFS**

1. Start at $s_0$ as **current node**
2. Expand all neighboring nodes of the **current node** and add them to the open set (stack).
3. Remove the **current node** from the open set and add it to the path.
4. Choose the most-recently expanded node from the open set as the **current node**.
5. Repeat steps 2 and 4 until the goal state is reached or the open set is empty.

**Example: DFS**

| Path | Open Set |
|---|---|
| | $\{A\}$ |
| $A$ | $\{AB, AC, AD\}$ |
| $AD$ | $\{AB, AC, ADA, ADE\}$ |
| $ADE$ | $\{AB, AC, ADA\}$ |

**Intra:**

| Path | Open Set |
|---|---|
| - | $\{A\}$ |
| $A$ | $\{AB, AC, AD\}$ |
| $AD$ | $\{AB, AC, ADE, \cancel{ADA}\}$ |
| $ADE$ | $\{AB, AC\}$ |

**Inter:**

| Path | Open Set | Closed Set |
|---|---|---|
| - | $\{A\}$ | - |
| $A$ | $\{AB, AC, AD\}$ | $\{A\}$ |
| $AD$ | $\{AB, AC, ADE, \cancel{ADA}\}$ | $\{A, D\}$ |
| $ADE$ | $\{AB, AC\}$ | $\{A, D, E\}$ |

**Process: IDDFS**

1. Start with a depth limit of 0.
2. Perform DFS up to the current depth limit.
3. If the goal state is not reached, increment the depth limit based on given fcn and repeat step 2.
4. Continue until the goal state is found or all nodes are explored.

**Example: IDDFS**

| Depth | Path | Open Set |
|---|---|---|
| 0 | | $\{A\}$ |
| 0 | A | $\{\}$ |
| 1 | $A$ | $\{AB, AC, AD\}$ |
| 1 | $AD$ | $\{AB, AC\}$ |
| 1 | $AC$ | $\{AB\}$ |
| 1 | $AB$ | $\{\}$ |
| 2 | $AB$ | $\{ABA, ABC\}$ |
| 2 | $ABC$ | $\{ABA\}$ |
| 2 | $ABA$ | $\{\}$ |
| 3 | $ABA$ | $\{ABAB, ABAC, ABAD\}$ |
| 3 | $ABAB$ | $\{ABAC, ABAD\}$ |
| 3 | $ABAC$ | $\{ABAD\}$ |
| 3 | $ABAD$ | $\{\}$ |
| 4 | $ABAD$ | $\{ABADA, ABADE\}$ |
| 4 | $ABADA$ | $\{ABADE\}$ |
| 4 | $ABADE$ | $\{\}$ |

**Process: CFS**

1. Start at $s_0$ as **current node**
2. Expand all neighboring nodes of the **current node** and add them to the open set (priority queue).
3. Remove the **current node** from the open set and add it to the path.
4. Choose the cheapest expanded node from the open set as the **current node**.
5. Repeat steps 2 and 4 until the goal state is reached or the open set is empty.

**Example: CFS**

| Path | Open Set |
|---|---|
| - | $\{A \mid 0\}$ |
| $A$ | $\{AB \mid 2,\ AC \mid 4,\ AD \mid 6\}$ |
| $AB$ | $\{AC \mid 4,\ AD \mid 6,\ ABC \mid 3,\ ABA \mid 4\}$ |
| $ABC$ | $\{AC \mid 4,\ AD \mid 6,\ ABA \mid 4,\ ABCE \mid 5,\ ABCD \mid 6\}$ |
| $AC$ | $\{AD \mid 6,\ ABA \mid 4,\ ABCE \mid 5,\ ABCD \mid 6,\ ACD \mid 7,\ ACE \mid 6\}$ |
| $ABA$ | $\{AD \mid 6,\ ABCE \mid 5,\ ABCD \mid 6,\ ACD \mid 7,\ ACE \mid 6,\ ABAB \mid 6,\ ABAC \mid 8,\ ABAD \mid 10\}$ |
| $ABCE$ | $\{AD \mid 6,\ ABCD \mid 6,\ ACD \mid 7,\ ACE \mid 6,\ ABAB \mid 6,\ ABAC \mid 8,\ ABAD \mid 10\}$ |

**Intra:**

| Path | Open Set |
|---|---|
| - | $\{A \mid 0\}$ |
| $A$ | $\{AB \mid 2,\ AC \mid 4,\ AD \mid 6\}$ |
| $AB$ | $\{AC \mid 4,\ AD \mid 6,\ ABC \mid 3,\ \cancel{ABA}\}$ |
| $ABC$ | $\{AC \mid 4,\ AD \mid 6,\ ABCE \mid 5,\ ABCD \mid 6\}$ |
| $AC$ | $\{AD \mid 6,\ ABCE \mid 5,\ ABCD \mid 6,\ ACD \mid 7,\ ACE \mid 6\}$ |
| $ABCE$ | $\{AD \mid 6,\ ABCD \mid 6,\ ACD \mid 7,\ ACE \mid 6\}$ |

**Inter:**

| Path | Open Set | Closed Set |
|---|---|---|
| - | $\{A \mid 0\}$ | - |
| $A$ | $\{AB \mid 2,\ AC \mid 4,\ AD \mid 6\}$ | $\{A\}$ |
| $AB$ | $\{AC \mid 4,\ AD \mid 6,\ ABC \mid 3,\ \cancel{ABA}\}$ | $\{A, B\}$ |
| $ABC$ | $\{AC \mid 4,\ AD \mid 6,\ ABCE \mid 5,\ ABCD \mid 6\}$ | $\{A, B, C\}$ |
| $AC$ | $\{AD \mid 6,\ ABCE \mid 5,\ ABCD \mid 6,\ ACD \mid 7,\ ACE \mid 6\}$ | $\{A, B, C\}$ |
| $ABCE$ | $\{AD \mid 6,\ ABCD \mid 6,\ ACD \mid 7,\ ACE \mid 6\}$ | $\{A, B, C, E\}$ |

**Process: HFS**

1. Start at $s_0$ as **current node**
2. Expand all neighboring nodes of the **current node** and add them to the open set (priority queue).
3. Remove the **current node** from the open set and add it to the path.
4. Choose the lowest heuristic value expanded node from the open set as the **current node**.
5. Repeat steps 2 and 4 until the goal state is reached or the open set is empty.

**Example: HFS**

| **Path** | **Open Set** |
|---|---|
| | $\{A \mid 3\}$ |
| $A$ | $\{AB \mid 2,\ AC \mid 1,\ AD \mid 1\}$ |
| $AC$ | $\{AB \mid 2,\ AD \mid 1,\ ACE \mid 0\}$ |
| $ACE$ | $\{AB \mid 2,\ AD \mid 1\}$ |

**Process: A***

1. Start at $s_0$ as **current node**
2. Expand all neighboring nodes of the **current node** and add them to the open set (priority queue).
3. Remove the **current node** from the open set and add it to the path.
4. Choose the lowest $\text{esct}(p) = \text{cst}(p) + \text{hur}(p)$ expanded node from the open set as the **current node**.
5. Repeat steps 2 and 4 until the goal state is reached or the open set is empty.

**Example: A***

| Path | Open Set |
|---|---|
| - | $\{A \mid 3\}$ |
| $A$ | $\{AB \mid 2+2,\ AC \mid 4+1,\ AD \mid 6+1\}$ |
| $AB$ | $\{AC \mid 5,\ AD \mid 7,\ ABC \mid (2+1)+1,\ ABA \mid (2+2)+3\}$ |
| $ABC$ | $\{AC \mid 5,\ AD \mid 7,\ ABA \mid 7,\ ABCD \mid (2+1+3)+1,\ ABCE \mid (2+1+2)+0,\}$ |
| $AC$ | $\{AD \mid 7,\ ABA \mid 7,\ ABCD \mid 7,\ ABCE \mid 5,\ ACD \mid (4+3)+1,\ ACE \mid (4+2)+0\}$ |
| $ABCE$ | $\{AD \mid 7,\ ABA \mid 7,\ ABCD \mid 7,\ ACD \mid 8,\ ACE \mid 6\}$ |

**Intra:**

| Path | Open Set |
|---|---|
| - | $\{A \mid 3\}$ |
| $A$ | $\{AB \mid 2+2,\ AC \mid 4+1,\ AD \mid 6+1\}$ |
| $AB$ | $\{AC \mid 5,\ AD \mid 7,\ ABC \mid (2+1)+1,\ \cancel{ABA}\}$ |
| $ABC$ | $\{AC \mid 5,\ AD \mid 7,\ ABCD \mid (2+1+3)+1,\ ABCE \mid (2+1+2)+0,\}$ |
| $AC$ | $\{AD \mid 7,\ ABCD \mid 7,\ ABCE \mid 5,\ ACD \mid (4+3)+1,\ ACE \mid (4+2)+0\}$ |
| $ABCE$ | $\{AD \mid 7,\ ABCD \mid 7,\ ACD \mid 8,\ ACE \mid 6\}$ |

**Inter:**

| Path | Open Set | Closed Set |
|---|---|---|
| - | $\{A \mid 3\}$ | - |
| $A$ | $\{AB \mid 2+2,\ AC \mid 4+1,\ AD \mid 6+1\}$ | $\{A\}$ |
| $AB$ | $\{AC \mid 5,\ AD \mid 7,\ ABC \mid (2+1)+1,\ \cancel{ABA}\}$ | $\{A, B\}$ |
| $ABC$ | $\{AC \mid 5,\ AD \mid 7,\ ABCD \mid (2+1+3)+1,\ ABCE \mid (2+1+2)+0,\}$ | $\{A, B, C\}$ |
| $AC$ | $\{AD \mid 7,\ ABCD \mid 7,\ ABCE \mid 5,\ ACD \mid (4+3)+1,\ ACE \mid (4+2)+0\}$ | $\{A, B, C\}$ |
| $ABCE$ | $\{AD \mid 7,\ ABCD \mid 7,\ ACD \mid 8,\ ACE \mid 6\}$ | $\{A, B, C, E\}$ |

**Process: IIA***

1. Start with a cost limit of 0.
2. Perform A* up to the current cost limit.
3. If the goal state is not reached, increment the cost limit based on given fcn and repeat step 2.
4. Continue until the goal state is found or all nodes are explored.

**Example: IIA***

| **Cost** | **Path** | **Open Set** |
|---|---|---|
| 0 | $\langle\rangle$ | $\{\}$ |
| 1 | $\langle\rangle$ | $\{\}$ |
| 2 | $\langle\rangle$ | $\{\}$ |
| 3 | $\langle\rangle$ | $\{A \mid 3\}$ |
| 3 | $A$ | $\{\}$ |
| 4 | $A$ | $\{AB \mid 2+2\}$ |
| 4 | $AB$ | $\{ABC \mid 3+1\}$ |
| 4 | $ABC$ | $\{\}$ |
| 5 | $ABC$ | $\{ABCE \mid 5+0\}$ |
| 5 | $ABCE$ | $\{\}$ |

**Process: WA***

1. Start at $s_0$ as **current node**
2. Expand all neighboring nodes of the **current node** and add them to the open set (priority queue).
3. Remove the **current node** from the open set and add it to the path.
4. Choose the lowest $\text{esct}(p) = w \cdot \text{cst}(p) + (1 - w) \cdot \text{hur}(p)$ expanded node from the open set as the **current node**.
5. Repeat steps 2 and 4 until the goal state is reached or the open set is empty.

**Process: How to Prove Consistent/Admissible Given a Search Graph?**
**Admissible:**
1. Given $\text{hur}(s)$ and search graph with $\text{cst}(s, a, \text{tr}(s, a))$. If consistent, then it is admissible.
2. Check $\forall s \in \mathcal{G}$, $\text{hur}(s) = 0$. If not, then it is not admissible.
3. For each $s \in \mathcal{S}$, calculate $\text{hur}^*(s)$ (i.e. actual cost of optimal soln.) using the search graph.
   (a) Start at $s$ and choose path that gives the lowest cost to $s \in \mathcal{G}$.
4. Check if $\text{hur}(s) \leq \text{hur}^*(s)$ $\forall s \in \mathcal{S}$. If not, then it is not admissible.
5. Repeat $\forall s \in \mathcal{S}$.
6. If all are true, then it is admissible.

**Consistent:**
1. Given $\text{hur}(s)$ and search graph with $\text{cst}(s, a, \text{tr}(s, a))$.
2. Check $\forall s \in \mathcal{G}$, $\text{hur}(s) = 0$. If not, then it is not consistent.
3. For each $s \in \mathcal{S}$, calculate $\text{hur}(s) - \text{hur}(\text{tr}(s, a))$.
   (a) check if it is $\leq \text{cst}(s, a, \text{tr}(s, a))$. If not, then it is not consistent.
   (b) Repeat $\forall a \in \mathcal{A}(s)$
4. Repeat $\forall s \in \mathcal{S}$.
5. If all are true, then it is consistent.

**Warning:** Be careful of bidirectional edges bc for consistency you need compute the cost of the heuristic edge in both directions.

**Example**:

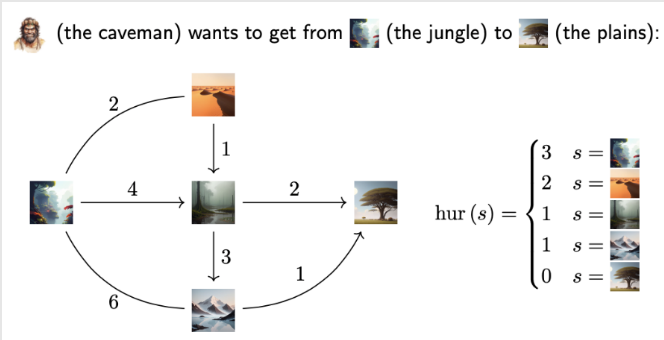



Figure 8: Jungle ($s^{(0)}$), Desert, Swamp, Mountain, Plains (Goal)

**Admissible:**

1. $s =$**Plains:** $\text{hur}(\text{Plains}) = 0$
2. $s =$**Jungle:** $\text{hur}(\text{Jungle}) = 3 \leq \text{hur}^*(\text{Jungle}) = 2 + 1 + 2 = 5$
3. $s =$**Desert:** $\text{hur}(\text{Desert}) = 2 \leq \text{hur}^*(\text{Desert}) = 1 + 2$
4. $s =$**Swamp:** $\text{hur}(\text{Swamp}) = 1 \leq \text{hur}^*(\text{Swamp}) = 2$
5. $s =$**Mountain:** $\text{hur}(\text{Mountain}) = 1 \leq \text{hur}^*(\text{Mountain}) = 1$
6. Therefore, it is admissible.

**Consistent:**

1. $s =$**Plains:** $\text{hur}(\text{Plains}) = 0$
2. $s =$**Jungle:**
   (a) $\text{hur}(\text{Jungle}) - \text{hur}(\text{Desert}) = 3 - 2 = 1 \leq \text{cst}(\text{Jungle}, \cdot, \text{Desert}) = 2$
   (b) $\text{hur}(\text{Jungle}) - \text{hur}(\text{Swamp}) = 3 - 1 = 2 \leq \text{cst}(\text{Jungle}, \cdot, \text{Swamp}) = 4$
   (c) $\text{hur}(\text{Jungle}) - \text{hur}(\text{Mountain}) = 3 - 1 = 2 \leq \text{cst}(\text{Jungle}, \cdot, \text{Mountain}) = 6$
3. $s =$**Desert:**
   (a) $\text{hur}(\text{Desert}) - \text{hur}(\text{Jungle}) = 2 - 3 = -1 \leq \text{cst}(\text{Desert}, \cdot, \text{Jungle}) = 2$
   (b) $\text{hur}(\text{Desert}) - \text{hur}(\text{Swamp}) = 2 - 1 = 1 \leq \text{cst}(\text{Desert}, \cdot, \text{Swamp}) = 1$
4. $s =$**Swamp:**
   (a) $\text{hur}(\text{Swamp}) - \text{hur}(\text{Mountain}) = 1 - 1 = 0 \leq \text{cst}(\text{Swamp}, \cdot, \text{Mountain}) = 3$
   (b) $\text{hur}(\text{Swamp}) - \text{hur}(\text{Plains}) = 1 - 0 = 1 \leq \text{cst}(\text{Swamp}, \cdot, \text{Plains}) = 2$
5. $s =$**Mountain:**
   (a) $\text{hur}(\text{Mountain}) - \text{hur}(\text{Jungle}) = 1 - 3 = -2 \leq \text{cst}(\text{Mountain}, \cdot, \text{Desert}) = 6$
   (b) $\text{hur}(\text{Mountain}) - \text{hur}(\text{Plains}) = 1 - 0 = 1 \leq \text{cst}(\text{Mountain}, \cdot, \text{Plains}) = 1$
6. Therefore, it is consistent.

**Process: How to Design Heuristic via Problem Relaxation?**

1. Make an assumption to simplify the problem as a relaxed problem.
2. Find the cost of the optimal solution of the relaxed problem, $\text{cst}_{\text{rel}}(s)$ from every state $s$ to the goal state.

**Example**:

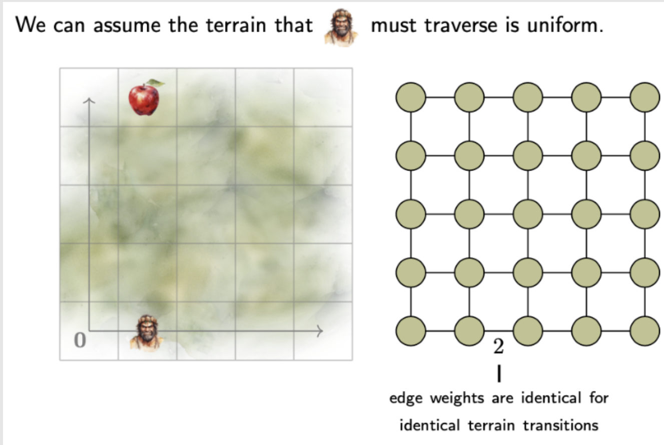



Figure 9