National University of Singapore
School of Computing
CS2109S: Introduction to AI and Machine Learning
Semester 2, 2024/2025

Tutorial 1 Solutions
Problem Formulation & Uninformed Search

Summary of Key Concepts

In this tutorial, we will discuss and explore the following key learning points/lessons:

1. Describing the problem environment with PEAS.
2. Formulating a search problem:
   (a) State representation
   (b) Actions
   (c) Goal state(s)
3. Uniform-cost Search
4. Iterative Deepening Search

A PEAS for AI chess player

1. Determine the PEAS (Performance measure, Environment, Actuators, Sensors) for an AI chess opponent in a mobile chess application.

Solution:

- **P (Performance Measure):** make "human" moves (ability to emulate human players with certain rankings (ELO)), win, draw, lose the game.
- **E (Environment):** chess board, position of game pieces, rules of the game, mobile operating system, human user's ranking (ELO), human user's playing style.
- **A (Actuators):** visual output to move a chess piece, or resign.
- **S (Sensors):** touch screen interactions to detect the change in position of pieces.

B Tower of Hanoi Problem

The Tower of Hanoi is a famous problem in computer science, described as follows: You are given 3 pegs (arranged left, middle, and right) and a set of n disks with holes in them (so they can slide onto the pegs). Every disk has a different diameter.

We start the problem with all the disks arranged on the left peg in order of size, with the smallest on top. The objective is to move all the disks from the left peg to the right peg in the *minimum number of moves*. This is done by:

- moving one disk at a time from one peg to another; and
- never placing a disk on top of another disk with a smaller diameter.

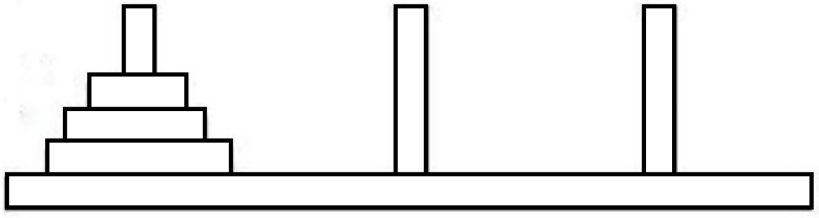

Figure 1: The initial state for $n = 3$

Figure 1 shows the initial state for $n = 3$. To get a better feel for the problem, you can try it here: https://www.mathsisfun.com/games/towerofhanoi.html.

1. (a) Propose a state representation for this problem if we want to formulate it as a search problem.

> **Solution:**
>
> **Important note:** While below, we have described a particular state representation for this problem, it is important to note that any state representation is acceptable as long as sufficient justification is given. Minimally, we must require that the state representation is **complete**: For any possible real-world configuration, there must be a corresponding state representation. In other words, a surjection must be formed between the set of all possible **valid** state representations (the validity of which is governed by your representation invariant), and the set of all possible real-world configurations.
>
> **State Representation:** We assign each disk an index from 1 to n, corresponding to its size (i.e., the largest disk is indexed as n, the second-largest disk as $n-1$, and so on, with the smallest disk indexed as 1).
>
> In this problem, we can define the state using 3 tuples (L, M, R): each tuple contains the indices of the disks on the left, middle, and right pegs, respectively, with each index from 1 to n contained in exactly one of these tuples.

   (b) Describe the representation invariant of your state (i.e., what condition(s) must a state satisfy to be considered valid). Compare the total number of possible configurations in the actual problem to that of your chosen state representation, with and without considering the representation invariant.

> **Solution:**
>
> **Representation Invariant:** Our representation invariant is that the indices in each tuple must be in **ascending** order. This invariant reduces the total number of unique, valid states of our representation from $n! \times \binom{n+2}{2} = \frac{(n+2)!}{2}$

(as proven in Appendix A) to exactly 3^n (as proven in Appendix B), matching the total number of configurations in the actual problem.

(c) Specify the initial and goal state(s) of your state representation.

Solution:

With this state representation in mind, our initial state is $((1, 2, \ldots, n), (), ())$ and the goal state is $((), (), (1, 2, \ldots, n))$.

(d) Define the actions.

Solution:

Actions: There are 6 actions in total:

1. Let a_1 denote the action of 'Transferring a disk from left to middle':
   $T(((l_1, \ldots), (m_1, \ldots), (\ldots)), a_1) \to ((\ldots), (l_1, m_1, \ldots), (\ldots)), l_1 < m_1$
   $T(((l_1, \ldots), (), (\ldots)), a_1) \to ((\ldots), (l_1), (\ldots))$
2. Let a_2 denote the action of 'Transferring a disk from left to right':
   $T(((l_1, \ldots), (\ldots), (r_1, \ldots)), a_2) \to ((\ldots), (\ldots), (l_1, r_1, \ldots)), l_1 < r_1$
   $T(((l_1, \ldots), (\ldots), ()), a_2) \to ((\ldots), (\ldots), (l_1))$
3. Let a_3 denote the action of 'Transferring a disk from middle to left':
   $T(((l_1, \ldots), (m_1, \ldots), (\ldots)), a_3) \to ((m_1, l_1, \ldots), (\ldots), (\ldots)), m_1 < l_1$
   $T((), (m_1, \ldots), (\ldots)), a_3) \to ((m_1), (\ldots), (\ldots))$
4. Let a_4 denote the action of 'Transferring a disk from middle to right':
   $T(((\ldots), (m_1, \ldots), (r_1, \ldots)), a_4) \to ((\ldots), (\ldots), (m_1, r_1, \ldots)), m_1 < r_1$
   $T(((\ldots), (m_1, \ldots), ()), a_4) \to ((\ldots), (\ldots), (m_1))$
5. Let a_5 denote the action of 'Transferring a disk from right to left':
   $T(((l_1, \ldots), (\ldots), (r_1, \ldots)), a_5) \to ((r_1, l_1, \ldots), (\ldots), (\ldots)), r_1 < l_1$
   $T((), (\ldots), (r_1, \ldots)), a_5) \to ((r_1), (\ldots), (\ldots))$
6. Let a_6 denote the action of 'Transferring a disk from right to middle':
   $T(((\ldots), (m_1, \ldots), (r_1, \ldots)), a_6) \to ((\ldots), (r_1, m_1, \ldots), (\ldots)), r_1 < m_1$
   $T(((\ldots), (), (r_1, \ldots)), a_6) \to ((\ldots), (r_1), (\ldots))$

Additionally, another way we could represent the state is with 3 **sets**, the difference being by definition, order does not matter for set equality. (e.g. $\{1, 2, 3\} = \{1, 3, 2\} = \{2, 1, 3\}$). Notice that inherently, the number of possible state representations is exactly 3^n even without enforcing a representation invariant.

(e) Using the state representation and actions defined above, state the transition function T (i.e., upon applying each of the actions defined above to a current state, what is the resulting next state?).

Solution:

Transition Function: Our transition function would be modified slightly.

In the case of a_1:

$$T((\{l_1, \ldots\}, \{\ldots\}, \{\ldots\}), a_1) \rightarrow (\{\ldots\}, \{l_1, \ldots\}, \{\ldots\}),$$
$$\forall l \in L \ (l_1 \leq l)$$
$$\text{and } \forall m \in M \ (l_1 < m)$$

Similar changes apply for the rest of the actions.

2. **Additional Question:** Given the nature of the Tower of Hanoi problem, which search algorithm would be most appropriate for finding the solution? Justify your answer.

Solution:

Branching Factor and Depth: The Tower of Hanoi problem has a branching factor of $b = 3$ since each valid action involves exactly 2 pegs, and for any pair among the k pegs, there is at most one valid action involving said pair (moving a top disk to an empty peg, or placing the smaller top disk on top of the larger top disk). Thus, $b = \binom{k}{2}$, and for $k = 3$, $b = \binom{3}{2} = 3$. The problem can result in infinite depth due to the potential for disks to move back and forth without progressing towards a solution.

Complexity of BFS: Breadth-First Search (BFS) explores all nodes at the current depth level before moving to the next level.

Time Complexity (Based on Nodes Generated): BFS has a time complexity of $O(b^d) = O(3^{2^n-1})$ since it generates all possible nodes at each level before moving deeper.

Memory Cost: BFS requires storing all nodes at the current depth level in memory, leading to a space complexity of $O(b^d) = O(3^{2^n-1})$.

Complexity of UCS: Uniform Cost Search (UCS) behaves similarly to BFS in this problem, since the cost for each move is uniform (1) .

Complexity of DFS: Depth-First Search (DFS) explores one branch fully before backtracking to explore others.

Time Complexity (Based on Nodes Generated): The time complexity of DFS is $O(b^m)$, where m is the maximum depth of the search tree. However, DFS can get trapped in deep, unproductive paths, leading to inefficiency in finding the optimal solution.

Memory Cost: DFS utilize $O(bm)$ memory in the frontier. This is because DFS must store all nodes along the current path from the root to the deepest node explored, along with the branching factor b at each level.

Most Suitable Algorithm: Depth-Limited Search (DLS)

Reasoning: Depth-Limited Search (DLS) is particularly suitable for the Tower of Hanoi problem because the optimal depth $d = 2^n - 1$ is known in advance. DLS avoids the inefficiencies of unbounded DFS while reducing memory usage compared to BFS.

Time Complexity (Based on Nodes Generated): DLS has a time complexity of $O(b^d) = O(3^{2^n-1})$, similar to BFS.

Memory Cost: The memory cost of DLS is $O(bd) = O(3 \times (2^n - 1))$ similarly as DFS but the depth is d as the optimal depth $d = 2^n - 1$ is known in advance.

- **Time Complexity Gap (DLS vs DFS):** For $n = 10$, the optimal depth d is $2^{10} - 1 = 1023$. DLS has a time complexity of $O(3^{1023})$, similar to BFS. In contrast, DFS could explore much deeper paths, leading to a higher time complexity.

- **Memory Cost Gap (DLS vs BFS):** While DLS requires $O(3 \times 1023)$ memory, BFS requires $O(3^{1023})$ memory, making DLS far more memory-efficient. (Same reason for UCS)

As n increases, these differences will become more significant, reinforcing DLS as the most suitable algorithm.

3. **Additional Question:** if there are k pegs instead of 3, what is the branching factor of the search tree?

Solution:

$b = \binom{k}{2}$. Each valid action involves exactly 2 pegs, and for any pair among the k pegs, there is at most one valid action involving said pair (moving a top disk to an empty peg, or placing the smaller top disk on top of the larger top disk). Thus, there are at most $\binom{k}{2}$ valid actions for any state.

C Uniform-Cost Search Analysis

Consider the undirected graph shown in Figure 2. Let S be the initial state and G be the goal state. The cost of each action is as indicated.

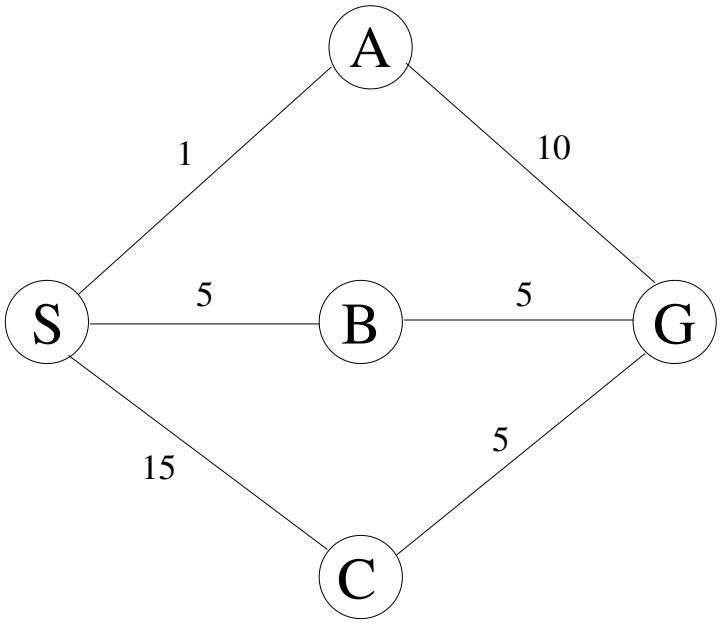

Figure 2: Graph of routes between S and G.

1. Give a trace of uniform-cost search using `SEARCH` (without visited memory) as im-

plemented in Figure 3. If the costs are the same, the tiebreaker is the alphabetical ordering of the letters, with A < Z.

Solution:

If we apply the `SEARCH` algorithm, the following are the nodes in the frontier at the beginning of the loop and at the end of each iteration of the loop: (A node n is followed by its path cost in parentheses)

frontier
S(0)
A(1) B(5) C(15)
S(2) B(5) G(11) C(15)
A(3) B(5) B(7) G(11) C(15) C(17)
S(4) B(5) B(7) G(11) G(13) C(15) C(17)
A(5) B(5) B(7) B(9) G(11) G(13) C(15) C(17) C(19)
B(5) S(6) B(7) B(9) G(11) G(13) C(15) G(15) C(17) C(19)
S(6) B(7) B(9) G(10) S(10) G(11) G(13) C(15) G(15) C(17) C(19)
⋮
G(10) S(10) S(10) B(11) G(11) G(12) S(12) B(13) G(13) G(14) S(14) C(15) G(15) C(17) G(17) C(19) G(19) C(21) C(23)

2. Give a trace of uniform-cost search using `SEARCH WITH VISITED MEMORY` as implemented in Figure 4.

Solution:

Search With Visited Memory Trace: Using the `SEARCH WITH VISITED MEMORY` algorithm, we obtain the following, which shows the nodes in the frontier and nodes expanded at the beginning of the loop and at the end of each iteration of the loop (a node n is followed by its path cost in parentheses):

Frontier	**Visited**
S(0)	
A(1) B(5) C(15)	S
S(2) B(5) G(11) C(15)	S, A
B(5) G(11) C(15)	S, A
G(10) S(10) G(11) C(15)	S, A ,B

3. When A generates G, the goal state, with a path cost of 11 in (b), why doesn't the algorithm halt and return the search result since the goal has been found? With your observation, discuss how uniform-cost search ensures that the shortest path solution is selected.

Solution:

After G is generated, it will be sorted together with the other nodes in the frontier. Since node B has lower path cost than G, it is then selected for expansion. Node B then expands to G with a path cost of 10, which gives the solution. By always selecting the node with the least path cost for expansion (i.e., nodes

are expanded in increasing order of path cost), uniform-cost search ensures that the first goal node selected for expansion is an optimal (i.e., shortest path) solution.

```
create frontier : priority queue (path cost)

insert Node(initial_state) to frontier
while frontier is not empty:
    node = frontier.pop()
    if node.state is goal: return solution



    for action in actions(node.state):
        next_state = transition(node.state, action)
        frontier.add(Node(next_state))
return failure
```

Figure 3: Uniform cost search (`SEARCH` implementation).

```
create frontier : priority queue (path cost)
create visited : set of states
insert Node(initial_state) to frontier
while frontier is not empty:
    node = frontier.pop()
    if node.state is goal: return solution
    if node.state in visited: continue
    visited.add(node.state)
    for action in actions(node.state):
        next_state = transition(node.state, action)
        frontier.add(Node(next_state))
return failure
```

Figure 4: Uniform cost search (`SEARCH WITH VISITED MEMORY` implementation).

D Comparison of Search Strategies

1. Describe a problem in which iterative deepening search performs much worse than depth-first search.

Solution:

Consider a problem with a branching factor b and all states at depth d are solutions while all states at shallower depths are not solutions.

Number of states expanded by DFS (`SEARCH WITH VISITED MEMORY` implementation) = $O(d)$.

Number of states expanded by IDS = $O(b^{d-1})$.

Note: While the number of states expanded by DFS is $O(d)$, the number of states generated is still $O(bd)$ due to the addition of unexpanded states in the frontier.

2. Describe a problem where `SEARCH` performs much better than `SEARCH WITH VISITED MEMORY`.

Solution:

Consider a search problem structured like a tree with branching factor b and maximum depth m, where each node has a unique path from the root, and no cycles exist. Assume the solution is located at depth d.

Imagine a search tree where each node branches into $b = 2$ child nodes. The tree has a maximum depth $m = 3$, and the solution is at depth $d = 1$. In this case, DFS explores paths until it reaches node 3. The DFS will expand the left children first.

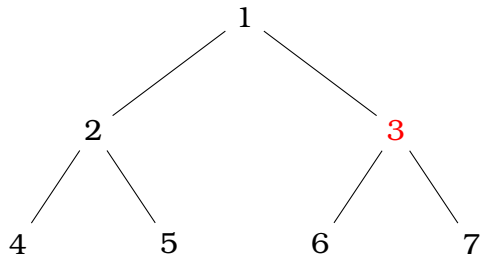

Figure 5: A binary tree with branching factor 2 and depth 2. The node 3 is highlighted in red, indicating the solution.

Memory usage in the frontier (`SEARCH` and `SEARCH WITH VISITED MEMORY`): $O(bm)$

Both `SEARCH` and `SEARCH WITH VISITED MEMORY` utilize $O(bm)$ memory in the frontier. This is because both algorithms need to store the current path from the root to the deepest node being explored. In the example above, the DFS would store up to 3 nodes in memory when reaching nodes 4 or 5, as it progresses down the tree.

Additional Memory Usage in `SEARCH WITH VISITED MEMORY`: $O(b^m)$

In addition to the $O(bm)$ memory for the frontier, `SEARCH WITH VISITED MEMORY` also requires additional memory to store all visited nodes to avoid revisiting states. This extra memory usage is $O(b^m)$, which can become significant as the tree depth increases. In the example, when the algorithm reaches the solution at node 3, it will store nodes 1, 2, 3, 4, and 5 in memory.

Memory Efficiency: While both `SEARCH` and `SEARCH WITH VISITED MEMORY` use $O(bm)$ memory in the frontier, `SEARCH WITH VISITED MEMORY` incurs an additional memory overhead of $O(b^m)$ to store visited nodes. As the maximum depth m increases, this additional memory usage can become substantial. For instance, if the branching factor $b = 2$ and the maximum depth $m = 10$, `SEARCH WITH VISITED MEMORY` would need to keep track of up to $b^m = 1024$ nodes. On the other hand, `SEARCH` continues to use only $O(bm)$ memory, which is linear in relation to the depth.

Appendix

Appendix A: Informal Proof for the total number of states, including invalid states, for Tower of Hanoi

The following is by no means a rigorous proof, but rather a sketch of a proof. However, it should still give you a good grasp of how the formula is obtained.

Let us properly define the problem: We want to find the number of ways to arrange n unique elements into 3 tuples: L, M, R. To do so, we shall adopt the following method of construction: "Shuffle" the n elements into a sequence $a_1, a_2, \ldots, a_n$, and split them into 3 partitions such that

$$\begin{aligned} L &= (a_1, a_2, \ldots, a_{n_1}), \\ M &= (a_{n_1+1}, a_{n_1+2}, \ldots, a_{n_1+n_2}), \\ R &= (a_{n_1+n_2+1}, a_{n_1+n_2+2}, \ldots, a_{n_1+n_2+n_3}), \end{aligned}$$

where $n_1 + n_2 + n_3 = n$. (Note that the order of the items matter: $(1, 2, 3) \neq (1, 3, 2)$). The number of unique configurations this method will yield can be calculated as the following:

$$\begin{aligned} &(\text{\# of ways to shuffle } n \text{ elements}) \times (\text{\# of ways to split } n \text{ elements into 3 partitions}) \\ &= n! \times \binom{n+2}{2} \\ &= \frac{(n+2)!}{2} \end{aligned}$$

Furthermore, it can be shown that this method of construction yields a *bijection* with the actual configurations of the problem. Hence, the number of ways to arrange n unique elements into 3 tuples: $\frac{(n+2)!}{2}$.

Appendix B: Informal Proof for the total number of valid states for Tower of Hanoi

The following is by no means a rigorous proof, but rather a sketch of a proof. However, it should still give you a good grasp of how the formula is obtained.

In order to show how the number of valid configurations for Tower of Hanoi is 3^n, we shall use a particular method of disk-placing: We shall sequentially place the disks onto any of the 3 pegs *in order from the largest disk to the smallest*. This method of disk-placing will yield 3^n unique configurations: For each disk, there are 3 pegs we can choose to put the disk on, therefore the number of unique configurations using this method: $3 \times 3 \times 3 \times \ldots = 3^n$.

Following this method of construction, we notice 2 things:

- This method of placing will *always* satisfy the constraints of the problem, regardless of which peg you decide to place each disk on
- It can be shown that there will be a *bijection* between this method of disk placing, and the valid configurations of the problem

Therefore, we can assert that the number of possible configurations for Tower of Hanoi is also 3^n.