CO541 – Assignment 3

E/19/166

Jayathunga W.W.K.

1. a)
   g(n) is the cost to reach node n,
   h(n) is the heuristic estimate from node
   w is a weight parameter.

   - When w=0:
     $$f(n)=(2-0)g(n)+0\cdot h(n)=2g(n)$$
     In this case, the function f(n) depends solely on the cost to reach node n scaled by a factor of 2. This is equivalent to **Uniform Cost Search (UCS) or Dijkstra's Algorithm**, which expands the node with the lowest path cost g(n). The factor of 2 is just a scaling factor and does not change the nature of the search.

   - When w=1:
     $$f(n)=(2-1)g(n)+1\cdot h(n)=g(n)+h(n)$$
     Here, the function f(n) is the sum of the path cost g(n) and the heuristic estimate h(n). This corresponds to the A* Search Algorithm, which balances the cost to reach the node and the estimated cost to the goal.

   - When w=2:
     $$f(n)=(2-2)g(n)+2\cdot h(n)=0\cdot g(n)+2h(n)=2h(n)$$
     In this scenario, the function f(n) depends solely on the heuristic estimate h(n), scaled by a factor of 2. This is equivalent to Greedy Best-First Search, which expands the node that appears to be closest to the goal based solely on the heuristic h(n).

   b)

   An algorithm is guaranteed to be optimal if it always finds the least-cost path to the goal. A key condition for optimality in search algorithms is the admissibility of the heuristic h(n), which means that h(n) never overestimates the true cost to reach the goal from node

   The given objective function is:

   $f(n)=(2-w)g(n)+wh(n)$

   - Admissibility of Heuristic:

     Since h(n) is admissible, $h(n) \leq h^*(n)$, where (n) is the true cost from node n to the goal.

- Effect of Different w Values:

  w=1: When w=1, the objective function becomes:

  $$f(n)=g(n)+h(n)$$

  This is the standard form for the A* search algorithm, which is guaranteed to be optimal when the heuristic h(n) is admissible. Therefore, the algorithm is optimal for w=1.

  $w \neq 1$: For other values of w, the function f(n) changes in a way that can impact the balance between g(n) and h(n). Specifically:

  When w<1: The term (2−w)g(n) becomes more dominant, making the algorithm focus more on the path cost already incurred. While this might still find an optimal path, it does not guarantee optimality as strictly as A* because the weight on the heuristic is reduced.

  When w>1: The term wh(n) becomes more dominant. This could lead to paths being chosen more for their heuristic value than their actual path cost, which can cause the algorithm to miss the optimal path.

The heuristic path algorithm is guaranteed to be optimal specifically when w=1, as this aligns it with the A* search algorithm, which is known to be optimal with an admissible heuristic. For $w \neq 1$, the guarantees of optimality are not as strong because the balance between the actual cost and the heuristic estimate is altered in ways that can affect the algorithm's performance.

2. a) Initial State: B B B B W W W E (Cost: 0)

First Level Moves:
Move E left: B B B W W E W (Cost: 1)
Jump E over W: B B B BW E W W (Cost: 2)
Jump E over two Ws: B B B E W W W (Cost: 5)

Second Level Moves:

<u>From B B B W W E W (Cost: 1):</u>
Move E left: B B B W E W W (Cost: 2)
Move W right: B B B W W W E (Cost: 2)
Jump E over W: B B B E W W W (Cost: 3)

<u>From B B B W E W W (Cost: 2):</u>
Move E left: B B B E W W W (Cost: 3)
Move W right: B B B W W E W (Cost: 3)
Jump E over W: B B B E W W W (Cost: 4)

<u>From B B B E W W W (Cost: 5):</u>
Move E left: B B E B W W W (Cost: 6)
Move W right: B B B W E W W (Cost: 6)

Jump E over W: B B E B W W W (Cost: 7)

Third Level Moves:

From B B B E W W W (Cost: 3):
Move E left: B B E B W W W (Cost: 4)
Move W right: B B B W W E W (Cost: 4)
Jump E over W: B B E B W W W (Cost: 5)

From B B B W W E W (Cost: 2):
Move E left: B B B W E W W (Cost: 3)
Move W right: B B B W W W E (Cost: 3)
Jump E over W: B B B E W W W (Cost: 4)

From B B B E B W W W (Cost: 6):
Move E left: B E B B W W W (Cost: 7)
Move W right: B B E B W W W (Cost: 7)
Jump E over W: B E B B W W W (Cost: 8)

For simplicity, the tree is pruned at nodes where the cost is increasing without reaching the goal. We can see that uniform cost search explores paths based on cumulative cost, expanding the lowest cost nodes first. The search continues until all paths lead to states where all Ws are to the left of all Bs, with the empty slot (E) positioned anywhere.

Final State:
Once all paths are expanded, we can find:
The goal state can be reached at the minimum cost of moving E to the left in steps that align W and B tiles in the required order. The lowest cost paths will naturally be expanded first, ensuring optimal moves.
This simplified tree gives an idea of how the uniform cost search explores different paths. The full tree for the given problem can get quite large, but the principle remains the same: explore all possible states generated by valid moves, expanding the lowest cost nodes first, until reaching the goal state.

b) No of Misplaced Tiles: Count the number of pairs (B,W) where B is to the left of W. This gives a measure of disorder.

c) **The start state:** B B B W W W E

Goal state: All Black tiles to the right of White tiles, E can be anywhere.

Initial State: B B B W W W E

g(n) = 0

h(n) = Number of misplaced pairs (4 pairs here)

f(n) = g(n) + h(n) = 0 + 4 = 4

First Level Moves:

Move E left:

B B B W W E W

g(n) = 1

h(n) = Number of misplaced pairs (still 4 pairs)

f(n) = 1 + 4 = 5

Jump E over W:

B B B W E W W

g(n) = 2

h(n) = Number of misplaced pairs (still 4 pairs)

f(n) = 2 + 4 = 6

Jump E over two Ws:

B B B E W W W

g(n) = 5

h(n) = Number of misplaced pairs (still 4 pairs)

f(n) = 5 + 4 = 9

Priority Queue (Sorted by f(n)):

B B B W W E W (f(n) = 5)

B B B W E W W (f(n) = 6)

B B B E W W W (f(n) = 9)

Second Level Moves:

From B B B B W W E W (f(n) = 5):

Move E left:

B B B W E W W

g(n) = 2

h(n) = Number of misplaced pairs (still 4 pairs)

f(n) = 2 + 4 = 6

Move W right:

B B B W W W E

g(n) = 2

h(n) = Number of misplaced pairs (still 4 pairs)

f(n) = 2 + 4 = 6

Jump E over W:

B B B E W W W

g(n) = 3

h(n) = Number of misplaced pairs (still 4 pairs)

f(n) = 3 + 4 = 7

Updated Priority Queue:

B B B W E W W (f(n) = 6)

B B B W E W W (f(n) = 6)

B B B E W W W (f(n) = 7)

B B B E W W W (f(n) = 9)

Third Level Moves:

From B B B B W E W W (f(n) = 6):

Move E left:

B B B E W W W

g(n) = 3

h(n) = Number of misplaced pairs (still 4 pairs)

f(n) = 3 + 4 = 7

Move W right:

B B B W W W E

g(n) = 3

h(n) = Number of misplaced pairs (still 4 pairs)

f(n) = 3 + 4 = 7

Jump E over W:

B B B E W W W

g(n) = 4

h(n) = Number of misplaced pairs (still 4 pairs)

f(n) = 4 + 4 = 8

Updated Priority Queue:

B B B E W W W (f(n) = 7)

B B B E W W W (f(n) = 7)

B B B E W W W (f(n) = 8)

B B B E W W W (f(n) = 9)

Fourth Level Moves:

From B B B B E W W W (f(n) = 7):

Move E left:

B B E B W W W

g(n) = 4

h(n) = Number of misplaced pairs (still 4 pairs)

f(n) = 4 + 4 = 8

Move W right:

B B B W W E W

g(n) = 4

h(n) = Number of misplaced pairs (still 4 pairs)

f(n) = 4 + 4 = 8

Jump E over W:

B B E B W W W

g(n) = 5

h(n) = Number of misplaced pairs (still 4 pairs)

f(n) = 5 + 4 = 9

Updated Priority Queue:

B B E B W W W (f(n) = 8)

B B B W W E W (f(n) = 8)

B B B E W W W (f(n) = 8)

B B B E W W W (f(n) = 9)

B B B E W W W (f(n) = 9)

Following the same process, the search continues by expanding nodes with the lowest f(n) value, applying moves to generate new states, and updating the priority queue accordingly. This expansion will eventually lead to a state where all W tiles are to the left of all B tiles, achieving the goal.

d) The heuristic h(n) we devised counts the number of misplaced pairs where a Black tile (B) is to the left of a White tile (W). The assumption is that each misplaced pair requires at least one move to correct:

h(n)=Number of misplaced (B, W) pairs

Analysis of Admissibility

Underestimation or Exact Estimation: By counting the number of misplaced pairs, we are estimating the minimum number of moves required to place all Black tiles to the right of all White tiles. Since each misplaced pair needs at least one move to be corrected, this heuristic does not overestimate the cost.

Consistency with Move Costs: Considering the move costs:

Moving one tile to the left or right costs 1 unit.

Jumping over one tile costs 2 units.

Jumping over two tiles costs 5 units.

The heuristic h(n) only considers the minimal move scenario (1 unit per pair). Even in the worst-case scenario, more complex moves (like jumps) would only confirm that the heuristic remains a lower bound, not an overestimate.

The heuristic h(n), which counts the number of misplaced (B, W) pairs, is admissible because:

- It never overestimates the actual cost to reach the goal.
- It provides a lower bound on the number of moves required to sort the tiles correctly.

Hence, h(n) is admissible and suitable for use in A* search to ensure optimality in finding the least-cost solution.

3. a) MST as a Lower Bound: An MST is a tree that connects all the vertices (cities) with the minimum possible total edge weight. Since it connects all cities with no cycles and the least cost, it provides a lower bound on the cost required to visit all cities. The MST does not necessarily include all edges needed to form a Hamiltonian cycle (required for TSP), so it underestimates the total cost. Any valid TSP tour is essentially a cycle, which is more costly than a tree connecting the same nodes.

   Inclusion of Start and Current City: The MST heuristic includes the start city and the current city, ensuring that the heuristic reflects the cost of connecting all relevant nodes from the current

state. The heuristic does not need to account for the cost of returning to the start city, as this will be included in the actual TSP tour but not necessarily in the MST, further ensuring it does not overestimate.

Example Analysis:

Suppose the current state of the TSP problem is at city C with unvisited cities {A,B,D,E}.The MST of these cities including C provides the minimum connection cost among these cities.
The actual TSP tour must include a round trip starting from C, visiting all other cities, and returning to C, which will be at least as costly as the MST since the TSP tour forms a cycle.

Comparison with Optimal Solution:

The cost of the optimal TSP solution, $h*(n)$, must be at least the cost of the MST plus the costs of additional edges needed to form a cycle. Therefore, the MST heuristic will always provide a cost estimate that is less than or equal to the actual minimum cost to complete the TSP tour from the current state.

Formal Argument
$h(n)$ be the MST heuristic cost. $h*(n)$ be the actual cost of the optimal TSP tour from the current state. Since the MST provides the minimum connection cost and the TSP tour is a cycle connecting all nodes, we have: $h(n) \leq h*(n)$
The MST heuristic is admissible for the TSP because it never overestimates the actual cost to complete the tour from any given state. The MST provides a lower bound on the cost of connecting all unvisited cities, ensuring that the heuristic is both optimistic and accurate for guiding the search towards the optimal solution.

b) The MST heuristic h(n) for the Traveling Salesperson Problem (TSP) estimates the cost by computing the MST of the graph connecting all unvisited cities and the current city.

Structure of the Problem:

n be the current state with the current city C and the set of unvisited cities U.
Let n'be the successor state after moving to city C '(one of the unvisited cities).
The MST heuristic h(n) is the cost of the MST for the set $\{C\} \cup U$.
The MST heuristic h(n') is the cost of the MST for the set $\{C'\} \cup (U \setminus \{C'\})$.

Consistency Check:

When moving from n to n ', the cost c(n,n') is the direct edge cost between C and C', denoted as d(C,C').
We need to show that:
$h(n) \leq d(C,C') + h(n')$

Detailed Argument

Initial MST Heuristic: h(n) is the cost of the MST of {C}∪U. New MST Heuristic: h(n′) is the cost of the MST of {C′}∪(U\{C′}).

Adding and Removing Nodes:
When transitioning from n to n′, we effectively remove C and add C′ to the set of nodes considered for the MST.

Cost Changes:

The MST of {C}∪U includes the minimum connections among C and all cities in U. C and adding C′ changes the MST, but the direct cost d(C,C′) must be accounted for. Consider the Following Cases

Direct Edge Inclusion: If d(C,C′) is included in the MST of {C,C′}∪(U\{C′}), then the total MST cost does not increase significantly. The cost of the MST with C′ might be lower or equal to adding the edge d(C,C′) to the remaining MST.

Edge Substitution: If the MST configuration changes by substituting edges, the new MST cost h(n′) plus the transition cost d(C,C′) will not exceed the original MST cost h(n).

Intuition and Formal Proof

The intuition is that replacing one node in the MST with another node should not increase the overall cost more than the direct transition cost because:

The MST is always the minimum possible connection, thus moving from one valid configuration to another by direct edge addition does not violate consistency.
Formally, since the MST heuristic always gives the minimum cost of connecting all nodes (and the MST of the subset of nodes plus the transition edge cost cannot exceed the original MST cost), it ensures: $h(n) \leq d(C,C') + h(n')$

The MST heuristic for the TSP is consistent because it satisfies the condition $h(n) \leq c(n,n') + h(n')$ for all transitions between states. Each step's cost to move to a new city and the MST estimate from that new city to the remaining unvisited cities ensures that the heuristic does not decrease by more than the step cost, thus maintaining the consistency property.

c) To show how the MST heuristic can be derived from a relaxed version of the Traveling Salesperson Problem (TSP), we need to define a version of TSP that is simpler to solve and from which the heuristic can be extracted. The relaxed problem should be easier than the original TSP but should still provide useful information about the solution to the TSP.
Relaxed TSP: The Minimum Spanning Tree (MST)

A Minimum Spanning Tree (MST) connects all nodes in a graph with the minimum possible total edge weight and without forming any cycles. The relaxed version of TSP removes the requirement to form a cycle and instead focuses on connecting all cities with the minimum cost.

Steps to Derive the MST Heuristic

Original Constraints:

Visit all cities exactly once.
Return to the starting city.
Form a cycle.

Relaxed Constraints:

Connect all cities.
Minimize the total connection cost.
No requirement to form a cycle (tree structure is sufficient).

Formulation of the Relaxed Problem

Vertices: The cities to be visited.
Edges: The possible routes between cities, with their respective costs.
Objective: Find a subset of edges that connects all vertices (cities) with the minimum total cost and forms no cycles.

Solution to the Relaxed Problem

The relaxed problem is solved by finding the Minimum Spanning Tree (MST) of the graph formed by the cities and their interconnecting edges. The MST provides a lower bound for the TSP because:

- An MST connects all cities with the minimum total edge weight.
- Any TSP tour, which is a cycle, must have a total cost at least as large as the MST (since the MST is the minimal way to connect all cities).

Using the MST as a Heuristic for TSP

In the context of the A* search algorithm for TSP, the MST heuristic h(n) estimates the remaining cost to complete the TSP tour from a given state n.

Steps in the A* Algorithm with MST Heuristic

Current State: Given a current city C and a set of unvisited cities U.Compute MST: Calculate the MST for the graph formed by {C}∪U.

Heuristic Value: Use the cost of this MST as the heuristic h(n).

The MST heuristic for TSP is derived from a relaxed version of the problem where the requirement to form a cycle is removed. Instead, we only need to connect all cities with the minimum cost, resulting in an MST. This heuristic is both admissible and consistent, making it suitable for use in algorithms like A* to find the optimal TSP tour.

d) Let's introduce another heuristic: the Minimum One-Two-Tree (MOTT) heuristic.

Minimum One-Two-Tree (MOTT) Heuristic
The MOTT heuristic is defined as the cost of the Minimum Spanning Tree (MST) plus the costs of the two cheapest edges connecting the starting node (or current node) to the MST.

Definition and Steps for MOTT Heuristic

Compute the MST:
Compute the MST for the graph formed by the unvisited cities and the current city.
Find the Two Cheapest Edges:
Identify the two cheapest edges connecting the current city to the nodes in the MST.
Calculate the MOTT Heuristic:
The MOTT heuristic $h_{MOTT}(n)$ is the sum of the MST cost and the two cheapest edges.

Comparison of MST and MOTT Heuristics
To determine which heuristic dominates the other, let's define what it means for one heuristic to dominate another.

Definition of Dominance
A heuristic h1(n) dominates another heuristic h2(n) if for every node n: h1(n)≥h2(n)
and there exists at least one node n for which: h1(n)>h2(n)

Analysis of Dominance

MST Heuristic:
h_{MST}(n): The cost of the MST connecting all unvisited cities and the current city.

MOTT Heuristic:
h_{MOTT}(n): The cost of the MST plus the costs of the two cheapest edges connecting the current city to the MST.

Dominance Analysis

Since the MOTT heuristic $h_{MOTT}(n)$ includes the cost of the MST plus additional edges, we can assert that: $h_{MOTT}(n) \geq h_{MST}(n)$
because the MST is a subset of the MOTT. Furthermore, for at least one node $h_{MOTT}(n) > h_{MST}(n)$ since the addition of the two cheapest edges generally increases the heuristic value.

Formal Proof of Dominance
For every node n:

MST Heuristic:
T be the MST for the set of cities including the current city.
The cost of the MST is $h_{MST}(n)$.

MOTT Heuristic:
Let the two cheapest edges connecting the current city to the MST be e1 and e2 with costs c1 and c2, respectively.
The cost of the MOTT heuristic is $h_{MOTT}(n) = h_{MST}(n) + c1 + c2$.
Clearly:
$h_{MOTT}(n) = h_{MST}(n) + c1 + c2$
Since $c1 \geq 0$ and $c2 \geq 0$:
$h_{MOTT}(n) \geq h_{MST}(n)$

Moreover, if $c1 > 0$ and $c2 > 0$:

$h_{MOTT}(n) > h_{MST}(n)$

The MOTT heuristic dominates the MST heuristic because it always provides a cost estimate that is at least as high as the MST heuristic and, in most cases, will be strictly higher due to the inclusion of the two cheapest connecting edges. This makes the MOTT heuristic more informative and potentially more efficient for guiding the search in solving the TSP.

e) To show a heuristic for the Traveling Salesperson Problem (TSP) that is not admissible, let's define a heuristic that can potentially overestimate the cost of reaching the goal. An inadmissible heuristic provides an estimate that is sometimes higher than the actual minimum cost to complete the tour.

Non-Admissible Heuristic: Maximum Edge Heuristic (MEH)
The Maximum Edge Heuristic (MEH) estimates the cost of completing the TSP tour by taking the cost of the maximum edge among the edges connecting the current city to any unvisited city and multiplying it by the number of unvisited cities.

Definition of the Maximum Edge Heuristic (MEH)

Identify the Maximum Edge:
For the current city C and the set of unvisited cities U, find the maximum edge cost dmax from C to any city in U.

Calculate the Heuristic Value:
Let |U| be the number of unvisited cities. Define the heuristic value $h_{MEH}(n)$ as: $h_{MEH}(n)=dmax\times|U|$

Example

Consider a TSP problem with the following cities and distances:
A-B: 2
A-C: 3
A-D: 1
B-C: 4
B-D: 5
C-D: 6
Suppose the current city is A, and the unvisited cities are B, C, and D.

Maximum Edge Identification:

The maximum edge from A to any unvisited city is max(2,3,1)=3.

Calculate Heuristic Value:

Number of unvisited cities |U|=3.The heuristic value is: $h_{MEH}(n)=3\times3=9$

Analysis of Admissibility
To check $h_{MEH}(n)$ is admissible, we need to ensure it never overestimates the actual minimum cost $h*(n)$ of completing the tour. If there exists any state where hMEH(n)>h*(n), the heuristic is not admissible.

Consider the actual TSP solution:

The minimum tour visiting all cities might be A-D-B-C-A with a total cost of: 1+5+4+3=13

For this example:

The maximum edge heuristic $h_{MEH}(n)$ estimates the cost as 9.

This example does not immediately show inadmissibility since 9 is less than 13. However, let's consider a situation where it clearly overestimates.
Clear Inadmissibility Example

Consider another setup:

Cities: A, B, C
Distances: A-B: 1, A-C: 1, B-C: 1 (equal distances for simplicity)
Suppose the current city is A, and the unvisited cities are B and C.

Maximum Edge Identification:

The maximum edge from A to any unvisited city is max(1,1)=1.

Calculate Heuristic Value:

Number of unvisited cities |U|=2.
The heuristic value is: hMEH(n)=1×2=2
The actual minimum TSP tour might be A-B-C-A with a total cost of: 1+1+1=3

In this case: hMEH(n)=2 underestimates the actual cost 3. However, to show non-admissibility, let's consider a modified heuristic.

Modified Maximum Edge Heuristic (MMEH)
Let's modify the heuristic to make it clearly inadmissible:

Modified Heuristic Calculation:
Define the heuristic value $h_{MMEH}(n)$ as: $h_{MMEH}(n)=d_{max}\times(|U|+1)$

Using the same example with cities A, B, and C: Maximum edge from A to unvisited cities B and C is 1.
Number of unvisited cities |U|=2.
The heuristic value is: $h_{MMEH}(n)=1\times(2+1)=3$
For the actual TSP tour A-B-C-A with a total cost of 3:

The heuristic $h_{MMEH}(n)=3$, which matches the actual cost but let's consider more cities.

Consider cities A, B, C, and D with:

A-B: 1, A-C: 1, A-D: 1, B-C: 2, B-D: 2, C-D: 2
Suppose the current city is A, and unvisited cities are B, C, and D.

Maximum Edge Identification:
Maximum edge from A to unvisited cities B, C, and D is 1.
Number of unvisited cities |U|=3.
Using hMMEH(n)=dmax×(|U|+1):
hMMEH(n)=1×(3+1)=4

For TSP tour A-B-C-D-A with total cost:1+2+2+1=6

Here, hMMEH(n) underestimates. Let's overestimate:

Clearly Overestimating Example

Use distances:

A-B: 5, A-C: 5, A-D: 5, B-C: 1, B-D: 1, C-D: 1

Current city A, unvisited: B, C, D.
Maximum Edge: 5 (to B, C, D).

Number unvisited |U|=3.
Using modified heuristic: $h_{MMEH}(n)=5\times(3+1)=20$

Actual minimum TSP tour might be: A−B−C−D−A or A−B−D−C−A

With cost: 5+1+1+5=12

Clearly: $h_{MMEH}(n)=20$

Overestimates: $h_{MMEH}(n)>h^*(n)=12$

The Maximum Edge Heuristic (MEH) with modification to multiply by |U|+1 (MMEH) is not admissible as it can overestimate. Example shows $h_{MMEH}(n)=20$ exceeding actual cost $h^*(n)=12$. Thus, heuristic not admissible.

4. Argument Breakdown

Initial Claim: "Given two admissible heuristics h1 and h2 where $h1(n)\geq h2(n)$ for all nodes n, it is obvious that A* using h1 will become more efficient than A* using h2."

Assumption and Proposal: "Now suppose I am given an admissible heuristic h2. If one can find a constant c such that heuristic $h1(n)=h2(n)+c$ is still admissible, then searching with h1 is better than searching with h2."

Detailed Critique

Initial Claim: Comparison of Admissible Heuristics

- Statement: "Given two admissible heuristics h1 and h2 where $h1(n)\geq h2(n)$ for all nodes n, it is obvious that A* using h1 will become more efficient than A* using h2."

- Critique:

  Correctness of Admissibility: Both heuristics h1 and h2 are admissible, meaning $h1(n)\leq h^*(n)$ and $h2(n)\leq h^*(n)$ for all nodes n, where $h^*(n)$ is the true cost to reach the goal from n.

  Efficiency in A*: It is generally true that a more informed (i.e., higher but still admissible) heuristic leads to fewer nodes being expanded by A*. This is because a higher heuristic value closer to the true cost provides better guidance towards the goal, reducing the search space.

  Obviousness: While it is not necessarily "obvious," it is a well-established result in AI search literature that a more informed heuristic (given admissibility) tends to make A* more efficient by reducing the number of expanded nodes. However, efficiency is not solely dependent on heuristic value; it also depends on the specific problem and the implementation details.

Assumption and Proposal: Adding a Constant to Create a Better Heuristic

- Statement: "Now suppose I am given an admissible heuristic h2. If one can find a constant c such that heuristic h1(n)=h2(n)+c is still admissible, then searching with h1 is better than searching with h2."

- Critique:
  Admissibility with Constant Addition:
  Definition: For h1(n)=h2(n)+c to be admissible, $h1(n) \leq h*(n)$ for all nodes n.

  Condition: Given h2 is admissible, $h2(n) \leq h*(n)$. Thus, for h1(n) to be admissible, $h2(n)+c \leq h*(n)$.

  Implication: This implies c must be non-positive ($c \leq 0$). Otherwise, h1(n) could exceed the true cost $h*(n)$, violating admissibility.
  Effect of c: If c is negative or zero, h1(n)=h2(n)+c does not necessarily provide better guidance than h2(n). In fact, it could degrade performance by making the heuristic less informative.

Efficiency Comparison:

General Heuristics: The proposal implicitly assumes that adding a constant c will improve efficiency, but this is not guaranteed. A constant addition does not change the relative ranking of nodes and, if negative, makes the heuristic less informative.
Better Guidance: For h1 to provide better guidance and be more efficient, it must not only be admissible but also be more informed in a meaningful way (e.g., closer to the true cost $h*(n)$).
Practical Heuristic: Practically, improving h2 to h1 should involve domain-specific enhancements that make h1 closer to $h*(n)$, not merely adding a constant c.

The argument presented has flaws in both steps:

Initial Claim: While $h1(n) \geq h2(n)$ generally leads to fewer node expansions in A*, the argument oversimplifies this by calling it "obvious" without considering the nuanced factors affecting search efficiency.
Assumption and Proposal: The suggestion that h1(n)=h2(n)+c improves efficiency is flawed. For h1 to remain admissible, c must be non-positive, which could degrade heuristic informativeness. The assumption ignores the practical aspects of heuristic design, where improvements should be based on more accurate estimations of the true cost rather than arbitrary constants.

Thus, the argument needs a more rigorous analysis and a deeper understanding of how heuristics influence A* search efficiency.

5. To illustrate a state space where A* using GRAPH-SEARCH returns a suboptimal solution due to an admissible but inconsistent heuristic, we need to construct an example where the inconsistency of the heuristic leads the algorithm astray. Here's how we can construct such a state space:

State Space Construction
Consider the following directed graph with nodes S,A,B, C, and G (goal state):

S (start node)
A, B, and C (intermediate nodes)
G (goal node)

Graph Structure

- Edges and Costs:
  S to A: cost 2
  S to B: cost 2
  A to G: cost 2
  B to C: cost 1
  C to G: cost 2

Heuristic Values h(n)
Let's define an admissible but inconsistent heuristic:

h(S)=3 (estimated cost from S to G)
h(A)=1 (estimated cost from A to G)
h(B)=4 (estimated cost from B to G)
h(C)=2 (estimated cost from C to G)
h(G)=0 (cost from G to G)

Explanation of Admissibility and Inconsistency

- Admissibility: For all nodes n, $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach G.

True costs:
$h^*(S)=4$ (via S→A→G)
$h^*(A)=2$ (via A→G)
$h^*(B)=3$ (via B→C→G)
$h^*(C)=2$ (via C→G)

Our heuristic values are all less than or equal to these true costs.

- Inconsistency: For some nodes n and their neighbors m, $h(n)>c(n,m)+h(m)$, violating the consistency condition.

Example: h(B)=4, but h(C)+c(B,C)=2+1=3, and 4>3.

Execution of A* with GRAPH-SEARCH
Step-by-Step Execution:

1. Initialization: Start with nodeS.

g(S)=0
f(S)=g(S)+h(S)=0+3=3

2. Expand S:

Successors: A and B

g(A)=2, f(A)=g(A)+h(A)=2+1=3

g(B)=2, f(B)=g(B)+h(B)=2+4=6

3. Choose node with lowest f-value: Expand A (since f(A)=3 and f(B)=6).

Successor: G

g(G via A)=g(A)+c(A,G)=2+2=4

f(G)=g(G)+h(G)=4+0=4

4. Goal test: G is the goal, so A* will return the pathS→A→G with cost 4.

However, consider the path through B:

5. Expand B:
   Successor: C
   g(C)=g(B)+c(B,C)=2+1=3
   f(C)=g(C)+h(C)=3+2=5
   Expand C:
   uccessor: G
   g(G via B and C)=g(C)+c(C,G)=3+2=5
   f(G)=g(G)+h(G)=5+0=5

   Comparing the paths:
   S→A→G with cost 4.
   S→B→C→G with cost 5.

   A* with GRAPH-SEARCH has expanded nodes in a manner such that it finds the path through A first due to lower f-value, but due to heuristic inconsistency, it does not explore the possibly optimal path through B efficiently.
   However, in this specific setup, inconsistency didn't lead to missing the optimal path because both paths were found, and the lowest cost path was chosen. To guarantee a scenario where the heuristic leads to suboptimality explicitly, consider adjusting such that graph search stops when finding G initially if another optimal path is viable but not yet fully explored.
   This example illustrates how A* with an admissible but inconsistent heuristic can lead to inefficiencies. Adjustments may yield a clear suboptimal result directly if expansion priority errors occur more sharply, e.g., modifying edge costs or heuristic to skew search path more significantly.

6. Best-First Search Using Comparison Method

When there is no good evaluation function but a good comparison method exists, we can still perform best-first search. Here's how we can adapt best-first search to work with a comparison method instead of numerical evaluations:

Best-First Search with Comparison Method

1. Initialization:

- Create an empty priority queue (often implemented as a min-heap or a priority queue).
- Insert the initial node into the priority queue.

2. Search Loop:

- While the priority queue is not empty:
  - Remove the node n from the priority queue (this is the node with the highest priority according to the comparison method).
  - If n is a goal node, return n as the solution.
  - Generate all successors of n.
  - For each successor, insert it into the priority queue using the comparison method to maintain the order.

3. Comparison Method:

- Instead of using a numerical evaluation function f(n) to compare nodes, use the provided comparison method to determine if one node is better than another.
- The priority queue operations (insert and remove) will rely on this comparison method to maintain the correct order.

Example Scenario

Assume we are solving a pathfinding problem in a grid. Instead of evaluating nodes with a numerical value, we have a comparison method that can determine which of two nodes is closer to the goal based on some heuristic (like a visual assessment of proximity to the goal):

1. Initial Node: Start at the initial position.
2. Priority Queue: Use the comparison method to keep the queue sorted.
3. Comparison Method: The method assesses two nodes and determines which one seems closer to the goal (without assigning actual numerical distances).

Adapting for A* Search

To determine if the comparison method can be used similarly for A* search, let's analyze the requirements of A*:

1. Evaluation Function in A*:

- A* uses an evaluation function f(n)=g(n)+h(n) where:
  - g(n) is the cost from the start node to n.
  - h(n) is the heuristic estimate of the cost from n to the goal.

2. Priority Queue in A*:

- A* relies on the numerical evaluation f(n) to prioritize nodes in the priority queue.

- The evaluation function ensures that nodes closer to the goal (both in terms of actual cost and heuristic estimate) are explored first.

Issues with Using Comparison Method for A*

- Inconsistency: The comparison method alone does not provide the numerical values necessary to compute the total estimated cost f(n).
- Optimality and Completeness: A* guarantees optimality if the heuristic is admissible and consistent. Using a comparison method might not maintain this guarantee because it doesn't provide a clear and consistent way to balance the actual cost (g(n)) and the heuristic estimate (h(n)).
- Combining Costs: Without numerical values, it's challenging to correctly combine actual path costs (g(n)) with heuristic estimates (h(n)) in a way that ensures the search is guided efficiently and correctly.

Conclusion

- Best-First Search: Yes, a comparison method can be used to perform best-first search. The priority queue can be maintained using the comparison method to determine the order of node expansion.
- A Search*: No, a comparison method alone cannot be used in the same way for A* search. A* requires numerical evaluations to calculate the total cost f(n) accurately, ensuring optimal and efficient pathfinding. The comparison method does not provide a way to balance the actual cost and heuristic estimate needed for A*'s guarantees.

In summary, while a comparison method is sufficient for best-first search, it is not adequate for A* due to the need for numerical evaluation to maintain the properties and guarantees of A*.

7. SMA* (Simplified Memory-Bounded A*) is considered an optimal search algorithm in the context of memory-bounded searches. To understand what it means for SMA* to be optimal, let's break down the key aspects of this claim:

   SMA* Overview
   SMA* is a variation of the A* search algorithm designed to handle situations where memory is limited. The primary goal of SMA* is to manage the available memory efficiently while still aiming to find an optimal solution.

   Characteristics of SMA*
   1. Memory Management: SMA* limits the amount of memory used by keeping only the most promising nodes in memory and discarding less promising ones when memory is full. When nodes are discarded, the algorithm saves enough information to regenerate them if needed.

2. Admissible Heuristic: SMA* relies on an admissible heuristic, meaning that the heuristic function h(n) never overestimates the cost to reach the goal from any node n.
3. Regeneration of Nodes: When nodes are discarded due to memory constraints, SMA* stores their path cost and heuristic value so they can be regenerated if revisited later.

Optimality of SMA*

Optimality in the context of SMA* means that, given sufficient memory, SMA* will find the least-cost path (optimal solution) from the start node to the goal node, assuming an admissible heuristic. Here are the details of this optimality:

1. Finding the Optimal Solution: SMA* guarantees that it will find the optimal solution if enough memory is available. If memory is not sufficient to hold all nodes required to find the optimal solution at once, SMA* will still operate but may revisit nodes, potentially increasing the search time.
2. Memory-Bounded Environment: In a memory-bounded environment, SMA* is designed to provide the best possible solution given the memory constraints. It systematically discards the least promising nodes while ensuring that information is stored to recover them if needed.
3. Admissibility and Consistency: Like A*, SMA* relies on the admissibility (and often consistency) of the heuristic to ensure that the path cost estimate is optimistic. This characteristic helps SMA* prioritize nodes correctly and converge on the optimal solution.

Practical Implications of SMA*'s Optimality

- Limited Memory Scenarios: In scenarios with limited memory, SMA* is optimal in the sense that it maximizes the use of available memory to find the best possible solution. It balances between memory usage and search efficiency, striving to find the optimal path even if it means regenerating some nodes.
- Trade-Offs: The optimality of SMA* also involves trade-offs. While it finds the optimal solution, the process might be slower due to the overhead of regenerating nodes and managing the memory constraints.
- Admissible Heuristic: The reliance on an admissible heuristic is crucial for SMA*'s optimality. The heuristic ensures that SMA* does not overlook potentially optimal paths by underestimating their costs.

Conclusion

SMA* is considered an optimal search algorithm because it guarantees finding the least-cost path (optimal solution) in a memory-bounded environment, given an admissible heuristic. Its design focuses on managing limited memory efficiently while still aiming for optimality. This makes SMA* particularly valuable in situations where memory constraints are a significant concern, allowing it to provide the best possible solutions within the given limitations.

8. The Simplified Memory-Bounded A* (SMA*) algorithm is specifically designed to handle search problems where memory constraints are a significant issue. Here are the characteristics of a search problem that can be solved by SMA* but not by a standard A* algorithm:

   Characteristics of Such Search Problems

   1. Large State Space:
      The problem involves a very large state space that cannot fit into memory all at once. A* would run out of memory trying to store all the necessary nodes, while SMA* can manage by keeping only the most promising nodes in memory and discarding others.
   2. High Branching Factor:
      The search tree has a high branching factor, resulting in an exponential growth of possible nodes to explore. This quickly overwhelms the memory capacity of A*, whereas SMA* is designed to handle such growth by limiting memory usage.
   3. Memory Constraints:
      The available memory is significantly limited relative to the size of the search space. A* requires enough memory to store all generated nodes, but SMA* can function within these constraints by discarding and regenerating nodes as needed.
   4. Iterative Improvement Needs:
      The problem may benefit from an algorithm that can make iterative improvements. SMA* can revisit previously discarded nodes and improve upon earlier solutions as it regenerates nodes and explores different paths.
   5. Non-critical Time Constraints:
      While SMA* is more memory efficient, it may be slower due to the overhead of regenerating nodes and managing limited memory. Problems where time efficiency is less critical compared to memory efficiency are suitable for SMA*.

Example Problem

Consider a navigation problem in a very large city with millions of potential routes, where each route segment and intersection can be a node in the state space. Here's how SMA* and A* would handle it:

- A*: The algorithm would start expanding nodes, but given the large number of possible routes and intersections, it would quickly run out of memory, especially if it needs to store all nodes and their evaluations.
- SMA*: This algorithm would handle the same problem by:
  - Keeping only the most promising routes in memory.
  - Discarding less promising nodes when memory limits are reached.
  - Storing just enough information to regenerate discarded nodes if they become relevant again.
  - Ensuring that even with limited memory, the algorithm can eventually find the optimal route by efficiently managing the memory.

Specific Characteristics Detailed

1. Memory Efficiency:
   SMA* uses a bounded amount of memory, discarding less promising paths while retaining enough information to regenerate them. This makes it capable of solving problems that A* cannot due to memory overflow.
2. Dynamic Node Management:
   SMA* dynamically manages its open list, prioritizing nodes with better evaluation while discarding and later regenerating others. This allows it to work within strict memory limits.
3. Optimality under Memory Constraints:
   Even with limited memory, SMA* can guarantee finding the optimal solution by efficiently using the available memory. It ensures that the most promising paths are always explored.
4. Adaptability to Memory Limitations:
   SMA* adapts to the given memory constraints, unlike A* which assumes enough memory is available to store all nodes. SMA*'s ability to operate within strict memory limits makes it suitable for very large problems.

Conclusion

Search problems suitable for SMA* but not A* are typically those with very large state spaces, high branching factors, and strict memory constraints. SMA* is designed to manage memory efficiently by keeping the most promising nodes and discarding and regenerating others as needed, thus enabling it to solve problems that A* cannot due to memory limitations.

9. a) hill-climbing search
   b) breadth-first search (BFS)
   c) hill-climbing search
   d) random search or random mutation hill-climbing