CSCI3230 Fundamentals of Artificial Intelligence

Written Assignment 2

Jonathan Lam – 1155093597

Question a

i.

Genetic Algorithm is a stochastic search algorithm which generates random states which will then evolve to search for the optima. Similar to the evolution of beings where *survival of the fittest* is true, the states will undergo genetic operations, such as crossover, mutation, selection, and the best of the individuals (evaluated by the fitness function) will survive, eventually converging into the global optimum of the problem.

ii.

g(n) is the path cost (so far) from the start node to n.
h(n) is the heuristic function which is the estimated cost from n to the goal.

iii.

Evaluation functions: Instead of searching all the possible moves as in the original Minimax algorithm which poses the limited source problem, the evaluation function focuses simply on the current position and estimates its expected utility. It agrees with the utility function on terminal states and gives an accurate reflection on the actual chances of winning.

Cutoff test: Since the max depth of the original Minimax algorithm may be too large to search through, a fixed depth limit (e.g. the time limit of the game) is set to search only a partial of moves. Iterative deepening can be used to achieve the same function.

iv.

***. Definition**

b	Branching factor – the maximum (or averaged) branching factor of the search tree. b is assumed to be finite.
d	Depth – the depth of the least-cost solution.
m	Max depth – the maximum depth of the state space or the search tree.

1. Breadth-first Search

Breadth-first search will expand the *shallowest* unexpanded node. The lower or the deeper the fringe is, the later it will be queued (FIFO). It has the same time and space complexity, but the memory requirement poses a bigger problem than the execution time.

Completeness

Yes. As it searches for the whole tree, it will complete. (Note: b is assumed to be finite.)

Time Complexity

$\mathbf{O}(b^{d+1})$**.*** The worst case would be the goal state is the bottom-rightmost node. Searching all $1 + b + b^2 + \cdots + b^d$ nodes in the tree with one more layer for the stopping condition, the time needed would be $b \times b^d = b^{d+1}$.

Space Complexity

$\mathbf{O}(b^{d+1})$**.*** Since each nodes are queued at the end of the queue and FIFO, every node is kept in memory. Hence, the space complexity is also $\mathbf{O}(b^{d+1})$**.**

Optimality

Yes, if each path cost is the same or non-decreasing towards the bottom. BFS searches for the shallowest goal, level by level; in order for it to be also the optimal goal, all path cost must be the same to ensure deeper levels has less optimal solutions.

* Remarks: Depending on the algorithm, the time and space complexity can be $\mathbf{O}(b^d)$ if the expansion is postponed (i.e. a goal test at the end of each loop).

2. Depth-first Search

Breadth-first search will expand the *deepest* unexpanded node. Successors are put at the front of the queue (LIFO). The disadvantage of DFS is 1) it cannot handle spaces with infinite-depth and loops, 2) time needed is huge if m is much larger than d, and 3) it is non-optimal. However, the space complexity of DFS is great since it only deals with linear space, rather than exponential in BFS.

Completeness

No. When encountered with infinite-depth spaces or loops, it will continuously expand on a single fringe and fails. It might complete in finite spaces where repeated states along the path are remembered and avoided.

Time Complexity

$\mathbf{O}(b^m)$**, if** m **is finite.** Since it will expand towards the deepest level before unwinding to its predecessors (previous node), the time needed is $\mathbf{O}(b^m)$.

Space Complexity

$\mathbf{O}(bm)$**.** DFS only has to remember a single path (m) with all the previously expanded nodes (b) in any given moment.

Optimality

No. DFS could return the deepest solution rather than the shallowest node.

v.

1. Consistency implies Admissibility

For a node n and it's successor node n', a heuristic is **consistent** if the estimated distance from n, $h(n)$, is less than or equal to the step cost from n to n', $c(n, n')$ and the estimated distance from n', $h(n')$, combined, i.e.

$$h(n) \leq c(n, n') + h(n')$$

and the estimated distance at the goal state, G, is 0.

$$h(G) = 0$$

A heuristic is admissible when the cost of reaching the goal is not overestimated,

$$h(n) \leq h^*(n)$$

where $h^*(n)$ is the actual cost from n to the goal.

Base Case

We first consider the goal state. Since $h(G) = h^*(G) = 0$, for a node G_{d+1} with length $d + 1$ that passes through the immediate successor goal G_d at layer d,

$$\begin{aligned} h(G) &\leq c(G_{d+1}, G_d) + h(G_d) \\ h(G) &\leq c(G_{d+1}, G_d) + h^*(G_d) \\ h(G) &\leq h^*(G_{d+1}) \end{aligned}$$

The heuristic is admissible at the goal state.

Hypothesis

The above holds true for all nodes n_{d+i+1} (with distance $i + 1$ from the goal state) that passes through an immediate successor n_{d+i}, i.e.

$$h(n_{d+i+1}) \leq h^*(n_{d+i})$$

Proof by Induction

From the rule of consistency,

$$h(n_{d+i+1}) \leq c(n_{d+i+1}, n_{d+i}) + h(n_{d+i})$$

and from the base case,

$$c(n_{d+i+1}, n_{d+i}) + h(n_{d+i}) \leq c(n_{d+i+1}, n_{d+i}) + h^*(n_{d+i})$$

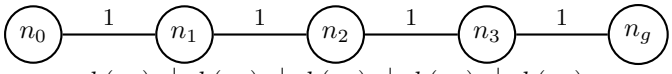
Since $h^*(n_{d+i+1}) = c(n_{d+i+1}, n_{d+i}) + h^*(n_{d+i})$

$$\begin{aligned} h(n_{d+i+1}) &\leq c(n_{d+i+1}, n_{d+i}) + h^*(n_{d+i}) \\ h(n_{d+i+1}) &\leq h^*(n_{d+i+1}) \end{aligned}$$

Thus, for heuristics in A^*, consistency implies admissibility.

2. Admissibility does NOT implies Consistency

Consider the following case:



$h(n_0)$	$h(n_1)$	$h(n_2)$	$h(n_3)$	$h(n_g)$
3	1	1	1	0
$h^*(n_0)$	$h^*(n_1)$	$h^*(n_2)$	$h^*(n_3)$	$h^*(n_g)$
4	3	2	1	0

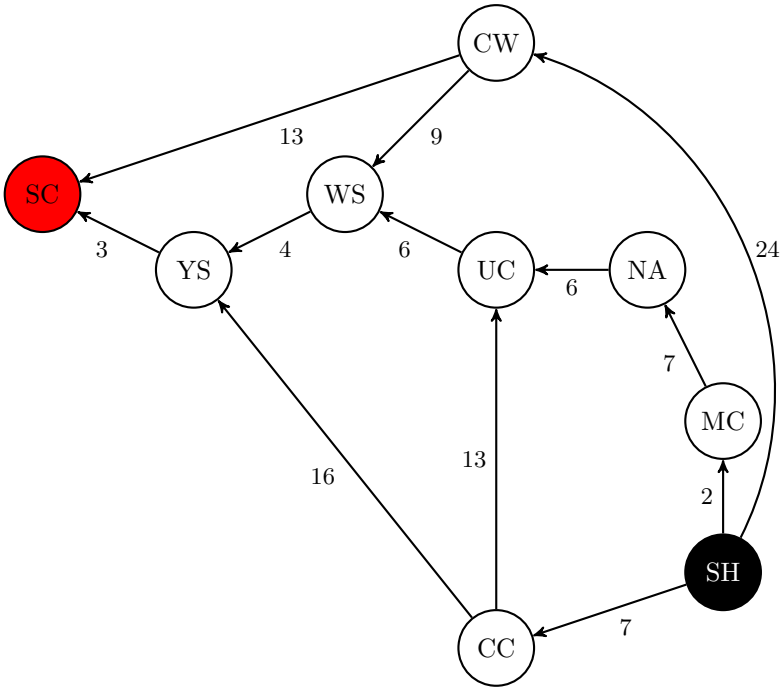
The heuristic is admissible, as for every node n, $h(n) \leq h^*(n)$, and $h(n_g) = h^*(n_g) = 0$. However, since

$$h(n_0) \geq c(n_0, n_1) + h(n_1)$$
$$3 \geq 1 + 1$$

it is not consistent where $h(n_0) \leq c(n_0, n_1) + h(n_1)$ should holds true.

Thus, for heuristics in A^*, admissibility does not implies consistency.

Question b



$h(SH)$	$h(MC)$	$h(CC)$	$h(NA)$	$h(CW)$	$h(UC)$	$h(WS)$	$h(YS)$
24	20	17	16	12	10	6	3

i. Uniform Cost Search

Labels

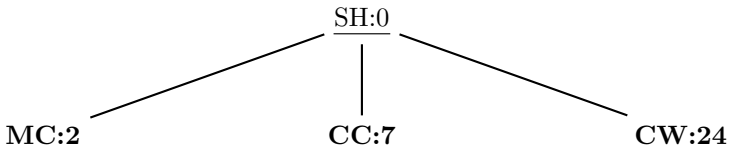
In uniform cost search,

Node N : Cost

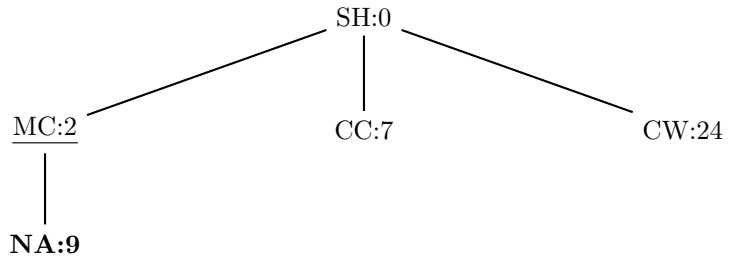
The cost, $g(N)$, is the sum of path cost from initial node to node N. Iteration 0: Initialize state

SH:0

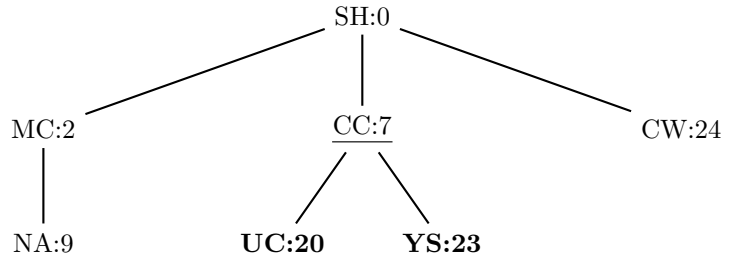
Iteration 1: Expand SH



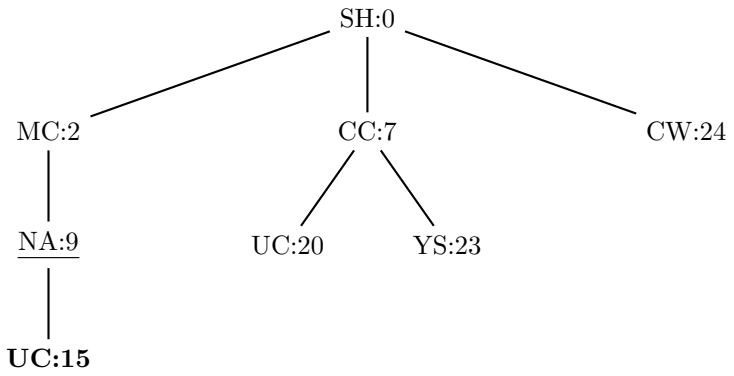
Iteration 2: Expand SH-MC



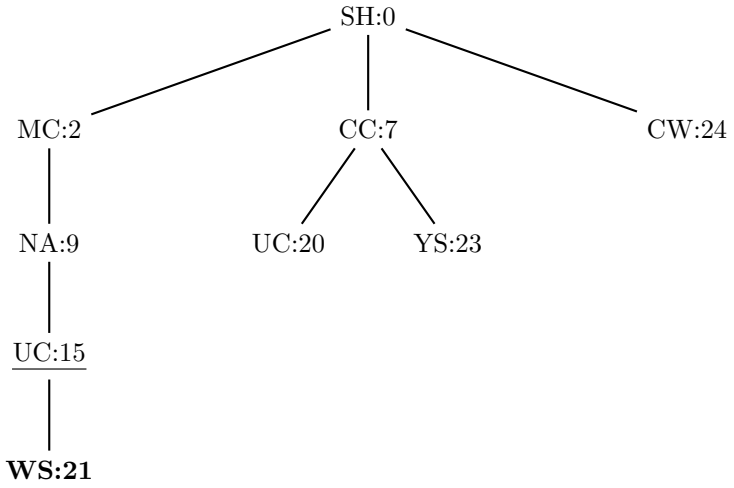
Iteration 3: Expand SH-CC



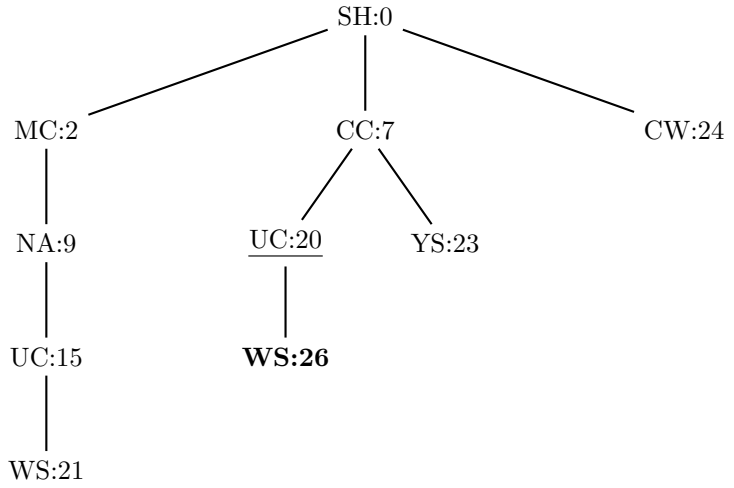
Iteration 4: Expand SH-MC-NA



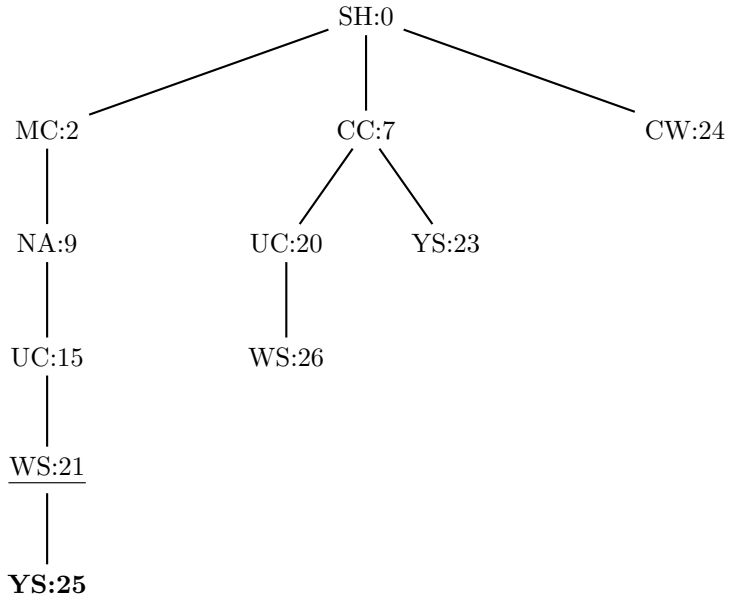
Iteration 5: Expand SH-MC-NA-UC



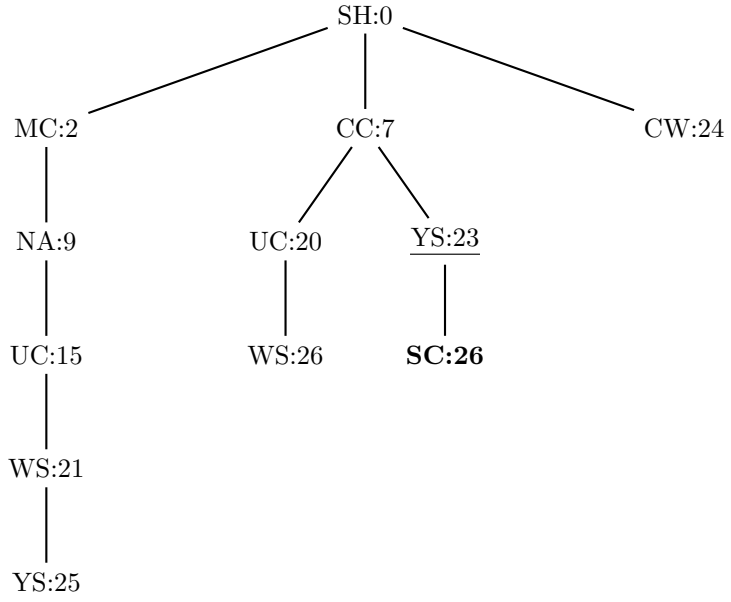
Iteration 6: Expand SH-CC-UC



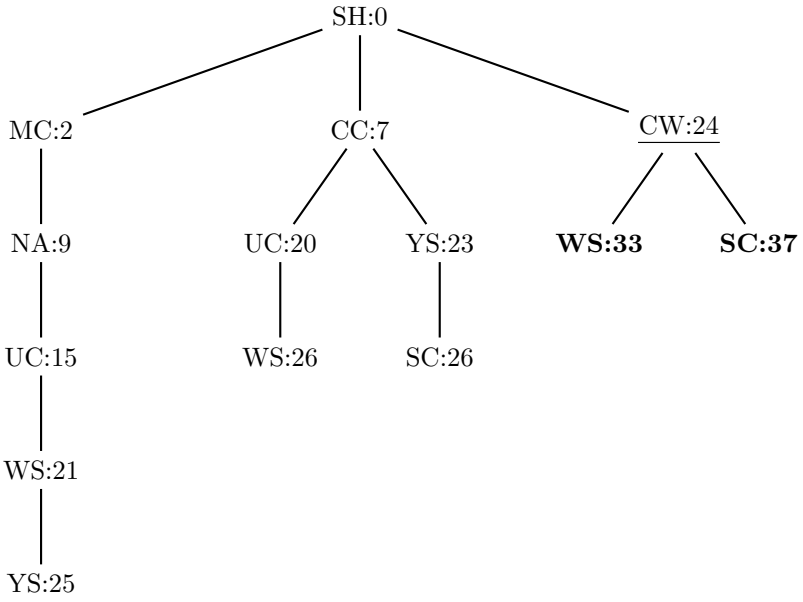
Iteration 7: Expand SH-MC-NA-UC-WS



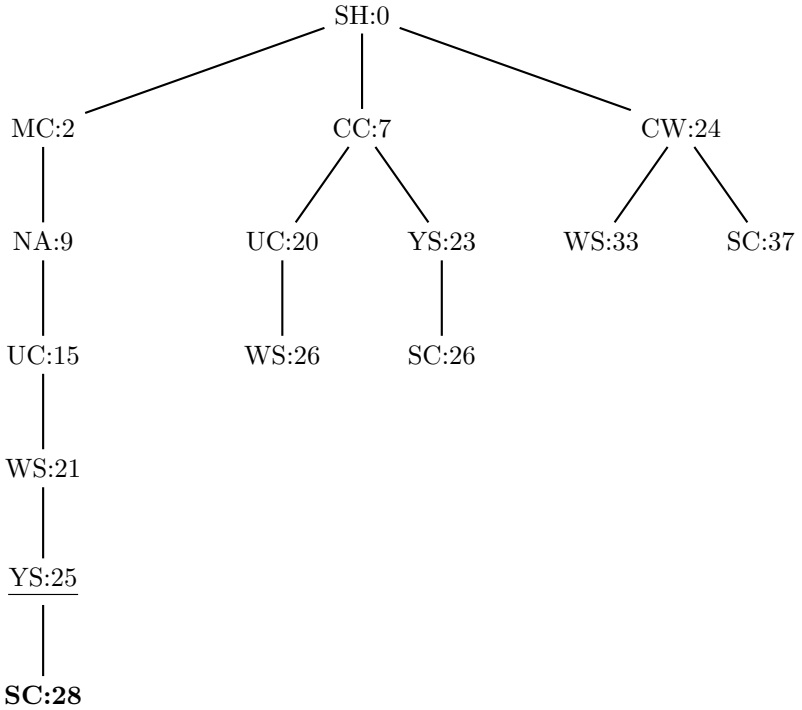
Iteration 8: Expand SH-CC-YS



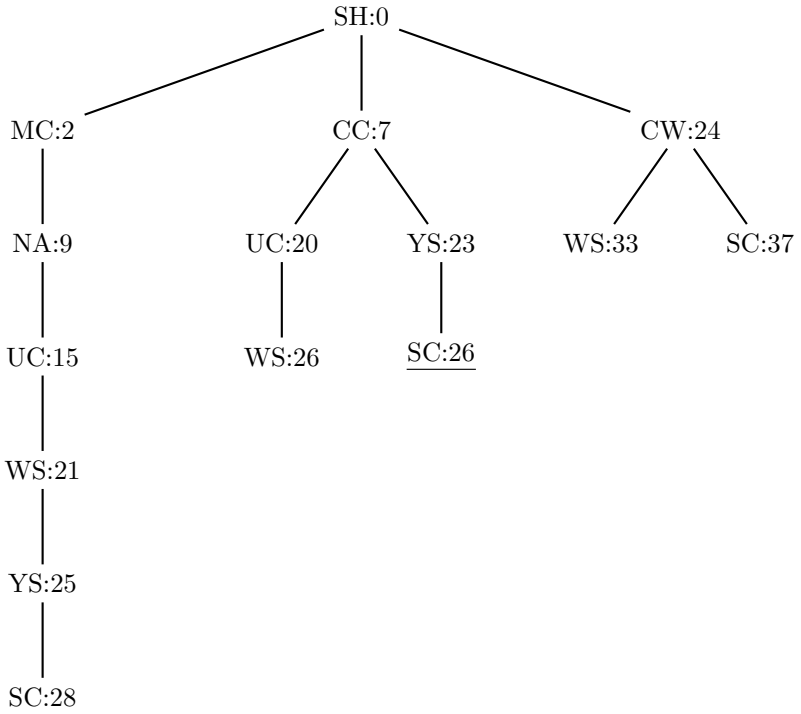
Iteration 9: Expand SH-CW



Iteration 10: Expand SH-MC-NA-UC-WS-YS



Iteration 11: Expand SH-CC-YS-SC → **Goal found**, end.



ii. Recursive Best-first Search

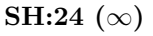
Labels

In recursive best first search,

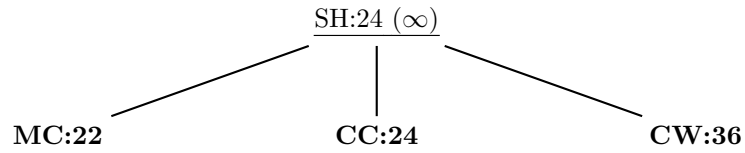
Node N : Cost (f-limit)

The cost, $f(N)$, is the sum of path cost from initial node to node N, $g(N)$, and the heuristic estimated distance from node N to goal state, $h(N)$. The f-limit stores the best alternative.

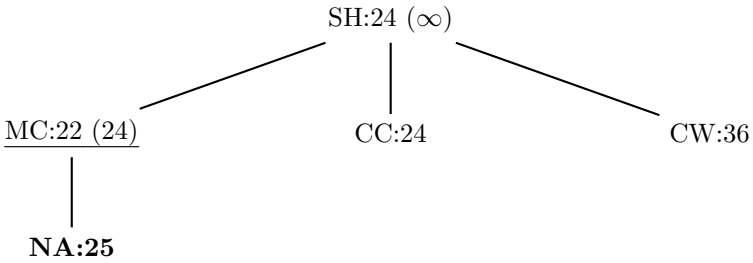
Iteration 0: Initialize state



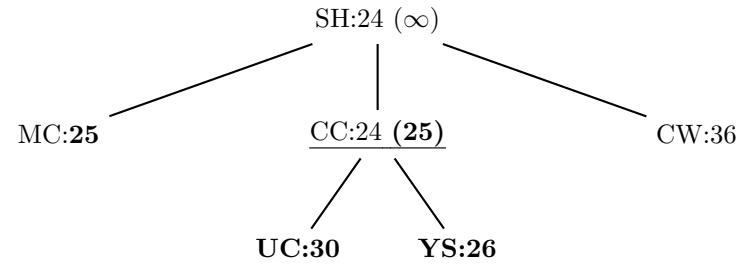
Iteration 1: Expand SH



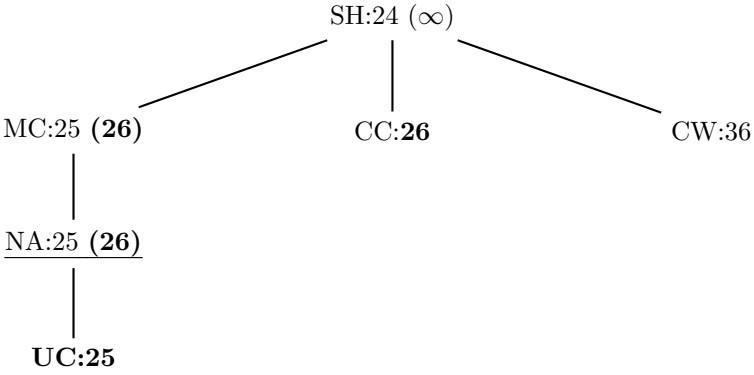
Iteration 2: Expand SH-MC



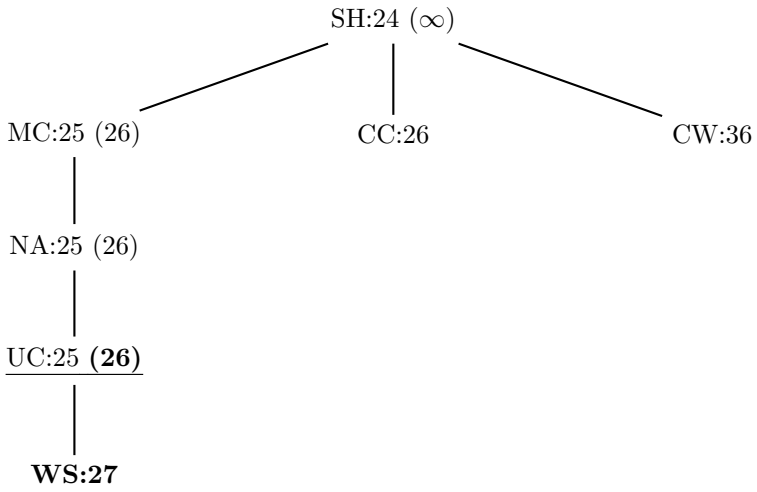
Iteration 3: Unwind, update cost of SH-MC, and expand SH-CC



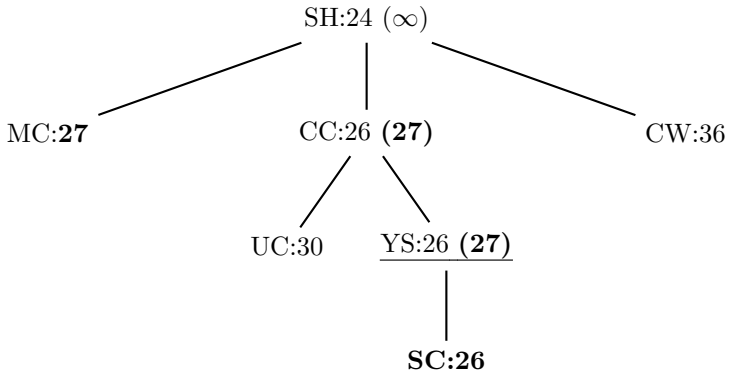
Iteration 4: Unwind, update cost of SH-CC, and expand SH-MC-NA



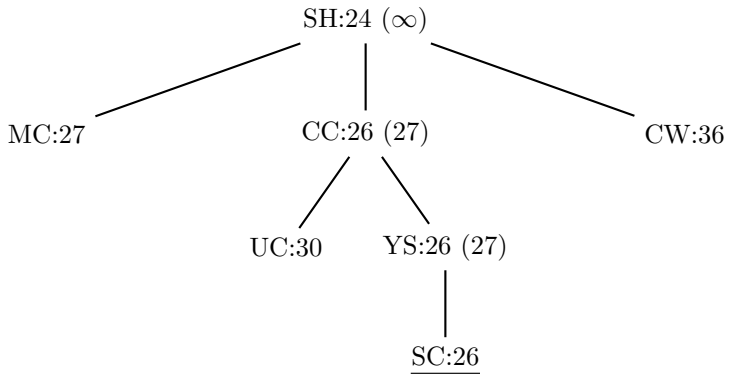
Iteration 5: Expand SH-MC-NA-UC



Iteration 6: Unwind, update cost of SH-MC, and expand SH-CC-YS



Iteration 7: Expand SH-CC-YS-SC → **Goal found**, end.



Question c

Concept of Pruning

If the search space or the number of nodes are too large, it may require too much time to transverse the whole search tree and find the optimal solution. **Pruning** is used to eliminate a branch (and its subsequent sub-tree) of the search tree, thus reducing its size and the time needed to search. The eliminated branches are irrelevant and do not affect the final result.

Basic Principle of alpha-beta Pruning

A player will not choose an outcome if they has a better option at hand. If the player knows enough information about the outcome of a particular path/node, and compares it to their best current option, they can choose to keep or discard (*prune*) the branch/sub-tree and search for other options.

In a Minimax algorithm, at any node n that is available to the player, if they has a better choice either at the the parent node of n or at any choice points further up, then the player will not choose it. The variable α and β respectively stores the minimum points (utility value) the maximizing player (the α-player) and the maximum points the minimizing player (the β-player) can get. Whenever a node with value v is larger than β (in max-level, i.e. β-player's turn) or smaller than α (in min-level, i.e. α-player's turn), the other player at the previous layer will not choose that path; thus, that sub-tree can be pruned.

$\alpha - \beta$ Pruning Example