HW 1 (Written Homework 1)
Worked by Jeong Yong Yang && Junho Son

Question 1:
States:

A state is represented by a tuple (j3, j8, j12) where:

1. j3 is the amount of water in the 3-gallon jug.
2. j8 is the amount of water in the 8-gallon jug.
3. j12 is the amount of water in the 12-gallon jug.

The initial state of the problem starts with 0 gallons of water in each jug: (0, 0, 0)
The goal state is reached when any one of the jugs contains exactly 1 gallon of water.

Actions:

There are three types of actions with sub-actions of the actions exist.

1. Fill the jug from sink
    a. Fill 3-gallon jug from the sink
    b. Fill 8-gallon jug from the sink.
    c. Fill 12-gallon jug from the sink.
2. Empty the jug to sink
    a. Empty 3-gallon jug to the sink
    b. Empty 8-gallon jug to the sink
    c. Empty 12-gallon jug to the sink
3. Transfer the water from one jug to another jug
    a. Pour from 3-gallon jug into 8-gallon jug without overflow.
        i. When pouring from 3-gallon jug into 8-gallon jug without overflow, if there is water remaining on the 3-gallon jug, the remaining water on the 3-gallon jug has to go into the 12-gallon jug and not cause overflow to the 12-gallon jug (since the 3-gallon jug has to be empty due to the fact that the rules state that the 3-gallon jug has to be empty after the pouring action and the remaining water from the 3-gallong jug cannot go into the sink)
    b. Pour from 3-gallon jug into 12-gallon jug without overflow.
        i. When pouring from 3-gallon jug into 12-gallon jug without overflow, if there is water remaining on the 3-gallon jug, the remaining water on the 3-gallon jug has to go into the 8-gallon jug and not cause overflow to the 8-gallon jug (since the 3-gallon jug has to be empty due to the fact that the rules state that the 3-gallon jug has to be empty after the pouring action and the remaining water from the 3-gallong jug cannot go into the sink)

c. Pour from 8-gallon jug into 3-gallon jug without overflow.
    i. When pouring from 8-gallon jug into 3-gallon jug without overflow, if there is water remaining on the 8-gallon jug, the remaining water on the 8-gallon jug has to go into the 12-gallon jug and not cause overflow to the 12-gallon jug (since the 8-gallon jug has to be empty due to the fact that the rules state that the 8-gallon jug has to be empty after the pouring action and the remaining water from the 8-gallong jug cannot go into the sink)

d. Pour from 8-gallon jug into 12-gallon jug without overflow.
    i. When pouring from 8-gallon jug into 12-gallon jug without overflow, if there is water remaining on the 8-gallon jug, the remaining water on the 8-gallon jug has to go into the 3-gallon jug and not cause overflow to the 3-gallon jug (since the 8-gallon jug has to be empty due to the fact that the rules state that the 8-gallon jug has to be empty after the pouring action and the remaining water from the 8-gallong jug cannot go into the sink)

e. Pour from 12-gallon jug into 3-gallon jug without overflow.
    i. When pouring from 12-gallon jug into 3-gallon jug without overflow, if there is water remaining on the 12-gallon jug, the remaining water on the 12-gallon jug has to go into the 8-gallon jug and not cause overflow to the 8-gallon jug (since the 12-gallon jug has to be empty due to the fact that the rules state that the 12-gallon jug has to be empty after the pouring action and the remaining water from the 12-gallong jug cannot go into the sink)

f. Pour from 12-gallon jug into 8-gallon jug without overflow.
    i. When pouring from 12-gallon jug into 8-gallon jug without overflow, if there is water remaining on the 12-gallon jug, the remaining water on the 12-gallon jug has to go into the 3-gallon jug and not cause overflow to the 3-gallon jug (since the 12-gallon jug has to be empty due to the fact that the rules state that the 12-gallon jug has to be empty after the pouring action and the remaining water from the 12-gallong jug cannot go into the sink)

Transition Model:

Transition model is the resulting state after an action. For example, if action is “empty the 8-gallon jug from the sink” in state (0, 5, 6), the result state would be (0, 0, 6).

Goal Test:

Here, we check if any of the jugs reach the goal. In other words, if any of the jugs contain exactly 1 gallon of water (j3 == 1, j8 == 1, or j12 == 1), then the goal condition is met.

Path Cost:

Path cost is a cost for doing one action. In this scenario, all of the actions have the same cost of 1.

Question 2:

According to the question, we are picking a distance function and need to engineer the edge costs in such a way that allows the distance function to be admissible and consistent. The edge cost of the graph should be greater than or equal to the maximum difference between the heuristic values of all adjacent nodes. For example, in the first programming assignment where the value of the heuristic function moving from one node to its adjacent node was set to be 99999 if the adjacent node was within the enemy attack radius, the edge cost should be greater than or equal to 99999.

We only need to prove that our heuristic function is consistent, since consistency naturally implies admissibility as well. Consistency refers to the triangle inequality but for action spaces. In other words, if there are vertices 'u' and 'v', the cost of vertex 'u' to the goal is less than or equal to cost of vertex 'v' to the goal added with cost of using action to go from from 'u' to 'v'. In mathematical equation,
$h(u) <= c(u,v) + h(v)$.

Reorganizing the equation above gives
$h(u) - h(v) <= c(u,v)$ for all adjacent vertices 'u' and 'v'.

Since it has to hold for all adjacent vertices 'u' and 'v', what we actually get is
Maximum of $h(u) - h(v) <= c(u,v)$ for all adjacent vertices 'u' and 'v'.

Therefore, to ensure that the heuristic function is consistency, and therefore admissible, it means that the edge weight should be greater than or equal to the maximum possible difference between the heuristic values of every two adjacent vertices.

Question 3:

All CSPs can be converted into binary CSPs.

All CSPs consist of variables, domains of the variables, and constraints. We can reduce a n-ary constraints by adding synthetic variable(s). Denote this newly invented variable as S. The domain of S, aka DS, will be the Cartesian product of the domains of the n original variables.

For a constraint involving n variables, let's introduce a synthetic variable S.
The domain of S, D_s , is the Cartesian product of the domains of the n original variables.

Therefore, given that

1. Variables (V): V1, V2, …, Vn
2. Domains (D): D1, D2, …, Dn
3. Constraints (C) involving variables V1, V2, …, Vn

We can create DS to be all possible cartesian combinations.
DS = DS1, DS2, …, DSn

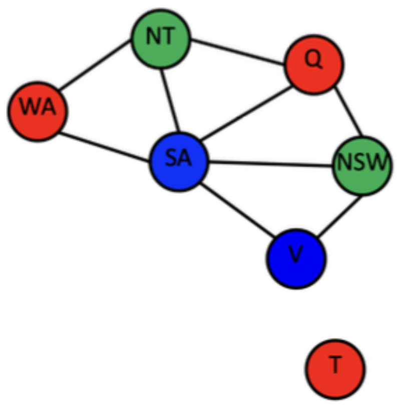
Where each element of DS is a tuple or list of domains.
For example,
DS1 = (d1, d2, …, dn) with each d1 through dn being values from the domain set D above (D1, D2, …, Dn).

Then, for each variable V: V1, V2, …, Vn in the original n-ary CSP, we can create a binary constraint between Vs and DS. By using this constraint, we can ensure that the ith component of the value assigned to DS is equal to the value assigned to Vs.

Then, we can prune the values from DS that does not satisfy with the constraint C provided by the original constraint on n-ary CSP. In other words, only the list or tuple that satisfies C remains → (D1, D2, …, Dn) remains on the list if and only if (D1 = V1, …, Dn = Vn) that satisfies C.

For example, in the class, we used coloring of the different regions of Australia with three distinct colors using Red (R), Blue (B), and Green (G) where no each adjacent regions have the same coloring.

We were given seven distinct regions of the picture:

→ 

Now, we denote a newly created variable DS where DS contains all possible cartesian product of the variables:

DS = {(RRRRRRR), (RRRRRRB), …, (GGGGGGG)} where the ordering goes WA, NT, SA, Q, NSW, V, T.

Here, WA will equal to the first component of the cartesian products of all DS, NT will equal to the second component of all DS, and so on. Then, we can delete all the combinations that do not satisfy the constraints where each region has to be different coloring from its adjacent regions. If we delete/prune them all, only the tuple that matches with the constraints will remain on DS.

Question 4:

Proof by induction (as recommended by the hint)

v = Minimax(s)
v ′ = Alpha-Beta-Pruning(s, α, β)

Claims :
• If α ≤ v ≤ β then v ′ = v
• If v ≤ α then v ′ ≤ α
• If v ≥ β then v ′ ≥ β

Base Case:
If s is a terminal state, the Minimax on s and Alpha-Beta Pruning on s both return the same utility value of s because they evaluate terminal states identically and therefore produce the same utility value.

Inductive Step:
If s is not a terminal state, we correctly assume that the claims above hold as true for all children of s. Using this assumption, we will prove that it also holds for s.

1. If α ≤ v ≤ β then v ′ = v where v = Minimax(s) and v ′ = Alpha-Beta-Pruning(s, α, β)

If s is at a maximizing state, then v = max {child for child in children(s)}. Using the inductive step, for every child of the state s, if α and β are set within the proper bound of α ≤ v(child) ≤ β, the utility value of v' (the Alpha-Beta-Pruning) will be equal to v(child) since it also wants to get the maximum utility value among the children. Therefore, when the Alpha-Beta-Pruning is ran on the non-terminal state s with initial values of α and β, it returns the same value as the minimax algorithm, giving v' = v.

Similarly, if s is at the minimizing state, then v = min {child for child in children(s)}. Using the inductive step, for every child of the state s, if α and β are set within the proper bound of α ≤ v(child) ≤ β, the utility value of v' will be equal to v(child) since it also wants to get the minimum utility value among the children, giving v' = v.

2. If v ≤ α then v ′ ≤ α

If s is at a maximizing state and the minimax algorithm returns a value of v ≤ α, this means that there exists a child or children of s with an utility value less than or equal to α. Therefore, when running the Alpha-Beta-Pruning algorithm, when it encounters the first child of the non-terminal state s that has an utility value less than or equal to α, it would naturally prune the remaining children without exploring the branch in depth since the value of the utility value of the remaining children would not be less than α. Therefore, if $v \leq \alpha$ then $v' \leq \alpha$.

If s is a minizing state, the Alpha-Beta-Pruning algorithm would return the value of v' that is less than or equal to α since it naturally returns the least (minimum) value among all the children. Therefore, if $v \leq \alpha$ then $v' \leq \alpha$.

3. If $v \geq \beta$ then $v' \geq \beta$

If s is a maximizing state, then the Alpha-Beta-Pruning algorithm would return the value of v' that is greater than or equal to β since it naturally returns the greatest (maximum) value among all the children. Therefore, if $v \geq \beta$ then $v' \geq \beta$.

If s is a minizing state and minimax algorithm returns a value of $v \geq \beta$, this refers to the fact that there exists a child or children of s with an utility value that are greater than or equal to β. When running the Alpha-Beta-Pruning algorithm, when it encounters first children that has an utility value greater than or equal to β, it would cause the Alpha-Beta-Pruning algorithm to prune the remaining children without exploring other children since the value of the utility value would not be greater than β. Therefore, if $v \geq \beta$ then $v' \geq \beta$

Using proof by induction, we can conclude that the three statements given above are all true for any state s.

This means that the Alpha-Beta-Pruning, when initialized with -infinity for α and +infinity for β, it would always return the same utility value as the Minimax algorithm would provide.

Extra Credit: Gradient Descent and Lipschitz Continuity (20 points)

First of all, we have the gradient descent equation:

$$\vec{x}_{t+1} = \vec{x}_t - \eta \nabla f(\vec{x}_t)$$

The question states that the gradient is Lipschitz continuous with the constant $c > 0$, we get that:

$$\forall \vec{x}, \vec{y} \quad \|f(\vec{x}) - f(\vec{y})\|_2 \leq c\|\vec{x} - \vec{y}\|_2 \quad \text{where} \quad \vec{x} \neq \vec{y}$$

$$\frac{\|f(\vec{x}) - f(\vec{y})\|_2}{\|\vec{x} - \vec{y}\|_2} \leq c$$

Denote $\vec{x} = \vec{y} + \delta$ where δ is the change, and we know that δ can't be 0 since $\vec{x} \neq \vec{y}$:

$$\lim_{\delta \to 0} \frac{\|f(\vec{y} + \delta) - f(\vec{y})\|_2}{\|\vec{y} + \delta - \vec{y}\|_2} \leq c$$

$$\lim_{\delta \to 0} \frac{\|f(\vec{y} + \delta) - f(\vec{y})\|_2}{\|\delta\|_2} \leq c$$

This is simply the derivative equation, giving:

$$\|f'(\vec{y})\|_2 \leq c$$

Applying this to the fact that the gradient descent is Lipschitz continuous, we get:

$$\|\nabla^2 f(\vec{x})\|_2 \leq c \cdot I \leq c \cdot 1$$

where 1 indicates the 1 matrix.

If we apply the 2nd degree polynomial of f centered around $f(\vec{x})$, we get:

$$f(\vec{y}) \leq f(\vec{x}) + \nabla f(\vec{x})^T(\vec{y} - \vec{x}) + \frac{1}{2}\nabla^2 f(\vec{x})(\vec{y} - \vec{x})(\vec{y} - \vec{x})^T$$

Plugging $\vec{y} = \vec{x}_{t+1}$, we get:

$$f(\vec{x}_{t+1}) \leq f(\vec{x}_t) + \nabla f(\vec{x}_t)^T(-\eta\nabla f(\vec{x}_t)) + \frac{1}{2}\nabla^2 f(\vec{x}_t)(-\eta\nabla f(\vec{x}_t))(-\eta\nabla f(\vec{x}_t))^T$$

$$= f(\vec{x}_t) - \eta\|\nabla f(\vec{x}_t)\|_2^2 + \frac{\eta^2}{2}\nabla^2 f(\vec{x}_t)\|\nabla f(\vec{x}_t)\|_2^2$$

From the previous fact about Lipschitz continuity, we know that $\nabla^2 f(\vec{x}_t) \leq c \cdot 1$. Therefore,

$$f(\vec{x}_{t+1}) \leq f(\vec{x}_t) - \eta \|\nabla f(\vec{x}_t)\|_2^2 + \frac{\eta^2 c}{2} \|\nabla f(\vec{x}_t)\|_2^2$$
$$\leq f(\vec{x}_t) + \eta \|\nabla f(\vec{x}_t)\|_2^2 \left(\frac{\eta c}{2} - 1\right)$$

Now, we know that the value of $\|\nabla f(\vec{x}_t)\|_2^2$ is always positive due to the square. We also know that $f(\vec{x}_{t+1})$ and $f(\vec{x}_t)$ are of the same signs.

If $f(\vec{x}_{t+1})$ and $f(\vec{x}_t)$ are positive, we get:

$$f(\vec{x}_{t+1}) \leq f(\vec{x}_t) + \eta \|\nabla f(\vec{x}_t)\|_2^2 \left(\frac{\eta c}{2} - 1\right)$$

Here, $f(\vec{x}_{t+1})$, $f(\vec{x}_t)$, and $\|\nabla f(\vec{x}_t)\|_2^2$ are positive, giving the information that $\eta \left(\frac{\eta c}{2} - 1\right)$ has to be less than or equal to 0 for the inequality to hold, in order for the gradient to converge.

$$\eta \left(\frac{\eta c}{2} - 1\right) \leq 0$$
$$\frac{\eta c}{2} - 1 \leq 0$$
$$\frac{\eta c}{2} \leq 1$$
$$\eta \leq \frac{2}{c}$$

If $f(\vec{x}_t)$ and $f(\vec{x}_{t+1})$ are both negative, we get:

$$f(\vec{x}_{t+1}) \geq f(\vec{x}_t) - \eta \|\nabla f(\vec{x}_t)\|_2^2 \left(\frac{\eta c}{2} - 1\right)$$

Since they are negative, I multiplied the sides by -1 to make them positive. Now, here $f(\vec{x}_{t+1})$, $f(\vec{x}_t)$, and $\|\nabla f(\vec{x}_t)\|_2^2$ are all positive again. This means that $-\eta \left(\frac{\eta c}{2} - 1\right)$ should be greater than or equal to 0 in order to make the gradient converge, giving:

$$-\eta \left(\frac{\eta c}{2} - 1\right) \geq 0$$
$$\eta \left(\frac{\eta c}{2} - 1\right) \leq 0$$
$$\frac{\eta c}{2} - 1 \leq 0$$
$$\frac{\eta c}{2} \leq 1$$
$$\eta \leq \frac{2}{c}$$

which is the same as the inequality of η, the step size.

Therefore, we here proved that the gradient is guaranteed to converge for specific values of the step size η by proving that it does converge when $\eta \leq \frac{2}{c}$.