

2021

IA FINAL PROJECT



Alejandro Berraondo

Marc Lozano

31-5-2021

Index

PATHFINDER..... 3

PLANNER 0

 World 0

 Actions..... 0

 Statistics 2

BEHAVIOR TREE..... 6

FINAL PROJECT 9

 What kind of world we have created..... 9

 Pathfinder specs..... 11

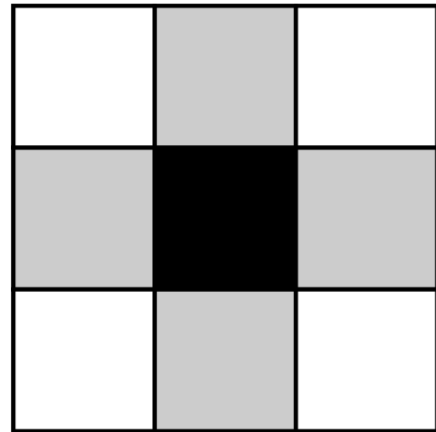
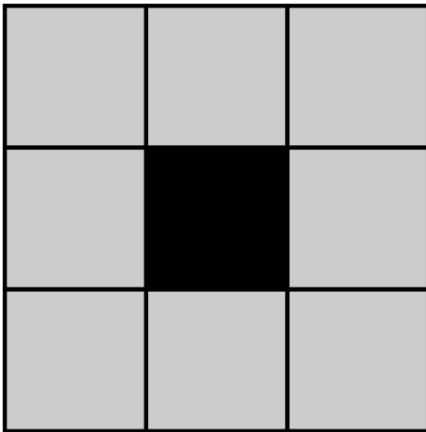
 Planner specs 12

PATHFINDER

The first thing we do in the FindPath function is to analyze the nodes that are in openSet and get the lowest fCost and then analyze it and configure it properly

```
for (int i = 1; i < openSet.Count; i++)
{
    if (openSet[i].fCost <= node.fCost && openSet[i].hCost < node.hCost)
    {
        node = openSet[i];
    }
}
```

Once we have added the node to the closedSet and verified that it is not the final destination node, we analyze the neighbors of this, here is important the euristica that we will use (Eight Connectivity or Four Connectivity) If we use Eight Connectivity we will analyze the neighbors as we can see in the left figure of the image below, but if we use Four Connectivity we will analyze the neighbors as in the right figure of the image below.



By analyzing the neighbors we can choose between them which is the most optimal way we must go to our destination, for this it must be transpassable, thus excluding walls or obstacles that prevent us from passing, and that this neighbor is in the closedSet.

Once we have decided this we must calculate the fCost to see if it is optimal to address that neighbor, for this we add the gCost, the real cost of the road traveled to reach that node from the initial node, the hCost, it represents the heuristic value of the node to be evaluated from the current to the end, and a cost multiplier, which tells us whether there is penalty for using that path or not, for example, moving an area of water or sand costs more than walking a paved road.

If this fCost obtained is less than the neighbor's gCost, it is best to go through that neighbor, so you will be assigned the new gCost as the value obtained above, the hCost as the new heuristic that depends on the connectivity used (we will talk about this function later) And the mParent as the node from which we came.

```

if (!neighbour.mWalkable || closedSet.Contains(neighbour)) continue;

float newMovementCostToNeighbour = node.gCost /* * node.mCostMultiplier */ + Heuristic(node, neighbour) * node.mCostMultiplier;
if (newMovementCostToNeighbour < neighbour.gCost || !openSet.Contains(neighbour))
{
    neighbour.gCost = newMovementCostToNeighbour;
    neighbour.hCost = Heuristic(neighbour, CurrentTargetNode);
    neighbour.mParent = node;

    if (!openSet.Contains(neighbour))
        openSet.Add(neighbour);
}

```

Once the target is found it is time to build the path, so we will pick up the final node and with the help of the process we have done before, we will be able to reach the initial node with the help of mParent, so we will add these nodes to a list that we will later reverse (because they are added in reverse order) to have a path that goes from the source to the destination.

```

NodePathfinding currentNode = endNode;

while (currentNode != startNode)
{
    path.Add(currentNode);
    currentNode = currentNode.mParent;
}

path.Reverse();
if(SmoothingPath) SmoothPath(path);

```

As for the heuristics function, this depends on the connectivity we have, if we use the Eight Connectivity we will be able to analyze not only the adjacent nodes but also the diagonal ones, but with Four Connectivity we will only analyze the adjacent ones. The latter is also known as Manhattan Distance, which consists of the sum of the difference of Node A and Node B in its absolute value, in eight connectivity, however, we use pythagoras, the square root of the sum of the squares of the difference of Node A and Node B

```

float Heuristic(NodePathfinding nodeA, NodePathfinding nodeB)
{
    // Heuristic function
    float dstX = nodeA.mGridX - nodeB.mGridX;
    float dstY = nodeA.mGridY - nodeB.mGridY;
    /***/
    if (EightConnectivity)
    {
        /***/
        return Mathf.Sqrt(dstX * dstX + dstY * dstY);
        /***/
    }
    else
    {
        /***/
        //return dstX * dstX + dstY * dstY;
        return Mathf.Abs(dstX) + Mathf.Abs(dstY);
        /***/
    }
}

/***/
}

```

As for the path smoothing we use Bresenham, for this we check the critical nodes. The critical nodes are those where the path curves or does not follow in a straight or linear way, for this we only obtain those points by discarding the rest and mostranthen separated by a straight line.

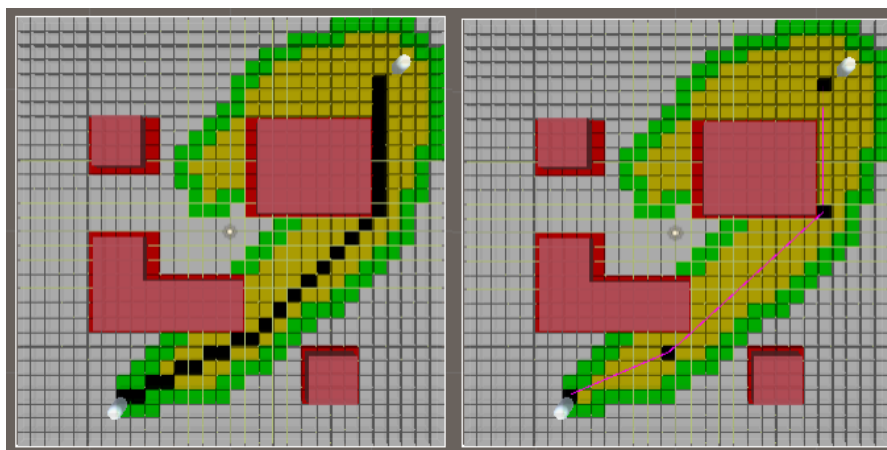
```
void SmoothPath(List<NodePathfinding> path)
{
    List<int> removables = new List<int>();
    NodePathfinding node = path[0];
    removables.Add(0);

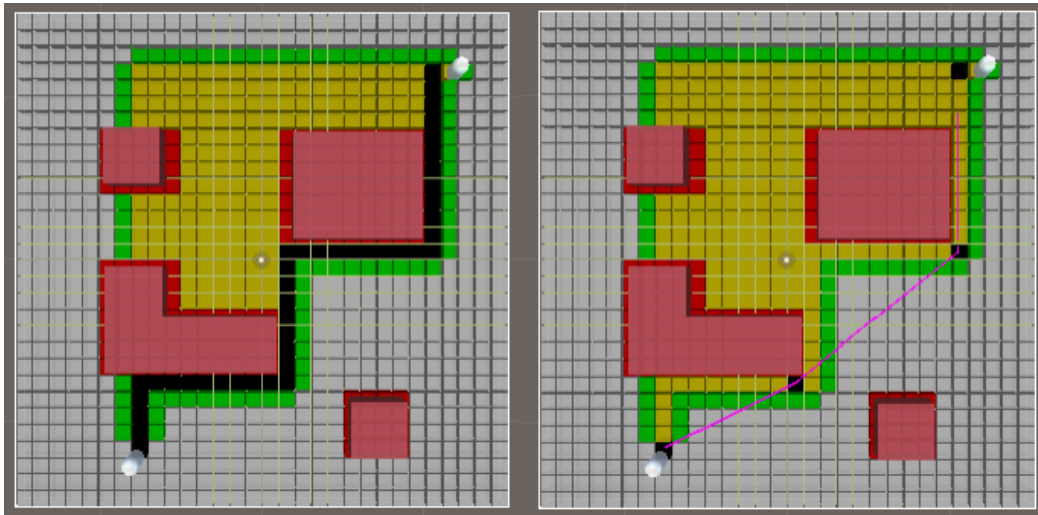
    for (int i = 1; i < path.Count; i++)
    {
        if (BresenhamWalkable(node.mGridX, node.mGridY, path[i].mGridX, path[i].mGridY))
        {
            if (i + 1 < path.Count && BresenhamWalkable(node.mGridX, node.mGridY, path[i + 1].mGridX, path[i + 1].mGridY))
            {
                removables.Add(i);
            }
        }
        else
        {
            // Reset node
            node = path[i];
            removables.Add(i);
        }
    }

    for (int j = removables.Count - 1; j > 0; j--)
    {
        int index = removables[j];
        path.Remove(path[index]);
    }
}

private void OnDrawGizmos()
{
    if(Grid.path!=null && SmoothingPath)
    {
        for (int i = 0; i < Grid.path.Count; i++)
        {
            if (i < Grid.path.Count - 1)
            {
                Gizmos.color = Color.magenta;
                Gizmos.DrawLine(Grid.path[i].mWorldPosition, Grid.path[i + 1].mWorldPosition);
            }
        }
    }
}
```

In the images below, we can see level 1 of the project where we can observe both the connectivity (first row Eight Connectivity, second row Four Connectivity) and the smoothing (first column No Smoothing, second column Smoothing).





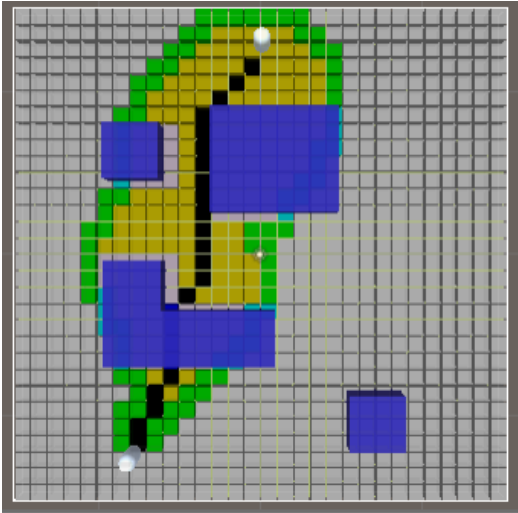
As we can see, in the images above, the path is quite similar between connectivities and smoothing, in terms of smoothing we can see the critical nodes that the pathfinding has selected and how it has joined them together more closely resembling the Eight Connectivity.

Respectively, the values are:

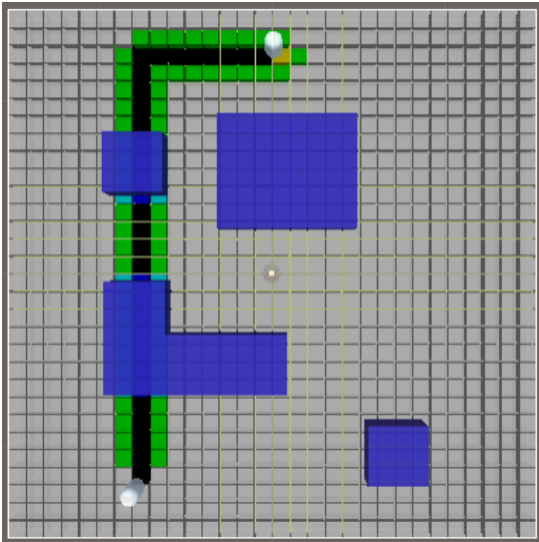
Four Connectivity	Eight Connectivity
Total nodes: 264	Total nodes: 270
Open nodes: 71	Open nodes: 102
Closed nodes: 193	Closed nodes: 168

We can analyze so that thanks to the Eight Connectivity the path uses fewer nodes but costs more because it has to analyze more neighbors, on the other hand the Four Connectivity analyzes fewer nodes but the path is longer.

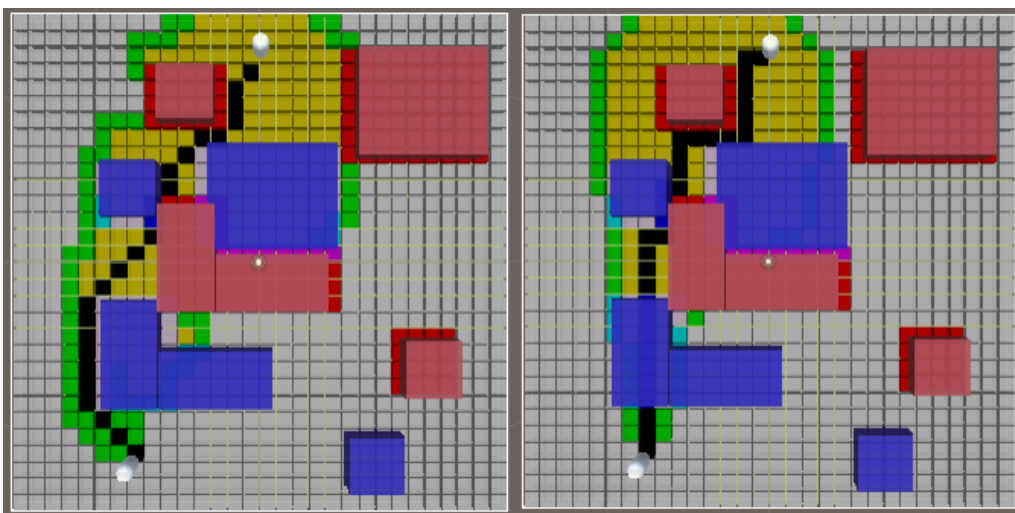
As for the cost of passage, level 2 we can see that in the first lake the pathfinding sees more optimal dodging the lake than while in the second lake, it makes the decision to cross the lake before surrounding it as it is more optimal.



If we do not use the multiplier the path is fully rectilinear and does not take into account the cost of walking through the water so it will go directly to the destination point

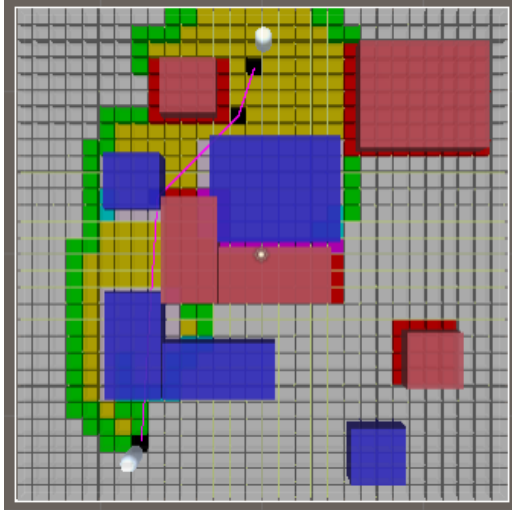


As for level 3, it combines the obstacles of level 1 and water of level 2 we can observe the difference between the connectivity since here they are even more marked, At Eight Connectivity



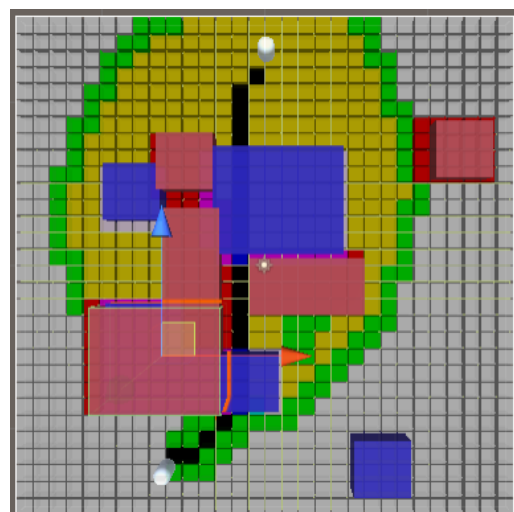
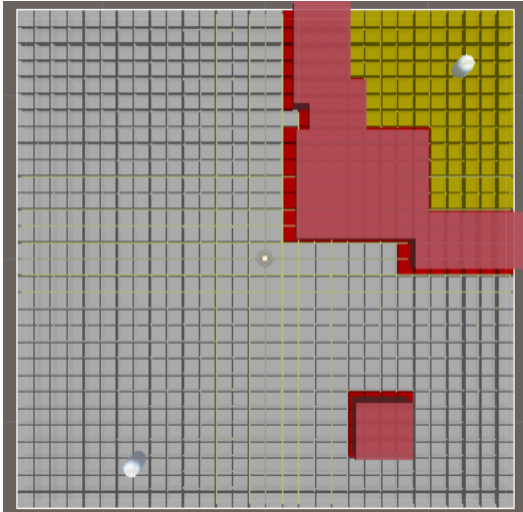
we can see that it surrounds the last lake while Four Connectivity decides to cross it and this is because of the number of nodes they can open at each iteration and the cost of doing so.

As for smoothing in this case a problem happens, and in any connectivity, the last lake decides to cross it in the middle, being that the Eight Connectivity prefers to surround it.



As for the worst cases seen during the course of practia are:

The impossibility of achieving the objective



Leave a direct path through the water or a rodeo, it chooses the water preferably even if you try the rodeo

PLANNER

World

The game is based on the survival of a shipwreck on a desert island, in it the shipwreck must try to survive by getting food, creating a shelter, fire... At the moment the island has a series of materials such as sticks, stones and ropes in addition to a tree which we can cut down if we have the right tools. The states of the world is a mask of 12 states:

- None
- Near tree
- Log out
- Axe owned
- Axe materials owned
- Stick owned
- Near stick
- Stone owned
- Near stone
- Rope owned
- Near rope
- Near crafting table

Actions

As we mentioned above the island is limited in resources, so the actions are also. At the moment the available actions are:

- Go near stick.
- Recollect stick.
- Go near stone.
- Recollect stone.
- Go near rope.
- Recollect rope.
- Go near tree.
- Cut tree.
- Go near crafting table.
- Craft axe

- Have axe materials.

As we can see, the actions are related to crafting such as obtaining materials and building tools for larger purposes such as obtaining wood. So, actions have been generated that are perfectly adaptable for the creation of more tools.

Each action has a precondition so that it can be executed and an effect that impacts on the world, in addition to this we have implemented negative preconditions and negative effects that make the actions can change as the plan is calculated.

```
mActionList.Add(
    new Action(
        Action.ActionType.ACTION_TYPE_CRAFT_AXE,
        WorldState.WORLD_STATE_AXE_MATERIALS_OWNED | WorldState.WORLD_STATE_NEAR_CRAFTING_TABLE,
        WorldState.WORLD_STATE_AXE_OWNED,
        WorldState.WORLD_STATE_AXE_OWNED,
        WorldState.WORLD_STATE_ROPE_OWNED | WorldState.WORLD_STATE_STONE_OWNED | WorldState.WORLD_STATE_STICK_OWNED | WorldState.WORLD_STATE_AXE_MATERIALS_OWNED,
        100.0f, "Axe crafted")
    );
```

For example, to make an axe we need to have all the materials for its fabrication and to be close to the crafting table, in turn the negative precondition tells us that to be able to execute this action we do not have to have an axe in our possession.

If this action is performed, the player/world will own an axe but in turn we will no longer have the rope neither the rope nor the stone nor the stick, therefore neither the materials to make another axe.

Plans

To make a plan it is necessary to know what state of the world we are leaving and which we want to get to.

```
FindPlan(World.WorldState.WORLD_STATE_NONE, World.WorldState.WORLD_STATE_AXE_MATERIALS_OWNED);
```

In this example we start from a null state (there has been no success) and we want to get the materials to build an axe.

```
public List<NodePlanning> GetNeighbours(NodePlanning node)
{
    List<NodePlanning> neighbours = new List<NodePlanning>();

    foreach (Action action in mActionList)
    {
        // If preconditions are met we can apply effects and the new state is valid
        if ((node.mWorldState & action.mPreconditions) == action.mPreconditions && ~(node.mWorldState & action.mNegativePreconditions) == action.mNegativePreconditions)
        {
            // Apply action and effects
            NodePlanning newNodePlanning = new NodePlanning((node.mWorldState | action.mEffects) & ~(action.mNegativeEffects), action);
            neighbours.Add(newNodePlanning);
        }
    }

    return neighbours;
}
```

The main executor of the plan is in the GetNeighbours function where we will check all available actions and if when performing them gives us a valid and optimal state to reach the final state. This check happens in the if, this if consists of two parts: The checking of the preconditions and the checking of the negative preconditions. Both are checked in the world to see if they are fulfilled. Once checked and accepted, the action is carried out and negative effects and effectors are applied to the world.

Obtaining such a result that:

```

[20:06:25] Planning...
UnityEngine.Debug.Log(Object)
[20:06:25] Statistics:
UnityEngine.Debug.Log(Object)
[20:06:25] Total nodes: 21
UnityEngine.Debug.LogFormat(String, Object[])
[20:06:25] Open nodes: 0
UnityEngine.Debug.LogFormat(String, Object[])
[20:06:25] Closed nodes: 21
UnityEngine.Debug.LogFormat(String, Object[])
[20:06:25] PLAN FOUND!
UnityEngine.Debug.Log(Object)
[20:06:25] Go to stick Accumulated cost: 1
UnityEngine.Debug.LogFormat(String, Object[])
[20:06:25] Stick collected Accumulated cost: 2
UnityEngine.Debug.LogFormat(String, Object[])
[20:06:25] Go to stone Accumulated cost: 4
UnityEngine.Debug.LogFormat(String, Object[])
[20:06:25] Stone collected Accumulated cost: 5
UnityEngine.Debug.LogFormat(String, Object[])
[20:06:25] Go to rope Accumulated cost: 8
UnityEngine.Debug.LogFormat(String, Object[])
[20:06:25] Rope collected Accumulated cost: 9
UnityEngine.Debug.LogFormat(String, Object[])
[20:06:25] All materials collected Accumulated cost: 29
UnityEngine.Debug.LogFormat(String, Object[])


```

Statistics

The result obtained in the previous section (both in the order and in the corresponding actions) has been thanks to the use of the cost of the shares and the pre-conditions.

If we change the cost of the actions, they can vary the order of the plan or if we change the preconditions the result can change the order or even become incapacitated the plan.

For example, we are going to change the order of material collection by changing the heat of the cost of the actions:



```

mActionList.Add(
    new Action(
        Action.ActionType.ACTION_TYPE_GO_NEAR_STICKS,
        WorldState.WORLD_STATE_NONE,
        WorldState.WORLD_STATE_STICK_OWNED | WorldStat
        WorldState.WORLD_STATE_NEAR_STICK,
        WorldState.WORLD_STATE_NONE,
        1.0f, "Go to stick")
);

mActionList.Add(
    new Action(
        Action.ActionType.ACTION_TYPE_GO_NEAR_STONES,
        WorldState.WORLD_STATE_NONE,
        WorldState.WORLD_STATE_STONE_OWNED | WorldStat
        WorldState.WORLD_STATE_NEAR_STONE,
        WorldState.WORLD_STATE_NONE,
        2.0f, "Go to stone")
);

mActionList.Add(
    new Action(
        Action.ActionType.ACTION_TYPE_GO_NEAR_ROPE,
        WorldState.WORLD_STATE_NONE,
        WorldState.WORLD_STATE_ROPE_OWNED | WorldState
        WorldState.WORLD_STATE_NEAR_ROPE,
        WorldState.WORLD_STATE_NONE,
        3.0f, "Go to rope")
);

```

```

mActionList.Add(
    new Action(
        Action.ActionType.ACTION_TYPE_GO_NEAR_STICKS,
        WorldState.WORLD_STATE_NONE,
        WorldState.WORLD_STATE_STICK_OWNED | WorldStat
        WorldState.WORLD_STATE_NEAR_STICK,
        WorldState.WORLD_STATE_NONE,
        3.0f, "Go to stick")
);

mActionList.Add(
    new Action(
        Action.ActionType.ACTION_TYPE_GO_NEAR_STONES,
        WorldState.WORLD_STATE_NONE,
        WorldState.WORLD_STATE_STONE_OWNED | WorldStat
        WorldState.WORLD_STATE_NEAR_STONE,
        WorldState.WORLD_STATE_NONE,
        1.0f, "Go to stone")
);

mActionList.Add(
    new Action(
        Action.ActionType.ACTION_TYPE_GO_NEAR_ROPE,
        WorldState.WORLD_STATE_NONE,
        WorldState.WORLD_STATE_ROPE_OWNED | WorldState
        WorldState.WORLD_STATE_NEAR_ROPE,
        WorldState.WORLD_STATE_NONE,
        2.0f, "Go to rope")
);

```

The plan becomes as follows:

Time	Event	Accumulated cost
[20:06:25]	Planning...	
[20:06:25]	Statistics:	
[20:06:25]	Total nodes: 21	
[20:06:25]	Open nodes: 0	
[20:06:25]	Closed nodes: 21	
[20:06:25]	PLAN FOUND!	
[20:06:25]	Go to stick	1
[20:06:25]	Stick collected	2
[20:06:25]	Go to stone	4
[20:06:25]	Stone collected	5
[20:06:25]	Go to rope	8
[20:06:25]	Rope collected	9
[20:06:25]	All materials collected	29

Time	Event	Accumulated cost
[20:22:39]	Planning...	
[20:22:39]	Statistics:	
[20:22:39]	Total nodes: 21	
[20:22:39]	Open nodes: 0	
[20:22:39]	Closed nodes: 21	
[20:22:39]	PLAN FOUND!	
[20:22:39]	Go to stone	1
[20:22:39]	Stone collected	2
[20:22:39]	Go to rope	4
[20:22:39]	Rope collected	5
[20:22:39]	Go to stick	8
[20:22:39]	Stick collected	9
[20:22:39]	All materials collected	29

As we can see we have gone from going to stick, stone, rope, stone, rope, stick changing only the costs.

We reset costs as before and then in the case of preconditions we will make that to go to catch the stick we will have to have a rope.

```
mActionList.Add(  
    new Action(  
        Action.ActionType.ACTION_TYPE_GO_NEAR_STICKS,  
        WorldState.WORLD_STATE_ROPE_OWNED,  
        WorldState.WORLD_STATE_STICK_OWNED | WorldState.WORLD_STATE_NEAR_STONE | WorldState.WORLD_STATE_NEAR_ROPE,  
        WorldState.WORLD_STATE_NEAR_STICK,  
        WorldState.WORLD_STATE_NONE,  
        1.0f, "Go to stick")  
    );
```

The plan becomes as follows :

[20:06:25] Planning...
UnityEngine.Debug:Log(Object)

[20:06:25] Statistics:
UnityEngine.Debug:Log(Object)

[20:06:25] Total nodes: 21
UnityEngine.Debug:LogFormat(String, Object[])

[20:06:25] Open nodes: 0
UnityEngine.Debug:LogFormat(String, Object[])

[20:06:25] Closed nodes: 21
UnityEngine.Debug:LogFormat(String, Object[])

[20:06:25] PLAN FOUND!
UnityEngine.Debug:Log(Object)

[20:06:25] Go to stick Accumulated cost: 1
UnityEngine.Debug:LogFormat(String, Object[])

[20:06:25] Stick collected Accumulated cost: 2
UnityEngine.Debug:LogFormat(String, Object[])

[20:06:25] Go to stone Accumulated cost: 4
UnityEngine.Debug:LogFormat(String, Object[])

[20:06:25] Stone collected Accumulated cost: 5
UnityEngine.Debug:LogFormat(String, Object[])

[20:06:25] Go to rope Accumulated cost: 8
UnityEngine.Debug:LogFormat(String, Object[])

[20:06:25] Rope collected Accumulated cost: 9
UnityEngine.Debug:LogFormat(String, Object[])

[20:06:25] All materials collected Accumulated cost: 29
UnityEngine.Debug:LogFormat(String, Object[])

[20:33:27] Planning...
UnityEngine.Debug:Log(Object)

[20:33:27] Statistics:
UnityEngine.Debug:Log(Object)

[20:33:27] Total nodes: 14
UnityEngine.Debug:LogFormat(String, Object[])

[20:33:27] Open nodes: 0
UnityEngine.Debug:LogFormat(String, Object[])

[20:33:27] Closed nodes: 14
UnityEngine.Debug:LogFormat(String, Object[])

[20:33:27] PLAN FOUND!
UnityEngine.Debug:Log(Object)

[20:33:27] Go to rope Accumulated cost: 3
UnityEngine.Debug:LogFormat(String, Object[])

[20:33:27] Rope collected Accumulated cost: 4
UnityEngine.Debug:LogFormat(String, Object[])

[20:33:27] Go to stick Accumulated cost: 5
UnityEngine.Debug:LogFormat(String, Object[])

[20:33:27] Stick collected Accumulated cost: 6
UnityEngine.Debug:LogFormat(String, Object[])

[20:33:27] Go to stone Accumulated cost: 8
UnityEngine.Debug:LogFormat(String, Object[])

[20:33:27] Stone collected Accumulated cost: 9
UnityEngine.Debug:LogFormat(String, Object[])

[20:33:27] All materials collected Accumulated cost: 29
UnityEngine.Debug:LogFormat(String, Object[])

We can see that the plan has varied and before we go for the stick has gone for the rope as we have clarified in the preconditions.

In the case of negative preconditions, it is only necessary to see a plan that is hard to execute as the case of going for a material that we already possess because you can only go for a material when you do not own.

```
FindPlan(World.WorldState.WORLD_STATE_STONE_OWNED, World.WorldState.WORLD_STATE_NEAR_STONE);
```

In this example we will try to go from the state “have a stone” to the state “go for a stone”.

```
mActionList.Add(
    new Action(
        Action.ActionType.ACTION_TYPE_GO_NEAR_STONES,
        WorldState.WORLD_STATE_NONE,
        WorldState.WORLD_STATE_STONE_OWNED | WorldState.WORLD_STATE_NEAR_STICK | WorldState.WORLD_STATE_NEAR_ROPE,
        WorldState.WORLD_STATE_NEAR_STONE,
        WorldState.WORLD_STATE_NONE,
        2.0f, "Go to stone")
);
```

```

[20:42:05] Planning...
UnityEngine.Debug:Log(Object)
[20:42:05] Statistics:
UnityEngine.Debug:Log(Object)
[20:42:05] Total nodes: 17
UnityEngine.Debug:LogFormat(String, Object[])
[20:42:05] Open nodes: 4
UnityEngine.Debug:LogFormat(String, Object[])
[20:42:05] Closed nodes: 13
UnityEngine.Debug:LogFormat(String, Object[])
[20:42:05] PLAN FOUND!
UnityEngine.Debug:Log(Object)
[20:42:05] Go to stick Accumulated cost: 1
UnityEngine.Debug:LogFormat(String, Object[])
[20:42:05] Stick collected Accumulated cost: 2
UnityEngine.Debug:LogFormat(String, Object[])
[20:42:05] Go to rope Accumulated cost: 5
UnityEngine.Debug:LogFormat(String, Object[])
[20:42:05] Rope collected Accumulated cost: 6
UnityEngine.Debug:LogFormat(String, Object[])
[20:42:05] All materials collected Accumulated cost: 26
UnityEngine.Debug:LogFormat(String, Object[])
[20:42:05] Go to crafting table Accumulated cost: 46
UnityEngine.Debug:LogFormat(String, Object[])
[20:42:05] Axe crafted Accumulated cost: 146
UnityEngine.Debug:LogFormat(String, Object[])
[20:42:05] Go to stone Accumulated cost: 148
UnityEngine.Debug:LogFormat(String, Object[])

```

In this case as we start from the possession of a material, we must get rid of it in order to go to look for it again, for it the planner decides to build an axe so that it can removes the stone so that it can go to search again.

```

[20:49:51] PLAN FOUND!
UnityEngine.Debug:Log(Object)
[20:49:51] Go to stick Accumulated cost: 1
UnityEngine.Debug:LogFormat(String, Object[])
[20:49:51] Stick collected Accumulated cost: 2
UnityEngine.Debug:LogFormat(String, Object[])
[20:49:51] Go to stone Accumulated cost: 4
UnityEngine.Debug:LogFormat(String, Object[])
[20:49:51] Stone collected Accumulated cost: 5
UnityEngine.Debug:LogFormat(String, Object[])
[20:49:51] Go to rope Accumulated cost: 8
UnityEngine.Debug:LogFormat(String, Object[])
[20:49:51] Rope collected Accumulated cost: 9
UnityEngine.Debug:LogFormat(String, Object[])
[20:49:51] All materials collected Accumulated cost: 29
UnityEngine.Debug:LogFormat(String, Object[])
[20:49:51] Go to crafting table Accumulated cost: 49
UnityEngine.Debug:LogFormat(String, Object[])
[20:49:51] Axe crafted Accumulated cost: 149
UnityEngine.Debug:LogFormat(String, Object[])
[20:49:51] Go to tree Accumulated cost: 169
UnityEngine.Debug:LogFormat(String, Object[])
[20:49:51] Cut tree Accumulated cost: 369
UnityEngine.Debug:LogFormat(String, Object[])

```

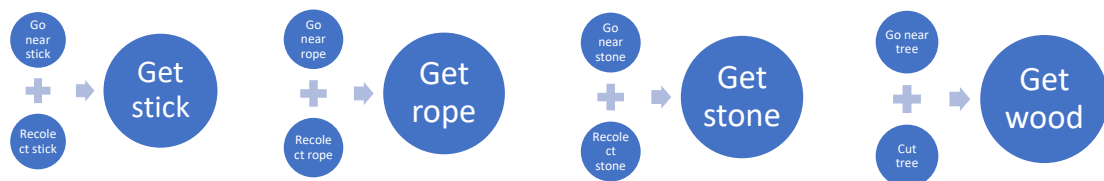
Finally, we will show the longest action that the planner possesses which is to start from nowhere to cut a tree.

BEHAVIOR TREE

The final design of our behavior tree

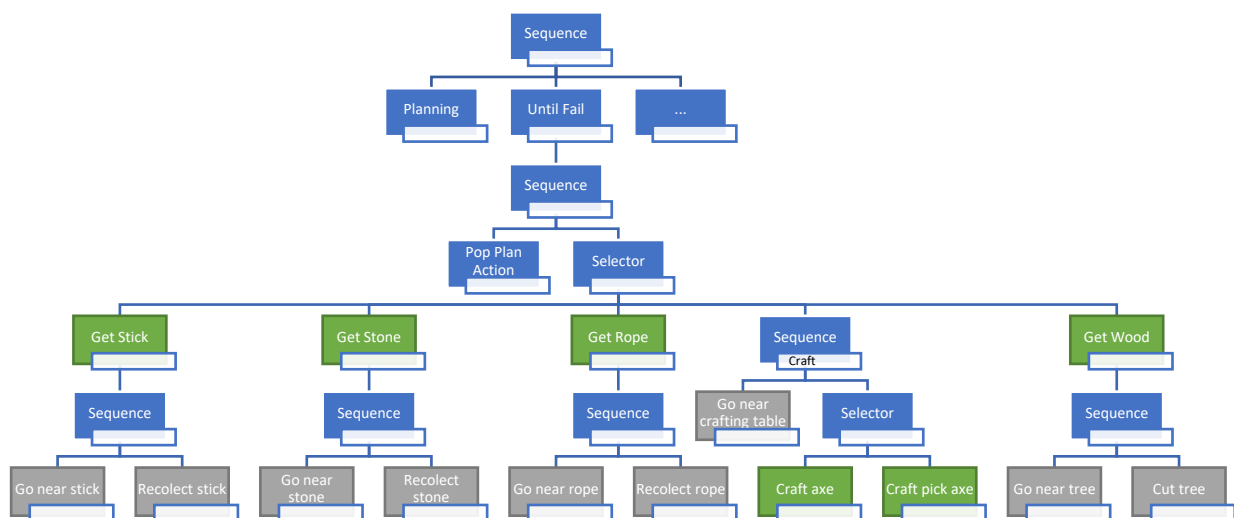
To reach the final design of the behavior tree we needed to change the actions of our already created planner in order to make everything work together. To reach that goal we needed to create higher level actions modifying the established ones to create a more approachable take of the behavior tree.

Our planner had a lot of actions that were useful when we only had the planner but now that we wanted to implement the behavior tree they were redundant or simply useless. In order to make our behavior tree work, we needed to transform the actions of our planner, joining some of them and then creating new actions for the higher levels of our system.



The final design of our behavior tree with everything updated looks like this:

- Planner action / High level action
- Behavior tree action / Low level action



List of actions



A couple of examples of plan executions.

For our longest execution plan the results are the following:

```
[18:00:42] Planning...
[18:00:42] Statistics:
[18:00:42] Total nodes: 24
[18:00:42] Open nodes: 7
[18:00:42] Closed nodes: 17
[18:00:42] PLAN FOUND!
[18:00:42] Get stick Accumulated cost: 1
[18:00:42] Get stone Accumulated cost: 3
[18:00:42] Get rope Accumulated cost: 6
[18:00:42] Axe crafted Accumulated cost: 106
[18:00:42] Get wood Accumulated cost: 126
[18:00:42] Planned in 5 steps
[18:00:44] Go near stick
[18:00:45] Recollect stick
[18:00:46] Go near stone
[18:00:47] Recollect stone
[18:00:48] Go near rope
[18:00:49] Recollect rope
[18:00:50] Go to crafting table
[18:00:51] Axe crafted
[18:00:52] Go to tree
[18:00:53] Cut tree
```


As we can see the program play a correct order of actions in order to reach our goal. Our world state at the beginning is none and our goal world state is to have wood. For that the planner finds the action that will reach that goal state and the behavior tree will do the rest. In our design we do not have selector so the execution will always be the same. If we had a selector at some point of the execution it will iterate trough the available actions until it gets the result that we need to reach our goal. The number of nodes and accumulated cost is correctly calculated and we will see how those numbers vary when changing goal states.

```
! [17:55:36] Planning...
! [17:55:36] Statistics:
! [17:55:36] Total nodes: 9
! [17:55:36] Open nodes: 0
! [17:55:36] Closed nodes: 9
! [17:55:36] PLAN FOUND!
! [17:55:36] Get stick Accumulated cost: 1
! [17:55:36] Get stone Accumulated cost: 3
! [17:55:36] Get rope Accumulated cost: 6
! [17:55:36] Axe crafted Accumulated cost: 106
! [17:55:36] Planned in 4 steps
! [17:55:37] Go near stick
! [17:55:38] Recolect stick
! [17:55:39] Go near stone
! [17:55:40] Recolect stone
! [17:55:41] Go near rope
! [17:55:42] Recolect rope
! [17:55:43] Go to crafting table
! [17:55:44] Axe crafted
```

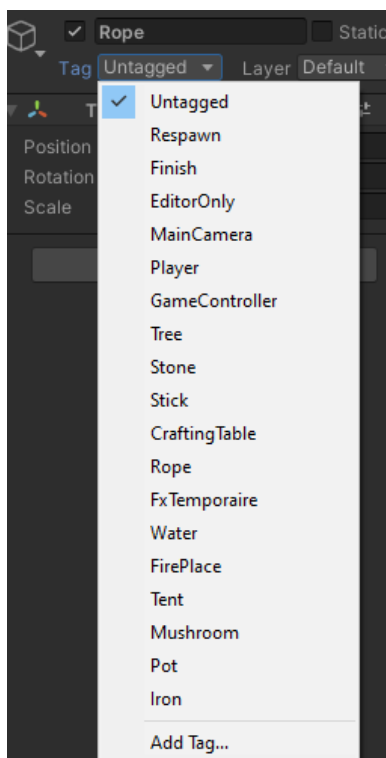
In this second execution we just want the axe crafted. As we can see the number of nodes reduced, this is obviously due to the reduced number of steps that we need to reach our goal compared to the first example. The total cost is also calculated correctly in this execution of the program.

FINAL PROJECT

What kind of world we have created

Our world tries to simulate the journey of a survivor in the forest. For that reason, we needed to create a lot of new actions in our Planner and also mix the Pathfinding, Planner and behavior Trees all together so the simulation can work correctly.

Before talking about the changes implemented to the pathfinder, the planner, and the behavior tree we are going to explain the changes that we did to our world so it could work with everything mixed. In our world we have trees, sticks, rope, and a lot of other materials that can be obtained/modified by the player. We get all this game objects by sorting them in our world with different tags and then creating arrays with only the objects that has a certain type of tag.



```
// referencias
public WorldState()
{
    trees = GameObject.FindGameObjectsWithTag("Tree");
    rope = GameObject.FindGameObjectsWithTag("Rope");
    sticks = GameObject.FindGameObjectsWithTag("Stick");
    stones = GameObject.FindGameObjectsWithTag("Stone");
    iron = GameObject.FindGameObjectsWithTag("Iron");
    mushrooms = GameObject.FindGameObjectsWithTag("Mushroom");
    water = GameObject.FindGameObjectsWithTag("Water");
    craftingTables = GameObject.FindGameObjectsWithTag("CraftingTable");
    firePlace = GameObject.FindGameObjectsWithTag("FirePlace");
    tent = GameObject.FindGameObjectsWithTag("Tent");
    pot = GameObject.FindGameObjectsWithTag("Pot");
    player = GameObject.FindGameObjectsWithTag("Player");

    nTrees = trees.Length;
}
```

To know the current state of our world we will save our world state in a int World Mask. This World Mask will be updated every time that our player does an action (or a set of actions) that changes the way that our player should behave.

```

99+ referencias
public enum WorldMask
{
    WORLD_STATE_NONE = 0,
    WORLD_STATE_AXE_OWNED = 1,
    WORLD_STATE_PICK_AXE_OWNED = 2,
    WORLD_STATE_NEAR_STICK = 4,
    WORLD_STATE_NEAR_STONE = 8,
    WORLD_STATE_NEAR_ROPE = 16,
    WORLD_STATE_NEAR_CRAFTING_TABLE = 32,
    WORLD_STATE_NEAR_TREE = 64,
    WORLD_STATE_FIREPLACE_OWNED = 128,
    WORLD_STATE_NEAR_FIREPLACE = 256,
    WORLD_STATE_NEAR_TENT = 512,
    WORLD_STATE_TENT_OWNED = 1024,
    WORLD_STATE_POT_OWNED = 2048,
    WORLD_STATE_HUNGER = 4096,
    WORLD_STATE_NEAR_WATER = 8192,
    WORLD_STATE_NEAR_MUSHROOM = 16384,
    WORLD_STATE_FISHING_ROD_OWNED = 32768,
    WORLD_STATE_NEAR_IRON = 65536,
    WORLD_STATE_STICK_OWNED = 131072,
    WORLD_STATE_STONE_OWNED = 262144,
    WORLD_STATE_ROPE_OWNED = 524288,
    WORLD_STATE_WOOD_OWNED = 1048576,
    WORLD_STATE_IRON_OWNED = 2097152,
    WORLD_STATE_FISH_OWNED = 4194304,
    WORLD_STATE_MUSHROOM_OWNED = 8388608
}

```

When we started with the planner our mask only contained the states needed to cut a tree but we wanted to add a lot more of complexity to our game so we created a lot of world states that can make us do exactly that.

Our WorldState class not only contains a mask but also a series of variables such as arrays of objects (already mentioned before) and int variables to store the number of objects acquired that make more complex the world by being able to count how many objects we own in the current state and generating more complex plans.

This mask is updated in the behavior tree and because we have a lot of different world states, we needed to define with complete security the positive preconditions, negative preconditions, positive effects and effects so we don't get stuck in an infinite loop or end up in a complete different state that we anticipated.

```

if (mPlan[mCurrentAction].mAction.mActionType == ActionPlanning.ActionType.ACTION_TYPE_GET_STONES)
{
    WorldState.WorldMask preconditions;
    WorldState.WorldMask negativePreconditions;
    WorldState.WorldMask effects;
    WorldState.WorldMask negativeEffects;
    if (complexWorld)
    {
        preconditions = WorldState.WorldMask.WORLD_STATE_NEAR_STONE;
        negativePreconditions = WorldState.WorldMask.WORLD_STATE_NONE;
        effects = WorldState.WorldMask.WORLD_STATE_NONE;
        negativeEffects = WorldState.WorldMask.WORLD_STATE_NEAR_STONE;
    }
    else
    {
        preconditions = WorldState.WorldMask.WORLD_STATE_NEAR_STONE;
        negativePreconditions = WorldState.WorldMask.WORLD_STATE_NONE;
        effects = WorldState.WorldMask.WORLD_STATE_STONE_OWNED;
        negativeEffects = WorldState.WorldMask.WORLD_STATE_NEAR_STONE;
    }
}

```

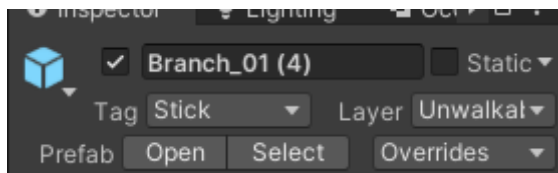
That last picture is an example of a behavior tree that changes the world state.

Pathfinder specs

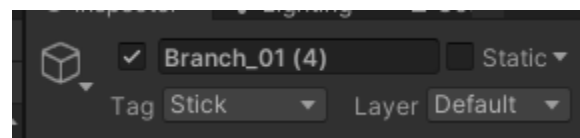
In the pathfinder the main change is that instead of doing all the process in the update we created a function so we can update correctly and whenever we want where our next goal is going to be.

```
public void DoPathfinding(Transform origin, Transform target)
{
    mSeeker = origin;
    mTarget = target;
    if (PathInvalid())
    {
        // Remove old path
        if (Grid.path != null)
        {
            Grid.path.Clear();
        }
        // Start calculating path again
        Iterations = -1;
        if (TimeBetweenSteps == 0.0f)
        {
            Iterations = -1;
        }
    }
    FindPath(mSeeker.position, mTarget.position, Iterations);
}
else
{
    // Path found?
    if (Iterations >= 0)
    {
        // One or more iterations?
        if (TimeBetweenSteps == 0.0f)
        {
            // One iteration, look until path is found
            Iterations = -1;
            FindPath(mSeeker.position, mTarget.position, Iterations);
        }
        else if (Time.time > LastStepTime + TimeBetweenSteps)
        {
            // Iterate increasing depth every time step
            LastStepTime = Time.time;
            Iterations++;
            FindPath(mSeeker.position, mTarget.position, Iterations);
        }
    }
}
```

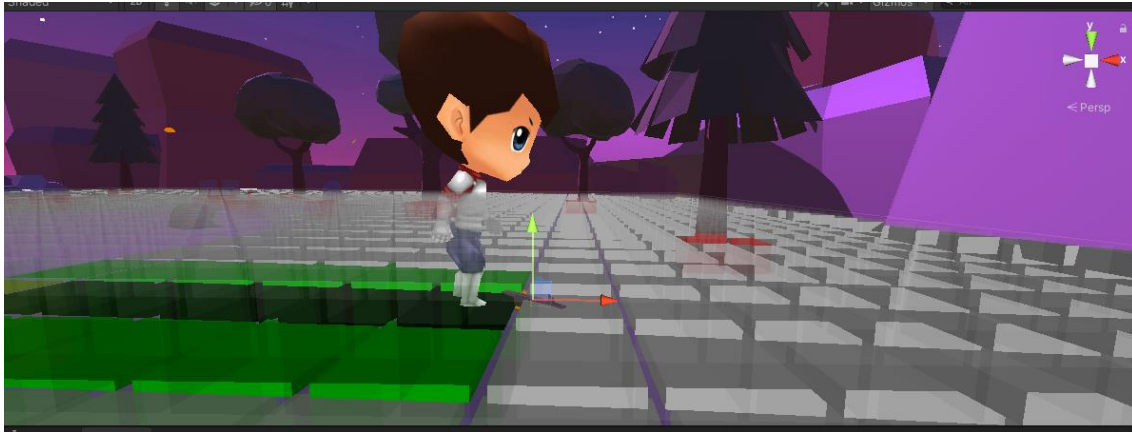
We access this function trough the behavior tree in every action that requires the player to move to a certain objective (GoNearRope, GoNearStone, GoNearStick, GoNearCraftingTable etc). We also needed to update the grid every time we changed the layer of the object we wanted to go because we wanted to walk over the item that we wanted to collect and also that when collected we could go trough the part where the item was.



Nearest stick before execution



Nearest stick during execution



The grid is white because the stick that we are going to grab is walkable but the tree is red because we can't pass through and we don't need to do anything with it now.

Planner specs

Now, with the planner we needed to redo almost everything. When the planner was created, we only had the number of actions needed to cut a tree, so we created a lot of new actions, at the end the actions that the planner had were the following ones (the missing actions are handled by the behavior tree).

Actions to obtain material:



Actions of building tools and objects:



Cooking actions:



When trying to plan the actions for a complex world (complex world being one in which we needed more than one material to craft something) our planner planned really slowly due to the huge number of nodes that needed to be opened in order to find the best set of actions to do it. We tried to improve the performance of our planner by creating a set of heuristic rules but it wasn't good enough.

That's why we needed a backwards planner instead of a forward planner, with the backwards planner a less (considerably less) number of nodes are opened and that was the solution of our problem. For that we duplicated the planner functions and we modified them so it follows a new set of rules to go from the goal state to the none state creating the in between actions needed to join all together.

To get the backwards planner to work correctly we needed to change the functions that defined how the forward planner worked and create some other ones to help us get the desired behavior. The first one that we created is `CompareState()`, due to the fact that we cant compare complex classes we were forced to create this function that simply makes sure that

the world mask and the item values are the same in the hypothetical world state and the actual world state.

```
public bool compareState(WorldState n)
{
    if(n.stick == n.nstick && n.stone == n.nstone && n.rope == n.nrope && n.wood == n.nwood && n.mushrooms == n.nmushrooms && n.fish == n.nfish && n.iron == n.niron && worldMask == n.worldMask)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

With that now we were able to modify the find plan function, with the complex world activated we need to take into account not only the world mask but also all the elements that are in our world state. The last change needed for this function to work is that now our current start node will be the target world and our current target node will be our start state.

```
WorldState wState = new WorldState();
wState.worldMask = (wState.worldMask | targetWorldState);
CurrentStartNode = new NodePlanning(wState, null);

wState = new WorldState();
wState.worldMask = (wState.worldMask | startWorldState);
CurrentTargetNode = new NodePlanning(wState, null);
```

Now after the find plan function, we needed to modify the retrace plan function, for this to work with backwards planning we simply comment the line where we reverse our plan because we don't need our plan reversed this time.

```
void RetracePlanBackwards(NodePlanning startNode, NodePlanning endNode)
{
    List<NodePlanning> plan = new List<NodePlanning>();

    NodePlanning currentNode = endNode;

    while (currentNode != startNode)
    {
        plan.Add(currentNode);
        currentNode = currentNode.mParent;
    }
    //plan.Reverse();
    World.plan = plan;
}
```

The last thing needed to get our backwards planner was the get neighbor function, this time we didn't need to check if the positive and negative preconditions of our world were the same as the action ones, this time we just needed to check every action with the world that we are in and if the effect that causes that action are the ones that causes our world state, we create an alternate world where the preconditions will be applied, in other words, we create a new world where the action can be done.

Here we have 2 examples about what we just explained, the first one is for the get stick action and the second one is for the craft axe action.

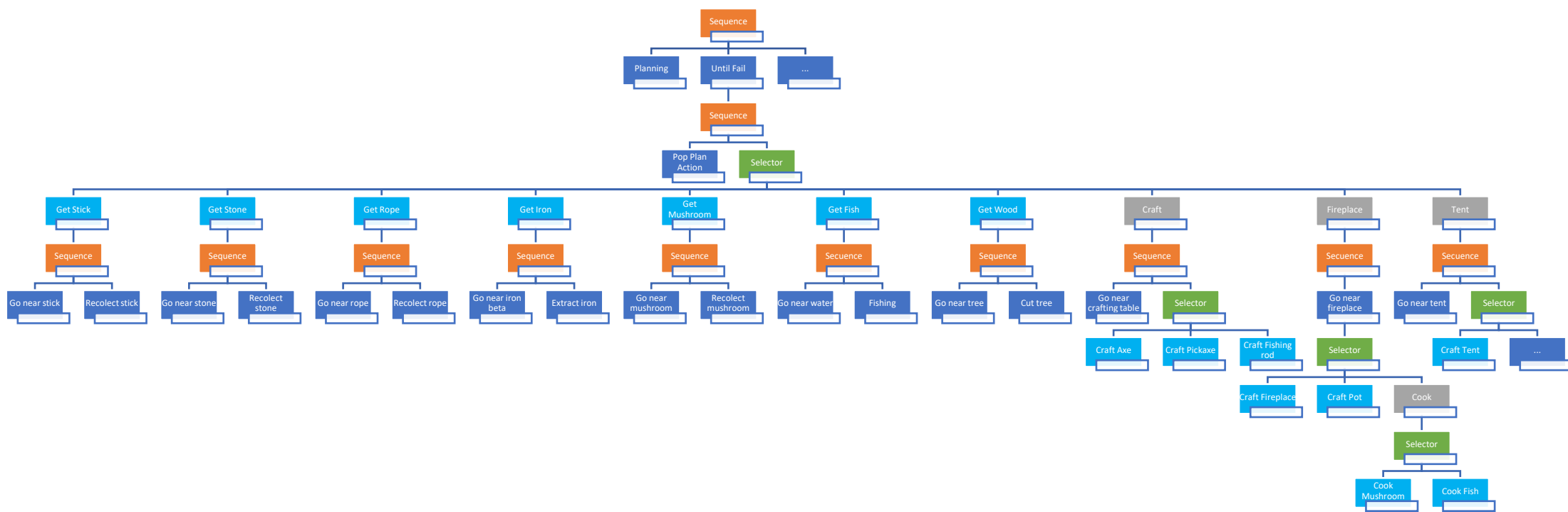
```
case ActionPlanning.ActionType.ACTION_TYPE_GET_STICK:
    if (origin.mWorldState.nStick != node.mWorldState.nStick)
    {
        wState.copyState(node.mWorldState, ((node.mWorldState.worldMask | action.mPreconditions) & ~(action.mNegativePreconditions)));
        wState.nStick--;
        // Apply action and effects
        newNodePlanning = new NodePlanning(wState, action);
        neighbours.Add(newNodePlanning);
    }
    break;
```

As we explained before, instead of adding a stick we are subtracting it so it can work.

```
case ActionPlanning.ActionType.ACTION_TYPE_CRAFT_AXE:
    if ((node.mWorldState.worldMask & action.mEffects) != 0)
    {
        wState.copyState(node.mWorldState, ((node.mWorldState.worldMask | action.mPreconditions) & ~(action.mNegativePreconditions)));
        wState.nStick += 1;
        wState.nStone += 1;
        wState.nRope += 1;
        // Apply action and effects
        newNodePlanning = new NodePlanning(wState, action);
        neighbours.Add(newNodePlanning);
    }
    break;
```

And here in the case of the crafting axe action we are adding the world state one of each material instead of subtracting it.

For the behavior tree we simply needed to create our main scheme of actions that we had already planned, the final tree ended up looking like this:



Here we also have to take into account if we are using a complex world or a more simplistic world because our preconditions, negative preconditions, effects and negative effects can be different depending on that decision. In case of the complex world we use the Backwards planner to better manage the number of open nodes, and in the simple world we use the Forward planner.

```
if (complexWorld)
{
    preconditions = WorldState.WorldMask.WORLD_STATE_NEAR_CRAFTING_TABLE;
    negativePreconditions = WorldState.WorldMask.WORLD_STATE_AXE_OWNED;
    effects = WorldState.WorldMask.WORLD_STATE_AXE_OWNED;
    negativeEffects = WorldState.WorldMask.WORLD_STATE_NEAR_CRAFTING_TABLE;
}
else
{
    preconditions = WorldState.WorldMask.WORLD_STATE_NEAR_CRAFTING_TABLE | WorldState.WorldMask.WORLD_STATE_STICK_OWNED | WorldState.WorldMask.WORLD_STATE_STONE_OWNED | WorldState.WorldMask.WORLD_STATE_ROPE_OWNED;
    negativePreconditions = WorldState.WorldMask.WORLD_STATE_AXE_OWNED;
    effects = WorldState.WorldMask.WORLD_STATE_AXE_OWNED;
    negativeEffects = WorldState.WorldMask.WORLD_STATE_NEAR_CRAFTING_TABLE | WorldState.WorldMask.WORLD_STATE_STICK_OWNED | WorldState.WorldMask.WORLD_STATE_STONE_OWNED | WorldState.WorldMask.WORLD_STATE_ROPE_OWNED;
}
```

In the last picture we can clearly see that for crafting an axe the preconditions and negative effects in the complex world are completely different for the one that isn't.

Here is also where we add the gameplay details like activate particles animations etc.

```
// If execution succeeded return "success". Otherwise return "failed".
if (timer >= timeToCraft)
{
    Debug.Log("Axe crafted");
    // Apply action and effects
    worldPlanning.GetWorld().mWorldState.worldMask = (worldPlanning.GetWorld().mWorldState.worldMask | effects) & ~(negativeEffects);
    worldPlanning.GetWorld().mWorldState.nStick -= 1;
    worldPlanning.GetWorld().mWorldState.nRope -= 1;
    worldPlanning.GetWorld().mWorldState.nStone -= 1;
    axe.SetActive(true);
    pickaxe.SetActive(false);
    fishingRod.SetActive(false);
    lightBulb.SetActive(false);
    Instantiate(collectMaterialsParticles, lightBulb.transform.position, lightBulb.transform.rotation);
    playerAnimator.SetBool("Craft", false);
    return Action.Result.SUCCESS;
}
else
{
    // Action in progress
    if (!lightBulb.activeSelf)
    {
        Instantiate(collectMaterialsParticles, lightBulb.transform.position, lightBulb.transform.rotation);
    }
    lightBulb.SetActive(true);
    playerAnimator.SetBool("Craft", true);
    return Action.Result.PROGRESS;
}
```

As we could see, depending if we completed the crafting or not we are applying particles and effects.