Goal State:
Dictionary<T, W> (By default dictionary of undefined types
In this case, Dictionary<string, object>
The following state is the desired state with the conditions we want to fulfill!

Conditions	Values
“collectedResourceAxe”	True

Sensors Update: Sensors handle all the updates to the memory state.
Workstation Sensor:

Get the memory state of the agent and set the following conditions:
(Note the blue text represents the type of a variable)

Conditions	Values
“seeWorkstation”	True or False (Ture)
“nearestWorkstation”	Workstation nearestStation
“nearestWorkstationPosition”	Vector3 nearestStation.transform.position

BankSensor:

Get the memory state of the agent and set the following conditions:

Conditions	Values
“seeBank”	True or False
“nearestBank”	Bank nearestBank
“nearestBankPosition”	Vector3 nearestBank.transform.position

ResourceBag Sensor:

Resource Bag is just a dictionary of <string, float> with the resource name and its quantity. Get the memory state of the agent, and set what resources are in the Resource Bag of the agent.

When the agent first spawns, it doesn’t have any resources in its bag, so the conditions “hasResource” and true or false doesn’t get updated to the memory state, until actions are performed that adds resources to the agent’s resource bag.

Conditions	Values
“hasResource” + resourceName	True or False (The resource has a capacity more than zero?)

“hasResourceOre” “hasResourceWood” “hasResourceTree” “hasResourceAxe”	…

MultipleResourcesSensor and Resource Sensor:
MultipleResourcesSensor handles the iteration of going through different ResourceManagers controlled by a MultipleResourcesManager class. Meanwhile, the ResourceSensor handles getting the position of each IResource type object handled by theResource Manager.

A Resource Manager shares the same name as the Resources they are holding, for example, the Tree ResourceManager handles Tree Resource objects, and each Tree Resource object has a string name “Tree” and a float capacity, in this case for tree 2.

Same for the Ore Resource Manager, has a list of Ore Resources and each one has a capacity of 5.

The MultipleResourcesManager handles those Resource Managers of Ore and Tree and MultipleResourcesSensor goes through them.

Get the memory state of the agent and set the following conditions:

Conditions	Value
“see” + resourceManager.GetResourceName() (could be either “seeTree” or “seeOre”	True or False (Resource Manager has at least one of that resource).
“nearest” + resourceManager.GetResourceName() (“nearestTree” or “nearestOre”)	IResource nearestResource (nearestResource could have a name of Tree or Ore)
“nearestTreePosition” or “nearestOrePosition”	Vector3 nearestResource.GetTransoform().position
…	…

Actions:
An action has a name, preconditions and effects which are represented as states, and a cost. Other helpful instances are action delegates to execute when an action finishes or fails, references the previous and next action, a reference to agent, and a flag to interrupt the action

when possible which could happen when calculating a new goal when another goal has higher priority than the current.

Upon asking when the game starts, the action will instantiate its effects and preconditions as empty states. Actions can be enabled when asking if the action IsActive() and disabled when called to Exit()/ Actions can be interrupted by AskForInterruption() and all actions by default are interruptible by calling IsInterruptible().

The public parameters of actions are the cost and name, so methods can be applied to the cost to dynamically change the cost in regards to the game. Lastly by default [Important] the Run() method of actions initializes local variables to the method:

- interruptWhenPossible = false;
- enabled = true; // set the action to active
- doneCallback = done; // method to push the next action in planner
- failCallback = fail; // method to tell agent to CalculateNewGoal()
- this.settings = settings;// personal dictionary for the action
- previousAction = previous;
- nextAction = next;

The defined actions must override the Run() method to actually run the action, also, define the preconditions and effects states. Note that some actions have preconditions and effects that are dynamic in which they change based on the next action's preconditions and effects

There are four defined actions:

- AddResourceToBankAction:
  - GetSettings(stackData):
    - If "collectedResource" condition is within goal state of passed stack data, then set <"ResourceName", resourceName> to the settings.
  - Effects(stackData):
    - <"collectedResource" + resourceName, true>
    - <"collectedResourceAxe", true>
  - Preconditions(stackData):
    - <"isAtPosition", Vector3 bankPosition>
    - < "hasResource" + resourceName, true>
    - < "hasResourceAxe", true>
  - Run():
    - Run the base Run() method
    - Get the bank from the "nearestBank" condition in the agent's memory state. Afterwards check if the bank is present and if the agent's resource bag has the resource to add. Add the resource to the bank and remove the resource from the bag.
- CraftRecipeAction:

[Important] The CraftRecipeAction is used twice by the agent to craft two recipes: CraftWoodAction & CraftAxeAction.
With the given recipe, check the needed resources in the recipe, and set them to preconditions. [Important] DO NOT USE FALSE PRECONDITIONS. Set to the effects that agent will have the resulting resource from crafting the recipe.

- Effects:
  - <“hasResource” + recipe.GetCraftedResource(), true”>
  - <“hasResourceAxe”, true>
  - Or <“hasResourceWood”, true>
- Preconditions(stackData):
  - <“hasResource” + pairKey, true>
    For Crafting an Axe:
  - <“hasResourceWood”, true>
  - <“hasResourceOre”, true>
    For Crafting Wood:
  - <“hasResourceTree”, true>
    Need to be at a WorkStation
  - <isAtPosition”, Vector3 nearestWorkStationPosition>
- Run:
  - Get the nearest workstation, and craft the recipe. If possible to craft, then call done, otherwise call fail.

- GatherResourceAction: (This action can be called multiple times in the planner to gather different resources. One Gather Action per resource).
  Checks what resource is needed in the goal state of a given stackData by searching for a condition that starts with “hasResource”, then it checks for the nearest location of that desired resource and sets to preconditions that the agent must be at that position in order to gather it. The effect is that the agent has that resource.
  - GetSettings(stackData):
    - Set to the settings the resource position of the nearest desired resource from the goal state of the given stack data: < “ResourcePosition”, Vector3 nearestNeededResourcePostion>
    - Set the wanted resource in settings:
      - < “Resource”, Tree>
      - <“Resource”, Ore>
  - CheckProceduralConditions(stackData):
    - [Important] By default this method returns true, but here we make sure that the agent has a resource bag to gather resources!
  - Effects:
    - <“hasResource” + newNeededResourceName, true”>
    - <“hasResourceTree”, true>

        - <“hasResourceOre”, true>
    - Preconditions(stackData):
        Need to be at the Needed Resource Position to gather
        - <isAtPosition”, Vector3 nearestNeededResourcePosition>
    - Run:
        - Get from the action’s personal settings the resource position of the nearest desired resource.
        - Get from the action’s personal settings the resource name.
        - If there is no desired resource or the capacity is less than what can be gathered, then we failed to gather, otherwise set a cool down.
    - Update:
        - If there is no desired resource or the capacity is less than what can be gathered, then we failed to gather, otherwise set a cool down.
        - If the gather cool down threshold is surpassed, remove 1 capacity of the resource, then add the resource to the agent’s bag, then call finished.
- GenericGoToAction:

    This references the defined state SmsGoTo in the Finite State Machine. In Awake(), effects, preconditions and settings are instantiated. Call to set the default effects: <“isAtPosition”, default(Vector3)>.
    - GetSettings(stackData):
        - Set to settings the <”ObjectivePosition”, GetWantedPositionFromState(stackData.goalState)
        - Return the settings
    - Effects(stackData):
        - Default: <“isAtPosition”, default(Vector3)>
        - <“isAtPosition”, goalWantedPosition> [Note: The goalWantedPosition is the retrieved position from a condition “isAtPosition” retrieved from the goal state of a given stackData]
    - Preconditions:
        - [Important] No preconditions are defined!
    - Run:
        - Look through the action’s personal settings, and search for the condition “ObjectivePosition”.
        - Get the Vector3 position of “ObjectivePosition” and pass it to the smsGoto.GoTo(), including the done and fail methods. The FSM’s SmsGoTo state will then handle physically moving the agent in the Game Scene.

Important things to consider:

- Actions can be defined to blacklist the current goal (set the OnFailureBlacklist flag to the agent to true)

Planner:

Planner performs the calculation for the goal of the highest priority, then it calculates a plan for that goal of highest priority. Goals that aren't blacklisted and are possible may be considered for calculator, and if the resulting goal doesn't have a viable plan, then the next queued highest priority goal will undergo the planning process.

The number of nodes we allow the planner to expand upon, and the number of iterations to check expanded nodes can be edited within the ReGoapPlannerSettings. This way for future projects, developers can place hard limits on how far the planning process can expand from nodes to nodes after applying actions.

At the start of the planning process, the goal state is converted to a node. The goal state node has no parent, no action, and no personal settings to give to an action. Every node has a goal state, so the root has a goal state which will be a copy of itself, and this state is compared to the current memory state. The differences between the memory state and the goal state are recorded in a state called goalMergedWithWorld which records the differences.

After instantiating the root node, the planner runs the AStar search algorithm with the node, a copy of the goal state, the number of max iterations and the flag to exit the plan early from the planner settings. Inside AStar, there are four key private global instances:

- Frontier: A priority queue that places the lowest cost node first. This queue holds nodes that have yet to be explored which means expanded as a parent to children nodes <INode<T>, float cost>
- stateToNode: A dictionary of nodes that have been iterated through <state, INode<T>>
- explored: A dictionary of nodes that have been expanded on as parents to produce child nodes <state, INode<T>>
- createdNodes: A list of created nodes

When AStar runs, it clears the frontier queue, stateToNode dictionary, the explored dictionary, and the createdNodes list; also, the root goal node is added to the createdNodes. Next the frontier appends the root node to its queue and its cost as a value.

While the frontier is not empty and the max iterations and max size is not exceeded, then dequeue the lowest cost node from frontier and conduct a base check if the node has accomplished having no differences with the current memory state. Since this is the root the base check fails. The root node is then added to the explored dictionary as it is about to be expanded upon.

When expanding a node a reference to the agent is retrieved from the planner, as well as the current action set of the agent is retrieved. An object called stackData is created of type GoapActionStackData. The action set is iterated, and for each action the stackData is passed, so the action references the agent and the personal settings of the action is called to instantiate.

With the settings of the action instantiated, depending on the Action there can be more than one defined setting ReGoapState, but for this example there is only one for every action. For each setting have the stackData.settings equal a copy of it, then get all preconditions and effects from the action.

With the effects and preconditions, if the effect has any condition and value pair that matches one in goal && there are no preconditions conflicting with the goal && there is no effect conflicting with the goal, then create a new goal variable equal to the current node's goal. Instantiate a new node and pass the planner, newGoal, this node, the action, and the action's personal settings as parameters. The resulting node will be added to an expanded node list.

Starting with the root node, an expanded child created will have the root as a parent, the action will be one with effects that fulfill some conditions in the goal state of the parent with no conflicting preconditions or effects, and the action's personal settings. Since the parent isn't null this time, the state of this node will clone the parent's state (remember that root copied the current memory state), and g is updated to equal the parent's g cost which is the total sum of node g's so far. Next action will reference the parent node's action, since we're planning backwards, the nodes' closer to the root are the next future actions in the plan. Action this time is not null, so with the passed newGoal we instantiate it to this node's goal instance. Next create a stackData, get the preconditions and effects of the action, update g with the addition of this action's cost. Next add the effects of the action to the node's state. Inside the node's goal state remove all conditions fulfilled by the effects, next add the preconditions. The preconditions become the next unfulfilled conditions to satisfy.

Calculate the heuristic of the goal state, then calculate the new cost for this node. Get the goalMergedWithWorld result - the dictionary of still conflicting conditions from the goal state compared to the current memory state.

Once all childs are made into the expanded list, iterate through the list. For each child in the expandList, base check if the child completes the goal (no remaining differences of its goal state to the current memory state), otherwise, get the cost and the child's state. With the state, we check if the state has been already explored (meaning it has been expanded upon), and if so continue to check the next child in the expandedList.

The stateToNode is a dictionary of types <ReGoapState, INode<T>>, and if the child's state matches any state in the stateToNode dictionary, then we check for the following:

- If the child node has a lesser cost while having a state similar to another similar node in stateToNode, then remove that similar node from the frontier as it is not worth exploring.
- Otherwise, the child has a greater cost to an already similar node, and the same may be so for its other siblings, so we don't bother exploring the parent Node any longer [Important: Even I'm not sure entirely about this as to why]

At the end, the child node is deemed worth considering to explore and is added to the frontier and the stateToNode dictionary. This process of expansion from nodes in frontier continues, until a node that accomplishes a goal is reached, and that node is returned, otherwise, the goal couldn't be validated with a reliable plan with the given actions.

AStar either returns a null or a leaf node that accomplishes the goal, or a leaf node that doesn't even accomplish the goal. With the leaf, the resulting plan can be calculated, since the leaf references a parent node and so on. The result is a queue of type ReGoapActionState with each ActionState defined by the node's held action and the action's personal settings. The following checks are done after the leaf node is made:

- If the leaf is null, then we continue to make a plan for the next priority goal
- If the current plan equals the resulting plan calculated from the leaf node, then we break, because likely the same previous goal is calculated, so the current priority goal inside planner is set to null and we break.
- If the resulting plan has no action states, then continue to calculate a plan for the next priority goal

Passing these checks means that that current priority goal is validated with a plan, and the resulting queue of action states is referenced by the current priority goal.

After planning two things happen:

- Run the callback method to formally have the planner manager tell the ReGoapLogger that planner completed planning, and to add the finished work to the queue of completed works, to then be later executed upon update in the planner Manager.
- None of the goals could be validated!

The Resulting Plan in the FSM Example:

In the FSM example, this is a macro plan of 10 action states. The plan is written in order of the agent executing the plan:

1. GenericGotoAction: Move to the nearest resource required by the goal state which is a Tree GameObject

2. GatherResourceAction: Delimit 1 from the Tree GameObject capacity, and add that to resource bag
3. GenericGoToAction: Move to the nearest resource required by the goal state which is a Ore GameObject
4. GatherResourceAction: Delimit 1 from the Ore GameObject capacity and add it to the agent's resource bag
5. GenericGoToAction: Move to the near Workstation location
6. CraftRecipeAction: Craft the Wood recipe which delimits 1 from the agent's resource bag to produce Wood.
7. CraftRecipeAction: Craft the Axe Recipe which delimits 1 of each an Ore and Wood from the agent's resource bag
8. GoToAction: Go to the nearest BankPosition
9. AddResourceToBankAction: Deposit an Axe to the bank

Preconditions and effects are only relevant for chaining action state nodes together during planning, and they don't directly affect the agent's memory state. The memory state is only updated by the agent's sensors. When an action is pushed for execution it runs and either fails or ends. When the action ends, there is a push for the next action, but if the action fails, then a new goal is to be calculated.

The agent will stop moving in the FSM example if it cannot validate any goal. The agent could fail its current plan if there are no trees to collect, no ores, no resources to build an axe or wood, and no axe to deposit into the bank. In this case, the plan fails immediately when there are no more trees to gather, as the trees each have a capacity of 2 compared to an ore with a capacity of 5.

FSM:

The FSM is defined with two states in the FSM Example: SmsGoTo and SmsIdle. The given FSM structure is defined by classes ISMState, SmState, and StateMachine. With no reference to the parent, the goal state is considered a root, and consequently its state is a copy of the current memory state. Also, as the root, the node doesn't reference the next Action. The goal of the node equals the goal state given. In the example the goal has only one condition, so the heuristic is 1, and as the root, so $g = 0$, thus the cost of the node is 1.