Jayden Lombardi + Drew Herlocher

Game AI 340-02

Professor Tolstenko

Goal Oriented Action Planning

Overview:

What exactly is goal-oriented action planning? It is an artificial intelligence technique used to allow an agent to dynamically plan a sequence of actions, in order to achieve a specific goal. It is the process of an agent (usually an enemy) finding the optimal sequence of actions between its current state and goal state while considering objectives, constraints, or priorities. If A* algorithm is one level higher than Dijkstra's algorithm, GOAP is one level higher than A*. GOAP combines traditional navigation methods, with higher-level decision making. Agents consider the fastest physical path, AS WELL as how their actions align with specific goals they need to achieve. Agents build plans by searching through possible actions. In other words, adding multiple goals of varying priority to an A* algorithm, while also taking into account current statistics (such as health, hunger, reload, etc.). Agents actively think, plan, and decide where to go AND what to do in order to meet a goal state.

Technique Explanation:

Source 1 - Key Terms and A*

From this source we are able to learn and understand the Key Terms and components of implementing Goal Oriented Action Planning, mainly the Goal, the Plan, the Action, and the Planner/Plan Formulation. The goal is a state or condition of the world that an agent seeks to satisfy. An agent can have multiple goals, with only one active at any time guiding their actions. Goals can be further categorized into relaxed, investigative, or aggressive (or other categories). GOAP takes in a goal and outputs what is known as the Plan. The plan is a sequence of actions created by the Planner taking an agent from its current state to the goal state. The plan is generated dynamically, and changes as environmental stimuli change. The Plan is created through a series of Actions, which are simply steps (reload, shoot, kill, etc.). Each action knows when it is valid to run (known as preconditions) and what it will do in the game world (effects). This allows for GOAP to chain actions into sequences, and check multiple actions at once. The Planner is responsible for linking these actions and returning a path. An agent generates a plan by supplying a goal to the planner, which then searches through a pool of action for the fastest/cheapest sequence of events that will take the agent to the goal state. By searching through a pool of actions, behaviors can be shared, changed, or maintained across different character types, as they are not hardcoded to one. Both a cyborg and a human can use the OpenDoor action, but the cyborg will smash the door off the hinges and a human will gently open it.

Another important component of GOAP is the use of the A* algorithm. The Planner uses the A* algorithm to calculate the costs and pathfinding within the pool of actions. A* algorithm is used to find the cheapest sequence of actions from the agent's current state to the goal state. A regressive A* search is used as it is more efficient and intuitive. Starting at the goal state, the planner checks its preconditions, if they are NOT met, the planner searches for an action within the action pool that has effects that satisfy these conditions. Then check the preconditions of the new action, and see if we have to regress another level. This is continued until all preconditions of the goal state have been met. A new plan is formulated when an agent completes a previous plan, it becomes invalid, another goal becomes more relevant, or something changes in the environment to alter the plan.

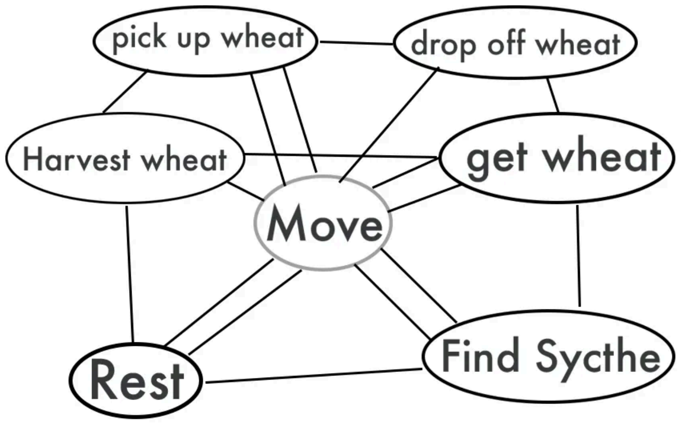
Source 2 - GDC Talk - Origin and Optimization

Where did Goal Oriented Action Planning come from? We can credit this awesome concept to the game F.E.A.R. created by Jeff Orkin and can be seen in various titles since. Instead of hardcoding every interaction, Orkin searched for a way that adapts behaviors dynamically. As mentioned in the previous source, the Planner uses a priority-based goal list to create a plan and is rerun when the environment changes. He created GOAP, allowing agents to adapt to unexpected situations or interactions. Making use of modular actions and goals allows for dynamic adaptability and easier scalability.

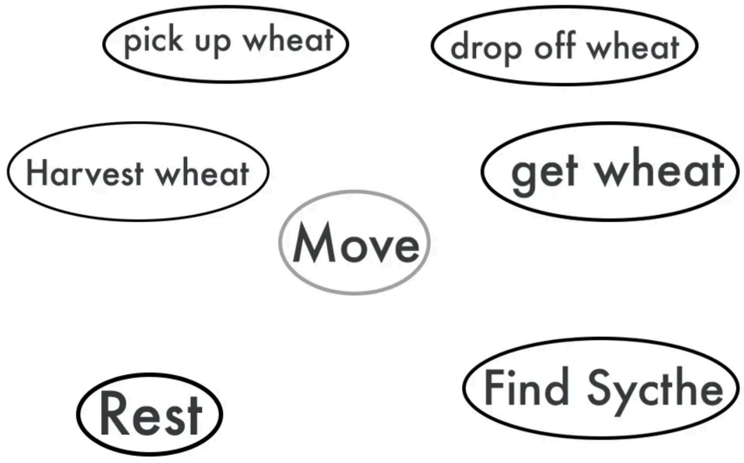
A few optimization techniques can be implemented to improve GOAP even more. Reducing the number of world state variables optimizes the process by simplifying the search space. Fewer variables mean a smaller space, allowing for faster computation at less computational cost. Another implementation to optimize GOAP would be having the agents be able to learn. Having agents track success rates for actions and sequences, and then adjusting their preferences accordingly would have agents learning from experience, and picking the best path more often. Even further down the optimization line, not ALL decisions need to use the Planner! GOAP involves searching the entire action pool, which is computationally expensive! Simple actions such as animations or straightforward state transitions can be handled by other, simpler state machines. They do not (and should not) use the entire GOAP system.

Source 3 - Better than FSM

So why are we using Goal Oriented Action Planning instead of a Finite State Machine? The simple answer is that when using FSMs in complex games, the hard coding can get extremely hard and redundant. GOAP is extremely modular and easier to maintain even when projects get to be extremely large and complex. Each action is self-contained, so editing one will not cause a cascading effect that will ruin your game. Instead of hardcoding interactions between every single state on a FSM like this:

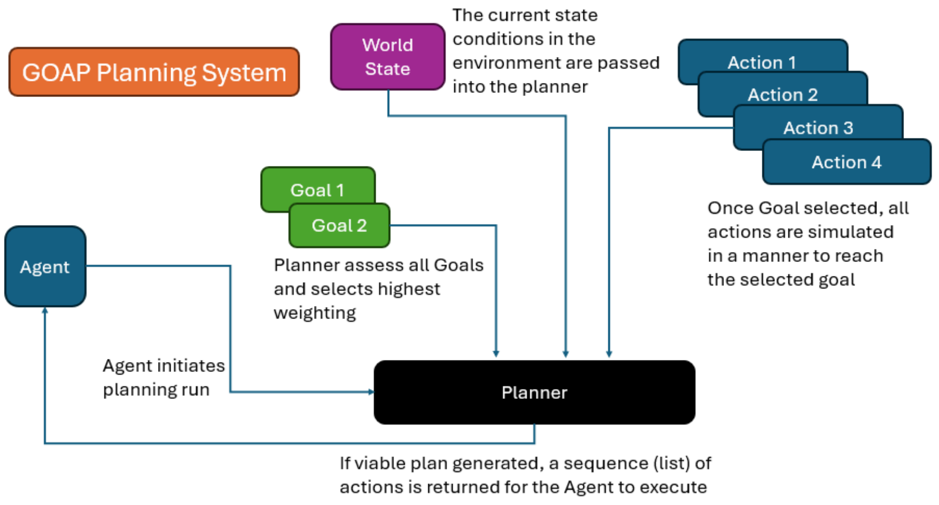


GOAP is able to define each individually and bridge the gap between them when necessary:

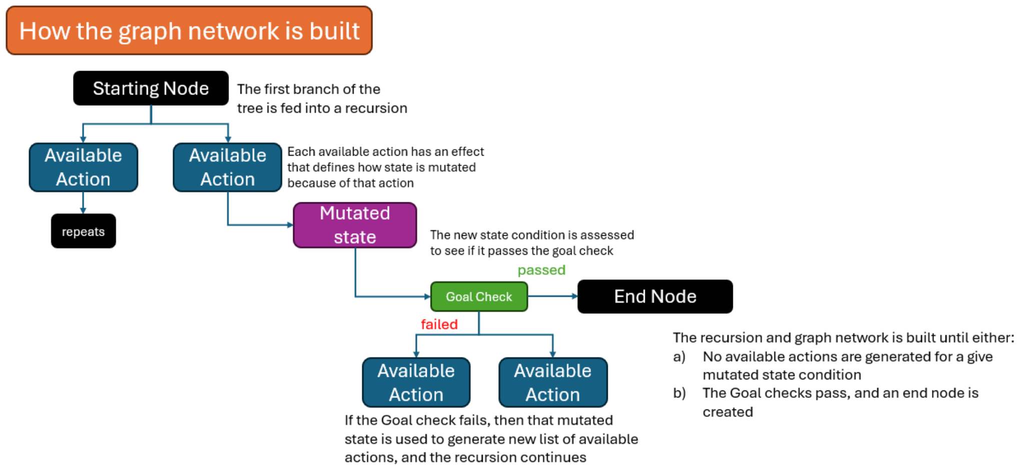


Source 4 - Visuals

Not too much new information was learned from this source, although I found the visuals on this website to be extremely helpful in understanding the key components of GOAP:


GOAP Planning System
World State
The current state conditions in the environment are passed into the planner
Action 1
Action 2
Action 3
Action 4
Goal 1
Goal 2
Agent
Planner assess all Goals and selects highest weighting
Once Goal selected, all actions are simulated in a manner to reach the selected goal
Agent initiates planning run
Planner
If viable plan generated, a sequence (list) of actions is returned for the Agent to execute

As well as helpful in understanding the the graph network that the Planner uses to search for actions within the action pool:


How the graph network is built
Starting Node
The first branch of the tree is fed into a recursion
Available Action
Available Action
Each available action has an effect that defines how state is mutated because of that action
repeats
Mutated state
The new state condition is assessed to see if it passes the goal check
passed
Goal Check
End Node
failed
Available Action
Available Action
If the Goal check fails, then that mutated state is used to generate new list of available actions, and the recursion continues
The recursion and graph network is built until either:
a) No available actions are generated for a give mutated state condition
b) The Goal checks pass, and an end node is created

Platform Suitability:

What platforms are <u>BEST</u> for GOAP?:

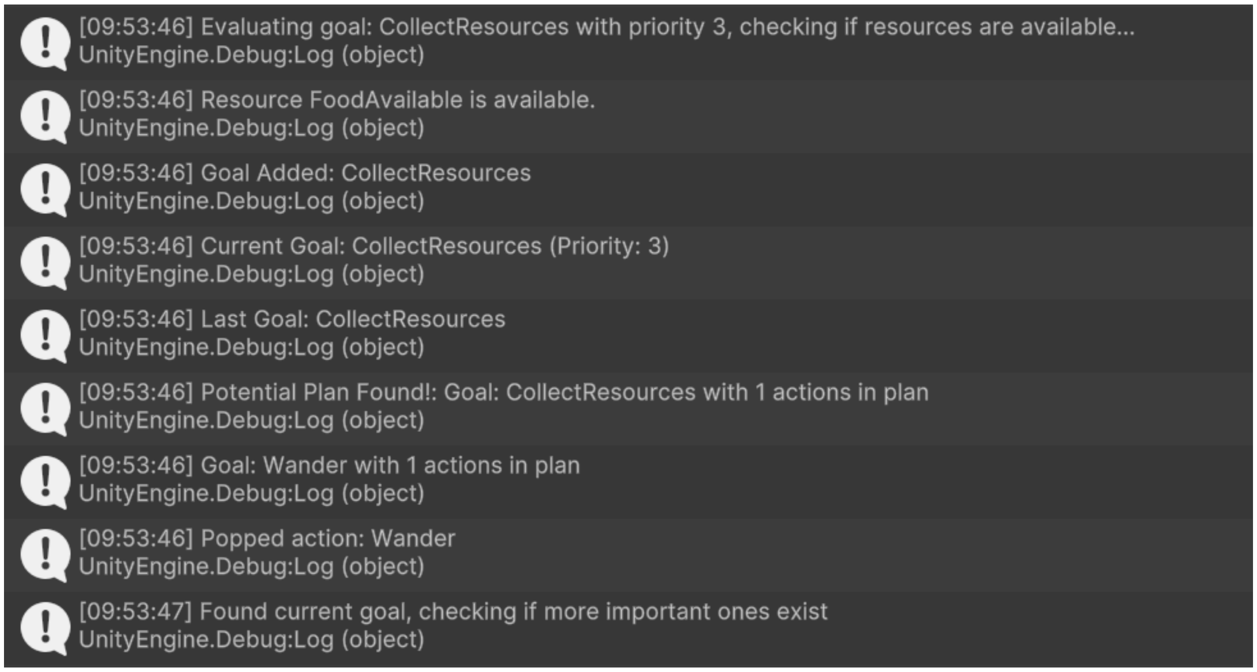
Goal Oriented Action Planning is best used on high-performance platforms such as newer-generation computers and consoles, capable of handling real-time decision-making. GOAP is best run within real-time strategy, life simulator, open-world, and Unity-based (what we are doing) games on these new-gen platforms. Within city-building games like The Sims, GOAP manages non-playable characters (NPCs), balancing objectives such as eating, resting, communicating, and working. It allows for constantly changing priorities and tasks within these types of games. GOAP truly excels within open-world games such as Breath of the Wild and Skyrim. In these types of games, GOAP is used to manage NPCs that are navigating the world, seamlessly switching between combat, exploration, reloading, and other tasks in an ever-changing world. Additionally, Unity-based games are extremely easy to implement GOAP into, which is what we have done! Unity has a built-in Navigational Mesh system that supports agents and pathfinding logic. The agents within this system can dynamically recalculate their paths based on changing environments. Navigational Mesh does NOT inherently support multiple weighted goals. That being said, GOAP can be implemented on top of NavMesh by scripting logic to evaluate multiple objectives of different weights. Adding the planning, thinking, and executing on top of the NavMesh will result in a basic implementation of GOAP!

What platforms are <u>WORST</u> for GOAP?:

Goal Oriented Action Planning should NOT be used on low-performance, low-resource platforms such as (some) mobile devices and older, outdated consoles. GOAP requires a significant amount of computational resources, as it constantly evaluates and prioritizes multiple goals and paths, especially when the environmental stimulus changes just as fast. Platforms like mobile devices or older consoles (PS1, Xbox360, etc.) with strict hardware limitations may be unable to run GOAP, and a simpler AI would have to be substituted. As for game genres, GOAP should NOT be used within constrained platformers or turn-based tactical games. In platformer games with rigid, constrained movements GOAP will be unnecessary and a bit overkill. If an agent only has one valid move (or is only allowed to perform one action) why would we need dynamically changing decision-making logic? We wouldn’t. An example of this being a Goomba enemy from the New Super Mario Bros Wii. These enemies will move back and forth on their current platform, following a “patrol”-like script. Therefore implementing GOAP here would be useless, as they are not intended to plan out action to achieve a goal, just move back and forth.

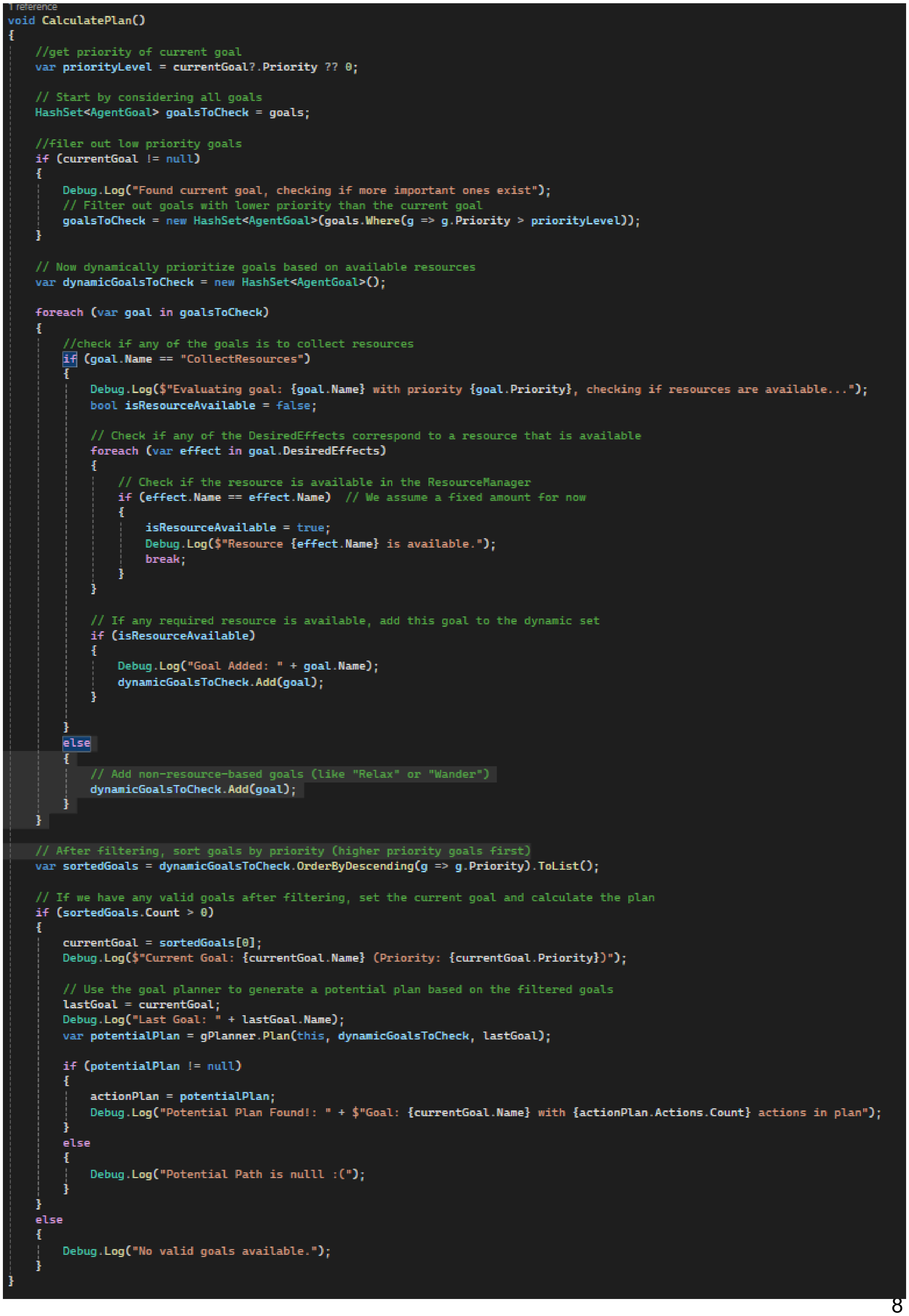
On the other hand, turn-based games (like the XCOM series) may seem like the perfect place to use GOAP. Having agents dynamically think/plan about how to win based on the player's previous move, but alas it is not. Often times these games use a simpler AI planning system, as the long pauses between each turn allow time for in-depth, thorough calculations. This means real-time dynamic updates are not needed, and the GOAP may not add too much to the AI.

Diagrams/Visuals:



```
[09:53:46] Evaluating goal: CollectResources with priority 3, checking if resources are available...
UnityEngine.Debug:Log (object)
[09:53:46] Resource FoodAvailable is available.
UnityEngine.Debug:Log (object)
[09:53:46] Goal Added: CollectResources
UnityEngine.Debug:Log (object)
[09:53:46] Current Goal: CollectResources (Priority: 3)
UnityEngine.Debug:Log (object)
[09:53:46] Last Goal: CollectResources
UnityEngine.Debug:Log (object)
[09:53:46] Potential Plan Found!: Goal: CollectResources with 1 actions in plan
UnityEngine.Debug:Log (object)
[09:53:46] Goal: Wander with 1 actions in plan
UnityEngine.Debug:Log (object)
[09:53:46] Popped action: Wander
UnityEngine.Debug:Log (object)
[09:53:47] Found current goal, checking if more important ones exist
UnityEngine.Debug:Log (object)
```

- Debug Logs explaining how the GOAP is adding Goals and Creating Plans (a bit messed up)



```
1 reference
void CalculatePlan()
{
    //get priority of current goal
    var priorityLevel = currentGoal?.Priority ?? 0;

    // Start by considering all goals
    HashSet<AgentGoal> goalsToCheck = goals;

    //filer out low priority goals
    if (currentGoal != null)
    {
        Debug.Log("Found current goal, checking if more important ones exist");
        // Filter out goals with lower priority than the current goal
        goalsToCheck = new HashSet<AgentGoal>(goals.Where(g => g.Priority > priorityLevel));
    }

    // Now dynamically prioritize goals based on available resources
    var dynamicGoalsToCheck = new HashSet<AgentGoal>();

    foreach (var goal in goalsToCheck)
    {
        //check if any of the goals is to collect resources
        if (goal.Name == "CollectResources")
        {
            Debug.Log($"Evaluating goal: {goal.Name} with priority {goal.Priority}, checking if resources are available...");
            bool isResourceAvailable = false;

            // Check if any of the DesiredEffects correspond to a resource that is available
            foreach (var effect in goal.DesiredEffects)
            {
                // Check if the resource is available in the ResourceManager
                if (effect.Name == effect.Name)  // We assume a fixed amount for now
                {
                    isResourceAvailable = true;
                    Debug.Log($"Resource {effect.Name} is available.");
                    break;
                }
            }

            // If any required resource is available, add this goal to the dynamic set
            if (isResourceAvailable)
            {
                Debug.Log("Goal Added: " + goal.Name);
                dynamicGoalsToCheck.Add(goal);
            }

        }
        else
        {
            // Add non-resource-based goals (like "Relax" or "Wander")
            dynamicGoalsToCheck.Add(goal);
        }
    }

    // After filtering, sort goals by priority (higher priority goals first)
    var sortedGoals = dynamicGoalsToCheck.OrderByDescending(g => g.Priority).ToList();

    // If we have any valid goals after filtering, set the current goal and calculate the plan
    if (sortedGoals.Count > 0)
    {
        currentGoal = sortedGoals[0];
        Debug.Log($"Current Goal: {currentGoal.Name} (Priority: {currentGoal.Priority})");

        // Use the goal planner to generate a potential plan based on the filtered goals
        lastGoal = currentGoal;
        Debug.Log("Last Goal: " + lastGoal.Name);
        var potentialPlan = gPlanner.Plan(this, dynamicGoalsToCheck, lastGoal);

        if (potentialPlan != null)
        {
            actionPlan = potentialPlan;
            Debug.Log("Potential Plan Found!: " + $"Goal: {currentGoal.Name} with {actionPlan.Actions.Count} actions in plan");
        }
        else
        {
            Debug.Log("Potential Path is nulll :(");
        }
    }
    else
    {
        Debug.Log("No valid goals available.");
    }
}
```

Above is the CalculatePlan function. This function is run withing GOAPAgent, and uses the GOAPPlanner to calculate a plan. This function specifically calculates which goal to pursue based on their priorities, and also checks if the resources are actually available for that goal. If resources are available, a plan is created to achieve the goal. If resources are not available, No valid goals is output to the debug log.