When beginning the assignment, I utilized the classes created in the week 6/7 lab as a base for the GOAP system. Core classes such as GOAPAction and AGOAPActor didn’t require altering initially, however the GatherAction and Gatherer classes were modified to incorporate the resources provided in the base code. These were expanded to be individual for the Tree and Stone resources, and later in the extended functionality the Metal resource, which resulted in a significant amount of duplicated code, which I will explain in detail later. Thankfully other classes such as DepositAction did not need to be duplicated, thanks to the inclusion of the Resource and MaxResource values in the parent AGOAPActor class. PickupFoodAction was similarly only a single class, as the health, maximum health and seek food values were set as fields in AGOAPActor, while their values per class were later defined in the appropriate constructors. The Builder class was unique and didn’t utilize DepositAction or a Gather class like the Stone and Tree gatherers, thus requiring multiple new classes. Aside from the ABuilder class itself, I also added the PickupMaterialsAction class, which retrieved materials from the village center, and the BuildAction class, which added those resources to the building. One decision worth noting at this point was a focus on movement efficiency. Although not specified in the assignment specifications, all actions prioritise moving the agent as little as possible, and thus agents will only deposit when their inventory is full, which means they return to gathering after feeding. The Builder similarly will not execute the BuildAction unless it can carry the maximum stone and wood, however per the requirements that “1 stone & 1 wood unit are consumed every second”, it will not finish an inventory if one resource runs out – which will always be the case as it holds 5 more wood than stone – thus meaning its movement is not efficient like the rest of the actors.

With the base code complete, I was able to move on to the extended functionality, which proved to be a significant challenge. I started by implementing the GatherMetalAction class, which was mostly identical to its tree and stone counterparts, and then added the ToolCrafter class. In order to have it integrate with the DepositAction class I decided to use the AGOAPActor fields “NumResource” and “MaxResource” to refer to the tools it was carrying, which resulted in some weird-looking code where MaxResource is set to 0, to ensure DepositAction doesn’t think the ToolCrafter has less than the maximum it can carry (see above on efficient movement). Metal was instead tracked with its own fields in ToolCrafter. I also had to create a ForgeToolAction class to handle the creation of tools, though I also chose to have it handle depositing materials at the Forge, which the base code indicated could store metal. Implementing tools was the next challenge, and I decided to approach this using an integer field in the AGOAPActor class. The choice of an integer was simple enough, as setting it to 0 on spawn, 75 when a tool was picked up and decreasing by 1 every successful resource gather was far easier than a more complicated approach like another class. My decision to use AGOAPActor to store the field, despite other children like the Builder and ToolCrafter not using it, was to avoid multiple casts every game tick – casting up to 8 times per tick would affect performance, though likely not enough to be noticed, as well as add far more unnecessary code. In order to ensure actors picked up tools before gathering, I added the “HasTool” Boolean as a precondition for gathering, and in AGOAPActor’s GetWorldState, added a field that sets this value to true or false based on if the ToolHealth value is greater than 0. Additional checks were added to StoneGatherAction and TreeGatherAction in CheckProceduralPrecondition, as well as a check every game tick to handle if the tool breaks during gathering.

On the topic of things I’d do differently, many of my technical decisions were made for the sake of achieving the assessment requirements as quickly as possible, and are thus not efficient or extensible. One such example is the similarity between the stone and tree gatherers/actions, which are almost identical and differ only in the provided resource, and for actions, the agent interacting with them. Ideally I would have liked to fully refactor the base code to utilize a single resource class, with the type of resource and other appropriate factors defined in blueprints, which would have then allowed for a single gatherer, gather action and deposit action capable of being used by all three, or at the very least made a parent class with the majority of the functionality in a virtual function. This would have prevented significant duplicate code, thus meaning changes in functionality didn’t need to be replicated across two or more classes, and also made extending functionality simpler by preventing the need to duplicate every single class again.

Another example is the lack of a custom GameState, which could have been used to make various actions more efficient, including all uses of GetActorsOfClass. This would additionally have allowed for a shared WorldState, allowing actors to make more informed decisions, and again providing greater extensibility if this project were to be revisited.

One of the more visible quirks is the method of finding targets used. Due to an early issue with the base code preventing the hitbox detection method being used to find valid targets, I implemented a different approach utilizing the aforementioned FindActor(s)OfClass. This successfully found the targets, but caused issues with clipping, as agents were pathfinding to the root of the object instead of the hitbox, and by the time the fix for the base code issue had been released, I’d already used the new method multiple times, and again in the interest of time deprioritized this fix. Subsequently this could be considered a bug, however it does not impact the AI. Another bug is the gathering timer used by the tree, stone and iron gatherers – the timer is initially set when the Reset function is called, however this can be called before movement, thus meaning when the agent reaches the resource they are able to immediately gather without waiting the specified time. This only occurs on the first harvest after an action is ‘accepted’, and the implementation of the PickupFoodAction and ForgeToolAction classes have a simple solution to this by setting the TargetTimer to null on reset and by default, then conditionally assigning it a TargetTime in PerformAction if TargetTime is null, however I didn’t get around to fixing this.

Aside from technical debt and visual quirks, perhaps my biggest regret was not using floating text to display the resources held by an agent. Despite the appropriate code existing in the week 6/7 tutorials, I opted not to include it, and in hindsight this would have saved me a tremendous amount of debugging, as clearly visible resource counts would make it clear what an actor was doing without the need to constantly step through code, while additionally making the simulation easier to watch.