SCHOOL OF COMPUTING

UNIVERSITY OF TEESSIDE

MIDDLESBROUGH

TS1 3BA

AI for Games

(COM3049)

ICA3 Group Report

Blackbird

Will Marshall

Rob Nicholds

Peter Southward

Table of Contents

1 System Architecture overview

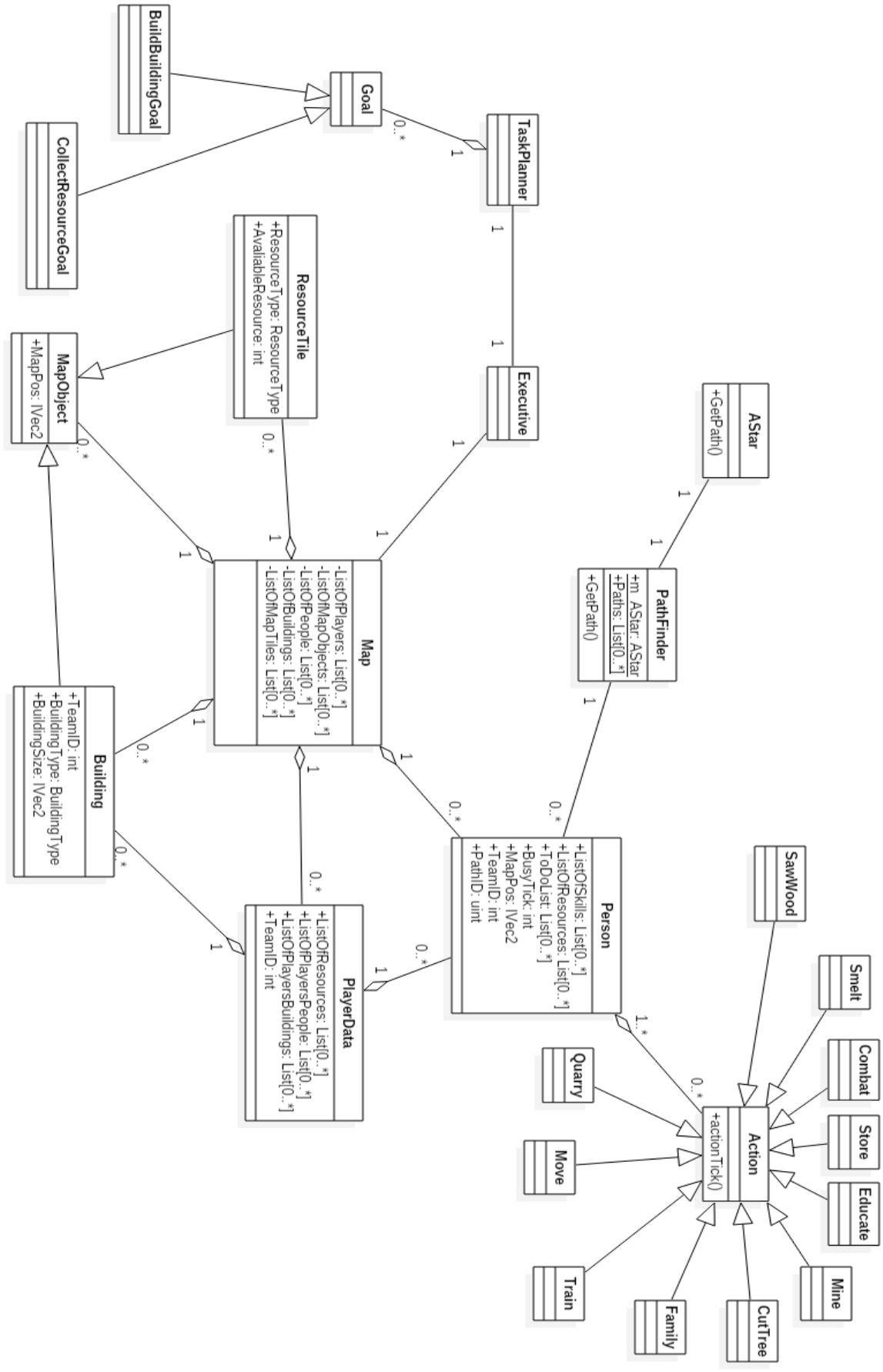
2 Motion planning

3 Task planning

4 Plan Executive

5 Results

1 SYSTEM ARCHITECTURE OVERVIEW



2 MOTION PLANNING

For the path planner we are using AStar with a Euclidean heuristic. The algorithm we have implemented has two Lists, one is an open queue that contains the Nodes to be looked at, the other is a close queue that contains both the nodes that have been looked at and the nodes in the queue. We have implemented the open and close lists like this as it is easier to check if a node has been looked at or is about to be looked at. While looking at a node there are four key steps the first is the check to see if the current node has reached the goal, second is to get the next available nodes form the current node, third is to check the new nodes against the close list and finally is the sort of the open list.

To check if the current node is at the goal is a simple check. The algorithm checks the position of the current node to see if it is the same as the goal position.

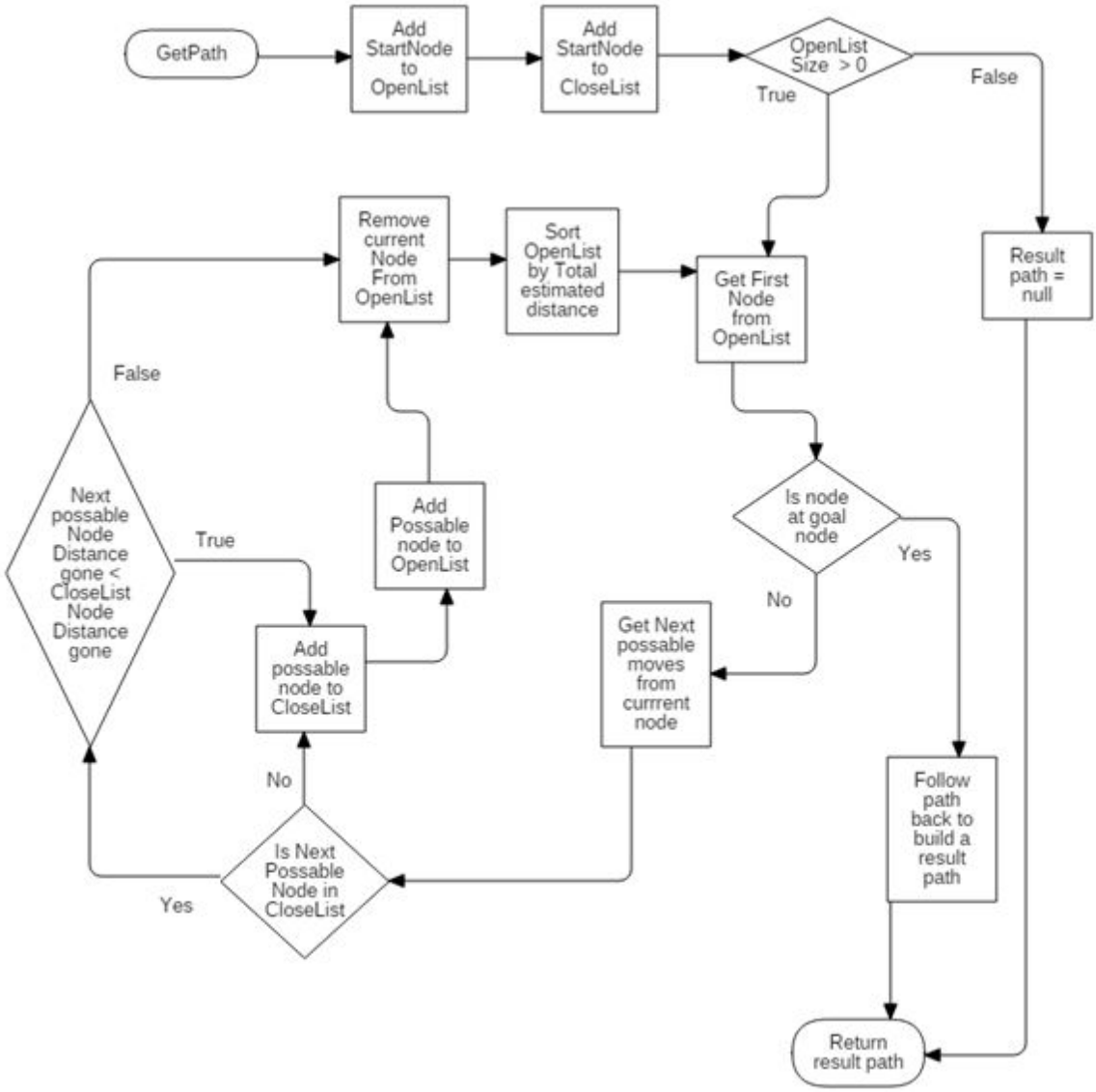
For getting the next position, is again very simple. The algorithm is just getting the positions in a three by three square around the current node. Once it has a new node it then checks to see if it is a traversable node, if it is, the node is then returned as a valid node.

When checking a node against the close list. We first check the position, if the position is not in the close list it returns that is it valid. If it doesn't find the position it then checks the distance gone. If the distance gone is less than the distance gone in close list we then remove the node from the close list and return that it is valid. If any of these check fail, we return that it is not valid.

When the close list check returns, if it is valid we then add the node to both the open and close list. If it was not valid we discard the node.

The final point is when we sort the open list, we sort it on results of the distance gone and the Euclidean heuristic value.

Below is a flow diagram of our AStar algorithm:



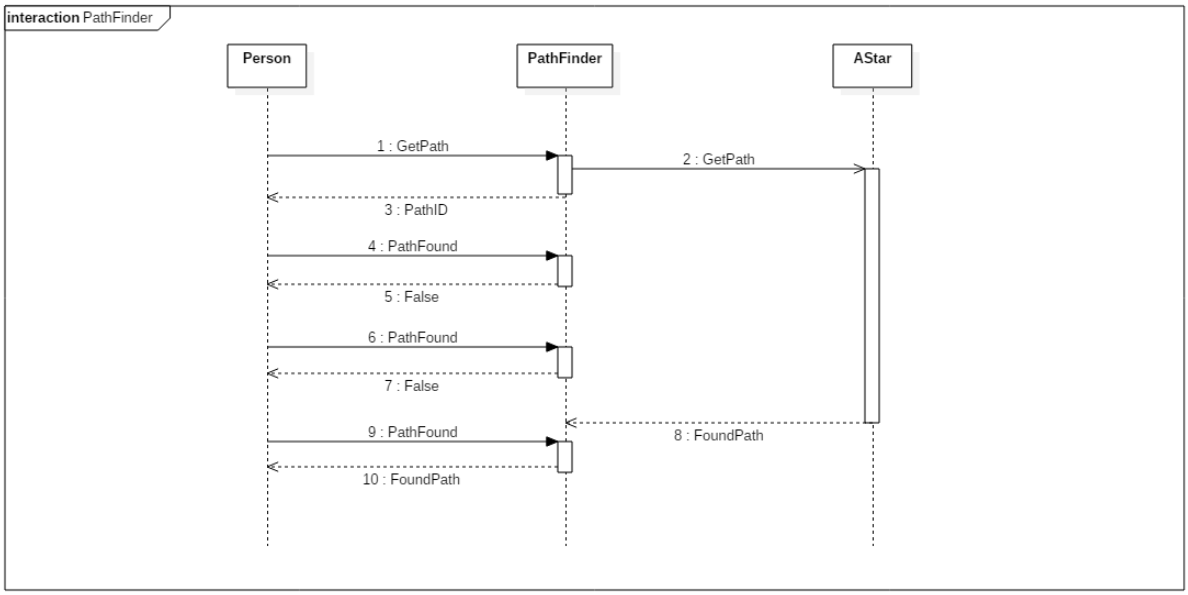
The way the path planner works in our application is when you call GetPath from the PathFinder class it creates a unique ID and used StartCoroutine to start the AStar algorithm with a set number of millisecond it can run per frame, the path finder then returns an ID for you to look up a resulting path. This does mean there can be a

delay between asking for a path and getting one but it also means that the game doesn't freeze while a path is found.

When a path is found it stores it in map which stores the result against the ID the path finder returned in the GetPath function. In our case we also pass in the person who requested the move path as it then adds the move commands to the todo list of the person. This means it will only move once per game tick. As the person travels along the path it removes the first node in the result list to make sure that the first node in the list is the next node it need to travel to.

The layering between the PathFinder class and the AStar class although very simple it keeps options open to being able to easily swap out different AI methods without having to changing the code anywhere else in the game. This method is used in many game engines to both keep the system modular and be able to easily swap to different APIs / methods.

Below is an interaction diagram of how the path planner is integrated in the game:



There are a few ways we could improve the path finding short of changing the method, one way would be to change it from a StartCoroutine to a thread as this will free the path finder from the per frame lock it currently has. Another way we could improve it is to look at ways of optimizing the code as we didn't have time to do any optimization.

Possible problems with the current solution include:

- The object that requests the path currently handles the deletion of the path
    - This can could cause issues with memory usage if not set up correctly
- If there are a lot of path requests it could cause the game to freeze
    - We haven't encounter this but knowing the code we know this could be a possibility as each path will take a set amount of milliseconds per frame

3 TASK PLANNING

When planning out the implementation of the task planner, it was decided with the executive that the planner would not need to handle concepts such as places, doing away with planning out movements. The reason for this is because when the planner decides it needs to cut down a tree, it is implied that movement needs to take place to a tree somewhere in the game world and the executive simply finds the nearest tree, this can be applied to all tasks generated by the planner.

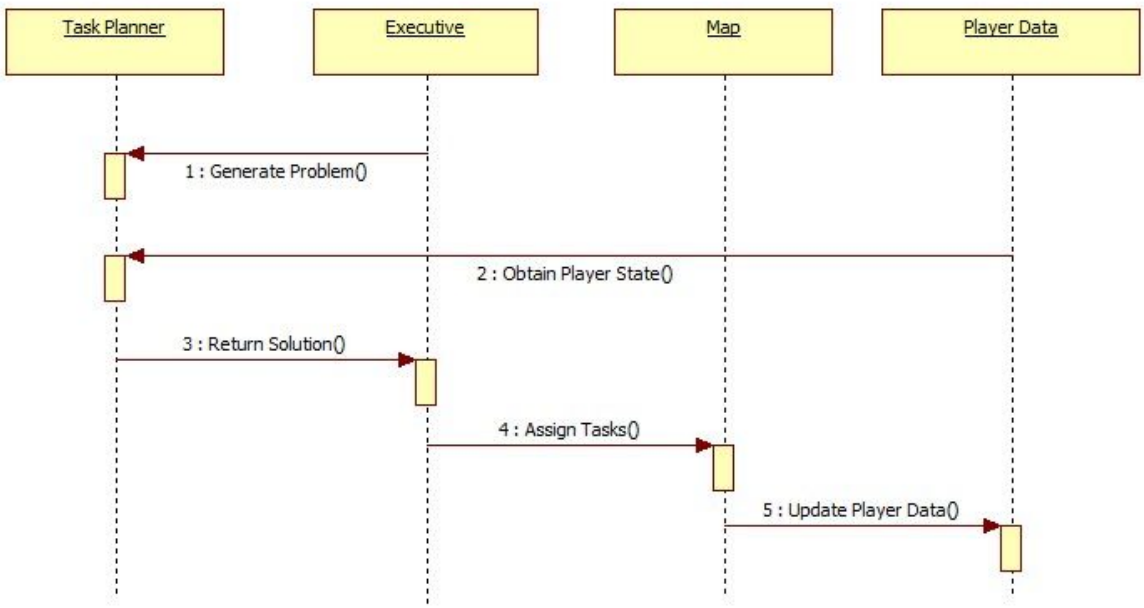
By eradicating actions such as moving to and from places in the task planner, means that problems that require a lot of actions can still be processed quickly and returned to the executive as fast as possible.

Generating problem files was the next challenge in development of the application. To inform the planner of what buildings each player has already, we loop through all the buildings that belong to each player, and write the appropriate predicate into the initial state of the problem file. For example if a player has a school stored within

their C# player data class, the predicate (has-school) is passed to the initial problem state.

The next step was to decide how the executive was going to pass goal states to the task planner. For this we created an abstract Goal class. The executive then simply passes a list of goals into the problem writer function that will be passed to the problem file. Two separate goal classes are created for the two distinct types of desired goals, the first being a goal to collect a resource of a specified type. So the executive only has to declare which resource they would like to collect and how much of it they require. The second type of goal is for building new buildings.

To generate correct solutions for all the possible problems the executive might require, values needed to be stored for the resources a player currently has, and then passed onto the planner. This means the planner will not give a solution that returns actions for obtaining resources that the player does not require. This is represented in the PDDL by the functions for each resource. The resource amounts are written into the problem file in a similar fashion to how buildings are past in, by looping through each resource type the player has and writing the amount to the problem file. Below is a diagram showing how the task planner interacts with the other systems within the application.



The PDDL domain contains all the necessary actions required in the assignment specification. Actions for constructing buildings also contain all the appropriate pre-conditions for construction. In the context of our task planner the preconditions are checking whether a person of an appropriate skill is available and if we have the correct amount of resources required. Below is a snip of an action for building an ore mine.

```
(:action buildOreMine
      :parameters(?resource - resource)
      :precondition(and(has-carpenter) (has-blacksmith) (ore_resource ?resource)  (not(has-oremine))
                  (>= (wood) 1)
                  (>= (iron) 1))
      :effect(and(has-oremine)  (decrease (wood) 1) (decrease (iron) 1))
)
```

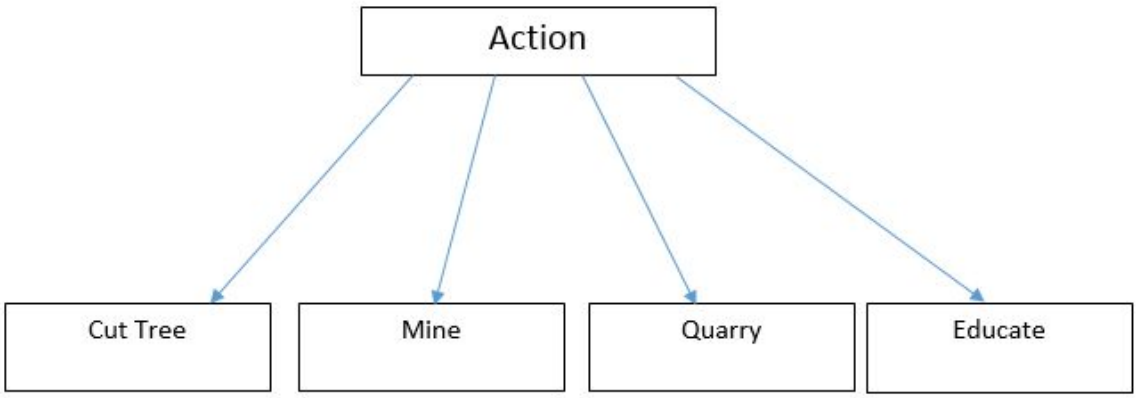
The above image shows that to build an ore mine we need to have a carpenter and a blacksmith to construct the mine. We also check that we have at least one wood and one iron resource available in global storage. Once the action is completed we then decrement the resources by the correct amount and inform the planner that we now have an oremine available for mining ore.

4 PLAN EXECUTIVE

The executive was in charge of providing the global goals of the AI players and passing these goals to the planner and obtaining viable solutions. The exectutive then uses exectues these solutions by interacting with the motion planner to move people around the map and perform tasks.

The decision was made that was made to have two AI existing in the game world playing against each other. Therefore each team's actions had to be processed separately. This was acheived by having a player data class that stores all data belonging to each individual player. The data stored includes a list of all the people belonging to that team, the buildings that team has access to and the number of resources the team has.

Moving on from the storage of team data, solving the problem of allowing people to perform actions was the next task. This is acheived by having a abstract Action class. With individual classes branching from this for every possible action.



Each person in existence in the game world then contains a to do list of actions that it needs to perform. These actions are read in from the solutions generated in the task planner. Once actions are assigned the process of carrying them out begins. In

the case of acquirring a resource, such as a tree. The applications looks for the nearest available resource of the given type. So in the example of acquirring timber, we look through the Map class and locate the nearest wood tile. Once an appropriate location is found we generate a path using the motion planner, and move to the location. To account for the time required for performing an action, we set a time that the person is busy for, meaning they do not move onto the next action in their queue until the previous one is completed.

One of the other problems encountered is the decision for building new buildings. First a valid location for the new building is decided, the location is decided by the person who has the task of construction assigned to them. This person will look in a radius around them and search for a valid location to place the building. The step is to check we have the correct amount of resources available for construction. If enough resources are available, we erect the building and deduct the required resources from the global pool.

5 RESULTS

In the final version of the application, several of the requirements have been met. The motion planner is fully functional and agents within the game world can move to their required locations. The motion planner also works fast and efficiently so the application does not wait upon it.

The task planner also meets its necessary requirements. Problems can be generated by the executive and the problems can be solved quickly by the planner. The planner can also handle being asked to obtain specified amounts of resources at the same time as constructing new building and training people in new skills

Finally the executive interacts with both of these systems to acheive globally specified goals. Several problems can be successfully generated and solved in

conjunction with the task planner and the motion planner. People in the simulation successfully acount for the resources required for meeting goals and the skills required for them.