Goals:

Create an agent that utilizes GOAP (Goal oriented action planning)

- ✓ Use Planning Ahead to consider states, events and actions that can have long term outcomes.
- ✓ Implement an Intelligent searching algorithm to opt best suitable path in a flexible manner while using independent actions.

Technologies, Tools, and Resources used:

- ➢ Python IDE(PyCharm) with python 3 installed
- ➢ Piglet Documentation http://pyglet.readthedocs.io/en/pyglet-1-3-maintenance/
- ➢ Python 3 Documentation http://docs.python.org/
- ➢ Using the code from previous lab tasks to implement GOAP combined.

Task Undertaken:

- Implement classes, states and on trigger events to remote the actions that can be performed or chosen to depict certain goal state based on the resources and considering effects of each actions to act intelligently to achieve the outcome. Similarly, there must be certain events that trigger off that state change so it would require some presets conditions and the effects of each off state event change that can be combined together into a cohesive plan between each state independently, as if it where to be voided the path-planning algorithm would not be able to opt the best suitable path of action to achieve the goal outcome.
- Now, the testing implementation part; instead of having to manually choosing the action after performing multiple action to get the outcome we'll use GOAP to search the best possible action to achieve the outcome without actually going through each individual action, this way the search algorithm would take the key initiative to asses the resource and the listed actions to achieve the goal outcome and then puts out the best possible action that checks ahead of time for the side effects if one action could have on the outcome of the goal.

```
def choose_action_goap(max_depth):
    states = [[State(goals, actions), list(actions.keys())[0]]]

    best_action = None
    best_value = 1000000000
    best_plan = []
    VERBOSE = True
    # Tells if the planning has moved back 2 depth levels or more for the right iteration
    moved_back = False
    if VERBOSE: print('Searching...')

    changed = True
    # The action from the previous depth, used for when the planning has moved back 2 depth levels or more
    previous_depth_action = states[0][1]
    # The action from the current depth, used for when the planning has moved back 1 depth level
    current_depth_action = list(actions.keys())[0]
    while states:
        # Get the discontentment value the current state can get from the planned action
        current_value = states[-1][0].discontentment(states[-1][1])
        # Apply action
        states[-1][0].apply_action(states[-1][1])

        # if the plan queue has reached the maximum depth, compare to see if the current plan is the best, then pop back one depth level
        if len(states) >= max_depth:
            if current_value < best_value:
                # Set a new best plan and the best action is the first action of the plan queue
                best_action = states[1][1]
                best_value = current_value
                best_plan = [state[1] for state in states if states[1]] + [best_value]

            states.pop()
            continue

        if moved_back:
            # If the planning has popped back 2 depth levels or more, then take the next action of the previous_depth_action
            next_action = get_next_action(previous_depth_action)
        else:
        else:
            # If the planning only popped back 1 depth level, then take the next action of the action from the current depth
            next_action = get_next_action(current_depth_action)
        if next_action:
            new_state = deepcopy(states[-1][0])
            states.append([new_state, None])
            states[-1][1] = next_action

            #new_state.apply_action_reset(next_action)
            if moved_back:
                moved_back = False
            else:
                current_depth_action = next_action

            changed = True
        else:
            # Reset the current depth's action
            current_depth_action = list(actions.keys())[0]
            try:
                # Get the action of the previous depth
                previous_depth_action = states[-1][1]
            except IndexError:
                pass
            # Set moved_back to True to tell that the planning has moved backward 2 levels or more
            moved_back = True
            states.pop()
    # Return the best action
    return best_action
```

What we found out:

The tasks undertaken were set to create a flexible set of actions that could be used with presets conditions and side effects they can have on the outcome, which allowed the searching algorithm to take the best suitable action to execute and get to goal, this mode of handling action resources is quite impressive and defines an extensible means for designing AI programmed NPCs/Enemies that can depict more intelligent and dynamic behaviour with easy and less time delays for the actual steps to achieve certain outcomes. This sort of behaviour can be implemented with Intelligent patrolling AI which is about path waypoints and stuff as this can help take actions to decide when to when to wander off from way points and when not to based on the resources in terms of energy and tier level it has compared to what aura it detects from the surrondings in terms of kill instincts. I believe this

will really help me with my Custom Project that I have in my mind as an update to the current one.

A little comparision has been done to provide with a better understanding and demonstration of achieve an outcome using Goal.Oriented.Action.Planning and without it to choose actions.

GOAP:

```
C:\Users\cools\AppData\Local\Programs\Python\Python38-32\python.exe "C:/Users/cools/Documents/AI for games
ACTIONS:
 * [get raw food]: {'Eat': -3}
 * [get snack]: {'Eat': -2}
 * [sleep in bed]: {'Sleep': -4}
 * [sleep on sofa]: {'Sleep': -2}
>> Start <<
---------------------------------------
GOALS: {'Eat': 4, 'Sleep': 3}
Searching...
BEST ACTION: sleep in bed
NEW GOALS: {'Eat': 0, 'Sleep': 0}
---------------------------------------
>> Done! <<

Process finished with exit code 0
```

Without GOAP:

```
C:\Users\cools\AppData\Local\Programs\Python\Python38-32\python.exe "C:/Users/cools/Documents/AI for games/
ACTIONS:
 * [get raw food]: {'Eat': -3}
 * [get snack]: {'Eat': -2}
 * [sleep in bed]: {'Sleep': -4}
 * [sleep on sofa]: {'Sleep': -2}
>> Start <<
---------------------------------------
GOALS: {'Eat': 4, 'Sleep': 3}
BEST_GOAL: Eat 4
BEST ACTION: get raw food
NEW GOALS: {'Eat': 1, 'Sleep': 3}
---------------------------------------
GOALS: {'Eat': 1, 'Sleep': 3}
BEST_GOAL: Sleep 3
BEST ACTION: sleep in bed
NEW GOALS: {'Eat': 1, 'Sleep': 0}
---------------------------------------
GOALS: {'Eat': 1, 'Sleep': 0}
BEST_GOAL: Eat 1
BEST ACTION: get raw food
NEW GOALS: {'Eat': 0, 'Sleep': 0}
---------------------------------------
>> Done! <<

Process finished with exit code 0
```

Althugh the output would seem simple at first, but the principles used in GOAP actually decouples the FSM allowing to easily pick new states and actions with the search and planning algorthim for best suited steps to achieve the outcome.