Spike: Spike_13
Title: **Goal Oriented Action Planning**

Author: Daniel Newman, 6449921

Goals / deliverables:
Utilising search algorithms, we can create an appropriate plan for executing independent actions to achieve a goal in a flexible manner.

Technologies, Tools, and Resources used:
List of information needed by someone trying to reproduce this work

- Python 3.6.4
- Python compatible IDE

Tasks undertaken:
To implement GOAP we had to declare some classes to facilitate the different states/events and actions we could use to trigger these actions on our way to getting to our goal state.

First we needed to define our GOAP state, and determine if we achieved any set preconditions we needed for the state change. It was also responsible for containing all the actions that were performed on the state, effectively functioning as a model of the world.

```
import copy
class GOAP_State:
    def __init__(self, states):
        self.states ={}
        for state in states:
            self.add_state(state)

    def add_state(self, name, value = False):
            self.states[name] = value

    def preconditions_met(self, action):
        for precondition in action.preconditions:
            if self.states[precondition['State']] != precondition['Needed']:
                return False
        return True

    def perform_action(self, action):
        for effect in action.effects:
            self.states[effect['State']] += effect['Result']
```

Then we had to create an action class that would dictate the behaviour the agent would do in order to trigger off the state change event. The precondition & effects are required to combine together into a cohesive plan between each independent state. Without them, the path-planning algorithm wouldn't be able to determine a suitable path of actions to complete its goal.

```
class GOAP_Action():
    def __init__(self, name, cost):
        self.name = name
        self.cost = cost
        self.preconditions = []
        self.effects = []

    def add_precondition(self, precondition, value = True):
        self.preconditions.append({'State':precondition, 'Needed': value})

    def add_effect(self, effect, value = True):
        self.effects.append({'State': effect,'Result': value})
```

Then our agent just had to plan a path of suitable actions that it could take to complete its goal objective. It did this utilising a DFS algorithm to plan out the set of actions to take, this was done to save memory utilisation.

```
class GOAP_Agent:
    def __init__(self):
        self.state = []
        self.actions = []
        self.running_cost = 0

        self.paths_evaluated = 0


    def perform_action(self, action):
        self.state.perform_action(action)
        self.running_cost += action.cost

    def plan(self, goal, state = None, path = None, start_action = None):

        if state is None:
            state = copy.deepcopy(self.state)

        if path is None:
            path = {'Actions': [], 'Cost': 0}
            self.paths_evaluated = 0

        if start_action:
            path['Actions'].append(start_action)
            path['Cost'] += start_action.cost
            state.perform_action(start_action)
```

What we found out:

Describe the outcomes, and how they relate to the spike topic + graphs/screenshots/outputs as needed

The steps taken earlier allowed us to create simple instantiations of actions and states, allowing us to create a flexible set actions that could be combined with preconditions and effects. This would then allow the planner system to then calculate the best approach to take and execute it. The use of GOAP creates an extensible approach to designing AI for NPC's and other enemies that can create more dynamic behaviour easily.

```
agent = GOAP_Agent()
agent.state = GOAP_State(
    [
        'HasFlour',
        'HasBread',
        'DoJob'
    ])
agent.actions = [
    bake_bread,
    get_flour,
    deliver_bread
]

get_flour.add_precondition('HasFlour', False)
get_flour.add_effect('HasFlour')
bake_bread.add_precondition('HasFlour')
bake_bread.add_precondition('HasBread', False)
bake_bread.add_effect('HasBread')
deliver_bread.add_precondition('HasBread')
deliver_bread.add_precondition('DoJob', False)
deliver_bread.add_effect('DoJob')



goal_state = 'DoJob'
path = agent.plan(goal_state)
```

While this was a simple plan, it outlined how GOAP is used to create a decoupled FSM, allowing us to easily plug in new actions and states which the planning algorithm utilised to create the best path based on the search method.

```
C:\Users\Dan\AppData\Local\Programs\Python\Python3
Goal: DoJob

1) get flour (5)
2) bake bread (10)
3) deliver bread (25)
Total cost: 40
Press any key to continue
```