

Spike: Task 4

Title: Goal-Oriented Behaviour

Author: Sam Huffer, 101633177

Goals / Deliverables

- Working code to demonstrate goal-oriented behaviour (GOB) using simple goal insistence (SGI). See “\04 – Spike – Goal Oriented Behaviour\SimpleGOB\SimpleGOB\SimpleGOB.py”.
- Demonstrations of where SGI does and does not work well.

Technologies, Tools, and Resources Used

- Visual Studio (VS) 2017 (for editing code)
- Learning materials on Canvas (for instructions and sample code)
- Command prompt (for executing and testing the code)

Tasks Undertaken

- Created a python3 project in VS and copied the sample code into it.
- Read the sample code to get a gist of what was going on, paying attention to comments denoting missing functionality.
- Compiled and ran the project to see what happened when the unaltered sample code ran.
- Went back to the comments about missing functionality, picked a feature, and implemented it (see below).
- Re-compiled the project and ran it again. If it broke or did not work as intended, I edited and re-ran the code until it worked as intended.
- Once one feature was done, I picked another piece of missing functionality and repeated.
- Once all missing features commented on in the code were implemented, I split the code into a Game class and an AI class according to object-oriented principles. I then ran the code and fixed any bugs that appeared.

Procedural Code Added / Edited

```
# Global (read-only) actions and effects
actions = {
    'get raw food': { 'Eat': -3 },
    'get snack': { 'Eat': -2 },
    'sleep in bed': { 'Sleep': -4 },
    'sleep on sofa': { 'Sleep': -2 }
}
```




```
# Global (read-only) actions and effects
actions = {
    'get raw food': { 'Eat': -3, 'Sleep': +2 },
    'get snack': { 'Eat': -1, 'Sleep': 0 },
    'sleep in bed': { 'Sleep': -4, 'Eat': +3 },
    'sleep on sofa': { 'Sleep': -2, 'Eat': +1 }
}
```

Figure 1: Adding side effects to each action listed.

```
# Find the most insistent goal - the 'Pythonic' way...
best_goal, best_goal_value = max(goals.items(), key=lambda item: item[1])

# ...or the non-Pythonic way. (This code is identical to the line above.)
#best_goal = None
#for key, value in goals.items():
#    if best_goal is None or value > goals[best_goal]:
#        best_goal = key
```




```
# Find the most insistent goal - the 'Pythonic' way...
#best_goal, best_goal_value = max(goals.items(), key=lambda item: item[1])

# ...or the non-Pythonic way. (This code is identical to the line above.)
best_goal = None

for key, value in goals.items():
    if best_goal is None or value > goals[best_goal]:
        best_goal = key
```

Figure 2: Swapping the “Pythonic” code for the “non-Pythonic” code for ease of reading.



```
def action_side_effects(action, goal):
    result = 0 # default is no side effects

    for g in actions[action]: # for each goal affected by the action
        if g is not goal: # if it is not the current goal
            result += -actions[action][g] # increment result by its effect

    return result
```

Figure 3: A function for returning the total side-effects of an action.

```

# Find the best (highest utility) action to take.
# (Not the Pythonic way... but you can change it if you like / want to learn)
best_action = None
best_utility = None
for key, value in actions.items():
    # Note, at this point:
    # - "key" is the action as a string,
    # - "value" is a dict of goal changes (see line 35)


    # Does this action change the "best goal" we need to change?
    if best_goal in value:

        # Do we currently have a "best action" to try? If not, use this one
        if best_action is None:
            pass
            ### 1. store the "key" as the current best_action
            ### ...
            ### 2. use the "action_utility" function to find the best_utility value of this best_action
            ### ...

        # Is this new action better than the current action?
        else:
            pass
            ### 1. use the "action_utility" function to find the utility value of this action
            ### ...
            ### 2. If it's the best action to take (utility > best_utility), keep it! (utility and action)
            ### ...

# Return the "best action"
return best_action

```



```

# Find the best (highest utility) action to take.
# (Not the Pythonic way... but you can change it if you like / want to learn)
best_action = None
best_utility = None
best_side_effects = None

for key, value in actions.items():
    # Note, at this point:
    # - "key" is the action as a string,
    # - "value" is a dict of goal changes (see line 35)

    # Does this action change the "best goal" we need to change?
    if best_goal in value:

        # Do we currently have a "best action" to try? If not, use this one
        if best_action is None:
            ### 1. store the "key" as the current best_action
            best_action = key
            ### 2. use the "action_utility" function to find the best_utility value of this best_action
            best_utility = action_utility(key, best_goal)

            best_side_effects = action_side_effects(key, best_goal)

        # Is this new action better than the current action?
        else:
            ### 1. use the "action_utility" function to find the utility value of this action
            utility = action_utility(key, best_goal)
            side_effects = action_side_effects(key, best_goal) # side effects is sum of magnitude of all side ef

            ### 2. If it's the best action to take (utility > best_utility), keep it! (utility and action)

            ### use this code if both eating and sleeping have -ve side effects of great enough magnitude that n
            ### the code is too concerned with managing the dominant goal
            #if utility > 0 and side_effects >= 0 and side_effects > best_side_effects: # if side effects are no
            #    best_action = key
            #    best_utility = action_utility(key, best_goal)
            #    best_side_effects = action_side_effects(key, best_goal)
            #el
            if (utility >= goals[best_goal] and best_utility >= goals[best_goal]) or (utility is best_utility):
                if side_effects > best_side_effects:
                    # print("Both beat goal; judging by side effects")
                    best_action = key
                    best_utility = action_utility(key, best_goal)
                    best_side_effects = action_side_effects(key, best_goal)

            elif utility > best_utility: # if utility beats best_utility
                # print("Updating best_utility based on maximising intended effect")
                best_action = key
                best_utility = action_utility(key, best_goal)
                best_side_effects = action_side_effects(key, best_goal)

# Return the "best action"
return best_action

```

Figure 4: Implemented code for selecting the best action to take.

OOP Code Screenshot

```
class Game:
    def __init__(self):
        self.VERBOSE = True

        # Global goals with initial values
        self.goals = {
            'Eat': 4,
            'Sleep': 3,
        }

        # Global (read-only) actions and effects
        self.actions = {
            'get raw food': { 'Eat': -3, 'Sleep': +2 },
            'get snack': { 'Eat': -1, 'Sleep': 0 },
            'sleep in bed': { 'Sleep': -4, 'Eat': +3 },
            'sleep on sofa': { 'Sleep': -2, 'Eat': +1 }
        }

    def print_actions(self, actions):
        print('ACTIONS:')

        for name, effects in actions.items():
            print(" * [%s]: %s" % (name, str(effects)))

        print()

def main(self):
    players = []
    players.append(AI(self.VERBOSE, self.actions, self.goals, "AI1"))

    for player in players:
        # print(player.actions)
        # print(player.actions.items())

        #for k, v in player.actions.items():
        #    print(k,v)

        print()
        print("Player name: " + player.name)
        print()
        self.print_actions(player.actions)

        player.play_until_all_goals_zero()

    print()
    print('>> All Players Done! <<')
```

```
class AI:
    def __init__(self, verbose, actions, goals, name):
        self.VERBOSE = verbose
        self.actions = actions
```

Figure 5: The code that ended up in the Game class, with the declaration of the AI class at the bottom; any code not in Game was placed in AI, and all code was tweaked as appropriate for object-oriented requirements.

```
print(HR)

input()

# finished
print('>> Done! <<')
```



```
Game().main()
```

Figure 6: The creation of the Game class outside all the classes, and the calling of its method main() to run the program.

What I Found Out

If the intention of the code provided to students was that we should modify it to address all the additional functionality listed in the comments and have the program reach a “Done” state:

- When only one value needed to be evaluated (the effect on the goal being worked towards), the program worked perfectly fine and reached a “Done” state within a few loops.
- When evaluating the intended effect *and* side effects of an action, it was dependent on the magnitude of all the effects of the actions as to whether the program got stuck in a loop where reducing to one goal to 0 required the other goal to be addressed, and so on.
- If multiple goals are to be completed through GOB and SGI in a manner based upon that used in this spike, there need to be actions that will satisfy one goal and not negatively impact others; this is not a concern if the AI is not expected to reach a “Done” state where all goals are completed and the program ends.

With regard to the creation of the object-oriented version of the code:

- Object-oriented programming encourages the division of fields and methods into more manageable segments that themselves can be passed around the code to access their encapsulated data or asked to perform tasks expected of them.
- Object-oriented programs can take advantage of inheritance and polymorphism such that different AIs have the same methods and fields but they’re implemented / populated differently, allowing for a variety of behaviours when the same actions of each sibling class are called by others.
- One downside of object-oriented python code specifically is the requirement of specifying `self.[FIELD NAME]` or `self.[METHOD NAME]` when a class accesses its own fields or methods, something that feels tiresome when other languages, such as C#, don’t require it as it is assumed that the field / method belongs to the class calling it unless otherwise specified.