Dungeon World: a dungeon simulation using GridWorld framework

Jack Valinsky, Johnny Jacobs, Shuyang Liu

April 28, 2017

1 Introduction

Dungeon World is a computer simulation of how an intellectual agent might behave in a preset dungeon environment. The simulation program is built on top of the GridWorld framework [LS12], which provides functionalities such as world knowledge representation and state node generation and searching. By using the GridWorld framework, an agent is created and is able to respond to various stimulus according to its internal states as well as external information in the simulated dungeon.

2 Program Components and Design

2.1 Background Story and Design

At the beginning of the program, the user can find the agent waking in the middle of a mysterious room, which has a door locked. The goal for both of the user and the agent would be exiting the room. Since the door is locked, a sub-goal for the user and agent is finding the key that can let it open the door. However, since the agent is not really goal-oriented when making the decision, the user have to influence its decision-making process by placing new objects during the middle of execution and eventually lead the agent to the door.

The time steps that the agent is allowed for taking actions is limited due to the fact that the room is continuously filling with gas at each step. Therefore, the user have to let the agent to exit the room (i.e. find the key and open the door) as soon as possible.

2.2 World Construction

Although Dungeon World is built using the basic GridWorld framework, we did not follow the exact instructions in the manual[Liu11] for constructing the world. Instead of having a graph-like map as the environment that the agent lives in, we implemented a grid-like map representing the dungeon. A room in the dungeon can be represented as a set of points (with x and y coordinates) and a set of edges connecting each pair of adjacent points. For example, a room can be represented as shown in Figure 1

Each point/vertex is represented as a symbol of the form:

$$< roomname > \& < x - coordinate > \& < y - coordinate >$$

Each edge/road is represented as a list of symbols of the form:

$$'(< point1 > \$ < point2 >< point1 > 1 < point2 >)$$

For example, a point with coordinate (3,3) at main room would be represented as `'MAIN&3&3`; an edge connecting (3,3) and (3,4) would be represented as `'(MAIN&3&3$MAIN&3&4 MAIN&3&3 1 MAIN&3&4)`

In this way, the agent can only move one distance unit at each step, which means that all of the edges are one unit long. The agent can only walk toward the direction that it is currently facing.

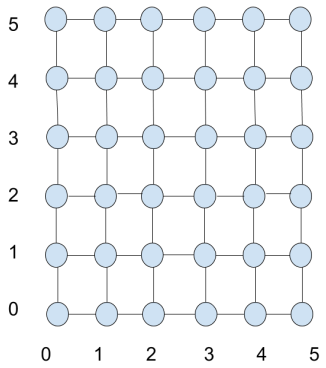

Figure 1: A sample grid room.

If it wants to move to another direction, it would consider turn its direction first (which also costs one time step) and then move to the location.

Since the length of each edge is a constant value, the value (or cost) for taking the action of $Walk$ would only depends on current fatigue level. More specifically, it is currently set to $3 - FatigueLevel$. For more descriptions about $Walk$ and other actions, please see Section 2.3.

2.3 External Object Creations and Action Definitions

Various external objects are created in DungeonWorld. All of the objects are created following the instructions from the GradWorld Manual[Liu11]. Types of objects are defined before objects are instantiated and placed in the map. Each type is associated with several "facts" and "properties" about the object that the agent knows initially.

For example, we defines *Door* with the following predicates:

1. `is_inanimated`
2. `is_accessible`

Later, when the agent is making decisions while seeing a door at adjacent positions, it automatically knows that *Door* is accessible and can attempt to take the action of $OpenDoor$.

Objects can have other objects associated with it. For example a container (such as a $Chest$), can have a list of items placed inside of it. We placed a $Chest$ in DungeonWorld, which has a Key object placed inside of it. Objects that are contained inside have the same location as the container.

Actions are defined in a similar way. There are two versions of definitions for each action. In the model version (used by the agent to plan), each action has a set of pre-defined preconditions, value for taking the action, and expected effects. In the actual version of the action (this is the one executed by GridWorld), it has a set of start conditions and stop conditions. In most of the cases, actions only take one time step. Therefore, for most of the actions, we define the effects of the action as the stop conditions.

Note that actions include both:

- Actions taken explicitly by the agent (such as $Walk$, $OpenDoor$)
- Actions done implicitly while taking other actions (such as See)

We call the second category of actions $ExogenousActions$. The above description only applies to the first category of actions. For more details on exogenous actions, please see Section 2.5.

2.4 Agent's Internal States and Representations

The agent has several internal states that are considered important factors during the decision making process. There are two types of internal states:

- Internal states that are represented in boolean values (either true or false)
- Internal states that are represented in degrees/levels

For example, we represent fatigue in terms of different levels. The fatigue level of the agent can affect the decision of whether or not to take the action of $Walk$. More specifically, the number that we use for representing the level of fatigue is used when calculating the value of taking a $Walk$ action

An example for internal states represented as predicates would be the emotions of the agent. We implemented several different types of emotions for the agent to have: $happiness$, $sastifcation$, and $surprise$. When the agent is interacting with the world, some of the emotions can be triggered based on the experience that the agent has.

Unfortunately we did not implement any functions that can give the agent a sense of personality. Through most of the readings in this course, it seems that many authors agree that personality traits can be very large influences on emotion triggering and decision making. This can be seen as possible extension for DungeonWorld in the future.

2.5 Exogenous Actions

The actions that the agent performs are not always the explicit actions that the agent "decides" to take. In fact, there are another type of action that the agent does automatically. We call this type of action Exogenous Actions.

An example of exogenous action is See in DungeonWorld. While the agent make its decision at each step and perform explicit actions, it also sees the objects around it automatically. It does not "decide" to see the object (unless it decides to close its eye). Therefore, the action of See is not an option during the decision making process. Instead, it "has to" perform this action at every time step. The consequence of having an exogenous action like this is that, the agent can start to have some level of "perceptions" about the world around it without actively making efforts to have feelings.

With some modifications, exogenous actions can also be used for implementing the Dual-processing architecture, which has a conscious process and an unconscious process running concurrently. While the explicit actions are those actions which the agent is aware of consciously, the exogenous actions can be seen as the unconscious process that runs in the background. (Although in this case, the term "action" may not be the best choice of word to describe unconscious processes).

In our current program, only See is implemented as an exogenous action. As possible extensions, it might be a good idea to also have other exogenous actions (such as $Hear$, and $Smell$) implemented in order to give the agent a full set of sensing mechanism.

Exogenous actions are executed by the handleExtOps-Dungeon function in gridworld-exops.lisp. This function is based off the handleExtOps function in the example Gridworld code but modified to run our exogenous actions. This function checks the start conditions, stop conditions, and starredStop conditions for starting and stopping exogenous actions. Starred fields denote conditions in the current world state that could end the action prematurely and how to handle adding and deleting for the action. handleExtOps starts and stops actions by calling the exogenous action's handler function and passing the action and a boolean value. The see action is handled by the handleSee function in gridworld-exops.lisp. Exogenous actions are not handled by the main Gridworld code like normal actions the agent can choose to perform. The handler function for an exogenous action must re-implement variable resolving and function execution in formulas. The see action updates a formula "can see" of all the objects in the current line of sight of the agent, so the handler must resolve variables for position, direction, and objects in front of the agent. The see action calculates the objects in front of the agent using a helper predicate saw? that takes the current position and direction of the agent.

2.6 Decision-Making Process and Goals

For simplicity, we did not modify much in this decision-making part from GridWorld. The original algorithm of forward-chaining is also used in DungeonWorld. By specifying a search beam of 3, with branching factors of 5, 4 and 3, the agent is able to infer the value of the actions based on future actions that they can lead to. Due to limited size of search beam, the agent is not able to see the futures that are very far away from it. This can lead to the consequence of being not able to walk toward the exit door simply because the it cannot anticipate the action of *OpenDoor* in the far future.

Since the original agent in GridWorld was not designed to be goal-oriented, outside of action values, there were not specific functions that take the responsibility of controlling the high-level motivations of the agent. Our current ad hoc method for making accessing through the door as a goal is by assigning a very high value to the action *OpenDoor*. If the agent is able to anticipate this action in the near future, it can certainly "pretend" to be goal-oriented and move toward the goal. However, as mentioned above, due to the limitation of search beam, it is not really able to obtain a long-term goal in our current implementation.

2.7 Graphical User Interface

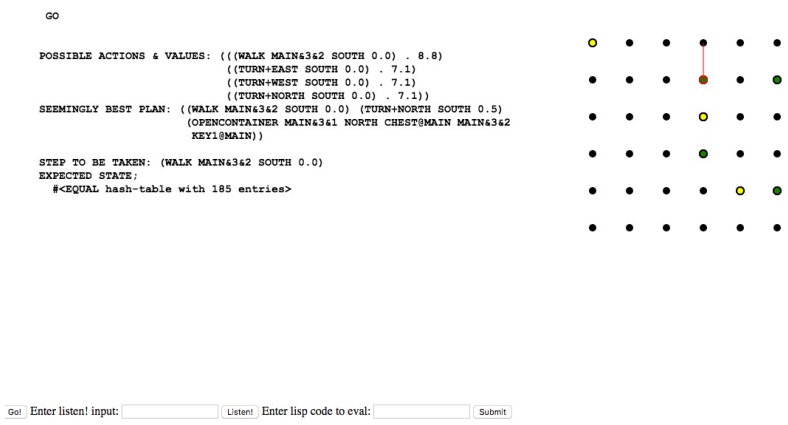

Figure 2: Web GUI for Dungeon World

We implemented a web-based graphical user interface for interacting with Dungeon World (see Fig. 2). It displays the points in the current world which are color coded according to which objects are located at that position. Black means no object, red is for the agent, green means there is an apple at that position, and yellow is the default object color. Currently the GUI can't show multiple objects at a single point, so in order to display where the agent is we color the edge of the point red. The point where the agent is located also has a sight line denoting where the agent is facing. On the left there is a text area that renders the current output of running gridworld from the console. There are three modes of input: a button to call the go! macro, a button and text field to call the listen! macro with user input, and an eval button and text field for submitting arbitrary lisp code to be run. The last form of input is not safe for running on a public server and is meant for debugging purposes only. The eval input does not check if string is valid lisp so there is a high potential in crashing the server if used.

The GUI was implemented as a REST API using the ningle (web framework), yason (json encoding), and clack (http, middleware) libraries for common lisp. The server serves the static files in the public folder and exposes several routes that the frontend uses to render the current world.

3 Limitations and Possible Extensions

Initially we wanted to have multiple rooms, connected by doors and hallways, but the limited memory we had access to precluded even a single large room. Another problem with larger rooms is that the agent can only plan three steps ahead, meaning anything more than 3 squares away is effectively ignored. This can be mitigated somewhat by having intermediate goals with high values, or by placing high value objects to lure the robot towards goals.

Additionally, Garbage Collection crashes the program whenever it runs. This effectively provides a time limitation, as if the GC feels it needs to reclaim memory, the whole program will end.

As further extensions, we wanted to include creatures who could interact with the Agent to provide information or useful items, or even fight with the agent. Additionally, we considered adding items that could also provide information, such as notes, or weapons to be used in fighting the hostile creatures.

A higher-level extension we considered was deeper emotional responses, such as having emotions influence actions directly, instead of being just side-effects. For example, when fighting a creature, high fear could cause the agent to run instead of continuing to fight. In concert with this, we would also like to add emergent emotions, such that more basal feelings such as fatigue or hunger added up to higher-level feelings such as contentment or frustration.

References

[Liu11] Daphne H Liu. Gridworld framework manual. 2011.

[LS12] Daphne H Liu and Lenhart Schubert. An infrastructure for self-motivated, continually planning agents in virtual worlds. 2012.