CS440 Project 3: Search and Destroy

Parth Patel (psp116)

April 2021

Introduction

Implementation of basic agents one, basic agent two and improved agent to solve search and destroy problems for a given map of size d.

Language used: *python*
Libraries required: *random*, *numpy*, *math*, *matplotlib*

Representation

- **Terrain representation**

  A map can have four different type of terrain:

  1. Flat : Represented as a 1
  2. Hilly : Represented as a 2
  3. Forested : Represented as a 3
  4. A maze of caves : Represented as a 4

- **Map representation**

  The map is generated using *numpy*, specifically using the method $full$. The method *make_map* handles the creation of the map and returns a map of given size *size* with cells being assigned a terrain based on respective probabilities.

```
def make_map(size)
```

Initially, the map is a $size * size$ 2D integer array that contains all 0s. Then, the method iterates through the entire map and using $random$, a value between 0 to 1 is calculated. That values works as a probability to determine the terrain type. Each terrain type has an assignment probability of 0.25. Consequently, if the $random$ value falls between $0-0.25$ it will be assigned 1 for $Flat$, $0.25-0.50$ leads to an assignment of 2 for $Hilly$, $0.50 - 0.75$ for 3 as $Forested$ and, lastly, $0.75 - 1$ for 4 as $A\ maze\ of\ caves$.

The last thing that the method does is set the target. Using $random.randint$, two values between 0 to $size - 1$ are generated and used as coordinates to set the target. The target is represented as (10 + $target\ cell's\ terrain\ value$).

Below is a representation of map and representation of what the agents sees. In Figure 2, FL represents Flat, HI represent Hills, FO for Forested and CA for Caves

```
[[ 1  2  1  4  2  4  3  1  3  3]
 [ 2  2  3  4  1  1  1  2  1  3]
 [ 3  4  3  1  4  2  2  4  4  3]
 [ 2  4  3  2  3  2  4  2  3  3]
 [ 3  3  2  3  2  2  4  2  2  4]
 [ 1  3  3  2  1  3  1  1  4  4]
 [ 2  3  4  3  3  3  2  2  1  1]
 [ 4  1  3  4  2  4  2  3  3  3]
 [ 4  2  3  4  3  3  3  3  2  3]
 [ 1 11  3  1  1  2  3  2  2  4]]
```

Figure 1: Actual map representation

FL	HI	FL	CA	HI	CA	FO	FL	FO	FO
HI	HI	FO	CA	FL	FL	FL	HI	FL	FO
FO	CA	FO	FL	CA	HI	HI	CA	CA	FO
HI	CA	FO	HI	FO	HI	CA	HI	FO	FO
FO	FO	HI	FO	HI	HI	CA	HI	HI	CA
FL	FO	FO	HI	FL	FO	FL	FL	CA	CA
HI	FO	CA	FO	FO	FO	HI	HI	FL	FL
CA	FL	FO	CA	HI	CA	HI	FO	FO	FO
CA	HI	FO	CA	FO	FO	FO	FO	HI	FO
FL	CA	FO	FL	FL	HI	FO	HI	HI	CA

Figure 2: What the agents sees

Problems

- **Updating the Bayesian network**

  The agent starts with creating a bayesian network of belief which represents everything observed so far. Therefore, Belief:

$$Belief[Cell_i] \; = \; P\;(Target\ in\ Cell_i \mid Observations\ till\ time\ t) \quad (1)$$

  Let's say we observe a failure in $cell_j$ at time $t+1$, this is how we would update our bayesian network:

$$P\;(Target\ in\ Cell_i \mid Observations_t\ \wedge\ Failure\ in\ Cell_j) \quad (2)$$

  1. $Target\ in\ Cell_i =$ t
  2. $Observations_t =$ o
  3. $Failure\ in\ Cell_j =$ f

$$= \; P\;(t \mid o\ \wedge\ f)\;.\;P\;(o\ \wedge\ f)\;/\;P\;(o\ \wedge\ f)$$

$$= \; P\;(o\ \wedge\ f \mid t)\;.\;P\;(t)\;/\;P\;(o\ \wedge\ f)$$

$$= \; P\;(o \mid t)\;.\;P\;(f \mid t)\;.\;P\;(t)\;/\;P\;(o\ \wedge\ f)$$

$$= \; P\;(o \mid t)\;.\;P\;(f \mid t)\;.\;P\;(t)\;/\;P\;(o)\;.\;P\;(f)$$

$$= \ P \ (t \mid o) \ . \ P \ (f \mid t) \ . \ P \ (o) \ / \ P \ (o) \ . \ P \ (f)$$

$$= \ P \ (t \mid o) \ P \ (f \mid t) \ / \ P \ (f)$$

$$= P \ (Target \ in \ Cell_i \mid Observations_t) \ . \ P \ (Failure \ in \ Cell_j \mid Target \ in \ Cell_i)$$

$$/ \ P \ (Failure \ in \ Cell_j) \tag{3}$$

If i == j:

$$Belief_t(Cell_i) \ . \ FNR(Cell_i) \ / \ P \ (Failure \ in \ Cell_j) \tag{4}$$

if i ! = j:

$$Belief_t(Cell_i) \ . \ 1 \ / \ P \ (Failure \ in \ Cell_j) \tag{5}$$

where FNR = False Negative Rate
and,

$$P \ (Failure \ in \ Cell_j) \tag{6}$$

$$= P \ (Failure \ in \ Cell_j \ \wedge \ Target \ in \ Cell_j)$$

$$+ \ P \ (Failure \ in \ Cell_j \ \wedge \ Target \ not \ in \ Cell_j)$$

$$= P \ (Failure \ in \ Cell_j \mid Target \ in \ Cell_j) \ . \ P \ (Target \ in \ Cell_j)$$

$$+ \ P \ (Failure \ in \ Cell_j \mid Target \ not \ in \ Cell_j) \ . \ P \ (Target \ not \ in \ Cell_j)$$

$$= \ FNR(Cell_j) \ . \ Belief_t(Cell_j) \ + \ 1 \ . \ (1 \ - \ Belief_t(Cell_j))$$

$$= \ FNR(Cell_j) \ . \ Belief_t(Cell_j) \ + \ 1 \ - \ Belief_t(Cell_j) \tag{7}$$

This could be demonstrated using a small example:

Suppose there's a 2x2 map and the agent has a belief system with intial belief of $1/(2 * 2)$ or $1/4$. Then, we observe a $FAILURE$ in cell (0, 0) which is a flat. Then, cell (0, 0)'s new belief:

$$0.1 * (1/4)/((0.1 * (1/4) + (1 - (1/4)))) = 1/31$$

The rest of the cell's new belief:

$$(1/4)/((0.1 * (1/4) + (1 - (1/4)))) = 10/31$$

It is worthy to note that the significance of having the same denominator for both cases is that it allows for normalization of the values. This makes sure that the all the values in the belief system sum to a one.

- **Contains probability**

  Problem number two inquires what is the probability that the target will be found in $Cell_i$ if searched given the belief state at time t. I will refer to this probability as the *contains* probability for ease of understanding. *Contains* probability can be denoted as:

$$P\ (Target\ found\ in\ Cell_i \mid Observations_t) \tag{8}$$

$$= \ P\ (Target\ is\ in\ Cell_i\ \wedge\ SUCCESS\ in\ Cell_i \mid Observations_t)$$

$$= \ P\ (Target\ is\ in\ Cell_i \mid Observations_t)\ .\ P\ (SUCCESS\ in\ Cell_i)$$

$$= \ Belief_t(Cell_i)\ .\ (1\ -\ FNR(Cell_i)) \tag{9}$$

  where FNR = False Negative Rate

  An important observation here is that the *contains* probability changes only if there is a change in the belief state. So, basically, after updating the bayesian network, each cell's *contains* probability would be multiplied by (1 - False Negative Rate).

- **Agent composition**

  All the agents go through the same flow which is basically:

  Pick a cell to search > Travel to that cell > Search the cell > Update beliefs

  The first thing that the agent does is create a $size * size$ array where size is the size of the map. This is the belief state and the initial belief of each cell is $1/(size * size)$.

  The function that handles cell selection has the definition as follows:

```
def find_max_cell(probability_array, size, agent_location)
```

$find\_max\_cell$ takes in the belief array or the *contains* probability array, its size and agent's location on the map. It searches for the cell with the maximum probability and returns it. In case of ties in belief/*contains* probability, it returns the one with the least Manhattan distance from agent's location and in case of ties in distance, a cell is chosen at random.

The agent also calculates its score as follows: Total manhattan distance travelled + number of searches. Subsequently, the agent travels to the cell and searches it. If the search results in $SUCCESS$, the game ends and the score is returned. Otherwise, you observe a $FAILURE$ and the observation is used to update the beliefs. This loop goes on until $SUCCESS$.

- **Basic Agent one**

  Basic agent one uses belief state to solve search and destroy. At every step, it passes the belief state to $find\_max\_cell$ to find the best cell to query.

- **Basic Agent two**

  Basic agent two uses the *contains* probability to solve search and destroy. At every step, it passes the *contains* probability array to $find\_max\_cell$ to find the best cell to query.

- **Basic agent comparison**

  The basic agent one uses the belief state at every point to solve the problem while basic agent two uses the *contains* probability. On average, basic agent two does better than basic agent one. This is partly due to the fact that agent two utilizes the map's terrain data more extensively than agent one.

  But there is a catch. The *contains* probability favors cell's with lower FNR (False Negative Rate). This is because how our *contains* probability is modeled. For every update to the belief state, the *contains* probability updates itself by factoring the cell's belief by its (1-FNR). Given two cells, one flat and another cave, the *contains* probability of former decreases by a factor of 0.9 while the latter's does by a whooping 0.1. That means basic agent two would find targets that are placed in lower FNR cells faster than basic agent one. This is where the analogy of the old joke, given by Professor Cowan, comes into play. There's an old joke about a guy who finds a drunk guy looking for his keys under a street lamp - he stops to help him, but after a while of not finding them says to the guy, are you sure you dropped them here? The guy says no, I dropped them in the park, but the light is better over here. This is good representation of how our model works. The drunk man searching for his keys in well lit areas while avoiding the park overlaps with our basic agent two since it avoids searching in areas with higher FNR even thought there is a good possibility that the target is in there.

So why does our basic agent two perform better? The reason is that after repeatedly searching in lower FNR cells, their respective beliefs get factored down enough for our agent to proceed to other cells.

Below is a bar graph that compares the performance of both the basic agents. Each agent solves the same map of size $50 * 50$ and the score is averaged over 25 runs. There are a total of 5 scenarios simulated: random target assignment, target assigned to a random $FLAT$ cell, target assigned to a random $HILLS$ cell, target assigned to a random $FORESTED$ cell, target assigned to a random $CAVES$ cell. Also, the **lower** the score, the **better** the agent

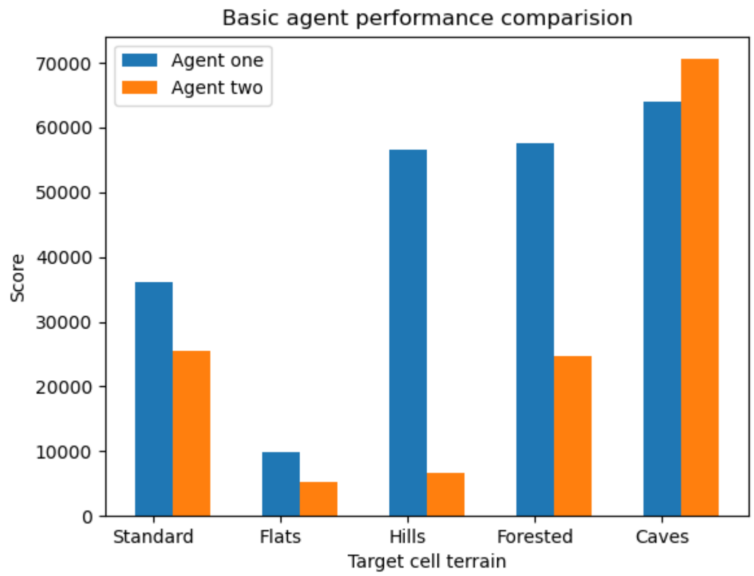

Figure 3: Basic Agent Comparison

This graph further reinforces my claim that basic agent two favors lower FNR cells. Looking at the graph, we can discern that when the target is in a $CAVES$ cell the agent struggles find the

- **Improved Agent**

  The improved agent is built upon basic agent two, therefore, utilizes *contains* probability to guide itself. It exploits the fact that searches increase the score by just one while moving the agent can possibly be quite costly. This helps tackle high FNR. The acronym that I am using for this agent is **T**errain **I**nformation **D**irected **A**gent (TIDA).

  The basic idea is to repeatedly search the cell and update the belief before moving on to choose another cell to query. The number of times we search depends on the cell's terrain type. The equation that decides how many times to search in a given Cell i is as follows:

  $$Number\ of\ searches\ =\ (cell\_type(Cell_i))^2$$

  The agent has access to terrain information of all cells. As per our representation discussed before, 1 is $FLAT$, 2 is $HILL$, 3 is $FORESTED$

and 4 is $CAVE$. We use the following method to know a cell's terrain type:

```
def cell_type(map, coordinates)
```

Below is a bar graph that compares the performance of all the agents. Each agent solves the same map of size $50*50$ and the score is averaged over 25 runs. There are a total of 5 scenarios simulated: random target assignment, target assigned to a random $FLAT$ cell, target assigned to a random $HILLS$ cell, target assigned to a random $FORESTED$ cell, target assigned to a random $CAVES$ cell. Also, the **lower** the score, the **better** the agent

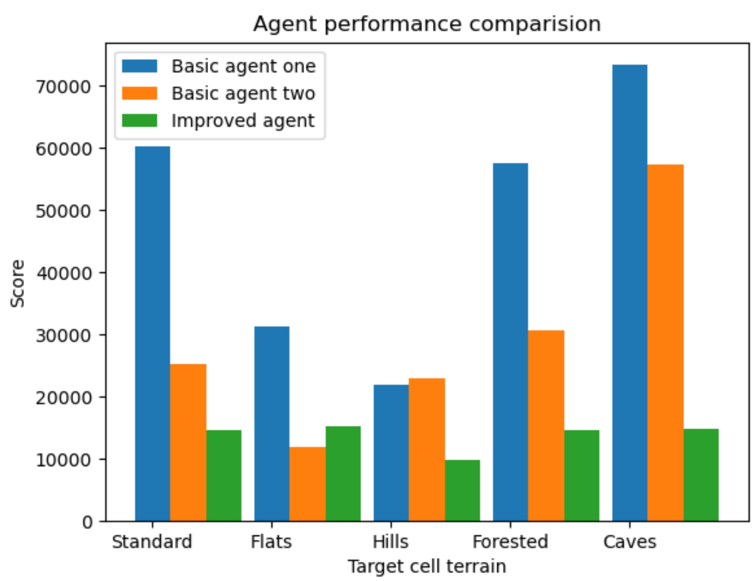

Figure 4: Agents Comparison

It is evident from the graph that **TIDA** does better than both agents.

Academic integrity

- I have read and abided by the rules laid out in the assignment prompt.

- I have not used anyone else's work for my project, my work is only mine.

  Signed by: Parth Patel