Dataiku, Take home project: What are the odds ?

Thibault Desfontaines

November 2019

Contents

1 Introduction

The solution project repository is available at the following github address:
https://github.com/tdesfont/what-are-the-odds.git

It contains the following file organisation:

```
what are the odds
├── helper_classes.py
├── myapp.py
├── solver.py
├── milleniumfalcon.json
├── universe.db
├── README.md
└── templates
    └── home.html
```

I chose to use Flask during this project to develop the web application. Flask is a simple framework for developing web application in development. Being not familiar to the Flask API, it was very straightforward to grasp the main features for our project (loading and reading a json, interacting with the database, handling the HTML templates).

The `milleniumfalcon.json` needs to be available at the start of the application and should contain the path to the database, in our case `universe.db`. At launch, the web application will read both the millenium json and universe.db and will then wait for a POST request to receive the empire.json. After each request, the user needs to refresh the page to see the updated probability to save the galaxy.

All the Flask and web application related features are located in the file `myapp.py`. In addition to this main file, the front end also interacts with the templates folder and the file `home.html` which will be the skeleton of the main page of our application.

As for the back-end, once the json that contains the empire intelligence is received and the database is loaded, the program that actually solves the problem statement is `solver.py`. This file contains the major functions for graph exploration and optimal path finding.

Finally, the `helper_classes.py` file contains two classes (Universe and Path) that will be both useful for object abstraction during the problem solving task.

2 The objects abstraction

The file `helper_classes.py` contains two classes: *Universe* and *Path*.

2.1 The *Universe* class

The **Universe** class is an abstraction that represents the galaxy as a graph and stores the presence of bounty. .

There is one Universe object for the whole simulation. This object stores the routes under the form of the graph and it also stores the bounty under the form of a Python dictionary where the keys are couples (Planet, Day) and the value is 1 if there is a bounty at the given planet location and day.

```python
class Universe:

    def __init__(self, routes, bounty):
        self.routes = routes
        self.bounty = bounty

    def get_destinations(self, source):
        if source not in self.routes:
            return []
        destinations = []
        for dest in self.routes[source]:
            destinations.append(dest)
        return destinations

    def get_travel_time(self, source, target):
        return self.routes[source][target]

    def get_bounty_presence(self, loc, day):
        key = (loc, day)
        if key in self.bounty:
            return self.bounty[key]
        else:
            return 0
```

2.2 The *Path* class

To solve the problem and get the maximal probability to save the galaxy, we will do an exhaustive path search in the graph. The Path class represents a sample path taken by the millennium falcon along the time. Through the methods of the Path class, we can compute all consequences of the actions available to the vessel (wait on the planet and travel to an other planet).

The path taken by the vessel are under constraints. A path is valid if the vessel has a positive or null value of autonomy at each step and if its time duration is lower than the empire countdown.

The following methods represent the different actions available to the vessel:

- `wait()`, Wait and refuel by default on the last planet
- `go_to(destination)`, Travel to planet *destination*

Those methods return boolean values to check that the path is valid or not. Those will be useful during our pruning and stopping criteria strategy in the path exploration.

- `is_out_of_fuel()`
- `is_out_of_time()`
- `has_reached_destination()`

```python
class Path:

    def __init__(self, autonomy, countdown, start, end, universe):
        self.path = [start]
        self.end = end
        self.day = 0
        self.autonomy = autonomy
        self.fuelCap = autonomy
        self.countdown = countdown
        self.universe = universe
        self.n_days_bounty = self.universe.get_bounty_presence(self.path[-1], self.
    day)

```

```
13
14      def wait(self):
15          assert len(self.path) > 0
16          self.autonomy = self.fuelCap
17          self.path += [self.path[-1]]
18          self.day += 1
19          self.n_days_bounty += self.universe.get_bounty_presence(self.path[-1], self.
    day)
20
21      def go_to(self, destination):
22          source = self.path[-1]
23          travel_time = self.universe.get_travel_time(source, destination)
24          self.autonomy -= travel_time
25          self.day += travel_time
26          self.path += [destination]
27          self.n_days_bounty += self.universe.get_bounty_presence(self.path[-1], self.
    day)
28
29      def get_destinations(self):
30          return self.universe.get_destinations(self.path[-1])
31
32      def is_out_of_fuel(self):
33          return self.autonomy < 0
34
35      def is_out_of_time(self):
36          return self.day > self.countdown
37
38      def has_reached_destination(self):
39          return self.path[-1] == self.end
```

3 Algorithmic solution proposal

Assume that we have the following database format (in `universe.db`):

ORIGIN	DESTINATION	TRAVEL TIME
Tatooine	Dagobah	4
Dagobah	Endor	1

When loading this database, we want to build the associated undirected graph as a convenient Python object. The planets will be nodes and the edges will be associated to weights equal to travel time for each (source, destination) pair. Note that each (source, destination) pair implies the presence of a (destination, source) edge in the graph as we will suppose that the graph is undirected.

To store the graph, I have chosen to use dictionary of dictionaries:
`{"Dagobah": {"Tatooine": 4, "Endor": 1}, "Endor": {"Dagobah": 1}, "Tatooine": {"Dagobah": 4}}`

This format enables easy access to the travel time using the following syntax:

```
>>> graph["Dagobah"]["Tatooine"]
 4
```

The proposed approach to solve the problem and compute the odds is an exhaustive search of the optimal path. Of course, this seems like a very brute force approach so we will come up with a pruning and stopping mechanism that will avoid to compute in depth solution that are proven to be sub optimal.

We need to compute the probability the millennium falcon has to save the galaxy before the countdown of the empire. Let's imagine that the vessel needs to go from Tatooine to Endor in 6 days. If a path exists between the two planets and take less than 6 days without any bounty

accross the path, then this path belongs to the optimal set of paths. The probability to save the galaxy is automatically one.

Now let's imagine that we have a path that links in time Tatooine to Endor, but there is k bounties along that path. Then the probability to be captured along that path is $\sum_{i=1}^{k} \frac{9^{i-1}}{10^i} = 1 - \frac{9^k}{10^k}$ and the probability to save the galaxy is a strictly decreasing function of k equals to $\frac{9^k}{10^k}$. The probability to save the universe is the maximum probability along all valid paths that links Tatooine to Endor. We showed that the optimal path in term of maximum of probability is also the path that has the lowest number of bounties encountered among all the valid paths. Also, the number of bounty encountered is increasing along the path. Therefore, if we already found a valid path with 2 bounties on it, and that we are examining an other path that already has 2 bounties on it, we can stop examining this path, because we already found an at most equally optimal path, the current path will never be superior to the already computed valid path.

Using this idea we now translate it in term of code. I used several global variables that will be updated during the computations. Those variables store the solution of the algorithm namely one of the optimal path, its associated probability (through the number of bounties encountered) to save the galaxy. I will also keep the number of examined path to see how depth we went into computations.

```
# Global update variable for optimal search
optimalPath = None  # Best path available
optimalBounty = +float("inf")  # Proba to be captured
n_explored_path = 0
stop_exploration = False
```

To compute the probability of being captured, we first need to get the number of times a bounty has been encountered on the path. We know that if we find a path that routes source to destination with no bounty encountered and in the time limit then we can stop to search alternatives and return this path with a probability of being captured equal to 0.

To control every recursive function call, we use the global variable `stop_exploration` as a boolean to be checked first. It will be updated only if we find a valid path with probability 1 to save the galaxy, in that case, there is no need to search for alternatives.

We then control the pruning of the exploration. We decide to delete the path under examination if one of the following condition is met:

- The millennium falcon will be out of fuel (negative autonomy) after travelling.
- The millennium falcon is out of time, the time taken by the path is strictly greater than the countdown of the empire.
- The path computed has a number of encountered bounties greater than a valid path from source to destination. As the number of encountered bounty is strictly increasing along the path, this path is not worth considering and will never be optimal.

```
    # Pruning
    if path.is_out_of_fuel() or path.is_out_of_time() or path.n_days_bounty >
    optimalBounty:
        n_explored_path += 1
        del path
        return None
```

The update of the optimal path and number of bounty encountered is done if the millennium falcon reached the target planet with a number of bounties encountered lower than the current minimum value of encountered bounties for a full path. In the case where the number of bounty is zero then

this path is optimal and we can stop exploration using the global variable `stop_exploration`.

```
    # Checking Optimal Condition
    if path.has_reached_destination() and path.n_days_bounty < optimalBounty:
        optimalPath = copy.deepcopy(path)
        optimalBounty = path.n_days_bounty
        n_explored_path += 1
        if path.n_days_bounty == 0:
            stop_exploration = True
        del path
        return None
```

If the function has not returned yet, it means that the path being computed is neither worthless considering neither complete and we need to pursue the exploration along this path.

To explore from a path, there are two options:

- Travel to every possible planet from the current planet
- Wait/Refuel on the current planet

```
    ###
    # Continue recursive exploration
    ###

    # Option 1: Travel from the planet
    available_destinations = path.get_destinations()
    for destination in available_destinations:
        new_path = copy.deepcopy(path)
        new_path.go_to(destination)
        dag_explorer(new_path)

    # Option 2: Stay on the planet
    path.wait()
    dag_explorer(path)
```

Here is the full program:

```
optimalPath = None  # Best path available
optimalBounty = +float("inf")  # Proba to be captured
n_explored_path = 0
stop_exploration = False

def dag_explorer(path):

    global optimalPath
    global optimalBounty
    global n_explored_path
    global stop_exploration

    # Stop all exploration in case optimal proba found of being captured is 0
    if stop_exploration:
        del path
        return None

    # Pruning
    if path.is_out_of_fuel() or path.is_out_of_time() or path.n_days_bounty >
    optimalBounty:
        n_explored_path += 1
        del path
        return None

    # Checking Optimality Condition
    if path.has_reached_destination() and path.n_days_bounty < optimalBounty:
        optimalPath = copy.deepcopy(path)
        optimalBounty = path.n_days_bounty
        n_explored_path += 1
        if path.n_days_bounty == 0:
            stop_exploration = True
        del path
        return None

    ###
    # Continue recursive exploration
    ###

    # Option 1: Travel from the planet
    available_destinations = path.get_destinations()
    for destination in available_destinations:
        new_path = copy.deepcopy(path)
        new_path.go_to(destination)
        dag_explorer(new_path)

    # Option 2: Stay on the planet
    path.wait()
    dag_explorer(path)
```

To measure the complexity of this algorithm, we will use the number of nodes in the graph **N** and the **countdown**. Assuming that all the edges are connected to each other, we now that if we have a number of step S in the graph exploration the time complexity can be up to $O(N^M)$ which is exponential. But we also now that the time constraint of the empire countdown bounds the number of steps of at most N steps because the travels from one planet to another takes from 1 to **autonomy** in term of days. So we can bound the time complexity to $O(N^{countdown})$.

We will now present the front-end application and the flask web application.

4 Flask Web Application

Recall the following file structures for our project:

```
what are the odds
├── helper_classes.py
├── myapp.py
├── solver.py
├── milleniumfalcon.json
├── universe.db
├── README.md
└── templates
    └── home.html
```

To prototype and launch the application, I used Flask that I did not know before but seems to be very efficient for development of web applications. The whole logic for the server is condensed in the file `myapp.py`. To start the application, open a terminal in the folder `what_are_the_odds` and launch the app using the following shell command:

```
python3 myapp.py
```

At launch, the main script of the Flask application will load the json with the millenium falcon properties, read the database and build the graph of the routes. Then it will wait for the user to send an HTTP POST request with the json containing the empire intelligence.

```
@app.route('/', methods=['GET', 'POST'])
def index():
    """
    Main file for the front end web application
    :return:
    """
    global proba
    global path
    global n_explored_path
    # Load millenium falcon file as json
    milleniumFalcon = json.load(open("millenium-falcon.json"))
    # Load the db path (e.g. universe.db)
    db_path = milleniumFalcon['routes_db']
    # Build the graph
    graph = graphFromDB(db_path)
    # Load empire intelligence json
    empire = request.get_json(silent=True)
    if empire:
        path, n_explored_path, proba = solver.odds(milleniumFalcon, graph, empire)
    return render_template("home.html", path=path,  percentage=proba*100,
    n_explored_path = n_explored_path)
```

4.1 Sending the empire.json to the web application

To launch the computation of the probability to save the galaxy before the countdown is over, the user can send the json via a HTTP POST request in the following format (curl or Postman are two different ways to send the json to the page):

```
curl --header "Content-Type: application/json" --request POST --data '{ "countdown":
    9,"bounty_hunters": [{ "planet": "Hoth", "day": 6}, { "planet": "Hoth", "day":
    7}, { "planet": "Hoth", "day": 8}]}' http://127.0.0.1:5000/
```

The url `http ://127.0.0.1:5000/` may vary depending on the used port and your localhost address, but the root page is the one to send the request to. After sending the request, the page needs to be refreshed to display the probability to save the galaxy.

5 Some resources I used during this project

During this project, I used several resources listed here:

https://smallbusiness.chron.com/use-sqlite-ubuntu-46774.html

http://flask.palletsprojects.com/en/1.1.x/patterns/sqlite3/

https://stackoverflow.com/questions/20001229/how-to-get-posted-json-in-flask

https://pythonise.com/series/learning-flask/working-with-json-in-flask