Simple Travel Plan by applying CBR

Ishtiak Zaman
Supervised by: Dr. David Leake

May 08, 2015

In this project, case based reasoning is used to get a plan for travelling from one location to another location. There are constraints for time and money that have to be satisfied by our plan. In the domain, we have actions such as travelling by foot, travelling by taxi, travelling by bus, travelling by flying, withdrawing money from bank etc.

The project has a Retriever module that retrieves the best matched case from the case based memory and the Modifier module modifies the retrieved case and adapt to our expected case. We verify that the modified plan really works as expected maintaining all the constraint. All the different parts of the project and the usage of the parts are described below:

Generator:

To avoid cold start problem, we initially have a file *generated_plan.txt* containing the case based memory we previously encountered along with the plan. We populate this file by running the *generator.py* file. We have to specify how many plans we want to generate. We can run the file as follow to generate 100 plans:

```
python generator.py 100
```

This will populate 100 random cases to our case based memory file *generated_plan.txt*. The *generator.py* file first randomly generates an initial state and a goal state. To find the plan between this two states it uses pyhop from the file *simple_travel_example.py*. More about pyhop in here: *https://bitbucket.org/dananau/pyhop*. Here is an extract of the case based file.

```
([
        ([  # Part 1: Initial State defined here
            'loc={\'me\': \'IMU\', \'bus\': \'bus_stop_3\'}',
            'balance_bank={\'me\': 4046.68}',
            'cash={\'me\': 2890.29}',
            'time={\'me\': 2037.3}',
        ]),
        ([  # Part 2: Goal State defined here
            'cash={\'me\': 415.0}',
            'loc={\'me\': \'London\'}'
        ])
    ],
    [   # Part 3: Generated plan defined here
        ('call_taxi', 'me', 'IMU'),
        ('ride_taxi', 'me', 'IMU', 'Chicago'),
        ('pay_cab_driver', 'me', 115.88),
        ('fly', 'me', 'Chicago', 'London'),
        ('pay_air_fare', 'me', 1134.53)
    ]
)
```

This is one case of the case based memory. It has three parts, first part is defining the initial state, second part is defining the goal state and the third part is the plan for achieving the goal state.

Client:

The *client.py* is the starting point of the project. In the file we define the initial state and the goal state for which we want to find a plan. Here is an example of initial state and goal state:

```
# Create the initial state here
currentState = State('current')
currentState.target = 'me'
currentState.loc = {'me':'IU Campus', 'bus':'bus_stop_2'}
currentState.cash = {'me':3050}
currentState.balance_bank = {'me':120}
currentState.time = {'me':1820}
currentState.nearest_bus_stop={'me': 'bus_stop_2'}


# Create the goal state here
goalState = State('goal')
goalState.loc = {'me':'Paris'}
goalState.cash = {'me':170}
goalState.target = 'me'
```

This case is telling us that the object 'me' wants to go to Paris from IU Campus. The object has amount in hand $3050, amount in bank $120, and 1820 minutes to spare. The final plan must take the object to Paris within 1820 minutes and with at least $170 in hand to spare.

After initiating the two states we find the best match case from the retriever and then modifies the retrieved plan by the modifier.

Retriever Module:

Retriever.py: The *Retriever.py* is the main retriever module. The *client.py* asks the retriever to get a best case match for current initial state and goal state. The retriever has access to the case based memory. It loops through all cases and compare with current initial state and goal state to the case from memory and returns the best matched case from the case based memory. The retriever is implemented independent of the problem domain. The retriever uses *state_similarity.txt*, which encodes information about the problem domain and *state_similarity.py*, which is used to calculate the similarity between two states of the problem domain. We can plug different *generated_plan.txt*, *state_similarity.txt* and *state_similarity.py* to the retriever to work for a different problem domain.

state_similarity.txt: This file holds domain related information required for the retriever. Such as min threshold value, similarity matrix, weights of properties. Here is a sample of the *state_similarity.py* file.

```
(
 {'threshold':20},  # Min threshold value
 [
  # Weight of every property
    'loc:30',
    'cash:8',
    'time:50'
 ],
 [
  # The similarity matrix
```

```
    'similarity={\'home\':{\'park\':0.5},
               \'home\':{\'restaurant\':0.3},
               \'London\':{\'Paris\':0.4},
               \'IU Campus\':{\'IMU\':0.94}}'
 ]
)
```
First part holds the minimum threshold. A case has to be at least this amount of similarity to get picked up by the retriever. The second part holds the weights of different properties of a state. We can adjust these weight values to prioritize which property is more important. In the example, similarity of time is most important and similarity of cash is least important. The third part is the similarity matrix of different locations. When we add new features to the problem domain, we will have to add that additional information into this file.

state_similarity.py: This file is used by the retriever to get the similarity between two states. This file use the *state_similarity.txt* to get the domain similarity info (as described above) and apply the rules to calculate the similarity according to those values.

If there is no case found in the case based memory that passes the minimum threshold requires, the retriever returns False.

Modifier:

The *modifier.py* holds the rules of modification we do after a case is retrieved from the retriever. In case, the retriever could not find a matched case, the modifier will run pyhop to get a plan to go from the initial state to the goal state, which is time consuming. Otherwise, there are two different strategies for modification of the case retrieved by the retriever.

Strategy #1: We try to adapt the retrieved case to our case. For example, if we want to go from point A to point D, but the retriever retrieved a plan that goes from point B to point C, we take this plan and replace B with A and C with D to get a plan that goes from point A to point D. After this adaptation we run a test on the new modified plan to see whether the plan really works and satisfies all the constraints such as money constraint, time constraint. If the plan passes the test, we take this plan as the final plan.

Strategy #2: If strategy #1 fails, we go to the second strategy. In this strategy we try to fill up the missing part in the beginning and ending part of the retrieved plan. For example, if we want to go from point A to point D, but the retriever retrieved a plan that goes from point B to point C, we try to find two more plans, one to go from A to B and another one to go from C to D, that will sum up to a plan to go from point A to point D. To find the intermediate two plans, we use pyhop. After this modification we run a test on the new modified plan to see whether the plan really works and satisfies all the constraints such as money constraint, time constraint. If the plan passes the test, we take this plan as the final plan.

In case both strategies fail, we return a partially modified plan we get during working on the strategies.

Miscellaneous:

State.py: This is a state object. Used by all classes when need to create a new state.

static_value.py: Holds defined values for the problem domain. In our domain, this file holds the information of the map. Distances between all the locations, information about bus stop and bus route are encoded in this file. We often load all this info into a state object so that the state object has all the information needed of this domain.

simple_travel_example.py: In this file, we use pyhop libraries to get a plan from initial state to goal state. We have the procedure *travel_by_goal_state* that we call to get a plan for a goal state from a initial state. For any modification of the domain, we add that modification in this file as well so that pyhop plan can generate a correct plan.

pyhop.py: The pyhop library. More about pyhop library here: *https://bitbucket.org/dananau/pyhop*.