CM3038 Artificial Intelligence for Problem Solving

Laboratory #2: Breadth-first Search & Depth-first Search

1. Aims

- To solve the Missionaries & Cannibals problem using breadth-first search with the given `aips` Java/Python library.
- Modify the given Java/Python search library to implement depth-first search.

2. Resources

Do it in Java/Python. Or try both if you want to.

2.1. Java

Download the `cm3038-java-lab02.zip` file from Campus Moodle. The ZIP file contains the followings:

`aips.search`:	This package contains the `aips` uninformed search library.
`aips.lab02.mc`:	This package contains classes for the Missionaries and Cannibals problem.

- Use an IDE of your choice. e.g. Eclipse or Netbeans.
- Copy the root of the Java source (i.e. the “`aips`” folder) into Java source of your project in Eclipse/Netbeans.

2.2. Python

Download the `aips-python-lab02.zip` file from Campus Moodle. The ZIP file contains the followings:

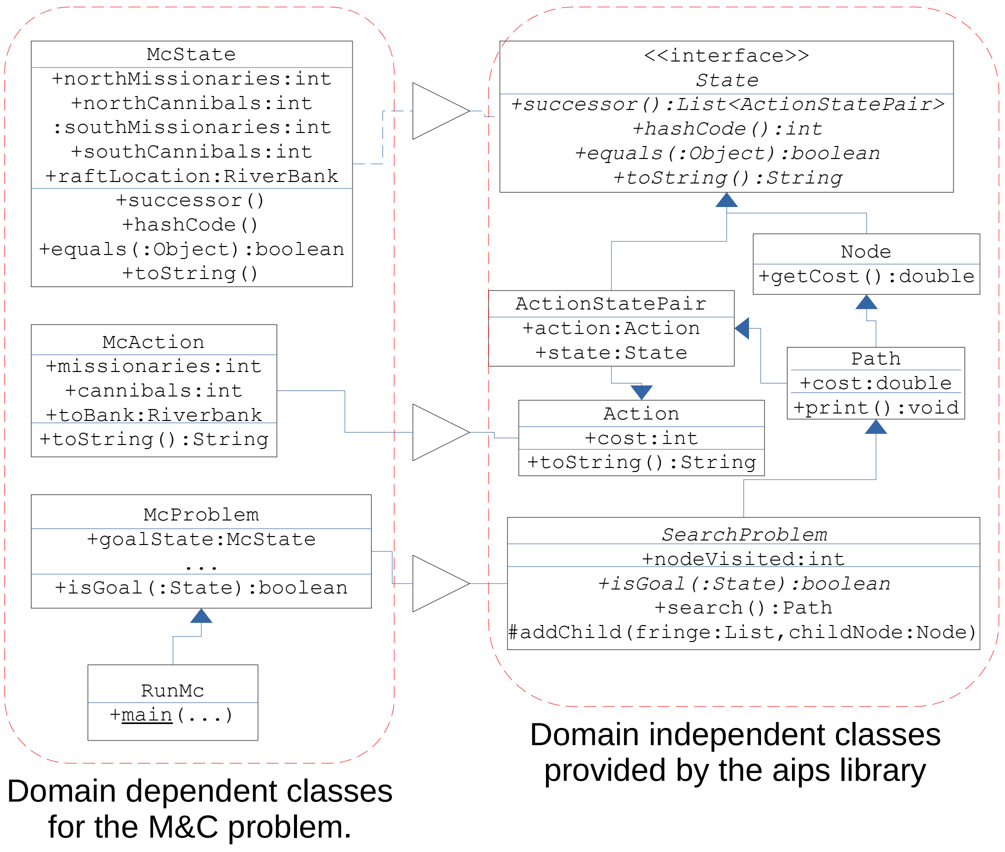
`search.py`:	This Python module implements the `aips` uninformed search library.
`mc.py`:	This file contains classes for the Missionaries and Cannibals problem.

- Use a development environment of your choice.
- Copy the “`aips`” folder into root of your Python source code.

3. Understanding the Domain-independent `aips` Search Library

Search problems share concepts which are common in all domains. On the other hand, there are domain-specific information which is different in each domain, and needs to be tailored. The `aips` library aims to provide reusable, common components across all search problems, while leaving you to provide the domain-specific program code only. The library can also be extended to support different search algorithms.

The following UML class diagram shows how the classes for the M&C problem implement or extend classes in the `aips` library:



For the full documentation of the `aips` Java library, download `aips-api.zip`. For Python, see comments in the Python `search.py` file.

Questions:

- On problem state:
  - Look at the `State` interface/class in the UML class diagram.
    - What are the methods defined on a `State` object?
      - `successor(), hashCode(), equals(…), toString()`
    - Are we sure that the `State` in any problem domain will have these methods?
      - Yes. No matter what problem domain it is, a state must have a `successor()` method that returns all children action-state

pairs. `hashCode()` is use to compute the hash value from the `State` object as we use a hash-table to store the history. `equals()` is needed to compare a `State` with another `State`. `toString()` is used to convert a `State` into a `String` for printing purpose.

- In general, the `hashCode()` method is not a must. However, it controls how objects stored in a hash table will be distributed over the "buckets". A good hashing function will evenly distribute the objects to give a good efficiency when we search in the hash table.

- While these methods may/may not exist in any problem state, can you write the program code of these methods without knowing what problem domain it is?
  - No. All these methods depend on the problem domain and reprentation.

◦ Look at the `McState` class in the UML class diagram.

- How is the world configuration (i.e. state) of the M&C problem modelled in `McState`?
  - The state in the M&C problem is represented by 4 `int` values storing the number of M&C in both sides, plus an enum value telling us where the rather is.
  - Some people may want to use a boolean value for the position of the raft. This is also fine. I use an enum type as it is more natural to define our own values.
- Why do we need to specialise/tailor the inherited methods in `McState`?
  - Because all these methods depend on our state representation. In different problem domains, the program code in these methods will be different.
  - e.g. Now we know there are 4 `int` values and an enum in `McState`, we can write `toString()` to compose and return a `String` that shows our world configuration.
  - If you are modelling the Monkey-and-Banana problem, values in your state will be different, and the program code in all these methods will be different.
- If you want to check if you have reached a goal state, how will you do it in Java/Python using the library?
  - In Java, you can invoke the `equals(…)` method of the state:

```
State stateA,goalState;
…
if (stateA.equals(goalState) ...
```

  - In Python, the `__eq__(...)` function of `McState` defines how the "==" equality operator is handle.

```
if stateA==goalState:
...
```

- Which method you have to program to specify the game rules in a specific search problem? (i.e. getting all valid actions and subsequent states)
  - You will need to modify the `successor()` method in `McState`:

        - From the current state, find out all valid actions based on the game rules.
        - For each valid action, create a new "next state" by applying the action to the current state.
            - Note: You DO NOT modify the current state directly. Instead you should have both the current state object and next state object for each action. Remember that we have a search tree. Each node in the search node points to/reference the state at this point of the search.
- On action:
    - Look at the `Action` class in the UML class diagram.
        - What are common across all actions in different problem domains? How are these captured/modelled in the `Action` class?
            - All action in any problem domain has a cost.
            - This is modelled as a `cost` attribute in `Action`.
            - However, we don't know what value `cost` has until we know which problem domain it is. So we assume a value of 1.0 which effectively counting the number of actions.
        - In different problems and actions, the action cost may be different. How do you assign the correct action cost in a Java/Python program?
            - When you have the domain-specific subclass that extends Action, you can simply assign the action cost value to the attribute. For example:
              `McAction myAction=new McAction(…);`
              `myAction.cost=5.4;`
            - Note that in Java, you cannot create an `Action` object directly as `Action` is an abstract class. You can only create objects of a concrete class. e.g. `McAction`.
    - Look at the `McAction` class in the UML class diagram.
        - What information is need to tell us about the action in the M&C? How is this modelled in `McAction`?
            - There are 2 `int` variables telling us how many M and C are being moved in this action, plus an enum value telling us where the raft is moving to.
        - We do not declare a `cost` attribute in `McAction`. Does `McAction` has a `cost` attribute or not?
            - Yes. `McAction` is a subclass of `Action`. It inherits the `cost` attribute from its superclass.
            - A common mistake is re-declaring the `cost` attribute in the `McAction` subclass. You MUST NOT do this.
        - If you want to customise the way a `McAction` is printed on the screen, which method should you change?
            - In Java, modify the `toString()` method which composes and returns the `McAction` as a `String`.
            - In Python, you modify the `__str__(…)` function.
- On search problem:
    - Look at the `SearchProblem` class in the UML class diagram.
        - What methods are defined in `SearchProblem`? Do these methods exist in all problem domains?

        - `isGoal(…)`, `search()`, and `addChild(…)`
        - These methods must exist in all search problems, despite the specific domain.
        - Read the Java API document to see what these methods do.
        - For Python, read the comments in `search.py`.
        - Can you program these methods without knowing what problem domain it is?
            - You cannot, as these methods are all domain-dependent. This is the reason why they are defined as abstract methods in `SearchProblem`.
    - Why don't we declare a `goalState` attribute in `SearchProblem`?
        - Because not all search problems can have an explicit goal state defined. Some domains use a goal test to compare the current state against some rules rather than an explicit goal state.
        - e.g. As far as you parking your car within the boundary of a parking space, you have reach the "goal".
        - There is an infinite number of position and orientation within the boundary of a parking space to be consider a goal. You cannot explicitly list out all possible goal states in this domain.
    - After a search:
        - How do you get the solution found (if any)?
            - Invoke the `search()` method of the `SearchProblem` object, which returns a `Path` object.
            - If a solution is found, the returned value from search is a non-`null` value.
            - Again, see the Java API document for full details.
        - How can you find the cost of the solution?
            - The `Path` object has a `cost` attribute. This is shown in the UML diagram.
        - How can you find the number of nodes explored?
            - After invoking `search()`, the number of nods explored is stored in the `nodeVisited` attribute of the `SearchProblem`.
            - Note that `nodeVisited` is not reset between multiple calls to `search()`. Manual resetting before the next call is needed.
    - The `addChild(...)` method of a `SearchProblem` controls how a new child `Node` is added into the fringe. The default strategy is breadth-first search.
        - In breadth-first search, where is a new child `Node` added into the fringe?
            - The end of the fringe (list/queue).
        - If you want to use depth-first search, what do you need to do?
            - Put the new child into the beginning of the fringe rather than the end.
- Look at the `McProblem` class in the UML class diagram.
    - Why do we need to declare a `goalState` attribute here?

        - Because the M&C problem uses an explicit goal state rather than testing rules, and the `SearchProblem` superclass does not have such attribute.
      - Why do we need to customise the `isGoal()` method here?
        - Yes, because `isGoal()` is an abstract method in the superclass.
        - You can see why `isGoal()` is defined as abstract in `SearchProblem` because if a domain uses goal testing rules, the rules are domain-dependent and need to be customised.
      - Do we need to modify the `addChild(…)` method in the `McProblem` class? Why?
        - No, because the `addChild(…)` method inherited gives a BFS strategy. You don't need to change it in `McProblem` if BFS is what you want to do.
- On node in the search tree:
  - Look at the `Node` class in the UML class diagram.
    - What method(s) does the `Node` class have?
      - The `getCost()` which returns the total cost from the root of the search tree to this node.
      - Note: There are other methods defined in Node but I am not showing them in the UML.
    - Do we need to modify the `Node` class in different problem domains?
      - No. You will see that the behaviour of `Node` is universal across all problem domains.
- Look at the `RunMc` class in the UML diagram.
  - There is a `main(…)` method in `RunMc`. Do we need a `main(…)` method in other classes? Why?
    - No, because you only need the `main(…)` method in the class that acts as the starting point of your program. Other classes are used to model the problem. They do not serve as starting points.
  - What do you need to do in the `main(…)` method to find the solution of a `McProblem` problem?
    - Create the initial state and goal state.
      Use them to create a `McProblem` object.
    - Then invoke `search()` on the `McProblem` object.
    - What is the returned result? How you do print the solution?
      - The returned value from `search()` is either `null` if no solution is found, or a `Path` object there is a solution.
      - Use the `print()` method of the `Path` object to print the solution path.
- If you are going to use the `aips` library to solve a different problem (e.g. Monkey-and-Banana), what are the steps involved?
  - Create your own set of subclasses from the library:
    - Create an `Action` subclass to model the actions in your domain.
      - Have suitable attributes in this subclass to capture information in an action.
      - Assign the correct `cost` attribute value based on the action.
    - Create a class that implements the `State` interface to model the state.

    - Have suitable attributes in this subclass to capture information in your problem's world configuration.
    - Customise `toString()` return your state as a `String`.
    - Customise `equals(…)` to check if your state equals to another object.
    - Customise `successor()` to return all valid actions from the current state, and the corresponding next state. Return all these as a list of action-state pair.
    - Optionally, customise the `hashCode()` method to return the hash value of your state as an `int`.
  - Create a `SearchProblem` subclass to model your search problem.
    - You may need an attribute to store the goal state here.
    - Customise the `isGoal(…)` method. In most domains it will be checking if a given state equals to the goal state.
  - Finally, in the main class' `main()` method, create an object from your `SearchProblem` subclass (defined above) with an initial and goal state. Invoke the `search()` method of the problem object.

4. Solving the Missionaries & Cannibals Problem Using Breadth-first Search

Classes in the `aips.lab02.mc` Java package (or `mc` Python module) implement the "Missionaries and Cannibals" problem using the `aips` search library. The following classes are developed to model the M&C domain:

Java Interface/Class	**Description**
`aips.lab02.mc.McState`	This class implements the `aips.search.State` interface to represent a problem state in the M&C domain. Examine `McState.java` to see how the problem state is modelled.
`aips.lab02.mc.McAction`	This class extends the `aips.search.Action` class to represent an action in the M&C domain. Examine `McAction.java` to see how an action is modelled.
`aips.lab02.mc.McProblem`	This subclass of `aips.search.SearchProblem` defines a specialised M&C search problem.
`aips.lab02.mc.RiverBank`	This enumerated type models the north and south river bank in the M&C domain. We have attributes/ variables using this data type when we need to represent the location of the raft. This data type is also use in the `McAction` class to tell where the raft is going. • The 2 river banks are north and south. It is better to model them as enumerated types instead of integer values. • The worst idea is to model them as String!
`aips.lab02.mc.RunMc`	This is the main class that creates a M&C problem and searches for its solution.

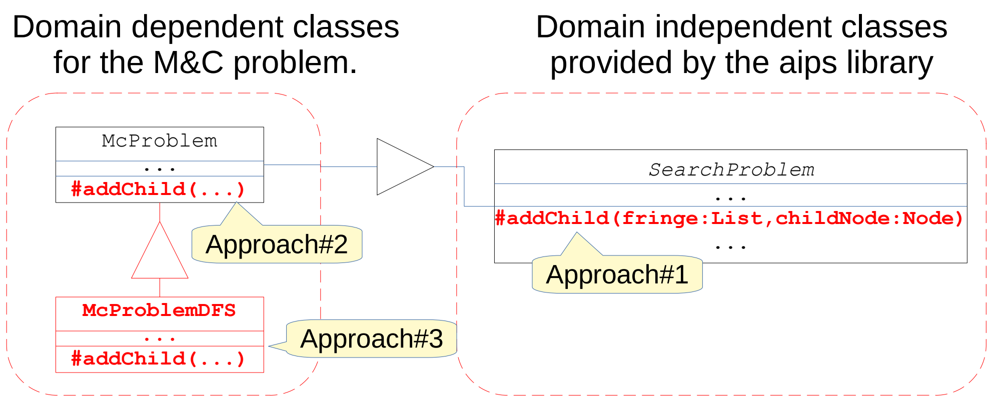
Corresponding Python classes are defined in the `mc.py` file.

- Run `aips.lab02.mc.RunMc` (Java), or `mc.py` (Python).
- Examine the solution. How does it compare to the one you worked out manually in Lab#01?
  - The program may explore the search tree in a different order from your manual solution. However, all feasible actions and dead-ends found should be the same as your hand-drawn tree.
- The default search strategy is breadth-first. Examine the cost reported. Do you think this is the lowest cost?
  - Breadth-first should find you the shallowest goal state which is closest to the root node. As we are taking each move as 1 cost unit, this should be the solution with the lowest cost.
- How many number of nodes are visited before the goal is found? Is it a reasonable figure? (i.e. at this length of the path, is this reasonable?)
  - My program has visited 31 nodes to give a solution of 11 steps. This is the default breadth-first search behaviour.

5. Implementing Depth-first Search

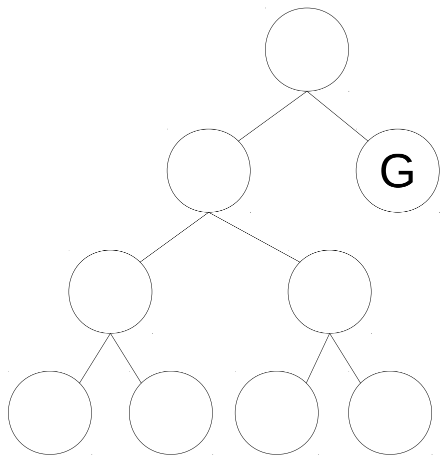
The `addChild(List<Node>,Node)` method in the `SearchProblem` class adds a new child node into the fringe (the list of `Node` to be expanded). The default method adds the child node to the end of the fringe, giving a breadth-first behaviour (BFS). By changing the way a new child node is added into the fringe, you can change the search strategy into depth-first search (DFS).

- Read about the `addChild(...)` method in the `SearchProblem` class.
  - For Java, you can find it in the API document of the `SearchProblem` class.
  - In Python, see the `SearchProblem` class in `search.py`.
- To do DFS, we need to change the `addChild(...)` method in the search problem. There are at least 3 ways to do it:


Domain dependent classes for the M&C problem.
Domain independent classes provided by the aips library
McProblem
...
#addChild(...)
Approach#2
McProblemDFS
...
#addChild(...)
Approach#3
SearchProblem
...
#addChild(fringe:List,childNode:Node)
...
Approach#1

  - Approach 1:
    - You can edit `SearchProblem.java` in the library to modify the `addChild(...)` method. **However, this is undesirable**, as you are not supposed to modify the library code, and it also removes the BFS strategy from the library.
  - Approach 2:
    - You can edit the `McProblem` class to add the `addChild(...)` method, overriding the inherited method.
    - If you use this approach, you will use only DFS in the M&C problem.
  - Approach 3:
    - You can create a subclass of `McProblem` (e.g. `McProblemDFS`) and override the `addChild(…)` method in this subclass.
    - If you use this approach, you can choose to use BFS or DFS to solve the M&C problem, depending on which problem class you use.
  - Hint:
    - In all approaches, you need to control where to put the new child `Node` into the fringe list which is a `List` of `Node` objects.
      - For Java, read the API document of `java.util.List` to see the methods available on a `List` object.

            - For Python, do a Google search to see how you can add an element to a specific position of a list.
- Run the modified program and compare the result.
    - Has the solution changed?
        - Interesting enough, depth-first search still give a solution of 11 steps. However, does it mean the 2 solutions are the same? Honestly we don't know until we compare the 2 solutions (1 from BFS, 1 from DFS) and see if all actions taken are the same. Remember that you may have different paths leading from the initial state to the goal state. Only if the 2 paths are exactly the same then the 2 solutions are the same.
    - Has the cost changed? Why? Is this expected?
        - Depth-first search still gives a solution of 11 steps. We are lucky that it is not worse than the optimal solution found by BFS. Maybe there is no longer/more expensive solution.
        - In general, we are not surprised if DFS finds a more expensive solution reported by BFS. Because BFS is optimal, the cost of 11 is the cheapest. You won't be able to find a cheaper solution than this. (However, you may be able to find a different solution which also costs 11 steps.)
        - Another factor that affects the search space is the order of the children when they are added into the fringe.
            - e.g. if N has children ABC, are we adding them as ABC or CBA or ...? This greatly affects the performance of depth-first search. Currently we are doing it in an arbitrary order.
        - Also, our search library is checking for repeating states. So it does not re-explore a node that it has already visited. The helps to avoid loops in searching for a solution.
    - How about the number of nodes visited in DFS? How does it compare to BFS?
        - The number of nodes by DFS visited is 27, compare to the 31 in BFS. This is expected as DFS does not explore all nodes in a level before proceeding to the next level. However, there is no guarantee that DFS will always explore fewer nodes than BFS.
        - In some situations, DFS may actually explore more nodes than BFS. Remember in our worst case analysis, BFS's time complexity is $O(b^d)$ where d is the depth of the goal. DFS's time complexity is $O(b^{m+1})$ where m is the maximum depth that DFS has reached. If m and d are close, then the difference in performance between the 2 algorithms will be small, and I guess this is the case in our M&C problem here (only a 4-node difference). If m and d differs greatly, then you will expect DFS to explore many more nodes than BFS. i.e. DFS wasted a long of time in the fruitless subtree before it comes back from the depth hitting a shallower goal. This is illustrated in the following example where m=3 and d=1. DFS needs to explore 8 nodes to find G, while BFS only need to explore 2:



        - That means when it comes to the no. of nodes explored (to get to the goal), and which goal you will get (the cost of the solution found), it is quite a matter of luck for DFS. It depends very much on the structure of the tree/search space.

6. General Questions

- Why do you think we ask you to use a given library? What do you think will happen if students are to develop the program from scratch?
    - I expect the followings will happen:
        - Case 1: Students can develop a domain-independent library like `aips`. This will be ideal but is very unlikely to happen.
        - Case 2: Students can develop a domain-specific program that works with the M&C problem only. This is also good.
        - Case 3: People cannot get the basic search algorithm to work, and get stuck. As a result, they fail to learn anything from any (future) lab exercise. I am sure this will happen for some.
        - Case 4: People go to Google/Github to download program code, defeating the whole purpose of the lab. This is the most likely outcome.
    - Now you may understand why I am forcing you to use my library:
        - Reason 1: To make sure that everyone gets a starting point. (Although I must emphasise that the basic algorithm is quite simple. I expect every student in the degree year to be able to program it from scratch.)
        - Reason 2: To stop people from Googling a solution. Most solutions you can find on the Internet are bound to a problem rather than using a domain-independent design like `aips`.
    - Some previous students wondered (or "complained) why we don't use a "standard library" here. The answer: There is no need for a standard as these algorithms are quite easy to program. People tend to create reusable libraries for complex but recurring tasks, so that they can focus on other more important issues. This is obviously not the case here.

- Finally, if you really want to program from scratch, just try it. However, in the coursework you must use the `aips` library.