G54DIA final report

Name: **Zongqi Wang**

Email address: psyzw1@nottingham.ac.uk

Student ID: **4143772**

2012/3/20

G54DIA final report

1. Introduction

This is the final report for the G54DIA, which will specifically describe the agent that has been successfully implemented to solve water delivering problem. To read this report, it is assumed that the reader had finished reading the relative interim report, which described the problem, the task environment and the software design for a single agent. To facilitate reading, the Section 2 will briefly recap the software design; In Section 3, the problem will be systematically analyzed; After reasoning about the problem, the Section 4 will introduce the algorithms; the software implementation will be explained in detail in Section 5; the single agent will be evaluated critically in Section 6; Section 7 will introduce the multi-agent version and will be evaluated; Finally, there will be a conclusion at the end of this report.

2. Software design for a single agent

Considering that the task for the agent is relatively complex, a deliberative architecture may be one of the feasible choices to complete the task.

2.1. High level

Due to the limitations of reactive architecture, more specifically: the lack of abilities to produce intelligent actions in a relatively large and complex environment, a deliberative architecture, which could perform deliberations over various choices of courses of actions subject to environment constraints, is considered as an alternative wise choice in this coursework.

- **Approach: problem solving (search)**

A search problem consists of a set of states, a set of operators and the state space.

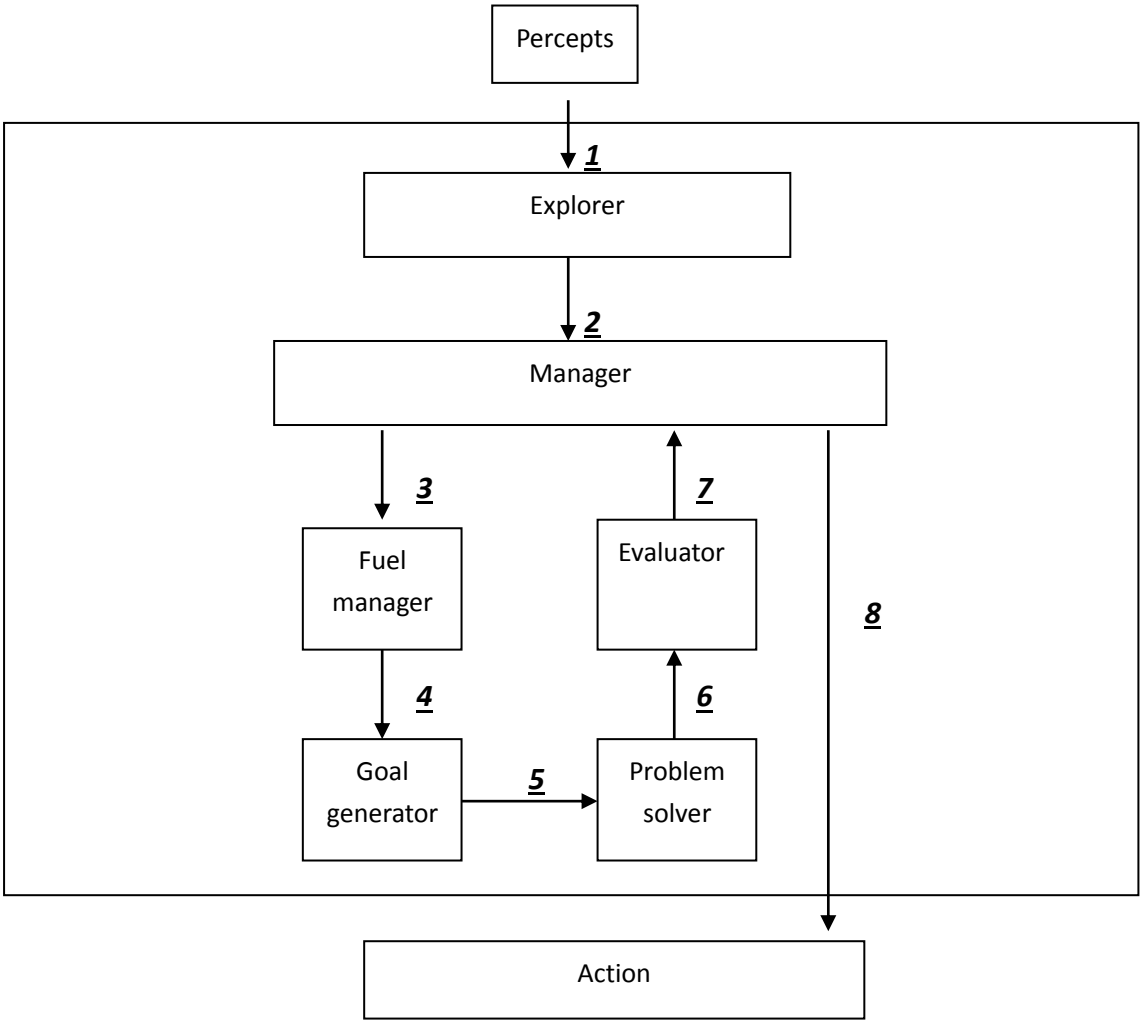
In order to make the design more flexible, the agent will be decomposed into several parts, with each of which has differing but co-operative functionalities. This approach may also facilitate the development of the agent because of adapting object-oriented thinking strategy. Note that, all functionalities listed here might be compromised in later implementation.
Agent components and operators:

1. Fuel manager:
    a) Description: manage the fuel.
    b) Functionality: ensure the problem of “out of fuel” would never occur; compute the cost of the amount fuel from a specific state to another.
2. Explorer:
    a) Description: explore the world and manage the world model according to what are explored.
    b) Functionality: explore world; gather information; manage world model.
3. Goal generator:

a) Description: automatically generate abstract goals for problem solver.
b) Functionality: generate abstract goals.

4. Problem solver:
   a) Description: given a goal, produce a sequence of fundamental actions.
   b) Functionality: produce a sequence of actions to realize a goal.
5. Evaluator:
   a) Description: evaluate the solution generated by the solver.
   b) Functionality: quantified evaluation of a solution.
6. Manager:
   a) Description: manage the inner state of the agent, make decisions of actions.
   b) Functionality: manage the state of the agent; decide which action to perform.

2.2. View of Process:



1. Explorer analyzes the environment and updates the model.
2. Manager analyzes and updates state.
3. Manager triggers fuel manager to generate fuel restrictions.
4. Fuel manager passes the fuel constraints to goal generator.
5. Generate appropriate goals and pass to solver.

6. Produce solutions to meet the goal.
7. Pass the evaluation of generated solutions to the manger for decision.
8. Choose an action to perform and update state.

3. Problem analysis

3.1. Purpose

As introduced in the coursework, the purpose is to deliver as much water to as many stations as possible. The evaluation formula is given by: (the amount of delivered water) *(the number of completed tasks).

3.2. Constraints

Some vital constraints from agent:

- Max Fuel Load: the maximum fuel level is 100.
- Max Water Load: the maximum water level is 10000;

Some important constraints from task environment:

- Limited view: the agent could only see 12 units in each of 8 directions from itself.
- Generated Task: the task is randomly generated with random amount of water requirement.
- Locations of Wells and Stations: the locations are unpredictable.
- Fixed refuel point: the fuel pump is only available in the centre point of the environment.

3.3. Reasoning

The only place, which is the centre point of the environment, for the agent to get refueled is the fuel pump. This indicates that: the possibly most duplicated state is the state where the agent is located at the fuel pump and performs actions to refuel.
To get score, the agent should try to complete tasks by providing water. This means the agent should try to deliver as much water as possible to complete tasks and therefore get higher score. However, the amount of water for an agent to hold at any time has an upper bound of 10000. This indicates that the agent could not load water if the water is full. Hence, it is reasonable to state that: in order to get higher score, the agent should try to load as much water as possible. This is reasonable because of the existence of the upper bound of water level.

3.4. Conclusions

Considering the facts described in (Section3.3), and the fact that the only way to decrease the water in agent is to deliver water to stations to complete tasks, it is therefore reasonable to state that: in order to get high score, the agent should perform at most 100 basic movements and

some other actions, such as load water and deliver water, to load as much water as possible and finally come back to fuel pump to get refueled. By repeating this strategy, the agent may perform a sequence of reasonably satisfied actions and will possibly be highly evaluated.

4. Algorithm

4.1. Shortest path

Input: A set of places to visit.
Output: A sorted sequence of places, which cost least to visit them all once.

Places ShortestPah(Places)
  place_shortest <- copy Places
  Do
    places_new ← permuate Places
    If (distance(places_ new) < distance(place_shortest))
      Then place_shortest ← places_ new
    End if
  Until (no more permutation for Places)

This algorithm ensures to produce a shortest path for a given set; however, the computational time grows more than exponential. Therefore, this algorithm is only used for a relatively small set. More detailed explanation is available in the Section 5.

5. Software implementation for single agent

5.1. Introduction

Java is chosen as the fundamental programming language in this software implementation. The software design is mostly kept in this implementation, therefore if you encountered unfamiliar concepts in this Section, please read Section 2 again. The section 5.2 will present the data flow to you, aiming to facilitate the understanding of the implementation. Also, the Section 5.3 will give a brief introduction to the classes and structure of implementation with relation to the software design. The rest of the each sub-section in 5.3 is structured to explain the implementation choices.
In Section 5.3, all classes and methods would be described. The implemented code would be in the appendix. All those code are well organized and with plenty of comments to help understanding. The section 5.3 would only explain the algorithm of some relatively complex

codes, instead of trying to explain all implemented code line by line.

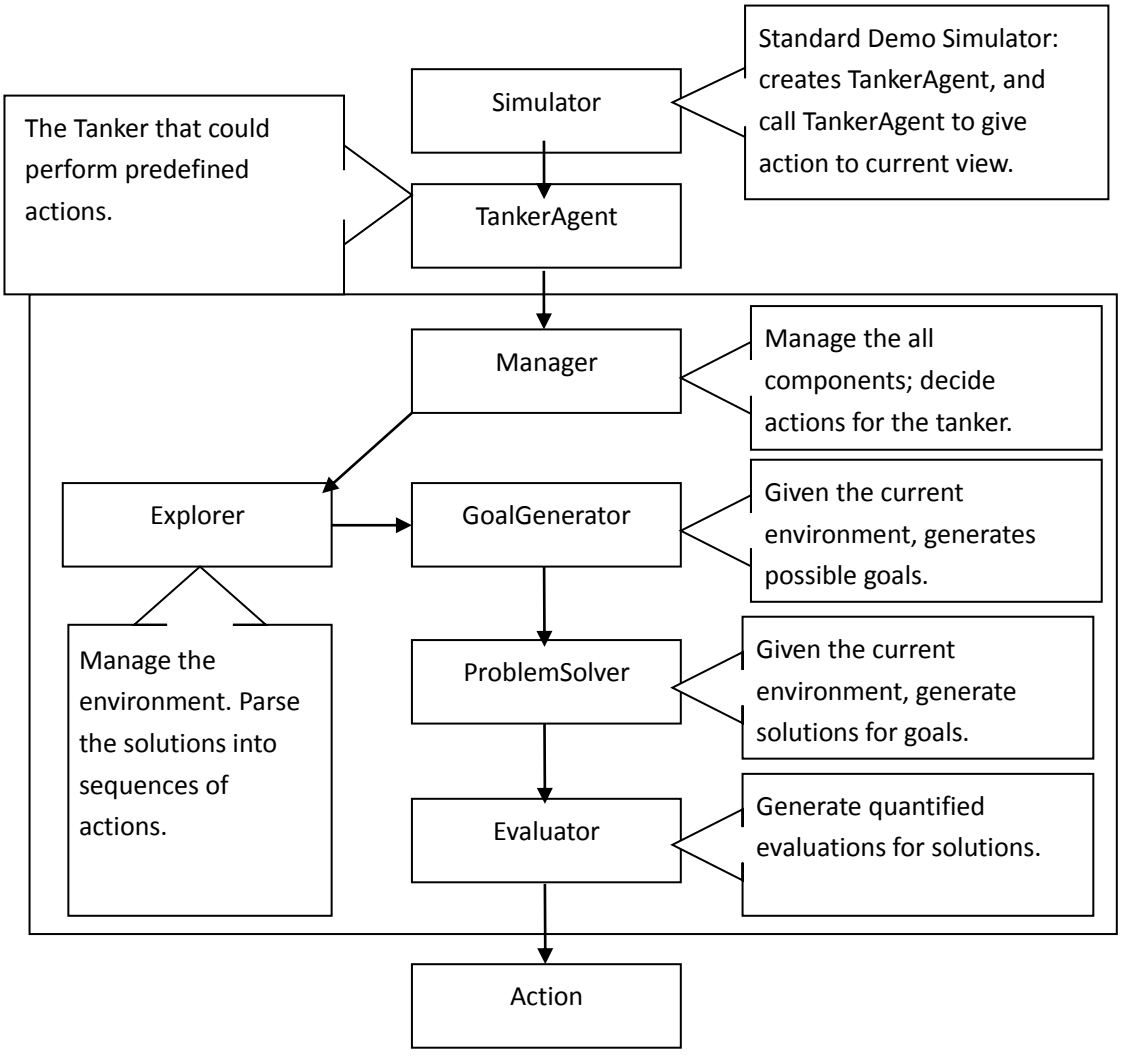
5.2. Overview of data flow

5.2.1. Components introduction

Notice that the FuelManager is implemented implicitly: embedded into manager to give fuel control over all components.

1. This is the main components diagram:

The actual implemented code would be presented and explained in Section 5.3.



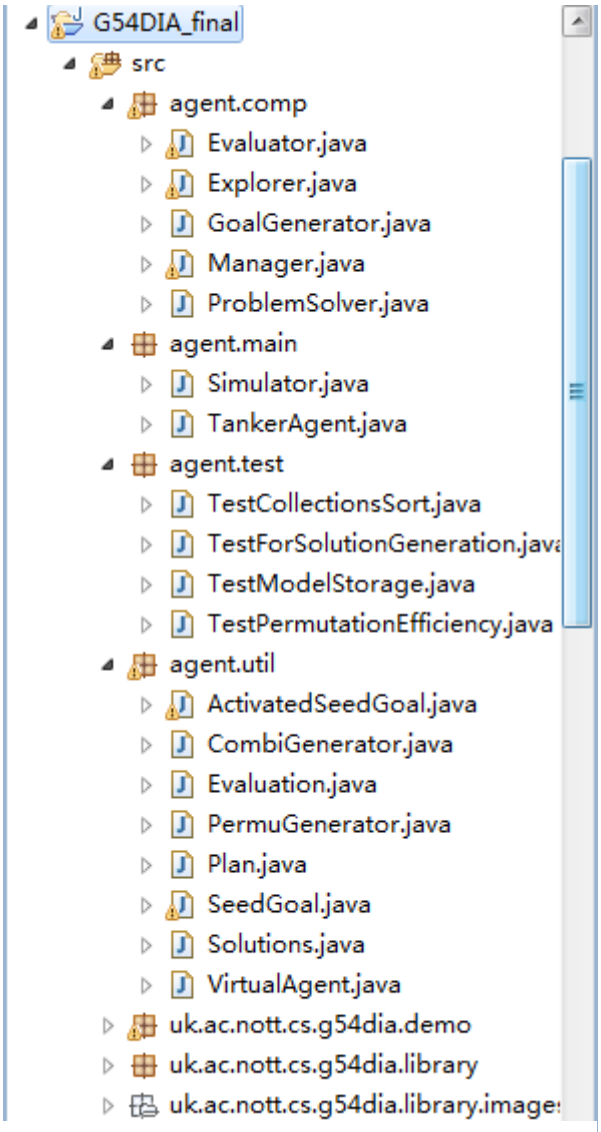
5.2.2. Data flow

1. The Simulator creates the TankerAgent and triggers the environment to tick. After each time step, the Simulator will call TankerAgent to give an action and tries to perform that action.
2. The TankerAgent received the tank view from Simulator and asks the manager to give an action.
3. The manager would firstly provide the Explorer the current view of tanker for update.
4. Once the Explorer completed information update, the manager would ask the GoalGenerator to produce goals.
5. The goals would then be passed to ProblemSolver to generate solutions.
6. These solutions would be evaluated by Evaluator.
7. Then, the manager would pick one of the best evaluated solutions and requires Explorer to parse the solution into a plan.
8. Finally, when the plan is finished, the manager would execute the plan by giving actions in sequence.

Notice that a lot of concepts are introduced without explanation, e.g. Solution, Plan. These concepts would be explained in detail in the next section with implemented code.

5.3. Classes and Code Implementation

Picture5.2_1 is the overview of the classes and packages. The development environment is [JavaSE-1.7]; the development tool is Standard Eclipse.



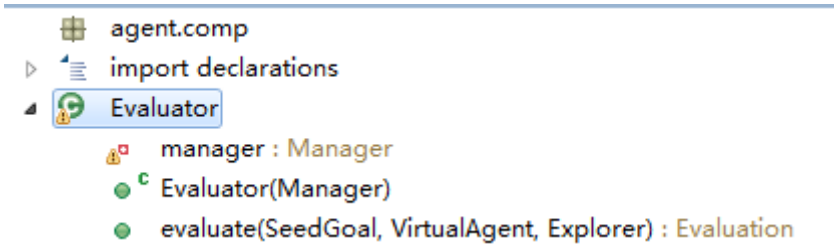
(Picture5.2_1: Classes and packages over view)

5.3.1. agent.comp package

This package contains all implemented components that have been introduced in Section 2 except the FuelManager, which is implemented implicitly, in other words: the FuelManager has been embedded into the Manager, controlling and managing the fuel issues. More detailed information is available in the following sub-sections.

5.3.1.1. Evaluator

- Overview of class outline



5.3.1.1-1Evaluator

The picture (5.3.1.1.1-1Evaluator) is the class outline.
The main task for the Evaluator is: given a solution, generate the evaluation for that solution.

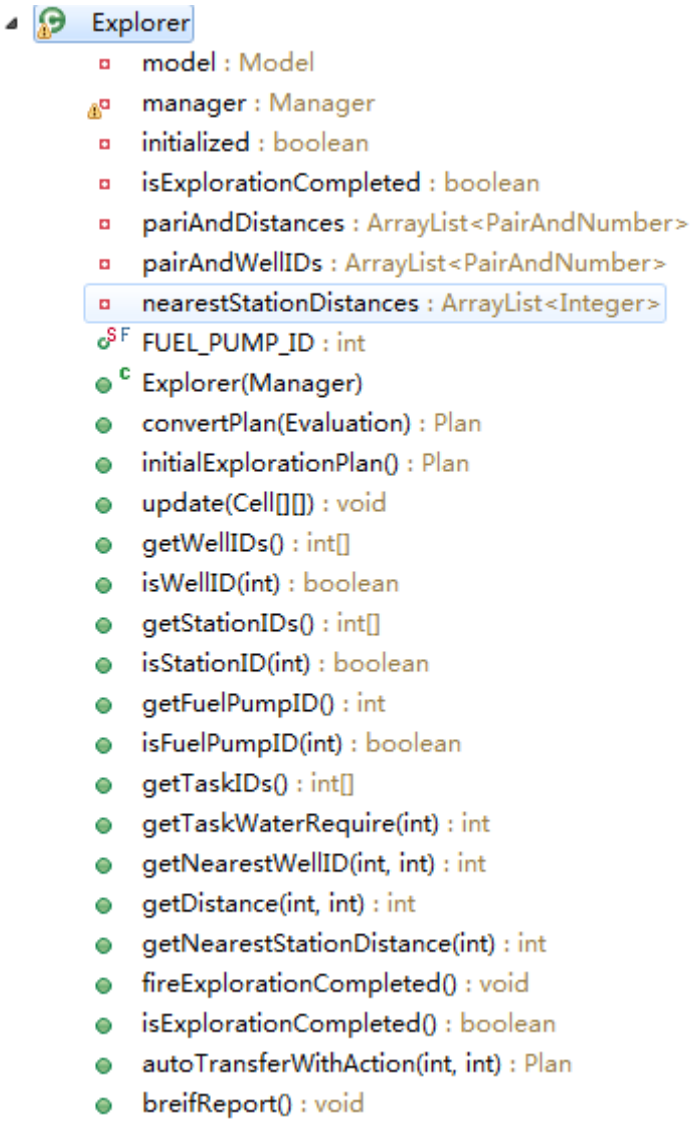
- Methods illustration:

```
/**
 * Generate Evaluation for a given Solution.
 *
 * @param seedGoal
 *           A solution.
 * @param vAgent
 *           The VirtualAgent which contains the same information as the
 *           current Tanker.
 * @param explorer
 *           The Explorer for this Evaluator.
 * @return An Evaluation for the given Solution.
 */
public Evaluation evaluate(SeedGoal seedGoal, VirtualAgent vAgent, Explorer explorer)
{
    Evaluation ← createEvaluation(seedGoal)

    Walk through the SeedGoal in sequence: {s1; s2; s3; …} {
        If s is a well, loadwater to maximum, and update vAgent
        If s is a task, complete that task and update water level in vAgent
    }

    While walking through the SeedGoal, if the vAgent didn't hold enough water to
complete a task, return the initial Evaluation
    If the distance cost more than the vAgent can afford, return return the initial
Evaluation

    If all goes well, return the updated Evaluation with the vAgent
}
```

5.3.1.2. Explorer

- Overview of class outline

The picture (5.3.1.1-1Explorer) is the class outline.

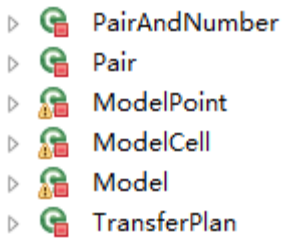


5.3.1.2-1Explorer

The Explorer works as one of the most vital components in the TankerAgent. The Explorer takes responsibilities of building and managing the mapping from the real environment to inner model. To facilitate calculations, this Explorer introduces ID systems to represent data. This ID system would make it possible to answer any query in computational complexity of O(1). More specifically, stations, wells and the fuel pump will be assigned unique IDs for identification purpose. This approach protects the inner data structure from the Manager. For example, the Manager would only know about the IDs, and the relations, such the distances, between these IDs. The manager would not be possible to know that, in this implementation, the environment is saved in Map of structure using java.awt.Point. Methods are provided to access the ID systems and the predefined relations among these IDs. For example, isWell(int ID) will tells if an ID represents a Well. In order to protect the data structure and facilitate calculation, the relations among the IDs are predefined and with O(1) to access them in all time once the method fireExplorationCompleted() is invoked by the manager. E.g. getDistance(int ID1, int ID2) will return the distance between the two given IDs in O(1).

- Structure of Explorer

The picture (5.3.1.2-3 Inner Classes) shows the complex structure of the Explorer. The design hides the Model that is implemented in this explorer, and greatly facilitates the model management.



5.3.1.2-2 Inner Classes

- Implementation illustration

1. Model management

The model is built by creating two vital mappings: `pointMap` maps from the environment points to model points; `modelMap` is the mapping from the model points to model cells that contains the cells from the original environment.

```
        /**
         * Mapping from origin environment points to built
 ModelPoint.
         */
        Map<Point, ModelPoint> pointMap;
        /**
         * Mapping from ModelPoint to ModelCell.
         */
        Map<ModelPoint, ModelCell> modelMap;
```

- The algorithm for updating the model from tanker view

```
/**
 * Update the model.
 *
 * @param view
 *           The current view of tanker.
 */
public void update(Cell[][] view) {
    Foreach cell in view {
        If the pointMap contains cell.point {
            ModelPoint ← get the relative ModelPoint from the pointMap
            ModelCell  ← get the ModelCell from the modelMap
            Update the ModelCell.cell using the cell
        } else {
            Point ← get the point of the cell
            ModelPoint ← create a relative ModelPoint using Point
            Save the mapping (Point → ModelPoint) in pointMap
            ModelCell ← create a ModelCell that contains the cell and ModelPoint
            Save the mapping (ModelPoint → ModelCell) in modelMap
        }
    }
}
```

2. ID system

If we want to find a well, which takes the smallest number of movement moving from one station to this well and to another target station, it should take O(n) to consider all wells. And if we repeat this requirement n times, the computational time becomes O(n^2). Another consideration is that: the computer process pure numbers faster than Objects. Therefore, in order to save computational time, to accelerate the deliberation process of a tanker agent, the ID system is a

wise choice.

There are three types of objects in the environment: the unique fuel pump, the stations, and the wells. Hence, one way to assign them unique ID could be: assign fuel pump 0, give stations positive IDs, and give all wells negative IDs. After using these IDs to represent the actual objects, we have to find a way to store them: that is a mapping from the ID to actual Objects in model. As stated above, the real implementation makes it possible to complete the mapping in O(1).

To create a mapping that takes O(1) to find a relative Object given an ID, we should find a mapping function, which could constantly map a number (The ID) to another number (The index of the Objects in the map that the ID maps to).

We define:

Map (ID, Object): A map that maps IDs to Objects

$f(\mathrm{ID}) = \mathrm{Index}$: The function takes constant time to calculate the index of the ID in the Map.

Recall that: if we want to find a number in a unsorted array, it takes O(n) to get the number, because we would have to walk through the array to see if the element stored in that position is the one we wanted. But if we want to find the a number in a well sorted continuous array, it takes O(1) to get it. For example: in a well ordered continuous array [0; 1; 2; 3; …], it is very easy to find any number we want. If we want 3, we just look at the 4th position of the array and there it is.

If we abstract that case, we could conclude that: in a well structured simple array, it is possible to create a function that takes linear time to calculate the index of a given value.

In the implementation, I want to build two relatively complex arrays:

```
/**
 * The structure to hold query results, which answers queries within O(1).
 */
private ArrayList<PairAndNumber> pariAndDistances, pairAndWellIDs;
```

The following is the generation code of `pariAndDistances` and `pairAndWellIDs`

```
int finalSize = sumTo(model.stations.size() + model.wells.size() + 1);
pariAndDistances = new ArrayList<PairAndNumber>(finalSize);
pairAndWellIDs = new ArrayList<PairAndNumber>(finalSize);
PairAndNumber pd = null;
Pair p = null;
int d = -1;
int wellID = -1;
int offset = model.wells.size();
for (int i = 0; i < list.size(); i++) {
    for (int j = i + 1; j < list.size(); j++) {
        p = new Pair(i - offset, j - offset);
        d = model.getDistance(list.get(i).mp, list.get(j).mp);
        pd = new PairAndNumber(p, d);
        pariAndDistances.add(pd);
        wellID = getNearestWellID(list.get(i).mp, list.get(j).mp);
        pd = new PairAndNumber(p, wellID);
        pairAndWellIDs.add(pd);
    }
}
```

The generation of the two arrays is using two for loops, and objects created are stored in sequence. Therefore I conclude that: these two arrays are well structured. It is possible to find a function that takes constant time to calculate the index of a given value.

```
    private int getDistance(Pair pair) {
        return pariAndDistances.get(calcIndex(pariAndDistances.size(), pair)).n;
    }

    private int getNearestWellID(Pair pair) {
        return pairAndWellIDs.get(calcIndex(pairAndWellIDs.size(), pair)).n;
    }
```

The two methods listed both use “calcIndex()” to calculate the index of a given value: Pair. The pair is just like the java.awt.Point, which contains two values x and y. Additionally, for the Pair to be ease of management, I added an additional rule: x <= y.

Now, let’s see the function “calcIndex()”

```
    private int calcIndex(int size, Pair p) {
        int base = model.stations.size() - p.x;
        int step = p.y - p.x - 1;

        int index = size - (sumTo(base) - step);

        return index;
```

(The sumTo(int n) represents the math formula: sumTo = 1+ 2+ 3+ .. + n)

As we can see, this function takes constants as its components, and thus the computational time is fixed.

Finally, I get the ID systems which has stored all required result from IDs, and most importantly, these information could be accessed in O(1), which greatly increases the speed of the deliberation for the tanker agent.

- **The algorithm to convert a solution into a plan**

```
public Plan convertPlan(solution) {
    {ID1; ID2; ID3;…} ← abstract the solution to get the sequence of IDs
    Plan ← travel from (Fuel Pump) to ID1 to ID2, …, to IDn to (Fuel Pump)
    If the ID represents a Fuel Pump, add RefuelAction to Plan
    If the ID represents a Well, add LoadWaterAction to Plan
    If the ID represents a Task, add DeliverWaterAction to Plan
    return Plan
}
```

G54DIA final report

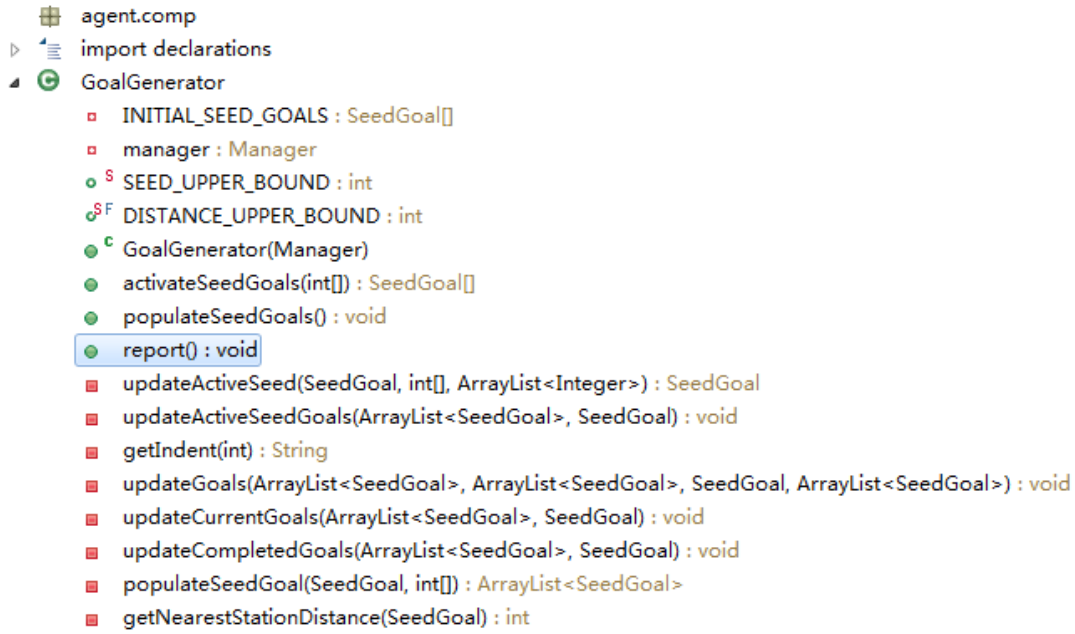
- The exploration strategy:

```
    /**
     * The Exploration Plan for this explorer. The plan contains 400 movements
     * and 4 refuel actions. More specifically, 12 diagonal movements, and
     * followed by a square of each length being 19, and 12 steps back to start
     * point. Then this strategy is performed in 4 differently areas to finally
     * search a space of 7569 in tanker view.
     *
     * @return An exploration Plan.
     */
    public Plan initialExplorationPlan() {
        initialized = true;
        Plan plan = new Plan();
        plan.setDescription("Initial search plan: using 400 movements and 4
refuel actions.");
        int[][] movePlan = { { MoveAction.NORTHEAST, 12 },
                { MoveAction.NORTH, 19 }, { MoveAction.EAST, 19 },
                { MoveAction.SOUTH, 19 }, { MoveAction.WEST, 19 },
                { MoveAction.SOUTHWEST, 12 }, { MoveAction.SOUTHEAST, 12 },
                { MoveAction.SOUTH, 19 }, { MoveAction.EAST, 19 },
                { MoveAction.NORTH, 19 }, { MoveAction.WEST, 19 },
                { MoveAction.NORTHWEST, 12 }, { MoveAction.SOUTHWEST, 12 },
                { MoveAction.SOUTH, 19 }, { MoveAction.WEST, 19 },
                { MoveAction.NORTH, 19 }, { MoveAction.EAST, 19 },
                { MoveAction.NORTHEAST, 12 }, { MoveAction.NORTHWEST, 12 },
                { MoveAction.NORTH, 19 }, { MoveAction.WEST, 19 },
                { MoveAction.SOUTH, 19 }, { MoveAction.EAST, 19 },
                { MoveAction.SOUTHEAST, 12 } };

        for (int i = 0; i < movePlan.length; i++) {
            for (int step = 0; step < movePlan[i][1]; step++) {
                plan.addAction(new MoveAction(movePlan[i][0]));
            }
            if (i != 0 && i % 6 == 5) {
                plan.addAction(new RefuelAction());
            }
        }

        return plan;
    }
```

5.3.1.3. GoalGenerator

- Overview of class outline

The picture (5.3.1.3-1 GoalGenerator) is the class outline.



5.3.1.3-2 GoalGenerator

This component is responsible for generating goals. For efficiency purpose, all possible goals would be generated at the end of the first exploration phase, and the result is stored in INITIAL_SEED_GOALS. Later, the goal generation would use this INITIAL_SEED_GOALS to populate activated goals, which would then be used to generate proper Solutions.

- Definition of the initial seed goals:

The initial goals are the systematically generated combinations of stations. All initial goals share the following properties:

1. The goals only contain stations. No wells are in consideration in the initial goals.
2. The distance of the goal is measured by: travel from the fuel pump, then visit all goals in sequence, finally come back to the fuel pump.
3. The distance never exceeds the tanker max fuel level, which is 100.
4. No duplicated seed goals in the initial seed goals. E.g. if goal1 = [1,2,3], to visit station 1, 2, 3 in sequence. Goal2=[2,3,1], to visit station 2, 3, 1 in sequence. Then this means goal1 and goal2 are duplicated.

- The reason for generate the initial seed goals:

It is explained in detail in (Section 3) that the aim is to deliver/load as much water as possible in each of up to 100 movements, which starts from fuel pump and ends in the fuel pump. If we relax the constraints: water is un-limited. Then the aim is to deliver as much water as possible in each tour. Due to the fact that the only way to deliver water is to visit stations, where tasks are allocated, our agent should visit as much stations as possible to complete the tasks.

The initial goals only consider the stations and the limited distance. By adding more constraints

to these goals, they could be gradually modified and finally become a valid goal that meets all constraints. For example, by providing the actual available tasks, come of the initial goals would be discarded, and the rest of the goals would be altered and survive. More specifically, there are two initial goals: goal1=[1,2,3], goal2=[2,3,4]. If only station 2 and 4 have tasks, then after adding this constraints, the goal2 would be altered to goal2_=[2,4], this goal2_ would then be called the second generation seed goals, and would be applied more constraints and finally changes to a valid goal. This is why, the goals are named "SeedGoal": they work like seed, after each selection, some will grow to adapt to the new constraints, and finally becomes a "tree".

- The algorithm for generating the initial seed goals.

```
public void populateSeedGoals() {
    StationIDs      ← call Explorer to get an array, contains all station IDs
    currentGoals    ← create empty array to hold SeedGoal
    finalizedGoals  ← create empty array to hold finalized SeedGoal
    newGoals        ← create empty array to hold new SeedGoal
    // add initial seeds to goals.
    Foreach stationID in stationIDs {
        SeedGoal ← create a seed goal using stationID
        currentGoals ← add SeedGoal to currentGoals
    }

    while (currentGoals is not empty) {
        SeedGoal ← remove the first element in currentGoals
        newGoals ← populateSeedGoal(SeedGoal, stationIDs)
        finalizedGoals ← updateGoals(finalizedGoals,
                                currentGoals, SeedGoal, newGoals)
    }
    INITIAL_SEED_GOALS ← convert finalizedGoals and save.
}

private SeedGoals populateSeedGoal (SeedGoal, stationIDs) {
    newGoals ← create new empty array to hold new goals
    uncompletedGoals ← the goals of stationIDs still not completed in SeedGoal
    Foreach uncompletedGoal in uncompletedGoals {
        newSeedGoal ← add uncompletedGoal to SeedGoal
        distance ← calculate the distance of newSeedGoal
        If (distance <= Tank.MAX_FUEL) then
            newGoals ← add newSeedGoal to newGoals
    }
    Return newGoals
}
```

- The algorithm for activating seed goals

```
/**
 * Given some task IDs, generate some SeedGoal that is activated. All
 * initial SeedGoal are suppressed. This process would generate the second
 * generation from the initial generation. The second generation is
 * activated, which could then be used to form solutions.
 *
 * @param taskIDs
 *          The available task IDs.
 * @return The activated Second Generation from {@code INITIAL_SEED_GOALS}.
 */
public SeedGoal[] activateSeedGoals(int[] taskIDs) {
    activeSeedGoals ← create empty array to hold SeedGoal.
    Foreach SeedGoal in INITIAL_SEED_GOALS {
        // intersects in set: f([1,2,3],[2,4,1]) = [1,2].
        activatedGoal ← intersects(SeedGoal, taskIDs)
        activeSeedGoals ← add activatedGoal to activeSeedGoals
    }
    return activeSeedGoals
}
```

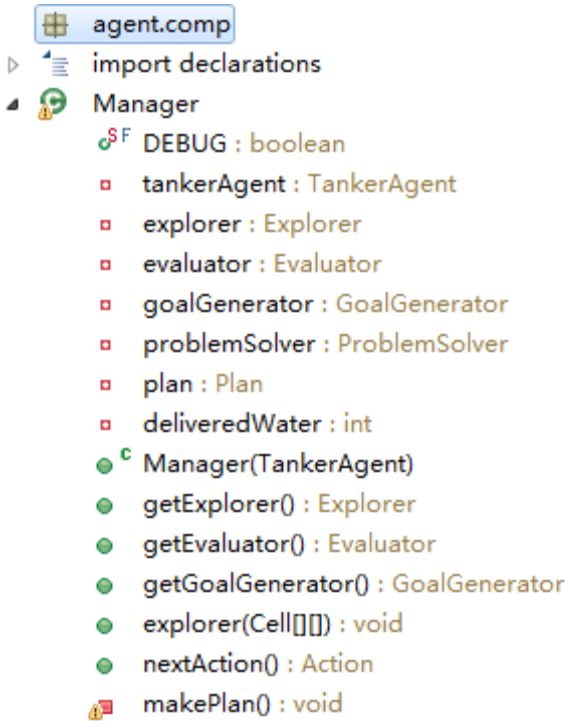
5.3.1.4. Manager

- Overview of class outline

The picture (5.3.1.4-2Manager) is the class outline.

The manager takes responsibilities of managing the components of the tanker, which includes the Explorer, Evaluator, GoalGenerator, ProblemSolver. The ultimate task for the Manager is to select one action for a tanker at any time when the method nextAction() is invoked.

Most of the work is done in the private method makePlan(). The rest of this section will explain how the manager works to co-operate with components and produce a sound plan.



5.3.1.3-1Manager

- The algorithm for the method: makePlan()

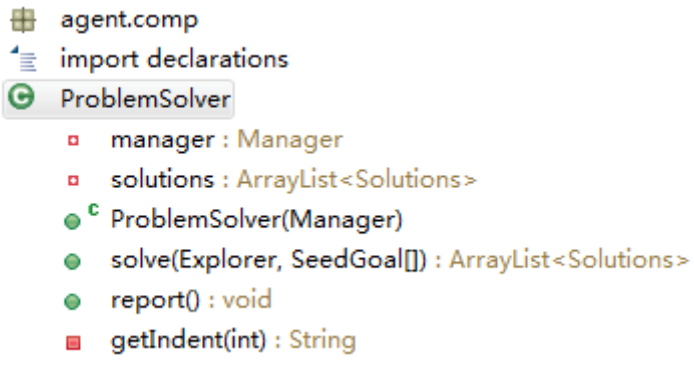
```
    /**
     * Make a new plan for the manager if required.
     */
Global variables:
Plan Explorer GoalGenerator ProblemSolver
private void makePlan() {
    If the Explorer has not performed a initial exploration plan {
        Then Plan ← Explorer.makeInitialPlan()
        Return
    } else if this is for the first time Explorer finished exploration {
        Explorer.buildIDSystem()
    }
    // after the model completing building ID systems...
    // abstract important information from the explorer.
    stationIDs ← explorer.getStationIDs()
    wellIDs    ← explorer.getWellIDs()
    taskIDs    ← explorer.getTaskIDs()
    // If there are no available tasks, stop the new plan.
    if (taskIDs.length == 0) {
        Return
    }
    avtiveSeedGoals ← GoalGenerator.activateSeedGoals(taskIDs)
    // now we have got the activeSeedGoals...
    // next thing to do: add wells to these tours...
    // call these solutions
    problemSolver ← create a  ProblemSolver
    solutions ← problemSolver.solve(explorer, avtiveSeedGoals)
    evaluations ← Evaluator.generateEvaluations(solutions)
    bestEvaluation ← pick the highest evaluated evaluation from evaluations
    // convert the solution to a plan by parsing the solution to explorer
    Plan ← explorer.convertPlan(bestEvaluation)
}
```

5.3.1.5. ProblemSolver

- Overview of class outline

The picture (5.3.1.5-2ProblemSolver) is the class outline.

The ProblemSolver is in charge of generating Solutions for activated goals.



5.3.1.5-1ProblemSolver

The main method is solve()

The following is the implemented code. As we could see, the solver actually pass the problem to Solution by create instances of Solution, and then store those solutions and then return. More details of Solution is available in later Section.

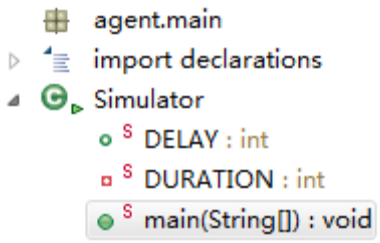
```
/**
 * Generate a set of solutions for some activated goals.
 *
 * @param explorer
 *            The explorer for the ProblemSolver.
 * @param avtiveSeedGoals
 *            The activated goals.
 * @return A set of solutions to the activated goals.
 */
public ArrayList<Solutions> solve(Explorer explorer,
        SeedGoal[] avtiveSeedGoals) {
    SeedGoal sg;
    Solutions s;
    VirtualAgent vAgent = manager.getVirtualAgent();
    for (int i = 0; i < avtiveSeedGoals.length; i++) {
        sg = avtiveSeedGoals[i];
        s = new Solutions((VirtualAgent) vAgent.clone(), explorer, sg);
        solutions.add(s);
    }
    return solutions;
}
```

5.3.2. agent.comp package

This package contains two classes: Simulator and TankerAgent.

5.3.2.1. Simulator

The simulator only contains a main method, which is provided in the coursework.



5.3.2-1Simulator

The only changed code is:

```
/**
         * Changed the code to create a tanker: the
TankerAgent.
         */
        Tanker t = new TankerAgent();
```

That is, replace the Tank with the TankerAgent, the one I developed for this simulator.

5.3.2.2. TankerAgent

The Tanker Agent developed for this course-work.

Basically, the tanker tasks the view and passes the information to the manager. Then the Manager would decide the next action by making reasonable plans. To see more details, please see the class Manager.

The implemented code is relative simple:

```
    /**
     * Get the current view and choose an action.
     *
     * @param view
     *           the cells the Tanker can currently see
     * @param timestep
     *           The current timestep
     * @return an action to perform
     */
    public Action senseAndAct(Cell[][] view, long timestep) {
        // update the explore information. i.e. the environment model for the
        // task environment.
        manager.explorer(view);
        // get the next action for the tanker.
        Action action = manager.nextAction();
        return action;
    }
```

5.3.3. agent.test package

All classes in this package are used to perform testing on problems in interest.

5.3.3.1. TestCollectionsSort

This class aims to test the sorting method for collection, which is: Collections.sort(Collection c). The testing result shows that: a collection of integer would be sorted into ascending order.

5.3.3.2. TestForSolutionGeneration

This class contains simplified version of generation “solutions” for a given initial seed goal.

The idea is that: a solution must be a power set of actions {deliver, load, doNothing}

The algorithm for generating the power set of fixed sequence tasksIDs.
for example “w” denotes a well, taskIDs = [1,2]
the output should be:
[] [1] [1 2] [2] [w] [w1] [w1 2] [w2] [1w] [1w2] [2w] [w1w] [w1w2] [w1w2w] [w2w]
Some ground rules for this power set generation:

1. The sequence of appearance of TaskID in power set should be in the same sequence as they occur in taskIDs.
2. The wells cannot be near to each other. I.e. there must be one or more tasks in between.

5.3.3.3. TestModelStorage

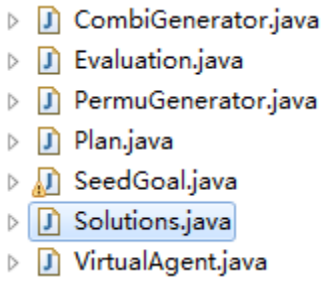
For efficiency of calculating, I decided to give the model the following properties. Well with negative ID, Pump with ID 0, Stations with positive ID. Wells = {-5,-4,-3,-2,-1} ; P = 0; Stations = {1,2,3,4,5,6,7,8,9,10}; to efficiently give distance, instead of using mapping, which generates the result by complexity of n^2, the following structure is used. This class aims to test if the storage strategy described in class Solution.

5.3.3.4. TestPermutationEfficiency

This class aims to tests the permutation efficiency. The result is: it is relatively efficient to generate the permutation up to 7. This suggests that, if the algorithm introduced in the section 4 (shortest path algorithm) works, the length of the set should be less or equal to 7.

5.3.4. agent.util package

The picture (5.3.4-2agent.util package) illustrates all members in this package. This section would introduce them in detail with the implemented code or algorithms.



5.3.3.4-1agent.util package

5.3.4.1. CombiGenerator

The class could generates all combinations for a given array.

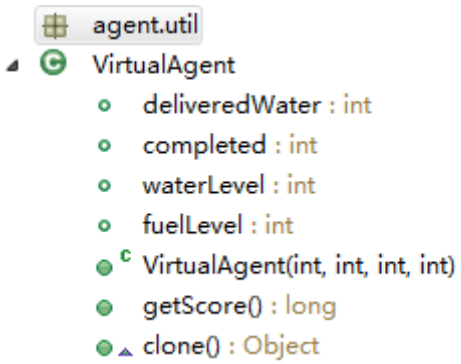
5.3.4.2. PermuGenerator

The class could generate all permutations for a given array.

5.3.4.3. VirtualAgent

This class works as a container, which copy the tanker information and store in it.

The picture (5.3.4.3-2VirtualAgent) is the class outline. From this outline, we could easily see that: the Virtual agent copies the delivered water, completed task, water level and fuel level. This class serves the purpose of management of fuel, and could play a virtual tour in a solution and in evaluation.



5.3.4.2-1VirtualAgent

5.3.4.4. Evaluation

The evaluation takes a solution and generates a score for that solution.

The main method for score generation is:

```
/**
 * Automatically generates an evaluation score for current Evaluation.
 */
```

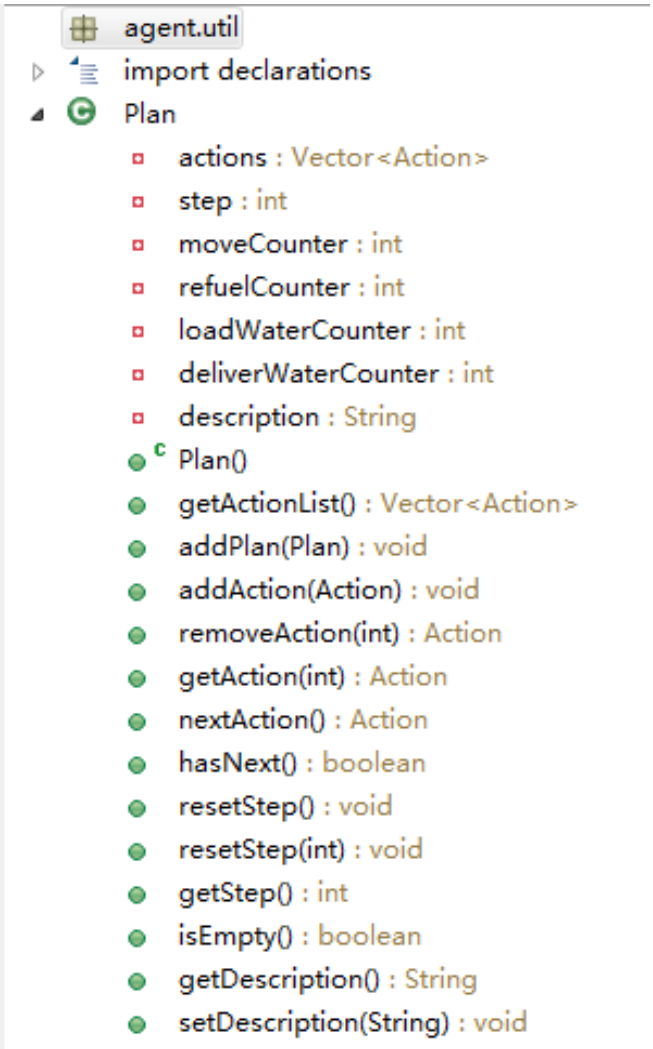
```
public void updateScoreAndEvalution() {
    if (taskCounter == 0) {
        return;
    }
    score = (long) taskCompleted * (long) deliveredWater;
    simpleEvaluation =
        ((long) taskCounter * (long) waterCounter + (long) waterLevel)
            / (long) timeEclipsed;
}
```

This `simpleEvaluation` is the value that represents the evaluation result of a solution. Basically, it states: load as much water as possible in the `timeEclipsed` .

5.3.4.5. Plan

The picture (5.3.4.5-2Plan) is the class outline.,
Basically, it stores a set of Action in Vector in `actions`. The `moveCounter`, `refuelCounter`, `loadWaterCounter` and `deliverWaterCounter` help to summarize information of the Plan.

As the code is well documented in java source, details of implemented code would be available in the appendix.



5.3.4.4-1Plan

5.3.4.6. SeedGoal

The SeedGoal contains several IDs, and has the ability to calculate the distance of the tour, moving from the first ID to the last ID, etc. The other functionality is to perform an upgrading, to search for shortest path for the given IDs.

The most important algorithm for method “updateShortestPath()” has been introduced in section 4.

5.3.4.7. Solutions

This class will generate and store the solutions for one activated goal.

The main algorithm for this class: populate solutions from a seed goal.
The general process and constraints are introduced in section 5.3.3.2

```
/**
 * The main method for solution generation for the {@code sg}.
 * The solutions are generated by generating power set of (DeliverWater,
 * LoadWater, DoNothing) with constraints applied: 1. maximum distance is no
 * larger than the Tanker.MAX_FUEL. 2. if water is not enough, go to well
 * first and then finish task.
 *
 * @param sg The seed goal
 */
private void initSolution(SeedGoal sg) {
    /*
     * Populate the initial solution for the seed.
     */
    tempSolutions1 ← initTempSolutions(seedGoal1);
    /*
     * Populate the solutions by generating power set of (DeliverWater,
     * LoadWater, DoNothing)
     */
    solutions1 ← populateSolutions(tempSolutions1, seedGoal1);
    /*
     * Finally, append one well before the agent goes home, also is
     * optional.
     */
    solutions1 ← appendFinalWell(solutions1, seedGoal1);
    seedGoals ← solutions1;
}
```

6. Evaluation of Single Agent

6.1. Result

The agent has been tested thoroughly, here shows the part of the results:

versionF Evaluation:simpleEvaluation = ((long) taskCounter * (long)waterCounter + (long) waterLevel) / (long)timeEclipsed;

index	completed	delivered	score	averageDeliver	stations	wells
1	1181	5834776	6.891E+09	4940.539	18	13
2	1344	6576247	8.838E+09	4893.041	20	15
3	1493	7636558	1.14E+10	5114.908	23	10
4	1420	7010445	9.955E+09	4936.933	22	9
5	1503	7455376	1.121E+10	4960.33	24	10
6	1498	7233666	1.084E+10	4828.883	23	10
7	1344	6711877	9.021E+09	4993.956	21	12
8	1549	7627476	1.181E+10	4924.129	24	8
9	1805	8943853	1.614E+10	4955.043	29	11
10	814	4047654	3.295E+09	4972.548	13	10
11	1925	9647193	1.857E+10	5011.529	31	15
12	1722	8611608	1.483E+10	5000.934	27	13
13	1091	5560410	6.066E+09	5096.618	16	10
14	1141	5654696	6.452E+09	4955.912	17	12
15	1780	8875770	1.58E+10	4986.388	28	16
16	1174	5807430	6.818E+09	4946.704	18	17
17	1562	7863326	1.228E+10	5034.14	23	14
18	1422	7150594	1.017E+10	5028.547	22	17
19	1871	9544323	1.786E+10	5101.188	28	5
20	1669	8265229	1.379E+10	4952.204	28	11
average	1465.4	7302925.35	1.11E+10	4981.724	22.75	11.9

This result shows:

On the average 20 tests, the environment generates 22-23 stations and 12 wells. And on average, the agent would complete 1465 tasks, deliver 730000 amount of water, and get 1.11E+10 as final score.

To compare with this agent, another version of agent is tested:

version1 First come, first serve. Until no other

index	completed	delivered	score	averageDeliver
1	1255	6221712	7.8E+09	4957.5394
2	1378	6883141	9.5E+09	4995.0225
3	1252	6116179	7.7E+09	4885.127
4	1152	5631771	6.5E+09	4888.6901
5	1401	7132203	1E+10	5090.7944

6	1479	7488228	1.1E+10	5063.0345
7	967	4682970	4.5E+09	4842.7818
8	831	4222187	3.5E+09	5080.8508
9	1856	9372340	1.7E+10	5049.7522
10	1292	6432679	8.3E+09	4978.8537
11	1186	5897689	7E+09	4972.7563
12	935	4643338	4.3E+09	4966.1369
13	1222	6087167	7.4E+09	4981.3151
average	1246.615385	6216277	8.1E+09	4980.9734

In this version, the agent tries to perform the best to finish tasks in sequence.
By comparison, we can find that the version implemented performs approximately 40% better than the first come first serve version.

6.2. Performance

The implemented agent will deliberate a lot longer in the first time than any other times. The first time, the agent would try to generate all possible tour among the stations, and therefore would take time to finish this calculation. However, this process is just like java compiler, which happens once and benefit from it forever. This implementation facilitates the later solution generation process.
The exploration takes 400 movements and 4 refuel actions to search a space of 7569. Compared with other strategy such as spiral, 33-movements strategy, this exploration plan gets a relatively high performance in exploration phase.

6.3. Limitation

As described, the first deliberation costs too much time, compared with later deliberations. This is due to the fact that the more stations there are the more combinations the agent has to consider. And this complexity grows more than exponential. Therefore, when the environment grows bigger, for example there are 1000 stations to be considered, the time the agent spend constructing the initial seed goals will be unpredictably huge. To summarize, for this version, the scalability is not satisfiable. The implementation and design ignore the scalability of the problem and thus the agent may be limited to solve relatively small task environment.

6.4. Conclusion

This single agent makes one deliberation on average 100 time steps. On each tour, the agent could complete 1 or multiple tasks. The result is satisfying; the performance in the given environment is also efficient. However, the lack of consideration on the scalability of this problem limits the design and the implementation, and thus makes this agent not applicable to huge environment. To sum up: this agent is suitable for relatively small environment with high performance.

7. Multi-Agent and Evaluation

For the multi-agent version, I choose the information broad cast to be the way for the agents to share information. There are two types of agents: workers and explorers. The explorers always searches for tasks in the environment; the worker will try to complete all tasks. If there are no available tasks, then this worker will also perform the exploration for once, and try to complete new found tasks.

7.1. The result

multi_f	two explorers, and one worker						
index	completed	delivered	score	averageDeliver	agent	stations	wells
1	1898	9522009	1.807E+10	5016.865	3	24	15
2	2053	10362791	2.127E+10	5047.633	3	26	18
3	1634	8221824	1.343E+10	5031.716	3	22	11
4	1298	6541581	8.491E+09	5039.739	3	17	5
5	1781	9002252	1.603E+10	5054.605	3	23	9
6	2060	10469860	2.157E+10	5082.456	3	28	9
7	1706	8404411	1.434E+10	4926.384	3	22	12
8	1815	9169897	1.664E+10	5052.285	3	25	11
9	2183	10832072	2.365E+10	4962.012	3	29	9
10	1368	6956607	9.517E+09	5085.239	3	19	7
11	1961	9817561	1.925E+10	5006.405	3	26	14
12	1122	5670984	6.363E+09	5054.353	3	15	12
13	1705	8412655	1.434E+10	4934.109	3	23	7
14	2330	11526676	2.686E+10	4947.071	3	30	12
15	1779	8945151	1.591E+10	5028.191	3	23	11
16	1329	6594041	8.763E+09	4961.656	3	18	9
17	1485	7353735	1.092E+10	4952.01	3	19	4
18	2506	12679665	3.178E+10	5059.723	3	34	12
19	1399	6881004	9.627E+09	4918.516	3	19	10
20	2493	12011541	2.994E+10	4818.107	3	32	13
average	1795.25	8968815.85	1.684E+10	4998.954	3	23.7	10.5

On average, this multi-agent version requires two explorers and one worker. The result shows that this version gets score 50% higher than the single agent version.

7.2. Evaluation

For most multi-agents, the more agents there are the better performance we will probably get.

In this version, it only requires 3 agents: 2 for exploration purpose, 1 for task completion purpose. This decision is made by the following facts:

- The single agent worker works efficiently to complete tasks, so that it complete tasks faster than the newly generated tasks.
  - By testing different number of explorers and workers, it shows that: one worker is sufficient in this relatively small environment, which may contain up to 37 stations. More workers do not give too much improvement in performance. Because for this environment, the task generation rate is too low, and therefore one single agent could handle them all and still have free time while there are no tasks.
- By testing the exploration, it is found that two explorers are sufficient for searching for new tasks in an area of 8000. More agents do not give a much better performance than just 2 explorers.
- By setting identical agents: all agents could explore and work to complete tasks:
  - The results shows: for the lack of knowledge of the environment, the newly found tasks are frequently completed, and the agents will have to explorer the environment. By testing, I find the score is not higher than the 2-explorer and 1-worker vision given three agents.

But this multi-agent still have problems to cope with huge environment as the single version.
Additionally, due to the fact that the agents shares the information by broad casting, when there are huge amount of agents, this multi-agent version would not work efficiently in theory and in reality, it will even not work properly.

7.3. Implementation

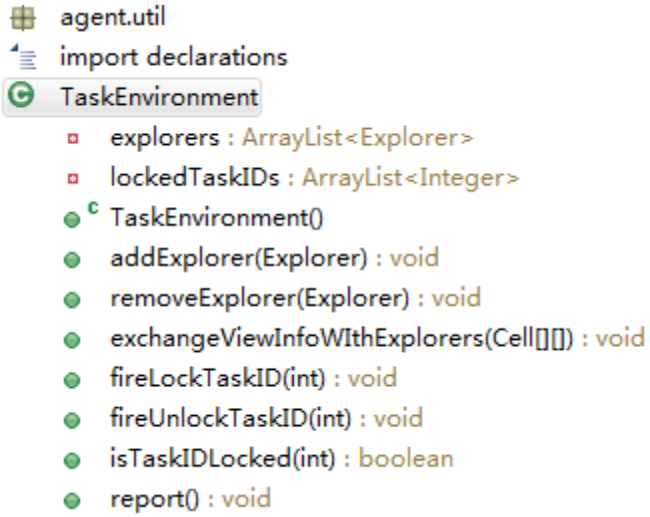
The only change to this multi-agent version compared with the single version is about the information sharing.

Firstly, I create a “TaskEnvironment”, and assign this object to each of the agent, such that they could communicate with each other. In this version, the strategy uses the broad casting.

The picture (7.3-1TaskEnvironment) shows the outline of the class.

This class will lock taskIDs once the agent decided to make a plan on those IDs and in execution, other agents would never be able to access those resources. Once the locked taskIDs are completed, those IDs will be unlocked for exploration.

As the implementation is easy, here just show the implemented code with some comments.



7.3-2TaskEnvironment

```
/**
 * Add an explorer to this environment for information sharing.
 * @param e The explorer.
 */
public void addExplorer(Explorer e) {
    explorers.add(e);
}
/**
 * Remove an explorer from this environment for information sharing.
 * @param e The explorer.
 */
public void removeExplorer(Explorer e) {
    explorers.remove(e);
}

/**
 * Share the information by passing the view to another explorer.
 * @param view The current view of one explorer.
 */
public void exchangeViewInfoWIthExplorers(Cell[][] view) {
    for (int i = 0; i < explorers.size(); i++) {
        explorers.get(i).updateExplorer(view);
    }
}

/**
 * Lock a task such that other explorers would ignore that task.
 * @param ID The task to be locked.
```

```
     */
    public void fireLockTaskID(int ID) {
        if (!lockedTaskIDs.contains(ID)) {
            lockedTaskIDs.add(ID);
        }
    }

    /**
     * Unlock a task once the task is completed by that explorer.
     * @param ID The task to be unlocked.
     */
    public void fireUnlockTaskID(int ID) {
        for (int i = 0; i < lockedTaskIDs.size(); i++) {
            if (lockedTaskIDs.get(i) == ID) {
                lockedTaskIDs.remove(i);
            }
        }
    }

    /**
     * For the explorer to test is a task is locked or not.
     * @param ID The task ID to be tested.
     * @return True if the task is locked by some explorer; otherwise false.
     */
    public boolean isTaskIDLocked(int ID) {
        return lockedTaskIDs.contains(ID);
    }
```

- The changes in the simulator:

The TaskEnvironment is created and assigned to all of the agents.

```
    // Create a fleet
    Fleet fleet = new Fleet();
    Tanker t;
    // create a TaskEnvironment for explorers to register and share information.
    TaskEnvironment taskEnvironment = new TaskEnvironment();
    for (int i = 0; i < FLEET_SIZE; i++) {
        // create a multiagent with this TaskEnvironment.
        // the i is for the ID of the agents.
        t = new MultiTankerAgent(taskEnvironment, i);
        fleet.add(t);
    }
```

8. Conclusion

The design and the implementation of the single agent have been explained in detail from section 2 to section 5. In section 6, the agent is critically evaluated. In section 7, a multi-agent solution is briefly introduced and evaluated.

A single agent has been designed and properly implemented to solve this problem. The performance for the single agent is satisfiable. To some level, it is plausible to claim that this single agent will perform perfectly well if the environment is totally known by that agent due to the fact that: the multi-agent version takes only one worker, and most of the time, the tasks in the environment will be cleared once they are detected. The efficient deliberation made it possible for the agent to complete tasks faster than the task generation rate. However, for the lack of consideration on the scalability of the environment as well as the problem, this agent is not applicable in a relatively large environment. For example, if there are 1000 tasks, then it would make this single agent version deliberate an unpredictable long time.
For this multi-agent version, it works generally well, at least better than the single agent version, by setting two explorers and one worker to co-operate with each other through information sharing. However, it also shares the same problem of scalability: this multi-agent is suitable for a relatively small environment, and is not applicable to big environment, e.g. an environment with hundreds of stations. This multi-agent version guarantees there would be no fatal error: the situation where more than 1 agent tries to completed one task and only one of them get to do it and another would also go to that station and do nothing. Thus, this multi-agent version successfully generates a simple co-operative environment for multiple agents.