A8 - Shipping Game

By Michael Patashnik and the CS2110 Course Staff

Contents

1. Introduction
2. Game overview
3. Installation and running
4. Using the GUI
5. Brief introduction to the classes
6. Your tasks
7. Concurrency
8. Grading

1. Introduction

You can work on A8 in groups of two. Form the group well before the assignment due date. Both must do something to form the group: one proposes, the other accepts. People in a group must work together. It is against the rules for one person to do programming on this assignment without the other person sitting nearby and helping. Take turns "driving" —using the keyboard and mouse.

With the exception of your CMS-registered group partner, you may not look at anyone else's code, in any form, or show your code to anyone else (except the course staff), in any form. You may not show or give your code to another student in the class.

We will pin a note on the Piazza, as we did for earlier projects, to discuss important points. Check it often, especially before you post a question.

2. Game overview

In Shipping Game, you play the part of a shipping company engineer who ensures that parcels get where they're going in a timely fashion. In a run of Shipping Game, you are presented with an undirected, weighted graph (see next page) that represents the world. The nodes in the graph are cities (labeled with their names), and the edges are highways between cities (labeled with their length). At each city, there may be some parcels, to be delivered to other cities. Thus, a parcel has a start city and a (different) destination city. The smaller filled-in circles that appear in some cities are the parcels that need to be picked up and delivered.

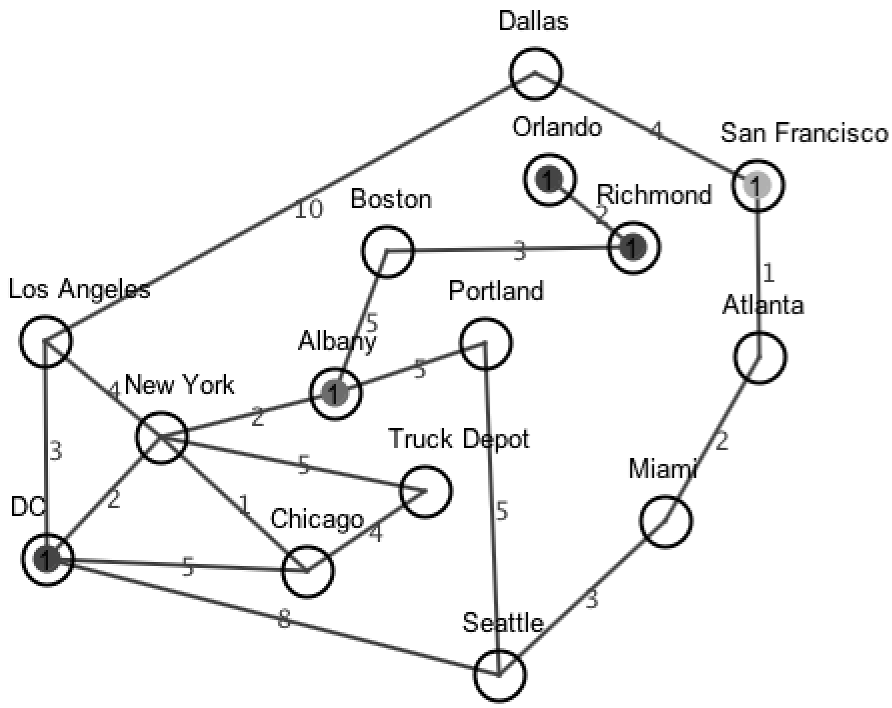

Figure 1: *A sample ShippingGame map. Finally, a rational explanation for why your flight from DC to Chicago was routed through New York*

You control a fleet of trucks, which can pick up and deliver parcels. You can give trucks instructions at the beginning of the game. Also, at various events, such as a truck reaching its destination, it will notify you so you can give it additional instructions. Trucks begin the game on the Truck Depot city (there is exactly one Truck Depot) and must return there after delivering all parcels for the game to end.

3. Installation and running

Installation

The code for this project is provided as a .jar file, without source files. Various data files are provided as text files in folders.

1. Download the A8 files from CMS or Piazza.

2. Create a new Java project in Eclipse, named, for example, A8.

3. Drag ShippingGame.jar and compressed folder data into your A8 folder, expand (unzip) the data folder. Don't change the names of any of the folders within data or the code won't be able to find the data. When this is done correctly, you should have a data folder with your project root, just as you did in A7.

4. Select ShippingGame.jar and right click on it. In the contextual menu that pops up, select Build Path -> Add to Build Path. This makes available all the .class files in the jar file.

5. Select ShippingGame.jar (now in Referenced Libraries) and right click it. In the contextual menu, select Build Path –> configure build path... In the menu that pops up, select Javadoc location and click the edit button on right... Select "Javadoc URL", then click Browse... –> Navigate to the "doc" folder (don't go into it) and click Open. Click ok on all the menus.

6. Create a package called student in folder src (right click src -> new... -> package) . All code you write goes in package student . Drag into package student your files MinHeap.java and Paths.java. You will have to put a statement "package student;" as the first line of each file, and you may have to put in a statement "import game.PQueue;". After doing this, there should be no syntax errors.

7. Create a class MyManager in package student. MyManager must extend game.Manager. MyManager is your manager class, and it is your main submission. You must use the name MyManager. Note: In the window that pops open, in which you type the name of the class and other things, in the Superclass field, type "game.Manager" and make sure the boxes "Constructor from superclass" and "Inherited abstract methods" are checked. This makes everything easier for you.

Running

1. Select A8 -> Run As... –> Run Configurations... –> New Launch Configuration (blank page with star in top left)

2. Click Search... next to Main class –> Select game.main

3. Click run.

From here on out, the settings are saved, so you can simply run the project by clicking the green run button as usual.

Arguments

You can provide arguments to the runner to have it do different things. All argument combinations have to start with the name of your manager (MyManager). With no arguments or just the manager classname provided, the game will launch in GUI mode. There are no other applicable arguments for this mode. Any other combination of arguments will launch the game in GameRunner mode instead, where the game will automatically run your manager on a sequence of maps.

To run GameRunner on a series of seeds <seed1> <seed2> <seed3> such as 15, 12345, 907230587, use each seed as an argument, separated by a space:

MyManager 15 12345 907230587

To run GameRunner on a series of n random seeds, use the $-r$ flag, followed by the number of random seeds to run (replace MyManager with the name of your Manager). For example, to run your manager on 10 random seeds:

MyManager -r 10

The GameRunner mode includes the GUI to show the progression of the run games. It also prints output of the games to the console. If you would like to just see the output and not show the GUI (headless), add the $-h$ tag. Thus, either:

MyManager -h 15 12345 907230587

or

MyManager -h -r 10

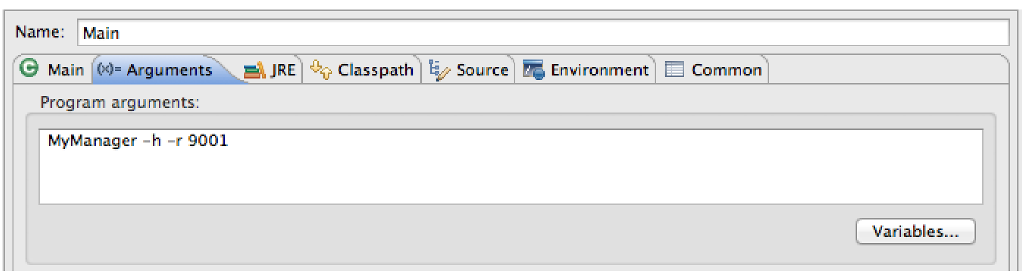

Figure 2: *Running MyManager in headless mode on 9001 random seeds.*

4. Using the GUI

Assuming the installation process went correctly and you have created a class that extends manager, you should be able to run the main class and see the gui. *(Note: the generated map may be different than pictured below)*. It is fully resizable. All cities (nodes) can be dragged to allow you to make the graph look cleaner, but this purely visual change has no impact on the board state. Here are some other features..

4a. Menu bar

- File -> Open...: Open a JFileChooser rooted at the data/maps directory. You can choose a map (in JSON format) to load into the gui.

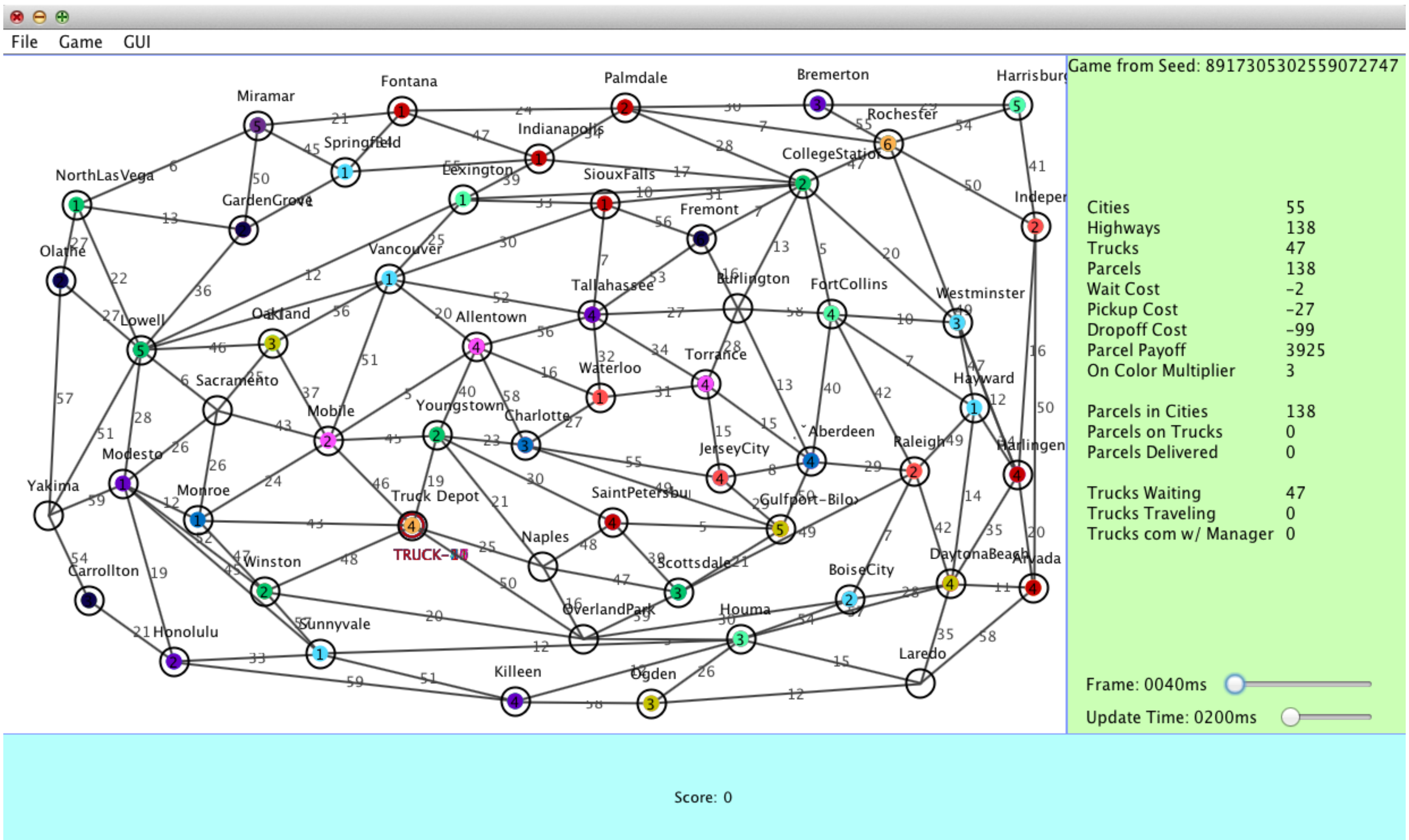

Figure 3: *The Shipping Game GUI. Now with 50% less (visual) sugar!*

- File -> Quit: Close the gui and terminate the program.
- Game -> Start: Begin the game. Clicking this again after the game has begun does nothing, but clicking it after the game has ended will restart it.
- Game -> Reset: Reset the displayed game to its initial state..
- Game -> New Random Map...: Generate a new random map with the seed (long int) given in the popup window. Exit this option by clicking cancel on the popup window.
- Game -> Print Game JSON: Print a JSON string represention of this game to the eclipse console. Useful to save an interesting board by resetting it, printing the game JSON, and creating a new text file in the data/maps directory with the game JSON as its contents.
- GUI -> Repaint: Repaint the gui. Try clicking this if the gui looks strange before trying other solutions.
- GUI -> Edge Coloring: Control how edges are colored on the gui.

  - Default: All edges are dark gray
  - Highlight-Travel: Edges being traveled by a truck are red, others are dark gray.
  - Distance Gradient: Colors edges by a linear interpolation from light to dark, based on their weights. Shorter (lighter) edges are painted in a lighter color; longer (heavier) ones are in a darker color.

4b. Info panel

The panel on the right is the info panel —it displays useful information about the currently displayed game. The top block of data is static and will not change as the game runs.

- Game From...: A description of the current game —either from seed, from file, or custom
- Cities: Number of nodes
- Highways: Number of edges
- Trucks: Number of available trucks
- Parcels: Number of parcels, including delivered parcels
- Wait Cost: Cost (per frame) of a truck doing nothing
- Pickup Cost: Singleton cost of a truck picking up a parcel
- Dropoff Cost: Singleton cost of a truck dropping off a parcel
- Parcel Payoff: Singleton value of dropping off a parcel at its location
- On Color Multiplier: Multiplier applied to the parcel payoff if the delivering truck and the parcel share a color.

The next two blocks of non-static data are changed as the game proceeds.

- Parcels in Cities: Number of undelivered parcels that are not on trucks
- Parcels on Trucks: Number of undelivered parcels that are on trucks
- Delivered Parcels: Number of delivered parcels (they are no longer on the map)
- Trucks Waiting: Number of trucks that are currently doing nothing
- Trucks Traveling: Number of trucks that are currently traveling an edge
- Trucks com w/ Manager: Number of trucks that are currently communicating with the Manager —i.e. have called truckNotification(...) and are awaiting a response.

You control how often non-static data is refreshed using bottom slider Update Time. Dragging the slider to the left updates the data more often at the cost of using a tiny bit more processing power. For debugging purposes, it is useful to have the data update as quickly as possible, whereas once your code works correctly, you can have the data rarely updated to get the best possible score.

Finally, slider Frame controls how quickly the game progresses. A higher value (drag right) causes the trucks to travel more slowly and favors more computationally complex solutions, and a lower value causes the trucks to travel more quickly and favors more computationally simple solutions. It can be useful in debugging to drag the slider far to the right to watch the trucks move in slow motion. However, altering the slider causes the score to change and change the color of the Frame label to red. (The label will stay black when the frame rate is unaltered).

5. Brief introduction to the classes

Assuming your installation worked correctly, you should be able to see the javadoc specifications of all public classes and methods in package game. You can do this by right-clicking index.html (in data/doc/) and selecting open with... -> Web Browser. If this doesn't work, consult section 2 of this guide or ask on Piazza/at office hours.

The following diagram shows the class structure. Each interface and class in the figure is explained briefly below. For a more thorough explanation, see the javadoc specs.

Interface Colorable

Colorable is implemented in classes that have a "color". Interface BoardElement extends Colorable, so all elements drawn on the map have a color.

Interface UserData

UserData is implemented in classes that allow the user (Your manager) to store data in them. You don't have to use UserData's methods, but it may prove useful in some graph algorithms.

Interface BoardElement

In addition to being a catenation of the requirements of interfaces Colorable and UserData, BoardElement specifies behaviors required for displaying an object on the GUI. Implementing classes are required to be able to provide various GUI-sensitive information, such as how to draw it, what its name on the GUI is, and if any trucks are currently "on it".

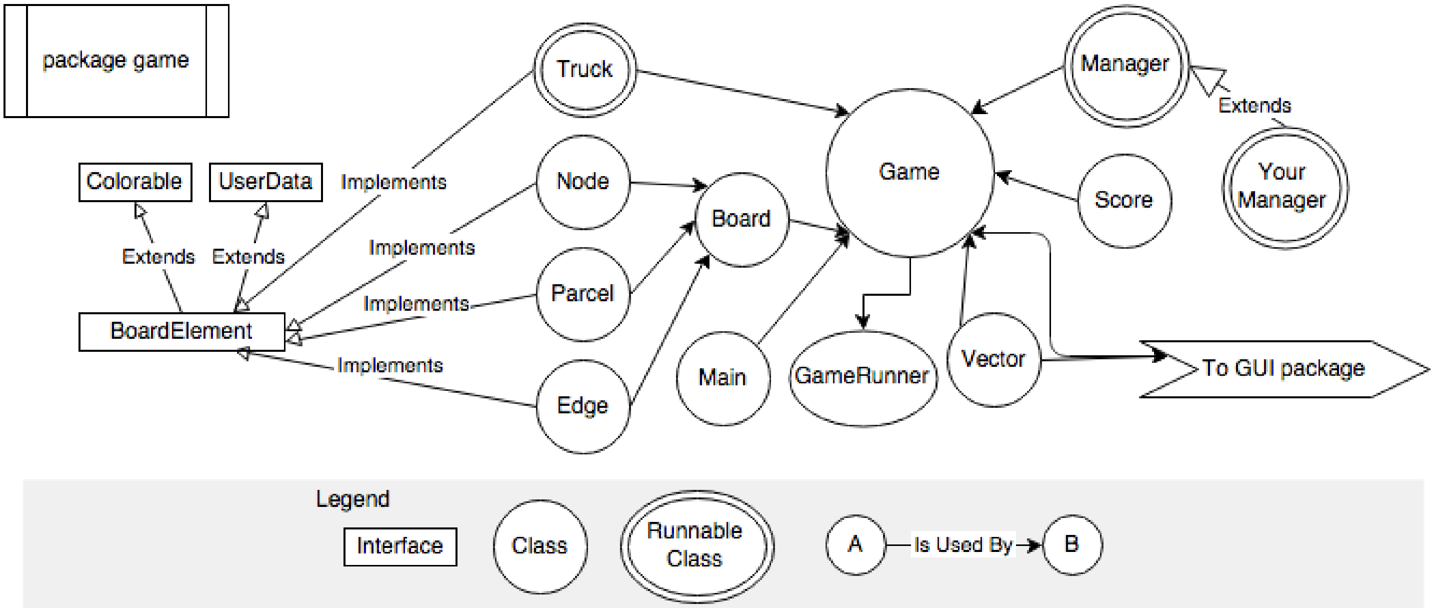

Figure 4: *Class Layout of package game. More lines and circles.*

Class Truck

Instances of Truck are the game pieces of ShippingGame. Trucks take instructions from Managers and can travel the map and pick up and drop off a one parcel at a time. Trucks are runnable: each instance of Truck runs in its own thread.

Class Node

Each Node on the map represents a city, which is connected to other cities by highways (Edges). Nodes contain parcels until trucks pick them up. Initially, all Trucks are at the special Node TruckDepot, and they must return there before the game can end.

Class Edge

Instances of Edge connect the Nodes in the Board. Each Edge represents a highway that connects two cities (Nodes). Edges are undirected (bidirectional) and weighted (have length). The weight of the Edge tells how long it takes for a truck to travel a given Edge.

Class Parcel

Instances of Parcel are the scoring pieces of ShippingGame. Each Parcel starts at a city. It awards points when it is delivered to its destination. The game can end only after all Parcels have been delivered.

Class Board

The Instance of Board for a game contains the Nodes, Edges, and Parcels associated with the game. It has the scoring constants related to various actions taken during the game and convenience methods such as getting a random Node or random Edge.

Class Score

An instance of Score encapsulates the score of the game. In addition to preventing the user (your manager) from altering the score, it has convenience methods for checking for a valid color and calculating the cost of a Truck Speed.

Class Vector

Vector is similar to class Point of Java except that it uses doubles instead of ints. It is used internally for calculation and is provided as a convenience.

Class Main

Main begins execution of the game —it has method main. It contains static convenience methods for use all over the project.

Abstract class Manager

You have to extend Manager. The non-abstract behavior of class Manager gives convenience methods for getting the map, trucks, and parcels in the game. It also gives enum Notification, which specifies the different reasons a Truck would notify the Manager of a change.

Class Game

An instance of Game represents the game as a whole. It is the unifying factor, tying all the other pieces together and communicating with the GUI to keep the visual version of the game up to date. It includes convenience methods for querying the current state of the game.

Class GameRunner

An instance of GameRunner provides for the automatic running of many games. This is useful to automatically test your manager on many boards, and it can be accessed via arguments given to main.

Your Manager

An instance of your manager fills in the missing behavior of class Manager to determine how the game runs. See the next section for more on what you have to do in your extension of class Manager.

6. Your tasks

Your primary task is write a subclass of abstract class Manager, putting it in package student. You must override the two abstract methods of class Manager: **run()** and **truckNotification(...)**, which determine the behavior of the trucks in the game. On their own, trucks don't do anything; it's up to you to write code to instruct them. The design of these two methods is up to you. We give a few details below.

Figure 5: *The average intelligence of your shipping boys*

Procedure run() is called as soon as the game begins. In run(), you can do initial computation and give trucks their initial instructions. Additionally, the body of run() is executed in a separate thread from the trucks, so you can continue to do computation after the trucks have begun their travel. Your implementation can either loop forever and continually add information to the trucks or execute a single time for initial instructions and then rely on method truckNotification for further interaction.

Truck t calls truckNotification(Truck t, Notification message) when it does something of note, to ask for more instructions. For example, upon arrival at a new city, t may send a notification that there is a parcel at the city. Perhaps you want t to pick up that parcel before continuing on its route. For a full list of the reasons a truck would call this method, see Manager.Notification in the previous section and in the javadoc. This method is called by truck t in t's thread.

7. Concurrency

Because every Truck the manager interacts with exists in its own thread, Shipping Game is inherently multithread. Your manager has to be able to handle several trucks calling truckNotification concurrently, without crashing or giving incorrect instructions. That said, depending on the complexity of your solution, you may not need to worry about concurrency at all. For example, consider the following pseudo-code approach to the shipping game problem.

Algorithm 1 Basic Preprocessing

```
for Parcel p do
    Choose arbitrary truck t to deliver p. Store p in a data structure in t's user
    data.
end for
```

Algorithm 2 Basic Truck Notification (t)

```
if Basic Preprocessing not done then
    return // It's best always to start with this if-statement
end if
if there is an Undelivered Parcel in game then
    if t is holding a parcel then
        Route t to the parcel's destination, then drop off the parcel
    else
        find the next parcel assigned to t, route to that parcel.
    end if
else
    Route t home
end if
```

This algorithm makes use of every truck, but because no data structure is accessed by several trucks and all internal data structures are guaranteed to be thread-safe, there is no need for worry about concurrency. Assuming no errors, a correctly implemented version of this algorithm would get at minimum a good grade, likely higher.

If, however, you're shooting for a perfect grade, you may have to do some amount of work to ensure that your complex network of information sharing between trucks remains thread safe. Concurrent modifications of most data structures cause incorrect states, usually leading to null pointers or array-index-out-of-bound errors. Failure to write thread safe code might lead to unpredictable and catastrophic errors.

In this vein, you have a few different options to keep code thread-safe. The more powerful the tool, however, the greater the possibility of creating new and more difficult problems. Ideally, come up with a strategy, than pick the least-comprehensive tool that gives you just enough capability to prevent potential thread safety problems.

Figure 6: *Don't rely on Spongebob to keep your code thread safe.*

Approaches to Concurrency

- **wait() and notify()** - Power: medium. Possibility for Error: very high. Java's built-in concurrency system. Threads can cause themselves to wait, using any object as a key, and wake up other threads waiting on a given object. Working out a system of wait() and notify() calls that correctly limits thread movement is difficult. At the end of the day, some higher level systems are able to accomplish the same tasks in a much safer way. Don't use wait/notify unless you really want to get your hands dirty.

- **Synchronized Collections** - Power: low. Possibility for error: none. Class java.util.Collections (API) provides the ability to create a synchronized collection (a synchronized set, list, map, etc.). These collections are inherently thread-safe for basic operations, including adding elements, removing elements, getting elements, and checking the size of the collection. Notably, you can't iterate over a synchronized collection without a synchronized block (see next section), so complex operations that require iterating aren't safe. If you can get away with just using Synchronized Collections, do so and won't have any problems.

- **Synchronized Blocks** - Power: medium. Possibility for Error: low. A synchronized block limits one thread to executing its body at a time. Other threads wait on before the statement until the executing thread finishes the block. Moreover, each synchronized block is called on some object in the form:

```
synchronized(obj){
//...Do things that only one thread should be doing.
}
```

And, only one thread can access any synchronized block that is associated with that object at a time. This means that two blocks synchronized on the same object:

```
synchronized(obj){
    processA();
}
synchronized(obj){ //Note - synchronized on same obj
    processB();
}
```

cannot execute processA() and processB() concurrently because a thread's presence in one of the two blocks prevents other threads from being in either of the two blocks.

The simplest way to use synchronized blocks on Synchronized collections to allow iteration (See API). That way only one complex (iterative) process can occur on the collection at a time, and nothing can go wrong. **This combination is probably the best solution for this project and should be considered first**.

- **Locks and Locking** - Power: High. Possibility for Error: High
  Locks (and Semaphores and other equivalent classes) take the two pieces of a synchronized block —preventing other threads from accessing code and letting other threads know that the going is clear— and separate them into two individual lines of code. Only one thread can "own" a mutex (mutual exclusion object) at a time (semaphores are generalized to allow N threads), so when a thread acquires a Lock lk, all other threads that attempt to call lk.lock() are forced to wait until the thread that owns the lock calls lk.unlock(), at which point an arbitrarily chosen thread that is waiting will call lk.lock() and proceed. This gives you ultimate control over which threads execute what, where, and when, but it is prone to issues such as deadlock in which two threads are each waiting to acquire a lock owned by the other. Locks and Semaphores are overkill for all but the most complex solutions and are most likely not necessary for this project.

8. Grading

This project is due on the last day of class, Tuesday, 5 May. Submit a single .zip file that contains the source of package student only. All classes your code needs to run must be in package student. For example, if your submission requires three files, package student contains should contain these three files and the zip file you submit should contain only those three files. There should be NO DIRECTORIES in your submission. Failure to adhere to this structure will lead to grading of your Submission being delayed and/or a penalty for failing to submit properly.

Grading will be done at a Frame rate of 1ms (very fast) in headless mode, so when checking your scores make sure that you are running in headless mode, which defaults to 1ms. Running the game in GUI mode with a frame rate of 1ms may produce a close but slightly altered score.

In order to grade your projects in a timely fashion, it is necessary to enforce a timeout on runnings —some maximum amount of time to allow before killing the process. The main purpose of this mechanism is to prevent broken solutions that have entered an infinite loop or infinite recursion from causing the grading program to break. The exact amount of time allowed is based on the map size (maps with more parcels/longer edges have higher allotted time), but every map is given at least 10 seconds. This should be plenty of time for even more advanced solutions. If you are worried that your solution may be taking too long, run your solution in headless mode on the seed that worries you and make sure that it never times out. The GameRunner used by the headless mode is the same one used for grading.