COMP2010 Assignment 2: Peacekeeping Assignments

ver 1.03

Mark Dras

May 12, 2025

1 Introduction

Sometime in a magical long-ago, the Second Dark Lord has just been defeated in the land of New Mordor. His orcs have all surrendered. The surrounding nations have agreed to put peacekeeping forces into the major cities of New Mordor to make sure the threat of the Second Dark Lord does not arise again. However, because the source of the Second Dark Lord's power, the Throne of Bone,[1] has not been recovered, the surrounding nations are all suspicious of each other, and want to make sure that no one nation's peacekeeping forces are able to band together to find and seize the Throne.[2] To do this, they have all agreed that peacekeeping forces in a city must come from a different nation than the peacekeeping forces in any of the neighbouring cities.

Assignment Aim Your assignment will be involve assessing and proposing peacekeeping troop assignments, based on the details in §2, by completing the specified methods described in §3. You can use the accompanying sample JUnit tests as part of a check that you have done this correctly; the JUnit tests that will be used in the automarking will be similar (but e.g. applied to different data).

2 The Specifics

New Mordor is defined by CITIES (denoted by integers $1 \dots N$), where city t_i ($t_i \in \{1 \dots N\}$) might be directly connected to city t_j ($t_j \in \{1 \dots N\}$) by a single road, or might not be directly connected. If city t_i is directly connected to city t_j, they are referred to as being NEIGHBOURS. A PATH consists of a sequence of connected roads where each road is traversed only once.

The troops for the peacekeeping mission come from surrounding NATIONS (denoted also by integers, $1 \dots M$). A TROOP ASSIGNMENT is some assignment of troops where each city t_i ($t_i \in \{1 \dots N\}$) has troops from at most one nation n_j ($n_j \in \{1 \dots M\}$). We'll use the function[3] $\text{assign}(\cdot)$ to indicate this relationship, so that $\text{assign}(t_i) = n_j$ means that city t_i is assigned troops from nation n_j. A COMPLETE TROOP ASSIGNMENT is where each city t_i has troops from *exactly* one nation n_j.

A troop assignment (whether partial or complete) is VALID if, for all pairs of neighbouring cities t_i and t_j, $\text{assign}(t_i) \neq \text{assign}(t_j)$. That is, neighbouring cities don't have troops from the same nations. (If a city doesn't have assigned troops, it won't violate this condition.)

Consider the example in Figure 1. This figure contains data from a sample map input file, data from a sample partial assignment input file, and a visualisation of this. The city with the label 1 is connected by road to cities with the labels 2, 3 and 5, but no others directly. From the partial assignment data, the city with label 2 has been given an assignment of troops from nation 1; the city with label 4 has been given an assignment of troops from nation 4; and the city with label 6 has been given an assignment of troops from nation 3.

One possible COMPLETE VALID TROOP ASSIGNMENT for this example would be for city 1 to have troops from nation 2; city 3 to have troops from nation 5; city 5 to have troops from nation 4; and city 7 to also have troops from nation 4. This complete valid troop assignment is shown in the top left of Figure 2. An alternative complete valid troop assignment is shown in the top right of the same figure.

[1]Or maybe it was the Starlight Crown, or maybe the Rod of Compelling Destiny, or maybe something else entirely. No one is sure.
[2]Or Crown or Rod or Other Thing.
[3]Technically, this is a partial function, since not every city necessarily has troops.

```
1 2
1 3
1 5
2 3
2 7
3 4
3 6
3 7
4 6
5 6
```

```
2 1
6 3
4 4
```

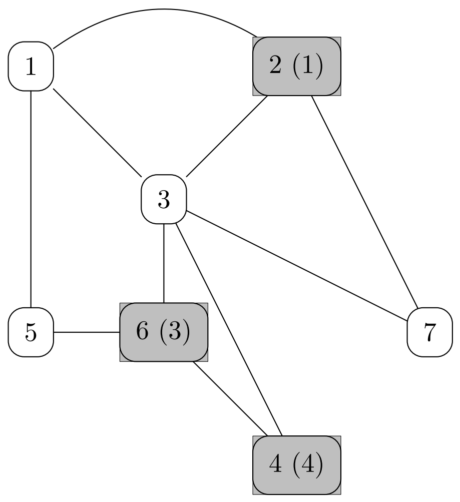

Figure 1: Sample data: (left) the format of the input map file; (middle) the form of the input partial assignment file; (right) a visualisation of this data. The input map file consists of a list of roads, defined by pairs of city numbers (so this example starts by defining a road between cities 1 and 2). The input partial assignment file consists of a list of city and troop nation pairings (so this example starts by defining an assignment of troops to city 2 from nation 1). The grey boxes in the visualisation indicate cities that are given assignments from the input partial assignment file. Troop assignments are indicated by numbers in parentheses.

An invalid troop assignment is shown in the bottom of Figure 2. In this, the assignment is invalid because neighbouring cities 2 and 3 both have troops assigned from nation 1.

In terms of the valid complete assignment examples, you'll see that the lefthand complete assignment uses troops from more nations (5) than the righthand one (4).

There are two more configurations of New Mordor with complete valid troop assignments in Figures 3 and 4.

3 Assignment Code Structure

Code Framework You will be working with a Java project that has 4 classes:

- `AssignControl`: This is the core class in the assignment, where all of the methods that you are required to complete are to be found, as well as one or more appropriate data members (class variables) to be chosen to store data. There are also some existing methods there related to some fundamental functions like reading in data from a file.
- `Road`: This is for storing data related to roads between cities, which defines the configuration of New Mordor. One instance of `Road` corresponds to a single road.
- `Assign`: This is for storing data related to assignments of troops to cities. One instance of `Assign` corresponds to the assignment to a single city.
- `FileNames`: This is for handling the naming schemes for input files. There are two kinds of input files, one with road definitions (in the form of pairs of Integers representing cities, as in the left of Figure 1), and one with initial assignments (also in the form of pairs of Integers, but here representing a city and a troop's nations, respectively, as in the middle of Figure 1).

For your tasks, you'll be adding attributes and methods to existing classes given in the code bundle accompanying these specs. Where it's given, **you should use exactly the method stub provided** for implementing your tasks. **Don't change the names or the parameters or exception handling.**

You can add more methods if you like. You can also add more classes if you like.

There is also a Java file of sample JUnit tests, `AllTests`. I encourage you to write your own additional JUnit tests.

Timing For the code to be completed below, it would be possible to write these in inefficient and/or time-consuming ways. **No method should run for more than 1 minute, and all tests together should run for no more than 3 minutes.** This probably won't be a problem. (My sample solution runs in milliseconds in aggregate.) Just in case you're worried that yours might exceed these limits, there's an example of how to use a stopwatch in Java included in the

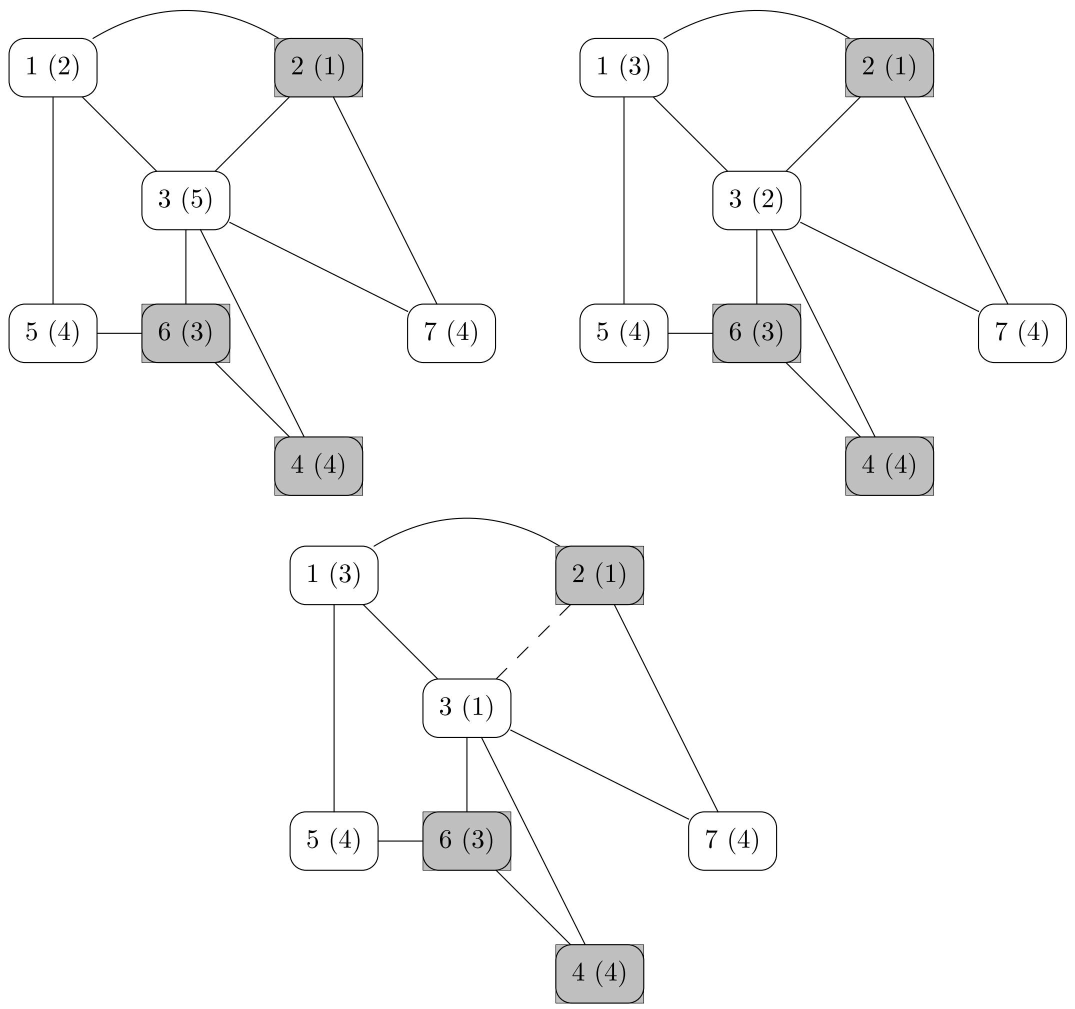

Figure 2: Sample complete troop assignments from the setup in Figure 1. The top two are valid assignments, but the bottom one is not; the problem there is the clash between between city 2 and city 3 (dotted line), both having troops from nation 1.

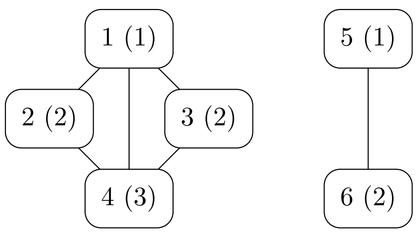

Figure 3: A second (small) configuration for New Mordor, with a valid complete troop assignment.

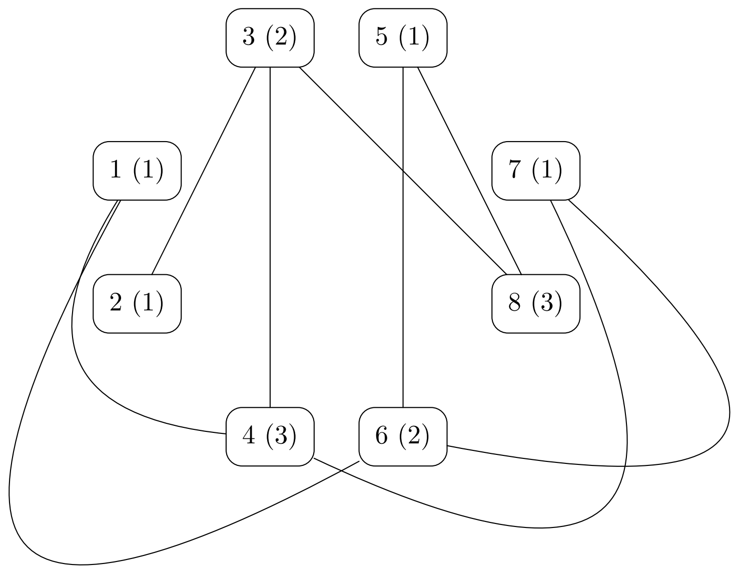

Figure 4: A third configuration for New Mordor, with a valid complete troop assignment.

`main()` of `AssignControl`.[4]

3.1 Pass Level

To achieve at least a Pass ($\geq 50\%$) for the assignment, you should do all of the following tasks, which involve filling in class `AssignControl`.

T1 The class needs one or more data members (class variables) to store data. Declare appropriate data member(s) in the class definition. Also define a suitable constructor that appropriately initialises the data members.

```
public AssignControl() {
// Constructor
}
```

T2 Given your chosen data members (class variables) from Task **T1**, write the following method to instantiate New Mordor with cities, roads, and initial assignments (i.e., those given in files).

```
public void instantiateMap(Vector<Road> roads, Vector<Assign> assigns) {
// PRE: -
// POST: New Mordor configuration is set up, including initial assignments
}
```

T3 Write the following methods for making and fetching a troop assignment for an individual city, as well as whether an individual city has a troop assignment or not.

```
public void setAssign(Integer city, Integer nation) {
// PRE: city exists
// POST: city is assigned troops from nation assign
}
```

```
public Integer getAssign(Integer city) {
// PRE: city exists
// POST: Returns nation that city is assigned troops from,
//         null if none
}
```

```
public Boolean isAssigned(Integer city) {
// PRE: city exists
// POST: Returns True if city has been assigned troops, false otherwise
}
```

T4 Write the following methods for verifying whether two cities are neighbours, and for returning a list of a given city's neighbours.

```
public Boolean isNeighbour(Integer city1, Integer city2) {
// PRE: city1, city2 exist
// POST: Returns True if city1 is a neighbour of city2, false otherwise
}
```

```
public Vector<Integer> getNeighbours(Integer city) {
// PRE: city exists
// POST: Returns vector of cities that are neighbours of city
//         (empty vector if no neighbours)
}
```

T5 Write the following method to determine if two cities have been assigned troops from the same nation.

```
public Boolean isAssignedSame(Integer city1, Integer city2) {
// PRE: city1, city2 exist
// POST: Returns True if city1 and city2 are assigned troops from the same nation,
//         False otherwise
}
```

T6 Write the following method for determining whether a (partial or complete) troop assignment is valid or not (i.e., has no violated constraints).

[4] There's more at `https://www.baeldung.com/java-measure-elapsed-time` if you're interested.

```
public Boolean isValidAssign() {
// PRE: -
// POST: Returns True if troop assignment is valid, False otherwise
}
```

T7 Write the following methods.

```
public Integer numDiffAssigns() {
// PRE: -
// POST: Returns the number of different nations from which troops are assigned
}
```

In the example configurations given above, `numDiffAssigns()` should give 3 for the partial initial assignment of Figure 1, 5 for the complete assignment at the top left in Figure 2, 4 for the complete assignment at the top right in Figure 2, 3 for the complete assignment in Figure 3, and 3 for the complete assignment in Figure 4.

```
public Boolean isEveryCityAssigned() {
// PRE: -
// POST: Returns True if every city is assigned some troops, False otherwise
}
```

T8 Write the following method.

```
public void giveAnyAssignment() {
// PRE: -
// POST: Gives a valid assignment of troops to every city that does not already have troops assigned
}
```

The top two assignments in Figure 2 would both be valid solutions for this method given the initial assignments of Figure 1.

3.2 Credit/Distinction Level

To achieve at least a Credit ($\geq 65\%$) for the assignment, you should complete the following tasks. You should also have completed all the Pass-level tasks.

T9 Write the following method.

```
public Boolean existsPath(Integer city1, Integer city2) {
// PRE: city1 and city2 exist
// POST: Returns True if there is a path between city1 and city2, False otherwise
}
```

For the example in Figure 3, this method should return True for `existsPath(1, 2)` and False for `existsPath(1, 5)`.

T10 Write the following methods for making valid assignments to individual cities.

```
public void assignCity(Integer city) {
// PRE: city not assigned
// POST: Troops from some nation are validly assigned to city
}
```

Starting with the initial partial assignment of Figure 1, city 1 could be assigned troops from nations 2, 3, 4, ... (but not nation 1, because city 1 is a neighbour of city 2, which is already assigned troops from nation 1).

```
public void assignCityLowest(Integer city) {
// PRE: city not assigned
// POST: Troops from the lowest possible numbered nation are validly assigned to city
}
```

Again starting with the initial partial assignment of Figure 1, under `assignCityLowest()` city 1 could only be assigned troops from nation 2.

T11 Write the following method.

```
public void giveGreedyCityOrderingAssignment() {
// PRE: -
// POST: Assigns troops greedily, starting at city1 and continuing to cityN,
//         subject to constraints
}
```

If the method `giveGreedyCityOrderingAssignment()` above were applied to the city network in Figure 3 starting from a null initial assignment, you would get the assignment given in Figure 3.

T12 Write the following method. It differs from the one in Task **T11** in the ordering chosen for considering cities for troop assignment.

```
public void giveGreedyRoadOrderingAssignment() {
// PRE: -
// POST: Assigns troops greedily, starting at with most roads and working through to least
//         (breaking ties by city ordering, so city i before city j if same number of
//         roads and i < j), subject to constraints
}
```

If the method `giveGreedyRoadOrderingAssignment()` above were applied to the city network in Figure 3 starting from a null initial assignment, the assignment would be similar to the one in Figure 3 except that cities 2 and 3 would be assigned troops from nation 3, and city 4 from nation 3. This is because the cities would be considered in the order $1 - 4 - 2 - 3 - 5 - 6$.

3.3 (High) Distinction Level

To achieve at least a mid-level Distinction ($80 - 100\%$) for the assignment, you should do the following tasks. You should also have completed all the Credit-level tasks.

T13 Write the following method.

```
public Boolean canDoWithNSources(Integer N) {
// PRE: -
// POST: Returns True if there is some complete and valid assignment of troops
//         that uses at most N nations,
//         instantiating the map with such a complete and valid assignment;
//         False otherwise
}
```

For the city and road network in Figure 3, `canDoWithNSources(3)` would be True (the assignment given the figure shows one assignment that uses troops from 3 nations), but `canDoWithNSources(2)` would be False (you can confirm for yourself that it's not possible to use only 2 nations).

T14 Write the following method. The notion of a better solution here is one that uses troops from fewer nations.

```
public Boolean canFindBetterSoln() {
// PRE: There is some initial complete assignment of troops
// POST: Returns True if there is some complete and valid assignment of troops
//         that is better than the initial,
//         instantiating the map with such a complete and valid assignment;
//         False otherwise
}
```

For the city and road network in Figure 2, there are better solutions than the one at the top left, which uses troops from 5 nations; one such better solution is at the top right, which uses troops from only 4 nations.

T15 Write the following method.

```
public Vector<Integer> pathWithAssign(Integer city1, Integer city2, Integer troopSource) {
// PRE: city1, city2 are existing cities; troopSource is an existing nation;
//        there is a complete valid assignment already
// POST: Returns a path between city1 and city2 that has at least one city assigned from troopSource,
//         if such a path exists, as a vector;
//         returns an empty vector otherwise
}
```

In Figure 4, a possible return value for `pathWithAssign(1, 5, 3)` would be the path $1-4-7-6-5$, as city 4 has troops assigned from nation 3. (Given the definition of a path in Section 2, $1-4-7-4-3-8-5$ is not a path, as there is backtracking along the road between cities 4 and 7.)

T16 Write the following method. It differs from Task **T15** in that there isn't a complete valid assignment already. (Note that you shouldn't override any existing initial assignments.) `troopSource` doesn't have to have already been used in any initial assignment.

```
public Vector<Integer> makePathWithAssign(Integer city1, Integer city2, Integer troopSource) {
// PRE: city1, city2 are existing cities; troopSource is an existing nation;
//       there is NOT a complete valid assignment already
// POST: Returns a path between city1 and city2 that has at least one city assigned from troopSource,
//        if such a path exists, as a vector;
//        returns an empty vector otherwise
}
```

4 The Code Bundle You'll Be Working With

There's a code bundle that contains the classes described in §3 that you need to complete. The code bundle has been saved as an archive zipfile (from the Eclipse IDE), so you should just need to open it as an archive file.

There are also some input data files containing specifications sets of players. Some of the JUnit tests in `AllTests.java` refer to a file location where you will store these files. The default location is from my own installation; you'll need to change it to yours. If you're on a Windows machine, it might look something like:

```
String dataDir = "C:\\Users\\YourAccountName\\path\\to\\your\\comp2010\\data"
```

At the start, the JUnit tests will give you all failures and errors, because the methods still have to be implemented.

The methods with non-void return types start with `return null` statements; these are just to make sure that the code compiles when you make a submission for any methods that you haven't completed. You should of course change this statement when you complete a method.

Sample Input Files On assignment release, there will be 4 sample inputs provided:

- `sample1`: Configuration like Figure 3, but with only cities $1, 2, 3, 4$, plus an initial troop assignment to city 2 from nation 1.
- `sample2`: Configuration in Figure 4.
- `sample3`: Configuration in Figure 4, with initial assignment for all cities.
- `sample4`: Configuration in Figure 3.

5 What To Hand In

In the submission page on iLearn for this assignment you must include the following:

Submit a zip file consisting of all the Java classes in the package from the original assignment code bundle. (Note that this is different from Assignment 1.) The Java classes should be in the form of the `.java` files.

You don't have to submit a JUnit test file. The automarker will use its own JUnit test file which will be similar but not identical to `AllTests.java`. (In particular, it will use additional data).

Your file must leave unchanged the specification of already implemented functions, and include your implementations of your selection of method stubs outlined above.

Do not change the names of the method stubs because the auto-tester assumes the names given. Do not change the package statement. You may however include additional auxiliary methods if you need them.

Please note that we are unable to check individual submissions and so it is very important to abide by the above submission instructions.