Assignment 2

Detective Academy 2521

Changelog

All important changes to the assignment specification and files will be listed here.

- [11/07 09:00] Assignment released
- [13/07 21:00]
  - Added note to game.c about not relying on the #defines in game.c
  - Updated test for MapSetName in testMap.c to reinforce that a copy of the given name should be made. See the most recent version of testMap.c here.

Admin

Marks	contributes 20% towards your final mark (see Assessment section for more details)
Submit	see the Submission section
Deadline	8pm on Friday of Week 10
Late penalty	0.2% per hour or part thereof deducted from the attained mark, submissions later than 5 days not accepted

- Graphs
  - Graph ADT
  - Graph Traversal
  - Weighted Graphs
    - Shortest Path

Background

In this assignment, you are the police! You will control four detectives as they travel through a network of cities trying to catch a thief.

The police are aiming for any of the four detectives to catch the thief before the thief escapes to the getaway city, and before the time runs out... and the aim of the thief is to reach the getaway city before they are caught.

The detectives have a map, but do not know where the thief is or where they are trying to get to. The thief also has a map but unfortunately they didn't take COMP2521, so they don't really know how to use it and they wander randomly through the cities trying to reach the getaway city. The detectives may employ different strategies depending on what they have been assigned to.

Game Rules

In this game, all the people (the four detectives and the thief) are known as *agents*, and the game consists of a series of turns, known as cycles.

Each agent starts in a city, determined by user configuration. Every cycle, each agent may move from their current city to an adjacent city by road. The goal of the detectives is to end up in the same city as the thief, which would allow them to catch the thief, while the goal of the thief is to reach the getaway city.

the length of the road between them.

Agents cannot travel along a road if they do not have the required level of stamina. This means it is possible for an agent to have no legal moves. If an agent has no legal moves due to not having enough stamina, they must remain in their current city for another cycle. Remaining in the same city for a turn resets the agent's stamina back to its initial level.

Each detective uses a set strategy to navigate the cities, determined by user configuration. Meanwhile, the thief always moves randomly.

The game ends if one of the following conditions is met:

- If a detective starts in the same city as the thief, the thief is caught immediately and the detectives win.
- If a detective is in the same city as the thief at the end of a turn the thief is caught and the detectives win.
- If the thief is in the getaway city at the end of a turn and there are no detectives there, the thief escapes, so the thief wins.
- If the time has run out, regardless of whether the thief was able to reach the getaway city, the trail has gone cold, so the thief wins.

Setting Up

Note: As this assignment uses random number generation, you may get different results if you run it on your local machine.

Change into the directory you created for the assignment and run the following command:

```
$ unzip /web/cs2521/23T2/ass/ass2/downloads/files.zip
```

You should now have the following files:

`Makefile`	This controls compilation of the program. You only need to modify this if you create additional files for your implementation.
`Map.h`	This is the interface to the Map ADT. It provides agents with information about the world including what cities and roads there are. Roads always go between two different cities and can always be traversed in both directions, and all the cities will be connected, either directly or indirectly via other cities. **You must not modify this file.**
`Map.c`	This is the implementation of the Map ADT. At the moment it is just a stub file that you need to implement yourself. You can use any of the lab code or adapt any of the code from lectures to do this.
`Agent.h`	This is the interface to the Agent ADT. An instance of the Agent ADT represents an 'agent' in the game. An agent represents a detective or a thief. This interface includes functions to allow the agents to interact with the client program. **You must not modify this file.**
`Agent.c`	This is the implementation of the Agent ADT. At the moment this only supplies the implementation for the `RANDOM` movement strategy, and will need to be completed by you. Make sure you understand what has already been supplied.
`testMap.c`	This is a main program for testing the Map ADT. You can modify this file to add more extensive tests.

	creates a map. The agents are also created and a game starts. **You must not modify this file.**
`data/`	This is a directory containing test data that can be used as input to the program. There are two types of data files, which are described below.
`cities*.data` **files**	Sample city data files to use as a starting point for your testing. Some data files will have small numbers of cities, some will have more; some have informants, some don't... but you should also create your own data files.
`agentsS*.data` **files**	Sample agent data files that you can use once you have completed the different stages of the assignment. `agentsS1.data` can be used if you have implemented stage 1 and above, `agentsS2.data` can be used if you have implemented stage 2 and above, and so on. Note that stage 3 must be tested by using one of the city data files with informants. And, of course, you should create your own `agents*.data` files for testing.

Note that you are permitted to create supporting files for Task 2. To ensure that these files are actually included in the compilation, you will need to edit the Makefile; the provided Makefile contains instructions on how to do this. Supporting files are not permitted for Task 1.

The Client Program

Inputs

```
./game <city data file> <agent data file> <cycles> [seed]
```

The program requires 3 command-line arguments with an optional fourth. The command-line arguments are

1. the name of the city data file
2. the name of the agent data file
3. the maximum number of cycles in the game
4. *(optional)* a seed value for the random number generator; by using the same seed, you can produce the same ordering of 'random' moves and repeat exactly the same situation.

City Data

The first line contains a single integer which is the number of cities. Then, for every city there will be a line of data. Each line begins with the ID of the city, which will always be between 0 and (the number of cities - 1), followed by pairs of integers indicating a road to another city of a certain length. After the roads are listed each line will contain either an 'n' or 'i'. An 'i' indicates that the city has an informant, while an 'n' indicates that it doesn't. At the end of each line is the name of the city.

For example, consider the following city data file:

```
10
0 5 29 1 41 6 60 7 50 8 40 n sydney
1 2 51 5 29 i adelaide
2 n melbourne
3 5 30 4 36 n perth
4 9 20 n darwin
5 6 10 n hobart
6 n auckland
7 n madrid
8 i new york
9 n brisbane
```

Adelaide of length 41, a road to Auckland of length 60, a road to Madrid of length 50 and a road to New York of length 40. There is an informant in Adelaide and New York.

Note that roads are bidirectional, so even though some cities do not have any roads in their lists in the city data file, they will still have roads to other cities due to them appearing in other cities' road lists. All cities are connected, either directly or indirectly via other cities.

Agent Data

The first line of data represents information about the thief. The first number represents the amount of stamina the thief starts with, which is also the maximum amount of stamina the thief can have. The second number represents the starting location of the thief. The third number indicates where the getaway city is. This is followed by a string representation (i.e., name) of the thief.

The next four lines represent the detectives. The first two numbers represent the initial/maximum amount of stamina and the starting location of the detective. The third number represents the strategy that the detective is assigned. This is followed by a string representation (i.e., name) of the detective.

For example, consider the following agent data file:

```
10 5 6 T
1 2 0 D1
1 3 1 D2
5 1 2 D3
5 4 2 D4
```

The first line indicates that the thief has a stamina of 10 and starts at city 5 (which would be Hobart if using the city data file above), and that the getaway city is city 6 (which would be Auckland if using the city data file above).

Commands

as follows:

Command	Description
`run`	This will run an entire simulation, printing out a trace of the agents' locations for each cycle of the game. It will print out how the game finished, i.e., with the thief being caught, getting away, or time running out.
`step`	This runs just one cycle of the game, printing out the new location of the agents for the next cycle. If the game finished in that cycle, it will also print out how the game finished. This allows the user to step through the game one cycle at a time.
`stats`	This prints out the status of each agent. This includes the name of the agents' current location and the agents' stamina.
`display`	This displays the current locations of all agents.
`map`	This prints out the map in a textual format, including the ID/name of each city, and the roads from each city and their length.
`quit`	Quits the game!

Task 1: Map Implementation

Your first task is to complete the implementation of the Map ADT in `Map.c`, which agents will use to get information about cities and roads.

The interface functions of the Map ADT are as follows:

Function	Description	Assumptions
`MapNew`	Creates a new map with the given number of cities and no	The given number of cities is positive

MapFree	Frees all memory allocated to the given map	
MapNumCities	Returns the number of cities on the given map	
MapNumRoads	Returns the number of roads on the given map	
MapSetName	Sets the name of the given city. If the city's name has already been set, renames it to the given name.	The given city and name are valid
MapGetName	Returns the name of the given city, or "unnamed" if the city's name has not been set (via MapSetName)	The given city is valid
MapInsertRoad	Inserts a road between two cities with the given length. Does nothing if there is already a road between the two cities.	The given cities are valid The given cities are the not the same The road length is positive
MapContainsRoad	Returns the length of the road between the two given cities, or 0 if no such road exists	The given cities are valid
MapGetRoadsFrom	Stores the roads connected to the given city in the given roads array in any order and returns the number of roads stored. The from field should be equal to the	The given city is valid The roads array is at least as large as the number of cities on the map

	array.	
`MapShow`	Displays the map (already implemented)	

Task 2: Agent Implementation

In this task, you must implement various strategies that the detectives can use to try and catch the thief in `Agent.c`. This will require you to modify existing functions and add new fields to the agent struct (to enable agents to *remember* things), so make sure you understand the existing code first.

Stage 0: `RANDOM` strategy

In stage 0, all agents use the **random strategy**. In the random strategy, each agent randomly selects an adjacent city that they have the required stamina to move to and move to it. If the agent does not have sufficient stamina to move to any city, they must remain in their current city for another cycle, which will completely replenish their stamina.

The random strategy has already been implemented, so you are not required to do anything to complete this stage. You should not alter the logic of the random strategy in `Agent.c`. You should also not use any random number generation in your implementation of the other strategies.

Stage 1: `CHEAPEST_LEAST_VISITED` strategy

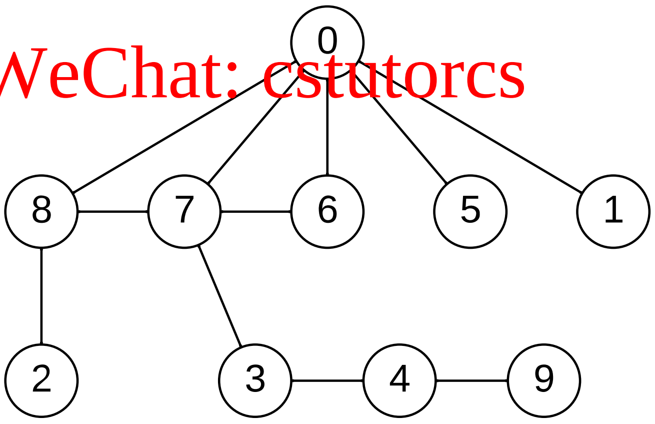
If a detective is assigned this strategy, it means that at every opportunity they have to move, they must move to the city they have visited the least number of times, out of the legal options that are available. This means the detective must work out what cities are actually adjacent to the current city that they have sufficient stamina to move to and pick from those the one that has been visited the least. If there is more than one city with the least number of visits, the city stamina, the city with the lowest ID among those should be chosen.

Note that at the beginning of the game, a detective is considered to have visited their starting city once. Also, if a detective must remain in their current city, this counts as an additional visit, even though the detective did not move.

Stage 2: `DFS` strategy

In this stage a DFS strategy must be implemented. When following this strategy, the detective maps out an entire route that will take them through every city on the map using the DFS algorithm. If the DFS has a choice between multiple cities, it must prioritise the city with the lowest ID. At every cycle, the detective attempts to move to the next city on the plan. If the detective does not have enough stamina, they must wait in the same city to recover. As soon as the detective has visited all cities at least once, a new DFS path from the final location is mapped out and is followed.

For example, consider the following arrangement of cities and roads:



If a detective using the `DFS` strategy starts at city 5, then they should devise the following route: 5 → 0 → 1 → 0 → 6 → 7 → 3 → 4 → 9 → 4 → 3 → 7 → 8 → 2. The route ends at city 2 because once the detective reaches city 2, they will have visited all the cities, and the next DFS would begin at city 2.

You can assume that the maximum stamina of each detective using the `DFS` strategy is greater than or equal to the length of the longest road, so no detective using the `DFS` strategy will be stuck forever at some city while trying to complete their route.

If a detective starts at, or moves to a city where there is an informant, they will discover where the thief is currently located (this is achieved by the `AgentTipOff` function being called). The detective must then find the path to that location that will take the least number of turns and then follow this path. Of course, the thief may be gone by the time the detective gets there, in which case the detective must restart their original strategy from their new location. Any cities the detective passes through on the shortest path are counted as being visited if the detective returns to the `CHEAPEST_LEAST_VISITED` strategy. The detective may also pass through a city with another informant in which the detective would find a new least turns path from the current location.

You must take into account the stamina of the detective. For example, if one path requires the detective to travel through 3 cities, but would have to rest twice (5 turns), that is more turns than the detective travelling through 4 cities but not having to rest (4 turns). If there are multiple paths that would take the least number of turns, the path that results in the agent having the most stamina at the end should be chosen. If there are multiple paths that would take the least number of turns and would also result in the agent having the same stamina, any of them may be chosen.

You can assume that all detectives will be able to reach every city from every other city. That is, for every pair of cities, there exists a route between them such that the length of the longest road on that route is less than or equal to the maximum stamina of each detective.

Examples

Stage 0

```
$ ./game data/citiesSmall.data data/agentsS0.data 10 6

DETECTIVE ACADEMY 2521

Red alert! A thief is on the run.
Detectives, to your cars. You have 10 hours.

Hour 0
  T  D1  D2  D3  D4
  3   1   1   1   1

Enter command: map
Number of cities: 4
Number of roads: 3
[0] melbourne has roads to: [1] perth (41), [2] darwin (900)
[1] perth has roads to: [0] melbourne (41)
[2] darwin has roads to: [0] melbourne (900), [3] california
(30)
[3] california has roads to: [2] darwin (30)

Enter command: stats
Hour 0
T is at [3] california with 1000 stamina
D1 is at [1] perth with 100000 stamina
D2 is at [1] perth with 5000 stamina
D3 is at [1] perth with 50 stamina
D4 is at [1] perth with 500 stamina

Enter command: step
Hour 1
  T  D1  D2  D3  D4
  2   0   0   0   0

Enter command: stats
Hour 1
T is at [2] darwin with 970 stamina
```

```
D4 is at [0] melbourne with 459 stamina

Enter command: step
Hour 2
  T  D1  D2  D3  D4
  3   1   2   0   1

Enter command: stats
Hour 2
T is at [3] california with 940 stamina
D1 is at [1] perth with 99918 stamina
D2 is at [2] darwin with 4059 stamina
D3 is at [0] melbourne with 50 stamina
D4 is at [1] perth with 418 stamina

Enter command: display
Hour 2
  T  D1  D2  D3  D4
  3   1   2   0   1

Enter command: run
Hour 3
  T  D1  D2  D3  D4
  2   0   3   1   0

Hour 4
  T  D1  D2  D3  D4
  0   1   2   1   1

T got away to melbourne (0)
GAME OVER: YOU LOSE - THIEF GOT TO GETAWAY
```

Stage 1

In this example, all detectives are using the `CHEAPEST_LEAST_VISITED` strategy.

```
DETECTIVE ACADEMY 2521

Red alert! A thief is on the run.
Detectives, to your cars. You have 10 hours.

Hour 0
  T  D1  D2  D3  D4
  9   8   7   6   2

Enter command: run
Hour 1
  T  D1  D2  D3  D4
  4   0   0   5   1

Hour 2
  T  D1  D2  D3  D4
  9   5   5   0   5

Hour 3
  T  D1  D2  D3  D4
  4   6   6   8   6

Hour 4
  T  D1  D2  D3  D4
  3   5   5   0   0

Hour 5
  T  D1  D2  D3  D4
  5   6   1   1   8

Hour 6
  T  D1  D2  D3  D4
  1   6   2   2   0

Hour 7
  T  D1  D2  D3  D4
  0   0   1   1   7
```

Stage 2

In this example, detectives 3 and 4 use the DFS strategy. In this strategy they do a depth-first traversal from their starting points and follow this at each cycle. If they do not have the stamina to go to the next city on their route, they remain at the same location to regain their stamina and continue on the set path. (This happens even if there were other options they did have the stamina for.)

Detectives 1 and 2 are still following the `CHEAPEST_LEAST_VISITED` strategy and the thief is of course using the `RANDOM` strategy.

```
$ ./game data/citiesMedium.data data/agentsS2.data 10 2

DETECTIVE ACADEMY 2521

Red alert! A thief is on the run.
Detectives, to your cars. You have 10 hours.

Hour 0
  T   D1   D2   D3   D4
  9    2    8    1    2

Enter command: run
Hour 1
  T   D1   D2   D3   D4
  4    1    0    0    1

Hour 2
  T   D1   D2   D3   D4
  9    5    5    5    0

Hour 3
  T   D1   D2   D3   D4
  4    6    6    3    0
```

```
   9   5   5   4   5

Hour 5
   T   D1  D2  D3  D4
   4   5   1   9   3

Hour 6
   T   D1  D2  D3  D4
   4   0   2   4   4

D3 caught the thief in darwin (4)
YOU WIN - THIEF CAUGHT!
```

Stage 3

In this example, detectives 1 and 2 start with the `CHEAPEST_LEAST_VISITED` strategy and detectives 3 and 4 start with the `DFS` strategy. However detectives 1 and 4 start off in a city with an informant (this is indicated by the `*` character on the display), so they immediately switch to the strategy of going along the shortest path to the thief's current location (city 9). They calculate the shortest path as follows:

- Detective 1 calculates the shortest path as 8 → 0 → 5 → 3 → rest → 4 → 9, which requires 6 turns.
- Detective 4 calculates the shortest path as 8 → 0 → 5 → 3 → 4 → 9, which requires 5 turns.

Detective 4 reaches the destination by hour 5 but the thief is no longer there. However the other detectives have found the thief by this stage anyway. Detective 1 is following the shortest path from 8 to 9, but has to stop at hour 4 to regain stamina.

Detective 2 finds an informant in city 1 in hour 1 and goes via the shortest path from city 1 to 4, which is calculated as 1 → 5 → rest → 3 → 4.

```
$ ./game data/citiesInformants.data data/agentsS3.data 10 2

DETECTIVE ACADEMY 2521

Red alert! A thief is on the run.
Detectives, to your cars. You have 10 hours.

Hour 0
  T   D1   D2   D3   D4
  9    8*   2    6    8*

Enter command: run
Hour 1
  T   D1   D2   D3   D4
  4    0    1*   0    0

Hour 2
  T   D1   D2   D3   D4
  9    5    5    0    5

Hour 3
  T   D1   D2   D3   D4
  4    3    5    1*   3

Hour 4
  T   D1   D2   D3   D4
  9    3    3    5    4

Hour 5
  T   D1   D2   D3   D4
  4    4    4    3    9

D1 caught the thief in darwin (4)
YOU WIN - THIEF CAUGHT!
```

Task 1

Task 1 can be tested in isolation using the `testMap.c` program.

To run these tests, first run `make`, which compiles your code and creates an executable called `testMap`, and then run `./testMap`. The program contains assert-based tests, which means that the program will exit as soon as a test fails.

As given, the `testMap.c` program contains very basic tests - it is recommended that you add more extensive tests. You can add more tests by creating additional functions in `testMap.c` and then calling them from the `main` function.

Task 2

Task 2 can be tested using the `./game` program.

To find out how to run the program, see "The Client Program" section of the spec. There are also example runs of the program in the "Example" section.

You can devise more test cases by creating new city data files and agent data files. It is recommended that you first draw out a map on paper (to double check that it actually tests the scenario that you want to test) and then create the corresponding city and agent data files.

A reference implementation is also available at:

```
/web/cs2521/23T2/ass/ass2/reference/game
```

Note that the `./game` program uses random number generation (for the `RANDOM` strategy), so to properly compare your program with the reference implementation, you must run your program on CSE machines.

If the reference implementation fails (e.g., segmentation fault, bus error, etc.) or behaves incorrectly according to the specification, please email

Debugging

Debugging is an important and inevitable aspect of programming. We expect everyone to have an understanding of basic debugging methods, including using print statements, knowing basic GDB commands and running Valgrind. In the forum and in help sessions, tutors will expect you to have made a reasonable attempt to debug your program yourself using these methods before asking for help.

You can learn about GDB and Valgrind in the Debugging with GDB and Valgrind lab exercise.

Frequently Asked Questions

- **Do the detectives communicate with each other at all?** No. Each detective operates independently of the other detectives.
- **Are we allowed to create our own functions?** You are always allowed to create your own functions. All additional functions you create should be made `static`.
- **Are we allowed to create our own `#define`s and structs?** Yes.
- **Are we allowed to `#include` any other libraries?** No. All the libraries required for this assignment are provided already.
- **Are we allowed to change the signatures of the given functions?** No. If you change these, your code won't compile and we won't be able to test it.
- **What errors do we need to handle?** You should handle common errors such as `NULL` returned from `malloc` by printing an error message to

- **What invalid inputs do we need to handle?** You are not required to handle invalid inputs, such as `NULL` being passed in as a Map or Agent. It is the user's responsibility to provide valid inputs.
- **Will we be assessed on our tests?** No. You will not be submitting any test files, and therefore you will not be assessed on your tests.
- **Are we allowed to share tests?** No. Testing is an important part of software development. Students that test their code more will likely have more correct code, so to ensure fairness, each student should independently develop their own tests.

Submission

You must submit the files `Map.c` and `Agent.c`. In addition, you can submit as many supporting files (`.c` and `.h`) as you want. Please note that none of your submitted files should contain a main function, and you must not submit `Map.h` or `Agent.h` as you are not permitted to modify these files. You can submit via the command line using the `give` command:

```
$ give cs2521 ass2 Map.c Agent.c your-supporting-files...
```

You can also submit via give's web interface. You can submit multiple times. Only your last submission will be marked. You can check the files you have submitted here.

WARNING:

After you submit, you **must** check that your submission was successful by going to your submissions page. Check that the timestamp and files are correct. If your submission does not appear under Last Submission or the timestamp is not correct, then resubmit.

Submitting non-compiling code leads to extra administrative overhead and will result in a 10% penalty.

Every time you make a submission, a dryrun test will be run on your code to check that it compiles. Please ensure that your final submission successfully compiles, even for parts that you have not completed.

Assessment Criteria

This assignment will contribute 20% to your final mark.

Correctness

80% of the marks for this assignment will be based on the correctness of your code, and will be based on autotesting. Marks for correctness will be distributed as follows:

Task		Weight
Map Implementation		15%
Agent Implementation	Stage 1	20%
	Stage 2	25%
	Stage 3	20%

There are no formal time complexity requirements. However, each test will still be run under time constraints, and solutions that are slower than $O(n^4)$, where n is the number of cities, will risk timing out.

Memory errors/leaks

You must ensure that your code does not contain memory errors or leaks, as code that contains memory errors is unsafe and it is bad practice to leave memory leaks in your program.

10% of the marks for that part. For example, the penalty for causing a memory error/leak in Stage 1 will be 2%. Note that our tests will always call `MapFree` at the end of the test (and `AgentFree` if appropriate) to free all memory allocated to the map and agent.

Style

20% of the marks for this assignment will come from hand marking of the readability of the code you have written. These marks will be awarded on the basis of clarity, commenting, elegance and style. The following is an indicative list of characteristics that will be assessed, though your program will be assessed wholistically so other characteristics may be assessed too (see the style guide for more details):

- Consistent and sensible indentation and spacing
- Using blank lines and whitespace
- Using functions to avoid repeating code
- Decomposing code into functions and not having overly long functions
- Using comments effectively and not leaving planning or debugging comments

The course staff may vary the assessment scheme after inspecting the assignment submissions but it will remain broadly similar to the description above.

Originality of Work

This is an individual assignment. The work you submit must be your own work and only your work apart from the exceptions below. Joint work is not permitted. At no point should a student read or have a copy of another student's assignment code.

comment.

You may use small amounts (< 10 lines) of general purpose code (not specific to the assignment) obtained from sites such as Stack Overflow or other publicly available resources. You must clearly attribute the source of this code in a comment.

You are not permitted to request help with the assignment apart from in the course forum, help sessions or from course lecturers or tutors.

Do not provide or show your assignment work to any other person (including by posting it publicly on the forum) apart from the teaching staff of COMP2521. When posting on the course forum, teaching staff will be able to view the assignment code you have recently submitted with give.

The work you submit must otherwise be entirely your own work. Submission of work partially or completely derived from any other person or jointly written with any other person is not permitted. The penalties for such an offence may include negative marks, automatic failure of the course and possibly other academic discipline. Assignment submissions will be examined both automatically and manually for such issues.

Relevant scholarship authorities will be informed if students holding scholarships are involved in an incident of plagiarism or other misconduct. If you knowingly provide or show your assignment work to another person for any reason, and work derived from it is submitted, then you may be penalised, even if the work was submitted without your knowledge or consent. This may apply even if your work is submitted by a third party unknown to you.