RMIT UNIVERSITY

Advanced Programming Techniques
COSC1076 | Semester 2 2020
Assignment 2 (v1.1) | Azul

Assessment Type	Group Assessment (Milestones 1, 2 & 4), Individual Assessment (Milestone 3). Clarifications/updates may be made via announcements/relevant discussion forums.
Due Date: Milestone 1	11.59pm Sunday 20 September 2020 (Week 8)
Due Date: Milestone 2	11.59pm Sunday 4 October 2020 (Week 10)
Due Date: Milestone 3	11.59pm Sunday 18 October 2020 (Week 12)
Due Date: Milestone 4	11.59pm Sunday 18 October 2020 (Week 12)
Silence Policy	From 2 business days before each due date
Weight (Group Component)	30% of the final course mark
Weight (Individual Component)	15% of the final course mark
Weight (Total)	45% of the final course mark
Submission	Online via Canvas. Submission instructions are provided on Canvas.

Change Log

1.1
- Enhancements Released
- Specified tile bag format for saved-game

1.0
- Initial Release
- Enhancements to be released at a later date

1 Overview

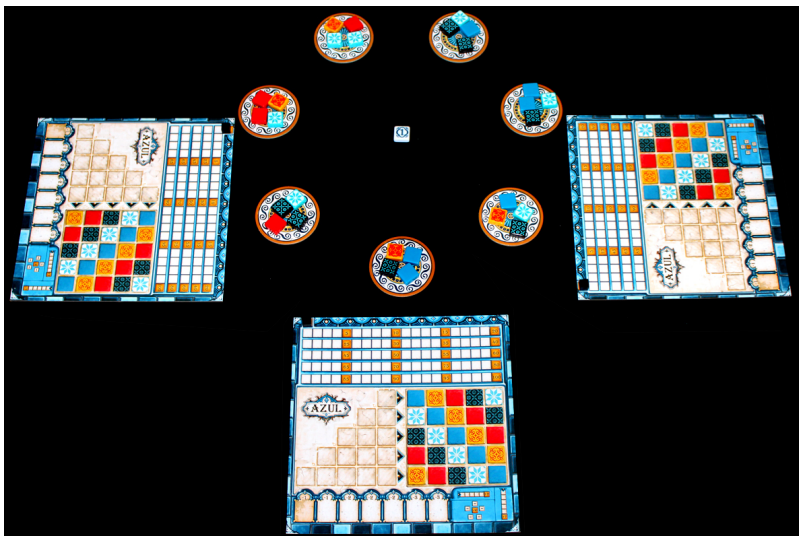
In this assignment you will implement a **simplified 2-player** text-based version of the Board Game **Azul**.

For an explanation of the rules and gameplay:

- Review and rules explanation: by SU&SD (Youtube)
- Rules explanation: by Dice Tower
- Official Rules: Online Website

In this assignment you will:

(a) `Azul` box and pieces



(b) Example game state

- Practice the programming skills covered throughout this course, such as:
  - ADTs
  - Data Structures (Arrays, Vectors, Linked Lists, Trees, etc.)
  - Dynamic Memory Management
  - File Processing
- Practice the use of testing
- Implement a medium size C++ program
- Work as a team

This assignment is divided into four Milestones:

- **Milestone 1, Group Progress Update (Group work)**: Your group will submit an update on their progress toward Milestone 2. This will require your group to have completed a list of activities. Milestone 1 will not be marked directly, however, Milestone 1 will influence the Milestone 2 grade which is worth 30% of the course mark.
- **Milestone 2, `Azul` Implementation (Group work)**: Your group will implement a simplified version of `Azul`, and write tests that show your implementation is correct. The group component is due 11.59pm Sunday 4 October 2020 (Week 10). The group work is worth 30% of the course mark.
- **Milestone 3, Enhancements (Individual work)**. You will *individually* extend upon your group's implementation with additional functionality (called enhancements). The individual component is due 11.59pm Sunday 18 October 2020 (Week 12). The individual work is worth 15% of the course mark.
- **Milestone 4, Written report (no more than 4 pages) & Demonstration**. You will write a report that analyses the design and implementation of your software. The report is due 11.59pm Sunday 18 October 2020 (Week 12). You will demonstrate your group and individual work. This is where your final work will be graded. Demonstrations will be held during Week 13.

1.1 Group Work

The group work must be completed in groups of 3.

1. You *may* form groups with any student in the course
2. We strongly recommend that you form groups from within your labs, because:
   (a) Your tutor will help you form groups, *but only within your lab.*
   (b) You will have plenty of opportunity to discuss your group's progress and get help from your tutor during the rest of the course. It will be extremely helpful for your whole group to be present, but this can't happen if you have group members outside the lab.

Groups for Assignment 2 must be **registered** with your tutor by the **your week 7 lab**. Your tutor "register" your group on Canvas.[1] If you are unable to find a group, discuss this with your tutor **as soon as possible**.

If at any point you have problems working with your group, **inform your tutor immediately**, so that issues may be resolved. This is especially important with the online delivery of the course. We will do our best to help

[1] If your group spans multiple labs, have one of your tutors register the group.

manage group issues, so that everybody receives a fair grade for their contributions. To help with managing your group work we will be requiring your group to use particular tools. These are detailed in Section 6.

> **!** There are **important requirements** about keeping your tutor informed if you have been unwell or otherwise unable to contribute to your group. Remember your actions affect **everybody in your group**.

1.2 Learning Outcomes

This assessment relates to all of the learning outcomes of the course which are:

- Analyse and Solve computing problems; Design and Develop suitable algorithmic solutions using software concepts and skills both (a) introduced in this course, and (b) taught in pre-requisite courses; Implement and Code the algorithmic solutions in the C++ programming language.
- Discuss and Analyse software design and development strategies; Make and Justify choices in software design and development; Explore underpinning concepts as related to both theoretical and practical applications of software design and development using advanced programming techniques.
- Discuss, Analyse, and Use appropriate strategies to develop error-free software including static code analysis, modern debugging skills and practices, and C++ debugging tools.
- Implement small to medium software programs of varying complexity; Demonstrate and Adhere to good programming style, and modern standards and practices; Appropriately Use typical features of the C++ language include basic language constructs, abstract data types, encapsulation and polymorphism, dynamic memory management, dynamic data structures, file management, and managing large projects containing multiple source files; Adhere to the C++14 ISO language features.
- Demonstrate and Adhere to the standards and practice of Professionalism and Ethics, such as described in the ACS Core Body of Knowledge (CBOK) for ICT Professionals.

2 Milestone 1: Group Progress Update

For Milestone 1, your group will need to submit an update on their progress towards completing Milestone 2. This update must include:

1. The group progress document:
   (a) The check-list of what the group has completed that is listed against the suggested schedule and
   (b) Description of the individual contributions of each group member
2. A zip file containing your group's code and completed work by the time Milestone 1 is due

Your tutor will evaluate how your group is progressing according to the suggested schedule (see Section 6.2), and record if your group is *ahead*, *on-track* or *behind* the schedule.

The final grade for the group component will be *informed* by this update. The marking rubric details how Milestone 1 impacts your final grade.

3 Milestone 2: `Azul` Implementation (Group Component)

In your groups you will implement a *simplified* version of `Azul`. This section lists the components that your group must implement. Generally, it is up to your group to decide the best way to implement these components. However, there are some **important requirements** that your group must satisfy.

> **!** Aspects of this specification are flexible and open to your interpretation. It is up to your group to determine the best course of action. You will need to **analyse** and **justify** your choices in the report.

3.1 Requirements

Your group will implement a game of `Azul` that:

- Is a **2-player** game.
- Both players are "human users" that play the game by interacting with the terminal, that is, through standard input/output.
- Using the default `Azul` Mosaic, with a fixed tile pattern. This is the mosaic pictured in Section 1. (Note this pattern is the same for *all* players).

- Use 5 factories (plus the central factory) as specified in the `Azul` rules for a 2-player game.
- Automatically moves tiles to the mosaic and scores points at the end of a round.

You will implement a simplified version of `Azul`. These changes are:

1. No randomness, including no shuffling of tiles. Section 3.4 describes how to play without randomness.
2. Remove the box-lid.
3. A game lasts exactly 5 rounds. You don't need to check for the "end-of-game condition".
4. No end-of-game scoring of bonus points

Your implementation should provide the following functionality:

- A main menu, that allows users to perform actions such as setting up a new game, loading an existing game, showing "credits" (details of the people who wrote the software), and quitting the program.
- Save a game to a file.
- Load a previously saved game from a file, and resume gameplay from the saved state.
- A way to represent and display the `Azul` mosaics and factories to the user.
- A User prompt for entering all commands from standard input.
- A special "testing mode", see Section 3.3.
- The program should terminate without crashing if the EOF character is given on standard input.
- Completely error free. Your program must not crash, segfault or otherwise contain logic errors.

Your group will also need to consider the **data structures** that are used to represent aspects of `Azul`, and functionality. It is up to your group to make this decision provided that you meet the following **requirements**:

- You must use *at least one* linked list to store or represent some aspect of the game.
- You must use *at least one* C++ STL vector to store or represent some aspect of the game.
- You must use *at least one* 1D or 2D array to store or represent some aspect of the game.
- You may only use the C++14 STL. You may not incorporate any additional libraries.

In your report your group will be marked on your **analysis** of the above choices.

Finally, remember to submit the group contribution spreadsheet described in 6.3.

3.2 Example Program

As an example, when you have implemented Milestone 2 your program might look as follows. Note that this example combines output from the program (ie to `std::cout`) and input from the user (ie to `std::cin`), where the user prompt is given by `>` and any text after this has been typed on standard input.

```
Welcome to Azul!
------------------
Menu
----
1. New Game
2. Load Game
3. Credits (Show student information)
4. Quit
> 3
---------------------------------
Name: <full name>
Student ID: <student number>
Email: <email address>
<Student 2, 3.>
---------------------------------

Menu
----
1. New Game
2. Load Game
3. Show student information
4. Quit
> 1
```

```
Starting a New Game

Enter a name for player 1
> <user enters name>

Enter a name for player 2
> <user enters name>

Let's Play!

=== Start Round ===
TURN FOR PLAYER: A
Factories:
0: F
1: R Y Y U
2: R B B B
3: B L L L
4: R R U U
5: R Y B L

Mosaic for A:
1:         . || . . . . .
2:       . . || . . . . .
3:     . . . || . . . . .
4:   . . . . || . . . . .
5: . . . . . || . . . . .
broken:

> turn 2 B 3
Turn successful.

TURN FOR PLAYER: B
Factories:
0: F R
1: R Y Y U
2:
3: B L L L
4: R R U U
5: R Y B L

Mosaic for B:
1:         . || . . . . .
2:       . . || . . . . .
3:     . . . || . . . . .
4:   . . . . || . . . . .
5: . . . . . || . . . . .
broken:
> turn 3 L 3
Turn successful.

> save savedGame

Game successfully saved to 'saveGame'

< turns continue until all tiles are taken >

=== END OF ROUND ===

=== Start Round ===

< Gameplay continues until the end of the game (5 rounds) >
```

```
=== GAME OVER ===

Final Scores:
Player A: 57
Player B: 98
Player B wins!
```

3.3 Testing Mode

Testing that your program is correct is very important. You will use saved-game files to test help your code (The format of saved-gamed files is described in Section 7). Briefly, a saved-game stores all of the turns that are played during a game. Thus when loading, the turns can be replayed to "re-construct" the game.

To use a saved-game file for testing your program you will need to implement a special "*testing mode*". A program is run in "testing mode" by using 2 command-line arguments:

```
$ ./azul -t <save-game-file>
```

The first argument `-t` activates the testing mode, and the second is a saved-game file. When run in "testing mode", your program should do the following:

1. Load a game from the saved game file
2. Print out the state of the game after loading including:
   - Player names
   - Factories
   - Player mosaics
   - Player scores
3. Quit (without crashing)

You can then use this output (to `std::cout`) to compare against an **expected output** of your program in a similar way that we tested the output of programs in assignment 1. For example, if the saved game file is:

```
<initial tile bag>
A
B
turn 2 B 3
turn 3 L 3
```

Then the output of your program when run in testing mode should be similar to:

```
Factories:
0: F R B
1: R Y Y U
2:
3:
4: R R U U
5: R Y B L

Score for Player A: 0
Mosaic for A:
1:         . || . . . . .
2:       . . || . . . . .
3:     B B B || . . . . .
4:   . . . . || . . . . .
5: . . . . . || . . . . .
broken:
```

```
Score for Player B: 0
Mosaic for B:
1:         . || . . . . .
2:       . . || . . . . .
3:     L L L || . . . . .
4:   . . . . || . . . . .
5: . . . . . || . . . . .
broken:
```

3.4 Tile Bag

> **!** This is very important to ensure you can run/test your code, and run any tests we give you

The simplified `Azul` you are implementing **does not** use any randomness. Thus, you will take the following approach for managing the tile bag:

1. You will use a *deterministic* (or fixed) ordering of tiles.
2. The tile bag must be treated as *queue*. That is:
   - Tiles are always drawn from the tile bag by taking tiles from the *front* of the queue
   - Tiles are always added back into the tile bag by adding tiles to the *back* of the queue
3. The box lid has been removed. Instead, when tiles should be added to the box lid (according to the original rules), the tiles are instead added to the *back* of the tile bag queue. Recall, that:
   (a) This only happens at the end of a round
   (b) The mosaic is "processed" from the 1st row down, finishing with the broken tiles.
   (c) You will also need to conduct the end-of-round scoring with the 1st player, and then the 2nd player.
4. The tile-bag order is determined when a new game is created. Use the same order every game.
5. In a saved-game file, the order of the tile-bag when the game was first created is provided.

3.5 Suggestions

This section provides suggestions that you might wish to consider for implementing the simplified `Azul`.

3.5.1 User Prompt

The user prompt (greater-than symbol (>), followed by a space) is displayed whenever input is required from the user. These suggestions assume that all user inputs are provided as a single line.

```
> 
```

If at any point the user enters input which is invalid then the program should print an error message and re-show the prompt so the user can try again.

```
> qwerty
Invalid Input
> 
```

If the user enters the EOF (end-of-file) character[2], then the program should Quit.

```
> ^D
Goodbye
```

3.5.2 Tiles

Tiles could be represented by a single-character code, based on their colour as in the table below. Special codes represent the 1st-player maker, and where a tile is not present.

[2]Reminder: this is **not** the two characters `^` and `D`, this is the representation of EOF when typing `control-D`

Colour	Colour Code
Red	`R`
Yellow	`Y`
Dark Blue	`B`
Light Blue	`L`
Black	`U`
`first-player`	`F`
`no-tile`	`.`

3.5.3 Game Board

The board has two elements:

1. The shared central area - "factories"
2. The individual player board - "mosaic".

The factories can be represented by listing all of the tiles on the factory. Factories are labelled with numbers so the user can refer to them. Factory `0` is the "centre" factory.

```
Factories:
0: F B U
1: R Y Y U
2:
3: B L L L
4: R R U U
5: R Y B L
```

This shows the state of an individual players mosaic, giving:

- Storage rows of unlaid tiles
- Completed grid of tiles.

This is an example of a mosaic. The "broken" tiles (including the 1st player marker) are listed at the bottom. Again, each storage row is numbered so the players can refer to it.

```
Mosaic for <player-name>:
1:         . || . . R . .
2:       Y Y || . . . R .
3:     B B B || . . . . .
4:   . . . . || . . L . .
5: . . U U U || . . . . .
broken: F Y
```

3.5.4 Launching the program

In "normal" (non-testing) mode your `Azul` program will be run from the terminal.

```
$ ./azul
```

3.5.5 Main Menu

The main menu shows the options of your `Azul` program. By default there should be 4 options (new game, load game from a file, credits and quit). The menu is followed by the user prompt.

3.5.6 Starting a New Game

The program should start a new game. As part of this you might want to get names for each of the players. Remember that the tile bag (Section 3.4) must use a fixed order!

3.5.7 Load a Game from a file

The program asks for a filename, where the filename is a *relative path* to the saved game file.

```
<main menu>
> 2

Enter the filename from which load a game
> <filename>

Azul game successfully loaded
<game play continues from here>
```

It is highly recommended to conduct validation checks such as:

1. Check that the file exists.
2. Check that the file contains a valid game.

Once the game has been loaded, gameplay resumes with the current player.

Hint: You should use the **same** logic as if the user(s) had typed the turns on the command line!

3.5.8 Credits (Show student information)

The program should print the name, student number, and email address of each student in the group.

3.5.9 Quit

The program should safely terminate *without crashing*.

3.5.10 Typical Gameplay

In `Azul`, 2 players take turns drawing tiles from factories and placing them in storage on their individual mosaic, starting with the first player. Once all of the factories are empty (including the centre factory), the round ends and scoring happens **automatically**. After 5 rounds, the game ends.

3.5.11 Starting a Round of `Azul`

At the beginning of a round, the factories need to be filled by drawing tiles from the Tile Bag. To ensure consistency, factories should be filled starting with factory `1`. Don't forget to add the 1st-player marker to the "centre" factory (number `0`).

3.5.12 A Player's Turn

A player might take their turn using the command such as

```
turn <factory> <tile-code> <storage row>
```

The command contains three elements:

1. A number of the factory to draw from
2. The code of the tile to take from the factory
3. The row in the mosaic storage to place the tiles

After the player enters the command, you should:

- Validate that the turn is *legal*, checking the player's action against the rules of `Azul`
- Update the game-state based on the player's turn, then continue with the next player's turn.

3.5.13 End-of-round

At the end of the round, starting with the 1st player your program should:

1. Move tiles from the player's storage to their completed mosaic grid, as per the rules of `Azul`.
2. Update the player's score.
3. Subtract points for broken tiles (Don't forget to move these tiles to back to the Tile Bag).

This is repeated for the 2nd player. You might also want to show how many points each player scored on that round and their total scores. The game then either proceeds to the next round or terminate if 5 rounds have been played.

3.5.14 End-of-game

The game ends **after 5 rounds**. You may then show the winner, the final scores and the final mosaics.

3.5.15 Saving the Game to a file

At any point, the current player may save the game to a file using a command such as:

```
save <filename>
```

Your program should save game to the given file (overwriting the file if it already exists), using the format described in Section 7.

4 Milestone 3: Enhancements (Individual Component)

In Milestone 3, as an **individual** you will make signification expansions(s) to the functionality of your group's `Azul` program. This milestone is your opportunity to showcase to us your skills, capabilities and knowledge! You will select your enhancements from the provided options. Additionally, if you group's Milestone 2 solution has *significant errors* or is *significantly incomplete*, please read Section 4.3.

Milestone 3 is a very open-ended. You are given some directives, however, there is a lot of room for you to make considered choices. However, this showcase of your skills is not just about "making the code work". A major focus is on *how* you choose to implement an enhancement, and the *justifications of the reasons why* you chose a given data structure, class hierarchy, language feature, or algorithm to name a few examples.

Enhancements are classified as **minor** or **major**. Enhancements must be *substantially functional* to get marks.

> **!** The list of major/minor enhancements will be released at the end of Week 10.

4.0.1 Configurable Enhancements

Where reasonably possible, your enhancements ts should be configurable. That is, it should be possible to enable/disable each enhancement at run-time (not through compilation). In particular, this means your enhancements can be "turned-off" so that your program runs the same as Milestone 2 (verbatim).

4.0.2 Changes to Saved Game file & Testing Mode

Some enhancements may require you to modify the format of the saved-game file. However, it is recommended that your program with enhancements *still supports* loading games from the default saved-game format used in Milestone 2.

To distinguish your new saved-game format from the "default" milestone 2 format, we recommend adding a special code at the start of *your new* saved-game format, such as:

```
#myformat
<initial tile bag>
...
```

4.1 Minor enhancements

4.1.1 Randomness & Box Lid

The simplified game does not shuffle the tiles in the tile bag, and does not use the box lid. This enhancement requires you to introduce the ability to shuffle the tiles (that is, randomise) the tiles according to the original rules of `Azul`. That is you should support these additional features:

1. When a new game is started, the tile bag should be *shuffled*. You will need to devise *your own algorithm* to "shuffle" the bag of tiles.
2. Instead of moving tiles back into the tile bag (as in our simplified version), the tiles are temporarily moved to the box lid
3. When the tile bag is empty, the tiles in the box lid are placed into the tile bag and the tile bag is *shuffled*
4. Update the format of the saved game file to cater for shuffling the tile bag

You will need to take careful note of how this enhancements impacts:

- the data structure for the tile bag and box lid
- loading games from a saved-game file
- the "testing mode" of your program.

You will need to store additional information in the saved-game file so the sequence of turns can be correctly replayed. This is left to your own invention.

Finally, you might want to be able to "turn the randomness off". We recommend that you use a command-line argument to take a fixed seed for your pseudo-random number generator.

```
$ ./azul -s <seed>
```

4.1.2 Data Structure Improvements

Additional data structures can be incorporated into your group's `Azul` program. This enhancement requires you to *significantly modify* and reconsider the data structures used in your `Azul` program. You must:

- Add a *Binary Search Tree*
- Ensure your modified program still contains:
  - You must use *at least one* binary search tree to store or represent some aspect of the game.
  - You must use *at least one* linked list to store or represent some aspect of the game.
  - You must use *at least one* C++ STL vector to store or represent some aspect of the game.
  - You must use *at least one* 1D or 2D array to store or represent some aspect of the game.

For this enhancement you must meet all of the above constraints. If any data structure is missing, this enhancement will be considered incomplete. As a suggestion, you may wish to use a binary search tree to for the tiles on the factories, as a user may find it hard to see the number of each tile on each factory. Could a binary search three improve displaying tiles and/or finding tiles in the factories?

You should also re-consider if your group's `Azul` program, and questions such as

- Are the most suitable data structures used to represent the elements of the `Azul` game?
- Can the use of abstraction and the design of the ADTs be improved?
- Can class inheritance and abstract classes be used to reduce code duplication?

If the use of data structures could be improved, then you must modify the program to use the most appropriate data structures. For example, if you group's report notes that the any of the above elements could be improved, then you should change these elements in this enhancement.

In your *individual* report you should justify why the data structures and code re-use has *significantly improved*.

4.2 Major enhancements

4.2.1 Advanced Azul (grey-board and 6-tile modes)

`Azul` can be played in more advanced modes that can bring additional strategy and challenge for players. The enhancement requires you implement two "advanced modes" for `Azul`, the "grey-board" and "6-tile" modes.

The "grey-board" is an advanced mode of `Azul` where the location that tiles are placed into a mosaic is not fixed. You should consult the `Azul` rules for the "grey-board" mode. Importantly, this mode requires players to manually place tiles onto their mosaic after all the factories are empty for end-of-round scoring, rather than the process being automatic. You will need to devise a command for players to place tiles onto their mosaic. You will also need to make sure you record these commands into the saved-game file, so that the game can be correctly re-loaded.

The "6-tile" mode introduces a 6th tile, which for the purposes of this enhancement will be denoted by the colour "orange". To permit play with the 6th tile, the following changes to the `Azul` board are made:

- The mosaic is expanded to a 6x6 board
- The 6th row has space to store 6 tiles
- An additional broken tile slot is added, worth `-4` points

For this enhancement, you may wish to consider:

- You may need to *redesign and change* the data structures that are used to represent `Azul`. However, you must *still* meet the mandatory minimal use of each data structure as given in Milestone 2.
- You will need to modify the saved-game format to support the advanced options.

- You may need additional commands to support the advanced modes.

In your *individual* report you should justify any *significant* changes to the choice of data structure(s), why the representation of information in the saved-game is *suitable*, describe any *significant* changes to your text-based UI, and justify any other necessary changes to your group's `Azul` program.

We recommend that the advanced mode is enabled using a command-line argument, such as:

```
$ ./azul --adv
```

(Note two dashed are used in due to the Unix command-line argument styles for multi-letter arguments.)

4.2.2 Write an AI

It would be nice to play `Azul` in a single-player mode against the computer. This enhancement requires you to develop an AI (Artificial Intelligence) so that your program can be used in single-player mode, and a person can play against the computer AI. When it's the AI's turn, it should make its move automatically without the user having to input a command.

An AI implies *intelligence*. An AI doesn't take random actions. It uses logic and *heuristics* to determine a "good" move, and hopefully the "best" action to take. Heuristics are ways to "guess" or "estimate" if an action is good or bad. Thus, your AI needs to have "intelligence" to figure out a good action to take on its turn. You may not be able to decide the "optimal" action, however, the choice must be better than a random action.

For this enhancement, you may wish to consider:

- You may need to *redesign and change* the data structures that are used to represent `Azul`. However, you must *still* meet the mandatory minimal use of each data structure as given in Milestone 2.
- The AI should not take a long time to calculate its move.
- You will need to modify the saved-game format to record if an AI is being used.

In your *individual* report you should justify why your AI has a "good" heuristic and is intelligent. You should also describe any *significant* changes to the choice of data structure(s) describe and justify any other necessary changes to your group's `Azul` program.

We recommend that the AI is enabled using a command-line argument, such as:

```
$ ./azul --ai
```

(Note two dashed are used in due to the Unix command-line argument styles for multi-letter arguments.)

4.3 Milestone 2 Code with Significant Errors

> **!** This section is a **general** statement. This is provided as a **starting point** from which to approach your tutor. This option may **only** be used if you have discussed the matter with your tutor.

It is possible that your group's `Azul` program may contain errors. If the errors in the your group's `Azul` program are small, while you should fix these in your individual work, they are not considered a *significant* change.

However, if your group's `Azul` program has significant errors or is missing significant functionality, then you **may be able to negotiate** with your tutor to fix these error as a **minor** enhancement. As each group's program is different, this will be determined on a case-by-case basis.

Please note that this does not excuse you from failing to make sufficient contributions to your group. While you may be able to "make-up" some functionality in this milestone, you should not use this as an excuse to make an insufficient contribution.

4.4 HD+ submissions

To receive a grade of `HD+` your submission must be **outstanding**. This means your work must stand apart from other submissions. If you would like to receive top marks, you will need to go above-and-beyond the minimum requirements of each enhancement.

5 Milestone 4: Written report & Demonstration

Your group must write a report, that *analyses* what your group has done in this assignment. Additionally, each **individual** must write a short report that *analyses* their individual enhancement(s). The report is due at the same time as the individual submission (11.59pm Sunday 18 October 2020 (Week 12)).

- The report should be A4, 11pt font, 2cm margins and single-space.
- The section of the report describing the group's work must be no more than **4 pages**.
- Each individual must add **1 additional page** about their enhancements.
- Thus the final report **must not exceed 7 pages** (for a group of 3).
- Only the first 4 pages (group), and 1 page (individual) will be marked. Modifying fonts and spacing will count as over length.
- Figures, Tables and References count towards these page limits.

In this assignment, you are marked on the analysis and justification of the choices you have made. Your report will be used (in conjunction with the demonstration) to determine the quality of your decisions and analysis.

Good analysis provides factual statements with *evidence* and *explanations*.
Statements such as:

> *"We did <xyz> because we felt that it was good"* or *"Feature <xyz> is more efficient"*

do not have any analysis. These are unjustified opinions. Instead, you should aim for:

> *"We did <xyz> because it is more efficient. It is more efficient because . . . "*

! We are asking for a combined report as it keeps the context of each individual's enhancements with the whole group's original implementation.

5.1 Group Component of the Report

In the **group** section of your report, you should discuss aspects such as:

- Your group's use of *at least one* linked list, and the reasons for where the linked list is used.
- Your group's use of *at least one* C++ STL vector, and the reasons for its use.
- Your group's use of *at least one* 1D or 2D array, and the reasons for its use.
- Your group's choices of ADTs, and how these leads to a "well designed" program.
- The efficiency of your implementation, such as the efficiency of your use of data structures.
- The reason for each of your tests.
- Your group co-ordination and management.

5.2 Individual Component of the Report

In the **individual** section of your report, you should discuss aspects such as:

- The design of your enhancements, including any changes (and additions) to the data structures and ADTs you had to make when enhancing your group's implementation.
- The efficiency of your enhancements, such as the efficiency of your use of data structures.
- Limitations and issues that you encountered with your group's implementation.

5.3 Demonstration

During Week 13, your group will demonstrate and discuss your work. In your demonstration you should:

- Demonstrate your `Azul` gameplay implementation by running the program.
- Demonstrate how your test cases prove your implementation is correct
- Discuss the design and efficiency of your software
- During the presentation, we may provide you with example saved game files for you to test live.

Additionally, each student will demonstrate their **individual** enhancements by running their program.

- Each individual student will be required to make a short demonstration.
- They should not be interrupted or assisted by other students during this time.

The purpose of the demonstrations is to mark your work. In your demonstration you should aim to convince your assessor of the quality of your work, and the grade that you should receive in each rubric category.

Each presentation will be 30 minutes. We recommend that you split up the demonstration time as follows:

- 10 minutes for the group demonstration
- 5 minutes for each individual student in the group
- 5 minutes for final questions

It is up to you and your group to decide how to best conduct this presentation. You should be prepared to:

- Share a screen with the group program running
- Share a screen with your individual enhancements
- Have test cases for demonstrating the program prepared in advance

5.3.1 Booking Demonstrations

Demonstration will be booked through an online system. Details will be provided closer to the demo period.

5.3.2 Running Demonstrations

Demonstrations will be held over MS Teams, **using the MS Team for your group**. Before the time your demonstration is scheduled, log into MS Teams, and use the "Meet Now" function for the "General Chat". Your assessors will join the meeting at the scheduled time. We recommend that your group creates the MS Team meeting at least **10 miuntes** before the scheduled time.

Ensure you are ready to go when the demonstration begins. You assessor should not be waiting for your group to set-up the demonstration, or explain how to use MS teams to share your screen.

6 Managing Group Work

! Having effective group work will be **critical** to the success of your group and reducing your stress levels. This 5 Minute Video from the Minute Physics YouTube channel contains a number of really good suggestions for how to work effectively as a team from home.

This group assignment will be conducted entirely online, without you ever meeting your group members face-to-face. This isn't a problem, with the available online tools. The challenge for you will be using these tools *effectively*. You will need to **make extra efforts** and be **very dedicated and diligent** in working with your team members. This will include setting up dedicated times for meetings and group programming sessions.

6.1 Group Work Tools

To help manage your group work, and demonstrate that you are consistently contributing to your group, we are going to require you to use a set of tools.

6.1.1 MS Teams

Each group will be required to create a team on the RMIT MS Teams platform. Your group must **add your tutor(s)** to your MS team[3].

Your MS team will be the *only official* communication platform for the assignment. This means you must:

- Only use the MS Team channels for group chats
- Hold all team meeting through MS Teams and record all team meetings
- Store any group files (not in Git) in the MS Team

If there are disputes, we will use the record on MS Teams as the source of evidence. You may not use other platforms (including Discord) as we cannot verify the identify of the users.

6.1.2 Git Repository

Your group must have a **private Git repository** that hosts your group's code. This may be on BitBucket or Github. Your group must **add your tutor(s)** to this repository[4]. This git repository will be used as the *evidence of your individual contribution* to your group.

For the individual component you should copy/fork the repository before commencing your individual work.

[3] If your group is from multiple labs with different tutors, add *all* of your tutors to the MS Team.
[4] If your group is from multiple labs with different tutors, add *all* of your tutors to the MS Team.

6.2 Suggested Weekly Schedule

On Canvas, we have provided a spreadsheet that contains a schedule of the suggested tasks your group should finish each week. Additionally, for Milestone 1 we will compare your group's progress to this schedule. The tasks for Milestone 1 are flexible if your group runs into issues, such as a student being sick. However, you **must inform your tutor** of any issues so that you can negotiate with your tutor to revise the Milestone 1 tasks.

6.3 Group Contributions

For Milestone 1 and 2 you will need to provide a record of your contribution to your group. This record is also contained on the schedule spreadsheet which has a location where each student can record their contributions. You should also note any issues that your have encountered on this spreadsheet.

! Include the filled-in weekly group schedule spreadsheet with your Milestone 1 and Milestone 2 submissions.

6.4 Notifying of Issues

If there are **any issues** that affect your work, such as being sick, you **must keep your group informed** in a timely manner. Your final grade is determined by you (and your group's) work over the entire period of the assignment. We will treat everybody fairly, and account for times when issues arise when marking your group work and contributions. However, you must be upfront and communicate with us.

! If you fail to inform us of issues in a **timely fashion** and we deem that your actions significantly harm your group's performance, we may award you a reduced grade. It is academic misconduct if you are *deliberately dishonest*, *lie to*, or *mislead* your group or teaching staff in a way that harms your group.

7 Saved Game Format

The format of a saved-game allows us to "re-construct" the state of a game by "replaying" all of the turns from a partially (or fully) completed game of `Azul`. A saved-game file contains the following in order:

1. The initial order of the tile bag when the game was created. (Note that while your `Azul` implementation will always use the same order when a new game is created, you must support using a *different* order when loading from a saved-game file.)
2. The player's names
3. The order of turns taken by the players

For example, the following stores one full round of `Azul` using the initial game state from Section 3.2.

```
<initial tile bag>
A
B
turn 2 B 3
turn 3 L 3
turn 2 Y 2
turn 5 Y 1
turn 4 U 5
turn 0 B 2
turn 0 L 1
turn 0 R 5
turn 0 U 5
```

UPDATE: The format initial tile bag is a sequence of character tile codes, on a single line without white space. For example the tile bag will look like the following (of course, for space reasons not all tiles are listed)

```
BURYYUBLLLRYBL
```

! You will need to carefully think about the information that needs to be stored in your program so that you can successfully create a saved game file.

8 Getting Started

This assignment requires you and your group to make a number of design choices including:

- Where to use linked lists, vectors and arrays
- How to create ADTs
- The format for getting input from the user

It is up to your group to determine the "best" way to implement the program. There isn't necessarily a single "right" way. The challenge in this assignment is about software design, not the actual gameplay. It is **very important** that you think about this design and your ADTs early in your work.

Since you may not have designed many pieces of software before, to help you get started, we recommend that you think about ADTs for the following aspects:

- `Game` - This store all of the state of a game of Azul. You will need to think about where you load and save a game state. This could be inside the Game ADT, or another part of the code that uses methods of the Game ADT.
- `Factories` - An ADT within `Game` that stores all information about the factories. You should think about if you need a further ADT for an individual factory.
- `Mosaic` - An ADT within `Game` that stores *one* mosaic for a player, including the storage rows, mosaic grid and broken tiles.

You can get help about your ideas and progress through the labs. Bring your ideas to your tutor and discuss these ideas with them.

9 Submission

To submit, follow the instructions on Canvas for Assignment 2.

Assessment declaration: When you submit work electronically, you agree to the assessment declaration.

9.1 Silence Period

A silence policy will take effect from **2 business days before each due date**. This means no questions about this assignment will be answered, whether they are asked on the discussion board, by email, or in person. Make sure you ask your questions with plenty of time for them to be answered.

9.2 Late Submissions & Extensions

Late submission accrue a penalty of 10% per day up to 3 days, after which you will lose ALL the assignment marks. The late penalty for Milestone 1, 2 and 4 is applied to the group component of the assignment mark. The late penalty for Milestone 3 is applied to the individual component of the assignment mark.

Extensions will not be given for this assignment.

9.3 Special Consideration

> ! In addition to special consideration processes, you are ***required*** to keep your group and tutor informed of when you are unable to contribute, ***in a timely manner***. If you apply for special consideration, but do not notify of issues before doing so, then you have *failed* to keep everybody informed.

Where special consideration is granted, generally it is awarded on an individual basis, not for the entire group.

Extensions of time will not be granted, instead an *equivalent assessment* will be conducted which may take the form of a written or practical test. This equivalent assessment covers the non-group components of the rubric. The group-work component will be assessed on your group participation for the duration of Assignment 2 for which you were unaffected and able to contribute. This will *take into consideration* how well you *kept your group informed* of your ability to work and contribute to the group.

9.4 Group Work Penalties

In severe cases of poor group work:

- We may apply an individual mark to any or all categories of the rubric.
- We may file a charge of academic misconduct if we determine that a student has been *deceitful* or *untruthful* in a manner than has a severe adverse academic impact on the other students in the group.

10 Marking guidelines

The detailed breakdown of this marking guidelines is provided on the rubric linked on Canvas. The marks are divided into the following categories:

- Group Component (30/45):
  - `Azul` Implementation: 15/45
  - Test Cases: 5/45
  - Analysis & Design: 5/45
  - Group Work: 5/45
- Individual Component (15/45):
  - Individual Enhancements Implementation: 10/45
  - Individual Enhancements Analysis: 5/45

11 Academic integrity and plagiarism (standard warning)

! CLO 6 for this course is: *Demonstrate and Adhere to the standards and practice of Professionalism and Ethics, such as described in the ACS Core Body of Knowledge (CBOK) for ICT Professionals.*

Academic integrity is about honest presentation of your academic work. It means acknowledging the work of others while developing your own insights, knowledge and ideas. You should take extreme care that you have:

- Acknowledged words, data, diagrams, models, frameworks and/or ideas of others you have quoted (i.e. directly copied), summarised, paraphrased, discussed or mentioned in your assessment through the appropriate referencing methods
- Provided a reference list of the publication details so your reader can locate the source if necessary. This includes material taken from Internet sites. If you do not acknowledge the sources of your material, you may be accused of plagiarism because you have passed off the work and ideas of another person without appropriate referencing, as if they were your own.

RMIT University treats plagiarism as a very serious offence constituting misconduct. Plagiarism covers a variety of inappropriate behaviours, including:

- Failure to properly document a source
- Copyright material from the internet or databases
- Collusion between students

For further information on our policies and procedures, please refer to the RMIT Academic Integrity Website.

The penalty for plagiarised assignments include zero marks for that assignment, or failure for this course. Please keep in mind that RMIT University uses plagiarism detection software.