UM EECS 281 Fall 2004
Programming Assignment 1
Mission to Mars

The Game
Your job is to program a new Mars rover named the **Vanguard**. The Vanguard is being deployed on a remote location on the planet Mars. The purpose of the rescue mission is to find the lost British rover **Beagle 2** and help Beagle 2 transmit its data back to Earth. Shortly after landing, unintelligible data was received from Beagle 2 and suddenly all contact was lost. It is believed that Beagle 2’s video recorders are still active and could provide key information regarding what went wrong. Satellite reconnaissance has located Beagle 2, and revealed that several key components of Beagle 2 are scattered about the surface of the Red Planet. Among these parts are the Apex Transmogrifier Unit (**ATU**) and the Wavelet Transponder Dish (**WTD**). Vanguard must gather these pieces and repair Beagle 2, so that Beagle 2 may beam its information back to Earth.

Vanguard must also discover a way to provide enough power for Beagle 2 to successfully send its data back to Earth. It is believed that a rare mineral, Praesodymium (aka **Crystal**) is located on Mars in crystal form. This crystal may be processed by the ATU and provide enough power for Beagle 2 to operate and send home its collected data. Therefore, part of the mission is to assist Vanguard in finding crystal to help power Beagle 2. In Summary, your mission is to program the Vanguard to find the ATU, WTD, and Crystal, and proceed to Beagle 2 for repairs after gathering these objects.

A Harsh and Barren World
The game world is divided into multiple **maps** where pieces of Beagle 2 might be located. A map may be thought about as a level in the game world. The number of maps in any particular game world may vary from 1 to 10.

The Vanguard can travel between these maps by use of new NASA technology, called **teleporters**, that instantly transport the rover to the specific, associated map. Teleporters, signified by grey tiles in a map, transport the Vanguard from the current location to the associated tile on another map specified by the grey teleporter tile. Teleporters are bidirectional; that is, they can transport from map X to map Y and from map Y to map X.

All maps are accessible by at least one teleporter, however, the \{**ATU**, **WTD**, **Crystal**, and **Beagle 2**\} may be distributed between multiple maps. **Figure 1** shows an example game world with three maps. In the first map there are two teleporters, one teleports between map one and map two, and the other teleport between map one and map three.
Your Tasks

You are to develop two path finding algorithms:

- A stack-based path finding algorithm;
- A queue-based path finding algorithm.

Your stack must use a linked list as its underlying data structure. Your queue must use an array as its underlying data structure.

You MUST develop your own stack and queue data structures for this programming assignment. That is, aside from the string class, you may not use the STL for any portion of this programming assignment. You can use the codes provided in the textbook, but you MUST include the textbook in your references if you use.

Grade Composition

Form

5%: Properly developed and commented code. For details, see Coding Style section near the end of this document.

Documentation

7.5%: Timing Analysis
7.5%: Test Report

Performance

40%: Working Stack algorithm
40%: Working Queue algorithm
What to Turn in

For the project, each student is required to turn in documentation in hardcopy, and source code to the autograder. Details about both types of submission follow.

Documentation

You must turn in a hard-copy of your Timing Analysis and Test Report results to the eecs281 drop box in 2420 EECS by the due date/time. Make sure your uniqname is on the hard-copy.

Timing Analysis

For timing analysis, we require that you run both algorithms on the given example test cases. Results will be turned in the form of an Excel graph. Collect the timing information for each of the given example test cases, for each of your algorithms. For each algorithm sort the timing numbers by the testcase map size. Then plot both sets of numbers on a single Excel graph. (If your algorithm detects some fatal errors on a given test case and exits with proper EXIT_CODE explained in File Output section, you don’t need to collect timing information for the test case.)

The timing information is available via the console command time. To display the current timing information, press tab to open up the console, then type in time and press enter. This will display the time that Vanguard spends executing the path-finding algorithm, including returning the solution tiles, but not including movement time.

To determine the time efficiency of your code, we will compare your run time with an average of the TA’s and instructor’s runtimes. Performance points will be assigned based on this comparison, using a sliding scale. If your runtime is not within X times the base runtime, you will lose points. The value of X will be announced soon.

Test Report

We will be grading for correctness primarily by running your program on a number of test cases. If you have a small, silly bug that causes most of the test cases to fail, then you may get a very low score on that part of the programming assignment even though you completed 95% of the work. As mentioned above, most of your grade will come from correctness testing. Therefore, it is imperative that you test your code thoroughly. To encourage some degree of correctness testing, we require that you hand in your test cases in the form of a 1-2 page Test Report. Your Test Report should follow the template that can be found on http://www.eecs.umich.edu/courses/eecs281/f04/projects.html

Source code

You need to submit source code to the autograder. The files you must submit are: ai.cpp, ai.h, and any other files you use.

The Autograder

You can submit your code to the autograder a total of 10 times per day. For the first 3 submissions per day you will receive correctness feedback by email. After your first 3
submissions, you will simply receive a response indicating that your submission has been received. **You should NOT use the autograder as a debugger!** You will only get 10 total submissions per day, so make sure you have tested your code thoroughly before even contemplating submission to the autograder. The autograder has only enough functionality to ensure that you are getting the basics right (compiles, runs, solves some trivial cases, etc.) **but you still need to test your code carefully and thoroughly on your own.**

If any code takes too long to run, then the autograder will simply terminate it. ‘Too long’ will be determined based upon the average run time of the best performing projects, including that of the staff. (Hint: your program should not take more than 10 seconds in path-finding for any of the given example test cases.) Every time you submit your code to the autograder, it will log your run time. We will periodically post the top 10 performance numbers on the course web site or forum.

Despite its name, the autograder does not assign grades, it merely tests for correctness against previously created test cases. Your grade will be assigned after all final submissions are received and graded.

Every time code is submitted to the autograder, the old copy is **overwritten. We do not** keep backup copy of anyone’s code. It is the student’s responsibility to back up code. If you have working code, make sure you make a backup copy before making any more “improvements”. The adventurous among you may try to learn how to use version control software for this purpose. Your last submission to the autograder before the deadline is considered your final submission and is the one used to assign a grade.

**Submission to the autograder**

To submit your code to the autograder, use the submit281 script. An instruction explaining how to use the submit281 script can be found on [http://www.eecs.umich.edu/courses/eecs281/f04/projects.html](http://www.eecs.umich.edu/courses/eecs281/f04/projects.html)
Development Environment

You can download the framework used for project 1 from the course website. Source code must be developed in a Windows environment using Microsoft Visual Studio .NET 2002 or newer because this project utilizes modern video game technology, DirectX 9.0. The DirectX 9.0 SDK must also be installed in order to compile. Nearly all (95%) of the CAEN labs have this software installed and ready for use. The project also requires a 32 MB video card. Most CAEN machines have this as well, but certain labs (such as the Pierpont basement) do not. Your source code must be ANSI-C++ compliant. **Be sure not to use any Microsoft only tricks** (such as MFC) as your code will be tested without the DX Framework graphical environment, **in Unix under gcc**.

Classes

The game world is represented through several C++ classes (described briefly here and in more detail later in the project spec):

- *IVECTOR2*: a simple data structure that is used to represent a grid location;
- *C_Map*: contains a 2-D array of *C_Tiles*;
- *C_Tile*: contains basic *tile* information such as *(x, y)* position, *tile* passability, etc;
- *C_Item*: contains basic *item* type and *(x, y)* position;
- *C_Player*: contains data relevant to the Vanguard, such as its current *location*, current *map*, and AI data;
- *C_AI*: contains variables and functions for Search and Rescue using Vanguard.

Do not be overwhelmed by the number of classes listed here, as the only class you must work in is *C_AI*. All the functionality required from the other classes is listed in this document. In fact, there are many more classes in the project not listed here. Do not touch or use these other classes, as it could cause your code to not function properly on the autograder.

For more information on all classes and their member functions used in this project, refer to the Doxygen documentation that can found on [http://www.eecs.umich.edu/courses/eecs281/f04/projects.html](http://www.eecs.umich.edu/courses/eecs281/f04/projects.html)

**IVECTOR2**

Position is represented through the use of the *IVECTOR2* class. This is a simple class with X and Y int members. Some examples on how to use the class:

```c++
IVECTOR2 variableName;     // declares an IVECTOR2
IVECTOR2 variableName(startingX, startingY);  // declares an IVECTOR2 with
// startingX, startingY as initial values
variableName.x;            // returns the x value (int)
variableName.y;            // returns the y value (int)
IVECTOR2(someX, someY);    // returns an IVECTOR2 with
// someX, someY values
```
**C_Map**

Each game world is made up of multiple maps and each map is defined by the *C_Map* class. The *C_Map* class encapsulates a multi-dimensional array of *C_Tiles*. The *C_Map* class also contains methods to retrieve a *C_Tile* pointer at a coordinate location, as well as the ability to query whether or not a *C_Item* exists at a *IVECTOR2* location. In order to represent the entire game world, an array of *C_Map* pointers called *gp_GameWorldMaps* is defined globally. This array may be accessed to determine information about any map in the entire game world.

For example, if you wish to know the size of map 4 (the array is indexed by zero), then you would call the following function: *gp_GameWorldMaps[3]->GetSize()*. Each map could have completely different dimensions - they don't have to be all the same size or shape.

Some of the important functions and variables are:

```c
static C_Map::numMaps // Number of maps in world
C_Tile* GetTileAt(const *IVECTOR2 pos);// Returns a pointer to a tile at // this location
C_Item* GetItemAt(const *IVECTOR2 loc);// Returns a pointer to an item at // this location if one exists
IVECTOR2 GetSize(); // Returns the size of this map
```

![Figure 2. A map showing coordinates and direction.](image)

**C_Tile**

Each map is a set of M x N tiles, where M and N are not necessarily equal; however, they will both always be greater than or equal to four. Tiles are aligned in a Euclidean grid system. NORTH is in the upper right direction in the game world (see above image for a compass). The origin of each map is always the most NORTH WEST tile. A tile’s (x,y) coordinates begin at (0,0) and increase in x values to the right of the origin and increase in y values to the left of the origin. **Figure 2** shows a picture of one map and its coordinate system.
Each tile is either passable or impassable. Impassable tiles are the equivalent of walls and therefore the Vanguard may only travel on passable tiles. Each tile is defined by the \textit{C\_Tile} class. Some of the important functions are:

\begin{verbatim}
bool   IsPassable();   // Whether or not the tile is impassable
IVECTOR2 GetLocation();  // Returns an (x,y) location of the tile
int    GetMap();      // Returns which map the tile is on
ITEM_TYPE GetItemType();  // Returns an enum specifying what type
  // of item is on this tile
C_Tile* GetTeleporterTarget(); // If target tile is teleporter, returns
  // pointer to tile that teleporter leads
  // to, otherwise returns NULL
\end{verbatim}

\textbf{C\_Item}

In order to complete its mission, the Vanguard must gather the necessary \textit{items} \{ATU, WTD, Crystal\} \textit{before} proceeding to the stranded Beagle 2. The Vanguard can move in four directions to any adjacent, passable tile. The items are indicated visually on the map, but the Vanguard will not know any item location before the mission. However, the Vanguard can query a tile to see what, if anything, is on it.

A list of all the different items that the rover will encounter on Mars is: \{ATU, WTD, Crystal, Beagle 2, and Teleporter\}. Each one of these items is identified by the following respective enum types defined in \textit{item.h}:

\begin{verbatim}
enum ITEM_TYPE {ITEM_NONE, ITEM_ATU, ITEM_WTD, ITEM_CRYSTAL, 
  ITEM_TELEPORTPAD, ITEM_BEAGLE2};
\end{verbatim}

Each item is defined by the \textit{C\_Item} class. Some of the important \textit{C\_Item} member functions are:

\begin{verbatim}
ITEM_TYPE   GetItemType() const;  // Returns what type the item is
IVECTOR2    GetLocation() const;  // Returns a (x,y) location of the item
\end{verbatim}

\textbf{Apex Transmogrifier Unit (ATU)}

The ATU is a simple item and is located on a passable tile. Driving the Vanguard over the ATU picks it up. (The movement is handled automatically by the engine. Returning a path to the tile with the ATU on it will acquire this item.)

\textbf{Wavelet Transponder Dish (WTD)}

The WTD is a simple item and is located on a passable tile. Driving the Vanguard over the WTD picks it up. (The movement is handled automatically by the engine. Returning a path to the tile with the WTD on it will acquire this item.)

\textbf{Praesodymium Crystal}

A Praesodymium Crystal is a simple item and is located on a passable tile. Driving the Vanguard over the crystal picks it up. (The movement is handled automatically by the engine. Returning a path to the tile with the Crystal on it will acquire this item.)
**Beagle 2**

The Beagle 2 is a simple item and is located on a passable tile. The Beagle 2 cannot be picked up by the Vanguard. (The movement is handled automatically by the engine. Returning a path to the tile with the Beagle 2 on it completes the mission, provided the other three items have been obtained.)

**Teleporters**

Teleporters are symmetric, i.e. the Vanguard can take a teleporter from map X to map Y, and return to the same teleporter at map Y and return to map X. Any two tiles can be directly connected by only one teleporter. When the rover is on a teleporter, it can 'see' the tiles adjacent to its current location, and also the tile to which the teleporter leads. This tile is retrieved by using the function `C_Tile* C_Tile::GetTeleporterTarget()`. You may step on a transport at location (3, 5) in map 1, but exit it at location (9, 12) in map 5. Stepping on a teleporter in one map and stepping off the corresponding tile in the other map costs the same as stepping on and off an open tile on a single level, that is O(1).

**C_Player**

The AI routine in which you must implement your algorithm will be given a pointer to the current player object, `C_Player *p_Player`. Through this pointer, you will be able to query the current `IVECTOR2` location of the player which is required to begin the search. The player may also be used to query the Vanguard’s current inventory to discern whether or not Vanguard has already collected an item. The Vanguard is defined by the `C_Player` class. Some of the `C_Player` member important functions are:

```cpp
bool s_SearchFailure; // controls running
C_Tile* GetCurrentTile(); // Returns pointer to tile rover is on
int GetCurrentMap(); // Returns the index of the current map
IVECTOR2 GetGoalLoc(); // Returns current goal location as set by user by clicking mouse on a tile
int GetGoalMap(); // Returns current goal map as set by user by clicking mouse on a tile
bool HasATU(); // Returns true if Vanguard has ATU
bool HasWTD(); // Returns true if Vanguard has WTD
bool HasCrystal(); // Returns true if Vanguard has crystal
```

**C_AI**

Two files (ai.cpp and ai.h) represent the `C_AI` class. This class represents the “brain” of the Vanguard and is where your path-finding algorithm will reside. `C_AI` consists of two main functions:

```cpp
C_Tile* RunOneStep(); // Run one “step” of ai
void Reset(); // Reset AI Search State
```

The `RunOneStep()` function should run one iteration of the path-finding algorithm. This function has a return type of `C_Tile*`. The algorithm returns a tile by tile solution path when the algorithm finds a correct path. Otherwise, it returns NULL. A loose psuedo-code description of how this works follows:
while (game is running) {
    while (ai is running) {
        rover's target tile = RunOneStep();
        //rover's target tile is processed here by game engine
        if( reached goal || player input || search failure )
            Reset();
    }
}

You must use the static member C_Play::aiMethod to determine which path finding algorithm to run at any given time. This variable is an enum with STACK and QUEUE values. So, somewhere in your code should be something like:

if( C_Play::aiMethod == STACK )
    //run stack based path finding
else if (C_Play::aiMethod == QUEUE)
    //run queue based path finding

RunOneStep() should run one iteration of the path finding algorithm as described in Steps 1-4 of the Algorithm Description section. At the end of any iteration, if the goal is still not found, RunOneStep() should return NULL. However, if a tile enqueued is a goal tile, then the algorithm must begin returning the solution path. The first C_Tile pointer returned should be a pointer to the goal tile itself, then a pointer to the tile which led to the goal tile, etc, all the way back to the start tile. Thus, the algorithm needs a way of remembering which tile led to which tile.

A method for creating output of the solution path is an interesting problem, and is left to the student to figure out.

If your search ever fails, (step 3 in Algorithm Description section), then the static variable C_Player::s_SearchFailure must be set to true. The game will stop calling RunOneStep(), and will call Reset(). This should be completed only after all error output is handled. (Otherwise, that part of the code might not get called).

Reset() should reset the code and prepare it to run again. For example, you may want to reset the stack / queue retaining any information about which tiles have been enqueued, etc. Reset will be called whenever Vanguard reaches a goal, whenever user input is detected, or whenever a search fails.

After your algorithm returns a pointer to the start tile, the Vanguard will automatically start moving. The path tiles that are returned are highlighted in a greenish color, to help visualize the path. When the Vanguard reaches its destination, it will stop and call Reset(), and then start calling RunOneStep() again.
**Path Finding**

**Goals**
There are four possible goals for the rover \{ATU, WTD, Crystal, Beagle2\}. However, the Beagle2 may only become a goal after the rover has picked up the other three items. That is why the `C_Player::HasATU()`, `C_Player::HasWTD()`, and `C_Player::HasCrystal()` functions are provided. The order in which the first three items are picked up does not matter, however the rover does not have time to spare - thus, the first path returned must lead to the first goal tile that was encountered while searching. It is likely that the stack and queue algorithms will visit the goals in a different order. After the rover obtains all three items and arrives at Beagle2, the AI will be reset and automatically stop running.

**Algorithm Description**
To implement path finding using the queue-based algorithm, initialize your queue with the a pointer to the start tile (i.e., `p_Player -> GetCurrentTile()`) and then do the following:

1. Dequeue the next tile.
2. Enqueue all passable tiles that are adjacent to the tile was just dequeued. **Do not enqueue a tile that was previously enqueued.**

Usually there are four adjacent tiles to enqueue, where ‘adjacent’ means tiles in the NORTH, SOUTH, EAST, and WEST directions. So from location (x, y) the tiles that are adjacent to location(x,y), would be (x, y-1) ; (x, y+1) ; (x+1, y) ; (x-1, y).

However, if a teleporter tile was dequeued, then the adjacent tiles are those adjacent to the teleporter on the current map, as well as the destination teleporter tile (which is on a different map). That is, there are four adjacent tiles from a non-teleporter tile and five adjacent tiles from a teleporter tile. Use the `GetTeleporterTarget()` member function of `C_Tile` to get the fifth tile from teleporters.

For the sake of consistency and ease of grading, you must always enqueue in the following order:

1. WEST - (x-1, y)
2. NORTH - (x, y-1)
3. EAST - (x+1, y)
4. SOUTH - (x, y+1)
5. Teleporter tile (if present)

*Failure to enqueue in this order will result in a different path than the autograder's solution and potentially, loss of all solution points.*

When a tile is enqueued, call the `C_Tile::Enqueue(int)` function as follows:

```c
tilePointer->Enqueue(depth)
```

Depth is the current level of the search. The depth starts at zero, and increments by one each time a tile is dequeued. That is, the depth for the initialization tile is 0, for all adjacent tiles it is 1, and so on. This provides helpful visual feedback, and is
used by the engine to detect when a search has failed. Undefined behavior may occur if Enqueue is not called on every tile that is enqueued.

As the algorithm enqueues a tile, it should check to see if it is a goal tile. To do this, check to see if a needed item is on the tile. For example, to check if a tile contains the ATU, use:

```c
if( tilePointer->GetItemType() == ITEM_ATU ) //found atu
```

Here ITEM_ATU is an ITEM_TYPE enum. All item types are defined in this enum. See the Item section for its exact definition. If no tiles enqueued are goal tiles, then the algorithm should go back to Step 1.

3. If the queue becomes empty before the target is reached, then the search failed (there is no path to the target). When this occurs, set C_Player::s_SearchFailure to true.

4. If a goal tile is found, the path taken to the tile must be determined. The path may not backtrack onto itself. That is, determining the path should not require visiting the same tile twice. When the goal is found, return C_Tile pointers to each tile on the path.

For the stack-based implementation, implement a similar algorithm using push and pop instead of enqueue and dequeue. The above algorithm will need to be modified, but you must follow the basic outline above for both the queue and stack implementations.

There may be many possible paths. One of your two algorithms must be able to find the path with the lowest cost (smallest number of tiles traversed).

Since the size of this project is very large, and possibly intimidating, you are responsible for only coding and modifying ai.cpp, ai.h and any other files that to help in your algorithm. For example, you may wish to create stack.h and queue.h files to encapsulate your data structures. You should not make any changes to any provided files (except ai.cpp/ai.h), as your code will not compile correctly on the autograder.

**Input/Output**

**User Input**

The Vanguard has been equipped with remote input to enable motion on Mars. The numpad keys may be used to move Vanguard. The following keys are mapped to directional movement:

- Numpad 1: SOUTH
- Numpad 3: EAST
- Numpad 7: WEST
- Numpad 9: NORTH
- Numpad 5: Teleport

In addition to manual input for direct movement, Vanguard has three methods of
controlling pathfinding movement:

- **F2**: Sets AI to RUN_ONE_STEP, then switches back to DONT_RUN. The “F2” key is mapped to your pathfinding algorithm. By pressing the “F2” key the rover will call C_AI::RunOneStep() thereby calling your algorithm one time. This will certainly be helpful in debugging.

- **F3**: Sets AI to RUN_FULL_SPEED. Also the key “F3” is set to call RunOneStep() repeatedly. This is the equivalent of the code running in a while loop. RUN_FULL_SPEED stops on three conditions. First, it will stop on user input, such as a mouse click or numpad input. Second, it will stop if the rover successfully completes the mission (Gets all three items and gets to Beagle2). Third, it will stop when C_Player::s_SearchFailure is set to true.

- **F4**: Toggles AI debugging information on/off. The F4 key toggles visual debugging information for the pathfinding algorithm. If the pathfinding information is turned on, then tiles will change color as you call C_Tile::Enqueue(int depth) for each tile. Each tile also displays its depth, as described above, and the sequence number the tile was enqueued. Also, the solution path is colored once the Vanguard begins moving.

The Vanguard is also set up to define a user-defined goal by left-clicking the mouse on a tile. While the autograder will not check to make sure the algorithm correctly handles pathfinding to user-defined targets, you may find it helpful to define this behavior first for debugging. The user-defined goal, if defined, is available by the C_Player member functions:

```cpp
int C_Player::GetGoalMap()
IVECTOR2 C_Player::GetGoalLoc()
```

So, instead of checking a tile to see that it is a desired item, one could check the tile’s location and map with those returned by GetGoalMap() and GetGoalLoc(). Once again, this is completely optional. Make sure this doesn't interfere with the required behavior before submitting.

**File Output**

Upon successful path-finding to any of the required objects {ATU, WTD, Crystal, Beagle2}, Vanguard should output each of the following codes to an output log file. The filename is the string C_Play::m_OutputDir + "log.txt". For the project, C_Play::m_OutputDir will contain "data/", and thus the log will go to data/log.txt. However, the autograder will modify this variable, therefore you must output your log file to C_Play::m_OutputDir + "log.txt". To access this variable, you must include play.h.

- **SUCCESS_ATU**: Vanguard successfully found the ATU
- **SUCCESS_WTD**: Vanguard successfully found the WTD
- **SUCCESS_CRYSTAL**: Vanguard successfully found the Crystal
- **SUCCESSMISSION**: Vanguard successfully moved to Beagle2 after acquiring
the other three items.

In addition, the following are error codes for all the fatal errors that must be caught. Upon catching a fatal error, the program must print the error code to the log file, and exit with the appropriate exit code described below. The exit codes are defined in AI.h.

**Exit Codes**

- **EXIT_NOPATH**: Vanguard determined that there is no path to at least one object.
- **EXIT_NOTFOUND**: Vanguard determined that at least one item has not been found in the map

If your program successfully completes the mission the project the AI will switch to DONT_RUN automatically. You do NOT have to exit for a complete mission.

**Error codes:**

- **ERR_ATU_NOTFOUND**: Apex Transmogrifier Unit is not found on any map
- **ERR_WTD_NOTFOUND**: Wavelet Transponder Dish is not found on any map
- **ERR_CRYSTAL_NOTFOUND**: Crystal is not found on any map
- **ERR_BEAGLE2_NOTFOUND**: Beagle 2 is not found on any map

exit(EXIT_NOTFOUND) after your program has output to the log file the status of each item. That is, after your program has determined which items do not exist, and output to the log file, exit(EXIT_NOTFOUND). The Vanguard need not implement path finding when it detects that some items are missing.

- **ERR_NOPATH_TO_ATU**: Vanguard cannot find path to the ATU
- **ERR_NOPATH_TO_WTD**: Vanguard cannot find path to the WTD
- **ERR_NOPATH_TO_CRYSTAL**: Vanguard cannot find path to the Crystal
- **ERR_NOPATH_TO_BEAGLE2**: Vanguard cannot find a path to the ATU

exit(EXIT_NOPATH) after your program has output to the log file the status of each item. That is, after your program has determined which items can not be pathed to, and output to the log file, exit(EXIT_NOPATH).

Please format output to the logfile in the following manner: For every error or success code, output the code then place a comma and newline character as a delimiter. For example, if the algorithm produces a successful mission (i.e., algorithm finds a path to each item and Beagle 2 in the order: ATU, Crystal, WTD, Beagle2), then the following is correct output:

```
SUCCESS_ATU,
SUCCESS_CRYSTAL,
SUCCESS_WTD,
SUCCESS_MISSION,
```

If the algorithm does not find any of the items on the map, then output an error for each missing item and exit(EXIT_NOTFOUND). For example, if the ATU or the Crystal was not found, then the program would output the following to the output directory and exit(EXIT_NOTFOUND ).
ERR_ATU_NOTFOUND,
ERRCRYSTAL_NOTFOUND,

If the algorithm cannot find a path to any items in the map, then the algorithm needs to output an error for each item that the rover cannot reach, and exit(EXIT_NOPATH). For example, if the algorithm finds the ATU and WTD but then fails, then the program would output the following to the output directory and then exit(EXIT_NOPATH):

SUCCESS_ATU,
SUCCESS_WTD,
ERR_NOPATH_TOCRYSTAL

The path will be output automatically to data/path.txt. You do not have to write any code to output the path. They are output as the tiles are returned in your C.AI::Run function.
Debugging Console

The game is equipped with a debugging console to help debug the algorithm. To view the console, press TAB. To hide it, press TAB again.

Outputting text to the console is very similar to cout. To use the console, there are two things that must be included in your files:

```cpp
#include "engine/console.h"
extern C_Console* gp_Console;
```

These lines are already included in ai.cpp for you. To output text to the console, simply do the following:

```cpp
*gp_Console<< "Hello World!"<<endl;
```

Output is also defined for integers, `IVECTOR2`, and other useful classes. For example:

```cpp
*gp_Console<<" Display 100: "<<100<<endl;
*gp_Console<<" Now Display (1,4): "<<IVECTOR2(1,4)<<endl;
```

displays:

```
Display 100: 100
Now Display (1,4): (1,4)
```

If you do not output an `endl`, your text will not display until the next `endl` is output. To overcome this, you may also use:

```cpp
*gp_Console<<"Hello World!;
gp_Console->Flush();
```

You MUST use either `Flush()` or `endl`, otherwise your debugging text may not display at all.

There are also several console commands available. To use these, type the command into the console, and simply press enter. A confirmation message will appear in the console window.

The commands are:

time: This command toggles the AI timing information display. The AI timing information is described in further detail elsewhere.

tile: This command toggles the tile information display. It displays the current rover and mouse positions.

clear: This command clears the console.

norender: This command stops the game from rendering (drawing to the screen) anything, but everything else (including your AI) still runs as normal. This is useful for debugging large maps, as it will speed up your AI process.

darkness: This command stops the game from simulating darkness. This may help speed up rendering.
Test Cases

Eight example test cases are included in the framework.

- atu
- beagle2
- crystals
- Dagobah
- RedPlanet
- SimpleMap
- TimeCube
- wtd

We may post other cases as they become available. You can also use the map editor to build your own test cases.
Coding style

We will check for empirical efficiency by measuring the memory usage and running time of your code and by reading the code. Given that this is a course on data structures and algorithms, we will focus on efficient use of data structures and algorithms, and not so much on the structure of the code beyond what has been pointed out below.

We will focus on whether you use an $O(n)$ algorithm or an $O(n^2)$ algorithm, but not whether you use printf's or cout's. In general, if the tradeoff is between illegible and fast code vs. pleasant to read code that is unnoticeably less efficient, we will prefer the latter. (Of course pleasant to read code that is also efficient would be best.)

Use a reasonable organization for your overall program:

Design a fairly reasonable class structure. On the one hand, don't stick everything into one class. On the other hand, don't be bureaucratic and require the reader to follow one class definition after another to find a single line of code wrapped in layers of methods, with each method doing nothing but calling the next one. Make reasonable use of constructors. Where needed be sure to supply a custom destructor. If the way you design your code feels sloppy to you, it probably is. Utilize multiple files in a way that is consistent with the general use of C++. Don't use more files than necessary.

Don't use literals!

Use either const, enum, or #define to give your literals meaningful names. We do deduct points for each occurrence of literals, even if it is the same one. The only exceptions will be for loop counter, command-line options, NULL(0) and TRUE/FALSE(1/0) testing/setting, and help and error messages printed out to user.

Use reasonable comments:

Explain what each class does and what each data member is used for. A one or two line description of most member functions is also desirable. Where you use non-standard coding techniques, document them. List your name and the date last modified for each file.

Remember that a useless comment is worse than no comment at all.

```
int temp; // declare temp. variable
```

would be an example of a useless comment which just makes code harder to read!

Use reasonable formatting:

From indentation alone, it should be obvious where a given code block ends. Use reasonable and informative variable names, but limit name size to a reasonable length. A 40-character name better have a very good reason to exist. Avoid lines that wrap in an 80 column display wherever possible. Your code should be tight, compact, and visually tidy.

Variable names:

Variable names like `i' and `j' can be reasonable, but you should not use such variables
to store meaningful long-term data. Other than LCV (loop control variables) you should use descriptive names for your variables, functions, classes, methods, structures, etc.

**Code reuse:**
If you find yourself rewriting basically the same code more than once, stop and try to see if you can somehow reuse the code by making it a function call.

**Hints and advice**

- Always think through your data structures and algorithms before you code them. It is important that you use efficient algorithms in this programming assignment and in this course, and coding before thinking often results in inefficient algorithms.

- This is not an easy programming assignment. Start it immediately.

**Motivation**

“Path-finding constitutes one of the primary *maps* of game AI and is a major concern in almost all modern computer games. The simple act of producing a route from A to B could introduce a significant CPU hit, depending on the complexity of the world and the manner in which the task is broken down.” (Board and Ducker, Dogfish Entertainment)

“A good path finding engine can be used for many more purposes than just moving units around the world. In the real-time strategy game *Empire Earth*, we used the pathfinding engine for tasks such as terrain analysis, choke-point detection, AI military route planning, weather creation/movement, AI wall building, animal migration routes, and of course, unit path finding.” (Higgins, Stainless Steel Studio)