CIDM 2315 Final Project: Hunt the Wumpus

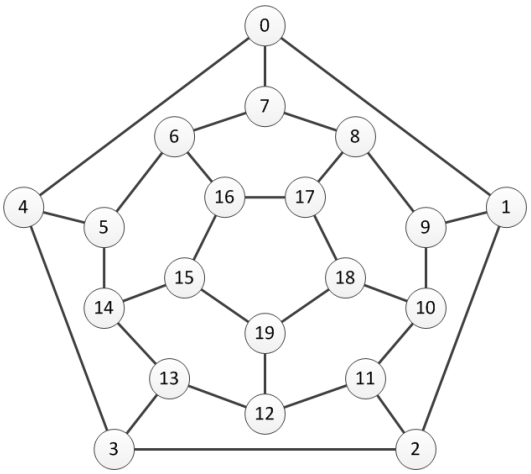
Description

You will implement the popular text adventure game ***Hunt the Wumpus***. Hunt the Wumpus was originally written in BASIC in 1972 by Gregory Yob. You can read the original description, instructions, and source code at https://www.atariarchives.org/bcc1/showpage.php?page=247. Our version will be a little bit different from the original. ***Hunt the Wumpus*** has since been ported to many other platforms, including, recently, the iPhone, Android, and Web.

I encourage you to download the game on your phone and try it out. You can play a web-based version at http://www.ifiction.org/games/playz.php?cat&game=249&mode=html.

In ***Hunt the Wumpus*** the player hunts a mysterious creature called the Wumpus. The Wumpus lives deep inside a dark cave network. The cave is shaped like a dodecahedron with 20 rooms and tunnels connecting them.

The player can move from room to room or shoot an arrow into a room hoping to hit and kill the Wumpus. Other hazards lurk in this cave as well! Bats can pick up and move the player to another room or the player can fall into a bottomless pit. The game gives clues to the location of these hazards. If the player is in an adjacent room, the game says, “Bats nearby”, “I feel a draft”, or “I smell a Wumpus”. Gameplay takes the form of exploration and deduction. It is your job to add all these features to the game.



Dodecahedron Cave Design

Learning Objectives

This project will require knowledge of the following programming techniques:

- Reading, understanding, and modifying existing code.
- Control Structures.
- Methods.
- Arrays and multi-dimensional arrays.
- Debugging, testing, and code maintenance techniques.
- Building a large, complex program with several parts.

Task 0: Read and Understand the Code

Thoroughly read through the existing code and documentation. Compile it. Run it. Move the player through the rooms. Become familiar with the design and methods. Answer the following questions completely in a separate Word document and submit those answers with your project.

1. Consider the static variables declared on lines 11-15:
    a) What kind of variables they (not the data type, but the kind based on their location in the code)?
    b) Why are those variables declared inside the class instead of inside a method?
2. In your own words, explain the design of the `adjacentRooms` rectangular array. What does the array represent?
3. Conceptually what does `adjacentRooms.GetLength(0)` represent? Be specific within the context of Hunt The Wumpus.
4. Conceptually what does `adjacentRooms.GetLength(1)` represent? Be specific within the context of Hunt The Wumpus.
5. Look at the loop condition for running the game (inside `Main()`). When will the game end?
6. Currently there are ten (10) methods besides `Main()`. List each method header and briefly describe what that method does.

Task 1: Shoot the Wumpus

Our player is unarmed and cannot defeat the Wumpus. Arm the player with arrows that he can shoot at the Wumpus. The arrows can be shot into any adjacent room. Write code that does the following:

1. Implement the 'shoot' command. If the player types 'shoot' the program should ask 'Which Room?' The player can then type in an adjacent room number to shoot into that room.
    - Detect if the player types in an invalid room and print out "You cannot shoot there."
    - **[Hint]** Look at the code for the 'move' command. How different is shoot from move?
    - Run the program. Verify that you handled invalid rooms correctly.
2. Once the player shoots the arrow into a room:
    - If the room HAS the Wumpus:
        i. Print out "ARGH…Splat!"
        ii. Print out "Congratulations. You killed the Wumpus! You Win." The game ends.
    - If the room does not have the Wumpus:
        i. Print out "Miss! But you startled the Wumpus".
        ii. Shooting the arrow startles the Wumpus and it may move to **an adjacent** room. Move the Wumpus to **any adjacent** room (it may stay in the same room also) including the room you are currently in -- in which case the player will die. Create a separate method to move the startled Wumpus.
        iii. Print out a trace message that tells you which room the Wumpus moved to.
3. The player only has three (3) arrows. Keep track of ammo and let the player know how much ammo they have. It is up to you to decide how to handle when the player runs out of ammo.
4. Run the program. Test both missing the Wumpus and hitting it.

Task 2: Add Bats

The Wumpus isn't the only creature that calls these caves home. There are other hazards. Superbats are large and scary bats that when startled pick you up and fly you to a random room. Implement the following to add superbats to the game:

1. Place bats in TWO random rooms at the start. Bats cannot be in room 0, the same room as the Wumpus, or the same room as other bats.
2. Print out trace messages that tell you which rooms the bats are in.
3. If you enter a room adjacent to the bats, you can hear them and you should print out "Bats nearby!" Implement this feature. Look at `InspectCurrentRoom()`to see how it is implemented for the Wumpus. Your code will be very similar to that.
4. Run the program. Test that you can detect the bats in an adjacent room.
5. If you enter a room with the bats, the bats fly you away to any random room. Look at `InspectCurrentRoom()`. That is where you will write the code to identify if you entered a room with bats. How can you add logic that will then move the player to another random room?
    - The game should display "Snatched by superbats!" when bats move you to a new room.
    - The new room might contain the Wumpus, a bottomless pit, or even other bats and you have to handle that. The bats moving you should work just like a normal move.
6. Run the program. Test that the bats will fly you to another room.

Task 3: Add Bottomless Pits

Another hazard in these caves is the bottomless pit. If you enter a room with a bottomless pit you will fall to your death. Add bottomless pits to your game:

1. Place bottomless pits in TWO random rooms at the start. A bottomless pit can be in any room except room 0.
    - If a bottomless pit and bats are in the same room, the bats will fly you to safety.
    - The Wumpus has sucker feet and won't fall into the bottomless pit if they are in the same room.
2. Print out trace messages that tell you which rooms have the bottomless pits.
3. If the player enters a room with a bottomless pit, the player will die and the game will end.
    - The game should display "YYYIIIIEEEE…fell in a pit."
4. Run the program. Go into a room with a bottomless pit and verify that it works.
5. If you enter a room adjacent to a bottomless pit you feel a draft and should print out "You feel a draft…" Implement this feature. It will be very similar to your implementation for the bats.
6. Run the program. Test that bottomless pits work as described.

Task 4: Implement the 'quit' command

We want to allow the player to quit mid-way through a game. Implement the 'quit' command. If the player types 'quit' the game should end and return to the main menu.

Task 5: Play again?

If the player dies from the Wumpus or falls into a pit, they may want to replay the same exact map again. Implement code that does that.

1. Once the game is over, prompt the user if they would like to replay the same map again.
2. If the player says yes, start a new game but on the exact same map with the Wumpus, bats, and bottomless pits all in the same places.
3. If the player says no, the game should end and return to the main menu.

Task 6: Keeping Score

Let's start to keep score now. Come up with your own scoring rules and implement them. Once the game ends, tell the player their score. Where should you declare the score variable?

Some possible scoring rules:

- The player should get points for killing the Wumpus.
- The player should lose points for getting eaten or falling into a bottomless pit.
- Maybe the player should get more points for killing the Wumpus quickly (less moves).
- Maybe the player should lose points for shooting but missing the Wumpus.

Task 7: Save the Score to Disk

Use the File I/O techniques we just learned to save the player's high score to a file called `WumpusHighScore.txt` on the Hard Drive.

1. After you display the high score to the player, save it to the disk.
2. You should append to the existing file so all previous high scores remain.
3. Run the program. Test that the high score is successfully written to disk. Test that multiple high scores will correctly append to the file.

Task 8: Implement ViewHighScores()

Implement the ViewHighScores() method to load the high score file from disk and display them one high score per line.

Task 9: Add Treasure

Add a treasure to the game. Place a gold bar somewhere in the cave. If the player finds it, they get extra points. Notify the player that they found gold bar.

Task 10: Change Starting Location

Tasks 10 and 11 try to make the game more dynamic and re-playable. Right now, the player always starts in room 0 and the cave layout is always the same. Change the gameplay mechanics so that the player can start in any random room. Be sure to adjust the starting location of the Wumpus, bats, and pits as well. They can now be in room 0 but not in the starting room of the player.

Task 11: Add a New Type of Cave

Our cave layout is a dodecahedron. It is static and the layout will never change. Implement one other type of cave that is a different layout. When the game starts, ask the player which cave layout they would like to use (or surprise them!) and build that type of cave network.

Consider the design of the `adjacentRooms` rectangular array and how you can initialize it to different values based on user input.

Some possible cave layouts are listed at the end of this document. Each number represents the room id starting at room 0. You can also make your own design.

Finishing Up Tasks

1. Remove all trace or debugging messages you wrote. This should make the game much more challenging and fun now -- ready for prime time.
2. Implement the `PrintInstructions()` method. Just display a few instructions to get a new player started in the game.

Extra Credit Tasks:

1. Add more cave types in task 11. Even more extra credit if you add caves with a different number of adjacent rooms. Maybe some rooms are dead ends and others are hubs to even more rooms?
2. Implement the “crooked arrow” feature. In the original game, the player’s arrows are “crooked” can be shot through five rooms at once.
3. Anything else exciting and neat that you can think of. As long as it requires some level of programming difficulty, just let me know. Think you know a good way to implement player health? Maybe they have other ammo or weapons? Maybe more unique hazards or other enemies? Want to try to add colors using the Console.Color methods? Go for it!

Submission Requirements:

1. Complete task 0. Write your answers in a Word document.
2. Complete all ten programming tasks including the finishing up tasks.
3. Comment your code for each task. Include in the comments what difficulties you encountered and how you overcame them or why you decided on a particular solution.
4. Zip the entire HuntTheWumpus Visual C# project into one zip file.
5. Upload both the zipped project and the Word Document for task 0 to the Final Project Drop Box.

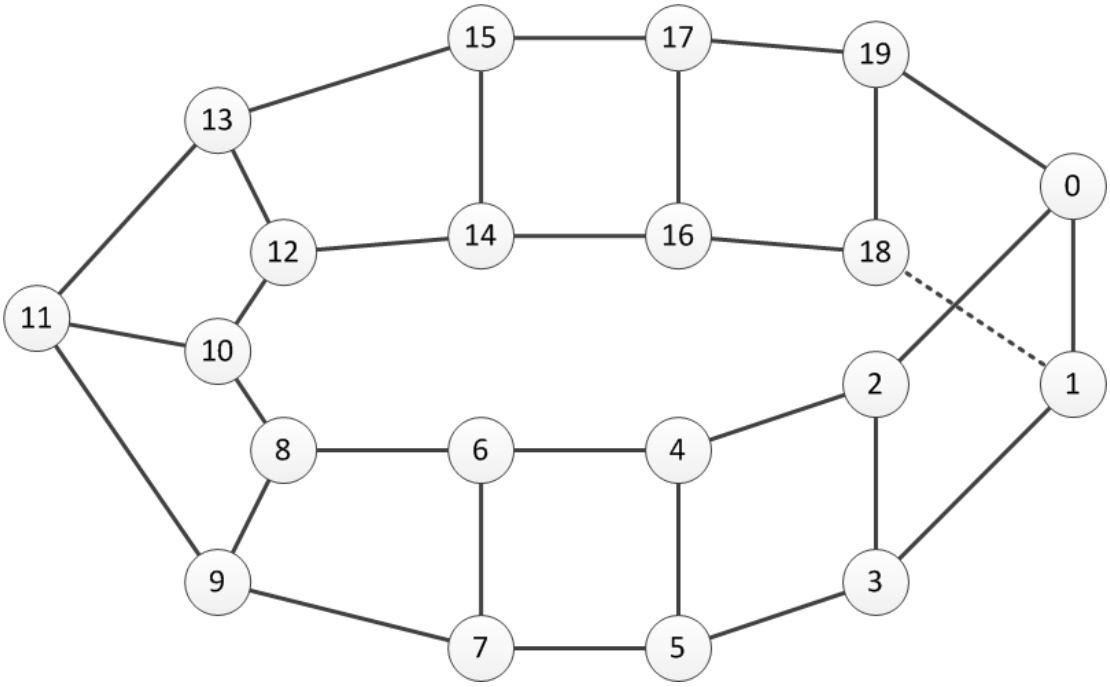
Evaluation Criteria

This final project will be evaluated similar to all previous homework assignments with a focus on two areas: correctness and design.

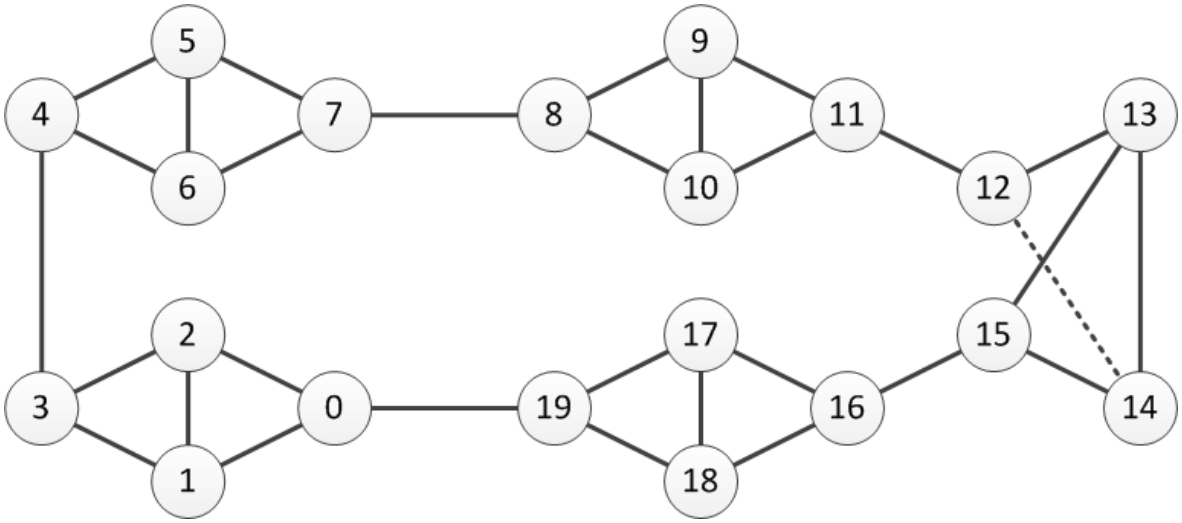
For correctness, I will check that you fully implemented the rules of the game as described and completed each programming task. The game must work and it must not crash. In particular I will also check whether you handle invalid input correctly.

For design, I will examine the source code and check that it follows reasonable control structure/method techniques. I will check whether your methods are small and self-contained. I will ensure that your code, methods, and variables are reasonably named, designed, and well commented. I will pay close attention to Task 11 as well.

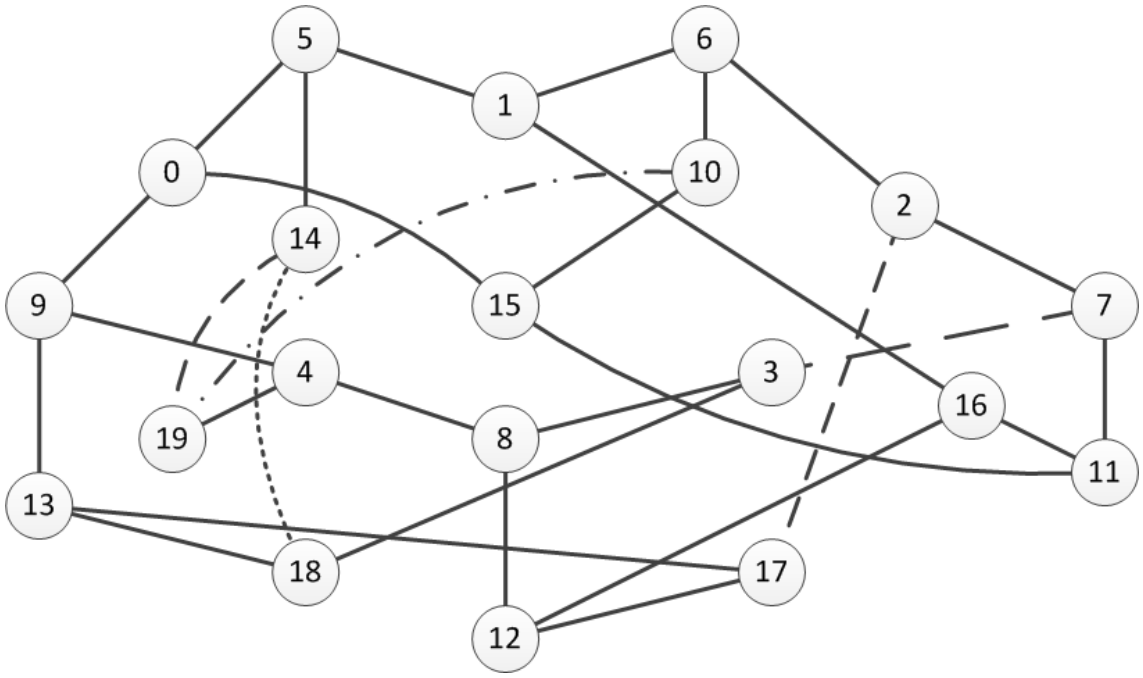
Possible New Cave Layouts for Task 11



Cave Layout 1 - The Mobius Strip



Cave Layout 2 - String of Beads



Cave Layout 3 - Toroidal Hex