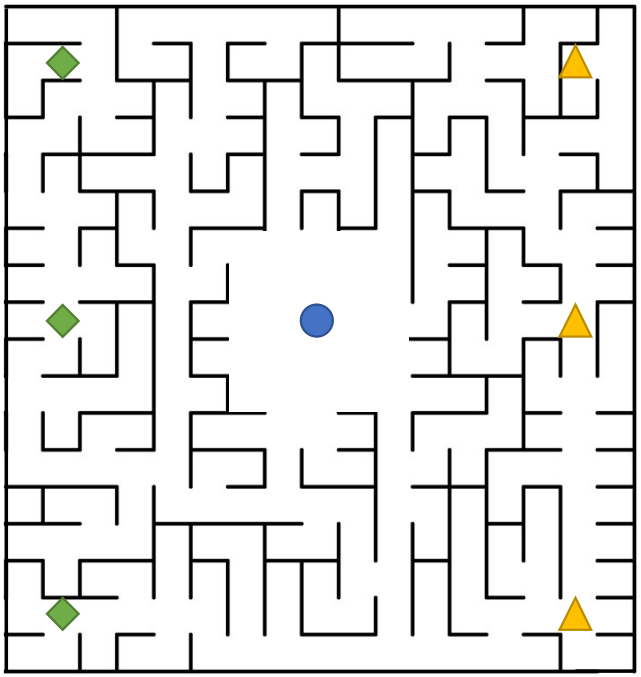
[9] Try it out! (20 minutes)



[9] Try it out! (20 minutes)

1. In the core package, create a new `GameState` class, which will hold a `gameEnded` flag (boolean, true if the game has ended, false otherwise), a maximum number of game ticks constant, a game tick counter (to be initialised to 0 in the `ForwardModel.setup()`), the button object, and 6 player objects in a data structure of your choice; no values to be assigned to these variables by default. Ignore the maze (actual location of objects) for now. Add an empty default constructor.
   - Add a method named `copy` which takes no arguments and returns a deep copy of the `GameState` object. You might need to add such methods on all custom objects as well (`Player`, `Button`).
2. In the core package, create a new `ForwardModel` class, which will contain the functionality of our game in several methods:
   - `void setup (Player[] players, GameState gameState) {}`
     - Sets up the initial state of the game (within the gameState object provided), including creating the `Button` instance.
     - Assign all variables not initialised in the constructor for `Player` objects: player ID, team ID and initial positions.
     - Consider all positions to be integers going from 0 to 287 (16x18 2D grid, counting from top-left corner by rows towards bottom-right corner).
     - The `Player` class should hold a new `ForwardModel` variable, which is passed to them in this method (passing the current object, `this`).
   - `void next (int[] actions, GameState gameState) {}`
     - Ignore player actions for now. Just increase the game tick in the game state by 1, and apply the game rules:
       - Check if any player pressed the button and trigger the correct functionality.
       - Check if the game has ended. If it did, give all players their correct win status (-2 = undecided, -1 = lose, 0 = tie, 1 = win) and change the gameEnded flag in the gameState.

[9] Try it out! (20 minutes)

3. Modify the `Player` class to receive a `GameState` object in the act method (**copy** of the real one)**.**
4. Modify the `Player, Button` and `GameState` classes to implement custom toString() methods each (the Player and Button should print their current states, and the GameState class should print nicely formatted information about the state of the button and all players).
5. In the `Run` file, create an array of `Player` objects containing **6** instances of `RandomPlayer`. Then, create an instance of the `GameState` and `ForwardModel` objects. In the GameState class, set the maximum number of game ticks to **5** for this exercise.
6. Then, setup the initial game state and print it out (using the custom `toString()` method in the `GameState` class).
7. Create the main game loop, which runs until the game has ended. Within the loop:
   a) All players who are still alive are asked for an action, given a copy of the current game state
   b) Their actions are added into an array, where index in array corresponds to player ID. If the player is out of the game (was not asked for an action), add -1 in the array; otherwise, add their returned action into the array.
   c) Call the `ForwardModel.next()` method with the array of player actions and the current game state.
   d) Print the game state.
8. When the game has ended, print the win status of all players.

3: Data Structures

Graphs & Trees