Seminar Assignment 1

Žan Mervič and Klemen Kerin

November 18, 2022

1 Description of the problem

We are given a vector of strings of length n which represents an $n \times n$ maze. Our goal is to find a path out of a maze where # characters represent a wall, . represent empty spaces, and S and E represent the starting and ending points. Later we also add a T character, which represents a treasure.

We can move through the maze in four directions, left, right, up and down. Our task is to use a genetic algorithm that will be able to find a path as short as possible out of any maze represented in such a way. The path is written as a string of moves from the start to the end of our maze (example: "LULLURDD"). Later addition includes a goal of treasure collection while finding the shortest path to the ending point. The mazes provided to us by the assignment and used for testing our algorithm are given at the end of this report.

2 Solution

As the main tool for our assignment we chose the R programming language, specifically, its GA function. The GA function uses a genetic algorithm for the maximization of the provided fitness function and is defined as

ga(type, fitness, nBits, maxiter, run, popSize, pmutation, mutation, population),

where

- type represents the type of vector used as a representation of our agents (in our case binary),
- fitness represents the fitness function which we're trying to maximize,
- nBits represents the number of bits our binary representation of agents holds,
- maxiter represents the maximal number of iterations of the genetic algorithm,
- run represents the number of runs after getting the same results before stopping the algorithm,
- popSize represents the population size,
- pmutation represents the mutation change,
- mutation represents the custom mutation function,
- population represents the custom population function.

Our goal was to find the right settings as well as to provide the right fitness, mutation and population functions such that the genetic algorithm would optimally solve the given maze and collect all the treasures.

We started by thinking about the right vector representation for our agents. We had to encode the characters which represent the moves in our path. We decided on a **binary representation** featuring the combination of 00 as 'U', 01 as 'R', 10 as 'D' and 11 as 'L'.

The idea of our fitness function is rather straightforward, the main difficulty was assigning the right amount of reward for getting close to the end or collecting the treasure and punishment for getting away from the end, into the wall or revisiting already visited points. As most of the treasures in mazes we tested on lie in the "blind alleys", which force the algorithm to walk away from the end and on already visited points, we had to set out the penalty rules just right if we wanted our algorithm to collect all the treasures while also coming to the end. The highlights of our fitness function are shown below.

```
fitness <- function(path){
  # We get the coordinate vector of our starting pozition,
  # we initialize all the other variables
  maze_matrix = map_reading(my_maze)
  start <- as.vector(t(which(maze_matrix == 'S', arr.ind = TRUE)))
  exit <- as.vector(t(which(maze_matrix == 'E', arr.ind = TRUE)))
  current = start
  width = length(maze_matrix[1,])
  penalties <- 0.0

  # We go through the path vector changing each encoded step
  # into a move and updating our current location
  for (i in seq(1,length(path), 2)) {
    move <- path[i : (i+1)]
    if(all(move == c(0,0))){ #UP"
      current <- current - c(1,0)
    } else if(all(move == c(0,1))){ #RIGHT
      current <- current + c(0,1)
    } else if(all(move == c(1,0))){ #DOWN
      current <- current + c(1,0)
    } else if(all(move == c(1,1))){ #LEFT
      current <- current - c(0,1)
    } else{
      print("Error in fitness")
    }
                                .
                                .
                                .

    if(maze_matrix[current[1], current[2]] == 'T'){
      # If we collect a treasure we get rewarded and the visited tiles reset
      penalties <- penalties - 200
      maze_matrix[maze_matrix == 'o'] <- '-'
      maze_matrix[maze_matrix == 'S'] <- 's'
    }

    if(maze_matrix[current[1], current[2]] == 'E'){
      # We made it to the exit, we reward the path with the least steps
      return (1000 + 1000/i - min(penalties, 0))
    }

    # We mark the tile as visited, so we don't visit it again
    maze_matrix[current[1], current[2]] <- "o"
  }
  # All moves have been done and we did not make it to the end
  return(100/sqrt((current[1] - exit[1])^2 + (current[2] - exit[2])^2) - penalties)
}
```

The idea of our custom mutation function is to choose a random move from the parent, check all the moves leading up to that move and if we hit a wall before we come to the randomly selected move, we change that move to a valid one, otherwise we change the randomly selected move to a new valid move. This idea

required another helper function which gives us a valid move for a specific position in our path. The full mutation function is shown below.

```
custom_mutation <- function(object, parent) {
  # We select the parent vector from the population
  mutate <- parent <- as.vector(object@population[parent,])
  n <- length(parent)
  # Sample a random vector move that should be changed
  # We make sure j is a odd number (first number of a move pair)
  j <- sample(1:n, size = 1)
  if (j %% 2 != 1){
    j <- j -1
  }

  # We now go from the first to the j-th bit and check if there is a wall hit.
  # If there is, we make a mutation on the bit responsible for the hit
  maze_matrix = map_reading(my_maze)
  current <- as.vector(t(which(maze_matrix == 'S', arr.ind = TRUE)))
  move <- c(0,0)
  for (i in seq(1, j, 2)) {
    move <- mutate[i : (i+1)]
    if(all(move == c(0,0))){ #UP
      current <- current - c(1,0)
    } else if(all(move == c(0,1))){ #RIGHT
      current <- current + c(0,1)
    } else if(all(move == c(1,0))){ #DOWN
      current <- current + c(1,0)
    } else if(all(move == c(1,1))){ #LEFT
      current <- current - c(0,1)
    } else{
      print("Error in custom mutation")
    }
    if(!all(current > 0) || !all(current < dim(maze_matrix)) || maze_matrix[current[1], current[2]] == '#'){
      # If we hit a wall before we get to the randomly selected j bit we change j to the current bit
      # So we make the mutation where we would have hit the wall
      j = i
      break
    }
    if(maze_matrix[current[1], current[2]] == 'E'){
      return(mutate)
    }
  }

  # Changing that element to a random move that is different from the invalid one
  new_move <- generate_valid_move(maze_matrix, move, current)[2,]

  mutate[j] <- new_move[1]
  mutate[j + 1] <- new_move[2]
  return(mutate)
}
```

The last custom function required for our genetic algorithm is the population function which generates the population of valid paths for a given maze. Here we start of by making a random binary matrix where each row represents one agent. We set the number of rows before in our settings and compute the appropriate length for our vectors using the following equation $\lceil \frac{n^2}{2} \rceil \cdot 2$, which ensures we have long enough vectors with an even number of elements. Finally, we loop through all the randomly generated vectors and check the validity of moves. If we find an invalid move we use the helper function from before to generate a new valid one. The highlights of our population function are shown bellow.

```
custom_population <- function(object,...){
  # We create a starting random matrix of our population
  my_population <- matrix(sample(0:1, my_nBits * my_popSize, replace = TRUE), nrow=my_popSize)

  # We set the minimum required steps for our entity to be valid
  required_steps <- my_required_steps

  # We check how many steps have been made before going into a wall,
  # if it's less than the required steps, we generate a new value
  maze_matrix = map_reading(my_maze)
  for(row in 1:my_starting_population_size){
    path <- my_population[row,]
    steps_made <- 0

    # We count the moves with our current path and change the path if invalid
    current <- as.vector(t(which(maze_matrix == 'S', arr.ind = TRUE)))
    for (i in seq(1, length(path), 2)) {
      move <- path[i : (i+1)]

                                        .
                                        .
                                        .

      # If we hit a wall we generate a different (valid) move
      if(!all(current > 0) || !all(current <= dim(maze_matrix)) || maze_matrix[current[1], current[2]] == '#'){
        temp <- generate_valid_move(maze_matrix, move, current)
        move <- temp[2,]
        path[i:(i+1)] <- move
        current <- temp[1,]
        if(!all(current > 0) || !all(current <= dim(maze_matrix)) || maze_matrix[current[1], current[2]] == '#'){
          print("Something went wrong, aborting")
          break
        }
      }

      if(maze_matrix[current[1], current[2]] == 'E'){
        # If we have a solution that reaches the finish we keep it
        break
      }

      steps_made <- steps_made + 1

      # If we made enough steps we stop the iteration
      if(steps_made >= required_steps){
        break
      }
    }

    my_population[row,] <- path
    steps_made = 0
  }
  return(my_population)
}
```

3 Results

3.1 Maze completion analysis

As the input mazes for the analysis of our algorithm, we used the mazes provided to us by the assignment available at the end of the report. First, we tested the mazes without treasures using the custom functions we described above and the following GA settings.

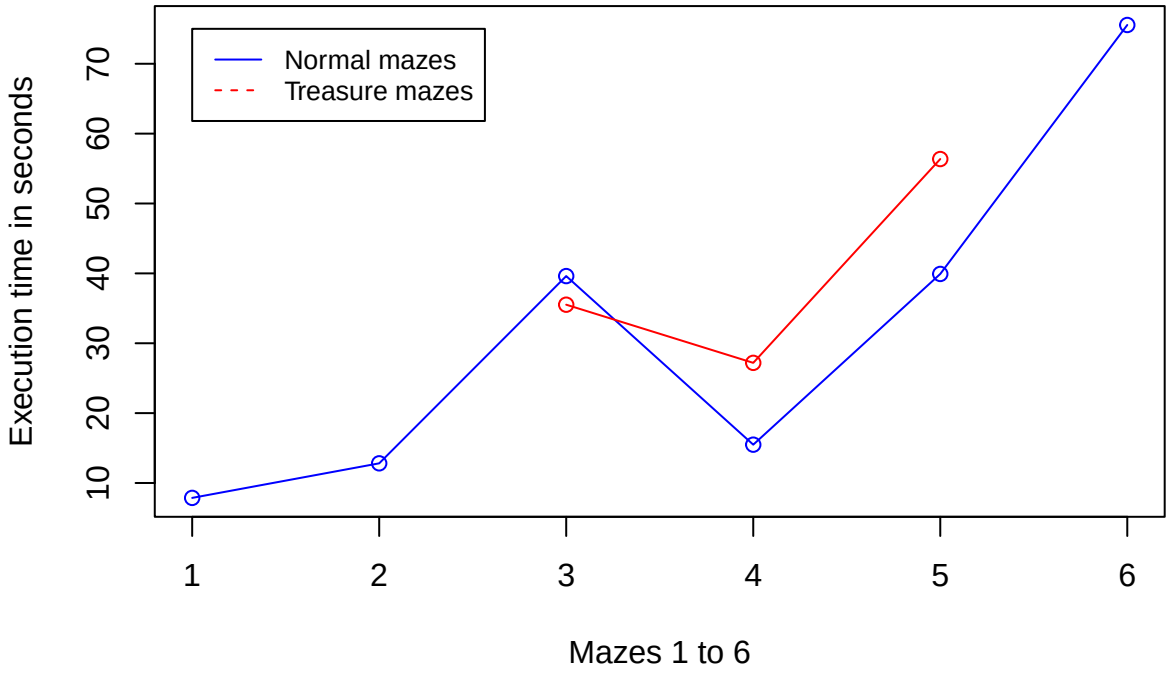
```
my_nBits <- ceiling((nchar(my_maze[1])^2) / 2) * 2 #number of bits per agent
my_popSize = 1000 #population size
my_maxiter = 1000 #number of iterations
my_run = 50 #number of runs
my_pmutation = 0.3 #mutation probability

# Starting population settings
my_required_steps = floor(my_nBits/2) #number of valid steps first gen entities need to make
my_starting_population_size = my_popSize #number of "trained" first gen entities
```

We concluded that our algorithm can consistently solve mazes 1, 2, 3, 4, 5 and 6. The execution time between mazes varied a lot, we measured it three times for each maze and computed the average execution time per maze.

After our initial maze analysis, we started to focus on mazes with treasures. First, we had to modify the starting population settings a bit, since we weren't getting the desired results with our usual settings. We set the number of valid steps for agents in our initial population to a maximum of 20 since we didn't want them to accidentally find the end in the first population and thus miss the treasure. We also set the "trained" starting population size to just one, allowing our GA to make a completely random population.

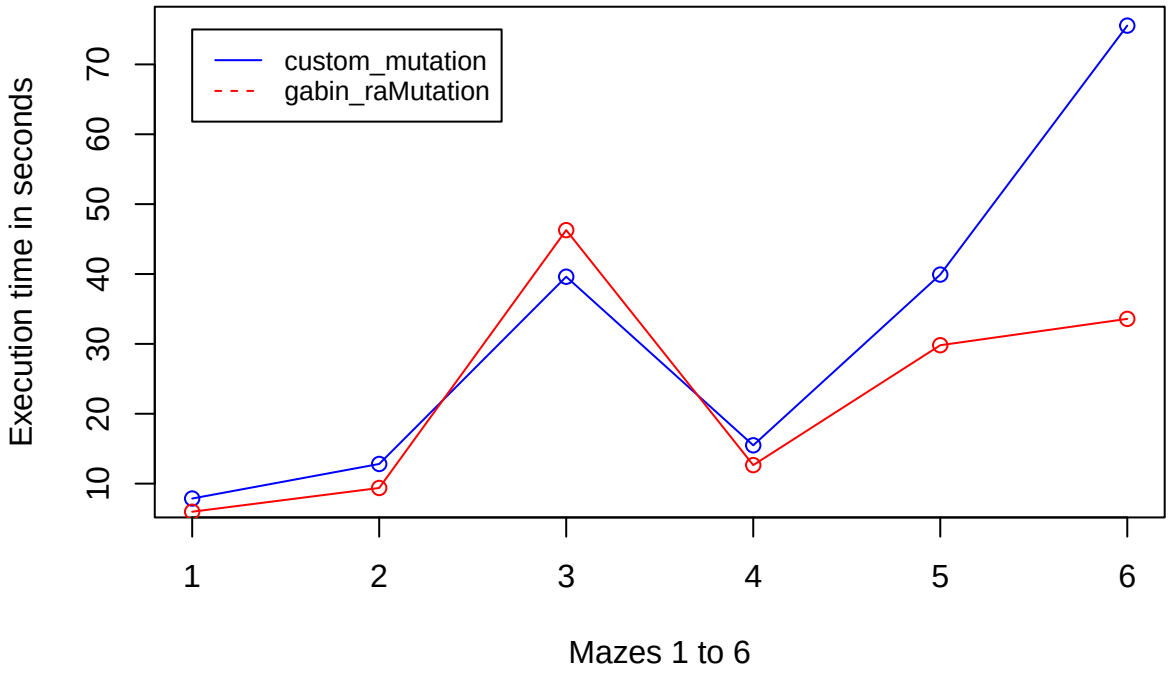
```
# Starting population setting for treasure mazes
my_required_steps = max((my_nBits / 2), 20) #low value is best for treasure mazes since it doesn't allow
#the first gen entities to reach the end thus getting many points and missing the treasure
my_starting_population_size = 1 #number of "trained" first gen entities
```

Our algorithm can consistently solve treasure mazes 3T, 4T and 5T. We used the same approach for measuring the time as before. The results for both types of mazes are shown in the graph below.



3.2 Mutation function analysis

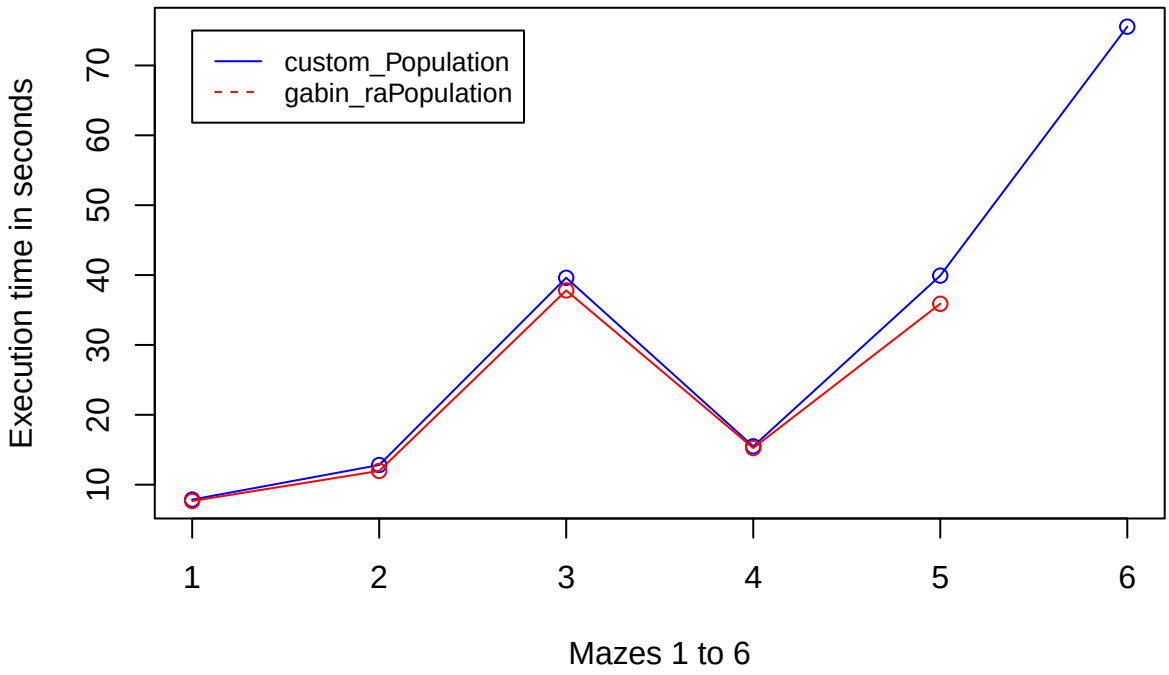
We analyzed the performance of our custom_mutation function by comparing it to the gabin_raMutation function included in the GA library. Using the new function our algorithm successfully solved non-treasure mazes 1, 2, 3, 4, 5 and 6. We computed the average execution time for each non-treasure maze and compared it to the previous results.



As we can see in the graph above, the gabin_raMutation function outperformed ours on almost all mazes that the algorithm could solve. It performed worse on the third maze, which is a quite complicated maze featuring many turns. Our function also did a lot better on maze number 7, where despite failing to complete the maze, the algorithm using our function optimized for a substantially better fitness value (275,8 versus 40,2). The reason for these results is the complexity of our function. Because of its feature to return a valid mutation and not just a random one, its time performance was worse, but the algorithm's ability to find a path was better because of it. We can see this in the example of mazes 7 and 3, where the algorithm found a path faster using our mutation function. A large credit for the successful completion of the six mazes using the gabin_raMutatino function also goes to our custom_population function, which provides an initial population of valid paths, which oftentimes include an already correct path.

3.3 Population function analysis

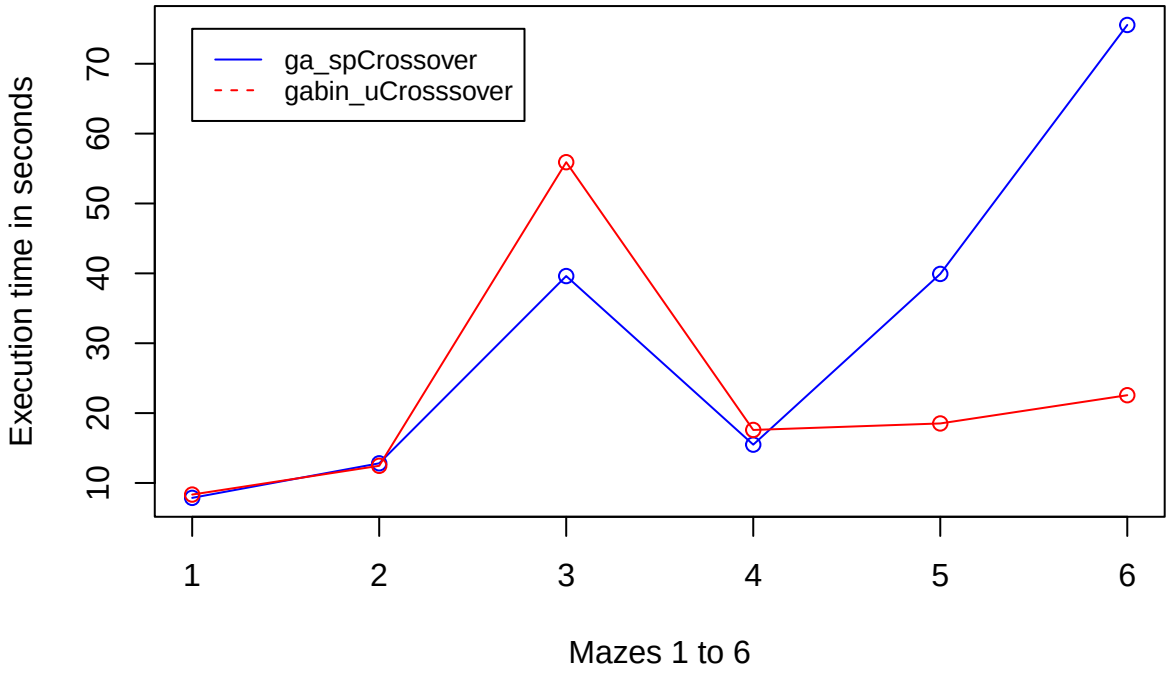
We compared the algorithm's performance using our custom_population function to its performance using the gabin_Population function from the GA library. Using the new function our algorithm successfully solved non-treasure mazes: 1, 2, 3, 4, and 5. We computed the average execution time for each non-treasure maze and compared it to the previous results.



The time performance results for mazes that the algorithm could solve are very similar, we can see a slightly better time performance from gabin_Population since it's probably a simpler function. The only difference when using the gabin_Population function is that the algorithm can't solve maze 6, the reason being in custom_Population building a population of valid paths, unlike the gabin_Population.

3.4 Crossover function analysis

Till now we used the default GA crossover function gabin_spCrossover. Because we didn't make our own, we decided to test it against the other binary crossover function from the GA library called gabin_uCrossover. We could solve mazes 1, 2, 3, 4, 5 and 6. Here are the average execution time results.



From the graph, we can see that there are some notable differences in execution time for the third, fifth and sixth mazes. Through testing the function, we found out that the new gabin_uCrosssover doesn't bring as big of a variety to the population as the ga_spCrossover does. This means that the algorithm needed more time to find out the path for maze 3. It also finished faster for mazes 5 and 6 because the path was already found in the initial population, but the algorithm didn't optimize the path as much as it does when using the gabin_spCrosssover (best fitness function value was lower).

```r
maze1 = c("##E##",
          "#...#",
          "#...#",
          "#S..#",
          "#####")

maze2 = c("#####E#",
          "#.#.#.#",
          "#.....#",
          "##.####",
          "#....S#",
          "#.#..##",
          "#######")

maze3 = c("##########",
          "#S.......#",
          "########.#",
          "#........#",
          "#.########",
          "#........#",
          "########.#",
          "#........#",
          "#.########",
          "#E########")

maze4 = c("#####E####",
          "#...#....#",
          "#.######.#",
          "#.#......#",
          "#.###.####",
          "#....S...#",
          "#####.##.#",
          "#..##.####",
          "##.......#",
          "##########")

maze5 = c("###E###########",
          "#........###..#",
          "########.....##",
          "#.....##.###.##",
          "#.###.....##..#",
          "#.....###...###",
          "####.####.#.###",
          "##......#.....#",
          "#######.#.###.#",
          "#.........#####",
          "#.####.S#....##",
          "##....#.#######",
          "#..##.........#",
          "##...###.##.###",
          "###############")

maze6 = c("##################",
          "##...#####...#####",
          "##.#.##....#.....#",
          "##.S....##...###.#",
          "###.####.#.#.....#",
          "#........#.##.####",
          "##.#######......##",
          "##......##########",
          "#####.####....##.#",
          "##.........##..#.#",
          "#######.#####.##.#",
          "####.##.##.....#.#",
          "####.##.########.#",
          "#................#",
          "#..###############",
          "##.#.##.#.##.##.##",
          "##...............#",
          "#########E########")

maze7 = c("#####################",
          "#...................#",
          "#.###############.###",
          "#.#########.......###",
          "#.#........#######..#",
          "#.##.##.##........##",
          "#.#...#.##.#######.##",
          "#.###.#.##.....##.##",
          "#.###..###########.##",
          "#.####.###.........#",
          "#.#....#...####.####",
          "#.#.####.####.....##",
          "#.#......#......####",
          "#.###.####.######...#",
          "#.#......#.....#.###",
          "#.######...######.#.#",
          "#.#.....##S#...#.#.#",
          "#.#########..#.....#",
          "#..........###.##.##",
          "##########E##########")


maze3_T = c("##########",
          "#S.......#",
          "########.#",
          "#..T.....#",
          "#.########",
          "#........#",
          "########.#",
          "#..T.....#",
          "#.########",
          "#E########")

maze4_T = c("#####E####",
          "#...#....#",
          "#.######.#",
          "#.#T.....#",
          "#.###.####",
          "#....S...#",
          "#####.##.#",
          "#..##.####",
          "##....T..#",
          "##########")

maze5_T = c("###E###########",
          "#........###..#",
          "########.....##",
          "#..T..##.###.##",
          "#.###.....##..#",
          "#.....###...###",
          "####.####.#.###",
          "##......#.....#",
          "#######.#.###.#",
          "#.........#####",
          "#.####.S#....##",
          "##....#.#######",
          "#..##T........#",
          "##...###.##.###",
          "###############")


maze7_T = c("#####################",
          "#...................#",
          "#.###############.###",
          "#.#########.......###",
          "#.#.T......#######..#",
          "#.##.##.##........##",
          "#.#...#.##.#######.##",
          "#.###.#.##....T##.##",
          "#.###..###########.##",
          "#.####.###.........#",
          "#.#....#...####.####",
          "#.#.####.####.....##",
          "#.#......#......####",
          "#.###.####.######...#",
          "#.#T.....#.....#.###",
          "#.######...######.#.#",
          "#.#.....##S....#.#.#",
          "#.#########..#...T.#",
          "#..........###.##.##",
          "##########E##########")
```