UPC
Universidad Peruana
de Ciencias Aplicadas

Tarea Académica 3

Programación Concurrente y Distribuida

Integrantes :

Guillén Rojas, Daniel Carlos	U201920113
Wu Pan, Tito Peng	U201921200
Sebastian Gonzales	U201923816

Profesor :

Jara Garcia, Carlos Alberto

Septiembre 2023

Índice

1. Descripción del Juego
2. Reglas del Juego
    2.1. Tablero del Laberinto
    2.2. Turnos y Movimientos
    2.3. Obstáculos
    2.4. Objetivo
    2.5. Modificaciones y Uso de Canales
3. Tareas a Implementar
    3.1. Implementación de Canales
    3.2. Lógica del Movimiento
    3.3. Sincronización y Gestión de Turnos
    3.4. Fin del Juego
4. Conclusión
5. Bibliografía

1. Descripción del Juego

El Ludo modificado es una versión ampliada y adaptada del popular juego de mesa Ludo. En esta versión, los jugadores compiten para llevar a sus personajes a través de un peligroso laberinto lleno de obstáculos y desafíos. Cada jugador tiene la tarea de guiar a sus personajes a través del laberinto y llegar a la meta antes que los demás.

2. Reglas del Juego

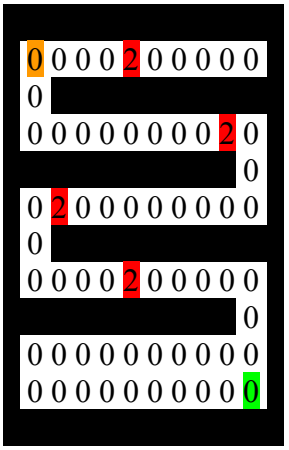
2.1. Tablero del Laberinto

El tablero se encuentra dividido en numerosas casillas que presentan una red de caminos y obstáculos.

```
12 12 1 1
1 1 1 1 1 1 1 1 1 1 1 1
1 0 0 0 0 2 0 0 0 0 0 1
1 0 1 1 1 1 1 1 1 1 1 1
1 0 0 0 0 0 0 0 0 2 0 1
1 1 1 1 1 1 1 1 1 1 0 1
1 0 2 0 0 0 0 0 0 0 0 1
1 0 1 1 1 1 1 1 1 1 1 1
1 0 0 0 0 2 0 0 0 0 0 1
1 1 1 1 1 1 1 1 1 1 0 1
1 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1
```

Laberinto del juego, donde:
0 - Representa espacio donde el jugador puede desplazarse
1 - Representa las paredes o límites del laberinto
2 - Representa las trampas

Presentación Gráfica



Los jugadores deben terminar el laberinto iniciando en las casillas naranjas hasta el verde.

Cada jugador tiene cuatro personajes que comienzan en puntos de partida específicos.

```
players = []Player{
        {Name: "Player 1", Position: pos{1, 1}, Pieces: 4,
Direction: int(Up), Turno: 0},
        {Name: "Player 2", Position: pos{1, 1}, Pieces: 4,
Direction: int(Up), Turno: 1},
        {Name: "Player 3", Position: pos{1, 1}, Pieces: 4,
Direction: int(Up), Turno: 2},
        {Name: "Player 4", Position: pos{1, 1}, Pieces: 4,
Direction: int(Up), Turno: 3},
    }
```

Se inicializan los jugadores con 4 personajes o piezas denominados “Pieces” en el juego.

2.2. Turnos y Movimientos

Los jugadores se turnan para lanzar un dado y mover a sus personajes.

```
for {
        playerChannel <- true
        if turno != player.Turno {
            <-playerChannel
            continue
        }

        turno = (turno + 1) % 4

        fmt.Printf("%s ", player.Name)
        player = move(player, rollDices(),
Direction(player.Direction))

        if exitCheck(player.Position) {
            player.Pieces--
            player.Position = pos{GameBoard.startRow,
GameBoard.startColumn}
            fmt.Printf("%s Finish 1 run, Rest (%d) Pieces.
\n", player.Name, player.Pieces)
            if player.Pieces == 0 {
```

```
                fmt.Printf("%s Win\n", player.Name)
                <-GameChaneel
                break
            }
        }
        <-playerChannel
    }
```

Los jugadores entran el el canal de “playerChaneel” donde si les toca su turno pueden tirar el dado y desplazarse en el laberinto, si jugadores entra y no es su turno, no se le deja jugar hasta que entre el jugador que debe tocarle su turno.

Los jugadores lanzan tres dados, dos dados normales (del 1 al 6) y uno con la operación (sumar o restar) para determinar cuántos pasos pueden avanzar o retroceder en su turno.

```
func rollDices() int {
    roll_1 := rand.Intn(6) + 1
    roll_2 := rand.Intn(6) + 1
    roll_3 := rand.Intn(1) + 1
    if roll_3 == 0 {
        fmt.Printf("rolled (+): %d\n", roll_1+roll_2)
        return roll_1 + roll_2
    } else {
        if roll_1-roll_2 > 0 {
            fmt.Printf("rolled (-): %d\n", roll_1-roll_2)
            return roll_1 - roll_2
        } else {
            fmt.Printf("rolled (-): %d\n", roll_1-roll_2)
            return 0
        }
    }
}
```

Se inicializan los 3 dados donde los primeros 2 tienen valor de 1 - 6 y el tercero 0 - 1, si sale 0 se le asigna “+” a la operación, en caso contrase se asigna “-”. En caso de salir negativo en las operaciones de “-”, se le asigna el valor de 0 de salida de los dados.

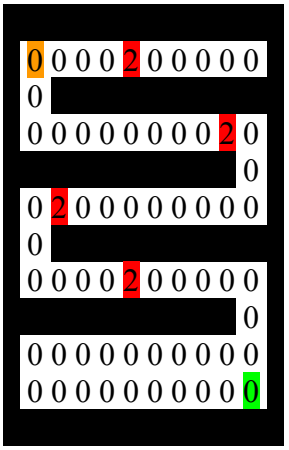
Los jugadores pueden mover un solo personaje por turno.

Los personajes deben avanzar exactamente la cantidad de pasos indicados por la operación de los dados (valor del primer dado y operador (+ -) con el valor del segundo dado).

2.3. Obstáculos

El laberinto está lleno de obstáculos como paredes, trampas y criaturas que bloquean el paso de los personajes en varias casillas.

Presentación Gráfica



Laberinto del juego, donde:
0 - Representa espacio donde el jugador puede desplazarse
1 - Representa las paredes o límites del laberinto
2 - Representa las trampas

Si al personaje le toca avanzar hacia una casilla con obstáculo entonces el jugador pierde el turno y continúa el siguiente jugador.

```
if GameBoard.maze[players.Position.i][players.Position.j]
== 2 {
            fmt.Printf("%s Moving at (%d, %d) in direction
%d\t", players.Name, players.Position.i,
players.Position.j, players.Direction)
            fmt.Printf("%s Fall in Tramp.\n",
players.Name)
            return players
        }
```

Si el jugador cae en un casilla donde se encuentra el trampa, automáticamente pierda su turno, es decir, si tiro 6 en los dados, solo avanza 3 pasos y cae en el trampa, ya no puede seguir jugando y pierde el turno

2.4. Objetivo

El objetivo es llevar a los cuatro personajes desde los puntos de partida hasta la meta en el menor número de turnos posible.

El primer jugador en llevar a todos sus personajes a la meta gana el juego.

```
if exitCheck(player.Position) {
            player.Pieces--
            player.Position = pos{GameBoard.startRow,
GameBoard.startColumn}
            fmt.Printf("%s Finish 1 run, Rest (%d) Pieces.
\n", player.Name, player.Pieces)
            if player.Pieces == 0 {
                fmt.Printf("%s Win\n", player.Name)
                <-GameChaneel
                break
            }
        }
```

Los jugadores deben dejar todos su piezas en la meta del juego, cuando uno de los jugadores ya no tengo piezas (su número de “Pieces” es 0), se le declara automáticamente como ganador.

2.5. Modificaciones y Uso de Canales

En esta versión modificada del juego, los jugadores y el tablero de juego están representados como entidades concurrentes separadas que se comunican a través de canales. Cada jugador tiene su propio canal de comunicación con el tablero del juego para enviar movimientos y recibir actualizaciones del estado del juego.

```
func main() {
    initGameBoard("GameBoard.in")
    playerChannel = make(chan bool, 1)
    GameChaneel = make(chan bool, 1)

    for i := range GameBoard.maze {
        for j := range GameBoard.maze[i] {
            fmt.Printf("%d\t", GameBoard.maze[i][j])
        }
        fmt.Println()
    }
```

```
    GameChaneel <- true

    go play(&players[0])
    go play(&players[1])
    go play(&players[2])
    go play(&players[3])

    GameChaneel <- true
}
```

Se implementa el uso de 2 canales, 1 para los jugadores y otro para el juego. El canal de los jugadores recibe a todos los jugadores que van a participar con su turno predeterminado. El canal del juego verifica cuando hay un ganador para terminar el proceso de todo si aparece uno.

3. Tareas a Implementar

3.1. Implementación de Canales

```
playerChannel chan bool
GameChaneel   chan bool

func play(player *Player) {

    for {
        playerChannel <- true
        if turno != player.Turno {
            <-playerChannel
            continue
        }

        turno = (turno + 1) % 4

        fmt.Printf("%s ", player.Name)
        player = move(player, rollDices(),
Direction(player.Direction))

        if exitCheck(player.Position) {
            player.Pieces--
            player.Position = pos{GameBoard.startRow,
GameBoard.startColumn}
```

```
            fmt.Printf("%s Finish 1 run, Rest (%d) Pieces.
\n", player.Name, player.Pieces)
            if player.Pieces == 0 {
                fmt.Printf("%s Win\n", player.Name)
                <-GameChaneel
                break
            }
        }
        <-playerChannel
    }
}

func main() {
    initGameBoard("GameBoard.in")
    playerChannel = make(chan bool, 1)
    GameChaneel = make(chan bool, 1)

    for i := range GameBoard.maze {
        for j := range GameBoard.maze[i] {
            fmt.Printf("%d\t", GameBoard.maze[i][j])
        }
        fmt.Println()
    }

    GameChaneel <- true

    go play(&players[0])
    go play(&players[1])
    go play(&players[2])
    go play(&players[3])

    GameChaneel <- true
}
```

Se implementa el uso de 2 canales, 1 para los jugadores y otro para el juego. El canal de los jugadores recibe a todos los jugadores que van a participar con su turno predeterminado. El canal del juego verifica cuando hay un ganador para terminar el proceso de todo si aparece uno.

La creación de un canal para cada jugador no fue necesario y es un tarea que enreda mucho la lógica del juego. Puesto que el comportamiento de utilizar 4 canales para cada jugador genera que cada uno de ellos quiera jugar al mismo tiempo sin respetar los turnos, lo que hace que después se tenga que generar otro canal para verificar los turnos.

3.2. Lógica del Movimiento

```
func move(players *Player, dice int, dir Direction)
*Player {

    players.Direction = int(dir)
    fmt.Printf("%s at (%d, %d) in direction %d\n",
players.Name, players.Position.i, players.Position.j,
players.Direction)

    for i := 0; i < dice; i++ {
        if Direction(players.Direction) == Up {
            //look left
            if
GameBoard.maze[players.Position.i][players.Position.j-1]
!= 1 {
                players.Position =
players.Position.move(direction[0].Left)
                players.Direction = int(Left)
                //look up
            } else if
GameBoard.maze[players.Position.i-1][players.Position.j]
!= 1 {
                players.Position =
players.Position.move(direction[0].Up)
                players.Direction = int(Up)
                //look right
            } else if
GameBoard.maze[players.Position.i][players.Position.j+1]
!= 1 {
                players.Position =
players.Position.move(direction[0].Right)
                players.Direction = int(Right)
                //look down
            } else if
GameBoard.maze[players.Position.i+1][players.Position.j]
!= 1 {
                players.Position =
players.Position.move(direction[0].Down)
                players.Direction = int(Down)
            }
```

```
        } else if Direction(players.Direction) == Left {
            //look down
            if
GameBoard.maze[players.Position.i+1][players.Position.j]
!= 1 {
                players.Position =
players.Position.move(direction[0].Down)
                players.Direction = int(Down)
                //look left
            } else if
GameBoard.maze[players.Position.i][players.Position.j-1]
!= 1 {
                players.Position =
players.Position.move(direction[0].Left)
                players.Direction = int(Left)
                //look up
            } else if
GameBoard.maze[players.Position.i-1][players.Position.j]
!= 1 {
                players.Position =
players.Position.move(direction[0].Up)
                players.Direction = int(Up)
                //look right
            } else if
GameBoard.maze[players.Position.i][players.Position.j+1]
!= 1 {
                players.Position =
players.Position.move(direction[0].Right)
                players.Direction = int(Right)
            }
        } else if Direction(players.Direction) == Right {
            //look up
            if
GameBoard.maze[players.Position.i-1][players.Position.j]
!= 1 {
                players.Position =
players.Position.move(direction[0].Up)
                players.Direction = int(Up)
                //look right
            } else if
GameBoard.maze[players.Position.i][players.Position.j+1]
!= 1 {
```

```
                players.Position =
players.Position.move(direction[0].Right)
                players.Direction = int(Right)
                //look down
            } else if
GameBoard.maze[players.Position.i+1][players.Position.j]
!= 1 {
                players.Position =
players.Position.move(direction[0].Down)
                players.Direction = int(Down)
                // look left
            } else if
GameBoard.maze[players.Position.i][players.Position.j-1]
!= 1 {
                players.Position =
players.Position.move(direction[0].Left)
                players.Direction = int(Left)
            }
        } else if Direction(players.Direction) == Down {
            //look right
            if
GameBoard.maze[players.Position.i][players.Position.j+1]
!= 1 {
                players.Position =
players.Position.move(direction[0].Right)
                players.Direction = int(Right)
                //look down
            } else if
GameBoard.maze[players.Position.i+1][players.Position.j]
!= 1 {
                players.Position =
players.Position.move(direction[0].Down)
                players.Direction = int(Down)
                //look left
            } else if
GameBoard.maze[players.Position.i][players.Position.j-1]
!= 1 {
                players.Position =
players.Position.move(direction[0].Left)
                players.Direction = int(Left)
                //look up
            } else if
```

```
GameBoard.maze[players.Position.i-1][players.Position.j]
!= 1 {
                players.Position =
players.Position.move(direction[0].Up)
                players.Direction = int(Up)
            }
        }

        if
GameBoard.maze[players.Position.i][players.Position.j] ==
2 {
            fmt.Printf("%s Moving at (%d, %d) in direction
%d\t", players.Name, players.Position.i,
players.Position.j, players.Direction)
            fmt.Printf("%s Fall in Tramp.\n",
players.Name)
            return players
        }

        if exitCheck(players.Position) {
            fmt.Printf("%s Moving at (%d, %d) in direction
%d\n", players.Name, players.Position.i,
players.Position.j, players.Direction)
            return players
        }
    }
    fmt.Printf("%s Moving at (%d, %d) in direction %d\n",
players.Name, players.Position.i, players.Position.j,
players.Direction)
    return players
}
```

La función de mover al jugador recibe de paramentos el jugador, el número que sacó de los dados y la dirección donde está mirando. En un for interactivo, el jugador se mueve de 1 paso a la vez por la cantidad de número que sacó del dado. Primero según la dirección donde la pieza del jugador esté mirando, va a verificar en sentido horario donde se puede mover (se puede mover en todos lados mientras que no se una pared “1”). Luego, se verifica con 2 condiciones; el primero si el jugador cae en una trampa, automáticamente pierde el turno y pasa el siguiente jugador; el segundo verifica si llego al meta, y llegó al meta termina el turno del jugador. Finalmente si ya se movio según el número de dados, entonces termina y devuelve los nuevos valores del jugador.

3.3. Sincronización y Gestión de Turnos

```
func play(player *Player) {
    for {
        playerChannel <- true
        if turno != player.Turno {
            <-playerChannel
            continue
        }

        turno = (turno + 1) % 4

        fmt.Printf("%s ", player.Name)
        player = move(player, rollDices(),
Direction(player.Direction))

        if exitCheck(player.Position) {
            player.Pieces--
            player.Position = pos{GameBoard.startRow,
GameBoard.startColumn}
            fmt.Printf("%s Finish 1 run, Rest (%d) Pieces.
\n", player.Name, player.Pieces)
            if player.Pieces == 0 {
                fmt.Printf("%s Win\n", player.Name)
                <-GameChaneel
                break
            }
        }
        <-playerChannel
    }
}
```

Todos los jugadores tiene un turno predeterminado, por lo que cada vez que el canal del jugador recibe a uno, verifica primero si es el turno de ese jugador, en caso de ser el turno de ese jugador, se ejecuta con normalidad y cambia el turno del siguiente jugador, en caso contrario no se le permite a ese jugador jugar y espera hasta que llegue el jugador del turno.

3.4. Fin del Juego

```
if exitCheck(player.Position) {
            player.Pieces--
            player.Position = pos{GameBoard.startRow,
GameBoard.startColumn}
            fmt.Printf("%s Finish 1 run, Rest (%d) Pieces.
\n", player.Name, player.Pieces)
            if player.Pieces == 0 {
                fmt.Printf("%s Win\n", player.Name)
                <-GameChaneel
                break
            }
        }
```

En caso de que un jugador tenga una pieza en la meta, se le resta 1 de sus piezas e inicia la siguiente pieza en la posición inicial. Cuando la cantidad de piezas de un jugador llega a 0, automáticamente se lo declara como ganador y termina la partida.

4. Conclusión

La implementación exitosa del juego de Ludo modificado mediante programación concurrente subraya la viabilidad y eficacia de aplicar estos conceptos en el desarrollo de sistemas interactivos y en tiempo real. Esto destaca la relevancia de comprender y aplicar los principios de concurrencia y comunicación entre procesos, lo que abre la puerta a futuras aplicaciones en el desarrollo de software y sistemas informáticos complejos.

La comprensión profunda y la aplicación exitosa de la concurrencia y la gestión de procesos en el contexto del juego de Ludo modificado demuestran la importancia de los fundamentos teóricos. Este proyecto enfatiza la necesidad de una comprensión sólida de los conceptos subyacentes, como la sincronización de procesos y la comunicación a través de canales, para abordar con éxito desafíos prácticos y situaciones de programación del mundo real.

5. Bibliografía

Repositorio de Github:

https://github.com/TitoWuPan/Trabajo-Final-Concurrente-y-Distribuida-CC65/tree/Tarea-Académica-3