Optimus Jr Robot Pathfinding - Complete Code Explanation

Problem Overview

This is a **pathfinding problem** where a robot needs to navigate from a starting point '@' to a destination '$' on a 2D grid with various obstacles and special mechanics.

Core Data Structures

1. Direction Enum

```go
type Direction int

const (
  SOUTH Direction = iota // 0
  EAST              // 1
  NORTH             // 2
  WEST              // 3
)
```

Why this approach?

- Using `iota` creates sequential integer constants
- Makes direction comparison fast (integer comparison)
- Easy to iterate through directions in priority order

2. Position Struct

```go
type Position struct {
  row, col int
}
```

Purpose: Represents a coordinate on the 2D grid. Simple but essential for tracking robot location.

3. State Struct

```go
```

```go
type State struct {
    pos       Position  // Current position
    direction Direction // Current facing direction
    inverted  bool      // Are direction priorities inverted?
    breaker   bool      // Is robot in breaker mode?
}
```

Critical for loop detection: This captures ALL aspects of robot state. Two identical states = potential infinite loop.

4. Robot Struct (Main Controller)

```go
type Robot struct {
    grid        [][]rune              // The map
    height      int                   // Map dimensions
    width       int
    startPos    Position              // Starting '@' location
    destPos     Position              // Destination '$' location
    teleporters map[rune][]Position   // Digit teleporters (1-9)
    state       State                 // Current robot state
    visited     map[string]bool       // For loop detection
    path        []Direction           // Solution path
}
```

Key Design Decisions:

- `[][]rune` for grid: Handles Unicode characters, mutable for wall destruction
- `map[rune][]Position` for teleporters: Each digit can have exactly 2 positions
- `map[string]bool` for visited: String keys for complex state comparison
- `[]Direction` for path: Records the solution sequence

Algorithm Flow

1. Initialization (NewRobot)

```go
```

```go
func NewRobot(grid [][]rune, height, width int) *Robot {
    // ... initialization code

    // Scan entire grid once to find special elements
    for i := 0; i < height; i++ {
        for j := 0; j < width; j++ {
            cell := grid[i][j]
            switch cell {
            case '@':
                robot.startPos = Position{i, j}
                robot.state.pos = Position{i, j}
                robot.state.direction = SOUTH  // Always starts facing SOUTH
            case '$':
                robot.destPos = Position{i, j}
            case '1', '2', '3', '4', '5', '6', '7', '8', '9':
                robot.teleporters[cell] = append(robot.teleporters[cell], Position{i, j})
            }
        }
    }
}
```

Interview Points:

- **Single-pass initialization:** Efficient O(H×W) setup
- **Automatic teleporter pairing:** No need to manually link teleporter pairs
- **Immutable references:** Start/dest positions never change

2. State Management & Loop Detection

```go
func (r *Robot) getStateKey() string {
    return fmt.Sprintf("%d,%d,%d,%t,%t",
        r.state.pos.row, r.state.pos.col,
        r.state.direction, r.state.inverted, r.state.breaker)
}
```

Why this approach?

- **Complete state capture:** Position + direction + mode flags
- **String key for hashing:** Easy map lookup, handles all data types
- **Loop prevention:** If we've seen this exact state before = infinite loop

3. Movement Validation

```go
func (r *Robot) canMoveTo(pos Position) bool {
    if !r.isValidPosition(pos) {
        return false // Out of bounds
    }

    cell := r.grid[pos.row][pos.col]

    // Can always move to empty spaces and special cells
    if cell == ' ' || cell == '@' || cell == '$' ||
       cell == 'S' || cell == 'E' || cell == 'N' || cell == 'W' ||
       cell == 'B' || cell == 'I' || (cell >= '1' && cell <= '9') {
        return true
    }

    if cell == '#' {
        return false // Unbreakable wall
    }

    if cell == 'X' && r.state.breaker {
        return true // Can break wall in breaker mode
    }

    if cell == 'X' {
        return false // Cannot break wall without breaker mode
    }

    return true
}
```

Key Logic:

- **Bounds checking first:** Prevents array out-of-bounds
- **Special cells always passable:** Robot can step on modifiers
- **Conditional wall breaking:** Only works in breaker mode

4. Direction Priority System

```go
```

```go
func (r *Robot) getPriorityDirections() []Direction {
  if r.state.inverted {
    return []Direction{WEST, NORTH, EAST, SOUTH} // Inverted
  }
  return []Direction{SOUTH, EAST, NORTH, WEST}   // Normal
}
```

Critical Rule: When robot hits obstacle, it tries directions in this priority order.

5. Main Pathfinding Algorithm

```go
go
```

```go
func (r *Robot) findPath() bool {
  for {  // Infinite loop until destination or loop detected

    // 1. Check win condition
    if r.state.pos == r.destPos {
      return true
    }

    // 2. Loop detection
    stateKey := r.getStateKey()
    if r.visited[stateKey] {
      return false // Loop detected
    }
    r.visited[stateKey] = true

    // 3. Process current cell effects
    cell := r.grid[r.state.pos.row][r.state.pos.col]
    switch cell {
    case 'S': r.state.direction = SOUTH
    case 'E': r.state.direction = EAST
    case 'N': r.state.direction = NORTH
    case 'W': r.state.direction = WEST
    case 'I': r.state.inverted = !r.state.inverted
    case 'B': r.state.breaker = !r.state.breaker
    case '1', '2', '3', '4', '5', '6', '7', '8', '9':
      // Teleport to matching number
      teleporters := r.teleporters[cell]
      for _, tp := range teleporters {
        if tp != r.state.pos {  // Don't teleport to same position
          r.state.pos = tp
          break
        }
      }
    }

    // 4. Try to move in current direction
    nextPos := r.getNextPosition(r.state.direction)
    if r.canMoveTo(nextPos) {
      // Handle wall destruction
      if r.grid[nextPos.row][nextPos.col] == 'X' && r.state.breaker {
        r.grid[nextPos.row][nextPos.col] = ' ' // Permanent destruction
      }
      r.state.pos = nextPos
      r.path = append(r.path, r.state.direction)
      continue  // Start next iteration
    }
```

```
        // 5. Cannot move forward, try priority directions
        priorities := r.getPriorityDirections()
        moved := false

        for _, dir := range priorities {
            nextPos := r.getNextPosition(dir)
            if r.canMoveTo(nextPos) {
                // Handle wall destruction
                if r.grid[nextPos.row][nextPos.col] == 'X' && r.state.breaker {
                    r.grid[nextPos.row][nextPos.col] = ' '
                }
                r.state.pos = nextPos
                r.state.direction = dir  // Change direction!
                r.path = append(r.path, dir)
                moved = true
                break
            }
        }

        // 6. Completely stuck
        if !moved {
            return false
        }
    }
}
```

Algorithm Analysis

Time Complexity

- **Best case:** O(H×W) - Direct path to destination
- **Worst case:** O(H×W×2²) - Visit every position in every possible state
    - H×W positions
    - 4 directions
    - 2 breaker states
    - 2 inverter states
    - Total states: H×W×4×2×2 = 16×H×W

Space Complexity

- **Grid storage:** O(H×W)
- **Visited states:** O(H×W×16) in worst case
- **Path storage:** O(H×W) maximum path length

- **Overall:** O(H×W)

Key Interview Topics

1. Why Use State-Based Loop Detection?

Position-only detection is insufficient:

```
Robot at (2,3) facing NORTH with breaker=true
vs
Robot at (2,3) facing SOUTH with breaker=false
```

These are different states and may lead to different outcomes.

2. Wall Destruction Logic

```go
if r.grid[nextPos.row][nextPos.col] == 'X' && r.state.breaker {
    r.grid[nextPos.row][nextPos.col] = ' ' // Permanent change
}
```

Why modify the grid? Destroyed walls stay destroyed. This affects future pathfinding.

3. Teleporter Implementation

```go
teleporters := r.teleporters[cell]
for _, tp := range teleporters {
    if tp != r.state.pos { // Critical check!
        r.state.pos = tp
        break
    }
}
```

Why the position check? Prevents teleporting to the same teleporter (infinite loop).

4. Direction Priority Logic

When hitting an obstacle, robot doesn't just turn around - it follows strict priority:

1. SOUTH (preferred)
2. EAST
3. NORTH
4. WEST (last resort)

This creates predictable, deterministic behavior.

5. Edge Cases Handled

- **Out of bounds movement**
- **Teleporter self-reference**
- **Permanent wall destruction**
- **State persistence through teleportation**
- **Multiple mode toggles (breaker, inverter)**

Common Interview Questions & Answers

Q: How do you detect infinite loops? A: By tracking complete robot state (position + direction + modes). If we ever revisit the same state, we're in a loop.

Q: Why not use BFS/DFS? A: The robot has deterministic movement rules. It doesn't choose paths - it follows fixed logic. This is simulation, not search.

Q: What if there are multiple teleporter pairs? A: Each digit (1-9) forms its own pair. The map stores all positions for each digit.

Q: How do you handle wall destruction permanently? A: Modify the original grid. Once 'X' becomes ' ', it stays that way for the entire simulation.

Q: What's the maximum possible path length? A: Theoretically H×W×16 (all positions in all states), but practically much shorter due to loop detection.

This solution demonstrates **state-space simulation**, **loop detection**, **grid manipulation**, and **deterministic pathfinding** - all valuable algorithmic concepts for technical interviews.