Implementing an intelligent version of the classical sliding-puzzle game for unix terminals using Golang's concurrency primitives

Pravendra Singh

Department of Computer Science and Engineering • Indian Institute of Technology, Roorkee, India

http://pravj.github.io • hackpravj@gmail.com

Abstract

A smarter version of the sliding-puzzle game is developed using the Go programming language. The game runs in computer system's terminals. Mainly, it was developed for UNIX-type systems but bacause of cross-platform compatibility of the programming language used, it works very well in nearly all the operating systems.

The game uses Go's concurrency primitives to simplify most of the hefty parts of the game. Real time notification functionality is also developed using language's built-in concurrency support.

Keywords: Artificial Intelligence, Concurrent Programming, Programming Languages, Golang, Game Development

---

1. Introduction

Sliding puzzles have their own reputation in the world of artificial intelligence and graph theory since a long. The oldest type of sliding puzzle game is known as fifteen puzzle. It is believed to be invented in 1874 by Noyes Palmer Chapman, a postmaster in New York state. The game consists of a 4x4 board with 16 tiles in it. Each tile have numbers drawn on it except one tile, which is either blank or sometimes have digit '0' on it. The task for the game is to re-order all the tiles in a particular manner, where you are only allowed to move the blank tile at a time.

This type of puzzles have many variants also. The 3x3 board puzzle being fairly popular among them, also known as the 8-puzzle game. In this type, the game board consists of 9 tiles. In this paper, we will use the 8-puzzle game board into consideration.

On the other hand, Go is a new programming language initially developed at Google, also known as Golang. It's a compiled and statically-typed language with built-in support for concurrent programming. Golang is gaining popularity as the language for system programming and developer operations.

2. Solvability

Here, the standard 8-puzzle game board is being considered, where the task of the game is to put the tiles in such a way that the last tile is blank and all other tiles have numbers in increasing order(from 1 to 8). A total of 362880(9!) board configurations are possible but only half of them are actually solvable according to our constraints.

So the game generates random configurations of the game board and uses an built-in package named 'scanner' to scan the board for its solvability. If the board is not really solvable then it generates a new one recursively.

For its implementation, package 'scanner' uses the discussion from the paper "Notes on the 15 puzzle" by Wm. Woolsey Johnson. A simplified version of it can be found on the "Analysis of Sixteen Puzzle" by Kevin Gong.

Package 'scanner' implements an algorithm to check for solvability of any board configuration. Its time complexity is $O(n^2)$. The algorithm returns a boolean value and an integer value, respectively implying whether a board is solvable or not and the index of blank tile in the board, if any.

```
func IsLegal(size int, values []int) (bool, int) {
    var inversions int
    n := len(values)

    for i := 0; i < (n - 1); i++ {
        if (values[i] != 0) && (values[i] != 1) {
            for j := i + 1; j < n; j++ {
                if (values[j] != 0) && (values[i] > values[j]) {
                    inversions++
                }
            }
        }
    }

    if (size%2 == 1) && (inversions%2 == 0) {
        return true, zeroIndex(values)
    }
    return false, -1
}
```

3. Puzzle Solution

The game comes with an built-in package named 'solver' that powers some features of it. For example, using the solver, the game shows optimal number of moves to solve any board configuration in real time. All the player moves are tracked and scored accordingly with the help of the solver.

The package 'solver' is implemented using Golang's native data structures and interfaces. It uses A-star algorithm for traversing game's state space.

3.1 Heuristic Function

The heuristic function used in the implementation is 'Misplaced Tiles'. Which is an admissible function, $h(n) \leq h^*(n)$. Then it will never overestimate the actual travelling distance and the solution will be always optimal.

```
func heuristicScore(b board.Board) int {
    var score int

    for i := 0; i < 3; i++ {
        for j := 0; j < 3; j++ {
            if b.Rows[i].Tiles[j].Value != ((3*i + j + 1) % 9) {
                score++
            }
        }
    }

    return score
}
```

3.2 Open List Data Structure

In the implementation, the game uses a custom data structure to accomplish the open-list required in the A-star algorithm execution. Open List maintains a collection of game-state nodes to be traversed at any given time.

```
type OpenList struct {
    nodeTable map[board.Board]Node
    table     map[board.Board]bool

    queue *PriorityQueue
}
```

3.2.1 nodeTable

Golang provides a built-in map type that implements a hash table.

A map is an unordered group of elements of one type, called the element type, indexed by a set of unique keys of another type, called the key type. The value of an uninitialized map is nil.

nodeTable maps a board configuration to a node in game's state space. No node appears twice when we move from start to goal state, so there is not any ambiguity. It helps us traversing the path once the search is complete.

3.2.2 table

table is also a map which maps board configurations a boolean value. It is used to check that whether a board configuration is present there in the open-list or not. By default it returns *false* whenever the board is absent.

3.2.3 queue

queue represents a *Priority Queue* data structure, precisely it is a *min-priority queue*. It is used to select a child node of node, having least travelling cost. Values of travelling cost are used as priorities for nodes. It is built using golang's 'container/heap' package.

3.3 Close List Data Structure

Similar to Open List, the solver also uses a Close List data structure. It is used to label nodes as *Already Traversed*.

```
type CloseList struct {
    table map[board.Board]bool
}
```

3.3.1 table

table here is similar to the one used in Open List. It maps a board configuration to a boolean value, indicating whether a board is present in closed list or not.

4. Path Traversing

The search algorithm terminates when the node to enter in the close list is the goal node. While traversing all the nodes in the game's state space, it keeps a track of all the nodes and their respective child nodes.

For this, it keeps a map from child-board configuration to parent-board configuration, generating a *many-to-one* mapping because a board configuration can have 2 to 4 new board configurations as child.

```
// mapping from child to parent board configurations
relation map[board.Board]board.Board
```

To collect the exact moves from start to the goal state, it forms a linked-list structure of board configurations, where the goal state is at the tail of the list and at the head it has the board to move next, from the current board configuration. It is built using the 'container/list' package of Golang.

```
// generating the linked-list representing path to move on
state := s.Goal

for s.relation[state] != start {
    state = s.relation[state]
    s.Path.PushFront(state)
}
s.Path.PushBack(s.Goal)
```

5. Scoring Function

The game has its own scoring function. The package 'score' helps implementing this. At any state space node(n), the score for the game can be calculated by the function, *score(n)*.

score(n) = ACS(n) / Total Moves

ACS(n) = *Accumulated Correct Score* for the node **n**

$$\text{where ACS(n)} = \begin{cases} 0; & \textit{if node n is start state} \\ \text{ACS(parent(n))} + 1; & \textit{if last move was correct} \\ \text{ACS(parent(n))} - 1; & \textit{if last move was wrong} \end{cases}$$

So in the game, the maximum possible score of 1 will be scored in only one case when all the moves from player were correct throughout the goal state. When the total number of moves is 0, the score will also be 0.

6. Game Interface

The graphic user interface for the game is built using Box-drawing characters, like it was used in early text-mode video hardware emulators, also known as Semigraphics.

It uses Golang port of the termbox library for writing text-based user interfaces. Coloring the interfaces is done using normal 8-colors, with foreground and background attributes for special formatting.

7. Real Time Notification

Golang provides built-in support to simplify concurrent programming with the help *goroutines* and *channels*.

A goroutine is a lightweight thread of execution. Channels are the pipes that connect concurrent goroutines. You can send values into channels from one goroutine and receive those values into another goroutine.

In this case, the game uses *goroutines* to provide an *asynchronous* or *non-blocking* experience. So that the graphic interface of game can be drawn and acted upon as soon as the game starts while the 'solver' stays busy solving the puzzle in the background at the same time.

Now, this can lead to a problem in slow systems. For example, when the game's interface is visible but the 'solver' is taking its time to solve the puzzle and the player gives any input event, the game is not likely to handle the request properly. Because the other sections of the game like 'score' package needs the puzzle to be solved first.

To solve this, the other *concurrency primitive, Channel* of the language is used. It works with the *goroutines* and implements the notification system for the game and helps manitaining the *asynchronous* characteristic of the game.

The game consists of packages 'notification' and 'surface' that implements this functionality. Package 'notification' has a channel named 'Tunnel', which let us flow *string* objects through it.

Golang has the keyword *go* to run any portion of code as a *goroutine* and *chan* is the keyword for *channels*.

```
type Notification struct {
    Tunnel chan string
}
```

The start of the game initiates a *goroutine* which listens to any object passing through the channel and update the notification in real time.

```
go func() {
    for e := range s.Notifier.Tunnel {
        s.solvableMoves = s.gameSolver.Path.Len()

        s.currentBoard = s.gameSolver.Path.Front()

        // updates the notification message
        s.Message = e
        s.drawBoard()
    }
}()
```

So whenever a player moves in wrong direction, the game starts a new *goroutine* to solve the new board configuration. If the user tries to move when the 'solver' has not yet solved the puzzle, it notifies the player to wait for a while. It invokes the 'Ready To Play' notification as soon as the 'solver' is done solving for the configuration to let the player know that he is ready to play. Here, you can see list of all the notifications used in the game.

```
// wrong move by player
s.gameSolver = solver.New(s.gameBoard)

go func() {
    s.gameSolver.Solve()
    s.solved = false
    s.solvableMoves = s.gameSolver.Path.Len()

    s.NotificationColor = termbox.ColorGreen

    // PASS THE NOTIFICATION INTO THE CHANNEL
    s.Notifier.Tunnel <- notification.ReadyToPlayMessage
}()
```

The notification channel *Tunnel* is closed when the game is complete or whenever player quits the game.

```
close(s.Notifier.Tunnel)
```

8. Acknowledgements

9. References