Graphs and Games: can Go take a ticket to ride?

Michele Caci
Senior Software Engineer at Amadeus

Games help you develop skills

About myself

I am a proud Gopher since 2018

- with experiences in other programming languages too

I work (in Go) in Amadeus

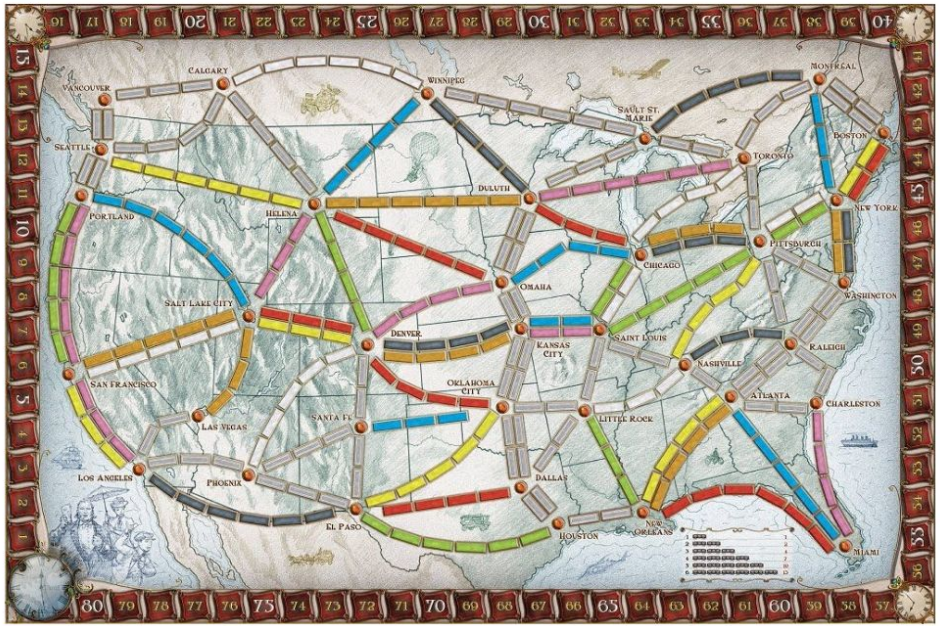
- a company that creates applications for the travel industry

I like sharing my Go knowledge at conferences

- But on my free time I enjoy swimming, cooking, learning languages and playing board games

Today's board game

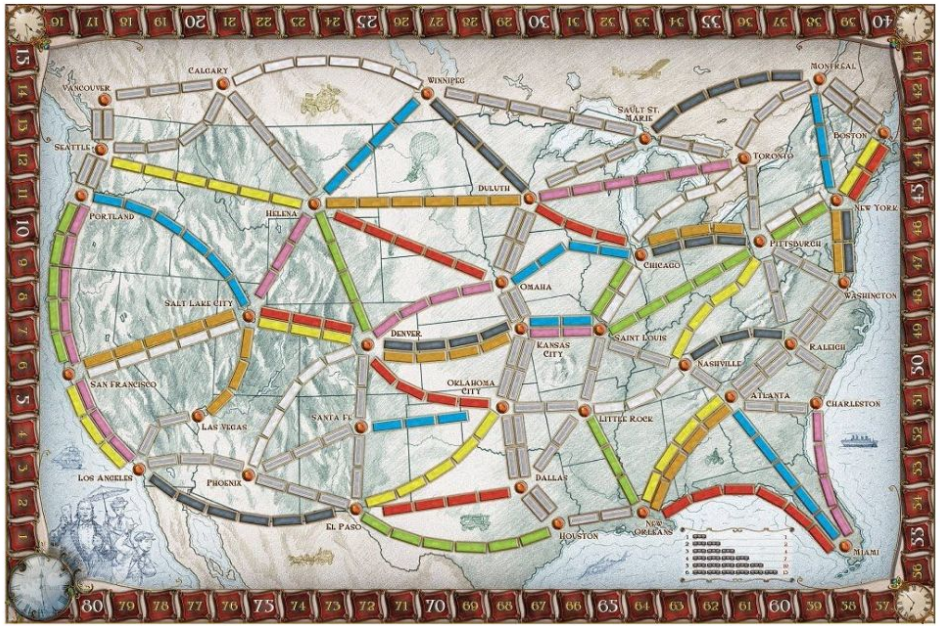
Ticket to Ride



Today's board game

Ticket to Ride

The board represents the United States map

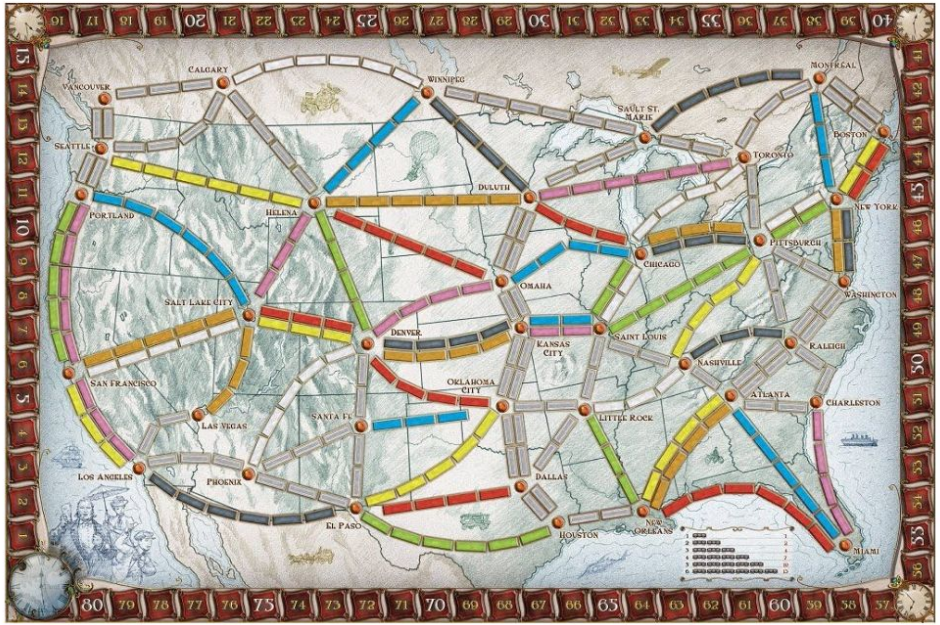


Today's board game

Ticket to Ride

The board represents the United States map

- The dots are cities/railway stations
- The lines are railway lines that connect them



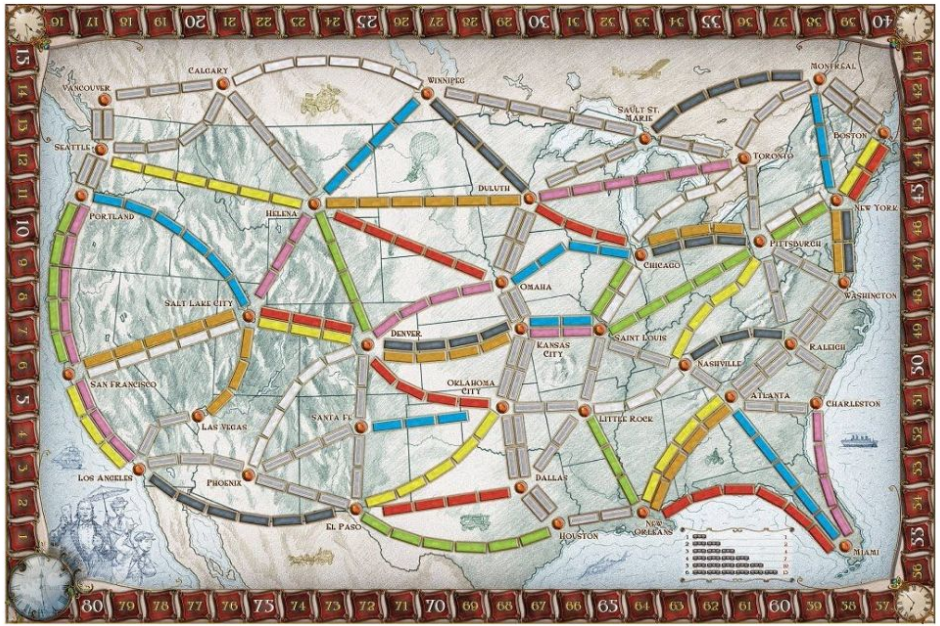
Today's board game

Ticket to Ride

The board represents the United States map

- The dots are cities/railway stations
- The lines are railway lines that connect them

The resources are train tokens and colored cards that are spent to occupy railway lines



Today's board game

Ticket to Ride

The board represents the United States map

- The dots are cities/railway stations
- The lines are railway lines that connect them

The resources are train tokens and colored cards that are spent to occupy railway lines

- For example to occupy the line between Miami and Atlanta you'll need to spend 5 trains tokens and 5 blue cards

Today's board game

Ticket to Ride

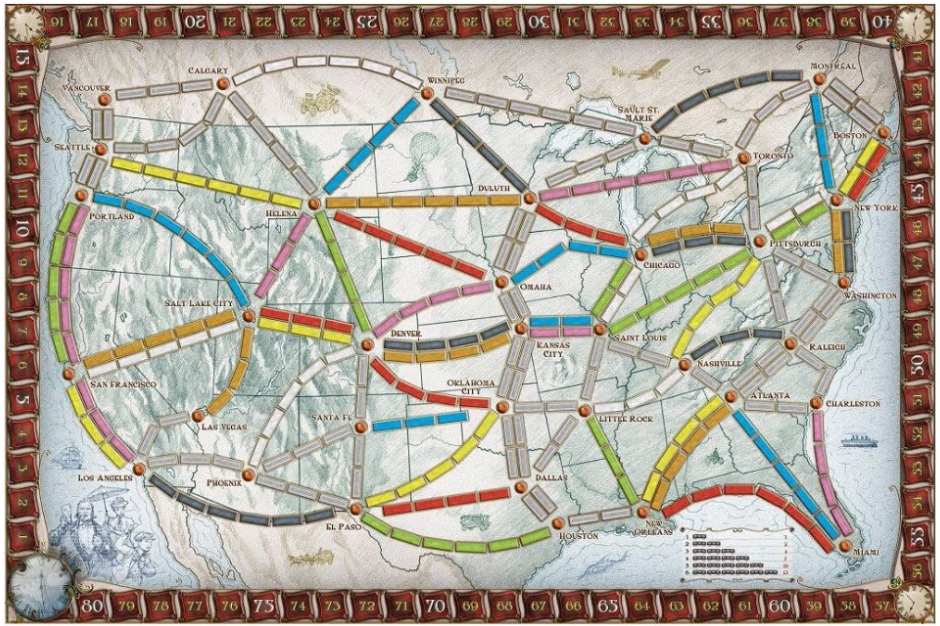
The board represents the United States map

- The dots are cities/railway stations
- The lines are railway lines that connect them

The resources are train tokens and colored cards that are spent to occupy railway lines

- For example to occupy the line between Miami and Atlanta you'll need to spend 5 trains tokens and 5 blue cards

The objective is to get the highest score



Today's board game

Ticket to Ride

The board represents the United States map

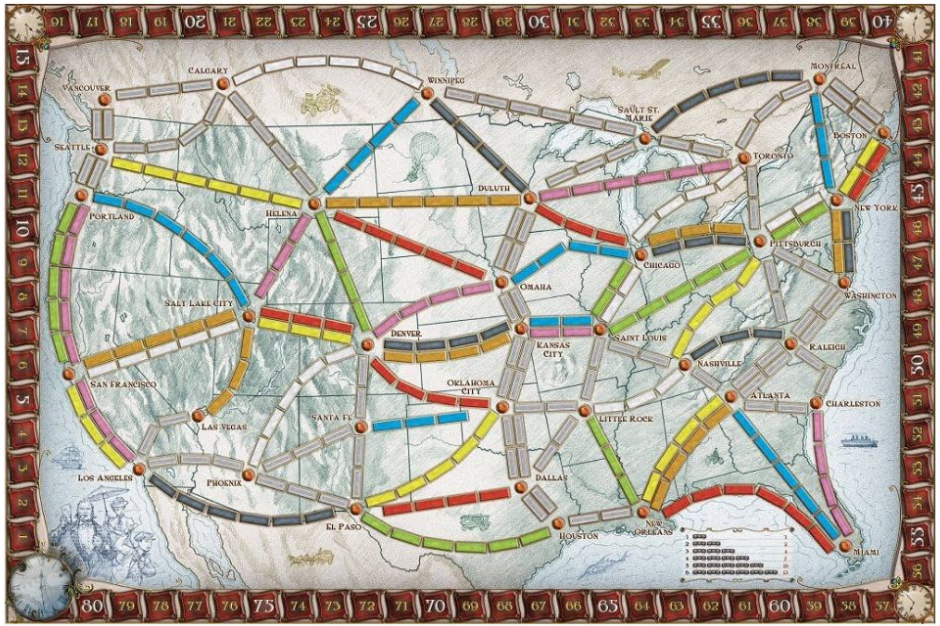
- The dots are cities/railway stations
- The lines are railway lines that connect them

The resources are train tokens and colored cards that are spent to occupy railway lines

- For example to occupy the line between Miami and Atlanta you'll need to spend 5 trains tokens and 5 blue cards

The objective is to get the highest score

- By occupying railway lines
- By connecting cities from objective cards



Let’s see how we can play the game in Go

Idea #1

We Go random and simplify a bit the rules

The number of player will be 2

The railway line chosen by each player will be **random**

Each player has unlimited resources

- which means that each player will take turns to select a line and occupy it

Each player has no objectives

- which means that the final score will be determined by which lines they occupy

Idea #1

Let's see the code

```
package main

func main() {
    // Collect all the railway lines
    railwaylines, err := data.RailwayLines()
    if err != nil { /* log and exit */ }

    // create the two players
    p1, p2 := player.NewRandom(1), player.NewRandom(2)
    // use a coin to select the player who takes the turn and play until all lines are occupied
    var coin bool
    for game.FreeRoutesAvailable(railwaylines) {
        playRound := p1.Play()
        if !coin {
            playRound = p2.Play()
        }
        playRound(railwaylines)
        // pass the turn
        coin = !coin
    }
    slog.Info("end game", "Score P1", player.Score(p1), "Score P2", player.Score(p2))
}
```

Idea #1

Let's see the code

```
1  package main
2
3  func main() {
4      // Collect all the railway lines
5      railwaylines, err := data.RailwayLines()
6      if err != nil { /* log and exit */ }
7
8      // create the two players
9      p1, p2 := player.NewRandom(1), player.NewRandom(2)
10     // use a coin to select the player who takes the turn and play until all lines are occupied
11     var coin bool
12     for game.FreeRoutesAvailable(railwaylines) {
13         playRound := p1.Play()
14         if !coin {
15             playRound = p2.Play()
16         }
17         playRound(railwaylines)
18         // pass the turn
19         coin = !coin
20     }
21     slog.Info("end game", "Score P1", player.Score(p1), "Score P2", player.Score(p2))
22 }
```

Idea #1

Let's see the code

```
package main

func main() {
    // Collect all the railway lines
    railwaylines, err := data.RailwayLines()
    if err != nil { /* log and exit */ }

    // create the two players
    p1, p2 := player.NewRandom(1), player.NewRandom(2)
    // use a coin to select the player who takes the turn and play until all lines are occupied
    var coin bool
    for game.FreeRoutesAvailable(railwaylines) {
        playRound := p1.Play()
        if !coin {
            playRound = p2.Play()
        }
        playRound(railwaylines)
        // pass the turn
        coin = !coin
    }
    slog.Info("end game", "Score P1", player.Score(p1), "Score P2", player.Score(p2))
}
```

Idea #1

Let's see the code

```go
package player

type Random struct {
    id    int
    owned []*game.TrainLine
}
func NewRandom(id int) *Random { return &Random{id: id} }
func (p *Random) Play() func(g game.Board) {
    return func(g game.Board) {
        // select and remove a random railway line from the board
        chosenLine := game.PopRandomLine(g)
        // add it to the owned list
        p.owned = append(p.owned, (*game.TrainLine)(chosenLine))
    }
}
// Score sums up the value of each owned railway line
func (p *Random) Score() int {
    var score int
    for i := range p.owned {
        score += p.owned[i].Value()
    }
    return score
}
```

Idea #1

Let's see the code

```go
package player

type Random struct {
    id    int
    owned []*game.TrainLine
}
func NewRandom(id int) *Random { return &Random{id: id} }
func (p *Random) Play() func(g game.Board) {
    return func(g game.Board) {
        // select and remove a random railway line from the board
        chosenLine := game.PopRandomLine(g)
        // add it to the owned list
        p.owned = append(p.owned, (*game.TrainLine)(chosenLine))
    }
}
// Score sums up the value of each owned railway line
func (p *Random) Score() int {
    var score int
    for i := range p.owned {
        score += p.owned[i].Value()
    }
    return score
}
```

Idea #1

Let's see the code

```
package player

type Random struct {
    id    int
    owned []*game.TrainLine
}
func NewRandom(id int) *Random { return &Random{id: id} }
func (p *Random) Play() func(g game.Board) {
    return func(g game.Board) {
        // select and remove a random railway line from the board
        chosenLine := game.PopRandomLine(g)
        // add it to the owned list
        p.owned = append(p.owned, (*game.TrainLine)(chosenLine))
    }
}
// Score sums up the value of each owned railway line
func (p *Random) Score() int {
    var score int
    for i := range p.owned {
        score += p.owned[i].Value()
    }
    return score
}
```

Idea #1

Let's see the code

```
package player

type Random struct {
    id    int
    owned []*game.TrainLine
}
func NewRandom(id int) *Random { return &Random{id: id} }
func (p *Random) Play() func(g game.Board) {
    return func(g game.Board) {
        // select and remove a random railway line from the board
        chosenLine := game.PopRandomLine(g)
        // add it to the owned list
        p.owned = append(p.owned, (*game.TrainLine)(chosenLine))
    }
}
// Score sums up the value of each owned railway line
func (p *Random) Score() int {
    var score int
    for i := range p.owned {
        score += p.owned[i].Value()
    }
    return score
}
```

Idea #1

Let's see the code

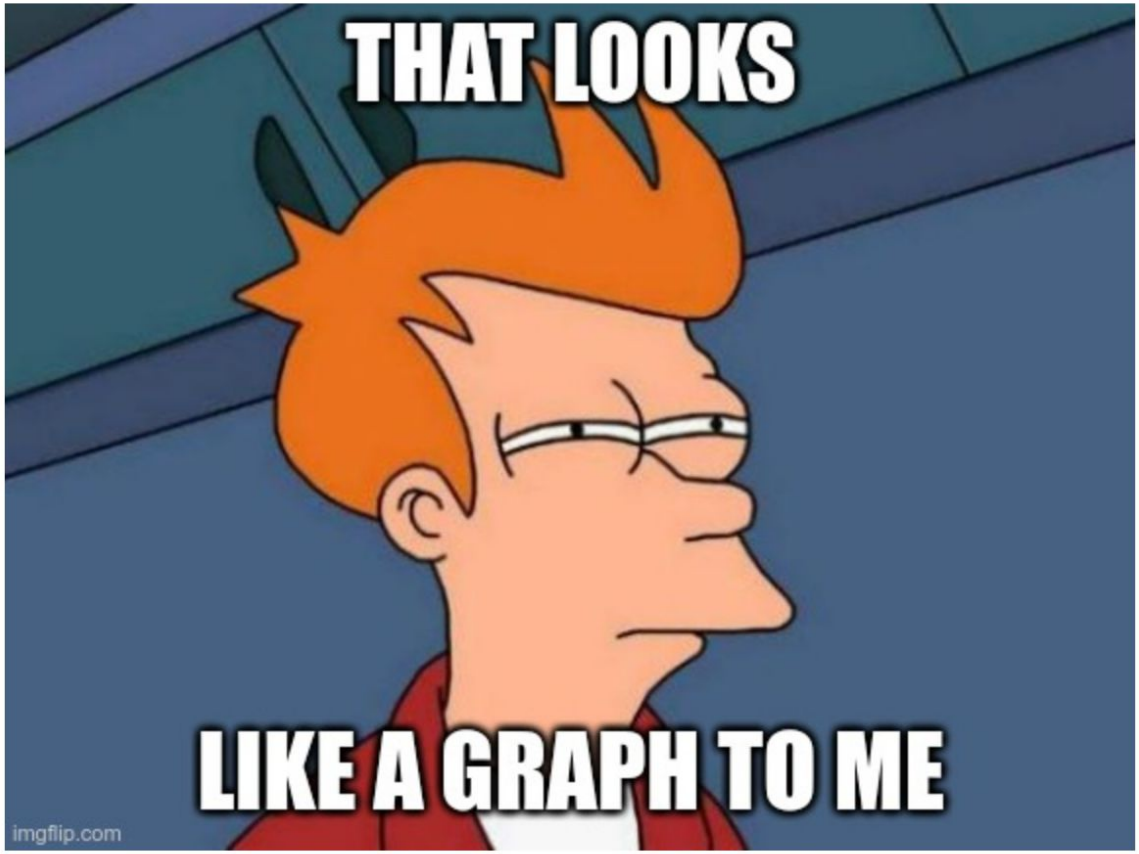
```go
package player

type Random struct {
    id    int
    owned []*game.TrainLine
}
func NewRandom(id int) *Random { return &Random{id: id} }
func (p *Random) Play() func(g game.Board) {
    return func(g game.Board) {
        // select and remove a random railway line from the board
        chosenLine := game.PopRandomLine(g)
        // add it to the owned list
        p.owned = append(p.owned, (*game.TrainLine)(chosenLine))
    }
}
// Score sums up the value of each owned railway line
func (p *Random) Score() int {
    var score int
    for i := range p.owned {
        score += p.owned[i].Value()
    }
    return score
}
```

Demo time!

Let’s focus on the board for one second


CALGARY
VANCOUVER
WINNIPEG
MONTREAL
SAULT ST. MARIE
SEATTLE
BOSTON
TORONTO
DULUTH
PORTLAND
HELENA
NEW YORK
PITTSBURGH
CHICAGO
OMAHA
WASHINGTON
SALT LAKE CITY
DENVER
SAINT LOUIS
KANSAS CITY
RALEIGH
NASHVILLE
SAN FRANCISCO
OKLAHOMA CITY
ATLANTA
CHARLESTON
SANTA FE
LITTLE ROCK
LAS VEGAS
LOS ANGELES
PHOENIX
DALLAS
EL PASO
NEW ORLEANS
HOUSTON
MIAMI


THAT LOOKS
LIKE A GRAPH TO ME
imgflip.com

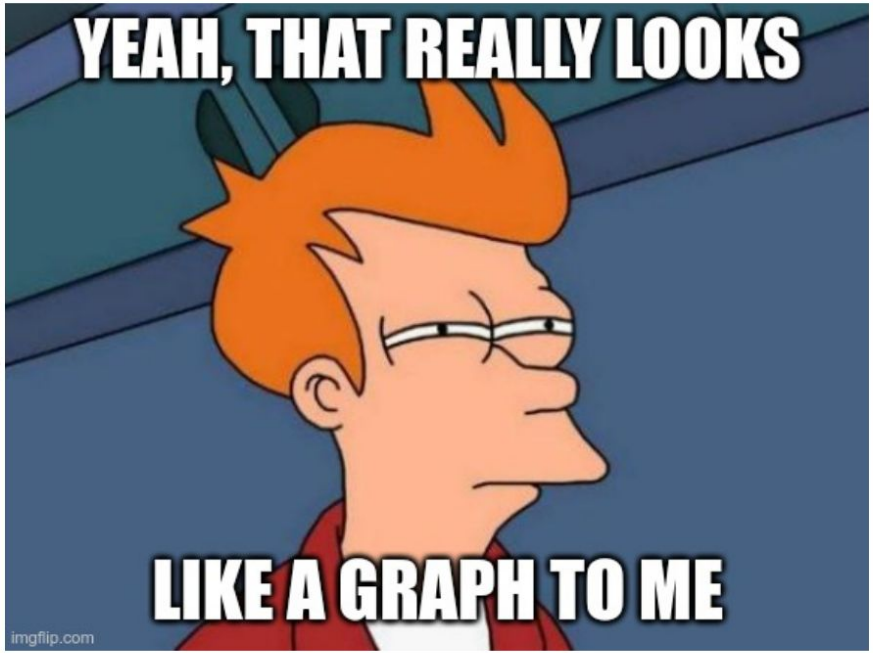

CALGARY
VANCOUVER
WINNIPEG
MONTREAL
SAULT ST. MARIE
SEATTLE
BOSTON
TORONTO
DULUTH
PORTLAND
HELENA
NEW YORK
PITTSBURGH
CHICAGO
OMAHA
WASHINGTON
SALT LAKE CITY
DENVER
SAINT LOUIS
KANSAS CITY
RALEIGH
NASHVILLE
SAN FRANCISCO
OKLAHOMA CITY
ATLANTA
CHARLESTON
SANTA FE
LITTLE ROCK
LAS VEGAS
LOS ANGELES
PHOENIX
DALLAS
EL PASO
NEW ORLEANS
HOUSTON
MIAMI

Idea #2

Let's model Ticket to Ride board as a graph

This is where we introduce graphs algorithms

- graphgo: my library to learn graph algorithms in Go
  - Use gonum instead of my library for appliaction that manages graphs
- go-ticket-to-ride: the implementation of the ticket to ride game in Go




GOOD NEWS EVERYONE
ALGORITHMS IN GO ARE VERY EASY
imgflip.com

Vertices, Edges and Graphs

Vertices and Edges

How we can implement them in Go and how they translate in Ticket to Ride

Vertices and Edges

How we can implement them in Go and how they translate in Ticket to Ride

```go
// A vertex is a node that is holding data, for simplicity we will have it comparable
type Vertex[T comparable] struct {
  E T
}
// An edge is a pair of vertices that can hold any property
type Edge[T comparable] struct {
    X, Y *Vertex[T]
    P    EdgeProperty
}
 type EdgeProperty any
```

Vertices and Edges

How we can implement them in Go and how they translate in Ticket to Ride

```go
// A vertex is a node that is holding data, for simplicity we will have it comparable
type Vertex[T comparable] struct {
  E T
}
// An edge is a pair of vertices that can hold any property
type Edge[T comparable] struct {
    X, Y *Vertex[T]
    P    EdgeProperty
}
 type EdgeProperty any
```

Vertices and Edges

How we can implement them in Go and how they translate in Ticket to Ride

```
1  // A Ticket to Ride example
2  // We create city stations as vertices of our Ticket to Ride graph
3  type City string
4  newYork := Vertex[City]{E: "New York"}
5  washington := Vertex[City]{E: "Washington"}
6
7  // We define a property for the Edge between city station vertices
8  type TrainLineProperty struct {
9    Distance int
10 }
11 // We create a train line as an edge between two city station vertices
12 newYorkWashington := Edge[City]{X: &newYork, Y: &washington, P: TrainLineProperty{Distance: 2}}
```

Vertices and Edges

How we can implement them in Go and how they translate in Ticket to Ride

```go
// A Ticket to Ride example
// We create city stations as vertices of our Ticket to Ride graph
type City string
newYork := Vertex[City]{E: "New York"}
washington := Vertex[City]{E: "Washington"}

// We define a property for the Edge between city station vertices
type TrainLineProperty struct {
  Distance int
}
// We create a train line as an edge between two city station vertices
newYorkWashington := Edge[City]{X: &newYork, Y: &washington, P: TrainLineProperty{Distance: 2}}
```

Vertices and Edges

How we can implement them in Go and how they translate in Ticket to Ride

```go
// A Ticket to Ride example
// We create city stations as vertices of our Ticket to Ride graph
type City string
newYork := Vertex[City]{E: "New York"}
washington := Vertex[City]{E: "Washington"}

// We define a property for the Edge between city station vertices
type TrainLineProperty struct {
  Distance int
}
// We create a train line as an edge between two city station vertices
newYorkWashington := Edge[City]{X: &newYork, Y: &washington, P: TrainLineProperty{Distance: 2}}
```

Graphs

How we can implement them in Go and how they translate in Ticket to Ride

```go
// ArcsList is graph representation of a collection of edges and vertices
type ArcsList[T comparable] struct {
    v        []*Vertex[T]
    e        []*Edge[T]
}
```

Graphs

How we can implement them in Go and how they translate in Ticket to Ride

```
// A Ticket to Ride example
newYork := Vertex[City]{E: "New York"}
washington := Vertex[City]{E: "Washington"}
newYorkWashington := Edge[City]{X: &newYork, Y: washington, P: TrainLineProperty{Distance: 2}}
// Keep adding cities (vertices) and railway lines (edges)
// And add all them to the board
board := ArcsList[City]{
  v: []*Vertex[City]{ &newYork ,&washington /*, ...*/ }
  e: []*Edge[City]{ &newYorkWashington /*, ...*/ }
 }
```

Graphs

How we can implement them in Go and how they translate in Ticket to Ride

```
// A Ticket to Ride example
newYork := Vertex[City]{E: "New York"}
washington := Vertex[City]{E: "Washington"}
newYorkWashington := Edge[City]{X: &newYork, Y: washington, P: TrainLineProperty{Distance: 2}}
// Keep adding cities (vertices) and railway lines (edges)
// And add all them to the board
board := ArcsList[City]{
  v: []*Vertex[City]{ &newYork ,&washington /*, ...*/ }
  e: []*Edge[City]{ &newYorkWashington /*, ...*/ }
 }
```

Graphs

How we can implement them in Go and how they translate in Ticket to Ride

```go
// ArcsList is graph representation of a collection of edges and vertices
type ArcsList[T comparable] struct {
    v        []*Vertex[T]
    e        []*Edge[T]
}
```

Graphs

How we can implement them in Go and how they translate in Ticket to Ride

```go
// ArcsList is graph representation of a collection of edges and vertices
type ArcsList[T comparable] struct {
    v       []*Vertex[T]
    e       []*Edge[T]
}
```

There are other graph representations and the choice of the representation is based on memory and time efficiency with respect to the operations done

All graph representations share a common behavior that can be captured by creating an interface

```go
type Graph[T comparable] interface {
  Vertices() []*Vertex[T]
  Edges() []*Edge[T]
  AddVertex(v *Vertex[T])
  RemoveVertex(v *Vertex[T])
  AddEdge(e *Edge[T])
  RemoveEdge(e *Edge[T])
  // ...
}
```

What algorithms can we use for Ticket to Ride?

Is there a path connecting a city to another one?

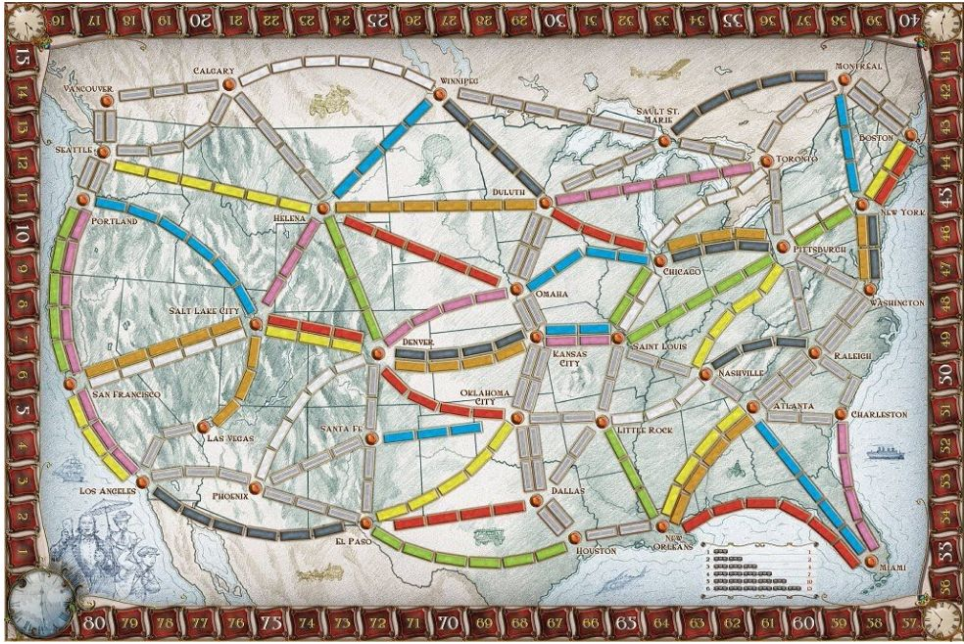
Connected vertices in a graph

The game starts with all of the cities connected by the edges representing the railway lines

As soon as players occupy railway lines, the correspondent edge is removed from the graph

To check if two cities are still connected by railway lines we'll use

- **visit** of a graph
- **connectivity** of two vertices in a graph



Is there a path connecting a city to another one?

Let’s see the code

Is there a path connecting a city to another one?

Let's see the code

```
// GenericVisit walks the graph from a source node, visiting each node it can visit only once
func GenericVisit[T comparable](g Graph[T], s *Vertex[T]) *Tree[T] {
    if !g.ContainsVertex(s) { return nil }
    s.Visit()
    t := &Tree[T]{element: &s.E}
    queue := []*Vertex[T]{s}
    for len(queue) != 0 {
        var next *Vertex[T]
        next, queue = queue[0], queue[1:]
        for _, v := range g.AdjacentNodes(next) {
            if v.Visited() {
                continue
            }
            v.Visit()
            queue = append(queue, v)
            parentNode := t.Find(&next.E)
            if subtree != nil {
                parentNode.children = append(parentNode.children, &Tree[T]{element: &v.E})
            }
        }
    }
    return t
}
```

Is there a path connecting a city to another one?

Let's see the code

```
// GenericVisit walks the graph from a source node, visiting each node it can visit only once
func GenericVisit[T comparable](g Graph[T], s *Vertex[T]) *Tree[T] {
    if !g.ContainsVertex(s) { return nil }
    s.Visit()
    t := &Tree[T]{element: &s.E}
    queue := []*Vertex[T]{s}
    for len(queue) != 0 {
        var next *Vertex[T]
        next, queue = queue[0], queue[1:]
        for _, v := range g.AdjacentNodes(next) {
            if v.Visited() {
                continue
            }
            v.Visit()
            queue = append(queue, v)
            parentNode := t.Find(&next.E)
            if subtree != nil {
                parentNode.children = append(parentNode.children, &Tree[T]{element: &v.E})
            }
        }
    }
    return t
}
```

Is there a path connecting a city to another one?

Let's see the code

```
// GenericVisit walks the graph from a source node, visiting each node it can visit only once
func GenericVisit[T comparable](g Graph[T], s *Vertex[T]) *Tree[T] {
    if !g.ContainsVertex(s) { return nil }
    s.Visit()
    t := &Tree[T]{element: &s.E}
    queue := []*Vertex[T]{s}
    for len(queue) != 0 {
        var next *Vertex[T]
        next, queue = queue[0], queue[1:]
        for _, v := range g.AdjacentNodes(next) {
            if v.Visited() {
                continue
            }
            v.Visit()
            queue = append(queue, v)
            parentNode := t.Find(&next.E)
            if subtree != nil {
                parentNode.children = append(parentNode.children, &Tree[T]{element: &v.E})
            }
        }
    }
    return t
}
```

Is there a path connecting a city to another one?

Let's see the code

```
// GenericVisit walks the graph from a source node, visiting each node it can visit only once
func GenericVisit[T comparable](g Graph[T], s *Vertex[T]) *Tree[T] {
    if !g.ContainsVertex(s) { return nil }
    s.Visit()
    t := &Tree[T]{element: &s.E}
    queue := []*Vertex[T]{s}
    for len(queue) != 0 {
        var next *Vertex[T]
        next, queue = queue[0], queue[1:]
        for _, v := range g.AdjacentNodes(next) {
            if v.Visited() {
                continue
            }
            v.Visit()
            queue = append(queue, v)
            parentNode := t.Find(&next.E)
            if subtree != nil {
                parentNode.children = append(parentNode.children, &Tree[T]{element: &v.E})
            }
        }
    }
    return t
}
```

Is there a path connecting a city to another one?

Let's see the code

```
// GenericVisit walks the graph from a source node, visiting each node it can visit only once
func GenericVisit[T comparable](g Graph[T], s *Vertex[T]) *Tree[T] {
    if !g.ContainsVertex(s) { return nil }
    s.Visit()
    t := &Tree[T]{element: &s.E}
    queue := []*Vertex[T]{s}
    for len(queue) != 0 {
        var next *Vertex[T]
        next, queue = queue[0], queue[1:]
        for _, v := range g.AdjacentNodes(next) {
            if v.Visited() {
                continue
            }
            v.Visit()
            queue = append(queue, v)
            parentNode := t.Find(&next.E)
            if subtree != nil {
                parentNode.children = append(parentNode.children, &Tree[T]{element: &v.E})
            }
        }
    }
    return t
}
```

Is there a path connecting a city to another one?

Let's see the code

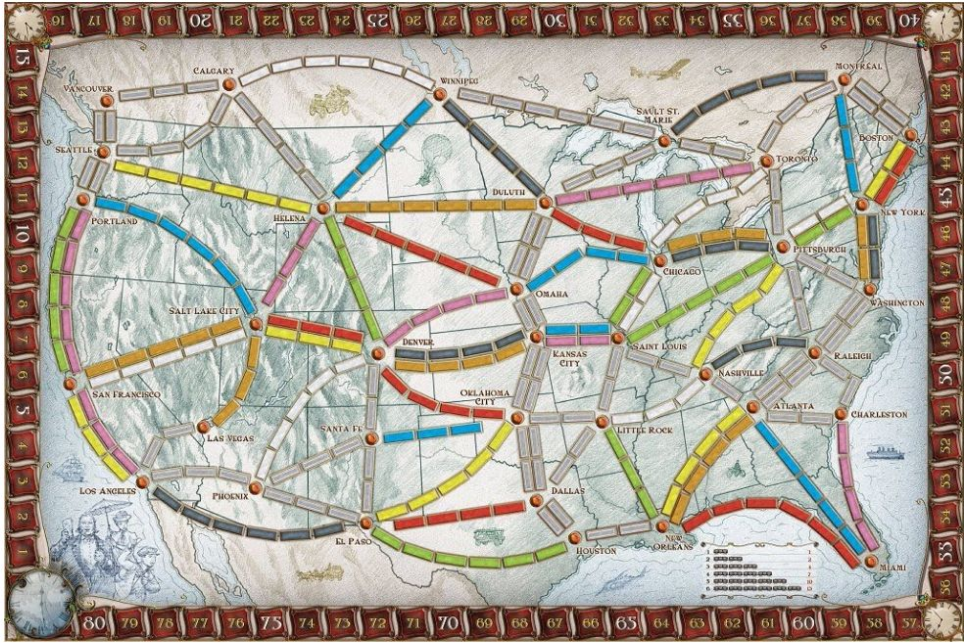
```
// Connected verifies that the vertices x and y are connected in the graph g
// by visiting g using x as a source and checking that the output tree contains the vertex v
func Connected[T comparable](g Graph[T], x, y *Vertex[T]) bool {
    return GenericVisit(g, x).Find(&y.E) != nil
}
```

Of all the routes connecting two cities, which one is the shortest?

Shortest path algorithm

If two cities are connected, there is at least one route between them

To check the shortest route between two cities we'll use the **Bellman-Ford algorithm**



Bellman-Ford algorithm for the shortest path

Let's see the code

```
func BellmanFordDistances[T comparable](g Graph[T], s *Vertex[T]) map[*Vertex[T]]*Distance[T] {
    d := make(map[*graph.Vertex[T]]*Distance[T]) // type Distance[T comparable] struct { v, u *Vertex[T]; d int }
    for _, v := range g.Vertices() {
        d[v] = &Distance[T]{v: s, u: v}
        if v != s {
            d[v].d = math.MaxInt
        }
    }
    canRelax := (x, y *graph.Vertex[T], w Weighter) bool { return d[x].d+w.Weight() < d[y].d && d[x].d+w.Weight() > 0 }
    relax    := (x, y *graph.Vertex[T], w Weighter)      { d[y].setDistance(w.Weight() + d[x].d)}
    for range g.Vertices() {
        for _, e := range g.Edges() {
            w := e.P.(Weighter) // type Weighter interface{ Weight() int }
            switch {
            case canRelax(e.X, e.Y, w):
                relax(e.X, e.Y, w)
            case canRelax(e.Y, e.X, w):
                relax(e.Y, e.X, w)
            }
        }
    }
    return d
}
```

Bellman-Ford algorithm for the shortest path

Let's see the code

```
func BellmanFordDistances[T comparable](g Graph[T], s *Vertex[T]) map[*Vertex[T]]*Distance[T] {
    d := make(map[*graph.Vertex[T]]*Distance[T]) // type Distance[T comparable] struct { v, u *Vertex[T]; d int }
    for _, v := range g.Vertices() {
        d[v] = &Distance[T]{v: s, u: v}
        if v != s {
            d[v].d = math.MaxInt
        }
    }
    canRelax := (x, y *graph.Vertex[T], w Weighter) bool { return d[x].d+w.Weight() < d[y].d && d[x].d+w.Weight() > 0 }
    relax    := (x, y *graph.Vertex[T], w Weighter)      { d[y].setDistance(w.Weight() + d[x].d)}
    for range g.Vertices() {
        for _, e := range g.Edges() {
            w := e.P.(Weighter) // type Weighter interface{ Weight() int }
            switch {
            case canRelax(e.X, e.Y, w):
                relax(e.X, e.Y, w)
            case canRelax(e.Y, e.X, w):
                relax(e.Y, e.X, w)
            }
        }
    }
    return d
}
```

Bellman-Ford algorithm for the shortest path

Let's see the code

```
1  func BellmanFordDistances[T comparable](g Graph[T], s *Vertex[T]) map[*Vertex[T]]*Distance[T] {
2      d := make(map[*graph.Vertex[T]]*Distance[T]) // type Distance[T comparable] struct { v, u *Vertex[T]; d int }
3      for _, v := range g.Vertices() {
4          d[v] = &Distance[T]{v: s, u: v}
5          if v != s {
6              d[v].d = math.MaxInt
7          }
8      }
9      canRelax := (x, y *graph.Vertex[T], w Weighter) bool { return d[x].d+w.Weight() < d[y].d && d[x].d+w.Weight() > 0 }
10     relax    := (x, y *graph.Vertex[T], w Weighter)      { d[y].setDistance(w.Weight() + d[x].d)}
11     for range g.Vertices() {
12         for _, e := range g.Edges() {
13             w := e.P.(Weighter) // type Weighter interface{ Weight() int }
14             switch {
15             case canRelax(e.X, e.Y, w):
16                 relax(e.X, e.Y, w)
17             case canRelax(e.Y, e.X, w):
18                 relax(e.Y, e.X, w)
19             }
20         }
21     }
22     return d
23 }
```

Bellman-Ford algorithm for the shortest path

Let's see the code

```
func BellmanFordDistances[T comparable](g Graph[T], s *Vertex[T]) map[*Vertex[T]]*Distance[T] {
    d := make(map[*graph.Vertex[T]]*Distance[T]) // type Distance[T comparable] struct { v, u *Vertex[T]; d int }
    for _, v := range g.Vertices() {
        d[v] = &Distance[T]{v: s, u: v}
        if v != s {
            d[v].d = math.MaxInt
        }
    }
    canRelax := (x, y *graph.Vertex[T], w Weighter) bool { return d[x].d+w.Weight() < d[y].d && d[x].d+w.Weight() > 0 }
    relax    := (x, y *graph.Vertex[T], w Weighter)      { d[y].setDistance(w.Weight() + d[x].d)}
    for range g.Vertices() {
        for _, e := range g.Edges() {
            w := e.P.(Weighter) // type Weighter interface{ Weight() int }
            switch {
            case canRelax(e.X, e.Y, w):
                relax(e.X, e.Y, w)
            case canRelax(e.Y, e.X, w):
                relax(e.Y, e.X, w)
            }
        }
    }
    return d
}
```

Bellman-Ford algorithm for the shortest path

Let's see the code

```
func Shortest[T comparable](g graph.Graph[T], d map[*graph.Vertex[T]]*Distance[T], x, y *graph.Vertex[T]) []*graph.Vertex[T] {
    if len(g.Vertices()) < 2 { return nil }
    isShortestDist := func(u, v *graph.Vertex[T], w Weighter) bool { return d[u].d+w.Weight() == d[v].d }
    isConnectingEdge := func(u, v *graph.Vertex[T], e *graph.Edge[T]) bool { return (e.X == u && e.Y == v) || (e.X == v && e.Y
    path := []*graph.Vertex[T]{y}
    v := y
    for v != x {
    neighbourSearch:
        for _, u := range g.AdjacentNodes(v) {
            for _, edge := range g.Edges() {
                if !isConnectingEdge(u, v, edge) {
                    continue
                }
                if !isShortestDist(u, v, edge.P.(Weighter)) {
                    continue
                }
                path = append([]*graph.Vertex[T]{u}, path...)
                v = u
                break neighbourSearch
            }
        }
    }
    return path
```

Bellman-Ford algorithm for the shortest path

Let’s see the code

```
func Shortest[T comparable](g graph.Graph[T], d map[*graph.Vertex[T]]*Distance[T], x, y *graph.Vertex[T]) []*graph.Vertex[T] {
    if len(g.Vertices()) < 2 { return nil }
    isShortestDist := func(u, v *graph.Vertex[T], w Weighter) bool { return d[u].d+w.Weight() == d[v].d }
    isConnectingEdge := func(u, v *graph.Vertex[T], e *graph.Edge[T]) bool { return (e.X == u && e.Y == v) || (e.X == v && e.Y
    path := []*graph.Vertex[T]{y}
    v := y
    for v != x {
    neighbourSearch:
        for _, u := range g.AdjacentNodes(v) {
            for _, edge := range g.Edges() {
                if !isConnectingEdge(u, v, edge) {
                    continue
                }
                if !isShortestDist(u, v, edge.P.(Weighter)) {
                    continue
                }
                path = append([]*graph.Vertex[T]{u}, path...)
                v = u
                break neighbourSearch
            }
        }
    }
    return path
```

Bellman-Ford algorithm for the shortest path

Let's see the code

```
func Shortest[T comparable](g graph.Graph[T], d map[*graph.Vertex[T]]*Distance[T], x, y *graph.Vertex[T]) []*graph.Vertex[T] {
    if len(g.Vertices()) < 2 { return nil }
    isShortestDist := func(u, v *graph.Vertex[T], w Weighter) bool { return d[u].d+w.Weight() == d[v].d }
    isConnectingEdge := func(u, v *graph.Vertex[T], e *graph.Edge[T]) bool { return (e.X == u && e.Y == v) || (e.X == v && e.Y
    path := []*graph.Vertex[T]{y}
    v := y
    for v != x {
    neighbourSearch:
        for _, u := range g.AdjacentNodes(v) {
            for _, edge := range g.Edges() {
                if !isConnectingEdge(u, v, edge) {
                    continue
                }
                if !isShortestDist(u, v, edge.P.(Weighter)) {
                    continue
                }
                path = append([]*graph.Vertex[T]{u}, path...)
                v = u
                break neighbourSearch
            }
        }
    }
    return path
```

Bellman-Ford algorithm for the shortest path

Let's see the code

```
func Shortest[T comparable](g graph.Graph[T], d map[*graph.Vertex[T]]*Distance[T], x, y *graph.Vertex[T]) []*graph.Vertex[T] {
    if len(g.Vertices()) < 2 { return nil }
    isShortestDist := func(u, v *graph.Vertex[T], w Weighter) bool { return d[u].d+w.Weight() == d[v].d }
    isConnectingEdge := func(u, v *graph.Vertex[T], e *graph.Edge[T]) bool { return (e.X == u && e.Y == v) || (e.X == v && e.Y
    path := []*graph.Vertex[T]{y}
    v := y
    for v != x {
    neighbourSearch:
        for _, u := range g.AdjacentNodes(v) {
            for _, edge := range g.Edges() {
                if !isConnectingEdge(u, v, edge) {
                    continue
                }
                if !isShortestDist(u, v, edge.P.(Weighter)) {
                    continue
                }
                path = append([]*graph.Vertex[T]{u}, path...)
                v = u
                break neighbourSearch
            }
        }
    }
    return path
```

Go’s simplicity vs the algorithms’ complexity

Back to Idea #2

Updated rules

- each player now has **3** objectives
  - which means the railway line chosen by each player will be made by **looking at the shortest path** available for the routes on their objective list

Back to Idea #2

Let's see the code

```
1  package main
2
3  func main() {
4      // Collect all the railway lines
5      railwaylines, err := data.RailwayLines()
6      if err != nil { /* log and exit */ }
7      // Collect all the tickets/objectives
8      tickets, err := data.Tickets()
9      if err != nil { /* log and exit */ }
10
11     // create the two players
12     p1, p2 := player.NewWithTickets(1, game.GetTickets(3, &tickets)),player.NewWithTickets(2, game.GetTickets(3, &tickets))
13     // use a coin to select the player who takes the turn and play until all lines are occupied
14     var coin bool
15     for game.FreeRoutesAvailable(railwaylines) {
16         play := p1.Play()
17         if !coin {
18             play = p2.Play()
19         }
20         play(railwaylines)
21         // pass the turn
22         coin = !coin
23     }
```

Back to Idea #2

Let's see the code

```
package main

func main() {
    // Collect all the railway lines
    railwaylines, err := data.RailwayLines()
    if err != nil { /* log and exit */ }
    // Collect all the tickets/objectives
    tickets, err := data.Tickets()
    if err != nil { /* log and exit */ }

    // create the two players
    p1, p2 := player.NewWithTickets(1, game.GetTickets(3, &tickets)),player.NewWithTickets(2, game.GetTickets(3, &tickets))
    // use a coin to select the player who takes the turn and play until all lines are occupied
    var coin bool
    for game.FreeRoutesAvailable(railwaylines) {
        play := p1.Play()
        if !coin {
            play = p2.Play()
        }
        play(railwaylines)
        // pass the turn
        coin = !coin
    }
```

Back to Idea #2

Let's see the code

```go
package player

type WithTickets struct {
    id         int
    ownedLines game.Board
    tickets    []game.Ticket
}
func NewWithTickets(id int, t []game.Ticket) *WithTickets {
    return &WithTickets{id: id, tickets: t, ownedLines: graph.NewUndirected[game.City](graph.ArcsListType)}
}
func (p *Random) Play() func(g game.Board) {
    randomSelection := func(b game.Board) {
        // same as the random player but storing ownedLines in the graph
    }
    shortestPath := func(b game.Board) { //...
    }

    if !p.HasTicketsToComplete() {
        return randomSelection
    }
    return shortestPath
}
```

Back to Idea #2

Let's see the code

```
package player

type WithTickets struct {
    id         int
    ownedLines game.Board
    tickets    []game.Ticket
}
func NewWithTickets(id int, t []game.Ticket) *WithTickets {
    return &WithTickets{id: id, tickets: t, ownedLines: graph.NewUndirected[game.City](graph.ArcsListType)}
}
func (p *Random) Play() func(g game.Board) {
    randomSelection := func(b game.Board) {
        // same as the random player but storing ownedLines in the graph
    }
    shortestPath := func(b game.Board) { //...
    }

    if !p.HasTicketsToComplete() {
        return randomSelection
    }
    return shortestPath
}
```

Back to Idea #2

Let’s see the code

```
1  package player
2
3  type WithTickets struct {
4      id         int
5      ownedLines game.Board
6      tickets    []game.Ticket
7  }
8  func NewWithTickets(id int, t []game.Ticket) *WithTickets {
9      return &WithTickets{id: id, tickets: t, ownedLines: graph.NewUndirected[game.City](graph.ArcsListType)}
10 }
11 func (p *Random) Play() func(g game.Board) {
12     randomSelection := func(b game.Board) {
13         // same as the random player but storing ownedLines in the graph
14     }
15     shortestPath := func(b game.Board) { //...
16     }
17
18     if !p.HasTicketsToComplete() {
19         return randomSelection
20     }
21     return shortestPath
22 }
```

Back to Idea #2

Let's see the code

```go
package player

type WithTickets struct {
    id         int
    ownedLines game.Board
    tickets    []game.Ticket
}
func NewWithTickets(id int, t []game.Ticket) *WithTickets {
    return &WithTickets{id: id, tickets: t, ownedLines: graph.NewUndirected[game.City](graph.ArcsListType)}
}
func (p *Random) Play() func(g game.Board) {
    randomSelection := func(b game.Board) {
        // same as the random player but storing ownedLines in the graph
    }
    shortestPath := func(b game.Board) { //...
    }

    if !p.HasTicketsToComplete() {
        return randomSelection
    }
    return shortestPath
}
```

Back to Idea #2

Let's see the code

```
shortestPath := func(b game.Board) {
    localBoard := graph.Copy(b)
updatedBoard:
    for len(localBoard.Edges()) > 0 {
        // Part 1: keep the door open to random selection if there are no available tickets
        ticket, err := p.NextAvailableTicket()
        if err != nil {  return randomSelection(localBoard) }

        //  Part 2: if there is no path between the two cities, the ticket is done and you move to the next one
        if !visit.ExistsPath(localBoard, ticket.X, ticket.Y) { ticket.Done = true; ticket.Ok = false; continue }

        // Part 3: if there is a path between the two cities in the objective select the shortest path and take the first segme
        shortest := path.Shortest(localBoard, path.BellmanFordDist(localBoard, ticket.X), ticket.X, ticket.Y)
        for i := 0; i < len(shortest)-1; i++ {
            chosenLine := game.FindLineFunc(game.ShortestSegment(ticket, shortest[i], shortest[i+1]), localBoard)
            // ...
        }
    }
    return
}
```

Back to Idea #2

Let's see the code

```
shortestPath := func(b game.Board) {
    // ...
    // Part 3: if there is a path between the two cities in the objective select the shortest path and take the first segment a
    shortest := path.Shortest(localBoard, path.BellmanFordDist(localBoard, ticket.X), ticket.X, ticket.Y)
    for i := 0; i < len(shortest)-1; i++ {
        chosenLine := game.FindLineFunc(game.ShortestSegment(ticket, shortest[i], shortest[i+1]), localBoard)
        // Is the line owned by me?
        owned := p.ownedLines.ContainsEdge(chosenLineEdge)
        if owned { continue }
        // Is the line owned by someone else?
        occupiedNotOwned := chosenLine.P.(*game.TrainLineProperty).Occupied
        if occupiedNotOwned {
            localBoard.RemoveEdge(chosenLineEdge); continue updatedBoard;
        }
        // Occupy the selected line
        chosenLine.P.(*game.TrainLineProperty).Occupy()
        p.ownedLines.AddVertex(chosenLine.X)
        p.ownedLines.AddVertex(chosenLine.Y)
        p.ownedLines.AddEdge(chosenLineEdge)
        // Check if ticket is completed after taking the line
        if visit.ExistsPath(p.ownedLines, tX, tY) {
            ticket.Done, ticket.Ok = true, true
        }
```

Demo time!

Conclusions

Can Go take the Ticket to Ride? Yes!

Conclusions

Can Go take the Ticket to Ride? Yes!

Games are a good opportunity to practise and learn new skills

Conclusions

Can Go take the Ticket to Ride? Yes!

Games are a good opportunity to practise and learn new skills

- About Go and beyond

Conclusions

Can Go take the Ticket to Ride? Yes!

Games are a good opportunity to practise and learn new skills

- About Go and beyond

Go makes it easy to translate pseudo-code in actual code and to implement algorithms

Conclusions

Can Go take the Ticket to Ride? Yes!

Games are a good opportunity to practise and learn new skills

- About Go and beyond

Go makes it easy to translate pseudo-code in actual code and to implement algorithms

- No matter how complex the algorithm is

Conclusions

Can Go take the Ticket to Ride? Yes!

Games are a good opportunity to practise and learn new skills

- About Go and beyond

Go makes it easy to translate pseudo-code in actual code and to implement algorithms

- No matter how complex the algorithm is

Take advantage of the simplicity that Go brings you

And you’ll be able to create awesome things with Go!

Thank you very much!

Michele Caci