École Polytechnique de Louvain-La-neuve

Project
Captain SonOZ

Authors
Maxime Leurquin - 57621700
Jonathan de Salle - 13861500

Professor
Pr. Peter Van Roy

Group 4
LINGI 1131: Computer Language Concepts
May 14, 2020

1 Introduction

Our objective was to write an implementation of the game "Captain Sonar" in the OZ programming language, according to the specifications given to us by Pr Van Roy[1].
This implementation must allow both simultaneous and turn-by-turn games and use the Message Passing programming paradigm.

2 Main File Design

2.1 Global Description

Our main file's functions and procedures can roughly be divised in 4 categories:

- Initialisation
- State
- Broadcast
- Game function

When the program is launched, we first call the initialisation functions which will initialize the players and the GUI and also create a first StateList[2].

After everything is set either call the function that launches the turn by turn or the one that launches the simultaneous game, depending on the parameters of the input file.

2.2 Initialization

This is composed of two main functions, $CreatePortSubmarine$ and $InitPosition$.

$CreatePortSubmarine$ will create a port for each player and create a first *StateList*, which is a list of states, used by the main function to keep track of each player. A state is defined by the following EBNF grammar:

- <state>::= nil| state(id:<id> port:port> surface:<surface>)
- <surface>::= surface(surface:<Boolean> timeLeft:<Integer>)

$InitPosition$ initializes the position of each player and the GUI.

2.3 State

As the name of their category imply, those functions are used to manipulate states and *StateLists* and return updated *StateLists*. It include the following functions:

- $SetState$
- $UpdateSurf$
- *Alive*
- $ProcessStream$
- $GetFinalState$

[1]`https://moodleucl.uclouvain.be/mod/resource/view.php?id=822539`
[2]see Initialization subsection for EBNF definition

2.4 Broadcast

There are three Broadcast functions:

- *Broadcast*: This one is actually a procedure, it does most of the broacast and is quite generic.
- *BroadcastFire*: This one is used specifically to broadcast when a player fires an item. It checks which Item is fired and acts accordingly, calling on the generic broadcast function on some occasions. The function returns <result>[3]
- *BroadcastMine*: Same as *BroadcastFire* except it is used when we want to explode a mine.

2.5 Game function

The main game function is *Turn*. This function allows a player to make all it's actions and is used by both the simultaneous and the turn by turn game functions. It is subdivised in smaller functions for each action (respectively Move, Charge, Fire and Mine), mostly for maintainability and readability. As result it returns the following:

- <result>::=result(surface:<Boolean> deads:<deadlist>)
- <deadlist>::=nil|<id>

2.5.1 Turn By Turn

The turn by turn function, called *PartieTT*, will call a turn for each alive player subsequently and return the state of the winner.

2.5.2 Simultaneous game

The simultaneous function, called *PartieSS* works almost in the same way. A thread is open for each player, launching its turn, the turn result being aggregated into a stream. After each player has played a turn at the same time, the stream is processed and each player can't play again.

3 Player stategies

3.1 Player004Random

As you could expect from the name of this section this player makes most of his decisions randomly. This player does not care about any information given by SayMove, SaySurface, SayMinePlaced,... He does not care either about answers received after sending a drone or a sonar. Accordingly this player does not send drones or sonars.

About moving this player moves completely randomly but still checks if his movement is valid, if he can't move anywhere he will surface.

About charging and firing items this player will charge the missiles if he can. If he can't - which will never happen in this implementation - he will try to charge the mine, then the drone and then the sonar. This player will always be able to charge his missiles because his firing strategy is to fire anything he has got as soon as possible to a random position which the item can reach. This strategy often ends up making this player commit suicide.
The reason why we still coded what would happen if the player couldn't charge or fire missiles is for the player to be easily improvable. This saved us some time when creating the smart player.

[3] See Game function subsection for the EBNF definition

In the same mindset the strategy to detonate mines is to choose a random mine in the mines we have at our disposition and make it explode regardless of it damaging us or not. That means we will stupidly detonate a mine a soon as we put it down. Again in this specific implementation we will never have any mines to detonates so it's not a problem.

About incoming sonars, the player will answer by giving randomly either the right x or the right y. The wrong x or y is randomly generated but won't be outside the grid.

About spawn selection it is randomly chosen amongst the available water squares.

3.2 Player004Smart

This player is much more smarter than the former. The reason is that it stores and use the information given by the $SayMove$ function. To record this information we added another argument to the *State* we record for the player, we called it *ennemyStateList*. This argument is updated each time the $SayMove$ function is called. *State.ennemyStateList* contains a list of records of type ennemyState(id:<id> direction:<directionlist>. The type <directionlist> is a list of <carddirection>. When the $SayMove$ function tells us that a player has moved in a direction, *State.ennemyStateList* is updated to record the player that has moved and the direction he took.

When this player is asked to move he will try to approach another player but not too much so he can still shoot at him without damaging itself. If he is too close to the other player it will try to move east, then west, then north, then south and then surface. In this current implementation this player can thus get stuck in $\cup$ shapes if the ennemy player is below him or in $\cap$ shapes if the ennemy player is above him. Same thing can happen in $\subset$ and $\supset$. However this is a rare case and can only happen if there is only one other player left, we are in a $\cup, \cap, \subset, \supset$ shape, enough turns have already been played so we know the other player position exactly and the other player happens to be just below, above, left, right of the shape.
We still don't use the sonar or drone[4]. However we scatter mines randomly, and then detonate them and fire missiles smartly. If we have a charged missile we will try to fire it smartly, if that's not possible we will try to place a mine down and then fire a drone or sonar but in this implementation the player will never be able fire a drone or sonar because we place mines down as soon as we get them.

To fire a missile smartly we pick a random ennemy, create a list of positions where he likely is and then pick a random element of this list that is in our range and won't hurt us as our fire position. If there are none we don't fire. We follow the same logic to detonate a mine.

The way we find where the other player likely is, is done by examining each point P of the grid the following way:
We know all the previous direction the ennemy player took. We try to apply the opposite of each direction to P. If while trying to apply the directions to P we go out of the map or hit an island then we stop and try another point until we have done it for every point of the grid. This gives us a list of the ennemy's likely positions but the process could take some time for big grids.

The player could be improved to be even smarter by using drones and sonars but we stopped there because this player is already pretty deadly.

[4]Again to allow the implementation of an even smarter Player on this player's basis, we still coded them.

4 Extensions

We made a simple extension that generates a random grid of the desired size. To do this we generate random lists of zeros and ones and make a list of these lists. We pay attention to generate more water squares than islands. However we don' t verify that the map generated does not contains closed shapes. This rare case never happened while we tested but it could happen in theory. If it ever does the user is simply advised to restart the game.

In addition we have drawn some 40x40 gifs with photoshop and added them in the GUI. Submarine and mines are displayed with beautiful drawings. Pictures for islands are selected randomly amongst 3 drawings (volcano, island with palm tree, rocks). Our modified GUI can be seen on the figure 1

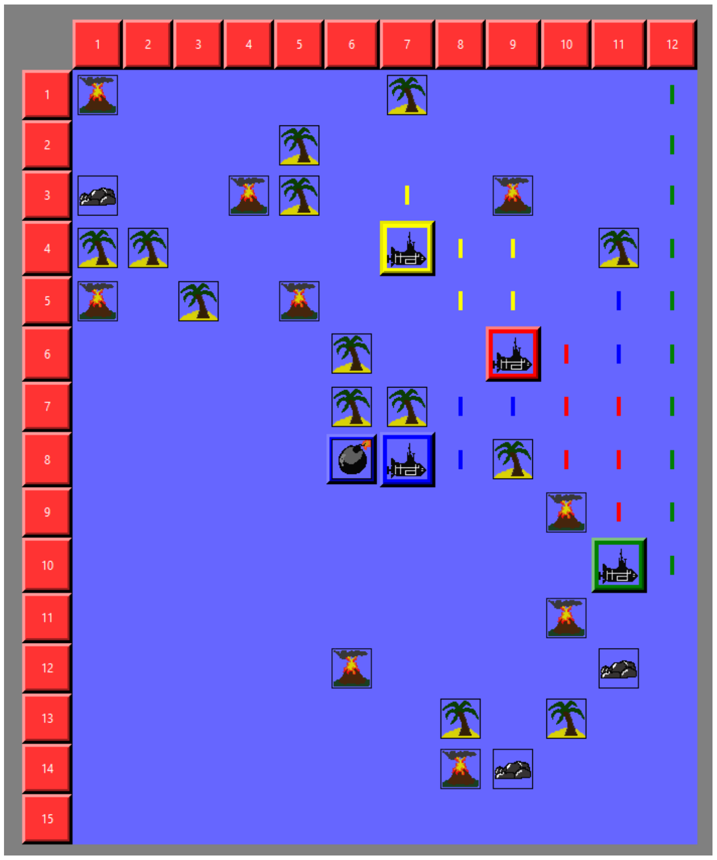

Figure 1: Our modified GUI. The map is randomly generated.