Developing the Game of Hex in a Declarative Environment

Programming Languages

Bart Middag
1st Master of Science in Computer Science Engineering
Academic year 2014-2015

1. INTRODUCTION

Hex is a strategy board game played on a hexagonal grid, theoretically of any size and several possible shapes, but traditionally an 11x11 rhombus. Each player has an allocated color: conventionally, the first player starts as red and the second as blue. Players take turns placing a stone of their color on a single cell within the overall playing board. The goal for each player is to form a connected path of his own stones linking the opposing sides of the board marked by his color, before their opponent connects his own sides with stones of his own color. The first player to complete his connection wins the game, as is the case for the player with the red stones in Figure 1.

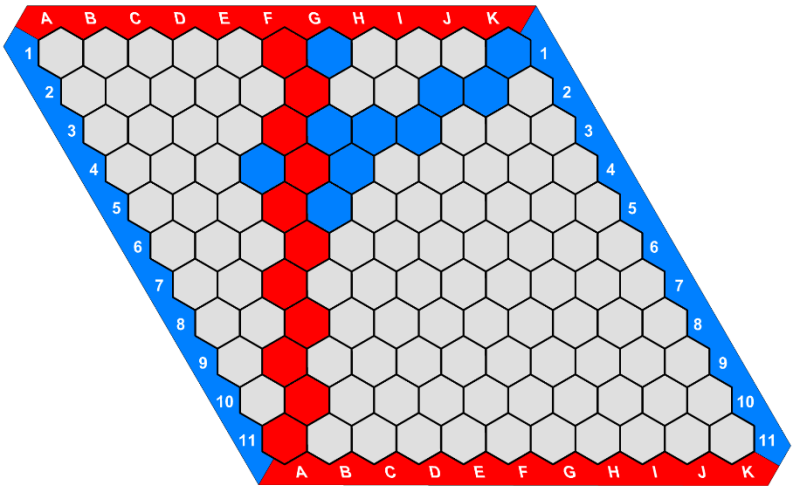

Figure 1: An example situation where the red player wins

Since the first player to move in Hex has a clear advantage, the *swap rule* can be implemented to give the second player a fair chance. This rule allows the second player to choose whether to switch positions with the first player after the first player makes the first move.

In this report, we propose a winning strategy for this game. After implementing this strategy using the declarative programming model and the message-passing concurrency model, we discuss the impact of these models on the code as well as its extensibility.

2. STRATEGY

2.1 Main algorithm: the bridge builder strategy

The strategy we implemented is what we call the “bridge builder” strategy. A bridge is a pair of stones that are not connected, but that are placed in a way that nothing can stop them from being connected, even if the opponent has the next move.

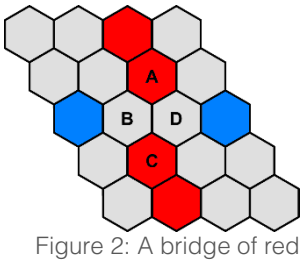

Figure 2: A bridge of red stones

In Figure 2, the red player has formed a bridge with the stones on positions A and C. If the blue player wants to prevent those two

stones from being connected, he may try to place a stone on position B or D, but then the red player will fill the other hex and connect the path.

Using this construction to our advantage, we can create a path out of these bridges and connect the bridges when we need to. In every turn, our strategy looks for path that requires the least amount of bridges to be added in order to obtain a certain victory. In the best scenario, this path can be formed in only 5 turns, as visible in Figure 3. When the path has been formed, the game is decided: no matter how much the opponent may try, he will not be able to break through the path, because the AI will keep closing the right gap when necessary.

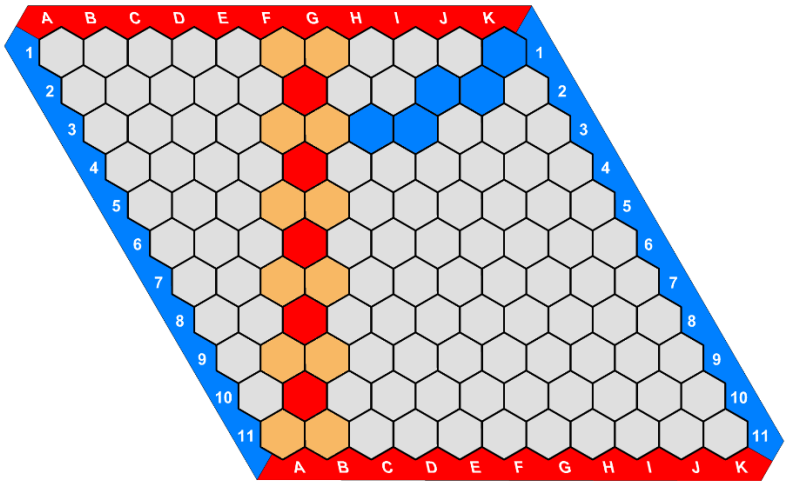

Figure 3: The red player has formed the shortest bridge path, decidng the game after only 5 turns. If the blue player now places a stone on one of the positions marked in orange, the red player will connect his path on the neighbouring position marked in orange, ensuring his victory in this session of Hex.

When the algorithm finds that a shortest bridge path exists and requires no more extra bridges to be placed, it will start connecting every bridge in that bridge path one by one until both sides are fully connected and the AI has won.

In a simplified form, the main algorithm looks like this:

```
BridgeList = list of all existing bridges for the current player
OpponentMove = position of latest move by opponent
if (OpponentMove is adjacent to both tiles in a bridge in BridgeList) {
  fillBridge(the bridge that it is adjacent to)
} else {
  ShortestBridgePath = GetShortestBridgePath()
  if (shortestBridgePath == null) {
    make random move
  } else if (amount of bridges to add to reach other side of board > 0) {
    add the next bridge along the path
  } else {
    connect the first unconnected bridge along the path
  }
}
```

2.2 Obtaining the shortest bridge path

The `GetShortestBridgePath` algorithm is the most complex helper function used in the bridge builder strategy. It is a branch-and-bound algorithm that starts on one of the sides that you want to connect and tries to find bridge paths from there. From every place it reaches, the algorithm travels across all of the bridges it can form, effectively checking every possible bridge path.

This is illustrated in Figure 4: if at any given time, the algorithm is at a position x, it will consider positions a, b, c, d, e and f as the next place to visit.

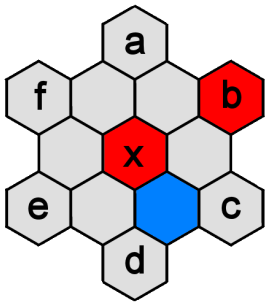

Figure 4: Places to consider for the shortest bridge path

In this case, however, the algorithm will not visit positions c and d, because one of the hexes between the bridge x-c or x-d is occupied by a stone of the opponent. If the main algorithm would decide to execute a move to position c or d, there would be no way to prevent the opponent from blocking our path at that bridge. We cannot let that happen, so the algorithm will not consider those positions as the next place to visit.

If that next place is empty, a new bridge will need to be added, but if the current player has a stone on that place, as is the case with place b on the figure, no bridge will need to be added. When all possibilities starting from the next place are checked, the algorithm continues with x.

When the other side is reached, it checks whether the amount of bridges to add to less than it was before (with 121 as the initial amount). If so, a new shortest bridge path has been found.

2.3 Flaws and possible improvements

Note that the `GetShortestBridgePath` algorithm looks for a path so that both hexes between every bridge (marked in orange in Figure 3) are free. If such a path cannot be found, that means the opponent's AI has also made a path containing a bridge. The opponent is then sure to win, *provided it can make use of the same bridge closing strategy*. That is why, regardless of whether a path can still be formed without the use of bridges, the algorithm will start executing random moves. Naturally, it would then be a much better solution to calculate the shortest path without using bridges and try to break through the opponent's path that way. We did not have enough time to implement this.

Another possible improvement would be to expand this strategy to n-*connected* paths: paths that can be connected in n moves no matter the opponent's moves. While it would require some research to properly expand the algorithm to n-connected paths, this would theoretically be able to defeat an optimal implementation of the bridge builder algorithm.

This strategy also does not currently work for boards of different shapes and sizes, as the size is hardcoded into a lot of helper functions. Different sizes would not be very hard to implement, though.

3. IMPLEMENTATION

The game of Hex was implemented in the Oz language using the Mozart 2.0 environment.

Two versions were developed: the standalone version and the version with the group's communication scheme. The reason why both versions were released is that there are too many disadvantages to the group's specifications, as we will discuss in section 3.3 and 3.4.

3.1 Modules: standalone

The standalone version of the application uses 5 modules (functors): the Main module, the Player module, the Strategy module, the Logic module and the Board module.

The *Main* module functions as the entry point of the application. It starts two threads of the Player module and gives them each other's ports so they can communicate.

The *Player* module contains the communication mechanism which will be discussed in more detail in section 3.2. To create a player, one will need to use `{Player.newPlayer Color ?PlayerPort OpponentPort OpponentType ?FinalBoard}`. The red player will then automatically send a message to the opponent containing the first move. It will then use folding to pass on the board to the next turn and it will keep replying to the opponent whenever a message is received until one of the two players has won.

The *Strategy* module contains all procedures that help a player determine which move to make. Multiple strategies can be implemented, as long as they follow the interface `{Strategy Board Player LastMoveByOpponent}`.
Currently, two strategies are available: the bridge builder strategy discussed in section 2 (as well as all its helper functions) and the random strategy, which places a random move. Both strategies return a single `(row#column)` tuple to the player so the player knows the coordinates of its next move.
As a result of this common interface, specific strategies can easily be assigned to players so that one player can play with `Strategy.bridge` and another can play with `Strategy.random`.[1]

The *Logic* module contains the algorithm that checks whether a player has won the game. It simply checks if a path can be formed between the specified player's two opposing sides of the board.

The *Board* module contains helper functions for all actions concerning the game board: creating the board; printing it; checking if a position is inside the bounds of the game board; checking whether a certain move is possible; executing a move by creating a new board based on the old one; getting the color of the stone on a certain place; and fetching a list of the positions of all stones of a certain color.

[1] However, the strategy choice was disabled for the tournament to prevent cheating.

3.2 Communication scheme: standalone

The standalone version uses a simple communication scheme. Only two types of messages are sent between the two threads: a `(row#column)` tuple (where row and column are 0-indexed for ease of use) and a `swap` message.

The board does not need to be passed on between the two threads, because each thread passes this on to the next turn internally using folding.

After a player sends a winning move to the other player, it will also send a `nil` message to itself so it knows the game is finished. When the other player receives the winning move, it will also terminate itself by sending a `nil` message.

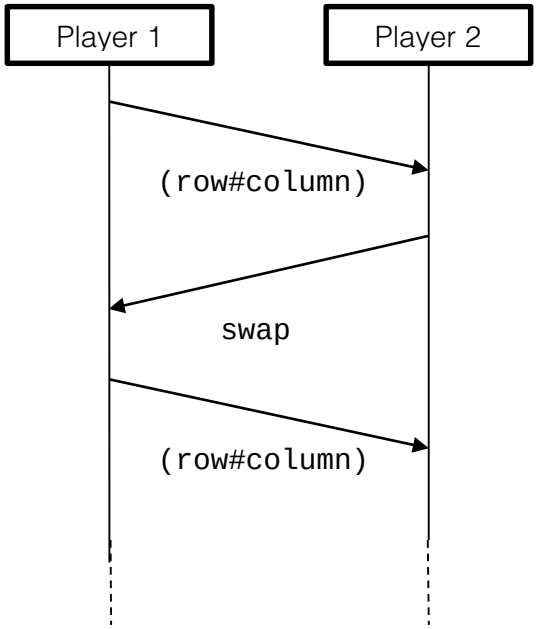

Figure 5: Communication scheme for the standalone version

This communication scheme is simple and clean and only requires a minimal amount of data to be sent between threads.

3.3 Modules: group's specifications

The group's specifications required two extra modules to be added: the App module and the Opponent module. Together, they replace the Main module from the standalone implementation.

The *App* module creates and communicates with the first player, as if the App was the second player. It also creates an Opponent through the Opponent module.
Because of the communication scheme described in section 3.4, it is required that there are three different threads active for the first player alone: the first player's thread; a thread that asks the opponent to execute a move and waits until the opponent is done before sending the result back to the third thread: the "dummy" second player thread which also reacts to the messages the opponent sends while deciding which move to make. Without the second thread, the App would not be able to respond to messages from the opponent that are sent while deciding which move to make.

The *Opponent* module works similarly: it creates and communicates with the second player, as if it is the first player itself. Two threads are necessary for this to work as intended.

3.4 Communication scheme: group's specifications

The group's specifications make communication more than a little difficult.

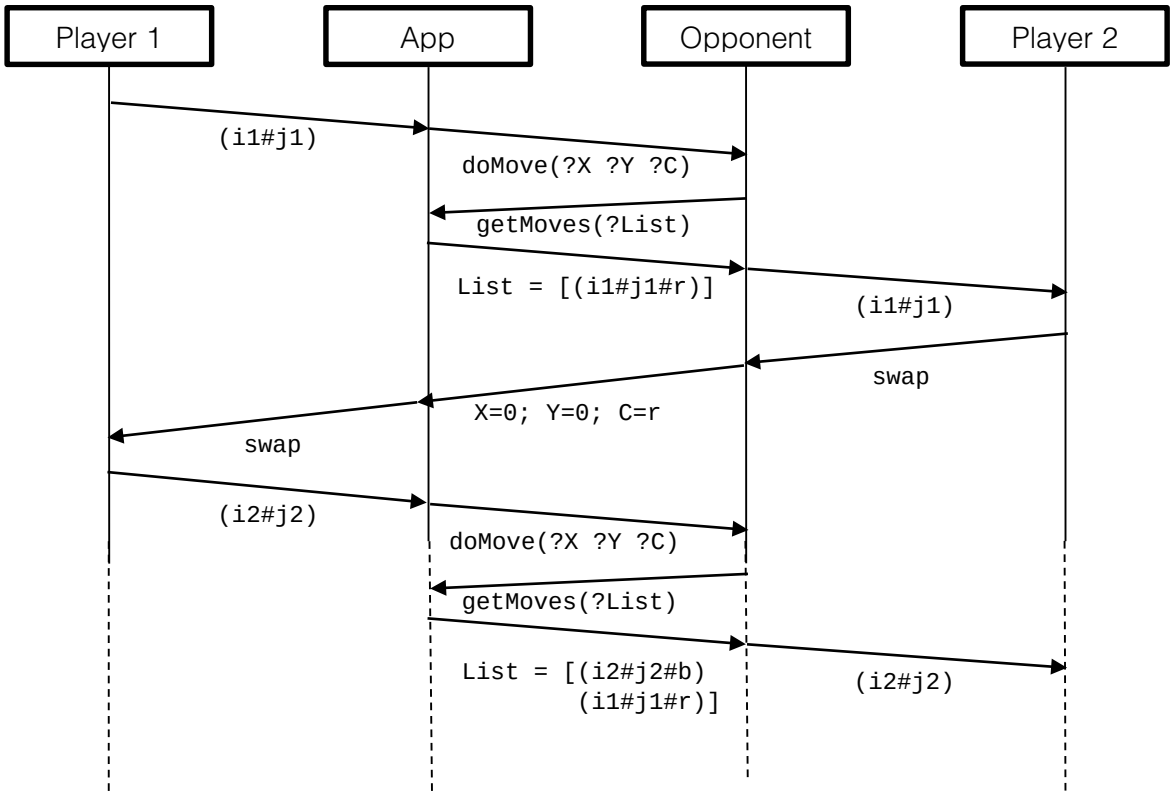

Figure 6: Simplified illustration of the group's communication scheme. In reality, App requires two threads, as it would not be able to respond to `getMoves(?List)` while waiting for the result of `doMove(x:?X y:?Y c:?C)` otherwise.

As illustrated in Figure 6, which shows the first three moves, there is a sharp contrast with the simple and clean communication scheme discussed in section 3.2 and illustrated in Figure 5.

The *App* can receive many messages from the opponent: `getMoves(?List)`, where `List` is bound to an *unsorted* list of all moves made by both players; and `getPos(X#Y ?Value)`, where `Value` is bound to the color on row `x` and column `y` (while `x` and `y` normally represent column and row respectively).

The *Opponent* can only receive the message `doMove(x:?X y:?Y c:?C)`. These variables are unbound when the message is received. It is the Opponent's task to bind them to the coordinates of its next move, where `x` is the row, `y` is the column and `c` is the color of the opponent.

As *Opponent* has does not receive any information about the first player's moves, it has no choice but to ask for a list of the first player's moves by using `getMoves(?List)`. This means all of the previous moves are transmitted every turn. Furthermore, the resulting list is unsorted, so the Opponent has to keep a list of the previous moves and

subtract that list from the new list to obtain the most recent move, so it can send this to the second player.

3.5 Comparison between communication schemes

It is clear that the communication scheme from the previous section is not only unnecessarily complicated, but also highly inefficient: if *App* and *Opponent* were to run on different machines, there would be a huge amount of overhead, because a list of all the moves is being sent over the link in every turn. The communication scheme discussed in section 3.2 sends only what is necessary, therefore making much better use of message-passing concurrency.

Because we were late to notice that a communication scheme had already been decided upon, we were not able to suggest improvements. However, a decent alternative that we had already started to implement would simply involve translating incoming and outgoing messages from the opponent's message type to our message type. The `Translate` procedure can be found in the *Player* module: `{Translate Board PlayerColor FromType ToType Message}`. Using this procedure, the code can already communicate with Rian's standalone code perfectly.

4. IMPACT OF PROGRAMMING MODELS

The game of Hex was implemented using the declarative programming model and the message-passing concurrency model. In this section, we will discuss the impact of this choice on the various aspects of the code.

4.1 Declarative programming: advantages and disadvantages

With the use of declarative programming, it is possible to make the code much shorter and easier to understand in some cases. By using procedures like `List.map` or `List.filter`, we were able to make the code much shorter: this is especially apparent in the *Board* module.

However, there is also a clear disadvantage: once a variable is bound, it cannot be updated (we need a cell for this). This means that every time the board needs to be updated, a copy of the board has to be made with only one hex being different from the previous board.

The implementation also contains a few algorithms that use backtracking, but because variables cannot be updated in the middle of a procedure, backtracking cannot be implemented by going back into the previous stage of the recursion.[2] Instead, backtracking is implemented using an extra procedure call. When that call is not at the tail of the procedure, the program has to store the return address, leading to an excessive amount of memory usage. This also feels very unintuitive for the programmer.

[2] This can be implemented, but for big problems it would require defining a lot of extra variables.

In this case, a stateful model may have been preferable, as it would not have these problems.

4.2 Message-passing concurrency: advantages and disadvantages

Message-passing concurrency made the communication very easy and intuitive to model. Since the communication itself takes care of the synchronization, a minimal amount of code was necessary to write the communication system. With message-passing concurrency, our application is also sure to run very well on distributed systems.

There are not many disadvantages to message-passing concurrency.
In contrast to shared-state concurrency, message-passing concurrency explicitly requires messages to be sent to the port of each thread that the memory needs to be shared with. This is only a problem if there are many threads, and as this is not the case in our application. Therefore, message-passing concurrency was very beneficial for this project.

4.3 Non-declarative parts of the code

The declarative model always behaves deterministically. If a program has observable nondeterminism, then it is not declarative.

Using message-passing concurrency, we use port objects for communication. Port objects are inherently non-declarative, as they are stateful.

Printing the results of the game also involves some nondeterminism due to interaction with the outside world in I/O components. Therefore, this is non-declarative.

The algorithm also uses `{Value.isFree X}` in a few helper procedures. This is a boolean function that tests whether a dataflow variable is bound or not. Since this allows using dataflow variables as a weak form of state, this is non-declarative. This boolean function was at first used for readability, as writing the same code without the function would not be very hard, but it would introduce more procedure calls and variables.

As mentioned in section 2.3, the current implementation of the bridge builder strategy will resort to random moves when no bridge path can be formed. Use of a random generator will introduce nondeterminism and is thus not declarative. Due to time constraints, this part of the code was not yet replaced with a deterministic algorithm to calculate the shortest path without using bridges, which would also have been a better strategy.

4.4 Impact on the rule modification

The implementation of the swap rule went very smoothly. Because the player code is very clear and intuitive, only a few extra lines of code were necessary. It would have been just as easy to implement using the stateful model.

4.5 Impact on the integration of other players' code

Integrating other players' code was difficult, but that was due to the odd specifications the group decided to use.

The declarative programming model did not impact the difficulty of integrating other players' code. Switching to the stateful model would not have impacted the code very much either.

Message-passing concurrency made it very easy to integrate other players' code, as using messages is very intuitive: messages can easily be translated to the target player's message type. Using shared memory would make this slightly more complicated.

5. TOURNAMENT RESULTS

A tournament was held in a small group of 5 people: Joris Van Innes, Rian Goossens, Bart Middag, Vincent Polfliet, Robbert Gurdeep Singh, and Lorin Werthen-Brabants.

The following results were obtained using the Windows x64 version of Mozart 2.0.0-alpha.0 (Build 4105) on a Intel® Core™ i7-3630QM CPU @ 2.40GHz with 8 GB RAM.

Because some of the group members' strategies are not fully deterministic, the game was executed 10 times for each pair of players. These results were then also double-checked by the other members of the group.

	Robbert	Joris	Vincent	Lorin	Bart	Rian
Robbert		0-10	4-6	0-10	0-10	0-10
Joris	10-0		10-0	0-10	0-10	0-10
Vincent	5-5	0-10		0-10	0-10	0-10
Lorin	10-0	10-0	10-0		0-10	0-10
Bart	10-0	10-0	10-0	10-0		10-0
Rian	10-0	10-0	9-1	0-10	0-10	

Figure 7: The results of the tournament. Different rows indicate different host players (player 1, who starts as red), different columns indicate different guest players (player 2, who starts as blue). The results are listed in the format "player 1 – player 2".

It is clear from the results in Figure 7 that our strategy outperforms every other algorithm in the group.

6. CONCLUSION

While the strategy we proposed can still be improved and our implementation is not optimal, it has proven to be a very challenging opponent in the tournament. By using declarative code, we can write code that is short, easy to read, and extensible, but a novice in declarative programming may find it hard to write code that does not keep state and may prefer to use both declarative and imperative programming. With message-passing concurrency, we can easily model an intuitive communication mechanism that is able to translate its messages to the messages other players expect to receive.