

Implementation of a strategy game in Prolog and the minimax algorithm for playing it

Artificial Intelligence Course Examination Project

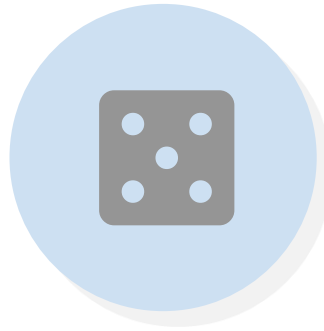
Valerio Morelli
Federica Paganica
Federico Staffolani
a.y. 2023/2024

Part 1



The game

What are the rules of the game?



Minimax implementation

How was the Minimax integrated?



The XPCE library

How was the GUI developed?



The Antiyoy game

The project focuses on the development of a strategy game **heavily inspired by Antiyoy**.

In the game theory, it can be classified as a **perfect information**, **non-deterministic**, **zero-sum** game.

The objective of the game is to **conquer as much of the map as possible**, invading enemy territories and killing their units.

Each turn, for each province, a player can **purchase resources** and **deploy units** to invade free or enemy territories

Finding a balance is the strategy of the game: **the more units a province owns, the lower its income, but the higher its offensive power.**

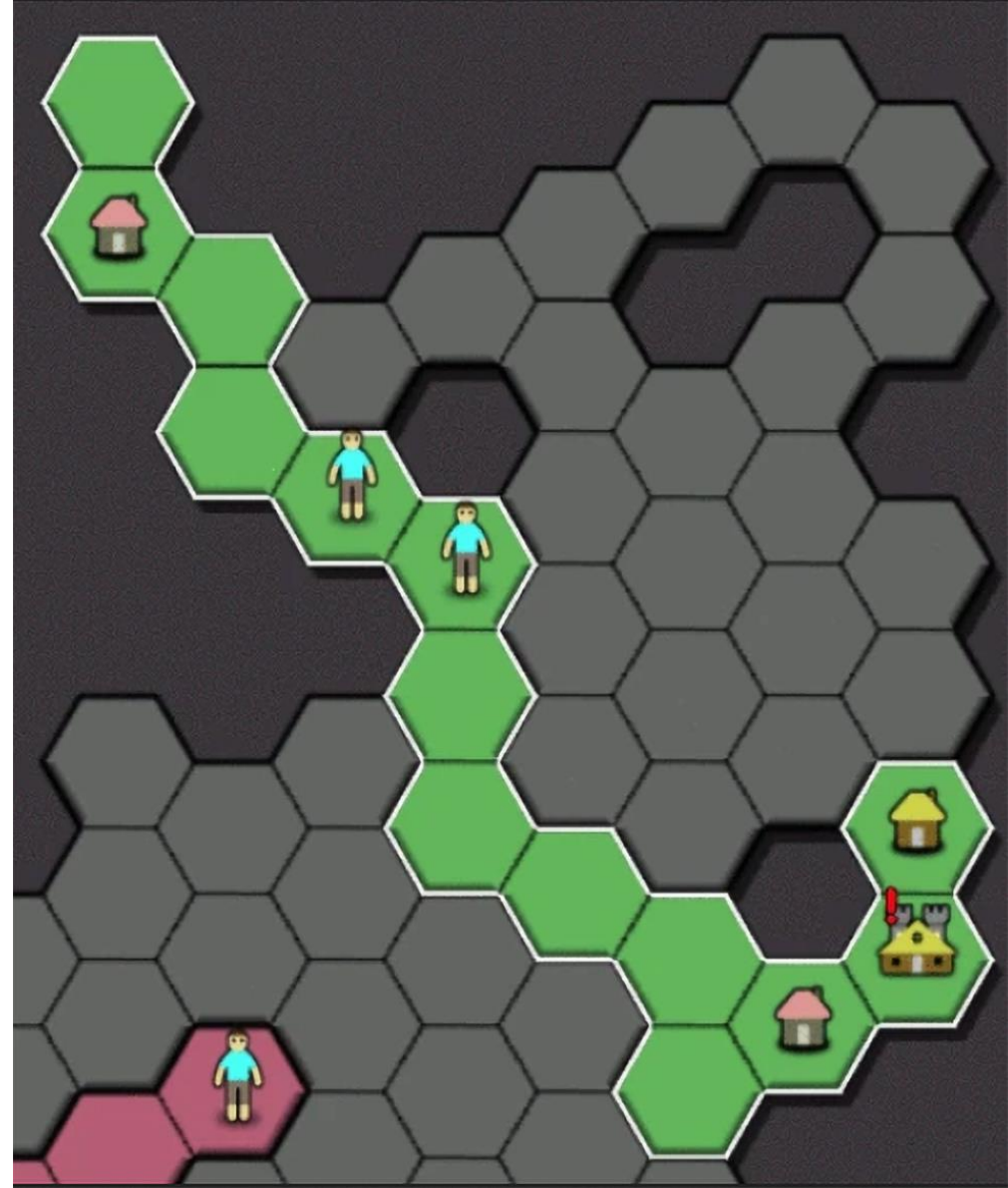
Province

A province is a **set of contiguous hexes of the same color.**


At the start of the game, all players start with one or two provinces, depending on the size of the map.




Each province has its own economy and must generate **enough income to support its own units.** Otherwise, it will go bankrupt and lose them all.

During the game's course, provinces can **split** and **merge**. When this happens, the economy is divided proportionally to their size.



Units and buildings

Icon	Unit	Strength	Protection	Cost	Income per turn
	Peasant	1	1	10 Coins	-2 Coins
	Spearman	2	2	20 Coins	-6 Coins
	Baron	3	3	30 Coins	-18 Coins
	Knight	4*	3	40 Coins	-36 Coins

Icon	Building	Protection	Cost	Income per turn
	Farm	0	12+2 Coins per Farm	+4 Coins
	Tower	2	15 Coins	-1 Coin
	Strong tower	3	35 Coins	-6 Coins

A unit can only capture a **neutral hex** or an **enemy's hex**, if it's adjacent to a friendly hex and the unit has more strength than the hex's protection. Two units may **merge into a stronger unit**.

A building can only be placed on a friendly hex. Farms are used to **generate income** and towers to **protect neighboring hexes**.



Antiyoy vs Antiyomacy

To implement the game in Prolog, it was necessary to make some changes to the original rules to simplify the development.

In the original game, trees can **randomly spread** to other hexes according to their spread rate.

Each tree on a conquered hex costs the owner 1 coin per turn, which makes the total income of that hex 0 coins per turn.

In Antiyomacy, **trees were removed** to make it a **perfect information** game, simplifying the implementation of the minmax algorithm.

Other changes

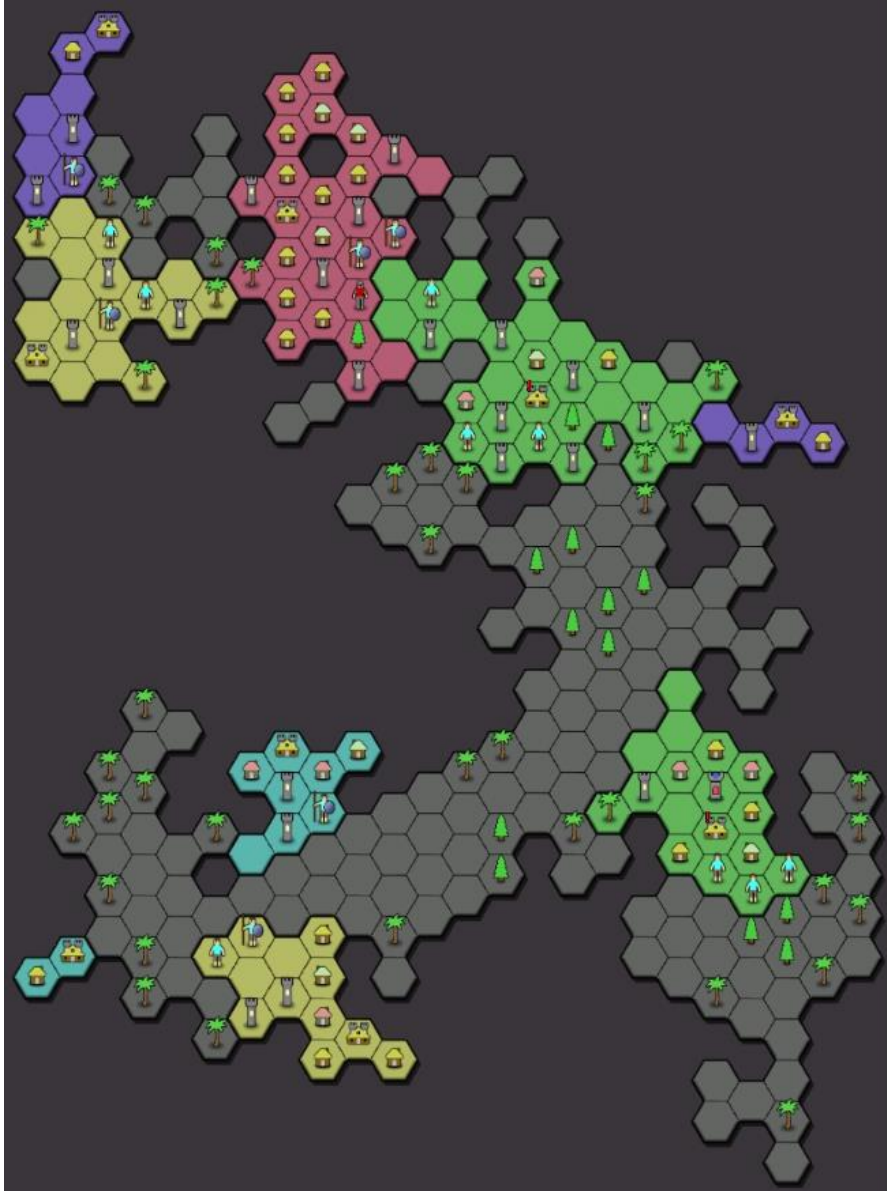
In Antiyomacy, the list of possible moves for each turn has been reduced due to **performance issues** when using Minmax.

The moveset now consists of **two options**:

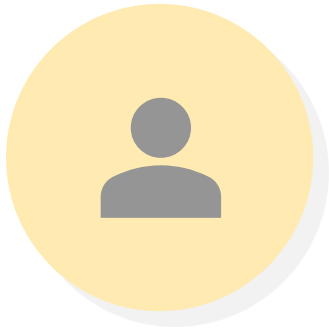
- **buying a unit or building** for each province where displacement has occurred
- **displacing a unit** for each province where no purchase has taken place.

It is also assumed that a unit can only be placed on the **inner or outer border** of a province, not within it.

Additionally, the game does not include a multiplayer mode, limiting it to **one-on-one** play.



Part 2



The game

What are the
rules of the
game?



The prolog implementation

How was the
Minimax
integrated?



The XPCE library

How was the
GUI
developed?

The game state

```
board(_Map, Provinces, Player, State, Conquests) :-  
    is_list(Provinces),  
    player(Player),  
    state(State),  
    Conquests = [_RedConquests, _BlueConquests].  
state(X) :- member(X, [play, win]).  
player(X) :- member(X, [red, blue]).
```

The *board/5* predicate represents a game state and, as such, retains all the information necessary to **uniquely define it**. It contains the **game map**, defined as a list of lists, one for each row of the grid; the **list of players provinces**, the **name of the player to move next**, the **winning state** of the game and the **number of invasions** for each player.

Game core functionalities

The Random Walker algorithm

```
% Simulate a walker random walk
% walk(+Map, +Coord, :StepAction, :WalkableCoordCondition, +Count, -NewMap)
walk(Map, _, _, _, Count, Map) :- Count =< 0. The walker dies
walk(Map, [X, Y], StepAction, WalkableCoordCondition, Count, NewMap) :-
    % Invoke the action on the current coordinate
    call(StepAction, Map, [X, Y], UpdatedMap),
    % Choose a random direction and move along it
    random_move([X,Y],[NewX,NewY]), Choose a random NSEW direction
    (
        % If the new position dwells within the map boundaries, continue the walk
        inside_map([NewX, NewY]),
        (
            call(WalkableCoordCondition, UpdatedMap, [NewX,NewY])
            -> % If the next step falls on a walkable tile, decrease the walker lifespan
                NewCount is Count - 1
            ; sea_in_map(Map) Infinite loop prevention
            -> % Else if there is at least one sea tile, do not decrease the walker lifespan
                NewCount=Count
            ; % Else, kill the walker
                NewCount = 0
        ),
        walk(UpdatedMap, [NewX, NewY], StepAction, WalkableCoordCondition, NewCount, NewMap) Take another step
    ; % Else, choose another step direction
        walk(Map, [X,Y], StepAction, WalkableCoordCondition, Count, NewMap) Choose another direction
    ).
```

How it works

The random walker algorithm creates a **connected territory**, which is exactly what is needed.

The first walker is placed at the center of the map, then it moves by randomly choosing between **one of the NSEW directions** and **placing a terrain tile at each step**.

Other walkers are spawned along the way, always at the location of the parent walker to maintain continuity, and, when a walker reaches its **lifespan limit**, it is destroyed.

The **number of walkers spawned**, and their **lifespan** determine **how jagged the map will be**.



The province search algorithm

```
% Find a province from a start hex using Breadth-first-like search
% province_bfs(+Map, +Owner, +StartHexes, [], -FoundHexes)
province_bfs(_, _, [], Hexes, Hexes).
province_bfs(Map, Owner, [Hex|Tail], Visited, Hexes) :-
    % Stop this branch if the hex is not owned by the player
    hex_owner(Hex, Owner), % Check
    hex_coord(Hex, [X,Y]), % Get
    % Scan the neighbor hexes
    % Note: We use near8/4 instead of near4/4 because units can move in the outer border
    %       and a player should never split any of their provinces after a move
    near8(Map, [X,Y], NeighborHexes),
    % Filter only the valid neighbor hexes
    findall(NeighborHex,
        (
            % Pick one neighbor hex to validate
            member(NeighborHex, NeighborHexes),
            % To be part of the province, the hex needs to be owned by the same owner
            hex_owner(NeighborHex, Owner),
            % Check that the hex has not already been visited
            \+ member(NeighborHex, Visited),
            % Check that the hex has not already been added to the ToVisit list
            \+ member(NeighborHex, Tail)
        ),
        ValidNeighborHexes),
    % All the valid neighbour hexes need to be expanded further
    append(Tail, ValidNeighborHexes, ToVisit),
    province_bfs(Map, Owner, ToVisit, [Hex|Visited], Hexes).
```

Find all the adjacent hexes that belong to the same province

Iterate the search on those hexes

How it works

During a game, players **invade each other's** territories, **split and merge provinces** and **go bankrupt**. In all these cases, the provinces must be recalculated from the map.

The algorithm works by starting from a given hex and **periodically checking and saving neighboring hexes** until no more hexes owned by the same player can be found.

Unfortunately, this method of recalculating a province **is rather expensive**, so efforts have been made to minimize its use.

refresh_province/4, instead, only refreshes the province's hexes from the map and is used when possible.

1 ₃	2 ₀					20 ₀		
4 ₀	3 ₁				11 ₂	15 ₁		
	5 ₁			8 ₂				26 ₀
		6 ₁	7 ₃	9 ₀	12 ₁	16 ₂	21 ₃	27 ₀
				10 ₂			22 ₀	28 ₀
				13 ₃	14 ₀			
			17 ₃	18 ₀	19 ₀			
		23 ₀	24 ₀	25 ₀				

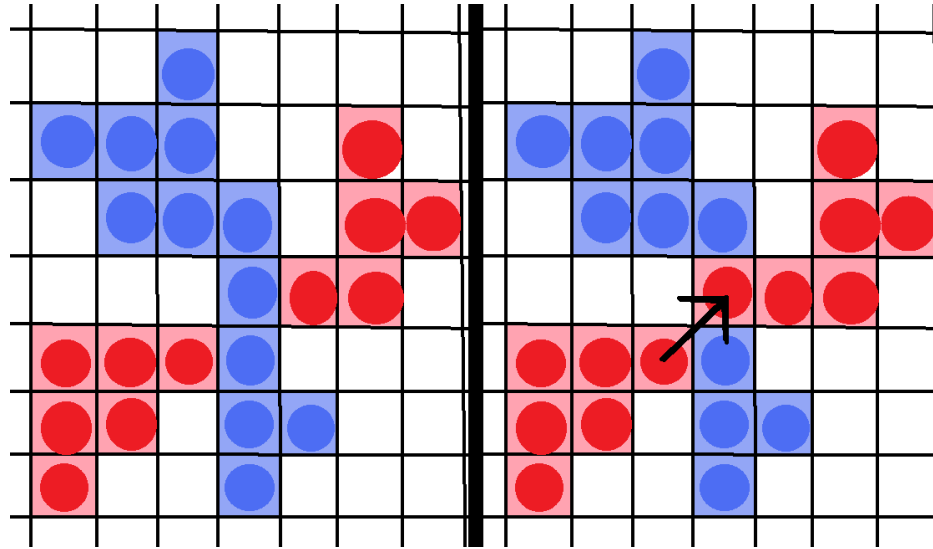
N

m

The number of the iteration in which the hex has been visited.

The number of adjacent hexes that this hex adds to the the list when inspected. A value of '0' indicates a leaf of the breadth-first search tree.

Detecting province splits and merges



Handling the possibility of merging or splitting provinces after a player invasion **is quite tricky**. Since the purpose of this project revolves around the Minimax, it would be nice to avoid recalculating all provinces on every move, since **performance is of utmost importance**. Furthermore, even if the players' provinces are recalculated after each move, there is still the problem of **identifying which provinces have split or merged with other provinces**.

Searching for a merge

1. A province merge can only occur after a unit has **conquered a free hex** or **invaded an enemy hex**.
2. If this is the case, **find all the player's provinces** in the list of the old provinces **that touch the area around the conquered hex**.
3. Calculate the money of the newly merged province as the sum of the money of these found provinces.
4. **Remove these provinces** from the list of the provinces.
5. **Add the new player's province** found with a **breadth-first search** around the conquered hex.

Searching for a split

1. An enemy province split can only occur after a unit has **invaded an enemy hex**
2. If this is the case, check if there are **any non-adjacent enemy hexes in the surrounding of the invaded hex**.
3. If this is the case, select one of these hexes and **check that it is not connected to all the other hexes surrounding the invaded hex**, using a **breadth-first search**.
4. If this is the case, **a split has certainly occurred**. Find all the provinces using a **breadth-first search** on the enemy hexes surrounding the invaded hex,
5. Select the split province from the list of the old provinces and use it to calculate the new split provinces' money.
6. **Remove the former and add the latter to the list of provinces**.
7. Otherwise, if the split hasn't happened, the invaded enemy province will still have to be recalculated due to the attack.

The Minimax integration

The four implementation steps

When applying Minimax to any game, there are four considerations to take:

1. How can the game be thought of as a two-player game?
Who is Min and who is Max?
2. How should it be decided when a **game state is terminal**?
3. How should one evaluate the **utility of a game state**?
4. How are **the children of a game state** determined?

-
- 1) In Antiyomacy there are two players: the **red** (Min) and the **blue** (Max).
 - 2) A game state is final when the last player to move **has at least 80% of the map** or has **conquered all the opponent's provinces**.
 - 3) The *eval/2* predicate assigns a score to a given board, considering the amount of **money**, the **income**, the number of **owned hexes**, the number of **invasions**, the **strength** and the **defense** of the provinces.
 - 4) The **non-deterministic** *move/2* predicate determine one possible player's move.

The alpha-beta pruning

It is a search algorithm that seeks to **decrease the number of nodes that are evaluated** by the minimax algorithm in its search tree.

The idea behind it is based on the rationality of the opponent, who, being Min or Max, always tries to choose his best move.

Unfortunately, the frequency of alpha-beta cuts **is not fixed** but depends on how much the list of possible states to analyze is **sorted in descending order**.

```
% Evaluates one Board checking for any alpha or beta cuts
% best_board(+LeftBoards, +AlphaBeta, +BoardVal, +Depth, -EnoughBoardVal)
best_board([], _, BoardVal, _, BoardVal) :- !.
best_board(_LeftBoards, [Alpha, Beta], [Board, Val], _, [Board, Val]) :-
    % Check beta test condition
    is_turn(Board, min),
    Val > Beta, !
;
    % Check alpha test condition
    is_turn(Board, max),
    Val < Alpha, !.
best_board(LeftBoards, AlphaBeta, BoardVal, Depth, EnoughBoardVal) :-
    update_alpha_beta(AlphaBeta, BoardVal, NewAlphaBeta),
    best_board(LeftBoards, NewAlphaBeta, Depth, OtherBestBoardVal),
    better(BoardVal, OtherBestBoardVal, EnoughBoardVal).
```

Inside the *minimax* module, the *best_board/4* evaluates a game state by **checking for any of the alpha or beta cuts**.

If no cut can be achieved, the predicate updates the alpha and beta values and **further explores the game state node**.

Depth-limited search with evaluation function

The Minimax algorithm can be classified as a depth-first search, and as such it **may not be able to search the entire game tree in an acceptable amount of time.**

This is especially true in this context, where both the **branching factor** and the **average number of moves to win** a game **are quite high.**

In particular, the former **increases exponentially** with the **number of units and provinces** a player has.

By introducing a **depth limit number** and decreasing it each time the *minimax/5* recursion is called, it is possible to stop the search when the depth value reaches zero and **call the evaluation function instead.**

Furthermore, a **time limit** of 5 seconds has been introduced because, as mentioned above, the number of moves to analyze to reach the given depth strongly depends on the phase of the game.

This results in a **vertical cut in the minimax tree**, so part of the player's possible move is not examined.

Generating the players' moves

```
% Get one possible move for each province (?non-deterministic?)
% move(+Board, -NextBoard)
move(board(Map, Provinces, Player, _, Conquests), board(NewMap, NewProvinces, NewPlayer, NewStat
% Get one possible move for a given province (?non-deterministic?)
% province_move(+Map, +Provinces, +Province, -NewMap, -NewProvinces)
province_move(Map, Provinces, Conquests, Province, NewMap, NewProvinces, NewConquests):- ...
% Get one possible move for a given unit and apply it (?non-deterministic?)
% unit_move(+Map, +Provinces, +Province, +HexWithUnit, -NewMap, -NewProvinces, -NewProvince)
unit_move(Map, Provinces, Province, [RedConq, BlueConq], HexWithUnit, NewMap, NewProvinces, NewP
;    % The unit moves to another hex...
% Purchase the given resource and place it in one possible location (?non-deterministic?)
% resource_buy(+Map, +Provinces, +Province, +ResourceName, -NewMap, -NewProvinces, -NewProvince)
resource_buy(Map, Provinces, Province, [RedConq, BlueConq], ResourceName, NewMap, NewProvinces,
```

The non-deterministic *move/2* predicate returns **one possible successor** of the current game state, in according to the following rules:

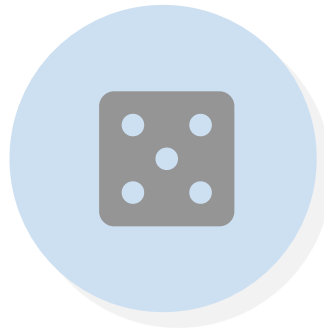
Move	How many times it can be made
<i>Buy a unit or building</i>	for each province, if no displacement has taken place, once
<i>Displace a unit</i>	for each province, if no purchase has taken place, once

Part 3



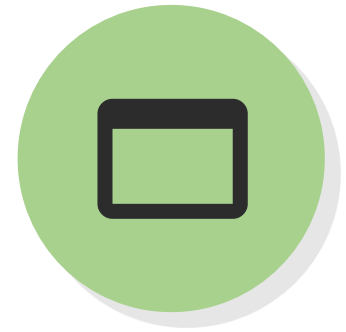
The game

What are the rules of the game?



Minimax implementation

How was the Minimax integrated?



The XPCE library

How was the GUI developed?

The need for a GUI

After integrating the minimax algorithm and **developing a terminal-based input system**, the game was ready to play.

```

      0 1 2 3 4 5 6 7
0 |   |r_|r_|r_|_| |   |   |
1 |   |   |r_|_|_|_|_| |   |
2 |   |   |   |r_|b_|   |_|_|
3 |   |   |   |r_p|b_|   |   |
4 |   |   |   |   |bf_|   |   |
5 |   |   |   |   |   |   |   |
6 |   |   |   |   |   |   |   |
7 |   |   |   |   |   |   |   |

Red provinces: -----
[Money = 2 | Income = 2 | Size = 6]

Blue provinces: -----
[Money = 0 | Income = 4 | Size = 3]
```

However, the gameplay experience **was not very satisfying** for two main reasons:

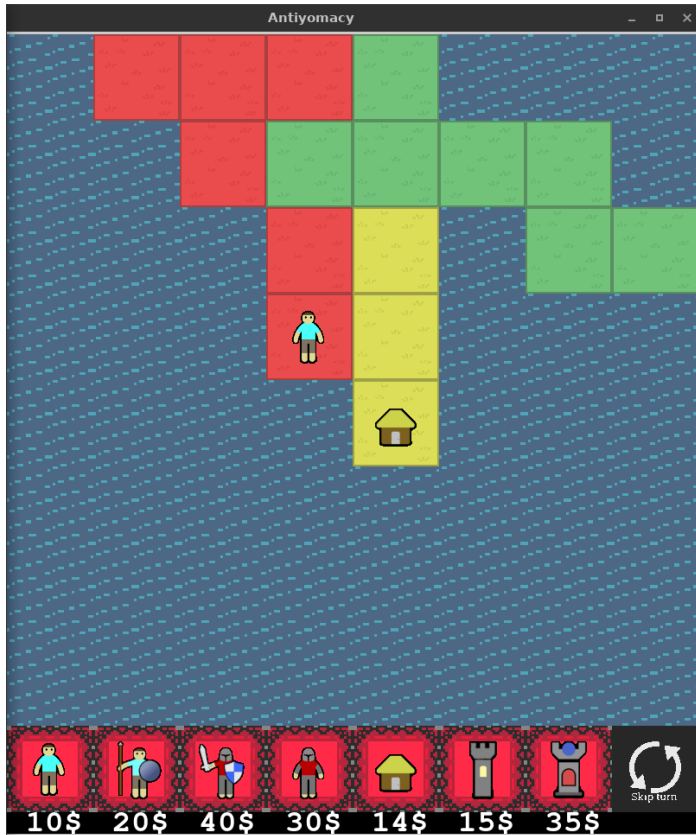
- The **map display** is not intuitive
- The **terminal-based input system** is difficult to use, as it requires the user to **enter the coordinates** of the hex in which the move is to be made each time.

XPCE is an object-oriented environment **written in C** and **fully compatible with Prolog**. The application runs on all X11 and Win32 compatible platforms.

The *swipl* installation **comes with XPCE**, as well as the emacs editor and some other useful demo in the `\xpce\proLog\demo` folder.

The gui module

To make the game **playable through direct input on the GUI**, the logic was moved from the *game* module to the *gui* module. Here, three are the main predicates:



- ***update_gui/0*** – This predicate **performs a full window refresh** based on the map, retrieved from the Prolog assert database, and some user input state flags. In accordance with the encapsulation principle, this predicate **is the only one the game module has to call** after the AI has chosen a move.
- ***on_tile_click/2*** – This predicate handles the clicks on the map tiles and **is used to perform moves**.
- ***on_frame_click/1*** – This predicate is invoked when the user clicks on one of the icons in the lower purchase menu. When an icon is selected, **the user's intention to buy it is stored** and is taken into account the next time the *on_tile_click/2* predicate is invoked.

Conclusions

The code of this project is fully documented and available in the following repository:
<https://github.com/MrPio/Antiyomacy> (GPL-3.0 license)

