UNIVERSITÀ POLITECNICA DELLE MARCHE
FACOLTÀ DI INGEGNERIA
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea in Ingegneria Informatica e dell'Automazione

ARTIFICIAL INTELLIGENCE COURSE EXAMINATION PROJECT

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

Authors

Valerio Morelli
Federica Paganica
Federico Staffolani

ACADEMIC YEAR 2023-2024

Contents

List of Figures

List of Tables

Part I

An overview of the game

CHAPTER 1

Introduction

This chapter introduces the project's goals and explores the Antiyoy game. Section 1.1 outlines the project's purpose, while Section 1.2 delves into the Antiyoy game, covering its key terms (1.2.1) and rules (1.2.2). Notably, Section 1.2.3 highlights changes introduced to enhance the Antiyomacy experience. This sets the stage for a detailed exploration of both the project's objectives and the intricacies of the Antiyomacy implementation in the following chapters.

1.1 Project Purpose

The main objective of this project is to develop an artificial intelligence capable of creating winning strategies for *Antiyomacy*, a slight different version of the *Antiyoy* game, using the *Minimax* algorithm. Both the game and the AI will be implemented in *Prolog*, by translating the rules of the game into first-order logic predicates.

1.2 The Antiyoy game

A first definition of the game is provided by the Fandom Wiki:

> Antiyoy is a mobile turn-based strategy game created by Yiotro. It is open source and also has an online version. This game is based on Slay, original game made by Sean O'Connor. [Fandom, 2021]

In its simplest form, Antiyoy, which logo is depicted in Figure 1.1, adheres to *perfect information* theory, with randomly spawning and spreading trees, which introduces an element of *casuality*.

The objective of the game is to conquer as much of the map as possible, invading enemy territories and killing their units. Each player starts the game with one or more provinces, defined as a set of contiguous, owned hexes, randomly placed on the map. Each turn, for each province, a player can purchase resources and deploy units to invade free or enemy territories. The strategy of the game lies in finding a balance between the money available, the number of units and buildings purchased, and the resulting income. In fact, the more units a province owns, the lower its income, defined as the money earned each turn. Furthermore, if a province goes bankrupt, i.e., if its money balance becomes negative, all the units in that province die.

Figure 1.1: The logo of the game

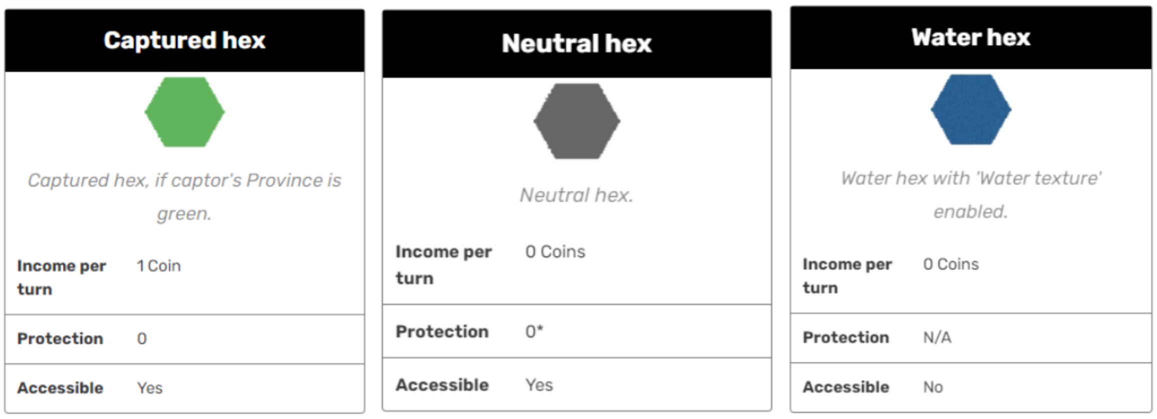
1.2.1 Rules definition

In Antiyoy, the players, two or more, take turns and in each player's turn, they can choose from three different types of move. These are detailed in figure 1.2

Move	How many times it can be made
Buy a unit	for each province, until its money runs out
Buy a building	for each province, until its money runs out
Displace a unit	for each unit in each province

Figure 1.2: The three types of moves available in the game

Then, as shown in Figure 1.3, there are three types of hexes: *neutral hexes*, *captured hexes*, and *sea hexes*. At the beginning of a game, neutral hexes are available for capture by any unit and, once captured, a hex cannot become neutral again and will change ownership. They also can be occupied by trees if the trees randomly spawn on them at the start of the game or the trees spread from captured hexes.



Captured hex

Captured hex, if captor's Province is green.

Income per turn	1 Coin
Protection	0
Accessible	Yes

Neutral hex

Neutral hex.

Income per turn	0 Coins
Protection	0*
Accessible	Yes

Water hex

Water hex with 'Water texture' enabled.

Income per turn	0 Coins
Protection	N/A
Accessible	No

Figure 1.3: The three hex types available in the game

To expand your territory and increase your coin income, it's important, in the early stages of the game, to focus on occupying neutral hexes, that provides the owner with an additional coin per turn, moving units upon them.

Conquering a hex requires placing a unit on it, provided it is allowed and there are no obstacles from other players blocking the action. Victory is achieved by conquering all hexes of the other players or possessing 80% of the total amount of hexes on the grid, excluding those of the sea type.

At the start of the game, players typically begin with one province on a small map, two on a medium map, and three on a large map. Each province has its own economy and must generate sufficient income to support its units and defensive buildings to avoid *bankruptcy*.

In fact, if a province's coin balance would be negative at the beginning of a turn, all units in that province will die; this means that it is possible to have a negative income even after death of all units.

It is also important to note how the game allows you to *merge* two (or more) provinces, simply by connecting their hexes placing a unit in a neutral hex between them, so their coin balances and coin incomes per turn are combined.

It also is possible for a province to be *split* into two or more parts following an enemy attack, with each part containing *at least two hexes*. However, if a province contains less than two hexes after a splitting, it is no longer considered a province, even though its color remains that of the previous owner. The total coins previously owned are then distributed among the newly created provinces in proportion to their size.

The following lines explain how troops move and engage in combat; all units move and fight in the same way: a unit can capture a neutral or enemy hex only if it is adjacent to a captured friendly hex and has greater strength than the hex's protection. Each unit can move within its province with a range of 4 based on the Manhattan distance and they differ in strength, protection, initial purchase cost, and ongoing cost per turn, as described in the following table.

Unit	**Strength**	**Protection**	**Cost**	**Coin income per turn**
Paesant	1	1	10	-2
Spearman	2	2	20	-6
Baron	3	3	30	-18
Knight	4	3	40	-36

Table 1.1: List of units with game values.

Note that the Knight's strength exceeds the protection it provides: to avoid stalemate situations, it can capture other Knights.

It's also possible to *merge two units* simply moving one unit onto another, or creating a new unit on the same hex as another, and they will combine into a more powerful unit with the same cost:

- Two peasants can combine to form Spearman;
- Two Spearman can be combined to form a Knight;
- A Baron can be formed by combining one Peasant and one Spearman;
- A Knight can also be formed by combining one Baron and one Peasant.

If the cost of the units exceeds 40, they cannot merge and move onto each other's hexes.

In the game, there are also various types of buildings that can be purchased with coins, just like units. Each building, as described in the following table, has specific characteristics and can be placed within its own province.

Note that the castle spawns randomly in each province and farms must be placed adjacent to the castle or, if present, to other farms. It's also important to notice that towers have a protection range of one adjacent hex and strong towers up to two adjacent hexes.

Building	Protection	Cost	Coin income per turn
Castle	1	0	0
Farm	0	12 + 2 for each farm	+4
Tower	2	15	-1
Strong tower	3	35	-6

Table 1.2: List of buildings with game values.

The last element present in Antiyoy is the *tree*, which can be of two types: *palm tree* or *pine tree*. At the start of the game, trees randomly spawns on the map on any hex (excluding water hexes) and randomly spread to other empty hexes according by their propagation rate.

If there's a tree on a hex, it incurs a cost of 1 coin per turn for the owner, resulting in a total income of 0 coins per turn for that hex. To cut down a tree, move a unit to the hex, which will immediately grant +3 coins. Trees will not spread to a hex that contains a unit or building, and buildings cannot be constructed on top of trees.

1.2.2 The variants of the game

Antiyoy has also some variants, that enrich the game's strategic landscape, which are described in the following:

- *Fog of War* restricts players' visibility, introducing an additional layer of uncertainty and challenging decision-making;
- *Diplomacy* introduces negotiation and alliances as victory conditions, altering the competitive dynamics. This variant allows players to engage in complex diplomatic interactions, including forming alliances, declaring war, and trading territories;
- The *Slay* variant, characterized by unique rules, disrupts the conventional gameplay. With altered dynamics such as the absence of farms and neutral hexes, knights unable to harm each other, and modified tree propagation rates, Slay introduces strategic nuances that require players to adapt their approaches.

1.2.3 Antiyomacy's introduced changes

In order to implement the game in Prolog, it was necessary to make some changes to the original rules to make the development of some parts easier.

Firstly, we neglected all previously mentioned variations and focused only on the simplest version of Antiyoy. This was necessary because the inclusion of features such as the 'Fog of War' would have made the game *imperfect information* and therefore difficult to adapt to the minmax algorithm and similar issues would have arisen with the other variations as well.

Furthermore, the multiplayer mode was not included, making it a one-on-one game and trees and castles were removed to make Antiyomacy a *deterministic* and *perfect information* game, in order to simplify the implementation of the minmax algorithm.

In Antiyomacy the list of possible moves for each turn has been reduced for performance issue when using Minmax, so now the moveset is made with two moves:

- buy a unit or building, for each province, if any displacement has taken place, *once*;
- displace a unit, for each province, if no purchase has taken place, *once*

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

To conclude, the following table highlights the main differences between the two games for simplicity.

Game features	Antiyoy	Antiyomacy
Grade of Information	Depends on mode	Always Perfect
Grade of chance	Trees spread randomly	None
Num of players	2 to 10	2
Alliances	Allowed in Diplomacy mode	None
Unit placements	Where allowed	Only on borders
Purchase limit	Only coins	Coins and once per turn
Displace limit per turn	Once per unit	Once per province

Table 1.3: List of differences between Antiyoy and Antiyomacy.

1.3 Vocabulary of terms

The definitions of the game keywords used above are summarised in Table 1.4. This will make it easier to understand the following chapters, where they will be used throughout to describe the implementation of the game features.

Element	Description
Hex	Basic unit of the game grid, which is made up of discrete hex units that are used for spatial representation and movement.
Coordinate	The hex's position on the game grid is represented by its coordinates in [X,Y] format.
Tile	Characterizes the type of terrain of a hex: it can be a land type, such as desert, wood, rock, sheep, clay, or hay, or sea, over which no movement or placement is allowed.
Owner	The owner of the territory is represented by the colour of a hex. For example, if a player who controls the red territory places a unit on an empty hex, it becomes red and the owner of the hex is also red.
Province	A distinct area made of connected hexes, controlled by the same owner.
Outer Border	A province's outer border consists of the hexes directly outside the province. These hexes may be neutral, sea hexes, or controlled by another player.
Inner Border	The province's internal border is formed by the last connected hexes that have the same owner as the other hexes.
Economy	The game's economy revolves around Coins and the concepts of coin balance and income. *Coin Balance* is the total amount of coins available to spend during a turn and it updates at the start of the player's turn. *Coin Income* is the amount of coins a Province earns or loses per turn.
Resource	A resource is an object that can be bought with coins. It can be a unit or a building and has its own cost and income per turn.

Table 1.4: The game's main elements and structures.

Part II

The prolog implementation

CHAPTER 2

Implementation of the game mechanics

After understanding the game rules, discussed in Chapter 1, we will now present how the implementation phase was carried out. Section 2.1 provides an overview of the project file structure and explains the motivation behind their organization in modules. Then, following the chronological order of the workflow, we will briefly explain in Section 2.2 the methods used to manage the primary functions of the game. In this section we will present the algorithm devised to detect a province split or merge after an invasion move into enemy territory. Subsequently, in Section 2.3, we will cover the implementation of the non deterministic players' moves validators, which will be used to list the possible transactions in the later minimax tree.

2.1 Code structure

An overview of the files that implement the core functionality of the game is given in Table 2.1.

2.1.1 Modules for the reduction of dependencies

> Because each predicate in a file can be called from anywhere in the program, it becomes very hard to find the dependencies and enhance the implementation of a predicate without risking to break the overall application. This is true for any language, but even worse for Prolog due to its frequent need for helper predicates. [SwiProlog, 2020c]

As the SWI Prolog documentation states, it is important to minimize mutual dependencies when defining a lot of predicates to implement different game mechanics. This can be achieved by assigning each file a specific responsibility and using module partitioning to expose only the necessary predicates to other modules. By doing so, problems caused by code changes can be reduced.

Following this philosophy, instead of `consult/1` , the predicates `module/2` and `use_module/1` have been used. The advantage of this approach is that it avoids the problems associated with circular imports.

2.1.2 Test module for mitigating code-breaking changes

Since the beginning of development, the `test.pl` file has been set up to contain a list of tests, one for each core game functionality. Each one of these tests creates an appropriate

File name	Description	Modules referenced
`map.pl`	The map manager. This file contains the predicates needed to create a random map, get one of its hexes,	`hex`
`hex.pl`	The definition of a map cell. This file contains all the operations that need to be performed on the hexes of the map.	`unit,building`
`province.pl`	The provinces manager. This file provides useful predicates for locating provinces on the map, performing a player move on them, and detecting any splits or merges after an invasion.	`map , hex , unit , building,economy`
`unit.pl`	The definition of a unit. This file contains the predicates needed to list all possible locations for a province's units on the map.	`hex , province , building`
`building.pl`	The definition of a building. This file defines the building placement rules and helps to list the locations where a purchased building can be placed. It also defines the behaviour of towers on enemy units.	`map,hex,province`
`economy.pl`	The Economy Manager. This file contains the rules that determine whether a purchase action can be carried out. It also defines how a province's money is divided when it splits or merges.	`hex,province,unit ,building`

Table 2.1: The four parameters required to establish a data binding link between a component attribute and a property of a binding context.

scenario in which the pertaining functionality is thoroughly tested. As shown in Figure 2.1, the `test/0` predicate calls all the tests in the module.

The tests were written as the features were being developed, and were run before each significant commit to ensure that the new code would not cause any problems with the features that had already been written.

2.2 Design of core game functionalities

Once the project repository was defined, the basic structures of the game were implemented, on top of which the higher-level structures would be designed.

2.2.1 Map generation using the random walker algorithm

The map is indeed one core component of the game, as every mechanic relies on it. Without much hesitation, it was decided to define a map as a list of lists, each representing a grid row. Each cell would then represent a conquerable territory or a non-conquerable sea tile, and as such would be described by the dedicated predicate `hex/4` , shown in Figure 2.2.

It is important to note that, in order to randomly generate a map, an algorithm that creates a connected territory is necessary. This is because in the case of island territories, a player on

```
test:-
    test_province,
    test_placements,
    test_purchases,
    test_farm,
    test_attack,
    test_end_turn,
    test_destroy_tower,
    test_displace_with_merge,
    test_share_money,
    test_manhattan_distance,
    test_destroy_province,
    nl, writeln('-- All the tests have succeeded! ---'), nl, !.
```

Figure 2.1: The `test.pl` predicate succeeds only if all the game tests pass

```
hex(Index, [X, Y], Tile, Owner, Building, Unit) :-
    number(Index),
    X >= 0, Y >= 0,
    tile(Tile),
    owner(Owner),
    \+ (building(Building, _, _, _), unit(Unit, _, _, _, _)).
```

Figure 2.2: The definition of a cell of the map grid

an island would be confined to it without any chance of conquering the enemy's tiles on the other land. The chosen algorithm for this task is the *Random Walker* algorithm. The function `generate_random_map/2` begins by covering the map with empty sea tiles. This assigns a proper index and coordinates to each hex using an incremented counter in the recursion. After the map has been populated with empty tiles, the *Random Walker* algorithm places a walker at the center of the map. The walker then moves by randomly choosing one of the four directions and placing a terrain tile at each step. Other walkers are spawned during the walk, always at the father walk location, to preserve continuity. When a walker reaches its lifespan limit, i.e. the number of steps it can take, it is destroyed. This algorithm allows for customization of the generated map through two parameters: the number of walkers spawned, and their lifespan. Increasing the former while decreasing the latter results in a smoother map, while having low long-living walkers creates a jagged map. To prevent maps from being too empty or too full at different sizes, these parameters are calculated as a function of map size, as shown in Figure 2.3.

```
map_size(5).
smooth(2).
walkers(X) :-      map_size(MapSize), smooth(Smooth), X is MapSize / 16 * Smooth.
walker_steps(X) :- map_size(MapSize), smooth(Smooth), X is MapSize *  8 / Smooth.
```

Figure 2.3: The *Random Walker* algorithm's two parameters

An example of the resulting map is shown in Figure 2.4.

To conclude this section, it should be noted that the same algorithm was also used in the `spawn_provinces/2` predicate to randomly generate the starting provinces of the two players on the newly generated map. To do this, it was necessary to resort to *high-order* programming, which is realised in Prolog by meta-calling with the *yall* library [SwiProlog, 2020b]. As shown in Figure 2.5, this library allows you to *define a predicate as a funtor* and pass it to another predicate, in this context the `walk/6`, which will meta-call it with the `call/[2..]` meta-predicate.

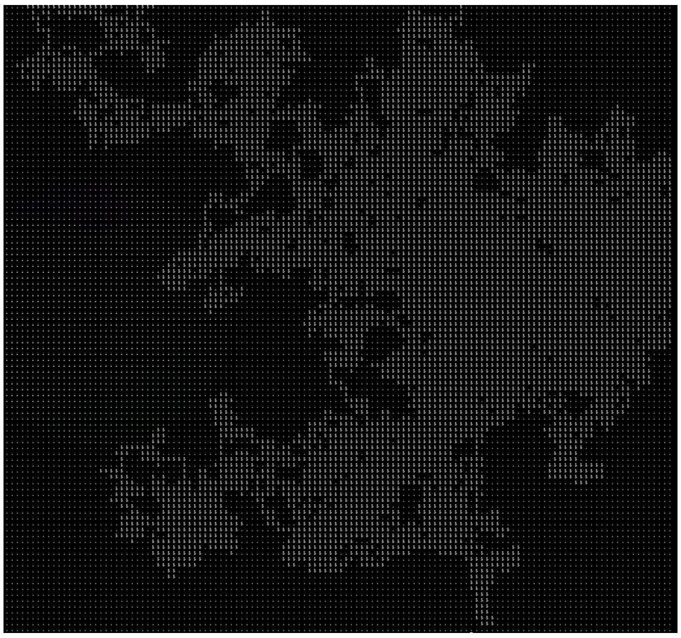


Figure 2.4: Example of a randomly generated map with the *Random Walker* algorithm

```
% Define the Random Walker action to be invoked at each step
% Note: the yall library is used here to define a lambda expression
StepAction = [ActionMap, ActionCoord, ActionNewMap] >>
    (   % Check if the walker is on a terrain hex
        get_hex(ActionMap, ActionCoord, Hex),
        hex_tile(Hex, terrain), % Check
        % Check if the hex is free or already owned by the player
        % Note: the second case is necessary to avoid cycling in a dead end
        (hex_owner(Hex, Player), ! ; hex_owner(Hex, none)), % Check
        % Change the owner to red player
        set_owner(ActionMap, ActionCoord, Player, ActionNewMap)
    ),
```

Figure 2.5: The use of *yall* library to achieve *high-order* programming

2.2.2 Finding provinces using breadth-first searching

During a game, players invade each other's territories, split provinces, and go bankrupt. Keeping the list of provinces synced with the map can be challenging because it is stored separately. In all of these cases, it is useful to recalculate the province from the map.

The algorithm devised to achieve this goal involves conducting a breadth-first search starting from a provided hex and periodically examining and storing adjacent hexes until no more hexes owned by the player can be found. When the search stops and the number of hexes found is at least two, a province has been found.

By repeating this process for all the cells on the map, it is possible to obtain the list of the players' provinces.

Unfortunately, this way of recalculating a province after a player move, although being very convenient, is rather expensive, since a lot of variables are instantiated over and over during the recursion. In the context of Minimax tree search, computational efficiency is of the utmost importance, as for each possible move all possible countermoves of the opponent are analysed to a certain depth. It is for this reason that an effort was made to minimise the use of the predicate `province_bfs`, except in cases of radical changes in province territories, such as after a split. In all the other cases, a smarter solution was devised.

As the Figure 2.7 shows, `refresh_province/4` differs from `update_province/3`,

```
% Find a province from a start hex using Breadth-first-like search
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 we don't want to create a new province after a unit has moved.
    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 no already been visited
                \+ member(NeighborHex, Visited),
                % Check that the hex has no 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).
```

Figure 2.6: The `province_bfs/5` predicate is used to scan the map and retrieve the list of provinces

which internally calls the `province_bfs/5` predicate, in that it only refreshes the hexes at the given coordinates. If the hex is already part of the province, its information will be updated. Otherwise, it will be added to the list of hexes for the province. Assuming that the caller knows in which hexes the changes made by the player's move are confined, this predicate performs significantly better than the previous one.

```
% Refresh only the provided coords on the given province, or add them if not already owned
% Note: this is faster than update_province
% refresh_province(+Map, +Province, +Coords, -NewProvince)
refresh_province(Map, Province, Coords, NewProvince):-
    province_hexes(Province, ProvinceHexes), % Get
    maplist(hex_coord, ProvinceHexes, ProvinceCoords), % Get
    refresh_province_(Map, ProvinceHexes, ProvinceCoords, Coords, NewProvinceHexes),
    change_province_hexes(Province, NewProvinceHexes, NewProvince).
refresh_province_(_, ProvinceHexes, _, [], NewProvinceHexes):- sort(ProvinceHexes, NewProvinceHexes).
refresh_province_(Map, ProvinceHexes, ProvinceCoords, [Coord|TCoords], NewProvinceHexes):-
    get_hex(Map, Coord, NewHex),
    (   member(Coord, ProvinceCoords) % Check
    ->  % The current hex is already part of the province
        member(OldHex, ProvinceHexes), % Get
        hex_coord(OldHex, Coord), % Check
        subtract(ProvinceHexes, [OldHex],ProvinceHexesWithoutOldHex),
        append(ProvinceHexesWithoutOldHex, [NewHex], NewProvinceHexes1)
    ;   % The current hex is not part of the province
        append(ProvinceHexes, [NewHex], NewProvinceHexes1)
    ),
    refresh_province_(Map, NewProvinceHexes1, ProvinceCoords, TCoords, NewProvinceHexes).
```

Figure 2.7: The `refresh_province/4` predicate is used to replace only the hexes that have changed in a specific province after a player move

2.2.3 The algorithm for detecting province splits and merges

As can be seen in Figure 2.8, handling the possibility of merging or splitting provinces after a player invasion is quite tricky. This is due to the need for a map scan after each attack. As previously mentioned, since the goal of this project revolves around the Minimax algorithm, it is clear that runtime performance is of utmost importance. This means that it would be nice to avoid recalculating all provinces on every move. 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.

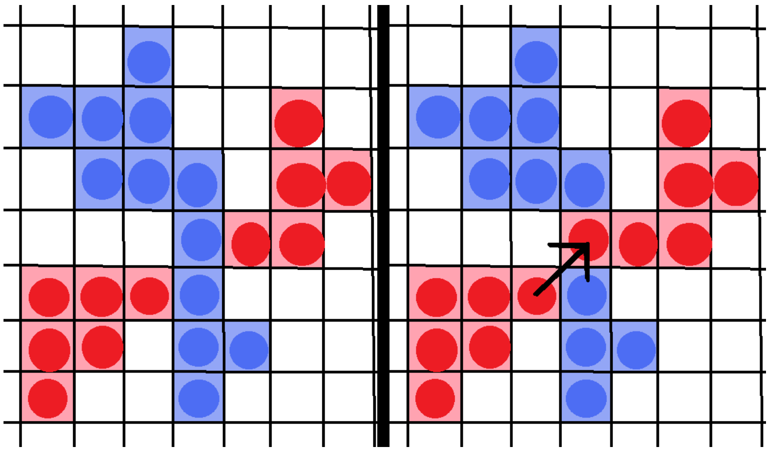


Figure 2.8: Example of both provinces splitting and merging after a red player invasion

That said, a lazy algorithm has been devised that takes advantage of sufficient conditions for both merge and split cases to reduce the use of breadth-first search for province recalculation. the algorithm breaks down as follows:

Searching for a merge

1. A province merge can only occur after a unit has conquered an unoccupied 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 use of these two predicates in sequences makes it possible to detect the occurrence of both simultaneous splitting and merging. These two algorithms are implemented in the predicates `province:check_for_merge/5` and `province:check_for_split/6` respectively.

As the algorithm presented above shows, the necessary conditions can be used to minimise the invocation of these two predicates. Figure 2.9 shows the last lines of the predicate `place_unit/8` . After the player's move has been applied and the map has been updated, the two merge and split checking algorithms are called. The former is called for both conquest and invasion, the latter only for invasion.

```
% Check for player merge in case of conquest or invasion
(   (OwnerBefore == none; OwnerBefore \= Player)
->  check_for_merge(NewMap1, Provinces, NewProvince, Hex, NewProvincesMerge)
;   subtract(Provinces, [Province], ProvincesWithoutNewProvince),
    append(ProvincesWithoutNewProvince, [NewProvince], NewProvincesMerge)
),
% Check for enemy split in case of invasion
(   (OwnerBefore \= none, OwnerBefore \= Player)
->  check_for_split(NewMap1, NewProvincesMerge, NewProvince, Hex, NewProvinces, NewMap)
;   NewProvinces = NewProvincesMerge,
    NewMap = NewMap1
).
```

Figure 2.9: The predicate `place_unit/8` checks whether mergers or splits have taken place only if the necessary conditions are met

2.2.4 Two-way unification with the CLP library

At some points in the development process, the need arose to generalise certain predicates from the order of "input" and "output" variables in arithmetic calculation. In other words, there was a need to solve equations instead of computing simple expressions with the built-in `is/2` predicate. This is because the latter should only be used with an unbound left operand. As illustrated in Figure 2.10, the `index2coord/2` predicate inside the `map` module allows bi-directional conversion between hex and coordinates.

```
% Convert index to coordinate and vice versa.
% Note: this can also be used as checker
% index2coord(?Index, ?Coord)
index2coord(Index, [X, Y]) :-
    map_size(MapSize),
    X #= Index // MapSize,
    Y #= Index mod MapSize,
    inside_map([X, Y]).
```

Figure 2.10: The predicate `index2coord/2` makes use of the `CLPFD` library to achieve two-way unification

2.3 Players' moves validators

The `unit_placement/5` and `building_placement/4` predicates form the core of player movement management. They both act as validators for the position of a resource on the map, according to the rules described in section 1.2.3. It is important to note that they do not edit the map or the players' provinces in any way, which is done by the `buy_and_place/8` and `displace_unit/9` predicates in the `province` module.

It is important to mention that, as stated in Section 1.2.3, the valid resource positions were restricted to the inner boundaries of the provinces to decrease the branching factor of the Minimax tree. The only exception is for units that need to move to the outer border in case of a conquest. The difference between the two types of boundaries is depicted in Figure 2.11.

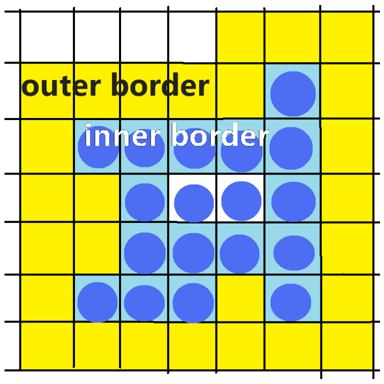


Figure 2.11: Definition of the inner border and the outer border of a province

Hence, to filter the hexes of a province using the outer_border/3 and inner_border/3, perform a recursive search on them.

The overall chain of calls that implement a player move is shown in Figure 2.12.

Buy a unit : `province:buy_and_place/8` ---> (`economy:check_buy/3` , `province:place_unit/5` ---> `unit:unit_placement/5` ---> `unit:unit_mergeable/4`)

Buy a building : `province:buy_and_place/8` ---> (`economy:check_buy/3` , `building:building_placement/4`)

Displace a unit : `province:displace_unit/8` ---> `province:place_unit/5` ---> `unit:unit_placement/5` ---> `unit:unit_mergeable/4`

Figure 2.12: Sequential visualization of predicates called to execute a player's move

Once the methods for validating the moves of buying and moving resources had been defined, it was very easy to implement the predicates responsible for generating all possible moves of a player. This is because the *backtracking* mechanism is intrinsically built into Prolog being a theorem prover.

In fact, since the `unit_placement/5` and `building_placement/4` are defined as non-deterministic predicates, meaning, as shown in Figure 2.13, that they make no claims about the number of solutions and whether or not they leave a choicepoint on the last solution [SwiProlog, 2020d], a list of possible placements can be obtained via the solution aggregators `findall/3`, `bagof/3` and `setof/3`.

It should be noted that the comments written in the code specify for the main predicates their determinism level, as depicted in Figure 2.14.

2.3.1 The building placement validator

The `building_placement/4` predicate has been designed with the following logic:

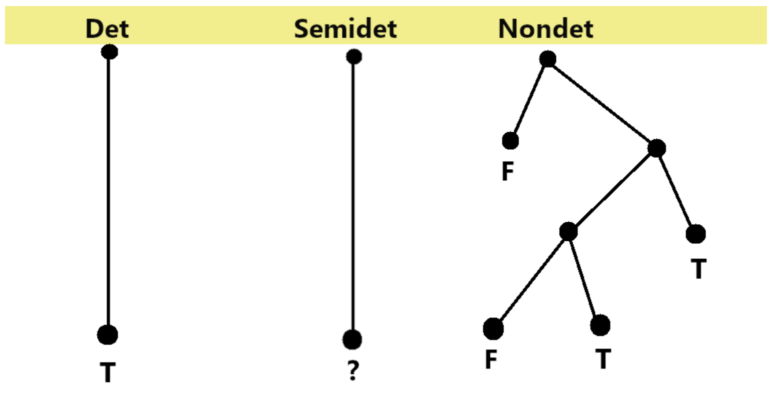
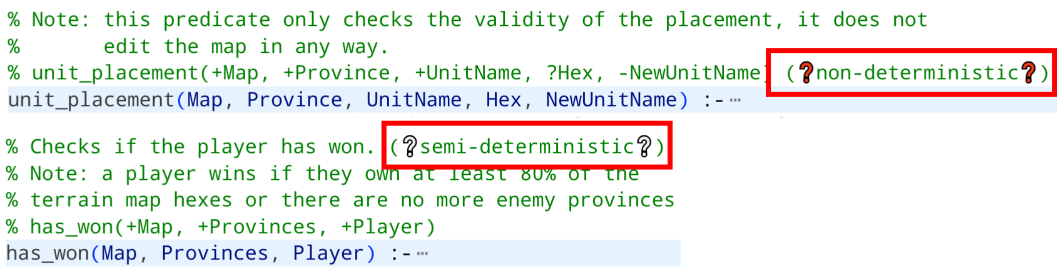


Figure 2.13: Representation of the three main forms of determinism in a theorem prover



```
% Note: this predicate only checks the validity of the placement, it does not
%       edit the map in any way.
% unit_placement(+Map, +Province, +UnitName, ?Hex, -NewUnitName) (❓non-deterministic❓)
unit_placement(Map, Province, UnitName, Hex, NewUnitName) :-⋯

% Checks if the player has won. (❓semi-deterministic❓)
% Note: a player wins if they own at least 80% of the
% terrain map hexes or there are no more enemy provinces
% has_won(+Map, +Provinces, +Player)
has_won(Map, Provinces, Player) :-⋯
```

Figure 2.14: Specification of determinism type in predicate documentation

1. To place a certain building in a particular hex on a particular map, the location should not host any units or buildings.

2. If the building to be placed is a farm, it should be adjacent to another farm, if any, or in one of the province's hexes.

3. Else, the location should be on the inner border of the province.

2.3.2 The unit placement validator

The `unit_placement/5` predicate has been designed with the following logic:

1. To place a certain unit in a particular hex on a particular map, the location should be on either the inner or outer border of the province.

2. The location should not host any units, or host a non-weaker enemy unit, or host a mergeable allied unit.

3. The location should not contain a non-destroyable enemy building or an allied building.

4. The location should not be near an enemy tower or strong tower, except for Baron units, which ignore towers, and Knight units, which ignore both of them.

CHAPTER 3

Adaptation of the Minimax algorithm

With the core functionality of the game implemented, it is now time to integrate the Minimax algorithm to teach the machine to play the game. In section 3.1, we will discuss the implementation of the `minimax` *module, including the alpha-beta pruning, and the introduction of the depth-limited search to cut the minimax tree with an evaluation function. Finally, in section 3.2 we present the predicates responsible for determining the children of a game state and discuss how human player input is handled.*

3.1 Implementation of Minimax with alpha-beta pruning

> Minmax [...] is a decision rule used in artificial intelligence, [...] for minimizing the possible loss for a worst case scenario [Wikipedia, 2024].

As the definition states, in its base version, the John von Neumann's Minimax algorithm is used in zero-sum games to find the best move in a certain state that minimizes the opponent's maximum payoff.

When applying Minimax to *Antiyomacy*, or any other game for that matter, there are four main considerations:

1. How can *Antiyomacy* 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?

Regarding the first question, *Antiyomacy* can be thought as a zero-sum game with two players: the red player and the blue player. By convention, the role of *Max* is assigned to the blue player and that of *Min* to the red player.

To answer the second question, a state is terminal if the player who made the last move has met the conditions required for victory. As described in Section 1.2.1, this means that the player owns at least 80% of the map or has conquered all enemy provinces.

The following sections address the last two questions, after providing an accurate definition of the representation of the game states.

```
% Board struct ====================================
board(_Map, Provinces, Player, State, Conquests) :-
    player(Player),
    is_list(Provinces),
    state(State),
    Conquests = [RedConquests, BlueConquests].
state(X) :- member(X, [play, win]).
```

Figure 3.1: The `board/5` predicate defines the game state

3.1.1 Definition of a game state

As Figure 3.1 shows, The concept of the board represents the state of the game. In fact, the map, the player and the winning state of the game are all that is needed to determine a distinct game state. However, for performance reasons, a list of provinces is stored in the board. This avoids having to recalculate the provinces every time a player changes the map, instead, the list is updated after each move using a *lazy* approach.

The number of conquered provinces for each player, on the other hand, refers to the number of invasions each player has achieved in enemy provinces. This information is stored in the game board and used in calculating the scoring function.

3.1.2 Alpha and Beta cuts and state choosing

The predicates written to implement the Minimax algorithm with the alpha-beta pruning are the following:

- `minimax/4` : this predicate finds the best move using Minimax with alpha-beta pruning;
- `best_board/5` : this predicate finds the best board from a list of possible state successors, whose value always remains within the Alpha and Beta limits, otherwise cuts the Minimax tree;
- `update_alpha_beta/3` , as the name suggests, this predicate is used to update the values of Alpha and Beta;
- `better/3` , this predicate returns the best game state based on who is playing. In the event of a tie, a random board will be chosen to introduce a veil of randomness into the face of multiple games with the same initial setup and choice of moves by the players.

The overall chain of calls between these recursive predicates is shown in Figure 3.2.

Minimax logic : `minimax/4` ---> (`best_board/5` ---> (`minimax/4` , (*Alfa/Beta cut* ; `best_board/5` ---> `better/3`)) ; `eval/2`)

Figure 3.2: Sequential visualization of predicates called to execute the Minimax algorithm

As mentioned above, and as can be seen in Figure 3.3, the `best_board/5` predicate stops the recursion if either of the alpha and beta cut conditions is met.

The basic logic behind the cuts is based on the rationality of the opponent, who always chooses their best move, which excludes the possibility of some game states manifesting themselves. Unfortunately, the frequency of alpha-beta cuts is not fixed, but depends on how much the list of possible states to analyse 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).
```

Figure 3.3: `best_board/5` exploits alpha-beta pruning to reduce the branching factor

3.1.3 Depth-limited search using the evaluation function

The Minimax algorithm, can be classified as a depth-first search. As one can guess, depending on the average number of transitions, this may result in an inability to search the entire game tree within an acceptable timeframe. In the context of this game, the branching factor, which is given by the number of moves a player can choose from in each turn, is relatively large and depends very much on the phase of the game. In particular, the combinations of unit movement options increase exponentially with the number of units a player has. The same applies, although to a lesser extent, to the possibilities of purchasing resources, as the amount of money available changes.

The solution to this problem was to change the depth-first search to a depth-limited search. By introducing a depth limit number as an argument to the `minimax/5` predicate and decreasing it each time the 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 has been introduced because, as mentioned above, the number of moves to analyse to reach the given depth in the minimax tree depends strongly on the state of the game, namely the number of provinces, their units and their money amounts. To do this, the *dynamic* predicate `start_time/1` was used, in combination with the `retract/1` and `assert/1` predicates, to store the timestamp of the start of the `minimax/5` recursion. The timestamp is retrieved using the built-in `get_time/1` predicate, which makes a system call to retrieve a system-dependent granularity timestamp [SwiProlog, 2020e].

The overall implementation of the `minimax/5` predicate is shown in Figure 3.4.

To conclude this section, the evaluation function deserves a few words. It has been implemented as a predicate that analyses a given board and calculates a score as a weighted sum of several factors such as the amount of money, the income, the number of owned hexes, the number of invasions and the strength and the defense of the provinces.

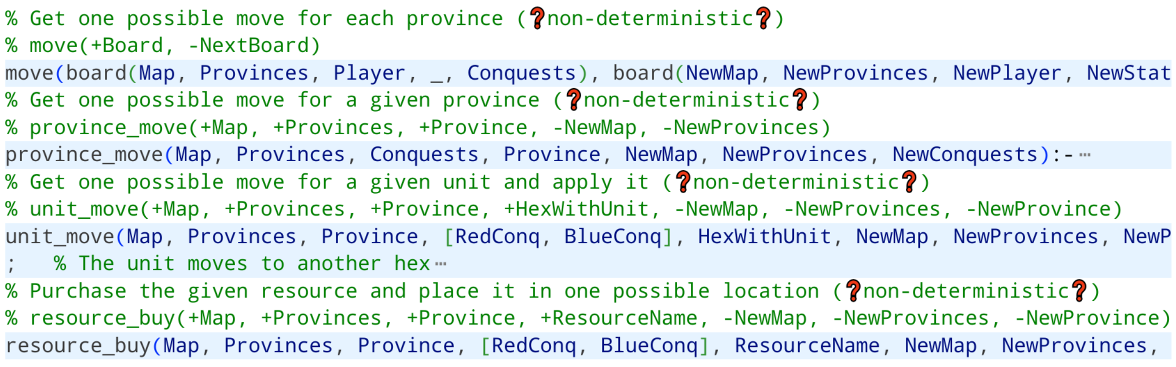
3.2 Generating the players' moves

The `move/2` predicate inside the `game` module is responsible for generating moves. As previously illustrated in 3.4, the `minimax/5` predicate actually retrieves the list of child states using the `set_of/3` solution aggregator, which was chosen over the `findall/3` aggregator due to its ability to filter duplicate solutions, on the `move/2` predicate. This implies that the latter should be implemented as a *non-deterministic* predicate. Indeed, as Figure 3.5 shows, a hierarchy of non-deterministic predicates has been implemented to

```
% Finds the best move using minimax with alpha-beta pruning
% minimax(+Board, +AlphaBeta, +Depth, -BestBoardVal)
minimax(Board, AlphaBeta, Depth, [BestBoard, Val]) :-
    % Check if the recursion has reached its depth limit
    Depth > 0,
    % Check if the minimax has reached its total time limit
    start_time(T1), get_time(T2),
    T is T2-T1, T < 10,
    % Find the list of available destination states
    setof(NextBoard, move(Board, NextBoard), NextBoards), !,
    % Choose the best state
    best_board(NextBoards, AlphaBeta, Depth, [BestBoard, Val])
;   % The depth has expired or there are no available moves
    eval(Board,Val).
```

Figure 3.4: `minimax/5` algorithm implemented as a depth-time limit search

determine a possible move for a player.



```
% 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,
```

Figure 3.5: The moves generation is implemented through four hierarchically subdivided predicates

3.2.1 Advanced recursion with the `univ/2` meta-predicate

In the original Antiyoy game, a player's move is determined by the steps shown in Figure 3.6.

- Select the player to play,
- For each of their provinces:
  - For each of its units:
    - Choose whether to move it and, if so, select a valid destination.
  - For each of its possible purchasable resource sets:
    - Choose whether or not to buy it.

Figure 3.6: A summary of the decisions required to make a move

Although, as mentioned above, the range of possible moves has been reduced for performance reasons, it has been decided, for the sake of possible future expansion, to implement them in full and then comment out certain lines of code to restrict the choice as desired. Therefore, according to the previous list, three levels of recursion are needed. Strictly speaking, one could create three related auxiliary predicates whose task is not to fulfil a particular functional requirement, but simply to implement a certain kind of recursion. In all three cases, a set of terms and a list are passed as input, and then a some variables are passed as output to the predicate. Finally, the predicate instantiates the output variables and the auxiliary recursive predicate passes them as new input variables to the next recursion. This chain ends when the list is empty and the last output terms are returned. As show in Figure 3.7, using the `univ/2` predicate, a more general solution has been found.

```
% Recursively pipe the calls of a predicate, taking the output
% of the previous step as input at each step, and returning
% the output as the input for the next step. (↑higher-order↑)
% pipe(:Functor, +InputList, +List, -LastOutput)
pipe(_, Out, [], Out).
pipe(Op, In, [H|T], Out):-
    % Instantiate the list of output vars
    length(Out, OutLength), length(Out1, OutLength),
    % Create the univ list with the the functor as
    % the first element and its arguments as the tail
    append([Op|In], [H|Out1], UnivList),
    % Instantiate the funtor with univ predicate
    Caller =.. UnivList,
    call(Caller),
    % Perform the recursion preserving the output vars
    pipe(Op, Out1, T, Out).
```

Figure 3.7: The `univ/2` predicate has been used to generalise the recursion of certain predicates

The *high-order* `pipe/4` predicate has been used in several occasions to remove redundancy and improve code readability, as demonstrated in the example in Figure 3.8.

Before

```
% helper_(+In1, +In2, +InList, -Out1, -Out2)
helper_(In1, In2, [], In1, In2).
helper_(In1, In2, [In3|T], Out1, Out2):-
    predicate_to_call(In1, In2, [In3|T], Out1_, Out2_),
    helper_(Out1_, Out2_, T, Out1, Out2).
```

After

```
pipe(predicate_to_call, [In1, In2], InList, [Out1, Out2]).
```

Figure 3.8: Use of `pipe/4` predicate to remove redundancy

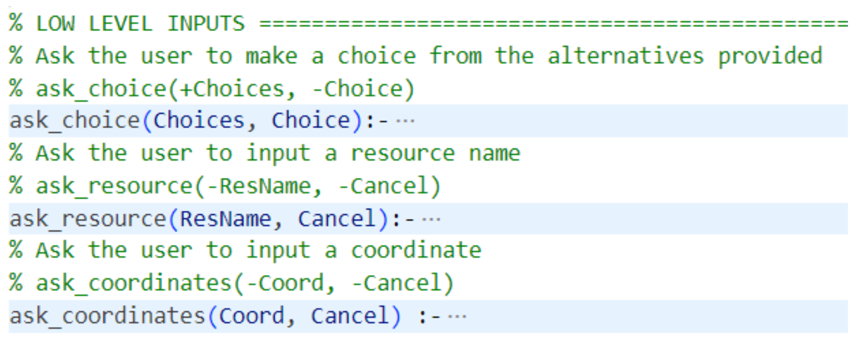
3.2.2 Processing user input for human moves

The `io` module contains the predicates for managing the input/output system and it is divided into two sections: one for *low-level* and one for *high-level* functionalities.

In the low-level section, as shown in Figure 3.9, the predicate `ask_choice/2` is called by the high-level predicates every time the user has to choose between multiple options provided in the input and returns the choice made. The predicates `ask_resource/2` and `ask_coordinates/2` manage the insertion of resources and coordinates by the user and return a boolean value `true` in the output variable 'Cancel' if the user types 'q', 'exit', 'quit', 'back', or 'esc' to cancel the previously selected move via high-level predicates.

The high-level predicates section, as shown in Figure 3.10, includes two primary predicates: `ask_displace/6` and `ask_purchase/5`. These predicates call previous predicates as required through auxiliary predicates.

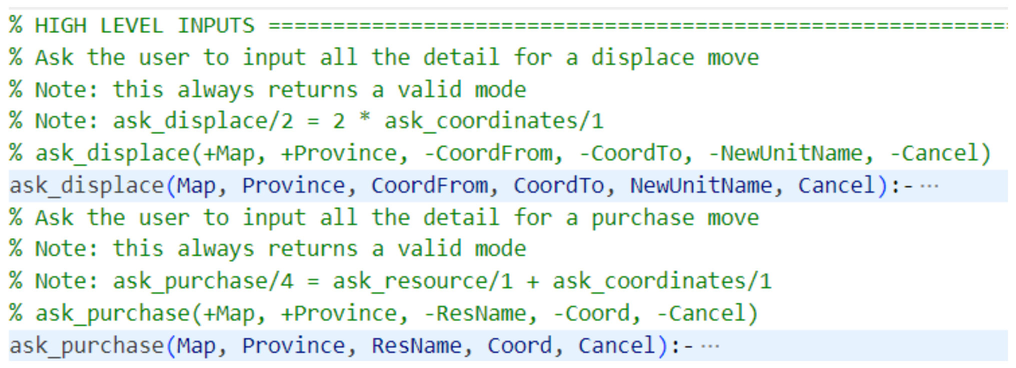
The `ask_displace/6` function assists users in selecting the starting and ending positions for a unit, represented in the 'X-Y' format. This is done in two steps: first, the starting



```
% LOW LEVEL INPUTS ==============================================
% Ask the user to make a choice from the alternatives provided
% ask_choice(+Choices, -Choice)
ask_choice(Choices, Choice):-···
% Ask the user to input a resource name
% ask_resource(-ResName, -Cancel)
ask_resource(ResName, Cancel):-···
% Ask the user to input a coordinate
% ask_coordinates(-Coord, -Cancel)
ask_coordinates(Coord, Cancel) :-···
```

Figure 3.9: Low level input predicates

position is requested, followed by the destination. Similarly, when users wish to make a purchase, they are prompted to select the desired resource type and placement location.

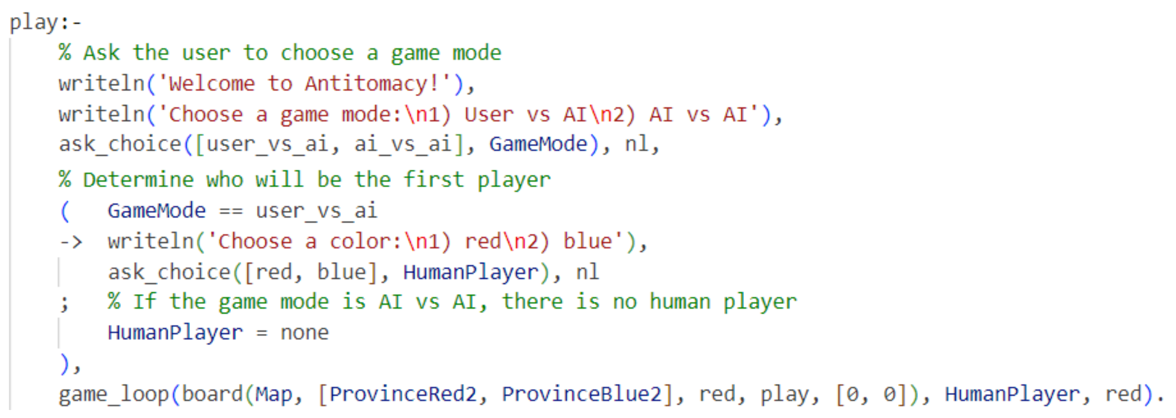


```
% HIGH LEVEL INPUTS ===============================================================
% Ask the user to input all the detail for a displace move
% Note: this always returns a valid mode
% Note: ask_displace/2 = 2 * ask_coordinates/1
% ask_displace(+Map, +Province, -CoordFrom, -CoordTo, -NewUnitName, -Cancel)
ask_displace(Map, Province, CoordFrom, CoordTo, NewUnitName, Cancel):-···
% Ask the user to input all the detail for a purchase move
% Note: this always returns a valid mode
% Note: ask_purchase/4 = ask_resource/1 + ask_coordinates/1
% ask_purchase(+Map, +Province, -ResName, -Coord, -Cancel)
ask_purchase(Map, Province, ResName, Coord, Cancel):-···
```

Figure 3.10: High level input predicates

This whole structure is then used by the callers which are the `play/0` , which initiates the game, and `game_loop/2` predicates of `game.pl` .

Initially, via `ask_choice/2` , first predicate prompts the user to choose between *player vs computer* or *computer vs computer* game modes (Figure 3.11).



```
play:-
    % Ask the user to choose a game mode
    writeln('Welcome to Antitomacy!'),
    writeln('Choose a game mode:\n1) User vs AI\n2) AI vs AI'),
    ask_choice([user_vs_ai, ai_vs_ai], GameMode), nl,
    % Determine who will be the first player
    (   GameMode == user_vs_ai
    ->  writeln('Choose a color:\n1) red\n2) blue'),
        ask_choice([red, blue], HumanPlayer), nl
    ;   % If the game mode is AI vs AI, there is no human player
        HumanPlayer = none
    ),
    game_loop(board(Map, [ProvinceRed2, ProvinceBlue2], red, play, [0, 0]), HumanPlayer, red).
```

Figure 3.11: Game starts with this predicate

Subsequently, `game_loop/2` is called with the human's chosen colour (or 'none' for computer vs computer) and the initial board as arguments. It manages turn-taking by invoking `minimax/4` for the computer's turn and `ask_provinces_moves/2` for the human's turn. This final predicate deals with managing and integrating the input/output logic described previously into the game.

CHAPTER 4

GUI implementation with XPCE library

This section is about the introduction of a Graphical User Interface (GUI) in the game Antiyomacy. The purpose of the GUI is to enhance the user experience by replacing the command line interface with a more intuitive and visual one. The GUI, developed using the XPCE library, facilitates resource management and game interaction, integrating game mechanics such as unit selection and resource purchase directly through graphical elements.

4.1 A GUI for user experience enhancement

In order to overcome the limitations of the command line interface and improve the in-game user experience, a Graphical User Interface was created at the end of the development. The primary purpose of the GUI is to provide an intuitive way to display the game map and manage interactions, rather than relying on terminal commands. The command-line interface is not very user-friendly, as it makes it difficult to visualize the current state of the game and interact with it, for the following reasons:

- *Map display*: the clarity and visual appeal of the map display are limited, making it difficult for players to understand the game's geography, including the location of units and buildings, and the ownership of each province. The game map should be displayed graphically, using colors and icons to show different types of terrain, units, buildings, and ownership statuses.
- *Game interaction*: the instructions sent from the terminal to control the gameplay are less intuitive and require the user to explicitly enter the coordinates to specify the cell on the map where the move should take place. Users should be able to interact with the game easily by clicking on GUI elements, namely, the map tiles and the purchase menu. This menu should allow players to manage the purchasing of resources.

To address the task of creating an intuitive and visually appealing GUI, the XPCE library was used to develop the interface directly in SWI-Prolog.

4.1.1 What is XPCE?

XPCE is an object-oriented library for building Graphical User Interfaces for symbolic or strongly typed languages. It provides high-level GUI specification primitives and dynamic modification of the program to allow for rapid development of interfaces. [Jan Wielemaker, 2002].

Although XPCE is compatible with languages such as Lisp and C++, its primary association is with Prolog. It acts as a bridge between the internal list of defined C structs, each representing a visual element, and the logical programming model of Prolog. XPCE enriches Prolog by introducing object-oriented programming, allowing the definition of classes using the `pce_begin/3` predicate and an appropriate syntax.

4.2 XPCE integration with SWI-Prolog

The communication between Prolog and XPCE is based on the concept of objects and messages. XPCE provides an object-oriented system for GUI management, which includes windows, images, texts, and more. Messages are sent to these objects to allow interaction between Prolog and XPCE, in more detail:

- *Object creation*: XPCE objects are created using the `new/2` predicate, where the first argument is a variable that will refer to the created object and the second argument is the object constructor call, as shown in the following code with the creation of images `new(@peasant, image('peasant.gif')).`
- *Messages sending*: to modify objects or request actions from them, messages are sent using the `send/2` or `get/3` for operations that return a value. This allows the manipulation of object attributes or the invocation of specific methods, such as sending the message `send(@window, open)` to open the interface window.

4.2.1 Images handling

In XPCE, images are treated as objects and their video output is controlled by a set of messages that define their attributes such as their placement in the application window.

Images are loaded using the *image* constructor, specifying the file path. Once the image is loaded, it can be assigned to global variables using the @ symbol for easy reference, making it easier to reuse within the interface.

The images are displayed by creating a bitmap from the image object. This bitmap is placed within the window or another graphical container. The position can be specified according to the window's dimensions and the layout of other graphical elements.

4.2.2 Events handling

XPCE simplifies event handling, such as mouse clicks or key presses, by using recognisers objects. This mechanism allows the definition of specific behaviors in response to user actions, integrating the application logic with the user interface.

In this context, a callback refers to a Prolog predicate, which can be linked to the events of specific objects. In the code `send(Bitmap, recogniser, click_gesture(left, '', single, MessageCallback))`, a click gesture is associated within a bitmap element, showing that the callback defined in `MessageCallback` should be triggered upon activation of the gesture.

Communication between the XPCE user interface and the Prolog engine presents a problem, namely the passing of complex arguments as functors through message callbacks. These callbacks are limited in their ability to communicate directly with Prolog, as they can only pass primitive data types such as integers and strings, precluding the possibility of passing more complex Prolog data structures directly. To overcome this limitation, an approach based on the use of dynamic variables and *assertz* and *retract* predicates can be used to manage the application state dynamically. The use of *assertz* and *retract* allows the Prolog knowledge base to be modified at runtime, thereby affecting the behavior of the user interface, in more detail:

- Use of *assertz*: the Prolog *assertz* predicate is used to add facts to the knowledge base, storing temporary or state information, such as the user's current selection. For example, `assertz(selected_resource(ResourceName))` records the resource, unit or building, selected by a user for the next action, such as placement on the map.
- Use of *retract*: the Prolog *retract* predicate is used to remove facts from the knowledge base. This helps to update the interface to reflect a new state of the game. For example, `retract(selected_tile(SelectedHex))` removes the current selection of a tile, allowing the user to make a new selection.

4.3 Integration with game mechanics

This section describes how the game Antiyomacy integrates its mechanics within the context of its graphical user interface. The `gui` module manages the display and interaction with the game map, units, buildings, and resource purchase menus. The various aspects and functionalities implemented in the `gui.pl` file are examined in detail below.

4.3.1 `update_gui/0` as the gui core predicate

The `init_gui` predicate plays a key role in initializing the graphical interface. It centralizes the definition of visual resources by assigning images to persistent global variables (e.g. `@peasant`, `@knight`, etc.). This approach helps to avoid redundancy when loading resources and simplifies memory management. Resources are loaded using the `new/2` construct, which associates each graphical entity (units, buildings, tiles) with a unique identifier represented by a global variable of the `image` type.

The `update_gui/0` predicate is the core for updating the graphical interface. It is responsible for redrawing the entire graphical layout based on the current state of the game, dynamically managing the display of maps, units, buildings and user interaction. Using predicates such as `send/2` and `send/3` from the XPCE library, `update_gui/0` manipulates the graphical elements predefined in `init_gui/0` and adapts them to the current state of the game. The control flow within `init_gui/0` includes clearing the game window with `send(@window, clear)`, positioning the graphical elements based on the game map, and managing interaction events such as the selection of tiles or units.

The player interactions with the graphical interface are handled by callbacks, associated with click events, to the `on_tile_click/2` and `on_frame_click/1` predicates, which manage the user's actions on the map and purchase menus respectively.

The predicate `on_tile_click/2` handles click events on map tiles. Its operating logic is articulated in several key steps:

- *Tile selection and deselection*: when a user clicks on a tile, the system checks if a tile has already been selected. If so, and if the clicked tile is different from the previous one, a unit movement can occur. Alternatively, a new tile can be selected for further action.
- *Resource purchase*: when the user has selected a resource to purchase and clicks on a tile, the system checks whether the clicked tile is valid for the placement of the selected

resource. If so, it will proceed with the purchase and placement of the resource on the tile.

- *Unit movement*: If a unit has been selected for movement and the user clicks on another tile, the system checks if the movement is valid and if so, moves the unit to the new tile.
- *GUI update*: After any click action that changes the state of the game, the GUI will be updated to reflect the new state of the game, such as new unit positions or the addition of buildings.

The `on_frame_click/1` predicate handles click events on menu items, such as buying resources or selecting specific actions. Its logic includes:

- *Resource selection for purchase*: When the user clicks on a resource icon that is available for purchase, the system marks that resource as selected. This allows the player to place it on a map tile later.
- *Resource deselection*: If the user clicks again on the same resource, it will be deselected and the intended purchase action will be canceled.
- *Skip turn*: If the player clicks on the *Skip Turn* icon, the system will proceed with the turn change and move to the next player according to the game rules.
- *GUI update*: In a similar way to `on_tile_click/1`, any action that changes the state of the game will cause the GUI updating to show the new options available or to reflect the changes made to the state of the game.

CHAPTER 5

Concluding remarks

Throughout the Antiyomacy project's development, many innovative concepts and expansion suggestions emerged. The purpose of Section 5.1 is to explore these ideas, focusing on innovations and improvements that have proven to be particularly complex to implement, or that have not yet been realized due to time limitations.

5.1 Future improvements

Each of the following points can be a useful starting point not only to enrich the gaming experience and increase strategic depth but also as a challenge from a research and technological development perspective. Some of most important developments for the future are listed below:

- The first step is to increase the variety of possible moves to approach the complexity of the original game. This requires improving the depth and efficiency of the *Minimax* algorithm. In addition, it involves the extension of the *Manhattan distance*, which is currently limited by the resulting high *combinatorial complexity*. This extension would expand the search space, challenging efficient management without compromising performance. To achieve these goals, it is essential to perform a *benchmarking* for each predicate. This will help to identify critical points, in performance, allowing optimization efforts to be focused on the most expensive predicates. *Benchmarking* predicates in Prolog is achieved by using the built-in `get_time/1` predicate, which provides an accurate timestamp used to measure the time interval between starting and finishing the execution of a predicate. This allows an accurate quantification of its impact on the overall performance of the system. The predicates for merging and splitting provinces are critical performance points, mainly due to the repeated calls to `update_province` which uses a breadth-first search algorithm to recalculate provinces based on the current map. Also, the size of the game state should be reduced to limit RAM usage. Currently, the log of all moves, each representing a complete board with map, list of provinces, and more, exceeds the gigabit of memory provided by SWI-prolog. To reduce RAM usage, the predicate `assert` can be used to permanently store the game's initial map, which contains static elements such as oceans and territories. This map serves as a fixed base upon which the dynamic variations related to the progress of the game are applied.

- One interesting possible development is the introduction of *machine learning* with *meta-level search*. This would involve using a persistent database to record knowledge gained from different gameplays. This database allows the system to learn from past matches by identifying which moves typically result in winning or losing. Instead of evaluating all possible moves from scratch, the system can consult the database and make decisions based on historical results, significantly improving performance and reducing dependence on the *Minimax* algorithm for every decision.

- A version of the game that includes *fog of war* mechanics, a variant of the Antiyoy game, as explained in Section 1.2.2, could be introduced. The concept of *fog of war* involves restricting the player's view of unexplored or distant territories from their units, simulating the limited knowledge one would have in a real battle situation. This feature adds an element of *imperfect knowledge* to the game, which increases strategic complexity as players must make decisions based on partial information. To navigate effectively in an unknown environment, *online search* algorithms are essential. *Online search* is distinguished by its ability to operate in contexts where the state space is not known in advance. The agent not only tries to reach the goal but also builds the map based on real-time information. The basic assumption in *online search* is that the agent has a full observational capacity of the states immediately surrounding it: the agent knows the actions it can take in a given state, the costs associated with those actions, and it can check whether it has reached its goal. However, it does not know in advance the *results* of its actions, it knows that it can perform a certain action in a state, but it does not know which new state this action will lead to until it performs it. Thanks to *online search*, the agent explores the map, making decisions step by step. The goal is to integrate the *online search* with the *Minimax* algorithm to handle the uncertainty and dynamics of the *fog of war* variant of the game.

- The introduction of random elements, such as trees that dynamically spawn at the end of each turn in *Antiyomacy*, is a significant challenge. This dynamic, inspired by the original game, adds a new dimension of uncertainty that standard *Minimax* is not equipped to handle, as it requires consideration of random events. To handle this added complexity, the use of the *Expectiminimax* algorithm is necessary. This variant of *Minimax* incorporates the management of randomness. Unlike *Minimax*, *Expectiminimax* calculates the weighted averages of possible outcomes, considering all probabilities associated with random events, namely the spawning of trees. The *Expectiminimax* algorithm introduces a third player, called "*casualty*", which does not follow a rational strategy of maximizing value. Instead, it simulates random events, such as the placement of trees, which affect the score of the game in a non-deterministic way. However, transitioning to *Expectiminimax* has several technical implications. It may lead to a degradation in performance due to the increase in the search space. This is because, for each possible move of the rational players, all possible actions of the "*casualty*" and their respective probabilistic outcomes must be considered. Also, the effectiveness of optimization through *alpha-beta pruning* may be compromised. *Alpha-beta pruning* is used to reduce the search space without losing the ability to find the optimal solution, but with *Expectiminimax*, the basic assumption of *alpha-beta pruning* is violated because the new actor acts without reason.

- As explained in Section 1.2.3, Antiyomacy is limited by only allowing two numbers of players. The inclusion of *evaluation vectors* could be a significant development for *Antiyomacy*, as it would allow the game to support more than two players. In this more complex *Minimax* variant, evaluation values are represented by a vector of values, where each value reflects the specific score for each player. In this configuration, decisions in the game are evaluated based on the full range of possible outcomes for all players involved, rather than just in terms of advantage or disadvantage relative to a single opponent. Therefore, the algorithm must navigate a much larger state space, where each move has multiple implications depending on its effect on the utility score of each player.

All the code for the project presented in this paper has been fully documented with comments and is available in the GitHub repository `https://github.com/MrPio/Antiyomacy`, under the GPL-3.0 license.

Bibliography

FANDOM (2021), «Antiyoy Wiki», https://antiyoy.fandom.com/wiki/Antiyoy_Wiki. (Cited at page 2)

JAN WIELEMAKER, A. A. (2002), «Programming in XPCE/Prolog», https://www.math.rug.nl/~piter/KR/userguide.pdf. (Cited at page 24)

SWIPROLOG (2020a), «Graphical applications», https://www.swi-prolog.org/Graphics.html.

SWIPROLOG (2020b), «library(yall): Lambda expressions», https://www.swi-prolog.org/pldoc/man?section=yall. (Cited at page 10)

SWIPROLOG (2020c), «Modules», https://www.swi-prolog.org/pldoc/man?section=modules. (Cited at page 8)

SWIPROLOG (2020d), «Notation of Predicate Descriptions», https://www.swi-prolog.org/pldoc/man?section=preddesc. (Cited at page 15)

SWIPROLOG (2020e), «Predicate get_time/1», https://www.swi-prolog.org/pldoc/doc_for?object=get_time/1. (Cited at page 19)

WIKIPEDIA (2024), «Minimax», https://en.wikipedia.org/wiki/Minimax. (Cited at page 17)