Maastricht University

Union find deletion in connection games

Student:
Tahmina Begum
Student Id: 06185670

Supervisor:
Dr.Cameron Browne

Report: Research Internship
February 25, 2020

Contents

1 Abstract

This research implements one modified version of the classical *union find* (UF) data structure - called Reconstruct with Weighted Quick Union and Path Compression (RWQUPC), which applies to Ludii game general system for connection games. The classical UF is an efficient data structure to solve dynamic connectivity problems. The classical UF contains 5 operations: *makeset*, *union*, *find*, *connect*, and *count*. However, this data structure has no efficient *deletion* operation. In this research, we design a framework with UF, which can support a *deletion* operation (it is called *union find deletion* (UFD)). As a result, the proposed framework helps to increase the applicability in different connection games. During the study period, we have designed 9 different ludemes (i.e.,game mechanisms defined by Java classes) by using UF, which we can directly use for game modeling in the Ludii game general system [PSS+19].

Some studies show that UF is already applied in some 2-player connection games, such as Hex, Go, etc. In our research, we have tried to increase the generability of the UF in different aspects of connection games, such as different game rules, various game board shapes and sizes, different numbers of players, different connectivity, and so on. This study is a small example of general game design to use a common framework. Moreover, to design a connection game with classical UF, players cannot access the undo button in the graphical user interface (UI). In our study, we also solve the problem of designing UI with the *deletion* operation.

Presently, UF and UFD frameworks are using in the Ludii game general system with 15 connection games. We analysed both frameworks in Ludii, where we found the RWQUPC has equal or better performance than the existing brute force methods for two games (Line of Action and Hex). We explain all the tests in the Experimental sections. All the code and test data are provided in the appendix section.

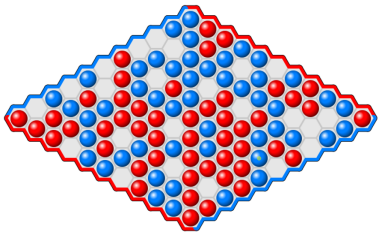
2 Introduction

A connection game is a board game in which players vie to develop or complete a specific type of connection with their pieces [Bro05]. The connection is a valuable property to play a connection game. Two popular connection games can be shown in figure-1. There are different types of game boards used in connection games, where 2 or more players can be involved. For example, in some 2-player games (for example, Hex, Y, Havannah), a player needs to connect different regions in the different types and shapes of the game board. Similarly, the aims of some other connection games (such as Line of Action, and Groups) to create a single group.

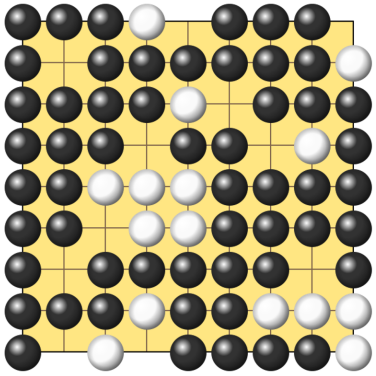
UF is an efficient and well known dynamic data structure, which has been successfully used in different types of applications, such as Network connectivity, Kruskal's minimum spanning tree algorithm, Percolation, and so on. Notably, there are various forms of the classical UF Algorithms, which are Quick Find (QF), Quick Union (QU), Ranked Quick Union (RQU), and Weighted Quick Union (WQU) [SW11][CLRS09]. The *union* operation is very convenient to create a set for some disjoint items. Besides, *find* operation is used to determine the specific set name of any particular item. The runtime of the data structure

is $\mathcal{O}(n+mlog(n))$, where n is the total items, and m is the total number of *find* operation using. The UF helps to speed up to create a connection in connection games, which is used in some 2-player connection games, such as Hex, Y [BT12], and Havannah [Ewa12].

As mentioned before, *deletion* operation is absent within the original data structure of UF. However, in the literature of general graph theory, some proposed algorithms discussed to modify the classical UF data structure, so that the *deletion* operation can be applicable. However, all the modified data structure of UF with *deletion* operation is not efficient same as the classical UF [KST02, ATG$^+$05, BY11]. Moreover, those are not suitable to apply in connection games. Because the connectivity of each game stone depends on its adjacent cell's game stone. For example, a game stone v can be connected with other game stone w, if and only if v and w are in the adjacent cells, or they have another connected path between them. In this research, we aim to analyze and design a general way to use *union find deletion* for connections game environment. Notably, we are interested in designing a system that can support the different number of players, various categories of game boards (i.e., types, sizes, and different cells), and different connectivity (such as orthogonal, all direction, etc.). As a result, a single framework can integrate more connection games in it.



(a) Hex.



(b) Go.

Figure 1: Two popular connection games.

2.1 Research Questions

To consider our research goals, we summarized the following research questions:

1. *Can we practically use the union find deletion data structure for connection games?*
2. *In which conditions union find deletion operation is applicable?*
3. *How efficient is it to use union find deletion method in comparison with the classical union find data structure?*

3 Related Work

The UF data structure was proposed by Bernard A. Galler and Michael J. Fischer in 1964 [GF64]. In 1973, the time complexity of UF was bounded to $\mathcal{O}(logn)$[1] by Hopcroft and Ullman. Robert Tarjan proved the upper bound complexity of this algorithm is $\mathcal{O}(\alpha(n))$ (where $\alpha \leq 4$) by using the inverse Ackermann's function in 1975 [CLRS09]. As mentioned before, the classical UF algorithms are divided into several types according to their performance. The first category is the Quick Find (QF) algorithm, where the runtime for the *union* is $\mathcal{O}(n)$ and find is $\mathcal{O}(1)$. The second variation is the Quick Union, the run time for the *union* and *find* operation is $\mathcal{O}(treeheight)$. The third one is the Weighted Quick Union (WQU) has performed both operations in $\mathcal{O}(logn)$. The last one is the Ranked Quick Union (RQU), which has also provided the same performance as the WQU. However, the WQU and the RQU can be used with a heuristics- called Path Compression. In this case, the *union* and the *find* operation take very nearly 1 in the amortized analysis [SW11].

In 2002, Tarjan et al. have proposed several different variations of the *union find deletion* data structures. These are *union find with deletions via k-ary trees*, *union find with deletions using incremental copying*, and *union find via path compression and linking by rank or size* [KST02]. Besides, all the variations, there includes a new operation called *insert*, which is able to add a global item into the data structure. In the first proposed variation (i.e., *union find with deletions via k-ary trees*) can be able to perform the *deletion* operation at $\mathcal{O}(logn/logk)$, where the number of children of the root or the intermediate nodes at least k in a union tree. The next algorithm is *union find with deletions with incremental copying*, where the worst-case performance of the *deletion(x)* operation depends on the summation of *find* and *insert* operation. The last modification of the *union find deletion* data structure is called *rebuilding method*, where the amortized cost for each delete is $\mathcal{O}(N)$, where N is the total number of items in the set, which contains the deleted item and $N \leq n$. In this research, the proposed method is based on this category, where we have used *reconstruct* topology instead of the *rebuilding*. In the method section, we describe more explanations about the *reconstruct* methods.

Alstrup et al. have demonstrated a constant time *delete* [2] operation of the *union find delete* data structures in 2005 [ATG+05]. However, in the worst case, the data structure contains twice more space than the actual items in the union tree. As a result, the size of the data structure does not implies the actual items in it, and the *find* operation takes a longer time than it is necessary. Amir and Simon have suggested another data structure with the constant time *delete* operation in 2010 [BY11].

In 2012, Browne et al. applied that the UF and the *Weighted Quick Having Union Find (WQHUF)* for the Hex and Y and also showed that both algorithms well perform in the *Per-Move win test* than other methods [BT12]. In the same year, Ewalds designed Havannah with the UF, where he used UF not only to detect a specific number of connections between the edges and the corners (as per the rules of the game Havannah) but also to recognize a pattern, which is a ring [Ewa12]. Notably, in all of those cases, the

[1] n is the total number of items.

[2] In this publication, authors used name delete instead of deletion.

deletion operation is not necessary.

In this study, we have focused on increasing the application areas of the UF with the *deletion* operation in connection games. Because all the proposed data structures with deletion operation designed to use in the general graph. In all the cases, after executing the *deletion* operation to delete one item from a union tree, then the number of items in that set is reduced by one. However, the union tree is not split into more subtrees [KST02, ATG$^+$05, BY11]. This means if we want to execute *deletion* to delete any item from a union tree T with N items (where, $N \geq 1$). After *deletion* executing, the size of T will be $N - 1$. However, the number of union trees in the universe remains the same. In connection games, if we delete an item from its existing tree, then it could happen the union tree divides into several union trees (which is proven in the theorem-2). To solve this problem, we introduce the *reconstruct* method, which helps to make overall implementation more efficient for general cases of connection games. We have tested the performance of the proposed data structure in a total of 15 connection games. All the tests takes 40 seconds of random playout per second (i.e., p/s) and moves per second (i.e., m/s) and in total, 20 times for each test runs in each game with a specific condition. The experimental result implies that the *union find delete* data structure is efficient enough to replace the classical UF data structure in connection games. As far as our knowledge, no research has been done to make *union find deletion* in the general application of connection games.

4 Applications

In this research, we have found four different specific cases in connection games to apply this *union find deletion* data structure. In the following, we describe all of them:

1. Firstly, this data structure can be used for those games, where the game stone swaps place at different game states. It means that first, we need to delete a game stone from its existing set; after that, we add it to a new position to make a member of a new set. For example, Line of Action and Groups game, where each game states one game stone moves from one location to others

2. The second application is in the capturing games, such as Line of Action, Go, Atari go, and so on. In this case, when any player captures one or more game stones in a game state, our proposed method can delete those from their existing set.

3. The next application is to design different types of connection games in a single framework. This method can help to optimize the practical run time and the space to design connection games.

4. Finally, if we design a connection game with classical UF, then it is not possible to develop an undo button in the user interface (UI). Because, if we add any piece into any union tree, there is no procedure to delete the last piece from the union tree. To use this proposed data structure, we can design an undo button for any UI.

5 Methods

In this research, we design two different frameworks. The first one is framework-A, where there is no *deletion* operation. It is an improved version of the classical UF data structure, where we maintain a list to store all the items in a union tree. However, framework-B contains the *deletion* operation. Both structures help us to find out the answer to the third research question, which is: *How efficient is it to use union find deletion method in comparison with the classical union find data structure?* Moreover, both frameworks are used the same optimization techniques. In the following, we discuss both structures, and the code is available in the appendix section.

5.1 Framework-A

Framework-A contains five operations, which are *makeset*, *connect*, *union*, *find*, and *groupSize*. This framework has no *deletion* operation.

5.1.1 MakeSet

In framework-A, the *makeset* is used to initialize the *parent* [3] array with a value. The size of the array depends on the size of the game board. If the game board has n cells then, the size of the array is n, where $n > 1$.

Algorithm 1 makeSet($final\ int\ totalVertices$, $final\ int\ numberOfPlayers$)

```
1:  parent = new int[numberOfPlayers + 2][];
2:  itemsList = new int[numberOfPlayers + 2][];
3:  for (int i = 1; i <= numberOfPlayers + 1; i++)
4:  {
5:      for (int j = 0; j < totalVertices; j++)
6:      {
7:          parent[i][j] = j;
8:          itemsList[i][j] = null;
9:      }
10: }
```

Explanation

As we mentioned before, framework-A is used in those games where the *deletion* operation is inapplicable, and it is closer to the classical version of UF. So, during the initialization, we have used each of the cells as an alive disjoint item (which is in the line: 7). Besides, each cells can be a root of union tree; so, we initialized a 2-dimensional *itemList* with null. There is another parameter used - called $numberOfPlayers$, which helps to create different objects of different players. To keep the overall implementation more general,

[3] *parent* is an array, which contains all the parent id of each array index.

we have considered two extra objects per game: one for the neutral players, and the other is used for all the common players information.

5.1.2 Union

Union is the main operation in the data structure. It contains some parameters, which are $activeIsLoop$, $dirnChoice$, and v. In this function, v is a cell number, and it contains a game piece that we need to add in any union tree. $activeIsLoop$ is a boolean flag that helps to detect isLoop operation (which we discuss later). $dirnChoice$ is used to select the direction of the connectivity of the union. This function does not return anything; however, after executing this function, the game stone in v position can be a member of any union tree.

Algorithm 2 union (*final boolean activeIsLoop, final DirectionChoice dirnChoice, final int v*)

```
1: Create : neighboursList for the valid position v.
2: Create : v as a singleton
3:
4: for (int i = 0; i < neighboursList.size(); i++)
5: {
6:     int i = neighbourList[i]
7:     for (int j = i + 1; j < neighboursList.size(); j++)
8:     {
9:         int j = neighbourList[j]
10:        if(ni and nj are connected)
11:            break;
12:        int rootP = find(ni, playerType)
13:        int rootQ = find(v, playerType)
14:
15:        if(rootP and rootQ are same)
16:                then return
17:
18:        if(groupSize(rootP) < groupSize(rootQ))
19:        {
20:            Set : rootP as the child of rootQ
21:            merge both itemLists of rootP and rootQ
22:        }
23:        else
24:        {
25:            Set : rootQ as the child of rootP
26:            merge both itemLists of rootP and rootQ
27:        }
28:    }
29: }
```

Explanation

In our implementation, we have used the special union algorithm, which can apply in connection games. In 2018, Browne has lectured this algorithm at the Master's course of Intelligent Search and Games (ISG) in Maastricht University [Bro18].

To generalize the implementation for connection games, we have designed a *union* operation, which can be used to connect the connection in any specific direction (i.e., all directions, orthogonal directions, forward slash directions, backward slash directions, etc.). According to the base of a game's requirement of connectivity, we select the valid adjacent positions of v. All the valid places are in the $neighbourList$. Initially, we have made the v as a singleton, which means a union tree with only one item (i.e. when v can be a singleton union tree, then the *parent* of v is v. The $itemList$ of v contains only one set bit, which is v.).

After that, we merge the singleton of v with all the existing union trees in the $neighbourList$ (which contains the valid adjacent cell numbers). At that time, we have used the WQUPC. For that reason, we first find the $rootP$ (which is the root of the ni) and $rootQ$ (which is the root of v). According to the principle of the WQUPC to minimize the depth of the new union tree, we select the $rootP$ or $rootQ$ as the new root of the union tree, which contains more items. If both trees have an equal number of items than arbitrarily select one root as a new root of the union tree. For example, union tree of $rootP$ contains more items than union tree $rootQ$. So, $parent[rootQ] = rootP$ and we merge both $itemlists$ of union tree $rootP$ and $rootQ$. If $rootP$ and $rootQ$ contain the same number of items, then $parent[rootP] = rootQ$ or $parent[rootQ] = rootP$ can be possible.

In the overall implementation, we have used the *union* operation two times for each newly arrive game stone: one time union to creates the current player's union tree and the second time to add it in a common union tree, which is for all existing players.

5.1.3 Find

This function is almost the same as the classical UF data structure [SW11].

Algorithm 3 find (*position*, *playerType*)

```
1: Set : parentId = parent of position
2:
3: if (parentId == position)
4:       then return position
5:
6: Otherwise, return find(parent[parentId], playerType)
```

Explanation

It is helped to identify the set name (i.e., the root of union tree) of *position* for the *playerType*, where *position* is a cell name. Line: 6 is used for the path compression

technique. Path compression is a heuristic technique, which helps to decrease the tree traversing time to find the root name of a union tree.

5.1.4 Connect

This function is a boolean function with two input parameters *position*1, and *position*2, which are the cell numbers of a game board. This function returns true if *position*1, and *position*2 are the member of the same union tree, otherwise false.

Algorithm 4 connect(*position*1, *position*2, *playerType*)

1: $Set : root1 = find(position1, playerType)$
2:
3: if (*position*2 *is in the itemList of* *root*1)
4: $\quad then, return\ true$
5:
6: $Otherwise, return\ false$

Explanation

Initially, we identify the root of cell *position*1 and name it as *root*1. We check whether *position*2 is true in the *root*1′s *itemlist* or not. If *position*2 is in the *itemlist* of *root*1 then this function returns true. In the classical UF, the *find* operation is used twice in a *connect* function, as there is no list for the items of the union tree. However, in this implementation, we use the *find* operation once, to check if the *position*2 exists in the same union tree or not.

5.1.5 GroupSize

This function returns the number of total items in a union tree.

Algorithm 5 groupSize(*position*, *playerType*)

1: $Set : root1 = find(position1, playerType)$
2: $\quad return\ itemList(root1, playerType).cardinality()$

Explanation

We determine the root of cell *position*1 as name *root*1. Then check the cardinality of that union tree and return it.

5.2 Framework-B

The framework-B contains 6 operations, which are *makeset*, *connect*, *union*, *find*, *groupSize*, and *deletion*. Here, the *connect*, and the *groupSize* operations are the same as framework-A. Thus, they are not explained here.

5.2.1 MakeSet

In framework-B, the *makeset* is used to initialize the *parent* array with a specific value, which is $Unused$. In this application, we define $Unused = -1$. Additionally, there are two lists used for each union tree, which are $itemsList$ and $itemWithOrthoNeighbors$. The first list is needed for the same purpose of framework-A, and the second list is used to keep the information of the orthogonal neighbors, which is used to calculate freedom in the territory types games, such as Go, Atari go, Pentalath, etc. The other criteria of this *makeset* is same as the *makeset* in framework-A.

Algorithm 6 makeSet(*final int totalVertices, final int numberOfPlayers*)

```
1:  parent = new int[numberOfPlayers + 2][];
2:  itemsList = new int[numberOfPlayers + 2][];
3:  itemWithOrthoNeighbors = new int[numberOfPlayers + 2][];
4:
5:  for (int i = 1; i <= numberOfPlayers + 1; i++)
6:  {
7:      for (int j = 0; j < totalVertices; j++)
8:      {
9:          parent[i][j] = Unused;
10:         itemsList[i][j] = null;
11:         itemWithOrthoNeighbor[i][j] = null;
12:     }
13: }
```

Explanation

In most connection games, the initial starting condition of the game board can be empty, or with some game stones in some specific cells of the game board. For example, Omega, Hex, and Havannah start with an empty game board; however, Line of Action, and Groups start with a fixed number of game stones in specific places. Thus, the presence and the absence of a game stone in a cell is significant. That is the reason there is an indicator, which is called $Unused$. The value of $Unused$ can be any integer. In the classical version of the UF, there is no parameter like $Unused$. Because all the existing disjoint data are considered as present in the universe, this is one of the limitations in the classical UF to integrate with connection game's environment. For example, an empty board within framework-A (where we follow the similar *makeset* of the classical UF) the groupProduct and the groupCount are 1 and equal to the board size respectively. However, it is invalid.

In framework-A, a singleton and a blank board cell have no difference. In the general game playing system, it is essential to distinguish between them. Besides, in 2002, Tarjan et al. showed some *union find deletion* methods, where they also used a new function-called *insert* to make each item alive. In that research, insert a new item at that union tree where it will be a member after executing *union* operation. Because in the classical UF operation, only two union trees can be merged in a single *union* operation. However, in our application, we can not follow their way as one game stone can be merged with more than one union tree, which is in Theorem-2.

Framework-B, we also introduced a new list - called $itemWithOrthoNeighbors$. This list is essential in territory games to performed the freedom calculation. It contains all the items and the orthogonal positions of a union tree.

5.2.2 Union

The *union* operation is used for similar purposes as the previous framework. However, this function needs two lists (i.e., $itemsList$, and $itemWithOrthoNeighbors$) instead of one.

Explanation

As mentioned before, the *union* in framework-B is an updated version of framework-A. Almost all the criteria are the same. However, we have used two lists here, so we need to merge both lists.

5.2.3 Find

It is used to identify the set name (i.e., the root of a union tree) of *position*, where *position* is a cell name.

Explanation

The *find* operation is same in both the framework-A and B. However, in framework-B, this function does not check the absence of a game stone in a cell. So, if any game, we access any empty cell of any player; in that case, the function returns *position*. The absence of a game stone is determined by using the content of the parent array.

Algorithm 7 union (*final boolean activeIsLoop, final DirectionChoice dirnChoice, final int v*)

```
1: Create : neighboursList for the valid position v.
2: Create : v as a singleton
3:
4: for (int i = 0; i < neighboursList.size(); i++)
5: {
6:     int i = neighbourList[i]
7:     for (int j = i + 1; j < neighboursList.size(); j++)
8:     {
9:         int j = neighbourList[j]
10:        if(ni and nj are connected)
11:            break;
12:
13:        int rootP = find(ni, playerType)
14:        int rootQ = find(v, playerType)
15:
16:        if(rootP and rootQ are same)
17:                then return
18:
19:        if(groupSize(rootP) < groupSize(rootQ))
20:        {
21:            Set : rootP as the child of rootQ
22:            merge both itemLists of rootP and rootQ
23:            merge both itemWithOrthoNeighbors of rootP and rootQ
24:        }
25:        else
26:        {
27:            Set : rootQ as the child of rootP
28:            merge both itemLists of rootP and rootQ
29:            merge both itemWithOrthoNeighbors of rootP and rootQ
30:        }
31:    }
32: }
```

Algorithm 8 find (*position*, *playerType*)

```
1: Set : parent = parent of position
2:
3: if (parent == Unused)
4:       then return position
5:
6: if (parent == position)
7:       then return position
8:
9: Otherwise, return find(parent[parent], playerType)
```

5.2.4 Deletion

This operation helps to delete an item or a group of items from a union tree. It is opposite to the *union* operation.

Explanation

In this function, there are 3 parameters, which are *deleteId*, *enemy*, and *groupDelete*. The *deleteId* is used as a reference for delete items. *enemy* is defined to indicate the owner of the union tree. If the *enemy* is true, then items need to delete from the opponent union tree or vice versa. The last parameter is the *groupDelete*, which helps to ensure how many items can be deleted. If *groupDelete* = false, then a single item delete, otherwise, a group of items deletes.

If we want to delete items from a union tree, then first determine the root of the *deleteId* for that player or opponent player (which depends on *enemy* flag). After that, copy the *itemlist* of that union tree in *bitsetsDeletePlayer* list. Then all the items of that particular union tree needs to be delete. If the *groupDelete* = false, then subtract *deleteId* from *bitsetsDeletePlayer* list. The *union* operation is applied to all the existing items in the *bitsetsDeletePlayer* list- this reconstructs the previous union tree without *deleteId*. At the reconstruction time, the same connectivity direction is used as previously. However, if the *groupDelete* = true, then all the items of the player's union tree is deleted. So, it is not necessary to reconstruct any item for that player's union tree. Similarly, in both cases, a similar procedure needs to follow in the common player's union tree. In this implementation, after executing *deletion* operation, the reconstruction part helps to produce the exact number of union trees. The newly produced union tree can be 0 or more. In the Theoretical Analysis section, we show the correctness of this algorithm in connection games environment.

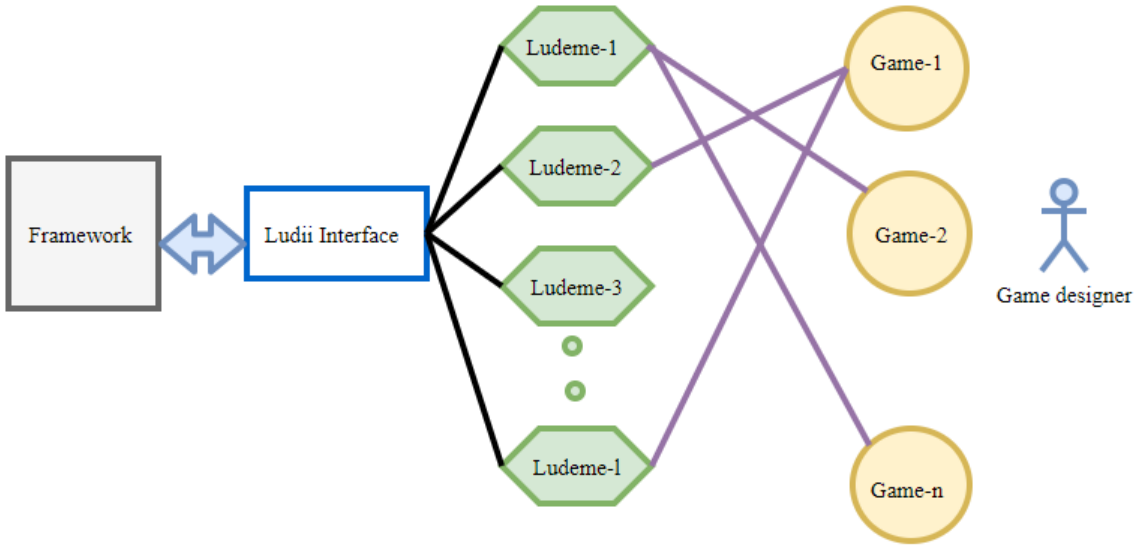
Algorithm 9 deletion ($final\ int\ deleteId$, $final\ boolean\ enemy$, $final\ boolean\ groupDelete$)

```
1: if(!groupDelete)
2:     {
3:         if(enemy)
4:         {
5:             Set : deletePlayer = currentOpponentplayerType
6:         }
7:     else
8:         {
9:             Set : deletePlayer = currentplayerType
10:        }
11:
12:    Set : deleteIdRoot = find (deleteId, deletePlayer)
13:    Set List : bitsetsDeletePlayer = itemlist of deleteId of deletePlayer
14:
15:    for each item i in bitsetsDeletePlayer
16:    {
17:        Clear : parent[i]
18:        Clear : itemList[i]
19:        Clear : itemWithOrthoNeighbors[i]
20:    }
21:    clear(deleteId) in the list bitsetsDeletePlayer
22:
23:    for each item i in bitsetsDeletePlayer
24:    {
25:        Apply : union operation for each i (reconstruction)
26:    }
27:    delete the same piece from common union tree
28:    reconstruct all existing items in common union tree
29: }
30: else
31: {
32:    Set : deletePlayer = currentOpponentplayerType
33:
34:    Set : deleteIdRoot = find (deleteId, deletePlayer)
35:    Set List : bitsetsDeletePlayer = itemlist of deleteId of deletePlayer
```

```
        for each item i in bitsetsDeletePlayer
        {
            Clear : parent[i]
            Clear : itemList[i]
            Clear : itemWithOrthoNeighbors[i]
        }
        delete the same group from common union tree
        reconstruct all existing items in common union tree
    }
    return false
```

6 Games Algorithms/Ludemes

This study focuses on increasing the applicability of the *union find deletion* in the general game system. All the proposed algorithm is implemented and tested in the Ludii game general system [PSS+19]. The basic integration system is shown in the figure-2. In this integration system, a game designer requires some user-friendly algorithm to design game modelling (figure-3). Those algorithms are called Ludemes, which are designed with java classes. Moreover, a game designer uses those ludemes to access the internal framework for any game, where those algorithms name uses as a keyword in each game modelling.



(a)

Figure 2: Basic integration system in Ludii.

In connection games, some common aspects make a difference between one game to

Hex.lud | Havannah.lud

```
(game "Havannah"
   (players 2)
   (equipment {
      (board (hexagon <1>) (hexagonal))
      (piece "Ball" Each)
   }
   )
   (rules
      (play (to (empty)))
      (end
         (if
            (or
                {
                    (isLoop)
                    (isConnect 3 SidesNoCorners)
                    (isConnect 2 Corners)
                }
            )
            (result Mover Win)
         )
      )
   )
)

(option <1> "Board Size/4" <4>)
(option <1> "Board Size/5" <5>)
(option <1> "Board Size/6" <6>)
(option <1> "Board Size/7" <7>)
(option <1> "Board Size/8" <8>)*
(option <1> "Board Size/9" <9>)
(option <1> "Board Size/10" <10>)

(metadata
  { "rules" "Havannah is a two players game.
  There are three winning conditions: 1. Connect between any three edges (without corner points),or,
   2. Make a bridge connection between any two corners, or, 3. loop around one or more non-friend cells."}
  { "source"   "https://en.wikipedia.org/wiki/Havannah"}
)

```

(a)

Figure 3: Havannah modeling in Ludii.

another game, such as connectivity, board types, cell types, and so on. Here, all ludemes can be supported in different connectivity[4], which is orthogonal connection, diagonal connection, and all (which means all adjacent cell connection). The algorithms are tested in all and orthogonal relationships. Moreover, the proposed algorithms are of two types. Some algorithms are used in each game state of a game. Some of them are used in the end game state to calculate the end score. It is possible to use all the ludemes for multiplayer games.

[4]dirnChoice uses to indicate the direction choice for any game.

6.1 IsConnect(*number*, *regions*, *role*, *regionType*)

IsConnect is a boolean type of ludeme with 4 parameters. It is used to check the connection between a specific number of regions.

- *number* is an optional parameter. It defines the minimum number of regions that need to connect.
- *regions* is used to the specific region in the game board.
- *role* refers to the type of player.
- *regionType* is defined any particular part of the game board, such as the north, south, east, west, and north-south.

Mathematical Explanations

Assume that we have a set of regions R, $R = \{r_1, r_2, r_3, ..., r_p\}$, and a *number*, where $2 \leq number \leq p$ (*number* is an optional parameter, If $number = null$, then $number = p$). The last move of the current game state is v. After executing *(v)*, v is the member of a union Tree T and the union tree T also increases its size. If the number of the intersection of the items of union tree T and the regions R is at least *number*, then the function returns true. This ludeme presently works with Hex, Havannah, Kensington havannah, Y-hex, Cross, Gonnect, and Chameleon.

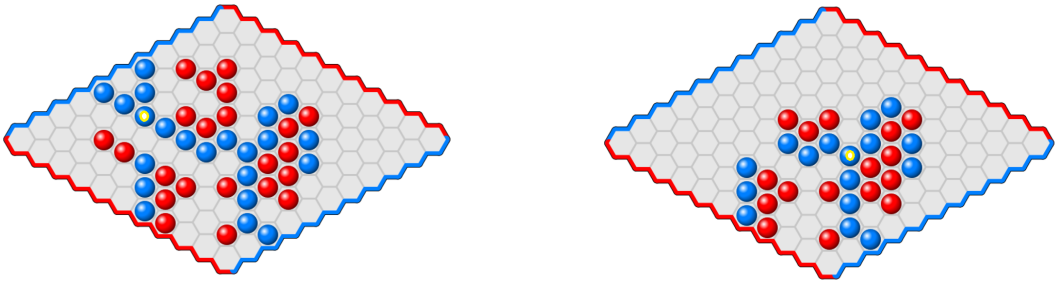
Algorithm 10 isConnect (*number*, *regions*, *role*, *regionType*)

```
1:  Set : v = last move
2:  Set : currentPlayer = role
3:  Union(v, currentPlayer)
4:  Set : connection = 0
5:
6:  if (number == 0)
7:      number = regiontype.size()
8:
9:  for (i = 0; i <= regiontype.size(); i++)
10: {
11:     if intersect(regiontype, itemList(v, currentPlayer))
12:         connection++
13:     if (number == connection)
14:         return true
15: }
16: return false
```

Explanation

IsConnect helps to ensure the connectivity between a specific number of regions. For example, in Hex, there are two separate regions for each player, and the winning condition is to connect both regions. In this case, first, we add each movement of each player in that player's union tree. After adding each stone into a union tree, this ludeme checks whether the latest produced union tree intersects with both regions or not. If the number of intersection satisfies the condition, then it returns true. In figure-4a, the last move v (with a yellow spot), merge two union trees, where the new union tree is intersecting two regions. However, in the next figure-4b, the last move v joins some union trees.

In this way, we check the winning strategy of the Hex. Similarly, for the winning strategy of havannah, each player needs to connect at least two corners or three edges or make a loop. For the first two winning strategies, one can use this ludeme.



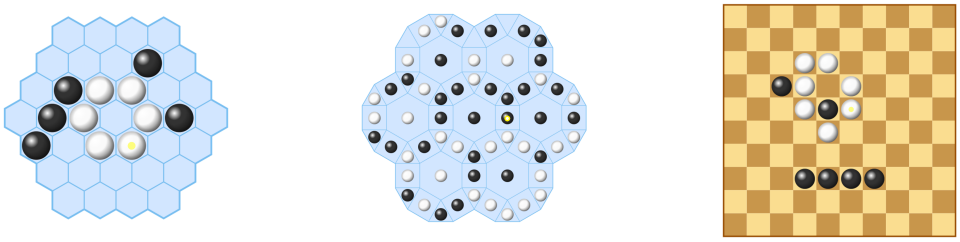
(a) v (yellow spot) merge two regions. (b) v has not connected two regions.

Figure 4: Two game states for Hex.

6.2 IsLoop (*all*, *enemy*, *empty*, *dirnChoice*)

IsLoop

In connection games [Bro05], a loop is a connected component, which is created with one or more player's stones. Sometimes a loop can also be called a ring. In figure-5a, there are two loops created by a white player in different game boards. A loop can be two types: open loop, or full loop. Inside the open loop, there is at least a cell, which can be empty or contains a non-friendly game stone. If an open loop contains at least an enemy piece inside, then it is called open loop with enemy pieces or cycle (figure-5c). Consequently, if an open loop contains at least an empty piece inside, it is called an open loop with empty pieces. There is an empty open loop, which can be seen at figure-5a. However, if the center of an open loop contains a friendly piece, then it is called full loop (figure-5b). The classification of the loop is shown in a tree in figure-6.



(a) In hex. (b) In rhombitrihexagonal. (c) In square.

Figure 5: Loop in different types of game board.

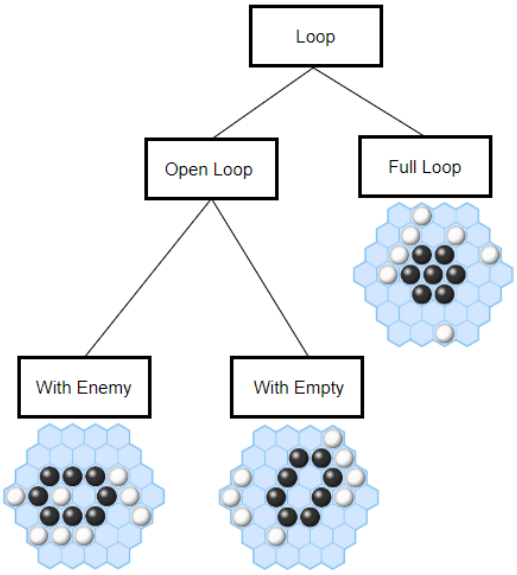
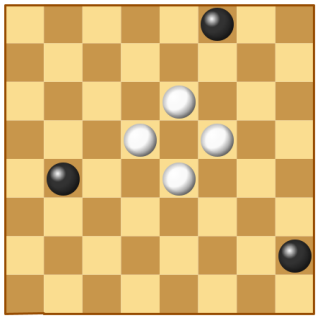
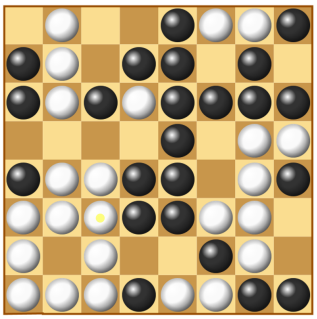


Figure 6: The classification of a loop in hex game board (all adjacent connection).

Importantly, the sizes, shapes, and the minimum number of the component to create a loop also depends on the type of connectivity. Figure-3 shows some examples of the different kinds of loops:

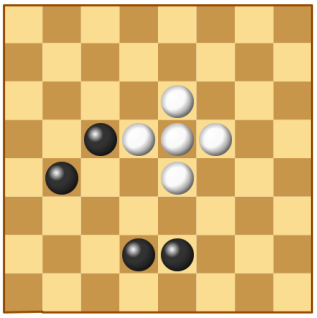


(a) All adjacent connection.

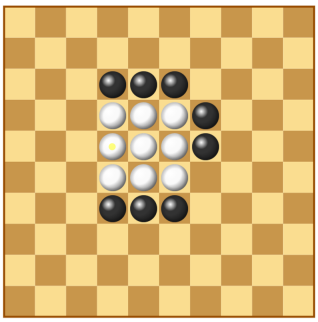


(b) Orthogonal connection.

Figure 7: Open loop (white) in square game board.

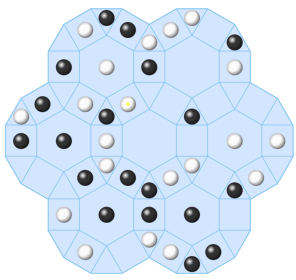


(a) All adjacent connection.

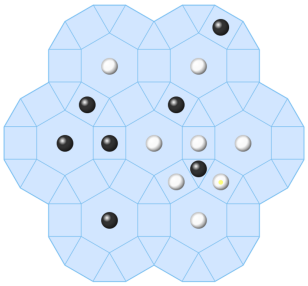


(b) Orthogonal connection.

Figure 8: full loop (white) in square game board.

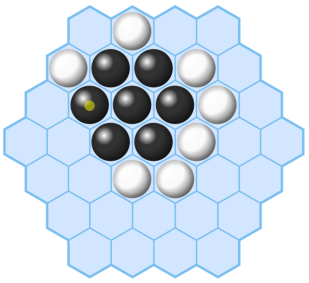


(a) All adjacent connection.

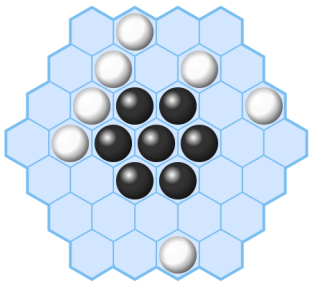


(b) Orthogonal connection.

Figure 9: open loop with enemy (white) in rhombitrihexagonal grid.



(a) All adjacent connection.



(b) Orthogonal connection.

Figure 10: Full-ring (black) in hex game board.

Mathematical Explanation

The game board can be described by an undirected and unweighted graph $G = (V, E)$, where, V is the set of vertices (in games, a vertex means a cell, which can hold a game stone) and $E \subseteq V \times V$, which is the set of edges. Assume that the last move in a game state is v, where the degree of v is k (where, $12 \geq k \geq 4$).

In this case, before adding v in any union tree, if all the adjacent friendly stones make more than one disjoint group and at last two disjoint sets are the member of the same union tree, then there is at least one open loop. Notably, to calculate the number of adjacent disjoint sets, the type of connectivity is the same as the game connectivity. Moreover, each set needs to contain at least one element related to v, which has the same connectivity relation. However, for the full loop around the last move v, all the adjacent friendly cells needs to be a connected open loop.

There is a special case for open loop in a game graph. In a game state for the last move v, if there is an adjacent vertex is u, where u contains a orthogonal degree k and $k = 3$. Then the previously mentioned rule of the open loop detection is not applicable for all adjacent direction connectivity. For example, if $dirnChoice$ = all directions, then only 3 friendly pieces can create an open loop around u (which can be seen at figure-9a). In this situation, we check all the orthogonal cells of u.

Common properties of a loop:

1. A loop is a connected component.
2. The *graph diameter* of a loop is at least 3.
3. The lastmove (i.e., v) is always on the circumference or in the center of the loop.
4. The lastmove (i.e., v) can make more than one loops (which is proven in Theorem-3).
5. Creation of an open loop v needs at least two disjoint sets of friendly neighbors, where at least one stone in each set has the same connectivity relation with v. Here, the connectivity means the specific connection of that game.

6. The last move v can create a full loop if all the adjacent friendly stones make an open loop or last move v helps to make an open loop, which is around any other friendly adjacent stone.

This ludeme present works with Andantino, Havannah, and Kensington havannah. This ludeme has some optional parameters (i.e., *all*, *enemy*, *empty*, and *dirnChoice*), which are describe in the following:

- *all* is defined to detect any types of loop, which is also set as a default type.
- *enemy* is used to return a loop with at least one enemy cell inside.
- *empty* determines a loop with at least one free cell inside.
- *dirnChoice* which can be any types of direction, such as all, orthogonal, diagonal, etc.

Algorithm

The algorithm divided into two separate boolean ludemes: 1. isLoopAux, and 2. isLoop for the purpose of loop detection. The isLoopAux is helped as an auxiliary part of isLoop. isLoopAux executes before the last move v add in the union tree. The isLoopAux ensures the possibility of an open loop in a single flag, which information saves into a boolean flag called ringFlag. In the second or the main algorithm uses the information about ringFlag to detect the loop. Here are two algorithms:

Algorithm 11 isLoopAux()

```
1: Set: neighbourList = adjacent items of last move.
2: if( dirnChoice == All)
3: {
4:     for (i = 0; i <= neighbourList.size(); i++)
5:         Apply : localUnion(neighbourList[i])
6:     for (i = 0; i <= neighbourList.size(); i++)
7:     {
8:         if( neighbourList[i] is a root) in localUnion
9:         {
10:            Set : RootI = neighbourList[i]
11:            for (j = i + 1; j <= neighbourList.size(); j++)
12:            {
13:                if( neighbourList[j] is a root) in localUnion
14:                {
15:                    Set : RootJ = neighbourList[j]
16:                    if : (RootI == RootJ) return true
17:                }
18:            }
19:        }
20:    }
21: }
22: if( dirnChoice == Orthogonal)
23: {
24:    for (i = 0; i <= neighbourList.size(); i++)
25:    {
26:        Apply : localUnion(neighbourList[i])
27:        Condition : atleast each set contains a orthogonal items with v
28:    }
29:    for (i = 0; i <= neighbourList.size(); i++)
30:    {
31:        if( neighbourList[i] is a root) in localUnion
32:        {
33:            Set : RootI = neighbourList[i]
34:            for (j = i + 1; j <= neighbourList.size(); j++)
35:            {
36:                if( neighbourList[j] is a root) in localUnion
37:                {
38:                    Set : RootJ = neighbourList[j]
39:                    if : (RootI == RootJ) return true
40:                }
41:            }
42:        }
43:    }
44: }
45: return false
```

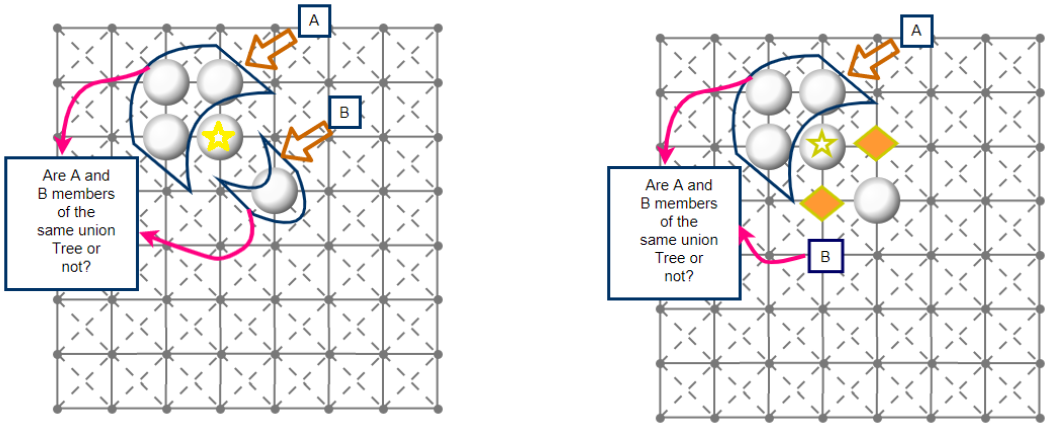
Algorithm 12 isLoop (*Role*, *dirnChoice*)

```
1: Set flag : fullring, enemy, empty, all, and defalt
2: Set : v = last move
3: Set : ringFlag = isLoopAux()
4: Union(v)
5:
6: if((allRing||default) && (ringFlag))
7:     return true
8:
9: if( dirnChoice == All)
10: {
11:     If there is an adjacent cell of v has degree 3
12:     Check the minimum open loop around it
13:     if(open − ring)
14:     {
15:         if(!(enemy||empty||fullring))
16:             return true
17:         if(enemy)
18:             if(checkInsideEnemy) return true
19:         if(empty)
20:             if(checkInsideEmpty) return true
21:     }
22: }
23:
24: for (i = 0; i <= neighbourList.size(); i + +)
25:     Apply : localUnion(neighbourList[i])
26:
27: Set : adjacentSetsnumber = disjoint Set in LocalUnion
28:
29: if(((adjacentSetsnumber > 1)&&(dirnChoice == All))
30: || ((adjacentSetsnumber > 0)&&(dirnChoice == Orthogonal)))
31: {
32:     if(enemy&&ringFlag)
33:         if(checkInsideEnemyWithDFS) return true
34:
35:     if(empty&&ringFlag)
36:         if(checkInsideEmptyWithDFS) return true
37:
38: }
```

```
39: if(fullring||all||default)
40: {
41:     i(adjacentSetsnumber > 1)
42:     {
43:     if(CheckOpenLoop(center(v)))
44:         return true
45:     if(CheckOpenLoop(circumference(v)))
46:         return true
47:     }
48: }
49: return false
```

Explanation

isLoop detects a loop. Users can specify type of loop to be used in game modeling. The loop can be any type, and the default value is all.



(a) (b)

Figure 11: Function of isLoopAux.

The isLoopAux is the auxiliary part of the implementation, which executes before the union operation execution. It identifies at least two disjoint sets in the *neighbourList* of the last move v, which are the members of the same union tree. This function is an essential part of detecting an open loop. However, each of the disjoint sets must have at least one item, which has the same connectivity relation with v. In figure-11a (which are the game graph of a square board), if disjoint sets A and B are the members of the same union tree, then there is at least one open loop (all adjacent connectivity). However, in the same figure, if we consider the orthogonal connection, then the position of A and B is not sufficient enough to make an open loop, because the last move v and B has no orthogonal relation. In this case, to create an open loop, at least one item of set B is in the orthogonal position of v (which can be shown in figure-11b with orange box).

This task is the primary function of the isLoopAux. The possibility of the open loop is returned to the isLoop via *union* operation by a flag called ringFlag.

As mentioned before, isLoop executes after last move v is added into a union tree and receives the output from isLoopAux. If a game needs open loop and if ringFlag is true, then returns true. However, if a user wants a open loop with enemy, then a DFS search helps to detect inside enemy piece. A similar purpose is applicable for the loop with empty cell. This ludeme is used for the game with the board and the board less games. For example, Andantino can be played as a boardless game. In a boardless game, the DFS iteration is controlled by a parameter-called dfsItr. The value of the dfsItr is twice the group size of an open loop. Because the DFS depth of the inside area of an open loop never exceeds two times the number of items in an open loop.

For the full loop detection, there are two possibilities. Firstly, the last move v can be at the center of a full loop, or secondly, the last move v is on the circumference of a full loop. So, if all the adjacent friendly cells (of last move v) have formed an open loop with corresponding connectivity, then it is a full loop. Similarly, the last move v can be on the circumference of a full loop; in this case, there is at least one adjacent vertex, which has an open loop at its adjoining list.

There is a particular case for triangular cell (which has 3 orthogonal adjacent vertices). To detect a minimum open loop around a triangular cell with all connectivity, the information from the isLoopAux is not helpful. For that situation, if there is an adjacent cell that is triangular, then we check all the orthogonal cells of that triangular cell for friendly items. If there is an open loop, then it returns true. Similarly, we use the same procedure to detect the enemy piece or empty cell inside of an open loop.

6.3 GroupSizeProduct(*playerType*)

This ludeme is used at the end of a game. It helps to calculate the product of all existing group size. This ludeme works with Omega.

Algorithm 13 groupSizeProduct (*playerType*)

```
1:  Set : value = 1
2:
3:  for each cell i in game board
4:      Apply : union(i, playerType)
5:
6:  for each cell i in game board
7:  {
8:      if (i == root of any union tree of playerType)
9:      {
10:         value = value * groupSize(i, playerType)
11:     }
12: }
13:
14: return value
```

Explanation

This ludeme is used as an end function. In the last game state, we initially add all the items in the *union find* data structure. Then we use the groupSize operation to obtain the size of each of the union trees and multiply all the sizes. Finally, this function returns the total value of the multiplication.

6.4 GroupCount(*playerType*, *min*)

This ludeme is used at the final game state for each player. There are 2 parameters: one is the *playerType* and the other is *min*. The *min* is an integer value, which is used to find out the more extensive groups than that. *min* is an optional parameter, and the default value is 0. Currently in Ludii, Odd is used with groupCount.

Algorithm 14 groupCount (*playerType*, *min*)

```
1:  Set : count = 0
2:
3:  if : min == null
4:      min = 0
5:
6:  for each cell i in game board
7:      Apply : union(i, playerType)
8:
9:  for each cell i in game board
10:     if (i == root of any union tree of playerType)
11:     {
12:         if (groupSize(root, playerType) > min)
13:         {
14:         count + = 1
15:         }
16:     }
17:
18: return value
```

Explanation

Initially, we create a union tree for the specific player (i.e., *playerType*). After that, the count variable is used to calculate how many union trees are sizeable than the *min* value and return the value of count.

6.5 GroupSize(*site*, *dirnChoice*)

After executing *union* operation, this ludeme returns to the group size of last move v. There are two parameters: one is the last move v, and the other is *dirnChoice*, which is the specific direction of the union tree. GroupSize works with Manalatha.

Algorithm 15 groupSize (v, *dirnChoice*)

```
1: if : v == null
2:       Set : v = last move
3:
4: Apply : union(v, playerType)
5:
6: return groupSize(v, playerType)
```

Explanation

This ludeme returns the group size of the last move v.

6.6 IsSingleGroup(*playerType*)

This ludeme returns a boolean value. If all the existing game stones of any player form a single group, then this operation returns true.

Algorithm 16 isSingleGroup(*playerType*)

```
1: Set : v = last move
2: Set : group = 0
3: Apply : union(v, playerType)
4: for each cell i in game board
5: {
6:       if (i == root of any union tree)
7:       {
8:             group++
9:       }
10: }
11: return (group == 1)
```

Explanation

This function recognizes the player, who create a single group. This operation works with Line of Action, and Groups.

6.7 Freedom()

This ludeme returns the number of freedom of any position. Freedom is the empty adjacent orthogonal position for a group of game stones. This ludeme works with Go, Arati-Go, and Pentalath.

Algorithm 17 freedom()

```
1: Crate : nList = all orthogonal positions
2:
3: if (all orthogonal positions have no game stone)
4:     return total number of orthogonal positions
5:
6: Crate : opponentAllBitset
7: Crate : sameAllBitset
8: Crate : nBitset
9:
10: for each cell i in nlist
11: {
12:     if (i == friendly piece)
13:     {
14:         set : nRoot = find (nList[i], playerType)
15:         sameAllBitset.or(itemsList(nRoot, playerType))
16:         nBitset.or(ItemWithOrthoNeighbors(nRoot, playerType))
17:     }
18: }
19:
20: Update : opponentAllBitset to store all the opponentitemList
21: Update : nBitset with all the orthogonal positions of v
22:
23: opponentAllBitset.and(nBitset)
24: nBitset.xor(opponentAllBitset)
25: nBitset.xor(sameAllBitset)
26: nBitset.clear(v)
27:
28: return nBitset.cardinality()
```

Explanation

In the proposed framework-B, there is a list-called *ItemWithOrthoNeighbors*. This list keeps the information of the items, and all adjacent orthogonal positions of a union tree. In each time, when *union* operation happens then, this list is updated as the same as the *ItemLists*. This list of information helps to calculate the freedom of any position with some boolean operations.

The calculation of freedom of any position v are 2 types. The simple one is if there is no friendly adjacent neighbor in the orthogonal position of the v (here, v is the specific position, which is used to calculate freedom). The other type is if there is at least one friendly neighbor at the direct location of v. In the first case, if there are no friendly pieces, then the number of empty orthogonal adjacent cells is the freedom of that position. However, for the second case, if there is at least one friendly piece in the orthogonal neighbor's position, then we create 3 lists. The first list is *sameAllAdBitset*, which is used to merge all the items of orthogonal adjacent friendly groups. The second list is*nBitset*, which contains all *itemsWithOrthogonalNeighbors* information of orthogonal adjacent friendly groups and all the orthogonal position of v. The last list is *opponentAllBitset*, which contains all the existing opponent's game stones information.

Initially, calculate all the overlapping positions of the *opponentAllBitset* and *nBitset* lists and store the overlapping position into *opponentAllBitset*. At that time, the *opponentAllBitset* contains all the opponent items list, which are in the orthogonal adjacent position of the group of v. Next update the *nBitset* to subtract all the items of the *opponentAllBitset* and *sameAllAdBitset* lists. As the *nBitset* initialised with all the items list of the group of v and their direct positions. So, after substracting, the *opponentAllBitset* and *sameAllAdBitset* lists from it, the *nBitset* contains all the freedom position and v. Finally, subtract the v from the *nBitset* list. As a result, the elements of the *nBitset* list are the freedom of v at that game state. This algorithm returns the number of freedom of the position v.

Importantly, this function works with *checkmove* operation. Here, we have not added v at any union tree, because to add an item and then immediately delete the same item would decrease the functional performance of a game. However, we have used all the necessary information from the data structure.

6.8 Enclosed()

This ludeme returns a legal move of the current player and helps to capture the adjacent opponent's group, which have no freedom. This algorithm is working with Go,and Pentalath.

Explanation

This operation captures the opponent adjacent groups, which have only one freedom. Here, the single freedom is important because, if we apply the current move, then that opponent group's freedom will be zero. At the starting of the algorithm, we set *sameAllBitset* list, which contains all the current player's friendly stones. Then check all the orthogonal neighbor's position of v to search the nonfriendly group with one freedom. For that reason, first, copy all the information of *itemsWithOrthogonalNeighbors* from all the adjacent union tree into a list, which is *numBit* list. Then subtract all the items of the opponent adjacent group from the *numBit* (where, $numBitset.xor(itemsList)$). At that point, *numBit* contains all the direct orthogonal positions of that opponent group. In this *numBit* list, if there any friendly game stones, then subtract them from *numBit*

Algorithm 18 enclosed()

```
1: Create : sameAllBitset to store all the existing friendly itemList
2:
3: for each cell i in valid neighborsList
4: {
5:     Set : surrounded == false
6:     if (neighbourList[i] is enemy pieces)
7:         Set : pieceundertheret = neighbourList[i],
8:
9:     Set : root = find (pieceundertheret, opponentplayerType)
10:    numBitset = itemWithOrthogonalNeighbours (pieceundertheret, opponentplayerType)
11:    numBitset.xor( itemsList (pieceundertheret, opponentplayerType))
12:    Set : tempBitset = numBitset
13:    tempBitset.and(sameAllBitset)
14:    numBitset.xor(tempBitset)
15:
16:    if(|numBitset| == 1)
17:        surrounded == true
18:
19:        if(surrounded)
20:        {
21:            deletion(all items of opponentplayerType)
22:        }
23: }
24: return moves
```

list. Now, the *numBit* list contains the total freedom of an opponent group. If the size of *numBit* is one, then the *enclosed* operation helps to remove that opponent group. So, for a move more that one opponent group can be removed from the game board. All the group items deletion can be done by the *deletion* operation.

6.9 SizeTerritory()

This ludeme is used to calculate the end score of the territory based games. It returns the total number of territories of each player. This operation is based on the local *union find*, which means this operation is not directly connected with framework-A or framework-B. This ludeme works with Go, and Pentalath.

Algorithm 19 sizeTerritory(role)

```
1: Initiallize : Array localParent as Unused with the boardSize
2: Initiallize : bitsetList localItemWithOrth as with the boardSize
3: Initiallize : Array rank as 0 with the boardSize
4: for each cell i in gameBoard
5: {
6:     if (cell[i] is free)
7:     Add : localunionRank(cell[i])
8:     Add : allOrthogonal position and (cell[i]) in localItemWithOrth List
9: }
10: Set : count == 0
11: for each unionTree i in gameBoard
12: {
13:     if there if any enemy piece in the localItemWithOrth List
14:         continue
15:     else
16:     {
17:         count + = emptyGroupsize
18:     }
19: }
20: return count
```

Explanation

In the territory based games, we need to calculate the total territories of each player. The territory is the empty space, which is enclosed by the friendly player's game stone. For that reason, we have calculated all the empty group in the game board at the final game state. In this function, we have used local *union find* with rank. There is a list (which called *localItemWithOrth*), which keeps all the information on the items list and their orthogonal neighbors. At the end of the game, when there is no valid move for the current player, then this function starts executing.

All the parameters (such as, *localParent*, *localItemWithOrth* list) are initialized with the same size as the board size. Then apply local union to make the group for all the empty space in the last game state. After adding all the empty space in the local union, group information is check. If there is a group which is surrounded by a friendly player's game stones, then calculate the number of empty cells using a *count* variable. However, if any opponent player's stone encloses an empty group, then ignore that group.

We use a separate UF structure for this implementation, as this operation is always executed at the end state of a game. If we use the proposed framework for this function, then all related information needs to be update (i.e., *union* and *deletion*) in each of the game states. As a result, the performance of a game would decline. So, to keep the total procedure efficient and straightforward, we keep this operation separate from the proposed framework.

7 Theoretical Analysis

In framework-B, the *makeset* operation performs in $\mathcal{O}(n^2)$ time, where n is the size of the game graph. In this operation, we initialize the game board with empty cells and each cell has two null lists. Furthermore, for the implementation of the data structure, we have used bitset [5] to make both lists. As a result, if the board size is huge (such as 41×41 in case of boardless games in ludii), the framework can perform properly.

In the union operation, we have used the WQUPC. It is the properties of the Weighted Quick Union, that when two different sized union tree merges, then the smaller tree's root is the child of the other tree's root. However, if both trees are of the same size, then arbitrarily choose one root as a parent of another root.

In this implementation, we initialize the universe as empty. So, when each game stone is placed on the game board, then the algorithm searches for friendly game stone in the valid neighbors. The valid neighbors depend on the direction of the connectivity (it can be orthogonal, diagonal, or all the adjacent cells). This connection relates to the specific game, where we use any related ludeme.

In a *union* operation, two cases can appear, firstly, the last move v may not have friendly stones in valid positions, or it can happen that there is one or more friendly game stone in its valid area. In the first case, the last move v makes a singleton with two lists. The first list is the *ItemLists*, which contains only one item, and the second list contains all the orthogonal cells of v. The second list is essential for territory games. In another case, it is possible that there is one or more friendly game stone in its valid position. Then we need to merge all the neighbors union trees into a single union tree.

To maintain the two lists in each position is very costly. So, at the union operation, when two lists merge into the root's list. Then we clear the child lists. As a result, practically, in any game state, if there is t number of union tree, then we need $2 \times t$ lists. This optimization technique makes the *union find deletion* efficient for connection games.

In this study, we are interested in designing a common framework for connection games, so we have tested the proposed framework into the Quick Union (QU), Ranked Quick Union with Path Compression (RQUPC), and Weighted Quick Union with Path Compression (WQUPC). In the application level, we have found the WQUPC works better than the others, which is discussed in the Experiments section. There are two primary differences between applying the *union find* in connection games and general applications. Firstly, in the graph, we add an edge to perform *union* operation between two vertices. However, in connection games, we add a vertex (i.e., the cell with game stone) and merge it with all the existing neighbors union trees. Secondly, in each of the *union* operation in the graph, we merge at most two union trees. Nevertheless, in connection game applications, one game stone merges $\lfloor k/2 \rfloor$ union trees (where k is the degree of any cell in a game board and $3 \leq k \leq 12$ [Bro05]).

As mentioned before, in the proposed *deletion* operation, we are able to delete more than one item together. So, for some connection games, especially where a group of stones can be capture, it is a great option. The *deletion* operation has some parameters, which are delete id v, enemy, and group. The first one is v, which is the id used to referred to

[5] BitSet is a class defined in the java.

delete item or items. The next one is enemy, which is a boolean flag, if the flag enemy is false, then we delete an item from the current player union tree. Otherwise, the item will delete from the next player's union tree. In most of the multiplayer games, the next player is the opponent of the current player. The last parameter is the group, which is the boolean flag used to indicate group of items that will be deleted. In the experiment time, we use a single item delete option. Currently, there is no interface function in the ludii system, which is able to delete more items.

This reconstructed deletion method is an invariant of the rebuilding method, which was proposed by Tarjan et al. [Tar72]. In the rebuilding method, there is a parameter α, whose value can be two or more; it depends on the application. If there is a union tree with size $|S|$ and α is 2. In this case, when we delete first $|S/\alpha|$ (or $|S/2|$) items, then the entire tree needs to be rebuilt again. This method could be a good option for the other application cases, where the connectivity of the union tree would not change even after removing some vertices. However, in our application, if we delete one game stone from an existing union tree, then the tree may need to be split into more subtrees to preserve the connectivity. Sometimes in our applications, one game stone merges with more than two union trees (which is proven in theorem-2). So, to keep an efficient way to remove a game stone or a group of game stones from any union tree, we need to reconstruct it. Let c be a parameter, which is the size of items needed to be deleted from a union tree with size $|S|$, then we apply reconstruction for $|S| - c$ items, where $1 \leq c \leq |S|$. The best case happens in the group capturing games (such as Go, Pentalath, etc.) when we delete a large group of items from a small union tree. However, the worst-case happens when we delete one vertex from a large union tree.

Theorem-1:

In a balance [6] *connection game (without capturing stone), if the game board contains n cells with p players, then the size of any union tree for each player's at most* $\lfloor n/p \rfloor$*, where* $p > 1$ *and* $n >> p$*.*

Proof:

The board size n for a connection game is always finite. However, in most cases, a game is started with an empty state. At that point, there is no union tree existing on the game board. As a result, for any number of players, all union tree sizes are equal, which is 0.

After the first round of a game, where all the players place one (or two) game stone, the size of each union tree can be 1 (or more) for each player. So, the size of each union tree is less than $\lfloor n/p \rfloor$ as $n >> p$.

Without loss of generality, we may assume that at a game state t, in the board contains m game stones, where, $m = t \times p$ and $m << n$. Here, c is a constant integer, which uses to explain the number of stones that can be placed in one game state by each player. As we know that in a balanced game, all the players have to place an equal number of the game stones in each turn, and most of the cases, it is 1. The total numbers of each player's stones are m/p. As we know, all the stones are on the game board, so $m << n$. That is the reason $m/p < \lfloor n/p \rfloor$. In the next game state $t + 1$, the game board contains a total $m\prime$ game stones.
where,

$$m\prime = (t + 1) \times p \times c$$

$$So, m\prime = t \times p \times c + p \times c$$

$$m\prime = m + p \times c$$

For a similar reason, at $t + 1$ game state, all the game pieces are on the board. So, $m\prime$ at most n. Thus, all the game stone of each player makes 1 union tree, then the size at most $\lfloor n/p \rfloor$. If there are more union trees, then the size is less than the $\lfloor n/p \rfloor$. □

[6] Balance means each of the players uses the same number of game stone in his turn.

Theorem-2:

If the last move v in a cell in a game graph with degree k, Then union operation can merge at most $\lfloor k/2 \rfloor$ union trees. For the same reason, deletion operation can be split one union tree at most $\lfloor k/2 \rfloor$.

Proof:

The game board can be a regular shape, semi-regular shape, or irregular shape. The game graph is based on the types of game board [Bro05]. The game graph is a connected graph, which can be a planer and a non planar graph. Let, at a game state t, there is the last move at v cell, and v has a degree k (where $3 \leq k \leq 12$ [Bro05]).

Now, let l is the number of friendly game stone at the valid neighbors of last move v. If $l = 0$, which means there are no friendly stones at the valid adjacent list of v. At that point, v is a singleton union tree.

If $l = 1$ or there is only one friendly stone, say w. then v will be a member of the w union tree.

Assume that $l \leq k/2$, then all the friendly stones can be separated from each other at the adjacency list. So each can be a member of a separate union tree. When last move v is adjacent to all friendly stones, then all of them merge into a union tree. At the figure-12, we have shown that the last move v can combine some union tree in a square board. From the first picture, it can be easily understandable that the last move v can merge 4 adjacent union tree, whereas the degree of cell v is 8. However, in the second figure, there are also 4 adjacent cells contain black stone, and there are 2 union trees. The last move v combines both trees.

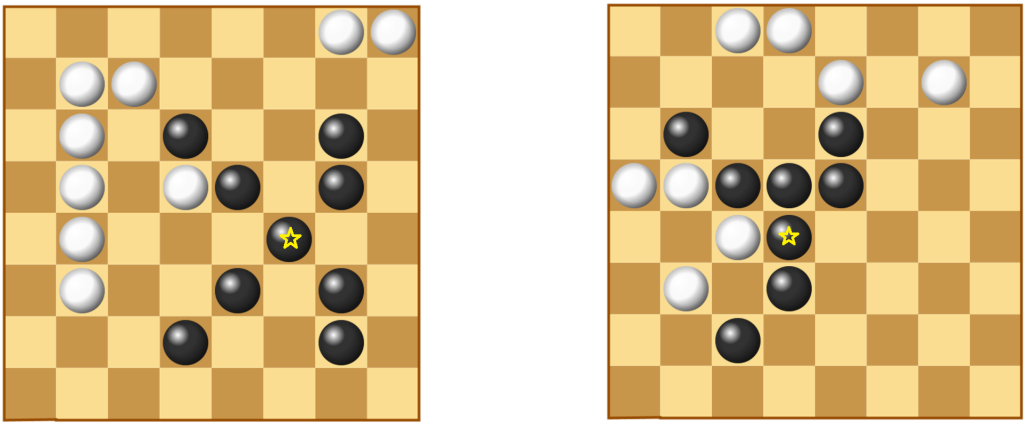


Figure 12: Last move v (with star) merge several union trees.

Without loss of generality, assume that $l > k/2$, then there the adjacent union trees can not be $\lfloor k/2 \rfloor$. The number of possible adjacent union tree is always less than $\lfloor k/2 \rfloor$. As a result, we can say, the last move v in a cell with a degree k, Then union(v) can merge at most $\lfloor k/2 \rfloor$ union trees.

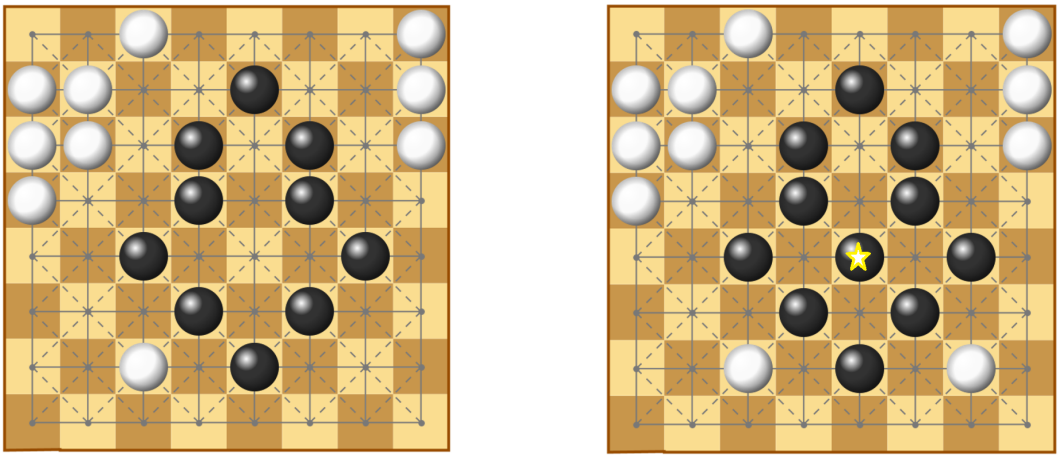
For a similar reason, if we perform the *delete* operation. Then one union tree can be divided into several sub-trees and the total number of sub-trees at most $\lfloor k/2 \rfloor$. □

Theorem-3:

In a game state, there are at most $\lfloor k/2 \rfloor - 1$ *new open loops possible, where* k *is the degree of the cell of last move* v*.*

Proof:

An open loop is a connected component with some friendly game stones, where there is at least a middle cell, which is not occupied by any friendly game stone.



(a) One open loop. (b) One open loop divide into 4.

Figure 13: Last move (with star) makes 3 new open loops.

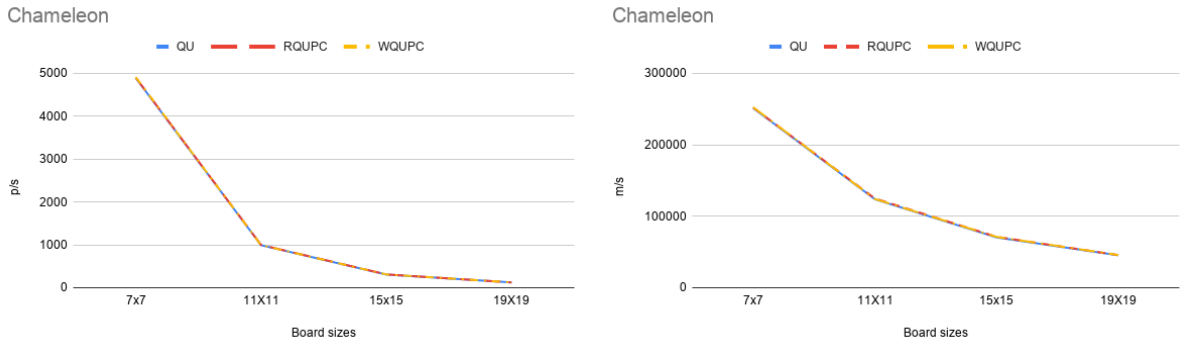
From theorem-2, it is clear that one game stone can merge at most $\lfloor k/2 \rfloor$ union trees in connection games. So, in the adjacent list of the v also has at most $\lfloor k/2 \rfloor$ disjoint cells that are occupied by friendly stones. There are at least has one member in each of the disjoint sets, which protects the connectivity properties with v. If two of the adjacent disjoint sets are the member of the same union tree, then it creates is an open loop. If all of the disjoint sets are members of the same union tree, then there are $\lfloor k/2 \rfloor$ of open loops around the v. Importantly, in the figure-13a, which is the previous game states of figure-13b (i.e., before arriving v), there is a loop with all the items and v helps to divide that open loop into $\lfloor k/2 \rfloor$ smaller open loops. As a result, the number of open loop is increased by the v is $\lfloor k/2 \rfloor - 1$. □

8 Experimental Section

All the experiments were done in the Ludii general game system [PSS+19] and the values of the number of playouts per second (i.e., p/s) and the moves per second (i.e., m/s) were calculated.

Test-1

The purpose of Test-1 was to compare the efficiency of the Quick Union (QU), the Ranked Quick Union with Path Compression (RQUPC), and the Weighted Quick Union with Path Compression (WQUPC) for connection games. For that purpose, we selected framework-A with two different games, which were Chameleon, and Hex. We selected Chemelon and Hex for the test because those use isconnect ludeme. The isconnect ludeme relates to union find framework. Both games are used the same types of board, and the number of players is equal. However, Chameleon uses one more ludeme (i.e., line).



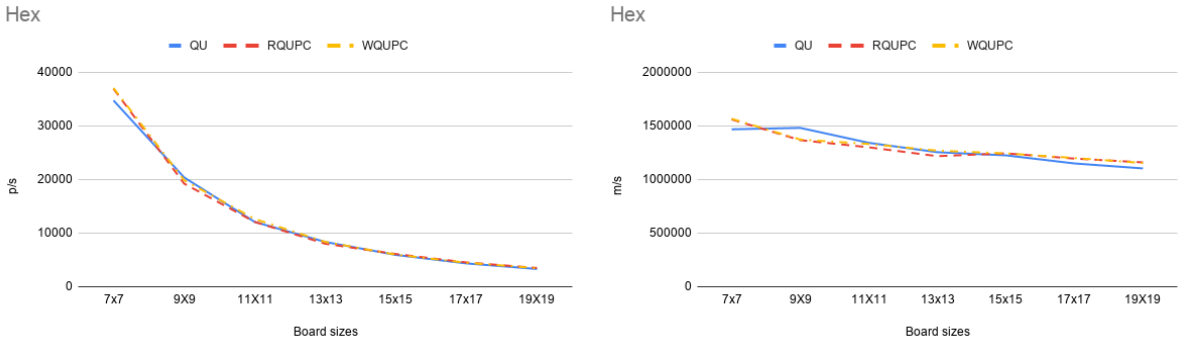
(a) Playouts per second (p/s).

(b) Moves per second (m/s).

Figure 14: Three different UF methods in Chameleon with different board sizes.

The figure-14 shows that the performance of all selected UF in p/s and m/s for 4 types of Chameleon (the board size are 7×7, 11×11, 15×15, and 19×19). Here, all three methods performed almost closer to each other.

For Hex, we selected odd-sized hex board ranging from size 7 to 19. In figure-15, the performce of QU varies than the other two methods on the smaller boards. However, when the board size is medium and large than for the p/s are the same for all. However, on larger board, the performances of QU declined. The performance of $RQUPC$ and $WQUPC$ are almost similar in all the tests.

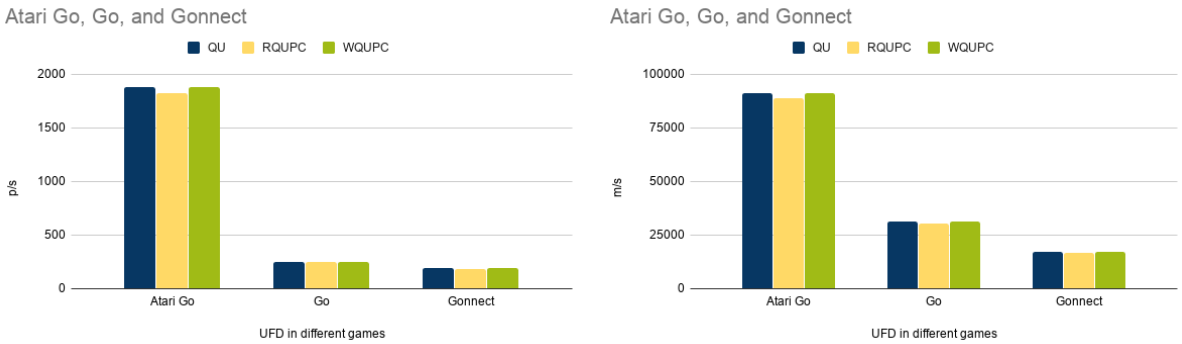


(a) Playouts per second (p/s). (b) Moves per second (m/s).

Figure 15: Three different UF methods in Hex with different board sizes.

Test-2

The correlation between all UF methods was tested using framework-B (i.e., UFD). This experiment was done with a board size 9×9 for games Atari Go, Go, and Gonnect. We selected those three games as all of them use some common ludemes (such as. freedom and enclosed). Moreover, all the games are used square shape game board and the game stone places in the intersection of the board lines.



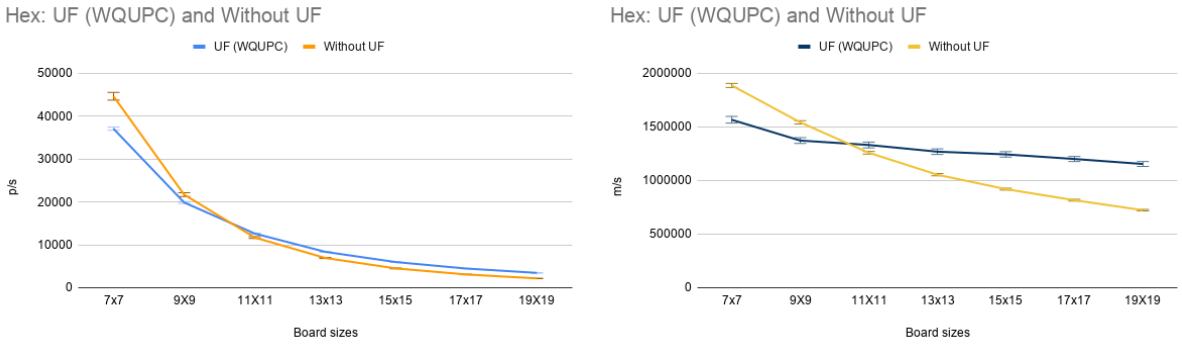
(a) Playouts per second (p/s). (b) Moves per second (m/s).

Figure 16: Three UF methods in framework-B with the same board sizes.

It can be shown in figure-16, the *WQUPC* performed a bit better than the *RQUPC* and the *QU* as the standard deviation of the data is very low. Here, all the selected games have different game rules; however, all of them use orthogonal connectivity. The Arati-Go is not a capturing game, so the p/s is higher than the other two games. Nevertheless, Go, and Gonnect have capturing facility, so each of the intersection (i.e., the position of a game stone) can be used more than one times, and *union* and *deletion* operations are frequently used in both games. Thus, p/s and m/s in Go and Gonnect are lower than Atari go.

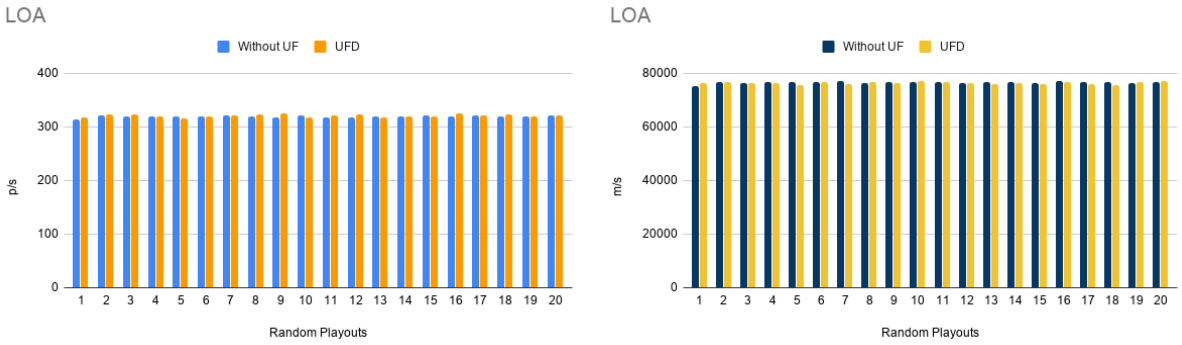
Test-3

The purpose of Test-3 was to compare the existing brute force methods with the UF frameworks. We analysed two separate tests during the experimental period. The first one was a brute force method and isConnect (in UF framework) with the game Hex. In the second test, we chose Line of Action with UFD. In each game state of line of Action, one game stone moves from one cell to another cell. So, the union and the deletion operation execute at least one time per game state. The ludeme isSingleGroup used to detect the winning strategy. However, a brute force algorithm checked all the game stones were in the same group or not.



(a) Playouts per second (p/s). (b) Moves per second (m/s).

Figure 17: Brute force method Vs Framework-A (UF).



(a) Playouts per second (p/s). (b) Moves per second (m/s).

Figure 18: Brute force method Vs Framework-B (UFD) with Line Of Action.

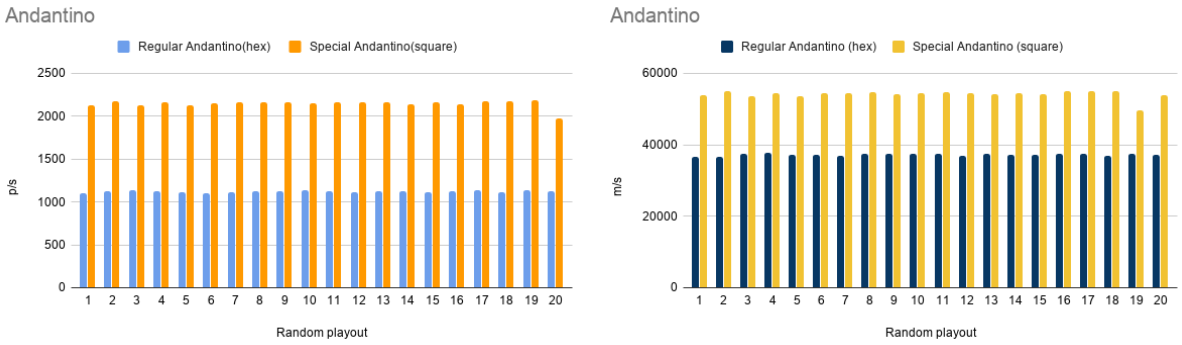
From figure-17a, it is apparent that in the smaller board size (7×7, and 9×9) the brute force method worked well than the UF. Ludii is a massive game general system, which is for board games, card games, dice games, mathematical games, simple video games, and so on. This system follows a state-based mechanism. It means each of the game state copy data from the previous state. For the smaller board size of Hex, to copy the games state in each time might reduce the performance of the UF. Nevertheless, in the

medium and the large game board sizes, the performance of the UF is better than the brute force method. Notably, if the game board size increased, then the m/s of the UF slowly declined (figure-17b). However, in that case, the m/s of the brute force method sharply dropped. It happens as UF took the almost same time to connect any piece to the union tree, which does not depend on the board size.

In the Line of Action, the performances of the UFD are same as the without UF (figure-18). In each game state, when a game stone moves one cell to other, then here reconstruct one of the existing union trees. Most of the time then the average size of the union tree could be 8 to 10 (in common player union tree) or less (in player's union tree). In our implementation, the UFD has a robust optimisation technique. So, the performance of the framework-B was almost the same as the brute force method.

Test-4

The average performance of Andantino with hex and square versions were compared by using framework-A. Andantino is generally played with hex pieces. Here, we have named it hex version. However, for the test, we redesigned the Andantino with the same game rules using square pieces, which is celled square version.



(a) Playouts per second (p/s).

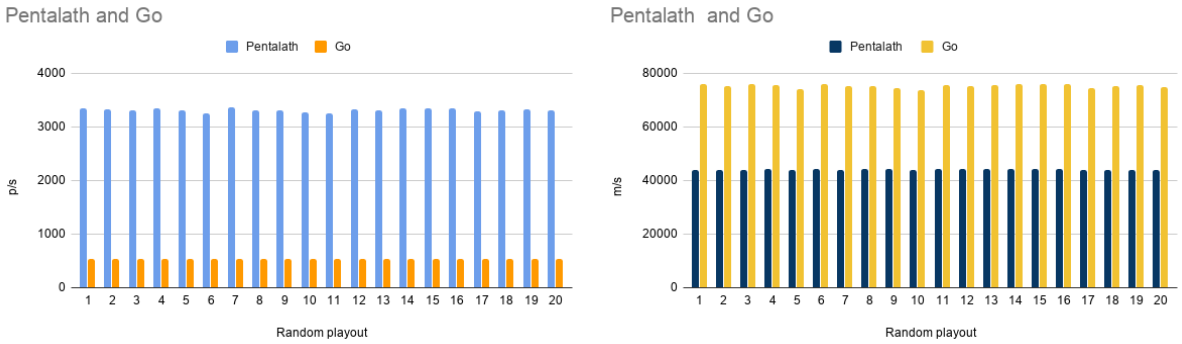
(b) Moves per second (m/s).

Figure 19: Andantino with different types of cells.

The game rules and the connectivity (i.e., all adjacent connection) of both versions of Andantino were the same. Andantino is originally a boardless game. However, in Ludii game general system uses 41×41 invisible board for both versions. The differences between hex and square connectivity in large graph are highlighted in figure-21a and figure-21b. In both cases, the square Andantino performed better than the hex Andantino. The winning strategy of Andantino is to make a cycle or make a line of 5. Our framework was connected with cycle detection. It is possible that in a square grid (where a cell with 8 adjacent cells) making a connection is quicker than in a hex grid.

Test-5

To compare the differences between hex and square cells with orthogonal connection, the games Pentalath and Go with framework-B were selected. For this test, we used Pentalath with base-5 (i.e., total cells 61) and Go with base-8 (i.e., total cells 64) with WQUPC.

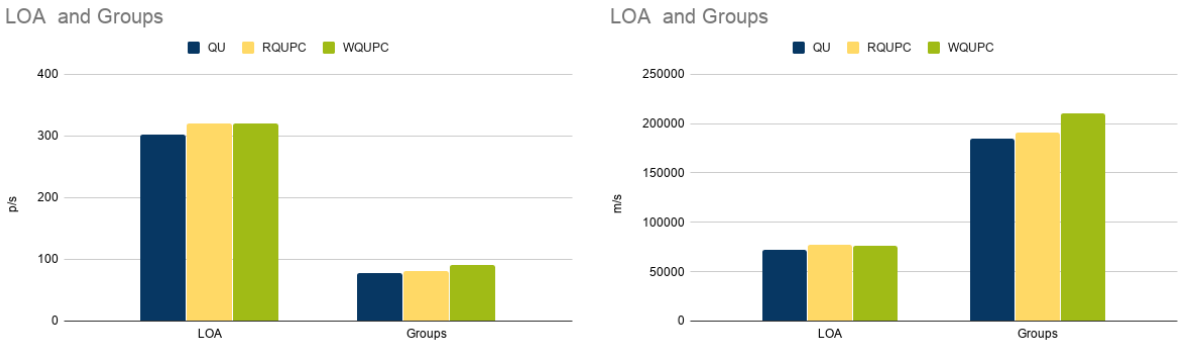


(a) Playouts per second (p/s). (b) Moves per second (m/s).

Figure 20: Pentalath and Go in Framework-B.

Both games have a plethora of similarities. However, Pentalath has an additional winning condition. The extra winning condition for the Pentalath is to create line 5. For this reason, Pentalath has a higher p/s than Go. Figure-20b, shows that the m/s of Go is better than Pentalath. The reason could be that for Go in any *union* operation merges at most 2 union trees. However, in Pentalath *union* operation merges at most 3 union trees.

Test-6



(a) Playouts per second (p/s). (b) Moves per second (m/s).

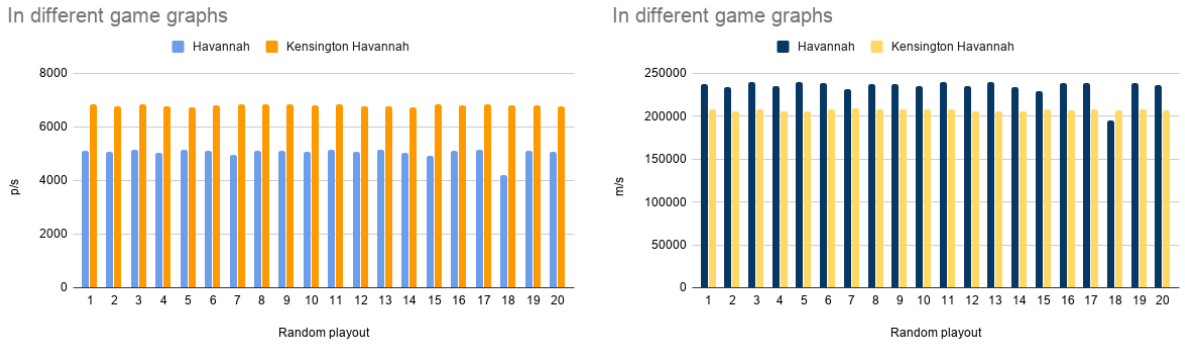
Figure 21: Line of Action and Groups in different UFD.

In order to assess the efficient UF method in framework-B, repeated-measures of the Line of Action and the Groups were used. In the Line of Action and the Groups, in each

game states the existing one game piece moves from its cell to another cell. As a result, one existing union tree needs to reconstruct in every movement, which is the worst case of the *deletion* operation. Moreover, both games have an identical winning condition with the same types and sizes of the game board. However, they have some dissimilarities, such as the connectivity, capturing facility, and the number of stones.

Figure-21 shows some of the main characteristics of the Line of Action, and the Groups in framework-B. The p/s of the Line of Action is higher than the Groups, as the Line of Action allows capturing pieces, so the number of stones decreases with the time. Moreover, this game has all adjacent connectivity, so it is faster to form a single group than the orthogonally connected game (Groups has orthogonal connectivity). However, the m/s of the Groups is higher than the Line of Action, because the Groups has a lower number of game stones than the Line of Action(i.e., where Line of Action has 12 stones, and Groups has 6 stones). Importantly, for both games, the RQUPC and WQUPC performed very carefully in p/s. Nevertheless, WQUPC performed better in m/s than the other two UF methods (figure-21b).

Test-7



(a) Playouts per second (p/s).

(b) Moves per second (m/s).

Figure 22: Framework-A in Havannah and Kensington havannah.

Test-7 was used to analyze the relationship between the different game graphs with the same size (i.e., total cells 61). We selected Havannah, and Kensington havannah [7]. Both games have the same winning conditions and the same connectivity.

There was a significant difference between the p/s and the m/s of both games. Kensington havannah showed a higher p/s than havannah (which can be shown at figure-22a). As, each player can win to make an open loop with 3 game stones, whereas for Havannah needs at least 6. However, in another figure, the m/s in Kensington havannah is lower than Havannah. Because in Kensington havannah, there are some cells with degree 3, so it needs time to check the particular case of a loop.

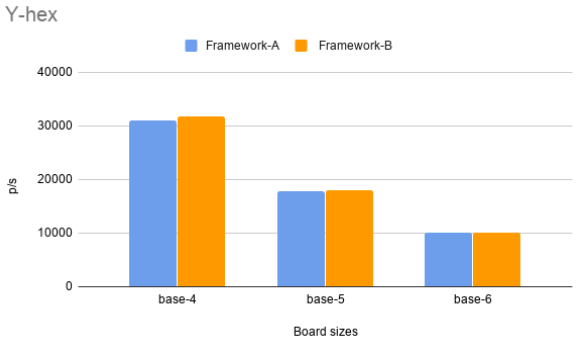
[7]It is a new game, which is invented by Cameron Browne, 2020.

Test-8

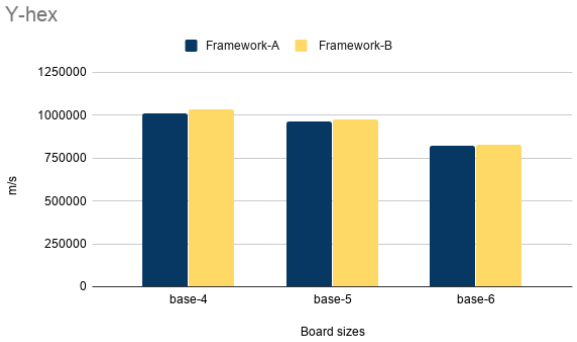
The last research question aim to find out how efficient framework-B is in comparison with framework-A for connection games. For this test, we selected Y-hex [8], Cross, and Omega with WQUPC. For Y-hex and Cross, the board sizes of 3 and 6 were used respectively. We selected both games as in each of the game state Y-hex is used isConnect one time. However, Cross is used isConnect 5 times. For Omega a standard board size of base-5 with the different number of players was used. We chose Omega as during the experiment time Omega is the only option to check the multi-players game.

Importantly, when we checked framework-A, then both frameworks existed in the Ludii game general system. Nevertheless, when framework-B was tested, then we ignored framework-A.

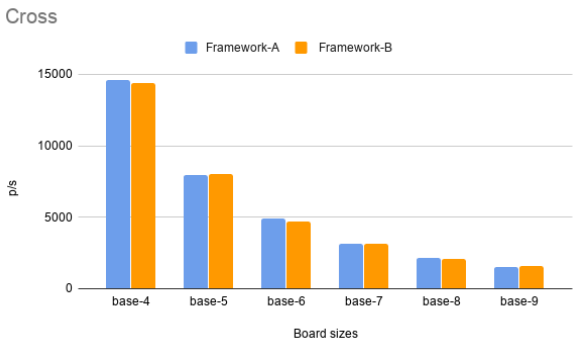
[8]This game is invented by Eric Piette, 2019. The game rules are the combination of popular games Y, and Hex.



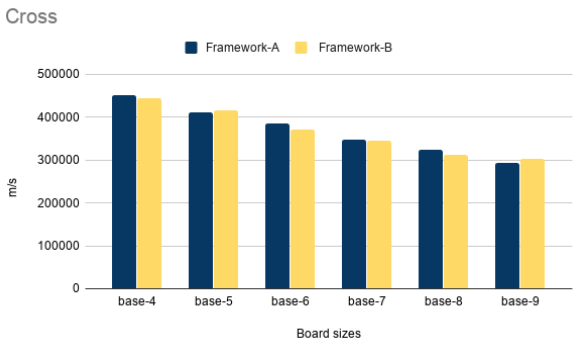
(a) In Y-hex (p/s).



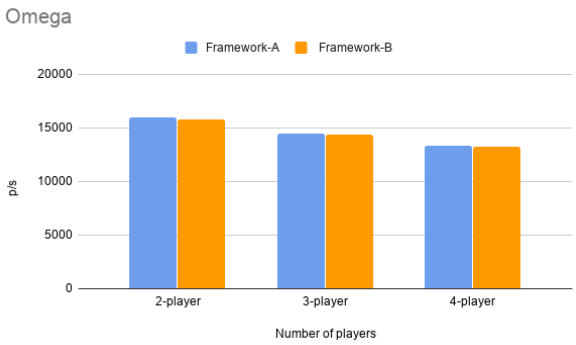
(b) In Y-hex (m/s).



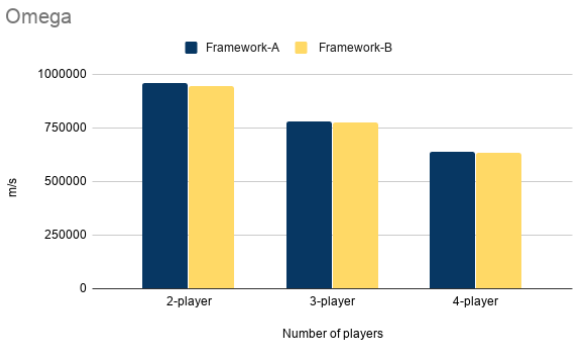
(c) In Cross (p/s).



(d) In Cross (m/s).



(e) In Omega (p/s).



(f) In Omega (m/s).

Figure 23: Framework-A Vs. Framework-B in different aspects.

The most striking result to emerge from figure-23 is that the average performance of both frameworks is the same. In the Y-hex and Cross, when the number of board size increases than p/s and m/s declined in both methods. Thus, if we repeatedly use UF related ludemes in any game modelling, then there is no significant gap between UF and UFD. In the multiplayer game, the result shows that if the number of player gain than p/s and m/s drop. As a result, we can summarize that in a general game system, we can easily replace the classical UF by the proposed method because the UFD has more applicability than the UF.

9 Discussion

Several publications have shown that the classical *union find* has been used in some 2-player connection games. However, no exact information was found regarding *union find deletion* data structure in connection games. This study set out with the aim of assessing the importance of *union find deletion* in general game system for connection games.

The current study found that *union find deletion* is practically applicable in connection games, which provides a plethora of facility to design different types of connection games in a single framework. It also helps to use a limited amount of memory in a general game system. Moreover, this new framework removes the common limitation to use the undo button in the graphical user interface.

The *union find deletion* data structure can be applied in different types of condition, such as any piece moves from one cell to another, stone captures in the capturing games, and also all the game condition, where previously classical *union find* was used. Test-3 shows that the performance of UFD is almost the same (in Line of Action) or better (in medium or large board size Hex) than the traditional brute force method. Because the implementation of UFD is designed with java bitset class and a robust optimization technique. However, for the capturing games (such as Go, Arati go, Pentalath), we have no information about any other alternative methods, which is comparable with our implementation. Moreover, the UFD is capable of operating on 2 or more player game and the different size and types of the game board. It is used to specify any connectivity direction in a game. It is also possible to use different connectivity for the various player in a single game (for example, the white player has forward slash connection, and the black player needs backwards slash connection). Additionally, UFD supports different types of game graph.

The third and most crucial question in this research was *how efficient is it to use union find deletion method in comparison with the classical union find data structure?* The results of this study indicate that in 2 or more players games, the UF and UFD have the same efficiency. Moreover, the UFD helps to integrate more games in a corresponding data structure than the UF. So, it could be possible to replace UF by UFD in Ludii game general system.

Presently, isloop is a boolean operation. However, it is possible to design this algorithm as an integer ludeme, where the new function will return the total number of the loop at the end state of a game. The properties of the integer isloop operation is explained in the Theorem-3.

Ludii is still growing general game system. During the testing period, there is a limited number of multiplayer connection game available. So, we have done only one test for that purpose (Test-8). In future, one can do the same analysis for the other types of games, which can be based on different ludeme in UFD.

In this study, UFD and UF provide an excellent result in the medium or large board size. However, in the small board size, the UF does not show a better performance than the classical brute force method (Test-3). It is possible that the UF or UFD framework would not provide better result to operate small groups, such as detect line 4 or line 5. Moreover, anyone can analysis the same framework in a minor general game system,

which is not state-based. The result could be different than our result.

10 Conclusion

The purpose of the current study was to determine the practical benefits of the *union find deletion* (UFD) data structure in connection games and to find the appropriate situations to apply it. The research has shown that it is possible to use the UFD in connection games, which gives more usability to design connection games in different aspects. The UFD data structure can be used in several cases of connection games, such as determine the number of connection, capturing pieces, piece movements, pattern recognition, number of group identification etc. The main goal of the current study was to determine the efficiency of using UFD in comparison to classical UF. These findings suggest that in Ludii game general system, the performance of both data structure are almost the same. As a result, we can use UFD instead of UF. Moreover, the UFD has al least equal effectiveness with the brute force method.

To use this framework, one user can reuse the same memory information in each game state for a single game. As a result, a game system designer needs not to consider the state base general game system. For that reason, the UFD is helpful to increase the performances of connection games.

Future works might be involved to apply the same method in different sectors, such as Wireless sensor networks, Telecommunications networks, etc. As in those areas also have some problems, which relates to *union find deletion*. The connectivity of a network grid has similar characteristics to the games board. This application is another idea worth exploring.

References

[ATG+05] Stephen Alstrup, Mikkel Thorup, Inge Li Gørtz, Theis Rauhe, and Uri Zwick. Union-find with constant time deletions. *ACM Transactions on Algorithms*, 11(1), 2005.

[Bro05] Cameron Browne. *Connection Games - Variations on a Theme*. A K Peters, 2005.

[Bro18] Cameron Browne. Mcts and deep learning, class lecture in course intelligence search and games. Maastricht University, 2018.

[BT12] Cameron Browne and Stephen Tavener. Bitwise-parallel reduction for connection tests. *IEEE Transactions on Computational Intelligence and AI in Games*, 4:112–119, 2012.

[BY11] Amir M. Ben-Amram and Simon Yoffe. A simple and efficient union-find-delete algorithm. *Theor. Comput. Sci.*, 412(4-5):487–492, 2011.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[Ewa12] Timo Ewalds. Playing and solving havannah. 2012.

[GF64] Bernard A. Galler and Michael J. Fischer. An improved equivalence algorithm. *Commun. ACM*, 7:301–303, 1964.

[KST02] Haim Kaplan, Nira Shafrir, and Robert E. Tarjan. Union-find with deletions. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 19–28, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[PSS+19] Eric Piette, Dennis JNJ Soemers, Matthew Stephenson, Chiara F Sironi, Mark HM Winands, and Cameron Browne. Ludii-the ludemic general game system. *arXiv preprint arXiv:1905.05013*, 2019.

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley Professional, 4th edition, 2011.

[Tar72] Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22:215–225, 1972.

Appendix A Framework-A

A.1 unionInfo.java

```
package util;

import java.io.Serializable;
import java.util.Arrays;
import java.util.BitSet;


/**
 * Contains all the info/storages for the Union-find.
 *
 * @author tahmina
 *
 */
public class UnionInfo implements Serializable
{
        private static final long serialVersionUID = 1L;

        protected int[][] parent;
        protected BitSet[][] itemsList;
        public int totalsize;

        /**
         * Constructor
         * @param totalVertices          The size of the game board.
         * @param numberOfPlayers        The total number of players.
         */

        public UnionInfo(final int totalVertices, final int numberOfPlayers)
        {
           parent          = new int [numberOfPlayers + 2][];
           itemsList = new BitSet[numberOfPlayers + 2][];
           totalsize = totalVertices;


           for (int i = 1; i <= numberOfPlayers + 1; i++)
           {
             parent[i] = new int[totalVertices];
             itemsList[i] = new BitSet[totalVertices];

             for (int j = 0; j < totalVertices; j++)
             {
                parent[i][j] = j;
                itemsList[i][j] = null;
```

```
    }
  }
}

public void setParent
(
                final int childIndex,
                final int parentIndex,
                final int player
)
{
        parent[player][childIndex] = parentIndex;
}

public int getParent
(
        final int childIndex,
        final int player
)
{
        return parent[player][childIndex];
}

public BitSet getItemsList
(
                final int parentIndex,
                final int player
)
{
        return itemsList[player][parentIndex];
}

public void setItem
(
                final int parentIndex,
                final int childIndex,
                final int player
)
{
        itemsList[player][parentIndex] = new BitSet(totalsize);
        itemsList[player][parentIndex].set(childIndex);
}

public void mergeItemsLists
(
                final int parentIndex1,
                final int parentIndex2,
```

```
                final int player
        )
        {
                itemsList[player][parentIndex1]
                                .or(itemsList[player][parentIndex2]);
                itemsList[player][parentIndex2].clear();
        }

        public boolean isSameGroup
        (
                final int parentIndex,
                final int childIndex,
                final int player
        )
        {

                if(itemsList[player][parentIndex] == null)
                        return false;

                return itemsList[player][parentIndex].get(childIndex);
        }

        public int getGroupSize
        (
                final int parentIndex,
                final int player
        )
        {
                if(itemsList[player][parentIndex] == null)
                        return 0;

                return itemsList[player][parentIndex].cardinality();
        }
}
```

A.2 unionFind.java

```
/**
 * Main file to create Union tree
 *
 * @author tahmina
 *
 */
public class UnionFind implements Serializable
{
  /**
   * @param siteId        The last move of the game.
   * @param state         Each state information.
   * @param uf            The object of the union-find.
   * @param whoSiteId     Player type of last move.
   * @param numPlayers    The number of players.
   * @param neighbourLis The adjacent list of our last movement.
   *
   * Remarks             None.
   *
   */
  private static void union
  (
        final int siteId,
        final ContainerState state,
        final UnionInfo uf,
        final int whoSiteId,
        final int numPlayers,
        final List<Vertex> neighbourList
  )
 {
        final int numNeighbours = neighbourList.size();

        uf.setItem(siteId, siteId, whoSiteId);
        uf.setParent(siteId, siteId, whoSiteId);

        for (int i = 0; i < numNeighbours; i++)
        {
           final int ni = neighbourList.get(i).index();
           boolean connect  = true;

           if
           (
             ((whoSiteId == numPlayers + 1) && (state.who (ni) != 0)) ||

             ((whoSiteId != numPlayers + 1) && (state.who (ni) == whoSiteId))
           )
```

```
            {
                for (int j = i + 1; j < numNeighbours; j++)
                {
                    final int nj = neighbourList.get(j).index();

                    if (connected(ni, nj, uf, whoSiteId))
                    {
                        connect = false;
                        break;
                    }
                }

                if (connect)
                {
                    final int rootP = find(ni, uf, whoSiteId);
                    final int rootQ = find(siteId, uf, whoSiteId);

                    if(rootP == rootQ)
                        return;

                    if (uf.getGroupSize(rootP, whoSiteId)
                                < uf.getGroupSize(rootQ, whoSiteId))
                    {
                        uf.setParent(rootP, rootQ, whoSiteId);
                        uf.mergeItemsLists(rootQ, rootP, whoSiteId);
                    }
                    else
                    {
                        uf.setParent(rootQ, rootP, whoSiteId);
                        uf.mergeItemsLists(rootP, rootQ, whoSiteId);
                    }
                }
            }
        }
    }
}
```

A.3 find.java

```
/**
*
* @param position   A cell number.
* @param uf         Object of union-find.
* @param whoSiteId  The current player type.
*
* @return  The root of the position.
*/
private static int find
(
        final int position,
        final UnionInfo uf,
        final int whoSiteId
)
{
   final int parent = uf.getParent(position, whoSiteId);

   if (parent == position)
      return position;
   else
      return find(uf.getParent(parent, whoSiteId), uf, whoSiteId);
}
```

A.4 connect.java

```
/**
 *
 * @param position1      Integer position.
 * @param position2      Integer position.
 * @param uf             Object of union-find.
 * @param whoSiteId      The current player type.
 *
 * @return check         Are the position1 and position1
                         in the same union tree or not?
 */
private static boolean connected
(
        final int position1,
        final int position2,
        final UnionInfo uf,
        final int whoSiteId
)
{
   final int root1 = find(position1, uf, whoSiteId);
   return uf.isSameGroup(root1, position2, whoSiteId);
}
```

Appendix B Framework-B

B.1 unionInfoD.java

```
package util;

import java.io.Serializable;
import java.util.Arrays;
import java.util.BitSet;

/**
 * Contains all the info/storages for the Union-find-delete.
 *
 * @author tahmina
 *
 *
 */
public class UnionInfoD implements Serializable
{
        private static final long serialVersionUID = 1L;
        private static int Unused = -1;

        protected int[][] parent;
        protected BitSet[][] itemsList;
        protected BitSet[][] itemWithOrthoNeighbors;
        public int totalsize;

        /**
         * Constructor
         * @param totalvertices          The size of the game board.
         * @param numberOfPlayers        The total number of players.
         */

        public UnionInfoD(final int totalVertices, final int numberOfPlayers)
        {
                parent    = new int[numberOfPlayers + 2][];
                itemsList = new BitSet[numberOfPlayers + 2][];
                itemWithOrthoNeighbors = new BitSet[numberOfPlayers + 2][];
                totalsize = totalVertices;

                for (int i = 1; i <= numberOfPlayers + 1; i++)
                {
                        parent[i] = new int[totalVertices];
                        itemsList[i] = new BitSet[totalVertices];
                        itemWithOrthoNeighbors[i] = new BitSet[totalVertices];

                        for (int j = 0; j < totalVertices; j++)
```

```
                {
                        parent[i][j]   = Unused;
                        itemsList[i][j] = null;
                        itemWithOrthoNeighbors[i][j] = null;
                }
        }
}


public void setParent
(
        final int childIndex,
        final int parentIndex,
        final int player
)
{
        parent[player][childIndex] = parentIndex;
}

public void clearParent
(
        final int childIndex,
        final int player
)
{
        parent[player][childIndex] = Unused;
}

public int getParent
(
        final int childIndex,
        final int player
)
{
        return parent[player][childIndex];
}

public BitSet getItemsList
(
        final int parentIndex,
        final int player
)
{
   if(itemsList[player][parentIndex] == null)
   {
     itemsList[player][parentIndex] = new BitSet(totalsize);
   }
```

```
    return itemsList[player][parentIndex];
}

public void clearItemsList
(
        final int parentIndex,
        final int player
)
{
    itemsList[player][parentIndex].clear();
}

public boolean isSameGroup
(
        final int parentIndex,
        final int childIndex,
        final int player
)
{
   if(itemsList[player][parentIndex] == null)
      return false;

   return itemsList[player][parentIndex].get(childIndex);
}

public void setItem
(
        final int parentIndex,
        final int childIndex,
        final int player
)
{
    if(itemsList[player][parentIndex] == null)
    {
      itemsList[player][parentIndex] = new BitSet(totalsize);
    }
     itemsList[player][parentIndex].set(childIndex);
}

public void mergeItemsLists
(
        final int parentIndex1,
        final int parentIndex2,
        final int player
)
{
```

```
    itemsList[player][parentIndex1]
                    .or(itemsList[player][parentIndex2]);
    itemsList[player][parentIndex2].clear();
}

public int getGroupSize
(
        final int parentIndex,
        final int player
)
{
    if(itemsList[player][parentIndex] == null)
        return 0;
    return itemsList[player][parentIndex].cardinality();
}

public BitSet getAllItemWithOrthoNeighbors
(
        final int parentIndex,
        final int player
)
{
   if(itemWithOrthoNeighbors[player][parentIndex] == null)
   {
       itemWithOrthoNeighbors[player][parentIndex]
                                    = new BitSet(totalsize);
   }
       return  itemWithOrthoNeighbors[player][parentIndex];
}

public void clearAllitemWithOrthoNeighbors
(
        final int parentIndex,
        final int player
)
{
     itemWithOrthoNeighbors[player][parentIndex].clear();
}

public void setItemWithOrthoNeighbors
(
        final int parentIndex,
        final int childIndex,
        final int player
)
{
   if(itemWithOrthoNeighbors[player][parentIndex] == null)
```

```
            {
              itemWithOrthoNeighbors[player][parentIndex]
                                          = new BitSet(totalsize);
            }
            itemWithOrthoNeighbors[player][parentIndex]
                                                  .set(childIndex);
        }

        public void mergeItemWithOrthoNeighbors
        (
                final int parentIndex1,
                final int parentIndex2,
                final int player
        )
        {
                itemWithOrthoNeighbors[player][parentIndex1]
                        .or(itemWithOrthoNeighbors[player][parentIndex2]);
                itemWithOrthoNeighbors[player][parentIndex2].clear();
        }

}
```

B.2 deletion.java

```
/**
*
* @param context          The current Context of the game board.
* @param deleteId         deleteId, which we want to delete from the union tree.
*/

public static void deletion
(
        final Context context,
        final int deleteId,
        final boolean enemy,
        final boolean groupDelete
)
{
   final Graph graph = context.graph();
   final int cid     = context.containerId()[deleteId];
   final ContainerState state = context.state().containerStates()[cid];
   int deletePlayer = state.who(deleteId);

   if(!groupDelete)
   {
    if(enemy)
    {
      deletePlayer = context.state().next();
    }

    final int deleteIdRoot = find(deleteId, state.unionInfoD()[deletePlayer]
                                                          , deletePlayer);
    final BitSet bitsetsDeletePlayer  = (BitSet) state.unionInfoD()
        [deletePlayer].getItemsList(deleteIdRoot, deletePlayer).clone();

    for (int i = bitsetsDeletePlayer.nextSetBit(0);
                        i >= 0;i = bitsetsDeletePlayer.nextSetBit(i + 1))
    {
        state.unionInfoD()[deletePlayer].clearParent(i, deletePlayer);
        state.unionInfoD()[deletePlayer].clearItemsList(i, deletePlayer);
        state.unionInfoD()[deletePlayer].clearAllitemWithOrthoNeighbors
                                                        (i, deletePlayer);
    }

    bitsetsDeletePlayer.clear(deleteId);

    for (int i = bitsetsDeletePlayer.nextSetBit(0);
                       i >= 0;i = bitsetsDeletePlayer.nextSetBit(i + 1))
    {
```

```
    final TIntArrayList nList;

    if (GameType.OrthogonalDeletionOnly) != 0L)
    {
       nList = validPosition(graph.vertices().get(i).orthogonal());
    }
    else
    {
       nList = validPosition(graph.vertices().get(i).adjacent());
    }

    final int nListSz = nList.size();
    final TIntArrayList neighbourList = new TIntArrayList(nListSz);

    for (int j = 0; j < nListSz; j++)
    {
       final int ni = nList.getQuick(j);
       if (state.who(ni) == deletePlayer)
       {
            neighbourList.add(ni);
       }
    }
    union(i, neighbourList, state.unionInfoD()[deletePlayer],
                                                    deletePlayer);
}

final int cPlayer = context.game().players().count() + 1;
final int delRootcPlayer = find(deleteId, state.unionInfoD()
                                              [cPlayer], cPlayer);
final BitSet itemsListDPlayer = (BitSet) state.unionInfoD()
          [cPlayer].getItemsList(delRootcPlayer, cPlayer).clone();

for (int i = itemsListDPlayer.nextSetBit(0);
                  i >= 0; i = itemsListDPlayer.nextSetBit(i + 1))
{
    state.unionInfoD()[cPlayer].clearParent(i, cPlayer);
    state.unionInfoD()[cPlayer].clearItemsList(i, cPlayer);
    state.unionInfoD()[cPlayer].clearAllitemWithOrthoNeighbors
                                                    (i, cPlayer);
}
itemsListDPlayer.clear(deleteId);

for (int i = itemsListDPlayer.nextSetBit(0); i >= 0;
                           i = itemsListDPlayer.nextSetBit(i + 1))
{
    final TIntArrayList nList;
```

```
    if (GameType.OrthogonalDeletionOnly) != 0L)
    {
        nList = validPosition(graph.vertices().get(i).orthogonal());
    }
    else
    {
        nList = validPosition(graph.vertices().get(i).adjacent());
    }

    final int nListSz = nList.size();
    final TIntArrayList neighbourList = new TIntArrayList(nListSz);

    for (int j = 0; j < nListSz; j++)
    {
            final int ni = nList.getQuick(j);
            if (state.who(ni) != 0)
            {
               neighbourList.add(ni);
            }
    }

    unioncPlayer(i, neighbourList, state.unionInfoD()[cPlayer], cPlayer);
    }
}
else
{
   final int deleteGPlayer = context.state().next();
   final int deleteIdRoot  = find(deleteId, state.unionInfoD()
                                  [deleteGPlayer], deleteGPlayer);
   final BitSet itemsList= (BitSet) state.unionInfoD()[deleteGPlayer]
                  .getItemsList(deleteIdRoot, deleteGPlayer).clone();

  for (int i = itemsList.nextSetBit(0); i >= 0;
                                  i = itemsList.nextSetBit(i + 1))
  {
     state.unionInfoD()[deleteGPlayer].clearParent(i, deleteGPlayer);
     state.unionInfoD()[deleteGPlayer].clearItemsList(i, deleteGPlayer);
     state.unionInfoD()[deleteGPlayer].clearAllitemWithOrthoNeighbors
                                        (i, deleteGPlayer);
  }

  final int commomPlayer    = context.game().players().count() + 1;
  final int delRootcPlayer  = find(deleteId, state.unionInfoD()
                                  [commomPlayer], commomPlayer);
  final BitSet itemsListcPlayer  = (BitSet) state.unionInfoD()[commomPlayer]
                  .getItemsList(delRootcPlayer, commomPlayer).clone();
```

```
        for (int i = itemsListcPlayer.nextSetBit(0); i >= 0
                                    ; i = itemsListcPlayer.nextSetBit(i + 1))
        {
          state.unionInfoD()[commomPlayer].clearParent(i, commomPlayer);
          state.unionInfoD()[commomPlayer].clearItemsList(i, commomPlayer);
          state.unionInfoD()[commomPlayer].clearAllitemWithOrthoNeighbors
                                                              (i, commomPlayer);
        }

      itemsListcPlayer.xor(itemsList);

      for (int i = itemsListcPlayer.nextSetBit(0);
                 i >= 0;i = itemsListcPlayer.nextSetBit(i + 1))
      {
         final TIntArrayList nList;

         if ((GameType.OrthogonalDeletionOnly) != 0L)
         {
            nList = validPosition(graph.vertices().get(i).orthogonal());
         }
         else
         {
            nList = validPosition(graph.vertices().get(i).adjacent());
         }

         final int nListSz = nList.size();
         final TIntArrayList neighbourList = new TIntArrayList(nListSz);

         for (int j = 0; j < nListSz; j++)
         {
            final int ni = nList.getQuick(j);
            if (state.who(ni) != 0)
            {
                 neighbourList.add(ni);
            }
         }

         unioncPlayer(i, neighbourList, state.unionInfoD()[commomPlayer],
                                                              commomPlayer);
      }
   }
}
```

B.3 find.java

```
/**
*
* @param position   A cell number.
* @param uf         Object of union-find.
* @param whoSiteId  The current player type.
*
* @return        The root of the position.
*/

private static int find
(
        final int position,
        final UnionInfoD uf,
        final int whoSiteId
)
{
  final int parentId = uf.getParent(position, whoSiteId);

  if (parentId == Unused)
        return position;

  if (parentId == position)
        return position;
  else
        return find(uf.getParent(parentId, whoSiteId), uf, whoSiteId);
}
```

B.4 connect.java

```
/**
 *
 * @param position1      Integer position.
 * @param position2      Integer position.
 * @param uf             Object of union-find.
 * @param whoSiteId      The current player type.
 *
 * @return check         Are the position1 and position1
                         in the same union tree or not?
 */
private static boolean connected
(
        final int position1,
        final int position2,
        final UnionInfoD uf,
        final int whoSiteId
)
{
   final int root1 = find(position1, uf, whoSiteId);
   return uf.isSameGroup(root1, position2, whoSiteId);
}
```

Appendix C Games Algorithms/ Ludemes

C.1 isConnect

```
/**
 * Connect with union-find
 *
 * @author tahmina
 *
 */
public final class IsConnect extends BaseBooleanFunction
{

    private final RegionFunction[] regionArray;

    private final IndexOf roleFunc;

    private List<BitSet[]> precomputedRegionsBitSets = null;

    private final Regions regions;

    private int number;

    /**
    * Constructor.
    * @param number     The minimum number of set need to connect.
    * @param regions    The disjoint regions set, which use for connection.
    * @param role       Type of player (i.e., black or white).
    * @param regionType Type of the regions.
    *
    */
   public IsConnect
   (
        @Opt final Integer number,
        @Or  final RegionFunction[] regions,
        @Or  final RoleType role,
        @Or  final RegionTypeStatic regionType
    )
    {
        this.number = (number == null) ? 0 : number.intValue();
        this.regionArray = regions;
        roleFunc = (role == null) ? null : new IndexOf(role);
        this.regions = (regionType == null) ? null
        : new game.equipment.other.Regions(null, null, null, null,
                                null, regionType, null, null);
     }
```

```
/**
 * @param context The current Context of the game board.
 *
 * In this algorithm, first we make an union tree with the last move.
 * check: Is the last move make a specific number of connection or not?
 *
 */
@Override
public boolean eval(final Context context)
{

  final int siteId = context.trial().lastMove().getTo();
  final boolean unionFlag = context.unionFindCalled();
  final boolean unionDeleteFlag = context.unionFindDeleteCalled();

  if(context.game().isUnionFindDelete())
  {
    if (!unionDeleteFlag)
    {
        UnionFindD.eval(context, siteId);
    }
  }
  else
  {
    if (!unionFlag)
    {
       UnionFind.eval(context, false, AbsoluteDirection
                                      .All, Constants.UNDEFINED);
    }
  }
  final int parentOfSiteid;

  if(context.game().isUnionFindDelete())
  {
    parentOfSiteid=find(siteId,state.unionInfoD()[whoSiteId],whoSiteId);
  }
  else
  {
    parentOfSiteid=find(siteId, state.unionInfo()[whoSiteId],whoSiteId);
  }

  int connection             = 0;
  final int numberOfRegions = sitesn.size();

  if (number == 0)
  {
    number = numberOfRegions;
```

```
        }

        if(context.game().isUnionFindDelete())
        {
          for (int i = 0; i < numberOfRegions; i++)
          {
             if (regionsBitSets[i].intersects(state.unionInfoD()
                     [whoSiteId].getItemsList(parentOfSiteid,whoSiteId)))
                connection++;

                if (connection == number)
                {
                   return true;
                }
             }
          }
          else
          {
             for (int i = 0; i < numberOfRegions; i++)
             {
                if (regionsBitSets[i].intersects(state.unionInfo()
                     [whoSiteId].getItemsList(parentOfSiteid,whoSiteId)))
                connection++;

                if (connection == number)
                {
                    return true;
                }
             }
          }
          return false;
    }
}
```

C.2 isLoopAux

```
/**
 *Helping file of the isLoop
 *
 * @author tahmina
 *
 */
public final class IsLoopAux extends BaseBooleanFunction
{
   private static final long serialVersionUID = 1L;

   /** Direction chosen. */
   private final AbsoluteDirection dirnChoice;


   public IsLoopAux
   (
     @Opt final AbsoluteDirection dirnChoice
   )
   {
     this.dirnChoice=(dirnChoice == null)
                  ?AbsoluteDirection.All : dirnChoice;
   }

/**

* @return Are the neighbor's position of the last move
*                is sufficient to create an Open loop?
*
*/
 @Override
 public boolean eval (final Context context)
  {
   final int siteId = context.trial().lastMove().getTo();
   if (siteId == Constants.UNDEFINED)
                        return false;

   final Graph graph = context.graph();
   final int cid = context.containerId()[siteId];
   final ContainerState state = context.state()
                                .containerStates()[cid];
   final int whoSiteId = state.who(siteId);

   List<Vertex> neighbourList = new ArrayList<Vertex>();

   if (dirnChoice == AbsoluteDirection.All)
```

```
    {
        neighbourList = graph.vertices().get(siteId).adjacent();
        return loop(siteId, graph, state, neighbourList,
                                        state.unionInfo()[whoSiteId]);
    }
    else
    {
        List<Vertex> orthoList = graph.vertices().get(siteId).orthogonal();
        int count = 0;

        for(int i = 0;i<orthoList.size();i++)
        {
                final int oi = orthoList.get(i).index();
                if(state.who(oi) == whoSiteId)
                {
                        count++;
                }
          }

          if(count > 1)
          {
              neighbourList = graph.vertices().get(siteId).adjacent();
              return loopOrtho(siteId, graph, state, neighbourList,
                                        state.unionInfo()[whoSiteId]);
          }
          else
              return false;
     }
}

//-----------------------------------------------------------------------

/**
* @param siteId          The last move of the current game state.
* @param graph           The graph of the present game board.
* @param state           The present state of the game board.
* @param neighbourList   The adjacent list of the last move.
* @param uf              The object of the union-find.
*
*
* @return               Is it loop or not?
*/
private static boolean loop
(
        final int siteId,
        final Graph graph,
        final ContainerState state,
```

```
        final List<Vertex> neighbourList,
        final UnionInfo uf
)
{
 final int whoSiteId        = state.who(siteId);
 final int numNeighbours     = neighbourList.size();
 final int[] localParent     = new int[numNeighbours];
 int adjacentSetsNumber      = 0;

 Arrays.fill(localParent, -1);
// use for the temporary union tree

 List<Vertex> kList;
 List<Vertex> interSectionList;

  for (int i = 0; i < numNeighbours; i++)
  {
    final int ni = neighbourList.get(i).index();
    if (state.who(ni) == whoSiteId)
    {
      if (localParent[i] == -1)
      {
        localParent[i] = i;
      }

    kList = graph.vertices().get(ni).adjacent();

    interSectionList = interSection(neighbourList, kList);

    for (int j = 0; j < interSectionList.size(); j++)
    {
    final int nj = interSectionList.get(j).index();

    if ((state.who (nj) == whoSiteId) && (ni != siteId))
    {
      for (int m = 0; m < numNeighbours; m++)
      {
        if ((m != i) && (nj == neighbourList.get(m).index()))
        {
          if (localParent[m] == -1)
          {
            localParent[m] = i;
            break;
          }
          else
          {
          int mRoot = m;
```

```
            int iRoot = i;
            while (mRoot != localParent[mRoot])
                        mRoot = localParent[mRoot];
            while (iRoot != localParent[iRoot])
                        iRoot = localParent[iRoot];

            localParent[iRoot] = localParent[mRoot];
            break;
          }
        }
      }
    }
  }
}

 for (int k = 0; k < numNeighbours; k++)
 {
  if (localParent[k] == k)
  {
        adjacentSetsNumber++;
  }
 }
 if (adjacentSetsNumber > 1)
 {
  for (int i = 0; i < numNeighbours; i++)
  {
   if (localParent[i] == i)
   {

    final int rootI=find(neighbourList.get(i)
                                .index(),uf,whoSiteId);

    for (int j = i + 1; j < numNeighbours; j++)
    {
     if (localParent[j] == j)
     {
        if (uf.isSameGroup(rootI, neighbourList.get(j)
                                .index(),whoSiteId))
        {
          return true;
        }
       }
     }
    }
   }
  }
```

```
 return false;
}

/**
* @param siteId          The last move of the current game state.
* @param graph           The graph of the present game board.
* @param state           The present state of the game board.
* @param neighbourList   The adjacent list of the last move.
* @param uf              The object of the union-find.
*
*
* @return                Is it loop or not?

private static boolean loopOrtho
(
        final int siteId,
        final Graph graph,
        final ContainerState state,
        final List<Vertex> neighbourList,
        final UnionInfo uf
)
{
        final int whoSiteId = state.who(siteId);
        final int numNeighbours= neighbourList.size();
        final int[] localParent= new int[numNeighbours];
        int adjacentSetsNumber = 0;

        Arrays.fill(localParent, -1);
// use for the temporary union tree

        List<Vertex> kList;
        List<Vertex> interSectionList;
        List<Vertex> orList = graph.vertices()
                              .get(siteId).orthogonal();


 for (int i = 0; i < numNeighbours; i++)
 {
  final int ni = neighbourList.get(i).index();
  if (state.who(ni) == whoSiteId)
  {
   boolean orthogonalPosition = false;
   for (int k = 0; k < orList.size(); k++)
   {
    final int oi = orList.get(k).index();

    if(ni == oi)
```

```
    {
      orthogonalPosition = true;
      break;
    }
   }
  if (orthogonalPosition)
  {
    if (localParent[i] == -1)
    {
      localParent[i] = i;
    }

    kList = graph.vertices().get(ni).orthogonal();
    interSectionList = interSection(neighbourList, kList);

   for (int j = 0; j < interSectionList.size(); j++)
   {
    final int nj = interSectionList.get(j).index();
    if ((state.who (nj) == whoSiteId) && (ni != siteId))
    {
     for (int m = 0; m < numNeighbours; m++)
     {
      if ((m != i) && (nj == neighbourList.get(m).index()))
      {
       if (localParent[m] == -1)
       {
        localParent[m] = i;
        break;
       }
       else
       {
        int mRoot = m;
        int iRoot = i;
        while (mRoot != localParent[mRoot])
                       mRoot = localParent[mRoot];
        while (iRoot != localParent[iRoot])
                       iRoot = localParent[iRoot];
        localParent[iRoot] = localParent[mRoot];
        break;
       }
      }
     }
    }
   }
  }
}
```

```
 for (int k = 0; k < numNeighbours; k++)
 {
  if (localParent[k] == k)
  {
    adjacentSetsNumber++;
  }
 }

 if (adjacentSetsNumber > 1)
 {

  for (int i = 0; i < numNeighbours; i++)
  {
   if (localParent[i] == i)
   {
     final int rootI=find(neighbourList.get(i).index(),uf,whoSiteId);

     for (int j = i + 1; j < numNeighbours; j++)
     {
      if (localParent[j] == j)
      {
       if (uf.isSameGroup(rootI,neighbourList.get(j).index(),whoSiteId))
       {
        return true;
       }
      }
     }
    }
   }
  }
  return false;
 }
}
```

C.3 isLoop

```
/**
 * Detection of a loop.
 *
 * @author tahmina.
 *
 */
public final class IsLoop extends BaseBooleanFunction
{

/** all types of ring */
private final  boolean all;

/** full ring */
private final  boolean fullRing;

/** ring with empty cell */
private final  boolean empty;

/** ring with inside enemy*/
private final  boolean enemy;

/** Direction chosen. */
private final AbsoluteDirection dirnChoice;

/**
 * The colour of the path.
 */
private final IntFunction colourFn;

/**
* The starting point of the loop.
*/
private final IntFunction startFn;

/**
* The starting points of the loop.
*/
private final RegionFunction regionStartFn;

private boolean opponent  = false;
private boolean edgeFound = false;
private boolean freeCell  = false;
private boolean emptyCheck;
private boolean enemyCheck;
```

```
public IsLoop
(
 @Opt @Name final Boolean all,
 @Opt @Name final Boolean full,
 @Opt @Name final Boolean empty,
 @Opt @Name final Boolean enemy,
 @Opt final AbsoluteDirection dirnChoice
)
{
 this.all = (all == null)?false:all.booleanValue();
 this.fullRing = (full == null)? false:full.booleanValue();
 this.empty =(empty == null)?false:empty.booleanValue();
 this.enemy =(enemy == null)?false:enemy.booleanValue();
 this.dirnChoice =(dirnChoice == null)?AbsoluteDirection.All:dirnChoice;
}

//-----------------------------------------------------------------
/**
* @param context The current Context of the game board.
* @return Is the last move create a ring or not?
*/

@Override
public boolean eval (final Context context)
{
 final int siteId  = startFn.eval(context);
 boolean ringflag       = false;
 final boolean unionFlag = context.unionFindCalled();
 final boolean defaultFlag =!(all||fullRing||empty||enemy);
 emptyCheck                = empty;
 enemyCheck                = enemy;
 final Graph graph = context.graph();
 final int cid  = context.containerId()[siteId];
 final ContainerState state = context.state()
                                     .containerStates()[cid];
 final int whoId = context.state().next();
 final int totalVertices= graph.vertices().size();
 final int whoSiteId = state.who(siteId);
 final List<Vertex> neighbourList = graph.vertices()
                                 .get(siteId).adjacent();
 final int whatSideId   = state.what(siteId);
 final int numNeighbourSize= neighbourList.size();

 if(!unionFlag)
 {
  ringflag = UnionFind.eval(context, true,
                        dirnChoice, Constants.UNDEFINED);
```

```
}

if((defaultFlag || all) && (ringflag))
{
 return true;
}

 if(!(emptyCheck ||enemyCheck))
 {
   if (dirnChoice == AbsoluteDirection.All)
   {
    if (!fullRing)
    {
     for(int i = 0 ; i < numNeighbourSize; i++)
     {
       final int ni = neighbourList.get(i).index();
       final int triValue = graph.vertices().get(ni)
                                                  .maxOrtho();

       if(triValue == 3)
       {
        final List<Vertex> nList = graph.vertices()
                                        .get(ni).orthogonal();
        final int nListSize = nList.size();
        int count = 0;

        for(int j = 0 ; j < nListSize; j++)
        {
         final int nj = nList.get(j).index();
         if(state.who(nj) == whoSiteId)
         {
           count++;
           if(count == triValue)
           {
               return true;
           }
               }
             }
           }
         }
       }
    }
  if(emptyCheck)
  {
   if (dirnChoice == AbsoluteDirection.All)
  {
```

```
    for(int i = 0 ; i < numNeighbourSize; i++)
    {
     final int ni = neighbourList.get(i).index();
     final int triValue = graph.vertices().get(ni).maxOrtho();

    if(triValue == 3)
    {
     final List<Vertex> nList = graph.vertices()
.get(ni).orthogonal()
     final int nListSize = nList.size();
     int count = 0;
     if(state.who(ni) == 0)
     {
      for(int j = 0 ; j < nListSize; j++)
     {
       final int nj = nList.get(j).index();
        if(state.who(nj) == whoSiteId)
        {
          count++;
          if(count == triValue)
          {
             return true;
          }
        }
       }
     }
    }
   }
  }
}

if(enemyCheck)
{
  if (dirnChoice == AbsoluteDirection.All)
  {
   for(int i = 0 ; i < numNeighbourSize; i++)
   {
     final int ni = neighbourList.get(i).index();
     final int triValue = graph.vertices().get(ni).maxOrtho();
     if(triValue == 3)
     {
      final List<Vertex> nList = graph.vertices()
.get(ni).orthogonal();
      final int nListSize = nList.size();
      int count = 0;

      if((state.who(ni) != 0)&&(state.who(ni) != whoSiteId))
```

```
      {
        for(int j = 0 ; j < nListSize; j++)
        {
          final int nj = nList.get(j).index();
          if(state.who(nj) == whoSiteId)
          {
            count++;
            if(count == triValue)
            {
                return true;
             }
            }
           }
          }
        }
      }
    }
}

final int[] dfsMarked = new int[totalVertices];
final int[] dfsParent = new int[totalVertices];
final int[] dfsColour = new int[totalVertices];

return loopWithEncircle(siteId, graph, state, whoId, ringflag,
    defaultFlag, state.unionInfo()[whoSiteId], neighbourList,
    dfsMarked, dfsParent, dfsColour);
}

//-----------------------------------------------------------------------

/**
* @param siteId          The last move of the current game state.
* @param graph           The graph of the present game board.
* @param state           The present state of the game board.
* @param whoId           The opponent player type of the last move.
* @param ringFlag        The flag of the ring from the isLoop.java.
* @param defaultFlag     The flag set for the default value;
* @param all             The flag of to check the all types of rings.
* @param fullRing        The flag of to check the full ring.
* @param encircle        The flag of to check enemy piece inside the loop.
* @param emptyCheck      The flag of to check at least one empty cell
                                                    inside the loop.
* @param uf              The object of the union-find.
* @param neighbourList   The list of neighbor vertices
* @param dfsMarked       use to visited vertices in dfs.
* @param dfsParent       use to store info for  parent vertices in dfs.
* @param dfsColor        use to store info for  cycle in dfs.
```

```
*
* @return        Is it desire Loop or not?
*
*/
private boolean loopWithEncircle
(
final int siteId,
final Graph graph,
final ContainerState state,
final int whoId,
final boolean ringflag,
final boolean defaultFlag,
final UnionInfo uf,
final List<Vertex> neighbourList,
final int[] dfsMarked,
final int[] dfsParent,
final int[] dfsColour
)
{
final int whoSiteId  = state.who(siteId);
final int numNeighbours = neighbourList.size();
final int[] localparent = new int[numNeighbours];
int adjacentSetsnumber     = 0;
List<Vertex> kList;

final int parentOfSiteId = find (siteId, uf, whoSiteId);
final int maximumBoardLessRadius = uf.getGroupSize
 (parentOfSiteId, whoSiteId);
final int dfsItr  = maximumBoardLessRadius * 2;

Arrays.fill (localparent, -1);
List<Vertex> klist;
List<Vertex> intersectionlist;

for (int i = 0; i < numNeighbours; i++)
{
  final int ni = neighbourList.get(i).index();
  if (state.who(ni) == whoSiteId)
  {
    if (localparent[i] == -1)
    {
       localparent[i] = i;
    }
klist = graph.vertices().get(ni).adjacent();
intersectionlist = interSection(neighbourList, klist);

for (int j = 0; j < intersectionlist.size(); j++)
```

```
{
  final int nj = intersectionlist.get(j).index();

  if ((state.who (nj) == whoSiteId) && (ni != siteId))
  {
   for (int m = 0; m < numNeighbours; m++)
   {
    if ((m != i) && (nj == neighbourList.get(m).index()))
   {
    if (localparent[m] == -1)
     {
        localparent[m] = i;
        break;
      }
     else
     {
       int mRoot = m;
       int iRoot = i;
       while (mRoot != localparent[mRoot])
                        mRoot = localparent[mRoot];
       while (iRoot != localparent[iRoot])
                        iRoot = localparent[iRoot];

       localparent[iRoot] = localparent[mRoot];
       break;
      }
     }
    }
   }
  }
 }
}

for (int k = 0; k < numNeighbours; k++)
{
  if (localparent[k] == k)
  {
    adjacentSetsnumber++;
  }
}
if (((adjacentSetsnumber > 1) &&
 (dirnChoice == AbsoluteDirection.All))
||
((adjacentSetsnumber > 0) &&
 (dirnChoice == AbsoluteDirection.Orthogonal)))
{
  if ((enemyCheck) || (emptyCheck))
```

```
  {
   if (ringflag)
   {
    for (int k = 0; k < numNeighbours; k++)
    {
    final int presentPosition
                = neighbourList.get(k).index();
    if((state.who (presentPosition) != whoSiteId))
    {
    opponent  = false;
    edgeFound = false;
      freeCell  = false;

     if (dfsMarked [presentPosition] == 0)
     {
      if (enemyCheck)
      {
        dfsCycle(presentPosition, siteId,
                -1, graph, state, whoSiteId, whoId, 0, dfsMarked,
                dfsParent, dfsColour, dfsItr, uf);

       if (opponent && !edgeFound)
       {
       return true;
       }
      }
      if (emptyCheck)
      {
        dfsCycle(presentPosition, siteId,
            -1, graph, state, whoSiteId, whoId, 0, dfsMarked,
                 dfsParent, dfsColour, dfsItr, uf);
         if (freeCell && !edgeFound)
         {
            return true;
          }
         }
        }
     }
    }
   }
  }
}

//-----------------------------------------------------------------------

if (fullRing || all || defaultFlag)
{
```

```
  if ((adjacentSetsnumber == 1) &&
   (dirnChoice == AbsoluteDirection.All))
    {
      final int maxOrthogonal = graph.vertices()
                                .get(siteId).maxOrtho();
      final List<Vertex> orthogonalList = graph.vertices()
                                .get(siteId).orthogonal();

       int sameColorNeighbour1 = 0;
       for (int i = 0; i < orthogonalList.size(); i++)
       {
         final int ni =  orthogonalList.get(i).index();
        if (state.who(ni) == whoSiteId)
        {
          sameColorNeighbour1++;
          if (sameColorNeighbour1 == maxOrthogonal)
         {
           return true;
         }
       }
    }
}
if (dirnChoice == AbsoluteDirection.Orthogonal)
{
  final int maxOrthogonal = graph.vertices()
.get(siteId).maxOrtho() * 2;
  final List<Vertex> nList = graph.vertices()
.get(siteId).adjacent();

   int sameColorNeighbour1 = 0;
    for (int i = 0; i < nList.size(); i++)
    {
      final int ni =  nList.get(i).index();
     if (state.who(ni) == whoSiteId)
     {
       sameColorNeighbour1++;
       if (sameColorNeighbour1 == maxOrthogonal)
       {
          return true;
        }
     }
  }
}

if ((adjacentSetsnumber == 1) &&
(dirnChoice == AbsoluteDirection.Orthogonal))
{
```

```
 final int maxOrthogonal = graph.vertices()
                           .get(siteId).maxOrtho();
 final int maxAdjacent = graph.vertices().get(siteId)
                                  .adjacent().size();
  if((maxOrthogonal == 6) && (maxAdjacent == 6))
  {
   final List<Vertex> orthogonalList = graph.vertices()
                                  .get(siteId).orthogonal();

   int sameColorNeighbour1 = 0;
   for (int i = 0; i < orthogonalList.size(); i++)
   {
     final int ni =  orthogonalList.get(i).index();
          if (state.who(ni) == whoSiteId)
          {
            sameColorNeighbour1++;
           if (sameColorNeighbour1 == maxOrthogonal)
           {
                return true;
           }
          }
      }
   }
}

if ((adjacentSetsnumber == 1) ||
 (adjacentSetsnumber == 2))
{
 for (int i = 0; i < numNeighbours; i++)
 {
  final int ni =  neighbourList.get(i).index();
  final int maxOrthogonal;

  if (state.who(ni) == whoSiteId)
  {
   if (dirnChoice == AbsoluteDirection.All)
   {
    kList  = graph.vertices().get(ni).orthogonal();
    maxOrthogonal = graph.vertices()
                                  .get(ni).maxOrtho();
    }
     else
    {
      kList  = graph.vertices().get(ni).adjacent();
      maxOrthogonal =  graph.vertices().get(ni)
                                  .maxOrtho() * 2;
    }
```

```
    int sameColorNeighbour = 0;

    for (int k = 0; k < kList.size(); k++)
    {
     if(state.who(kList.get(k).index())== whoSiteId)
     {
       sameColorNeighbour++;
       if (sameColorNeighbour == maxOrthogonal)
       {
         return true;
       }
     }
    }
   }
 }
}

if ((adjacentSetsnumber == 1) &&
(dirnChoice == AbsoluteDirection.Orthogonal))
{
  final List<Vertex> orthoList = graph.vertices()
                        .get(siteId).orthogonal();

  for (int i = 0; i < orthoList.size(); i++)
  {
    final int ni =  orthoList.get(i).index();
    int maxOrthogonal;

    if (state.who(ni) == whoSiteId)
    {
     kList  = graph.vertices().get(ni).adjacent();

     maxOrthogonal =  graph.vertices()
                                 .get(ni).maxOrtho();
     int numberFull = maxOrthogonal * 2;

     if((maxOrthogonal == 6) && (kList.size() == 6))
     {
        numberFull = 6;
      }
      int sameColorNeighbour = 0;

      for (int k = 0; k < kList.size(); k++)
      {
       if (state.who(kList.get(k).index()) == whoSiteId)
       {
        sameColorNeighbour++;
```

```
        if (sameColorNeighbour == numberFull)
        {
        return true;
       }
      }
     }
    }
   }
  }
 }
}
return false;
}

//-----------------------------------------------------------------------

/**
*
* @param presentPosition  The present vertex of the DFS traverse.
* @param siteId   The last move of the current game state.
* @param p  Parent of each vertex.
* @param graph Present status of graph.
* @param state  Present game state.
* @param whoSiteId   The last move player's type.
* @param whoId       The opponent player type of the last move.
* @param countItr    Maximum iteration range for the board Less game.
* @param dfsMarked   use to visited vertices in dfs.
* @param dfsParent   use to store info for  parent vertices in dfs.
* @param dfsColor    use to store info for  cycle in dfs.
* @param dfsItr     use to make a limit of iteration for the Boardless games.
* @param uf        The object of the union-find.
*
* Remarks   dfscycle() use to set some global flag, such as freeCell,
*           edgeFound, and inside of the ring, there is any
*           opponent player or not.
*/
private void dfsCycle
(
 final int presentPosition,
 final int siteId,
 final int p,
 final Graph graph,
 final ContainerState state,
 final int whoSiteId,
 final int whoId,
 final int countItr,
 final int[] dfsMarked,
```

```
 final int[] dfsParent,
 final int[] dfsColour,
 final int dfsItr,
 final UnionInfo uf
)
{
 final TIntArrayList verticesList;
 int newcountItr        = countItr;

 if (edgeFound)
 {
  return;
 }

  if (newcountItr >= dfsItr)
 {
  edgeFound = true;
  return;
 }

if(dirnChoice == AbsoluteDirection.Orthogonal)
{
 verticesList = validPosition(graph.vertices()
                .get(presentPosition).adjacent());
 }
 else
 {
 verticesList = validPosition(graph.vertices()
                .get(presentPosition).orthogonal());
}

if (verticesList == null)
{
 if ((state.who(presentPosition) == whoSiteId))
 return;
 edgeFound = true;
 return;
}

if(dirnChoice == AbsoluteDirection.Orthogonal)
{
 if(state.who(presentPosition) == whoSiteId)
 {
  final int rootPresentPosition = find (
                        presentPosition, uf, whoSiteId);
  final int rootSiteId = find (siteId, uf, whoSiteId);
```

```
    if(rootSiteId != rootPresentPosition)
    {
      edgeFound = true;
      return;
    }
   }
}

if ((presentPosition == siteId) ||
(state.who(presentPosition) == whoSiteId))
{
  return;
}

final List<Vertex> outerList = graph.outer();

for(int i = 0; i < outerList.size(); i++)
{
 final int edgevertex = outerList.get(i).index();
 if(edgevertex == presentPosition)
 {
  if ((state.who(presentPosition) == whoSiteId))
  return;
  edgeFound = true;
  return;
 }
}
// all is visited
if (dfsColour[presentPosition] == 2)
{
  return;
}
// already visited but not finished (but this is a cycle)
if (dfsColour[presentPosition] == 1)
{
  int cur  = p;
  dfsMarked[cur] = 1;
  while (cur != presentPosition)
  {
   cur = dfsParent[cur];
   dfsMarked[cur] = 1;
  }
  return;
 }
if (state.who(presentPosition) == whoId)
{
  opponent = true;
```

```
}

if ((state.who(presentPosition) != whoId) &&
(state.who(presentPosition) != whoSiteId))
{
  freeCell = true;
}
dfsParent[presentPosition] = p;
dfsColour[presentPosition] = 1;

for (int i = 0; i < verticesList.size(); i++)
{
  final int nextPosition = verticesList.getQuick(i);

   if (nextPosition == dfsParent[presentPosition])
    continue;

   newcountItr++;
   dfsCycle(nextPosition, siteId, presentPosition,
   graph, state, whoSiteId, whoId, newcountItr,
   dfsMarked, dfsParent,  dfsColour, dfsItr, uf);
  }
 dfsColour[presentPosition] = 2;
 }
}
```

C.4 groupSizeProduct

```
/**
 * It returns a multiplication value of the all group size.
 * (End function)
 *
 * @author tahmina
 */
public final class GroupProduct extends BaseIntFunction
{
 private static final long serialVersionUID = 1L;

 /** the type of player**/
 private final RoleType who;

 //---------------------------------------------------------------

 /**
 * Constructor.
 * @param who : Player's type
 */
 public GroupProduct
 (
  final RoleType who
 )
 {
  this.who = who;
 }
//---------------------------------------------------------------------
/**
* @param context The current Context of the game board.
*
* @return The multiplied value of the all group size of a player.
*
*/

 @Override
 public int eval(final Context context)
 {
  final Graph graph = context.graph();
  final ContainerState state = context.state().containerStates()[0];
  final int totalvertices= graph.vertices().size();
  final int whoSiteId = new IndexOf(who).eval(context);
  final int numPlayers = context.game().players().count();
  int mulValue = 1;

  for (int i = 0; i < totalvertices; i++)
```

```
  {
   final List<Vertex> neighbours = graph.vertices().get(i).adjacent();
   if (state.who(i) != 0)
   {
    unionAll(i, state, neighbours, state.unionInfo()[numPlayers + 1],
                                        numPlayers + 1, numPlayers);
   }
   if (state.who(i) == whoSiteId)
   {
    unionAll(i, state, neighbours, state.unionInfo()[whoSiteId],
                                        whoSiteId, numPlayers);
   }
  }
  for (int i = 0; i < totalvertices; i++)
  {
   if (i == state.unionInfo()[whoSiteId].getParent(i, whoSiteId))
   {
    final int eachGroupSize = state.unionInfo()[whoSiteId]
                                        .getGroupSize(i, whoSiteId);
    if(eachGroupSize > 0)
    {
     mulValue = mulValue * eachGroupSize;
    }
   }
  }
 return mulValue;
}

//-------------------------------------------------------------------------

/**
*
* @param siteId The each move of the game.
* @param state  Each state information.
* @param validatePositions  The adjacent list of our last movement.
* @param uf     The object of the union-find.
* @param whoSiteId The last move player's type.
*
* Remarks  This function will not return any things.
*
*/
private static void unionAll
(
 final int siteId,
 final ContainerState state,
 final List<Vertex> validatePositions,
 final UnionInfo uf,
```

```
 final int whoSiteId,
 final int numPlayers
)
{

 final int validatePositionsSize  = validatePositions.size();

 uf.setItem(siteId, siteId, whoSiteId);
 uf.setParent(siteId, siteId, whoSiteId);

 for (int i = 0; i < validatePositionsSize; i++)
 {
  final int ni = validatePositions.get(i).index();
  boolean connect = true;

  if
  (
   (( ni < siteId) && (state.who (ni) == whoSiteId) &&
   (whoSiteId != numPlayers + 1)) ||
   ((whoSiteId == numPlayers + 1) &&
   (state.who (ni) != 0) && ( ni < siteId))
   )
   {
    for (int j = i + 1; j < validatePositionsSize; j++)
    {
     final int nj = validatePositions.get(j).index();
     if (connected(ni, nj, uf, whoSiteId))
     {
      connect = false;
      break;
     }
    }
   if (connect)
   {

    final int rootP = find (ni, uf, whoSiteId);
    final int rootQ = find (siteId, uf, whoSiteId);

    if (uf.getGroupSize(rootP, whoSiteId)
      < uf.getGroupSize(rootQ, whoSiteId))
    {
        uf.setParent (rootP, rootQ, whoSiteId);
        uf.mergeItemsLists(rootQ, rootP, whoSiteId);
    }
    else
    {
        uf.setParent (rootQ, rootP, whoSiteId);
```

```
            uf.mergeItemsLists(rootP, rootQ, whoSiteId);
        }
      }
    }
  }
}
}
```

C.5 groupCount

```
/**
 * It returns the number of the all group.
 *
 * @author tahmina
 */
public final class GroupCount extends BaseIntFunction
{
 private static final long serialVersionUID = 1L;

 /** The type of player **/
 private final RoleType who;

 /** The minimum size of a group */
 private final IntFunction minFn;

 /** The total number of players **/
 private int numPlayers;

// ---------------------------------------------------------------------

/**
 * Constructor.
 *
 * @param who Player's type
 */
 public GroupCount
(
    final RoleType who,
    @Opt @Name final IntFunction min
)
{
  this.who = who;
  this.minFn = (min == null) ? new IntConstant(0) : min;
}
// ---------------------------------------------------------------------
@Override
public int eval(final Context context)
{
 final Graph graph = context.graph();
 final ContainerState state = context.state().containerStates()[0];
 final int totalvertices = graph.vertices().size();
 final int whoSiteId = new IndexOf(who).eval(context);
 this.numPlayers = context.game().players().count();
 final int min = minFn.eval(context);
 int count = 0;
```

```
 for (int i = 0; i < totalvertices; i++)
 {
  if (state.who(i) != 0)
  {
   unionAll(i, state, graph.vertices().get(i).adjacent(),
      state.unionInfo()[numPlayers + 1], numPlayers + 1);
  }
  if (state.who(i) == whoSiteId)
  {
   unionAll(i, state, graph.vertices().get(i).adjacent(),
      state.unionInfo()[whoSiteId], whoSiteId);
   }
  }
  for (int i = 0; i < totalvertices; i++)
  {
   final int eachGroupSize = state.unionInfo()[whoSiteId]
                                  .getGroupSize(i, whoSiteId);
   if (i == state.unionInfo()[whoSiteId].getParent(i, whoSiteId))
   {
    if (eachGroupSize >= min)
     count += 1;
   }
  }
 return count;
}
```

C.6 groupSize

```
/**
 * It returns the size of the all groups from a site.
 *
 * @author tahmina
 */
public final class GroupSizeUF extends BaseIntFunction
{
 private static final long serialVersionUID = 1L;
 /** the type of player **/
 private final IntFunction siteFn;
 /** Direction of the connection. */
 private final AbsoluteDirection direction;

// ---------------------------------------------------
/**
* Constructor.
*/
 public GroupSizeUF
 (
  @Opt    final IntFunction site,
  @Opt final AbsoluteDirection direction
 )
 {
  this.siteFn = (site == null) ? new LastToMove(null) : site;
  this.direction = (direction == null)
                        ? AbsoluteDirection.All : direction;
 }

// ---------------------------------------------------------
 @Override
 public int eval(final Context context)
 {
  final ContainerState state = context.state().containerStates()[0];
  final int site = siteFn.eval(context);
  if(site == Constants.Off)
   return Constants.UNDEFINED;

  final int indexComponent = state.what(site);
  if (indexComponent == 0)
   return 0;

  final int whoSiteId = state.who(site);
  final boolean unionFlag = context.unionFindCalled();

  if (!unionFlag)
```

```
    {
     UnionFind.eval(context, false, direction, Constants.UNDEFINED);
    }

    final int parentOfSiteid  = find(site,
                    state.unionInfo()[whoSiteId], whoSiteId);
    return  state.unionInfo()[whoSiteId].getGroupSize(parentOfSiteid,
                                                      whoSiteId);
  }
}
```

C.7 isSingleGroup

```
/**
 * To know if all the pieces of a player are connected and
 * form a group.
 * @author tahmina
 */
public final class IsSingleGroup extends BaseBooleanFunction
{
 private static final long serialVersionUID = 1L;
 /** Player Index */
 private final IntFunction playerId;

 /** Direction of the connection. */
 private final AbsoluteDirection direction;

// ---------------------------------------------------------------

/**
* Constructor.
* @param role    Type of player (i.e. black or white).
* @param direction The direction of the connection.
* @formatter:off
*/
 public IsSingleGroup
 (
  @Opt @Or final IntFunction indexPlayer,
  @Opt @Or final RoleType role,
  @Opt    final AbsoluteDirection direction
 )
 {
 int numNonNull = 0;
 if (indexPlayer != null)
                          numNonNull++;
 if (role != null)
                          numNonNull++;
 if (numNonNull == 0)
 {
  playerId = null;
 }
 else
 {
  if (indexPlayer != null)
        this.playerId = indexPlayer;
  else
   this.playerId = new IndexOf(role);
  }
```

```
  this.direction = (direction == null)
      ? AbsoluteDirection.All : direction;
 }

//-----------------------------------------------------------------
/**
* @param context The current Context of the game board.
*
* Is all the game stones make one group or not?
*
*/
 @Override
 public boolean eval(final Context context)
 {
  final int siteIdTo          = context.trial().lastMove().getTo();
  final Graph graph           = context.graph();
  final int cid               = context.containerId()[siteIdTo];
  final ContainerState state = context.state().containerStates()[cid];
  final int whoSiteId         = state.who(siteIdTo);
  final int totalvertices     = graph.vertices().size();
  int groupsize               = 0;

  for (int i = 0; i < totalvertices ; i++)
  {
   if (state.unionInfoD()[whoSiteId].getParent(i, whoSiteId) == i)
   {
    groupsize++;
   }
  }
  return (groupsize == 1);
 }
}
```

C.8 freedom

```
/**
 * It returns total number of freedom for last move.
 *
 * @author tahmina
 */
public final class Freedom extends BaseIntFunction
{
 private static final long serialVersionUID = 1L;
 private static int Unused = -1;
 /**
  * Constructor.
  */
 public Freedom
 (
 )
 {
 // Do nothing
 }

//-----------------------------------------------------------

/**
 * @param context The current Context of the game board.
 *
 * @return Total number of freedom for last move.
 *
 */

 @Override
 public int eval(final Context context)
 {
  final int siteId  = context.trial().lastMove().getTo();
  final Graph graph = context.graph();
  final int cid     = context.containerId()[siteId];
  final ContainerState state = context.state().containerStates()[cid];
  final int whoSiteId = state.who(siteId);
  final int whoSiteIdNext= context.state().next();
  boolean unionActive = false;
  int freedom = 0;
  final TIntArrayList nList;

  nList = validPosition(graph.vertices().get(siteId).orthogonal());

  final int nListSz = nList.size();
```

```
 for (int i = 0; i < nListSz; i++)
 {
  final int who = state.who(nList.getQuick(i));
  if (who != whoSiteIdNext)
  {
   if (who != whoSiteId)
   {
    freedom++;
   }
   else
   {
    unionActive = true;
   }
  }
 }
 if (!unionActive)
 {
  return freedom;
 }

 final int totalVertices = graph.vertices().size();
 final BitSet opponentAllBitset = new BitSet(totalVertices);
 final BitSet sameAllBitset = new BitSet(totalVertices);
 final BitSet nBitset = new BitSet(totalVertices);

 for (int i = 0; i < nListSz; i++)
 {
  final int ni = nList.getQuick(i);
  if (state.who(ni) == whoSiteId)
  {
   final int nRoot = find(ni, state.unionInfoD()
                                        [whoSiteId], whoSiteId);
   sameAllBitset.or(state.unionInfoD()[whoSiteId]
                                .getItemsList(nRoot, whoSiteId));
   nBitset.or(state.unionInfoD()[whoSiteId]
               .getAllItemWithOrthoNeighbors(nRoot, whoSiteId));
  }
 }
for (int j = 0; j < totalVertices; j++)
{
 if (j == state.unionInfoD()[whoSiteIdNext]
                                    .getParent(j, whoSiteIdNext))
 {
  opponentAllBitset.or(state.unionInfoD()[whoSiteIdNext]
                               .getItemsList(j, whoSiteIdNext));
 }
}
```

```
 for (int i = 0; i < nListSz; i++)
 {
  nBitset.set(nList.getQuick(i));
 }
 opponentAllBitset.and(nBitset);
 nBitset.xor(opponentAllBitset);
 nBitset.xor(sameAllBitset);
 nBitset.clear(siteId);
 return nBitset.cardinality();
 }
}
```

C.9 enclosed

```
/**
 * Moves applied to any enclosed group
 *
 * @author Eric Piette & Tahmina
 */
public final class Enclosed extends Moves
{
 private static final long serialVersionUID = 1L;
 private static int Unused = -1;
//------------------------------------------------------

 /** Location of the piece. */
 private final IntFunction startLocationFn;

 /** Direction chosen. */
 private final AbsoluteDirection dirnChoice;

 /** The piece to surround. */
 private final BooleanFunction targetRule;

 /** Moves applied after that one. */
 private final Moves next;

//------------------------------------------------------

/**
* Constructor.
*/
public Enclosed
(
 @Opt        final IntFunction startLocationFn,
 @Opt        final AbsoluteDirection dirnChoice,
 @Opt @Name final BooleanFunction of,
 @Opt @Name final Moves effect,
 @Opt        final Then consequences
)
{
 super(consequences);
 this.startLocationFn = (startLocationFn == null)
                        ? new From() : startLocationFn;
 this.dirnChoice = (dirnChoice == null)
                   ? AbsoluteDirection.All : dirnChoice;
 this.targetRule = (of == null)
                ? new IsEnemy(To.instance(), null) : of;
 this.next= (effect == null)
```

```
        ? new Remove(To.instance(), null, null) : effect;
}

//-------------------------------------------------------------------

@Override
public final Moves eval(final Context context)
{
 final Moves moves = new BaseMoves(super.consequents());
 final Graph graph = context.graph();
 final int siteIdTo= context.trial().lastMove().getTo();
 final int from = startLocationFn.eval(context);
 final int cid = context.containerId()[siteIdTo];
 final ContainerState state = context.state().containerStates()[cid];
 final int whoSiteId = state.who(siteIdTo);
 final int whoSiteIdNext = context.state().next();
 final int fromOrig = context.from();
 final int toOrig = context.to();
 final int stepOrig = context.step();
 final int totalVertices = graph.vertices().size();
 final Vertex fromV = graph.vertices().get(from);
 final BitSet sameAllBitset = new BitSet(totalVertices);


 final int[] directionIndices = context.board().tiling()
                                  .getChosenDirectionIndices(dirnChoice);

 for (int i = 0; i < totalVertices; i++)
 {
  if (i == state.unionInfoD()[whoSiteId].getParent(i, whoSiteId))
  {
   if (state.unionInfoD()[whoSiteId].getGroupSize(i, whoSiteId) != 0)
   {
    sameAllBitset.or(state.unionInfoD()[whoSiteId]
                                        .getItemsList(i, whoSiteId));
   }
  }
 }
 for (final int dirn : directionIndices)
 {
  for (int indexTo = 0; indexTo < fromV.indexedRadials()[dirn]
                                                  .length; indexTo++)
  {
  final int[] path = fromV.indexedRadials()[dirn][indexTo];
   if (path.length < 2)
        continue;
  final int pieceUnderThreat = path[1];
```

```
  boolean surrounded = false;
  final int root = find(pieceUnderThreat, state.unionInfoD()
                                          [whoSiteIdNext], whoSiteIdNext);
  if (root == Unused)
        continue;

  final BitSet numBitset = (BitSet) state.unionInfoD()[whoSiteIdNext]
                .getAllItemWithOrthoNeighbors(root, whoSiteIdNext).clone();
  numBitset.xor(state.unionInfoD()[whoSiteIdNext]
                                          .getItemsList(root, whoSiteIdNext));
  final BitSet tempBitset = (BitSet) numBitset.clone();
  tempBitset.and(sameAllBitset);
  numBitset.xor(tempBitset);

  if (numBitset.cardinality() == 1)
  {
   surrounded = true;
  }

 if (surrounded)
 {
  final BitSet bitsetsPlayer=(BitSet) state
  .unionInfoD()[whoSiteIdNext].getItemsList(root, whoSiteIdNext).clone();
  for (int i = bitsetsPlayer.nextSetBit(0);
  i >= 0;i = bitsetsPlayer.nextSetBit(i + 1))
  {
   boolean alreadyOnIt = false;
   for (final Move m : moves.moves())
   {
    if (m.getTo() == i)
    {
     alreadyOnIt = true;
     break;
    }
    }
    if (!alreadyOnIt)
    //INTERFACE DELETION
    }
   }
  }
 }
  context.setStep(stepOrig);
  context.setFrom(fromOrig);
  context.setTo(toOrig);
  return moves;
 }
}
```

C.10 sizeTerritory

```
/**
* Presently: It returns total number of Territory of a specific Player.
*
* @author tahmina
*/
public final class SizeTerritory extends BaseIntFunction
{
 private static final long serialVersionUID = 1L;
 private static int Unused = -1;

 private final IntFunction indexPlayer;

/**
* Constructor.
* @param role Player's type.
*/
 public SizeTerritory
 (
  final RoleType role
 )
 {
  indexPlayer = new IndexOf(role);
}
//-----------------------------------------------------------------------------
/**
* @param context  The current Context of the game board.
*
* @return The multiplied value of the all group size of a player.
*
*/

@Override
public int eval(final Context context)
{
 final Graph graph = context.graph();
 final ContainerState state = context.state().containerStates()[0];
 final int whoSiteId        = indexPlayer.eval(context);
 final int totalVertices    = graph.vertices().size();
 final int[] localParent    = new int [totalVertices];
 final int[] rank  = new int [totalVertices];
 final BitSet[] localItemWithOrth     = new BitSet[totalVertices];
 int sizeTerritory  = 0;

for (int i = 0; i < totalVertices; i++)
{
```

```
 localItemWithOrth[i]  = new BitSet(totalVertices);
 localParent[i]  = Unused;
 rank[i]    = 0;
}

for (int k = 0; k < totalVertices; k++)
{
 if (state.who (k) == 0)
 {
 localParent[k] = k;
 localItemWithOrth[k].set(k);

 final TIntArrayList nList = validPositionAll(graph
                        .vertices().get(k).orthogonal());

 for(int i = 0; i < nList.size(); i++)
 {
  localItemWithOrth[k].set(nList.get(i));
 }

for (int i = 0; i < nList.size(); i++)
{
  final int ni = nList.get(i);
  boolean connect = true;

  if ((state.who (ni) == 0) && (ni < k))
  {
   for (int j = i + 1; j < nList.size(); j++)
   {
     final int nj = nList.get(j);
     if (state.who(nj) == 0)
     {
       if (connected(ni, nj, localParent))
      {
        connect = false;
        break;
      }
    }
   }
    if (connect)
    {
     final int rootP = find(ni, localParent);
     final int rootQ = find(k, localParent);

     if (rank[rootP] < rank[rootQ])
     {
       localParent[rootP] = rootQ;
```

```
        localItemWithOrth[rootQ].or(localItemWithOrth[rootP]);
       }
       else
      {
     localParent[rootQ] = rootP;
     localItemWithOrth[rootP].or(localItemWithOrth[rootQ]);

     if (rank[rootP] == rank[rootQ])
     {
       rank[rootP]++;
     }
    }
   }
  }
}
}
}

for (int i = 0; i < totalVertices; i++)
{
  if (i == localParent[i])
  {
   boolean flagTerritory = true;
   int count = 0;

    for (int j = localItemWithOrth[i].nextSetBit(0); j >= 0;
                        j = localItemWithOrth[i].nextSetBit(j + 1))
    {
      if(state.who(j) == 0)
     {
       count++;
      }
      if((state.who(j) != whoSiteId) && (state.who(j) != 0))
      {
        flagTerritory = false;
      }
     }

     if(flagTerritory)
     {
       sizeTerritory += count;
      }
    }
  }
  return sizeTerritory;
 }
}
```

Appendix D Experimental Data

D.1 TEST-1

Table 1: p/s for Chameleon.

Test-1(Chameleon)	7x7	11X11	15x15	19X19
QU				
Mean	4888.55	992.6	307.45	124.2
STDEV	8.035939011	1.846761034	2.089447169	1.151657844
RUQPC				
Mean	4900	998.4	311.1	124.25
STDEV	8.111071057	6.451438028	2.789076436	1.860248993
WQUPC				
Mean	4908.05	993.65	310.15	125
SD	16.35936879	6.06347998	0.8750939799	0.3244428423

Table 2: m/s for Chameleon.

Test-1(Chameleon)	7x7	11X11	15x15	19X19
QU				
Mean	251389.45	123831.25	70401.05	45395.65
STDEV	441.8884683	536.4841563	617.0585978	493.1779091
RQUPC				
Mean	252009.35	124890.55	71258.6	45694.3
STDEV	394.7961866	764.8718142	577.3445929	632.1804456
WQUPC				
Mean	252549.3	124123.75	71168.25	45836.25
STDEV	496.9442519	558.1020351	124.825299	396.5451955

Table 3: p/s for Hex.

Test-1(Hex)	7x7	9X9	11X11	13x13	15x15	17x17	19X19
QU							
Mean	34742.4	20396.1	12094.35	8357.65	5936.4	4355.1	3314
STDEV	897.5447445	782.4434669	690.0225988	146.2101138	25.49179435	94.39943131	69.27519492
RQUPC							
Mean	36924.85	19244.45	12096.4	8023.4	6105.65	4536.25	3506.7
STDEV	556.7675765	723.1540911	243.7847195	232.2377185	116.3765553	22.30618276	60.03604181
WQUPC							
Mean	37069.7	19881.5	12645.65	8373.8	5986.35	4475.75	3468.9
STDEV	1061.666818	700.8116723	437.1355655	192.1719625	150.4996153	154.8985661	137.2166554

Table 4: m/s for Hex.

Test-1(Hex)	7x7	9X9	11X11	13x13	15x15	17x17	19X19
QU							
Mean	1468339.45	1481544.85	1347359.65	1254867.2	1225512.95	1149981.75	1103584.6
STDEV	37954.21497	53929.77592	84806.72493	29435.38635	20048.28731	25859.36141	32387.94976
RQUPC							
Mean	1560617.85	1367104.55	1300539.55	1217911.9	1243966	1195180.45	1159932.6
STDEV	23490.30074	51277.64871	26270.28403	35237.05542	24531.70496	11902.2948	21468.94209
WQUPC							
Mean	1566698.8	1372936.75	1331633.8	1269169.25	1243711.45	1200878.05	1154100.6
STDEV	44927.24019	53903.50177	46992.2683	31693.60818	34421.50652	47946.59165	46153.97185

D.2 TEST-2

Table 5: p/s for Atari Go,and Gonnect.

	Atari Go	Go	Gonnect
UQ			
Mean	1880.3	250.25	190.35
STDEV	16.56279915	1.585294261	1.631111988
RQUPC:			
Mean	1827.95	247.8	184.65
STDEV	2.981963324	2.706716792	2.75824124
WQUPC			
Mean	1879.75	252.3	191.25
STDEV	16.22173655	4.268612495	2.48945143

Table 6: m/s for Atari Go,and Gonnect.

Test-2	Atari Go	Go	Gonnect
UQ			
Mean	91251	31198.45	17180.1
STDEV	818.903246	274.7717665	69.50191213
RQUPC:			
Mean	88704.15	30398.65	16508.9
STDEV	461.8223941	427.8007745	248.5022292
WQUPC			
Mean	91363.35	31412.85	17397.5
STDEV	291.150219	479.9902713	239.5602331

D.3 TEST-3

Table 7: p/s for Hex.

	7x7	9X9	11X11	13x13	15x15	17x17	19X19
With UF							
Mean	37069.7	19881.5	12645.65	8373.8	5986.35	4475.75	3468.9
STDEV	1061.666818	700.8116723	437.1355655	192.1719625	150.4996153	154.8985661	137.2166554
Without UF							
Mean	44642.35	21720.05	11712	6944.2	4510.9	3094.45	2180.95
STDEV	170.1498643	119.744234	22.9645035	26.32509309	25.85567229	6.621138398	7.472581461

Table 8: m/s for Hex.

	7x7	9X9	11X11	13x13	15x15	17x17	19X19
With UF							
Mean	1566698.8	1372936.75	1331633.8	1269169.25	1243711.45	1200878.05	1154100.6
STDEV	44927.24019	53903.50177	46992.2683	31693.60818	34421.50652	47946.59165	46153.97185
Without UF							
Mean	1886749.65	1542598	1259186.7	1054271.85	919796.4	816506.45	723386.7
STDEV	7182.492019	8513.686962	2457.93707	3935.424678	5312.164703	1737.477133	2455.47528

D.4 TEST-4

Table 9: p/s and m/s for Andantino (hex) and Andantino (square).

WQUPC	Regular Andantino(hex-p/s)	Special Andantino(square-p/s)	Regular Andantino (hex-m/s)	Special Andantino (square-m/s)
mean	1123	2145.8	37218.2	54148.7
STDEV	8.83771821	44.94628373	318.9823622	1131.247569

D.5 TEST-5

Table 10: p/s and m/s for Pentalath and Go.

WQUPC	Pentalath(61 cells) p/s	Go(64 cells) p/s	Pentalath(61 cells) m/s	Go(64 cells) m/s
Mean	3315.4	536.75	44064.5	75294.7
STDEV	31.52342487	2.48945143	133.7584232	692.3062222

D.6 TEST-6

Table 11: p/s and m/s for Line Of Action (LOA) and Groups.

	UQ p/s	RQUPC-p/s	WQUPC-p/s	UQ m/s	RQUPC-m/s	WQUPC-m/s
LOA						
Mean	301.85	320.4	320.95	71890.5	76761.3	76391.95
STDEV	2.007223796	1.759186415	2.645253943	455.9500781	249.6013875	423.2813516
Groups						
Mean	78.1	81.45	90.1	184451.6	190926	210829.6
STDEV	2.221900562	1.637552731	1.618966532	2238.535503	2866.69867	3053.212207

D.7 TEST-7

Table 12: p/s and m/s for Havannah and Kensington havannah.

	Havannah-p/s	Kensington havannah-p/s	havannah-m/s	Kensington Havannah-m/s
WQUPC				
Mean	5040.85	6801.8	234589.1	206968.5
STDEV	209.3182606	38.24794456	9741.40467	1148.854235

D.8 TEST-8

Table 13: p/s and m/s for Y-hex.

	UF(p/s)	UFD(m/s)	UF(p/s)	UFD(m/s)
base-4				
Mean	31071.90476	31705.38095	1010420.857	1030991.571
STDEV:	87.37385465	89.73988867	2856.295525	2964.120385
base-5:				
Mean	17803.95	18026.35	965232.3	977341.9
STDEV:	94.92350106	111.8891721	5136.078507	6021.635378
base:6				
Mean	10039.7	10150.85	820334.75	829436.95
STDEV:	402.8471174	49.0469968	32903.80699	4024.177202

Table 14: p/s and m/s for Cross.

	UF(p/s)	UFD(m/s)	UF(p/s)	UFD(m/s)
base-4:				
Mean	14621.6	14390.2	452151.3	444983.2
STDEV	88.33989383	140.8902522	2709.594708	4343.330624
base-5:				
Mean	7938.95	8027.65	412072.75	416737.3
STDEV	78.5469856	59.55738056	4061.859217	3116.160443
base-6:				
Mean	4895.2	4728.1	385008.85	371872.7
STDEV	35.01368154	27.62131138	2736.495094	2166.804781
base-7:				
Mean	3121.15	3111.35	347039.1	345890.95
STDEV	21.25602082	26.83531648	2365.63889	2977.356519
base-8:				
Mean	2161	2094.8	323203.9	313302.05
STDEV	21.38863446	26.05379051	3192.195564	3891.620776
base-9:				
Mean	1511.85	1556.75	292940.55	301660.4
STDEV	29.79451557	15.28630278	5772.377996	2931.805433

Table 15: p/s and m/s for Omega.

Omega	UF(p/s)	UFD(m/s)	UF(p/s)	UFD(m/s)
2-Player				
Mean	15942.6	15758.5	956584.95	945544.5
STDEV	309.4047596	38.03806957	18560.24182	2277.57751
3-player				
Mean	14421.45	14334.4	778780.1	774086.3
STDEV	129.4675229	54.76591333	6987.999231	2956.566396
4-player				
Mean	13310.15	13224.55	638906.25	634804.65
STDEV	28.99051569	23.99226849	1394.371236	1150.288992