WARSAW UNIVERSITY OF TECHNOLOGY

Advanced algorithms

Final Documentation

Debelo Milkiyas Duguma,
Swetha Suresh Kumar

February 8, 2023

Contents

1 List of modifications

As modifications, we implemented new features including a *random point generator*, the ability to *input points from a file*, the capability to *clear all points*, and the option to *edit existing points* along with additional content that should have been added. The contents are:

- **Glossary of terms**
- **Task and its size**
- **Solution architecture**

2 Glossary of terms

The following terms are used in this documentation:

- **ball** - a point on a two-dimensional plane from the set $\mathbf{B}$;
- **hole** - a point on a two-dimensional plane from the set $\mathbf{H}$;
- **stroke** - segment connecting the ball and the hole on a two-dimensional plane;
- **balanced shot** - a shot in which there are as many balls as holes on each side of the half-planes separated by the straight line containing the shot;
- **match** (ball $\mathbf{B}$ to holes $\mathbf{H}$) - set of strokes covering sets of points $\mathbf{B}$ and $\mathbf{H}$;
- **planar match** - a match where no pair of strikes have a point in common;

3 Task and its size

3.1 Task

Two sets $\mathbf{B}$ and $\mathbf{H}$ different, with three non-collinear points of size n each from the space $\mathbb{R}^2$ (representing the positions of the balls and the positions of the holes, respectively).

3.2 Job size

The total number of points is $2n$. However, taking into account the fact that we are considering ball-hole pairs and for the sake of simplicity of further analysis, we can assume. that the size of the task does not depend on the constant and is n.

4 Technical Documentation

4.1 Solution architecture

The *Task* class was created to represent a task of a certain size. It contains collections of *Ball* and *Hole* objects. These objects inherit from the abstract class *Point* and correspond to the balls and holes of the problem under consideration. Derived from the Point class contain a point type field and *Value* appropriate for that type.

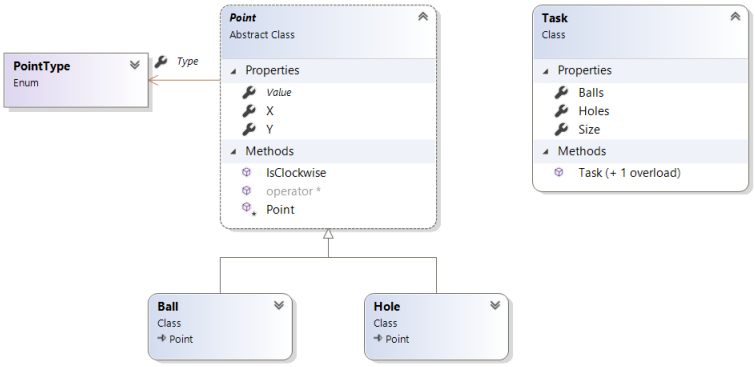

Figure 1: Structure of objects.

To implement the algorithm, the *IBalancedHitFinder* and *IPlanarMatchingFinder* interfaces were implemented. The former is implemented by the *BalancedHitFinder* class and has a *FindBalancedHit* method for finding a balanced shot, taking collections of objects corresponding to balls and holes as arguments.

This interface is used by the *PlanarMatchingFinder* class, which takes an object implementing it in the constructor. The *FindPlanarMatching* method calls a solution to the problem, and if successful, the result is returned using the *Matching* class, which contains a collection of *Hit* objects containing information about the pair of points (ball and hole) between which the hit is made .

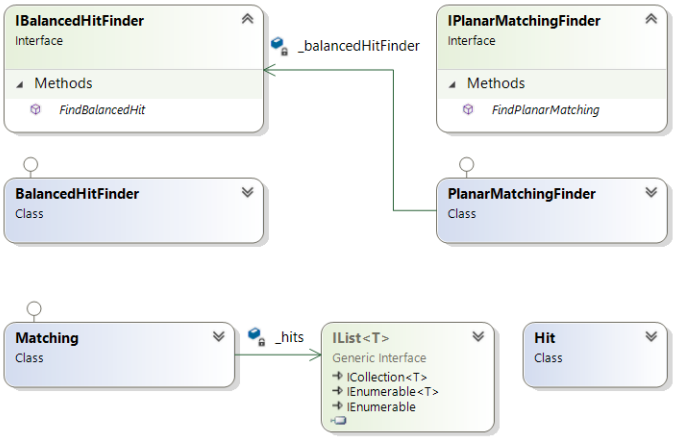

Figure 2: Objects that implement the algorithm.

Input operations are performed using the *TextFileTaskParser* class that implements the *ITaskParser* interface. The *Parse* method takes as an argument the path of a text file in which the input data for a task is saved in the appropriate format and returns a *Task* object representing this task.

Saving the current task is done by the *TextFileTaskSaver* class that implements the *TextFileTaskSaver* interface. Using the *Save* method, which takes an object of the *Task* type as an argument containing task data and saves them in a specific format to a text file.
On the other hand, saving the found solution is performed by the *TextFileMatchingPrinter* class that implements the *IMatchingPrinter* interface. It contains the *Print* method, which takes an object of the *Matching* class as an argument and writes the strokes it contains to a text file in the specified format.

4.2 Tests

The *GolfAppTests* project contains a unit test for the *GolfApp* project. They check the correctness of the logic of the classes executing the algorithm and the behavior in the case of various (including incorrect) public method arguments.

The tests were performed using the *xUnit* testing framework. Below is a screenshot showing the list of tests and their execution status.

In order to test the correctness of the implemented algorithm, we have created an API to generate tasks of a given size. On its basis, we can conduct both acceptance and performance tests. The tests were carried out for tasks with sizes from 0 to 500. They are located in the Correctness Tests section.

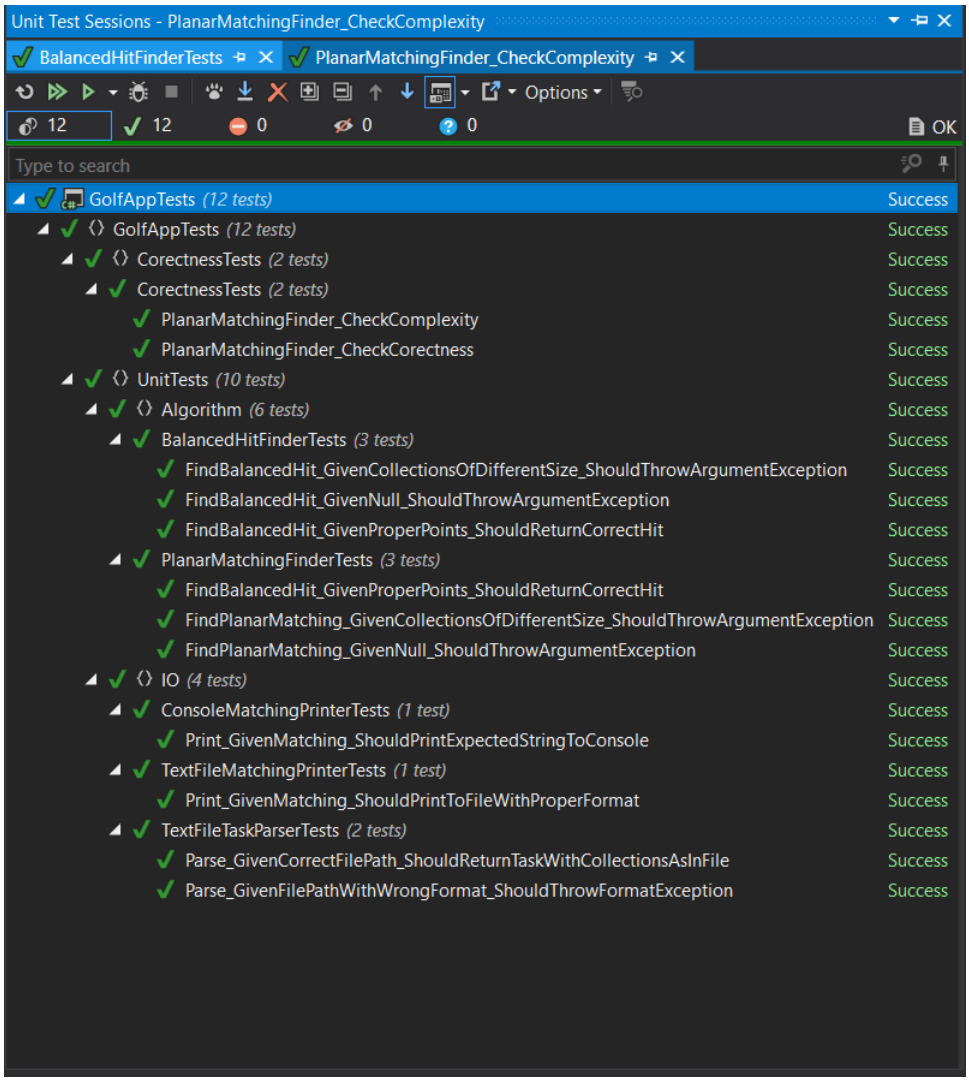

Figure 3: Unit tests.

Based on the results of measuring the speed of our algorithm for tasks of various sizes, we created a graph showing the relationship between the size of the task and the average execution time of the algorithm measured as the number of processor ticks.

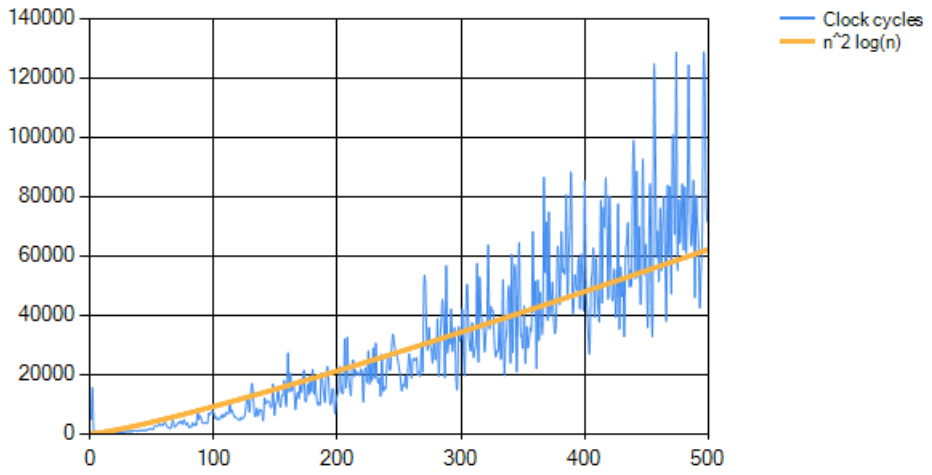

Figure 4: Graph of the relationship between the size of the task and the average execution time of the algorithm.

4.3 Division of work

In the project documentation, Debelo Milkiyas Duguma made a description of the algorithm, proved the correctness and computational complexity of the algorithm and prepared a user manual. Swetha Suresh Kumar described the algorithm along with the size of the task, created the pseudocode of the algorithm, described the way of presenting input and output data, solved a sample task and made technical documentation of the project.

In the programming project, Swetha Suresh Kumar implemented the algorithm, designed the structures used to calculate the program, created a prototype of the graphical interface. Debelo Milkiyas Duguma implemented input and output support in the program, significantly improved the graphical interface and prepared unit, performance ,acceptance tests and also prepared an API for testing.

5 Conclusions

Proofs provide us with mathematical confirmation, while tests allow us to examine the correctness and complexity of the algorithm in a pragmatic way. While validation tests should pass regardless of external factors, it should be remembered that complexity tests are subject to errors resulting from running the algorithm in some operating system environment where the management of processor time and other resources distributes processing power and memory among various processes that do not can always be turned off.

However, both the formal proof and the tests of correctness and complexity show that the proposed algorithm for finding a planar match of vertices from two different classes in a bipartite graph is correct and finds this match in no more than $O(n^2 \log n)$.

6 User manual

To run the application, run *Golf.exe inside GOLFUI folder*. After successful launch, the initial window of the program should appear.

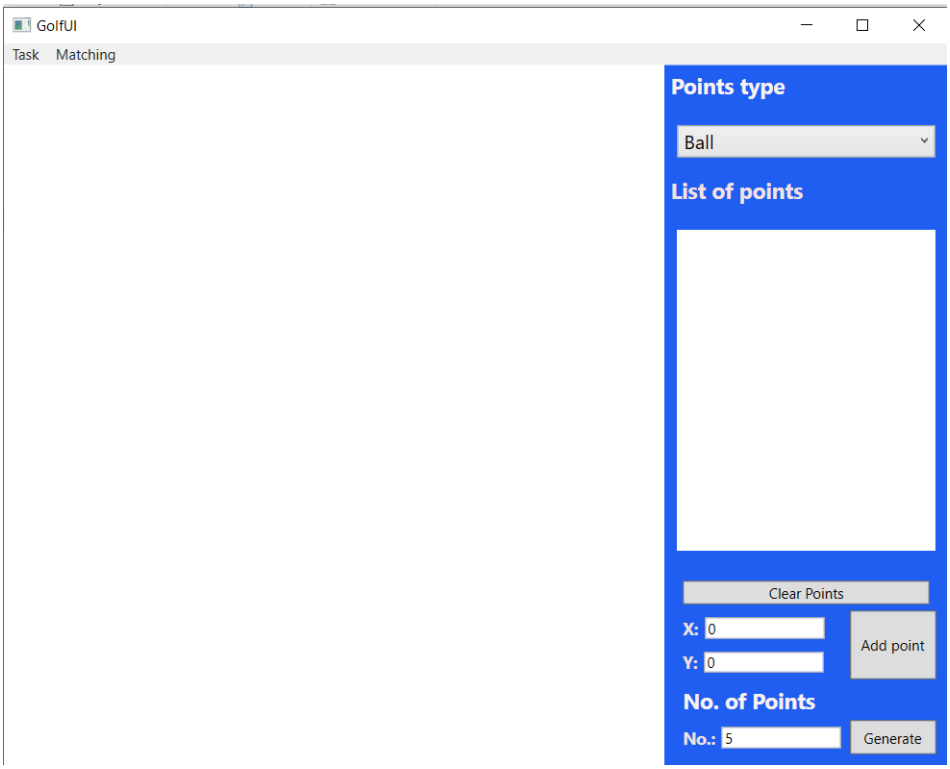

Figure 5: Initial window.

The following items are available in the menu:

- Task

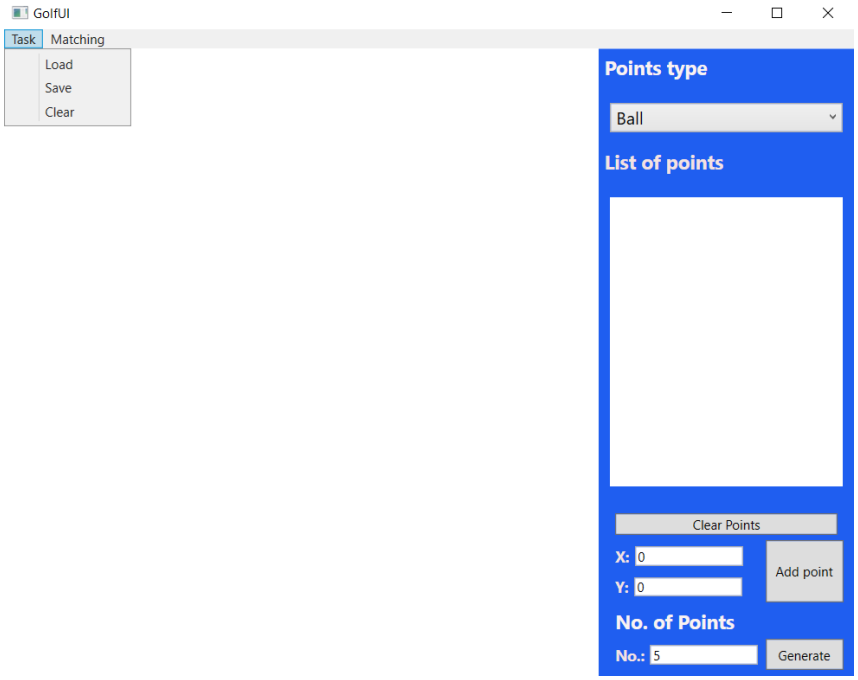

Figure 6: Task menu options.

– The *Load* option allows you to specify a file containing the task's input data and load it into the program. This data should be in the following format:

```
[N - number of balls]
[X coordinate of 1st ball], [Y coordinate of 1st ball]
...
[X coordinate of the Nth ball], [Y coordinate of the Nth ball]
[X coordinate of 1st hole], [Y coordinate of 1st hole]
...
[X coordinate of the Nth hole], [Y coordinate of the Nth hole]
```

Example:

```
3
10,10
0, 5
20,50
50,20
90,30
5, 5
```

The path to the test file: `GolfAppTests\TestFiles\TextFileTaskParserTest.txt`

– The *Save* option saves the current task to a specified file. The recording takes place in the format mentioned in the previous option.

– The *Clear* option: Selecting the "Clear" option will initiate a confirmation message to appear on the screen. This message will ask if you are sure you want to clear all points. To proceed with clearing all points, simply press "Yes" on the confirmation message.

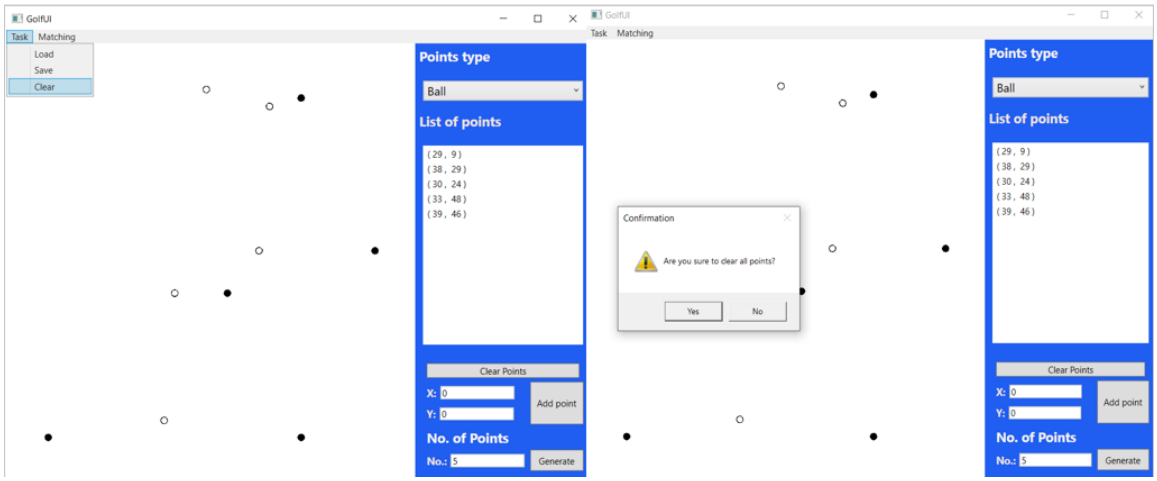

Figure 7: Clear option.

- Matching

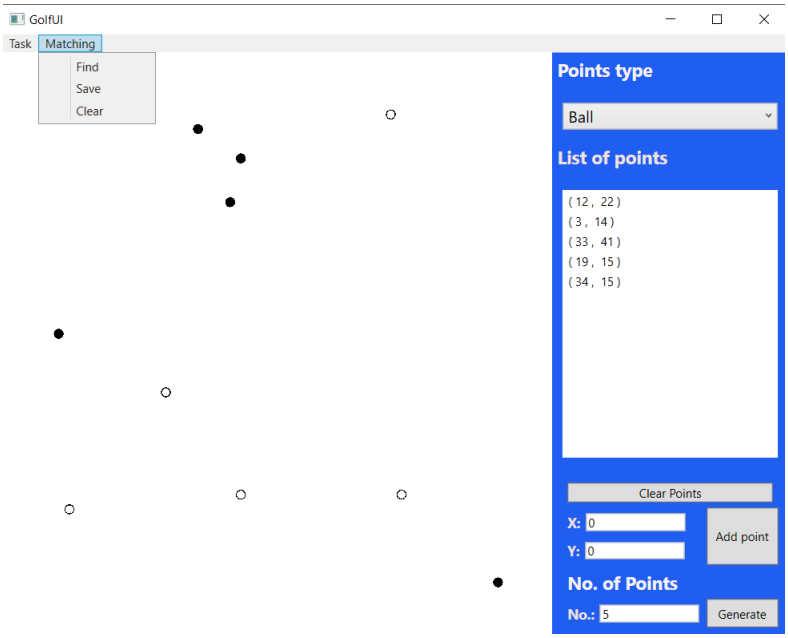

Figure 8: Matching menu options.

– The *Find* option finds strokes between the holes and balls defined in the current job.
– The *Save* option saves the solution to the problem (hit), if one has already been found, to the indicated text file.
– *Clear* option when selected, presents a pop-up that prompts for confirmation. If you press "Yes" on this pop-up, the currently found solution will be cleared.

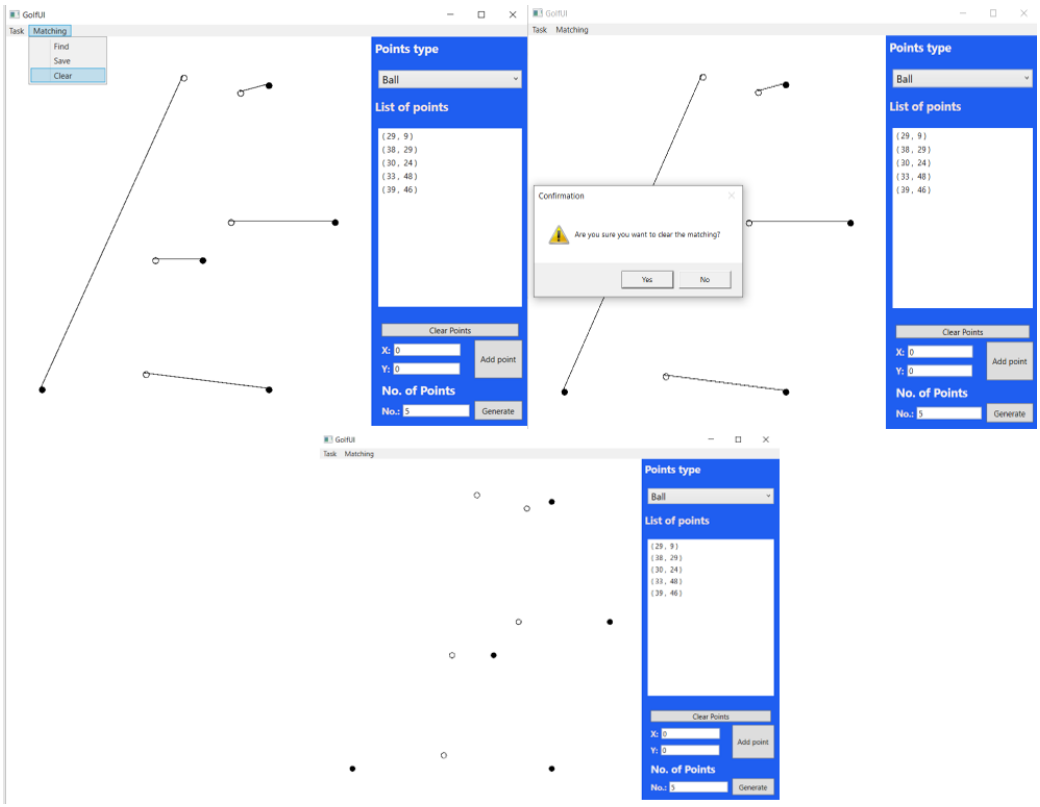

Figure 9: clear option.

Task points are presented in the list on the right side of the screen (*List of points*). However, only one point type is displayed in the list, which can be selected using the *Points type* drop-down list.

- Using the *Add point* button, it is possible to add new points to the currently selected list. Enter the coordinates of the new point in the text boxes next to the button and add the point by clicking the button. If a point with the specified coordinates exists in any of the lists, an error message will be displayed.

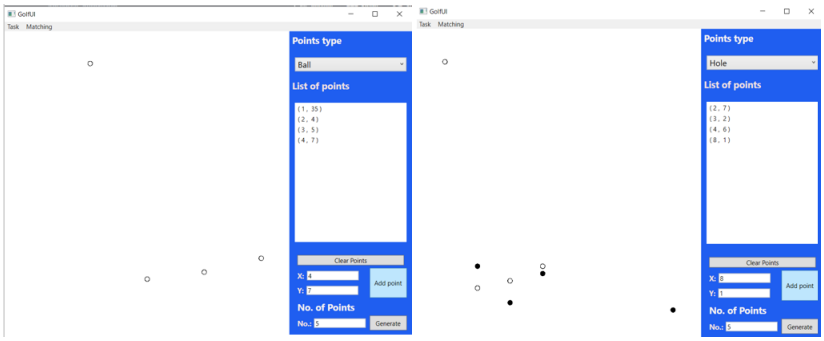

Figure 10: Adding a point.

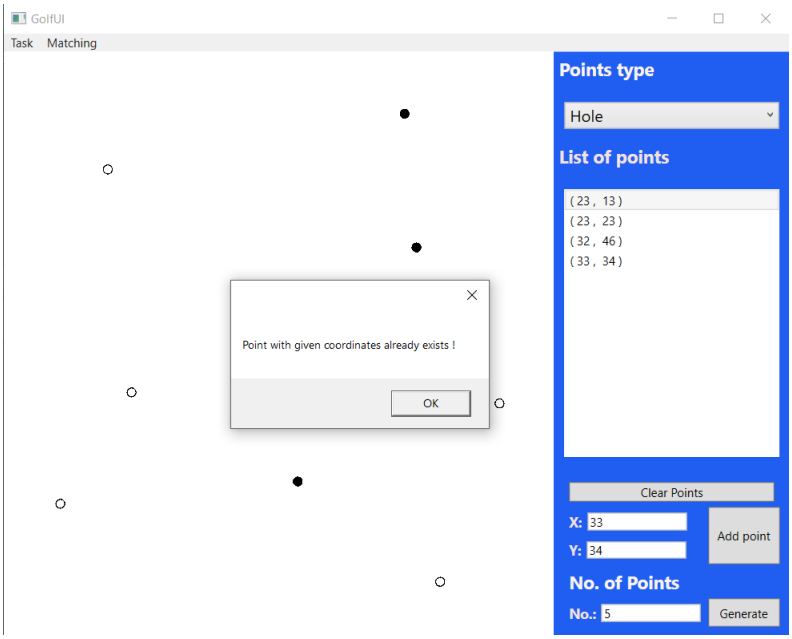

Figure 11: Error message.

- You can delete a point by right-clicking on any of the list items. A context menu will appear with *Remove* available. If this option is selected, the indicated element will be removed from the list of points.

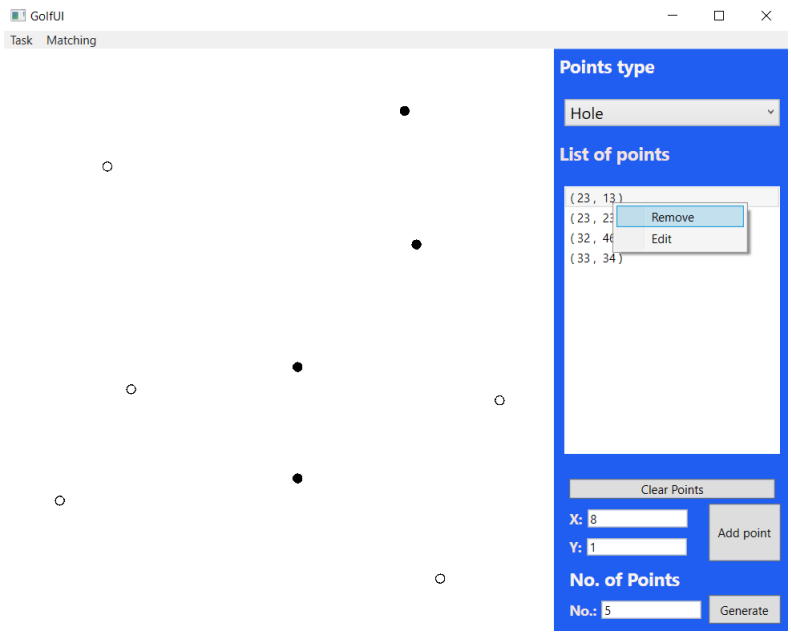

Figure 12: Deleting a point.

- To edit a point, right-click on the desired item in the list of points. A context menu will appear with the *Edit* option. If this option is chosen, a pop-up will appear allowing you to change the coordinates of the selected point. After making the desired changes, press "OK" to save and apply the edits.

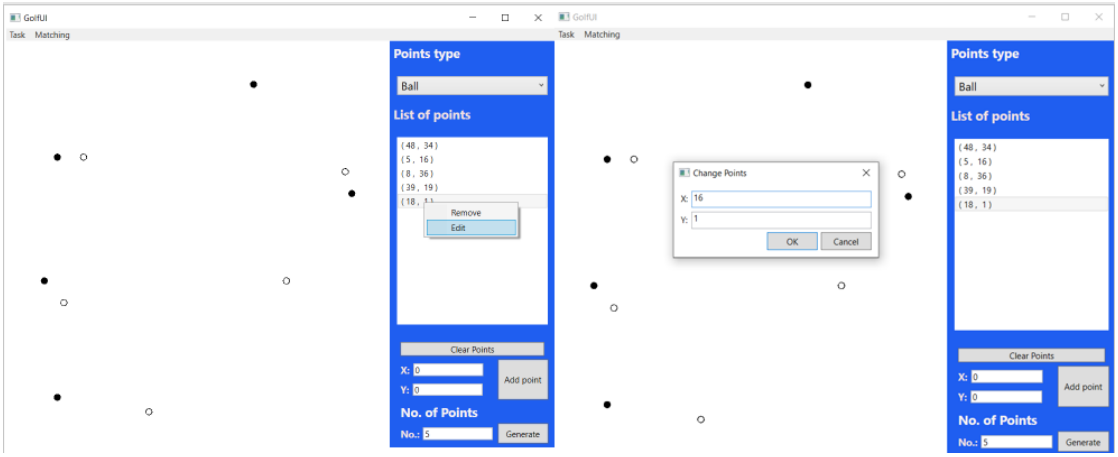

Figure 13: modifying a point.

- You may also produce random points by entering the desired number of points in the *No.:* box and clicking *generate* to obtain random points of the specified quantity.

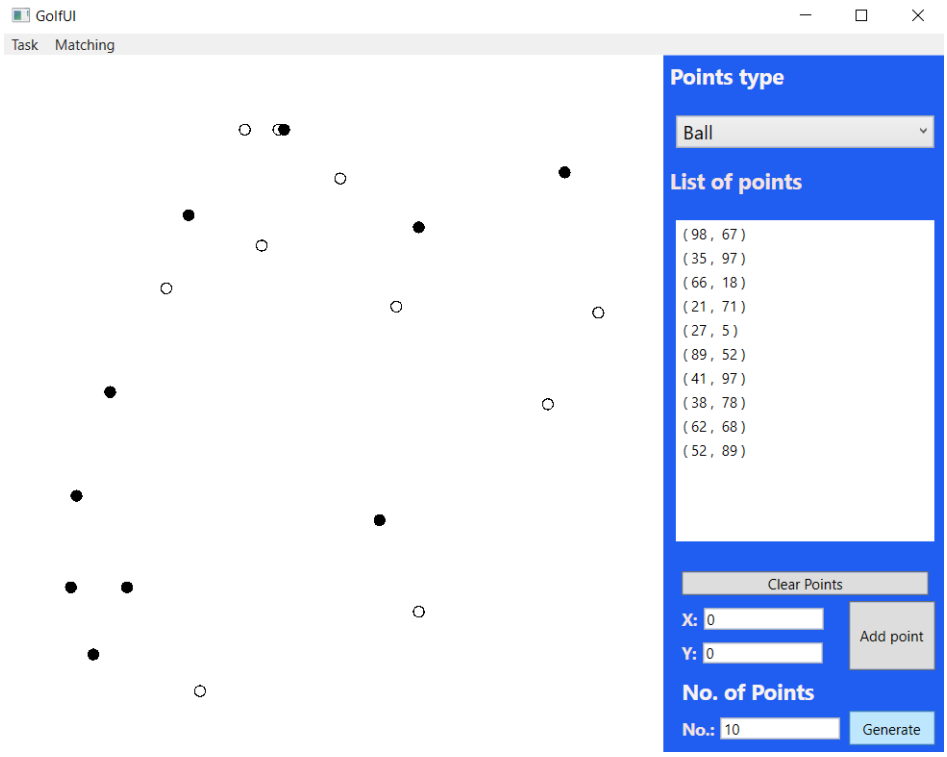

Figure 14: Random point generator

- Another way to remove all balls and holes from the board, the *Clear Points* button can be clicked. This button provides an easy and convenient way to clear everything on the board.

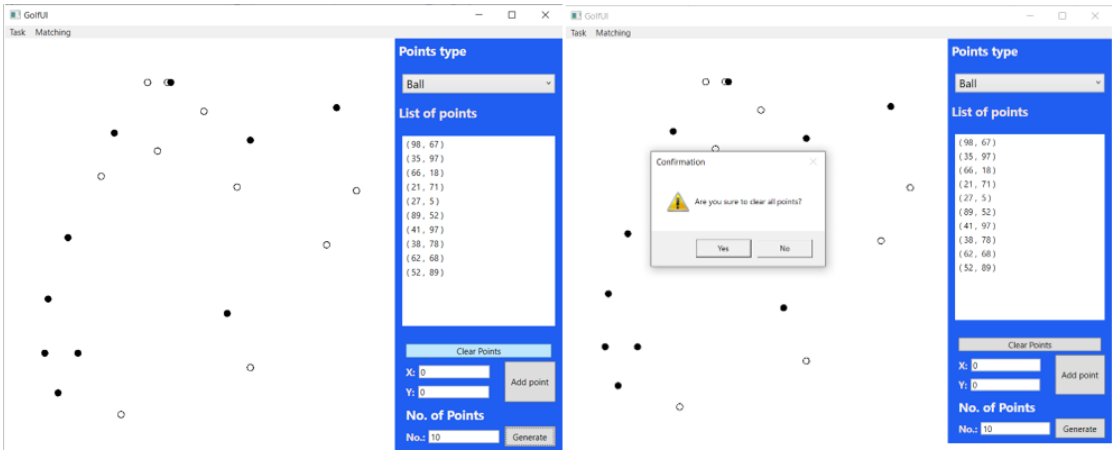

Figure 15: clear point option