Final Documentation - Wells Assignment Project

Piotr Krzemiński 305785
Zofia Wrona 305803

1 Modifications

The initial documentation has been revised and updated based on the requests specified in the emails until the submission date. As a result, the source code in the initial documentation closely mirrors the provided source code up until the delivery of the final documentation. The only key modification utilizes optimizing Python code by using numpy arrays with lower access times, and introduction of caching of key variables that reduced the previous time complexity from $O((nk)^4)$ to $O((nk)^3)$.

2 User's manual

2.1 Requirements

The application requires the Python executable version 3.7 with numpy version $>= 1.21$ and matplotlib version $>= 3.0.0$.

2.2 Execution Guidelines

To execute the application with the provided argument parser, follow the steps outlined below:

Command-Line Options

The application supports the following command-line options:

- `-m, -mode`: Specifies the application mode. The default is `GENERATE_AND_RUN`. Options include:
  - `GENERATE_INPUT`: Generates a new input file.
  - `GENERATE_AND_RUN`: Generates a new input file and runs the Hungarian algorithm.
  - `READ_INPUT`: Reads the provided input file and runs the Hungarian algorithm. For this case, n and k values are ignored.
  - `BENCHMARK`: Executes the Hungarian algorithm with each pair of parameters $(wells\_count, houses\_per\_well)$, where $wells\_count \in [1, n]$ and $houses\_per\_well \in [1, k]$.
- `-n`: Sets the value for the number of wells. The default value is 3, and you can adjust it by providing a positive integer value. Note that a combination of (n, k) with values greater or equal to 5 can result in the runtime of around an hour for complex cases.
- `-k`: Sets the value for the number of houses per well. The default value is 3, and you can adjust it by providing a positive integer value. Note that a combination of (n, k) with values greater or equal to 5 can result in a runtime of around an hour for complex cases.
- `-i, -input_file`: Specifies the input file name for the application. The default file name is `input.txt`.
- `-o, -output_file`: Specifies the output file name for the application. The default file name is `output.txt`.

Example Usage

To run the application with custom settings, use the following command:

```
python main.py -m <MODE> -n <n> -k <k> -i <file.txt> -o <file.txt>
```

As an example, this is how the main function is executed for default parameters:

```
python main.py -m GENERATE_AND_RUN -n 5 -k 5 -i input.txt -o output.txt
```

Adjust the values and file paths according to your specific requirements. Any missing argument will take its predefined default value.

3 Tests

To validate the algorithm's correctness, we conducted multiple tests as described below. The total cost was verified to be correct for each graph by comparing results with a brute force technique. Running the command:

```
python tests.py
```

executes these tests and saves the outputs to the ***Pictures*** folder.

1. Example from the initial documentation. Runtime of less than one second.

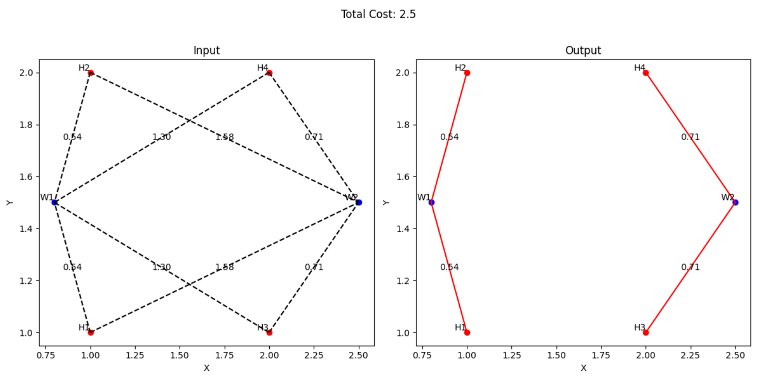

Figure 1: Initial documentation example

2. Randomly generated example with 2 wells and 10 houses per well - verification of correctness for significant amount of houses per well. Runtime of less than one millisecond.

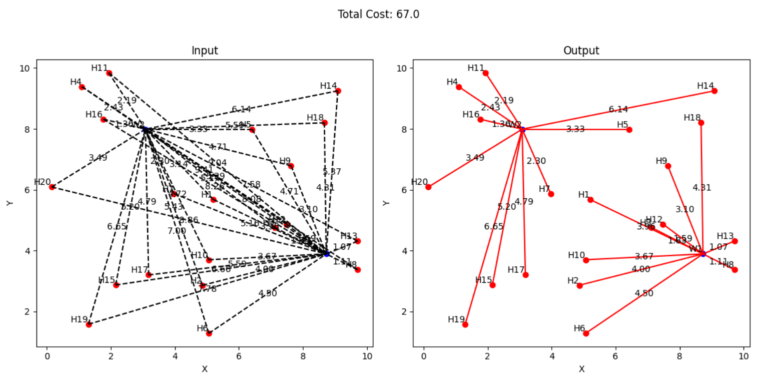

Figure 2: 2 wells, 10 houses per well

3. Randomly generated example with 10 wells and 2 houses per well - verification of correctness for significant amount of wells. Runtime of less than one millisecond.

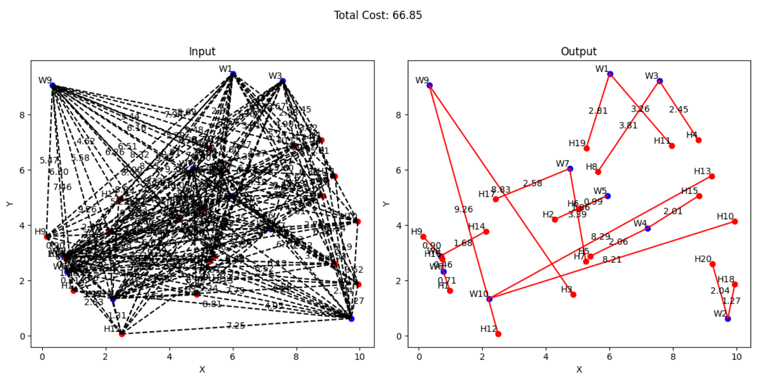

Figure 3: 10 wells, 2 houses per well

4. Randomly generated example with 4 wells and 4 houses per well - verification of correctness for a simple random case. Runtime of around 5 milliseconds.

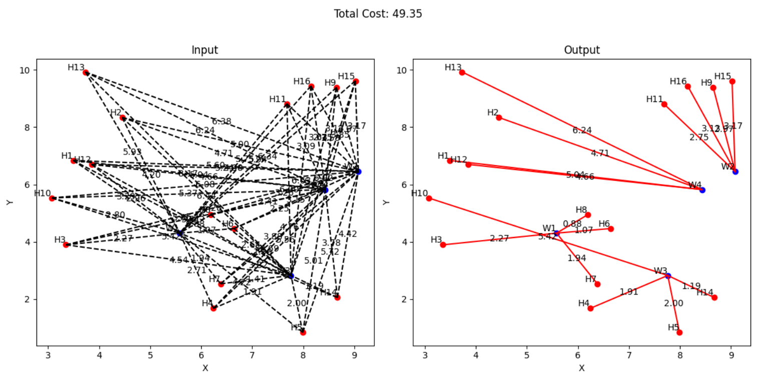

Figure 4: 4 wells, 4 houses per well

5. Medium example with 7 wells and 4 houses per well. Random example that is still clear in showing algorithm correctness. Runtime of around 10 milliseconds.

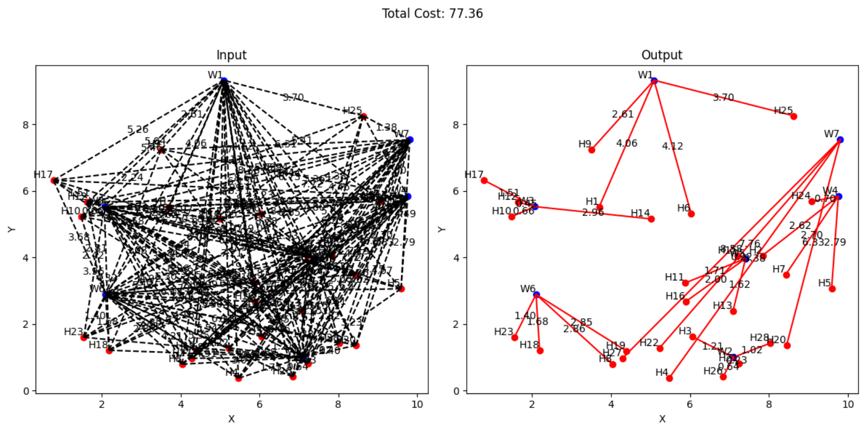

Figure 5: 7 wells, 4 houses per well

6. Large example with 10 wells and 10 houses per well. Runtime of around one tenth of a second.

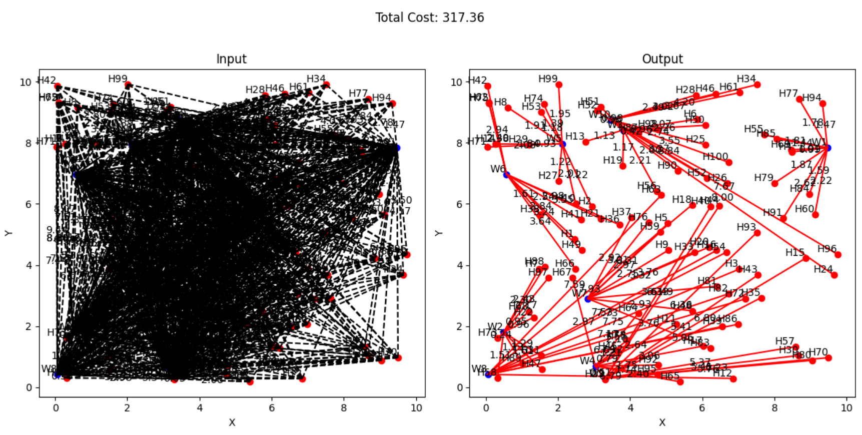

Figure 6: 10 wells, 10 houses per well

7. The final example an extreme case with 30 wells and 30 houses per well. The runtime of the algorithm for the graph of this size reaches one minute.

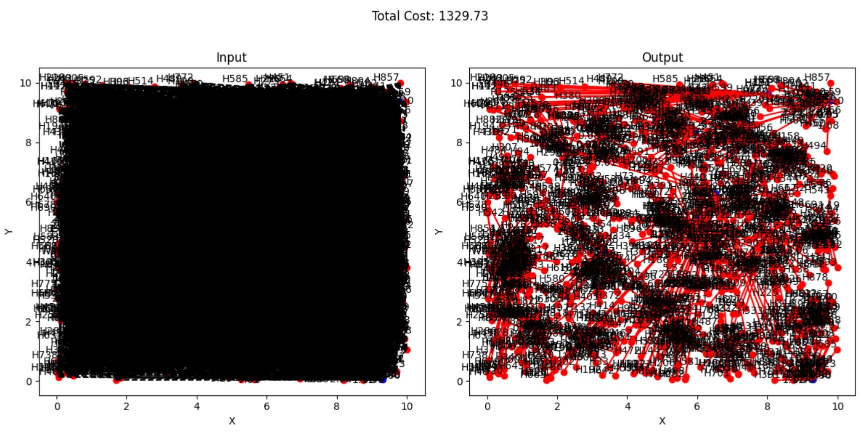

Figure 7: 30 wells, 30 houses per well

The above examples correctly assigned houses to wells with the cost minimized, confirming the algorithm's accuracy.

4 Time Complexity Measurements

We measured the execution time for each $(wells\_count, houses\_per\_well)$ pair, with $wells\_count \in [1, 30]$ and $houses\_per\_well \in [1, 30]$. The measurements were taken 100 times, and the results averaged. The resulting surface, depicted in green, is shown in Figure 8. We compared it with the predicted worst-time complexity $O((nk)^3)$, represented by the red surface.

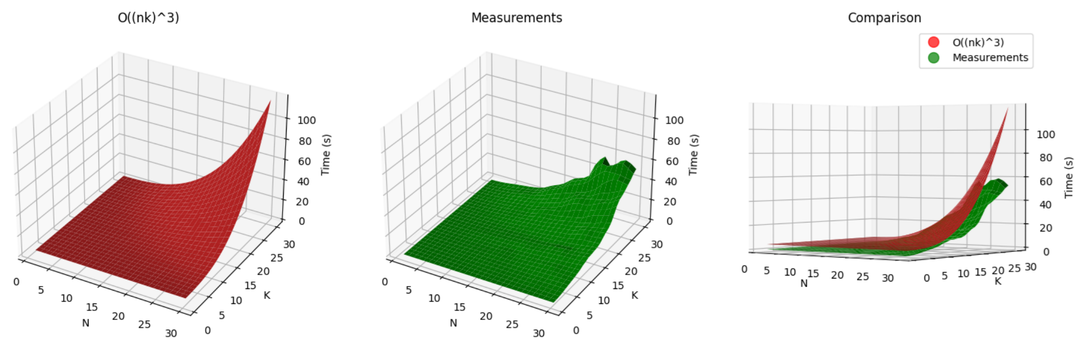

Figure 8: Time complexity measurements

The plots affirm that the worst-case runtime of the algorithm aligns with the expected $O((nk)^3)$ complexity.

5 Work Division

We strategically divided the workload to maintain an equal distribution of 50%, and both authors agree that this objective was successfully achieved.

Name	Responsibilities
Zofia	Implement the Hungarian algorithm
	Prepare initial documentation: • Correctness and optimality proofs • Mathematical formulation • Matrix algorithm version pseudo-code
	Improve code clarity and readability
	Conduct tests for correctness verification
	Prepare final documentation: • Test results • Division of work
Piotr	Implement the Hungarian algorithm
	Prepare initial documentation: • Time complexity proof • Problem formulation • Graph algorithm version pseudo-code
	Create visualizations for input, output, and time complexity
	Optimize Time Complexity with Label and Path Caching
	Prepare final documentation: • Execution instructions • Benchmarks, Time complexity

Table 1: Work division

6 Conclusions

In conclusion, the modified Hungarian algorithm implemented in this project has demonstrated its reliability and efficiency in solving the Wells Assignment problem. Rigorous testing across various scenarios affirms its correctness and scalability, with a time complexity of $O((nk)^3)$. The algorithm in itself however is extremely complex and requires a substantial amount of knowledge in rarely used mathematical concepts to fully grasp. Moreover, the absence of comprehensive resources delving into the optimization process, coupled with the insufficient availability of any correctness and time complexity proofs for crucial algorithmic steps, introduces challenges in implementing the algorithm and renders the verification of concepts exceptionally demanding. While the algorithm has proven effective in solving the Wells Assignment problem, its intricate nature may pose a barrier for those seeking a straightforward understanding and implementation.