



## Algorytmy zaawansowane

---

### Dokumentacja końcowa

Dawid Maksymowski

Artur Michalski

17 czerwca 2023

#### Streszczenie

Niniejszy dokument stanowi dokumentację końcową projektu dotyczącego rozwiązywania problemu studni. W celu znajdowania rozwiązań dla różnych instancji problemu został zaimplementowany algorytm węgierski, który został opisany w ramach dokumentacji wstępnej realizowanego projektu. Wszelkie poczynione zmiany względem dokumentacji wstępnej zostały przedstawione w tym dokumencie. Została również dodana instrukcja korzystania z programu określająca format plików wejściowych, dostępne dla użytkownika opcje oraz format plików wyjściowych. Ponadto zostały przeprowadzone serie testów mające na celu zweryfikowanie poprawności zaimplementowanego algorytmu oraz wpływ wartości parametrów  $N$  oraz  $K$  na uzyskiwany czas wykonania. W tym celu jako algorytm referencyjny wykorzystano implementację algorytmu węgierskiego dostępną w ramach biblioteki QuikGraph[1]. Wszelkie przeprowadzone testy zostały udokumentowane w niniejszym raporcie, a poczynione obserwacje oraz wyciągnięte wnioski przedstawione w ramach osobnej sekcji dokumentu.

# Spis treści

<b>1</b>	<b>Wprowadzenie</b>	<b>3</b>
<b>2</b>	<b>Zmiany w algorytmie</b>	<b>3</b>
2.1	Napotkane trudności . . . . .	4
2.1.1	Arytmetyka zmiennoprzecinkowa . . . . .	4
2.1.2	Aktualizacja grafu krawędzi ciasnych . . . . .	5
2.1.3	Sprawdzenie $N_p(S) == T$ i kolejna pętla . . . . .	5
<b>3</b>	<b>Program</b>	<b>6</b>
3.1	Format plików wejściowych . . . . .	6
3.2	Wywołanie programu . . . . .	6
3.3	Format plików wyjściowych . . . . .	7
<b>4</b>	<b>Przeprowadzone testy</b>	<b>7</b>
4.1	Metodyka przeprowadzonych testów . . . . .	7
4.2	Generowanie pojedynczej instancji problemu . . . . .	8
4.3	Przykładowa instancja problemu . . . . .	8
4.4	Poprawność działania algorytmu . . . . .	9
4.5	Wpływ liczby studni na czas wykonywania algorytmu . . . . .	9
4.6	Wpływ liczby domów zaopatrywanych przez pojedynczą studnię na czas wykonywania algorytmu . . . . .	10
<b>5</b>	<b>Wnioski</b>	<b>12</b>
<b>6</b>	<b>Podział prac</b>	<b>12</b>

# 1 Wprowadzenie

Niniejszy dokument stanowi podsumowanie wyników implementacji algorytmu węgierskiego, rozwiązującego problem **Studni** w następującym sformułowaniu:

**Studnie** Na płaszczyźnie znajduje się  $n$  studni oraz  $kn$  domów. Każda studnia zapewnia wodę  $k$  domom, a każdy dom może pobierać wodę tylko bezpośrednio ze studni (nie dopuszczamy pobierania wody za pośrednictwem innych domów). Należy znaleźć najtańsze sumaryczne doprowadzenie wody do domów, gdzie koszt doprowadzenia wody do domu  $d_i$  ze studni  $s_j$  jest równy odległości euklidesowej między punktami  $d_i$  i  $s_j$ .

## 2 Zmiany w algorytmie

Przypomnijmy skonstruowany wcześniej pseudokod algorytmu. Połączenie algorytmów 5 i 6 z Dokumentacji Wstępnej zostało przedstawione jako algorytm 1. Przedstawia ono główną ideę algorytmu, bez szczegółów implementacyjnych zaciemniających ogólny obraz.

Algorytm miał składać się z dwóch pętli - zewnętrznej oraz wewnętrznej. W każdej iteracji zewnętrznej pętli wielkość skojarzenia powiększa się o 1; na jej początku przygotowywane i resetowane są wszystkie struktury potrzebne do działania pętli wewnętrznej, po czym następuje poszukiwanie ścieżki powiększającej, i na końcu skojarzenie jest względem znalezionej ścieżki powiększane.

W pętli wewnętrznej natomiast główną strukturą, na której operujemy jest graf  $\vec{G}_p$ . Na jego podstawie, w pętli, budujemy drzewo naprzemienne o korzeniu w niepokrytym przez  $M$  wierzchołku  $u$  (w implementacji: `free_s`). Drzewo to budujemy w osobnym grafie, by mieć łatwo dostęp do jego krawędzi. Zbiory  $S$  oraz  $T$  natomiast stanowią szybki dostęp do osiągniętych w tym drzewie wierzchołków - stąd struktury te muszą być ze sobą dobrze zsynchronizowane. Dodając do drzewa kolejne krawędzie, należy pamiętać o dodaniu odpowiadających im wierzchołków również do zbiorów  $S$ ,  $T$ .

---

**Algorithm 1** Hungarian

---

**Input:**  $G = (V, E)$  $V = W \cup H$  - podział wierzchołków na klasy dwudzielności $w$  - funkcja wag krawędzi**Output:**  $M$  - skojarzenie doskonałe o minimalnym koszcie

```
1:  $M \leftarrow \emptyset$ 
2:  $p[] \leftarrow$  startowa funkcja potencjału
3:  $\vec{G}_p \leftarrow$  graf skierowany składający się z krawędzi ciasnych grafu  $G$ 
   (poprawność grafu utrzymywana w trakcie działania algorytmu)
4:
5: while  $M$  nie jest doskonałe do
6:    $u \leftarrow$  dowolny niepokryty wierzchołek z  $W$ 
7:    $S \leftarrow \{u\}$ 
8:    $T \leftarrow \emptyset$ 
9:   while  $path$  is NULL do
10:    if  $N_p(S) = T$  then
11:       $\Delta \leftarrow \min \{w(s, y) - p(s) - p(y) : s \in S, y \notin T\}$ 
12:       $p \leftarrow \text{ImprovePotential}(p, \Delta)$ 
13:    end if
14:     $y \leftarrow$  dowolny wierzchołek z  $N_p(S) - T$ 
15:    if  $y$  - niepokryty przez  $M$  then
16:       $path \leftarrow$  ścieżka od  $u$  do  $y$  ▷ Przerwanie pętli while
17:    else  $yz \in M$ 
18:       $S \leftarrow S \cup \{z\}$ 
19:       $T \leftarrow T \cup \{y\}$ 
20:    end if
21:  end while
22:   $M \leftarrow$  powiększ  $M$  względem  $path$ 
23: end while
24: return  $M$ 
```

---

## 2.1 Napotkane trudności

Ta sekcja opisuje trudności napotkane podczas implementacji algorytmu, które sprawiły, że zmienił nieznacznie swój kształt w stosunku do przewidywanego w dokumentacji wstępnej. Został również przedstawiony sposób poradzenia sobie z nimi a także usprawnienia samego algorytmu, które wynikły z ich napotkania.

### 2.1.1 Arytmetyka zmiennoprzecinkowa

Wagi krawędzi tworzonego w algorytmie grafu, z natury problemu Studni, są liczbami zmiennoprzecinkowymi. Operowanie na tych liczbach bezpośrednio, bez wzięcia pod uwagę niedokładności arytmetyki tych liczb, prowadzi do błędów lub całkowitego zatrzymania się algorytmu. Problem ten rozwiązaliśmy, korzystając z wbudowany w język C# typ liczbowy `decimal`. Wykorzystanie tego typu liczbowego rozwiązuje problem. Jako usprawnienie algorytmu sugerujemy skorzystanie z typu `double` o mniejszym rozmiarze oraz porównywanie liczb z określoną dokładnością, przy skorzystaniu z funkcji `Math.Round()`. Aby znaleźć odpowiedni poziom przybliżenia, który należy zastosować, należy przeanalizować algorytm pod kątem maksymalnej liczby operacji wykonywanych na wartościach potencjału wierzchołków (wykonują się one tylko w operacji w linii 12 pseudokodu. Inną opcją jest przetłumaczenie danych wejściowych na wersję dyskretną. W ostatecznej aplikacji można np. ograniczyć format przyjmowanych danych wprost do odległości (czyli wag w grafie), które muszą być liczbami całkowitymi. W wersji dyskretnego algorytmu problem ten nie występuje.

### 2.1.2 Aktualizacja grafu krawędzi ciasnych

Pominiętą podczas wstępnej analizy algorytmu kwestią było utrzymywanie w trakcie działania algorytmu poprawności grafu krawędzi ciasnych. Według pierwotnej myśli modyfikacja potencjału nie miała modyfikować dotychczas zbudowanego grafu  $\vec{G}_p$ , a jedynie *rozbudowywać* go o kolejne krawędzie. Nie jest to prawdą. Aktualizacja funkcji potencjału jest przeprowadzona w taki sposób, że:

- nie modyfikuje ona wyznaczonego dotychczas skojarzenia oraz
- funkcja potencjału pozostaje poprawna, tj.  $\forall uv \in E(G) p(u) + p(v) \leq w(uv)$ .

Okazuje się jednak, że niektóre z krawędzi grafu  $\vec{G}_p$  mogą z niego po tej modyfikacji *wypaść*. Oznaczmy jako  $S$  i  $T$  zbiory, między którymi modyfikujemy potencjał oraz  $\bar{S}$  i  $\bar{T}$  jako ich dopełnienia w odpowiadających im klasach dwudzielności. Sposób, w jaki budowany jest graf  $\vec{G}_p$  i zbiory  $S, T$  sprawiają, że mamy pewność co do następującego:

- krawędzie między  $S$  i  $T$  pozostają w  $\vec{G}_p$  (zgodnie z modyfikacją),
- krawędzie między  $\bar{S}$  i  $\bar{T}$  pozostają w  $\vec{G}_p$  (te wierzchołki nie są modyfikowane),
- krawędzie z  $S$  do  $\bar{T}$  nie istnieją,
- krawędzie z  $T$  do  $\bar{S}$  nie istnieją,
- krawędzie z  $\bar{T}$  do  $S$  nie istnieją (można dowieść nie wprost),
- krawędzie z  $\bar{S}$  do  $T$  **mogą wypaść** z  $\vec{G}_p$

Możliwość wypadnięcia krawędzi sprawia, że każdorazowe ulepszenie potencjału wymaga sprawdzenia wszystkich krawędzi z  $\bar{S}$  do  $T$ . Operacja ta niestety ma złożoność  $O(n^2)$ .

### 2.1.3 Sprawdzenie $N_p(S) == T$ i kolejna pętla

Sprawdzenie warunku wewnętrznej pętli, czy wszystkie wierzchołki sąsiadujące z wierzchołkami z  $S$  znajdują się w  $T$ , wymaga w naiwnej implementacji przejrzenia wszystkich krawędzi incydentnych – złożoność  $O(n^2)$ . Warto jednak zwrócić uwagę, że jeśli dodamy do  $T$  wszystkich sąsiadów wierzchołka  $s \in S$ , to nie ma potrzeby kolejny raz go sprawdzać. Stąd w implementacji użyto tablicę wartości bool dla każdego  $s \in S$ . Jest ona ustawiana na **true**, po dodaniu wszystkich sąsiadów wierzchołka do  $T$ . Dzięki takiemu zabiegowi złożoność sprawdzenia jest liniowa.

W konsekwencji tego usprawnienia należy wprowadzić jeszcze jedną modyfikację. Zauważmy, że na skutek operacji polepszenia potencjału do grafu  $\vec{G}_p$  mogły zostać dodane więcej niż jedna krawędzie. Wg pseudokodu tylko jedna z tych krawędzi zostałaaby dodana do drzewa naprzemiennego (oraz, równocześnie, tylko jeden wierzchołek  $y$  zostałaby dodany do  $T$  (linia 14)). Oczywiście, jeśli mamy kilka wierzchołków  $y_1, y_2, \dots$  z  $N_p(S) - T$ , to zostaną one *odnalezione* w kolejnych iteracjach wew. pętli. Nie ma jednak potrzeby kolejnego wywoływania sprawdzenia warunku tej pętli, skoro wiemy, jaki będzie jej wynik. Możemy „zbierać” wszystkie wierzchołki (oraz krawędzie), które będziemy dodawać do  $T$  (do drzewa) w wyniku modyfikacji potencjału oraz wykonywać dla nich wszystkich kod z linii 14-20 w pętli.

Modyfikacja ta jest natomiast **niezbędna** do zastosowania, jeśli chcemy skorzystać z tablicy bool wspomnianej wyżej. W przeciwnym przypadku oznaczylibyśmy dany wierzchołek  $s \in S$  jako *sprawdzony* nawet, kiedy modyfikacja potencjału spowodowałaby dodanie do niego więcej niż jednego sąsiada.  $s$  nie zostałby ponownie sprawdzony w warunku pętli w przypadku, kiedy jego wartość bool została już ostawiona na **true**.

## 3 Program

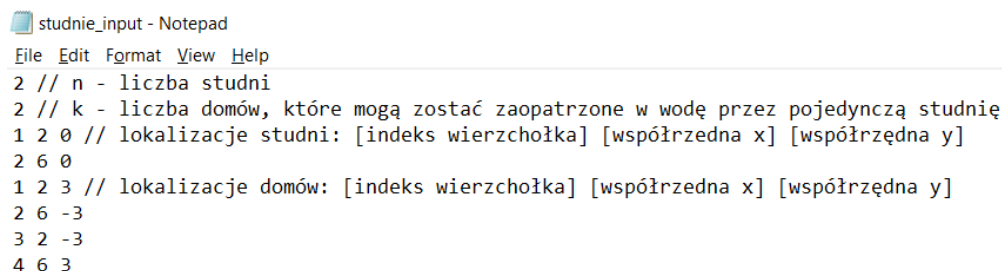
Zaimplementowany program to aplikacja konsolowa uruchamiana poprzez wiersz poleceń. Format plików wejściowych, wspieranych opcji, oraz format plików wyjściowych został szczegółowo opisany w kolejnych podsekcjach dokumentu.

### 3.1 Format plików wejściowych

Wejście programu stanowi plik zawierający następujące informacje:

- liczność zbioru reprezentującego studnie ( $n$ )
- liczba domów, do których może być doprowadzona woda przy użyciu jednej studni ( $k$ )
- lista zawierająca lokalizacje wszystkich studni
- lista zawierająca lokalizacje wszystkich domów

Przykładowy plik wejściowy w poprawnym formacie został przedstawiony na rysunku 1:



```
studnie_input - Notepad
File Edit Format View Help
2 // n - liczba studni
2 // k - liczba domów, które mogą zostać zaopatrzone w wodę przez pojedynczą studnię
1 2 0 // lokalizacje studni: [indeks wierzchołka] [współrzędna x] [współrzędna y]
2 6 0
1 2 3 // lokalizacje domów: [indeks wierzchołka] [współrzędna x] [współrzędna y]
2 6 -3
3 2 -3
4 6 3
```

Rysunek 1: Przykładowy plik wejściowy

Wywołanie programu odbywa się poprzez uruchomienie pliku wykonywalnego i wskazanie ścieżki zawierającej instancję problemu jaki należy rozwiązać.

### 3.2 Wywołanie programu

W celu uruchomienia programu należy przejść do folderu w którym znajduje się plik wykonywalny aplikacji i korzystając z wiersza poleceń wywołać następującą komendę:

```
./Application.exe -i 2_2.txt
```

Spowoduje to uruchomienie algorytmu dla pliku wejściowego o nazwie 2\_2.txt. Przy wywoływaniu programu wspierane są następujące opcje:

- **-i (input)** ścieżka do pliku wejściowego skąd wczytywana jest instancja problemu
- **-o (output)** ścieżka do pliku wyjściowego gdzie zapisywane jest rozwiązanie zadanej instancji problemu (opcjonalne)
- **-n (n)** liczba studni (opcjonalne, domyślnie = 2)
- **-k (k)** liczba domów do których może być dostarczana woda z wykorzystaniem jednej studni (opcjonalne, domyślnie = 2)
- **-v (v)** czy powinny być wyświetlane logi informujące o aktualnym stanie programu (opcjonalne, domyślnie opcja wyłączona)
- **-s (s)** czy znalezione rozwiązanie ma być wypisywane w konsoli (opcjonalne, domyślnie opcja wyłączona)

Kolejność wprowadzania argumentów wejściowych nie ma znaczenia. Każda z opcji musi zostać wprowadzona indywidualnie (nie jest wpierane łącznie opcji na przykład w postaci *-vs*). W przypadku gdy została zdefiniowana wartość dla  $k$  lub  $n$  zostanie wygenerowana nowa instancja problemu i zapisana pod wskazaną ścieżką do pliku wyjściowego. W przypadku gdy wartość dla  $k$  lub dla  $n$  nie zostanie wyspecyfikowana zostanie użyta wartość domyślna równa 2. W przypadku gdy nie została zdefiniowana nazwa pliku wyjściowego otrzymane rozwiązanie zapisywane jest w tym samym folderze oraz z taką samą nazwą jak plik wejściowy ale z sufiksem *output.txt*.

Pełne wywołanie programu (wraz z generowaniem instancji problemu dla 5 studni, 30 domów, wyświetlaniem logów, wypisywaniem rozwiązania na konsolę oraz zapisem rozwiązania pod wskazaną nazwą pliku):

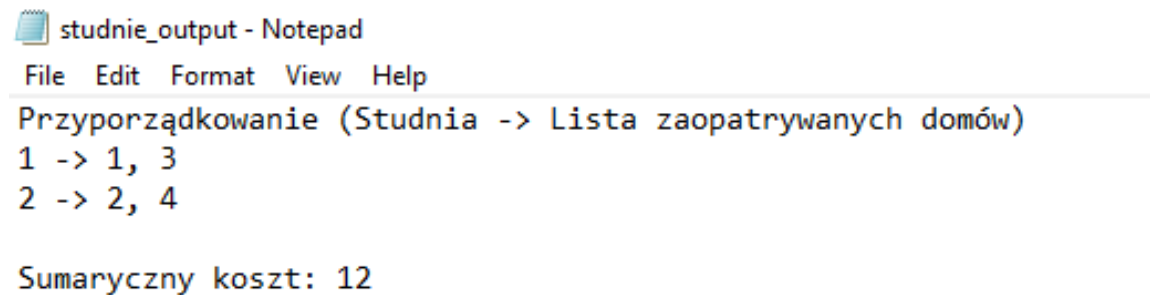
```
./Application.exe -i 5_6_generated.txt -o 5_6_solution -n 5 -k 6 -v -s
```

### 3.3 Format plików wyjściowych

Wyjście programu stanowi plik zawierający następujące informacje:

- lista zawierająca kolejne studnie oraz zaopatrywane przez nią domy
- minimalny sumaryczny koszt dostarczenia wody do każdego domu (rozumiany jako suma odległości euklidesowych wszystkich połączeń)

Przykładowy plik wejściowy w poprawnym formacie został przedstawiony na rysunku 2:



Rysunek 2: Przykładowy plik wyjściowy

## 4 Przeprowadzone testy

W celu zweryfikowania skuteczności działania opracowanego algorytmu zostały przeprowadzone serie testów weryfikujące zarówno poprawność otrzymywanych przez algorytm rezultatów jak i czasu wykonania w porównaniu z wersją biblioteczną algorytmu węgierskiego. Jako rozwiązanie referencyjne względem którego zostało przeprowadzone porównanie algorytmów została wykorzystana implementacja algorytmu węgierskiego dostępna w ramach biblioteki QuikGraph[1]. Uzyskane wyniki zostały przedstawione w kolejnych podsekcjach dokumentu.

### 4.1 Metodyka przeprowadzonych testów

Pojedynczy test polegał na wygenerowaniu dla ustalonych wartości  $N$  (liczba studni) oraz  $K$  (liczba domów, które mogą być zaopatrywane w wodę przez jedną studnię) pewnej liczby różnych instancji problemu. Różnicowanie generowanych instancji odbywało się poprzez zdefiniowanie różnej wartości ziarna inicjalizującego generator liczb losowych. Każda tak wygenerowana instancja problemu była

następnie przekazywana jako wejście dla zaimplementowanej wersji algorytmu węgierskiego oraz wersji bibliotecznej. W trakcie znajdowania przez algorytmy rozwiązań realizowane były również pomiary czasu działania każdego z algorytmów. Dla par wartości  $N$  oraz  $K$  rozpatrywana została następująca liczba testów:

$N$	$K$	Liczba wykonanych testów
[1,9]	[1,10]	10
10,20	[1,10]	10
50	[1,5]	10
100	[1,4]	5
200	[1,3]	5
500	[1,2]	5
1000	1	5

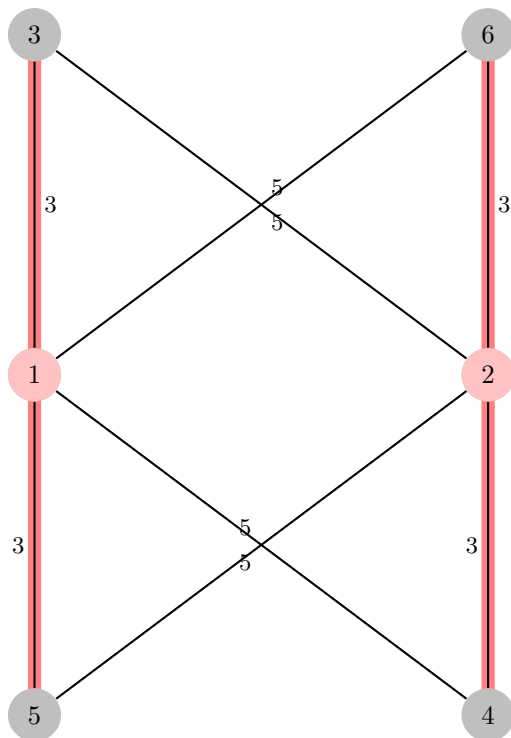
Wszystkie przeprowadzone eksperymenty mogą zostać zreprodukowane uruchamiając należący do solucji projekt TestRunner. Ponadto w celu zbadania wpływu parametrów  $N$  oraz  $K$  na złożoność całego algorytmu przeprowadzono dwie dodatkowe serie testów.

## 4.2 Generowanie pojedynczej instancji problemu

Na potrzeby przeprowadzanych testów został również zaimplementowany generator instancji problemów. Wejście generatora stanowią parametry  $N$  i  $K$  mówiące o rozmiarze generowanej instancji problemu oraz wartość ziarna inicjalizującego generator liczb losowych. Następnie generowana jest para współrzędnych z zakresu  $(-100,100)$  dla kolejno  $N$  studni oraz  $N \cdot K$  domów. Tak otrzymana instancja problemu może następnie zostać zapisana do pliku, po czym obliczana jest dla niej macierz dystansów, zawierająca informacje o odległości pomiędzy dowolnym domem, a dowolną studnią. Obliczona macierz stanowi następnie wejście dla zaimplementowanego algorytmu węgierskiego.

## 4.3 Przykładowa instancja problemu

Rozpatrzmy graf przedstawiony poniżej będący instancją problemu dla  $N=2$  oraz  $K=2$ .





Kolorem czerwonym oznaczono wierzchołki należące do zbioru studni, podczas gdy kolor szary oznacza przynależność wierzchołka do zbioru reprezentującego domy. Zgodnie z poczynionymi wcześniej obserwacjami zbiory te tworzą dwie oddzielne klasy dwudzielności, a problem studni jest tożsamy z problemem znajdowania skojarzenia doskonałego o minimalnym łącznym koszcie w grafie dwudzielnym rozszerzonym tak, aby liczność zbioru zawierającego wierzchołki reprezentujące studnie odpowiadała liczności zbioru zawierającego wierzchołki reprezentujące domy. Dla przedstawionego grafu rozwiązanie optymalne stanowi przypisanie wierzchołków 3 i 5 do studni oznaczonej wierzchołkiem 1 oraz przypisanie wierzchołków 4 i 6 do studni oznaczonej wierzchołkiem 2. Łączny koszt takiego przypisania wynosi 12. Zaimplementowany algorytm znajduje poprawne przypisanie domów do studni oraz poprawnie wyznacza łączny koszt.

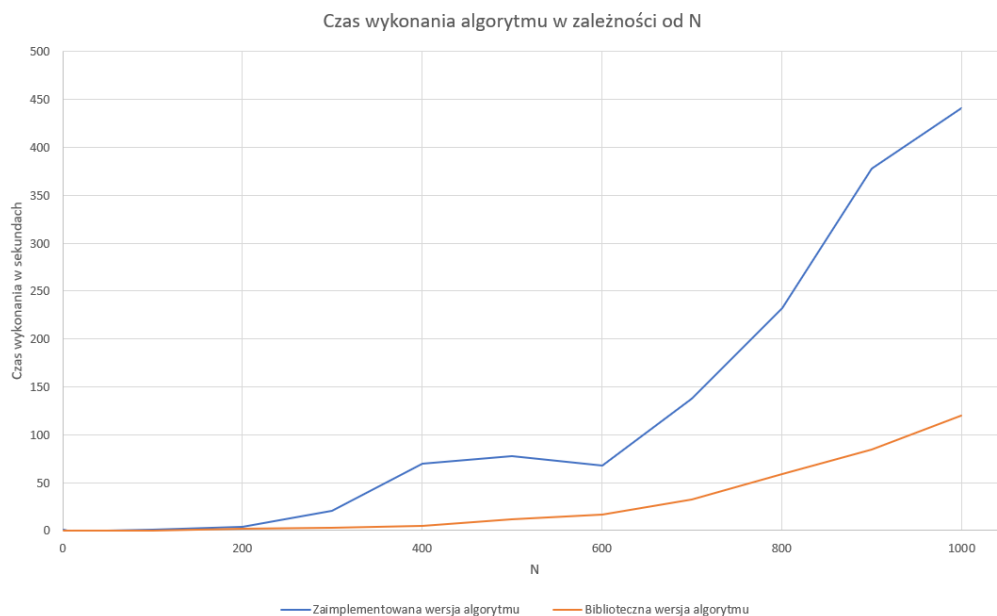
#### 4.4 Poprawność działania algorytmu

Poprawność algorytmu została zweryfikowana poprzez zestawienie łącznego, sumarycznego kosztu skojarzenia doskonałego wyznaczonego przez zaimplementowaną wersję algorytmu z biblioteczną wersją algorytmu. Dla rozpatrywanych przykładów nie zaobserwowano różnic w kosztach rozwiązań. Zauważono jednak różnicę w bezpośrednim przyporządkowaniu domów do studni. Różnice te jednak wynikają wyłącznie z istnienia dla niektórych instancji problemu więcej niż jednego optymalnego rozwiązania, a sumaryczny koszt doprowadzenia wody do studni jest identyczny. W początkowych fazach prac nad algorytmem do weryfikacji poprawności stosowano również podejście brute force polegające na rozpatrywaniu wszystkich możliwych permutacji zbioru reprezentującego domy, co definiowało przyporządkowanie domów do studni. Takie rozwiązanie ze względu na złożoność  $O(K*N)$  pozwoliło jednakże wyłącznie na sprawdzenie w zadowalającym czasie poprawności wyłącznie dla niewielkich instancji problemu, gdzie iloczyn wartości  $K$  oraz  $N$  nie przekraczał wartości 13.

#### 4.5 Wpływ liczby studni na czas wykonywania algorytmu

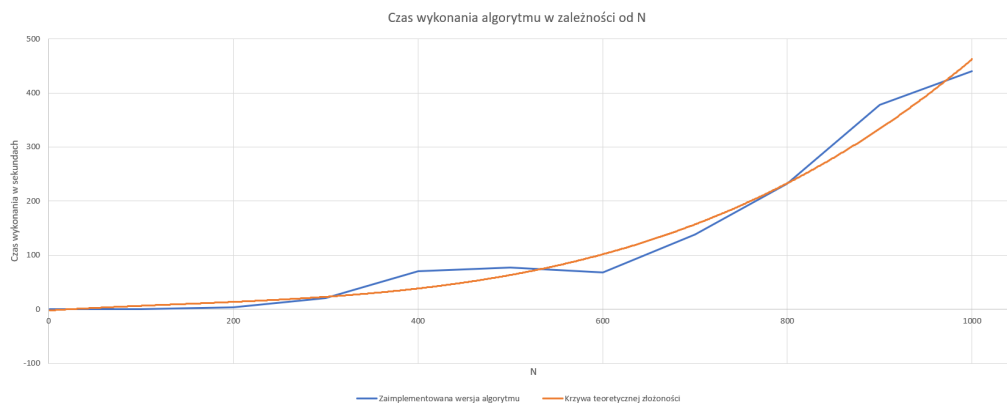
Aspektem, który był również brany pod uwagę podczas testów był wpływ liczby studni (wielkości parametru  $N$ ) na złożoność algorytmu. W tym celu przeprowadzono serię testów, gdzie dla ustalonej wartości  $K=1$  mierzono czas działania algorytmu dla wartości  $N$  z przedziału  $(1, 1000)$ . Uzyskane wyniki zostały zebrane w poniższej tabeli oraz przedstawione na wykresie 3.

N	Zaimplementowana wersja[s]	Wersja biblioteczna[s]
1	0,734	0,099
5	0,019	0,009
10	0,006	0,001
15	0,056	0,001
20	0,107	0,001
25	0,038	0,002
50	0,213	0,030
100	0,844	0,300
200	3,618	2,163
300	20,774	3,255
400	69,846	4,506
500	77,332	11,826
600	67,737	16,424
700	137,850	32,606
800	232,263	59,032
900	377,936	84,980
1000	440,708	119,835



Rysunek 3: Wpływ wartości parametru N na czas wykonania algorytmu

Dla uzyskanych wartości pomiarowych dopasowano z wykorzystaniem metody najmniejszych kwadratów krzywą określającą teoretyczną złożoność algorytmu, co przedstawia rysunek 4. Przy wyznaczaniu krzywej zgodnie z dokumentacją wstępną przyjęto założenie, że złożoność algorytmu jest rzędu pewnego wielomianu trzeciego stopnia.

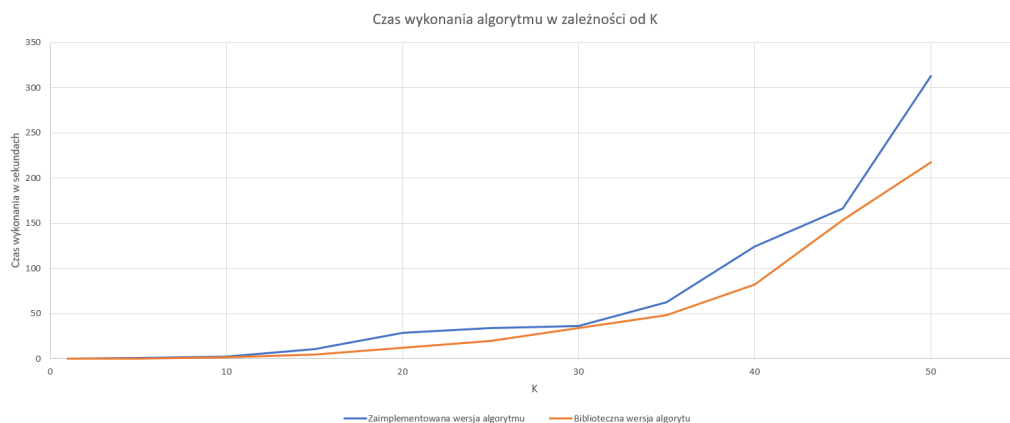


Rysunek 4: Teoretyczna krzywa złożoności w zależności od N

#### 4.6 Wpływ liczby domów zaopatrywanych przez pojedynczą studnię na czas wykonywania algorytmu

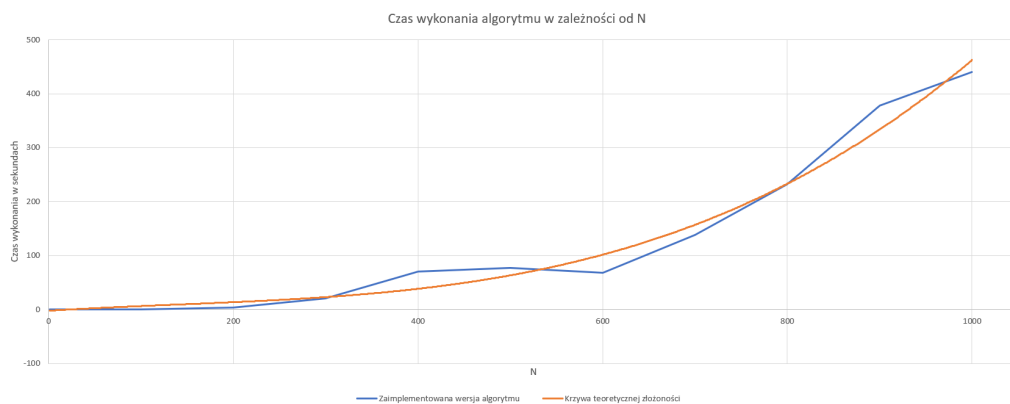
Innym aspektem, również wpływającym na złożoność algorytmu jest parametr określający jak wiele domów może być zaopatrywanych w wodę przez pojedynczą studnię. W celu zbadania jego wpływu na czas wykonania algorytmu przeprowadzono serię testów, gdzie dla ustalonej wartości  $N=10$  mierzono czas działania algorytmu dla wartości  $K$  z przedziału  $(1, 50)$ . Uzyskane wyniki zostały zebrane w poniższej tabeli oraz przedstawione na wykresie 5.

K	Zaimplementowana wersja[s]	Wersja biblioteczna[s]
1	0,240	0,171
5	1,240	0,378
10	2,710	1,519
15	11,130	4,548
20	28,590	12,021
25	34,400	19,908
30	36,070	34,417
35	62,490	48,682
40	124,160	82,530
45	166,450	153,730
50	313,080	217,234



Rysunek 5: Wpływ wartości parametru K na czas wykonania algorytmu

Dla uzyskanych wartości pomiarowych dopasowano z wykorzystaniem metody najmniejszych kwadratów krzywą określającą teoretyczną złożoność algorytmu, co przedstawia rysunek 6. Przy wyznaczaniu krzywej zgodnie z dokumentacją wstępną przyjęto założenie, że złożoność algorytmu jest rzędu pewnego wielomianu trzeciego stopnia.



Rysunek 6: Teoretyczna krzywa złożoności w zależności od K

## 5 Wnioski

Algorytm węgierski pozwala na rozwiązywanie problemów polegających na przypisaniu pewnych zadań, zasobów do pracowników w sposób optymalny. Należy zwrócić uwagę, że rozwiązanie polegające na rozpatrzeniu wszystkich możliwych przypisań wiąże się ze złożonością  $O(n!)$ , gdzie przez  $n$  rozumiana jest liczba dostępnych zadań/zasobów. Tymczasem złożoność algorytmu węgierskiego po modyfikacjach Emdondsa i Karpa wynosi  $O(n^3)$ .

Pierwszym i największym wnioskiem z naszej pracy nad analizą algorytmu jest to, że jest on trudny! Dogłębne zrozumienie działania tego typu algorytmu grafowego jest czasochłonne, a jednocześnie niezbędne do tego, żeby możliwe byłoby zaprojektowanie go w zakładanej, optymalnej złożoności pesymistycznej  $O(n^3)$ . Niewątpliwie, większe doświadczenie w pracy teoretycznej z grafami oraz algorytmami na nich byłoby pomocne. To oraz nieprzewidzenie faktu, że graf krawędzi ciasnych może *tracić* krawędzie po modyfikacji potencjału wierzchołków (jak opisano w 2.1.2) sprawiło, że zakładana pesymistyczna złożoność obliczeniowa najpewniej nie została osiągnięta. Ze względu na operację przeglądania wszystkich krawędzi grafu skierowanego wewnątrz pętli wewnętrznej, jest ona rzędu  $O(n^4)$ . Wydaje się możliwa implementacja algorytmu z całkowitym pominięciem budowy grafu  $\overrightarrow{G_p}$ , co mogłoby rozwiązać problem. Niestety ze względu na ograniczony czas, nie zostało to przetestowane.

Innym cennym wnioskiem z pracy jest to, że błędy numeryczne wynikające z wykorzystywanej w komputerach arytmetyki zmiennoprzecinkowej są ważne i należy brać je pod uwagę – albo implementować algorytmy w wersjach dyskretnych.

## 6 Podział prac

Dawid Maksymowski:

- Główny algorytm
- Usprawnienia w algorytmie i poprawki złożoności
- Badanie poprawności algosa
- Przeprowadzenie i opracowanie testów
- Zaprojektowanie architektury
- Opis zmian względem dokumentacji wstępnej

Artur Michalski:

- Opis złożoności oraz dowód poprawności algorytmu
- Instrukcja użytkownika - opis wejścia, wyjścia, wspieranych opcji
- Implementacja TestRunnera pozwalającego na automatyzację testów
- Przeprowadzenie i opracowanie testów
- Implementacja interfejsu użytkownika
- Implementacja algorytmu brute force
- Integracja z biblioteczną wersją algorytmu węgierskiego
- Implementacja generatora plików wejściowych
- Zaprojektowanie architektury

## Literatura

- [1] Alexandre Rabérin. Quikgraph - generic library for graph data structures.  
<https://github.com/KeRNeLith/QuikGraph>.