Sprawozdanie z laboratorium Optymalizacji Kombinatorycznej: Problem Job-shop wykonany metodą wielokrotnego wspinania się (hill climbing)

Jakub Gosławski 141222 jakub.goslawski@student.put.poznan.pl
Michał Wiśniewski 141355 michal.janu.wisniewski@student.put.poznan.pl

22 listopada 2019

1 Implementacja

Implementacja problemu została wykonana obiektowo w języku C++. Kod dzieli się na trzy zasadnicze części, interpretację plików wejściowych, algorytm wyznaczający nowe rozwiązanie oraz algorytm optymalizacyjny wybierający i analizujący rozwiązania metodą wielokrotnego wspinania się (Hill Climbing).

```
while (getline( &: input,  &: line) && ((maxjobs == -1) || jobID < maxjobs))
{
    if (line.empty()) {continue;}
    ss<<line;
    Job j(jobID);

    while(ss >> temp)
    {
        int machine = temp;
        int duration;
        ss>>duration;
        Task task(machine, duration, jobID);
        j.add_Task(task);
    }

    this->add_Job(j);

    ++jobID;
    ss.clear();

}
```

Rysunek 1: Interpretacja danych instancji typu orlib

Obiekt Factory (fragment konstruktora przedstawiony powyżej) składa się z obiektów Job, które składają się z obiektów Task, wszystkie przechowywane w wektorach, aby zachować kolejność przedstawioną w pliku wejściowym.

Algorytm wyznaczający nowe rozwiązanie to zmodyfikowana wersja algorytmu zachłannego, który za parametr przyjmuję permutację zadań (Jobs) i według niej przypisuje kolejne zadania (Tasks). Algorytm optymalizacyjny rozpoczyna się włączeniem pomiaru czasu i wygenerowaniem pierwszego rozwiązania na podstawie wcześniej wylosowanej permutacji. Następnie generowana jest następna wg porządku leksykograficznego permutacja zadań, i dopóki wynik rozwiązania się ulepsza, tak długo dokonuje kolejnych prób z następnymi permutacjami i przyjmuje je jako lokalne maksimum. Następnie algorytm wraca do pierwotnie wylosowanej permutacji i porusza się tym razem w drugą stronę (generując poprzednie wg porządku leksykograficznego permutacje zadań), tak długo jak wynik będzie się ulepszał. O ile upłynięty czas jest krótszy aniżeli ten z argumentu funkcji algorytmu, zostaje wygenerowane nowe losowe rozwiązanie i względem niego powtarzane są powyższe kroki wpisania się w obie strony. Ponieważ generowanie nowych fabryk przy każdym rozwiązaniu wiązałoby się z ogromną złożonością pamięciową, algorytm optymalizacyjny pracuje na

```
127     void greedy_solve(){
128         int time = 0;
129         int next_time = 0;
130         vector<Machine> machines;
131         for (int i=0; i<number_of_machines; i++){
132             machines.push_back(Machine(i));
133         }
134         int temp;
135         int done = 0;
136         int task_n;
137         while (done >=time) {
138             next_time = (int) time;
139             for (auto &j : this->jobs) {
140                 task_n = j.get_curr_task();
141                 if (task_n >= number_of_machines) { continue; }
142                 auto &t = j.get_Task(task_n);
143                 temp = t.get_machine_ID();
144                 if (machines[temp].check_if_free(time) && j.get_end_time() <= time) {
145                     t.set_start_time(time);
146                     machines[temp].set_end_time( time: t.get_duration() + time);
147                     machines[temp].set_busy();
148                     j.next_task();
149                     j.set_end_time( time: t.get_duration() + time);
150                     if (done < t.get_duration() + time) { done = t.get_duration() + time; }
151 
152                 }
153                 if ((   machines[t.get_machine_ID()].check_if_free(time) &&
154                         machines[t.get_machine_ID()].get_end_time() < next_time )||
155                         next_time == time){
156                    if(time <= machines[t.get_machine_ID()].get_end_time()) {next_time = machines[t.get_machine_ID()].get_end_time();}
157                 }
158                 if (j.get_end_time() < next_time && j.get_end_time() > time) {next_time = j.get_end_time();}
159             }
160             if (next_time != time) {
161             time = next_time;
162             }else{
163                 time++;
164             }
165         }
166         for (auto &j : this->jobs){
167             if (exec_len < j.get_end_time()){
168                 exec_len = j.get_end_time();
169             }
170         }
171     }
```

Rysunek 2: Algorytm optymalizacyjny

jednej wczytanej fabryce i przed kolejnym rozwiązaniem wywoływana jest funkcja resetująca. Do pomiaru bieżącego czasu wykorzystywany jest system_clock z biblioteki std::chrono. Losowanie wykorzystuje funkcję shuffle oraz deflaut_radnom_engine z biblioteki <random>, a przemieszanie się po kolejnych permutacjach funkcje next_permutation i prev_permutation z biblioteki <algorithm>.

1.1 Użycie

Program wykonuje się z linii poleceń i przyjmuje parametry: nazwa pliku, format instancji (o - orlib, t -tailard), oraz opcjonalną liczbę wczytanych pierwszych zadań, a zapisuje rozwiązanie w nowym pliku o nazwie [nazwa instancji]_answer.txt.

2 Testy

2.1 Testy jakości

Testy czasu wykorzystywały klasę high_resolution_clock z biblioteki std::chrono i mierzyły czas z dokładnością 1 nanosekundy. Program wykorzystany do testów na każdą fabrykę wykonywał 1000 prób i zapisywał od razu uśredniony wynik do pliku tekstowego.

Testy czasu zostały wykonane na dwóch maszynach:

Tablica 1: Testy czasu wykonane dla instancji tai20-25, w nanosekundach

Instancja	Średni czas wykonania na maszynie 1	Średni czas wykonania na maszynie 2
tai20	295108	294899
tai21	427876	431494
tai22	392651	380268
tai23	371395	370940
tai24	414031	399364
tai25	395496	396084

1. Maszyna Wirtualna Linux Kubuntu 64-bit, 4 GB pamięci RAM
2. UnixLab

Obie maszyny wykazały bardzo podobne wyniki. Zdecydowanie najszybciej wykonywała się instancja tai20, wszystkie pozostałe wykonywały się około 100 mikrosekund szybciej.

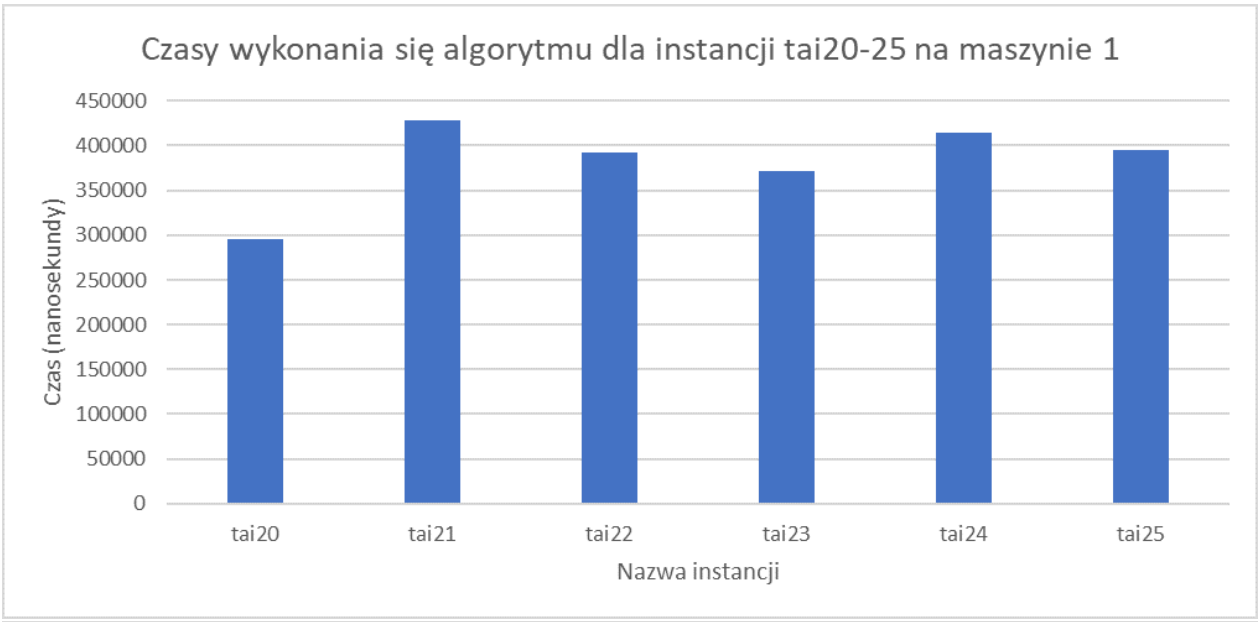
Może mieć to związek z wyznaczonym dolnym ograniczeniem wartości funkcji celu, gdyż i ten parametr był zdecydowanie najniższy dla tai20 spośród badanych instancji, jednak nie możemy tego jednoznacznie stwierdzić na podstawie tego jednego faktu.

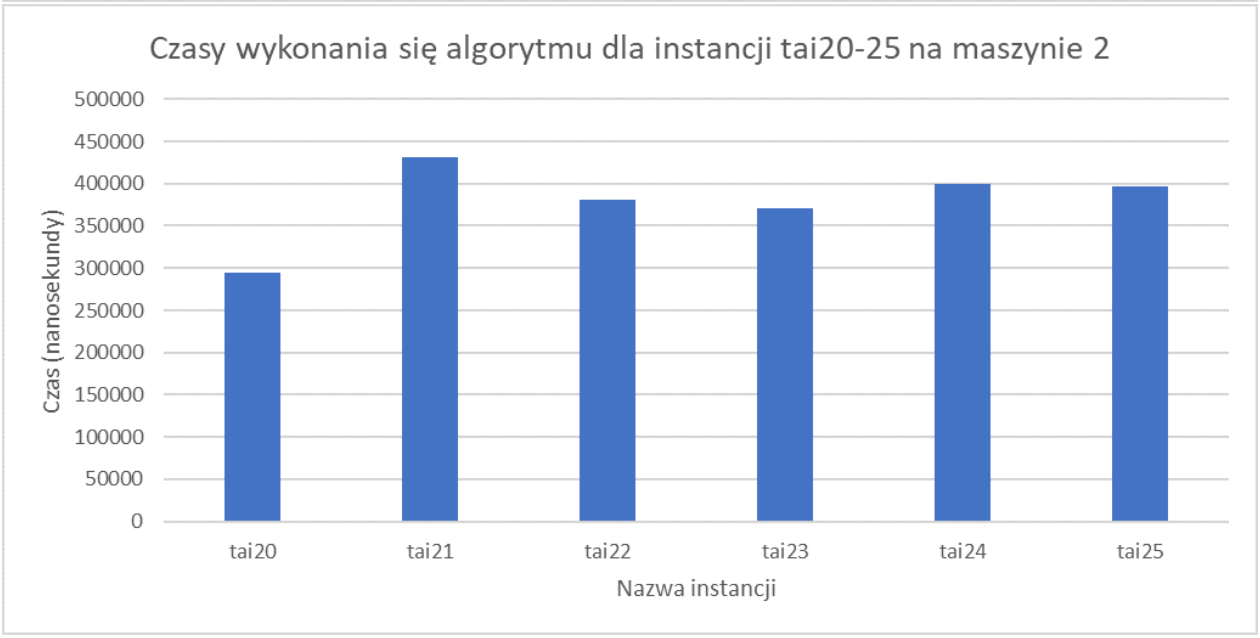
Tablica 2: Testy czasu dla różnych ilości wczytanych pierwszych zadań dla instancji tai25, w nanosekundach

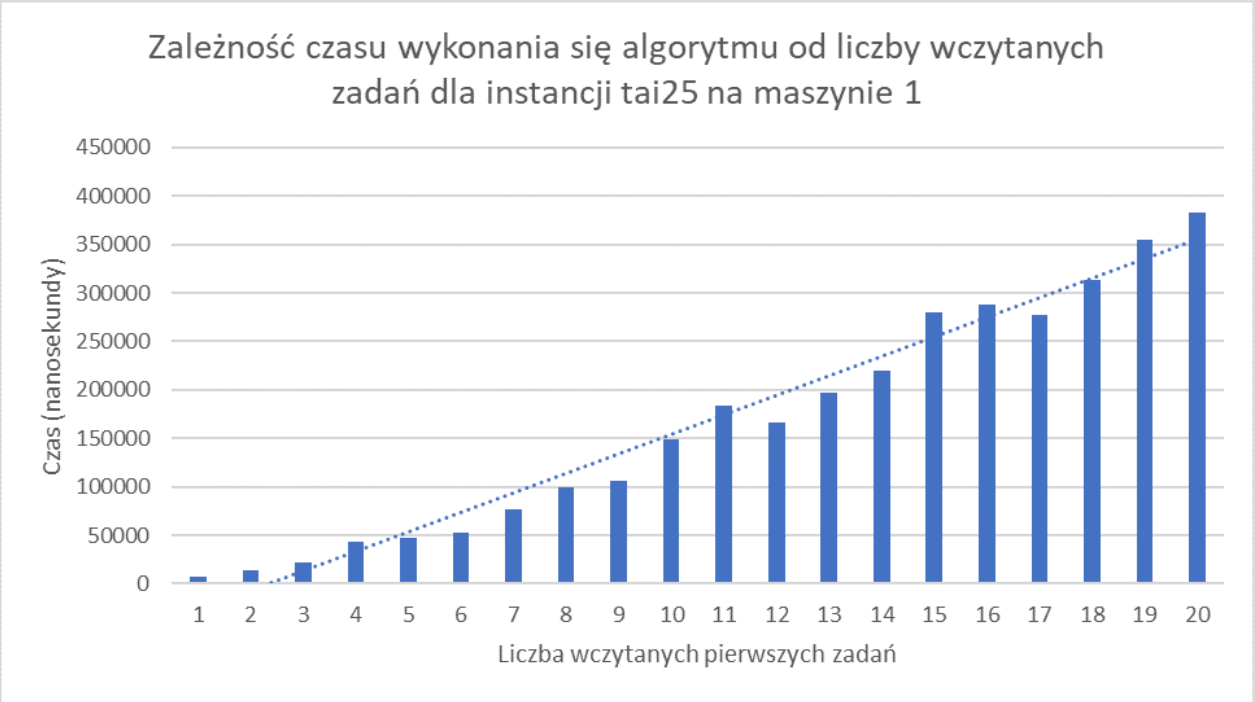
Liczba wczytanych pierwszych zadań	Średni czas wykonania na maszynie 1	Średni czas wykonania na maszynie 2
1	7054	7631
2	13431	13346
3	22463	23135
4	42760	42685
5	47133	49130
6	52133	53746
7	76962	77403
8	98902	97968
9	106721	112050
10	149067	150758
11	183171	183362
12	166781	174118
13	196710	197134
14	219794	223420
15	279682	277783
16	288087	275100
17	277483	287026
18	313943	332040
19	354713	402332
20	383007	404426

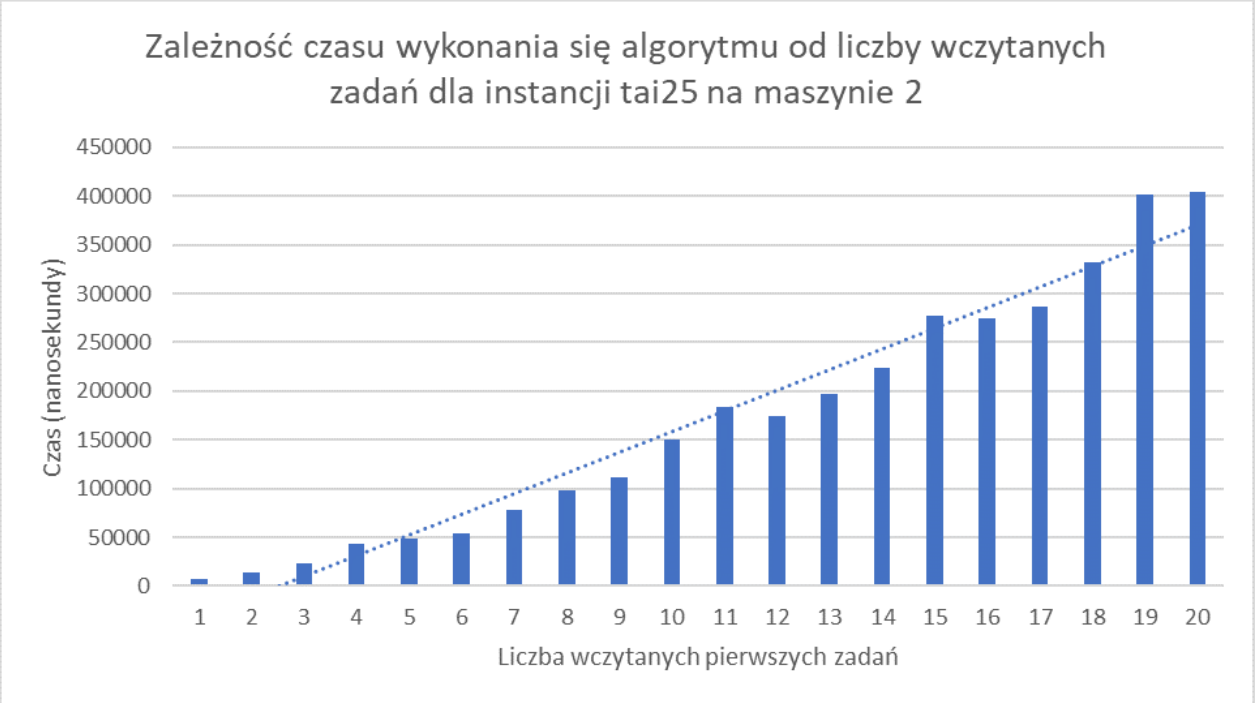
```cpp
int main()
{
    typedef chrono::high_resolution_clock clock;
    clock::time_point start, end;
    ofstream output;
    int testy = 1000;
    output.open("testy.txt");
    output << " file pomiar1 pomiar 2 pomiar 3... in nanoseconds" << endl ;
    string tab[6] = {"tai20.txt", "tai21.txt", "tai22.txt", "tai23.txt", "tai24.txt", "tai25.txt"};
    for (auto f : tab)
    {
        output << f << " ";
        long long result = 0;
        for (int i = 0; i < testy; ++i)
        {
            Factory factory(f, 't');

            start = clock::now();
            factory.greedy_solve();
            end = clock::now();
            result += std::chrono::duration_cast<std::chrono::nanoseconds>(end - start).count() / testy;



        }
        output << result << " ";
        output << endl ;

    }
    output << endl ;
    long long result = 0;
    for (int jobs = 1; jobs <=20 ; ++jobs)
    {   result = 0;
        if (jobs == 20) {++jobs;}
        output << jobs << " ";
        for (int i = 0; i < testy; ++i)
        {
        Factory factory("tai25.txt", 't', jobs);
        start = clock::now();
        factory.greedy_solve();
        end = clock::now();
        result += std::chrono::duration_cast<std::chrono::nanoseconds>(end - start).count() / testy;

        }
        output << result << " ";
        output << endl;

    }
```

Rysunek 3: Kod wykorzystany do wykonania pomiarów czasu


Czasy wykonania się algorytmu dla instancji tai20-25 na maszynie 1
Czas (nanosekundy)
450000
400000
350000
300000
250000
200000
150000
100000
50000
0
tai20
tai21
tai22
tai23
tai24
tai25
Nazwa instancji


Czasy wykonania się algorytmu dla instancji tai20-25 na maszynie 2
Czas (nanosekundy)
500000
450000
400000
350000
300000
250000
200000
150000
100000
50000
0
tai20
tai21
tai22
tai23
tai24
tai25
Nazwa instancji


Zależność czasu wykonania się algorytmu od liczby wczytanych zadań dla instancji tai25 na maszynie 1
Czas (nanosekundy)
450000
400000
350000
300000
250000
200000
150000
100000
50000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Liczba wczytanych pierwszych zadań


Zależność czasu wykonania się algorytmu od liczby wczytanych zadań dla instancji tai25 na maszynie 2
Czas (nanosekundy)
450000
400000
350000
300000
250000
200000
150000
100000
50000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Liczba wczytanych pierwszych zadań

2.1.1 Wnioski

Obie maszyny wykazały, że czas wykonywania się algorytmu rośnie w zależności od liczby wczytanych instancji i jest to zależność bliska liniowej. Nie ma to jednak definitywnego wpływu i zależy równie mocno od poszczególnych danych np. 12 zadań w badanej instancji wykonywało się zawsze średnio szybciej niż 11 i kolejne zadanie najwyraźniej nieznacznie usprawniało pracę algorytmu. Dla większych różnic ilości wczytanych zadań widać jednak wyraźne różnice czasów wykonania.

2.2 Testy czasu

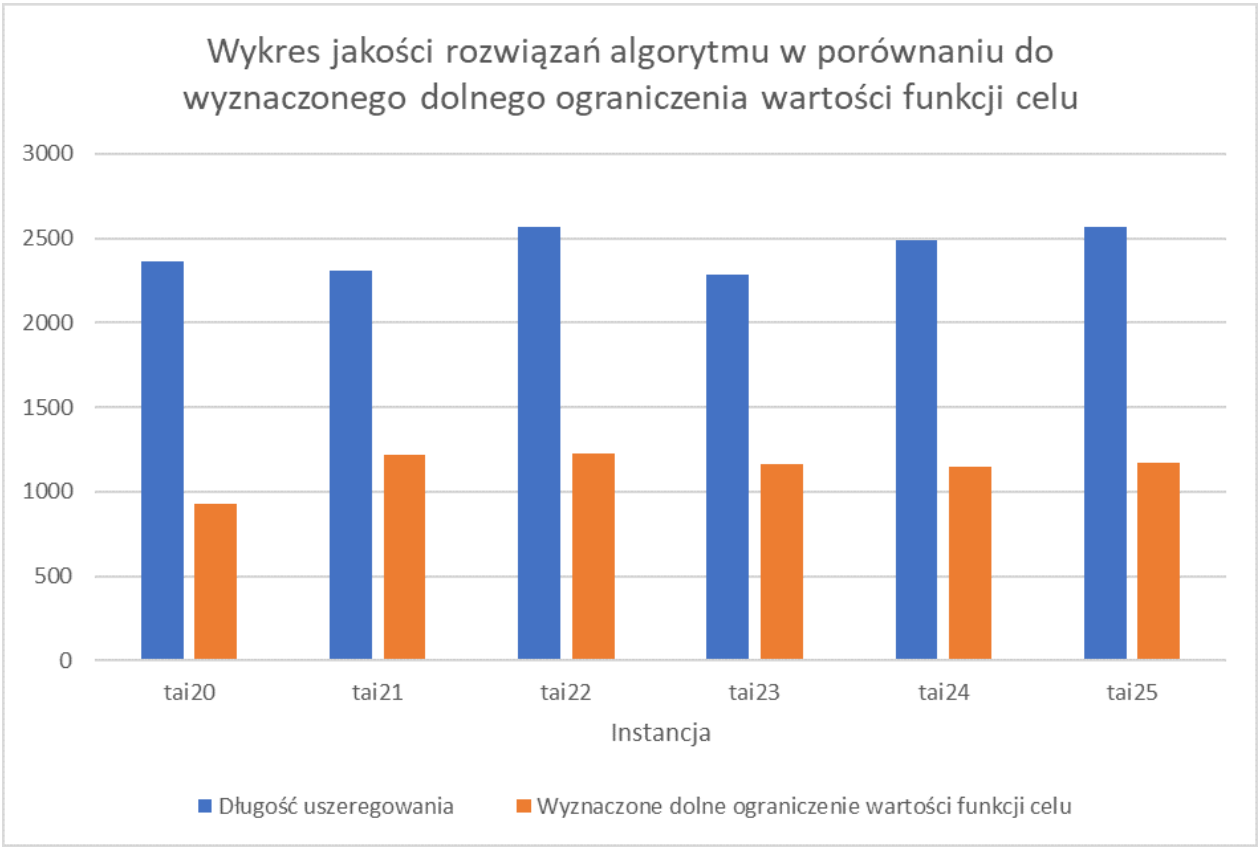
Testy czasu wykorzystywały klasę high_resolution_clock z biblioteki std::chrono i mierzyły czas z dokładnością 1 nanosekundy.

```
351  int main()
352  {
353      typedef chrono::high_resolution_clock clock;
354      clock::time_point start, end;
355      ofstream output;
356      output.open("testj.txt");
357      output << " file rozwiązanie lower bump" << endl ;
358      string tab[6] = {"tai20.txt", "tai21.txt", "tai22.txt", "tai23.txt", "tai24.txt", "tai25.txt"};
359      for (auto f : tab)
360      {
361          output << f << " ";
362          long long result = 0;
363
364          Factory factory(f, 't');
365          factory.greedy_solve();
366          output << factory.get_exec_len() << " " << factory.lower_bound();
367          output << endl ;
368
369      }
370      output << endl ;
371
372      return 0;
373  }
```

Rysunek 4: Kod użyty do wyznaczenia pomiarów jakościowych

Tablica 3: Testy jakości rozwiązań dla instancji tai20-25

Instancja	Wynik	Wyznaczony lower bump
tai20	2360	928
tai21	2308	1217
tai22	2567	1223
tai23	2282	1164
tai24	2487	1151
tai25	2566	1170


Wykres jakości rozwiązań algorytmu w porównaniu do wyznaczonego dolnego ograniczenia wartości funkcji celu
3000
2500
2000
1500
1000
500
0
tai20
tai21
tai22
tai23
tai24
tai25
Instancja
Długość uszeregowania
Wyznaczone dolne ograniczenie wartości funkcji celu