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

```
230     void opti_solve(int time)
231     {
232         chrono::seconds duration(time);
233         chrono::time_point<chrono::system_clock> end = chrono::system_clock::now() + duration;
234         auto start = chrono::steady_clock::now();
235 
236         vector<int> jobs_order = {};
237         int i=0;
238         while (i < this->get_number_of_jobs()) {jobs_order.push_back(i); ++i;}
239 
240         auto randomizer = default_random_engine {};
241         shuffle(jobs_order.begin(),jobs_order.end(), randomizer);
242         int min = this->greedy_solve(jobs_order);
243         vector<int> best_jobs_order = jobs_order;
244 
245         int res;
246         while(chrono::system_clock::now() < end)
247         {
248             this->reset();
249             shuffle(jobs_order.begin(),jobs_order.end(), randomizer);
250             res = this->greedy_solve(jobs_order);
251             int local_min = res;
252             int local_start_res = res;
253             if (res < min) {min = res; best_jobs_order = jobs_order;}
254             vector<int> local_order = jobs_order;
255             bool better = true;
256             while (better == true && chrono::system_clock::now() < end)
257             {
258                 next_permutation(jobs_order.begin(),jobs_order.end());
259                 this->reset();
260                 res = this->greedy_solve(jobs_order);
261                 if (res < local_min)
262                 {
263                     local_min = res;
264                     if (res < min) {min = res; best_jobs_order = jobs_order; }
265                 }
266                 else {better = false;}
267             }
268             jobs_order = local_order;
269             local_min = local_start_res;
270             better = true;
271             while (better == true && chrono::system_clock::now() < end)
272             {
273                 prev_permutation(jobs_order.begin(),jobs_order.end());
274                 this->reset();
275                 res = this->greedy_solve(jobs_order);
276                 if (res < local_min)
277                 {
278                     local_min = res;
279                     if (res < min) {min = res; best_jobs_order = jobs_order;}
280                 }
281                 else {better = false;}
282             }
283         }
284         this->reset();
285         this->greedy_solve(best_jobs_order);
286 
287     }
```

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

2.2 Testy jakości w zależności od czasu

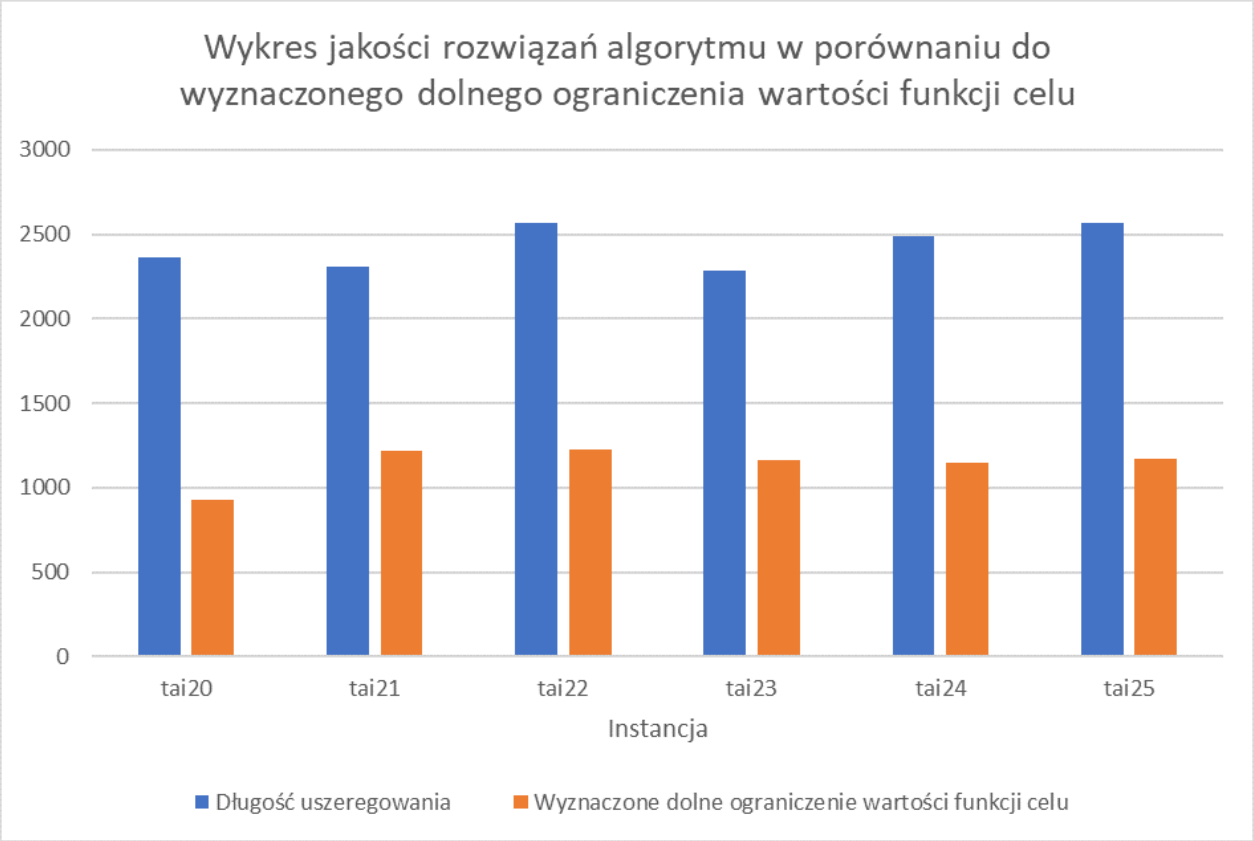
Testy czasu wykorzystywały klasę high_resolution_clock z biblioteki std::chrono i mierzyły czas z dokładnością 1 nanosekundy.

```
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 3: Kod użyty do wyznaczenia pomiarów jakościowych

Tablica 1: 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