Projektowanie efektywnych algorytmów Projekt
Etap 2

Marcin Kapuściński 248910

Grudzień 2021

1 Tabu search - przeszukiwanie lokalne z zakazami

Przeszukiwanie tabu to metoda będąca szczególnym rodzajem przeszukiwania lokalnego. Przeszukiwanie lokalne polega na szukaniu globalnie optymalnych rozwiązań w sąsiedztwie aktualnie rozpatrywanego rozwiązania. Sąsiedztwo rozwiązania jest definiowane przez relację sąsiedztwa na parach rozwiązań. Definicja tej relacji zależy od specyficznego dla problemu formatu rozwiązań, oraz dopuszczalnych operacji przejścia między rozwiązaniami. Dla listy danych, sąsiadami mogą być listy otrzymane po zamianie wartościami dwóch pozycji. Z każdą iteracją algorytmu, przechodzi się do sąsiada bieżącego rozwiązania, najlepszego w sąsiedztwie. Tak więc, osiągając optimum lokalne, algorytm będzie w okół niego krążył, nie mogąc kontynuować przeszukiwania innych sąsiedztw. Ten problem można rozwiązać resetując algorytm do nowego startowego rozwiązania, zadając tym samym nowy rejon przestrzeni rozwiązań - nowe sąsiedztwo do przeszukiwań. Można również zabronić powrotu do optimum lokalnego, zmuszając algorytm do oddalenia się od niego, wybierając gorszych sąsiadów. Takie podejście reprezentuje przeszukiwanie tabu.

Tabu search zachowuje w pamięci długotrwałej zbiór sąsiadów, lub danych wejściowych operacji przejścia do nich, które są zabronione. Pamięć długotrwała jest zachowywana między iteracjami algorytmu. Iteracją jest znalezienie nowego sąsiada, przejście do niego i ewentualne dodatkowe operacje. Owe operacje to aktualizacja znalezionego najlepszego rozwiązania globalnego, intensyfikacja i dywersyfikacja, aspiracja oraz przejście do nowego sąsiedztwa.

Intensyfikacja przeszukiwania oznacza zawężenie rozpatrywanego sąsiedztwa. Taka operacja może okazać się opłacalna jeżeli zaobserwowaliśmy, korzystając z danych w pamięci długotrwałej, że obecne sąsiedztwo, pewne cechy rozwiązań, skutkują dobrymi wynikami. Dywersyfikacja to operacja odwrotna, używana jeżeli obecne sąsiedztwo najprawdopodobniej nie będzie zawierać dobrych wyników.

Aspiracja to złamanie tabu, pozwolenie na wykonanie zabronionego ruchu, przejście do wykluczonego sąsiada. Taki wyjątek może okazać się opłacalny w przypadku, gdy zabroniony sąsiad jest lepszy od obecnego znalezionego rozwiązania globalnego lub gdy nie ma żadnych legalnych ruchów dla bieżącego sąsiada i sąsiedztwa. Taka operacja zazwyczaj dozwalana jest po spełnieniu zdefiniowanego kryterium aspiracji.

2 Implementacja

Implementacja algorytmu miała miejsce w języku `C++`. Dla zadanego problemu komiwojażera, element przestrzeni rozwiązań to lista reprezentująca cykl Hamiltona dla zadanej instancji problemu. Lista zawiera podwojony pierwszy wierzchołek, na końcu listy. Skutkuje to długością ścieżki $n + 1$, gdzie n to wielkość instancji problemu. Relację sąsiedztwa definiuje operacja `swap()`, zamieniająca dwa elementy rozwiązania miejscami, poza pierwszym i ostatnim. Warunek stopu algorytmu to 60 sekund czasu wykonania. Ze względu na charakter sąsiedztwa, jego przeszukanie powinna cechować złożoność $O(\lfloor \frac{n^2}{2} \rfloor - \lfloor \frac{n}{2} \rfloor - n + 1) \approx O(n^2)$.

```
32  inline void swap_(int g, int h, vector<int>& path){
33      int buff = path[g];
34      path[g] = path[h];
35      path[h] = buff;
36  }
37
38  inline int getSwappedCost(int** adjMatrix, int g, int h, int n, vector<int>& path, int pathCost){
39      pathCost -= adjMatrix[ path[g-1] ][ path[g] ];
40      pathCost += adjMatrix[ path[g-1] ][ path[h] ];
41
42      pathCost -= adjMatrix[ path[h] ][ path[h+1] ];
43      pathCost += adjMatrix[ path[g] ][ path[h+1] ];
44      //czy zamieniane miasta sasiaduja?
45      if(g + 1 != h){//nie sasiaduja
46          pathCost -= adjMatrix[ path[g] ][ path[g+1] ];
47          pathCost += adjMatrix[ path[h] ][ path[g+1] ];
48
49          pathCost -= adjMatrix[ path[h-1] ][ path[h] ];
50          pathCost += adjMatrix[ path[h-1] ][ path[g] ];
51      } else {//sasiaduja
52          pathCost -= adjMatrix[ path[g] ][ path[h] ];
53          pathCost += adjMatrix[ path[h] ][ path[g] ];
54      }
55      return pathCost;
56  }
```

Rysunek 1: Metody swap() i getSwappedCost()

Powyższy listing przedstawia metody odpowiedzialne za przejścia między rozwiązaniami, poruszanie się w sąsiedztwie. Funkcja `swap()` zamienia miejscami dwa elementy zadanej ścieżki, o zadanych indeksach. Funkcja `getSwappedCost()` oblicza nowy całkowity koszt rozwiązania jeżeli zamieni się elementy ścieżki o zadanych indeksach. Rozpatruje się dwa przypadki: indeksy sąsiadują ze sobą lub znajduje się między nimi przynajmniej jeden element. Jest to wymagane aby uniknąć dodawania wag krawędzi z elementów do nich samych ($g < h$, linijki 47. i 50.).

```
58  vector<int> TabuSearch::genPathBFS(int** adjMatrix, int n, int start){
59      vector<int> path = vector<int>();
60      path.reserve(n + 1);
61      vector<bool> visited = vector<bool>();
62      visited.resize(n, false);
63      int currCity = start;
64      visited[currCity] = true;
65      path.push_back(currCity);
66      int closestCurrentCityIndex = 0;
67      int closestCurrentCityDist = INT_MAX;
68
69      for(int pathLength = 1; pathLength < n; ++pathLength){
70          closestCurrentCityDist = INT_MAX;
71          for(int dest = 0; dest < n; ++dest){
72              if(adjMatrix[currCity][dest] < closestCurrentCityDist && visited[dest] == false){
73                  closestCurrentCityDist = adjMatrix[currCity][dest];
74                  closestCurrentCityIndex = dest;
75              }
76          }
77          currCity = closestCurrentCityIndex;
78          visited[currCity] = true;
79          path.push_back(currCity);
80      }
81      path.push_back(start);
82      return path;
83  }
```

Rysunek 2: Metoda genPathBFS()

Powyżej zaprezentowano implementację metody budującej rozwiązanie metodą *best first search*, z zadanym wierzchołkiem startowym. Owa funkcja jest wykorzystywana przy inicjalizacji algorytmu, wygenerowane rozwiązanie jest pierwszym rozpatrywanym.

```
85  vector<int> TabuSearch::genPathRand(int n){
86      vector<int> path;
87      path.reserve(n + 1);
88      for(int it = 0; it < n; ++it) {
89          path.push_back(it);
90      }
91
92      std::mt19937 generator(std::chrono::high_resolution_clock::now().time_since_epoch().count());
93
94      std::shuffle(path.begin(), path.end(), generator);
95      path.push_back(path[0]);
96
97      return path;
98  }
```

Rysunek 3: Metoda genPathRand()

Funkcja `genPathRand()` generuje rozwiązanie dla zadanej wielkości instancji problemu n. Kolejność wierzchołków rozwiązania jest losowana, z wyjątkiem pierwszego i ostatniego - te mają taką samą wartość. Generator liczb losowych jest inicjalizowany *czasem unixowym* wysokiej rozdzielczości, pozwalającej generować ścieżki z nowym ziarnem wystarczająco często. Niestety klasa `random_device` nie działa poprawnie dla kompilatora `g++` 8.1.0, jako części projektu `MinGW`. Powyższa metoda jest wykorzystywana przy zmianie rozpatrywanego sąsiedztwa, generując nowe rozpatrywane rozwiązanie.

```
20  TabuSearch::Timer::Timer(int s){
21      this->period = s;
22      this->startEpoch = std::time(nullptr);
23  }
24
25  inline bool TabuSearch::Timer::elapsed() {
26      if(std::time(nullptr) - this->startEpoch > this->period){
27          return true;
28      }
29      return false;
30  }
```

Rysunek 4: Metody klasy Timer

Klasa `Timer` w momencie inicjalizacji zapisuje bieżący moment w czasie, z dokładnością do sekundy. Jeżeli minął zadany w konstruktorze okres czasu, funkcja `elapsed()` zwróci wartość `false`. Owa klasa jest wykorzystywana przy sprawdzaniu warunku stopu algorytmu.

```cpp
vector<int> TabuSearch::findShortestHCycle(int** adjMatrix, int n, int memoryLength,
int itWithNoImprovLimit, int lightestHCycle, int threadN, int start)
{

    TabuSearch::Timer timer(60);
    std::string firstImprov = "g";
    Printer printer;

    //ile jeszcze iteracji, kombinacja operatorow jest tabu?
    vector<vector<int>> tabuOperands = vector<vector<int>>();
    tabuOperands.resize(n, vector<int>());
    for(int it = 0; it < n; ++it){
        tabuOperands[it].resize(n, 0);
    }
    vector<int> bestCurrPath;
    int bestCurrCost = INT_MAX;
    vector<int> currPath = genPathBFS(adjMatrix, n, start);
    int currCost = getPathCost(adjMatrix, currPath);
    int buffCost;

    int bestFoundCost;
    int bestFoundG;
    int bestFoundH;

    bool foundAnyNeighbour;
    int itWithNoImprov = 0;
    int iteration = 0;

    while(!timer.elapsed()) {
        //dywersyfikacja
        if(itWithNoImprov >= itWithNoImprovLimit){
            itWithNoImprov = 0;
            currPath = genPathRand(n);
            currCost = getPathCost(adjMatrix, currPath);
        }

        //szukanie najlepszego sasiada
        bestFoundCost = INT_MAX;
        foundAnyNeighbour = false;
        for(int g = 1; g < n - 1; ++g){
            for(int h = g + 1; h < n; ++h){
                if(tabuOperands[g][h] == 0
                   && (buffCost = getSwappedCost(adjMatrix, g, h, n, currPath, currCost)) < bestFoundCost){
                    bestFoundCost = buffCost;
                    bestFoundG = g;
                    bestFoundH = h;
                    foundAnyNeighbour = true;
                }
            }
        }

        if(foundAnyNeighbour){
            swap_(bestFoundG, bestFoundH, currPath);
            currCost = bestFoundCost;
            tabuOperands[bestFoundG][bestFoundH] = memoryLength;
            //tabuOperands[h][g] = memoryLength;//niepotrzebne
        } else {//wszystkie przejscia sa tabu
            for(int it = 0; it < n; ++it){
                for(int itt = 0; itt < n; ++itt){
                    if(tabuOperands[it][itt] > 0){
                        --tabuOperands[it][itt];
                    }
                }
            }
        }

        //czy znaleziony najlepszy lokalny sasiad jest lepszy od najlepszego znalezionego rozwiazania
        if(currCost < bestCurrCost) {
            bestCurrPath = currPath;
            bestCurrCost = currCost;
            float prd = 100 * (bestCurrCost - lightestHCycle)/(float)lightestHCycle;
            std::string feed = std::to_string(iteration) +  "@" + std::to_string(threadN) + "\t\t" +
             std::to_string(bestCurrCost) + "\tPRD = " + std::to_string(prd) + "%\n";
            printer.printSync(feed);
            if(firstImprov.at(0) == 'g') {
                firstImprov = feed;
            }
        } else {
            ++itWithNoImprov;
        }

        //zapominanie
        for(int it = 0; it < n; ++it){
            for(int itt = 0; itt < n; ++itt){
                if(tabuOperands[it][itt] > 0){
                    --tabuOperands[it][itt];
                }
            }
        }
        ++iteration;
    }
    printer.printSync(firstImprov);
    return bestCurrPath;
}
```

Rysunek 5: Metoda findShortestHCycle()

Główna funkcja implementująca algorytm. Funkcja przyjmuje:

1. macierz wag dla instancji problemu sczytaną z pliku wejściowego
2. wielkość instancji problemu n - ilość wierzchołków
3. ilość iteracji przechowywania przejścia (pary indeksów dla `swap`) w tablicy tabu
4. limit ilości iteracji bez poprawy najlepszego globalnego rozwiązania, po którego przekroczeniu rozpatrywane jest nowe sąsiedztwo,
5. optimum globalne dla instancji problemu, podawane w pliku z danymi wejściowymi
6. przypisany numer wątku
7. wierzchołek startowy dla algorytmu

Na początku wykonywania inicjalizowany jest licznik sprawdzany w ramach warunku stopu algorytmu. Po przekroczeniu minuty, algorytm kończy wykonanie po zakończeniu iteracji. Następnie definiowana i inicjalizowana jest tablica tabu w postaci tablicy dwuwymiarowej. Komórki tej tablicy przechowują wartość mówiącą, ile jeszcze iteracji dana kombinacja indeksów dla operacji `swap` będzie zabroniona. Owa tablica może mieć niezerowe wartości tylko powyżej przekątnej. Pierwszy z indeksów `g` dla zamiany zawsze jest mniejszy od drugiego `h`. Przed główną pętlą algorytmu deklarowane są używane zmienne. Warunkiem jej zakończenia jest minięcie okresu czasu zadanego obiektowi klasy `Timer` na początku metody.

Linia 129., pierwszą czynnością w nowej iteracji pętli jest sprawdzenie czy licznik iteracji algorytmu bez poprawy najlepszego globalnie rozwiązania przekroczył zadany limit. Jeżeli tak, licznik jest resetowany i generowana jest losowe nowe rozwiązanie, którego sąsiedztwo będzie przeszukiwane. Jest to dywersyfikacja nie poprzez rozszerzenie badanego sąsiedztwa, a przez jego zmianę.

Linia 136., następuje szukanie najlepszego sąsiada bieżącego rozwiązania, do którego można przejść. Obliczany jest koszt ścieżki po zamianie elementów o zadanych indeksach, będących iteratorami zagnieżdżonej pętli. Jeżeli prognozowany koszt jest mniejszy niż najmniejszy znaleziony w tej iteracji algorytmu - kombinacja indeksów prowadzących do sprawdzanego sąsiada jest zapisywana jako aktualnie najlepsza. Dozwolone są zamiany elementów o indeksach $\in \langle 1, n-1 \rangle$, nie można zamieniać pierwszego i ostatniego wierzchołka rozwiązania z żadnym innym. Są one zmieniane jedynie przy przechodzeniu do nowego sąsiedztwa wygenerowanego losowo rozwiązania.

Linia 151., jeżeli znaleziono sąsiada do którego można przejść, następuje przejście oraz aktualizacja tablicy tabu, ustawiając licznik pary zamienionych indeksów na wartość zadaną funkcji. Jeżeli nie ma możliwego ruchu, nie można przejść do żadnego sąsiada ze względu na tabu - następuje dekrementacja wszystkich liczników w tablicy tabu. Jest to wolna aspiracja. Dla zadanej trwałości pamięci tp równej n, zawsze będzie przynajmniej jedno dozwolone przejście dla wielkości instancji problemu $n > 4$. Wynika to ze wzoru biorącego pod uwagę legalne wartości indeksów `g` i `h` i rozmiar tablicy tabu: $\lfloor \frac{n^2}{2} \rfloor - \lfloor \frac{n}{2} \rfloor - n + 1 > tp$. Zadając trwałość pamięci równą lub większą od n, ta funkcjonalność jest bezużyteczna.

Linia 167., sprawdzane jest, czy znaleziony najlepszy sąsiad jest lepszy od najlepszego znalezionego globalnego rozwiązania. Jeżeli tak jest, jest to odnotowywane w pamięci długotrwałej oraz wypisywane do standardowego wyjścia. Jeżeli nie polepszono globalnego rozwiązania, inkrementowany jest licznik iteracji bez poprawy globalnego wyniku.

Linia 182., dekrementowane są wartości w komórkach tablicy tabu, jeżeli są nieujemne. Jest to równoważne z zapominaniem lub przedawnianiem zakazów. Ostatecznie inkrementowany jest ogólny licznik iteracji algorytmu, przydatny przy raportowaniu wyników.

```
9   int main(int argc, char *argv[])
10  {
11      if(argc != 5){
12          cout << "pass: <filePath> <nOfThreads> <memoryLength> <noImprovItLimit>" << endl;
13          return 1;
14      }
15      Input input = Input::readInput(argv[1]);
16
17      int nOfThreads = std::stoi(argv[2]);
18      std::vector<std::future<std::vector<int>>> threads;
19      for(int it = 0; it < nOfThreads; ++it){
20          threads.push_back(std::async(TabuSearch::findShortestHCycle,
21              input.adjacencyMatrix,
22              input.nOfCities,
23              std::stoi(argv[3]),//memory length
24              std::stoi(argv[4]),//limit bez poprawy globalnej
25              input.shortestHCycleLength,
26              it,
27              it * (input.nOfCities/nOfThreads)
28          ));
29      }
30
31      int lowestFoundCost = INT_MAX;
32      vector<int> shortestFoundPath;
33      vector<int> buff;
34      for(int it = 0; it < threads.size(); ++it){
35          if(TabuSearch::getPathCost(input.adjacencyMatrix ,buff = threads[it].get()) < lowestFoundCost){
36              lowestFoundCost = TabuSearch::getPathCost(input.adjacencyMatrix , buff);
37              shortestFoundPath = buff;
38          }
39      }
40
41      for(int it = 0; it < shortestFoundPath.size(); ++it){
42          cout << shortestFoundPath[it] << " ";
43      }
44      cout << "\ncost: " << TabuSearch::getPathCost(input.adjacencyMatrix, shortestFoundPath);
45      string b;
46      cin >> b;
47      return 0;
48  }
```

Rysunek 6: Metoda main() dla implementacji zrównoleglonej

Zrównoleglenie polega jedynie na uruchomieniu instancji algorytmu dla wielu rozwiązań startowych o pierwszym wierzchołku dyktowanym numerem wątku, ze wzoru w linii 27. Następnie, wyniki wykonania algorytmu są zbierane z uruchomionych wcześniej wątków, wybierane jest najlepsze znalezione rozwiązanie.

3 Pomiary

Poprawność działania algorytmu jest wnioskowana na podstawie poprawnie obliczanych kosztów ścieżek oraz raportowania jedynie coraz to lepszych znalezionych rozwiązań.

Pomiary zostały wykonane po skompilowaniu z opcją optymalizacji 3 poziomu, na procesorze o 8 rdzeniach logicznych oraz maksymalnym taktowaniu 3.4 GHz. Przy każdym uruchomieniu programu eksploatowane były 4 wątki. Na rzecz pomiarów przyjęto warunek stopu w postaci limitu czasu wykonania do 60 sekund. Pomiar dla każdej konfiguracji został wykonany 4 razy. Kolumny tablic reprezentują limity iteracji bez poprawy najlepszego wyniku globalnego *revLim* (n, n^2, n^3, n^4), wiersze odpowiadają różnym trwałościom pamięci (wartości ustawianego licznika tabu).

3.1 ftv64.atsp

	n	n^2	n^3	n^4	n^5
n/8	22.5	20.6	22.5	22.5	22.5
n/4	22.0	9.26	12.4	12.5	12.5
n/2	19.0	3.44	2.19	2.18	2.18
n	19.0	4.50	2.41	2.34	2.34
2n	19.0	7.98	6.60	6.58	6.58
10n	19.0	19.0	6.44	19.0	19.0
20n	19.0	19.0	19.0	19.0	19.0

Rysunek 7: Mapa ciepła dla uśrednionych odchyleń od optimum, w procentach, dla instancji ftv64.atsp

	n^3
1.	2.45
2.	2.83
3.	2.56
4.	0.92

Rysunek 8: poszczególne wyniki dla trwałości pamięci $n/2$ i *revLim* równe n^3

3.2 gr120.tsp

	n	n^2	n^3
n/4	18.9	9.25	9.05
n/2	19.7	7.55	7.58
n	19.8	11.4	9.49
2n	19.8	17.0	16.2
10n	19.8	19.8	19.8
20n	19.8	19.8	19.8

Rysunek 9: Mapa ciepła dla uśrednionych odchyleń od optimum, w procentach, dla instancji gr120.tsp

3.3 ftv170.atsp

	n	n^2	n^3
n/4	32.3	22.9	22.9
n/2	32.3	19.7	19.4
n	32.3	23.5	18.0
2n	23.5	32.3	28.4
10n	32.3	32.3	32.3
20n	32.3	32.3	32.3

Rysunek 10: Mapa ciepła dla uśrednionych odchyleń od optimum, w procentach, dla instancji ftv170.atsp

4 Wnioski

Jak można wywnioskować z tabeli 7, najlepsze wyniki dla danej instancji osiągnięte zostały dla trwałości pamięci bliskiej n i *revLim* większego od n^2. Warto zauważyć, że dla wielkości zadanej instancji, n^4 to aż 16777216. Ogólna liczba iteracji przy wykonaniu instancji algorytmu rzadko przekracza 3 miliony. Oznacza to, że dla *revLim* większych od 64^3, dywersyfikacja w postaci losowania nowego rozpatrywanego rozwiązania w ogóle nie ma miejsca, wynik znajdowany jest korzystając jedynie z listy tabu i ewentualnej aspiracji. Skutkuje to deterministycznym algorytmem dla dwóch ostatnich kolumn, co widać po pojedynczych pomiarach - ich uśrednianie jest zbędne, każde wykonanie algorytmu daje taki sam wynik. Taka sama sytuacja ma miejsce dla każdej żółtej i czerwonej komórki (wiersze $20n$, $10n$ bez n^3, kolumna n, wiersz $n/8$ bez n^2)(wyniki 19, 22, 22.5). Identyczne wyniki dla różnych parametrów czasem wynikają ze znajdowania dobrych wyników w początkowych iteracjach, szukając wokół rozwiązania generowanego metodą *best first seatch* (np. dla wyniku 19). Ma to miejsce zawsze dla *revLim* równego n, algorytm za szybko generuje nowe sąsiedztwo, nie znajdując potem dobrych rozwiązań. Czasem jednak, brak dywersyfikacji nie przeszkadza osiągnięciu zadowalających wyników, na przykład dla trwałości pamięci równej n i *revLim* o wartości n^4.
Dla kolumny n ewidentnie *revLim* jest za niski, wyniki są uniwersalnie złej jakości, losowanie ma miejsce przed znalezieniem oddalonego lokalnego optimum. Dla niektórych z komórek, *revLim* jest wystarczająco mały, natomiast trwałość pamięci okazuje się zbyt mała ($n/8$) lub duża ($10n$).

Dywersyfikacja w postaci losowania nowego rozwiązania jako centrum sąsiedztwa czasami się opłaca. W ten sposób możliwe jest osiągnięcie rozwiązania lepszego niż algorytm deterministyczny, nigdy nie losujący nowego sąsiedztwa. Jest to widoczne porównując najlepszy wynik w 8 z każdym wynikiem dla kolumn reprezentujących brak dywersyfikacji ($revLim > n^3$). Aby mieć większe szanse na wylosowanie dobrego sąsiedztwa, warto równolegle uruchomić wiele instancji algorytmu. Biorąc pod uwagę najgorszy wynik z tabeli 8, jeżeli jesteśmy pewni jakości implementacji tabu, aspiracji i dobrania trwałości pamięci, możemy zrezygnować z niedeterministycznej dywersyfikacji aby wyeliminować możliwość znacznie gorszych wyników dla pojedynczych wykonań. Dla instancji `ftv170.atsp` brak dywersyfikacji miał miejsce już przy *revLim* wielkości n^3. Wygląda na to, że wraz ze wzrostem wielkości instancji problemu należy zmniejszać limit iteracji bez poprawy globalnie najlepszego wyniku.

Równoległe uruchamianie wielu instancji algorytmu opłaci się zawsze jeżeli ma miejsce losowa dywersyfikacja, ze względu na większą liczbę losowań. Jeżeli *revLim* efektywnie neguje dywersyfikację, wielowątkowość nic nie wnosi, algorytm jest deterministyczny. Rozpatrując tabelę 8 (dane wejściowe umożliwiające dywersyfikację), widać bardzo duży potencjał na poprawę wyniku. Dla implementacji jednowątkowej, moglibyśmy otrzymać najgorszy osiągnięty wynik - 2.83% odchylenia od optimum. Dla większej ilości instancji algorytmu, mamy szansę znaleźć lepsze rozwiązanie, na przykład prowadzące do wyniku o odchyleniu 0.92% od optimum.

Reasumując, zastosowana implementacja dywersyfikacji może okazać się tym bardziej opłacalna, im więcej instancji algorytmu zostanie uruchomionych. Implementacja tablicy tabu i aspiracji jest wystarczająco dobra aby uzyskać satysfakcjonujące wyniki bez zastosowania dywersyfikacji. Biorąc pod uwagę również pomiary dla instancji `gr120.tsp` i `ftv170.atsp`, najlepsze wyniki uzyskiwane są dla trwałości pamięci bliskiej n lub $n/2$. Przy zastosowaniu wielowątkowości, warto zapewnić limit iteracji bez poprawy najlepszego globalnie wyniku większy od n i mniejszy od n^4 dla mniejszych instancji, n^3 dla większych.