SPRAWOZDANIE

Projektowanie efektywnych algorytmów

Zadanie projektowe nr 2

Implementacja i analiza efektywności algorytmu Tabu Search i symulowanego wyżarzania dla problemu komiwojażera (TSP)

Autor:
Patryk Dulęba 259213
Czwartek 11:15

Prowadzący:
Mgr inż. Antoni Sterna

Spis treści

1 Opis rozpatrywanego problemu

Celem tego zadania projektowego było zaimplementowanie oraz późniejsza analiza efektywności algorytmów znajdujących rozwiązania dla problemu komiwojażera. Problem komiwojażera polega na poszukiwaniu w grafie takiego cyklu, który zawiera wszystkie wierzchołki (każdy tylko raz), kończy się i zaczyna w tym samym punkcie i ma jak najmniejszy koszt całej drogi. Inaczej mówiąc problem komiwojażera polega na znalezieniu cyklu Hammiltona o najmniejszej wadze.

Aktualnie nie jest znany efektywny, czyli działający w czasie co najwyżej wielomianowym algorytm dający gwarancję znalezienia optymalnego rozwiązania problemu komiwojażera. Z tego powodu problem ten jest zaliczany do klasy problemów NP-trudnych. Dzieje się tak z powodu ogromnej liczby kombinacji do sprawdzenia przy grafach o większej liczbie wierzchołków. W grafie pełnym mającym n wierzchołków liczba możliwych cykli Hammiltona wynosi aż $\frac{(n-1!)}{2}$.

2 Rozpatrywane algorytmy rozwiązujące problem TSP

Poniżej zostały omówione wszystkie algorytmy rozwiązujące problem komiwojażera zaimplementowane w programie w ramach tego zadania projektowego.

2.1 Tabu Search (TS)

Tabu Search to metoda optymalizacji, która polega na wyszukiwaniu najlepszego rozwiązania danego problemu poprzez przeszukiwanie przestrzeni rozwiązań w sposób iteracyjny i stopniowe poprawianie już znalezionych rozwiązań. Kluczowym elementem tej metody jest stosowanie tzw. "tabu list", czyli listy zakazanych ruchów, które pozwalają uniknąć zapętlenia się w rozwiązaniach już przeszukanych, a tym samym zwiększyć szansę na znalezienie optymalnego rozwiązania. W praktyce Tabu Search jest często stosowany do rozwiązywania złożonych problemów optymalizacyjnych, w których trudno znaleźć rozwiązanie metodami heurystycznymi lub innymi metodami wyszukiwania ewolucyjnego.

2.1.1 Opis działania algorytmu

Działanie Tabu Search opisane w krokach:

1. Wybór początkowej solucji.
2. Wybór operatora przekształcenia.
3. Wybór sąsiedniej solucji.
4. Sprawdzenie, czy sąsiednia solucja jest lepsza od aktualnego optymalnego rozwiązania. Jeśli tak, to przechodzimy do kroku 5. Jeśli nie, to przechodzimy do kroku 6.
5. Aktualizacja aktualnego optymalnego rozwiązania i przejście do kroku 2.
6. Sprawdzenie, czy sąsiednia solucja jest w tabu. Jeśli tak, to przechodzimy do kroku 7. Jeśli nie, to wracamy do kroku 4.
7. Ustalenie czasu tabu dla sąsiedniej solucji i przejście do kroku 2.
8. Zakończenie algorytmu po osiągnięciu zadanego limitu iteracji lub zadanego limitu czasu.

2.1.2 Najważniejsze elementy Tabu Search

- **Lista tabu** - jest zbiorem elementów, które zostały już wybrane lub rozważane w danym momencie pracy algorytmu i nie mogą być ponownie uwzględnione przy wyborze kolejnego rozwiązania. Dzięki temu algorytm unika zapętlenia się w już sprawdzonych opcjach i jest w stanie znaleźć lepsze rozwiązanie danego problemu.

- **Dywersyfikacja** - polega to na unikaniu sytuacji, w której algorytm skupia się na jednej grupie rozwiązań i ignoruje inne opcje, co może ograniczyć jego skuteczność. W praktyce polega na wygenerowaniu nowego rozwiązania początkowego po kilku iteracjach, które nie przyniosły żadnych zmian.
- **Definicja sąsiedztwa** - jest to sposób przeszukiwania sąsiednich rozwiązań danego rozwiązania. Wyróżniamy 3 możliwe strategie :
  - **insert(i,j)** – przeniesienie j-tego elementu na pozycję i-tą
  - **swap(i,j)** – zamiana miejscami i-tego elementu z j-tym
  - **invert(i,j)** – odwrócenie kolejności w podciągu zaczynającym się na i-tej pozycji i kończącym się na pozycji j-tej

2.2 Symulowane Wyżarzanie (SA)

Symulowane wyżarzanie (SA) jest algorytmem optymalizacyjnym, który jest stosowany w różnych dziedzinach, takich jak optymalizacja, przetwarzanie obrazu i inżynieria wiedzy. Jego głównym celem jest znalezienie minimalnej lub maksymalnej wartości dla danego problemu, używając metody, która jest analogiczna do naturalnego procesu wyżarzania. Wyżarzanie jest procesem zmiany materiału lub struktury poprzez podgrzanie materiału do wysokiej temperatury i powolne chłodzenie go. W algorytmie SA jest to odwzorowane poprzez losowe szukanie różnych rozwiązań, a następnie wybór najlepszego z nich.

2.2.1 Opis działania algorytmu

Działanie Symulowanego wyżarzania opisane w krokach:

1. Wybór początkowej solucji.
2. Wyznaczenie temperatury początkowej.
3. Wybierz losowe sąsiednie rozwiązanie.
4. Oblicz prawdopodobieństwo (P) przyjęcia nowego rozwiązania.
5. Losuj liczbę z zakresu 0-1.
6. Jeśli liczba losowa jest mniejsza niż P, przyjmij nowe rozwiązanie. W przeciwnym wypadku pozostań z początkowym rozwiązaniem.
7. Zmniejsz obecną temperaturę, mnożąc ją przez temperaturę chłodzenia.
8. Powtarzaj kroki 2-7, aż osiągniesz określoną liczbę iteracji lub aż temperatura spadnie poniżej określonego progu.
9. Zakończ algorytm i zwróć najlepszy znaleziony punkt.

2.2.2 Najważniejsze elementy Symulowanego wyżarzania

- **Definicja sąsiedztwa** - jest to sposób losowego wyboru sąsiedniego rozwiązania. Wyróżniamy 3 możliwe strategie :
  - **insert(i,j)** – przeniesienie j-tego elementu na pozycję i-tą
  - **swap(i,j)** – zamiana miejscami i-tego elementu z j-tym
  - **invert(i,j)** – odwrócenie kolejności w podciągu zaczynającym się na i-tej pozycji i kończącym się na pozycji j-tej

3 Opis najważniejszych klas w projekcie

3.1 Klasa main

Klasa zawierająca menu główne napisanego programu.

```cpp
//Glowne menu programu
int main(int, char**) {

    AdjacencyMatrix mat = *(new AdjacencyMatrix());
    srand(time(NULL));
    bool koniec = false;
    char wybor;

    system("cls");

    while (!koniec) {

        system("cls");
        cout << " ============MENU============ " << endl;
        cout << "------------------------------" << endl;
        cout << "------------------------------" << endl;
        cout << "1.  Wczytaj dane z pliku" << endl;
        cout << "2.  Automatyczne generowanie grafu" << endl;
        cout << "x.  Zamkniecie programu " << endl;
        cout << "------------------------------" << endl;
        cout << "------------------------------" << endl;
        cout << "Wybor: ";
        cin >> wybor;
```

Rysunek 1: Klasa main

3.2 Klasa AdjacencyMatrix

Klasa odpowiedzialna za generowanie, wczytywanie oraz przechowywanie danych o grafie.

```cpp
#include <iostream>

using namespace std;

class AdjacencyMatrix{
    private:
        int **Matrix;
        int size;
        double time;
        bool dywer, neighborType;

    public:
        AdjacencyMatrix();
        ~AdjacencyMatrix();
        int** getMatrix();
        void choice();
        void choice2();
        int getSize();
        void setTime(double time);
        bool readMatrix(string name);
        void show();
        double getTime();
        void setDywer(bool temp);
        bool getDywer();
        void setNeighBorType(bool temp);
        bool getNeighborType();
        void generateMatrix(int mainSize);

};
```

Rysunek 2: Klasa AdjacencyMatrix

3.3 Klasa SA

Klasa odpowiedzialna za wykonywanie algorytmu Symulowanego wyżarzania.

```
#include <iostream>
#include <vector>

using namespace std;

class SA{

    private:
        int size, length2, length1, solution;
        double t_min, t_cool, t_curr, t;

    public:
        SA();
        int path_distance(const vector<int>& path, int** Matrix);
        void randomSwap(vector<int> &order);
        void randomInvert(vector<int> &temp);
        bool probability(int length, int length1, double temperature);
        int TSP( double max_iterations, int** mainMatrix, int mainSize, bool neighborType);
};
```

Rysunek 3: Klasa SA

3.4 Klasa TS

Klasa odpowiedzialna za wykonywanie algorytmu Tabu Search.

```
#include <iostream>
#include <vector>

using namespace std;

class TS{
    struct tabu{
        int x;
        int y;
        int time;
    };

    private:
        int size, critical_events, length, best_length, tabuTime;
        int bestChange[2];
        vector<tabu> tab;
        double t;

    public:
        TS();
        bool contains(int x, int y);
        void insert(int x, int y);
        int path_distance(const vector<int>& path, int** Matrix);
        vector<int> findBestNeighbor(vector<int> order, int** Matrix);
        vector<int> findBestNeighbor2(vector<int> order, int** Matrix);
        void updateTabuList();
        void randomSwap(vector<int> &order);
        int TSP( double max_iterations, int** mainMatrix, int mainSize, bool dywer, bool neighborType);
};
```

Rysunek 4: Klasa TS

3.5 Klasa Time

Klasa odpowiedzialna za pomiar czasu wykonywania algorytmów.

```cpp
#include <iostream>
#include <chrono>

using namespace std;
using namespace std::chrono;

class Timer {
public:
    high_resolution_clock::time_point startTime;
    high_resolution_clock::time_point endTime;

    void start();
    void stop();
    double timeCount();
};
```

Rysunek 5: Klasa Time

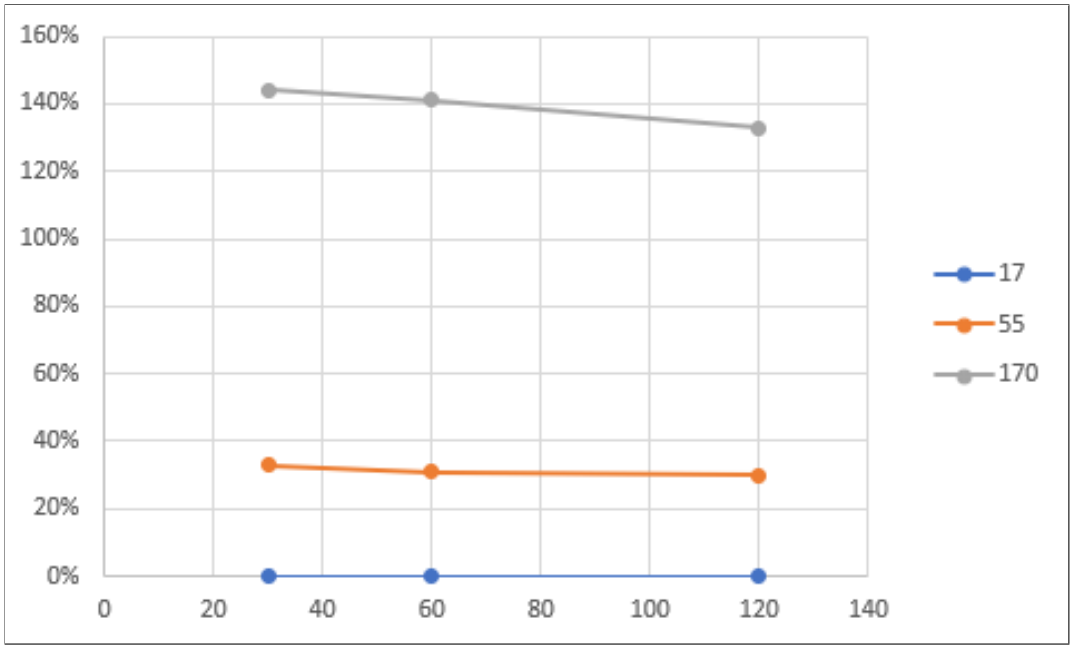
4 Wyniki eksperymentu

4.1 Tabu Search (TS)

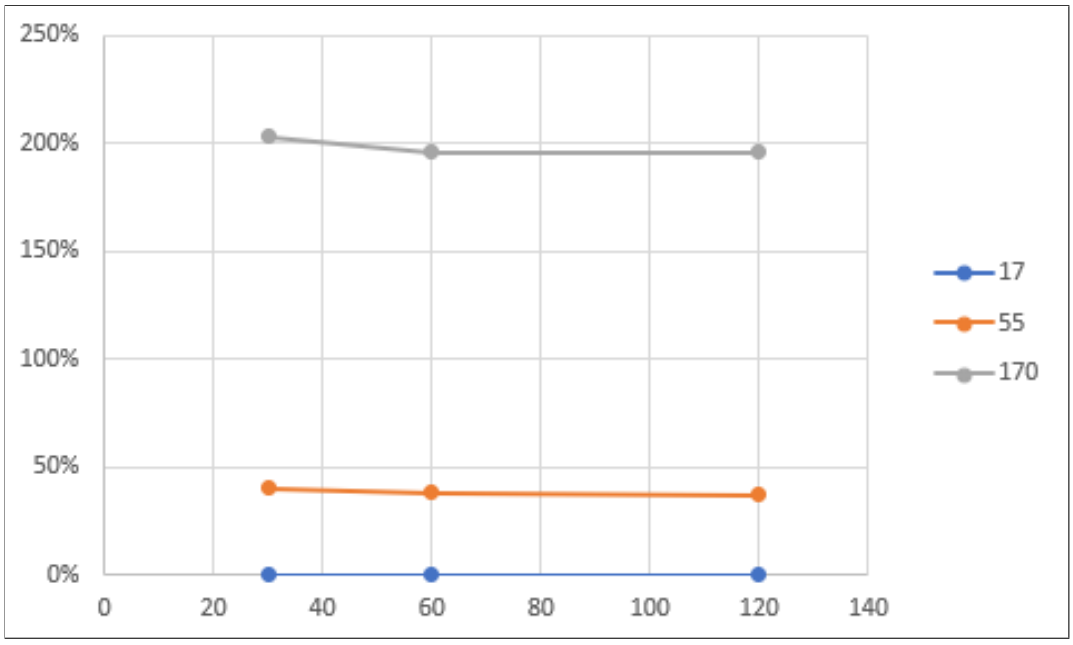
Liczba wierzchołków	Czas [s]	Z dywersyfikacją		Bez dywersyfikacji	
		SWAP	INVERT	SWAP	INVERT
17	30	0%	0%	56%	15%
	60	0%	0%	38%	0%
	120	0%	0%	8%	0%
55	30	33%	40%	95%	131%
	60	31%	38%	92%	100%
	120	30%	37%	84%	92%
170	30	144%	203%	216%	274%
	60	141%	196%	195%	242%
	120	133%	196%	182%	240%

Tabela 1: Wyniki pomiarów błędu względnego algorytmu Tabu Search w zależności od ustawionych parametrów.

4.1.1 Wykresy Tabu Search z dywersyfikacją

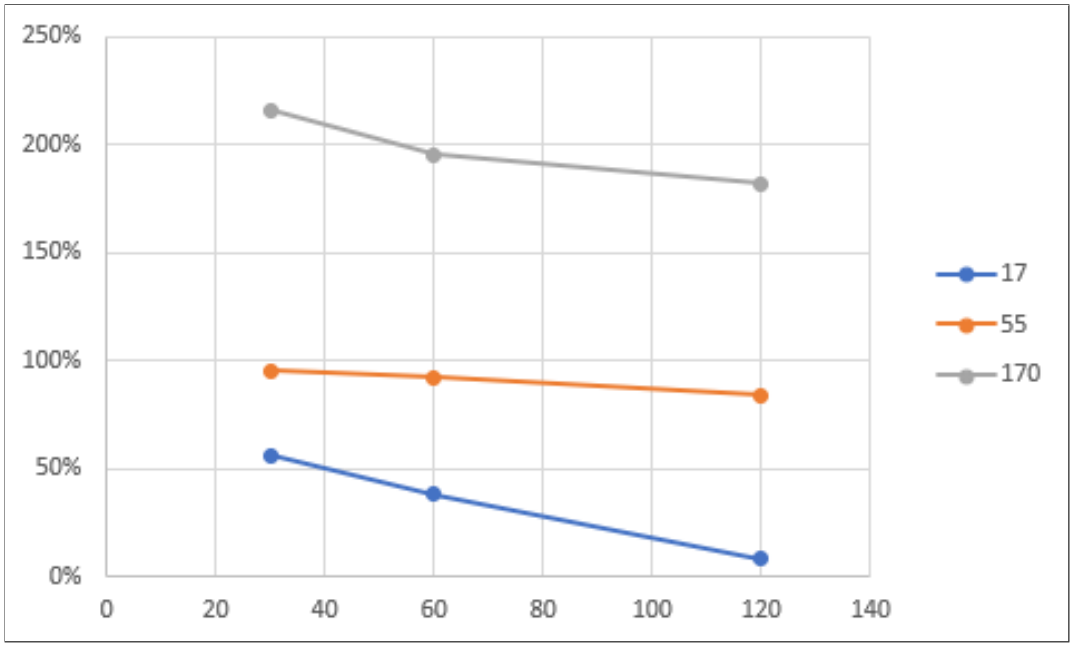


Rysunek 6: Zależność błędu względnego od czasu wykonywania algorytmu i ilości wierzchołków dla przypadku z dywersyfikacją i typem sąsiedztwa SWAP

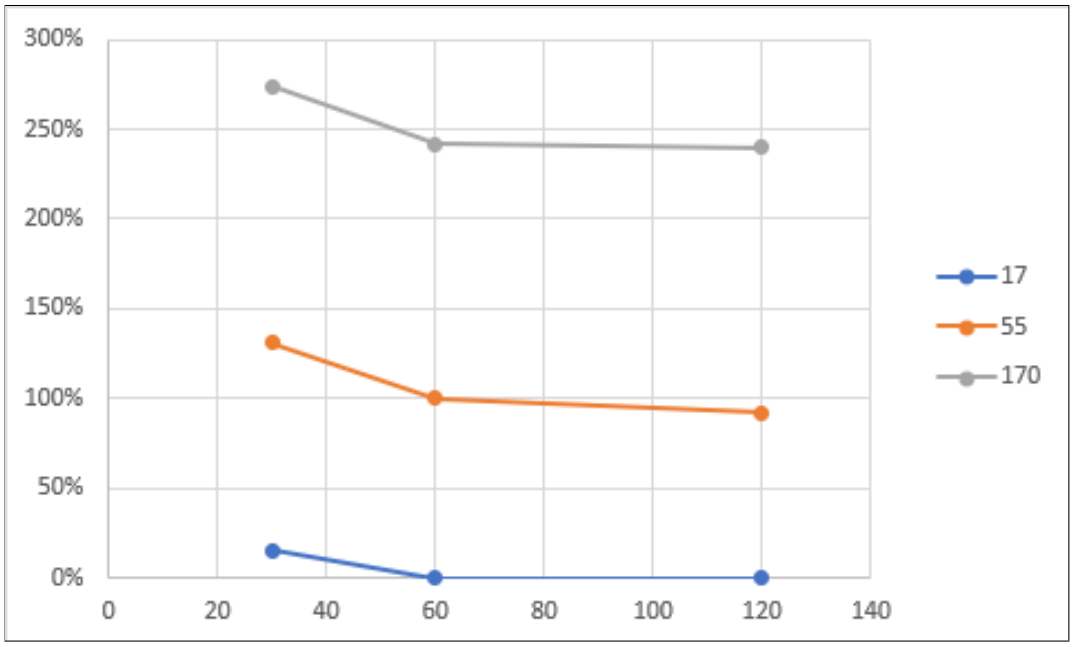


Rysunek 7: Zależność błędu względnego od czasu wykonywania algorytmu i ilości wierzchołków dla przypadku z dywersyfikacją i typem sąsiedztwa INVERT

4.1.2 Wykresy Tabu Search bez dywersyfikacji



Rysunek 8: Zależność błędu względnego od czasu wykonywania algorytmu i ilości wierzchołków dla przypadku bez dywersyfikacji i typem sąsiedztwa SWAP



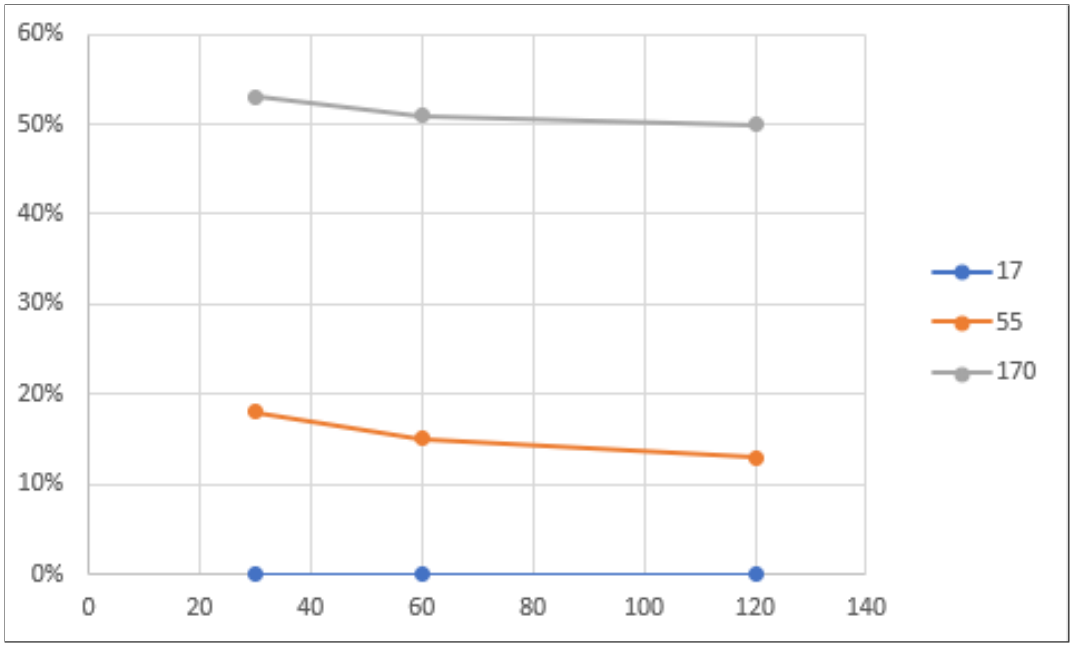
Rysunek 9: Zależność błędu względnego od czasu wykonywania algorytmu i ilości wierzchołków dla przypadku bez dywersyfikacji i typem sąsiedztwa INVERT

4.2 Symulowane wyżarzanie (SA)

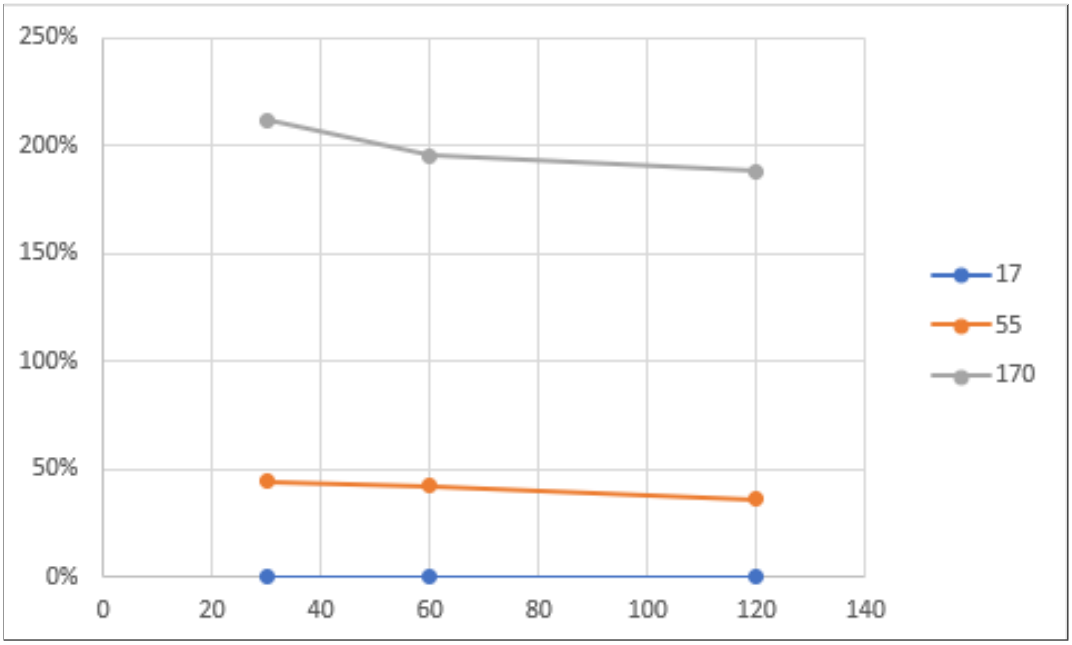
Liczba wierzchołków	Czas [s]	SWAP	INVERT
17	30	0%	0%
	60	0%	0%
	120	0%	0%
55	30	18%	44%
	60	15%	42%
	120	13%	36%
170	30	53%	212%
	60	51%	195%
	120	50%	188%

Tabela 2: Wyniki pomiarów błędu względnego algorytmu Symulowanego wyżarzania w zależności od ustawionych parametrów.

4.2.1 Wykresy Symulowanego wyżarzania



Rysunek 10: Zależność błędu względnego od czasu wykonywania algorytmu i ilości wierzchołków dla przypadku z typem sąsiedztwa SWAP



Rysunek 11: Zależność błędu względnego od czasu wykonywania algorytmu i ilości wierzchołków dla przypadku z typem sąsiedztwa INVERT

5 Wnioski

Na podstawie otrzymanych wyników można stwierdzić, że żaden z zaimplementowanych algorytmów nie daje jednoznacznego rozwiązania. Błąd względny otrzymywanych wyników zależy od czasu działania algorytmu. Im dłuższy ustawimy czas tym mamy większą szansę na otrzymanie najbardziej optymalnego rozwiązania.

W przypadku metody Tabu Search wyłączenie dywersyfikacji znacznie pogarsza efektywność algorytmu. Gorsze wyniki w tym samym czasie również uzyskujemy wykorzystując typ sąsiedztwa INVERT.

Algorytm Symulowanego wyżarzania tak jak wcześniejszy daje lepsze wyniki, używając typu sąsiedztwa SWAP.

Porównując oba algorytmy lepsze wyniki uzyskujemy stosując algorytm SA.