(3) Simulated Annealing

Projektowanie Efektywnych Algorytmow

Projekt

18/11/2022

259222 Bartosz Ostrowski

|  |  |
| --- | --- |
| spis treści | strona |
| Sformułowanie zadania | 2 |
| Metoda | 3,4 |
| Algorytm | 5,6 |
| Dane testowe | 7 |
| Procedura badawcza | 8,9 |
| Wyniki | 10,11,12,13 |
| Analiza wyników i wnioski | 14 |

1. Sformułowanie zadania

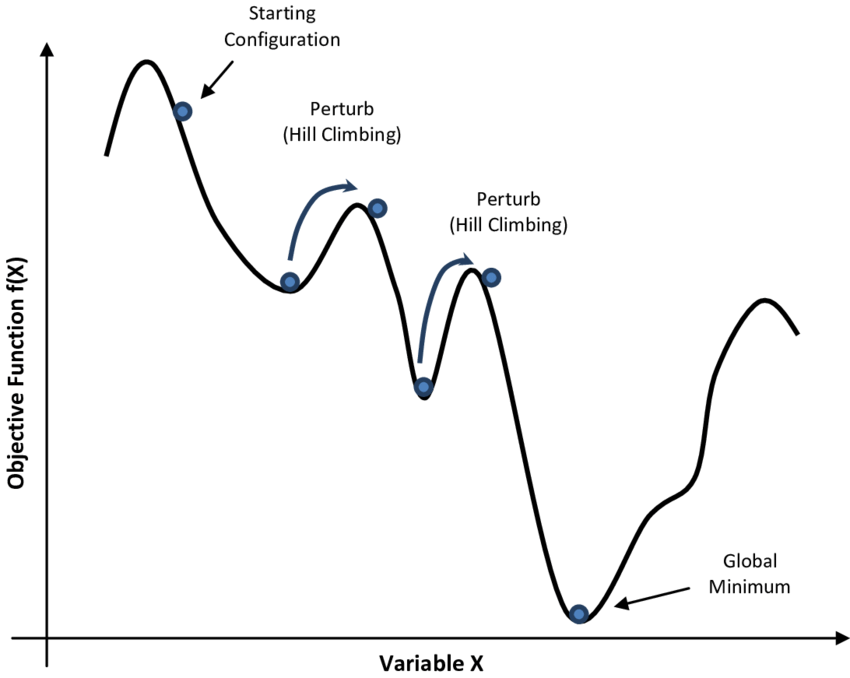
Algorytm *Symulowanego wyżarzania (Simulated Annealing)*, jest jedną z technik projektowania algorytmów heurystycznych, do określenia przybliżonego optimum dla danego problemu, koncepcja wywodzi się z metody obróbki cieplnej, polegającym na nagrzaniu materiału do określonej temperatury, wytrzymaniu przy tej temperaturze oraz następnym powolnym studzeniu.

Symulowane wyżarzanie jest metaheurystyką, zatem nie jest to szczegółowo opisany algorytm, a jedynie ogólna koncepcja. W zależności od problemu do rozwiązania, poszczególne elementy algorytmu mogą być zdefiniowane różnie. Przykładowo, przy rozwiązywaniu problemu komiwojażera pobliskim rozwiązaniem może być zamiana miejscami dwóch węzłów. Odległość między aktualnym a nowym rozwiązaniem w takim przypadku nie musi zależeć od temperatury.

Należy wziąć pod uwagę fakt, że algorytm ten nie gwarantuje znalezienia dokładnego rozwiązania problemu, ponieważ głównym elementem jest poleganie na losowym dobieraniu następnych rozwiązań (w granicy sąsiedztwa do aktualnego rozwiązania), a następnie przy pomocy dobranego kryterium rozwiązania, dobranie prawdopodobieństwa i decyzja, czy należy przyjąć niekorzystne rozwiązanie, bądź na podstawie tego prawdopodobieństwa je porzucić.

1. Metoda

Symulowane wyżarzanie jest rozszerzoną wersją przeszukiwania lokalnego (local search). Przeszukiwanie lokalne zatrzymuje się przy pierwszym minimum (bądź maksimum), badając całą przestrzeń od punktu startowego, jednak ograniczając się do najbardziej korzystnych lokalnie wyników. Koncepcję tę rozszerzamy o możliwość wyskoczenia z naszych ograniczeń, stosujemy do tego technikę, która jest pochodną przeszukiwania lokalnego tzw. hill climbing, modyfikujemy przy tym zasadę przeskoku która w przypadku SA jest obierana losowo.



Rysunek 1: Przykład zasady działania Symulowanego wyżarzania, dla funkcji w płaszczyźnie euklidesowej.

Dzięki Hill climing’owi jesteśmy w stanie ominąć nasze minima lokalne i jest możliwość (w zależności od parametrów) dotrzeć do naszego globalnego minimum, na rysunku 1 zobrazowany jest właśnie to zagadnienie.

Symulowane wyżarzanie w TSP, rozpoczyna się od wygenerowania rozwiązania (R) początkowego, następnie na podstawie tego rozwiązania tworzone jest rozwiązanie sąsiednie (R\*) – następnie porównujemy ze sobą długości obu rozwiązań Δ = F(R) - F(R\*). Jeśli Δ < 0, rozwiązanie sąsiednie (R\*) zostaje zaakceptowane jako nowe początkowe rozwiązanie w następnej iteracji, z racji, że posiada krótszą drogę. W przeciwnym wypadku jeśli Δ >= 0, rozwiązanie sąsiednie (R\*) też może zostać zaakceptowane z określonym prawdopodobieństwem exp(-Δ / T) – gdzie T jest temperaturą. Temperatura jest zmniejszanie, zgodnie z obranym schematem chłodzenia.

Kluczowym zagadnieniem w przypadku algorytmu Symulowanego wyżarzania jest dobranie parametrów dla danej instancji, aby zapewnić, że losowe zmiany podczas wykonywania, a istnieją odgórne proponowane parametry startowe, która mogą się sprawdzić, ale nie dla wszystkich instancji, wszystko zależy od losowego rozkładu naszej instancji.

W celu zminimalizowania błędnych wyników znalezienia optymalnego rozwiązania, stosuje się parametry, które ustalamy w zależności od instancji, parametrami są:

1. Temperatura początkowa – T­­0
2. Temperatura końcowa – T­­f
3. Metoda chłodzenia (oraz również jej zmienne pomocnicze)
4. Rozwiązanie początkowe – c0
5. Sposób przekształcania rozwiązań (insert, invert, swap)
6. Liczba prób w ramach jednej epoki – f

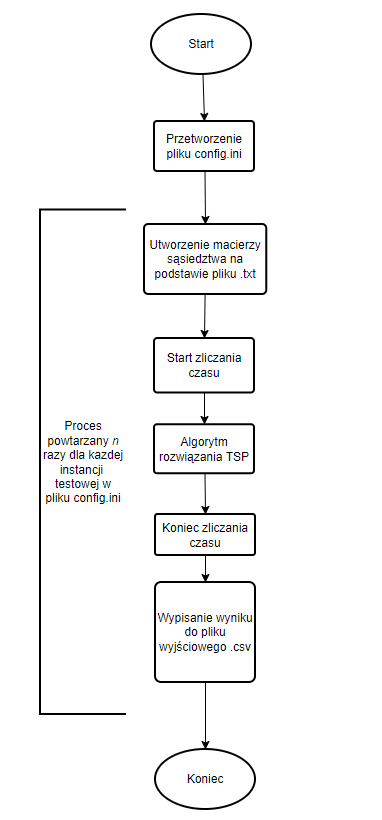
Temperaturę początkową wybieramy w zależności od wielkości instancji dla TSP

Jak widać na rysunku 2, przestrzeń startowa przy metodzie zstępującej jest dużo większa, od metody wstępującej. Algorytm Helda-Karpa używa metody w myśl zasady wstępującej.

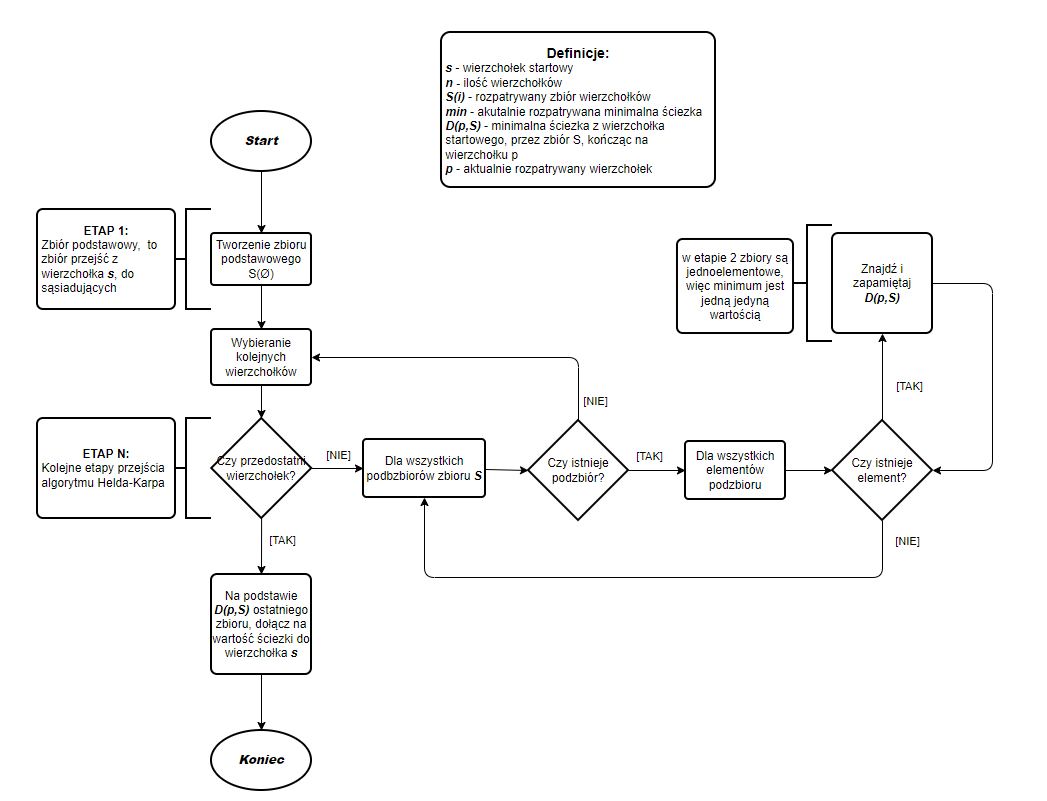
Algorytm ma złożoność czasową **O(n22n)** i złożoność pamięciową **O(n2n)**. Jest to algorytm znacząco lepszy od prezentowanego w poprzednim zadaniu (***zadanie 1: brute-force***), którego złożoność czasowa była wykładnicza - taką złożoność przy warunkach rozwiązywania problemu TSP, dla dużej liczby wierzchołków można uznać za nieefektywną.

Można dodatkowo napomknąć, że proces przeglądu zupełnego był w metodzie wstępującej, bez zapamiętywania etapów.

1. Algorytm



Rysunek 4: Ogólny schemat blokowy programu.



Rysunek 5: Schemat blokowy algorytmu Helda-Karpa dla problemu komiwojażera.

1. Dane testowe

Przy metodzie Helda-Karpa, nie ustawiamy dodatkowych parametrów. Pliki są sczytywane i przechowywane w formie macierzy sąsiedztwa, należy uwzględniać konkretny format zapisu danej instancji - gdzie w pierwszej linii zawieramy informację o liczbie wierzchołków *n*, a w następnych liniach pojawia się pełna macierz sąsiedztwa, o zadanym rozmiarze *n*, sczytywana po linii. Do wykonania badań wybrano następujący zestaw instancji:

1. tsp\_10.txt 212
2. tsp\_12.txt 264
3. tsp\_13.txt 269
4. tsp\_14.txt 282
5. tsp\_15.txt 291, http://jaroslaw.mierzwa.staff.iiar.pwr.wroc.pl/pea-stud/tsp/
6. atsp\_br\_17.txt 39, http://elib.zib.de/pub/mp-testdata/tsp/tsplib/atsp/index.html
7. tsp\_gr21.txt 2707
8. tsp\_gr24.txt 1272
9. tsp\_bays29.txt ?, http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/index.html
10. Procedura badawcza

Należało zbadać zależność czasu rozwiązania problemu od liczby wierzchołków.

W przypadku algorytmu Helda-Karpa przestrzeni rozwiązań dopuszczalnych nie występowały parametry programu, które mogły mieć wpływ na czas i jakość uzyskanego wyniku.

W związku z tym procedura badawcza polegała na uruchomieniu programu sterowanego plikiem inicjującym config.ini (format pliku: nazwa\_instancji liczba\_wykonań rozwiązanie\_optymalne [ścieżka optymalna]; nazwa\_pliku\_wyjściowego).

tsp\_10.txt 100 212 [0 3 4 2 8 7 6 9 1 5 0]

tsp\_12.txt 100 264 [0 1 8 4 6 2 11 9 7 5 3 10 0]

tsp\_13.txt 100 269 [0 10 3 5 7 9 11 2 6 4 8 1 12 0]

tsp\_14.txt 100 282 [0 10 3 5 7 9 13 11 2 6 4 8 1 12 0]

tsp\_15.txt 50 291 [0 10 3 5 7 9 13 11 2 6 4 8 14 1 12 0]

atsp\_br\_17.txt 25 39 [0 16 7 8 3 4 14 6 15 5 12 9 10 1 13 2 11 0]

tsp\_gr21.txt 10 2707 [0 6 7 5 15 4 8 2 1 20 14 13 12 17 9 16 18 19 10 3 11 0]

tsp\_gr24.txt 5 1272 [0 15 10 2 6 5 23 7 20 4 9 16 21 17 18 14 1 19 13 12 8 22 3 11 0]

wyniki.csv

Oraz osobny test (który nie był w stanie się wykonać, więcej o tym we wnioskach):  
tsp\_bays29.txt 1 ? [nie zbadano]

Każda instancja została wykonana określoną ilość razy, aby uzyskać zestaw wyników do analizy, obliczyć średnią i ewentualnie wykluczyć konkretne wyniki, które odbiegały od normy (tzw. błędy grube) – na podstawie wyliczonej średniej, odchylenie maksymalnie 10%. Wykluczenie konkretnych wyników i dalsza analiza została przeprowadzona w programie MS Excel, wraz z tworzeniem wykresu.

Na samym końcu pliku wynikowego musi być koniecznie znak nowej linii, aby mieć łatwy dostęp do odczytu nazwy pliku wyjściowego. W celu elastycznego odczytywania z pliku config.ini ów nazwy pliku. Poniżej przedstawiony fragment zawartości pliku wyjściowego:

tsp\_10.txt 100 212 [0 3 4 2 8 7 6 9 1 5 0];  
0,0007968;  
0,0007376;  
…  
tsp\_12.txt 100 264 [0 1 8 4 6 2 11 9 7 5 3 10 0];  
0,0228583;  
0,0047210;  
…

tsp\_gr21.txt 10 2707 [0 6 7 5 15 4 8 2 1 20 14 13 12 17 9 16 18 19 10 3 11 0];  
5,9047400;  
6,1227600;  
…

Testy zostały wykonywane na specyfikacji:

Windows 10 Pro N (v 10.0.19045)

Procesor Intel Core i3-8100 CPU @ 3.60 GHz

16 GB RAM

W celu przeprowadzenia jak najbardziej wydajnych pomiarów, testy przeprowadzone w środowisku konsolowym, bez debugowania (bezpośrednio uruchomiony plik main.exe), bez niepotrzebnych programów w tle.

Schemat procedury badawczej: wykorzystywana własna funkcja *testbench*, wywoływana dla każdej instancji w pliku config.ini (określoną *n* liczbę razy, w), w której uruchamiane są funkcje do liczenia czasu, oraz osobno funkcje do wypisywania do pliku wyjściowego (takie rozróżnienie podfunkcji w *testbench* jest bardzo ważne, ze względu na to, aby podczas zliczaniu czasu nie brać pod uwagę wpisywania do pliku).

Wykorzystujemy udostępniony w bibliotece systemowej *Win32* (*profileapi.h*) funkcję *QueryPerformanceFrequency (QPF)* – funkcje zwracającą częstotliwość licznika wydajności. Częstotliwość licznika wydajności jest ustalona podczas uruchamiania systemu i jest spójna dla wszystkich procesorów.

Funkcja ta zlicza taktowanie zegara. Podczas testów w pętli głównej, zagnieżdżamy wywołanie funkcji naszego algorytmu Helda-Karpa pomiędzy funkcje StartCounter() oraz GetCounter() (funkcje zbudowane na podstawie QPF).

Następnie w zależności od potrzeb bądź/lub otrzymywanych wyników skaluje się odpowiedną zmienną aby otrzymać wyniki w odpowiedniej precyzji, w tym zadaniu skalujemy PCFreq do sekund czyli zapis: PCFreq = double(li.QuadPart)/1000; zwraca nam czas w milisekundach.

1. Wyniki
   1. Złożoność czasowa

Wyniki zgromadzone zostały w plikach: wyniki.csv. Wyniki przedstawione zostały w postaci wykresu zależności czasu uzyskania rozwiązania problemu od wielkości instancji (rysunek 3).

Rysunek 6: Wykresy zależności czasu od liczby wierzchołków dla TSP bruteforce oraz TSP Helda-Karpa.

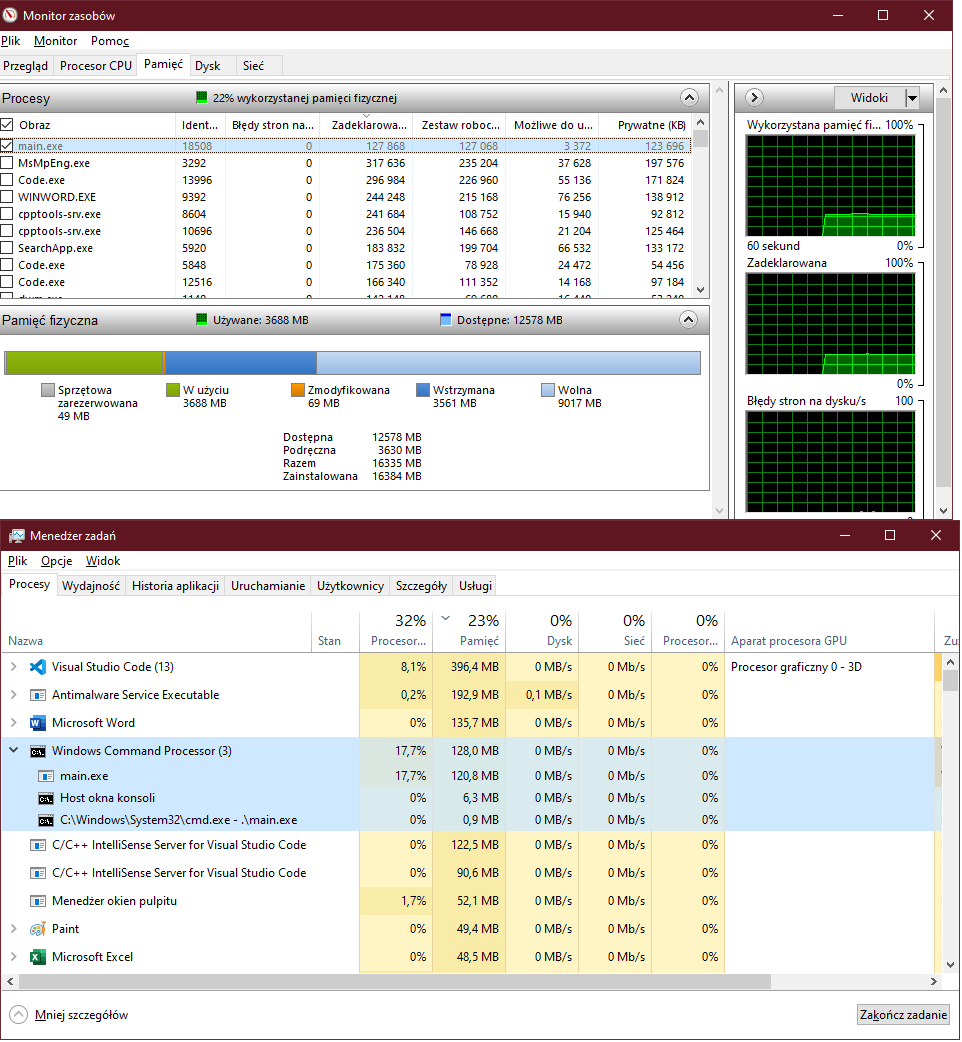
Tabela 1: Średnie pomiarów czasu dla Helda-Karpa i Bruteforce.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 10 | 12 | 13 | 14 | 15 | 17 | 21 | 24 |
| HeldKarp | 0,0007922 | 0,0041805 | 0,0098471 | 0,0216527 | 0,0503602 | 0,2954507 | 6,0969680 | 61,3752200 |
| Bruteforce | 0,003629 | 0,479002 | 6,227021 | 87,17829 | 1307,674 | … | … | … |

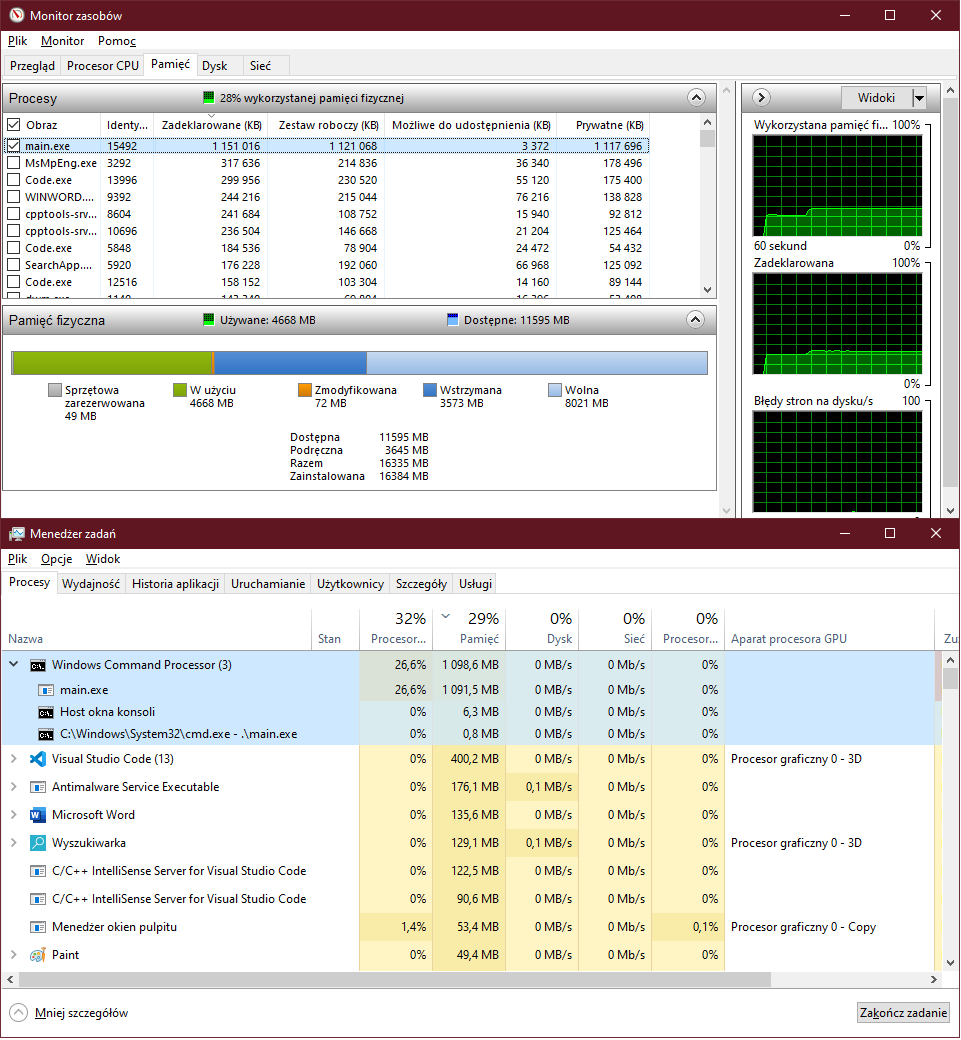
* 1. Złożoność pamięciowa

Dużo ciekawszym zagadnieniem do testowania w przypadku Helda-Karpa jest złożoność pamięciowa. Wszelkie testy zostały przeprowadzone przy pomocy narzędzi Windows 10 tj. menedżer urządzeń i monitor zasobów. Głównym parametrem na który zwracamy uwagę jest zużycie pamięci RAM, która jest kluczowa dla przechowywania wartości ścieżek już poznanych. Niżej testy dla 3 instancji, które najlepiej ukazują wzrost użytej pamięci na instancję:

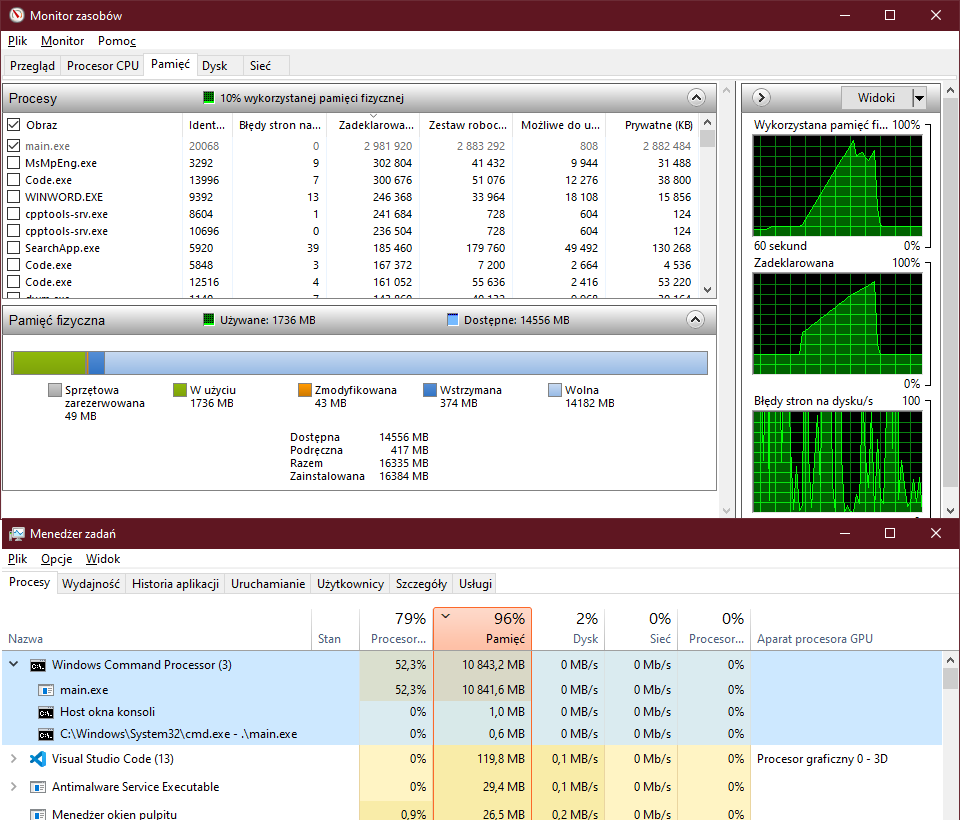
tsp\_gr21.txt 10 2707 [0 6 7 5 15 4 8 2 1 20 14 13 12 17 9 16 18 19 10 3 11 0]  
tsp\_gr24.txt 5 1272 [0 15 10 2 6 5 23 7 20 4 9 16 21 17 18 14 1 19 13 12 8 22 3 11 0]  
tsp\_bays29.txt 1 ? [nie zbadano]



Rysunek 7: Pomiary pamięci dla instancji tsp\_gr21.txt



Rysunek 8: Pomiary pamięci dla tsp\_gr24.txt



Rysunek 9: Pomiary pamięci dla tsp\_bays29.txt

1. Analiza wyników i wnioski

Jak wynika z rysunku 5 oraz tabeli 1, algorytm Helda-Karpa jest zasadniczo szybszy od metody przeglądu zupełnego. Przypominając, przegląd zupełny ma złożoność czasową: ***O(n!)***, więc już w przypadku 13 wierzchołków potrafił zająć parę godzin. W przypadku Helda-Karpa otrzymujemy szybsze rozwiązanie, dla większych instancji, złożoność wynosi: **O(n22n).** Jest to ogromne przyśpieszenie i zaoszczędzenie czasu, jednak w dalszej analizie powstał pewien kłopot, a tyczy się on złożoności pamięciowej, teoretycznie wynosi ona: **O(n2n).** Przechowanie takich rozmiarów w pamięci sprawia, że przy domowym sprzęcie 16 GB RAM dla instancji w okolicach 30 wierzchołków, program jest w stanie zabrać całą pamięć. Dlatego też bardziej prawdopodobnie szybciej dojdzie do przepełnienia pamięci, niż do długiego czasu oczekiwania na wynik dla większych instancji.

Testy pamięci na rysunkach 6, 7 i 8 ukazują duży wzrost pamięci, posłużyły one w celu znalezienia granicy, przy której program nie będzie miał już zasobów. Na moim sprzęcie instancja 29 wierzchołkowa nie była w stanie zostać obsłużona. Praktycznie 10 GB zajęte, przez program. To ukazuje największy minus algorytmu Helda-Karpa. Stosowanie go jednak dla mniejszych instancji niż 30 jest jak najbardziej poprawne, podobnie odpowiednie zastosowanie bruteforce dla odpowiednio małych instancji, też może dać zadowalające efekty. Jednak trzeba pamiętać, że są algorytmy bardziej efektywne, więc raczej te aktualnie przedstawione nie są one w użytku ogólnym.