Bogusław Malewski
Jakub Pietrowcew

Dokumentacja do projektu
Szeregowanie zadań

1. Treść zadania

Zaimplementować algorytm ewolucyjny planujący wykonanie N zadań na M procesorach w taki sposób aby suma opóźnień w wykonaniu była jak najmniejsza. Na jedno zadanie składa się: czas trwania T i deadline wykonania - wartość D. Obliczenia przerwać po pewnej liczbie generacji bez poprawy wyniku. WE: plik z listą zadań, liczba M procesorów. WY: harmonogram zadań i suma opóźnień.

2. Przyjęte założenia

Przyjęto następujący format pliku wejściowego: w pierwszej linii znajduje się liczba procesorów M, natomiast w każdej kolejnej linii umieszczone są dwie liczby - w kolejności określające czas wykonania T oraz deadline zadania D. Plik powinien być w formacie .txt

Każdemu zadaniu nadawany jest numer ID zgodny z jego kolejnością.

Podział odpowiedzialności w zespole

Bogusław: Opracowanie algorytmu, wstępna implementacja i ulepszanie kodu, przeprowadzenie testów, opracowanie dokumentacji

Jakub: rozwinięcie i zoptymalizowanie implementacji, opracowanie wyników testów, opracowanie dokumentacji

3. Opis algorytmu

Zadania są przechowywane w klasach które mają atrybut *nowy_proc* określający czy dany obiekt klasy jest zadaniem czy jest znacznikiem symbolizującym, że kolejne zadania są przydzielone do następnego procesora.

Przykładowo następujący wektor zadań:

id_zadania:	*1*	*0*	*2*	*0*	*3*	*4*
nowy_proc:	*false*	*true*	*false*	*true*	*false*	*false*

zostanie zinterpretowany jako następujący przydział zadań do procesorów

Procesor 1: 1

Procesor 2: 2

Procesor 3: 3, 4

W programie został zastosowany algorytm programowania ewolucyjnego:

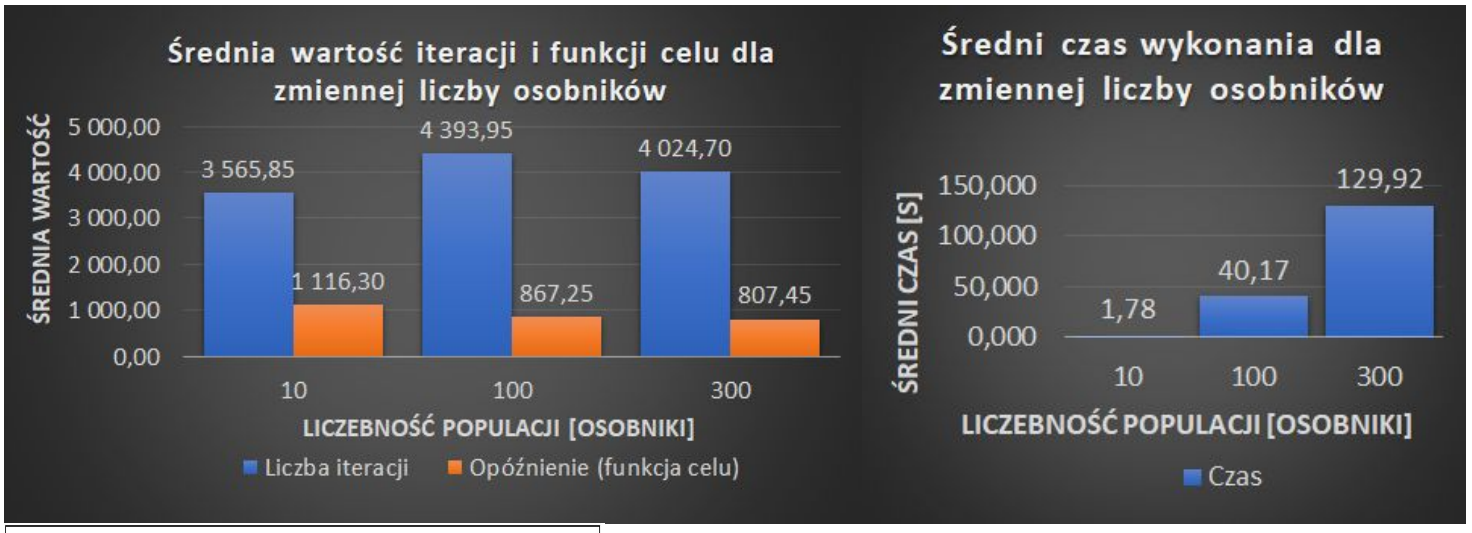
0. Wygeneruj P - populację MI osobników
1. Reprodukuj z P MI-elementową populację potomną R, stosując mutację dla każdego osobnika z P. Najlepszych 90% z P jest mutowanych poprzez zamianę miejsc 2 losowych zadań, natomiast 10% najgorszych z P jest mutowanych poprzez przesunięcie losowego zadania rozdzielającego na procesory o jedną pozycję w lewo bądź w prawo, a następnie zamianie tego punktu podziału z losowym zadaniem (takim które nie jest zadaniem rozdzielającym).
2. Utwórz nowe P jako MI osobników wybranych z $P \cup R$.
3. Jeśli w ostatnich N generacjach nie było poprawy minimalnej wartości funkcji celu, zakończ działanie algorytmu. W przeciwnym razie wróć do punktu 1.

Zastosowanie algorytmu z krzyżowaniem, nie było możliwe ponieważ, umiejscowienie jednego zadania w danym miejscu na procesorze nakłada ograniczenia na pozostałe zadania, więc nie można połączyć części uszeregowanych zadań od jednego rodzica i części od drugiego.

Opisane powyżej mutowanie 90% populacji zapewnia eksploatację, a 10% eksplorację.

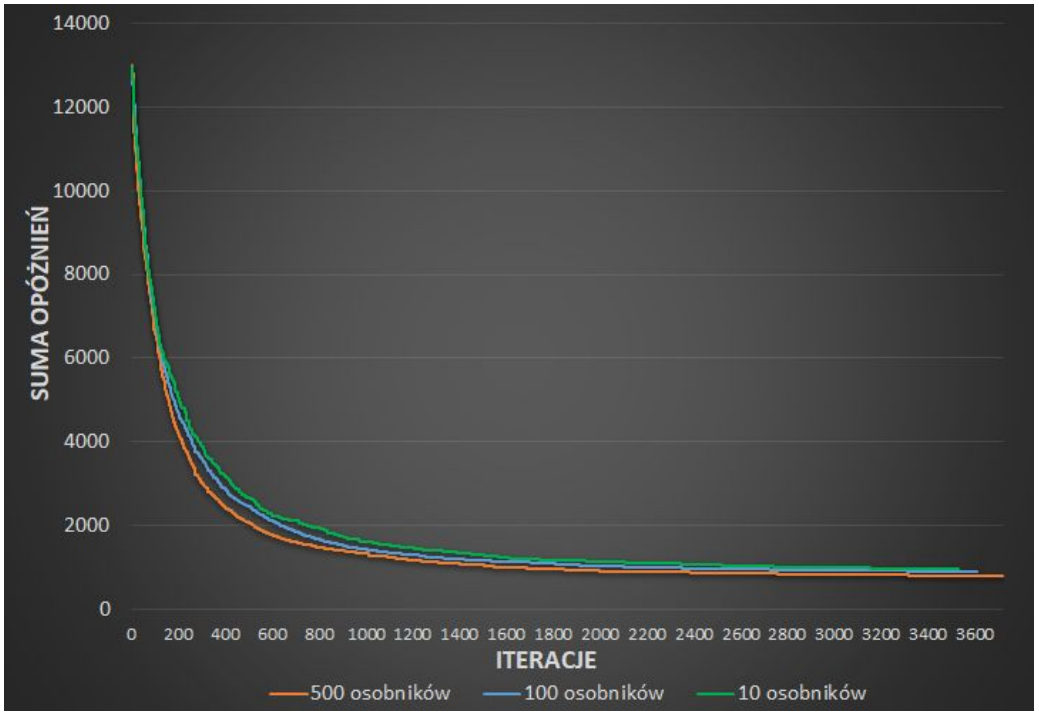
4. Przebieg testów oraz otrzymane wyniki

Dla każdego z następujących eksperymentów algorytm wykonano 25 razy i zebrano liczbę iteracji (po których algorytm się zatrzymał i zwrócił wynik), czas wykonania algorytmu oraz uzyskaną w wyniku działania minimalną wartość funkcji celu, czyli suma opóźnień.
Prezentujemy 2 najciekawsze eksperymenty z tych które przeprowadziliśmy.
a). Parametry: 200 zadań, 10 procesorów, 100 iteracji bez poprawy
Liczba osobników w populacji była zmienna i wynosiła: 10, 100 i 300.



Odchylenie standardowe dla poszczególnych parametrów			
parametr\osobniki	10	100	300
iteracje	784,57	887,49	548,14
czas	0,378	8,289	17,596
opóźnienie	157,13	63,29	52,74

Z powyższych wykresów wynika, że zwiększając liczbę osobników, zwiększa się jakość rozwiązania, jak również zwiększa się czas wykonania algorytmu. Jest to zgodne z intuicją. Mając więcej osobników do mutacji możemy w każdej iteracji zbadać więcej możliwych rozwiązań co zajmuje więcej czasu, ale daje lepsze rezultaty. Natomiast zależność pomiędzy liczbą osobników i liczbą iteracji jest bardziej skomplikowana. Początkowo zwiększając liczbę osobników liczba iteracji rośnie, a następnie zaczyna maleć. Można to wytłumaczyć w ten sposób, że dla "średniej" ilości osobników, algorytm więcej eksploruje niż dla "małej" liczby, a następnie potrzebuje więcej iteracji by eksploatować osobników otrzymanych w tych eksploracjach. Dla "dużej" liczby zachodzi mocna eksploatacja i mocna eksploracja w każdej iteracji, więc algorytm potrzebuje mniej iteracji.



Po lewej stronie przedstawiono przykładowe przebiegi algorytmu dla różnej liczby osobników. Można zauważyć, że w początkowej fazie działania algorytmu istnieje mała różnica pomiędzy trzema przypadkami. Dopiero w okolicach 200 iteracji można zauważyć przewagę większej liczby osobników. Krzywe na wykresie przypominają parabole o wzorze $y = \frac{a}{x}$, gdzie dla większej liczby osobników większy jest współczynnik a.

b) Parametry: 200 zadań, 100 iteracji, 100 osobników w populacji
Liczba procesorów była zmienna i wynosiła: 5, 10, 15

Odchylenie standardowe dla poszczególnych parametrów			
parametr\procesory	5	10	15
iteracje	653,55	887,49	691,01
czas[s]	6,787	8,289	6,629
opóźnienie	154,28	63,29	21,95

Z powyższych wykresów wynika, że algorytm radzi sobie lepiej z harmonogramowaniem zadań na większej ilości procesorów. Wraz ze wzrostem procesorów maleje liczba iteracji i czas wykonania.

5. Wnioski i przemyślenia

Dla stosunkowo małej liczby zadań np.100 można stosować całkiem duże liczby osobników np. 500 ,by uzyskać dobrą jakość rozwiązania w rozsądnej ilości czasu. Dla zadań bardzo dużych jak 20000, duża liczba osobników np. 500, dałaby dobrą jakość rozwiązania lecz czas pracy byłby bardzo długi. W sytuacjach w których czas znalezienia rozwiązania jest ważny, a jakość rozwiązania stanowi mniejszy priorytet warto zastosować minimalną liczbę osobników tj. 10. Dla 10 osobników stosunek czasu wykonania do jakości rozwiązania jest **bardzo** korzystny.
Potencjalnym kierunkiem rozwoju algorytmu jest zmodyfikowanie go tak, by na początku działania bardziej eksplorował, a z biegiem czasu coraz bardziej eksploatował.

6. Instrukcja użytkownika

Program jest uruchamiany z konsoli i przyjmuje 3 argumenty:
szereguj.out [nazwa pliku wejściowego] [iteracje bez poprawy] [liczba osobników]

[nazwa pliku wejściowego] - nazwa pliku z którego zostaną odczytane dane.
[iteracje bez poprawy] - liczba iteracji w których nie było poprawy wyniku i po których program zakończy działanie.
[liczba osobników] - liczba osobników populacji.
z czego wymagany jest tylko [nazwa pliku wejściowego]. Jeśli nie poda się pozostałych argumentów przyjmowane są domyślne wartości czyli: szereguj.exe [nazwa pliku wejściowego] 100 100.
[iteracje bez poprawy] przyjmuje wartości z zakresu [30;500].
[liczba osobników] przyjmuje wartości z zakresu [10;500].

Program będzie na bieżąco wypisywał przebieg działania programu, a po zakończeniu działania algorytmu wypisze wynik do konsoli i do pliku o nazwie [nazwa pliku wejściowego]_WYNIK.txt.