POLITECHNIKA BIAŁOSTOCKA

WYDZIAŁ INFORMATKI
KATEDRA OPROGRAMOWANIA

PRACA DYPLOMOWA MAGISTERSKA

TEMAT: ZASTOSOWANIE ALGORYTMU GENERALIZED EXTREMAL OPTIMIZATION W PROBLEMIE SZEREGOWANIA ZADAŃ

WYKONAWCA: PAWEŁ KACZANOWSKI

PODPIS: .....

PROMOTOR: DR INŻ. ANNA PIWOŃSKA

PODPIS: .....

BIAŁYSTOK 2013 ROK

Karta dyplomowa

<table>
<tr><td>POLITECHNIKA BIAŁOSTOCKA

Wydział Informatyki

Katedra Oprogramowania</td><td>Studia stacjonarne
II stopnia</td><td>Nr albumu studenta: 68911

Rok akademicki: 2012/2013

Kierunek studiów: Informatyka

Specjalność: Inżynieria Oprogramowania</td></tr>
<tr><td colspan="3">KACZANOWSKI PAWEŁ

TEMAT PRACY DYPLOMOWEJ:

Zastosowanie algorytmu generalized extremal optimization w problemie szeregowania zadań

Zakres pracy:
1. Opis problemu szeregowania zadań w systemach wieloprocesorowych
2. Przedstawienie koncepcji i zasady działania algorytmu generalized extremal optimization
3. Zaprojektowanie i zaimplementowanie aplikacji wykorzystującej algorytm generalized extremal optimization do rozwiązywania problemu szeregowania zadań
4. Wykonanie badań eksperymentalnych na danych testowych
Słowa kluczowe: szeregowanie zadań, algorytm GEO

dr inż. Anna Piwońska
Imię i nazwisko, stopień/ tytuł promotora - podpis

prof. dr hab. Leon Bobrowski
Imię i nazwisko kierownika katedry - podpis</td></tr>
<tr><td>.....
Data wydania tematu pracy dyplomowej - podpis promotora</td><td>.....
Regulaminowy termin złożenia pracy dyplomowej</td><td>.....
Data złożenia pracy dyplomowej - potwierdzenie dziekanatu</td></tr>
<tr><td colspan="3">.....
Ocena promotora

.....
Podpis promotora</td></tr>
<tr><td>.....
Imię i nazwisko, stopień/ tytuł recenzenta</td><td>.....
Ocena recenzenta</td><td>.....
Podpis recenzenta</td></tr>
</table>

Thesis topic:

Heuristic generalized extremal optimization (GEO) can be successfully applied to task scheduling problem to allow finding shortest execution time of specified program's graph.

SUMMARY

With fast development of high cost computers and multiprocessors systems, it has become necessary to optimise the use of these resources. The scheduling problem concerns such ordering of tasks that maximizes the use of all resources or executes all tasks in the shortest possible time. This dissertation focuses on finding the most efficient ordering to minimize execution time.

For the purpose of this task, gradient algorithms are used due to the relatively sizeable search space as well as them belonging to NP-complete problem set (i.e. such that cannot be solved in polynomial time).

Instead, this dissertation utilizes heuristic and meta-heuristic algorithms, which are based on the natural world observation. Genetic algorithms or simulated annealing algorithms, whose behavior is similar to that of genetic mutation process, ants' behaviour or annealing of metals, allow limiting the search space. An example of such heuristic algorithms, which this thesis uses, is generalized extremal optimization method (GEO).

GEO is easy to implement and can be applied simultaneously to several types of variables (district, integer and continuous). It allows solving unconstrained and constrained problems in both convex and disjoint space. Its fault, however, is that it cannot always guarantee finding the best solution – which is the case with all heuristic algorithms.

This paper focuses on design and implementation on a GEO algorithm that solves the scheduling problem. Effectively the algorithm should allow quick and efficient search result for the optimal solution or one closest to it.

This dissertation presents theoretical principles behind scheduling and GEO algorithm. It proposes and closely describes a GEO algorithm that is suitable for solving scheduling problem. The final chapter presents and analyses the results of a study carried out with the use of testing program graphs. It also explains different behaviour of the algorithm depending on parameters as well as statistical data pertaining to the odds of finding the optimal solution and the required number of objective function executions.

In next chapters describe basics of scheduling problem and GEO algorithm. Propose and describe GEO algorithm for scheduling problem. In last sections illustrated results of conducted experiments on testing program graphs. Describe behavior of algorithm

dependent of input parameters. Presents statistic chances to find optimal solution and necessary number of objective function executions.

Spis treści

1. Wstęp

W wielu różnych dziedzinach badacze muszą zmierzyć się z problemami obliczeniowymi. Często muszą przeanalizować ogromne ilości danych wykorzystując do tego zasoby komputerowe i bardzo zaawanasowane algorytmy. Dzięki powstaniu systemów wieloprocesorowych i wielkich klastrów złożonych z wielu jednostek obliczeniowych takie algorytmy mogą być wykonywane równolegle. Powodując skrócenie czasu analizy danych ułatwiają pracę naukowców. Koszty ogromnych systemów są bardzo wysokie i bardzo ważne jest, aby ich wykorzystanie było maksymalne.

Problem szeregowania zadań dotyczy optymalnego wykorzystania zasobów komputerowych oraz skrócenia czasu wykonywania poszczególnych modułów złożonego algorytmu. Poniższa praca skupia się na znalezieniu takiego uszeregowania zadań, aby czas ich wykonywania był najkrótszy. Okazuje się jednak, że rozwiązanie problemu szeregowania jest problemem NP-zupełnym, co oznacza że nie są znane algorytmy rozwiązujące ten problem w czasie wielomianowym.

Powstanie algorytmów heurystycznych i metaheurystycznych takich jak algorytmy genetyczne spowodowało, że problemy klasy NP mogą być rozwiązywane w krótszym czasie, ich złożoność obliczeniowa jest dużo niższa. Podstawową wadą tych algorytmów jest brak 100% pewności, że znalezione rozwiązanie jest najlepsze. Jedną z metod heurystycznych jest uogólniona ekstremalna optymalizacja (*generalized extreme optimization,* GEO) pozwalająca na szybkie rozwiązywanie złożonych problemów.

Celem pracy jest zaprojektowanie i zaimplementowanie aplikacji wykorzystującej algorytm uogólnionej ekstremalnej optymalizacji do rozwiązania problemu szeregowania oraz przeprowadzenie badań eksperymentalnych na danych testowych.

W rozdziale drugim opisano problem szeregowania. Zaproponowano sposób reprezentacji programu równoległego oraz zależności procesorów, tak aby problem mógł zostać zinterpretowany w prosty sposób przez algorytm programu komputerowego. Omówiono także najlepsze polityki szeregowania mające bardzo ważną rolę przy optymalizacji czasu wykonania.

Rozdział trzeci prezentuje algorytm uogólnionej ekstremalnej optymalizacji oraz jego genezę. Przedstawia także model danych wykorzystywany przez algorytm i sposób adaptacji różnych problemów do tego modelu.

Szczegółowe omówienie proponowanej metody implementacji problemu szeregowania z wykorzystaniem GEO znajduje się w rozdziale czwartym. Zdefiniowano w nim szczegółowo problem, opisano sposób tworzenia modelu. Wymieniono także kolejne kroki działania procedury wraz z dogłębnym opisem.

Wyniki badań znajdują się w rozdziale piątym. Opisano w nim poddane eksperymentom grafy programów, które powinny zostać uszeregowane. Przedstawiono typowe przebiegi algorytmu dla systemów z 2, 4 i 8 procesorami. Zestawiono także wyniki pracy z wynikami działania algorytmu genetycznego.

Ostatni rozdział podsumowuje wyniki pracy i odpowiada na pytanie, czy algorytm GEO może być konkurencyjny w stosunku do algorytmów genetycznych przy implementacji dla problemu szeregowania. Odwołując się do wyników badań przedstawia w jaki sposób konfigurować algorytm, omawia jego słabe i mocne strony.

2. Problem szeregowania zadań

Problemy szeregowania dotyczą wielu dziedzin życia takich jak rozdzielanie procesów przez system komputerowy, zarządzanie zespołem projektowym przez przydzielanie zadań i kolejności ich wykonywania dla poszczególnych osób, wykorzystanie maszyn w przedsiębiorstwie produkcyjnym. Odpowiednie uszeregowania pozwala na zaoszczędzenia czasu pracy, maksymalne wykorzystanie zasobów, a co za tym idzie poprawienie wydajności w danej dziedzinie.

W przypadku planowania wykonania pewnych czynności (zadań) w momencie korzystania z ograniczonej liczby zasobów (procesorów) można stwierdzić, że kolejność wykonanych zadań, ich uszeregowanie, pozwala na osiągnięcie lepszych lub gorszych rezultatów. Oczekiwania mogą dotyczyć równomiernego rozkładu zadań na zasoby lub czasów realizacji całego przedsięwzięcia. Poniższa praca dotyczy optymalizacji czasu szeregowania [1]. Przykładowymi sytuacjami w których takie szeregowanie ma istotne znaczenie są:

- określenie kolejności zadań, obliczeń w wykonywanym algorytmie programu komputerowego. W tym wypadku zadaniem jest krok algorytmu, a zasobem czas procesora,
- stworzenie harmonogramu prac inwestycji budowlanej [2]. Zasobami są ludzie, sprzęt budowlany, a szeregowaniu podlegają prace budowlane takie jak np. położenie powierzchni asfaltowej,
- zdefiniowanie modelu produkcyjnego pewnych dóbr złożonych z wielu podzespołów. Zasobem może być tu praca ludzka, dostęp do urządzeń koniecznych do wykonania podzespołów, uszeregowaniu podlega tworzenie kolejnych podzespołów.

Poniższa praca skupia się na problemie szeregowania w systemach komputerowych. W zależności od warunków istnieje wiele algorytmów, które pozwalają na znalezienie optymalnego rozdzielenia zadań na zasoby [3]. Pierwszego podziału można dokonać na dwa typy szeregowania: dynamiczne i statyczne. Z szeregowaniem dynamicznym najczęściej można się spotkać przy rozdzielaniu czasu procesora pomiędzy procesy, gdy pewne zmienne np. czasy zakończenia zadania są nieznane przed rozpoczęciem programu.

Najczęściej stosowanymi algorytmami przy tym typie szeregowania są: FIFO[1], wg najkrótszego czasu szeregowania (SJF - ***S**hortest **J**ob **F**irst*) i wg najkrótszego czasu pozostałego do wykonania (SRTF - ***S**hortest **R**emaining **T**ime **F**irst*). Z kolei szeregowanie statyczne pojawia się, gdy znany jest czas wykonania wszystkich zadań, znane są także zasoby i koszty przesyłania zadań. Poniższa praca skupia się na problemie statycznego szeregowania.

Następnego podziału można dokonać ze względu na liczbę zasobów potrzebnych do wykonania czynności. Ich liczba może być ograniczona lub nieskończona. Drugi przypadek jest rzadko spotykany w świecie rzeczywistym. Najistotniejsze jednak w tym przypadku byłoby znalezienie najkrótszego czasu szeregowania biorąc pod uwagę wyłącznie zależności pomiędzy czynnościami oraz koszt transmisji danych pomiędzy zasobami. Praca rozważa problem ograniczonych zasobów, w którym wielokrotnie kilka zadań mogłoby być wykonanych równocześnie, jednak ze względu na brak wolnych procesorów nie jest to możliwe.

Kosztem przesyłania danych pomiędzy zasobami określa się czas jaki jest konieczny na transmisję danych, produktów wykonanego zadania do zasobu, na którym zostanie wykonane kolejne zadanie. Koszt musi zostać uwzględniony wyłącznie jeśli zadania są zależne. Częstym założeniem, istotnym w tej pracy, jest zerowy koszt w przypadku gdy zadania były wykonywane na tym samym procesorze, zasobie. Z praktycznego punktu widzenia jeśli informacja może zostać zachowana w danym zasobie, bez kosztu jej składowania, nie ma potrzeby wykonywania dodatkowych czynności.

Rodzaje algorytmów szeregowania możemy podzielić również ze względu na uwzględniające koszty przesyłania przy szeregowaniu (np. w systemach z pamięcią rozproszoną – *message passing system*) lub je ignorujące (np. w systemach z pamięcią współdzieloną – *shared memory system*). Także istotne jest czy przesyłanie danych może odbywać się równolegle, czy musi być ono kolejkowane.

Zadania, które podlegają szeregowaniu, można podzielić na przerywalne i nieprzerywalne. Pierwsze z wymienionych dotyczą przypadku, gdy wykonywaną czynność w dowolnym momencie można rozbić na wiele zadań, następnie można je wznowić na innych procesorach. Z teoretycznego punktu widzenia powinno to pozwolić osiągnąć lepsze czasy szeregowania, jednak należy uwzględnić dodatkowe koszty związane z przesłaniem podzielonego zadania, które mogą być niewspółmierne do zysku związanego

[1] FIFO (z ang. *first in first out*) – oznacza że zadania są wykonywane w kolejności w jakiej się pojawiły

ze wcześniejszym zakończeniem innych zadań [3]. Koszty takiego podziału nie mogą być również obliczone na starcie algorytmu, a dopiero w momencie przenoszenia podzielonego zadania na inny zasób. Pamiętać należy także o tym, iż nie w każdych warunkach jest możliwe podzielenie zadania na części. Jest to także bardzo trudny przypadek do implementacji.

Stworzono algorytmy, które można wykorzystać w sytuacji, gdy czasy wykonania wszystkich procesów są stałe. Mimo że nie jest to często spotykane w rzeczywistych problemach, to wartym podkreślenia faktem jest możliwość rozwiązania takiego problemu przy dodatkowych warunkach jak problemu klasy P-zupełnego[2] [4]. W poniższej pracy czas wykonywania poszczególnych zadań nie jest obłożony tym ograniczeniem, istotne jest że nie może być ujemny lub zerowy.

Udowodniono, że problem szeregowania jest problemem NP-zupełnym [5], z wyjątkiem trzech sytuacji:

- przy drzewiastej strukturze grafu zależności zadań z arbitralną wartością procesorów [4],
- dla grafów procesów z jednakowym czasem wykonania zadań na dwóch procesorach [6],
- dla uporządkowanych grafów przedziałowych (*interval-ordered*) [7] [8] z jednakowym czasem wykonania zadań i stałą liczbą procesorów.

Oznacza to, że nie jest możliwe szybkie znalezienie najlepszego rozwiązania i głównie używa się algorytmów heurystycznych [9], które próbują znaleźć optimum we względnie krótkim czasie, nie gwarantując jednak jego odnalezienia.

2.1 Graf programu równoległego

W przypadku szeregowania statycznego z uwzględnieniem kosztów przesłania danych najefektywniejszym i najczytelniejszym sposobem prezentacji zależności pomiędzy zadaniami jest użycie acyklicznego, ważonego grafu skierowanego $G_p=(V_p,E_p)$, zwanego grafem programu. V_p jest zbiorem N_p wierzchołków grafu G_p reprezentujących elementarne zadania. E_p jest zbiorem krawędzi grafu opisującym zależności pomiędzy modułami.

[2] Problem P-zupełny to problem, którego rozwiązanie można znaleźć w czasie wielomianowym oraz należy do klasy P i dowolny problem należący do P może być do niego zredukowany w czasie wielomianowym

Przykład takiego grafu jest widoczny na Rys. 2.1. Węzeł grafu symbolizuje proces, zaś krawędź zależność pomiędzy zadaniami. W węźle widoczne są dwie wartości: numer procesu oraz czas wykonania (wyróżniany w komórce z szarym tłem). Zadanie numer 0 wykona się w czasie 3 jednostek, z kolei 2-gie zadanie w czasie 4 jednostek. Krawędź prowadząca od węzła nr 0 do węzła nr 2 oznacza, że możliwe jest wykonanie zadania nr 2 dopiero, gdy zakończy się zadanie 0. Wartość przy krawędzi oznacza koszt przesłania danych pomiędzy modułami. Należy zwrócić uwagę, że koszt przesłania liczymy według wzoru:

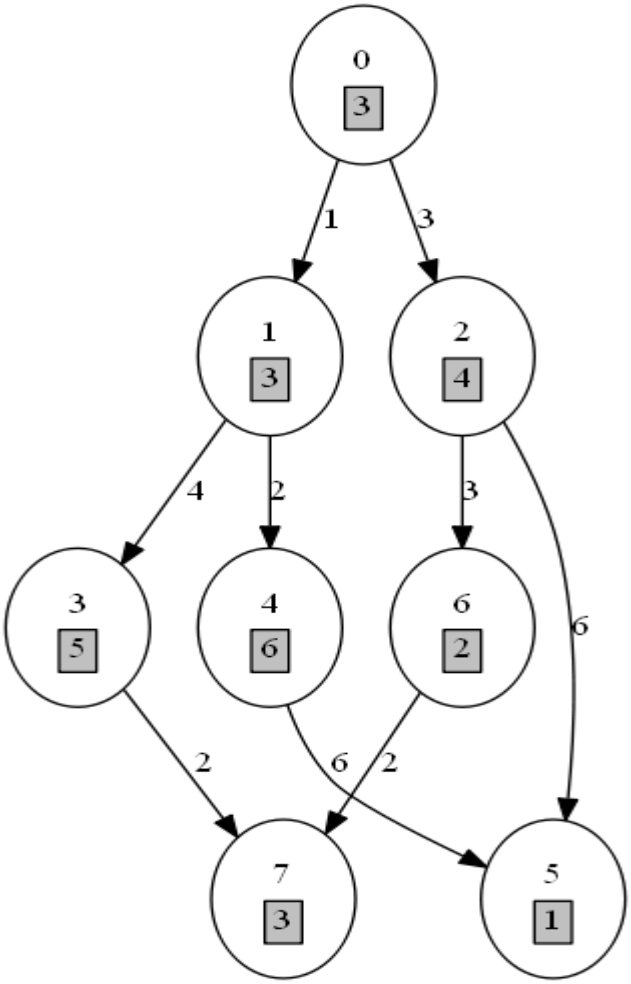
$$t_{kl} = a_{kl} * d_{kl} \tag{1}$$

gdzie: t_{kl} – czas komunikacji między modułami k i l,

d_{kl} – minimalna odległość pomiędzy zasobami wykorzystywanymi przez moduły k i l, odległość definiuje się według grafu procesów opisanego w rozdziale 2.2,

a_{kl} – waga krawędzi pomiędzy węzłami k i l.

Jeśli procesy są alokowane na tym samym procesorze, odległość d_{kl} wynosi 0 i koszt także jest zerowy. Z kolei, gdy procesory nie są połączone bezpośrednio, należy zsumować odległość pomiędzy nimi uwzględniając koszt komunikacji (może być różny pomiędzy procesorami). W pracy założono, że czas wykonywania poszczególnych modułów jest niezależny od procesora na którym zostanie wykonany.

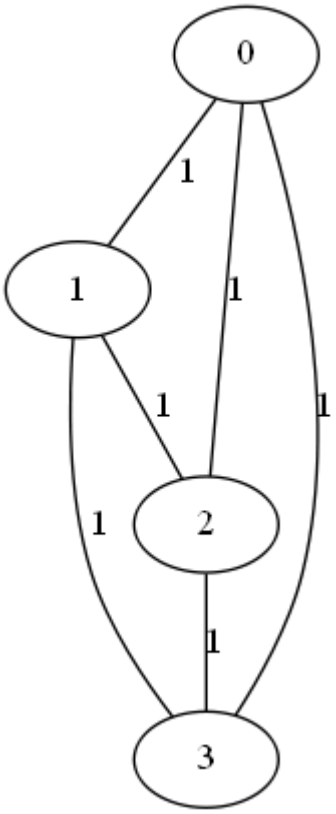


Rys. 2.1 Przykładowy graf procesów rozpatrywany w ramach problemu szeregowania

2.2 Graf procesorów

Zasoby można podobnie jak zadania przedstawić w formie nieskierowanego, nieważonego grafu $G_s=(V_s,E_s)$, zwanego grafem systemu. V_s to zbiór wierzchołków grafu G_s reprezentujących procesory. E_s jest zbiorem krawędzi grafu G_s stanowiących reprezentację dwukierunkowych połączeń między procesorami.

Rys. 2.2 pokazuje przykładowy graf cykliczny dla 4 procesorów. Wszystkie procesory są ze sobą połączone, co oznacza że można wymieniać dane pomiędzy każdym z nich. Koszt w każdym wypadku wynosi 1 (wartość przy krawędzie).



Rys. 2.2 Graf przedstawiający powiązanie procesorów wraz z kosztem zmiany procesora

Liczba procesorów dla problemu szeregowania jest nieograniczona, przy czym minimalną logiczną wartością są 2 procesory. Także koszty komunikacji mogą być różne i zależne od przypadku. Na potrzebny poniższej pracy skorzystano z 2, 4 i 8 procesorów z założeniem, że komunikacja może odbywać się na zasadzie każdy z każdym, a koszt transmisji danych wynosi 1.

2.3 Polityka szeregowania

Czas szeregowania grafu procesów zależy od alokacji zadań na procesorach oraz od kolejności w jakiej zadania zostaną uruchomione na procesorach. Polityka szeregowania powinna określać jaka będzie kolejność wykonywania czynności na procesorze, które zadanie powinno zostać wykonane najpierw w przypadku gdy w jednej chwili więcej niż jeden moduł może zostać wykonany.

Ogólnie politykę szeregowania, w przypadku opisywanego problemu, można określić w następujący sposób. Najpierw wybierane są zadania, które mogą zostać uruchomione w danej chwili. Jeśli modułów gotowych do uszeregowania jest więcej niż jeden pojawia się konflikt i należy wybrać jeden z nich. Ich wybór może zostać określony na wiele sposobów, np. wybierając zadania o najkrótszym czasie wykonania, o najniższym

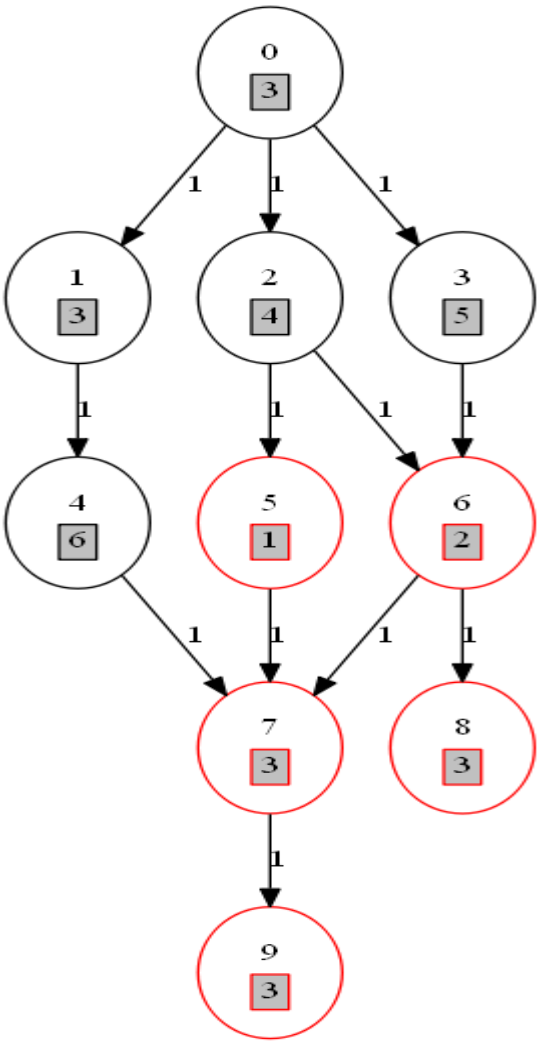
numerze czy wręcz losowo. Jednak ze względów jakościowych tylko kilka rozwiązań warto zastosować w praktyce, są to polityki oparte na:

- liczbie następników liczonej rekurencyjnie,
- wysokości,
- wysokości dynamicznej.

Metoda oparta na liczbie następników liczonych rekurencyjnie polega na wyborze zadania, które ma więcej węzłów-potomków. Licząc potomków ze wzoru (2) bierzemy pod uwagę wszystkich pośrednich potomków, czyli wszystkie zadania zależne od bazowego. Na Rys. 2.3 zaznaczono kolorem czerwonym wszystkich następników modułu 2, ich liczba jest równa 7, gdyż część zadań zostanie zliczona dwukrotnie.

$$p_k = nast_k + \sum_{i=1}^{nast_k} p_{k_i} \tag{2}$$

gdzie: p_k – liczba następników węzła k,
$nast_k$ – liczba bezpośrednich następników węzła k,
k_i – i-ty następnik węzła k.



Rys. 2.3 Moduły będące następnikami węzła 2

Obliczając wysokość dla danego modułu wybiera się maksymalną wartość z sumy wag wierzchołków i krawędzi na ścieżce prowadzącej do dowolnego modułu wyjściowego. Do sumy włącza się wagę węzła wejściowego i wyjściowego. Na przykład na Rys. 2.3 istnieją 3 ścieżki prowadzące z węzła 2 do węzłów wyjściowych 9 i 8, są nimi drogi:

- 2 -> 6 -> 8, wysokość wynosi 4 + 1 + 2 + 1 + 3 = 11,
- 2 -> 6 -> 7 -> 9, wysokość wynosi 4 + 1 + 2 + 1 + 3 + 1 + 3 = 15,
- 2 -> 5 -> 7 -> 9, wysokość wynosi 4 + 1 + 1 + 1 + 3 +1 + 3 =14.

Wysokość dla procesu 2 wynosi *max*(11, 15, 14) = 15.

Warunkiem obliczenia wysokości dynamicznej jest wiedza na temat przypisania procesów modułom. Wartość wysokości dynamicznej to suma wag wierzchołków i krawędzi grafu z uwzględnieniem kosztu zmiany procesora. W przypadku procesu

alokowanego na tym samym procesorze koszt zmiany wynosi 0. Przyjmując że koszt zmiany procesora wynosi 1, a przyporządkowanie modułów to:

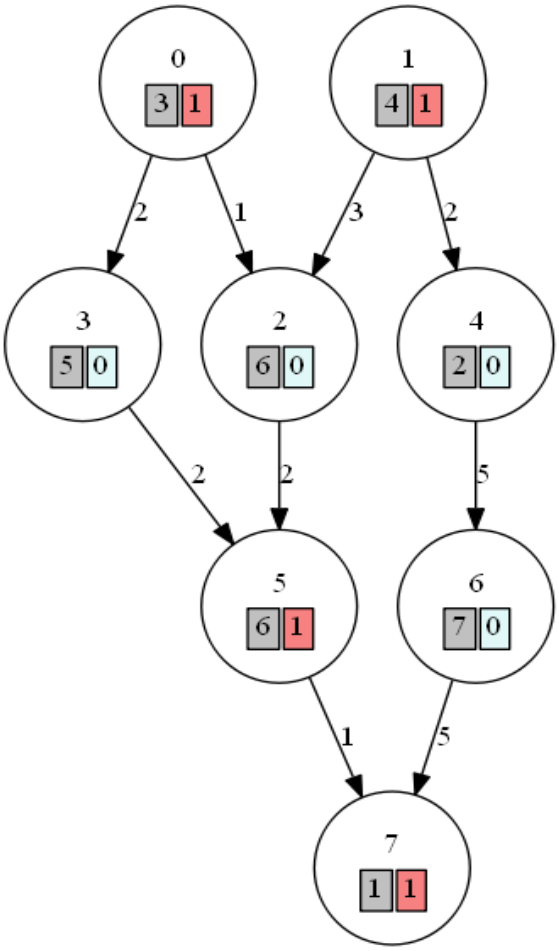
- procesor 0: 2,6,7,9,
- procesor 1: 5,8,

dla wierzchołka 2 pojawiają się 3 ścieżki:

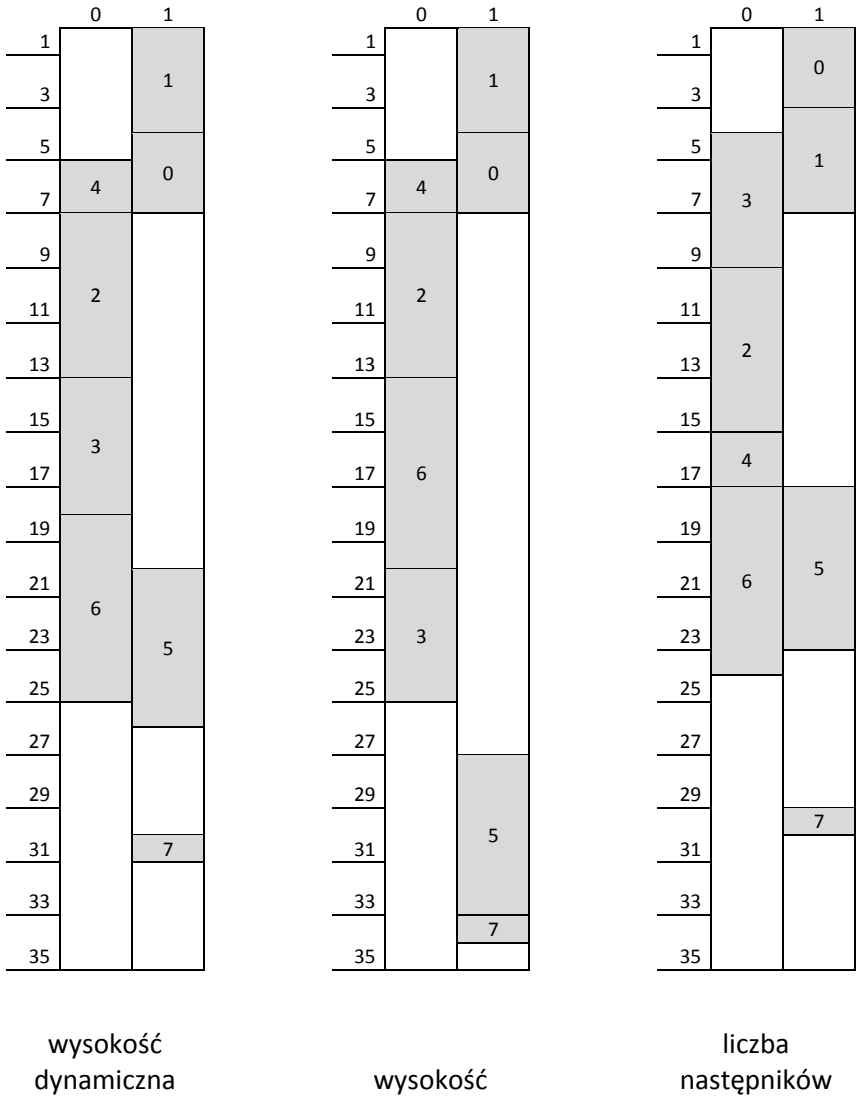
- 2 -> 6 -> 8, wysokość wynosi 4 + 1 * 0 + 2 + 1 * 1 + 3 = 10,
- 2 -> 6 -> 7 -> 9, wysokość wynosi 4 + 1 * 0 + 2 + 1 * 0 + 3 +1 * 0 + 3 = 12,
- 2 -> 5 -> 7 -> 9, wysokość wynosi 4 + 1 * 1 + 1 + 1 * 1 + 3 +1 * 0 + 3 =13.

Tym razem najlepszą ścieżką okazała się 3 droga z wysokością równą 13.

Wyniki badań eksperymentalnych pokazały, że najlepsze wyniki otrzymuje się przy wykorzystaniu metody obliczania dynamicznej wysokości i została ona wykorzystana przy implementacji algorytmu szeregowania na potrzebny poniżej pracy. W przypadku, gdy dla kilku modułów wysokości dynamiczne są identyczne wybierany jest moduł o niższym numerze.



Rys. 2.4 Graf testowy z uszeregowaniem na dwa procesory



Rys. 2.5 Diagram Gantta szeregowania dla trzech różnych polityk szeregowania

Rys. 2.4 przedstawia przykładowy graf programu z podziałem na dwa procesory. Z kolei na Rys. 2.5 pokazano trzy diagramy Gantta dla tego grafu z użyciem różnych polityk szeregowania. Jak można zauważyć przy różnych sposobach wyboru kolejności wykonywania zadań osiągnięto diametralnie różne czasy szeregowania.

3. Algorytm uogólnionej ekstremalnej optymalizacji

Ewolucja w naturze skutkuje tym, że organizmy dostosowują się do bardzo skomplikowanych problemów w środowisku naturalnym. Ich zmiany pozwalają np. na oszczędzenie energii, lepsze dostosowanie do środowiska w którym żyją. Natura wykształca samoregulujące się, bardzo złożone mechanizmy rozwiązujące skomplikowane problemy. Funkcjonowanie mózgu, wyżarzanie metali, a nawet obserwacja zachowania mrówek dostarcza wielu interesujących spostrzeżeń. Od ponad 30 lat badacze zainspirowani tymi mechanizmami tworzą algorytmy numeryczne pozwalające rozwiązywać problemy optymalizacyjne, podobnie jak ma to miejsce w przyrodzie.

Wśród metod optymalizacyjnych powstałych na bazie obserwacji środowiska znajdują się algorytmy genetyczne [10] [11], algorytmy symulowanego wyżarzania (*simulated annealing*) oraz wiele innych im pochodnych, które są obecnie jednymi z najczęściej stosowanych metod rozwiązywania problemów optymalizacyjnych w inżynierii i nauce. Zastosowanie tych rozwiązań pozwala często na uniknięcie ograniczeń spotykanych w tradycyjnych algorytmach gradientowych takich jak multimodalne lub rozłączne przestrzenie projektowania, używanie zmiennych ciągłych i dyskretnych, wystąpienie silnych nieliniowości w funkcji celu lub wielokrotnych ograniczeń. Także zdolność łatwej adaptacji tych algorytmów do szerokiej gamy problemów optymalizacyjnych wyróżnia je na tle innych metod. Wadą tych rozwiązań jest często konieczność wielokrotnego wykonania funkcji celu w poszukiwaniu najlepszego rozwiązania, aby działanie algorytmu było efektywne. Jeśli funkcja celu jest złożona i jej wykonanie zajmuje dużo czasu to użycie algorytmu może okazać się niepraktyczne. Jednak ze względu na rosnące możliwości zasobów obliczeniowych i możność wykorzystania metod hybrydowych, algorytmy te mogą być wykorzystywane w coraz większym stopniu także przy tak trudnych klasach problemów.

Opracowano także algorytm [12] ekstremalnej optymalizacji (*extremal optimization, EO*) bazujący na zasadach naturalnej selekcji, który nie wykorzystuje typowej dla algorytmów genetycznych reprodukcji populacji. Bazuje on na zaproponowanym przez Baka i Sneppena [13] modelu ewolucyjnym do rozwiązywania trudnych problemów optymalizacji kombinatorycznej. Szkielet algorytmu oparto na uproszczonym modelu naturalnej selekcji tak, aby pokazać powstanie samoorganizującej się krytyczności

(*self-organized criticality, SOC*) w ekosystemie. Ewolucja w tym modelu powstaje w wyniku procesu nieprzerwanej mutacji najsłabszych gatunków oraz ich najbliższych sąsiadów. W celu uniknięcia zagłębiania się wyłącznie w optimum lokalnym, zwiększenia wydajności oraz uniknięcia mutacji tylko najsłabszych gatunków wprowadzono regulowany parametr τ, a sam algorytm nazwano τ-EO.

Wadą τ-EO jest konieczność definiowania własnej funkcji przystosowania (*fitness)* dla każdego problemu optymalizacji oraz fakt, że algorytm został użyty wyłącznie przy problemach kombinatorycznych, bez dostosowania go dla funkcji ciągłych. Rozwiązaniem tych problemów jest stworzona przez Sousa i Ramosa [14] [15] uogólniona metoda ekstremalnej optymalizacji (*generalized extremal optimization, GEO*), która może być stosowana dla szerokiej gamy problemów optymalizacyjnych bez ograniczeń co do funkcji przystosowania oraz zmiennych projektowych (*design variables*). GEO to stochastyczna metoda optymalizacji, która może być zastosowana do wypukłych lub rozłącznych problemów, w których występuje dowolna kombinacja zmiennych ciągłych i dyskretnych.

3.1 Model Baka-Snappena

Według teorii samoorganizującej się krytyczności (SOC) duże interaktywne systemy ewoluują w naturalny sposób do stanu krytycznego, gdzie pojedyncza zmiana w jednym z ich elementów powoduje lawinowe zdarzenia, które mogą dosięgnąć dowolną liczbę elementów w systemie. Model Baka-Snappena pokazuje, że SOC wyjaśnia właściwości takiego systemu jak naturalna ewolucja. Definiując system w którym gatunki leżą obok siebie oraz pierwszy jest połączony z ostatnim tworząc koło, stworzono prosty model ekosystemu. Dla każdego z gatunków losowana jest wartość przystosowania z jednostajnego rozkładu w przedziale [0, 1]. Gatunek z najniższą wartością, najsłabiej dopasowany, jest zmuszany do mutacji i przypisuje się mu nową, losową wartość. Zmiana najsłabszego ogniwa pociąga za sobą zmianę sąsiednich elementów, dla których także losowana jest nowa wartość, niezależnie od przystosowania danego elementu. W ten sposób po kilku iteracjach wartość przystosowania dla wszystkich gatunków przekracza próg krytyczny. Jednak ze względu na dynamikę systemu dochodzi do sytuacji, w których pewna liczba osobników spada poniżej progu krytycznego w wyniku wystąpienia lawinowych zmian. Wyjaśnieniem tego procesu jest fakt, że w każdej iteracji najgorszy gatunek zawsze podlega mutacji, a podczas ewolucji najsłabiej przystosowane gatunki

systematycznie, ze statystycznego punktu widzenia, ewoluują zwiększając wartość przystosowania.

3.2 Metoda ekstremalnej optymalizacji

W metodzie ekstremalnej optymalizacji każda zmienna reprezentuje pojedynczy gatunek, dla którego przypisana jest wartość funkcji przystosowania (*fitness*). W przypadku implementacji *τ-EO* według wartości funkcji przystosowania ustalany jest ranking dla *N* zmiennych, a następnie wybierany jest gatunek, który ulegnie mutacji z prawdopodobieństwem P_k:

$$P_k = k^{-\tau} \tag{3}$$

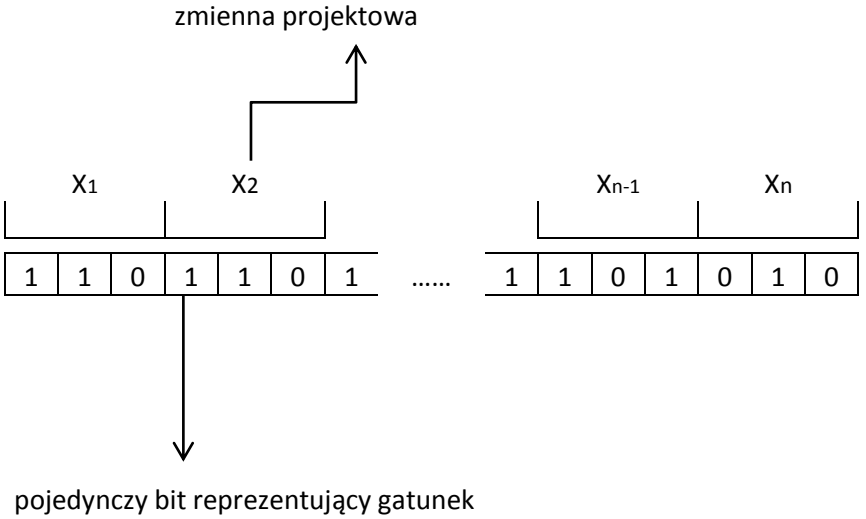
gdzie: k – ranking zmiennej, numer 1 - dla najsłabiej przystosowanego osobnika, numer N – dla najlepiej przystosowanego osobnika,
τ – konfigurowalny parametr.

Algorytm pozwala na mutację wszystkich zmiennych, jednak największe szanse na mutację ma gatunek o najsłabszej wartości przystosowania. Dzięki możliwości mutacji wszystkich zmiennych, algorytm pozwoli na opuszczenie minimum lokalnego. W przypadku, gdy wartość $\tau \rightarrow 0$ wybór osobnika staje się bardzo losowy i rosną szanse na ewolucję najlepszych osobników, z kolei przy $\tau \rightarrow \infty$ spadają szanse na ewolucję lepszych gatunków i poszukiwanie optimum skupi się na ewolucji najsłabszych osobników.

Podstawowy EO oraz τ-EO są z powodzeniem wykorzystywane przy poszukiwaniu rozwiązań trudnych problemów kombinatorycznych. Jednak ogólna definicja funkcji przystosowania nie jest optymalna, a wręcz w niektórych przypadkach nie jest możliwe jej zastosowanie [12] i konieczne jest tworzenie nowej funkcji przystosowania dla nowych problemów podlegających optymalizacji. Nie istnieje również implementacja dla zmiennych ciągłych. Aby zapewnić możliwość wykorzystanie EO z ogólną funkcją przystosowania dla wszystkich rodzajów zmiennych stworzono algorytm uogólnionej ekstremalnej optymalizacji (GEO) [14] [15].

3.3 Algorytm GEO

W przeciwieństwie do EO funkcja przystosowanie nie jest definiowana dla zaprojektowanej zmiennej, ale dla całej populacji, która koduje zmienne. Każdemu gatunkowi przypisuje się wartość przystosowania, a mutacji dokonuje się według ogólnych reguł. Populacja reprezentowana jest przez ciąg bitów, a pojedynczy gatunek przez 1 bit w ciągu, jak na Rys. 3.1. Każdy bit w ciągu ma przypisaną wartość przystosowania, na podstawie tej wartości określany jest ranking według którego wybierany jest bit ulegający mutacji.



Rys. 3.1 Ciąg bitów tworzący populację. Zmienna projektowa złożona z 3 bitów

Algorytm GEO składa się z następujących kroków [16]:

1. losowo inicjalizowany jest ciąg binarny o długości L, który koduje N zmiennych projektowych o długości równej L/N,
2. dla obecnej konfiguracji C ciągu bitów obliczana jest wartość funkcji celu V i inicjalizowane są zmienne najlepszej konfiguracji $C_{best} = C$ oraz najwyższej (najniższej w przypadku poszukiwania minimum) wartości funkcji celu $V_{best} = V$,
3. iterując po wszystkich bitach i=1..N w danej konfiguracji C:
    a. należy zanegować wartość bitu i, tzn. zamienić 0 na 1 i odwrotnie. Następnie należy obliczyć wartość funkcji celu V_i dla konfiguracji ciągu ze zmienionym bitem C_i,

b. wartość przystosowania bitu $F_i=V_i^3$,
c. przywracana jest wartość bitu *i*,

4. tworzony jest ranking dla *N* bitów wg wartości funkcji przystosowania F_i. Najgorzej przystosowany bity otrzymuje ranking k=1, a najlepiej k=*N*. Przy problemie minimalizacji wyższa wartość będzie miała wyższy ranking, i odwrotnie w przypadku maksymalizacji. Jeśli wartość jest identyczna dla dwóch bitów kolejność ustalana jest w sposób losowy,
5. wybierany jest bit *i*, który zostanie zmutowany. Wyboru dokonuje się poprzez:
   a. wylosowanie wartości *rand* z przedziału [0,1),
   b. następuje losowy wybór jednego z bitów *i*,
   c. sprawdzany jest ranking *k* wybranego bitu i następuje sprawdzenie czy bit zostanie poddany mutacji przez porównanie *rand* z wartością prawdopodobieństwa $P_i=k^{-\tau}$, gdzie τ jest konfigurowalnym parametrem,
   d. jeśli *rand* < P_i bit zostanie poddany mutacji, w przeciwnym wypadku należy przejść do kroku 5.b i ponownie wylosować bit,
6. ustawienie aktualnej konfiguracji $C=C_i$ oraz $V=V_i$,
7. jeśli $F_1 < F_{best}$[4] (w przypadku problemu maksymalizacji będzie to $F_1 > F_{best}$) ustawienie nowych wartości $F_{best}=F_1$ oraz $C_{best}=C_0$,
8. powtarzane są kroki 3-7 do osiągnięcia kryterium stopu, którym może być liczba wykonanych iteracji, liczba wykonanych funkcji celu lub otrzymanie zadanej minimalnej (lub maksymalnej) wartości funkcji celu,
9. wynikiem algorytmu jest najlepsza konfiguracja C_{best}, wartość funkcji celu V_{best} oraz przystosowania F_{best}.

Podczas wybierania bitu do mutacji może zostać wybrany każdy z nich. Na początku losowany jest jeden z *N* bitów, a następnie sprawdzana jest wartość prawdopodobieństwa mutacji P_i tego bitu. Wartość prawdopodobieństwa jest zależna od pozycji w rankingu na

[3] W literaturze wartość funkcji definiowana jest także jako $F_i=V_i - R$, gdzie *R* jest dowolną stałą lub $F_i=V_i - V_{best}$, z tego względu, że sprawdzane jest dopasowanie bitu, a nie wartość funkcji celu. Jednak ze względu na fakt, że podczas jednej iteracji wartość V_{best} jest stała można traktować ją jako *R*. Z kolei odjęcie stałej *R* od wartości funkcji celu V_i nie zmieni kolejności bitów wg funkcji przystosowania, dlatego w poniższej pracy przyjęto $R=0$

[4] W niektórych wersjach algorytmu porównywana jest wartość funkcji przystosowania wybranego bitu F_i do najlepszej aktualnej wartości F_{best}. Jednak taki warunek wiąże się z pominięciem sprawdzenia najlepszej konfiguracji, gdy nie zostanie wybrany bit o rankingu 0. Z kolei porównanie do F_0 nie wiąże się z koniecznością wykonania żadnych dodatkowych obliczeń. Z tych powodów nieporównywanie najlepszej populacji do wybranej wydaje się błędnym założeniem.

którym znajduje się wylosowany bit oraz od parametru τ. W przypadku, gdy wartość parametru wynosi 5.0 szansa, że po wylosowaniu bit na pozycji drugiej zostanie zmutowany to 3%, a trzeci – 0.4%. Z kolei przy τ=0.1 szansa, że bit o pozycji 100 zostanie zmutowany wynosi ponad 63%. Bit z pozycji 1 zawsze podlega mutacji, gdy zostanie wylosowany, gdyż P_1=1. Możliwość wyboru osobników których mutacja nie polepszy wartości funkcji celu, pozwala na wyjście poza minima lokalne. Z drugiej strony przy zbyt niskiej wartości parametru τ algorytm zaczyna zachowywać się w sposób bardzo losowy.

3.4 Zdefiniowanie ilości potrzebnych bitów

Pierwszym krokiem przy opracowaniu algorytmu GEO dla problemu optymalizacji jest konieczność zdefiniowania ilości bitów kodujących zmienną projektową. Ilość bitów powinna pozwolić na zakodowanie wszystkich możliwych stanów osobnika. W przypadku zmiennych ciągłych konieczne jest określenie minimalnej precyzji. Dla takich zmiennych minimalną liczbę bitów m do osiągnięcia pożądanej precyzji można wyliczyć według wzoru:

$$2^m \geq \left\lceil \frac{(x_j^u - x_j^l)}{p} + 1 \right\rceil \tag{4}$$

gdzie: x_j^u – górna granica wartości zmiennej j,

x_j^l – dolna granica wartości zmiennej j,

x_j – zmienna projektowa $j \in \{1, 2, .. N\}$,

p – pożądana precyzja.

Aby odkodować wartość zmiennej projektowej zapisanej przez m bitów należy skorzystać z równania:

$$x_j = x_j^l + \left(x_j^u - x_j^l\right) \cdot \frac{I_j}{(2^{l_j} - 1)} \tag{5}$$

gdzie: I_j – to wartość zmiennej j po przekształceniu z postaci dwójkowej do dziesiętnej.

W przypadku zmiennych dyskretnych i całkowitych przyjmuje się, że precyzja $p = 1$. Gdy ilość możliwych stanów wynosi 2^n dla dowolnej naturalnej wartości n, zakodowanie i odkodowanie wartości zmiennej projektowej jest bardzo proste i służy do tego poniższy wzór:

$$x_j = x_j^l + I_j \tag{6}$$

Do wartości zmienne I_j w postaci dziesiętnej należy dodać minimalną oczekiwaną wartość. Powyższy warunek na liczbę stanów można sformułować w postaci:

$$x_j^u - x_j^l = 2^N - 1 \tag{7}$$

Gdy warunek przedstawiony w równaniu (7) nie jest spełniony należy wykorzystać minimalną liczbę bitów m, która spełnia nierówność:

$$2^m > \left(x_j^u - x_j^l\right) + 1 \tag{8}$$

Dla $2^m - N$ bitów w ciągu przypisuje się liczby całkowite z przedziału dopuszczalnych rozwiązań lub spoza możliwych wartości. Może to doprowadzić do sytuacji, w której jedna lub więcej zmiennych będą skojarzone z więcej niż jednym ciągiem bitów. Takie rozwiązanie pozwala uniknąć konieczności nakładania dodatkowych ograniczeń, jednak zwiększa szansę mutacji niektórych bitów przez co zaburza działanie algorytmu GEO. Zmienne dyskretne wymagają ostatecznie dodatkowego mapowania na wartości całkowite.

4. Zastosowanie algorytmu GEO w problemie szeregowania

Algorytm GEO wydaje się doskonałym narzędziem, dzięki zastosowaniu którego możliwe jest szybkie znalezienie najlepszej lub zbliżonej do optymalnej ścieżki szeregowania z minimalnym czasem szeregowania. Jest to metoda heurystyczna, dzięki czemu wymaga przeszukania tylko części rozwiązań wśród ogromnego ich zbioru. Wielkość tego zbioru zależy od ilości procesorów na które można przydzielać zadania oraz od ilości zadań podlegających szeregowaniu. Ilość możliwych kombinacji rozwiązań, co w przypadku próby sprawdzenia wszystkich możliwych rozwiązań wymagałoby wykonania takiej samej ilości funkcji celu, wynosi:

$$\bar{V}_p^t = p^t \tag{9}$$

gdzie: p – liczba procesorów,
t – liczba szeregowanych modułów.

Liczbę wykonań funkcji celu w przypadku implementacji algorytmu GEO dla tego problemu można opisać według wzoru:

$$\lceil \log_2 p \rceil \cdot t \cdot n_{iteracji} \tag{10}$$

gdzie: $n_{iteracji}$ – maksymalna liczba iteracji skonfigurowana przy uruchomieniu algorytmu.

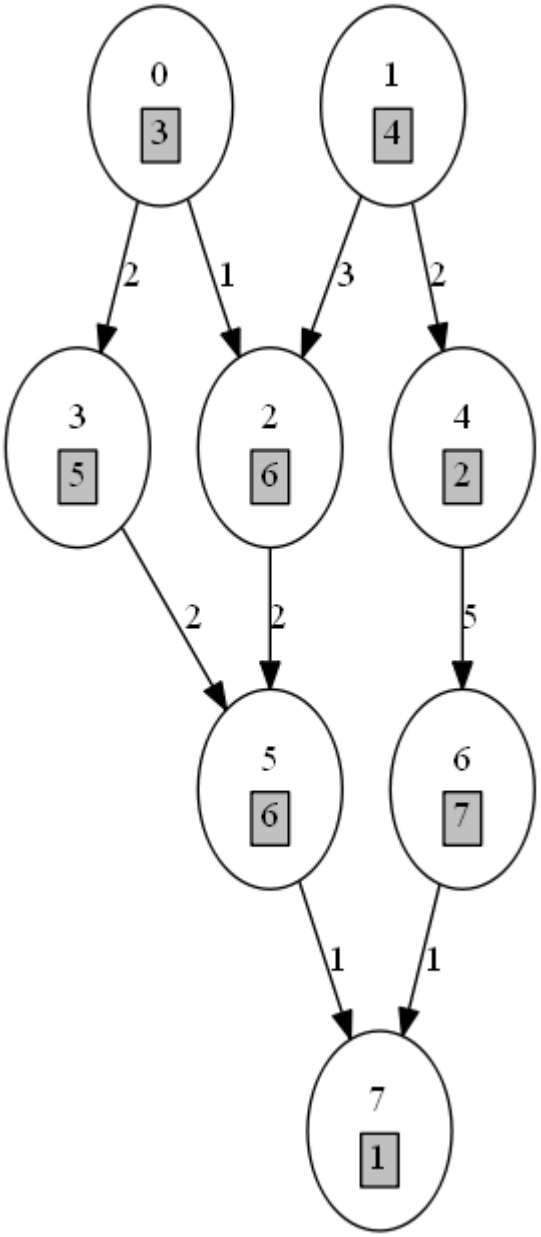
Implementując algorytm GEO dla problemu szeregowania [17] należy wykonać poniższe kroki:

1. zainicjowanie maksymalnej liczby iteracji $n_{iteracji}$ algorytmu oraz zmiennej τ,
2. określenie długości ciągu bitów N, które kodują konfigurację uruchomionych zadań. Konfiguracja to określenie na którym procesorze zostaną wykonane kolejne zadania (kolejność wykonania jest determinowana na podstawie polityki szeregowania, patrz rozdział 2.3),

3. zainicjowanie początkowego ciągu C, np. poprzez wylosowanie wartości dla każdego z bitów,
4. wyznaczenie wartości funkcji celu (czasu szeregowania) V dla początkowego ciągu z zastosowaniem określonej polityki szeregowania, ustawienie wartości V jako V_{best} oraz konfiguracji C jako C_{best} ,
5. zmienianie kolejno bitów i=1..N w konfiguracji C poprzez jego negację i zapamiętanie konfiguracji C_i
    a. wyznaczenie czasu szeregowania V_i dla nowej konfiguracji,
    b. przywrócenie wartości bitu i,
6. posortowanie utworzonych konfiguracji na podstawie wartości funkcji celu V_i, ranking o numerze 1 przypada konfiguracji z najlepszym czasem szeregowania, najgorsza konfiguracja znajdzie się na pozycji N,
7. wybór konfiguracji, która zastąpi obecną C
    a. wylosowanie progu mutacji c_p z rozkładu równomiernego w przedziale [0, 1],
    b. wylosowanie liczby k z przedziału 1..N,
    c. jeśli wartość prawdopodobieństwa dla k liczona według wzoru (3) przekracza próg c_p, to konfiguracja $C = C_k$, następuje przejście do kroku 8, w przeciwnym wypadku następuje ponowne losowanie w kroku 7a,
8. gdy $V_1 < V_{best}$, $V_{best} = V_1$, a $C_{best} = C_1$,
9. jeśli osiągnięto maksymalną liczbę iteracji (kryterium stopu) przerwanie algorytmu i zwrócenie V_{best} oraz C_{best}. W przeciwnym wypadku wykonanie kolejnej iteracji poprzez powrót do kroku 5.

4.1 Zdefiniowanie problemu

Załóżmy, że należy znaleźć najlepszy czas szeregowania i kolejność wykonywania zadań na poszczególnych procesorach dla grafu programu przedstawionego na Rys. 4.1. Przyjmijmy, że szeregowanie nastąpi na 4 procesorach w konfiguracji FULL4 (Rys. 5.2).



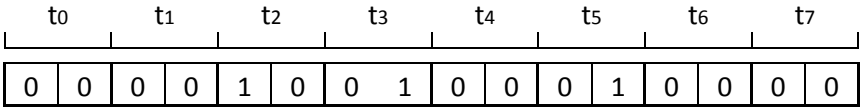
Rys. 4.1 Graf procesów dla przykładowej implementacji algorytmu

4.2 Sposób reprezentacji populacji

Rozdział 3.4 opisuje w jaki sposób wyznaczyć ilość potrzebnych bitów na zdefiniowanie jednego osobnika i całej populacji. W przypadku problemu szeregowania osobnikiem jest numer procesora dla określonego zadania, populację stanowi konfiguracja która definiuje procesory dla wszystkich kolejnych modułów. W przypadku 2 procesorów z równania (4) wynika, że potrzebny jest pojedynczy bit. W przypadku liczby procesorów równej 3 lub 4 wymagane są 2 bity. Gdy ilość procesorów znajdzie się w przedziale [5;8]

potrzebne są już 3 bity. Wraz ze wzrostem ilości potrzebnych bitów proporcjonalnie wzrasta liczba wykonań funkcji celu w każdej iteracji.

W rozważanym przypadku potrzebne są 2 bity na zakodowanie zmiennej projektowej, numeru procesora na którym zostanie wykonane zadanie. Cały ciąg będzie składał się z 16 bitów. Przedstawiono to na Rys. 4.2.



Rys. 4.2 Przykładowa konfiguracja ciągu bitów

4.3 Inicjowanie początkowej konfiguracji

W przypadku algorytmów heurystycznych i metaheurystycznych często bardzo ważny jest punkt startowy. Gdy algorytm rozpoczyna wykonywania w miejscu leżącym blisko optimum globalnego, może bardzo szybko znaleźć optymalne rozwiązanie. Jednak gdy wartość początkowa znajduje się w obrębie minimum lokalnego, czas potrzebny na znalezienie najlepszego rozwiązania może znacznie się wydłużyć. Istnieje wiele sposobów inicjalizacji wartości początkowej, są one zależne od rozważanego problemu i stosowanego algorytmu. GEO w przypadku problemu szeregowania pozwala na dokonanie inicjalizacji na poniższe sposoby:

- wszystkie zadania będą uruchamiane na 1 procesorze, wszystkie bity w ciągu będą miały wartość 0,
- ciąg zostanie zainicjowany przez losowe wartości, zadaniu przypisywany jest dowolny procesor,
- inicjalizacja poprzez wykonanie innego algorytmu lub jako konfiguracja wprowadzona przez użytkownika algorytmu, który pozwoli na starcie zbliżyć się do minimum.

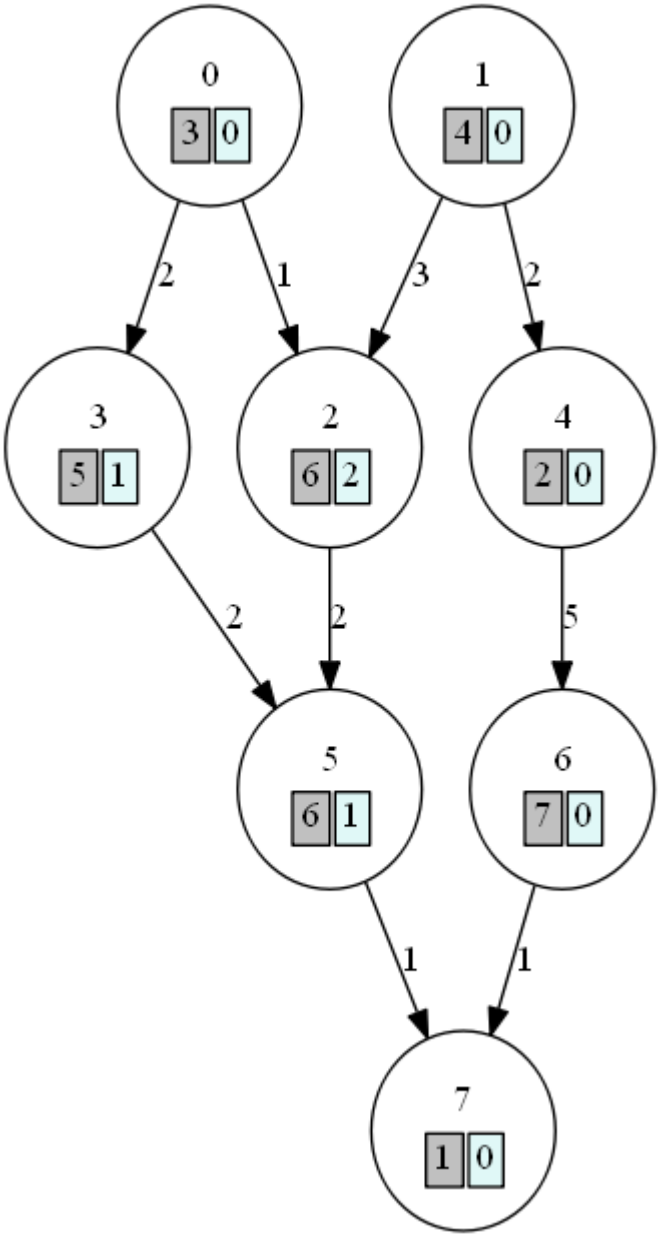
W poniżej pracy inicjalizacja odbywa się w sposób losowy. Przypadek inicjowania zadań na jednym procesorze może być traktowany jako przypadek losowy i nie przybliża do szybszego znalezienia rozwiązania w przypadku uruchamiania algorytmu na różnych grafach programów. Uruchomienie innego algorytmu do określenia punktu początkowego jest czasochłonne i w przypadku problemu szeregowania nie ma praktycznego

uzasadnienia dla takiego sposobu inicjalizacji. Wprowadzenie przez użytkownika wartości początkowej mogą wydawać się dobrym pomysłem, zbliżającym do optymalnego rozwiązania, aczkolwiek nie w każdym przypadku wykorzystania algorytmu może pojawić się na to czas. Człowiek jest również istotą omylną, a mózg człowieka nie jest w stanie w sposób szybki przeanalizować graf program składający się z ponad kilkunastu zadań.

Tabela 4.1 przestawia przykładową konfigurację startową dla rozważanego grafu programu. Rys. 4.3 pokazuje konfigurację startową na grafie, w komórce na niebieskim tle znajduje się numer procesora na którym zostanie uruchomione zdanie. Na szarym tle znajduje się koszt wykonania zadania.

Tabela 4.1 Konfiguracja startowa

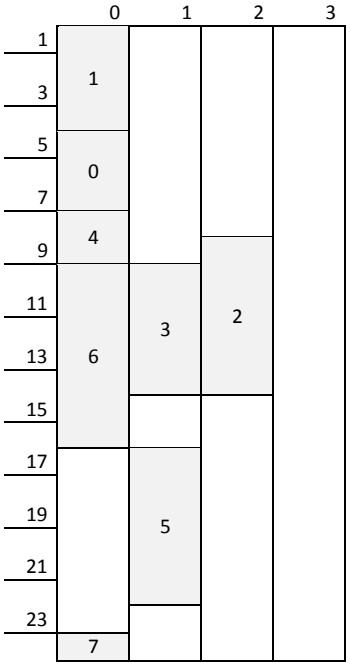
zadanie	0		1		2		3		4		5		6		7	
bity	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0	0



Rys. 4.3 Zainicjowana konfiguracja w postaci grafu

4.4 Wyznaczenie czasu szeregowania dla określonej konfiguracji

W poniższej pracy wykorzystano politykę szeregowania opisaną w rozdziale 2.3 opartą na wysokości dynamicznej. Według tej polityki uszeregowanie początkowe wygląda jak na Rys. 4.4. Czas szeregowanie *T* tej konfiguracji wynosi 24.



Rys. 4.4 Diagram Gantta dla konfiguracji początkowej

Dwa pierwsze zadania, które mogą być uruchomione na starcie jednocześnie na procesorze 0. Tabela 4.2 przedstawia możliwe ścieżki rozpatrywane przy tym konkretnym dylemacie. Gdy kilka zadań może rozpocząć wykonywanie na tym samym procesorze jednocześnie zostanie wybrane te z najdłuższą ścieżką, w tym przypadku będzie to moduł 1.

Tabela 4.2 Wybór zadania gotowego do wykonania

zadanie początkowe	ścieżka	wzór T	T
0	0 -> 2 -> 5 -> 7	3 + 1 * 1 + 6 + 2 * 1 + 6 + 1 * 1 + 1	20
0	0 -> 3 -> 5 -> 7	3 + 2 * 1 + 5 + 2 * 0 + 6 + 1 * 1 + 1	18
1	1 -> 2 -> 5 -> 7	4 + 3 * 1 + 6 + 2 * 1 + 6 + 1 * 1 + 1	23
1	1 -> 4 -> 6 -> 7	4 + 2 * 0 + 2 + 5 * 0 + 7 + 1 * 0 + 1	14

4.5 Pierwsza iteracja – mutacja konfiguracji

Rozpoczynając iterację należy wygenerować osobniki zmutowane poprzez zmianę pojedynczego bitu. Dla konfiguracji przedstawionej w Tabela 4.1 w wyniku mutacji powstaną osobniki przedstawione w Tabela 4.3.

Tabela 4.3 Przykład mutacji w iteracji

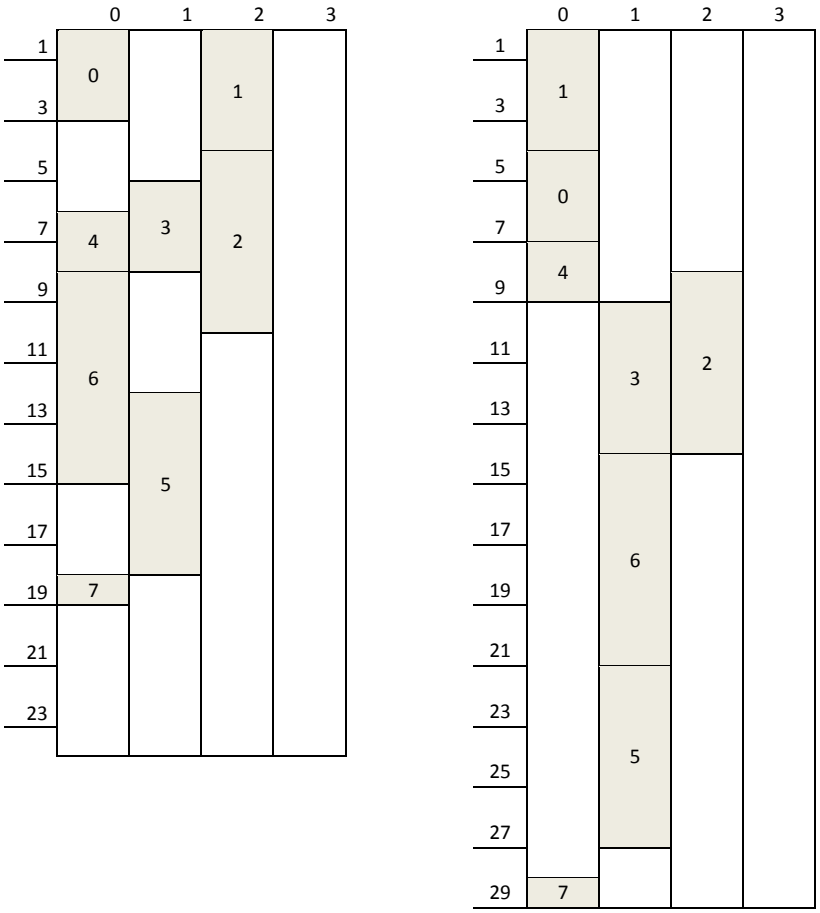
mutujący bit	zadania																T
	0		1		2		3		4		5		6		7		
1	1	0	0	0	1	0	0	1	0	0	0	1	0	0	0	0	23
2	0	1	0	0	1	0	0	1	0	0	0	1	0	0	0	0	23
3	0	0	1	0	1	0	0	1	0	0	0	1	0	0	0	0	20
4	0	0	0	1	1	0	0	1	0	0	0	1	0	0	0	0	23
5	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	23
6	0	0	0	0	1	1	0	1	0	0	0	1	0	0	0	0	24
7	0	0	0	0	1	0	1	1	0	0	0	1	0	0	0	0	24
8	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	24
9	0	0	0	0	1	0	0	1	1	0	0	1	0	0	0	0	24
10	0	0	0	0	1	0	0	1	0	1	0	1	0	0	0	0	24
11	0	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	24
12	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	24
13	0	0	0	0	1	0	0	1	0	0	0	1	1	0	0	0	23
14	0	0	0	0	1	0	0	1	0	0	0	1	0	1	0	0	29
15	0	0	0	0	1	0	0	1	0	0	0	1	0	0	1	0	24
16	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0	1	23

W następnym kroku konieczne jest ustalenie rankingu mutowanych bitów. Należy posortować konfiguracje według kolumny *T*, której wartość jest równa wartości funkcji przystosowania F_i, tak jak w Tabela 4.4. Jak można zauważyć najlepszy wynik otrzymamy przy mutacji bitu numer 3 (podział przedstawiono na Rys. 4.6). Warto zwrócić uwagę na rozbieżność czasów szeregowania przy zmianie przypisania procesora tylko do jednego zadania. Wartości *T* wahają się w przedziale [20;29], przy bazowej konfiguracji z czasem *T*=24.

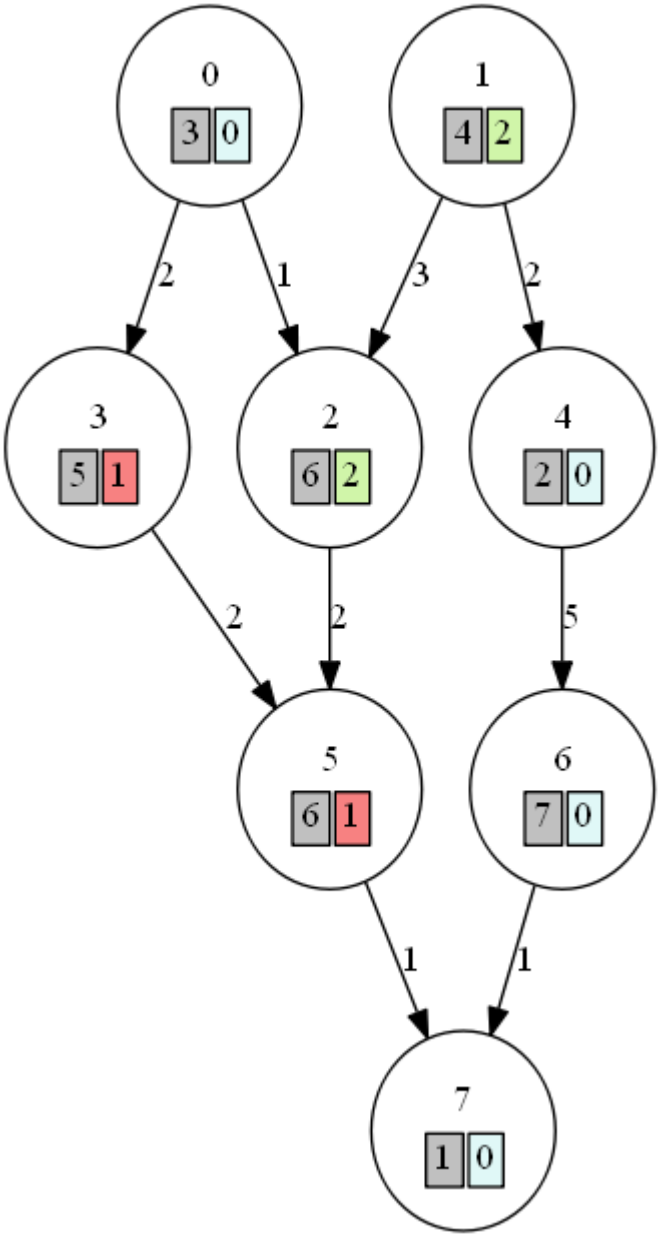
Tabela 4.4 Konfiguracje ułożone w rankingu

ranking	mutujący bit	zadania																T
		0		1		2		3		4		5		6		7		
1	3	0	0	1	0	1	0	0	1	0	0	0	1	0	0	0	0	20
2	1	1	0	0	0	1	0	0	1	0	0	0	1	0	0	0	0	23
3	2	0	1	0	0	1	0	0	1	0	0	0	1	0	0	0	0	23
4	4	0	0	0	1	1	0	0	1	0	0	0	1	0	0	0	0	23
5	5	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	23
6	13	0	0	0	0	1	0	0	1	0	0	0	1	1	0	0	0	23
7	16	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0	1	23
8	6	0	0	0	0	1	1	0	1	0	0	0	1	0	0	0	0	24
9	7	0	0	0	0	1	0	1	1	0	0	0	1	0	0	0	0	24
10	8	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	24
11	9	0	0	0	0	1	0	0	1	1	0	0	1	0	0	0	0	24
12	10	0	0	0	0	1	0	0	1	0	1	0	1	0	0	0	0	24
13	11	0	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	24
14	12	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	24
15	15	0	0	0	0	1	0	0	1	0	0	0	1	0	0	1	0	24
16	14	0	0	0	0	1	0	0	1	0	0	0	1	0	1	0	0	29

Na Rys. 4.5 przedstawiono porównanie diagramów Gantta dwóch skrajnie różnych konfiguracji po mutacji genu. Różnią się one dwoma zadaniami położonymi na innych procesorach.



Rys. 4.5 Porównanie diagramów Gantta dwóch skrajnych konfiguracji



Rys. 4.6 Diagram programu z uwzględnieniem podziału według najlepszej mutacji w iteracji

4.6 Wybór mutowanego bitu

Gdy ranking został ustalony, należy wybrać bit który zostanie poddany mutacji. W przykładzie ustalono wartość parametru τ=0.8, dla której nastąpiła próba wyboru bitu. W pierwszej kolejności wylosowano wartość graniczną dla mutacji równą 0.416. Następnie losowano bity, aż do momentu gdy wartość $k^{-\tau}$ nie przekroczyła progu. Wszystkie próby przedstawia Tabela 4.5.

Tabela 4.5 Próby wyboru mutowanego bitu

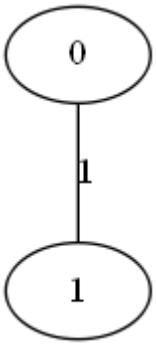
Ranking k	$k^{-\tau}$
3	0,415244
15	0,114585
5	0,275946
8	0,189465
10	0,158489
13	0,128483
2	0,574349

Próg został przekroczony dopiero dla rankingu o numerze 2, wybrano konfigurację w której mutowany jest pierwszy bit. Nie jest to najlepsza konfiguracja. Warto zwrócić uwagę, że gdyby τ byłoby równe 0.5 wybrano by bit znajdujący się na 3 miejscu w rankingu, z kolei przy τ = 5.0 mógłby zostać wybrany do mutacji wyłącznie bit z pozycji pierwszej.

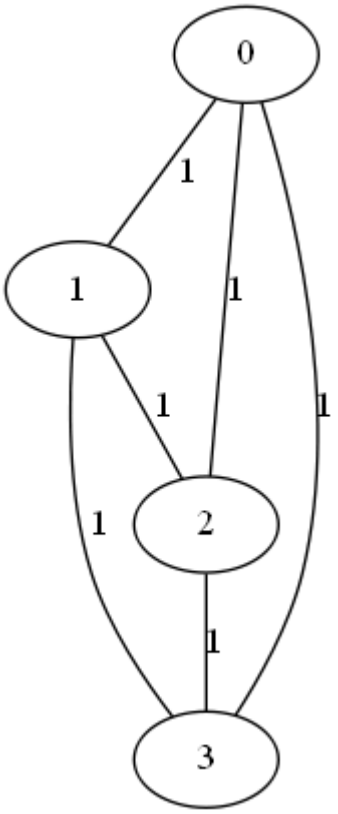
5. Wyniki eksperymentów

5.1 Konfiguracje procesorów

Eksperymenty zostały przeprowadzone na trzech konfiguracjach procesorów. Każdą z nich można przedstawić za pomocą grafu pełnego, gdzie każdy węzeł łączy się z pozostałymi. Waga każdej z krawędzi ma wartość 1, co oznacza na podstawie wzoru (1), że dane z jednego procesora można przenieść na inny z kosztem równym a_{kl}. Taka konfiguracja powoduje, że gdy na procesorze p_0 umieścimy zadania t_0 i t_2, a na p_1 zadania t_1, t_3 i t_4 to czas wykonania oraz startu poszczególnych zadań będzie taki sam jakbyśmy przenieśli wszystkie moduły z procesora p_0 na p_1 i odwrotnie. Przykładowe konfiguracje przedstawiono na Rys. 5.1 i Rys. 5.2.



Rys. 5.1 Graf procesorów FULL2



Rys. 5.2 Graf procesorów FULL4

5.2 Rozkład prawdopodobieństwa mutacji w przypadkach testowych

Przeprowadzając testy uruchomiono 1000 razy algorytm GEO do szeregowania na różnych grafach w następujących konfiguracjach parametru prawdopodobieństwa τ: 0.1, 0.2, 0.5, 0.8, 1.0, 1.5, 2.0, 5.0 i 8.0.

Tabela 5.1 Prawdopodobieństwo wyboru konfiguracji do dalszej analizy w zależności od parametru prawdopodobieństwa

	0,1	0,2	0,5	0,8	1	1,5	2	5	8
1	1,0000	1,0000	1,0000	1,0000	1,0000	1,0000	1,0000	1,0000	1,0000
2	0,9330	0,8706	0,7071	0,5743	0,5000	0,3536	0,2500	0,0313	0,0039
3	0,8960	0,8027	0,5774	0,4152	0,3333	0,1925	0,1111	0,0041	0,0002
4	0,8706	0,7579	0,5000	0,3299	0,2500	0,1250	0,0625	0,0010	0,0000
5	0,8513	0,7248	0,4472	0,2759	0,2000	0,0894	0,0400	0,0003	0,0000
6	0,8360	0,6988	0,4082	0,2385	0,1667	0,0680	0,0278	0,0001	0,0000
7	0,8232	0,6776	0,3780	0,2108	0,1429	0,0540	0,0204	0,0001	0,0000
8	0,8123	0,6598	0,3536	0,1895	0,1250	0,0442	0,0156	0,0000	0,0000
9	0,8027	0,6444	0,3333	0,1724	0,1111	0,0370	0,0123	0,0000	0,0000
10	0,7943	0,6310	0,3162	0,1585	0,1000	0,0316	0,0100	0,0000	0,0000
11	0,7868	0,6190	0,3015	0,1469	0,0909	0,0274	0,0083	0,0000	0,0000
12	0,7800	0,6084	0,2887	0,1370	0,0833	0,0241	0,0069	0,0000	0,0000
13	0,7738	0,5987	0,2774	0,1285	0,0769	0,0213	0,0059	0,0000	0,0000
14	0,7680	0,5899	0,2673	0,1211	0,0714	0,0191	0,0051	0,0000	0,0000
15	0,7628	0,5818	0,2582	0,1146	0,0667	0,0172	0,0044	0,0000	0,0000

W Tabela 5.1 przedstawiono prawdopodobieństwo z jakim może zostać wybrana konfiguracja podziału zadań na procesorach do następnej iteracji jako wartość bazowa. Oznacza to również szansę, że wybrana konfiguracja zostanie wyselekcjonowana do porównania z aktualnie najlepszym ustawieniem i może zastąpić je w przypadku, gdy czas szeregowania będzie lepszy.

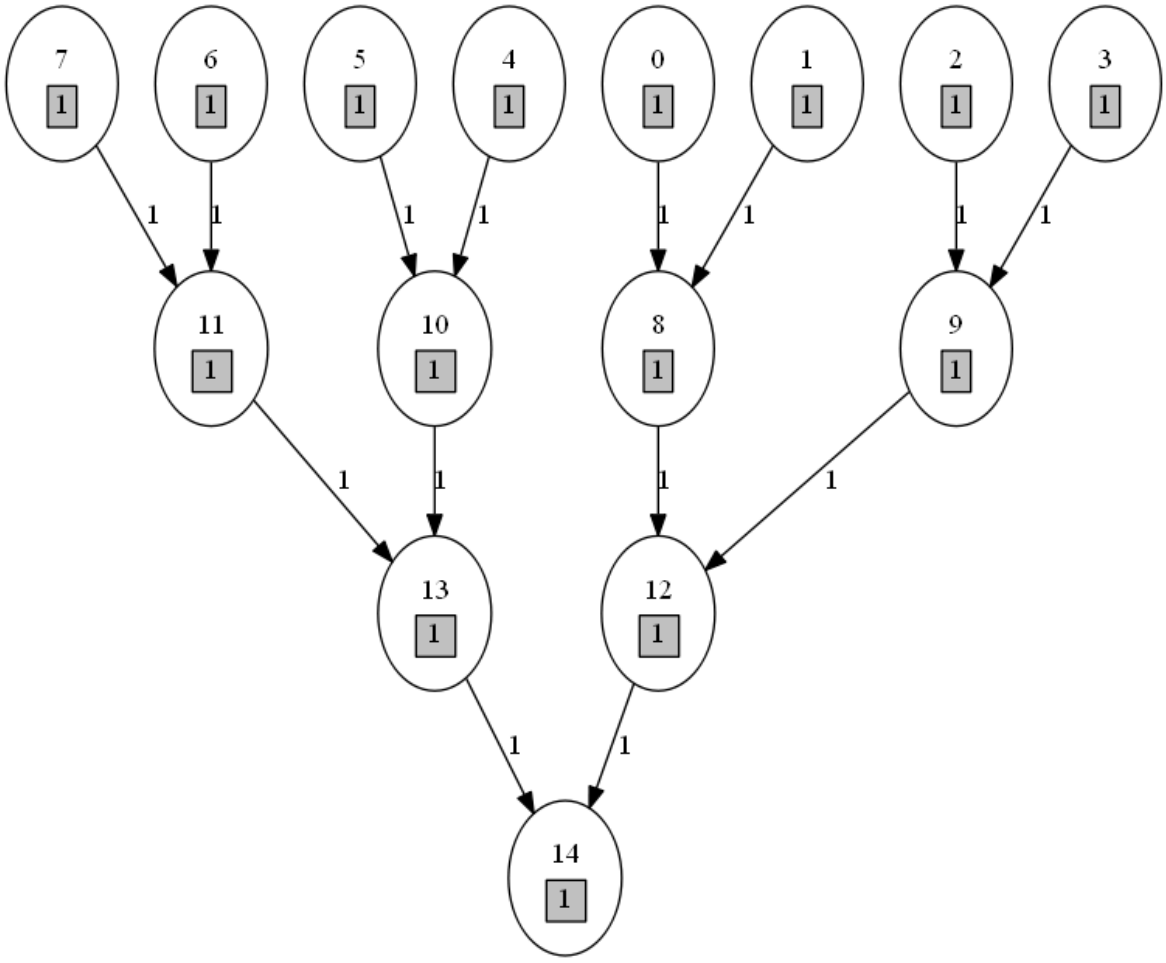
Istotne jest że szansa wybrania danej konfiguracji pojawia się dopiero w sytuacji gdy wcześniej podana konfiguracja zostanie wylosowana. Rozkład prawdopodobieństwa jest w tym wypadku stały i wynosi *1/k*, gdzie *k* jest ilością konfiguracji. Oznacza to, że przykładowo gdy wartość parametru τ=1, a ilość konfiguracji wynosi 15, to szansa że do następnej konfiguracji przejdzie konfiguracja numer 1 wynosi 1/15 * 1, z kolei szansa wyboru konfiguracji numer 10 wynosi 1/15 * 1/10 = 1/150. Istnieje także niezerowa szansa, że w losowaniu nie zostanie wybrana konfiguracji i będzie należało ponowić wybór. Taka sytuacja będzie miała miejsce w znacznej większości przypadków gdy τ=8.

Bardzo ważną zależnością widoczną w tabeli jest fakt zmniejszania szansy na wybór gorszych konfiguracji (im wyższy numer tym gorsze ustawienie) wraz ze wzrostem wartości parametru τ. W przypadku τ=8 szansa, że zostanie wybrane ustawienie inne od najlepszego wynosi < 0.1% i w przypadku wykonania 100 iteracji prawdopodobnie nie zostanie wybrana inna konfiguracja niż najlepsza do dalszego przetwarzania. Można to

zaobserwować na Wykres 5.1. Z kolei na Wykres 5.2 dla τ=0.1 widoczna jest skokowość przebiegu algorytmu i częsty wybór gorszych konfiguracji.

5.3 Proste grafy drzewiaste

Grafy procesów tree15 (Rys. 5.4) oraz intree15 (Rys. 5.3) są jednymi z najprostszych do uszeregowania ze względu na swoją budowę, brak złożonych zależności, ten sam czas szeregowania wszystkich procesów i równy koszt zmiany procesora. Oba grafy mają najlepszy czas szeregowania T = 9 w przypadku użycia dwóch procesorów w konfiguracji FULL2. Wykonując testy na obu grafach wykonano 1000 uruchomień algorytmu dla każdego z grafów. Limit iteracji ustawiono na 100, co oznaczało że liczba ewolucji wykonania funkcji przystosowania (*fitness*) wyniosła 1500 (100 iteracji * 15 mutujących konfiguracji na iterację).



Rys. 5.3 Graf procesów intree15

W Tabela 5.2 przestawiono dane statystyczne po wykonaniu eksperymentów na grafie intree15. Dane pokazują zależność pomiędzy ilością iteracji potrzebną do

znalezienia optymalnego rozwiązania a parametrem prawdopodobieństwa τ. Jak pokazuje kolumna średnia *T* i min *T*, które oznaczają odpowiednio średni i najlepszy czas szeregowania, w obu przypadkach wartość wynosi 9 i jest to najlepszy czas uszeregowania dla tego grafu. Spoglądając na średnią i wariancję liczby iteracji potrzebnych do znalezienia optymalnego rozwiązania, można zauważyć że najlepsze wyniki otrzymano gdy wartość τ wynosiła 5. Przy tej wartości parametru średnia wyniosła 1.474 i jest znacznie lepsza od średniej dla wartości parametru równej 0.1 i wynoszącej 4.322. W tym przypadku także wariancja potwierdza, że różnica między potrzebnymi iteracjami jest nieznaczna. Wartość mediany, kwartylu górnego oraz maksymalnej liczby iteracji potwierdzają te obserwacje. Porównując wartości τ równego 0.1 i 5.0 mamy odpowiednio:

- medianę równą 3 i 1, co oznacza, że w 50% uruchomień potrzebne były co najmniej 3 iteracje w przypadku τ=0.1 i tylko 1 w przypadku τ=5.0,
- kwartyle górne równe 6 i 2, jest to duża różnica jeśli chodzi o ilość potrzebnych iteracji,
- maksymalną liczbę iteracji równą 42 i 6.

W przypadku dominanty sytuacja prawie we wszystkich przypadkach wygląda podobnie. Najczęściej potrzeba tylko jednej iteracji do znalezienia optymalnego rozwiązania. Odstępstwem jest uruchomienie algorytmu z parametrem τ=0.1. Tutaj większe prawdopodobieństwo ma znalezienie optimum przy losowaniu bazowej konfiguracji (wynosi 21.1%), niż po pierwszej iteracji (wynosi ono 17%). Dla τ=5.0 prawdopodobieństwo znalezienie najlepszego rozwiązania w 1 iteracji wynosi 38.1% a łącznie z szansą wylosowania w zerowej iteracji osiąga 56.5%, dla τ=8.0 wartości te to odpowiednio 40.5% oraz 58.2%. Mimo dużej rozbieżności w wynikach dla wszystkich konfiguracji udało się osiągnąć najlepszy rezultat w 100 iteracjach.

Analizując powyższe dane nasuwa się wniosek, że najlepszą wartością parametru τ dla grafu intree15 jest 5.0 i 8.0. Powodem jest to, że graf intree15:

- posiada proste zależności procesów,
- koszt zmiany procesora jest niewielki i wynosi 1.

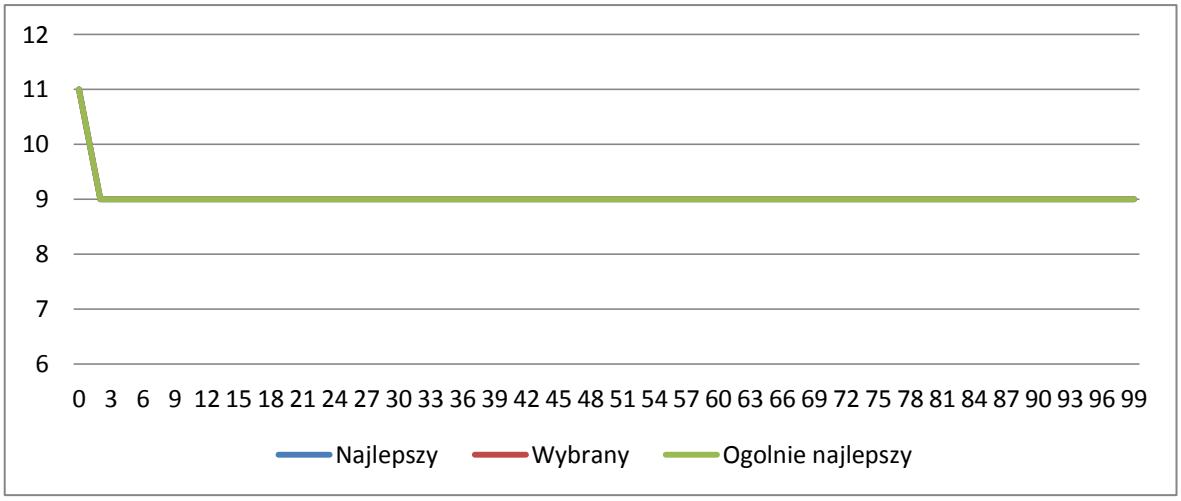
Te czynniki powodują, że istnieje znaczny procent rozwiązań, które dają najlepszy czas szeregowania. Przejście z gorszej do najlepszej konfiguracji wymaga wykonania tylko kilku kroków co oznacza konieczność zmieniania tylko 2-3 zadań na procesorze. Nasuwa się stwierdzenie, że w przypadku algorytmu GEO nie istnieje minimum lokalne wśród zbioru konfiguracji. To sprawia, że łatwiej jest znaleźć najlepsze rozwiązanie zawsze

wybierając minimum w aktualnym zbiorze rozwiązań (τ=8.0), niż przeszukiwać wybierając losowo kolejny zbiór (τ=0.1). Druga ewentualność w przypadku tego grafu prowadzi do zbytniego skupienia się na ominięciu minimum lokalnego, które nie istnieje i próbie przeszukania rozwiązań wśród zbyt wielu konfiguracji.

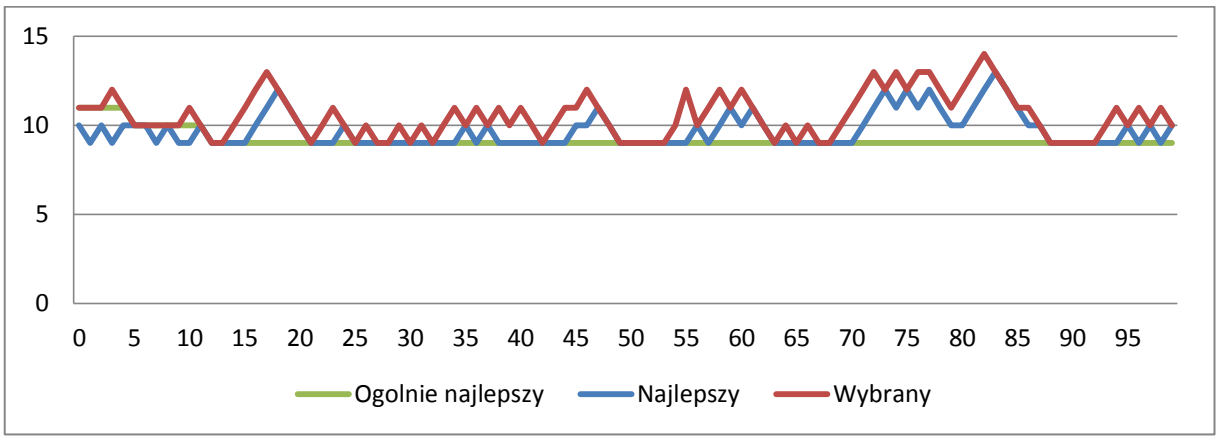
Tabela 5.2 Dane statystyczne testów dla grafu intree15 z 2 procesorami

τ	średnia iteracji	wariancja iteracji	min	Q1	mediana	Q3	max	moda	moda (%)	najlepsze wyniki (%)	śr. T	min T
0.1	4,322	27,097	0	1	3	6	42	0	21,1	100	9	9
0.2	3,502	14,579	0	1	2	5	32	1	19,4	100	9	9
0.5	2,03	3,669	0	1	2	3	11	1	26,3	100	9	9
0.8	1,749	2,498	0	1	1	2	10	1	30,3	100	9	9
1.0	1,723	2,056	0	1	1	2	12	1	33,4	100	9	9
1.5	1,519	1,381	0	1	1	2	9	1	37	100	9	9
2.0	1,497	1,279	0	1	1	2	6	1	33,3	100	9	9
5.0	1,474	1,183	0	1	1	2	6	1	38,1	100	9	9
8.0	1,483	1,247	0	1	1	2	6	1	40,5	100	9	9

Na Wykres 5.1 oraz Wykres 5.2 przedstawiono przebiegi algorytmu w zależności od parametru prawdopodobieństwa. Na pierwszym wykresie wszystkie linie pokrywają się, zawsze wybierany jest do mutacji bit o rankingu 1. W przypadku gdy jego wartość jest mała (np. 0.1 na Wykres 5.2), można zauważyć że nie zawsze wybierana jest do następnej iteracji najlepsza konfiguracja procesów. Jest to różnica między linią niebieską, obrazującą najlepszy czas szeregowania w danej iteracji, a czerwoną – wartość czasu szeregowania dla wybranej (wylosowanej) konfiguracji. Druga zauważalna różnica, to odejście od najlepszej konfiguracji do innej, skrajnie najgorszej. Widoczne jest to przy odchyleniach linii czerwonej od zielonej. Taki przebieg pozwala na wyjście z minimum lokalnego, co może prowadzić do znalezienia najlepszej konfiguracji w następnych iteracjach.



Wykres 5.1 Przebieg algorytmu dla grafu intree15 i τ=8.0, linie pokrywają się

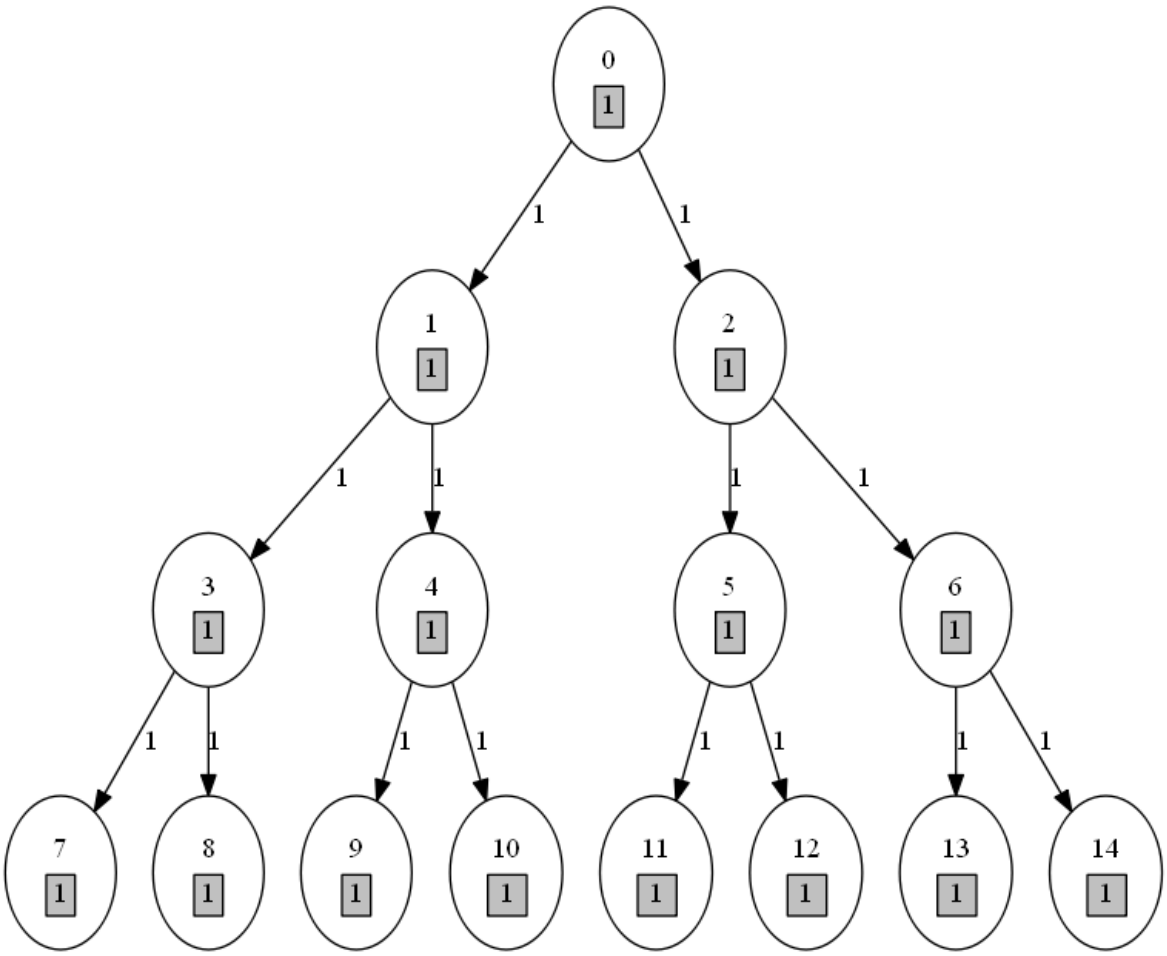


Wykres 5.2: Przebieg algorytmu dla grafu intree15 i τ=0.1

Warto też zauważyć, że nie dochodzi w przypadku tego grafu do nagłych zmian wybranej konfiguracji. Przy przejściu od lepszego do gorszego czasu szeregowania i odwrotnie nie następuje skok o kilka wartości lecz o 1 lub 2 jednostki czasu. Ma to związek z typem grafu. Procesy są tutaj powiązane w linii prostej, nie ma skomplikowanych zależności, a koszty zmiany procesora są niewielkie. To skutkuje tym, że generowanie nowych konfiguracji nie powoduje powstania dużo słabszych, a jedynie takich z niewielkimi odchyleniami. Z kolei na wykresie dla dużego prawdopodobieństwa (np. 8.0) brak jest jakichkolwiek odchyleń, wszystkie linie nakładają się na siebie. Przyczyny są tutaj dwie:

- nie wybrano konfiguracji o rankingu niższym niż 2 (wykorzystano dwie najlepsze w kolejności),
- graf procesów jest prosty, koszty zmiany procesu są niskie (wynoszą 1) i istnieje wiele konfiguracji, które dają najlepszy czas szeregowania. Przez to

zmiana procesora na którym wykonane zostanie jedno zadanie, a pozostawienie pozostałych bez zmian często nie spowoduje zmiany czasu szeregowania.



Rys. 5.4 Graf procesów tree15

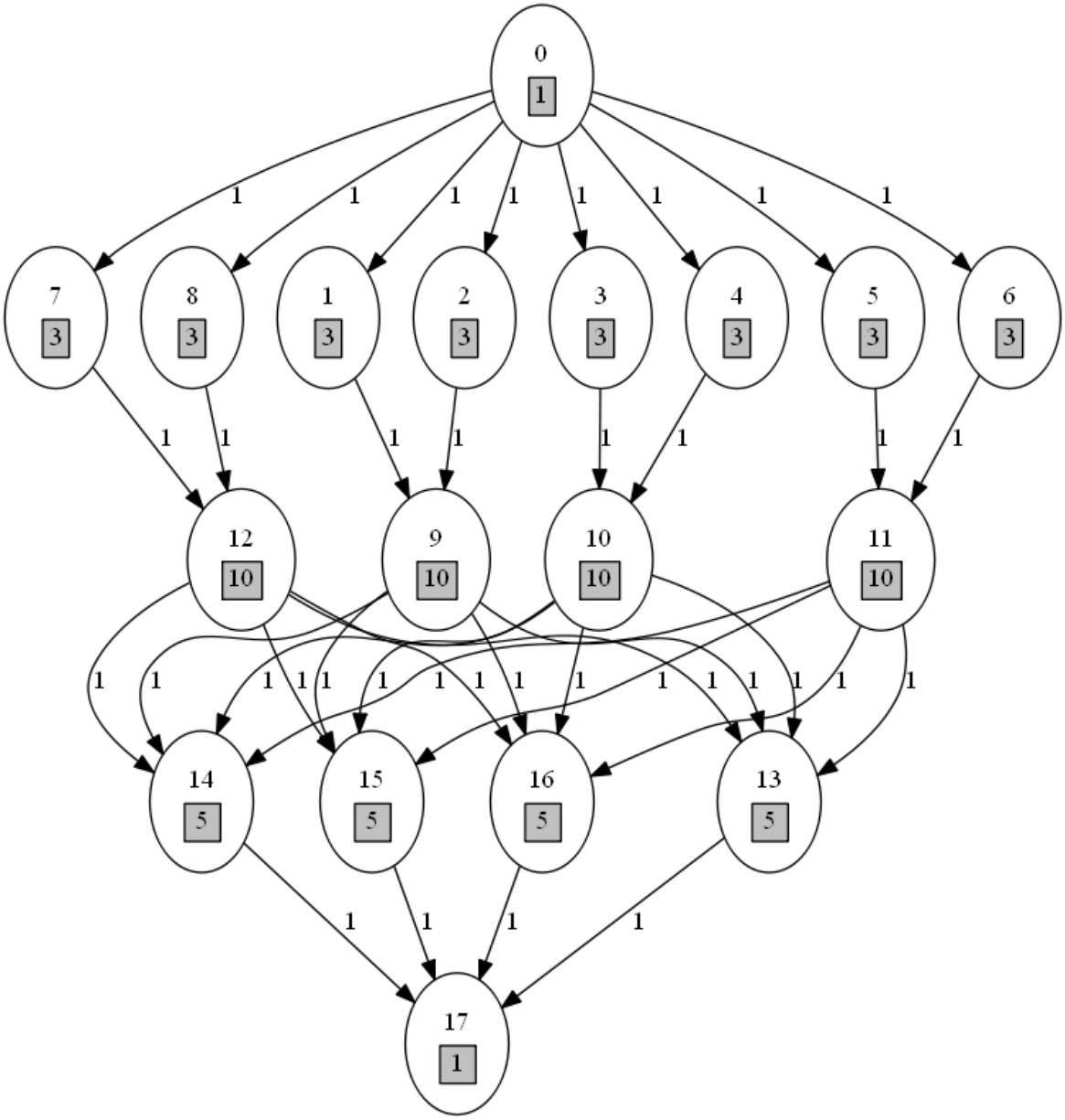
Wyniki przeprowadzonych testów na grafie tree15 są podobne do wyników otrzymanych na drzewie intree15. Podstawową przyczyną jest podobieństwo grafów, intree jest odwróceniem grafu tree. Tutaj także najlepsze wyniki pojawiają się przy wprowadzeniu parametru prawdopodobieństwa równego 5.0 lub 8.0. Jak pokazuje Tabela 5.3 wszystkie uruchomienia zakończyły się sukcesem i zawsze otrzymano najlepszy wynik wynoszący 9.

Tabela 5.3 Dane statystyczne testów na grafie tree15 z 2 procesorami

τ	średnia iteracji	wariancja iteracji	min	Q1	mediana	Q3	max	moda	moda (%)	najlepsze wyniki (%)	śr. T	min T
0.1	4,114	21,388	0	1	3	6	34	0	21	100	9	9
0.2	3,073	11,363	0	1	2	5	22	0	22	100	9	9
0.5	2,152	3,781	0	1	2	3	16	1	27,8	100	9	9
0.8	1,642	2,03	0	1	1	2	8	1	33,4	100	9	9
1.0	1,673	1,744	0	1	1	2	8	1	33,8	100	9	9
1.5	1,443	1,43	0	1	1	2	6	1	36,3	100	9	9
2.0	1,433	1,209	0	1	1	2	5	1	35,4	100	9	9
5.0	1,418	1,192	0	1	1	2	6	1	41,8	100	9	9
8.0	1,394	1,138	0	1	1	2	6	1	38	100	9	9

5.4 Graf g18 - 2 procesory

Graf g18 [18] widoczny na Rys. 5.5 jest znacznie bardziej złożony niż opisane wcześniej intree15 i tree15. Wagi krawędzi są identyczne i wynoszą 1 - koszt zmiany procesora jest niewielki. Oznaczone na szarym tle czasy wykonywania pojedynczych modułów są zróżnicowane i inne na różnych poziomach grafu. Zależności są tak skonstruowane, że przejście do następnego poziomu n+1 w większości przypadków wymaga wykonania wszystkich zadań na poziomie n. Powoduje to, że wybierając najlepszą ścieżkę, ważne jest równoległe wykonywanie zadań na poziomie n i podział zadań w taki sposób, żeby przejście do poziomu n+1 powodowało niewielką stratę czasową związaną z przesłaniem danych między procesorami. Najlepszy czas szeregowanie na 2 procesorach dla tego grafu wynosi 46.

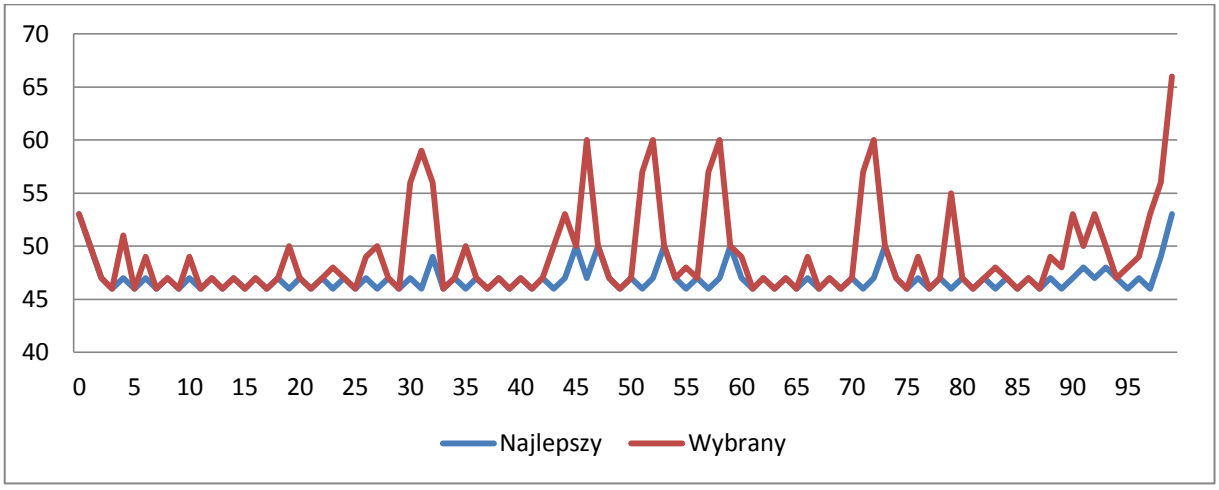


Rys. 5.5 Graf procesów g18

Dane statystyczne przedstawione w Tabela 5.4 znacznie odbiegają od wyników otrzymanych przy grafach intree15 i tree15. Uruchomienia dla tego grafu najlepsze wyniki osiągały przy τ=0.5, gdy ilość iteracji potrzebnych do osiągnięcia optimum była najmniejsza (średnia = 7.095, mediana 5) i wszystkie uruchomienia zakończyły się sukcesem.

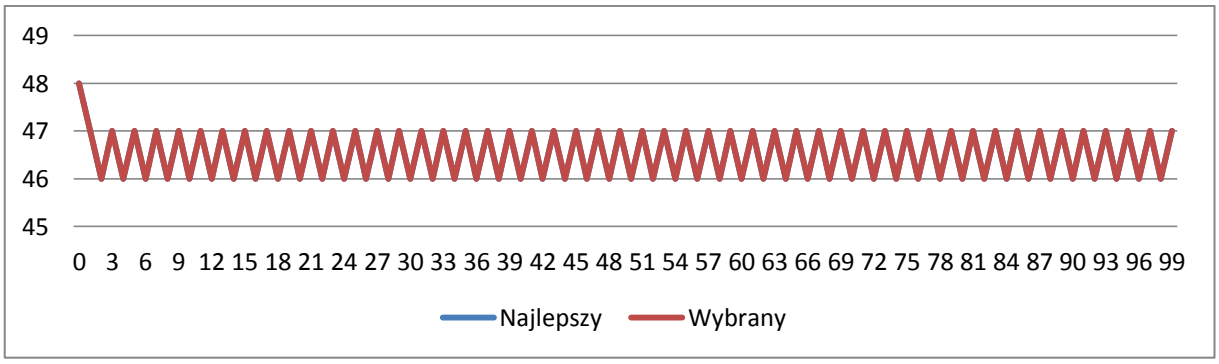
Tabela 5.4 Dane statystyczne testów na grafie g18 z 2 procesorami

τ	średnia iteracji	wariancja iteracji	min	Q1	mediana	Q3	max	moda	moda (%)	najlepsze wyniki (%)	śr. T	min T
0.1	21,813	454,781	0	7	15	29	100	4	5,4	98,5	46	46
0.2	13,55	172,434	0	4	9	19	82	3	6,2	100	46	46
0.5	7,095	80,985	0	3	4	8	80	3	15,2	100	46	46
0.8	8,765	268,376	0	2	4	7	100	3	18	99	46	46
1.0	10,234	468,444	0	2	3,5	5	100	3	20,9	96,5	46	46
1.5	16,896	1067,881	0	2	3	5	100	3	23,2	87,7	46,2	46
2.0	16,625	1116,619	0	2	3	4	100	3	25,3	86,7	46,2	46
5.0	17,341	1193,436	0	2	3	4	100	3	24,3	85,4	46,2	46
8.0	18,643	1292,1	0	2	3	4	100	3	26,7	83,7	46,3	46



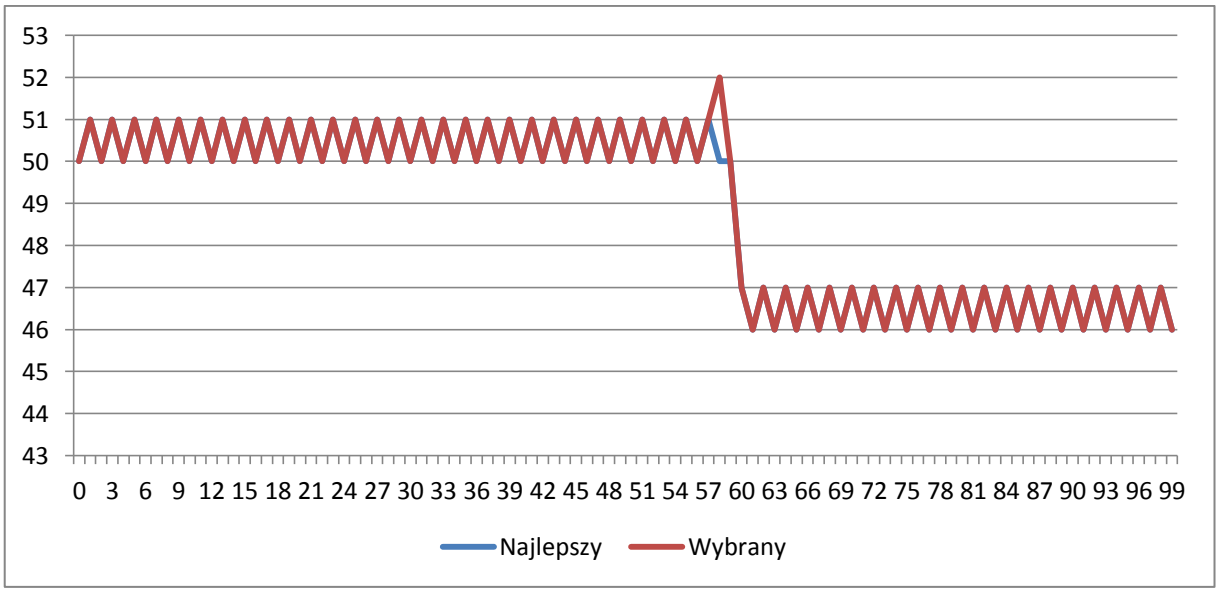
Wykres 5.3: Przebieg algorytmu τ=0.5, najlepszy rezultat osiągnięty w iteracji 4

Typowy przebieg algorytmu został przedstawiony na Wykres 5.3. Można zauważyć, że wielokrotnie wybierana jest słabsza konfiguracja (dłuższy czas wykonania), jednak w efekcie w większości przypadków w kilku kolejnych krokach algorytm znów osiąga optymalny rezultat.



Wykres 5.4: Przebieg algorytmu τ=8.0, optimum osiągnięto w 3 iteracji, linie nakładają się na siebie

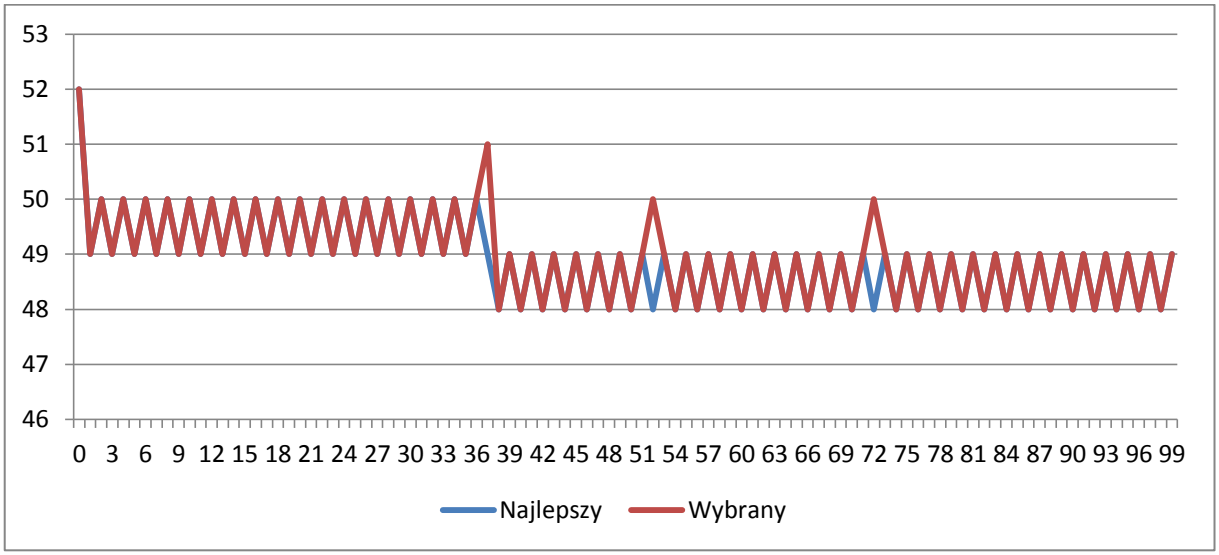
Analizując dane wynikowe należy zwrócić uwagę na dominantę ilości potrzebnych iteracji oraz procent wystąpień tej wartości. W przypadku, gdy wartość parametru jest równa 8.0 przebieg algorytmu skupia się na wybieraniu najlepszego wyniku do następnej iteracji co przedstawia Wykres 5.4 na którym linie czerwona i niebieska nakładają się na siebie. Taki sposób postępowania okazał się sukcesem w 83.7% przypadków. Pozostałe uruchomienia nie zakończyły się sukcesem ze względu na minima lokalne w których algorytm zapętlił się. Zobrazowane jest to na Wykres 5.5 i Wykres 5.6 przedstawiających 2 uruchomienia dla parametru τ=5.0. Pierwszy z nich pokazuje przebieg, w którym znaleziono optimum. Jak można zauważyć algorytm próbuje wybierać konfiguracje zbliżone do optimum (linia czerwona) cały czas wybierając do następnej iteracji najlepszy wynik (linia niebieska, na którą nakłada się czerwona). Dopiero w iteracji 58 zostaje wylosowana słabsza konfiguracja i następuje wyjście z minimum lokalnego, w efekcie prowadzi to do znalezienia w kilku następnych iteracjach minimum globalnego.



Wykres 5.5: Przebieg algorytmu τ=5.0, ilość iteracji do osiągnięcia minimum = 61

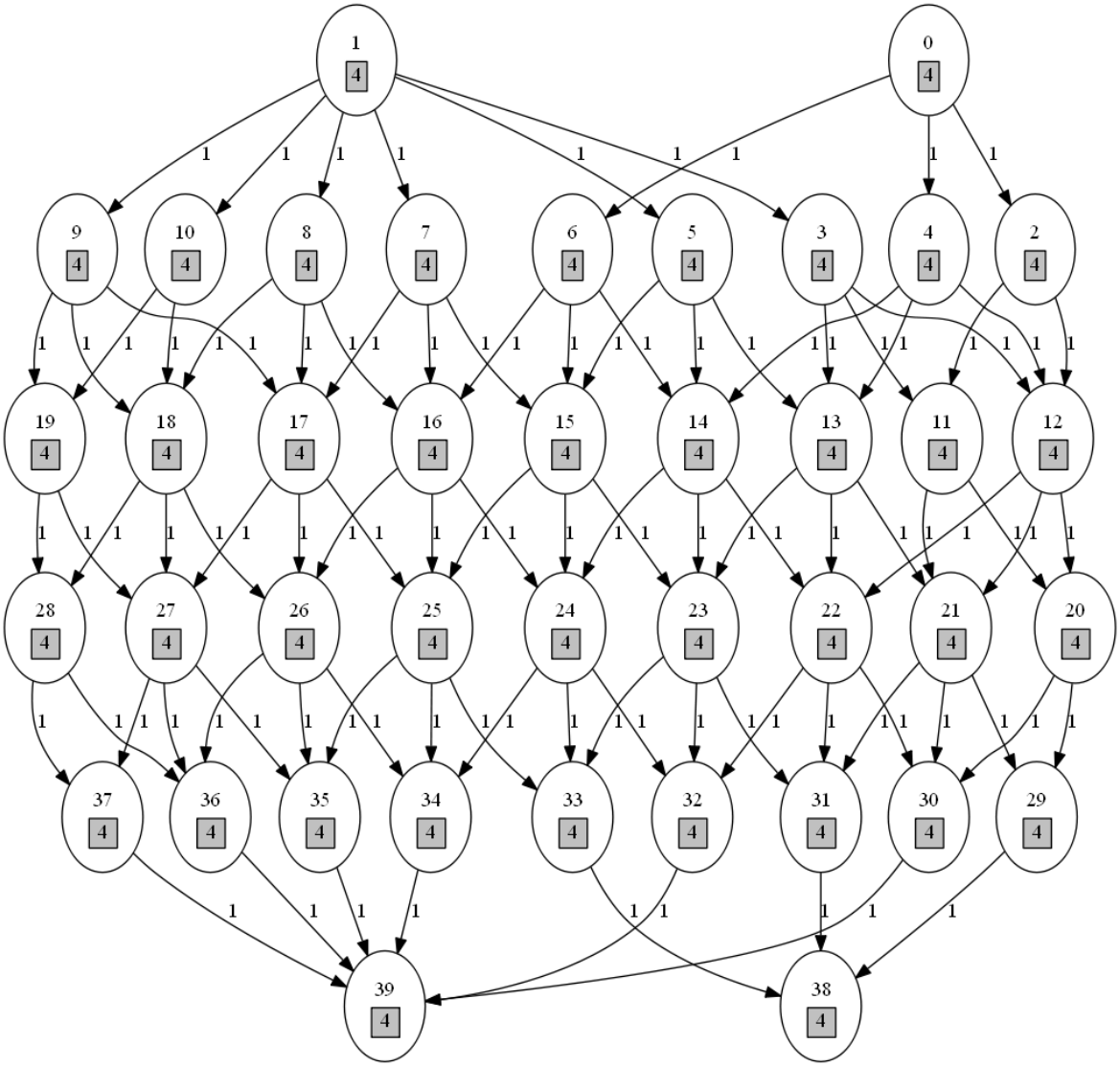
Minimum lokalne w przypadku algorytmu GEO to sytuacja w której wybranie najlepszej konfiguracji spośród możliwych w iteracji *n*, spowoduje że w iteracji *n*+1 najlepszą konfiguracją będzie ta z iteracji *n*. Wybierając zawsze najlepsze rozwiązanie następuje zapętlenie. Jest to widoczne na drugim wykresie. W trakcie przebiegu trzykrotnie nastąpiła próba wyjścia z minimum lokalnego. W iteracji 37 próba pozwoliła na wyjście z jednego minimum lokalnego do drugiego, co w rezultacie poprawiło wynik algorytmu. Następnie w iteracjach 52 i 72 nastąpiła nieudana próba wyjścia poza minimum

lokalne i w rezultacie algorytm zakończył się po 100 iteracjach z wyznaczonym czasem szeregowania 48, nie osiągnięto optymalnego wyniku.



Wykres 5.6: Przebieg algorytmu τ=5.0, nie znaleziono minimum globalnego

5.5 Graf g40 – 2 procesory



Rys. 5.6 Graf procesów g40

Na przedstawionym na Rys. 5.6 grafie g40 [19] podobnie jak na grafie g18 można zauważyć podział zadań na poziomy, z tą różnicą że są bardziej złożone i jest ich więcej. Graf składa się z 40 procesów o czasie wykonania 4. Minimalny czas szeregowania zadań dwóch procesorach typu FULL2 wynosi 80, co w praktyce powoduje że żaden z procesorów nie musi bezczynnie oczekiwać na przesłanie danych.

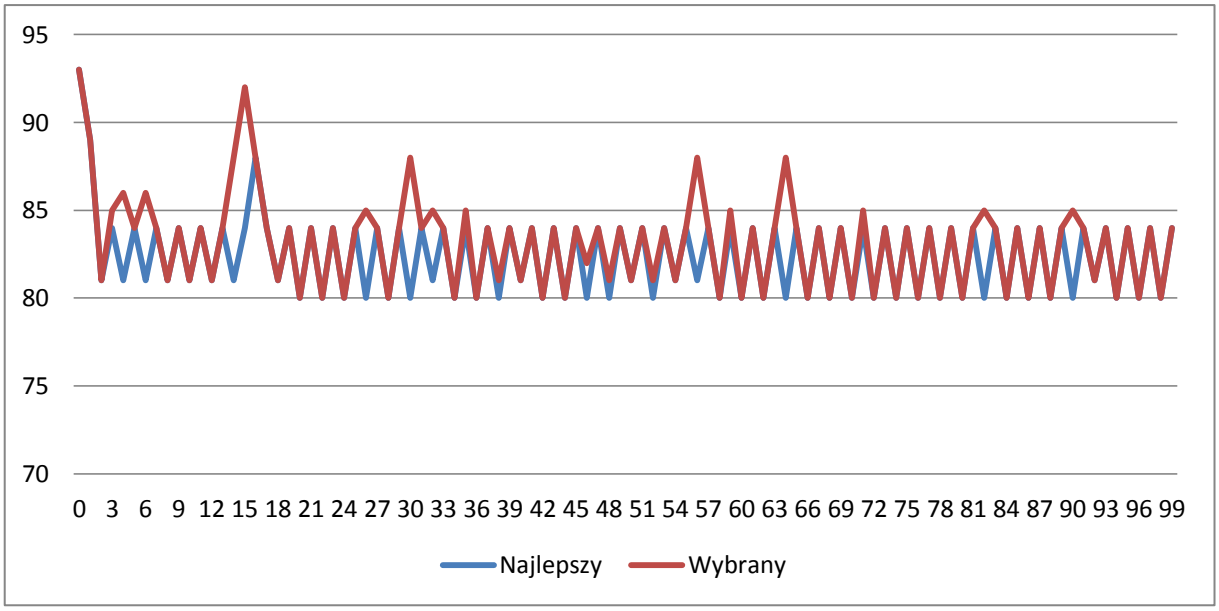
Na podstawie danych widocznych w Tabela 5.5 można stwierdzić, że w żadnej z konfiguracji skuteczność nie wyniosła 100%. Dla τ równego 0.2 i 0.5 w jednym przypadku na 1000 nie udało się znaleźć optymalnego rozwiązania w czasie 100 iteracji (wykonanie 4000 funkcji przystosowania). Kwartyle 1, 2 i 3 wskazują, że najlepszą wartością parametru będzie 1.0. Przy zastosowaniu takiej wartości zmiennej byłoby konieczne

wykonanie najmniejszej liczby iteracji w celu znalezienia optimum. Potwierdza to także średnia. Jednak wariancja wskazuje najmniejszą rozbieżność wyników dla τ=0.5, skuteczność algorytmu także jest najlepsza dla tej wartości, a czas wykonania *T* jest równy 80.001. Celem implementacji algorytmu jest uruchamianie go na grafie o nieznanym czasie szeregowania, dlatego ważne byłoby kilkukrotne uruchomienie programu w różnym zakresie parametru τ w przedziale [0.5, 1.0] z różną ilością iteracji. Dopiero zestawienie wyników otrzymanych w tym procesie pozwoliłoby na wybranie najlepszej proponowanej konfiguracji.

Tabela 5.5 Dane statystyczne testów na grafie g40 z 2 procesorami

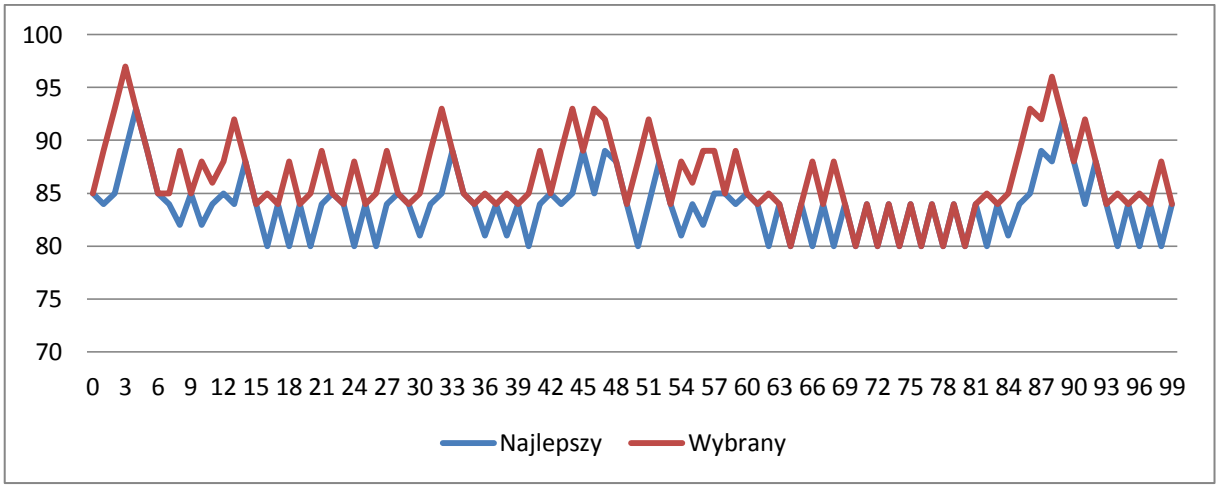
τ	średnia iteracji	wariancja iteracji	min	Q1	mediana	Q3	max	moda	moda (%)	najlepsze wyniki (%)	śr. T	min T
0.1	28,964	611,51	0	10	21	40	100	4	3,2	97,2	80,04	80
0.2	15,945	175,063	0	6	12	22	100	7	5,4	99,9	80,00	80
0.5	9,705	96,671	0	4	7	12	100	5	8,9	99,9	80,00	80
0.8	8,948	134,232	0	3	6	10	100	3	12,8	99,8	80,00	80
1.0	8,529	123,577	0	3	5	9	100	4	12,4	99,6	80,00	80
1.5	10,167	247,521	0	3	5	9	100	3	15,5	98,8	80,01	80
2.0	14,134	528,551	0	3	5	12	100	3	13,9	96,5	80,04	80
5.0	31,425	1733,472	0	3	5	85	100	100	23,9	76,1	80,26	80
8.0	38,93	2122,003	0	3	5	100	100	100	35,6	64,4	80,38	80

Wykres 5.7 przedstawia typowy przebieg algorytmu na grafie g40. Różni się on od wykresów dla grafów intree15, tree15 i g18 większą różnicą między czasem szeregowania w kolejnych iteracjach. Na przykład w iteracji 42 czas szeregowania wyniósł 80, a w następnej czas ten wynosił 84, w kolejnej znów 80. Oznacza to, że konfiguracje są bardzo czułe na zmianę przypisania procesorów. Pamiętając o tym, że w jednej iteracji zmieniany zostaje tylko jeden bit (procesor dla pojedynczego zadania), można stwierdzić, że taka mutacja będzie skutkować częstym zagłębieniem się w minimum lokalnym. Można zauważyć, że przy iteracjach 13 i 55 wyjście z takiego minimum przez wybór gorszej konfiguracji pozwala w następnych iteracjach znaleźć optymalny wynik.



Wykres 5.7: Przebieg algorytmu na grafie g40 τ=0.5 - 100 iteracji, najlepszy wynik osiągnięto w 21 iteracji

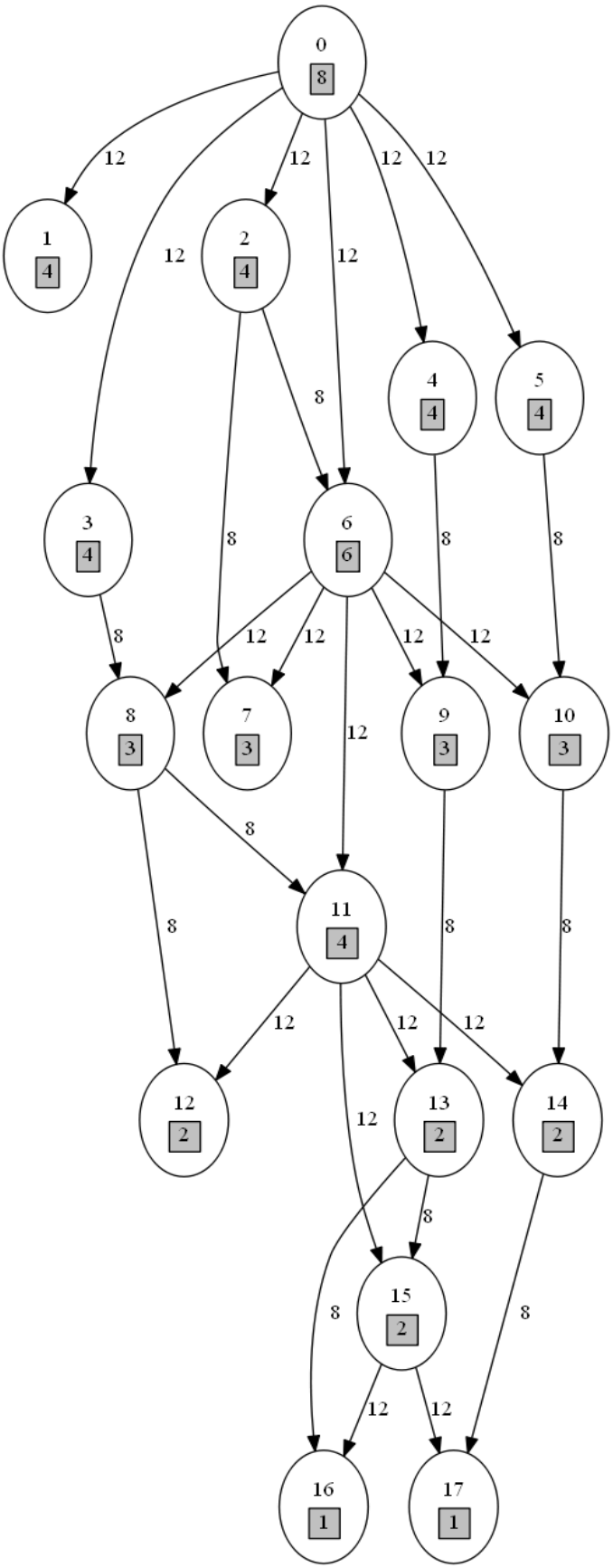
Algorytm nie wychodzi poza rozwiązania optymalne (minima lokalne) na więcej niż 2 iteracje. Kontrprzykładem jest Wykres 5.8 przedstawiający przebieg dla τ=0.1. Tutaj widoczne jest między iteracjami 44-49 i 86-93 odejście od minimum lokalnego i bezowocne wykonywanie algorytmu bez zbliżania się celu. Czas jest tracony na wykonywanie bezcelowych obliczeń na słabszych konfiguracjach.



Wykres 5.8: Przebieg algorytmu na grafie g40 τ=0.1 - 100 iteracji, najlepszy wynik osiągnięto w 18 iteracji

5.6 Graf gauss18 – 2 procesory

Graf gauss18 [20] przedstawiony na Rys. 5.7 jest najbardziej złożonym grafem spośród testowanych. Koszty zmiany procesora i czasy wykonania poszczególnych zadań są bardzo zróżnicowane. Bardziej optymalne w takiej sytuacji jest wykonanie dwóch zadań na jednym procesorze niż przesłanie na drugi i wykonanie równoległe.



Rys. 5.7 Graf procesów gauss18

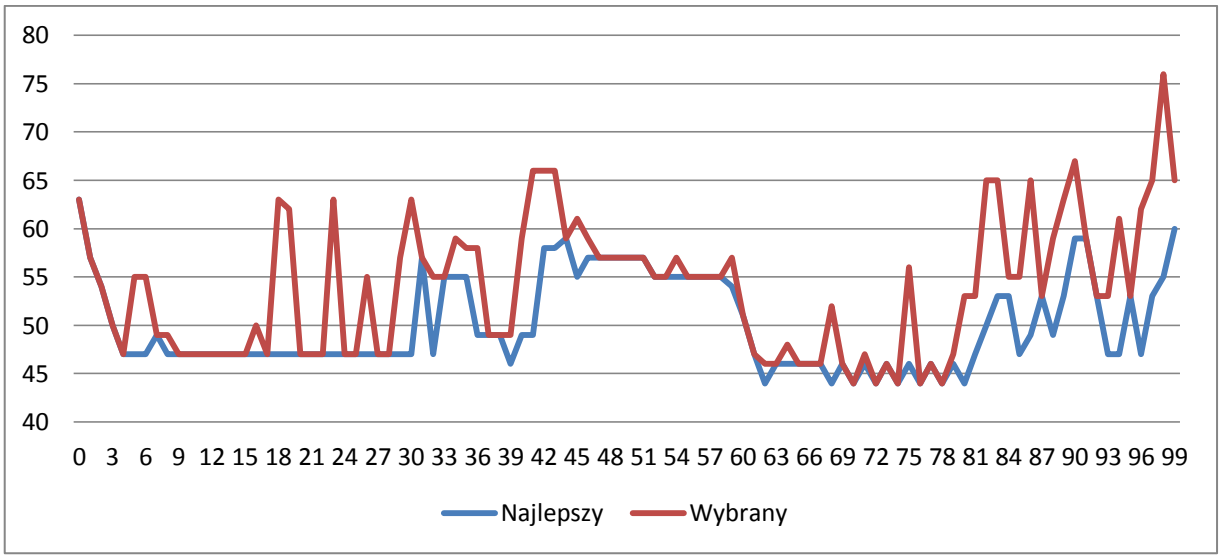
Pierwsze próby znalezienia optymalnego rozwiązania przy 100 iteracjach nie przyniosły dobrych rezultatów i najlepsze wyniki otrzymano tylko w 46.4% przypadków. Dopiero zwiększenie liczby iteracji do 400 spowodowało, że odnaleziono najlepszą konfigurację w większości przypadków. Wyniki przedstawione w Tabela 5.6 pokazują, że w takim wypadku liczba znalezionych poprawnych wyników T=44 stanowiła 95%, a średni czas był równy 44.13. Średnia liczba iteracji potrzebna do znalezienia optimum wyniosła 124 przy bardzo dużej wariancji wynoszącej 12447. Jest to bardzo dobry wynik, jednak w efekcie przy takiej liczbie iteracji należało sprawdzić 7200 przypadków, co stanowi 2,75% wszystkich możliwych kombinacji dla tego grafu. W przypadku 100 iteracji konieczne jest sprawdzenie 1800 konfiguracji - 0,69% możliwych kombinacji.

Tabela 5.6 Dane statystyczne testów na grafie gauss18 z 2 procesorami

τ	średnia iteracji	wariancja iteracji	min	Q1	mediana	Q3	max	moda	moda (%)	najlepsze wyniki (%)	śr. T	min T
0.1	98,866	71,419	2	100	100	100	100	100	97,6	2,4	49,67	44
0.2	94,264	314,152	8	100	100	100	100	100	87,9	12,2	47,34	44
0.5	74,294	1127,879	5	41	100	100	100	100	53,9	46,4	45,65	44
0.5 (400 it.)	124,08	12447,13	3	30	92,5	189	400	400	5	95	44,13	44
0.8	72,011	1325,61	3	31	100	100	100	100	56,4	43,7	46,05	44
1.0	71,972	1413,949	3	28	100	100	100	100	60,3	39,7	46,41	44
1.5	75,241	1450,866	3	31	100	100	100	100	68,2	31,8	47,36	44
2.0	78,849	1295,428	2	58	100	100	100	100	71,9	28,1	47,90	44
5.0	86,383	1006,164	2	100	100	100	100	100	82,6	17,4	49,56	44
8.0	88,933	901,31	2	100	100	100	100	100	87,8	12,2	50,07	44

Na Wykres 5.9 przedstawiono typowy przebieg algorytmu dla parametru τ=0.5. W porównaniu do pozostałych algorytmów wyraźnie zauważalna jest różnica pomiędzy linią pokazującą najlepszą konfigurację w iteracji oraz wybraną. Dzieje się tak ze względu na złożoność grafu, w którym często zmiana w szeregowaniu przypisania pojedynczego zadania do procesu skutkuje znaczną zmianą czasu szeregowania. W grafach tree15, intree15, g18 i g40 taka zmiana wielokrotnie nie wpływała na czas wykonania lub powodowała, że konfiguracja szeregowania miała czas różniący się o nieznaczną wielkość. Konfiguracje można było pogrupować i wybranie konfiguracji 3 lub 4 oznaczało wybranie jednej z wielu optymalnych konfiguracji. W przypadku tego grafu wybranie konfiguracji 3 i wyższej oznacza wybranie gorszego rozwiązania. Spowodowało to także, że poczynając od iteracji 28 do iteracji 47, gdy nastąpił częsty wybór słabszych konfiguracji znacznie pogorszył się najlepszy czas wykonania algorytmu w danych iteracjach. Podobna sytuacja

rozpoczęła się przy 80 iteracji. Jednak tak jak dla poprzednich grafów znalezienie najlepszej konfiguracji wiąże się z wyjściem z minimum lokalnego i poszukiwaniem optimum globalnego.



Wykres 5.9: Przebieg algorytmu na grafie gauss18 τ=0.5 - 100 iteracji, najlepszy wynik osiągnięto w 74 iteracji

5.7 Porównanie algorytmu GEO ze standardowym algorytmem genetycznym

W Tabela 5.7 dokonano porównaniu otrzymanych wyników średnich czasów szeregowania algorytmu genetycznego z wynikami otrzymanymi w poniższej pracy. Wyniki są bardzo zbliżone. W przeciwieństwie do algorytmu generycznego GEO nie zawsze potrafił znaleźć najlepsze uszeregowanie.

Tabela 5.7 Średnie czasy szeregowania dla grafów testowych

graf programu	standardowy algorytm genetyczny	algorytm GEO
tree15	9	9
intree15	9	9
gauss18	44	44,128
g18	46	46
g40	80	80,001

5.8 Wyniki na systemach czteroprocesorowych

Algorytm GEO pozwala na wykorzystanie dowolnej liczby procesorów podczas szeregowania. Jednak ze względu na łatwość implementacji w przypadku wykorzystania liczby procesorów będącej potęgą 2 postanowiono wykonać testy dla systemu z 4 procesorami pomijając system trójprocesorowy. W poniższym rozdziale przedstawiono wyniki dla konfiguracji FULL4.

Tabela 5.8 Dane statystyczne testów na grafie tree15 z 4 procesorami

τ	średnia iteracji	wariancja iteracji	min	Q1	mediana	Q3	max	moda	moda (%)	najlepsze wyniki (%)	śr. T	min T
0.1	15,371	218,848	0	5	11	21	100	2	5,5	99,9	7,00	7
0.2	9,547	74,568	0	3	7	14	82	2	8,2	100	7,00	7
0.5	5,665	30,449	0	2	4	7	40	2	15,8	100	7,00	7
0.8	5,437	42,483	0	2	3	6	56	2	21	100	7,00	7
1.0	5,826	78,779	0	2	3	6	77	2	21,8	100	7,00	7
1.5	6,744	137,396	0	2	3	6	100	2	22,8	99,7	7,00	7
2.0	9,091	289,268	0	1	3	6	100	2	23,5	98,9	7,01	7
5.0	21,867	1345,315	0	1	2	15	100	1	24,1	85,5	7,15	7
8.0	27,411	1808,969	0	2	2	88	100	2	26,7	75,4	7,25	7

W Tabela 5.8 przedstawiono wyniki wykonania algorytmu dla grafu tree15. Szeregowanie dla 4 procesorów okazało się trudniejszym zadaniem dla algorytmu i potrzebnych było więcej iteracji do osiągnięcia najlepszych rezultatów niż przy 2 procesorach. Jednak i w tym wypadku algorytm przy odpowiedniej wartości parametru τ nie ma problemów ze znalezieniem optymalnych konfiguracji z czasem szeregowania równym 7, robi to również ze 100% skutecznością.

Tabela 5.9 Dane statystyczne testów na grafie g18 z 4 procesorami

τ	średnia iteracji	wariancja iteracji	min	Q1	mediana	Q3	max	moda	moda (%)	najlepsze wyniki (%)	śr. T	min T
0.1	92,31	396,288	1	100	100	100	100	100	82,7	17,5	27,72	26
0.2	71,391	1067,936	2	40	90	100	100	100	45,4	55,2	26,52	26
0.5	43,694	999,588	1	16	35	67	100	100	12,2	88,2	26,12	26
0.8	45,37	1170,101	2	14	36	75	100	100	16,6	83,4	26,24	26
1.0	49,32	1386,778	1	12	40	99	100	100	24,8	75,5	26,35	26
1.5	63,95	1604,576	1	16	88	100	100	100	47,5	52,8	27,03	26
2.0	70,397	1622,892	1	22	100	100	100	100	60,6	39,5	27,64	26
5.0	79,22	1446,24	1	100	100	100	100	100	75,5	24,5	28,73	26
8.0	83,414	1271,34	2	100	100	100	100	100	81,9	18,1	29,52	26

Tabela 5.9 przedstawia wyniki szeregowania dla grafu g18. W tym wypadku 100 iteracji nie było wystarczające, aby algorytm miał 100% skuteczność niezależnie od wartości parametru prawdopodobieństwa. Dla τ=0.5 w 88.2% przypadków znaleziono najlepsze rozwiązanie T=26, a średni czas szeregowania wynoszący 26.12 okazał się dobrą wartością. Średnio należało wykonać 43 iteracje w celu znalezienia optymalnego wyniku.

Tabela 5.10 Dane statystyczne testów na grafie g40 z 4 procesorami

τ	średnia iteracji	wariancja iteracji	min	Q1	mediana	Q3	max	moda	moda (%)	najlepsze wyniki (%)	śr. T	min T
0.5	331,1	29014,77	125	127	361,5	500	500	500	40	60	45,40	45

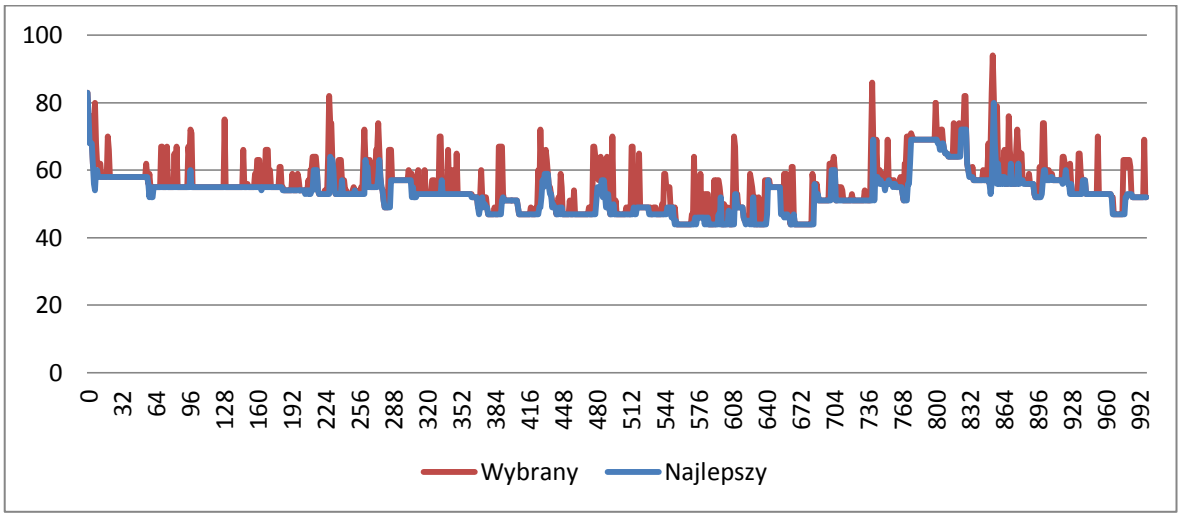
Wyniki dla grafu g40 okazały się bardzo słabe przy 100 iteracjach, dopiero zwiększenie liczby kroków do 500 pozwoliło na osiągnięcie zadowalających rezultatów. Wyniki w Tabela 5.10 pokazują, że średni czas szeregowania wyniósł 45.4 oraz skuteczność algorytmu na poziomie 60%.

Tabela 5.11 Dane statystyczne testów na grafie gauss18 z 4 procesorami

τ	średnia iteracji	wariancja iteracji	min	Q1	mediana	Q3	max	moda	moda (%)	najlepsze wyniki (%)	śr. T	min T
0.5	666,5	91185,61	172	453	622	1000	1000	1000	30	70	44,80	44

Tabela 5.11 pokazuje wyniki uruchomień dla grafu gauss18 przy uruchomieniu programu dla 1000 iteracji. 70|% przypadków przy wartości parametru τ=0.5 osiągnęło najlepszy wynik szeregowania równy 44, średnia wszystkich uruchomień wyniosła 44.80. Duża liczba iteracji potrzebna do znalezienia optimum > 660 jest spowodowana specyfiką grafu oraz bardzo dużą kombinacją uszeregowań. Należy pamiętać, że w przypadku

algorytmu GEO w trakcie jednej iteracji na 4 procesorach sprawdzane jest ustawienie procesu wyłącznie na dwóch z procesorów. Na Wykres 5.10 pokazano typowy przebieg algorytmu dla grafu gauss18. W tym wypadku przy 4 procesorach można zauważyć, że algorytm często wybiera gorsze ustawienia (linia czerwona jest oddalona od niebieskiej) jednak w dalszym ciągu pozostając w pewnym minimum, które trudno jest opuścić. Od 670 iteracji można zaobserwować, że algorytm zaczyna przeszukiwać inny obszar rozwiązań, który znacznie oddala go od optimum.



Wykres 5.10 Typowy przebieg algorytmu dla grafu gauss18 z 4 procesorami

Tabela 5.12 przedstawia zestawienie średnich końcowych czasów szeregowania algorytmu GEO oraz średnich uzyskanych przez standardowy algorytm genetyczny.

Tabela 5.12 Średnie końcowe czasy szeregowania dla grafów testowych – 4 procesory

graf programu	standardowy algorytm genetyczny	algorytm GEO
tree15	7.0	7.0
g18	26.0	26.12
g40	45.0	45.4
gauss18	44.0	44.8

5.9 Testy algorytmu GEO na ośmiu procesorach

Spośród testowanych grafów dla g18 i g40 czasy szeregowania mogą zostać poprawione, wykorzystanie większej liczby procesorów pozwala na wykonanie procesów szybciej. Czasy dla tree15 i gauss18 nie mogą zostać poprawione ze względu na ich strukturę. W przypadku grafu drzewiastego, gdzie czas przesłania jest równy czasowi wykonania, czas szeregowania poprawia się do momentu gdy liczba procesorów jest równa połowie liczby końcowych procesów (węzły bez potomków). Czas dla grafu gauss18 nie

może zostać poprawiony ze względu na złożone zależności, które pozwalają w praktyce na wykorzystanie tylko dwóch procesorów jednocześnie.

Tabela 5.13 Dane statystyczne testów na grafie tree15 z 8 procesorami

τ	średnia iteracji	wariancja iteracji	min	Q1	mediana	Q3	max	moda	moda (%)	najlepsze wyniki (%)	śr. T	min T
0.5	3,757	23,768	0	1	2	5	52	1	24,5	100	7	7
0.8	4,042	49,97	0	1	2	4	92	1	29,9	100	7	7

W Tabela 5.13 przedstawiono wyniki wykonania algorytmu na grafie tree15. Graf okazał się łatwy w uszeregowaniu i wystarczyło kilka iteracji, aby znaleźć najlepsze rozwiązanie *T*=7. Jak wspomniano wcześniej czas wykonania pojedynczego zadania jest równy czasowi przesłania danych pomiędzy procesorami, to powoduje, że wykonanie dwóch niezależnych zadań ze wspólnym przodkiem na jednym procesorze A będzie równe czasowy wykonania dwóch zadań na różnych procesorach A i B pod warunkiem, że będzie konieczne przesłanie danych z procesora A na B.

Tabela 5.14 Dane statystyczne testów na grafie g18 z 8 procesorami

τ	średnia iteracji	wariancja iteracji	min	Q1	mediana	Q3	max	moda	moda (%)	najlepsze wyniki (%)	śr. T	min T
0.5	25,256	365,382	1	11	21	33	100	6	4,2	99,3	24,02	24
0.8	23,138	398,726	1	9	17	31	100	6	4,3	99,6	24,01	24

Tabela 5.14 zawiera wyniki szeregowania dla grafu g18. W tym wypadku z ponad 99% skutecznością znajdowano najlepsze rozwiązanie *T*=24 podczas wykonywania 100 iteracji. Szeregowanie okazało się lepsze od szeregowania na 4 procesorach dla tego grafu. Przyczyną jest większy zbiór optymalnych rozwiązań w przypadku wykorzystania 8 procesorów.

Tabela 5.15 Dane statystyczne testów na grafie g40 z 8 procesorami

τ	średnia iteracji	wariancja iteracji	min	Q1	mediana	Q3	max	moda	moda (%)	najlepsze wyniki (%)	śr. T	min T
0.5	204,5	25955,61	62	79	155	298	529	62	10	100	33	33

Szeregowanie dla grafu g40 przy ustawieniu limitu 1000 iteracji znalazło najlepsze konfiguracje z czasem T=33 we wszystkich uruchomieniach. Średnia ich liczba to 204.5, ale w najgorszym wypadku należało wykonać 529 iteracji co oznaczało konieczność

wykonania 63480 funkcji przystosowania (3 bity na procesor * 40 szeregowanych zadań * 529 iteracji). Jednak jest to nieznaczny ułamek możliwych kombinacji wszystkich rozwiązań wynoszących 2^{120}.

Tabela 5.16 Dane statystyczne testów na grafie gauss18 z 8 procesorami

τ	średnia iteracji	wariancja iteracji	min	Q1	mediana	Q3	max	moda	moda (%)	najlepsze wyniki (%)	śr. T	min T
0.5	3394,1	944672,1	1731	2237	4000	4000	4000	4000	60	40	45,7	44

Pokazane w Tabela 5.16 dane dla grafu gauss18 pokazują, że aby odnaleźć w tym wypadku najlepsze rozwiązanie konieczne jest wykonanie około 2200 iteracji. Ustawienie limitu na 4000 pokazało, że tylko 40% uruchomień jest w stanie znaleźć najlepsze rozwiązanie T=44. Patrząc na średnią czasów równą 45.7 zauważalne jest, że algorytm często nie zbliża się nawet do najlepszego rozwiązania. Najlepszy czas szeregowania nie zmienia się w zależności od liczby procesorów, ze względu na fakt, że najlepsze szeregowanie można osiągnąć wykorzystując 2 procesory. Zwiększenie liczby procesorów powoduje znaczny wzrost liczby konfiguracji wśród których algorytm spróbuje znaleźć optymalne, jednak zbiór najlepszych rozwiązań nie zmienia się w takim tempie. Z drugiej strony trudno jest algorytmowi znaleźć rozwiązanie, które okazuje się najlepsze w przypadku konfiguracji, gdzie większość zadań uruchamianych jest na 1 procesorze i tylko końcowe procesy są uruchamiana na innych procesorach. Przyczyną takiej optymalnej konfiguracji jest wysoki kosz przesyłania zadań w stosunku do czasu wykonania samego zadania.

Program rozpoczyna pracę od losowej konfiguracji, w której zadania są często rozdzielone pomiędzy wiele procesorów. Przez to więcej czasu zajmuje dojście do optimum przy zmienianiu tylko jednego przypisania procesora do zadania na iterację. W czasie jednej iteracji mutacja dotyczy pojedynczego bitu, co oznacza też że dla jednego zadania zostaną sprawdzone 3 konfiguracje z innym numerem procesora, a nie wszystkie możliwe procesory. To także spowalnia wyszukiwanie najlepszego rozwiązania dla tego grafu, gdyż podczas iteracji często nie jest wybierana konfiguracja, która przybliża go bezpośrednio do najlepszego rozwiązania.

Tabela 5.17 Średnie końcowe czasy szeregowania dla grafów testowych – 8 procesorów

graf programu	standardowy algorytm genetyczny	algorytm GEO
tree15	7.0	7.0
g18	24.0	24.01
g40	33.0	33.0
gauss18	44.0	45.7

W Tabela 5.17 porównano średnie czasu szeregowania dla algorytmów genetycznych i algorytmu GEO. W przypadku algorytmu GEO najlepsze wyniki znaleziono dla każdego grafu, średnie otrzymanych wyników także są zadawalające, mimo tego że nie zawsze udawało się znaleźć optymalny wynik.

5.10 Podsumowanie testów

Wyniki przedstawione w poniższym rozdziale pokazują, że algorytm GEO potrafi znaleźć najlepsze czasy szeregowania zarówno dla prostych jak i złożonych grafów programów. Może być stosowany przy różnej liczbie procesorów.

Statystyki przeprowadzonych testów pokazały, że bardzo ważnym jest dobór odpowiedniej wartości parametru τ. Najlepsze wyniki otrzymywano dla τ=0.5, jednak w praktyce zalecane jest uruchomienie algorytmu z różnymi wartościami tego parametru zawartymi w przedziale [0.5, 1]. Wybór odpowiedniej wartości τ pozwoli na szybsze odnalezienie optymalnego wyniku. Zbyt niska wartość np. 0.1 spowoduje, że algorytm będzie przeszukiwał wśród zbyt wielu rozwiązań, nie dążąc do zoptymalizowania wyniku. Podobnie wprowadzając zbyt wysoką wartość jak np. 5.0 algorytm będzie miał problem z wyjściem poza minimum lokalne i odnalezieniem najlepszego globalnie rozwiązania.

Przy wielu eksperymentach widać także, że bardzo istotny czynnik stanowi wprowadzenie odpowiedniego limitu iteracji które wykona program. Dla prostych grafów jak np. tree15 lub g18 liczba iteracji koniecznych do znalezienie najlepszego szeregowania była znacznie mniejsza niż w przypadku g40 i gauss18. Grafy z dużą liczbą złożonych zależności potrzebują większego czasu do znalezienia optimum.

Liczba procesorów także odgrywa istotną rolę jeśli chodzi o szybkość znajdowania najlepszego rozwiązania. Wraz ze wzrostem ich liczby konieczne jest wykonanie dużo większej liczby funkcji celu. Spowodowane jest to wykładniczym wzrostem liczby możliwych uszeregowań oraz samym algorytmem, który podczas jednej iteracji sprawdza tylko część procesorów dla danego zadania. Bardzo widoczne było to przy grafie gauss18,

gdzie optymalne rozwiązanie to takie w którym większość zadań musi być wykonana na jednym procesorze. Przy większej liczbie procesorów należy zwiększyć limit iteracji proporcjonalnie do ich liczby, zwłaszcza w przypadku grafów, gdzie koszt zmiany procesora jest znacznie wyższy od czasu wykonania zadania. Należy jednak uważać, gdyż w każdym grafie programu istnieje graniczna liczba procesorów po przekroczeniu której dodanie nowych procesorów nie spowoduje skrócenia czasu szeregowania, a znacznie wydłuży czas wykonywania algorytmu. Zwiększenie ilości procesorów wiąże się bezpośrednio ze znacznym wzrostem możliwych rozwiązań według wzoru (9), maleje przy tym stosunek liczby najlepszych rozwiązań do całego zbioru rozwiązań.

6. Podsumowanie

Celem pracy było zaprojektowanie algorytm GEO do problemu szeregowania oraz porównanie wyników z wynikami otrzymywanymi przy zastosowaniu algorytmów genetycznych. Przeprowadzone eksperymenty pokazały, że GEO dorównuje algorytmom genetycznym. We wszystkich przeprowadzonych eksperymentach znaleziono najlepsze czasy szeregowania przeszukując przy tym przestrzeń rozwiązań odpowiadającą od 0.5% do 2.5% ogóły możliwych kombinacji. Łatwość implementacji również wyróżnia go na tle innych algorytmów heurystycznych. GEO jest doskonałym narzędziem, które można zastosować w problemie szeregowania. Może być stosowany przy prostych jak i bardzo skomplikowanych grafach programów z dużą liczbą zadań i zależności.

Bardzo istotne jest wprowadzenie dwóch parametrów w algorytmie. Wartość τ powinna znaleźć się w przedziale [0.5, 1], gdyż tylko w takich warunkach algorytm nie błądzi losowo po przestrzeni rozwiązań, ani nie gubi się w minimach lokalnych. Wprowadzenie optymalnej liczby iteracji, po której algorytm przerwie działanie, może pozwolić na szybsze wykonanie algorytmu i znalezienie optymalnego rozwiązania. Zbyt mała wartość może prowadzić do sytuacji, gdy GEO nie odnajdzie optimum, a zbyt duża wydłuży czas oczekiwania na rozwiązanie. GEO potrafi bez problemów znaleźć najlepsze rozwiązanie dla każdej liczby procesorów. Należy jednak zwrócić uwagę, że zwiększenie ich liczby wiąże się ze wzrostem czasu wykonywania algorytmu GEO.

Wadą algorytmu GEO zauważalną przy szeregowaniu zadań na ponad 2 procesorach była zbyt wolna mutacja. Pozwala na zmianę tylko jednego bitu, a tym samym przypisania pojedynczego zadania do procesora, w czasie jednej iteracji. Wartym rozważenia jest modyfikacja pozwalająca na mutowanie większej liczby zmiennych projektowych (nie dwóch bitów) podczas jednej iteracji. Można byłoby wybrać 2 bity z rankingu odpowiedzialne za zmianę przypisania dwóch różnych zadań, po czym sprawdzić czy zmutowanie tych bitów jednocześnie spowoduje poprawnie czasu szeregowania.

Nieodpowiedni dobór parametru τ skutkuje losowym działaniem algorytmu lub utknięciem w minimum lokalnym. Można byłoby zrezygnować z tego parametru i zastąpić go odpowiednią funkcją kary, która pozwalałaby na seryjny wybór lepszych rozwiązań. W momencie wykrycia sytuacji utknięcia w minimum lokalnym pozwalałaby na opuszczenie go przez wybór konfiguracji z końca rankingu.

Literatura

1. Janiak Adam. *Wybrane problemy i algorytmy szeregowania zadań i rozdziału zasobów.* Warszawa : kademicka Oficyna Wydawnicza PLJ, 1999. ISBN 83-7101-415-5.
2. Krzemiński Michał i Nowak Paweł. *Modyfikacja kosztów algorytmu johnsona do szeregowania zadań budowlanych.* Budownictwo i Inżynieria Środowiska. 2, 2011, strony 323-326.
3. Kwok Yu-kwong i Ahmad Ishfaq. *Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors.* ACM Computing Surveys. Grudzień 1999, Tom 31, 4, strony 407-471.
4. HU T. C. *Parallel sequencing and assembly.* Journal of Operations Research. 1961, Tom 9, 6, strony 841-848.
5. Ullman Jeffrey D. *NP-complete scheduling problems.* Journal of Computer and System Sciences. 1975, Tom 10, 3, strony 384-393 .
6. Coffman Edward G. i Graham Ronald Lewis. *Optimal scheduling for two-processor systems.* Acta Informatica. 1972, Tom 1, 3, strony 200-213.
7. Papadimitriou Christos H. i Yannakakis Mihalis. *Scheduling interval-ordered tasks.* SIAM Journal on Computing. 1979, 8, strony 405–409.
8. Fishburn Peter C. *Interval orders and interval graphs : a study of partially ordered sets.* New York : John Wiley and Sons, Inc., 1984. 9780471812845.
9. Mccreary C.L. , i inni. *A comparison of heuristics for scheduling DAGs on multiprocessors.* Cancun : ICPP 1994. International Conference on, 1994. strony 446 - 451. 0-8186-5602-6.
10. Goldberg David Edward. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Reading, Massachusetts : Addison-Wesley Professional, 1989. ISBN-13: 978-0201157673.
11. Michalewicz Zbigniew. *Algorytmy genetyczne + struktury danych = programy ewolucyjne.* Warszawa : Wydawnictwa Naukowo-Techniczne, 2003. ISBN 83-2042-881-5.
12. Boettcher Stefan i Percus Allon G. *Optimization with extremal dynamics.* Physical Review Letters. 86, 2001, strony 5211–5214.

13. Bak P. i K. Sneppen. *Punctuated equilibrium and criticality in a simple model of evolution.* Physical Review Letters. 71, 1993, Tom 24, strony 4083–4086.
14. Sousa F.L. i Ramos F.M. *Function optimization using extremal dynamics.* Rio de Janeiro : Proceedings of the 4th International Conference on Inverse Problems in Engineering, 2002.
15. Sousa F.L., i inni. *New stochastic algorithm for design optimization.* AIAA Journal. 2003, Tom 41, 9, strony 1808–1818.
16. Sousa F.L., Vlassov Valeri i Ramos F.M. *Generalized extremal optimization: An application in heat pipe design.* Applied Mathematical Modelling. 2004, Tom 28, 10, strony 911–931.
17. Switalski Piotr i Seredynski Franciszek. *Multiprocessor scheduling by generalized extremal optimization.* Journal of Scheduling. 2010, Tom 13, 5, strony 531–543.
18. El-Rewini Hesham i Lewis T.G. *Scheduling parallel program tasks onto arbitrary target machines.* Journal of Parallel and Distributed Computing. 1990, Tom 9, 2, strony 138–153.
19. Schwehm Markus i Walter Thomas. *Mapping and Scheduling by Genetic Algorithms.* Linz, Austria : Third Joint International Conference on Vector and Parallel Processing , 1994. strony 832-841. ISBN: 978-3-540-58430-8.
20. Kwok Yu-kwong i Ahmad Ishfaq. *Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors.* IEEE Transactions on Parallel and Distributed Systems. 1996, Tom 7, 5, strony 506-521.

Spis rysunków

Spis wykresów

Spis tabel