Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Wydział Informatyki, Elektroniki i Telekomunikacji

Katedra Informatyki

Rozprawa Doktorska

Algorytmy stadne w optymalizacji problemu przepływowego szeregowania zadań

Wiesław Popielarski

Promotor:
prof. dr hab. inż Bogusław Filipowicz

Kraków 2013

ukochanej żonie Elżbiecie za zachęty, cierpliwość i poświęcenie

Spis treści

1 Wprowadzenie **6**

1.1 Cele pracy 9

1.2 Organizacja pracy 10

1.3 Definicje 11

2 Modele szeregowania zadań dla wielu maszyn **14**

2.1 Ogólny model przetwarzania równoległego $Pm||C_{max}$ 15

2.2 Model $Fm||C_{max}$ 15

2.3 Model $Jm||C_{max}$ 20

2.4 Model $Om||C_{max}$ 23

2.5 Wprowadzenie do modeli stochastycznych 24

2.6 Model $P2||E(C_{max})$ 27

2.7 Model $Fm|prmu, p_{ij} = p_j|C_{max}$ 29

3 Wprowadzenie do algorytmów stadnych **30**

3.1 Zbieżność stochastyczna 31

3.2 Tempo osiągania zbieżności stochastycznej 33

3.3 Ogólny schemat algorytmu stadnego 36

3.4 Algorytm pszczeli 37

3.5 Algorytm kukułki 40

4 Określenie parametrów algorytmu stadnego dla problemu optymalizacji modelu *flow shop* **42**

4.1 Problem zbieżności 42

4.1.1 Sąsiedztwo stanu 43

4.1.2 Eksploatacja w algorytmie pszczelim 45
4.1.3 Eksploatacja w algorytmie kukułki 47
4.2 Problem próbkowania 49
4.2.1 Odległość stochastyczna i łańcuchy sprzężone 49
4.2.2 Wyznaczenie T dla sąsiedztwa stanu 51
4.3 Liczba iteracji 54
4.4 Zrównoleglanie algorytmu stadnego dla N procesorów 55
4.4.1 Przypadek szczególny $t_{ji} \geqslant t_{(j+1)(i-1)}$ 58

5 Realizacja algorytmów stadnych dla modelu $Fm|prmu|C_{max}$ 62
5.1 Implementacja 62
5.1.1 Środowisko uruchomieniowe 64
5.1.2 Struktura programu 64
5.1.3 Odpowiedzialności obiektów infrastrukturalnych 65
5.1.4 Obiekty i procedury w implementacji algorytmu pszczelego 68
5.1.5 Obiekty i procedury w implementacji algorytmu kukułki 74
5.1.6 Inicjalizowanie algorytmu listą zadań 78
5.2 Przykładowe wyniki 79
5.2.1 Ustawienia parametrów testów dla algorytmu pszczelego 80
5.2.2 Ustawienia parametrów testów dla algorytmu kukułki . 82
5.2.3 Ustawienia dla inicjalizacji wynikiem działania algorytmu *NEH* 83
5.2.4 Badanie zbieżności stochastycznej 84

6 Dyskusja wyników i dalsze kierunki badań 85
6.1 Obserwacje dotyczące równoległego poszukiwania optymalnego harmonogramu 87
6.2 Ogólne obserwacje dotyczące algorytmów inicjalizowanych uszeregowaniem losowym 88
6.2.1 Porównanie efektywności działania algorytmu pszczelego z algorytmem kukułki 94
6.3 Wpływ początkowej permutacji na wartość końcową 95

6.4 Zbieżność stochastyczna w badanych algorytmach 98
6.5 Podsumowanie 100

Streszczenie

W pracy zaproponowano nowatorskie podejście do analizy algorytmów stadnych na przykładach algorytmów: pszczelego oraz kukułki, w kontekście problemu szeregowania zadań dla modelu przepływowego (permutacyjny flow shop).

Obok wprowadzenia w modele szeregowania zadań z jednej oraz algorytmy stadne z drugiej strony, praca porusza bardzo ważną problematykę wyznaczania liczby iteracji w zależności od przyjętego sposobu definiowania sąsiedztwa w przestrzeniach kombinatorycznych.

Analizę przeprowadzono dla algorytmów zrównoleglonych zaimplementowanych w języku *Scala* w oparciu o testy zespołu Erica Taillarda.

Abstract

This thesis takes the reader on the fascinating journey through the world of task scheduling problems. First, it introduces the most common parallel flow models: flow shop, job shop and open shop - in their deterministic and stochastic versions. We focus here on proofs that they belong to NP-complete complexity class.

Then we move to the herd (known as 'boids' also) algorithms realm. This part presents the main requirements for every stochastic algorithm and describes in details two representatives of this algorithms family: bee and cuckoo algorithms.

Next, we deliberate about neighborhoods, the combinatorial space of their solution models, and how those models influence the objective function shape. We discuss the methods of computing iteration numbers for the above algorithms using Chernoff inequality and coupled Markov chains. The conclusions are then used in concrete implementations.
We have a practical implementation in Scala. The main idea is funded on the actors model (similar to the agents model) which allows for asynchronous, multithreaded computations. Scala provides strong support for distributed computations, and was thusly chosen as the implementation language.

Finally, we present the results of the application runs on Taillard's test for permutative parallel flow shop model.

Rozdział 1

Wprowadzenie

Problem szeregowania zadań jako odrębna grupa problemów algorytmicznych wywodzi się z analizy zagadnienia optymalnego przydziału zasobów, które z kolei pochodzi wprost z problemu efektywnego wykorzystania mocy produkcyjnych. Pierwsze metodologie szeregowania starano się opracować na początku XX wieku. Prekursorem był Henry Gantt, który pracując jako inżynier w Bethlehem Steel Corporation opracowywał dla brygadzistów dzienne harmonogramy prac. Obecnie metodologie te są wciąż używane i nazywane potocznie diagramami Gantta. Inną osobą zajmującą się optymalnym wykorzystaniem środków produkcyjnych był Polak, Karol Adamiecki. Podobnie jak H. Gantt swoje wyliczenia prowadził dla prac przeprowadzanych w walcowniach i stalowniach. Niestety ze względu na niewielką ilość samych publikacji oraz tłumaczeń na języki głównych krajów przemysłowych tamtego okresu (szczególnie angielskiego i niemieckiego, ponieważ nie należy zapominać, że już z początkiem I Wojny Światowej to właśnie USA i Niemcy były największymi gospodarkami ówczesnego świata), pozostał trochę zapomniany.

Kolejny impuls rozwoju metod szeregowania dał przełom lat 50 i 60, w których w badanie problemów kombinatorycznych tej klasy pewien udział miały służby logistyczne Armii Stanów Zjednoczonych. Wtedy też powstały prace dr. S. H. Johnsona oraz R. Grahama.

Lata 70te to prace o złożoności obliczeniowej klas problemów, w których

ponownie można wyróżnić R. Grahama za dowód, że ogólny problem Job-Shop należy do klasy złożoności *NP*.

Od lat 80tych główny wysiłek badaczy skierowany został na poszukiwanie efektywnych algorytmów stochastycznych oraz heurystyk i algorytmów aproksymacyjnych.

Rola zagadnień szeregowania zadań w obecnym świecie jest nie do przecenienia. Praktycznie w każdej dziedzinie życia można pośrednio lub bezpośrednio spotkać się z problemem, który sprowadza się do optymalizacji szeregowania zadań. Począwszy od systemów operacyjnych, poprzez ciągi technologiczne, organizację pracy na modelach biznesowych skończywszy.
Przykład pierwszy. Linia lotnicza chce zoptymalizować koszt utrzymania załóg swojej floty. W tym celu dla każdego członka załogi (pilotów, osób obsługi lotu), określa się tygodniowy harmonogram pracy, który musi być określony tak, aby każdy lot miał pełną obsługę. Dodatkowo czas pracy, dobowy i tygodniowy, każdej z osób nie może przekroczyć norm określonych w regulacjach prawnych. Z punktu widzenia teorii szeregowania zadań powyższy problem zalicza się do deterministycznego modelu ***Job Shop***.
Przykład drugi. Firma prowadzi sprzedaż z dostawą do drzwi klienta. Stąd też chce zoptymalizować globalny koszt dostawy zamówionych towarów. W tym celu dla każdego zlecenia określa się dwa harmonogramy. Pierwszy przeznaczony jest dla osób pracujących w magazynie, który optymalizuje czas dostarczenia poszczególnych pozycji zamówienia do punktu wydawania magazynu. Drugi przeznaczony jest dla służby logistycznej, który optymalizuje koszt dostarczenia zamówienia do drzwi klienta. Dodatkowo wprowadza się ograniczenie na całkowity czas realizacji zamówienia. Z kolei ten problem w teorii szeregowania zadań zalicza się do stochastycznego modelu ***Open Shop***.

Ze względu na to, że dla większości funkcji kosztu zarówno w modelach jedno- jak i wieloprocesorowych ogólne problemy należą do klasy *NP*, jak już wspomniano, współczesne kierunki zadań koncentrują się nad znajdywaniem heurystyk lub algorytmów aproksymacyjnych dla specyficznych, w danej domenie, modeli. Dzięki temu możliwe staje się opracowanie rozwiązań o akceptowalnym czasie obliczeń i ich błędzie. Niestety, takie podejście

powoduje, że zbudowanie ogólnego narzędzia do rozwiazywania szerokiego spektrum zagadnień jest praktycznie niemożliwe. Z drugiej strony, budując model matematyczny problemu, można posłużyć się narzędziami do programowania liniowego (w tym programowania całkowitoliczbowego), jednakże złożoność obliczeniowa w takim przypadku dla algorytmu *simpleks* wynosi $2^{O(\sqrt{m})}$, gdzie m jest liczbą ograniczeń (wynik z 1997 roku otrzymany przez Matouska, Welzla, Sharira i Kalaia). W przypadku algorymu *elipsoidalnego* zaproponowanego przez Leonida Chacziana (Leonid Khachiyan,1979) złożność wynosi $n + m + \phi$ (gdzie n liczba zmiennych, m liczba ograniczeń, natomiast $\phi = \max(a_{ij}, b_i, c_j)$). Ze względu na ϕ algorytm Chacziana ma złożoność pseudowielomianową i wciąż jest otwartą kwestią, czy jest silnie wielomianowy. Stąd też wynikła potrzeba znalezienia zupełnie innego podejścia do problemu szeregowania zadań, a jednym z obiecujących rezultatów jest cała grupa algorytmów naśladujących (inspirowanych przez) naturę, zarówno procesy fizyczne (jak w przypadku symulowanego wyżarzania) czy zachowania organizmów żywych (genetyczne, odzwierzęce), zaliczanych do grupy algorytmów stochastycznych. Spośród nich szczególnie algorytmy odzwierzęce charakteryzują się zadziwiającą prostotą, pomimo której są wydajnymi narzędziami do rozwiązywania problemów *NP*. Dodatkowo interesującymi cechami tej grupy algorytmów jest ich bardzo podobna struktura oraz łatwość adaptacji do szczególnego problemu (podobnie jak w przypadku programowania liniowego). Dlatego też są one tematem wielu (w tym ninejszej) prac.

Jest jeszcze jedna cecha algorytmów stochastycznych, która sprawia, że zwracają one na siebie uwagę. Tą cechą jest łatwość z jaką poddają się one zrównoleglaniu. Jest to o tyle istotne, że obecnie zwiększenie ilości operacji na jednostkę czasu wykonywanych przez procesor osiąga się raczej przez zastosowanie architektury wielordzeniowej niż proste zwiększenie częstości zegara. Stąd też algorytmy stochastyczne lepiej niż inne, może poza algorytmami typu *dziel i ograniczaj*, są w stanie wykorzystać rozproszone środowisko obliczeniowe (wiele procesorów wielordzeniowych). Nawet niezbyt duże klastry jednostek obliczeniowych (do 500 procesorów) są w stanie zmniejszyć czas obliczeń nawet o 3 rzędy wielkości. Z punktu widzenia teorii złożono-

ści obliczeniowej nie jest to na pewno spektakularne osiągnięcie, jednakże z inżynierskiej perspektywy jest to jakościowy skok. Wystarczy pomyśleć, że obliczenia, które wymagały poprzednio tysiąca godzin pracy (41 dni) od tej chwili są wykonywane w ciągu jednej godziny. Pozwala to w wielu przypadkach reagować na nieoczekiwane zmiany w procesie niemalże na bieżąco czyniąc z tych procesów systemy czasu rzeczywistego.

1.1 Cele pracy

Spośród algorytmów odzwierzęcych, jako jeden z pierwszych (początek lat 90tych) został zaproponowany algorytm mrówkowy [19]. Obecnie, obok algorytmu mrówkowego, bada się własności zachowania stadnego innych grup zwierząt jak świetliki, ptaki czy pszczoły. I właśnie algorytmy **pszczeli** oraz **kukułki** zostały wykorzystane w niniejszej pracy do znajdowania rozwiązania możliwie najbliższego optimum funkcji kosztu dla problemu szeregowania zadań. W pracy dokonano porównania efektywności znajdowania wartości optymalnej przed ich **zrównoleglone** wersje. W przypadku algorytmu **kukułki** nie są znane implementacje obliczeń równoległych dla problemu szeregowania zadań (w tym *flow shop*). Stąd też zaprezentowane autorskie rozwiązanie jest być może jednym z pierwszych w ogóle.

W niniejszej pracy zostaną jedynie poruszone zagadnienia szeregowania zadań dla modeli wieloprocesorowych, a więc takich, w których liczba jednostek przetwarzających jest większa od jedności. Zdecydowano się na taki wybór z dwóch zasadniczych powodów. Po pierwsze, modele jednoprocesorowe zostały już bardzo dobrze zanalizowane. Dla większości funkcji kosztu zostały znalezione reguły szeregowania, które dają rozwiązanie optymalne w czasie wielomianowym. Warto w tym miejscu wspomnieć, że pomimo iż model jednoprocesorowy jest relatywnie prosty, to istnieją takie funkcje kosztu, dla których znane rozwiązania należą do klasy *NP*. Jednakże dla większości z nich znaleziono heurystyki lub algorytmy aproksymacyjne dające rozsądne przybliżenia wartości optymalnej.

Z punktu widzenia obecnych systemów biznesowych i produkcyjnych to właśnie modele wieloprocesorowe są ich naturalnym sposobem opisu. Mode-

le jednoprocesorowe pozostają głównie domeną akademickich rozważaniach. Oczywiście, nie umniejsza to ich walorów poznawczych, gdyż jako uproszczenie pozwalają na analizę podstawowych własności zastosowanej funkcji kosztu. Modele jednoprocesorowe można traktować też z reguły jako kres dolny złożoności obliczeniowej dla algorytmu wyznaczającego wartość optymalną dla zadanej funkcji kosztu. Innymi słowy znalezienie rozwiązania optymalnego dla tej samej funkcji celu w modelu wieloprocesorowym jest obliczeniowo co najmniej tak trudne jak w modelu jednoprocesorowym.

W pracy pominięto zagadnienia wywłaszczania aktualnie wykonywanego zadania, ponieważ z reguły model taki jest prostszy w analizie i często udaje się znaleźć dla niego optymalne rozwiązanie wielomianowe [43]. Można powiedzieć, że modele bez wywłaszczania są w pewnym sensie generalizacją modeli z wywłaszczaniem. Sens ten sprowadza się do rezygnacji z ograniczenia wymuszającego nieprzerwane wykonywanie danego zadania na wybranym procesorze. W ten sposób z modelu bez wywłaszczania otrzymuje się model z wywłaszczaniem (*relaksacja* modelu). Jednakże sposób przeprowadzania dalszej analizy takiego modelu jest odmienny.

W pracy postawiono tezę, że **zagadnienie szeregowania zadań dla modelu przepływowego (*flow shop*) wielomaszynowego, będącego problemem należącym do klasy *NP*-zupełnej, jest podatne na efektywne zrównoleglenie jeżeli zastosuje się do jego rozwiązania algorytm stochastyczny, w szczególności stadny oraz że istnieje sposób badania efektywności takiego algorytmu.** Ostatecznym rezultatem pracy jest zaprezentowanie działającej wielowątkowej aplikacji.

1.2 Organizacja pracy

W pierwszej części pracy przedstawiono własności deterministycznych modeli szeregowania zadań począwszy od podstawowego modelu równoległego poprzez *flow shop* oraz *job shop* skónczywszy na *open shop*. Następnie scharakteryzowano wybrane modele stochastyczne i problemy z nimi związane.

W drugiej części pracy przybliżono pojęcia algorytmów pszczelego i kukułki

jako szczególnej klasy algorytmów stadnych i w ogólności stochastycznych. Trzecia i czwarta część pracy zawierają autorską analizę złożoności obliczeniowej algorytmów stochastycznych w oparciu o nierówność Chernoffa, procesy Markowa i algorytm Metropolisa oraz rozważania na temat zrównoleglenia algorytmu stadnego. Wyznaczona liczba kroków oraz wnioski z rozważań nad przetwarzaniem równoległym dla modelu *flow shop* zostały wykorzystane przy konstrukcji prezentowanego algorytmu.
W końcu ostatnia część zawiera opis implementacji algorytmu, wyniki doświadczeń oraz propozycje dalszych badań.

1.3 Definicje

Na wstępie należy zwrócić uwagę na cokolwiek dwuznaczną terminologię używaną w opisie rozważanych modeli wieloprocesorowych. W literaturze polskojęzycznej najczęściej przyjmuje się, że szeregowane są **zadania** rozumiane jako jednostki pracy. W pracach anglojęzycznych odpowiada im określenie **job**, które tłumaczone jest po prostu jako *praca*. Każda taka jednostka pracy może składać się z *podzadań* wykonywanych na poszczególnych procesorach. W terminologii anglojęzycznej odpowiada im pojęcie **task** tłumaczone jako *zadanie*. W dalszej części pracy używane będą słowa **zadanie** oraz **podzadanie**.

We wszystkich rozważanych modelach szeregowania przyjęto, że liczba oczekujących na zaszeregowanie zadań jak i liczba przetwarzających **procesorów**, nazywanych też **maszynami**, są skończone. Liczbę określającą ilość zadań opisuje się literą n, z kolei m określa ilość procesorów w modelu. Indeks j zazwyczaj odnosi się do zadania, natomiast indeks i do procesora. Stąd też para (i, j) odnosi się do pewnej charakterystyki procesora i w kontekście zadania j lub operacji wykonywanej na procesorze i na rzecz zadania j. Istotnym pojęciem z dziedziny szeregowania (charakterystyką procesora) jest **czas przetwarzania** p_{ij} zadania j na procesorze i. Jeżeli czas ten jest taki sam dla wszystkich procesorów w modelu to indeks i może być pominięty.

Problem szeregowania opisuje trójka **a**|**b**|**c**, gdzie **a** określa model środo-

wiska, **b** dostarcza informacji o charakterystyce przetwarzania, natomiast **c** reprezentuje funkcję celu (kosztu), dla której poszukiwane jest minimum. Możliwe modele środowiska wyszczególnione w niniejszej pracy:

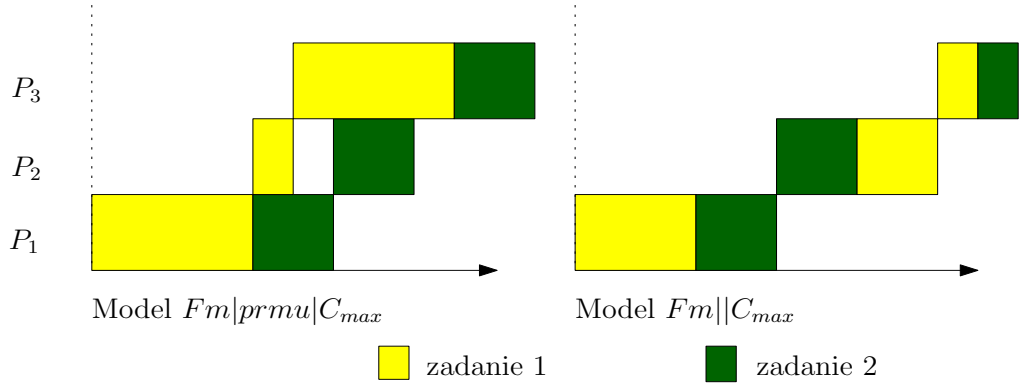
Identyczne procesory pracujące równolegle (*Pm*) Model zawiera m pracujących równolegle procesorów. Zadanie j może być przetworzone przez dowolny aktualnie wolny procesor. W przypadku tym przy definiowaniu p_{ij} indeks i może zostać pominięty.

Flow shop (*Fm*) Model m pracujących procesorów i n zadań. Każde zadanie j musi być przetworzone przez każdy z m procesorów. Wszystkie zadania muszą być przetworzone w tej samej, ustalonej a priori, kolejności. Przykładowo niech będzie dane m równe 3 procesory oraz n równe 2 zadania. Niech kolejność przetwarzania będzie określona jako **3-1-2**. Wówczas każde z n zadań musi być kolejno przetworzone przez procesor 3, następnie 2 i w końcu 1. Dodatkowo wprowadza się ograniczenie, że zadanie nie może być przetwarzane przez kolejny procesor, jeżeli jego przetwarzanie przez bieżący procesor nie zostało zakończone.

Job shop (*Jm*) Model ten jest uogólnieniem modelu **flow shop** z tym, że w odróżnieniu od niego każde zadanie ma swoją własną, definiowaną a priori, kolejność przetwarzania. Czyli modyfikując przykład powyżej, zadanie 1 mogłoby być przetworzone w kolejności **3-1-2**, natomiast zadanie 2 w kolejności **1-3-2**.

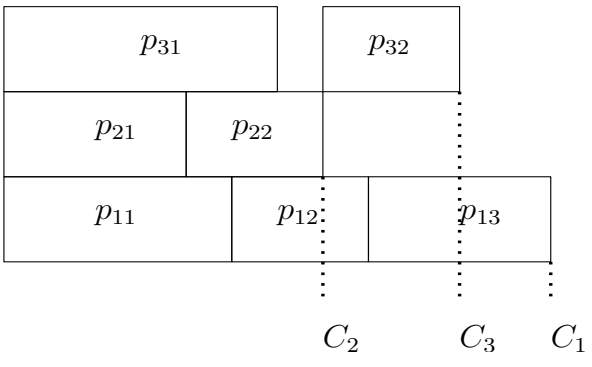
Open shop (*Om*) Model jest uogólnieniem modelu **Job shop** z tym, że kolejność przetwarzania danego zadania j nie jest ustalona a priori, lecz może być ustalana (obliczana) dynamicznie.

Jeżeli w rozważaniach pojawi się potrzeba określenia jakiejkolwiek charakterystyki przetwarzania dla któregokolwiek z powyższych modeli, to będzie to jedynie **permutacja**. **Permutacja** (*prmu*) jest ograniczeniem stosowanym w przypadku **flow shop**, które determinuje kolejność przetwarzania zadań przez dany procesor w porządku FIFO. Wynika stąd, że porządek przetwarzania przez pierwszą maszynę jest zachowany przez cały system. Pojęcie *prmu* to zostało zobrazowane na rysunku 1.1.



Rysunek 1.1: Permutacyjny *flow shop*. Po prawej stronie pokazano przykład niepermutacyjnego modelu. *Źródło: opracowanie własne.*

W pracy skupiono się na zagadnieniu optymalizacji tylko jednej funkcji kosztu, **maksymalnego czasu przetwarzania** (C_{max}), zdefiniowanej jako $max(C_1, \ldots, C_m)$. Ma to uzasadnienie w tym, że funkcja ta określa stopień wykorzystania procesorów podczas przetwarzania. Mała jej wartość oznacza wysoki poziom wykorzystania zbioru procesorów. Innymi słowy, minimalizacja C_{max} oznacza także minimalizację całkowitego czasu pracy jałowej procesorów w systemie (modelu). Definicja C_{max} została pokazana na rysunku 1.2.



Rysunek 1.2: Definicja C_{max} (w prezentowanym przypadku $C_{max} = \max(C_1, C_3, C_2) = C_1$). *Źródło: opracowanie własne.*

Rozdział 2

Modele szeregowania zadań dla wielu maszyn

Z biznesowego punktu widzenia modele wieloprocesorowe są modelami, które wierniej, a co za tym idzie, lepiej odzwierciedlają rzeczywistość. Pozwalają one nie tylko potwierdzać pewne empiryczne obserwacje lub intuicje, ale także przewidywać rozwiązanie lub przynajmniej określić, czy istnieje algorytm optymalizujący zadaną funkcję kosztu w danym modelu (który determinuje konfigurację procesorów i charakterystykę przetwarzania). Dlatego też każde nowe opracowanie któregokolwiek z modeli wieloprocesorowych jest wartościowe. Z drugiej strony warto zaznaczyć, że przetwarzanie w modelach wieloprocesorowych jest uogólnieniem przetwarzania w modelach jednoprocesorowych, stąd też, mimo, że ich analiza jest zdecydowanie trudniejsza, to otrzymane wyniki często również można zastosować w modelach jednoprocesorowych (jednakże nie zawsze jest to możliwe).

W dalszych częściach zostaną krótko przedstawione następujące deterministyczne modele: $Pm||C_{max}$, $Fm||C_{max}$, $Fm|prmu|C_{max}$, $Jm||C_{max}$ oraz $Om||C_{max}$. W ramach opisu przekonać się będzie można o tym, że algorytm, który optymalizowałby C_{max}, w przypadku każdego z powyższych modeli, ma złożoność ponadwielomianową (należy do klasy złożoności NP-zupełnej). W opisie nie będą przedstawione heurystyki ani algorytmy aproksymacyjne dające przybliżenia wartości optymalnych.

Przymiotnik *deterministyczne* przy słowie modele, oznacza, że czas przetwarzania zadania j przez maszynę i jest stały i znany a priori.

2.1 Ogólny model przetwarzania równoległego $Pm||C_{max}$

W modelu $Pm||C_{max}$ jest zadane m procesorów oraz n zadań. Każde zadanie musi być przetworzone co najwyżej raz na jednym z m procesorów (można też powiedzieć, że każde z zadań musi być przetworzone przez jeden z m procesorów z tym, że czas przetwarzania $p_{ij} = p_j \geqslant 0$) oraz czas przetwarzania danego zadania nie zależy od wybranego procesora, czyli $p_{ij} = p_j$ [13, 43]. Model ogólny jest podstawą do analizy własności szeregowania zadań m.in. w modułach szeregowania zadań w systemach operacyjnych.

Twierdzenie 1: Problem $P2||C_{max}$ należy do klasy NP-zupełnej

Dowód: Przez redukcję problemu podziału zbioru (PARTITION) do $P2||C_{max}$

Niech będą procesory *M1* oraz *M2*, oraz niech będzie dany zbiór n zadań $a_1, \ldots, a_n$. Niech wartość optymalna funkcji kosztu C_{max} wynosi $b = \frac{\sum_{j=1}^{n} a_j}{2}$ wówczas rozwiązanie istnieje wtedy i tylko wtedy, gdy istnieje taki podział zbioru zadań a_j na rozłączne podzbiory indeksów S_r i S_q, że $\sum_{r \in S_r} a_r = \sum_{q \in S_q} a_q = b$.
Ponieważ problem $Pm||C_{max}$ jest przynajmniej tak samo trudny jak $P2||C_{max}$, stąd wniosek, że problem $Pm||C_{max}$ należy również do klasy NP-zupełnej.
Z wniosku z *twierdzenia 1* wynika fakt, że jest mało prawdopodobne (co z kolei wynika z hipotezy, że $P \neq NP$), aby istniał algorytm, który znajdowałby wartość optymalną dla problemu $Pm||C_{max}$ w czasie wielomianowym.

2.2 Model $Fm||C_{max}$

W modelu *Flow shop* dana jest liczba n zadań i m procesorów. Każde zadanie musi być przetworzone co najwyżej raz przez każdy z procesorów

(można powiedzieć także, że każde zadanie musi być przetworzone dokładnie raz przez każdy z procesorów z tym, że czas przetwarzania $p_{ij} \geqslant 0$). Kolejność przetwarzania każdego z n zadań przez wszystkie m procesorów musi być taka sama [13, 43].

Ze względu na to, że $F2||C_{max}$ jest problemem, dla którego znany jest algorytm wielomianowy (dokładniej o złożoności $O(n \log n)$) odkryty przez Selmera Martina Johnsona w połowie lat 50tych, w dalszej części nie będzie on przedmiotem rozważań. Nasuwa się jednak pytanie, czy gdy zwiększona zostanie ilość procesorów do trzech problem $F3||C_{max}$ będzie należał do klasy P.

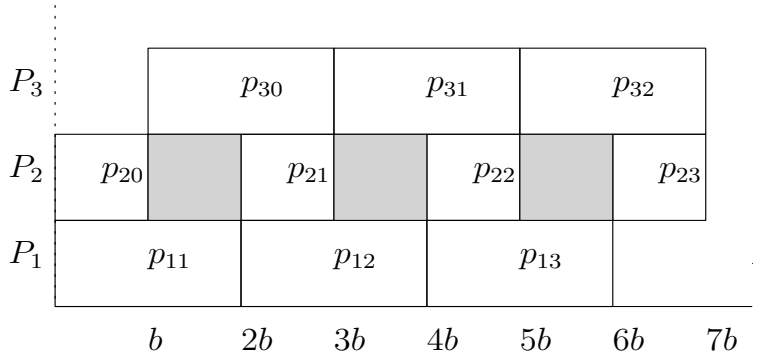
Twierdzenie 2: Problem $F3||C_{max}$ należy do klasy NP-zupełnej.

Dowód: Poprzez redukcję problemu 3-PARTITION do $F3||C_{max}$.

Problem 3-PARTITION jest problemem decyzyjnym podziału zbioru S, którego wszystkie elementy e_i spełniają warunek $\frac{b}{4} < e_i < \frac{b}{2}$ oraz $\sum_i e_i = kb$, na 3-elementowe podzbiory $S_1, S_2, \ldots, S_k$ takie, że suma elementów w każdym z tych podzbiorów jest równa pewnej zadanej wartości b.
Niech będzie dane $n = 4t + 1$ zadań oraz niech czasy przetwarzania poszczególnych zadań przez procesory będą określone poniższym układem równań:

$$\begin{array}{lll} p_{10} = 0, & p_{20} = b, & p_{30} = 2b \\ p_{1k} = 2b, & p_{2k} = b, & p_{3k} = 2b, 1 \leqslant k \leqslant t-1 \\ p_{1t} = 2b, & p_{2t} = b, & p_{3t} = 0 \\ p_{1(t+k)} = 0, & p_{2(t+k)} = a_k, & p_{3(t+k)} = 0, 1 \leqslant k \leqslant 3t \end{array}$$

co zostało też pokazane na rysunku 2.1. Niech problemem decyzyjnym będzie zbadanie, czy istnieje takie przypisanie zadań $a_1, a_2, \ldots, a_k, \ldots, a_{3t}$, że $C_{max} = (2t + 1)b$.
Warto zwrócić uwagę, że wszystkie a_k zadań odnoszą się do procesora drugiego. Jak można zauważyć, na procesorze 2 istnieje dokładnie t slotów o długości b między $t + 1$ zadań również o długości b. Stąd też problem decyzyjny będzie zweryfikowany pozytywnie (prawdziwy) wtedy i tylko wtedy,



Rysunek 2.1: Redukcja 3-PARTITION do $F3||C_{max}$ dla $t = 3$ oraz $C_{max} = 7b$. W wyszarzone miejsca, każde o długości b, przypisuje się pozostałe zadania o czasie przetwarzania a_k. *Źródło: opracowanie własne.*

gdy istnieje taki podział zadań $a_1, a_2, \ldots, a_k, \ldots, a_{3t}$ na t 3-elementowych podzbiorów, dla których suma elementów każdego z nich wynosi b.

Wniosek: Nie istnieje algorytm o złożoności wielomianowej, który rozwiązywałby ogólny problem *flow shop* $Fm||C_{max}$ (przy założeniu, że $NP \neq P$).

Wiedząc, że $Fm||C_{max}$ jest problemem należącym do klasy NP-zupełnej można by zapytać, czy pośród modeli bez wywłaszczania (bo tylko takie należą do klasy NP-zupełnej) istnieje (bądź istnieją) taki, dla którego algorytm optymalizacyjny należałby do klasy P. To pytanie jest o tyle istotne, że może dać wskazówkę do ewentualnej analizy pewnych warunków narzuconych na model (bądź modele) [5, 18, 38, 39]. Istnienie co najmniej jednego takiego modelu wiązałoby się z potrzebą wydzielenia pewnych podklas problemów, dla których nie byłoby potrzeby stosowania heurystyk bądź algorytmów aproksymacyjnych, aby znaleźć rozwiązanie bliskie optymalnemu. Z drugiej strony, podklasy te mogą być bardzo dobrymi testami szczególnie dla algorytmów stochastycznych, dla których analiza poprawności wyniku jest nietrywialna. Jako przykład niech posłuży problem *flow shop*, w którym czas przetwarzania j-tego zadania nie zależy od procesora, czyli $p_{ij} = p_j$. Dodatkowo zakładając, że kolejność przetwarzania dla każdego zadania na każdym procesorze musi być zachowana (tzn. że rozpatrywany jest jeden z modeli permutacyjnych $Fm|prmu, p_{ij} = p_j|C_{max}$) to okazuje się, że model taki jest problemem nale-

żącym do klasy P, a wartość C_{max} można wyznaczyć z poniższej zależności:

$$C_{max} = \sum_{j=1}^{n} p_j + \max(p_1, \ldots, p_n)(m-1)$$

Dowód: Z własności modelu wynika, że dla każdej permutacji zadań na którymkolwiek pierwszym procesorze całkowity czas przetwarzania wynosi:

$$\sum C_i = \sum_{j=1}^{n} p_j$$

Niech n zadań będzie uszeregowane zgodnie z rosnącą wartością p_j. Warto zwrócić uwagę, że takie uszeregowanie jest uszeregowaniem optymalnym (obok innych optymalnych harmonogramów), ponieważ na każdym kolejnym procesorze żadne zadanie nie będzie czekać z rozpoczęciem przetwarzania na zakończenie przetwarzania tego samego zadania na maszynie poprzedzającej. Wówczas czas rozpoczęcia zadania pierwszego na maszynie drugiej t_{21} równa się p_1, t_{22} będzie wynosić $p_1 + p_2$, $t_{23} = p_1 + p_2 + p_3$, w końcu $t_{2n} = \sum_{k=1}^{n} p_k$. Dla maszyny trzeciej czasy te będą wynosić odpowiednio:

$$\begin{aligned}
t_{31} &= p_1 + p_1, \\
t_{32} &= p_1 + p_2 + p_3 + p_3, \\
&\vdots \\
t_{3n} &= \sum_{k=1}^{n} p_k + p_n
\end{aligned}$$

W końcu czasy rozpoczęcia na maszynie m:

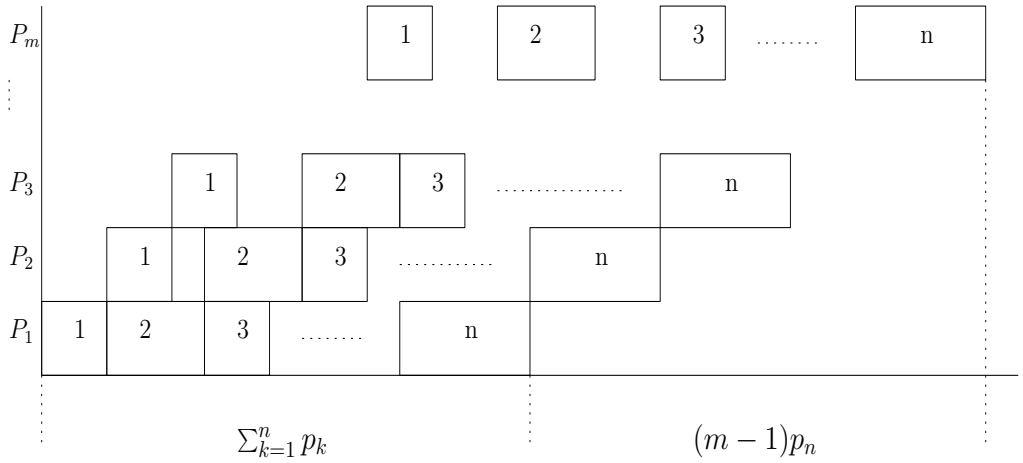
$$\begin{aligned}
t_{m1} &= p_1 + (m-2)p_1, \\
t_{m2} &= p_1 + p_2 + p_3 + (m-2)p_3, \\
&\vdots \\
t_{mn} &= \sum_{k=1}^{n} p_k + (m-2)p_n
\end{aligned}$$

Stąd też czas zakończenia ostatniego zadania n przez procesor m jest szukaną

wartością optymalną i wynosi:

$$C_{max} = \sum_{k=1}^{n} p_k + (m-2)p_n + p_n = \sum_{k=1}^{n} p_k + (m-1)p_n$$

gdzie $p_n = max(p_1, \ldots, p_n)$. Harmonogram ten został przedstawiony na rysunku 2.2.



Rysunek 2.2: Przykładowy optymalny harmonogram zadań dla $Fm|prmu, p_{ij} = p_j|C_{max}$ (według najkrótszego czasu przetwarzania). *Źródło: opracowanie własne.*

Można również znaleźć inne uszeregowanie zadań, które jest także optymalnym. Rozważmy zadania uszeregowane w następujący sposób:

$$p_1 \leqslant p_2 \leqslant \ldots \leqslant p_k \geqslant p_{k+1} \geqslant \ldots \geqslant p_n$$

Jak łatwo stwierdzić, żadne z zadań nie musi czekać na zakończenie przetwarzania przez procesor poprzedzający. Wówczas zakładając, że zadania prze-

twarzane są kolejno przez procesory $1, 2, ..., m$ otrzymujemy:

$$C_1 = p_1 + p_2 + \ldots + p_k + \ldots + p_n = \sum_{j=1}^{n} p_j$$

$$C_2 = p_1 + p_2 + \ldots + p_k + p_k + \ldots + p_n = \sum_{j=1}^{n} p_j + p_k$$

$$C_3 = p_1 + p_2 + \ldots + p_k + p_k + p_k + \ldots + p_n = \sum_{j=1}^{n} p_j + 2p_k$$

$$\vdots$$

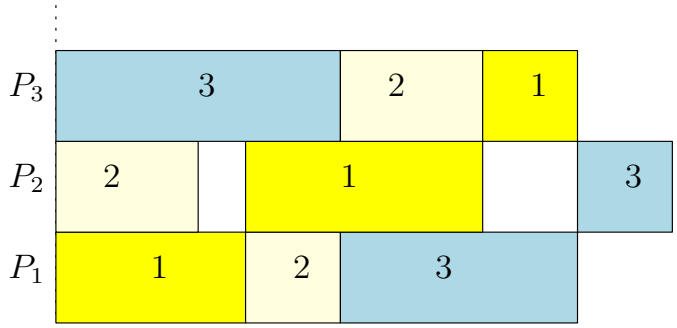
$$C_m = p_1 + p_2 + \ldots + p_k + \ldots + p_k + \ldots + p_n = \sum_{j=1}^{n} p_j + (m-1)p_k$$

Stąd też $C_m = C_{max}$ oraz $p_k = \max(p_1, \ldots, p_n)$.
Z powyższego wynika, że dla modelu $Fm|prmu, p_{ij} = p_j|C_{max}$ można bardzo łatwo znaleźć wartość optymalną i może posłużyć jako test dla badanej heurystyki.

2.3 Model $Jm||C_{max}$

Uogólnieniem modelu *flow shop* jest model ***job shop*** ($Jm||C_{max}$) [43]. W modelu *job shop* kolejność przetwarzania, mimo że jest określona a priori dla każdego zadania, może być różna dla każdych dwóch zadań (przykład na rysunku 2.3). *Job shop* jest uogólnieniem modelu *flow shop*. Z tego względu model ogólny $Jm||C_{max}$ jest przynajmniej tak samo trudny do rozwiązania jak $Fm||C_{max}$ (tzn. należy do klasy problemów NP-zupełnych).

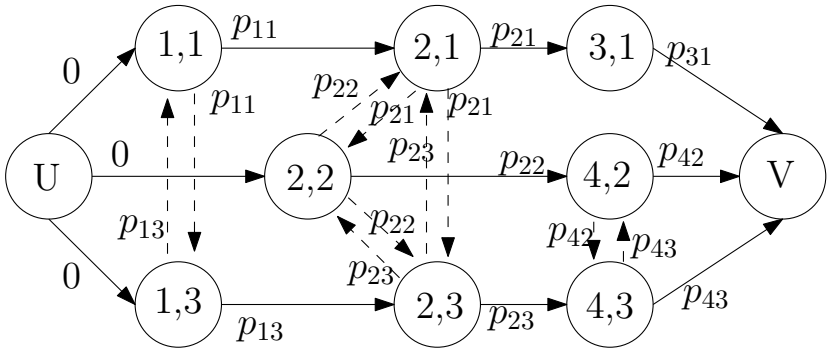
Ciekawym podejściem w znajdowaniu wartości optymalnej jest przedstawienie problemu $Jm||C_{max}$ w postaci grafu skierowanego. Poraz pierwszy taka formuła została zastosowana w problemie dziesięciu zadań postawionym przez H. Fishera i G.L. Thompsona w 1963 roku. W grafie (oznaczonym jako G) wyróżnia się dwa rodzaje łuków: zadań, które definiują zadaną kolejność przetwarzania zadania j przez poszczególne procesory oraz procesorów, które określają zależności pomiędzy zadaniami wykonywanymi na procesorze i. Łuki procesorów tworzą kliki, których ilość jest równa ilości procesorów m.



Rysunek 2.3: Przykładowy harmonogram zadań dla $J3||C_{max}$. Zadanie *1* ma harmonogram 1-2-3, zadanie *2* 2-1-3 natomiast *3* 3-1-2. *Źródło: opracowanie własne.*

Rysunek 2.4 przedstawia przykład takiego grafu dla n równemu 3 zadania oraz następujących zadanych kolejnościach przetwarzania na m równym 4 procesory (można założyć, że dla pominiętych procesorów $p_{ij} = 0$):

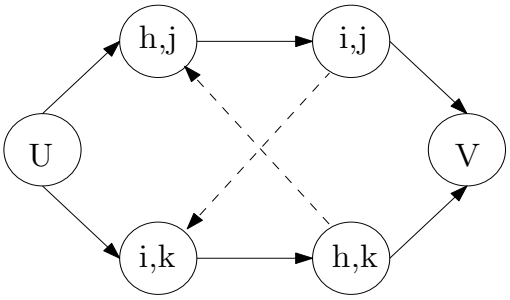
zadanie	kolejność przetwarzania
1	p_{11}, p_{21}, p_{31}
2	p_{22}, p_{42}
3	p_{13}, p_{23}, p_{43}



Rysunek 2.4: Przykładowy harmonogram zadań dla $J3||C_{max}$ przedstawiony jako graf. *Źródło: opracowanie własne.*

Rozważmy teraz warunki, jakie musi spełniać graf g powstały z rozważanego G składającego się z wszystkich łuków zadań i wybranych łuków procesorów (nazywanych też łukami alternatywnymi, ponieważ z każdej ich pary pomiędzy węzłami A_i i A_j tylko jeden jest wybrany do g), aby istniało przynajmniej

jedno rozwiązanie dopuszczalne.
Po pierwsze można sobie zadać pytanie, czy mogą w takim grafie istnieć cykle. Wobec tego niech (h, j) i (i, j) oznaczają dwie następujące po sobie operacje (na procesorach h oraz i) na zadaniu j. Niech (i, k) i (h, k) oznaczają dwie następujące po sobie operacje na zadaniu k. Oraz niech (i, j) poprzedza (i, k) oraz (h, k) poprzedza (i, k). Wówczas powstaje cykl (jak na rysunku 2.5), który z punktu widzenia szeregowania zadań oznacza sprzeczność. Stąd



Rysunek 2.5: Cykliczna zależność między zadaniami j oraz k przetwarzanymi przez procesory h oraz i [43].

też każdy graf g powstały z G, zawierający cykl, jest grafem zawierającym tylko rozwiązania niedopuszczalne. Natomiast każdy skierowany graf acykliczny (DAG) powstały z G zawiera rozwiązanie dopuszczalne, którym jest najdłuższa ścieżka w grafie g. Wagami łuków w g (wychodzących z węzłów) są wartości p_{ij}, natomiast w przypadku węzłów U i V wagi te wynoszą zero. Ponieważ dla DAG istnieje algorytm znajdujący najdłuższe ścieżki w czasie wielomianowym, wydawać by się mogło, że problem $Jm||C_{max}$ również należy do klasy P. Rozważmy teraz generację wszystkich grafów g z grafu G. Ilość wygenerowanych podgrafów z k-elementowej kliki to $\sum_{i=0}^{k} 2^i - 1$. Oczywiście, część z tych podgrafów może zostać odrzucona, ponieważ będą tworzyć cykle (również z łukami zadań), jednakże w przypadku pesymistycznym należy przejrzeć je wszystkie. Stąd też w najgorszym przypadku należy obliczyć najdłuższą ścieżkę dla $m \sum_{i=0}^{n} 2^i - 1$ grafów, co powoduje, że złożność rozważanego algorytmu optymalizującego $Jm||C_{max}$ jest przynajmniej wykładnicza.

Innym podejściem w próbie rozwiązania $Jm||C_{max}$ jest zapisanie zależ-

ności pomiędzy zadaniami w postaci następującego modelu programowania całkowitoliczbowego (MILP), w którym t_{ij} oznacza czas rozpoczęcia przetwarzania zadania j przez maszynę i:

$\min C_{max}$ z ograniczeniami:
$t_{kj} - t_{ij} \geqslant p_{ij}$, dla wszystkich (i, j) poprzedzających (k, j) (ograniczenie na łuki zadań),
$C_{max} - t_{ij} \geqslant p_{ij}$, dla wszystkich (i, j),
$t_{ij} \geqslant a_{ij}(p_{ik} + t_{ik})$, dla wszystkich (i, j) i (i, k) (ograniczenie na łuki pocesorów)
$t_{ik} \geqslant (1 - a_{ij})(p_{ij} + t_{ij})$,
$a_{ij} = \{0, 1\}$,
$t_{ij} \geqslant 0$

Ponieważ, jak wspomniano, jest to zadanie MILP, tak więc w ogólności ma złożoność *NP* [14, 38].

2.4 Model $Om||C_{max}$

Model ***open shop*** jest uogólnieniem modelu *job shop*, w którym kolejność każdego z n zadań przetwarzanego przez m maszyn nie jest określona a priori, czyli żadne zadanie nie ma predefiniowanego harmonogramu [43].
Ponieważ *open shop* jest uogólnieniem $Jm||C_{max}$, który dla przypadku $m > 2$ należy do klasy *NP*-zupełnej, można się spodziewać, że jest przynajmniej tak samo trudny jak $Jm||C_{max}$. Żeby się o tym przekonać, niech będzie rozważony przypadek $O3||C_{max}$.

Twierdzenie 3: $O3||C_{max}$ należy do klasy problemów *NP*-zupełnych

Dowód: Przez redukcję problemu podziału zbioru na dwa podzbiory, których suma elementów jest sobie równa (PARTITION).
Rozważmy $m = 3t + 1$ zadań, których czasy przetwarzania określone są

następująco:

$$\begin{aligned} p_{1j} = a_j, \quad p_{2j} = 0, \quad p_{3j} = 0; 1 \leqslant j \leqslant t, \\ p_{1j} = 0, \quad p_{2j} = a_j, \quad p_{3j} = 0; t+1 \leqslant j \leqslant 2t, \\ p_{1j} = 0, \quad p_{2j} = 0, \quad p_{3j} = a_j; 2t+1 \leqslant j \leqslant 3t \end{aligned}$$

oraz niech

$$p_{1(3t+1)} = p_{2(3t+1)} = p_{3(3t+1)} = b$$

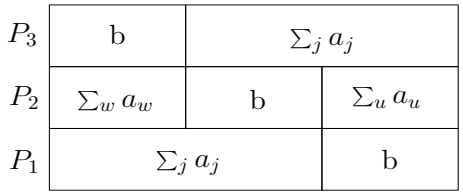
i

$$\sum_{j=1}^{t} a_j = \sum_{j=t+1}^{2t} a_j = \sum_{j=2t+1}^{3t} a_j = 2b.$$

Należy rozwiązać problem decyzyjny $C_{max} = 3b$.

Niech zadanie $(3t + 1)$ jest przetwarzane kolejno przez maszyny 1, 2 i 3 (rysunek 2.6). Wówczas, aby $C_{max} = 3b$ musi istnieć taki podział zbioru S elementów a_j dla $t + 1 \leqslant j \leqslant 2t$ na rozłączne podzbiory W oraz U, żeby $\sum_w a_w = \sum_u a_u = b$, gdzie $\{au\} \cup \{aw\} = S$.

Wniosek: Jeżeli każdy problem $Om||C_{max}$ dla $m > 3$ jest przynajmniej tak samo trudny jak $O3||C_{max}$, to $Om||C_{max}$ należy również do klasy *NP*.



Rysunek 2.6: Rozważany harmonogram dla $O3||C_{max}$. *Źródło: opracowanie własne.*

2.5 Wprowadzenie do modeli stochastycznych

Podstawowe deterministyczne modele opisujące harmonogramowanie zadań już dla trzech procesorów są problemami o złożoności należącej do klasy *NP* [13, 43]. Stąd też nie należy się spodziewać, że będzie istniał algorytm,

który w czasie wielomianowym byłby w stanie wyznaczyć wartość optymalną (zakładając, że $P \neq NP$). Jest to mocna przesłanka, aby próbować znaleźć algorytm (algorytmy), który w rozsądnym czasie byłby zdolny wyznaczyć wartość bliską optymalnej przez sięgnięcie do wersji stochastycznych przedstawionych modeli.

Do opisu modeli stochastycznych używa się tej samej notacji jak w przypadku modeli deterministycznych. Jedyną znaczącą różnicą jest, jak to już zostało wyżej wspomniane, czas przetwarzania zadania j na maszynie i. Ponieważ wartość ta nie jest liczbą a zmienną losową, stąd też oznacza się ją jako $\mathbf{X_{ij}}$ ($\mathbf{P}$ jest z reguły zarezerwowane dla prawdopodobieństwa).

Przyjmijmy, że dla czasu dyskretnego prawdopodobieństwo zakończenia przetwarzania zadania j przez maszynę i w każdej chwili jest takie samo i wynosi p oraz jest tylko jedno takie zdarzenie (ponieważ przetwarzanie można zakończyć dokładnie raz), stąd prawdopodobieństwo, że zadanie j zakończy się dokładnie w chwili t wyrazić można zależnością:

$$\mathbf{P}(X_{ij} = t) = (1-p)^{t-1}p.$$

Stąd też X_{ij} w dziedzinie czasu dyskretnego można opisać za pomocą rozkładu geometrycznego [25, 28].
Niech teraz $p = \frac{\lambda}{k}$, gdzie k oznacza ilość doświadczeń w jednostce czasu. Wówczas prawdopobieństwo zakończenia przetwarzania zadania j w dowolnym momencie t-tej jednostki czasu będzie wynosić:

$$\mathbf{P}(t) = \frac{\lambda}{k} + (1-\frac{\lambda}{k})\frac{\lambda}{k} + (1-\frac{\lambda}{k})^2\frac{\lambda}{k} + \ldots + (1-\frac{\lambda}{k})^k\frac{\lambda}{k} = \frac{\lambda}{k}\sum_{i=0}^{k}(1-\frac{\lambda}{k})^i,$$

stąd

$$\mathbf{P}(t) = \frac{\lambda}{k}\frac{1-(1-\frac{\lambda}{k})^k}{1-(1-\frac{\lambda}{k})} = 1-(1-\frac{\lambda}{k})^k,$$

mając na uwadze, że w jednostkach czasu poprzedzających t zadanie j nie zostało zakończone otrzymujemy poniższą zależność:

$$\mathbf{P}(X_{ij} = t) = (1-(1-(1-\frac{\lambda}{k})^k))^{t-1}(1-(1-\frac{\lambda}{k})^k) = (1-\frac{\lambda}{k})^{k(t-1)}(1-(1-\frac{\lambda}{k})^k).$$

Jeżeli teraz ilość doświadczeń w jednostce czasu będzie zwiększana do nieskończoności wówczas:

$$\mathbf{P}(X_{ij} = t) = \lim_{k \to +\infty} [(1 - \frac{\lambda}{k})^{k(t-1)}(1 - (1 - \frac{\lambda}{k})^{k})] = e^{-\lambda(t-1)}(1 - e^{-\lambda})$$

$$\mathbf{P}(X_{ij} = t) = e^{-\lambda(t-1)} - e^{-\lambda t} = \int_{t-1}^{t} \lambda e^{-\lambda x}\, dx,$$

w którym wyrażenie $\lambda e^{-\lambda x}$ jest rozkładem wykładniczym. Stąd też w dziedzinie czasu ciągłego X_{ij} można opisać za pomocą rozkładu wykładniczego.

Podsumowując, do opisu X_{ij} w dziedzinie czasu dyskretnego będzie użyty rozkład geometryczny, a w przypadku czasu ciągłego odpowiadający mu rozkład wykładniczy.
X_{ij} można opisać również innymi rozkładami. Wiąże się to jednak z dwoma problemami. Po pierwsze należałoby znać dokładniej proces przetwarzania, ponieważ do tej pory zakładano, że przetwarzanie może zakończyć się w dowolnym momencie z takim samym prawdopodobieństwem. Po drugie, analiza takich zmiennych losowych jest dużo trudniejsza, dlatego w dalszej części pracy będą one pominięte.

Sposób wyznaczania C_{max} różni się w porównaniu do modeli deterministycznych. Przede wszystkim, ze względu na to, że czas przetwarzania X_{ij} jest zmienną losową, poszukiwana wartość optymalna jest wartością oczekiwaną. Jako przykład niech będzie rozważony model $P2||C_{max}$ z dwoma zadaniami, których czas przetwarzania wynosi $p_1 = p_2$ równy 1. Wówczas dla modelu deterministycznego $C_{max} = \min(C_1, C_2) = 1$, natomiast dla modelu stochastycznego należy rozważyć cztery przypadki, których prawdopodobieństwa są sobie równe i wynoszą $p = 0.25$:

- zadania 1 i 2 przydzielone zostały do pierwszego procesora,
- zadania 1 i 2 przydzielone zostały do drugiego procesora,
- zadanie 1 zostało przydzielone do pierwszego procesora natomiast 2 do drugiego,
- zadanie 2 zostało przydzielone do pierwszego procesora natomiast 1 do drugiego.

Zakładając, że X_{ij} są określone przez rozkład wykładniczy o wartości $\lambda = 1$, wartość oczekiwaną $E(C_{max})$ wylicza się następująco:

$$E(C_{max}) = p(E(X_{11}X_{12})) + p(E(X_{21}X_{22})) + p(\max(E(X_{11}), E(X_{22}))) \\ + p(\max(E(X_{21}), E(X_{12}))).$$

Aby wyznaczyć $E(X_{11}X_{12}))$ należy obliczyć splot gęstości prawdopodobieństw zmiennych X_{11} oraz X_{12}, stąd

$$f(X_{11}X_{12}) = f(X_{11}) * f(X_{12}) = \int_0^t e^{-\tau} e^{-(t-\tau)}\, d\tau = te^{-t}.$$

Wartość oczekiwana zmiennej $X_{11}X_{12}$ wynosi:

$$E(X_{11}X_{12}) = \int_0^\infty t^2 e^{-t}\, dt = 2$$

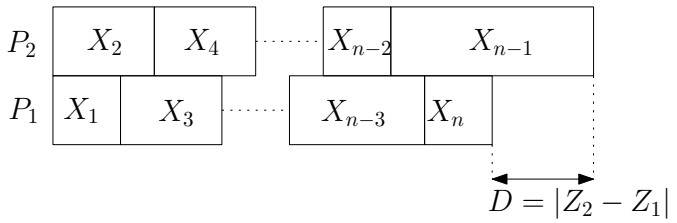
i jest taka sama dla $X_{21}X_{22}$, ponieważ ich funkcje gęstości są takie same. Podstawiając można ostateczne wyznaczyć $E(C_{max})$, która wynosi 1.5.

Jak można się przekonać wartości C_{max} oraz $E(C_{max})$ są nie tylko rozbieżne, ale dodatkowo wyznacza się w zupełnie inny sposób. W przypadku stochastycznym należy brać pod uwagę funkcje gęstości prawdopodobieństw i ich splotów, co znacząco utrudnia obliczenia oraz dowodzenie własności. Na marginesie warto wspomnieć, że w ogólności $E(C_{max})$ może zawierać się przedziale $1 \leqslant E(C_{max}) \leqslant 2$ i zależy to od funkcji gęstości zmiennych losowych X_{ij} ($E(C_{max}) = 1$ odpowiada przypadkowi deterministycznemu).

2.6 Model $P2||E(C_{max})$

Niech będzie dane n zadań $X_1, X_2, \ldots, X_n$, z których każde jest zmienną losową o rozkładzie wykładniczym z λ_j (gdzie $1 \leqslant j \leqslant n$). Niech będą one przypisane do poszczególnych maszyn w taki sposób, że zadanie $j + 1$ będzie wykonywane przez aktualnie wolną maszynę, co zostało zilustrowane na rysunku 2.7. Wówczas optymalny harmonogram będzie znaleziony tylko wtedy, gdy wartość oczekiwana zmiennej losowej D będącej różnicą zmiennych losowych Z_1 i Z_2 takich, że $Z_1 = \sum_k X_{k(1)}$ (gdzie $X_{k(1)}$ są zadaniami wyko-

nywanymi na maszynie pierwszej) i $Z_2 = \sum_k X_{k(2)}$ (gdzie $X_{k(2)}$ są zadaniami wykonywanymi na maszynie drugiej) będzie minimalna [43].



Rysunek 2.7: Harmonogram dla $P2||E(C_{max})$.

Można zastanowić się, jak zachowuje się wartość oczekiwana zmiennej losowej D w zależności od kolejności X_j. Otóż okazuje się, że dla dowolnie wybranej zmiennej X_0 z λ_0 zachodzi:

$$E(D(X_0, X_1, X_2, \ldots, X_n)) \leqslant E(D(X_0, X_2, X_1, \ldots, X_n))$$

wtedy, gdy wartość

$$\lambda_1 = \min(\lambda_1, \lambda_2, \ldots, \lambda_n).$$

Stąd też aby znaleźć optymalny harmonogram wystarczy uszeregować je zgodnie z najdłuższym oczekiwanym czasem przetwarzania. Jest to o tyle zaskakujące, że model deterministyczny $P2||C_{max}$ należy do klasy problemów *NP*.

Niestety, trudno przenieść rezultat powyższej analizy na uogólniony model $Pm||C_{max}$ ze względu na złożone zależności pomiędzy zmiennymi losowymi nawet dla dość prostego rozkładu jakim jest rozkład wykładniczy. Stwarza to trudności z zastosowaniem podejścia stochastycznego w modelowaniu rzeczywistych procesów, w których bardziej naturalne jest posługiwanie się zmiennymi losowymi a nie arbitralnie wybranymi wartościami. Dlatego też w części poświęconej zastosowaniu algorytmów stadnych w harmonogramowaniu zadań, pomimo obiecującego wyniku dla $P2||C_{max}$, będą rozpatrywane tylko modele deterministyczne.

2.7 Model $Fm|prmu, p_{ij} = p_j|C_{max}$

Ogólne modele $Fm||C_{max}$, $Jm||C_{max}$ czy $Om||C_{max}$ w wersji stochastycznej są przynajmniej tak samo trudne w analizie jak $Pm||C_{max}$, dla którego nie znaleziono żadnego wielomianowego algorytmu, ani też nie wykazano, że problem takiego algorytmu nie posiada. To z kolei powoduje, że modele typu *shop* podobnie jak $Pm||C_{max}$, nie mają zastosowania w analizie rzeczywistych procesów. Jednym z niewielu modeli tej klasy, dla którego udało się znaleźć rozwiązanie ze względu na C_{max} jest $Fm|prmu, p_{ij} = p_j|C_{max}$ [43]. Wartość oczekiwana C_{max} w tym przypadku wynosi:

$$E(C_{max}) = \sum_{j=1}^{n} \frac{1}{\lambda_j} + (m-1)\frac{1}{\lambda_k},$$

a sposób w jaki zadania są harmonogramowane jest dokładnie taki sam jak w przypadku deterministycznym, czyli wartości $\frac{1}{\lambda_j}$ powinny spełniać zależność:

$$\frac{1}{\lambda_1} \leqslant \frac{1}{\lambda_2} \leqslant \ldots \leqslant \frac{1}{\lambda_k} \geqslant \ldots \geqslant \frac{1}{\lambda_{n-1}} \geqslant \frac{1}{\lambda_n}.$$

Stąd też wniosek, że $Fm|prmu, p_{ij} = p_j|C_{max}$ może być użyty jako rodzaj testu dla permutacyjnego problemu ogólnego.

Rozdział 3

Wprowadzenie do algorytmów stadnych

Algorytmy stadne (z ang. boids algorithms) należą do grupy algorytmów stochastycznych inspirowanych przez naturę. Inspiracja ta motywowana jest obserwacjami, że niektóre problemy optymalizacyjne zostały rozwiązane przez strategie wykorzystywane głównie przez organizmy żywe lub grupy (kolonie) organizmów. Najbardziej znanym z takich rozwiązań jest struktura pszczelego plastra, która minimalizuje ilość materiału potrzebnego do jego budowy. Jest to o tyle zaskakujące, że trudno oczekiwać od pszczół jakichkolwiek zdolności matematycznych, a jednak natura wykształciła w nich taką umiejętność. Co ciekawe, tylko niektóre gatunki rodziny Apis budują plastry w optymalny sposób. Przykładowo plastry trzmieli nie są już tak regularne. Innym aspektem jest analiza zachowań grup (kolonii) organizmów. W przypadku niektórych gatunków owadów eusocjalnych (mrówki, pszczoły) lub parasocjalnych (karaczany) natura wykształciła pewne heurystyki pozwalające na efektywne wyznaczanie drogi do źródła pokarmu (w przypadku pszczół - pożytku). Algorytmy wzorujące się na tego typu zachowaniach grupowych nazywane są algorytmami stadnymi, a ich badania rozpoczęły się z początkiem lat 90-tych od analizy kolonii mrówek przez Marco Dorigo. Obecnie opracowuje się bardziej złożone modele zachowań mrówek, pszczół, karaczanów, świetlików a nawet zwierząt wyższych jak ptaki czy ssaki. Jednakże w

przypadku tych ostatnich modelowanie zachowań społecznych jest wyjątkowo trudne a uzyskiwane rezultaty problematyczne.

Oczywiście poszukiwania takich rozwiązań nie ograniczają się tylko do świata materii ożywionej. Pewne procesy fizyczne również mogą być wykorzystane w algorytmach optymalizacyjnych. W tym przypadku bardzo znanym przykładem jest symulowane wyżarzanie wykorzystujące analogię do procesu osiągania stanu równowagi cieplnej w taki sposób, aby wartość energii całego układu cząstek była globalnie najmniejsza. Wiele nadziei wiązano również z różnego rodzaju algorytmami ewolucyjnymi, które u swych początków wzorowały się na procesach ewolucji (obecnie panuje konsensus, że algorytmy te są jedynie daleką analogią tych procesów) i w których rozwiązanie optymalne powinno być odkryte przez naturę reprezentowaną przez funkcje przystosowania, celu oraz wybrany arbitralnie zasięg mutacji.
Tym, co łączy każde z powyższych podejść to implementacja pewnego wybranego mechanizmu łamiącego lokalność przeszukiwania przestrzeni rozwiązań w oparciu o prawdopodobieństwo. W przypadku algorytmów stadnych jest to funkcja werbunkowa, w przypadku algorytmów ewolucyjnych to operator mutacji, natomiast w symulowanym wyżarzaniu funkcja ta jest złożeniem funkcji sąsiedztwa, tranzycji oraz akceptacji lokalnego rozwiązania. Właśnie ze względu na losowość wszystkie one zalicza się do grupy algorytmów stochastycznych.

3.1 Zbieżność stochastyczna

Każdy poprawnie skonstruowany algorytm stochastyczny powinien charakteryzować się przynajmniej jedną z poniższych zbieżności stochastycznych [13, 25].

Najmocniejszą zbieżnością określającą relację między wyznaczoną wartością a wartością optymalną to:

$$P\{\lim_{n\to\infty} a_n = opt\} = 1,$$

czyli, że jest prawie pewne, że dla odpowiednio długiego czasu działania algo-

rytmu znaleziona wartość a_n jest wartością optymalną *opt*. Warto zauważyć, że zbieżnością tego rodzaju charakteryzuje się najprostszy z wszystkich algorytmów stochastycznych - algorytm błądzący (random walk).

Powyższą zbieżność można osłabić przez założenie, że istnieje pewna wartość $\epsilon > 0$, dla której

$$\lim_{n\to\infty} P\{|a_n - opt| > \epsilon\} = 0,$$

z czego również wynika, że

$$\sum_{n=1}^{\infty} P\{|a_n - opt| > \epsilon\} < \infty.$$

Interesującym faktem jest, że algorytm, którego wyznaczone wartości zbiegałyby w przedstawiony sposób, przypomina deterministyczny algorytm ϵ-aproksymacyjny (dla którego $\epsilon \geqslant \frac{|a_n - opt|}{opt}$), w tym sensie, że oba wyznaczają kres górny błędu (odpowiednio bezwzględnego lub względnego) [18].

Najsłabszą zbieżnością, która może zostać do określenia zależności pomiędzy a_n a *opt* jest zbieżność względem wartości oczekiwanej zdefiniowanej jako

$$\lim_{n\to\infty} |E(a_n) - opt| = 0,$$

gdzie $E(a_n)$ jest wartością oczekiwaną ciągu prób. Zbieżność ta jest o tyle interesująca, że nie wymaga się od algorytmu znalezienia dokładnej wartości, ale wystarczy, aby sąsiedztwo wartości optymalnej było dobrze zbadane. Z drugiej strony dobre określenie sąsiedztwa szczególnie w problemach kombinatorycznych jest z reguły zadaniem nietrywialnym. Dlatego też chociaż powyższa zbieżność jest najsłabsza, to może się okazać, że zbudowanie algorytmu o takiej zbieżności jest trudniejsze niż w pozostałych przypadkach (przykładowo wspomniany algorytm błądzący mimo, że jest najprostszy, charakteryzuje się najsilniejszą zbieżnością).

Dowodzenie zbieżności z reguły jest nietrywialne chociaż można posłużyć się pewnym schematem. Przede wszystkim należy zapewnić własność, że wybrany sposób wyznacznia prawdopodobieństw pozwala na osiągnięcie wszystkich możliwych stanów w układzie. By osiągnąć ten cel, w analizie

można posłużyć się procesem markowowskim. Jeżeli zostanie wykazane, że odpowiadający sposobowi wyznaczania prawdopodobieństw proces Markowa jest nieredukowalny, wówczas mamy pewność, że przyjęty sposób eksploruje całą badaną przestrzeń rozwiązań. Co więcej, oznacza to również, że algorytm jest algorytmem nieobciążonym tzn. że potencjalnie znalezione rozwiązanie jest niezależne od przyjętego początkowego przeszukiwanego podzbioru zbioru rozwiązań dopuszczalnych (czyli od danych początkowych). *Potencjał* oznacza w tym przypadku odpowiednio długi czas działania algorytmu tak, aby zbieżność stochastyczna była możliwa do zaobserwowania [28, 32, 34].

3.2 Tempo osiągania zbieżności stochastycznej

Obok dowodu zbieżności algorytmu ważne jest, żeby następowała ona w rozsądnym (wielomianowym) czasie. Wykazanie, że rzeczywiście zbieżność następuje w czasie wielomianowym jest z reguły trudniejsze od wykazania zbieżności jako takiej [34]. Pomocne będzie następujące twierdzenie:

Twierdzenie 4: Niech $X_1, X_2, \ldots, X_m$ będą niezależnymi zmiennymi losowymi o jednakowym rozkładzie i niech $\mu = E(X_i)$. Jeżeli $m \geqslant 3ln(\frac{2}{\delta})/(\epsilon^2\mu)$, to

$$P\left(\left|\frac{1}{m}\sum_{i=1}^{m} X_i - \mu\right| \geqslant \epsilon\mu\right) \leqslant \delta,$$

co oznacza, że m próbek zapewnia (ϵ, δ)-aproksymację wartości oczekiwanej μ.

Dowód: Niech dane będą niezależne zmienne losowe $X_1, X_2, \ldots, X_m$ oraz niech $X = \sum_{i=1}^{m} X_i$ i niech $\mu = E(X)$ wówczas z nierówności Chernoffa

$$P(|X - \mu| \geqslant \epsilon\mu) \leqslant 2e^{-\mu\epsilon^2/3}.$$

Ponieważ $\mu = E(X) = \sum_{i=1}^{m} E(X_i)$ i uwzględniając fakt, że zmienne X_i mają ten sam rozkład o wartości oczekiwanej $E(X_i) = \mu'$ wówczas $\mu = m\mu'$, stąd

podstawiając:

$$P(|X-\mu| \geqslant \epsilon\mu) = P(|X-m\mu'| \geqslant \epsilon m\mu') \leqslant 2e^{-m\mu'\epsilon^2/3},$$

co z kolei można zapisać w postaci:

$$P\left(\left|\frac{1}{m}\sum_{i=1}^{m} X_i - \mu'\right| \geqslant \epsilon\mu'\right) \leqslant 2e^{-m\mu'\epsilon^2/3} \leqslant \delta.$$

Przekształcając $2e^{-m\mu'\epsilon^2/3} \leqslant \delta$ tak, aby znaleźć m otrzymujemy

$$m \geqslant 3ln(\frac{2}{\delta})/(\epsilon^2\mu')$$

co kończy dowód.

Pierwszą rzeczą, na którą należy zwrócić uwagę, jest założenie dotyczące takich samych rozkładów zmiennych losowych. Wydaje się ono być zbyt restrykcyjne, jednak z drugiej strony pozwala na dość dokładne oszacowanie ilości próbek potrzebnych do wyznaczenia wartości różniącej się o nie więcej niż ϵ z prawdopodobieństwem nie większym od δ od rzeczywistej wartości optymalnej. W przeciwnym wypadku jedynymi dostępnymi narzędziami byłyby nierówności Czebyszewa i Markowa, których oszacowania charakteryzują się dużo mniejszą dokładnością. Zachodzi jednak pytanie, jak powinien być skonstruowany sposób losowania próbki, aby każde z losowań opisane mogło zostać przez dostatecznie bliskie sobie (lub takie same) funkcje rozkładu. Do tego celu stosuje się metodę *Monte Carlo* opartą o łańcuchy Markowa, która zostanie przybliżona w dalszej części pracy przy okazji analizy zbieżności algorytmu stadnego zastosowanego do optymalizacji modelu przepływowego *flow shop*.

W tym momencie uwaga zostanie skupiona na drugim ważnym wniosku, wynikającym z wyrażenia na ilość m próbek. Jak można się przekonać potrzebna ilość losowań uzależniona jest od funkcji wartości oczekiwanej μ. Jako ilustracja niech posłuży problem zliczania wartościowań spełniających formułę (klauzulę) boolowską w dysjunktywnej postaci normalnej [34]. Sam problem zliczania należy do klasy *NP*-zupełnej, co jest o tyle korzystne, że może dać

ogólną wskazówkę do rozwiązywania innych problemów tej klasy. Jak zostało to wykazane powyżej, każdy ogólny model przepływowy (*flow shop*, *job shop* oraz *open shop*) również należy do klasy problemów *NP*-zupełnych.
Niech będzie dana dysjunktywna postać normalna (DPN) o t klauzulach i niech w każdej z nich będzie użytych k_i wyrażeń z n. Przykładowo dla poniższej DPN t wynosi cztery $k_1 = k_2 = 3$, $k_3 = 2$ a $k_4 = 4$ oraz $n = 4$,

$$(x_1 \wedge \bar{x}_2 \wedge \bar{x}_3) \vee (\bar{x}_1 \wedge x_2 \wedge \bar{x}_3) \vee (x_1 \wedge x_2) \vee (x_1 \wedge \bar{x}_2 \wedge x_3 \wedge x_4).$$

Niech c będzie szukaną ilością wartościowań oraz niech dane wartościowanie będzie losowane ze zbioru wszystkich wartościowań o liczności 2^n. Niech zmienna losowa $X_i = 1$, jeżeli wylosowane wartościowanie spełnia przynajmniej jedną klauzulę oraz $X_i = 0$ w przeciwnym wypadku. Wówczas wartość oczekiwana $X = \sum_{i=1}^{m} X_i$ wynosi:

$$E(X) = E(\sum_{i=1}^{m} X_i) = E(\sum_{i=1}^{m} E(X_i)) = E(\sum_{i=1}^{m} \frac{c}{2^n}) = m\frac{c}{2^n}.$$

Stąd też liczność próby m wynosi:

$$m \geqslant 2^n 3ln(\frac{2}{\delta})/(c\epsilon^2).$$

Ponieważ nie można powiedzieć niczego pewnego o wartości c, można w szczególności przyjąć, że jest ona wielomianowa względem n. Co z kolei implikuje, że m zależy wykładniczo od wartości n dla przyjętego sposobu znajdowania wartościowania. Jak można zatem inaczej podejść do problemu?
Pierwszą rzeczą jaką można zauważyć jest to, że znany jest zbiór wartościowań spełniających daną DPN, którego liczność wynosi $|\Omega| = \sum_{i=1}^{t} 2^{n-k_i}$, gdzie 2^{n-k_i} jest licznością zbioru wartościowań spełniających i-tą formułę. Stąd też prawdopodobieństwo, że wylosowane wartościowanie spełnia j-tą formułę wynosi

$$p_j = 2^{n-k_j}/(\sum_{i=1}^{t} 2^{n-k_i}).$$

Może się zdarzyć, że wylosowane wartościowanie spełnia również inną klauzulę, stąd też można oszacować, że $p_j = 2^{n-k_j}/(\sum_{i=1}^{t} 2^{n-k_i}) \geqslant 1/t$ (wylosowane wartościowanie spełnia przynajmniej jedną klauzulę z t). Jeżeli $X = \sum_{i=1}^{m} X_i$ jest zmienną losową (zdefiniowaną jak w przypadku powyżej) to wówczas wartość oczekiwana $E(X)$ wynosi

$$E(X) = E(\sum_{i=1}^{m} X_i) = \sum_{i=1}^{m} p_i \geqslant \frac{m}{t}.$$

W końcu wyznaczając m z nierówności Chernoffa otrzymujemy

$$m \geqslant t3ln(\frac{2}{\delta})/(\epsilon^2).$$

Jak widać wartość m zależy liniowo od t, czyli ilości formuł w dysjunktywnej postaci normalnej, dzięki czemu można się przekonać, że dla powyższego problemu NP istnieje efektywny algorytm (ϵ, δ)-aproksymacyjny.

3.3 Ogólny schemat algorytmu stadnego

Algorytmy stadne, mimo że naśladują bardzo różne zachowania, charakteryzują się wspólnym schematem przetwarzania, który można przedstawić w następujących krokach [4, 21, 36]:

1. Inicjalizacja populacji początkowej - wstępne określenie parametrów algorytmu takich jak wielkość populacji, liczba iteracji, określenie wskaźnika jakości;

2. Ocena jakości znalezionego rozwiązania (bądź znalezionych rozwiązań) dla populacji początkowej - wyznaczenie wskaźnika jakości dla każdej wartości znalezionej w iteracji, który decyduje o jej zakwalifikowaniu do dalszego przetwarzania;

3. Eksploatacja zakwalifikowanych najlepszych rozwiązań - przeszukiwanie przestrzeni rozwiązań w sąsiedztwie (otoczeniu) lokalnych wartości optymalnych;

4. Eksploracja pozostałej przestrzeni rozwiązań;

5. Ocena jakości znalezionych rozwiązań - krok identyczny jak w punkcie 2;

6. Iteracje - jeżeli spełniony warunek stopu to program kończy działanie, w przeciwnym przypadku skocz do punktu 3;

Struktura ta, ze względu na kroki eksploracji i eksploatacji, jest również bliska algorytmom genetycznym w ich ogólnej postaci. W ogólności istnieje wiele analogii pomiędzy algorytmami genetycznymi a stadnymi, jednakże podstawową cechą, która je różni to brak odpowiednika operatora krzyżowania w algorytmach stadnych.
Niestety, z przedstawionej powyżej struktury wszystko, co można powiedzieć to to, że algorytm stadny jest algorytmem iteracyjnym, który wyznacza optima lokalne w sąsiedztwach losowo wyznaczanych punktach przestrzeni dopuszczalnych rozwiązań, wymagającym dobrze zdefiniowanego warunku stopu. Stąd też istnieje wiele implementacji ogólnego schematu, z reguły dostosowanych do szczególnych klas problemów, a przede wszystkim do struktury przeszukiwanej przestrzeni (jeżeli można o niej cokolwiek wnioskować).
W dalszej części pracy będzie pokazane jak zbudować taką implementację dla problemu przepływowego flow shop o liczbie procesorów większej od 2, o którym wiemy, że należy do klasy *NP*-zupełnej, co zostało udowodnione w jednym z poprzednich rozdziałów (patrz dowód *twierdzenia 2*). Jako przykłady implementacji użyte zostaną algorytmy: pszczeli, wykorzystujący strategię poszukiwania pożytku pszczół, oraz kukułki, wykorzystujący z kolei strategię poszukiwania gniazd ptaków gospodarzy, do przeszukiwania kombinatorycznej przestrzeni rozwiązań.

3.4 Algorytm pszczeli

Entomologia daje nam pewne wyobrażenie o strategii pozyskiwania pokarmu przez pszczoły. Początkowo część robotnic (zwiadowcy) poszukuje źródeł pokarmu wybierając w sposób losowy kierunek i obszar poszukiwań. Po

znalezieniu potencjalnego źródła pożywienia, wraca do ula, by informacje o kierunku, odległości, obszarze i zasobności źródła pokarmu przekazać innym pszczołom w specyficznym rodzaju tańca (ang. *waggle dance*). Pozostałe pszczoły oceniają taniec zwiadowców i decydują się lub też nie na przyłączenie się do najlepiej tańczącego spośród nich. Generalnie im taniec jest dłuższy i bardziej ekspresyjny, tym więcej pszczół podąży za zwiadowcą. Zrobią to również te, które aktualnie eksploatują inne źródło, a które uznają, że nowo odnaleziony obszar rokuje na większą ilość pokarmu i efektywniejszy jego zbiór. W ten sposób kolonie pszczół rozwiązują problem maksymalizowania zbiorów przy ograniczonych zasobach.
Algorytm pszczeli, w swojej podstawowej implementacji, jest algorytmem iteracyjnym i z reguły jedynym kryterium stopu jest liczba iteracji podobnie, jak w przypadku innych algorytmów losowych [21, 36]. Dodatkowo jak każdy algorytm losowy (genetyczny, symulowane wyżarzanie) podstawowy algorytm pszczeli może być podatny na problem przedwczesnej zbieżności w zależności od przyjętej strategii przeszukiwania. W końcu ogólne działanie algorytmu można opisać w następujących, zmodyfikowanych względem schematu ogólnego dla algorytmów stadnych, krokach [21]:

1. Inicjalizacja populacji początkowej – wstępne określenie parametrów algorytmu takich jak wielkość populacji – n, liczba wyznaczonych rokujących obszarów w iteracji – m, liczba najlepszych rozwiązań w iteracji – e oraz inne dodatkowe (liczba pszczół elitarnych przeszukujących rozwiązania najlepsze - ne, w końcu liczba pszczół nm przeszukujących obszary rokujące); określenie wskaźnika jakości; przypisanie losowo wybranych obszarów przestrzeni rozwiązań poszczególnym zwiadowcom;

2. Ocena jakości znalezionych rozwiązań – wyznaczenie wskaźnika jakości dla każdej wartości znalezionej przez zwiadowcę; wskaźnik określa, czy zwiadowca odnalazł rokujący obszar; m najlepszych obszarów przechodzi do następnego kroku algorytmu;

3. Eksploracja najlepszych m rozwiązań – dla oznaczonych e spośród m rozwiązań przydzielamy większą ilość osobników do przeszukiwania otoczenia;

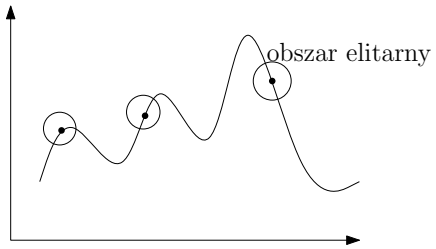
4. Wybór najlepszych e rozwiązań spośród m przeszukiwanych – rozwiązania te biorą udział w dalszej części algorytmu; tych e bieżacych rozwiązań może się różnić od e początkowo znalezionych;

5. Eksploracja przestrzeni niezatrudnionymi osobnikami – rozlosowanie obszarów niezatrudnionym osobnikom;

6. Ocena jakości znalezionych rozwiązań - krok identyczny z krokiem 2

7. Iteracje – jeśli spełniono warunek stopu – zakończ; w przeciwnym wypadku skocz do kroku 3;

Na rysunku 3.1 pokazano przykład przeszukiwanego obszaru rozwiązań z zaznaczoną interpretacją wartości m oraz e.



Rysunek 3.1: Interpretacja wartości m i e dla $e = 1$ oraz $m = 3$. Punktami zaznaczono obszary elitarny oraz rokujące, które przechodzą do następnej iteracji. Kółkami zaznaczono ich eksploatowane sąsiedztwa. *Źródło: opracowanie własne.*

Powyżej wspomniano o tym, że algorytm pszczeli (a w ogólnym przypadku algorytm stadny) w swojej podstawowej implementacji przyjmuje arbitralnie liczbę iteracji oraz, że może być podatny na problem przedwczesnej zbieżności. W ujęciu klasycznym w celu wyznaczenia warunku stopu można posłużyć się błędem względym wartości otrzymanej w danej iteracji do aktualnie najlepszego znalezionego rozwiazania. Jeżeli błąd ten jest mniejszy niż przyjęte ϵ to algorytm zatrzyma się. Jednak w tym podejściu nic nie można powiedzieć o czasie wykonywania algorytmu, stąd też dodatkowo wyznacza się maksymalną ilość iteracji. Niestety, nic nie wiadomo także o tym, jak bardzo znalezione najlepsze rozwiązanie różni się od rzeczywistego oraz

czy jeżeli pozwolimy działać algorytmowi dowolnie długi czas to, czy wartość znaleziona będzie zbiegać do optymalnej w którykolwiek ze sposobów przedstawionych wcześniej.

3.5 Algorytm kukułki

Drugim algorytmem badanym w pracy jest algorytm kukułki. Poraz pierwszy został on zaprezentowany przez Xin-she Yang'a i Suash'a Deb'a w 2009 roku. Inspiracją dla twórów było zachowanie pewnych gatunków kukułki, zwane pasożytniczym obowiązkiem wychowania potomstwa, oznaczające po prostu podrzucanie swoich jaj do gniazd innych gatunków ptaków (gospodarzy). Same kukułki gniazd nie budują. Oczywiście ptaki starają się bronić przed tym rodzajem pasożytnictwa. Jeżeli rozpoznają w swoim gnieździe obce jajo, wówczas albo wyrzucają je z gniazda, albo porzucają gniazdo i budują nowe w nowym, innym miejcu. Z drugiej strony, niektóre gatunki kukułki starają specjalizować w kilku gatunkach ptaków gospodarzy tak, że kolor i wzór ich jaj naśladują kolor i wzór jaj gospodarzy. Okazuje się, że idealizacja zachowań kukułki może zostać wykorzystana do różnych problemów optymalizacji.

Algorytm kukułki bazuje na czterech wyidealizowanych regułach:

1. każda kukułka składa jedno jajo w czasie i podrzuca je do losowo wybranego gniazda

2. najlepsze gniazda (reprezentujące najlepsze rozwiązania) przechodzą do kolejnej iteracji

3. liczba dostępnych gniazd gospodarzy jest stała

4. obce jajo może zostać rozpoznane przez gospodarza z prawdopodobieństwem $p_a \in (0, 1)$. Wówczas gniazdo takie jest porzucone, a gospodarz buduje nowe w losowo wybranym miejscu

Dodatkowo Yang and Deb odkryli, że sposób w jaki kukułki poszukują kolejnego gniazda gospodarza lepiej opisuje lot Levy'ego jak podstawowy spacer

losowy. Silną stroną algorytmu jest jego prostota. Praktycznie poza rozmiarem populacji n jedynym znaczącym parametrem jest p_a. Powyższe ustalenia można opisać za pomocą następującego pseudokodu:

Niech będzie dana funkcja celu: $f(x) : x = (x_1, x_2, \ldots, x_d)$;
Wygeneruj populację początkową n gniazd gospodarzy;
while not (warunek STOPu)
do
- Wylosuj jedną (i-tą) kukułkę;
- Zastąp jej obecne rozwiązanie poprzez wykonanie lotu Levy'ego;
- Wyznacz wartość funkcji jakości F_i ($F_i \propto f(x_i)$);
- Wylosuj jedno (j-te) z n gniazd gospodarzy;
- **if** ($F_i > F_j$)
  - **then** Zastąp zawartość gniazda przed dane kukułki;
  - **else**
    - Odrzuć dane kukułki
    - z prawdopodobieństwem p_a;
    - Zbuduj nowe gniazdo;
- Posortuj rozwiązania i zachowaj najlepsze z nich;
- Przekaż najlepsze rozwiązania do nastepnej iteracji;

return (najlepsze rozwiązanie)

Kończąc warto zwrócić uwagę na następujące wnioski wynikające z przedstawionego algorytmu:

1. gniazda przechowują stan algorytmu
2. nie jest istotna ilość kukułek, dlatego można przyjąć, że jest ona równa ilości gniazd
3. nie jest istotny stan kukułki, dlatego można przjąć, że jest on równy stanowi wybranego gniazda.

Rozdział 4

Określenie parametrów algorytmu stadnego dla problemu optymalizacji modelu *flow shop*

4.1 Problem zbieżności

Występujące w literaturze [4, 21, 36] pojęcie *problem zbieżności* jest rozumiane z reguły jako problem przedwczesnej zbieżności i związane jest przede wszystkim z algorytmami genetycznymi lub szerzej ewolucyjnymi. Problem ten może, ale nie musi, dotyczyć również algorytmów stadnych. Przedwczesną zbieżność obserwuje się wtedy, gdy algorytm traci zdolność przeszukiwania przestrzeni rozwiązań przed znalezieniem globalnego optimum. Algorytmy charakteryzujące się przedwczesną zbieżnością posiadają dwie zasadnicze wady. Przede wszystkim nie są one w stanie powiedzieć jak dobre jest znalezione rozwiązanie w odniesieniu do wartości optymalnej. Jedynym praktycznym miernikiem jakości jest stosunek przeszukanej przestrzeni rozwiązań przez algorytm do całkowitej przestrzeni rozwiązań. Drugą niekorzystną cechą wynikającą z przedwczesnej zbieżności jest prawdopodobna (ponieważ zależy także od rodzaju problemu) zależność znalezionego rozwiązania od obszaru,

w którym algorytm rozpoczyna przeszukiwanie. Istnieją opracowane sposoby zapobiegania przedwczesnej zbieżności, które nie będą przedstawiane ze względu na ograniczenia pracy. W dalszej części główna uwaga będzie skierowana na problem zbieżności w rozumieniu stochastycznym [23, 32, 34]. Oznacza to, że będziemy zastanawiać się, jak można próbkować przestrzeń rozwiązań, aby rozwiązania znalezione przez algorytm działający w odpowiednio długim czasie przybliżały się do wartości optymalnej. Następnie spróbujemy odpowiedzieć sobie, czy przestrzeń permutacji zadań w modelu *flow shop* jest (ϵ, δ)-aproksymacyjna (tzn. czy może istnieć efektywny algorytm, którego rozwiązanie nie będzie się różnić bardziej niż o ϵ z prawdopodobieństwem δ) i, jeżeli taki algorytm istnieje, ile iteracji będzie wymagać.

Przypomnijmy, że problem $Fm||C_{max}$ należy do klasy problemów *NP*-zupełnych, a rozwiązaniem jest taka kolejność zadań, która minimalizuje maksymalny czas przetwarzania C_{max}. Innymi słowy interesuje nas znalezienie właśnie takiej permutacji z n zadań (permutacja określa kolejność zadań). Zachodzi pytanie, jak powinniśmy poruszać się po stanach przestrzeni rozwiązań, aby być prawie pewnym, że znalezione rozwiązanie będzie stochastycznie zbiegać do optimum. Aby na nie odpowiedzieć, zastosowana zostanie metoda *Monte Carlo* oparta o łańcuchy Markowa. Podstawowy pomysł polega na zdefiniowaniu ergodycznego łańcucha Markowa, którego zbiorem stanów jest przestrzeń próbek, a rozkład stacjonarny tego łańcucha wymaganym rozkładem próbkowania [23, 32, 34].

4.1.1 Sąsiedztwo stanu

Intuicja podpowiada, że w konstruowanym łańcuchu każdy stan musi być osiągalny z każdego innego stanu. Formalnie można powiedzieć, że każde dwa stany są wzajemnie komunikującymi się a sam łańcuch jest nieredukowalny. Gdyby ten warunek nie był spełniony, istniałoby niezerowe prawdopodobieństwo tego, że algorytm utknąłby w pewnym lokalnym optimum. Z drugiej strony wiemy, że w modelu $Fm||C_{max}$ ilości zadań i procesorów są liczbami skończonymi. Stąd też liczba stanów konstruowanego łańcucha Markowa również musi być skończona. Obserwacja ta powoduje, że odpowiedni łańcuch

będzie szukany pośród tych wszystkich, które są nieredukowalne i skończone. Pierwszym krokiem będzie zaprojektowanie zbioru ruchów, który będzie gwarantować, że przestrzeń stanów będzie nieredukowalna względem tego łańcucha Markowa. Niech zbiór stanów osiągalnych w jednym kroku ze stanu x (z wykluczeniem x) tworzy sąsiedztwo x i będzie oznaczane jako $N(x)$ oraz jeżeli $y \in N(x)$ to $x \in N(y)$. Chcemy także, żeby $N(x)$ było relatywnie małe i dające się łatwo obliczyć. Zdefiniujmy sąsiedztwo w następujący sposób. Niech $x = j_1 j_2 j_3 \dots j_n$, gdzie j_k jest k-tym zadaniem. Wówczas y będzie w $N(x)$ wtedy, gdy pierwsze zadanie (z indekesem 1) w x zostało wymienione z którymkolwiek z zadań na dalszej pozycji. Czyli:

$$y \in N(x) = \{j_2 j_1 j_3 \dots j_n, j_3 j_2 j_1 \dots j_n, \dots, j_n j_2 j_3 \dots j_1\}.$$

Z takiego określenia sąsiedztwa wynika również, że dwa dowolne stany są osiągalne w co najwyżej $2(n-2)+1$ krokach.

Dowód: Niech będą dane uporządkowania a i b takie, że a i b różnią się na wszystkich pozycjach. W pierwszym kroku znajdujemy w b zadanie j_k takie, że w a ma ono indeks n i zamieniamy w b j_k z zadaniem o indeksie 1. Następnie zamieniamy pozycje 1 oraz n-tą w b. Te same kroki należy powtórzyć dla zadania kolejno o indeksie $n-1$ w a, $n-2$ w a, a w najgorszym przypadku te dwie operacje powtarzane będą dla $n-2$ pozycji, ponieważ ostatnie dwie pozycje o indeksach 2 i 1 wymagają już tylko jednego kroku. Stąd też całkowita liczba operacji wymaganych do przekształcenia b w a wynosi $2(n-2)+1$.

Po określeniu sąsiedztwa musimy wybrać prawdopodobieństwa przejść. Jednym ze sposobów jest spacer losowy po stanach wyznaczonego sąsiedztwa, a ponieważ żaden ze stanów nie jest wyróżniony, można przyjąć, że prawdopodobieństwa przejść z x do któregokolwiek z $y \in N(x)$ są sobie równe i wynoszą $(n-1)^{-1}$. Ponieważ x zostało wykluczone z sąsiedztwa (prawdopodobieństwo przejścia ze stanu x do x wynosi zero), odpowiadający przejściom łańcuch Markowa jest okresowy (o okresie 2). Jeżeli zmodyfikujemy spacer losowy dodając każdemu stanowi x prawdopodobieństwo pętli własnej, to można uzyskać rozkład stacjonarny, co ma potwierdzenie w po-

niższym twierdzeniu:

Twierdzenie 5: Dla skończonej przestrzeni stanów Ω i struktury sąsiedztwa $\{N(X) : x \in \Omega\}$ niech $N = \max_{x\in\Omega} |N(x)|$. Niech M będzie dowolną liczbą taką, że $M \geqslant N$. Rozważmy łańcuch Markowa, dla którego

$$P_{xy} = 1/M \text{ jeżeli } x \neq y \text{ i } y \in N(x),$$
$$P_{xy} = 0 \text{ jeżeli } x \neq y \text{ i } y \notin N(x),$$
$$P_{xy} = 1 - N(x)/M \text{ jeżeli } x = y.$$

Jeśli łańcuch jest nieredukowalny i nieokresowy, to rozkład stacjonarny jest rozkładem jednostajnym [32, 34].

Dowód: Jeżeli dla $x \neq y$ $\pi_x = \pi_y$, wówczas $\pi_x P_{xy} = \pi_y P_{yx}$, ponieważ $P_{xy} = P_{yx} = 1/M$. Stąd wynika, że łańcuch jest łańcuchem czasowo odwracalnym o rozkładzie stacjonarnym $\pi = (\pi_0, \ldots, \pi_x, \pi_y, \ldots, \pi_n)$, gdzie $\pi_x = 1/|\Omega|$.

Wniosek: W rozważanym modelu $Fm||C_{max}$ dla wszystkich stanów x wartości $|N(x)|$ są sobie równe i wynoszą $N = 1/(n-1)$. Niech $M = N$, wówczas prawdopodobieństwo pętli własnej wynosi również $1/(n-1)$, a próbkowanie będzie tożsame z wylosowaniem któregokolwiek ze stanów należących do $x \cup N(x)$ z prawdopodobieństwen $1/(n-1)$. Warto też zauważyć, że tak wyznaczony sposób próbkowania gwarantuje nie tylko zbieżność stochastyczną do wartości optymalnej, ale także niezależność znalezionej wartości od początkowego obszaru przeszukiwania. Do oszacowania po ilu próbkach rzeczywiście stan końcowy nie będzie zależał od poczatkowego obszaru przeszukiwania zastosujemy teorię sprzężenia łańcuchów Markowa. Jednak zanim to uczynimy należy rozważyć jeszcze jeden problem przy zdefiniowanych jak powyżej prawdopodobieństwach przejść.

4.1.2 Eksploatacja w algorytmie pszczelim

Jeżeli spojrzeć na drugi krok algorytmu pszczelego, to zwraca uwagę akcja, w której wyznaczany jest wskaźnik jakości dla każdej wartości znalezionej przez zwiadowcę. Wskaźnik ten, w kroku trzecim, jest użyty do wyznacze-

nia funkcji werbunkowej, która to jest funkcją zmiennej losowej. Werbowane pszczoły mogą podążyć za zwiadowcą z pewnym prawdopodobieństwem, zależnym od wyznaczonego wcześniej wskaźnika jakości, co z punktu widzenia algorytmu oznacza akceptację, lub jej brak, dla przejścia ze stanu x do stanu y. Stąd też znalezione prawdopodobieństwa przejść, będa mieć nstępującą postać:

$$P_{xy} = \Psi(x,y)f(x,y) \text{ jeżeli } x \neq y \text{ i } y \in N(x),$$
$$P_{xy} = 0 \text{ jeżeli } x \neq y \text{ i } y \notin N(x),$$
$$P_{xy} = 1 - \sum_{y \neq x} \Psi(x,y)f(x,y) \text{ jeżeli } x = y.$$

Aby powyższy łańcuch był odwracalny w czasie musi być spełniony poniższy warunek:

$$\pi(x)\Psi(x,y)f(x,y) = \pi(y)\Psi(y,x)f(y,x).$$

Jeżeli założyć, że Ψ jest symetryczna wówczas powyższe równanie uprości się do

$$\pi(x)f(x,y) = \pi(y)f(y,x).$$

Wprowadźmy oznaczenie $b(x,y) = \pi(x)f(x,y)$. Ponieważ $f(x,y)$ jest funkcją prawdopodobieństwa, której wartości są mniejsze lub równe 1, to $b(x,y) \leqslant \pi(x)$, a ponieważ $b(x,y) = b(y,x)$ to $b(x,y) \leqslant \pi(y)$.
Ponieważ zależy nam na tym, aby akceptować Ψ z jak największym prawdopodobieństwem, wynika stąd, że $b(x,y)$ jest ograniczone przez mniejszą w wartości π. Czyli $b(x,y) = \min(\pi(x), \pi(y))$, a stąd otrzymujemy

$$f(x,y) = \min(1, \pi(y)/\pi(x)).$$

Tym sposobem doszliśmy do algorytmu Metropolisa, którego postać formalnie można zapisać w następujący sposób [32, 34]:

$$P_{xy} = \Psi(x,y)\min(1, \pi(y)/\pi(x)) \text{ jeżeli } x \neq y \text{ i } y \in N(x),$$
$$P_{xy} = 1 - \sum_{y \neq x} \Psi(x,y)\min(1, \pi(y)/\pi(x)) \text{ jeżeli } x = y,$$
$$P_{xy} = 0 \text{ w pozostałych przypadkach.}$$

Wniosek: W prezentowanym w pracy rozwiązaniu, funkcja werbunkowa zastąpiona została przez algorytm Metropolisa, w którym reprezentują wskaźniki jakości aktualnie badanego obszaru przestrzeni rozwiązań oraz potencjalnie badanego jego sąsiedztwa. Przyjmując $w(x)$ jako funkcję jakości w x, $M = |N(x)| + 1$ otrzymujemy poniższą formułę na prawdopodobieństwa przejść:

$$P_{xy} = (1/M)\min(1, w(y)/w(x)) \text{ jeżeli } x \neq y \text{ i } y \in N(x),$$
$$P_{xy} = (1 - (M-1)/M)\min(1, w(y)/w(x)) \text{ jeżeli } x = y,$$
$$P_{xy} = 0 \text{ w pozostałych przypadkach.}$$

Pierwsze i drugie równanie są takie same, stąd też ostatecznie otrzymujemy:

$$P_{xy} = (1/M)\min(1, w(y)/w(x)) \text{ jeżeli } x \neq y \wedge y \in N(x) \vee x = y,$$
$$P_{xy} = 0 \text{ w pozostałych przypadkach.}$$

Formuła ta została zastosowana w implementowanym algorytmie do eksploatacji sąsiedztwa.

4.1.3 Eksploatacja w algorytmie kukułki

Implementacja algorytmu kukułki przedstawiona przez Yang'a i Dob'a dotyczyła problemu, w którym przeszukiwana przestrzeń rozwiązań była ciągła i różniczkowalna. Stąd też w swojej pracy mogli użyć Euklidesowej definicji odległości oraz wyznaczać gradient dalszego poszukiwania. W przypadku przestrzeni kombinatorycznej, a z taką mamy do czynienia w modelu

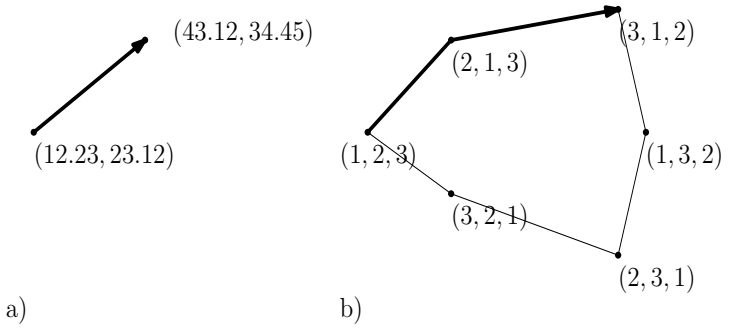
$F_m|prmu|C_{max}$, zarówno pojęcie odległości jak i gradientu, a ogólniej mówiąc kierunku przeszukiwania, nie są tak jednoznacznie zdefiniowane. W rozdziale 4.1.1 pokazano jak zdefiniowane zostało sąsiedztwo danego punktu x. Jasnym jest, że sposób eksploatacji przestrzeni jest zdeterminowany przez sposób wyznaczania tego sąsiedztwa.
Niech operacja $walk$ permutacji $from$ na permutacji $toward$ będzie określona następująco:

1. znajdź indeks k taki, że $from(1) = toward(k)$
2. zamień ze sobą wartości na pozycjach 1 oraz k w permutacji $from$
3. po zamianie nowa permutacja będzie postaci: $from_1(k, 2, 3, \ldots, 1, k + 1, \ldots, n)$

wówczas powiedzieć można, że odległość pomiędzy $from_1$ a $toward$ będzie o jeden mniejsza jak w przypadku $from$ a $toward$, zaś samą odległość można zdefiniować jako:

$$\begin{aligned} dist(from, toward) = k \text{ takie, że} \\ & walk_1(from, toward) \circ walk_2(from_1, toward) \circ \cdots \\ & \circ walk_k(from_{k-1}, toward) \\ & = toward \end{aligned}$$

gdzie $\circ$ jest złożeniem funkcji wykonanym k-krotnie na wyniku $from_i$ poprzedniej operacji $walk_i = walk$.
Powtarzając sposób dowodu z rozdziału 4.1.1 można wykazać, że maksymalna odległość pomiędzy permutacjami $from$ a $toward$ wynosi $n-1$ oraz, że jest ona najmniejsza. Druga obserwacja jest istotna ze względu na to, że można ją wykorzystać do określenia kierunku dalszego przeszukiwania. W pierwotnej implementacji Yang'a i Dob'a kierunkek ten jest po prostu wyznaczany jako różnica wektorów obecnie najlepszego ($\vec{b}$) i właśnie wyznaczonego ($\vec{n}$) $\vec{b} - \vec{n}$. **W przypadku przestrzeni kombinatorycznej przyjęto w pracy, że kierunek jest wyznaczany przez funkcję** $walk(from, toward)$**, co zostało pokazane na rysunku 4.1.**



Rysunek 4.1: Na rysunku a) pokazano różnicę wektorów $\vec{b}$ oraz $\vec{n}$ w przestrzeni ciągłej dla pierwotnego problemu Yang'a-Dob'a. Z kolei na rysunku b) pokazano wyznaczanie kierunku w przestrzeni kombinatorycznej dla problemu $F_m|prmu|C_{\max}$. *Źródło: opracowanie własne.*

4.2 Problem próbkowania

W poprzednim rozdziale rozważany był problem wyznaczania sąsiedztwa. Ważne jest, aby sposób, w jaki jest ono wyznaczane, gwarantowało zbieżność stochastyczną. W bieżącym rozdziale zostanie przeprowadzona analiza, która odpowie nam ile razy należy losować, aby otrzymać próbki nieobciążone. Do jej przeprowadzenia użyte zostaną teorie sprzężania łańcuchów Markowa i ich mieszania [32, 34].

4.2.1 Odległość stochastyczna i łańcuchy sprzężone

Rozkład stacjonarny jest rozkładem granicznym łańcucha Markowa, gdy liczba kroków dąży do nieskończoności. W rzeczywistości obserwujemy skończoną liczbę kroków. Zachodzi pytanie ile kroków należy wykonać do chwili, w której obserwowany łańcuch był bliski łańcuchowi z rozkładem granicznym. Aby określić ilościowo pojęcie bliskości łańcuchów należy wprowadzić miarę odległości.
Odległość w sensie wariacji pomiędzy dwoma rozkładami prawdopodobieństwa d_1 oraz d_2 na przestrzeni Ω definiuje się jako:

$$\|d_1 - d_2\| = \max |d_1(A) - d_2(A)|, \text{ gdzie } A \subseteq \Omega.$$

Definicja ta jest jawnie probabilistyczną, ponieważ odległość między d_1 a d_2 jest największą różnicą pomiędzy prawdopodobieństwami wyznaczonymi dla tego samego stanu A przez te dwa rozkłady. Definicja odległości w sensie wariacji jest silnie powiązana z (ϵ, δ)-aproksymacją z *twierdzenia 4*. Algorytm generuje ϵ-jednostajną próbę z przestrzeni Ω wtedy i tylko wtedy, gdy odległość w sensie wariacji między wyjściowym rozkładem d a rozkładem jednostajnym u spełnia nierówność $\|d - u\| \leqslant \epsilon$. W odniesieniu do *twierdzenia 4* oznacza to, że chcemy, aby algorytm generował próbki, które spełniają następującą nierówność:

$$P(|d(x) - u(x)| \leqslant \epsilon) \leqslant 1 - \delta.$$

Jeżeli uda się wykazać, że własność taka zachodzi, to będzie to oznaczać, że algorytm potrafi efektywnie poruszać się po przestrzeni rozwiązań dopuszczalnych.
Oszacujmy teraz odległość w sensie wariacji po t krokach. Wprowadźmy najpierw poniższe definicje.

Definicja 1: Niech π będzie rozkładem stacjonarnym nieredukowalnego, nieokresowego łańcucha Markowa na przestrzeni stanów S. Niech p_x^t reprezentuje rozkład stanu łańcucha po t krokach, gdy zaczyna on w stanie x. Wówczas $\Delta(t) = max\|p_x^t - \pi\|$, dla każdego $x \in S$, jest największą odległością w sensie wariacji pomiędzy rozkładem p_x^t a π, a liczbę kroków $\tau \geqslant t$, po których $\Delta(\tau) \leqslant \epsilon$ nazywa się czasem mieszania się łańcucha Markowa i oznaczamy go przez $\tau(\epsilon)$.
Mówimy, że łańcuch Markowa jest szybko mieszający się, gdy $\tau(\epsilon)$ jest wielomianową funkcją $\log(1/\epsilon)$ i rozmiaru problemu.

Definicja 2: Sprzężeniem łańcucha Markowa M_t na przestrzeni stanów S jest łańcuch Markowa $Z_t = (X_t, Y_t)$ na przestrzeni stanów $S \times S$ taki, że

$$P(X_{t+1} = x' | Z_t = (x, y)) = P(M_{t+1} = x' | M_t = x)$$
$$P(Y_{t+1} = y' | Z_t = (x, y)) = P(M_{t+1} = y' | M_t = y)$$

Oznacza to, że sprzężenie składa się z dwóch kopii łańcucha Markowa M przebiegającymi jednocześnie. Te dwie kopie nie muszą być w tym samym stanie oraz nie muszą wykonywać tych samych ruchów, jednak pod względem prawdopodobieństwa przejść muszą zachowywać się tak jak łańcuch pierwotny.
W dalszych rozważaniach interesować nas będą sprzężenia, które, po pierwsze, doprowadzają kopie do tego samego stanu, i po drugie, powodują, że od tego momentu łańcuchy wykonują identyczne ruchy. Gdy dwie kopie osiągną ten sam stan wówczas nastąpiło ich sprzężenie. Podejście takie ma uzasadnienie w twierdzeniu:

Twierdzenie 6: Niech $Z_t = (X_t, Y_t)$ będzie sprzężeniem łańcucha Markowa M na przestrzeni stanów S. Załóżmy, że istnieje T takie, że:

$$\forall x, y \in SP(X_T \neq Y_T | X_0 = x, Y_0 = y) \leqslant \epsilon$$

wówczas

$$\tau(\epsilon) \leqslant T, \text{ natomiast } \Delta(t) \leqslant \max(P_{x,y}(\tau > t)) \text{ dla } x, y \in \Omega.$$

Oznacza to, że dla dowolnego stanu początkowego odległość w sensie wariacji między rozkładem łańcucha po T krokach a rozkładem stacjonarnym jest co najwyżej równa ϵ.
Dowód twierdzenia będzie pominięty.

4.2.2 Wyznaczenie T dla sąsiedztwa stanu

Pozostało nam teraz znalezienie wartości T dla założonego sposobu wyznaczania sąsiedztwa i poruszaniu się po nim. Przypomnijmy, że w przypadku rozważanego algorytmu stadnego, sąsiedztwo danej permutacji wyznacza operacja zamiany elementów na pozycjach o indeksach 1 oraz k. Z kolei prawdopodobieństwa przejść wynoszą $(1/M)\Psi$ (gdzie Ψ jest rozkładem prawdopodobieństwa akceptacji stanu docelowego), gdy stan docelowy należy do sąsiedztwa bieżącego stanu oraz zero w przeciwnym przypadku. Niech pierw-

sza kopia rozważanego procesu dokładnie odtwarza algorytm poruszania się po sąsiedztwie, a druga niech wykonuje następujące ruchy:

- wyszukaj indeks m elementu o wartości równej elementowi o indeksie 1 w pierwszej kopii;
- zamień elementy na pozycjach m oraz k zgodnie z Ψ;

Ponieważ Ψ jest takie samo dla obu kopii, stąd też nie ma ono wpływu na poszczególne kroki. Wykażmy teraz, że druga kopia jest prawidłowa. Zauważmy, że prawdopodobieństwo tego, że element będzie znaleziony na pozycji m jest takie samo jak wylosowanie indeksu k w przypadku pierwszej kopii i wynosi $1/M$. Stąd można wnioskować, że oba łańcuchy ze względu na prawdopodobieństwo zachowują się identycznie. Po drugie wyznaczmy po ilu krokach druga kopia stanie się identyczna z kopią pierwszą.
Niech permutacje (o n pozycjach) obu kopii różnią się na wszystkich pozycjach i niech będzie wylosowany indeks i. Wówczas, po iteracji, w najgorszym przypadku permutacje te będą różnić się na $n-1$ pozycjach. Jeżeli w następnym kroku wylosowany zostanie którykolwiek z poprzednio wylosowanych indeksów, wówczas ilość zgodnych pozycji się nie zmieni. Jeżeli z kolei wylosowany zostanie indeks poprzednio niewylosowany to, po j-tej iteracji, liczba różniących się pozycji będzie wynosić $n_{j-1}-1$. Stąd wniosek, że permutacje będą zgodne wtedy i tylko wtedy, gdy każdy indeks zostanie wylosowany przynajmniej jeden raz.
Niech dane będą dwie początkowe permutacje odpowiednio dla procesu p_1 naśladującego algorytm wyznaczania sąsiedztwa oraz p_2 dla procesu z zamianą elementów na pozycjach m i k. Wówczas przykładowy proces sprzęgania ma postać jak w tabeli 4.1.

Jak to już zostało powiedziane, permutacje będą zgodne wtedy i tylko wtedy, gdy każdy indeks zostanie wylosowany przynajmniej jeden raz. Stąd też należy zadać sobie pytanie, ile razy trzeba losować, aby być dostatecznie pewnym, że każdy z indeksów był wylosowany przynajmniej raz? Problem ten można sprowadzić do problemu kolekcjonera kuponów, który wyznacza wartość oczekiwaną liczby losowań tak, aby wśród wyciągniętych kuponów były wszystkie n różne. Wyznaczymy teraz tę wartość. Interesuje nas ciąg

nr	index k	kopia 1	kopia 2
1	2	$\underline{1}\ \underline{2}\ 3\ 4 \rightarrow 2\ 1\ 3\ 4$	$3\ \underline{1}\ 2\ 4 \rightarrow 3\ 1\ 2\ 4$
2	4	$\underline{2}\ 1\ 3\ \underline{4} \rightarrow 4\ 1\ 3\ 2$	$3\ 1\ \underline{2}\ \underline{4} \rightarrow 3\ 1\ 4\ 2$
3	4	$\underline{4}\ 1\ 3\ \underline{2} \rightarrow 2\ 1\ 3\ 4$	$3\ 1\ \underline{4}\ \underline{2} \rightarrow 3\ 1\ 2\ 4$
4	3	$\underline{2}\ 1\ \underline{3}\ 4 \rightarrow 3\ 1\ 2\ 4$	$3\ 1\ \underline{2}\ 4 \rightarrow 3\ 1\ 2\ 4$
5	2	$\underline{3}\ \underline{1}\ 2\ 4 \rightarrow 2\ 1\ 3\ 4$	$\underline{3}\ \underline{1}\ 2\ 4 \rightarrow 2\ 1\ 3\ 4$
6	1	$\underline{2}\ 1\ 3\ 4 \rightarrow 2\ 1\ 3\ 4$	$\underline{2}\ 1\ 3\ 4 \rightarrow 2\ 1\ 3\ 4$

Tabela 4.1: Kolejne wylosowane indeksy k. Podkreśleniem zaznaczono wymieniane elementy. Szarym kolorem oznaczono moment sprzężenia łańcuchów.

zmiennych losowych $X = \sum_{i=1}^{n} X_i$ takich, że w każdym kolejnym kroku liczba różnych kuponów wzrasta o jeden. Wiadomo też, że X_i na rozkład geometryczny. Prawdopodobieństwo tego, że mając zebranych $i-1$ spośród n różnych kuponów, wylosowany zostanie i-ty wynosi $p_i = 1 - \frac{i-1}{n}$,a stąd $X_i = 1/p_i$. Wyznaczając wartość oczekiwaną otrzymujemy:

$$\begin{aligned} E(X) = E(\sum_{i=1}^{n} X_i) &= \sum_{i=1}^{n} \frac{n}{n-i+1} = n \sum_{i=1}^{n} \frac{1}{n-i+1} = n \sum_{i=1}^{n} \frac{1}{i} \\ &= n \ln(n) + cn,\ 0 \leqslant c \leqslant 1 \end{aligned}$$

Mając wyznaczoną wartość oczekiwaną liczby kroków można teraz wyznaczyć odległość $\Delta(t)$, gdzie $t = n \ln(n) + cn$. Z *twierdzenia 6* wiadomo, że

$$\Delta(t) \leqslant P(\tau > n \ln(n) + cn) \text{ i } \Delta(t) \leqslant \epsilon,$$

z kolei dla problemu kolekcjonera kuponów

$$P(\tau > n \ln(n) + cn) \leqslant e^{-c},$$

stąd $e^{-c} \leqslant \epsilon$, a po przekształceniu względem c otrzymujemy

$$c \geqslant \ln(\epsilon^{-1}).$$

Ostatecznie otrzymujemy

$$t = n \ln(n) + \ln(\epsilon^{-1})n,$$

co oznacza, że prawdopodobieństwo tego, że łańcuchy nie będą sprzężone po czasie t wynosi co najwyżej ϵ.

4.3 Liczba iteracji

Powyższe rozważania posłużą do wyznaczenia liczby iteracji w zaimplementowanym algorytmie stadnym. W rozdziale 3 przedstawiono wniosek z nierówności Chernoffa, który zostanie teraz zastosowany do konkretnego algorytmu.

Przypomnijmy postać z dowodu twierdzenia:

$$P\left(\left|\frac{1}{m}\sum_{i=1}^{m} X_i - \mu'\right| \geqslant \epsilon\mu'\right) \leqslant 2e^{-m\mu'\epsilon^2/3} \leqslant \delta,$$

a stąd

$$m \geqslant 3\ln(\frac{2}{\delta})/(\epsilon^2\mu').$$

Aby reguła ta mogła być zastosowana, zmienne losowe X_i muszą mieć jednakowe rozkłady. Z drugiej strony wykazaliśmy, że rozkłady zbiegają do swojego rozkładu stacjonarnego, a liczba kroków, po którym rozkłady różnią się między sobą o ϵ w sensie wariancji wynosi

$$t = n\ln(n\epsilon^{-1}).$$

Z algorytmu Metropolisa otrzymaliśmy następującą formułę na prawdopodobieństwa przejść pomiędzy stanami:

$$P_{xy} = (1/M)\min(1, w(y)/w(x)) \text{ jeżeli } x \neq y \wedge y \in N(x) \vee x = y,$$
$$P_{xy} = 0, \text{ w pozostałych przypadkach.}$$

Niestety, nie możemy w szczególności założyć, że rozkładem stacjonarnym jest rozkład jednostajny. Jednak warto zwrócić uwagę, że w ogólności, żaden z kierunków przeszukiwania nie jest wyróżniony. Oznacza to, że zaczynając próbkowanie z dowolnych punktów przestrzeni rozwiązań, w granicy, dowol-

ne sąsiedztwo będzie wybierane z takim samym prawdopodobieństwem, stąd przyjmuje się, że $\mu' = 1/M = 1/(n+1)$. **Z powyższego można wnioskować, że całkowita liczba iteracji w algorytmie musi być większa lub co najmniej równa wartości** mt **wynoszącej:**

$$mt \geqslant \frac{3\ln(\frac{2}{\delta})}{\frac{\epsilon^2}{n+1}} n \ln(n\alpha^{-1}) = 3(n+1)n\epsilon^{-2} ln(n\alpha^{-1}) ln(\frac{2}{\delta}).$$

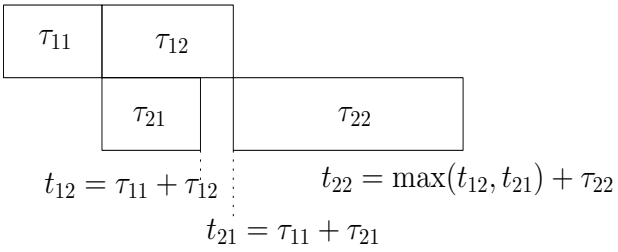
Jak widać, algorytm ma złożoność kwadratową ze względu na rozmiar problemu, którym w przypadku modelu *flow shop* jest ilość zadań do uporządkowania. Dodatkowo zależy od od trzech współczynników jakości α, δ oraz ϵ.

4.4 Zrównoleglanie algorytmu stadnego dla N procesorów

Algorytmy stochastyczne łatwo poddają się zrównoleglaniu, jeżeli tylko zmienne losowe są niezależne [25, 52]. Podejście takie jednak nie zmienia złożoności problemu [5, 39], a zysk jaki można uzyskać, wynika z prawa Amdalh'a. Ponieważ algorytm stadny jest algorytmem stochastycznym, w którym przynajmniej m z mt próbek jest niezależnych (ponieważ potrzeba t kroków, aby zdekorelować stany), każda wylosowana permutacja może być przetwarzana w osobnym wątku, procesie a w końcu procesorze. Przetwarzanie polega na wyznaczeniu C_{max} dla danej permutacji zadań. W permutacyjnym modelu przepływowym *flow shop* jest to o tyle proste, że każdy procesor musi przetwarzać zadania w tej samej kolejności. Wynika stąd, że czas zakończenia wykonywania się zadania i przez procesor j t_{ji} można wyznaczyć z poniższej zależności rekurencyjnej:

$$t_{ji} = \tau_{ji} + \max(t_{(j-1)i}, t_{j(i-1)}),$$

gdzie τ_{ji} oznacza czas wykonywania zadania i przez j-ty procesor. Rysunek 4.2 pokazuje tę zależność graficznie. Wykażmy najpierw, że obliczone w ten



Rysunek 4.2: Sposób wyznaczania wartości t_{ij} dla $i = j = 2$. *Źródło: opracowanie własne.*

sposób C_{max} jest rzeczywiście minimalne dla danej permutacji zadań. Z własności permutacyjnego modelu *flow shop* wynika, że zadanie i na procesorze j nie może rozpocząć się przed czasem zakończenia przetwarzania zadania i na procesorze $j-1$ oraz że procesor j nie może rozpocząć przetwarzać zadania i przed zakończeniem przetwarzania zadania $i-1$. Stąd:

$$t_{ji}^{start} \geqslant t_{(j-1)i} \wedge t_{ji}^{start} \geqslant t_{j(i-1)} \Rightarrow t_{ji}^{start} = \max(t_{(j-1)i}, t_{j(i-1)})$$

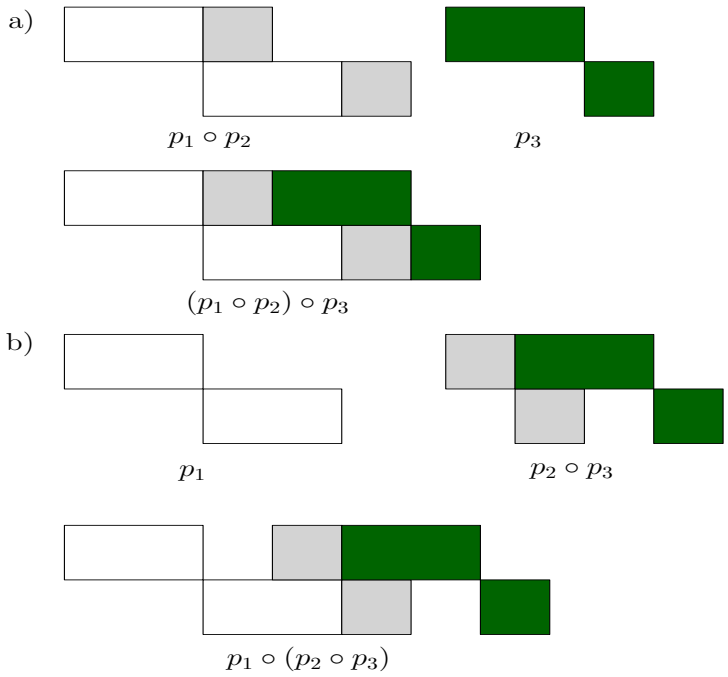
i ostateczne:

$$t_{ji} = \tau_{ji} + \max(t_{(j-1)i,} t_{j(i-1)}).$$

Znając formułę na wyznaczenie C_{max} można teraz zastanowić się czy istnieje dla niej sposób na zrównoleglenie obliczeń. Innymi słowy chcielibyśmy wiedzieć, czy istnieje algorytm, którego złożoność obliczeniowa będzie zredukowana z $o(mn)$ do polilogarytmicznej. Ogólniej, czy działanie wyznaczające t_{ji} należy do klasy problemów NC w odróżnieniu do problemów P-zupełnych (oznaczanych w literaturze również jako $\#P$), które mimo wielomianowej złożoności, nie są podatne na zrównoleglenie. Przykładem tego ostatniego jest problem MAX FLOW.

Jeżeli $p_1 = \{t_{11}, t_{12}, \ldots, t_{1n}\}$ będzie zbiorem zadań (dokładniej ich czasów) przetwarzanych przez pierwszy procesor, natomiast $p_2 = \{t_{21}, t_{22}, \ldots, t_{2n}\}$ i $p_3 = \{t_{31}, t_{32}, \ldots, t_{3n}\}$ odpowiednio przez drugi i trzeci, wówczas jeżeli $(p_1 \circ p_2) \circ p_3 = p_1 \circ (p_2 \circ p_3)$, gdzie $\circ$ oznacza funkcję t_{ij}, to obliczenia można zrównoleglić, a złożoność czasowa z $o(mn)$ w ogólności zostanie zredukowana do $o(n \log_2 m)$.

Niestety można udowodnić (poprzez podanie przykładu), że w ogólności działanie $\circ$ nie jest łączne. Przede wszystkim należy zwrócić uwagę, że wyznaczonych wartości t_{23j}, będących rezultatem $(p_2 \circ p_3)$, nie można użyć jako kroku pośredniego do obliczenia $p_1 \circ (p_2 \circ p_3)$, ponieważ nie uwzględniają one zależności czasowych pomiędzy t_{1j} oraz t_{2j}. Na rysunku 4.3 przedstawiono przykład ilustrujący problem dla n równego 3 zadania. Na rysunku a)



Rysunek 4.3: Szczególny przypadek (jeden z wielu) dowodzący, że działanie $\circ$ nie jest łączne. Na rysunku a) przedstawiono wynik działania $(p_1 \circ p_2) \circ p_3$, natomiast na rysunku b) najpierw wynik $p_2 \circ p_3$ a następnie $p_1 \circ (p_2 \circ p_3)$. Wartość optymalna w przypadku a) wynosi 6, z kolei w przypadku b) 7. *Źródło: opracowanie własne.*

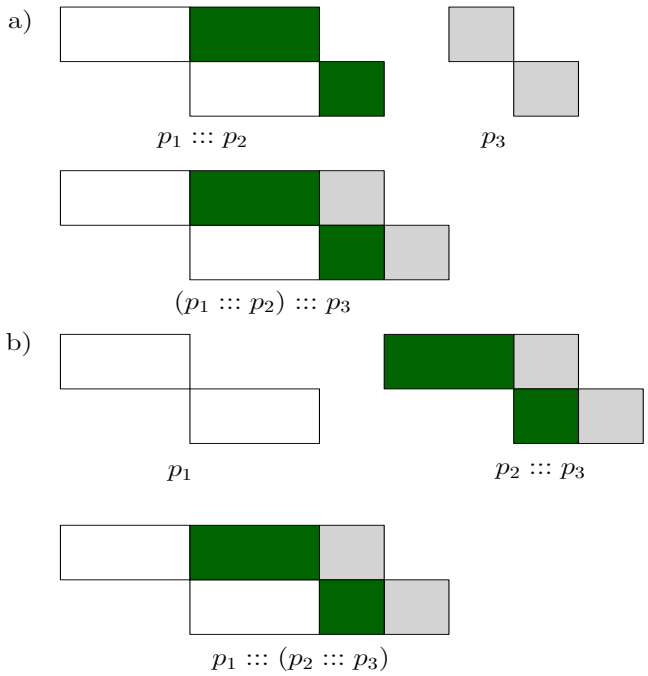
przedstawiono $(p_1 \circ p_2) \circ p_3$, która wyznaczona jest prawidłowo, natomiast b) przedstawia przypadek $p_1 \circ (p_2 \circ p_3)$, dla którego z wartości $t_{2(n-2)}$ oraz $t_{2(n-1)}$ nie można wyznaczyć nowej wartości t'_{2n}. Stąd wniosek, że formuła $t_{ji} = \tau_{ji} + \max(t_{(j-1)i}, t_{j(i-1)})$ nie jest podatna na zrównoleglenie.

4.4.1 Przypadek szczególny $t_{ji} \geqslant t_{(j+1)(i-1)}$

Jeżeli zachodzi warunek:

$$\forall(j,i) t_{ji} + \tau_{ji} \geqslant t_{(j+1)(i-1)}$$

wówczas w permutacyjnym modelu flow shop problem obliczania wartości C_{max} można rozwiązać w inny sposób (rysunek 4.4). Ponieważ zawsze zadanie



Rysunek 4.4: W przypadku, gdy $t_{ji} \geqslant t_{(j+1)(i-1)}$, wówczas problem *flow shop* należy do klasy złożoności *NC*. Kolejność złożenia zadań zachowuje łączność, jak przykładowo pokazano na rysunkach a) i b). *Źródło: opracowanie własne.*

t_{ij} musi wykonać się przed $t_{(i+1)j}$ oraz t_{ij} musi wykonać się przed $t_{i(j+1)}$ oznacza to, że dla każdego zadania k można najpierw wyznaczyć ciągi $t_k = (t_{1k}, t_{2k}, \ldots, t_{mk})$, które następnie użyte są do wyznaczenia swoich złożeń. W szczególności procedura ta przebiega następująco. W pierwszym kroku ciąg t_k służy do wyznaczenia par czasów rozpoczęcia (t^b_{ik}) i zakończenia (t^e_{ik}) przetwarzania zadania k przez procesory z założeniami, że:

1. $t^e_{ik} = t^b_{ik} + t_{ik}$,

2. $t^b_{1k} = 0$,

3. $t^b_{(i+1)k} = t^e_{ik}$.

Stąd też przekształcony ciąg będzie miał postać:

$$t_k = ((t^b_{1k}, t^e_{1k}), (t^b_{2k}, t^e_{2k}), \ldots, (t^b_{mk}, t^e_{mk})).$$

Z punktu widzenia złożoności wyznaczenie wszystkich t_k wymaga $o(mn)$ operacji. Jednakże warto zauważyć, że każde z t_k jest obliczane niezależnie od pozostałych $n-1$ zadań, co z kolei prowadzi do wniosku, że w przypadku zrównoleglenia złożoność czasowa będzie ograniczona przez $o(mn/c)$. Wartość c określa ilość dostępnych jednostek obliczeniowych i w skrajnym przypadku może być równa n, wówczas złożoność będzie wynosić $o(m)$.
Zanim zostanie wyznaczone złożenie t_k oraz t_{k+1}, zdefiniujmy samą operację ::: (). Niech będą dane złożenia $T_l = (t_w, t_{w+1}, \ldots, t_k)$ oraz $T_r = (t_{k+1}, t_{k+2}, \ldots, t_u)$, wówczas

$$\begin{aligned}
&:::(T_l, T_r) \\
&\quad = T_l \text{ concat } map(T_r, \tau_{offset}) \\
&\quad = T_l \text{ concat } (map(t_{k+1}, \tau_{offset}), map(t_{k+2}, \tau_{offset}), \ldots, map(t_u, \tau_{offset})) \\
&\quad = (t_w, t_{w+1}, \ldots, t_k) \text{ concat } (map(t_{k+1}, \tau_{offset}), map(t_{k+2}, \tau_{offset}), \ldots, \\
&\quad map(t_u, \tau_{offset})) \\
&\quad = (t_w, t_{w+1}, \ldots, t_k, map(t_{k+1}, \tau_{offset}), map(t_{k+2}, \tau_{offset}), \ldots, \\
&\quad map(t_u, \tau_{offset}))
\end{aligned}$$

nazywamy funkcją złożenia T_l oraz T_r, natomiast map definiuje się jako

$$\begin{aligned}
&map(t_k, \tau_{offset}) \\
&\qquad = ((t^b_{1k} + \tau_{offset}, t^e_{1k} + \tau_{offset}), (t^b_{2k} + \tau_{offset}, t^e_{2k} + \tau_{offset}), \ldots, \\
&\qquad (t^b_{mk} + \tau_{offset}, t^e_{mk} + \tau_{offset})).
\end{aligned}$$

Wartość τ_{offset} definiuje się następująco:

$$\tau_{offset} = t^e_{ik} - t^b_{i(k+1)} : 1 \leqslant i \leqslant m, \tau_{offset} \geqslant 0, \forall \text{ j } t^e_{jk} \leqslant t^b_{j(k+1)} + \tau_{offset}.$$

Wyznaczmy teraz złożenie $T_l = (t_k)$ oraz $T_r = (t_{k+1})$.

$$::: (T_l, T_r) = (t_k, map(t_{k+1}, \tau_{offset})),$$

z własności permutacyjnego modelu flow shop wynika, że τ_{offset} jest również minimalnym przesunięciem, dla którego czasy ciągów t_k i t_{k+1} nie zachodzą na siebie. Stąd wniosek, że dla danej permutacji zadań, C_{max} wyrażone przez $t^e_{m(k+1)}$ jest minimalne.
Sprawdźmy teraz, czy tak zdefiniowane złożenie jest działaniem łącznym. Niech

$$\begin{aligned} T_1 &= (t_w, t_{w+1}, \ldots, t_k), \\ T_2 &= (t_{k+1}, t_{k+2}, \ldots, t_u), \\ T_3 &= (t_{u+1}, t_{u+2}, \ldots, t_z). \end{aligned}$$

Wówczas

$$\begin{aligned} &:::(T_1, ::: (T_2, T_3)) \\ &\quad =::: (T_1, (t_{k+1}, t_{k+2}, \ldots, t_u, map((t_{u+1}, t_{u+2}, \ldots, t_z), \tau))) \\ &\quad = (t_w, t_{w+1}, \ldots, t_k, map(t_{k+1}, t_{k+2}, \ldots, t_u, map((t_{u+1}, t_{u+2}, \ldots, t_z), \tau), \upsilon)) \\ &\quad = (t_w, t_{w+1}, \ldots, t_k, map((t_{k+1}, t_{k+2}, \ldots, t_u), \upsilon), map((t_{u+1}, t_{u+2}, \ldots, t_z), \upsilon + \tau)); \end{aligned}$$

z kolei

$$\begin{aligned} &:::(::: (T_1, T_2), T_3) \\ &\quad =::: ((t_w, t_{w+1}, \ldots, t_k, map((t_{k+1}, t_{k+2}, \ldots, t_u), \upsilon)), T_3) \\ &\quad = (t_w, t_{w+1}, \ldots, t_k, map((t_{k+1}, t_{k+2}, \ldots, t_u), \upsilon), \\ &\quad map((t_{u+1}, t_{u+2}, \ldots, t_z), \upsilon + \tau)). \end{aligned}$$

Stąd też złożenie ::: () jest działaniem łącznym.
Dzięki powyższej własności można zastosować poniższy schemat równoległych obliczeń. Niech na początku będą dane $T_1, T_2, \ldots, T_n$, gdzie $T_k = (t_k)$. Niech pierwsza jednostka obliczeniowa (wątek, proces) wyznacza złożenie ::: $(T_1, T_2) = T_{12}$, druga ::: $(T_3, T_4) = T_{34}$, a $n/2$ z kolei ::: $(T_{n-1}, T_n) = T_{(n-1)n}$ (przypadek, gdy n jest nieparzyste jest nieistotny z punktu widzenia ogólności rozważań). Następnie niech ponownie pierwsza jednostka będzie wyznaczać ::: $(T_{12}, T_{34}) = T_{1234}$, druga ::: $(T_{56}, T_{78}) = T_{5678}$ itd. Stąd też w granicznym wypadku $n/2$ jednostek będzie potrzebować $m \log_2 n$ kroków na wyznaczenie C_{max} dla danej permutacji n zadań na m procesorach, gdzie m przed logarytmem wynika z wyznaczenia τ_{offset}.
Pozostaje jeszcze oszacowanie operacji *map*. Wydawać by się mogło, że ponieważ $t_k = ((t^b_{1k}, t^e_{1k}), (t^b_{2k}, t^e_{2k}), \ldots, (t^b_{mk}, t^e_{mk}))$ oraz $T_r = (t_{k+1}, t_{k+2}, \ldots, t_u)$ to całkowity koszt *map* wynosi $m\mathrm{f}(n)$. Jednakże zamiast dodawać przesunięcie do każdego elementu t_k wystarczy je zapamiętać i zwiększać w przypadku kolejnego złożenia. Stąd m może być pominięte w dalszych szacowaniach złożoności. f(n) jest całkowitą ilością operacji zwiększania przesunięcia ($offsetu$) przez jednostkę obliczeniową. Ponieważ podczas każdego złożenia należy zmodyfikować połowę wartości przesunięć, stąd też średnio podczas wszystkich złożeń będzie modyfikowanych $\log_2 n$ wartości przesunięć, co w ostateczności prowadzi do wniosku, że f(n) jest ograniczone przez $o(\log_2 n)$.

Znając złożoność poszczególnych obliczeń można wyznaczyć całkowitą złożoność obliczeń równoległych, która wynosi:

$$o(m + m \log_2 n\mathrm{f}(n)) = o(m + m \log_2^2 n),$$

w którym pierwsze m wynika z przekształcenia t_k w T_k.
Kończąc rozważania warto zauważyć, że dopiero od $n \geqslant 16$ zaproponowany schemat równoległy daje przewagę nad podejściem sekwencyjnym $t_{ji} = \tau_{ji} + \max(t_{(j-1)i}, t_{j(i-1)})$.

Rozdział 5

Realizacja algorytmów stadnych dla modelu $Fm|prmu|C_{max}$

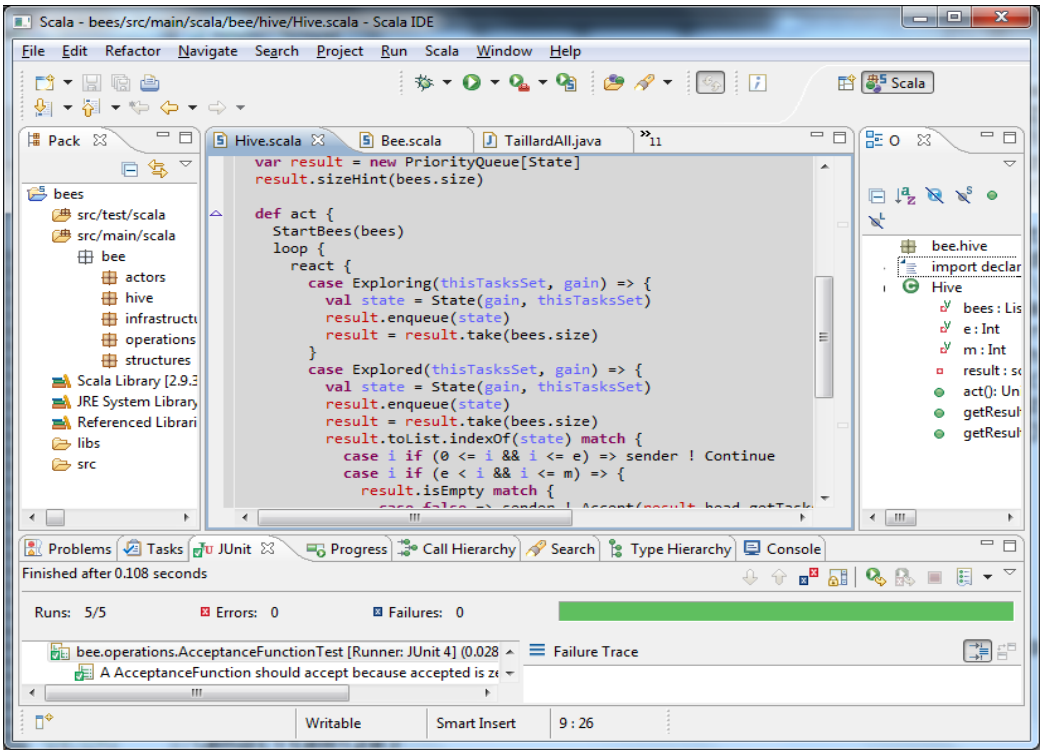
5.1 Implementacja

Zrównoleglony algorytm pszczeli dla problemu *flow shop* zaimplementowano w języku *Scala* (w wersji 2.9) [1, 16, 48, 49]. Na jego wybór miało wpływ przede wszystkim to, że należy on do języków funkcyjnych, których konstrukcja mocno wspiera tworzenie aplikacji dla systemów rozproszonych. W systemach tego rodzaju bardzo ważnym jest, aby wykonywane instrukcje pozbawione były efektów ubocznych (ang. *side effects*) a paradgmat funkcyjny rozwiązuje ten problem (używa się też pojęcia *referencial transparency*, którego rozumienie jest takie samo). Z praktycznego, programistycznego, punktu widzenia przejawia się to przez używanie przede wszystkim zmiennych stałych (ang. *immutables*) oraz zastępowaniem pętli przez wywołania rekurencyjne (pętla z reguły modyfikuje lokalny stan). Z drugiej strony *Scala* pozwala na stosowanie imperatywnego sposobu tworzenia kodu, charakterystycznego dla takich języków obiektowych jak *C++* i *Java*, dzięki czemu dużo efektywniej rozwiązuje się problemy przede wszystkim z wydajnością przetwarzania. Jest to związane z tym, że tworzenie obiektów niezmiennych

(*immutables*) jest ciągle dużo kosztowniejsze od przypisania nowej wartości obiektowi zmiennemu (*mutable*). Stąd też można powiedzieć, że *Scala* łączy zalety paradygnatów imperatywnego i funkcyjnego bez ich wad.
Bardzo ważną cechą przemawiającą za zastosowaniem *Scali* jest to, że do tworzonych fragmentów programu można stosować metodę *indukcji strukturalnej*. Dzięki temu w wielu wypadkach upraszcza się dowodzenie poprawności programu i ogranicza ilość niezbędnych testów.
Do realizacji rozwiązań rozproszonych, oprócz tradycyjnych wątków z *Javy*, *Scala* oferuje obiekty nazwane *aktorami*. Jest to abstrakcja (zapożyczona z języka *Erlang*, również bliska pojęciu *agent*), w której obiekty komunikują się za pomocą niemutowalnych asynchronicznych wiadomości. Dzięki temu rozwiązuje się problem współdzielenia stanu pomiędzy asynchronicznie działającymi obiektami, co z kolei implikuje, że system jest łatwiejszy w analizie. Jest to istotne zwłaszcza w przypadku wychwytywania potencjalnych zakleszczeń czy zagłodzeń. Jest oczywiste, że nawet rezygnacja ze współdzielonego stanu może nie uchronić od potencjalnych problemów, ponieważ *aktorzy* zaimplementowani są za pomocą tradycyjnych wątków. Z drugiej strony to oddelegowanie implementacji chroni programistę od pisania bardzo skomplikowanego i wrażliwego kodu, dostarczanego przez zewnętrzną specjalizowaną bibliotekę, pozostawiając mu więcej czasu na realizację wymaganego zadania.
Ponieważ, jak to już wspomniano, *Scala* jest językiem funkcyjnym, stąd też implementacja modelu *aktorów* jest dużo bardziej efektywna niż w przypadku języków imperatywnych. Oczywiście, *aktor* może być, ale nie musi, tożsamy z wątkiem. *Scala* dostarcza zmienne systemowe (*actors.maxPoolSize* oraz *actors.corePoolSize*), które kontrolują rzeczywistą ilość wątków wykorzystywanych przez aplikację i potrafi ją adaptować do potrzeb. Warto też zaznaczyć, że od wersji 2.10 dostarczany model jest w pełni rozproszony, tzn. możliwe jest tworzenie całych klastrów aktorów na maszynach zdalnych. Wszysto, co jest wówczas wymagane od programisty to dostarczenie konfiguracji zapisanej w specjalizowanym DSLu (który również napisany jest w *Scali*).

5.1.1 Środowisko uruchomieniowe

Jako środowiska programistycznego i uruchomieniowego użyto *Typesafe IDE* będącą aplikacją umożliwiającą pisanie programów oraz testów jednostkowych w *Scali*, i która opiera się o platformę *Eclipse*. Na rysunku 5.1 przedstawiono ogólny widok wykorzystanej aplikacji.



Rysunek 5.1: Ogólny widok *Typesafe IDE. Źródło: opracowanie własne.*

5.1.2 Struktura programu

Z punktu widzenia organizacji kodu program został podzielony na następujące funkcjonalne pakiety:

actors zawierający przede wszystim implementację pszczoły (*Bee class*),

hive**,** który definiuje ul (*Hive class*),

cuckoo zawierający implementację algorytmu *kukułki* wraz z pomocniczymi obiektami,

infrastructure, w którym zawarto implementacje elementów pomocniczych z punktu widzenia *Scali* takie jak konwertery typów i implementacje *loggerów*. W pakiecie tym zlokalizowany jest też *Runner* będący wygodnym API dla programów uruchomieniowych i testów jednostkowych,

operations grupuje operacje wykonywane przez algorytm takie jak wyznaczanie uszeregowania zadań czy obliczanie C_{max} dla znalezionego uszeregowania,

structures definiuje struktury wiadomości i danych użyte w aplikacji.

Warto też dodać, że funkcjonalności poszczególnych klas i obiektów zostały pokryte 84 testami jednostkowymi znajdującymi się w folderze *src/test/scala*. Warto zaznaczyć, że tworzą one specyfikację aplikacji.

5.1.3 Odpowiedzialności obiektów infrastrukturalnych

Podstawowymi obiektami biblioteki są:

Runner Jest on wygodnym API do tworzenia obiektów *Hive* oraz *Bees*. Jest to robione poprzez wywołanie konstruktora o sygnaturze:

```
Runner(tasks: List[List[Double]], epsilon: Double = 0.1,
    delta: Double = 0.01, alpha: Double = 0.01, bees: Int,
    elite: Int, potential: Int)
```

Jak widać wielkości *alpha*, *delta* i *epsilon* są predefiniowane, z kolei *bees* determinuje liczność roju pszczół. Są one użyte do wyznaczenia liczby iteracji dla pojedynczej pszczoły zgodnie z zależnoscią wyznaczoną w rozdziale 4.3

$$mt \geqslant 3(n+1)epsilon^{-2}n\ln(alpha^{-1}n)\ln(\frac{2}{delta})/bees.$$

Nieobecność wyznaczonych rozdziale 3.2 wielkości *ne* (liczność pszczół elitarnych) oraz *nm* (liczność pszczół rokujących) wynika stąd, że ze względu na małą liczność roju przyjęto w algorytmie, iż $ne = e$ oraz $nm = m - e$. Stąd też nie ma potrzeby ekspozycji tych dwóch wielkości w API algorytmu.
Po stworzeniu i zainicjowaniu obiektów *Hive* oraz *Bees Runner* oczekuje na wywołanie metody *run* w celu rozpoczęcia obliczeń (wymiany wiadomości pomiędzy pszczołami a ulem), która zdefiniowana jest następująco:

```
def run: List[State]
```

Wynikiem działania tej metody jest lista obiektów *State* (plik *State.scala*) reprezentujących końcowe stany pszczół o sygnaturze:

```
class State(computedGain: Double, foundTasksOrder:
    List[TaskOffset[Double]])
```

Przykładowy kod użycia można znaleźć w *RunnerTest.scala*, w pakiecie testów Tailard'owskich oraz w poniższym przykładzie:

```
// data
val tasks = List(
    List(1, 2, 3.5),
    List(4, 0, 1.54)
)
// Runner instantiation with implementation of Logger
val runner = new Runner(tasks = tasks, bees = 5, elite = 1,
    potential = 1)
    with Logger { def trace(message: String) =
        println(message) }
// run search and collect results
val result = runner.run
println(result)
```

Dodatkowo do przetwarzania zadanych danych początkowych wyrażonych jako ciąg czasów przetwarzania do postaci wymaganej przez aplikację *Runner* definiuje metodę w obiekcie towarzyszącym o sygnaturze:

```
def transformToTasks(taillardData: String, tasksNumber: Int,
    log: (List[List[Double]]) => Unit): List[List[Double]]
```

CuckooRunner jest odpowiednikiem *Runnera* dla *algorytmu kukułki* tworzącym obiekty *Forrest* oraz *Cuckoos*. Sygnatura konstruktora tego obiektu jest następującej postaci:

```
CuckooRunner(tasks: List[List[Double]], epsilon: Double =
    0.1, delta: Double = 0.01, alpha: Double = 0.01, cuckoos:
    Int, discoveredEggTreshold: Double = 0.25, bestNests: Int,
    acceptBestNest: Double = 0.8)
```

Wartości *epsilon*, *delta* oraz *alpha* są użyte od wynaczenia liczby iteracji w analogiczny sposób jak w przypadku algorytmu pszczelego, która wynosi:

$$mt \geqslant 3(n+1)epsilon^{-2}n\ln(alpha^{-1}n)\ln(\frac{2}{delta})/cuckoos.$$

discoveredEggTreshold określa prawdopodobieństwo z jakim gospodarz może odkryć podrzucone jajo w gnieździe natomiast *bestNests* jest ilością zachowywanych najlepszych rozwiązań (gniazd).
Dodatkowo wprowadzono dodatkowy parametr *acceptBestNest* aby zapobiec przedwczesnej zbieżności w następujący sposób:

procedure GETBESTNEST($bestNests, acceptBestNest$)
 $accept \leftarrow$ RANDOM$(0,1)$;
 if $(accept < acceptBestNest)$
  then return $(bestNests.head)$
  else return $(bestNests[\text{RANDOM}(1, bestNests.size - 1)])$

Wynik powyższej operacji jest użyty przez *kukułkę* do określenia kierunku przeszukiwania przestrzeni tak, jak to zostało przedstawione w

rozdziale 4.1.3.

Podobnie jak w przypadku *Runnera* wymagane jest wywołanie funkcji *run* zwracającej obiekt *State*. Przykładowy kod użycia można znaleźć w testach Taillard'owskich dla *algorytmu kukułki* (pakiet *boids.cuckoo.taillard*) oraz w przykładzie poniżej:

```
// data
val tasks = List(
    List(1, 2, 3.5),
    List(4, 0, 1.54)
)
// Runner instantiation with implementation of Logger
val runner = new CuckooRunner(tasks, epsilon = 0.1, delta =
    0.01, alpha = 0.01, cuckoos = 50, discoveredEggTreshold =
    0.2, bestNests = 5, acceptBestNest = 0.6)
  with Logger { def trace(message: String) = println(message)
     }
// run search and collect results
val result = runner.run
println(result)
```

5.1.4 Obiekty i procedury w implementacji algorytmu pszczelego

Głownymi aktorami (także w rozumieniu jednostek przetwarzających) implementacji algorytmu pszczelego są ul (*hive*) oraz jego pszczoły (*bees*).

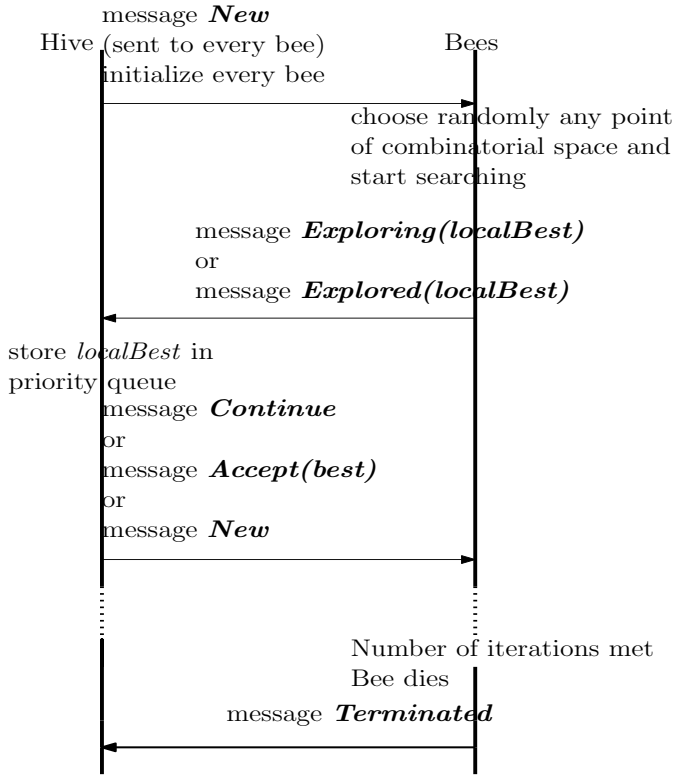
Hive odpowiada za przetwarzanie wiadomości otrzymywanych od pszczół oraz utrzymuje najlepsze wyniki w kolejce priorytetowej o ustalonej długości n równej wielkości roju. Schemat wymienianych wiadomości pomiędzy ulem a pszczołami przedstawiono na rysunku 5.2.

Jak widać, ul może odpowiedzieć na *Explored* jednym z trzech komunikatów. Decyzja, która z odpowiedzi jest wysyłana z ula do pszczoły, podejmowana jest dynamicznie w zależności od bieżącej wartości od-

powiedzi i zawartości kolejki priorytetowej zgodnie z następującą procedurą:

procedure REACT($queue, localBestSchedule$)
 comment: kolejka sortowana jest zgodnie z najmniejszym C_{max}
 $localBestIndex \leftarrow$ ENQUEUE($queue, localBestSchedule$)
 if ($localBestIndex \leqslant e$)
  then SEND($Continue$)
  else if ($e < localBestIndex \leqslant m$)
  then SEND($Accept(queue[\text{RANDOM}(0, e)])$)
  else SEND(New)

Jedyną reakcją ula na wiadomość *Exploring* jest zapisanie otrzyma-



Rysunek 5.2: Komunikacja pomiędzy ulem a pszczołami. *Źródło: opracowanie własne.*

nego wyniku do kolejki priorytetowej. Jest to też informacja dla ula,

że pszczoła w tym momencie przetwarza rokujące sąsiedztwo lokalnego minimum.

Ul kończy działanie w momencie, gdy ostatnia z żyjących pszczół wyśle komunikat *Terminated*.

Bee przeszukuje zadaną przestrzeń rozwiązań wyznaczając wartość C_{max}, o czym zawiadamia ul.

Jeżeli pszczoła otrzyma wiadomość *New*, losuje punkt przestrzeni rozwiązań zgodnie z procedurą:

procedure RANDOMORDER($preOrdered : Buffer$)
$newOrder \leftarrow [\,]$
while not $preOrdered.isEmpty$
do $\begin{cases} var \leftarrow preOrdered.remove(\text{RANDOM}(0, preOrdered.size - 1)) \\ newOrder.add(var) \end{cases}$
return $(newOrder)$

dla którego wyznacza C_{max}, a następnie odpowiada ulowi komunikatem *Explored*.

W przypadku wiadomości *Accept* pszczoła decyduje czy zaakceptować zaproponowany przez ul porządek zadań i rozpocząć przeszukiwanie jego sąsiedztwa w promieniu Markowowskim czy też kontynuować przeszukiwanie sąsiedztwa bieżącego lokalnego rozwiązania. Decyzja wyznaczana jest poprzez poniższą regułę (algorytm Metropolisa):

procedure ACCEPT($localCmax, globalCmax$)
$ratio \leftarrow globalCmax/localCmax$
if $(ratio \geqslant 1)$
then return (**true**)
else return $(\text{RANDOM}(0, 1) \leqslant ratio)$

Z kolei wiadomość *Continue* nakazuje pszczole przeszukiwanie sąsiedztwa bieżącego lokalnego rozwiązania.

Algorytm przeszukiwania sąsiedztwa zaimplementowano następująco:

comment: *schedule* - permutacja zadań

comment: *dist* - Markowowski promień sąsiedztwa

comment: *iters* - ilość cykli życia pszczoły

global *hive*

procedure MARKOVSEARCH($schedule, dist, iters$)

if $(iters == 0)$

then SENDTO($hive, Terminate$); EXIT()

else if $(distance == 0)$

then $\begin{cases} \text{SENDTO}(hive, Explored(schedule)) \\ \textbf{return } (schedule) \end{cases}$

else $\begin{cases} sched \leftarrow \text{SWAPRANDOMLYFIRSTANDANYTASK}(schedule) \\ sched \leftarrow \text{COMPUTECMAX}(sched) \\ best \leftarrow \text{MIN}(schedule, sched) \\ \textbf{if } (best! = schedule) \\ \quad \textbf{then } \begin{cases} \text{SENDTO}(hive, Exploring(best)) \\ \textbf{return } (\text{GREEDYSEARCH}(best, iters - 1)) \end{cases} \\ \quad \textbf{else return } (\text{MARKOVSEARCH}(sched, dist - 1, iters - 1)) \end{cases}$

Nazwa procedury nawiązuje do sposobu w jaki została zdefiniowana odległość pomiędzy sąsiednimi permutacjami (jak pamiętamy jest ona zdefiniowana w sensie wariacji) oraz ilości kroków potrzebnych do zdekorelowania stanów, które wyznaczono za pomocą analizy sprzężonych łańcuchów Markowa. Jak widać, algorytm będzie wykonywać się aż nie zostanie znalezione uszeregowanie lepsze od bieżącego. Wówczas wykonywane jest przeszukiwanie zachłanne sąsiedztwa nowego stanu.

procedure GREEDYSEARCH($schedule, iters$)

```
if (iters == 0)
  then sendTo(hive, Terminate); exit()
  else { neighbors ← findNeighbors(schedule)
       { best ← schedule
       { for each neighbor ∈ nieghbors
       {   do best = min(best, neighbor)
       { if (best == schedule)
       {   then return (globalSearch(schedule, iters − 1))
       {   else { sendTo(hive, Exploring(best))
       {        { return (greedySearch(best, iters − 1))
```

gdzie *findNeighbors* jest zdefiniowane jako:

```
procedure findNeightbors(schedule)
 neighbors ← [ ]
 numberOfTasks ← schedule.size
 for index = 1 to numberOfTasks − 1
   do { neighbor ← swapFirstAndNthTasks(schedule, index)
      { neighbor ← computeCMax(neighbor)
      { neighbors.add(neighbor)
 return (neighbors)
```

Przeszukiwanie zachłanne wykonuje się aż do momentu, w którym dla każdego z sąsiednich uszeregowań bieżąca wartość C_{max} jest niewiększa od znalezionych w sąsiedztwie. Wówczas algorytm losuje dowolne uszeregowanie z przestrzeni rozwiązań. Procedurę *globalSearch* można opisać następująco:

```
procedure globalSearch(schedule, iters)
```

if $(iters == 0)$

then sendTo$(hive, Terminate)$; exit()

else
- $schedule \leftarrow$ randomOrder$(schedule)$
- $schedule \leftarrow$ computeCMax$(schedule)$
- sendTo$(hive, Explored(schedule))$
- **return** $(schedule)$

Sprowadza się ona do wyznaczenia C_{max} dla nowowygenerowanego uszeregowania zadań i przesłania wiadomości o tym do ula.

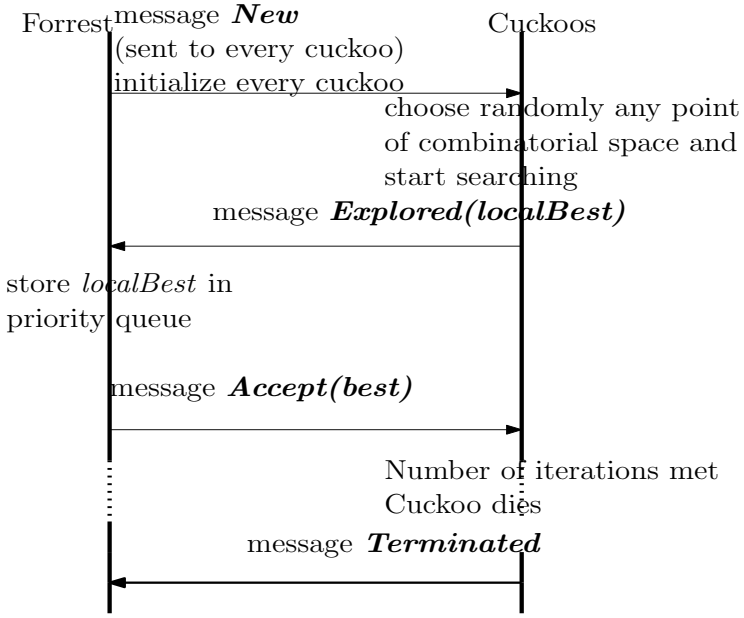
W pliku *Bee.scala* oprócz wymienionych powyżej algorytmów (zmodyfikowany algorytm Metropolisa *markovSearch* oraz algorytm zachłanny *greedySearch*) zaimplementowano również inne podejscia do rozwiązania problemu takie jak lokalny spacer losowy czy wyczerpujące przeszukiwanie lokalne. Ze względu na to, że znajdowane przez nie wartosci uszeregowań były znacząco gorsze zrezygnowano z ich włączenia do obliczeń. Z drugiej strony mogą być one podstawą do dalszych modyfikacji.
Poniżej przedstawiono sygnatury metod oraz odpowiadające im procedury:

```
abstract class Bee
// markovSearch procedure
def localMarkovNeighborhoodSearch(sender: OutputChannel[Any],
    context: RunningContext, decorrelateSteps: Int)(best:
    List[TaskOffset[Double]]): List[TaskOffset[Double]]
// greedySearch procedure
def localGreedySearch(sender: OutputChannel[Any], context:
    RunningContext)(best: List[TaskOffset[Double]]):
    List[TaskOffset[Double]]
// globalSearch procedure
def globalSearch()(taskSet: List[TaskOffset[Double]]):
    List[TaskOffset[Double]]
```

5.1.5 Obiekty i procedury w implementacji algorytmu kukułki

Analogicznie do algorytmu pszczelego, w implementacji algorytmu kukułki głównymi aktorami (również z rozumieniu jednostek przetwarzających) są las (*forrest*) oraz zamieszkujące go kukułki (*cuckoos*).

Forrest odpowiada za przetwarzanie wiadomości otrzymywanych od kukułek (założono, że liczba kukułek jest tożsama z liczbą gniazd zgodnie z wnioskami z rozdziału 3.5) oraz utrzymuje najlepsze wyniki w kolejce priorytetowej o ustalonej długości n równej liczności kukułek. Schemat wymienianych wiadomości pomiędzy lasem a kukułkami przedstawiono na rysunku 5.3.



Rysunek 5.3: Komunikacja pomiędzy lasem a kukułkami. *Źródło: opracowanie własne.*

W porównaniu do algorytmu pszczelego schemat komunikacji jest uproszczony. Wiadomość *New* jest wysyłana do każdej kukułki tylko raz w celu określenia jej stanu początkowego.

Kukułka zawsze odpowiada wiadomością *Explored(localBest)* chyba, że

kukułka zakończyła swój cykl życia. Wówczas przesyłana jest wiadomość *Terminated*.
Za każdym razem, gdy kukułka prześle komunikat *Explored(localBest)* las proponuje kukułce kierunek dalszego przeszukiwania odsyłając wiadomość *Accept(best)*. Wartość *best* wybierana jest przez las zgodnie ze wspomnianą już procedurą *getBestNest* (rozdział 5.1.3).
Gdy ostatnia z kukułek zakończy swoje życie, wówczas las zwraca najlepsze znalezione uszeregowanie zadań.

Cuckoo odpowiada za przeszukiwanie przestrzeni rozwiązań i wyznacza C_{max}. Informację o znalezionych uszeregowaniach przesyła do obiektu *Forrest* (las). Pierwszą wiadomością otrzymywaną przez kukułkę jest *New*, która obsługiwana jest przez kod, który można opisać następującą prostą procedurą:

procedure REACTNEW($initialSchedule, iters$)
return (GLOBALSEARCH($initialSchedule, iters$))

Procedura *globalSearch* została już zdefiniowana dla algorytmu pszczelego. Ta sama jest ponownie użyta w bieżącej implementacji. Jedyna różnica dotyczy obiektu, do którego wysyłane są wiadomości *Explored*. W tym przypadku jest to *Forrest* zamiast *Hive*.
W dalszym swoim życiu kukułka otrzymuje jedynie wiadomości *Accept(best)*. Pierwszą rzeczą jaką robi, jest sprawdzenie czy poprzednie uszeregowanie zostało zaakceptowane przez gospodarza. Jeżeli tak się nie stało, kukułka jeszcze raz losuje dowolną permutację zadań (*globalSearch*), natomiast w przeciwnym wypadku aplikuje procedurę *cuckooSearch*.

global $currentSchedule, discoveredEggTreshold$
procedure REACTACCEPT($schedule, iters$)
if (RANDOM$(0,1) <$ $discoveredEggTreshold$)
then return (GLOBALSEARCH($currentSchedule, iters-1$))
else return (CUCKOOSEARCH($currentSchedule, schedule, iters-1$))

cuckooSearch jest jądrem algorytmu kukułki:

global *beta*
procedure CUCKOOSEARCH($currentSchedule, schedule, iters$)
if $(iters == 0)$
then SENDTO($forrest, Terminated$); EXIT()
$distance \leftarrow$ DIST($curentSchedule, schedule$)
$numberOfSteps \leftarrow$ LEVYFLIGHT($distance, beta$)
$newSchedule \leftarrow$ WALK($currentSchedule, schedule, numberOfSteps$)
if $(newSchedule == currentSchedule)$
then return (GLOBALSEARCH($currentSchedule, iters$))
else
$newSchedule \leftarrow$ COMPUTECMAX($newSchedule$)
SENDTO($forrest, Explored(newSchedule)$)
return $(newSchedule)$

Podstawową funkcją przedstawionej powyżej procedury jest wyznaczanie odległości pomiędzy bieżącym rozwiązaniem a rozwiązaniem najlepszym, przesłanym przez *Forrest* w wiadomości *Accept* i wyznaczenie dla tak znalezionego uszeregowania wartości C_{max}. Ponieważ problem odległości oraz poruszania się w przestrzeni kombinatorycznej w kierunku zadanego (najlepszego) uszeregowania został przedstawiony w rozdziale 4.1.3, uwaga zostanie skupiona na funkcji *LevyFlight*. Funkcja ta losuje liczbę kroków z przedziału $(0, distance)$ zgodnie z dystrybucją Levy'ego. Podstawą implementacji *LevyFlight* jest [61] napisana w języku *MatLab*, a która przepisana w *Scali* ma następującą postać:

```
// Levy flight implementation
object LevyFlight {
  def apply(distance: Int, beta: Double = 0.25) : Int = {
    val u = abs(nextGaussian * Sigma(beta))
    val v = abs(nextGaussian);
    val prob = min(1, pow(u / (u + v), 1 / beta))
    (distance.toDouble * prob).toInt
  }
```

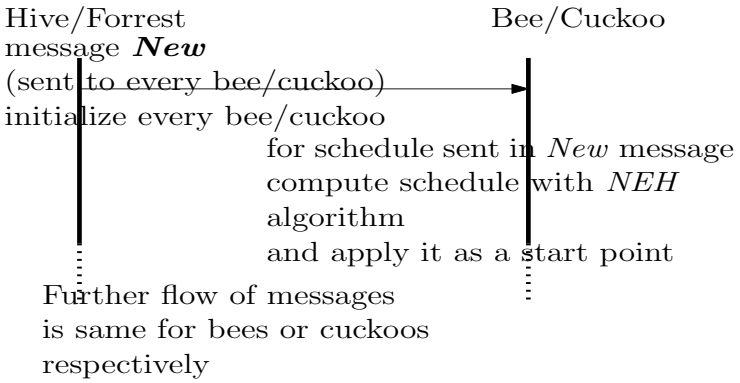
```
}
// with Sigma
object Sigma {
  import Gamma._
  import math._
  def apply(beta: Double) : Double = {
        pow(stGamma(1 + beta) * sin(Pi * beta / 2) /
            (stGamma((1 + beta) / 2) * beta * pow(2, ((beta -
            1) / 2))), 1 / beta)
  }
}
// and Gamma function
object Gamma {
  import scala.math._
  def stGamma(x:Double): Double = sqrt(2 * Pi / x) * pow((x /
      E), x)
}
```

Zauważmy, że do wyznaczenia $\Gamma(x)$ posłużono się aproksymacją Stirlinga (*stGamma*) o postaci $\sqrt{2\pi}n^{n+\frac{1}{2}}\exp^{-n}$.
Kończąc analizę procedury *cuckooSearch* warto zwrócić uwagę, że w przypadku gdy nowe uszeregowanie jest tożsame z bieżącym, wywoływana jest procedura *globalSearch* wyznaczająca nowy punkt początkowy do przeszukiwania przestrzeni rozwiązań.

Porównując implementacje algorytów pszczelego i kukułki, można przekonać się, że ten pierwszy jest nieznacznie bardziej skomplikowany. Z pewnością dużym ułatwieniem jest bardzo podobna architetura oraz możliwość wykorzystania tych samych metod (funkcji). Z punktu widzenia inżynierii oprogramowania można by w przyszłości zastanowić się nad wprowadzeniem wspólnej platformy programistycznej dla tego typu algorytmów.

5.1.6 Inicjalizowanie algorytmu listą zadań

Przedstawione powyżej algorytmy można zainicjalizować listą zadań otrzymaną w wyniku działania innej heurystyki. Jako przykład zaimplementowano algorytm *NEH* będący jednym z najefektywniejszych narzędzi do znajdowania uszeregowań bliskich optimum. Jego implementacja znajduje się w pakiecie *boids.neh*. Zastosowanie inicjalizacji wymaga niewielkich zmian w przedstawionych powyżej procedurach wywoływanych zarówno dla obiektów *bees* jak i *cuckoos* w momencie otrzymania wiadomości *New*. Rysunek 5.4



Rysunek 5.4: Inicjalizacja uszeregowaniem znalezionym przy pomocy algorytmu NEH. *Źródło: opracowanie własne.*

przedstawia zmianę w diagramach obsługi wiadomości, polegającą jedynie na zastąpieniu losowania permutacji początkowej przez wyznaczenie jej za pomocą algorytmu *NEH*. Procedury wywoływane w przypadku otrzymania wiadomości *New* są zaimplementowane następująco:

- dla algorytmu pszczelego
  procedure NEHORDER(*initialSchedule*)
  nehSchedule ← FINDSCHEDULEWITHNEHALGO(*initialSchedule*)
  return (*nehSchedule*)

- dla algorytmu kukułki
  procedure REACTNEW(*initialSchedule*, *iters*)

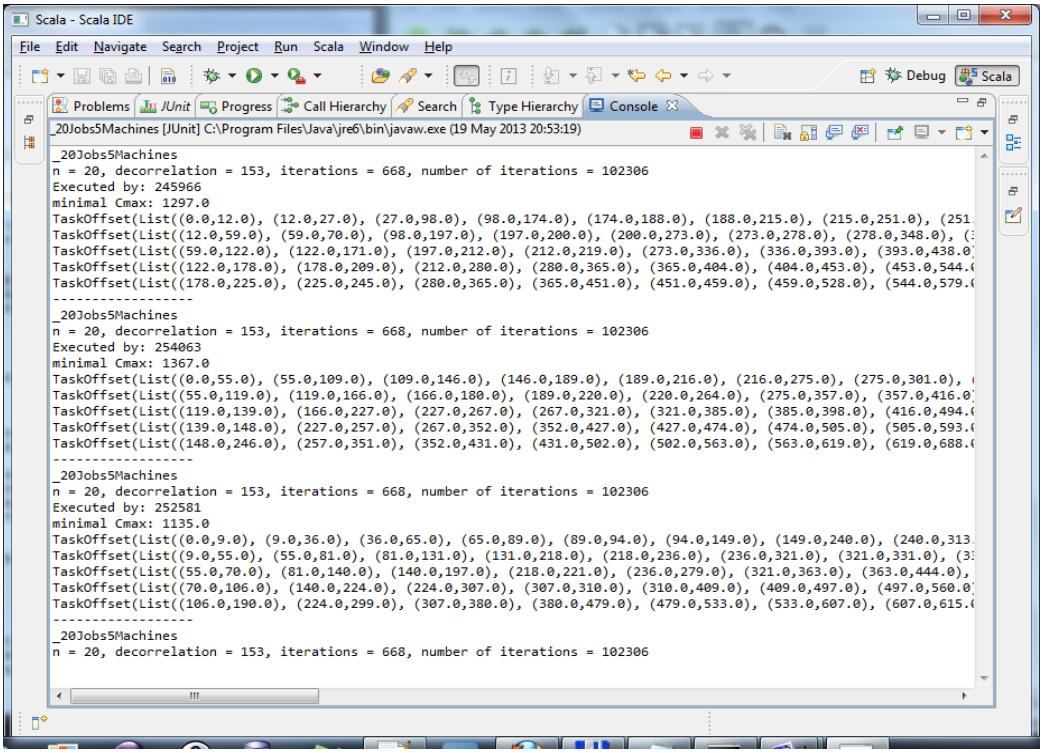
$nehSchedule \leftarrow$ findScheduleWithNehAlgo($initialSchedule$)
return (computeCMax($nehSchedule, iters$))

5.2 Przykładowe wyniki

Badania przeprowadzono na pojedynczej maszynie PC wyposażonej w processor Intel i7 z 8-ma logicznymi rdzeniami i 8GB pamięci.
Na rysunku 5.5 przedstawiono widok aplikacji podczas wykonywania testów algorytmu kukułki dla 20stu zadań i 5ciu procesorów (*_20Jobs5Machines.scala* w pakiecie *boids.cuckoo.taillard*).
Do testowania poprawności znajdowanych rozwiązań użyto prób zespołu Erica Taillarda [54]. Badania przeprowadzono na zbiorach danych testowych dla:

- 20stu zadań i odpowiednio 5ciu oraz 10ciu maszyn
- 50ciu zadań i odpowiednio 5ciu oraz 10ciu maszyn
- 100 zadań i odpowiednio 5ciu oraz 20stu maszyn

Ze względów czasowych nie zdecydowano się na badania większej ilości zadań.
Badania przeprowadzano dla przypadków inicjalizowania przeszukiwań permutacyjnej przestrzeni rozwiązań uszeregowaniem losowym oraz znalezionym przy pomocy algorytmu *NEH*.
Warto zauważyć, że próby zespołu Erica Taillarda uznaje się za jedne z trudniejszych przestrzeni kombinatorycznych dla algorytmów stochastycznych rozwiązujących problem $Fm|prmu|C_{max}$. Cecha ta czyni je dobrym wskaźnikiem siły testowanego algorytmu i stąd też zostały wybrane do weryfikacji zaimplementowanych algorytmów.



Rysunek 5.5: Ogólny widok *Typesafe IDE* podczas wykonywania testów. *Źródło: opracowanie własne.*

5.2.1 Ustawienia parametrów testów dla algorytmu pszczelego

- pozostawino domyślne ustawienia parametrów *actors.corePoolSize* równy 4 oraz *actors.maxPoolSize* równy 256;
- liczbę pszczół *bees* ustawiono arbitralnie na 50 badając obciążenie procesora (tak, aby był wykorzystywany w granicach 40-80%). Długość kolejki priorytetowej z zapamiętanymi najlepszymi wynikami jest równa liczbie pszczół;
- ze względu na relatywnie niewielką ilość pszczół przyjęto, że liczba elitarnych pszczół przeszukujących najlepsze rozwiązania ne jest równa liczbie najlepszych rozwiązań w iteracji e. Podobnie liczba pszczół przeszukujących obszary rokujące nm jest równa liczbie obszarów rokują-

cych (z wyłączeniem najlepszych) $m - e$;

- przyjęto arbitralnie $alpha = 0.1$, $delta = 0.01$, $epsilon = 0.01$, liczba obszarów najlepszych $e = 5$, liczba obszarów rokujących (włączając obszary najlepsze) $m = 10$;

Wszystkie przypadki zostały zapisane jako testy jednostkowe i są dostępne w lokalizacji *src/test/scala/boids/taillard*. Dodatkowo zdefiniowano test *TaillardAll.scala*, który uruchamia wszystkie testy.

Średnie czasy wykonania $t_{i,j}^{avg}$ (gdzie i oznacza ilosć maszyn a j zadań) poszczególnych testów wynoszą odpowiednio:

- dla $i = 5$ oraz $j = 20$ $t_{5,20}^{avg} = 461$ sekund
- dla $i = 5$ oraz $j = 50$ $t_{5,50}^{avg} = 1073$ sekundy
- dla $i = 5$ oraz $j = 100$ $t_{5,100}^{avg} = 1975$ sekund
- dla $i = 10$ oraz $j = 20$ $t_{10,20}^{avg} = 487$ sekund
- dla $i = 10$ oraz $j = 50$ $t_{10,50}^{avg} = 1207$ sekund
- dla $i = 20$ oraz $j = 100$ $t_{20,100}^{avg} = 2975$ sekund

Ze względu na stosunkowo długie czasy wykonywania każdy z testów powtórzony był $k = 5$ razy. Ponieważ taka próba jest zbyt mała, aby można z niej wnioskować o zależnosciach statystycznych, ograniczono się jedynie do wyznaczenia średniego błędu względnego, jego wartosci minimalnej i maksymalnej w celu zobrazowania potencjalnych możliwosci algorytmu. Dodatkowo dla przypadku, w którym maksymalny błąd był największy badanie powtórzono dla $\epsilon = 0.001$.
W tabelach 6.1, 6.2, 6.3 oraz 6.4 (rozdział 6.2) przedstawiono otrzymane rezultaty działania aplikacji.

5.2.2 Ustawienia parametrów testów dla algorytmu kukułki

- pozostawino domyślne ustawienia parametrów *actors.corePoolSize* równy 4 oraz *actors.maxPoolSize* równy 256;
- liczbę kukułek *cuckoos* ustawiono arbitralnie na 50 badając obciążenie procesora (tak, aby był wykorzystywany w granicach 40-80%). Długość kolejki priorytetowej z zapamiętanymi najlepszymi wynikami jest kontrolowana przez parametr *bestNests* i arbitralnie ustawiona na 5;
- przyjęto arbitralnie $alpha = 0.1$, $delta = 0.01$, $epsilon = 0.01$, prawdopodobieństwo odkrycia obcego jaja przez gospodarza $discoveredEggTreshold = 0.25$, prawdopodobieństwo akceptacji najlepszego gniazda przez kukułkę $acceptBestNest = 0.8$, przyjęto za [61] $beta = 0.25$;

Wszystkie przypadki zostały zapisane jako testy jednostkowe i są dostępne w lokalizacji *src/test/scala/boids/cuckoo/taillard*. Dodatkowo zdefiniowano test *TaillardAll.scala*, który uruchamia wszystkie testy.

Średnie czasy wykonania $t_{i,j}^{avg}$ (gdzie i oznacza ilosć maszyn a j zadań) poszczególnych testów wynoszą odpowiednio:

- dla $i = 5$ oraz $j = 20$ $t_{5,20}^{avg} = 264$ sekund
- dla $i = 5$ oraz $j = 50$ $t_{5,50}^{avg} = 4023$ sekundy
- dla $i = 5$ oraz $j = 100$ $t_{5,100}^{avg} = 3411$ sekund
- dla $i = 10$ oraz $j = 20$ $t_{10,20}^{avg} = 389$ sekund
- dla $i = 10$ oraz $j = 50$ $t_{10,50}^{avg} = 2873$ sekund
- dla $i = 20$ oraz $j = 100$ $t_{20,100}^{avg} = 4850$ sekund

Ze względu na stosunkowo długie czasy wykonywania każdy z testów powtórzony był $k = 4$ razy. Podobnie jak w przypadku algorytmu pszczelego próba

jest zbyt mała, aby można z niej wnioskować o zależnosciach statystycznych, dlatego ograniczono się jedynie do wyznaczenia średniego błędu względnego, jego wartosci minimalnej i maksymalnej w celu zobrazowania potencjalnych możliwosci algorytmu.
W tabelach 6.5, 6.6, 6.7 oraz 6.8 (rozdział 6.2) przedstawiono otrzymane rezultaty działania aplikacji.

5.2.3 Ustawienia dla inicjalizacji wynikiem działania algorytmu *NEH*

W pracy badano pobieżnie zależność między otrzymaną wartością optymalną a początkowym uszeregowaniem zadań. W tym celu zmodyfidowano ich implementacje tak, aby były inicjalizowane permutacją zadań otrzymanych w wyniku działania heurystyki *NEH*.
Ustawienia parametrów pozostawiono bez zmian.
Dla algorytmu pszczelego testy wykonano dla przypadków:

- 5 maszyn i 20 zadań
- 5 maszyn i 50 zadań
- 10 maszyn i 20 zadań

Dla algorytmu kukułki testy wykonano dla przypadku:

- 5 maszyn i 20 zadań

Średnie czasy wykonania $t_{i,j}^{avg}$ (gdzie i oznacza ilosć maszyn a j zadań) poszczególnych testów wynoszą odpowiednio:

- w przypadku algorytmu pszczelego
  - dla $i = 5$ oraz $j = 20$ $t_{5,20}^{avg} = 321$ sekund
  - dla $i = 5$ oraz $j = 50$ $t_{5,50}^{avg} = 2825$ sekund
  - dla $i = 10$ oraz $j = 20$ $t_{10,20}^{avg} = 1602$ sekundy
- w przypadku algorytmu kukułki

– dla $i = 5$ oraz $j = 20$ $t^{avg}_{5,20} = 347$ sekund

Każdy z testów powtórzony był $k = 2$ razy. Wyznaczono różnicę pomiędzy najlepszym rozwiązaniem znalezionym przez odpowiedni algorytm inicjalizowany przez uszeregowanie losowe oraz znalezione przez *NEH*.
W tabelach 6.9, 6.10 oraz 6.11 (rozdział 6.3) przedstawiono otrzymane rezultaty działania aplikacji.

5.2.4 Badanie zbieżności stochastycznej

W końcu w pracy badano pobieżnie zbieżność stochastyczną algorytmów inicjalizowanych uszeregowaniem losowym.
Testy wykonano na dziewiątym przypadku Taillard'a dla 5 maszyn i 20 zadań.
Implementacja testów znajduje się w pakiecie *src/test/scala/boids/cohesion*.
Każdy z dwóch testów (odpowiednio dla algorytmu pszczelego oraz kukułki) został powtórzony $k = 20$ razy.
Rysunki 6.3 oraz 6.4 (rozdział 6.4) prezentują otrzymane wyniki badania.

Rozdział 6

Dyskusja wyników i dalsze kierunki badań

Zanim zinterpretowane zostaną wyniki doświadczeń, zastanówmy się jeszcze przez chwilę o czym mówi nierówność Chernoffa zastosowana do wyznaczania liczby iteracji. Jak pamiętamy ma ona postać

$$P(|X - \mu| \geqslant \epsilon\mu) \leqslant \delta,$$

gdzie $\mu = E(X)$. Warto zauważyć, że chociaż μ nie jest wartością optymalną, to daje o niej pewną wskazówkę. Wyobraźmy sobie dyskretną przestrzeń, w której wszystkie punkty poza jednym mają taką samą wartość. Gdy przestrzeń taka jest odpowiednio duża to z reguły wyznaczona wartość oczekiwana będzie po prostu równa wartości punktów (poza jednym). Co więcej, można powiedzieć, że dla pozostałych, nieprzejrzanych punktów przestrzeni istnieje jedynie pewne prawdopodobieństwo δ, dla którego będą się one różnić bardziej niż o ϵ od tych przeszukanych. Stąd też **z reguły** wkład wartości optymalnej do wartości oczekiwanej nie powinien być znaczący. Oczywiście w granicznym przypadku, powyższa przestrzeń może mieć rozkład Diraca i wtedy jakiekolwiek badanie stochastyczne (poza bezpośrednim wylosowaniem optimum) będzie błędne. Z drugiej strony i inne podejścia, w tym deterministyczne z przeszukiwaniem przestrzeni rozwiązań, są równie bezradne. Stąd wniosek, że nierówność Chernoffa można wykorzystać do szacowania ilości

kroków iteracji z tym, że trzeba pamiętać, że dla pewnych przestrzeni może ona okazać się niewystarczająca.
Porównując wyniki otrzymane dla zaimplementowanych algorytmów pszczelego oraz kukułki okazuje się, że z reguły ten pierwszy lepiej radzi sobie z przeszukiwanymi przestrzeniami. Jest to związane z trudnością analizy przestrzeni kombinatorycznych, w których odległość (tak jak została zdefiniowana) jest słabo powiązana z topografią funkcji celu. Innymi słowy wydaje się, że to co jest uznawane za sąsiedztwo w rzeczywistości nim nie jest. Prawdopodobnie stąd też bardziej efektywny wydaje się być spacer losowy od przeszukiwania ukierunkowanego przez lokalnie najlepsze rozwiązanie. Interesujące podejście do wyznaczenia kroku pośredniego π_{12} pomiędzy permutacjami π_1 oraz π_2 przedstawiono w pracy [22], które można opisać następującym algorytmem:

comment: γ arbitralny współczynnik proporcjonalności

procedure FINDPERMUTATIONBETWEEN(π_1, π_2)
$dist \leftarrow$ HAMMINGDISTANCE(π_1, π_2)
$\beta \leftarrow 1/(1 + \gamma dist)$
$\pi_{12} \leftarrow$ [ ] o rozmiarze π
przepisz do π_{12} te pozycje i, gdzie $\pi_1(i) = \pi_2(i)$
for $k \leftarrow 0$ **to** $(dist - 1)$
do
- wylosuj z p-stwem $1/(dist - k)$ indeks j niewypełnionej pozycji
- $p \leftarrow$ RANDOM$(0, 1)$
- **if** $p > \beta$
  - **then**
    - **if** $\pi_1(j) \notin \pi_{12}$
      - **then** $\pi_{12}(j) \leftarrow \pi_1(j)$
  - **else**
    - **if** $\pi_2(j) \notin \pi_{12}$
      - **then** $\pi_{12}(j) \leftarrow \pi_2(j)$

jeżeli pozostały w π_{12} nieobsadzone pozycje
dolosuj dla nich niewykorzystane wartości

Podejście to mogłoby być użyte w algorytmie kukułki zamiast istniejącego ukierunkowanego spaceru. Należy jednak zwrócić uwagę, że sąsiedztwo nie jest w tym przypadku ściśle zdefiniowane. To z kolei wpływa na wyznaczenie

ilości kroków algorytmu z nierówności Chernoffa, ponieważ nie wiadomo jak zdefiniować operację sprzężenia dla takiego przypadku. **Stąd wniosek, że w przyszłych badaniach można zastanowić się nad problemem ilości próbek dla tak zdefiniowanego spaceru.**

6.1 Obserwacje dotyczące równoległego poszukiwania optymalnego harmonogramu

Ze względu na to, że zaimplementowane algorytmy badane były na komputerze jednoprocesorowym (z ośmioma rdzeniami logicznymi), trudno jest przeprowadzić dokładniejszą analizę zależności czasowych pomiędzy wątkami *ula/lasu* (*hive/forrest*) a *pszczół/kukułek* (*bees/cuckoos*). Domyślny algorytm tworzenia i przydziału wątków wbudowany w bibliotekę języka *Scala* tworzy tyle wątków, ile jest dostępnych rdzeni czyli w rozważanym przypadku osiem. Stąd wynika, że wiele *aktorów* współdzieliło ten sam wątek. Pomimo tego ograniczenia udało się zaobserwować, że czas wymiany wiadomości pomiędzy *ulem/lasem* a *pszczołami/kukułkami* relatywnie maleje w stosunku do czasu przetwarzania uszeregowania przez *pszczołę/kukułkę*. Oznacza to, że nakład obliczeń potrzebnych do wyznaczenia nowego harmonogramu rośnie szybciej niż obsługa wiadomości otrzymanej przez *ul/las*. Stąd też wynika, że dla badanego zakresu danych wejściowych to wątki wyliczające uszeregowania wpływają decydująco na całkowity czas obliczeń. Ten czas wynosi kT, gdzie T jest czasem dla obliczeń wykonywanych sekwencyjnie, natomiast $k \in (\frac{1}{c-1}, \frac{1}{c})$, gdzie c z kolei jest liczbą rdzeni procesora. Dzieje się tak dlatego, że złożoność obliczeniowa operacji wykonywanych przez *ul/las* jest mniejsza od tych, które są wykonywane przez *pszczołę/kukułkę*. Wystarczy przypomnieć sobie, że złożoność wyznaczania uszeregowania wynosi $o(nm)$, z kolei operacja wstawiania znalezionego rozwiązania do kolejki priorytetowej (wykonywana przez *ul/las*) ma złożoność $o(d)$, gdzie d jest długością kolejki (w badaniach d wynosiło 50, natomiast $nm \in [100, 2000]$).
Ze względu na to, że wielkość przesyłanych wiadomości nie przekraczała 100kB, czas potrzebny na alokację bufora można również zaniedbać.

6.2 Ogólne obserwacje dotyczące algorytmów inicjalizowanych uszeregowaniem losowym

W tabelach od 6.1 do 6.8 zebrano wyniki doświadczeń dla algorytów pszczelego i kukułki inicjalizowanych uszeregowaniem losowym.
Rozpatując przypadek pięciu maszyn widać, że **minimalny** błąd względny może nawet nie przekraczać wartości **0.003** (tabela 6.1), co sprawia, że zaproponowane rozwiązania wydają się być bardzo obiecujące. Jednak z drugiej strony **maksimum** błędu wynosi aż **0.0496** (tabela 6.5), co znacznie przekracza założoną wartość $\epsilon = 0.01$. Dla najgorszych przypadków (opowiednio dla algorytmu pszczelego i kukułki) dokonano dodatkowego badania dla $\epsilon = 0.001$ (co oznacza 100 razy więcej próbek). Krok ten nie przyczynił się jednak do poprawy znalezionej wartości optymalnej. Może to oznaczać, że w tym konkretnym przypadku przestrzeń rozwiązań jest niepodatna na którykolwiek z dwóch zaimplementowanych algorytmów i należałoby zastosować inne podejście do problemu.
Patrząc na wyniki dla dziesięciu i dwudziestu maszyn tym, co uderza najbardziej, to wartość błędu względnego, dużo większa od przyjętego ϵ. Podczas analizy problemu i implementacji algorytmów założono, że obliczenia nie zależą od m (ilości maszyn), ponieważ C_i (maksymalny czas przetwarzania zadań przez *i-tą* maszynę) jest zdeterminowane jedynie przez kolejność zadań. Z otrzymanych wyników można wnioskować, że założenie to nie jest w pełni poprawne. Właściwszym byłoby założenie, że wartości C_i (dla wszystkich maszyn) są również zmiennymi losowymi. W rozdziale 4 pokazano, że złożoność obliczeniowa dla zaimplementowanego algorytmu wynosi $o(c(n,m)n^2 \log n)$, gdzie $c(n,m)$ jest kosztem wyznaczenia C_{max} oraz że $o(c(n,m)) = o(nm)$. Stąd też przez prostą ekstrapolację można by domniemywać, że, po uwzględnieniu postulatu odnośnie natury C_i, złożoność powinna wynosić $o((nm)n^2m^2 \log n \log m)$. Rzeczywista analiza jest jednak trudniejsza, ponieważ powinna ona uwzględniać fakt, że C_i jest zmienną losową zależną od uszeregowania. W dalszej pracy nad usprawnianiem algorytmu należałoby zbadać również zależność stochastyczną pomiędzy uszeregowaniem a wyznaczoną wartością $\mathbf{C_i}$ i uwzględnić ją podczas wyznaczania liczby ite-

racji w algorytmie.

Oba te problemy bardzo mocno wpływają na złożoność algorytmu. Nie należy także zapominać o parametrach α, δ i ϵ, z których ten ostatni jest szczególnie istotny ze względu na to, że jest poniesiony do kwadratu. Jeżeli przyjąć, że C_i jest zmienną losową, a złożoność będzie wynosić $o((nm)n^2m^2 \log n \log m)$ to dla $n = m = 100$ liczba operacji będzie niemniejsza jak $4 * 10^{15}$ a liczba iteracji odpowiednio $4 * 10^{11}$. Liczby te są na tyle duże, że obliczenia na pojedynczym komputerze będą trwać długie godziny (stan na 2013). Z drugiej strony dyskutowana implementacja algorytmów stadnych jest wielowątkowa, a więc do zastosowania w obliczeniach rozproszonych. Podczas testów ilość pszczół ustawiono na 50, natomiast rzeczywiste wielkości rojów zawierają się w przedziale od 20-stu do nawet 50-ciu tysięcy osobników. Stąd też liczba pszczół eksplorujących jak i eksploatujących powinna być o około trzy rzędy wielkości większa, tak aby strategia wypracowana przez ewolucję zachowania pszczół mogła zostać rzeczywiście zasymulowana. Z nieco inną sytuacją mamy do czynienia w przypadku algorytmu kukułki. Ustalona wielkość populacji równa 50 mogłaby być również większa, jednakże nie odbiega ona znacząco od obserwacji w naturze. Z drugiej strony należy pamiętać, że jest to jedynie model, którego zasadnicza różnica w porównaniu z algorytmem pszczelim wyraża się w zastąpieniu spaceru losowego lotem Levy'ego. W dalszym rozwoju aplikacji należałoby przenieść obie implementacje do wersji 2.10 języka *Scala* i testować ją w rzeczywistym środowisku rozproszonym (biblioteka *Akka*) dla dużo większych populacji. Dodatkowo należałoby zbadać wpływ ustawień me oraz mn dla algorytmu pszczelego, a β i $discoveredEggTreshold$ w przypadku algorytmu kukułki, na znalezioną wartość optymalną uszeregowania.

W rozdziale 4.4 dyskutowano o złożoności wyznaczania C_{max}. Wykazano, że działanie $\circ$ nie jest działaniem przemiennym. Stąd też nie należy do klasy *NC*, a jego złożoność wynosi $o(nm)$. Można zastanowić się nad tym, czy istnieje równoległy algorytm wyznaczania C_{max} (o złożoności polilogarytmicznej) albo udowodnić, że taki algorytm nie istnieje. Jednakże czynnik wnoszony do ogólnej złożoności jest relatywnie niewielki ($o(m \log n)$ w miejsce $o(mn)$) i raczej należy się spodziewać, że ulepszenie tej części implementacji nie spo-

woduje radykalnego skrócenia czasu pracy algorytmu.

C_{max}	C^{opt}_{max}	$\frac{C_{max}-C^{opt}_{max}}{C^{opt}_{max}}$	C_{max}	C^{opt}_{max}	$\frac{C_{max}-C^{opt}_{max}}{C^{opt}_{max}}$
$j = 20$			$j = 50$		
1297	1278	0.0148	2763	2724	0.0143
1363	1359	0.0029	2920	2834	0.0303
1130	1081	0.0453	2700	2621	0.0301
1346	1293	0.0409	2846	2751	0.0345
1258	1235	0.0186	2890	2863	0.0094
1227	1195	0.0267	2875	2829	0.0162
1251	1234	0.0137	2823	2725	0.0359
1255	1206	0.0406	2755	2683	0.0268
1277	1230	0.0382	2647	2552	0.0372
1161	1108	0.0478	2833	2782	0.0183
średnia błędów		0.029	średnia błędów		0.0253
błąd min		0.0029	błąd min		0.0094
błąd max		0.0478	błąd max		0.0372

Tabela 6.1: **Algorytm pszczeli.** Znalezione wartości C_{max} dla $i = 5$ maszyn oraz $j = 20$ i $j = 50$ zadań.

C_{max}	C^{opt}_{max}	$\frac{C_{max}-C^{opt}_{max}}{C^{opt}_{max}}$
$j = 100$		
5560	5493	0.0121
5364	5268	0.0182
5312	5175	0.0264
5126	5014	0.0223
5395	5250	0.0276
5215	5135	0.0155
5346	5246	0.0190
5222	5094	0.0251
5570	5448	0.0223
5449	5322	0.0238
średnia błędów		0.021
błąd min		0.012
błąd max		0.027

Tabela 6.2: **Algorytm pszczeli.** Znalezione wartości C_{max} dla $i = 5$ maszyn oraz $j = 100$ zadań.

C_{max}	C^{opt}_{max}	$\frac{C_{max}-C^{opt}_{max}}{C^{opt}_{max}}$	C_{max}	C^{opt}_{max}	$\frac{C_{max}-C^{opt}_{max}}{C^{opt}_{max}}$
$j = 20$			$j = 50$		
1682	1582	0.0632	3334	2991	0.1146
1777	1659	0.0711	3226	2867	0.1252
1609	1496	0.0755	3233	2839	0.1387
1486	1377	0.0791	3346	3063	0.0923
1540	1419	0.0852	3330	2976	0.1189
1484	1397	0.0622	3328	3006	0.1071
1572	1484	0.0592	3346	3093	0.0817
1663	1538	0.0812	3303	3037	0.0875
1656	1593	0.0395	3206	2897	0.1066
1680	1591	0.0559	3372	3065	0.1001
średnia błędów		0.0672	średnia błędów		0.1073
błąd min		0.0395	błąd min		0.0817
błąd max		0.0852	błąd max		0.1387

Tabela 6.3: **Algorytm pszczeli.** Znalezione wartości C_{max} dla $i = 10$ maszyn oraz $j = 20$ i $j = 50$ zadań.

C_{max}	C^{opt}_{max}	$\frac{C_{max}-C^{opt}_{max}}{C^{opt}_{max}}$
$j = 100$		
7011	6202	0.1304
7123	6183	0.1520
7058	6271	0.1254
7116	6269	0.1351
7170	6314	0.1355
7211	6364	0.1330
7207	6268	0.1498
7319	6401	0.1434
7180	6275	0.1442
7236	6434	0.1246
średnia błędów		0.1373
błąd min		0.1246
błąd max		0.1520

Tabela 6.4: **Algorytm pszczeli.** Znalezione wartości C_{max} dla $i = 20$ maszyn oraz $j = 100$ zadań.

C_{max}	C^{opt}_{max}	$\frac{C_{max}-C^{opt}_{max}}{C^{opt}_{max}}$	C_{max}	C^{opt}_{max}	$\frac{C_{max}-C^{opt}_{max}}{C^{opt}_{max}}$
$j = 20$			$j = 50$		
1297	1278	0.0148	2752	2724	0.0103
1366	1359	0.0051	2936	2834	0.0359
1132	1081	0.0471	2696	2621	0.0286
1351	1293	0.0448	2837	2751	0.0312
1258	1235	0.0186	2903	2863	0.0139
1226	1195	0.0259	2909	2829	0.0282
1250	1234	0.0129	2831	2725	0.0388
1264	1206	0.048	2778	2683	0.0354
1281	1230	0.0414	2636	2552	0.0329
1163	1108	0.0496	2839	2782	0.0204
średnia błędów		0.031	średnia błędów		0.0276
błąd min		0.0051	błąd min		0.0103
błąd max		0.0496	błąd max		0.0388

Tabela 6.5: **Algorytm kukułki.** Znalezione wartości C_{max} dla $i = 5$ maszyn oraz $j = 20$ i $j = 50$ zadań.

C_{max}	C^{opt}_{max}	$\frac{C_{max}-C^{opt}_{max}}{C^{opt}_{max}}$
$j = 100$		
5564	5493	0.0129
5383	5268	0.0218
5313	5175	0.0266
5139	5014	0.0249
5409	5250	0.0302
5229	5135	0.0183
5368	5246	0.0232
5244	5094	0.0294
5585	5448	0.0251
5456	5322	0.0251
średnia błędów		0.0237
błąd min		0.0129
błąd max		0.0302

Tabela 6.6: **Algorytm kukułki.** Znalezione wartości C_{max} dla $i = 5$ maszyn oraz $j = 100$ zadań.

C_{max}	C^{opt}_{max}	$\frac{C_{max}-C^{opt}_{max}}{C^{opt}_{max}}$	C_{max}	C^{opt}_{max}	$\frac{C_{max}-C^{opt}_{max}}{C^{opt}_{max}}$
$j = 20$			$j = 50$		
1684	1582	0.0644	3386	2991	0.132
1765	1659	0.0638	3213	2867	0.1206
1594	1496	0.0655	3262	2839	0.1489
1472	1377	0.0689	3381	3063	0.1038
1518	1419	0.0697	3379	2976	0.1354
1490	1397	0.0665	3369	3006	0.1207
1581	1484	0.0653	3414	3093	0.1037
1641	1538	0.0669	3345	3037	0.1014
1672	1593	0.0495	3287	2897	0.1346
1705	1591	0.0716	3426	3065	0.1177
średnia błędów		0.0652	średnia błędów		0.1219
błąd min		0.0495	błąd min		0.1014
błąd max		0.0716	błąd max		0.1489

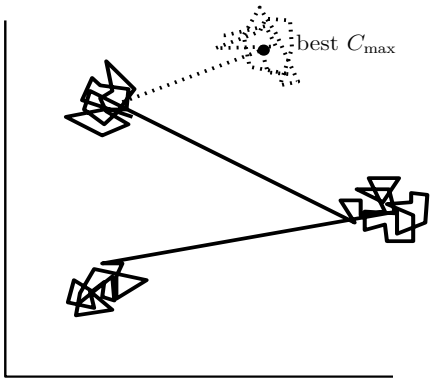
Tabela 6.7: **Algorytm kukułki.** Znalezione wartości C_{max} dla $i = 10$ maszyn oraz $j = 20$ i $j = 50$ zadań.

C_{max}	C^{opt}_{max}	$\frac{C_{max}-C^{opt}_{max}}{C^{opt}_{max}}$
$j = 100$		
7135	6202	0.1504
7196	6183	0.1638
7228	6271	0.1526
7130	6269	0.1373
7198	6314	0.14
7236	6364	0.137
7253	6268	0.1571
7324	6401	0.1441
7093	6275	0.1303
7270	6434	0.1299
średnia błędów		0.1442
błąd min		0.1299
błąd max		0.1638

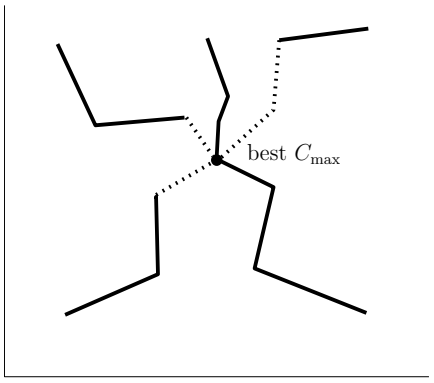
Tabela 6.8: **Algorytm kukułki.** Znalezione wartości C_{max} dla $i = 20$ maszyn oraz $j = 100$ zadań.

6.2.1 Porównanie efektywności działania algorytmu pszczelego z algorytmem kukułki

Patrząc na wyniki doświadczeń zebranych w tabelach 6.1 do 6.8 można postawić hipotezę, że przeszukiwanie przestrzeni rozwiązań algorytmem pszczelim jest efektywniejszym sposobem na znalezienie wartości optymalnej od algorytmu kukułki. Rysunki 6.1 oraz 6.2 mogą dać pewne wyobrażenie, dlaczego tak się dzieje.



Rysunek 6.1: Trajektoria przeszukiwań permutacyjnej przestrzeni rozwiązań przez pszczołę. *Źródło: opracowanie własne.*



Rysunek 6.2: Trajektoria przeszukiwań przestrzeni rozwiązań przez kukułkę dla 4-elementowej przestrzeni permutacyjnej. Oznacza to, że najmniejsza odległość pomiędzy punktami (tak jak została zdefiniowana w rozdziale 4.1.3) wynosi 3. *Źródło: opracowanie własne.*

Przypomnijmy, że w przypadku algorytmu pszczelego, pojedyncza pszczoła przeszukuje przestrzeń losując jedno z sąsiedztw bieżącego punktu. Jak pamiętamy, liczbę tych losowań określono z użyciem sprzężonych łańcuchów Markowa. Ta faza przeszukiwania sąsiedztwa jest przedstawiona na rysunku 6.1 jako kłębki trajektorii w odległości nie większej niż zadane r od bieżącego punktu. Po tej fazie (w której znajduje się stochastycznie lokalnie najlepsze rozwiązanie) następuje decyzja albo o akceptacji najlepszego globalnie rozwiązania (*best* $C_{\max}$), albo o losowym wyborze innego punktu przestrzeni, którego sąsiedztwo jest z kolei badane przez pszczołę. Na rysunku 6.1 skoki losowe zaznaczono liniami ciągłymi, natomiast skok wynikający z akceptacji globalnie najlepszego rozwiązania linią przerywaną.
W przypadku algorytmu kukułki schemat przeszukiwania jest odmienny. Najpierw losowany jest dowolny punkt przeszukiwanej przestrzeni p a następnie, kukułka losuje liczbę kroków k z przedziału $[0, n]$ w kierunku aktualnie globanie najlepszego rozwiązania. W przypadku wylosowania zera lub n losowanie punktu p jest powtórzone. Następnie liczona jest wartość funkcji celu w punkcie $p + k$. Schemat ten został przedstawiony na rysunku 6.2.
Teraz wystarczy odpowiedzieć sobie na pytanie, który z tych schematów bardziej nadaje się do przeszukiwania zadanej przestrzeni. Generanie można rozważyć w tym miejscu dwa skrajne przypadki. Przestrzeń zdominowaną przez bardzo strome, wąskie góry oraz przestrzeń zdominowaną przez rozległe, łagodne pagórki. Dość oczywistą konstatacją będzie powiedzenie, że pierwsza z przestrzeni efektywniej będzie przeszukiwana przez algorytm pszczeli natomiast druga przez algorytm kukułki. Natomiast z przewagi algorytmu pszczelego nad kukułki wynika, że w przypadku testów Taillard'a, strome góry dominują łagodne pagórki (dla zdefiniowanego w pracy sąsiedztwa).

6.3 Wpływ początkowej permutacji na wartość końcową

W tabelach 6.9 do 6.11 zebrano wyniki testów dla algorytmów inicjalizowanych rozwiązaniem znalezionym za pomocą heurystyki *NEH*. Przede

wszystkim, inicjalizacja przez *NEH* z reguły prowadziła do znalezienia lepszego rozwiązania optymalnego w porównaniu do algorytmu inicjalizowanego losowym harmonogramem. Z obserwacji tej wynika, że badane implementacje algorytmów pszczelego oraz, w mniejszym zakresie, kukułki są algorymami obciążonymi, w rozumieniu, że znaleziona wartość optymalna (która, jak to już zostało wyjaśnione, nie jest tożsama z wartością oczekiwaną) zależy od punktu początkowego poszukiwań. Ponieważ względny błąd znalezionego w ten sposób rozwiązania może być nawet blisko dwa razy mniejszy jak w przypadku inicjalizacji losowej, stąd też, w ramach ulepszenia algorytmu, można się zastanowić nad innymi heurystykami, których rezultaty mogłyby tworzyć zbiór punktów początkowych poszukiwań.

NEH	C_{max}	C_{max}^{opt}	$\frac{C_{max}-C_{max}^{opt}}{C_{max}^{opt}}$	NEH	C_{max}	C_{max}^{opt}	$\frac{C_{max}-C_{max}^{opt}}{C_{max}^{opt}}$
$j = 20$				$j = 50$			
1286	1286	1278	0.0062	2733	2729	2724	0.0018
1365	1365	1359	0.0044	2882	2868	2834	0.0119
1140	1127	1081	0.0425	2625	2625	2621	0.0015
1340	1329	1293	0.0278	2782	2782	2751	0.0112
1305	1261	1235	0.0210	2868	2868	2863	0.0017
1228	1224	1195	0.0242	2840	2840	2829	0.0038
1279	1251	1234	0.0137	2776	2746	2725	0.0077
1235	1223	1206	0.0140	2703	2703	2683	0.0074
1291	1262	1230	0.0260	2574	2574	2552	0.0086
1151	1136	1108	0.0252	2822	2809	2782	0.0097
średnia błędów			0.0206	średnia błędów			0.0065
błąd min			0.0044	błąd min			0.0015
błąd max			0.0425	błąd max			0.012

Tabela 6.9: **Algorytm pszczeli inicjalizowany heurystyką NEH.** Znalezione wartości C_{max} dla $i = 5$ maszyn oraz $j = 20$ i $j = 50$ zadań.

NEH	C_{max}	C_{max}^{opt}	$\frac{C_{max}-C_{max}^{opt}}{C_{max}^{opt}}$
$j = 10$			
1680	1659	1582	0.0486
1786	1739	1659	0.0482
1557	1556	1496	0.0401
1450	1442	1377	0.0472
1502	1502	1419	0.0584
1453	1441	1397	0.0314
1562	1540	1484	0.0377
1609	1591	1538	0.0344
1647	1637	1593	0.0276
1653	1649	1591	0.0364
średnia błędów			0.041
błąd min			0.0276
błąd max			0.0584

Tabela 6.10: **Algorytm pszczeli inicjalizowany heurystyką NEH.** Znalezione wartości C_{max} dla $i = 20$ maszyn oraz $j = 10$ zadań.

NEH	C_{max}	C_{max}^{opt}	$\frac{C_{max}-C_{max}^{opt}}{C_{max}^{opt}}$
$j = 5$			
1286	1286	1278	0.0062
1365	1365	1359	0.0044
1140	1126	1081	0.0416
1340	1340	1293	0.0363
1305	1275	1235	0.0323
1228	1228	1195	0.0276
1279	1254	1234	0.0162
1235	1235	1206	0.0240
1291	1286	1230	0.0455
1151	1151	1108	0.0388
średnia błędów			0.0273
błąd min			0.0044
błąd max			0.0455

Tabela 6.11: **Algorytm kukułki inicjalizowany heurystyką NEH.** Znalezione wartości C_{max} dla $i = 20$ maszyn oraz $j = 5$ zadań.

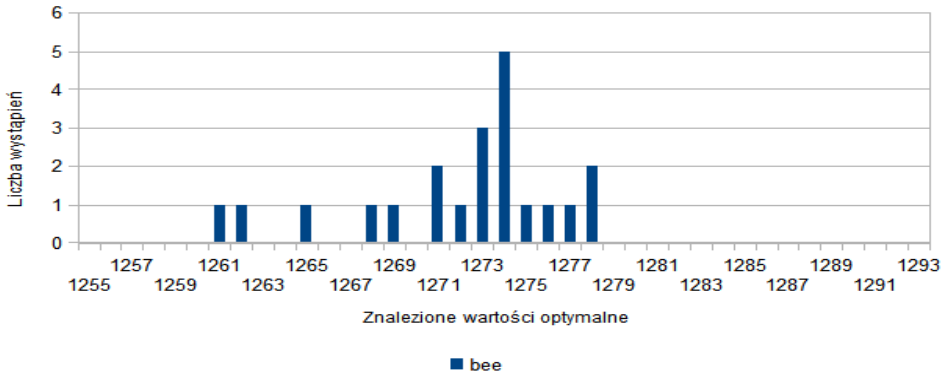
6.4 Zbieżność stochastyczna w badanych algorytmach

Rysunki 6.3 oraz 6.4 pokazują wynik doświadczenia, w którym badano zbieżność stochastyczną. Weryfikacja tej cechy algorytmów jest konieczna ze względu na to, że jak to już zostało wyjaśnione, wartość oczekiwana wykorzystywana przy estymacji liczby iteracji nie jest tożsama z poszukiwaną wartością $C_{\max}$. Stąd też warto chociaż spróbować odpowiedzieć sobie na pytanie, czy istnieje jakakolwiek zależność pomiędzy wartościami μ, ϵ i δ a $C_{\max}$.
Może się wydawać, że odpowiednim narzędziem do analizy otrzymanych wyników mógłby być jeden z testów statystycznych przykładowo test istotności dla małej próby ($k < 30$), o nieznanych wartościach $E(X)$ i $D(X)$ oraz o normalnym rozkładzie populacji wyznaczonych wartości średnich. Jednak biorąc pod rozwagę, że $C_{\max}$ jest skrajną, najmniejszą znalezioną wartością ze wszystkich doświadczeń, stąd też nie podlega ona prawom statystyki.
Załóżmy jednak, że w jakiejś mierze, wartości C_i, gdzie $i \in 1, \ldots, m$ tworzą ciąg zmiennych losowych a rysunki 6.3 i 6.4 przedstawiają rozkłady prawdopodobieństw odpowiednio dla algorytmu pszczelego i kukułki. Oraz niech μ będzie wyznaczoną wartością średnią z C_i i $P(|C_{\max} - \mu| < \epsilon\mu) > 1 - \delta$, wówczas:

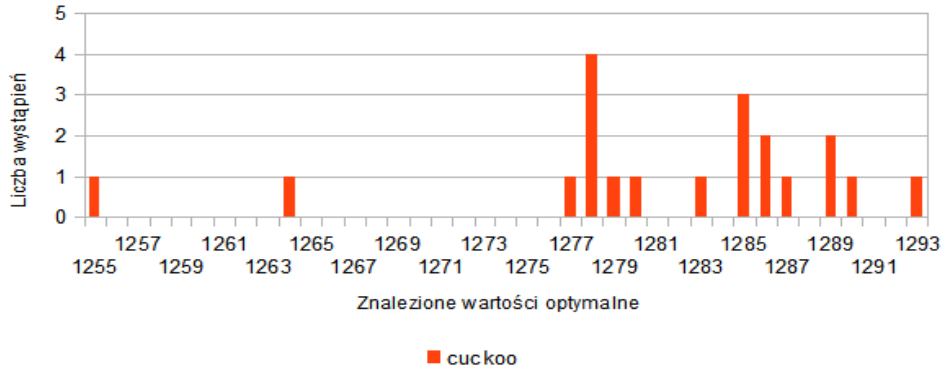
- dla algorytmu pszczelego $\mu = 1272$, $\epsilon\mu = 12.72$, stąd potencjalnie $C_{\max}$ powinno zawierać się w przedziale $(1259, 1285)$ z prawdopodobieństwem $1 - \delta = 0.99$,
- dla algorytmu kukułki $\mu = 1281$, $\epsilon\mu = 12.81$, stąd potencjalnie $C_{\max}$ powinno zawierać się w przedziale $(1268, 1294)$ z prawdopodobieństwem $1 - \delta = 0.99$.

W przypadku algorytmu pszczelego wszystkie otrzymane wyniki C_i zawierają się w wyznaczonym przedziale, co pozwala powiedzieć, że jest (przynajmniej w pewnym sensie) stochastycznie zbieżny. Natomiast dla algorytmu kukułki zależność $P(|C_{\max} - \mu| < \epsilon\mu) > 1 - \delta$ nie jest spełniona. Warto również

zwrócić uwagę na to, że rzeczywista wartość optymalna (1230) w obu przypadkach znajduje się poza przedziałem prawdopodobnych wartości C_i.
Powyższe obserwacje prowadzą do dwóch istotnych wniosków. Po pierwsze, schemat poszukiwań w algorytmie kukułki wymaga innego podejścia w wyznaczaniu liczby iteracji, która powinna być znacząco większa. O ile, na to mogą odpowiedzieć przyszłe badania. Po drugie, należałoby poszukać innych metod szacujących poprawność otrzymanych wartości optymalnych.



Rysunek 6.3: Wyniki poszukiwania wartości optymalnej przez algorytm pszczeli zainicjalizowany harmonogramem losowym. *Źródło: opracowanie własne.*



Rysunek 6.4: Wyniki poszukiwania wartości optymalnej przez algorytm kukułki zainicjalizowany harmonogramem losowym. *Źródło: opracowanie własne.*

6.5 Podsumowanie

W pracy zaprezentowano autorskie implementacje w języku *Scala* dwóch zrównoleglonych algorytmów stadnych: pszczelego i kukułki. W przypadku algorytmu kukułki nie znaleziono w literaturze przykładów implementacji zrównoleglonych, co nie oznacza, że takie nie istnieją.

W rozdziale 4 przeprowadzono autorską analizę zbieżności algorytmów za pomocą sprzężonych łańcuchów Markowa i algorytmu Metropolisa. Wykazano, że algorytmy pszczeli i kukułki, jako przykłady algorytmów stochastycznych, przy pewnych założeniach są zbieżne stochastycznie. Dzięki temu mogą być również algorytmami nieobciążonymi. Następnie wyznaczono ilość iteracji w algorytmach posługując się nierównością Chernoffa. Wykazano również, że sposób wyznaczania $C_{\max}$ zaimplementowany w pracy nie należy do klasy polilogarytmicznej i nie jest podatny na zrównoleglenie. Nie rozstrzygnięto natomiast, czy taki algorytm istnieje. Podano również warunek (razem z dowodem), jaki muszą spełniać zadania, aby obliczenia $C_{\max}$ wykonać równolegle.

Rozdział 5 poświęcono prezentacji implementacji algorytmów. Program został stworzony w języku *Scala* z wykorzystaniem modelu *aktorów*, dzięki czemu może on wykorzystać zalety przetwarzania wielowątkowego. Dodatkowo przedstawiono wyniki przykładowych doświadczeń oraz wnioski z nich płynące.

Każdy z zaprezentowanych wniosków w pracy może stać się wstępem do dalszych, bardziej ukierunkowanych, lecz głębszych badań. Począwszy od zagadnień związanych z analizą stochastyczną przestrzeni rozwiązań i w ten sposób doboru schematu jej przeszukiwania, poprzez nietrywialne zagadnienie sąsiedztwa i odległości w przestrzeniach permutacyjnych, a w ten sposób wyznaczania liczby iteracji, kończąc na implementacji rzeczywiście rozproszonej aplikacji (być może w języku *Scala* w wersji 2.10).

Podsumowując należy stwierdzić, że zaprezentowane implementacje zrównoleglonych algorytmów stadnych oraz przedstawione wnioski z badań zawierają w sobie potencjał adaptacji do znajdowa-

nia wartości bliskich optymalnej nie tylko dla problemu $Fm|prmu|C_{max}$, ale również każdego innego problemu szeregowania zadań. W przyszłości mogą się one stać konkurencyjnymi dla obecnie znanych heurystyk. Co do celu pracy to można stwierdzić, że został realizowany w większym zakresie niż początkowo zakładano, to znaczy:

- **algorytmy zaimplementowano z użyciem wątków,**
- **podano analizę liczby iteracji i spodziewanej dokładności znalezionych wartości optymalnych,**
- **opisano kardynalne różnice w sposobie przeszukiwania przestrzeni rozwiązań przez zaimplementowane algorytmy.**

Tezy i obserwacje zawarte w pracy mogą być rozwinięte w dalszych badaniach.

Bibliografia

[1] Akka Documentation for Scala, *http://akka.io/docs/*

[2] Akkan, C., Karabat, S.i *The Two-Machine Flowshop Total Completion Time Problem: Improved Lower Bounds and a Branch-and-Bound Algorithm*, European Journal of Operational Research, Vol. 159, pp. 420–429, 2004

[3] Aldous, D., *Random walks on finite groups and rapidly mixing Markov chains*, Seminar on probability, XVII, Lecture Notes in Math., vol. 986, Springer, Berlin, pp. 243–297, 1983

[4] Arabas, J., *Wykłady z algorytmów ewolucyjnych*, Wydawnictwa Naukowo-Techniczne, 2004

[5] Arora, S., Barak, B., *Computational Complexity: A Modern Approach*, Published draft of book, Jan. 2007

[6] Aytug, H., Khouja, M., Vergara, F., *Use of Genetic Algorithms to Solve Production and Operations Management Problems: a Review*, International Journal of Production Research, Vol. 41, pp. 3955–4009, 2003

[7] Bacanin, N., Brajevic, I., Tuba, M. *Firefly algorithm applied to Integer Programming Problems* Megatrend University Belgrade, 2013

[8] Badr, A., Fahmy, A., *A proof of convergence for Ant Algorithms*, IJICIS, Jan. 2003

[9] Boryczka, U., *Algorytmy optymalizacji mrówkowej*, Wydawnictwo Uniwersytetu Śląskiego, 2006

[10] Brown, A., Lomnicki, Z., *Some Applications of the Branch and Bound Algorithmto the Machine Sequencing Problem*, Operational Research Quarterly, Vol. 17, pp. 173–186, 1966

[11] Bubley, R., Dyer, M., *Path coupling: A technique for proving rapid mixing in Markov chains*, Proceedings of the 38th Annual Symposium on Foundations of Computer Science, pp. 223–231, 1997

[12] Burnwal, S., Deb, S., *Scheduling optimization of flexible manufacturing system using cuckoo search-based approach*, Int. J. Adv Manuf Technol, 2012

[13] Błazewicz, J., Ecker, K. H., Schmidt, G., Węglarz, J., *Scheduling in Computer and Manufacturing Systems* Springer, 1994

[14] Carlier, J., Pinson, E., *An algorithm for solving the job-shop problem*, Management Science, Feb. 1989

[15] Chen, B., Pottsand, C., Woeginger, J., *A Review of Machine Scheduling: Complexity, Algorithms, and Approximability*, in Handbookof Combinatorial Optimization, D.-Z. Du and P. Pardalos(eds.), pp. 21–169, Kluwer Academic Press, Boston, 1998

[16] Chiusano, P., Bjarnason, R., *Functional Programming in Scala*, Early access edition, 2012

[17] Chong, C., Low, M. Y. H., Sivakumar, A. I., Gay, K. L., *A Bee Colony Optimization Algorithm to Job Shop Scheduling*, Simulation Conference, WSC 06. Monterey, CA, 2006

[18] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C., *Wprowadzenie do algorytmów*, Wydawnictwa Naukowo-Techniczne, 2005

[19] Dorigo, M., Di Caro, G., *The Ant Colony Optimization Meta-Heuristic*, In D. Corne, M. Dorigo and F. Glover (eds.), New Ideas in Optimization McGraw-Hill, 11-32, 1999

[20] Dorigo, M., Gambardella, L., *Ant Colonies for the Travelling Salesman Problem*, BioSystems, 43;78-81, 1997

[21] Dorigo, M., Stutzle, T., *The ACO metaheuristic: Algorithms, Applications, and Advances*, Handbook of metaheuristics, 2002

[22] Durkota, K., *Implementation of a discrete firefly algorithm for the QAP problem within seage framework*, Chech Technical University in Pague, 2011

[23] Eberle, A., *Stochastic Analysis. An Introduction*, University of Bonn, 2011

[24] Garey, M. R., Johnson, D. S., Seth, R., *The Complexity of Flowshop and Jobshop Scheduling*,Mathematics of Operations Research, Vol.1, pp.117–129, 1976

[25] Gerstenkorn, T., Śródka, T., *Kombinatoryka i rachunek prawdopodobieństwa*, Państwowe Wydawnictwo Naukowe, 1980

[26] Gutowski, M., *Lévy flights as an underlying mechanism for global optimization algorithms*, ArXiv Mathematical Physics e-Prints, June, 2001

[27] Jerrum, M., Sinclair, A., *The Markov chain Monte Carlo method: an approach to approximate counting and integration*, Approximation Algorithms for NP-hard Problems, 1996

[28] Karatzas, I., *A tutorial introduction to stochastic analysis and its application*, Dep. of Statistics, Columbia University, 1988

[29] Layeb, A., *A novel quantum inspired cuckoo search for knapsack problems, Int. J. Bio-Inspired Computation*, Vol. 3, pp. 297-305, 2011

[30] Lenstra, J.K., Rinnooy Kan, A., *Computational Complexity of Discrete Optimization Problems*, Annals of Discrete Mathematics, Vol.4, pp. 121–140, 1977

[31] Leung, J., Pinedo, M., *Scheduling Jobs on Parallel Machines that are subject to Breakdowns*, Naval Research Logistics, Vol. 51, pp. 60–71, 2004

[32] Levin, D. A., Peres, Y., Wilmer, E. L., *Markov Chains and Mixing Times*, University of Oregon, 2009

[33] Mantegna, R., *Fast, accurate algorithm for numerical simulation of Lévy stable stochastic processes*, Physical Review E, Vol.49, 4677-4683, 1994

[34] Mitzenmacher, M., Upfal, E., *Metody probabilistyczne i obliczenia*, Wydawnictwa Naukowo-Techniczne, 2009

[35] Nowicki, E., Smutnicki, C., *A Fast Taboo Search algorithm for the Job Shop Problem*, Management Science, Vol. 42, pp. 797–813, 1996

[36] Nyree Lemmens, S., de Jong, K., Tuyls, A. N., *A Bee Algorithm for Multi-Agents System*, 2007

[37] Odersky, M., Spoon, L., Venners, B., *Programming in Scala, Second Edition*, Artima, 2010

[38] Papadimitriou, C. H., Steiglitz, K., *Combinatorial Optimization: Algorithms and Complexity*, Dover Pub., 1998

[39] Papadimitriou, C. H., *Złożoność obliczeniowa*, Wydawnictwa Naukowo-Techniczne, 2002

[40] Parker, R. G., *Deterministic Scheduling Theory*, Chapman&Hall, London, 1995

[41] Pham, D. T., , Kog, E., Ghanbarzadeh, A., Otri S., Rahim, S., Zaidi, Z., *The Bees Algorithm – A Novel Tool for Complex Optimisation Problems*, 2nd international virtual conference on Intelligent PROduction Machines and Systems IPROMS, Oxford, Elsevier, 2006

[42] Pham, D. T., Koç, E., Lee, J. Y., Phrueksanant, J., *Using the Bees Algorithm to schedule jobs for a machine*, Proc Eighth International Conference on Laser Metrology, CMM and Machine Tool Performance, LAMDAMAP, Euspen, UK, Cardiff, pp.430–439, 2007

[43] Pinedo, M. L., *Scheduling: Theory, Algorithms, and Systems*, Springer, 2008

[44] Pinedo, M., Weiss, G., *Scheduling of Stochastic Tasks on Two Parallel Processors*, Naval Research Logistics Quarterly, Vol. 26, pp. 527–535, 1979

[45] Queyranne, M., *Structure of a Simple Scheduling Polyhedron*, Mathematical Programming, Vol. 58, pp. 263–286, 1993

[46] Roemer, T. A., *A Note on the Complexity of the Concurrent Open Shop Problem*, Journal of Scheduling, Vol. 9, pp. 389–396, 2006

[47] Ross, S. M., *Introduction to Stochastic Dynamic Programming*, Academic Press, New York, 1983

[48] Scala Actors: A short tutorial, *http://www.scala-lang.org/node/242*

[49] Scala Language Documentation, *http://www.scala-lang.org/node/197*

[50] Sevastianov, S. V., Woeginger, G., *Makespan Minimization in Open Shops: a Polynomial Time Approximation Scheme*, Mathematical Programming, Vol. 82, pp. 191–198, 1998

[51] Shmoys, D. B., Wein, J., Williamson, D. P., *Scheduling Parallel Machines On-line*, SIAM Journal of Computing, Vol. 24, pp. 1313–1331, 1995

[52] Skiena, S. S., *The Algorithm Design Manual*, Springer, 2008

[53] Sriskandarajah, C., Sethi, S. P., *Scheduling Algorithms for Flexible Flow Shops: Worst and Average Case Performance*, European Journal of Operational Research, Vol. 43, pp. 143–160, 1989

[54] Taillard, E., *Some Efficient Heuristic Methods for the Flow Shop Sequencing Problem*, European Journal of Operational Research, Vol.47, pp.65–74, 1990

[55] Taylor, H., Karlin, S., *An Introduction to Stochastic Modeling* Academic Press, 1998

[56] Walton, S., Hassan, O., Morgan, K., Brown, M., *Modified cuckoo search: A new gradient free optimisation algorithm*, Chaos, Solitons & Fractals, 2011

[57] Weber, R. R., *Scheduling Jobs with Stochastic Processing Requirements on Parallel Machines to Minimize Makespan or Flow Time*, Journal of Applied Probability, Vol. 19, pp. 167–182, 1982

[58] Yang, X-S., Deb. S, *Cuckoo search via Levy flights*, IEEE Publications pp. 210-214, arXiv:1003.1594v1, 2009

[59] Yang, X. S., Deb, S., *Engineering optimisation by cuckoo search*, Int. J. Mathematical Modelling and Numerical Optimisation, Vol. 1, No. 4, 330-343, 2010 (*http://arxiv.org/abs/1005.2908*)

[60] Yang, X. S., *Engineering Optimizations Via Nature-Inspired Virtual Bee Algorithms*. Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach, pp. 317-323 , Springer Berlin/Heidelberg, 2005

[61] Yang, X.-S., Deb, S., *Cuckoo Search algorithm by Xin-She Yang and Suesh Deb programmed by Xin-She Yang*, Cambridge University, 2009

[62] Zhang, W., Dietterich, T., *A Reinforcement Learning Approach to Job-Shop Scheduling*, in Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95) , C.S. Mellish (ed.), pp. 1114–1120, Conference held in Montreal, Canada, Morgan Kaufmann Publishers, San Francisco, California, 1995