POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: Automatyka i Robotyka (AIR)
SPECJALNOŚĆ: Technologie informacyjne w systemach automatyki (ART)

PRACA DYPLOMOWA MAGISTERSKA

Korygowanie harmonogramów
z uwzględnieniem awarii maszyn

Re-scheduling with regard to machine failure

AUTOR:
inż. Kamil Niemczyk

PROWADZĄCY PRACĘ:
dr hab. Wojciech Bożejko Prof. PWr, W4/K-8

OCENA PRACY:

WROCŁAW 2016

Spis treści

1 Przegląd literatury **2**
1.1 Optymalizacja dyskretna oparta na uczeniu z nauczaniem 2
1.2 Podejście ewolucyjne w harmonogramowaniu proaktywnym 3
1.3 Wyznaczanie harmonogramów od nowa 4
1.4 Analiza wydajności linii produkcyjnych ze skończonym buforem 6
1.5 Potrzeba ponownej obróbki elementów 7
1.6 Harmonogramowanie odporne 7

2 Sformułowanie problemu **10**

3 Opis algorytmów **12**
3.1 Model danych 12
3.2 Rozwiązanie początkowe 13
3.3 Tabu Search z nawrotami 13
3.4 Awarie 16

4 Eksperymenty obliczeniowe **18**
4.1 Wizualizacja wyników 18
4.2 Parametryzacja 18
4.3 Statystyki 19

Wnioski i uwagi **21**

Dodatek A Biblioteka do generowania HTML dla C++ **22**
A.1 Interfejs `ContentHTML` 22
A.2 Klasa `TextContentHTML.h` 22
A.3 Klasa `TagContentHTML` 23
A.4 Klasa `PageHTML` 27
A.5 Style CSS 29
A.6 JavaScript 29

Bibliografia **31**

Spis rysunków **33**

Spis tabel **34**

Rozdział 1

Przegląd literatury

Pierwsze próby rozwiązania problemu harmonogramowania zadań w procesach produkcyjnych sięgają końca lat 70 XX wieku [6]. Tak wcześnie jak zaczęto problemem się zajmować, wkrótce zakwalifikowano go jako problem NP trudny [6, 17]. Trzy ogólne problemy, które wydzielono w tym zagadnieniu, to:

- problem przepływowy (flow shop),
- problem gniazdowy (job shop),
- problem otwarty (open shop).

Klasyfikacja problemów harmonogramowania jako NP trudne oznacza, że metody dokładne stosowane do rozwiązania tych problemów posiadają ponad-wielomianową złożoność obliczeniową, np. przegląd zupełny cechuje złożoność $O(n!)$. Z tego względu zaczęto stosować różnego rodzaju metaheurystyki, jak np. Tabu Search, Algorytm Genetyczny, czy Symulowane Wyżarzanie. W międzyczasie zaobserwowano, że w rzeczywistości procesom produkcyjnym towarzyszą różnego rodzaju zakłócenia, stąd rozpoczęto prowadzenie badań nad korygowaniem harmonogramów.

1.1 Optymalizacja dyskretna oparta na uczeniu z nauczaniem

W [9] autorzy proponują optymalizację dyskretną opartą na uczeniu z nauczaniem (DTLBO, ang. *Discrete teaching-learning based optimization*) dla realistycznych problemów przepływowych. Uwzględniono tam jednoczesny wpływ 5 rodzajów zakłóceń:

- awaria maszyny,
- nadejście nowego zadania,
- anulowanie zadań,
- zmienny czas przetwarzania zadań,
- zmienny czas *stygnięcia* zadania.

Odnośnie awarii maszyn, zakłada się tutaj, że nie wiemy kiedy występują, ani jak długo trwają. Ponadto, zadanie przerwane przez taką awarię, po jej usunięciu jest kontynuowane

od momentu jego przerwania. W zależności od tego czy dana operacja została zakończona przed wystąpieniem awarii, przerwana przez awarię lub miała się dopiero wykonać, zostaje przypisana do odpowiedniej grupy. Wszystkie przerwane zadania pozostają na swoich maszynach, a kolejne zadania są przesuwane w czasie, w zależności od oszacowania czasu naprawy usterki. W przypadku nadejścia nowego zadania stosuje się reaktywne procedury, które odpowiednio plasują je w harmonogramie. Poza sytuacjami wystąpienia zakłóceń nie jest dopuszczalne przerywanie zadań, a kolejne operacje danego zadania nie mogą się na siebie nakładać. Poza fazami nauczania i uczenia, charakterystycznymi dla TLBO wykorzystano tutaj również iteracyjny algorytm zachłanny (IG) w celu przyspieszenia procesu ewolucji. Celem porównania wybranej przez autorów metody w zależności od jej parametryzacji, skonstruowano sześć wariantów TLBO, spośród których wyłoniono najlepszy i wykorzystano do konstrukcji kolejnych czterech wariantów DTLBO. Ponownie w testach wybrano najwydajniejszy wariant i ostatecznie porównano go z algorytmami:

- hGA - hybrydowy algorytm genetyczny,
- IG - iteracyjny algorytm zachłanny,
- ILS - iteracyjny algorytm poszukiwań lokalnych,
- PSO - optymalizacja rojem cząstek.

Wyniki zestawiono na podstawie testów na 90 instancjach Taillard'a z modelem zakłóceń zapożyczonym z [12]. Jako kryterium porównawcze zastosowano względne odchylenie procentowe (*RPD - Relative prcentage deviation*) od najlepszego wyniku dla danej instancji. Z uwagi na wykorzystany w funkcji celu współczynnik kary, wyniki podzielono na 3 kategorie, w których współczynniki kary wynosiły odpowiednio 0.1, 0.5 oraz 0.9. We wszystkich przypadkach algorytm DTLBO pozwolił uzystać autorom najlepsze wyniki, niestety jednak kosztem znacząco wyższej złożoności obliczeniowej (średnie $RPD = 18,79$, podczas gdy ILS uzyskało wynik $RPD = 1,29$).

1.2 Podejście ewolucyjne w harmonogramowaniu proaktywnym

Kolejna pozycja literaturowa [20] rozważa harmonogramowanie proaktywne w odpowiedzi na losowe (stochastyczne) awarie maszyn w pogarszających się środowiskach produkcyjnych, gdzie właściwy czas przetwarzania zadania wydłuża się proporcjonalnie do używania maszyny i jej wieku. Zaznacza się tu istotność odporności rozwiązania (ang. *solution robustness*), z uwagi na obszary, których problem ten dotyka. Może być to sytuacja, kiedy firmy udostępniają przez Internet swoje harmonogramy produkcji swoim dostawcom surowców, aby dostawy mogły się odbywać w filozofii Just In Time. Ma to również znaczenie, jeśli chcieć posługiwać się tymi odpornymi rozwiązaniami problemu do budowy wskaźnika wydajności z punktu widzenia kierownictwa i operatorów hal produkcyjnych. Co również istotne, zasugerowano że umożliwiłyby one wgląd w niedaleką przyszłość pozwalając tym samym na sugerowanie konkurencyjnych terminów dostaw dla klientów. Podobnie jak w przypadku [9], tutaj również analizowano problem przepływowy, bez możliwości bezpośredniego przerywania zadań, gdzie awarie maszyn występują zgodnie z pewnym rozkładem prawdopodobieństwa, z tym że w tym artykule ustalany jest on na podstawie statystycznych danych historycznych. Co odróżnia tą publikację od poprzedniej, jest uwzględnienie stopniowego pogarszania czasu przetwarzania operacji z

uwagi na czas pracy i wiek maszyny oraz fakt, że reakcja na awarię maszyny jest natychmiastowa, przerwane zadanie musi zostać wykonane jeszcze raz (od samego początku) oraz, jeśli chodzi o aspekt starzenia się maszyn, po naprawie wszystkie uszeregowane zadania na naprawionej maszynie wykonują się w oryginalnie założonym przedziale czasu, bez pogorszenia wynikającego z opisanych wyżej przyczyn. Głównym wyznacznikiem jakości zbudowanego algorytmu (autorzy nazwali go ADK/SA-NSGA-II) była analiza wariancji (ANOVA). Jako punkt odniesienia przyjęto wielokryterialny algorytm ewolucyjny (MOEA/D, ang. *Multi objective evolutionary algorithm*). W rezultacie rozwiązanie autorów uzyskało znacząco lepszą zbieżność dla mniejszych instancji (30-200 zadań), jednak powyżej tego pułapu (300-500 zadań) przegrało z konkurentem.

1.3 Wyznaczanie harmonogramów od nowa

Jeszcze inne podejście analizuje się w artykule [8], gdzie proponuje się strategię sterowaną zdarzeniami, nie tyle korygując harmonogram, co za każdym razem tworząc go na nowo. Problem dotyczy harmonogramowania m identycznych równolegle pracujących maszyn z uwzględnieniem czasu przezbrojeń każdej z maszyn między wykonywanymi zadaniami przez pojedynczy serwer. W pracy podano definicję korygowania harmonogramów:

> *„Aktualizacja istniejącego harmonogramu produkcyjnego w odpowiedzi na nieprzewidywalne zdarzenia czasu rzeczywistego, aby zminimalizować ich wpływ na wydajność systemu."*

Pytanie jakie należałoby sobie postawić to: „Jak i kiedy reagować na te zdarzenia czasu rzeczywistego?". Odpowiedź nie jest jednoznaczna, co sugeruje też dalsza treść artykułu. Sugeruje się różne podejścia. Odnośnie tego „jak" proponuje się naprawę harmonogramu, polegającą na pewnych lokalnych dopasowaniach, bądź też ustalenie harmonogramu zupełnie od nowa. Natomiast, jeśli spytać „kiedy" reagować, przedstawiane są trzy podejścia: periodyczne, polegające na korygowaniu cyklicznym, sterowane zdarzeniami, czyli korekty są wprowadzane tylko w sytuacji zarejestrowania określonych zdarzeń, oraz hybrydowe, będące połączeniem dwóch poprzednich.

Wracając do tematyki problemu przedstawionego w artykule, ponieważ zakłada się istnienie tylko jednego serwera, wydzielono trzy oszacowania czasu przezbrojenia maszyn:

Mniejszy (*minor*) – kiedy maszyna będzie przetwarzać pierwsze zadanie na początku harmonogramu; jeśli awaria maszyny przerwała jakieś zadanie, tego typu przezbrojenie będzie również wymagane.

Średni (*medium*) – jeśli poprzednio wykonywane na maszynie zadanie jest tego samego typu co kolejne, które ma się wykonać na tej samej maszynie.

Większy (*major*) – w przypadku gdy kolejne zadanie do wykonania na tej samej maszynie różni się typem od zadania, które zostało na niej ukończone.

Dopuszcza się przerywanie zadań. Stochastyczną awaryjność maszyn określają dwa współczynniki o rozkładzie wykładniczym: średni czas między awariami (MTBF, ang. *Mean Time Between Failure*) oraz średni czas do wykonania naprawy (MTTR, ang. *Mean time to repair*). Z uwagi na fakt większej złożoności obliczeniowej podczas wyznaczania harmonogramu za każdym razem od nowa w porównaniu z korygowaniem harmonogramu,

przyjęto strategię kolejkowania nadchodzących zadań i układania harmonogramu jedynie dla części z nich. Autorzy wydzielili cztery zdarzenia, które powodują uruchomienie procedury ustalenia nowego harmonogramu:

Zdarzenie #1: nadejście nowego zadania – w ten sposób system zapełnia pulę harmonogramowanych zadań aż do osiągnięcia zadanej górnej granicy liczby harmonogramowanych zadań.

Zdarzenie #2: awaria maszyny – zadanie które zostało przerwane zostaje dodane do puli zadań będących w harmonogramie i ustala się nowy harmonogram z pominięciem maszyny, która uległa awarii.

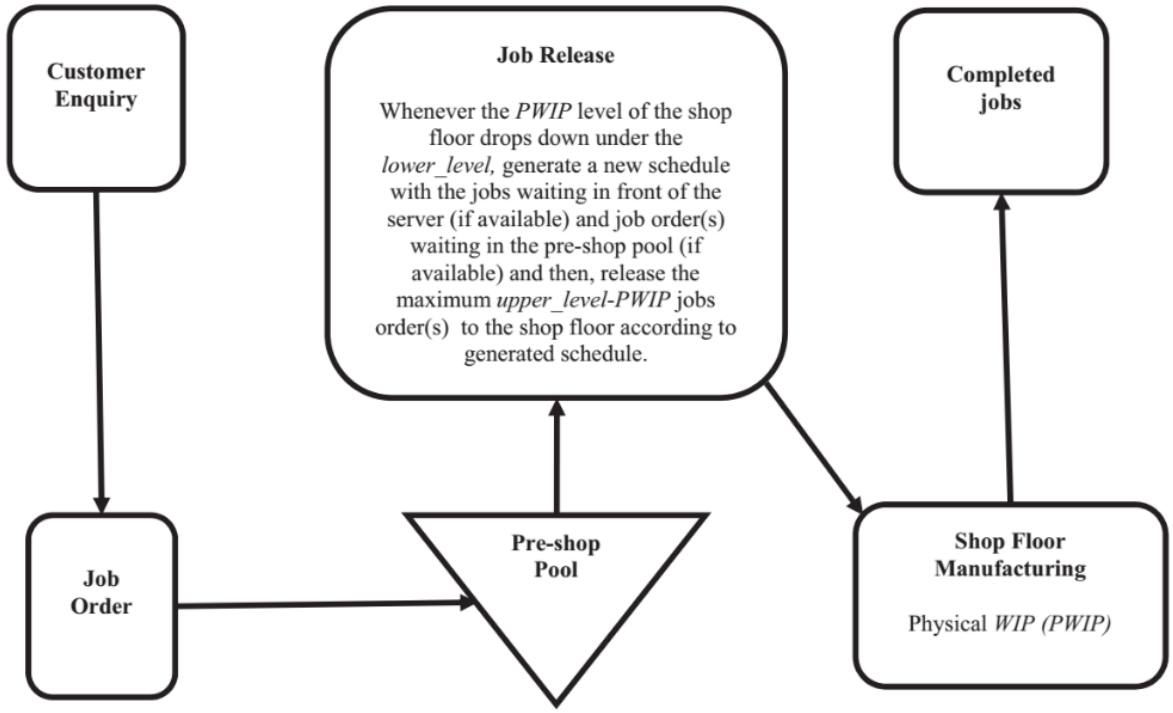
Zdarzenie #3: naprawienie maszyny – ustala się nowy harmonogram dla zadań już uszeregowanych, ale których wykonywanie jeszcze się nie rozpoczęło - tym razem już z uwzględnieniem maszyny, która wcześniej uległa awarii.

Zdarzenie #4 – aktywuje się na koniec **zdarzenia #1** oraz w momencie kiedy liczba zadań w puli tych już uszeregowanych na maszynach spada poniżej pewnej dolnej granicy. Powoduje uwolnienie z kolejki określonej liczby zadań i dodanie do zadań będących w puli już uszeregowanych zadań oraz wyznaczenie nowego harmonogramu.

Układanie harmonogramu wyzwolone każdym z powyższych zdarzeń uwzględnia jedynie maszyny, które są dostępne (mogą pracować). W przypadku gdy choć jedna maszyna nie jest dostępna (np. na skutek awarii) harmonogram nie zostaje wyznaczony. Schemat dodawania zadań z kolejki do ułożenia harmonogramu pokazano na rysunku 1.1. Z wykorzystaniem algorytmu symulowanego wyżarzania (SA) oraz reguł planowania przydziału procesora (DR, ang. *Dispatching Rules*), takich jak SPT, LPT oraz FIFO, wykonano symulacje dla proponowanego rozwiązania przedstawionego problemu. Przyjęto, że:

- na hali produkcyjnej pracuje równolegle 10 identycznych maszyn,
- istnieje 50 różnych typów zadań,
- czas przetwarzania każdego zadania mieści się w zakresie 100-500,
- **większe** czasy przezbrojenia mieszczą się w zakresie 20-100,
- **średnie** czasy przezbrojenia wynoszą 10,
- **mniejsze** czasy przezbrojenia wynoszą 5,
- wartość średnia rozkładu wykładniczego dla MTBF wynosi 12000,
- wartość średnia rozkładu wykładniczego dla MTTR wynosi 3000.

Ostatecznie, na podstawie przeprowadzonych eksperymentów, wywnioskowano, że algorytm symulowanego wyżarzania osiągnął lepsze rezultaty niż FIFO, LPT czy SPT.



Rysunek 1.1 Schemat kolejkowania nadchodzących zadań oraz dodawanie ich do puli zadań, dla których zostaje wyznaczony nowy harmonogram. Na diagramie za kolejnością strzałek: klient składa ofertę, pula wszystkich zadań, kolejka zadań oczekujących na dodanie do puli zadań w harmonogramie, wyzwolenie zadań (zdarzenie #4), fizyczne wykonywanie zadań na hali produkcyjnej, wykonane zadania. Źródło: [8]

1.4 Analiza wydajności linii produkcyjnych ze skończonym buforem

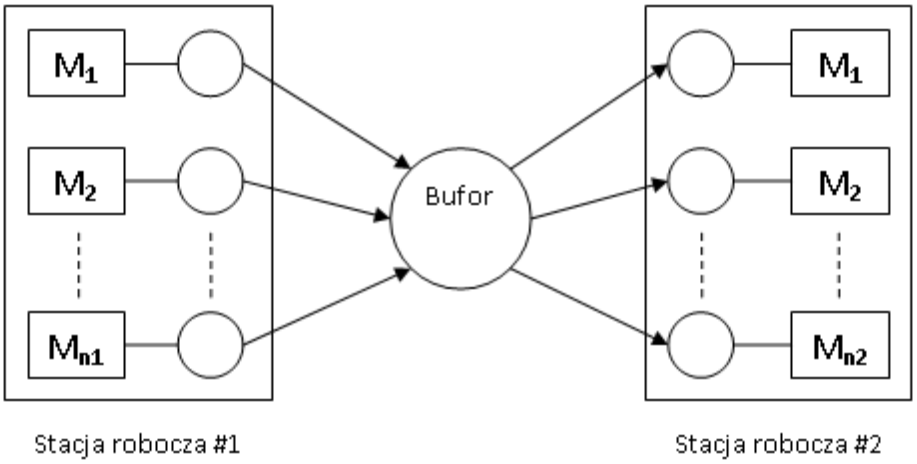
Pozycja [14] pokazuje jeszcze inny problem produkcyjny związany z awarią maszyn. Analizuje się tutaj wydajność linii produkcyjnej składającej się z dwóch stacji roboczych połączonych wspólnym buforem o skończonej pojemności, gdzie na każdą ze stacji składają się równolegle niezależnie pracujące maszyny. Ilustracja problemu została przedstawiona na Rysunku 1.2.
Artykuł opisuje możliwe stany każdej z maszyn, których możliwe stany przedstawiono poniżej:

Pracuje (*working*) – czyli obrabia pewien element,

Jest zagłodzona (*starving*) – w sytuacji gdy bufor przed stacją roboczą nie zawiera dostępnych do obróbki elementów,

Jest zablokowana (*blocked*) – jeżeli skończyła obróbkę elementu, ale bufor za stacją roboczą jest pełny.

Odnośnie awaryjności maszyn, w pracy zakłada się, że tylko pracująca maszyna może ulec awarii, a zagłodzona lub zablokowana już nie. Dodatkowo, czego nie uwzględniono na powyższym schemacie, przed stacją roboczą #1 oraz za #2, istnieją bufory o nieskończonych



Rysunek 1.2 Poglądowy schemat analizowanej w pracy [14] linii produkcyjnej.

pojemnościach. Autorzy porównali zaproponowany trójstanowy model maszyn z modelem dwustanowym, który opracowali Diamantidis i Papadopoulos w [4], oraz z modelem symulacyjnym WITNESS. Model trójstanowy okazał się dawać rezultaty bardzo zbliżone jakościowo do wyników działania na modelu symulacyjnego - miejscami nawet lepsze.

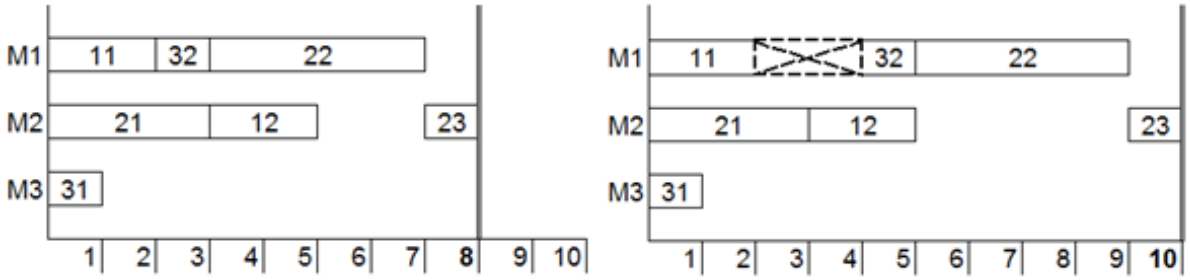
1.5 Potrzeba ponownej obróbki elementów

Co zostało już zaznaczone we wcześniej wspomnianych tytułach, również autorzy pracy [15] zauważyli inne przyczyny potrzeby korygowania harmonogramów. Poza tymi związanymi z zasobami (awarie lub zawieszanie się maszyn, przestoje lub blokady linii produkcyjnej, opóźnienia w dostawach, itp.) istnieją również takie, które mają związek z samymi zadaniami wykonywanymi na maszynach (anulowanie lub nadejście nowych zadań, zmiany ich priorytetów, czasów obróbki, czy też potrzeba poprawek lub przeróbek). W takich sytuacjach należy znaleźć kompromis między wydajnością (koszty harmonogramowania) a stabilnością (koszty zakłóceń). Koszty zakłóceń typowo mierzy się badając jak często oraz w jakim stopniu zmieniają się: kolejność wykonywanych zadań, czasy rozpoczęcia oraz zakończenia zadań, a także obciążenie maszyn w poprzednim oraz nowym harmonogramie. Natomiast koszty harmonogramowania wyrażają się bardzo prosto, w całkowitym czasie wykonania harmonogramu. Artykuł skoncentrował się nie tyle na metodach wyznaczania harmonogramów (zastosowano tu prostą metodę planistyczną SPT), co na samym zagadnieniu zakłóceń wynikających z potrzeby ponownej obróbki przetwarzanych na maszynach elementów.

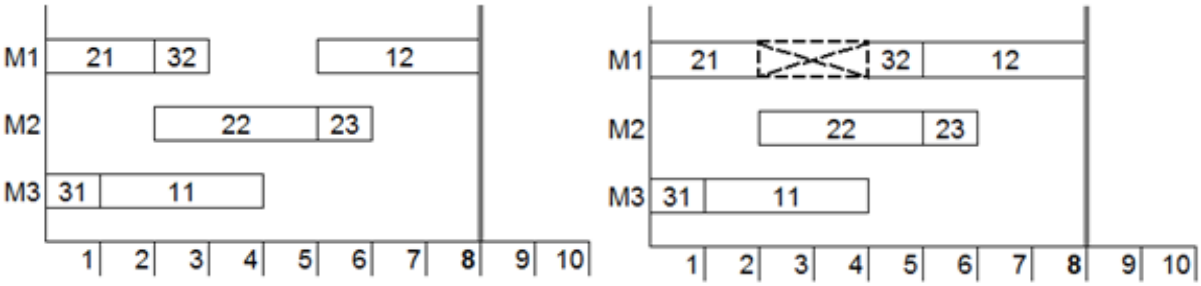
1.6 Harmonogramowanie odporne

Artykuły [2] oraz [21] poruszają interesujący temat alternatywnych rozwiązań w poszukiwaniu optymalnych harmonogramów. Odporne harmonogramowanie w problemach gniazdowych z uwzględnieniem losowych awarii maszyn [2] zostało tutaj zrealizowane z wykorzystaniem hybrydowego algorytmu genetycznego. Odporność algorytmu polega na tym, że generowane harmonogramy nie są tak ciasno upakowane, jak te które otrzymuje się

szeregując zadania w podejściu statycznym. Dzięki temu awaria maszyny nie zawsze musi oznaczać opóźnienie całego harmonogramu. Widać to na Rysunkach 1.3 i 1.4 W przypadku harmonogramowania statycznego (lewa strona na rysunkach) obydwa rozwiązania są takie same. Jednak z punktu widzenia harmonogramowania dynamicznego (prawa strona na rysunkach), np. wiemy, że są duże szanse awarii maszyny M1, rozwiązanie przedstawione na rysunku 1.4 jest lepsze pod względem odporności na nieplanowane opóźnienia pierwotnego harmonogramu. Proponowanym podejściem jest rozwiązanie elastycznego



Rysunek 1.3 Tradycyjne harmonogramowanie. Gęsto upakowane zadania usztywniają harmonogram. Awaria maszyny M1 w punkcie czasu 2 powoduje opóźnienie czasu wykonania harmonogramu.



Rysunek 1.4 Harmonogramowanie odporne. Luźno rozmieszczenie zadań pozwala na minimalne zmiany w harmonogramie oraz zachowanie czasu jego wykonania w przypadku awarii maszyny M1.

problemu gniazdowego za pomocą hybrydowego algorytmu genetycznego opracowanego przez autorów [2] w ich wcześniejszym artykule [1]. Korzysta się z niego dwukrotnie - za pierwszym razem do stworzenia harmonogramu, który minimalizuje czas wykonania wszystkich zadań, tak jak w klasycznym elastycznym problemie gniazdowym, a następnie uruchamiany jest ponownie, w celu zwiększenia odporności harmonogramu na późniejsze zmiany (uwzględnienie możliwości awarii maszyn). Reprezentacja genów chromosomów to trójki liczb (k, i, j) oznaczające kolejno:

- k – numer maszyny,
- i – numer zadania,
- j – numer operacji zadania i-tego.

Przykładowo: chromosom (2,1,1)-(1,2,1)-(1,1,2)-(2,2,2) oznacza, że na maszynie 1. wykonają się kolejno operacja 1. zadania 2. oraz operacja 2. zadania 1., a na maszynie 2.

operacja 1. zadania 1. oraz operacja 2. zadania 2. Dzięki takiej reprezentacji oraz zastosowaniu w algorytmie odpowiednio zaprojektowanych operatorów krzyżowania i mutacji pozwalają zapobiec osiągnięciu zabronionych harmonogramów, co pozwoliło autorom pominąć etap walidacji chromosomów otrzymanych w wyniku wykorzystania tych operatorów. Przypadki testowe zostały zapożyczone z prac [11, 16, 13, 10, 3]. Oprócz analizy wariancji otrzymanych w eksperymentach populacji (ANOVA) zbadano również wydajność harmonogramu pod kątem stabilności i odporności oraz czasu wykonania wszystkich zadań (ang. *makespan*).

***

Potrzeba korygowania harmonogramów jest bardzo obszernie analizowana w literaturze. Proponowane metody to najczęściej poszechnie znane heurystyki, takie jak Tabu Search, Symulowane Wyżarzanie, Algorytm Genetyczny, czy Algorytm Ewolucyjny, sparametryzowane pod kątem konkretnych problemów. Trudno określić jednoznacznie, która z nich jest najlepsza.

Rozdział 2

Sformułowanie problemu

Elastyczny Problem Gniazdowy (ang. *Flexible Job Shop Problem*, *FJSP*) jest uogólnieniem klasycznego Problemu Gniazdowego. Różnica poleta na tym, że nie uwzględnia się w nim ograniczenia dotyczącego maszyny, która ma wykonać konkretną operację w zadaniu. W konsekwencji, dla każdej z operacji w trakcie układania harmonogramu należy podjąć dodatkową decyzję o wyborze maszyny, która przetworzy daną operację. Ponieważ już klasyczny Problem Gniazdowy został zakwalifikowany do problemów silnie NP-trudnych, z uwagi na dodatkowe skomplikowanie FJSP także zakwalifikowano jako problem silnie NP-trudny [19]. Sformułowanie problemu deterministycznego Elastycznego Problemu Gniazdowego podane w [1] przedstawia się jak poniżej:

- Mamy n niezależnych wzajemnie zadań J indeksowanych przez i.
- Wszystkie zadania są gotowe do rozpoczęcia przetwarzania w chwili czasu 0. Każde zadanie J_i składa się z O_i operacji, których kolejność wykonania jest określona przez O_{ij}, gdzie $j = 1, ..., O_i$.
- Mamy m maszyn M indeksowanych przez k.
- Maszyny nigdy nie ulegają awarii i są zawsze dostępne.
- Każda operacja O_{ij} ma zdefiniowany zbiór maszyn $M_{kij} \subseteq 1, ..., m$, które mogą ją przetwarzać.
- Poszczególne czasy przetwarzania operacji O_{ij} na maszynach M_{kij} są zdefiniowane w zbiorze P_{ij} o tym samym rozmiarze co M_{kij} – przetwarzanie na maszynie k zdefiniowane jest przez p_{ijk}.
- Odpowiednio, jako czas rozpoczęcia uszeregowanej operacji oznacza się S_{ij}, a czas jej zakończenia C_{ij} wyraża się przez sumę $S_{ij} + p_{ijk}$.
- Czasy przezbrojenia maszyn są niezależne od kolejności operacji na nich wykonywanych i wliczone są w ich czasy przetwarzania.
- Nie można przerywać przetwarzania operacji.
- Każda z maszyn może przetwarzać jednocześnie co najwyżej jedną operację.
- Ograniczenia kolejnościowe przetwarzania operacji w zadaniach może zostać zdefiniowane dla dowolnej pary operacji.

Celem jest minimalizacja całkowitego czasu wykonania harmonogramu C_{max}, w literaturze anglojęzycznej częściej określanym jako *makespan*. Jendak w rzeczywistych warunkach produkcyjnych ustalony harmonogram może zostać zakłócony, na przykład na skutek awarii maszyny. Pozycja [2] zakłada pewną losowość występowania awarii maszyn. Jednak ta losowość jest zdeterminowana przez założenie posiadania historycznych danych ze środowiska produkcyjnego, a samo prawdopodobieństwo wystąpienia awarii jest wprost proporcjonalne do łącznego czasu pracy maszyny (suma czasów przetwarzania zakończonych operacji). Podobny efekt można uzyskać planując momenty wystąpienia awarii – może to być na przykład zaplanowany serwis maszyny. W pracy zdecydowano się na zastosowanie tego drugiego podejścia. Dalszy jego opis można znaleźć w rozdziale 3.

Rozdział 3

Opis algorytmów

Opracowany algorytm inspirowany jest interesującym podejściem harmonogramowania odpornego. Korzysta on z procedury wyznaczającej pierwszy harmonogram metodą wstawień dostosowaną do elastycznych problemów gniazdowych (*IniPopGen*) zaproponowaną przez Al-Hinai oraz ElMekkawy w [1]. Choć *IniPopGen* zaprojektowano do działania z algorytmem genetycznym, to praktycznie bez problemu udało się go zaadoptować do wykorzystania z metodą Tabu Search z nawrotami (*TSAB*), którą pierwszy raz opisano w [18]. Decyzję o rezygnacji z wykorzystania algorytmu genetycznego proponowanego przez autorów *IniPopGen* podjęto z uwagi na wysoką złożoność obliczeniową metody. O ile dla małych problemów, rzędu kilkudziesięciu operacji nie było to problemem, to już przy jednej z najmniejszych instancji Taillard'a (225 operacji, 15x15) jedynie inicjalizacja populacji 500 osobników zajmowała minutę. Dla porównania, algorytm *TSAB* wykonuje wszystkie obliczenia dla tej samej instancji średnio w sekundę. Jego szybkość spowodowana jest mocno ograniczonym sąsiedztwem w każdej iteracji – brane są pod uwagę tylko takie ruchy, które mają szansę na skrócenie ścieżki krytycznej harmonogramu poprzez zamianę operacji w blokach krytycznych [7]. Widać to w eksperymentach – w ciągu kilku minut otrzymano przybliżone rozwiązania kolejno wszystkich 80 instancji opublikowanych przez Taillard'a dla problemu gniazdowego. Szczegółowe wyniki eksperymentów opisano w rozdziale 4. Zaprojektowany algorytm pozwala rozwiązywać instancje Elastycznego Problemu Gniazdowego oraz jego szczególnych odmian, takich jak klasyczny Problem Gniazdowy, czy Problem Przepływowy.

3.1 Model danych

Podstawowym elementem harmonogramu jest operacja. W tabeli 3.1 zestawiono jej cechy charakterystyczne. Kilka operacji o ustalonej kolejności wykonania tworzy zadanie. Z programistycznego punktu widzenia, zadanie jest tablicą operacji. Strukturę diagramu Gantt'a opisują dwie listy sąsiedztwa. Pierwsza z nich LS_M definiująca kolejność wykonania na maszynach o rozmiarze m, gdzie m jest liczbą maszyn. Natomiast druga, LS_Z będąca kompletną kopią wszystkich operacji o rozmiarze n, gdzie n jest liczbą zadań, i opisuje kolejność wykonania operacji w każdym z poszczególnych zadań. Połączenie tych dwóch zestawów danych, opisujących harmonogram oraz zadania, pozwala wygodnie poruszać się po harmonogramie. Taka reprezentacja danych bardzo przypomina graf skierowany opisany przez Nowickiego i Smutnickiego w [18]. Pozwala ona na łatwą konwersję harmonogramu do formatu HTML, dzięki czemu uzyskano interaktywną graficzną prezentację wyników. Więcej na ten temat opisano w rozdziale 4.

PID	numer zadania, którego elementem jest operacja
ID	numer kolejnościowy operacji w zadaniu **PID**
M	numer maszyny, do której aktualnie przyporządkowano operację
S	czas rozpoczęcia wykonywania operacji w harmonogramie
P	mapa czasów przetwarzania operacji na poszczególnych maszynach, na których można przetwarzać daną operację
C	czas zakończenia operacji na maszynie **M**
B	bufor czasowy (domyślnie = 0, więcej na ten temat w sekcji 3.4)

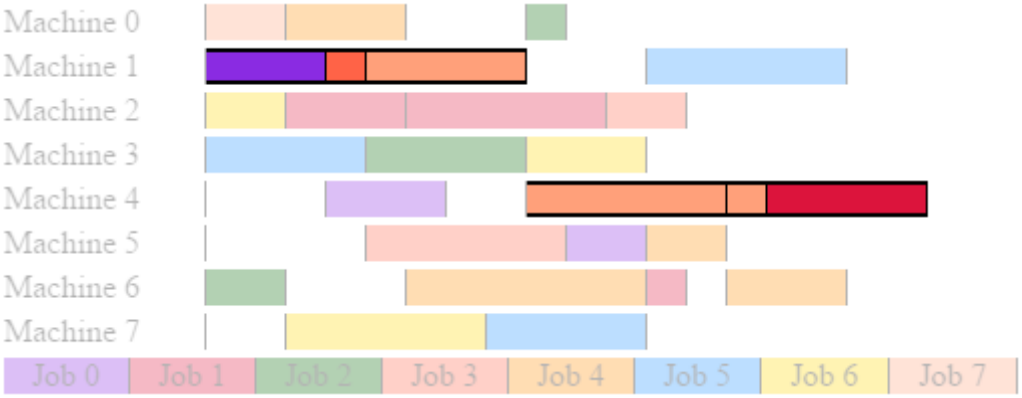
Tabela. 3.1 Elementy opisujące pojedynczą operację harmonogramu.

3.2 Rozwiązanie początkowe

W pierwszej kolejności z pliku wczytywana jest pojedyncza instancja problemu i zapisywana jest do listy sąsiedztwa zadań LS_Z. Następnie, LS_M tworzona jest przy pomocy algorytmu *IniPopGen*. Otrzymana w wyniku tej metody lista LS_M jest w pełni deterministyczna. Oznacza to, że za każdym razem przy określonej kolejności wykonania zadań wylosowanej na początku algorytmu skutkuje identyczną listą LS_M. W opisie tego algorytmu pominięto jedynie kroki związane z genami i chromosomami, które w metodzie Tabu Search nie mają żadnego zastosowania. Nazwę zachowano, aby ułatwić identyfikację w odniesieniu do źródła, z którego pochodzi.

3.3 Tabu Search z nawrotami

Podstawowym elementem na którym bazuje algorytm *TSAB* jest ścieżka krytyczna. Jest to taki ciąg operacji, że opóźnienie którejkolwiek z nich spowodowałoby opóźnienie całego harmonogramu. Harmonogramy w Problemach Gniazdowych mogą mieć więcej niż jedną ścieżkę krytyczną. Każda ścieżka krytyczna dzieli się na bloki [7] – ciągi operacji wykonywanych bez odstępów czasowych na określonej maszynie.



Rysunek 3.1 Przykładowy harmonogram wraz z wyróżnioną ścieżką krytyczną zawierającą dwa bloki krytyczne, każdy składający się z trzech operacji, na maszynach 1 i 4.

Algorytm 1: IniPopGen. Źródło: [1].

Zainicjuj strukturę diagramu Gantt'a;
Wygeneruj losową kolejność wykonania zadań J (np.: J_2, J_1, J_3);
Ustaw $operCount = 1$ oraz $oper = 1$;
Dopóki $(operCount \leqslant totNOper)$ **wykonuj**
    Dla wszystkich *zadań* $i \in J$ **wykonaj**
        Znajdź wszystkie maszyny M, które mogą przetwarzać operację $O_{i,oper}$ (np. M_1, M_3);
        Wygeneruj tablicę czasów zakończenia T operacji $O_{i,oper}$ o rozmiarze równym rozmiarowi M;
        Zainicjuj t_0 i t_1;
        Jeżeli $O_{i,oper}$ *jest pierwszą operacją* **to**
            Ustaw $t_0 = 0$;
        Przeciwnie
            Ustaw t_0 równe czasowi zakończenia przetwarzania poprzednika technologicznego $O_{i,oper-1}$;
        Koniec warunku
        Dla wszystkich *maszyn* $k \in M$ **wykonaj**
            Zidentyfikuj czas przetwarzania $t_{i,oper,k}$ operacji $O_{i,oper}$ na maszynie M_k;
            Jeżeli $t_1 \leqslant t_0$ **to**
                Ustaw $t_1 = 0$;
            Przeciwnie
                Ustaw t_0 równe czasowi zakończenia przetwarzania ostatniej operacji na M_k;
            Koniec warunku
            Jeżeli $t_1 \leqslant t_0$ **to**
                Dodaj $O_{i,oper}$ na M_k z czasem rozpoczęcia t_0 i ustaw $T_{oper,k} = t_0 + t_{i,oper,k}$;
            Przeciwnie, jeżeli $t_{i,oper,k}$ *zawiera się w przedziale czasu* $[t_0, t_1]$*, czyli może być wykonane między uszeregowanymi już operacjami na* M_k **to**
                Wstaw $O_{i,oper}$ na M_k z możliwie najwcześniejszym czasem rozpoczęcia $t_s \geqslant t0$ oraz ustaw $T_{oper,k} = t_s + t_{i,oper,k}$;
            Przeciwnie
                Dodaj $O_{i,oper}$ na M_k z czasem rozpoczęcia t_1 oraz ustaw $T_{oper,k} = t_1 + t_{i,oper,k}$;
            Koniec warunku
        Koniec pętli
        Znajdź minimalny czas zakończenia w tablicy T i przydziel $O_{i,oper}$ do odpowiadającej temu elementowi maszyny. Jeśli istnieje kilka maszyn, spełniające takie przyporządkowanie, należy wybrać pierwszą z nich;
        ++operCount;
    Koniec pętli
    ++oper;
Koniec pętli

W algorytmie Tabu Search, aby był on jak najbardziej wydajny, najważniejsze jest obserwowane otoczenie (sąsiedztwo) bieżącego w danej iteracji rozwiązania oraz wybór naj-

lepszego ruchu, który należy wykonać, aby przetransformować aktualne rozwiązanie do tego wybranego. Za ten obszar w algorytmie *TSAB* odpowiedzialny jest jest zmodyfikowany algorytm *NSP* [18]. Bazuje on na blokach krytycznych składających się z więcej niż jednej operacji. Do wygenerowania otoczenia wykonuje tylko takie ruchy, które polegają na zamianie kolejnością dwóch pierwszych lub dwóch ostatnich operacji w danym bloku. Pomija się jedynie sąsiadów, którzy mogliby zostać wygenerowani przez zamianę dwóch pierwszych operacji w pierwszym bloku ścieżki krytycznej oraz dwóch ostatnich w ostatnim bloku krytycznym. Modyfikacja algorytmu polega na tym, że sprawdza się dopuszczalność wykonania danego ruchu, tzn. czy wykonanie ruchu nie naruszy ograniczeń kolejnościowych w jakimkolwiek zdaniu w harmonogramie. Pierwotnie, algorytm *NSP* nie potrzebował takiego sprawdzenia, ponieważ nie był stosowany do Elastycznych Problemów Gniazdowych.

Algorytm 2: NSP. Źródło: [18].

Przyjmuje: Permutację π, zestaw możliwych ruchów $V(\pi) \neq \emptyset$, listę tabu T oraz najkrótszy osiągnięty czas wykonania harmonogramu C^*_{max}.

Zwraca: Wybrany (najlepszy) ruch v', powstałą przez wykonanie ruchu v' permutację sąsiada π' oraz zmodyfikowaną listę tabu T'.

1 Znajdź zestaw zakazanych ruchów, które poprawiają aktualne rozwiązanie $A = \{v \in V(\pi) \cap T : C_{max}(Q(\pi, v)) < C^*_{max}\}$, gdzie $Q(\pi, v)$ zwraca wynik transformacji harmonogramu z permutacji π poprzez wykonanie ruchu v;
2 **Jeżeli** $(V(\pi) \setminus T) \cup A \neq \emptyset$ **to**
3 | wybierz $v' \in (V(\pi) \sim T) \cup A$, takie że $C_{max}(Q(\pi, v')) = min\{C_{max}(Q(\pi, v)) : v \in (V(\pi) \setminus T) \cup A\}$
4 | i przejdź do kroku 14;
5 **Koniec warunku**
6 **Jeżeli** $|V(\pi)| = 1$ **to**
7 | wybierz $v' \in V(\pi)$;
8 **Przeciwnie**
9 | **Powtarzaj**
10 | | dopisuj do listy tabu T ostatnio dodany do niej element T_{maxt};
11 | **dopóki** $(V(\pi) \setminus T) \neq \emptyset$;
12 | a następnie wybierz $v' \in (V(\pi) \setminus T)$;
13 **Koniec warunku**
14 Ustaw $\pi' := Q(\pi, v')$ oraz $T' := T \cup \{\bar{v'}\}$, gdzie $\bar{v'}$ jest ruchem odwrotnym do v', tzn. $Q(Q(\pi, \bar{v'}), v') = Q(Q(\pi, v'), \bar{v'}) = \pi$;

Obserwując konstrukcję algorytmu *NSP* nietrudno zauważyć, że aby móc oceniać wygenerowane sąsiedztwo aktualnego harmonogramu wymagane jest wyznaczenie czasów rozpoczęcia (pośrednio również czasów zakończenia) wszystkich operacji za każdym razem kiedy wykonamy jakikolwiek ruch (zamiana kolejności operacji na maszynie). Złożoność obliczeniowa algorytmu naprawy jest o tyle istotna, że jest to najczęściej wywoływany element w trakcie pracy algorytmu *NSP*, który z kolei jest później wielokrotnie wywoływany w *TSAB*. Zaprojektowany algorytm *RepairPerm* posiada złożoność liniową $O(totNOper)$, gdzie $totNOper$ jest łączną liczbą operacji w całym harmonogramie. Ważne jest, że algorytm ten zapętli się jeżeli podana na jego wejściu struktura diagramu Gantta G zawiera niedopuszczalną listę sąsiedztwa LS_M, tzn. gdy naruszone zostaną ograniczenia kolejnościowe. Należy więc upewnić się, że wykonanie ruchu v' wygeneruje poprawną permutację.

Algorytm 3: RepairPerm.

Przyjmuje: Strukturę diagramu Gantta G o nieustalonych czasach rozpoczęcia (i zakończenia) operacji na listach sąsiedztwa LS_M i LS_Z oraz liczbę wszystkich operacji $totNOper$

Zwraca: Naprawioną permutację

1 Utwórz macierz zerową O o liczbie wierszy równej liczbie wierszy LS_Z i liczbie kolumn o 1 większej niż w tej liście. Pierwszy element każdego wiersza tej macierzy zainicjuj wartością 1;
2 Utwórz tablicę U o długości m (liczba maszyn) i ją również wypełnij zerami;
3 Ustaw $uszeregowano := 0$;
4 **Dopóki** $uszeregowano < totNOper$ **wykonuj**
5 **Dla wszystkich** *maszyn* $M_k \in M$ **wykonaj**
6 Ustaw indeks operacji do uszeregowania $l := U_k$;
7 **Jeżeli** $l < |M_k|$ **to**
8 Pobierz operację $op := M_{k,l}$;
9 **Jeżeli** $O_{op.PID,op.ID} = 1$ **to**
10 Oznacz operację jako uszeregowaną: $O_{op.PID,op.ID+1} = 1$, $++U_k$, $++uszeregowano$;
11 Ustaw czas rozpoczęcia operacji $op.S := max\left\{0, LS_{M(k,l-1)}.C, LS_{Z(op.PID,op.ID-1)}.C\right\}$;
12 Zapisz nowe czasy rozpoczęcia na listach sąsiedztwa: $LS_{M(k,l)}.S := op.S$ $LS_{Z(op.PID,op.ID)}.S := op.S$;
13 **Koniec warunku**
14 **Koniec warunku**
15 **Koniec pętli**
16 **Koniec pętli**

Idea algorytmu *TSAB*, polega na wyznaczeniu w każdej iteracji najlepszego kierunku poszukiwań za pomocą algorytmu *NSP*. Tradycyjny algorytm *TS* zakończyłby pracę po *maxIter* liczbie operacji, jednak algorytm *TSAB*, oprócz listy tabu posiada również listę L zawierającą *maxBt* ostatnio zarejestrowanych najlepszych rozwiązań. Po osiągnięciu *maxIter* operacji następuje powrót do najdawniej zapisanego rozwiązania z listy L, przy czym z listy możliwych do wykonania ruchów usuwa się wszystkie dotychczas wykonane z tego rozwiązania ruchy.

3.4 Awarie

Wybrany model zaplanowanych awarii wymaga jedynie, aby uzupełnić strukturę diagramu Gantta o pojedynczą *operację*, która opisuje czas rozpoczęcia **S**, czas trwania **P(M)** oraz czas zakończenia **C** awarii na określonej maszynie **M**. W ten sposób, dodając taką operację *na sztywno* do listy sąsiedztwa LS_M wszystkie poprzednio zadeklarowane algorytmy mogą działać bez zmian.

Algorytm 4: TSAB. Źródło [18].

Przyjmuje: Najlepszą dotychczas uzyskaną permutację π^*, aktualną permutację $\pi = \pi^*$, najkrótszy osiągnięty czas wykonania harmonogramu $C^* = C_{max}(\pi^*)$, pustą listę tabu $T = \emptyset$, pustą listę nawrotów $L = \emptyset$, $iter = 0$, $l = 0$, $zapisz = 1$.

Zwraca: Wybraną (w przybliżeniu najlepszą) permutację uzyskaną na podstawie danych wejściowych o czasie wykonania $C_{max}(\pi^*)$

1 **Powtarzaj**
2 | ++iter;
3 | Znajdź zbiór możliwych do wykonania ruchów $V(\pi)$;
4 | **Jeżeli** $V(\pi) = \emptyset$ **to**
5 | | STOP. OPTIMUM;
6 | **Koniec warunku**
7 | Znajdź ruch $v' \in V(\pi)$, sąsiada $\pi' = Q(\pi, v')$ i zmodyfikowaną tablicę tabu T' za pomocą algorytmu *NSP*;
8 | **Jeżeli** $zapisz = 1$ *oraz* $V(\pi) \setminus \{v'\} \neq \emptyset$ **to**
9 | | $++l$; $L := L \cup (\pi, V(\pi) \setminus \{v'\}, T)$;
10 | **Koniec warunku**
11 | Ustaw $\pi := \pi'$, $T := T'$, $zapisz := 0$;
12 | **Jeżeli** $C_{max}(\pi) < C^*$ **to**
13 | | Ustaw $\pi := \pi'$, $C^* := C_{max}(\pi)$, $iter := 0$, $zapisz := 1$;
14 | **Koniec warunku**
15 | **Jeżeli** $iter \leqslant maxIter$ **to**
16 | | Idź do kroku 1;
17 | **Koniec warunku**
18 | **Jeżeli** $l \neq 0$ **to**
19 | | Ustaw $(\pi, V(\pi), T) := L_l$, $--l$, $iter := 1$, $zapisz := 1$;
20 | **Koniec warunku**
21 **dopóki** $iter \leqslant maxIter$;

Rozdział 4

Eksperymenty obliczeniowe

Zaimplementowany algorytm przetestowano na wszystkich 80 instancjach Taillarda do klasycznego Problemu Gniazdowego, oraz na jednej z instancji Elastycznego Problemu Gniazdowego przedstawionego w [10].

4.1 Wizualizacja wyników

W celu automatyzacji graficznej prezentacji otrzymanych w wynikach harmonogramów napisano prostą bibliotekę pozwalającą na budowanie stron HTML. Składa się ona z czterech klas:

ContentHTML interfejs po którym dziedziczą wszystkie klasy opisane dalej.

TextContentHTML implementuje `ContentHTML`. Klasa opisująca element HTML będący zwykłym tekstem.

TagContentHTML implementuje `ContentHTML`. Klasa opisująca tag HTML. Posiada on swoją nazwę, mapę parametrów oraz listę elementów potomnych typu `ContentHTML`.

PageHTML implementuje `ContentHTML`. Klasa opisująca stronę HTML. Składa się z tagów HTML `HEAD` oraz `BODY`.

Ich pliki źródłowe dołączono do dodatku A. Podczas projektowania tej biblioteki wykorzystano wzorce projektowe Kompozyt oraz Builder [5]. Pozwoliło to na znaczące uproszczenie struktury klas i lepszy podział ich odpowiedzialności. Co najważniejsze, niezależnie od poziomu zagłębienia w kodzie HTML, na każdym jego elemencie można wywołać metodę `toString()`, która zwróci wygenerowany kod HTML danego elementu i wszystkich jego elementów potomnych. Dzięki temu, po poprawnym zbudowaniu struktury strony HTML, aby otrzymać jej kod źródłowy wystarczy wywołać metodę `toString()` na głównym obiekcie agregującym typu `PageHTML`.

4.2 Parametryzacja

Przyjęto:

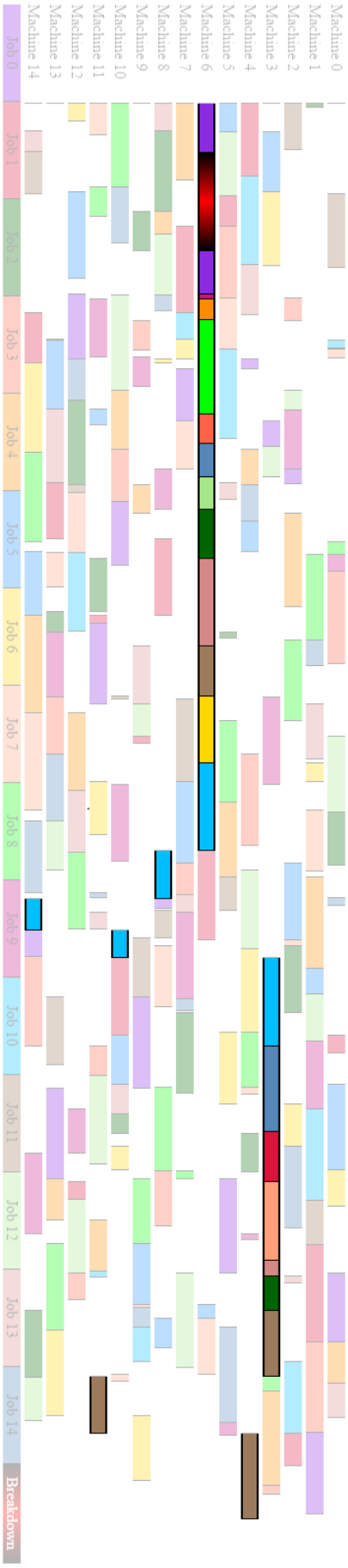
- długość listy tabu: 7,
- długość listy najlepszych rozwiązań: 5,
- $maxIter = 10$.

4.3 Statystyki

Dla wszystkich instancji aplikację uruchomiono 5 razy. Uzyskane wyniki zestawiono w tabeli 4.1.

Nr instancji	**Średni czas obliczeń [s]**	C_{max} – **MIN**	C_{max} – **MAX**
0	1,08	1412	1433

Tabela. 4.1 Statystyki po 5 uruchomieniach aplikacji. W każdym przypadku awaria nastąpiła w chwili czasu 50 na maszynie nr 6. i trwała 100 jednostek czasu.



Rysunek 4.1 Wygenerowany za pomocą napisanej biblioteki diagram Gantta. Czarno-czerwonym gradientem zaznaczono moment awarii maszyny 6.

Wnioski i uwagi

Wszystkie postawione cele projektu zostały zrealizowane. W przyszłości można by rozszerzyć możliwe do wykonania ruchy (algorytm NSP) o zamianę operacji między maszynami, aby uzyskać jeszcze krótsze harmonogramy przy zachowaniu ich odporności na zmiany wynikające z awarii.

Wszystkie pliki źródłowe można znaleźć w repozytorium `https://github.com/kejn/job-shop-HGA/tree/tabuSearch/jobshop_HGA`.

Dodatek A

Biblioteka do generowania HTML dla C++

A.1 Interfejs ContentHTML

```
/*
 * ContentHTML.h
 *
 *  Created on: 22 kwi 2016
 *      Author: Kamil
 */

#ifndef HTML_CONTENTHTML_H_
#define HTML_CONTENTHTML_H_

#include <string>

class ContentHTML {
protected:
  ContentHTML(){}
public:
  virtual std::string toString() = 0;
  virtual ~ContentHTML(){}
};

#endif /* CONTENTHTML_H_ */
```

A.2 Klasa TextContentHTML.h

```
/*
 * TextContentHTML.h
 *
 *  Created on: 22 kwi 2016
 *      Author: Kamil
 */
```

```
#ifndef HTML_TEXTCONTENTHTML_H_
#define HTML_TEXTCONTENTHTML_H_

#include <string>

#include "ContentHTML.h"

class TextContentHTML: public ContentHTML {
protected:
  std::string text;
public:
  TextContentHTML(const std::string& text) :
      text(text) {}
  virtual std::string toString();
  virtual ~TextContentHTML() {
  }
};

#endif /* HTML_TEXTCONTENTHTML_H_ */
```

```
/*
 * TextContentHTML.cpp
 *
 *  Created on: 22 kwi 2016
 *      Author: Kamil
 */

#include "../../inc/html/TextContentHTML.h"

std::string TextContentHTML::toString() {
  return text;
}
```

A.3 Klasa TagContentHTML

```
/*
 * TagHTML.h
 *
 *  Created on: 20 kwi 2016
 *      Author: Kamil
 */

#ifndef HTML_TAGCONTENTHTML_H_
#define HTML_TAGCONTENTHTML_H_

#include <map>
#include <string>
#include <vector>

#include "ContentHTML.h"
```

```
#include "../Oper.h"

using uint = unsigned int;

class TagContentHTML: virtual public ContentHTML {
protected:
  std::string name;
  std::map<std::string, std::string> params;
  std::vector<ContentHTML*> children;
public:
  TagContentHTML(const std::string &name) :
      name(name) {
  }
  TagContentHTML(const std::string &name,
      std::map<std::string, std::string> params) :
      name(name), params(params) {
  }
  TagContentHTML(const std::string &name,
      std::map<std::string, std::string> params,
      std::vector<ContentHTML*> children) :
      name(name), params(params), children(children) {
  }

  static TagContentHTML * forTDEmptyOperation(uint width);
  static TagContentHTML * forTDOperation(Oper oper, std::string bgColor,
      uint htmlScale, bool inCriticalPath = false);
  static TagContentHTML * forTDJobLegend(uint jobNumber, std::string bgColor,
      uint cMax, std::string text="");

  void addParam(const std::string &param, const std::string &value);
  void addChild(ContentHTML * const & childElement);

  virtual std::string toString();
  virtual ~TagContentHTML();
};

#endif /* HTML_TAGCONTENTHTML_H_ */
```

```
/*
 * TagHTML.cpp
 *
 *  Created on: 20 kwi 2016
 *      Author: Kamil
 */

#include "../../inc/html/TagContentHTML.h"

#include <iterator>
#include <sstream>
#include <utility>
```

```
#include "../../inc/html/TextContentHTML.h"
#include "../../inc/util/stringUtil.h"

using namespace std;

void TagContentHTML::addParam(const string& param, const string& value) {
  params.insert(pair<string, string>(param, value));
}

void TagContentHTML::addChild(ContentHTML * const & childElement) {
  if (childElement == nullptr) {
    return;
  }
  children.push_back(childElement);
}

std::string TagContentHTML::toString() {
  stringstream ss;
  ss << "<" << name;
  for (const auto & param : params) {
    ss << " " << param.first << "=\"" << param.second << "\"";
  }
  if (children.empty() && name.compare("body") && name.compare("head")) {
    ss << " />\n";
  } else {
    ss << ">";
    for (const auto & childElement : children) {
      ss << '\n' << childElement->toString();
    }
    ss << " </" << name << ">";
  }
  return ss.str();
}

TagContentHTML* TagContentHTML::forTDOperation(Oper oper, string bgColor,
    uint htmlScale, bool inCriticalPath) {
  TagContentHTML* tdOperationTag = new TagContentHTML("td");
  string style = "width: "
      + stringUtil::toString(htmlScale * oper.getProcessingTime() - 1)
      + "px;" + "background-color: " + bgColor + ";";

  if (inCriticalPath) {
    style += " border-top: 2px black ridge;";
    style += " border-bottom: 2px black ridge;";
  }

  tdOperationTag->addParam("style", style);
  tdOperationTag->addParam("title", oper.toString());

  string jobNumber = "job" + stringUtil::toString(oper.getPid());
  if (inCriticalPath) {
    tdOperationTag->addParam("class", jobNumber + " cpath");
```

```
    tdOperationTag->addParam("onclick", "cpath()");
  } else {
    tdOperationTag->addParam("class", jobNumber);
  }
  tdOperationTag->addParam("onmouseover", "mOver('" + jobNumber + "')");
  tdOperationTag->addParam("onmouseout", "mOut()");

  return tdOperationTag;
}

TagContentHTML::~TagContentHTML() {
  vector<ContentHTML*>::iterator iter = children.begin();
  for (; iter != children.end(); ++iter) {
    if (*iter != nullptr) {
      delete *iter;
    }
  }
}

TagContentHTML* TagContentHTML::forTDEmptyOperation(uint width) {
  TagContentHTML *tdTag = new TagContentHTML("td");

  string style("width: ");
  style += stringUtil::toString(width) + "px;";

  tdTag->addParam("style", style);
  return tdTag;
}

TagContentHTML* TagContentHTML::forTDJobLegend(uint jobNumber,
    std::string bgColor, uint cMax, std::string text) {
  TagContentHTML *td = new TagContentHTML("td");
  string style = "text-align: center; width: 100px; background-color: "
      + bgColor + ";";
  td->addParam("style", style);

  string jobNumberClass = "job" + stringUtil::toString(jobNumber);
  string cMaxString = "cMax: " + stringUtil::toString(cMax);
  td->addParam("title", cMaxString);
  td->addParam("class", jobNumberClass);
  td->addParam("onmouseover", "mOver('" + jobNumberClass + "')");
  td->addParam("onmouseout", "mOut()");

  if (text.empty()) {
    td->addChild(
        new TextContentHTML("Job " + stringUtil::toString(jobNumber)));
  } else {
    td->addChild(new TextContentHTML(text));
  }

  return td;
}
```

A.4 Klasa PageHTML

```
/*
 * GanttHTML.h
 *
 *  Created on: 20 kwi 2016
 *      Author: Kamil
 */

#ifndef HTML_PAGEHTML_H_
#define HTML_PAGEHTML_H_

#include <string>

#include "TagContentHTML.h"

class PageHTML: public ContentHTML {
  TagContentHTML head;
  TagContentHTML body;
public:
  PageHTML() :
      head("head"), body("body") {}

  enum Section {
    HEAD,
    BODY
  };

  void add(const PageHTML::Section& section,
      ContentHTML * const & childElement);

  void addContentType(std::string contentType);
  void addTitle(std::string title);
  void addStyle(std::string url);
  void addScript(std::string url);

  virtual std::string toString();
  virtual ~PageHTML() {
  }
};

#endif /* HTML_PAGEHTML_H_ */
```

```
/*
 * GanttHTML.cpp
 *
 *  Created on: 20 kwi 2016
 *      Author: Kamil
 */

#include "../../inc/html/PageHTML.h"
```

```
#include <sstream>
#include <string>
#include <vector>

#include "../../inc/html/TextContentHTML.h"

using namespace std;

void PageHTML::add(const PageHTML::Section& section,
    ContentHTML * const & childElement) {
  if (section == Section::HEAD) {
    head.addChild(childElement);
  } else if (section == Section::BODY) {
    body.addChild(childElement);
  }
}

void PageHTML::addContentType(string contentType) {
  TagContentHTML* meta = new TagContentHTML("meta");
  meta->addParam("http-equiv", "Content-Type");
  meta->addParam("content", contentType);
  add(Section::HEAD, meta);
}

void PageHTML::addTitle(string title) {
  TagContentHTML* titleTag = new TagContentHTML("title");
  titleTag->addChild(new TextContentHTML(title));
  add(Section::HEAD, titleTag);
}

void PageHTML::addStyle(string url) {
  TagContentHTML* linkTag = new TagContentHTML("link");
  linkTag->addParam("rel", "stylesheet");
  linkTag->addParam("type", "text/css");
  linkTag->addParam("href", url);
  add(Section::HEAD, linkTag);
}

void PageHTML::addScript(string url) {
  TagContentHTML* scriptTag = new TagContentHTML("script");
  scriptTag->addChild(new TextContentHTML(""));
  scriptTag->addParam("src", url);
  add(Section::HEAD, scriptTag);
}

string PageHTML::toString() {
  stringstream ss;
  ss << "<!doctype html>\n";
  ss << "<html>\n";
  ss << head.toString() << endl;
  ss << body.toString() << endl;
```

```
  ss << "</html>" << endl;

  return ss.str();
}
```

A.5 Style CSS

```
body,div,table,tr,td {
  margin: 0px;
  padding: 0px;
}

div {
  width: 6000px;
}

table {
  border-spacing: 0px;
  padding: 2px;
}

td {
  border-right: 1px black solid;
}

/* breakdown */
td.job4294967295 {
  color: white;
  background-image: linear-gradient(to left, black, red, black);
}
```

A.6 JavaScript

```
var tds = document.getElementsByTagName('td');
var over = 0;
var clicked = 0;

function mOver(classname) {
  if (!clicked) {
    over = 1;
    var jobs = document.getElementsByClassName(classname);
    for (var i = 0; i < tds.length; ++i) {
      tds[i].style.opacity = 0.3;
    }
    for (var i = 0; i < jobs.length; ++i) {
      jobs[i].style.opacity = 1.0;
    }
  }
```

```
}

function mOut() {
  if (!clicked) {
    over = 0;
    setTimeout(function() {
      if (over == 1)
        return;
      for (var i = 0; i < tds.length; ++i) {
        tds[i].style.opacity = 1.0;
      }
    }, 500);
  }
}

function cpath() {
  if (!clicked) {
    clicked = 1;
    for (var i = 0; i < tds.length; ++i) {
      tds[i].style.opacity = 0.3;
    }
    var jobs = document.getElementsByClassName('cpath');
    for (var i = 0; i < jobs.length; ++i) {
      jobs[i].style.opacity = 1.0;
    }
  } else {
    clicked = 0;
  }
}
```

Bibliografia

[1] N. Al-Hinai, T. Y. ElMekkawy. An efficient hybridized genetic algorithm architecture for the flexible job-shop scheduling problem. *Flexible Services and Manufacturing Journal*, 23:64–85, 2011.

[2] N. Al-Hinai, T. Y. ElMekkawy. Robust and stable flexible job-shop scheduling with random machine breakdowns using a hybrid genetic algorithm. *International Journal of Production Economics*, 132(2):279–291, August 2011.

[3] P. Brandimarte. Routing and scheduling in a flexible job shop by tabu search. *Annals of Operations Research*, 41:157–183, 1993.

[4] A. C. Diamantidis, C. T. Papadopoulos. Exact analysis of a two-station one-buffer flow line with parallel unreliable machines. *European Journal of Operational Research*, 197:572–580, 2009.

[5] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[6] M. R. Garey, D. S. Johnson, R. Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2):117–129, 1976.

[7] J. Grabowski, E. Nowicki, S. Zdrzałka. A block approach for single machine scheduling with release dates and due dates. *European Journal of Operational Research*, 26:278–285, 1986.

[8] A. Hamzadayi, G. Yidiz. Event driven strategy based complete rescheduling approaches for dynamic m identical parallel machines scheduling problem with a common server. *Computers & Industrial Engineering*, 91:66–84, January 2016.

[9] L. Jun-qing, P. Quan-ke, M. Kun. A discrete teaching-learning-based optimisation algorithm for realistic flowshop rescheduling problems. *Engineering Applications of Artificial Inteligence*, 37(25):279–292, January 2015.

[10] I. Kacem, S. Hammadi, P. Brone. Approach by localization and multi-objective evolutionary optimization for flexible job-shop scheduling problems. *IEEE Transactions on Systems, Man and Cybernetics*, 32(1):1–13, 2002.

[11] I. Kacem, S. Hammadi, P. Brone. Pareto-optimality approach for flexible job-shop scheduling problems: hybridization of evolutionary algorithms and fuzzy logic. *Mathematics and Computers in Simulation*, 60:245–276, 2002.

[12] K. Katragjinia, E. Valladaa, R. Ruiza. Flow shop rescheduling under different types of disruption. *International Journal of Production Research*, 51(3):780–797, 2013.

[13] D. Y. Lee, F. DiCesare. Scheduling flexible manufacturing systems using petri nets and heuristic search. *IEEE Transactions on Robotics and Automation*, 10(2):123–132, 1994.

[14] J. Liu, S. Yang, A. Wu, S. J. Hu. Multi-state throughput analysis of two-stage manufacturing system with parallel unreliable machines and a finite buffer. *European Journal of Operational Research*, 219(2):296–304, June 2012.

[15] L. Liu, H. Zhou. On the identical parallel-machine rescheduling with job rework disruption. *Computers & Industrial Engineering*, 66:186–198, September 2013.

[16] K. Mesghouni, S. Hammadi, P. Brone. Evolution programs for job-shop scheduling. *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics, Orlando, Florida*, 1:720–725, 1997.

[17] M. Nawaz, E. E. Enscore Jr, I. Ham. A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *Omega*, 11(1):91–95, 1983.

[18] E. Nowicki, C. Smutnicki. A fast taboo search algorithm for the job shop problem. *Management Science*, 42(6):797–813, 1996.

[19] J. C. Tay, D. Wibowo. An effective chromosome representation for evaluating flexible job shop schedules. *Genetic and Evolutionary Computation*, 3103:210–221, 2004.

[20] D.-J. Wang, F. Lio, Y.-Z. Wang, Y. Jin. A knowledge-based evolutionary proactive scheduling approach in the presence of machine breakdown and deterioration effect. *Knowledge-Based Systems*, 90:70–80, December 2015.

[21] J. Xiong, L. ning Xing, Y. wu Chen. Robust scheduling for multi-objective flexible job-shop problems with random machine breakdowns. *International Journal of Production Economics*, 141(1):112–126, January 2013.

Spis rysunków

1.1 Schemat kolejkowania nadchodzących zadań oraz dodawanie ich do puli zadań, dla których zostaje wyznaczony nowy harmonogram 6
1.2 Poglądowy schemat analizowanej w pracy [14] linii produkcyjnej. 7
1.3 Tradycyjne harmonogramowanie 8
1.4 Harmonogramowanie odporne 8

3.1 Przykładowy harmonogram wraz z wyróżnioną ścieżką krytyczną 13

4.1 Wygenerowany za pomocą napisanej biblioteki diagram Gantta. 20

Spis tabel

3.1 Elementy opisujące pojedynczą operację harmonogramu. 13

4.1 Statystyki po 5 uruchomieniach aplikacji. W każdym przypadku awaria nastąpiła w chwili czasu 50 na maszynie nr 6. i trwała 100 jednostek czasu. 19