Rozdział 1

Wykłady 3 i 4

1.1 Poszukiwanie ewolucyjne

Celem rozważanego podejścia jest znalezienie wartości $\operatorname{argmin}_{x \in X} F(x)$. Metody ewolucyjne się używa najczęściej wtedy gdy a) problem nie jest ciągły (nie mamy do dyspozycji gradientu), b) mamy małą wiedzę/intuicję o badanym przedmiocie (nie znamy lepszej metody optymalizacyjnej). Minusy metody to wolna zbieżność do rozwiązań optymalnych (szczególnie wtedy gdy jest duży wymiar przestrzni atrybutów). W praktycznych zastosowaniach do uzyskania dobrych efektów często się stosuje metody hybrydowe.

Poszukiwanie ewolucyjne polega na stworzeniu zbioru rozwiązań $W \subset X$ zwanego populacją. Każde rozwiązanie jest kodowane przez zbiór atrybutów (domyślnie chodzi o geny, chromosomy, etc). Następnie w każdej epoce dopuszcza się procesy mające podstawę w prawdziwej ewolucji, a mianowicie występuję reprodukcja, krzyżowanie i mutacja oraz przeżycie lub selekcja.

Do implementacji GS (genetic search) potrzebne jest dookreślenie następujących punktów:

- *sposobu kodowania atrybutów:* każde rozwiązanie x jest kodowanie przez jego zbiór atrybutów;
- *postać funkcji adaptacji:* opisuje na ile dobre jest dane rozwiązanie, typowo określone przez funkcję kosztu;
- *schemat wyboru puli rodzicielskiej:* sposób wybierania osobników którzy będą dalej przekazywać geny;
- *schemat kojarzenia rodziców:* staramy sie nie wybierać od siebie za bardzo oddalonych genetycznie;
- *operatory krzyżowania:* opisuje sposób tworzenia genów potomstwa;
- *schemat mutacji:* aby umożliwić przeszukiwanie szerszej przestrzeni potencjalnych rozwiązań konieczne są mutacje;
- *schemat przeżywania:* do następnego pokolenia przechodzą tylko lepiej dostosowane osobniki.

Schemat klasycznego podejścia do algorytmów genetycznych jest dosyć jasny[1]:

[1] https://ibug.doc.ic.ac.uk/media/uploads/documents/courses/GeneticAlgorithm-tutorial.pdf

1. generowanie populacji początkowej: w klasycznych algorytmach genetycznych każdy element jest ciągiem binarnym ustalonej długości L, nazywamy genem/chromosomem;

2. po stworzeniu populacji, każdy element jest opisywany przy pomocy funkcji adaptacji, w klasycznym podejściu jest definiowane jako $p_i = f_i/\bar{f}$, gdzie f_i jest ewaluacją funkcji kosztu na każdym elemencie, i $\bar{f}$ jest średnią wartością na elementach w populacji;

3. wykonanie algorytmu genetycznego widzimy jako proces dwuetapowy. Startujemy z bieżacej populacji. Dokonujemy selekcji by wybrać pulę rodzicielską (populacja pośrednia). Następnie tworzymy następną populację przez odpowiednie kojarzenie rodziców i mutację. Klasycznie wybieramy punkt x_i z prawdopodobieństwem p_i

Ogólnie o algorytmach genetycznych jest w Handbook:[Cha00] (tam jest też rozdział 8 w którym jest dokładnie opisany schemat GA dla problemu gniazdowego, który będzie nas szczególnie interesował).

1.2 Wstęp do szeregowania zadań

1.2.1 Czym jest problem szeregowania zadań?

Obecnie w przemyśle przeważającym modelem produkcji jest wytwarzanie w postaci zadań produkcyjnych, poprzez budowę części i ich montaż [Smu12]. Potrzebne są wobec tego metody takiego planowania produkcji (ang. scheduling, production planning), które będzie optymalizowało wykorzystanie zasobów.

Trzeba pamietać, że problem jest problemem trudnym o wysokiej złożoności obliczeniowej, zwykle wymagając algorytmów o wykładniczej złożonosci obliczeniowej dla znalezienia rozwiązania optymalnego. Konieczne jest więc korzystanie z szeregu różnego rodzaju uproszczeń i założeń w definicji, dla znalezienia rozwiązań sub-optymalnych. Konieczne będzie wykorzystanie metod programowania matematycznego, programowania dynamicznego, także z ograniczeniami (ang. constraints), a także wykorzystanie heurystyk.

Zadanie szeregowania zadań można najlepiej opisać przez przedstawienie go w środowisku przemysłowym. Ze względu na literaturę przedmiotu, która jest w przeważającej części w języku angielskim, będziemy dodawać angielskie tłumaczenia w nawiasach. Będziemy też używać najczęstrszego słownictwa występującego w literaturze polskiej.

Zlecenia (ang. orders) składane są do zakładu przemysłowego. Te zlecenia są zwykle opisane w postaci szeregu *zadań* (ang. jobs), z których każde jest przetwarzane jako ciąg *operacji* (ang. tasks, processes). Każda z operacji ma określony czas wykonania (precyzyjnie lub też probabilistycznie w postaci rozkładu), a kolejność operacji musi być zachowana. W oczywisty sposób mogą nastąpić nieoczekiwane wydarzenia, które zaburzą przebieg wykonania zlecenia [Pin00].

W takiej sytuacji celem jest rozwiązanie problemu szeregowania zadań poprzez zbudowanie takiego *harmonogramu* (ang. schedule), który w największym stopniu pozwoli na precyzyjne dotrzymanie zobowiązań.

Hala produkcyjna (fabryka, ang. job-floor czy shop-floor) składa się z szeregu *zasobów* (ang. resources) takich jak urządzenia (maszyny, ang. machines), personel, materiały, czy kapitał.

Ponieważ wszystko odbywa się w czasie (który jest zwykle dyskretyzowany), poszczególne zadania i operacje mają swoje atrybuty, z których najważniejszymi są

- *czas gotowości* (ang. release date),
- *czas zakończenia* (ang. due time) (to może być także *żądany* czas wykonania),
- *czas przetwarzania* (ang. processing time) zwykle odnoszący się do pojedynczych operacji na konkretnych maszynach,
- *priorytet* (ang. weight) każdego zadania lub zlecenia,
- *czas rozpoczęcia* (ang. starting time) wykonania zadania na danej maszynie, czas zakończenia (ang. completion time) wykonania zadania na maszynie (czyli operacji).

Uwaga 1.1. W większości problemów zakładamy, że dane zadanie składając się z wielu operacji każdą wykonuje na innej maszynie. Jest to, oczywiście, pewne uproszczenie problemu pozwalającego na jego rozwiązanie.

Innymi atrybutami są także możliwość *przerywalności* i *podzielności* zadań i operacji. Pozwalają on, odpowiednio, na przerwanie zadania w trakcie sesji, oraz na wykonanie zadania (operacji) na osobnych maszynach.

Wykonanie zadań podlega szeregu ograniczeń (ang. constraints). Dotyczą zwykle zasobów – liczby maszyn, dostępności materiałów, ograniczenia personelu, np. braku osób umiejących obsługiwać konkretne maszyny. Powoduje to kolejne utrudnienie problemu.

Dla prawidlowego sformułowania problemu należy zdefiniować różne warianty modeli, możliwe ograniczenia, co jest celem budowy harmonogramu i w jaki sposób można dwa harmonogramy porównać.

1.2.2 Modele

Należy najpierw wprowadzić najważniejsze modele w rosnącym stopniu złożoności (częściowo za [Pin00]).

Pojedynczej maszyny gdzie dostępne jest tylko jedno urządzenie (czy zasób) przetwarzające. To zadanie jest rozwiązywane zwykle przy wykorzystaniu szeregu heurystyk i rozwiązanie jest wykorzystywane po redukcji zadań bardziej złożonych.

Równoległych maszyn gdzie dostępnych jest wiele maszyn (czy innych zasobów) tego samego typu. Uogólnienie poprzedniego i także wykorzystywane przy redukcji bardziej złożonych.

Przepływowy (ang. flow shop) w którym obsługiwanych jest wiele maszyn i wiele zadań, jednak wszystkie mają taką samą sekwencję operacji na maszynach. Zadania mogą być przestawiane między maszynami.

Gniazdowy (ang. job shop) składający z wielu maszyn i wielu zadań o *różnych* sekwencjach. Ograniczeniem (upraszczającym problem) może być zastrzeżenie, że jedno zadanie może być przetwarzane na pojedynczej maszynie tylko raz.
Dany jest zbiór zadań $\{J_1, \ldots, J_n\}$ *na zbiorze maszyn/zasobów* $\{R_1, \ldots, R_q\}$. *Każda praca* J_j *składa się z ciągu operacji (tasks)* $\{t_{i_1 j}, \ldots, t_{i_m j}\}$ *które musza być wykonane kolejno, i każda operacja rezerwuje jeden z zasobów (oznaczamy przez* $R(t)$ *zasób rezerwowany dla operacji* t*).*
Każda operacja zadania j *na maszynie* i *ma określony wstępnie czas wykonania* p_{ij} *i start* s_{ij}*, który musimy dobrać przy budowie harmonogramu.*

Uogólniony gniazdowy jest uogólnieniem modelu gniazdowego, w którym kolejność wykonywania tasków jest zadana za pomocą odpowiedniego grafu skierowanego, a nie ciągu (listy) [zostanie doprecyzowane później].

W naszym wykładzie będziemy się interesować najbardziej problemem gniazdowym (oraz uogólnionym gniazdowym)

1.2.3 Ograniczenia

Podstawową rolę w tworzeniu harmonogramu grają ograniczenia (ang. constraints). Oto najważniejsze z nich.

precedensji (ang. precedence) operacji oznaczające, że dane zadanie (operacja) może być uruchomione dopiero gdy pewien zbiór innych zadań będzie ukończony $s_{ij} + p_{ij} \leqslant s_{ik}$ gdzie k jest indeksem operacji następujacej po j,

pojemności określajace, że dwie operacje nie mogą w jednym czasie wykorzystywać tego samego zasobu $s_{ij} + p_{ij} \leqslant s_{ik} \, \forall \, k \neq j$

wybieralność (ang. eligibility) maszyn, gdy dane zadania mogą być wykonywane tylko na konkretnych maszynach; np. jeśli jest to zwiazane z jakością wykonania zależną od maszyny,

personelu gdy tylko podgrupa personelu może obsługiwać niektóre maszyny, lub też personelu jest ograniczona liczba,

rutowania (ang. routing) gdy zadanie ma określoną kolejność wykonywania operacji,

dostawy i przechowywania materiałów,

transportowe.

1.2.4 Cele optymalizacji i metody porównywania

W zadaniach harmonogramowania określone konieczne jest zdefiniowania *funkcji kosztu*, którą należy minimalizować. W zależności od jej wyboru, możliwe będzie wykorzystanie pewnej grupy algorytmów, i różnić się będzie rozwiązanie. Oto najczęstsze z nich.

maksymalizacja przepustowości polegająca na maksymalizacji liczby wykonanych zadań w jak najkrótszym czasie; odpowiada do minimalizacji **makespan** C_{max}, a więc maksymalnego czasu ukończenia ostatniego ze skończonej liczby zadań,

minimalizacja opóźnienia (ang. lateness) $L_{max} = \max(c_j - d_j, \ldots, c_n - d_n)$ czyli różnicy miedzy czasem zakończenia,

minimalizacja opieszałości (ang. tardiness) $T_{max} = \sum_j \max(c_j - d_j, 0)$ czyli łącznego opóźnienia dla zadań; często ważona przez priorytet zadania w_j,

minimalizacja kosztów przezbrojenia związanych z rozpoczynaniem produkcji, na przykład na maszynie, która zużywa sporo materiałów przed rozpoczęciem pracy,

minimalizacja kosztów przechowywania ukończonych produktów, co daje w efekcie produkcję typu *just-in-time*,

minimalizacja kosztów transportu materiałów, produktów, etc.

1.2.5 Notacja dla najważniejszych pojęć

O_k	*zlecenie* (ang. order) składane do zakładu przemysłowego opisane przez zbiór zadań
J_j	*zadanie* (ang. job) opisuje wykonanie pojedynczego lub partii produktów
t_{ij}	*operacja* (ang. task) pojedyncze przetwarzanie zadania j na jednym zasobie R_i
R_i	*zasób* (ang. resource) maszyn (wtedy często M_i), personelu, zasobów
p_{ij}	czas przetwarzania (processing) operacji j na maszynie i; w systemach probabilistycznych czas jest dany przez podanie rozkładu, zwykle jako para $(\mu_{ij}, \sigma_{ij}^2)$ średniego czasu wykonania wraz z wariancją [BW07]
r_j	czas rozpoczęcia (release) zadania j
d_j	czas obiecanego zakończenia (due) zadania j
w_j	priorytet (priority) zadania j
s_{ij}	czas rozpoczęcia (start) zadania j na maszynie i
s_j	czas rozpoczęcia (start) zadania j
c_{ij}	czas zakończenia (completion) zadania j na maszynie i
c_j	czas zakończenia (completion) zadania j
$C_{max} = \max_j c_j$	czas ukończenia ostatniego ze skończonej liczby n zadań
L_{max}	opóźnienie (ang. lateness)
T	opieszałość (ang. tardiness)

1.2.6 Reprezentacja problemu gniazdowego

Do reprezentacji problemu gniazdowego można wykorzystać strukturę grafową typu "czynność na węźle" AON (ang. activity on node). Zbiór V zawiera węzły reprezentujące opercje [Kli16], zbiór E zawiera łuki opisujące relacje kolejnościowe między operacjami typu koniec-początek bez zwłoki (ang. finish-start zero-lag precedence). Operacja $i \in V$ musi być realizowana na maszynie $m_i \in M$. Maszyna m_i może wykonywać co najwyżej k_i operacji w danej chwili czasu. Operacja i ma czas trwania d_i. Minimalizacja czasu trwania przedsięwzięcia $C_{\max}$ (makespan), równa czasowi dotarcia do wierzchołka końcowego. Rozwiązaniem problemu harmonogramowania projektu z ograniczonymi zasobami z kryterium minimalizacji czasu trwania prac jest wektor czasów rozpoczęcia lub zakończenia czynności tzw. reprezentacja bezpośrednia.

Do znalezienia rozwiązania można wykorzystać reprezentacje pośrednie:

- **Lista czynności.** Reprezentacja w postaci listy czynności (ang. activity list), w której rozwiązanie jest permutacją numerów kolejnych czynności przy uwzględnieniu zależności kolejnościowych. Rozwiązania w reprezentacji pośredniej muszą być przekształcane do reprezentacji bezpośredniej przy użyciu procedur dekodujących. Opracowywane są schematy generowania uszeregowania SGS (ang. Schedule Generation Scheme), które dekodują listę czynności w realizowalne harmonogramy (ustalane są wektory czasów rozpoczęcia zadań) przy uwzględnieniu ograniczeń czasowych i zasobowych.

  Do procedur SGS zaliczyć można [Kol96]:

  - procedurę szeregową (ang. serial SGS), w której w kolejnych chwilach t ustalany jest czas rozpoczęcia dla pierwszego nieuszeregowanego zadania z listy czynności, w najwcześniejszym możliwym czasie przy uwzględnieniu ograniczeń kolejnościowych i zasobowych,

– procedurę równoległą (ang. parallel SGS), w której w kolejnych chwilach t rozpoczynane są wszystkie nieuszeregowane zadania (rozpatrywane w kolejności na liście czynności), które mogą być rozpoczęte w chwili t przy uwzględnieniu ograniczeń kolejnościowych i zasobowych.

Przy dekodowaniu listy stosowane mogą być różne techniki harmonogramowania:

– w przód (ang. forward scheduling) – szeregowanie kolejnych czynności od początku listy czynności,

– wstecz (ang. backward scheduling) – szeregowanie kolejnych zadań od końca listy czynności, ustalając czasy rozpoczęcia zadań przy przyjętym terminie zakończenia projektu (due date).

- **Permutacja zadań.** Innym sposobem jest kodowanie operacji za pomocą permutacji z powtórzeniami. Załóżmy dla przykładu (patrz Rysunek 1.1), że mamy spermutowane operacje $(t_{21}, t_{31}, t_{22}, t_{11}, t_{23}, t_{12}, t_{13}, t_{23})$. Z tej permutacji tworzymy osobnika zastępując identyfikator operacji przez numer zadania, czyli dostajemy $(2, 3, 2, 1, 3, 1, 1, 2)$. To jest kluczowe, gdyż okazuej się, że każdy taki ciąg tworzy prawidłowe rozwiązanie.

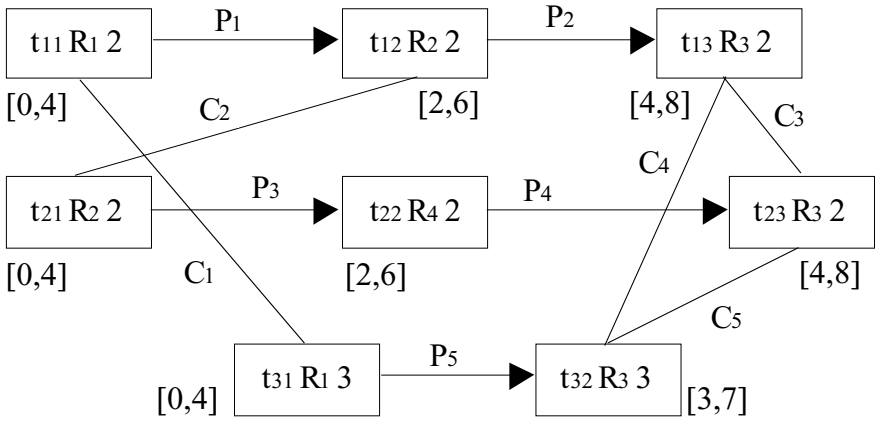


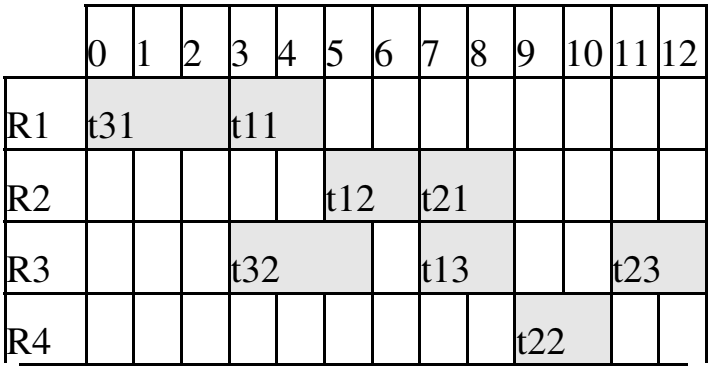
Figure 8.1 A JSS problem instance with three jobs. The release dates have value 0 for every task and the due dates have value 10. The duration of the tasks is indicated within the boxes, together with the task identification and the resource requirement

Figure 8.1 depicts an example with three jobs *{J1,J2,J3}* and four physical resources *{R1,R2,R3,R4}*. It is assumed that the tasks of the first two jobs have duration of two time units, whereas the tasks of the third one have duration of three time units. The release time is 0 and the due date is 10. Label *Pi* represents a precedence constraint and label *Cj* represents a capacity constraint. Start time values constrained by the release and due dates and the duration time of tasks are represented as intervals. For instance, [0,4] represents all start times between time 0 and time 4, as allowed by the time granularity, namely {0,1,2,3,4}.

Rysunek 1.1: źródło: [Cha00, Figure 8.1]

Do odzyskania rozwiązania potrzebujemy funkcji dekodującej, która z takiego ciągu nam tworzy odpowiedni harmonogram. Sa dwie strategie (opisuję domyślną która ma mniejszą złożoność, opis obu jest w [Cha00, Rozdział 8]). Ciąg operacji jest ponumerowany zgodnie z kolejnością w której występuje w danym osobniku. Dla każdej operacji jest przyporządkowywany czas startu zgodnie z poprzednimi operacjami, a mianowicie: maksimum z czasów zakończenia poprzednich operacji które dzielą wspólne ograniczenie (precednecy lub capacity).

Dla zobaczenia wygodnie jest zobaczyć następujący przykład: rozpatrzmy osobnika (33111222). Wtedy on tworzy nam $(t_{31}t_{32}t_{11}t_{12}t_{13}t_{21}t_{22}t_{23})$. Musimy stworzyć dopuszczalny harmonogram (czyli wyliczyć odpowiednie czasy), patrz obrazek:



Rysunek 1.2: [Cha00, Figure 8.2 a)]

Dalsze szczegóły patrz [Cha00, Rozdział 8].

1.2.7 Obserwacje i wnioski

Problemy projektowania harmonogramów są bardzo złożone. Poza najprostszymi przypadkami, które zupełnie nie są praktyczne w rzeczywistych aplikacjach ze względu na ograniczenia liczby zadań i zasobów do kilku (!), algorytmy nie są do wykonania w czasie wielomianowym, a wymagają pełnego przeszukiwania przestrzeni rozwiązań.

Dodatkowo zadania nie są zwykle statyczne, ponieważ nie można określić np. czasów przetwarzania dokładnie, a w rzeczywistych procesach produkcyjnych sytuacja może zmienić się nagle. Funkcja kosztu ma zwykle wiele celów. Klasyczna jest sytuacja, w której należy zoptymalizować zarówno czas przetworzenia wszystkich zamówień, a jednocześnie zminimalizować koszty przetwarzania, dowozu materiałów, eksploatacji maszyn, itd. Nie można tych zadań rozwiązywać osobno otrzymując zwykle sprzeczne rozwiązania, a jednocześnie. A dzieje się to kosztem otrzymania zadania, które ma wiele minimów lokalnych. Metody minimalizacji dyskretnej są trudne.

Z tego względu regułą jest dzielenie problemów na prostsze, np. pełnego zbioru zamówień na zadania wykonywane na jednej maszynie i rozwiązywanie ich osobno (np. z problemu gniazdowego do wielu problemów pojedynczej maszyny czy problemów przepływowych).

W tym celu należy wykorzystać podejścia heurystyczne oraz podejscia związane z metodami uczenia maszynowego. Nie należy zapominać, że dla wielu prostszych zadań, wykonywa-

nych po ewentualnym podziale, można wykorzystać znane już rozwiązania dla których znane są ograniczenia i zapewnione warunki osiągnięcia rozwiązań optymalnych.

1.2.8 Algorytm ewolucyjny (GA=genetic algorithm): problem gniazdowy

Opisujemy w skrócie klasyczne podejście, Rysunek 1.1. Kodowanie osobników jest stworzone za pomocą **permutacji z powtórzeniami** (patrz Sekcja 1.2.6). Załóżmy dla przykładu (patrz [Cha00, Figure 8.1]), że mamy spermutowane taski $(t_{21}, t_{31}, t_{22}, t_{11}, t_{23}, t_{12}, t_{13}, t_{23})$. Z tej permutacji tworzymy osobnika zastępując identyfikator operacji przez numer pracy/zadania, czyli dostajemy $(2, 3, 2, 1, 3, 1, 1, 2)$.

Operator mutacji polega na tym, że dwie operacje są losowo wybierane, i ich pozycje są wymieniane. Operatory krzyżowania (cross-over) są zazwyczaj stosowane dwa:

GOX *generalized order crossover:* załóżmy, że mamy rodziców (parents) $P1 = (12\underline{21}21)$ oraz $P2 = (1\underline{1}2\underline{2}12)$. Z pierwszego wybieramy dowolny podciąg (dla przykładu druga dwójka i pierwsza jedynka), i usuwamy go z $P2$ (odpowiednie podkreślenia). Następnie wkładamy ten podciąg (w jednym ciągu, czyli w całości) na pozycji gdzie zaczynał się w pierwszym rodzicu.

GPX *generalized position crossover:* analogicznie, ale wkładamy dokładnie tam, gdzie był w $P1$ [mi się bardziej podoba, bo wtedy dziecko identycznych rodziców jest ich kopią].

Jako funkcję adaptacji (fitness function) bierzemy całkowity makespan (czyli czas do kiedy się skończy ostanie zadanie).

Dalsze szczegóły patrz [Cha00, Rozdział 8]. Na podstawie literatury, w szczególności [Cha00, Rozdział 8], wynika, że GA ma sens tylko i wyłącznie w połączeniu z innymi metodami (symulowane wyżarzanie, shifting bottleneck).

1.2.9 GA: uogólniony problem gniazdowy

W naszym przypadku nie zakładamy linearności operacji (tasks) wewnątrz zadań (jobs). W konsekwencji każde zadanie jest reprezentowany jako spójny etykietowany graf skierowany (bez cykli). Ma specjalną strukturę - ma jeden wierzchołek [? ostateczny potomek], taki, że poruszając się zgodnie z kierunkiem krawędzi do niego zawsze trafiamy [? czy to jedyny warunek].

Wierzchołki opowiadaja zadaniom, krawędź z wierzchołka e do f oznacza, że zadanie odpowiadające wierzchołkowi f może byc wykonywane dopiero po wykonaniu e. W każdym wierzchołku mamy etykiety – wersja minimum to czas wykonywania+maszyna, ale w wersji rozszerzonej może być jeszcze na przykład więcej maszyn, pracownik który to zadanie wykonuje, może jakiś koszt, etc. Wierzchołki ktore nie mają rodziców nazywam *początkowymi*.

GENOM. Pokażemy, że każdy harmonogram jest jednoznacznie opisany przez podanie w każdym wierzchołku jakiejś liczby rzeczywistej (zakładamy, że te liczby są parami różne).

DEKODOWANIE. Opiszę sposób dekodowania. Bierzemy z wszystkich zadań (rozumianych jako grafy) wierzchołki początkowe (pamiętamy wskaźniki do elementu). Sortujemy względem liczby rzeczywistej opisującej. Bierzemy pierwszy element, znajdujemy w grafie jego wszystkich potomków, następnie go z grafu usuwamy. Spośród potomków bierzemy te które są w tym nowym grafie elementami początkowymi, i dosortowujemy je do naszego ciągu.

Mamy (podobnie jak w klasycznym podejściu), dwie opcje:

- (prostsze w implementacji) pamiętamy kiedy wszystkie maszyny są już wolne (skończyły pracę), i wtedy dokładamy czas tego zadania do maszyny na której ma być wykonywane, oczywiście po maks z czasów wszystkich rodziców (musimy oczywiście wpisywać kiedy zaczynamy i kiedy kończymy każdą operację); nie wymaga żadanego przeszukiwania

- (trudniejsze, ale się stosuje) rozważamy pełne wypełnienie każdej maszyny, i ustawiamy na pierwszym/największym z dopuszczalnych miejsc gdzie się zmieści (tu jest trochę koszt, że musimy tą przestrzeń dopuszczalnych przedziałów jakoś przeszukiwać. najprościej chyba sortujemy względem czasu rozpoczęcia i patrzymy kolejno kiedy znajdziemy odpowiednią).

Teraz jak mamy takie DNA, to możemy stosować jakieś podstawowe metody optymalizacji, oraz jakieś preferencje/heurezy (jak należy ustawiać wartości dla jednej maszyny, etc, między maszynami, etc).

1.3 Poszukiwanie rojem cząstek

1.3.1 Wprowadzenie

Przeszukiwanie rojowe (Particle Swarm Optimization, PSO) [Ken95] jest algorytmem optymalizacyjnym inspirowane ruchem roju (swarm) osobników np. stada ptaków, które poszukują pożywienia. Pozycja cząstki to jakieś rozwiązanie dopuszczalne. Cząstki zmieniają swoje pozycje (przeszukują przestrzeń rozwiązań) tak aby finalnie osiągnąć pozycję o najlepszym rozwiązaniu.

Algorytm ma charakter iteracyjny, kończy się po określonej liczbie iteracji bądź osiągnięciu satysfakcjonującego rozwiązania. Na początku każda cząstka ma losową pozycję i losowy wektor prędkości. W kolejnych iteracjach cząstka podąża zgodnie z wektorem prędości zależnym od:

- poprzedniego przesunięcia

- przesunięcia kierującego go do najlepszego rozwiązania dotychczas znalezionego przez tą cząstkę

- przesunięcia kierującego go do najlepszego rozwiązania dotychczas znalezionego rozwiązania przez cały rój.

PSO jest zwykle stosowany do optymalizacji ciągłych nieliniowych problemów, jako że pozycja cząstki jest dowolnym wektorem w rzeczywistej przestrzeni wektorowej.

1.3.2 Stan wiedzy

PSO jest częstym algorytmem wykorzystywanym w szeregowaniu zadań [RK06].

Typowy sposób zastosowania PSO polega na transformacji permutacji operacji do postaci wektorowej, gdzie często używa się reprezentacji za pomocą losowego klucza (random key representation) [Bea94, Bie95] bądź algorytmu przedstawionego w [GT60]. W zastosowaniu PSO do problemu harmonogramowania może się zdarzyć, że bliskie położenia cząstek będą dawać znacząco różne harmonogramy. Aby temu zapobiec dokonuje się modyfikacji zarówno

w reprezentacji opsiu oraz aktualizacji położenia cząstki [SH06]. Ciekawe podejście redefiniujące podobieństwo cząstek i wektora prędkości zostało przedstawione w [GSLQ08].

Różnica w realizacji PSO tkwi często w ilości generowania nowych cząstek do przeszukiwania rozwiązań [PK09, PK11]. Algorytmy PSO są często hybrydyzowane i łączone z symulowanym wyżarzaniem [GDQ07], algorytmami immunologicznymi [GSLQ08], bądź lokalnym przeszukiwaniem [LHK+10].

1.3.3 Algorytm rojowy: problem gniazdowy

Rozważamy ogólny problem gniazdowy (patrz 1.2.2). Mamy zestaw n zadań $J_1, \ldots, J_n$ i m maszyn $M_1, \ldots, M_m$. Praca składaja się z operacji z ograniczeniami poprzedzania. Każda operacja musi być wykonywana na określonej maszynie (dokładnie tak jak w opisie poprzednich algorytmów). Chcemy znaleźć rozwiązania minimalizujące makespan.

Zbiór operacji możemy przedstawić w postaci macierzy $T = (t_{ij})_{ij} \in \mathbb{R}^{m \times n}$. Macierz T można zapisać w postaci wektorowej, spisując ją kolumnowo. Wówczas wszystkie operacje zadania j będą na tych pozycjach k wektora które spełniają

$$j = k \bmod n$$

licząc pozycje wektora od zera tzn. $j, j+n, j+2n$, itd..

Będziemy chcieli stowrzyć permutację wektora operacji, która to wyznaczy nam kolejność operacji (stosujemy reprezentacje za pomocą **permutacji operacji**). Dokładniej, przykładową permutację operacji $(o_{12}, o_{13}, o_{22}, o_{11}, o_{32}, o_{21}, o_{31}, o_{32})$ możemy przekształcić zastępując identyfikator operacji przez numer zadania, czyli dostajemy $(2, 3, 2, 1, 3, 1, 1, 2)$. Każdy taki wektor (permutacji zadań) definiuje prawidłowe rozwiązanie (jako pierwsza będzie rozplanowana pierwsza operacja zadania 2, jako druga będzie pierwsza operacja zadania 3, itd.).

Permutacje operacji tworzymy przypisując operacji pewną liczbę rzeczywistą oznaczającą priorytet (operacja z niższym priorytetem ma szanse być rozplanowana wcześniej). Dla przykładu weźmy 3 zadania, każde wykonywane na 2 maszynach, co daje macierz $T \in \mathbb{R}^{2\times 3}$. Wówczas przykładowy wektor priorytetów (pierwszy wiersz) wyznacza kolejność wykonań operacji (drugi wiersz), patrz Rysunek 1.3.3. Jako że ograniczenia kolejnościowe mogą uniemożliwić wykonanie dokładnie takiej sekwencji operacji w ramach danego zadania, to operacje zastępujemy numerami zadań (trzeci wiersz). Ostatecznie, dla każdego zadania z powstałego wektora bierzemy jego pierwszą operację która nie powoduje złamania ograniczeń (4 wiersz).

Położenie cząstki opisane jest wektorem priorytetów – wektor priorytetów dekoduje się do harmonogramu jak pokazano powyżej. Zatem dla danego wektora priorytetów można policzyć makespan. PSO dąży do takiej modyfikacji wektora priorytetów, aby minimalozować makespan.

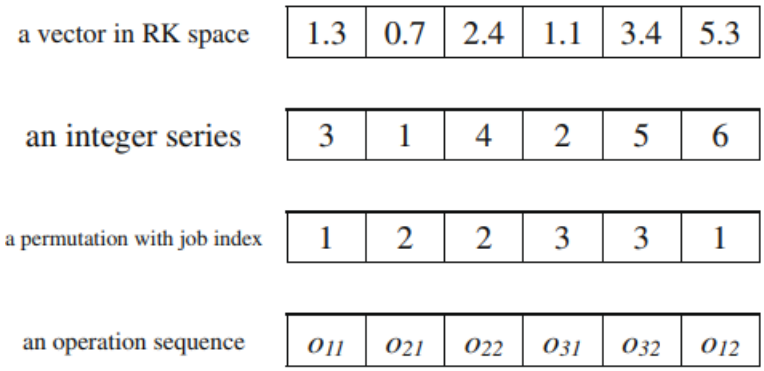
Przyjmijmy, że x_i oznacza położenie cząstki i, a v_i jej wektor prędkości. Zmiana położenia odbywa się za pomocą następującego równiania:

$$x_i = x_i + v_i,$$

natomiast wektor prędkości zmienia się następująco:

$$v_i = \omega v_i + c_1 r_1 (x_i^{best} - x_i) + c_2 r_2 (x^{best} - x_i),$$

gdzie x_i^{best} to najlepsze położenie (rozwiązanie) znalezione dotychczas przez cząstkę i, a x^{best} to najlepsze położenie znalezione przez cały rój. Współczynniki ω, c_1, c_2 są predefiniowane, a



Rysunek 1.3: [LHK+10] (należy zamienić indeksy operacji z indeksami maszyn, bo notacja w źródle była inna)

$r_1, r_2 \in (0, 1)$ są wybierane losowo w każdym kroku. Dodatkowo można ograniczyć możliwe wartości wektora prędkości v_i.

Algorytm PSO ma postać dla przeszukiwania rozwiązania w N wymiarowej przestrzeni ma postać:

Inicjalizacja:
Wybierz pozycje i wektory prędkości N cząstek
Wyznacz wartości rozwiązań cząstek
Ustaw x^{best} oraz x_i^{best} dla każdej cząstki
Procedura:
repeat
  for $i = 1, \ldots, N$ **do**
    Wyznacz położenie x_i cząstki i oraz wartość jej rozwiązania
    Zaktualizuj x_i^{best} oraz x^{best}
  end for
until wymagana liczba iteracji nie została osiągnięta (bądź nie nastąpił inny warunek stopu)
Zwróć:
Harmonogram S^* odpowiadający x^*

1.3.4 Modyfikacje / Parametryzacja

W celu osiągnięcia lepszego rozwiązania parametry ω, c_1, c_2 mogą zmieniać się wraz z czasem. Pokazano, że lepsze rozwiązanie można osiągnąć obniżając wartości ω liniowo [KHK+09, XW06]:

$$\omega = \omega_{\max} - Iter\frac{\omega_{\max} - \omega_{\min}}{MaxIter}$$

gdzie $MaxIter$ to maksymalna ilość iteracji, $Iter$ to aktualna ilość, $\omega_{\max}$ to początkowa wartość dla ω, a $\omega_{\min}$ to końcowa wartość.

Kodowanie permutacji za pomocą wektora może powodować że dwa bliskie wektory dają bardzo różne harmonogramy. Aby temu zapobiec redefiniuje się podobieństwo i wektor prędkości w kontekście harmonogramów [GSLQ08].

Uwaga 1.2. Ze względu na prostotę implementacji dla problemu harmonogramowania (podobnie jak algorytmu genetyczne), PSO jest często używany. Jednak najlepsze efekty daje w połączeniu z innymi metodami np. metodami lokalnych przeszukiwań takich jak symulowane wyżarzanie [LHK+10].

Bibliografia

[Bea94] James C Bean. Genetic algorithms and random keys for sequencing and optimization. *ORSA journal on computing*, 6(2):154–160, 1994.

[Bie95] Christian Bierwirth. A generalized permutation approach to job shop scheduling with genetic algorithms. *Operations-Research-Spektrum*, 17(2-3):87–92, 1995.

[BW07] J. Ch. Beck and N. Wilson. Proactive algorithms for job shop scheduling with probabilistic durations. *Journal of Artificial Intelligence Research*, 28:183–232, 2007.

[Cha00] Lance D Chambers. *The practical handbook of genetic algorithms: applications*. Chapman and Hall/CRC, 2000.

[GDQ07] Hongwei Ge, Wenli Du, and Feng Qian. A hybrid algorithm based on particle swarm optimization and simulated annealing for job shop scheduling. In *Natural Computation, 2007. ICNC 2007. Third International Conference on*, volume 3, pages 715–719. IEEE, 2007.

[GSLQ08] Hong-Wei Ge, Liang Sun, Yan-Chun Liang, and Feng Qian. An effective pso and ais-based hybrid intelligent algorithm for job-shop scheduling. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 38(2):358–368, 2008.

[GT60] Bernard Giffler and Gerald Luther Thompson. Algorithms for solving production-scheduling problems. *Operations research*, 8(4):487–503, 1960.

[Ken95] R Kennedy. J. and eberhart, particle swarm optimization. In *Proceedings of IEEE International Conference on Neural Networks IV, pages*, volume 1000, 1995.

[KHK+09] I-Hong Kuo, Shi-Jinn Horng, Tzong-Wann Kao, Tsung-Lieh Lin, Cheng-Ling Lee, Takao Terano, and Yi Pan. An efficient flow-shop scheduling algorithm based on a hybrid particle swarm optimization model. *Expert systems with applications*, 36(3):7027–7032, 2009.

[Kli16] Marcin Klimek. Symulowane wyżarzanie dla problemu harmonogramowania projektu z ograniczonymi zasobami. *Zeszyty Naukowe Warszawskiej Wyższej Szkoły Informatyki*, 2016.

[Kol96] Rainer Kolisch. Serial and parallel resource-constrained project scheduling methods revisited: Theory and computation. *European Journal of Operational Research*, 90(2):320–333, 1996.

[LHK+10] Tsung-Lieh Lin, Shi-Jinn Horng, Tzong-Wann Kao, Yuan-Hsin Chen, Ray-Shine Run, Rong-Jian Chen, Jui-Lin Lai, and I-Hong Kuo. An efficient job-shop scheduling algorithm based on particle swarm optimization. *Expert Systems with Applications*, 37(3):2629–2636, 2010.

[Pin00] Michael Pinedo. *Planning and scheduling in Manufacturing and services*. Springer, 2 edition, 2000.

[PK09] Pisut Pongchairerks and Voratas Kachitvichyanukul. A two-level particle swarm optimisation algorithm on job-shop scheduling problems. *International Journal of Operational Research*, 4(4):390–411, 2009.

[PK11] Thongchai Pratchayaborirak and Voratas Kachitvichyanukul. A two-stage pso algorithm for job shop scheduling problem. *International Journal of Management Science and Engineering Management*, 6(2):83–92, 2011.

[RK06] L Rookapibal and V Kachitvichyanukul. Particle swarm optimization for job shop scheduling. In *Proceedings of the international computers and industrial engineering conference*, 2006.

[SH06] DY Sha and Cheng-Yu Hsu. A hybrid particle swarm optimization for job shop scheduling problem. *Computers & Industrial Engineering*, 51(4):791–808, 2006.

[Smu12] Czesław Smutnicki. *Algorytmy szeregowania zadań*. Oficyna Wydawnicza Politechniki Wrocławskiej, 2012.

[XW06] Wei-jun Xia and Zhi-ming Wu. A hybrid particle swarm optimization approach for the job-shop scheduling problem. *The International Journal of Advanced Manufacturing Technology*, 29(3-4):360–366, 2006.