Projektowanie efektywnych algorytmów

Szeregowanie zadań

Branch and Bound

# Wstęp

W tym etapie projektu podjąłem się próby implementacji problemu jednomaszynowego szeregowania zadań przy kryterium minimalizacji ważonej sumy opóźnień zadań.

Całość została wykonana w języku C++ w standardzie C++14 z użyciem Visual Studio 2015 i Visual Compiler 8.1.

## Podstawowy algorytm

Metoda Podziału i Ograniczeń (Branch and Bound) polega na budowaniu rozwiązania rozszerzając rozwiązania częściowe o kolejne elementy rozwiązania, dla problemu jednomaszynowego szeregowania zadań oznacza to dokładanie kolejnych zadań. Ponieważ dla n zdań w drzewie rozwiązań istnieje węzłów(licząc pusty korzeń), rozsądnym jest liczenie funkcji celu również inkrementacyjnie, a nie dla każdego węzła niezależnie.

## Problem jednomaszynowego szeregowania zadań przy kryterium minimalizacji ważonej sumy opóźnień zadań.

Instancja problemu składa się ze zbioru zadań posiadających następujące parametry: czas wykonywania(processing time), wagę(weight) i termin wykonania(deadline). Jeżeli zadanie zostanie skończone po terminie wykonania, czas o jaki został przekroczony termin jest mnożony razy wagę i zapamiętywany jako tardiness(opieszałość).

# Implementacja algorytmu

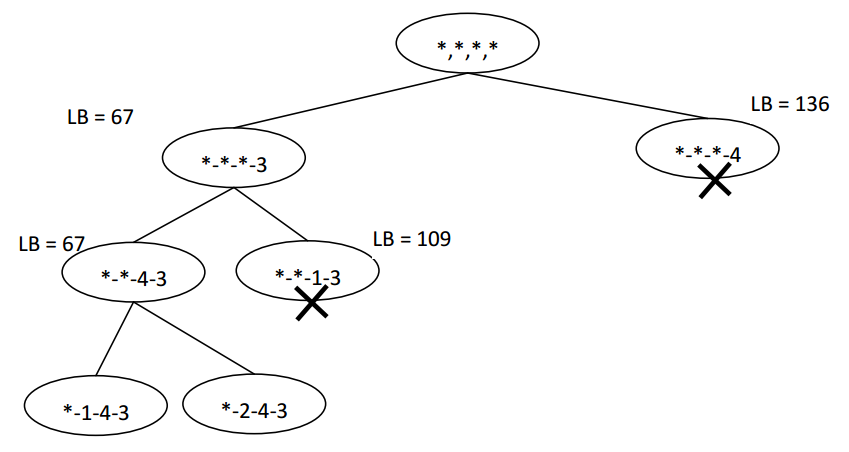
## Branch And Bound

Pseudokod:

1. Utwórz pusty zbiór rozwiązań
2. Dla każdego dostępnego zadania
   1. Policz różnicę w funkcji celu po dodaniu zadania, jeśli nie osiągniesz górnego ograniczenia, rozwiń rozwiązanie o aktualnie rozpatrywane zadanie
   2. Jeśli rozwiązanie jest pełne, zapamiętaj je jako najlepsze i zapamiętaj aktualną wartość funkcji celu jako górne ograniczenie. W przeciwnym razie, dla aktualnego zbioru rozwiązań wykonaj 2

## Strategia rozwijania rozwiązań

Pierwszym co przychodzi na myśl, jest budowanie drzewa rozwiązań przez dokładanie ich od początku, jednak biorąc pod uwagę specyfikę problemu, dla większości przypadków cała opieszałość pochodzi z zadań wykonanych jako ostatnie. Żeby odwiedzić jak najmniej węzłów, powinnyśmy możliwie szybko obcinać gałęzie, a co za tym idzie, zwiększać funkcję celu na wyższych poziomach drzewa. Aby to osiągnąć wystarczy dokładać zadanie „od końca”(patrz rysunek)



## Estymowanie górnego ograniczenia

Żeby skrócić czas działania algorytmu, wyznaczam górne ogranicznie algorytmem aproksymującym działającego według następującego schematu

Dla n zdań:

1. Posortuj zadania według terminu wykonania(deadline), umieść je w zbiorze zadań dostępnych
2. Utwórz n rozwiązań, dla każdego wybierz inne zadanie ze zbioru zadań dostępnych i umieść na początku zbioru zadań wykonanych
3. Z istniejących rozwiązań utwórz zbiór rozwiązań o najniższej wartości funkcji celu
4. Dla każdego zadania w zbiorze utwórz rozgałęzienia w liczbie zadań dostępnych dla obecnego poziomu drzewa.
5. Jeśli rozwiązania w obecnym poziomie drzewa w zbiorze zadań zamkniętych mają n zadań, zakończ. W przeciwnym razie, wróć do punktu trzeciego.

# Podejście pierwsze – przeszukiwanie drzewa wszerz układając od końca

Pierwotnie moim pomysłem było napisanie algorytmu działającego według następującego schematu:

Rozwiązania (częściowe) składają się z części otwartej (która będzie modyfikowana dla kolejnych rozgałęzień) – „zadania dostępne” i części zamkniętej (niezmiennej dla następnych rozgałęzień) – „zadania wykonane”. Podczas liczenia funkcji celu, zbiór zadań dostępnych znajduje się przed zbiorem zadań wykonanych, tj. algorytm układa zadania do wykonania od końca osi czasu.

### Pseudokod:

Ponieważ ważona suma opóźnień (dalej będę używać określenia „tardiness”) jest najbardziej uzależniona od zadań wykonanych jako ostatnie, powyższy algorytm początkowo odcina wszystkie gałęzie oprócz jednej, jednak po kilku piętrach drzewo rozwiązań gwałtownie rozrasta się. Dla dużych instancji algorytm zużywa mnóstwo pamięci i nie udało się nim policzyć problemu dla 40 zadań, ale po kilku poziomach drzewa wylicza lower bound bardzo bliski optymalnemu rozwiązaniu(930 gdy w optymalnym rozwiązaniu tardiness to 917). Dla małych, czterozadaniowej działa bez zarzutu.

Szczegółowa implementacja do obejrzenia w klasie BBScheduler na branchy <https://github.com/greeboCherry/PEA_scheduling/tree/BFS>

# Podejście drugie – przeszukiwanie drzewa w głąb układając od początku

Po konsultacjach z Prowadzącym spróbowałem podejścia rekurencyjnego, wybierania zadań od początku osi czasu, i liczeniu funkcji celu wraz z doklejaniem kolejnych zadań. Do wyliczenia lower bound użyłem algorytmu z punktu pierwszego zatrzymującego się po pierwszym poziomie drzewa po którym nie spadła wartość funkcji celu.

Ponieważ pierwsze kilka zadań potrafi nie generować żadnego tardiness, algorytm chociaż nie dochodził do prawie żadnego liścia, obcinał gałęzie bardzo nisko, będąc wcale niewiele lepszym od przeglądu zupełnego, ze względu na dużą ilość instrukcji warunkowych w każdym kroku, instrukcje mogły nie być idealnie potokowane przez procesor. Ponadto, ponieważ nie mam dużego doświadczenia z funkcjami rekurencyjnymi, nie wiem dokładnie jak je optymalizować ani jak optymalizuje je kompilator

Szczegółowa implementacja w klasie BBDFSScheduler w załączonym projekcie/na masterze podanego wcześniej repozytorium.

# Podejście trzecie – połączeniu obu metod

Zauważyłem że algorytm z punktu trzeciego często schodzi tylko do jednego liścia, ale odwiedza po drodze mnóstwo węzłów. Ostatecznie wartość funkcji celu pozostaje taka jaką oszacował dla niej algorytm z punktu pierwszego. Ostatecznie moim rozwiązaniem jest:

* 1. Przeszukując drzewo wszerz znaleźć końcową część rozwiązania. Algorytm zatrzymuje się po określonej ilości poziomów których pokonanie nie obniżyło lower bound’a.
  2. Z pośród dostępnych rozwiązań częściowych o najniższej wartości funkcji celu wybierz jedno. Jego zbiór otwarty przekaż algorytmowi rekurencyjnemu jako problem do rozwiązania, wraz z wyliczoną wartości funkcji celu dla tego zbioru. Zbiór zamknięty zapamiętaj jako druga część rozwiązania
  3. Przeszukując drzewo w głąb rozwiąż otrzymany problem
  4. Złącz rozwiązania wyliczone przez oba algorytmy

Ponieważ dwa bazowe algorytmy pisałem z różnym podejściem (wskaźniki przeciwko indeksom) ich złączenie wymagało poświęcenia dużo czasu na integracje, ale efekt końcowy mnie satysfakcjonuje.

# Testowanie

Wszystkie algorytmy poprawnie policzyły następującą instancję problemu:

Task 1: Time: 12, Weight: 4, Deadline: 16

Task 2: Time: 8, Weight: 5, Deadline: 26

Task 3: Time: 9, Weight: 5, Deadline: 27

Task 4: Time: 15, Weight: 3, Deadline: 25

Czasy wykonania:

Algorytm przeszukujący drzewo w głąb: 1.12743 ms + 370.713 ms na wyliczenie upper bound

Algorytm przeszukujący drzewo wszerz: 444.351 ms

Algorytm hybrydowy: 569.385 ms + 53.0492 ms

Instancję:   
Task 1: Time: 9, Weight: 10, Deadline: 1446

Task 2: Time: 70, Weight: 5, Deadline: 1481

Task 3: Time: 90, Weight: 4, Deadline: 1484

Task 4: Time: 35, Weight: 4, Deadline: 1487

Task 5: Time: 68, Weight: 7, Deadline: 1497

Task 6: Time: 95, Weight: 5, Deadline: 1509

Task 7: Time: 52, Weight: 3, Deadline: 1510

Task 8: Time: 36, Weight: 5, Deadline: 1512

Task 9: Time: 85, Weight: 9, Deadline: 1522

Task 10: Time: 12, Weight: 5, Deadline: 1528

Task 11: Time: 46, Weight: 3, Deadline: 1539

Task 12: Time: 30, Weight: 7, Deadline: 1541

Task 13: Time: 55, Weight: 7, Deadline: 1559

Task 14: Time: 73, Weight: 3, Deadline: 1566

Task 15: Time: 64, Weight: 7, Deadline: 1582

Task 16: Time: 26, Weight: 1, Deadline: 1588

Task 17: Time: 14, Weight: 2, Deadline: 1598

Task 18: Time: 86, Weight: 7, Deadline: 1599

Task 19: Time: 24, Weight: 10, Deadline: 1620

Task 20: Time: 39, Weight: 9, Deadline: 1627

Task 21: Time: 86, Weight: 4, Deadline: 1628

Task 22: Time: 67, Weight: 3, Deadline: 1636

Task 23: Time: 40, Weight: 3, Deadline: 1645

Task 24: Time: 44, Weight: 7, Deadline: 1650

Task 25: Time: 78, Weight: 8, Deadline: 1658

Task 26: Time: 94, Weight: 4, Deadline: 1660

Task 27: Time: 32, Weight: 10, Deadline: 1694

Task 28: Time: 29, Weight: 10, Deadline: 1709

Task 29: Time: 14, Weight: 10, Deadline: 1727

Task 30: Time: 79, Weight: 9, Deadline: 1731

Task 31: Time: 35, Weight: 5, Deadline: 1772

Task 32: Time: 46, Weight: 10, Deadline: 1773

Task 33: Time: 69, Weight: 4, Deadline: 1795

Task 34: Time: 50, Weight: 3, Deadline: 1814

Task 35: Time: 37, Weight: 10, Deadline: 1826

Task 36: Time: 27, Weight: 7, Deadline: 1836

Task 37: Time: 28, Weight: 4, Deadline: 1833

Task 38: Time: 74, Weight: 2, Deadline: 1844

Task 39: Time: 49, Weight: 1, Deadline: 1579

Task 40: Time: 78, Weight: 1, Deadline: 1855

Algorytm hybrydowy rozwiązał w 775.563 ms + 2289.93 ms.  
Algorytm przeszukujący drzewo w głąb potrzebował na to 320 minut.  
Algorytm przeszukujący drzewo wszerz potrzebował alokować zbyt wiele pamięci.

Otrzymana wartość funkcji celu różni się od optymalnej podanej przez OR-LIB.

# Konkluzje

Ten projekt nauczył mnie że:

1. Mieszanie nowoczesnego c++ z raw pointerami i indeksami wydłuża czas tworzenia programu bardziej niż Test Driven Developemnt.
2. Brakuje mi dyscypliny do TDD.
3. Nie pisanie w TDD = nie pisanie testów jednostkowych w ogóle.

A o samym zadaniu

1. Najwięcej (a w niektórych przypadkach cały) tardiness generują zadania na końcu kolejki.
2. Metoda podziału i ograniczeń liczona od przodu jest zwyczajnie nieefektywna bo obcięcia są zbyt nisko.
3. Większa wariancja terminów realizacji wypływa pozytywnie na jakość zwykłego BnB - odcięcia następują szybciej, funkcja rekurencyjna wywołuje się rzadziej.