Dokumentacja końcowa
AAL - Analiza algorytmów

Treść:

Dany jest zbiór sal wykładowych oraz zbiór zamówień, określających czas rozpoczęcia i zakończenia wykładu. Ułożyć plan wykorzystania sal, akceptując pewne wykłady i odrzucając inne, tak aby sumaryczny czas wykorzystania sal był jak najdłuższy. Porównać czas obliczeń i wyniki różnych metod.

Struktury danych:

Używane struktury ze standardowej biblioteki:

- std::vector
- std::list
- std::map

Własne struktury:

- *Model* - zawierający wektor sal i zamówień celem zarządzania nimi
- *Classroom* - zawierające ID w postaci ciągu znaków, identyfikujące daną salę wykładową, całkowitą zajętość sali liczoną w trakcie dodawania nowych zajęć, oraz listę przyporządkowanych do niej wykładów
- *Order* - prosta klasa zawierająca ID w postaci ciągu znaków, identyfikująca dane zamówienie, czas rozpoczęcia i czas zakończenia wykładu
- *Result* - klasa przechowująca mapę sal wykładowych z kluczem jako ID danej sali oraz wartością jako jej instancją dla danego rozwiązania (tj. ze znalezioną listą zamówień). Wykorzystywana jest do trzymania całkowitego czasu wykorzystania sal w danym rozwiązaniu oraz prezentacji wyników w postaci planu zajęć jako pliku *.html*
- *PartContainer* - struktura pomocnicza, używana w ostatnim algorytmie, do podziału wektora czasu na nakładające się wykłady. Zawiera początek i koniec okresu nakładających się zajęć oraz maksymalną ilość występujących wykładów w jednej godzinie = ilość potrzebnych sal do ułożenia wszystkich wykładów z tej części.

Algorytmy:

Pomocnicze algorytmy ze standardowej biblioteki użyte w programie:

- std::sort
- std::find
- std::remove

Metody rozwiązania (n – liczba zamówień, k – liczba sal wykładowych):

1. Algorytm programowania dynamicznego:

Opis:

Dla każdej sali przyporządkowanie dla niej optymalnego jej wykorzystania używając wykładów nie umieszczonych w salach już rozważonych. Działanie optymalne dla liczby sal = 1, dla większej ilości sal możliwe przypadki znalezienia nieoptymalnego wyniku dla całego rozważanego problemu.

Działanie:

Zajęcia sortowane są wg czasu zakończenia *~(n * log(n))*. Dla każdej sali *(k * (..)* sprawdzamy optimum czasowe gdybyśmy przypisali wykład *(.. * n + ..)* do rozważanej sali. W tym celu należy porównać dwie liczby a i b, gdzie a jest czasem trwania rozważanego wykładu, b jest sumą czasu trwania rozważanego wykładu i największego wykorzystania sali uzyskanego po dołączeniu wcześniejszych wykładów, który znajdujemy spośród wykładów kończących się nie później niż w chwili rozpoczęcia rozważanego wykładu. Ułożenie zajęć w następnych salach następuje z wyłączeniem już wstawionych zajęć do sal poprzedzających. Następnie dla najlepszego wyniku wykorzystania sali usuwane są *(.. + (n * ..)* użyte wykłady *(.. * n))*.

Przybliżona spodziewana złożoność obliczeniowa:

$$T_{(n)} \approx n \cdot \log n + 2 \cdot k \cdot n^2$$

$$O_{(T_{(n)})} \approx k \cdot n^2$$

2. Algorytm zachłanny:

Opis:

Wykonywany równolegle dla wszystkich sal. Nie gwarantuje to optymalnego układu, aczkolwiek zapewnia szybkie działanie.

Działanie:

Zajęcia posortowane wg czasu zakończenia *~(n * log(n))*. Następnie przechodząc po kolejnych wykładach *(.. + n * ..)* dopasowanie do sali nr. 1, jak nie pasuje (nakłada się) to do sali nr. 2 itd. *(.. * k)*.

Przybliżona spodziewana złożoność obliczeniowa:

$$T_{(n)} \approx n \cdot \log n + n \cdot k$$

$$O_{(T_{(n)})} \approx n \cdot \log n$$

3. Metoda *Brute Force*:

Opis:

Sprawdzenie wszystkich możliwych kombinacji i wybranie najlepszej możliwej. Dla dużych ilości zamówień i sal narzut czasowy obliczeń jest ogromny.

Działanie:

Rekurencyjnie sprawdzamy kolejne możliwości ulokowania zajęć, odrzucając kombinacje z nakładającymi się zajęciami jako nie-akceptowalne rozwiązanie, co zmniejsza ilość możliwości.

Przybliżona spodziewana złożoność obliczeniowa:

$$T_{(n)} \approx k + n + T_{(n-1,k)} + T_{(n,k-1)}$$

$$O_{(T_{(n)})} \approx ?\ (nie\ udało\ mi\ się\ oszacować)$$

4. Połączenie ww. algorytmów:

Opis:

Używane na mniejszych pod-problemach. Algorytm w niektórych przypadkach łańcuchowego nakładania się wykładów, sprowadza się do działania jednego w ww. podejść.

Działanie:

Tworzymy wektor, o długości *24 * liczba dni (5),* odpowiadający kolejnym godzinom. Do każdej godziny *(c * ..)* tworzymy listę do której przyporządkowujemy informację o wykładzie odbywającym się w tym czasie *(.. * n + ..)*. Przechodząc po wektorze *(.. + c * (..)* dzielimy go na części z odrębnie nakładającymi się (lub nie) wykładami *(.. n + ..)* i dodajemy unikalne wykłady do pomocniczej struktury *(.. + n * n) + ..)*. Jeżeli maksymalna długość listy jest mniejsza równa ilości dostępnych sal, to można przyporządkować sale po kolei *(n)*. Jeżeli nie to dla skupisk *(.. + c * ..)* nakładających się sal w miejscach gdzie nakłada się więcej zajęć niż jest sal *(.. * k * n * Talg)* wykonujemy jeden z wyżej wymienionych algorytmów w celu znalezienia odpowiedniego układu:

- Dla liczby nakładających się zajęć < k: - algorytm zachłanny
- Dla n < 10 i k < 5: - metoda *Brute Force*
- w p.p.: - algorytm programowania dynamicznego

Przybliżona spodziewana złożoność obliczeniowa:

$$T_{(n)} \approx c \cdot n + c \cdot (n + n \cdot n?) + c \cdot n \cdot k \cdot Talg_{(n)}$$

$$O_{(T_{(n)})} \approx k \cdot n^2$$

Przykładowe wyniki pomiarów czasu:

Ze względu na losowy charakter generowanych danych, w zależności od stopnia złożoności algorytmu dane zostały wygenerowane kilkakrotnie dla danego rozmiaru problemu i wybrany do porównań został czas najdłuższy.

Algorytm 1:

Przy zbliżonym rozmiarze problemu algorytm zdaje się dosyć sprawnie wykonywać w oszacowanym czasie. Jednakże dla większych instancji problemu w niektórych przypadkach można zaobserwować przeszacowanie złożoności. Aczkolwiek dla ograniczonej puli czasów w rozpatrywanym problemie, generowanie większej ilości danych może powodować w różnych przypadkach, wolniejsze bądź niekiedy szybsze dotarcie do rozwiązania.

```
./aal -t 61 8 1 1 10

 n   t(n)[ms]        q(n)
61        180     1.07569
62        180     1.04127
63        190      1.0645
64        190     1.03149
65        190           1
66        200     1.02098
67        200    0.990726
68        200    0.961801
69        220     1.02754
```

```
./aal -t 41 6 5 1 6

 n   t(n)[ms]        q(n)
41        180     1.19929
46        210     1.11153
51        280    0.964552
56        350           1
61        490    0.983248
66        550    0.942761
71        700    0.888713
```

```
./aal -t 51 6 5 1 6

 n   t(n)[ms]        q(n)
51        280     1.04206
56        370     1.14209
61        480     1.04058
66        540           1
71        720    0.987559
76        790    0.945687
81       1010    0.931337
```

```
./aal -t 51 6 10 1 5

 n   t(n)[ms]        q(n)
51        200     1.10575
61        360     1.15939
71        520     1.05956
81        730           1
91        940    0.906859
101      1610      1.1348
111      2240     1.18836
```

```
./aal -t 71 6 10 1 10

 n   t(n)[ms]        q(n)
71        710     1.25153
81       1000     1.18506
91       1280     1.06828
101      1640           1
111      2260     1.03721
121      2860     1.01254
131      3530     0.98421
```

Algorytm 2:

Ze względu na swoją prostotę, algorytm zdaje się utrzymywać podobną złożoność dla generowanych problemów.

```
./aal -t 161 6 1 2 10
```

n	t(n)[ms]	q(n)
161	230	1.01863
162	230	1.01235
163	240	1.04988
164	230	1
165	240	1.03715
166	240	1.03091
167	250	1.06743

```
./aal -t 161 6 10 2 10
```

n	t(n)[ms]	q(n)
161	230	1.01256
171	260	1.0143
181	280	0.974639
191	320	1
201	360	1.01558
211	380	0.972565
221	420	0.979651

```
./aal -t 201 6 20 2 6
```

n	t(n)[ms]	q(n)
201	350	0.984701
221	440	1.02353
241	520	1.0168
261	600	1
281	700	1.00623
301	820	1.02704
321	920	1.01297

Algorytm 4:

Odstępstwa od złożoności w algorytmie spowodowane są najprawdopodobniej przez rozkład wygenerowanych danych i podziału na różnej wielkości pod-problemy w trakcie szukania rozwiązania.

Jednakże wynik z ostatniego przykładu może być powodem *zacięcia* komputera podczas generowania, gdyż zaobserwowałem *brak mrugnięcia kursora* podczas obliczeń.

```
./aal -t 71 6 1 4 6

 n   t(n)[ms]      q(n)
71        310  0.990443
72        320   0.99419
73        330  0.997362
74        340         1
75        340  0.973511
76        350  0.975945
77        360  0.977925
```

```
./aal -t 71 6 5 4 6

 n   t(n)[ms]      q(n)
 71       310  0.753329
 76       470  0.996806
 81       630   1.02924
 86       690         1
 91       860  0.989489
 96       910  0.940793
101      1140  0.958297
```

```
./aal -t 71 6 10 4 6

 n   t(n)[ms]      q(n)
 71       300   1.04489
 81       450    1.0537
 91       620   1.02242
101       830         1
111      1040  0.943103
121      1320  0.923393
131      1680  0.925525
```

```
./aal -t 71 6 20 4 6

 n   t(n)[ms]      q(n)
 71       310    1.1597
 91       630   1.11587
111      1050    1.0227
131      1690         1
151      2670   1.03054
171      3900   1.03567
191      7370   1.40361
```