# Job shop

Wersja 1 – algorytm zachłanny i metoda losowa

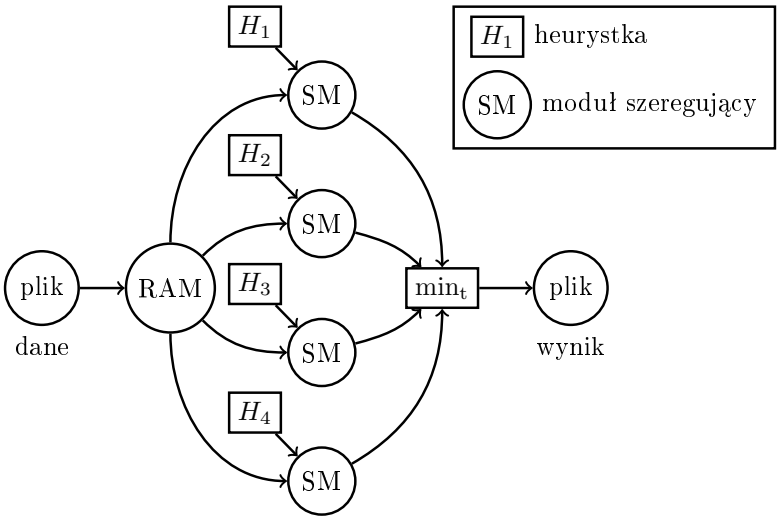
Alex Pawelski (147412) <alex.pawelski@student.put.poznan.pl>
Stanisław Puzio (151886) <stanislaw.puzio@student.put.poznan.pl>

25 listopada 2022

## Spis treści

# 1 Opis metody rozwiązania

Rdzeniem rozwiązania jest moduł szeregujący sterowany zadaną heurystyką. Opracowane zostały dwie heurystyki, z których każda ma po dwa warianty. Procedura startowa odczytuje dane z pliku do pamięci, a następnie przekazuje instancję modułowi szeregującemu – dla każdej heurystyki po kolei. Z czterech wygenerowanych w ten sposób rozwiązań, wybierane jest to o najkrótszym czasie uszeregowania.

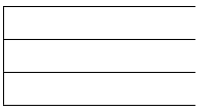



**Rysunek 1:** Schemat działania programu.

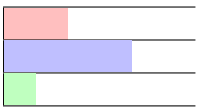
## 1.1 Moduł szeregujący

Moduł ten odpowiada za umieszczanie operacji na najwcześniejszych możliwych miejscach na odpowiednich osiach czasu – tak, aby nie kolidowały ze sobą, ani żeby czas wykonywania wstawianej operacji nie nakładał się na czasy wcześniej uszeregowanych operacji tego samego zadania. Jego implementacja w sposób bezpośredni odzwierciedla diagram Gantta. Poniższy schemat przedstawia zasadę działania modułu na przykładzie instancji z trzema maszynami i trzema zadaniami:

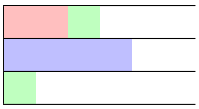
**Stan po początkowy.** Diagram został wypełniony nieskończenie[1] długimi przedziałami czasu oznaczonymi jako wolne.



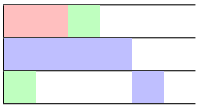
**Stan po wstawieniu pierwszych operacji każdego zadania.** Przedziałom dotąd wolnym zostały przypisane operacje, a ich granice zostały odpowiednio pomniejszone. Nowe, wolne, nieskończone przedziały zostały dodane za nimi.



**Stan po wstawieniu operacji zadania oznaczonego kolorem zielonym.** Operacja została wstawiona na najbliższe wolne miejsce.



**Stan po wstawieniu operacji zadania oznaczonego kolorem niebieskim.** Operacja została wstawiona w taki sposób, aby nie nachodziła na wcześniejszą operację tego zadania. Została przypisana dotąd wolnemu przedziałowi, w którym się znalazła, a jego granice zostały odpowiednio pomniejszone. Nowe, wolne, przedziały zostały dodane przed i za operacją.



**Stan po wstawieniu operacji zadania oznaczonego kolorem czerwonym.**

[1] Przedziały są nieskończenie długie tylko w teorii. W praktyce, prawa granica „nieskończonego" przedziału zdefiniowana jest jako maksymalna wartość typu całkowitego przechowującego wartości czasowe – w tym przypadku jest to $2^{32} - 1$

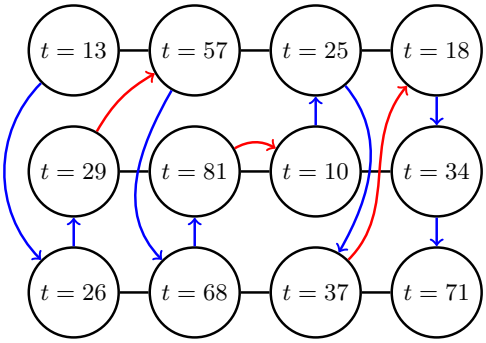
.

## 1.2 Heurystyki

Heurystyki odpowiadają za kolejność, w której moduł szeregujący rozpatruje operacje. Zaimplementowane zostały jako komparatory parametryzujące sortowanie. Na przedstawionych schematach narysowano przykładowe instancje z 3 zadaniami, z których każde składa się z 4 operacji. Zapis $t = x$ oznacza, że czas operacji jest równy $x$. Niebieskie strzałki oznaczają kolejność szeregowania w ramach jednej serii operacji ($i$-tych operacji każdego zadania), a czerwone – przejście do kolejnej serii (zwiększenie $i$ o 1).

### 1.2.1 Algorytm zachłanny

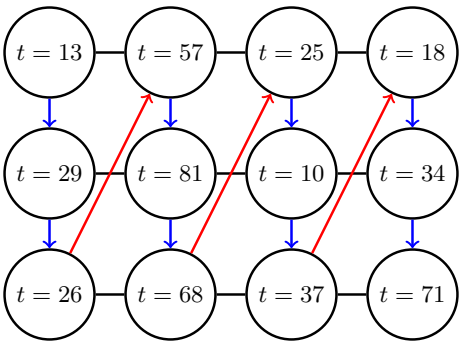
Metoda ta polega na dzieleniu problemu szeregowania operacji wszystkich zadań na podproblemy szeregowania $i$-tych operacji każdego z zadań (gdzie $i$ na początku przyjmuje wartość 1, a na końcu wartość równą ilości operacji w jednym zadaniu). Na każdym $i$-tym etapie operacje są sortowane według czasu ich trwania ich $i$-tych operacji. Wariant pierwszy sortuje operacje niemalejąco, wariant drugi – nierosnąco. Po wykonanym sortowaniu, operacje te przekazywane są do wcześniej opisanego modułu szeregującego. Opisany algorytm prezentuje przykład poniżej:




**Rysunek 2:** Schemat działania algorytmu zachłannego (wariant sortujący niemalejąco).

### 1.2.2 Metoda losowa

Ta prosta metoda polega na przechodzeniu po zadaniach w takiej kolejności, w jakiej umieszczone są w instancji. W jej wariancie, operacje odczytywane są w kolejności odwrotnej.




**Rysunek 3:** Schemat działania metody losowej (wariant idący do przodu).

# 2 Badania efektywnościowe

## 2.1 Badanie czasu wykonywania

Przeprowadzonych zostało kilka testów w celu sprawdzenia szybkości działania algorytmu.

### 2.1.1 Badanie kontrolne

Dla instancji: `tai20`, `tai21`, `tai22`, `tai23`, `tai24` oraz `tai25` zostały wykonane pomiary kontrolne czasu wykonywania. Algorytm dla każdej instancji wykonany został 10000 razy. Poniżej przedstawione są średnie wyniki pomiarów czasu:

| symbol instancji | śr. czas wykonywania [µs] |
|---|---|
| `tai20` | $1131 \pm 69$ |
| `tai21` | $1430 \pm 68$ |
| `tai22` | $1497 \pm 112$ |
| `tai23` | $1471 \pm 54$ |
| `tai24` | $1430 \pm 107$ |
| `tai25` | $1496 \pm 86$ |

Dodatkowo, obliczenie średniej i odchylenia standardowego zostało przeprowadzone po odrzuceniu 30 pierwszych pomiarów.

| symbol instancji | śr. czas wykonywania [µs] |
|---|---|
| `tai20` | $1128 \pm 42$ |
| `tai21` | $1427 \pm 25$ |
| `tai22` | $1494 \pm 91$ |
| `tai23` | $1469 \pm 34$ |
| `tai24` | $1427 \pm 83$ |
| `tai25` | $1494 \pm 49$ |

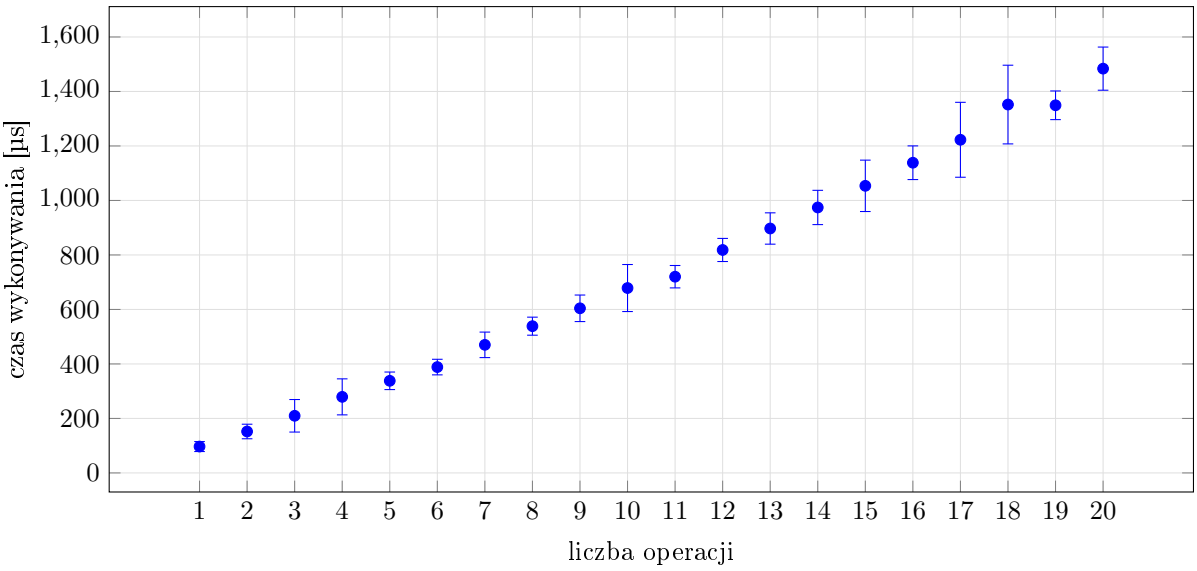
### 2.1.2 Badanie zależności czasu wykonywania od rozmiaru instancji

W celu zbadania zależności czasu wykonywania algorytmu od rozmiaru danych, zostało przeprowadzonych 20 testów dla instancji `tai25`, w których liczba uwzględnianych operacji zmieniała się w zakresie od 1 do 20. W każdym teście, algorytm został wykonany 10000 razy. W tabeli i na wykresie poniżej prezentowane są średnie wyniki pomiarów czasu ($\mu$ – średnia wyników, $\sigma$ – odchylenie standardowe):

| | Liczba zadań | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| $\mu$ [µs] | 96.8 | 152.1 | 209.7 | 279.2 | 338.1 | 388.5 | 470.1 | 538.6 | 604.2 | 678.6 |
| $\sigma$ [µs] | 18.3 | 26.7 | 59.7 | 66.1 | 32.2 | 28.6 | 46.8 | 33.1 | 48.7 | 86.3 |

| | Liczba zadań | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| $\mu$ [µs] | 720.2 | 818.3 | 897.1 | 974.3 | 1053.7 | 1138.5 | 1222.7 | 1352.0 | 1349.3 | 1483.8 |
| $\sigma$ [µs] | 41.1 | 42.4 | 57.5 | 62.9 | 94.3 | 61.8 | 137.5 | 144.4 | 52.6 | 79.2 |

**Tabela 1:** Średnie wyniki pomiarów czasu dla instancji `tai25`; dla liczby operacji od 1 do 20.




**Rysunek 4:** Wykres czasu wykonywania algorytmu od liczby operacji w zadanej instancji z zaznaczonym odchyleniem standardowym dla każdej średniej pomiarów.

### 2.2 Badanie jakości wyniku

Algorytm został przetestowany dla instancji: `tai20`, `tai21`, `tai22`, `tai23`, `tai24` oraz `tai25` pod kątem jakości wyników. Miara oceny jakości zdefiniowana została jako $\frac{r_o}{r_a}$, gdzie $r_o$ jest wartością dolnej granicy funkcji celu podaną dla danej instancji, a $r_a$ jest czasem uszeregowania wyliczonym przez algorytm. Im wyższa wartość oceny tym lepszy wynik. Otrzymane wyniki prezentowane są w tabeli poniżej:

| symbol instancji | dolna granica f. celu | uzyskany wynik | ocena jakości wyniku |
|---|---|---|---|
| `tai20` | 1213 | 1729 | 0.70 |
| `tai21` | 1217 | 2011 | 0.61 |
| `tai22` | 1314 | 2040 | 0.64 |
| `tai23` | 1248 | 1983 | 0.63 |
| `tai24` | 1284 | 1957 | 0.66 |
| `tai25` | 1256 | 1933 | 0.65 |

**Tabela 2:** Ocena jakości wyniku dla sześciu przykładowych instancji.

## 3 Wnioski

1. Zaprezentowany algorytm znajduje wynik w czasie wielomianowym, jednak nie daje on gwarancji optymalnego rozwiązania. Złożoność czasowa oszacowana została jako $O(m^2 * n \log(n))$, gdzie $n$ – ilość zadań; $m$ – ilość operacji w jednym zadaniu (przy założeniu, że wszystkie zadania składają się z jednakowej ilości operacji).

2. Wyniki uzyskane przez algorytm dla instancji 20x15 i 20x20 nie odbiegają od dolnej granicy funkcji celu o więcej niż 50%. Najlepszy wynik uzyskało rozwiązanie instancji 20x15, co może wskazywać na korelację ujemną rozmiaru instancji względem jakości wyniku.

3. W pomiarze kontrolnym, znaczna różnica w odchyleniu standardowym po odrzuceniu 30 pierwszych pomiarów wynika stąd, że pierwsze pomiary były przeważnie kilkukrotnie większe niż średnia wszystkich pomiarów. Przyczyna tego zjawiska nie została ustalona.

# 4 Kompilacja i użycie

## 4.1 Kompilacja

**macOS / Linux**
Należy użyć kompilatora Clang w wersji 10 albo GCC w wersji lub 8 nowszej, wywołując polecenia:

```
~$ make job_shop
```

**Cygwin**
Należy użyć kompilatora GCC w wersji 10.2.0 lub nowszej, wywołując polecenia:

```
~$ make exe
```

## 4.2 Opis programów

**Program *job_ shop*** – program rozwiązujący problem job shop.

```
./job_shop [-d input] [-o output] [-l jobs] [-r count] [-g] [-t]
```

Objaśnienie opcji:

- `-d input` – ścieżka do pliku z instancją; domyślnie – `./data.txt`
- `-o output` – ścieżka do pliku wyjściowego; w przypadku braku, wynik wypisywany jest tylko na wyjściu standardowym
- `-l jobs` – liczba pierwszych zadań branych pod uwagę; w przypadku braku lub podania wartości `0`, brane pod uwagę są wszystkie zadania zawarte w pliku
- `-r count` – liczba powtórzeń algorytmu (przydatne do testów wydajnościowych); w przypadku braku algorytm wykonywany jest raz
- `-g` – wyświetlenie diagramu Gantta (niezalecane dla instancji z długimi operacjami)
- `-t` – wypisanie czasu wykonania algorytmu zamiast wyniku

**Program *convert*** – konwerter instancji w formacie Tailarda na format J. E. Beasleya.

```
./convert <input> <output> [-d]
```

Objaśnienie opcji:

- `input` – plik (lub folder) z instancją/ami w formacie Tailarda
- `output` – plik (lub folder) z wynikiem/ami konwersji
- `-d` – w przypadku podania tej opcji jako ostatni argument, argumenty `input` i `output` interpretowane są jako foldery (w przypadku braku, zawsze interpretowane są jako pliki)

**Program *test*** – narzędzie do przeprowadzania automatycznych testów poprawności wyników z użyciem programu sprawdzającego `chk_jsorl`.

```
./test <solver_exec> <data_dir> <output_dir> [checker_exec]
```

Objaśnienie opcji:

- `solver_exec` – ścieżka do programu `job_shop`
- `data_dir` – folder z instancjami w formacie J. E. Beasleya
- `output_dir` – folder, do którego zapisane zostaną wyniki
- `checker_exec` – ścieżka do programu `chk_jsorl` (w przypadku braku, sprawdzanie poprawności wyników nie jest przeprowadzane)