Algorytmy zastępowania stron

Ich implementacja i testowanie

Wprowadzenie do systemów komputerowych

Antoni Cichoń (279471) | 07.06.2024

Cyberbezpieczeństwo (I st.) II semestr

Oświadczam, że sprawozdanie zostało przeze mnie wykonane osobiście i samodzielnie

Spis treści

Informacje wstępne

Celem tej symulacji jest poznanie wydajności podstawowych algorytmów zamiany stron w pamięci wirtualnej.

Do stworzenia symulacji wybrałem język python. Głównym powodem była obsługa wielu pomocnych bibliotek. Symulacja również nie miała być optymalizowana pod kątem wydajności czy złożoności obliczeniowej, co utwierdziło mnie w przekonaniu że python będzie odpowiednim wyborem. Do napisania programu symulacyjnego użyłem bibliotek matplotlib oraz numpy.

Program symulacyjny najpierw generuje zestaw danych testowych a następnie przeprowadza testy na każdym z zadanych algorytmów. Wyniki są zapisywane w plikach .csv oraz generowane są histogramy prezentujące uzyskane dane w czytelny sposób.

Generacja danych testowych

Generator danych testowych przyjmuje od użytkownika 3 argumenty:

1. Liczbę stron do wygenerowania
2. Zakres odwołań do stron
3. Opcjonalnie ziarno generatora liczb losowych. Przy pominięciu tego argumentu generator używa losowego ziarna

Do przeprowadzenia testów użyłem poniższych wartości:

1. Liczba generowanych stron: 512
2. Zakres wygenerowanych stron: 1 – 32
3. Ziarno: 1234

Generator najpierw usuwana wszystkie istniejące foldery od „test1” do „testn”, gdzie n jest ostatnim testem w folderze symulacji. Następnie generowane są nowe dane testowe. Użytkownik ma wpływ na sposoby generacji danych, poprzez modyfikację listy „genfunctions”. Może on usuwać algorytmy generujące oraz dodawać własne. Ja użyłem poniższych algorytmów:

```
genfunctions = [ascending, uniform, binomial, exponential]
```

inputgen.py, line 19

1. „ascending” – generuje strony z zadanego zakresu w kolejności rosnącej
2. „uniform” – generuje strony wg rozkładu jednostajnego dyskretnego
3. „binomial” – generuje strony wg rozkładu dwumianowego
4. „exponential” – generuje strony wg rozkładu wykładniczego

Po wygenerowaniu zestawu danych testowych, generator tworzy w folderze symulacji folder o nazwie „testx”, gdzie x jest numerem algorytmu za pomocą którego generowane były strony. Dane testowe zapisywane są do pliku „input.txt”. Pierwsza linijka tego pliku zawiera sposób generacji danych. Każda kolejna linijka zawiera żądaną stronę.

Dodatkowo dla każdego testu tworzony jest histogram o nazwie „inputvisual.png” wizualizujący sumę wystąpień danej strony w wygenerowanym teście.

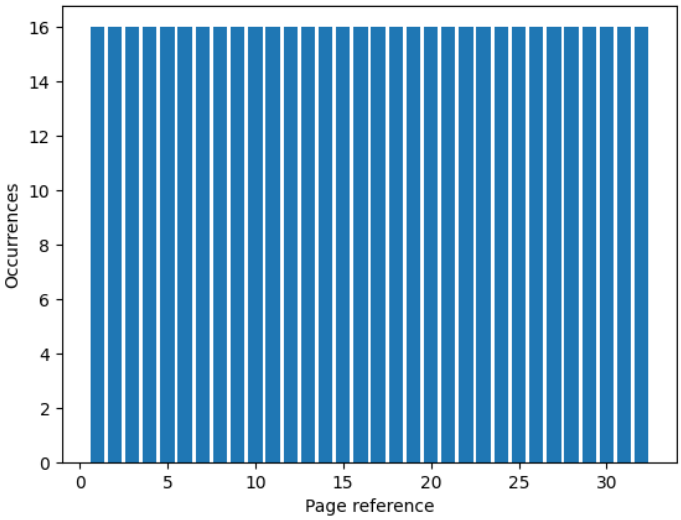
Na koniec generator zwraca listę ścieżek do utworzonych folderów testowych. Program informuje użytkownika o każdym stworzonym folderze. W przypadku niepowodzenia, przekazuje którego testu nie mógł stworzyć oraz pełną ścieżkę do wadliwego folderu i awaryjnie kończy swoje działanie. Kontynuowanie generacji teoretycznie może nastąpić, jednak zdecydowałem się przerwać program ze względu na potencjalne wystąpienie niezgodności między numeracją testów a zadeklarowaną listą funkcji generujących.

Dla moich danych wejściowych wykresy wystąpień stron wyglądają następująco:

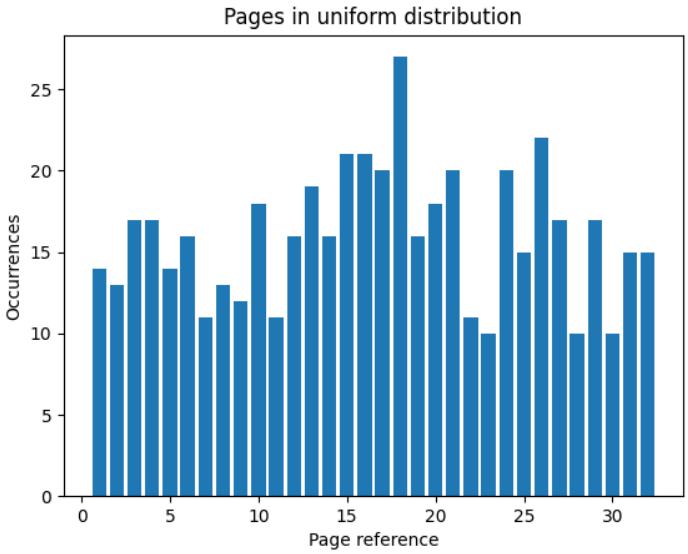
test 1 (ascending):

każda strona występuje dokładnie tyle samo razy

/test1/inputvisual.png



/test2/inputvisual.png



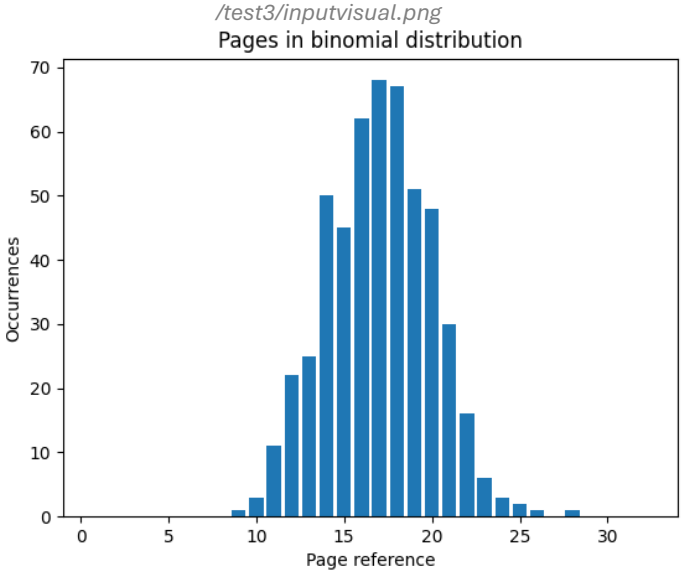
test 2 (uniform):

wystąpienia stron generowane są losowo i każda strona ma równe prawdopodobieństwo na wystąpienie

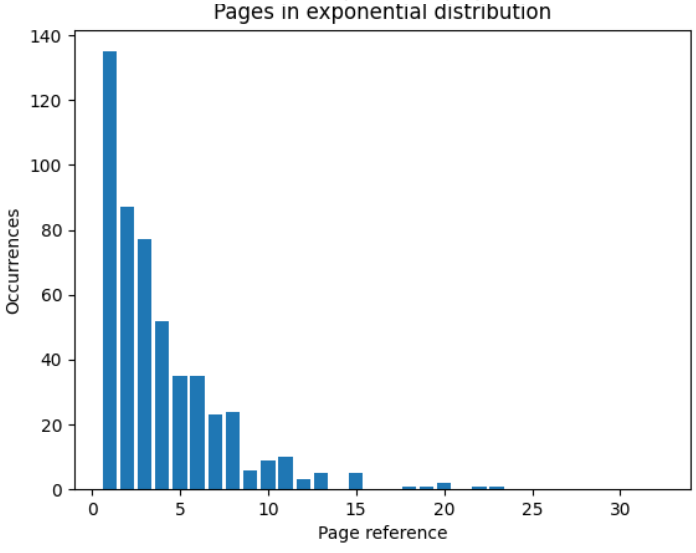
test 3 (binomial):

najczęściej występują strony ze środka zadaneg zakresu.

/test3/inputvisual.png



/test4/inputvisual.png



test 4 (exponential):

prawopodobieństwo wystąpienia strony jest równe 0,75 * prawdopodobieństwo poprzedniej strony. Prawdopodobieństwo strony nr 1 wynosi 25%.

Przeprowadzanie testu

Po uzyskaniu ścieżek do wygenerowanych testów, program rozpoczyna wynonywanie symulacji. Testowane są algorytmy zawarte w słowniku „usealgorithms”. Klucz w słowniku to nazwa algorytmu a wartościami są funkcje zaimportowane z odpowiednich plików. Użytkownik może dodawać własne bądź usuwać je z listy. Ja użyłem 4 algorytmów zamiany stron:

1. First in first out
2. Least frequently used
3. Last recently used
4. Most frequently used

```
usealgorithms = {
    "First in first out" : fifo,
    "Least frequently used" : lfu,
    "Last recently used" : lru,
    "Most frequently used": mfu}
```

main.py, line 16

(Szczegółowe omówienie każdego z algorytmu znajduje się na następnych stronach)

Każdy z tych algorytmów przyjmuje zakres dostępnych ramek pamięci a następnie sprawdza ile razy wystąpił brak strony dla dostępnych ramek z zadanego zakresu.

Algorytmy zastępowania stron

Zadaniem algorytmu zastępowania stron jest zarządzanie stronami w pamięci w taki sposób, aby żądana strona się w niej znalazła. Wybrane algorytmy na podstawie swoich kryteriów wybierają stronę w pamięci do usunięcia a następnie ładują żądaną stronę w zwolnioną ramkę. Błąd zdefiniowany jest jako brak żądanej strony w pamięci.

Anomalia Belady'ego

Warto również wspomnieć o tzw. „Anomalii Belady'ego". Polega ona na zwiększeniu się liczby błędów po zwiększeniu się dostępnych ramek pamięci. Przeczy to intuicyjnemu myśleniu „więcej ramek musi oznaczać mniej błędów". Dzieje się tak ze względu na definicję błędu – braku żądanej strony w pamięci. Za błąd uznaje się więc zarówno usunięcie wybranej strony z ramki i załadowanie żądanej, jak i załadowanie żądanej strony do pustej ramki. Im więcej dostępnych ramek, tym więcej błędów powstaje przy pierwszym ładowaniu stron (do zapełnienia pustych ramek).

1. First in first out

Jest to najprostszy algorytm z użytych. W momencie pojawienia się strony, która nie znajduje się w pamięci, usuwana jest strona znajdująca się w pamięci najdłużej.

First in first out

kolejka stron: 1 3 0 3 5 6 3

obecnie żądana strona:		1	3	0	3	5	6	3
ramka:	1			→0	0	0	0	0→3
	2		→3	3	3	3	3→6	6
	3	→1	1	1	1	1→5	5	5
		błąd	błąd	błąd		błąd	błąd	błąd

łącznie błędów pamięci: 6

+ prosty w implementacji
+ niski koszt w zasobach
+ „sprawiedliwe" traktowanie stron

- zazwyczaj mniej wydajny od reszty
- podatny na anomalię Belady'ego

2. Least frequently used

Algorytm śledzi sumę żądań dla każdej strony i w momencie konieczności opróżnienia ramki pamięci, wybiera tę w której znajduje się najrzadziej (do tej pory) żądana strona.

Least frequently used

kolejka stron: 1 3 0 3 5 6 3

obecnie żądana strona:		1	3	0	3	5	6	3
ramka:	1			→0	0	0	0→6	6
	2		→3	3	3	3	3	3
	3	→1	1	1	1	1→5	5	5
		błąd	błąd	błąd		błąd	błąd	

łącznie błędów pamięci: 5

+ najwyższa wydajność
+ szczególnie wydajny przy wielu powtarzających się stronach

- duży koszt zasobów
- istnieje możliwość pozostania strony w pamięci na długi czas

3. Least recently used

Algorytm śledzi kolej w jakiej przychodziły strony i w momencie konieczności opróżnienia ramki pamięci, wybiera tę w której znajduje się najdawniej zażądana strona.

Least recently used

kolejka stron: 1 3 0 3 5 6 3

obecnie żądana strona:		1	3	0	3	5	6	3
ramka:	1			→0	0	0	0→6	6
	2		→3	3	3	3	3	3
	3	→1	1	1	1	1→5	5	5
		błąd	błąd	błąd		błąd	błąd	

łącznie błędów pamięci: 5

+ prosta implementacja
+ wydajny przy dużej częstotliwości konkretnej strony lub stron
+ eliminuje możliwość długotrwałego zajmowania ramki występującą przy algorytmie least frequently used

- duży koszt zasobów
- zazwyczaj mniej wydajny niż algorytm least frequently used

4. Most frequently used

Algorytm śledzi częstotliwość żądań dla każdej strony i w momencie konieczności opróżnienia ramki pamięci, wybiera tę w której znajduje się najczęściej (do tej pory) żądana strona.

Most frequently used

kolejka stron: 1 3 0 3 5 6 3

obecnie żądana strona:		1	3	0	3	5	6	3
ramka:	1			→0	0	0	0	0→3
	2		→3	3	3	3→5	5	5
	3	→1	1	1	1	1	1→6	6
		błąd	błąd	błąd		błąd	błąd	błąd

łącznie błędów pamięci: 6

\+ wydajny przy małej częstotliwości Konkretnej strony lub stron
\+ eliminuje możliwość długotrwałego zajmowania ramki występującą przy algorytmie least frequently used

\- duży koszt zasobów
\- bardzo niska stabilność
\- w testach wypadł najgorzej
\- najsłabsze skalowanie wraz ze wzrostem dostępnych ramek
\- podatny na anomalię Belady'ego

Reprezentacja wyników

Po przeprowadzeniu testu dla każdego algorytmu wyniki zapisywane są w pliku .csv z nazwą odpowiadającą testowanemu algorytmowi (np. Firstinfirstout-result.csv). Pierwsza linijka zawiera nagłówki prezentowanych danych, tj. dostępne ramki i sumę błędów. Każda kolejna linijka zawiera dane a kolumny rozdzielone są średnikami.

Dodatkowo, generowane są wykresy przedstawiające wydajność każdego z algorytmów o nazwie „resultgraph.png”. Program informuje użytkownika o zapisie wyników. W przypadku niepowodzenia, obecny test jest pomijany i następuje przejście do kolejnego.

Po przeprowadzeniu testu, zawartość folderu tego testu będzie wyglądać w następujący sposób:

```
C:\Users\Antek\Desktop\pisanie\SK\PageReplacement\test1>dir /b /oe
Mostfrequentlyused-result.csv
Leastrecentlyused-result.csv
Leastfrequentlyused-result.csv
Firstinfirstout-result.csv
inputvisual.png
resultgraph.png
input.txt
```

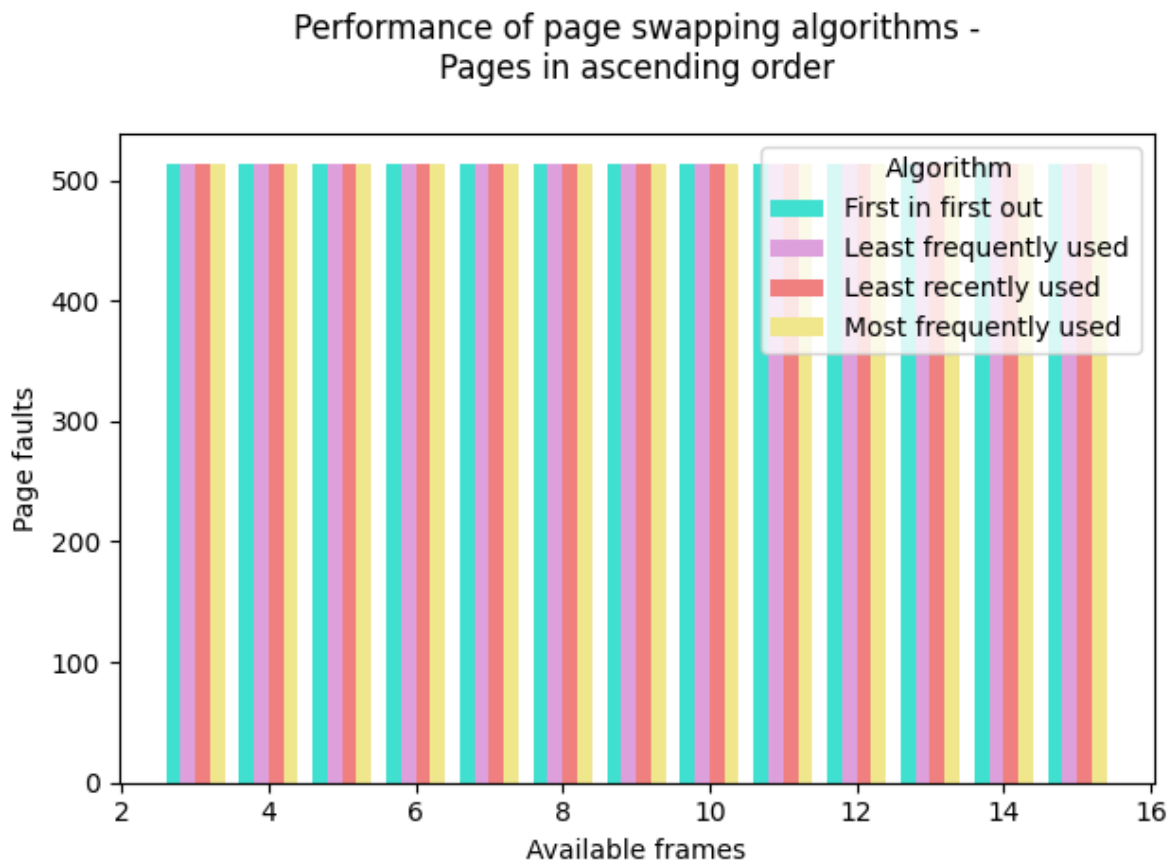
Z oczywistych powodów ścieżka do folderu będzie się różnić

Po przeprowadzeniu wszystkich testów (ich lista kontrolowana jest przez generator danych) program oblicza czas wykonania i zwraca go do konsoli.

Interpretacja wyników

Wyniki poszczególnych testów (przy użyciu ziarna „1234” podczas generowania danych) wyglądają następująco

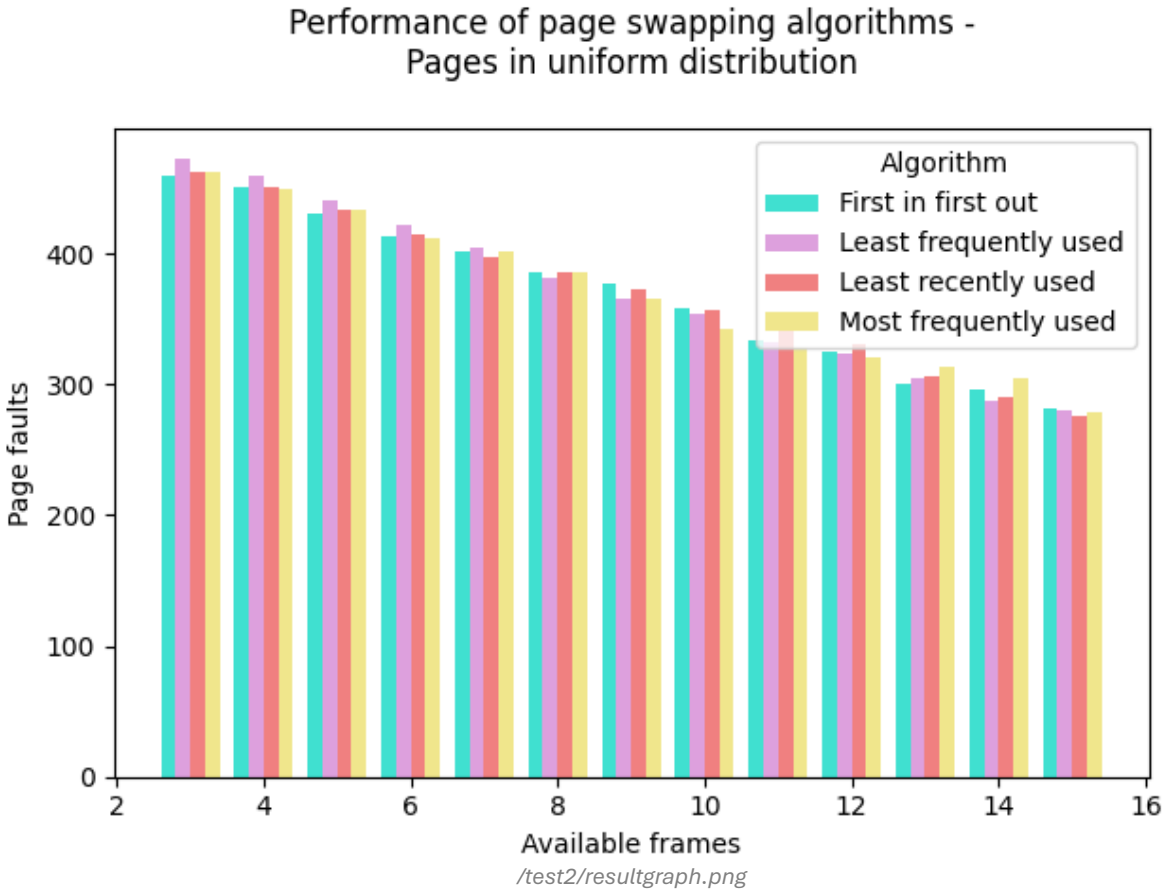
1. Strony w rosnącej kolejności



/test1/resultgraph.png

Żaden z zaimplementowanych algorytmów nie poradził sobie z tak wygenerowanymi stronami. W przypadku tak specyficznych danych należałoby stworzyć algorytm który bierze pod uwagę nie tylko pojedyncze strony ale też wzorce i zależności między występowaniem konkretnych stron po sobie. Jest to jednak skrajny i bardzo nienaturalny przypadek względem realnych scenariuszy.

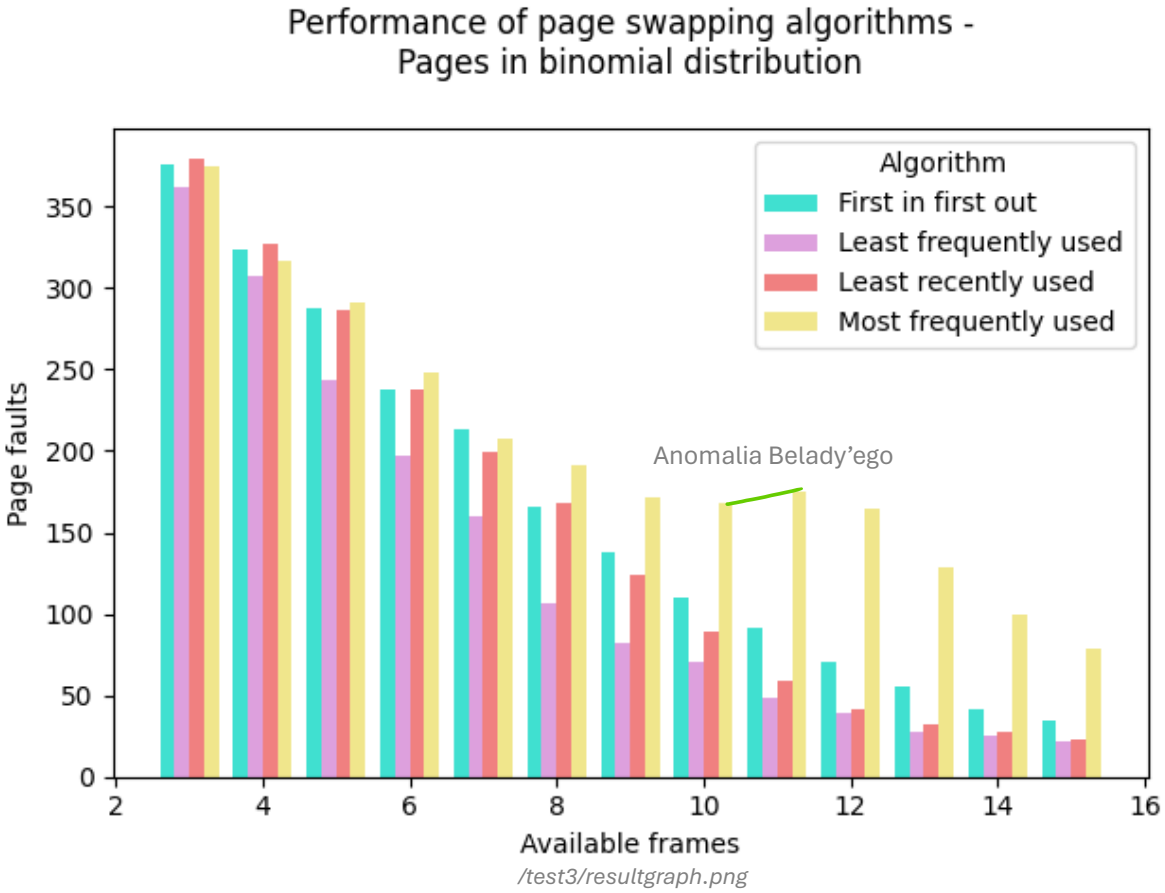
2. Strony wygenerowane losowo



/test2/resultgraph.png

W przypadku losowo generowanych stron również można zauważyć, że żaden z algorytmów nie jest wybitnie lepszy od reszty. Wynika to z losowości stron, przez co z założenia strony występują z taką samą częstotliwością i regularnością. Przy niewielu dostępnych ramkach najgorzej radzi sobie LFU, jednak wraz ze wzrostem dostępnych ramek jego wydajność staje się coraz lepsza – najlepiej się skaluje. Najgorzej pod tym względem wypada MFU. Wyniki te są na tyle do siebie zbliżone, że można uznać algorytmy za porównywalnie wydajne dla takiego scenariusza.

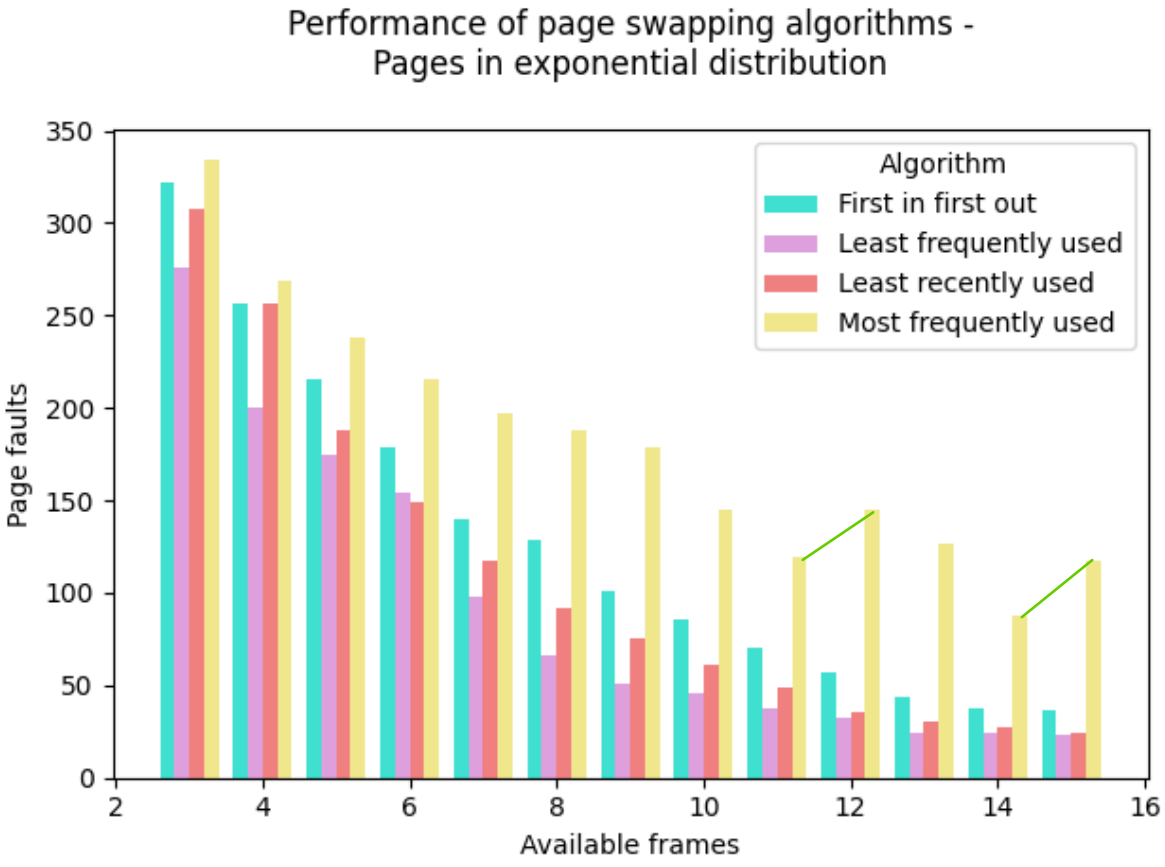
3. Strony występujące wg rozkładu dwumianowego



/test3/resultgraph.png

Ten test można uznać za najbardziej zbliżony do rzeczywistych scenariuszy. Najwydajniejszym algorytmem okazał się LFU. FIFO i LRU wypadły bardzo podobnie, jednak ze wzrostem ramek, LRU był nieznacznie wydajniejszy od FIFO. Najgorzej prezentuje się MFU nie tylko w pojedynczych przypadkach, ale też pod względem skalowania. Idealnie również widać jak doprowadził on do powstania anomalii Belady'ego.

4. Strony występujące wg rozkładu wykładniczego



/test4/resultgraph.png

Wyniki tego testu wygladają podobnie do tych ze stronami występującymi wg tozkładu dwumianowego, jednak są bardziej spolaryzowane między poszczególnymi algorytmami. Algorytm MFU ponownie poradził sobie najgorzej, dwukrotnie generując anomalię B. (zaznaczone na zielono). Najlepiej wypadł LFU.

Wnioski

Przed wyciągnięciem wniosków należy wziąć pod uwagę fakt, że powyższe testy są jedynie syntetycznymi scenariuszami i różnią się od realnych sytuacji.

Z przeprowadzonych testów wynika, że algorytm „Least frequently used” jest najwydajniejszy spośród wybranych czterech. Różnice są szczególnie widoczne przy stosunkowo niewielu dostępnych ramkach. Wraz z ich wzrostem różnie między LFU a „Least recently used” stają się coraz mniejsze. Algorytm „First in first out” ze względu na swoje ograniczenia zazwyczaj generował więcej błędów. W testach najgorzej wypadł „Most frequently used”, co wynika ze sposobu w jaki dane testowe były generowane – nie został przygotowany test, który pozwoliłby MFU się „wykazać”. Oznacza to, że MFU jest specyficznym algorytmem i aby w pełni wykorzystać jego potencjał, należałoby użyć go w konkretnych sytuacjach, co czyni go mało uniwersalnym.

Podsumowanie

Każdy z przetestowanych algorytmów ma swoje zalety i wady. W przypadku potrzeby ich implementacji nie można wybrać obiektywnie najlepszego z nich. Należy odpowiednio dopasować nasz wybór do parametrów takich jak dostępność zasobów, typ danych do przetworzenia czy prostota implementacji. Można się również zastanowić nad hybrydyzacją tych algorytmów w celu zwiększenia wydajności przy pojawieniu się skrajnych przypadków.

Użycie języka python do stworzenia symulacji było odpowiednią decyzją. Dzięki prostocie nie musiałem rozwiązywać trywialnych problemów i mogłem skupić się na implementacji algorytmów i generacji danych. Zbawienna okazała się biblioteka matplotlib, która umożliwiła czytelną prezencję danych zarówno wejściowych jak i wyjściowych.