Nazwa kursu: Projektowanie Algorytmów i Metody Sztucznej Inteligencji

Tytuł: Projekt 1 (sortowania)

Data oddania: 03.04.2019 r.

Termin zajęć: Środa, 11-13

Prowadzący: dr inż. Łukasz Jeleń

Dane studenta: Patryk Szczygieł 241578

1. Cel ćwiczenia

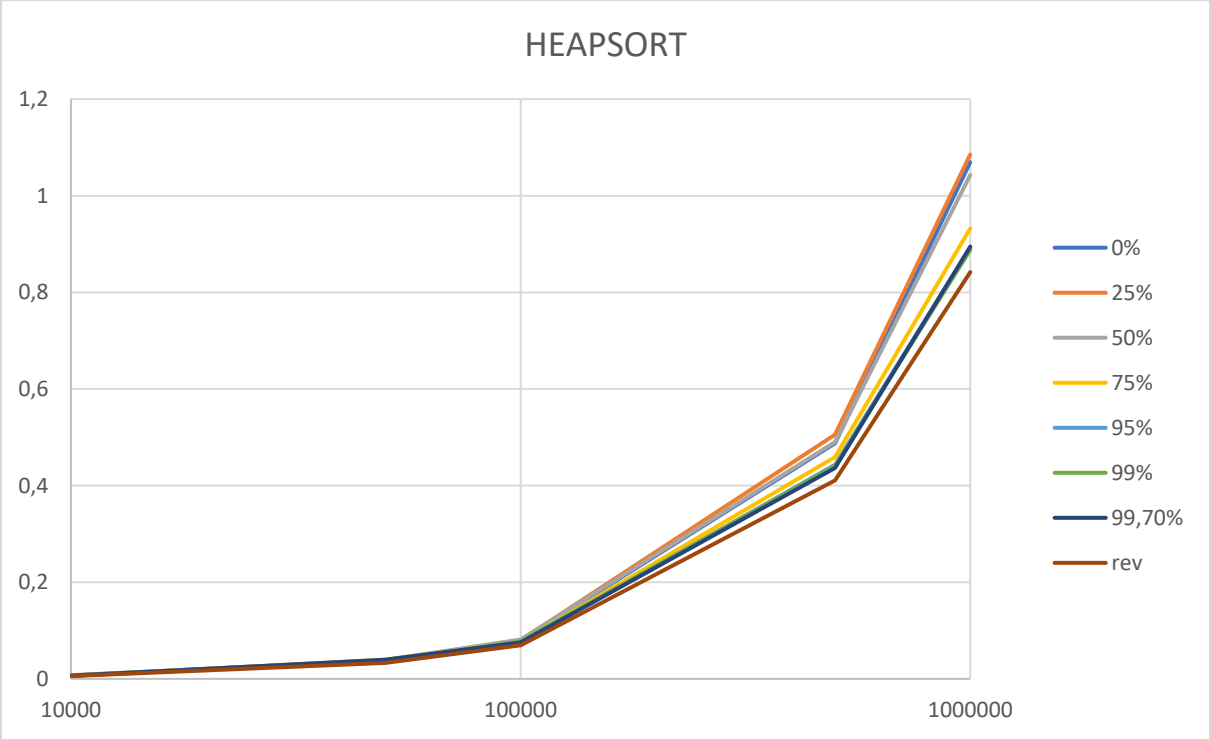
Celem projektu było zaimplementowanie trzech algorytmów (odpowiednio wybranych z tabeli w zadaniu projektowym) sortowania: sortowanie przez scalanie (Merge Sort), sortowanie przez kopcowanie (Heap Sort) oraz sortowanie Shella (Shell Sort). Należało porównać algorytmy, przeprowadzając pomiary czasu sortowania stu tablic o ilości elementów 10.000, 50.000, 100.000, 500.000 i 1.000.000 w różnych stopniach wstępnego posortowania: 0%,25%,50%,75%,95%,99%,99.7% oraz tablicy posortowanej, lecz w odwrotnej kolejności.

2. Sortowanie przez kopcowanie (Heap Sort)

Sortowanie przez kopcowanie jest jednym z algorytmów szybkich. Do sortowania wykorzystuje tzw. kopiec, czyli kompletne drzewo binarne (jest ono zapełnione elementami maksymalnie do lewej strony). Pierwszym elementem algorytmu jest stworzenie kopca maksymalnego – takiego, w którym każdy korzeń ma wartość większą od każdego ze swoich dwóch poddrzew. W drugim etapie algorytmu następuje sortowanie właściwe. Polega ono na usunięciu wierzchołka kopca, zawierającego element maksymalny, a następnie wstawieniu w jego miejsce elementu z końca kopca i odtworzenie porządku kopca (kopiec maksymalny). W zwolnione w ten sposób miejsce, zaraz za końcem zmniejszonego kopca wstawia się usunięty element maksymalny. Operacje te powtarza się aż do wyczerpania elementów w kopcu. Każde wstawienie nowego elementu podczas budowania maksymalnego kopca może wymagać tyle przesunięć, ile jest węzłów kopca na ścieżce od miejsca wstawiania do korzenia drzewa. Łatwo więc można zauważyć, że liczba ta jest równa lub mniejsza od wysokości kopca, czyli log2n. Ponieważ operację tę wykonujemy (n - 1) razy, to dostajemy górne oszacowanie właśnie O(n log n).

		10000	50000	100000	500000	1000000
0%	min	0,006	0,035	0,075	0,466	1,006
	max	0,026	0,053	0,084	0,57	1,202
	średnia	0,00747	0,03816	0,07935	0,48757	1,06949
25%	min	0,005	0,035	0,077	0,479	1,01
	max	0,01	0,053	0,089	0,583	1,363
	średnia	0,00667	0,03858	0,08032	0,50542	1,08534
50%	min	0,005	0,035	0,077	0,461	0,996
	max	0,01	0,073	0,094	0,599	1,302
	średnia	0,00659	0,03921	0,08155	0,48993	1,04283
75%	min	0,005	0,034	0,072	0,44	0,92
	max	0,008	0,107	0,085	0,513	0,968
	średnia	0,00636	0,03991	0,07699	0,4589	0,93227
95%	min	0,005	0,034	0,072	0,426	0,881
	max	0,008	0,051	0,08	0,472	0,937
	średnia	0,00623	0,03634	0,07543	0,44298	0,89203
99%	min	0,006	0,033	0,072	0,417	0,879
	max	0,008	0,041	0,085	0,506	0,905
	średnia	0,00666	0,03616	0,07768	0,44123	0,88777
99,70%	min	0,005	0,034	0,072	0,413	0,874
	max	0,008	0,072	0,085	0,499	0,97
	średnia	0,00653	0,0396	0,07565	0,43637	0,89471
rev	min	0,005	0,03	0,066	0,399	0,831
	max	0,007	0,039	0,075	0,463	0,917
	średnia	0,00573	0,03285	0,06964	0,41098	0,8419

Tab. 1. Wyniki pomiarów dla sortowania przez kopcowanie



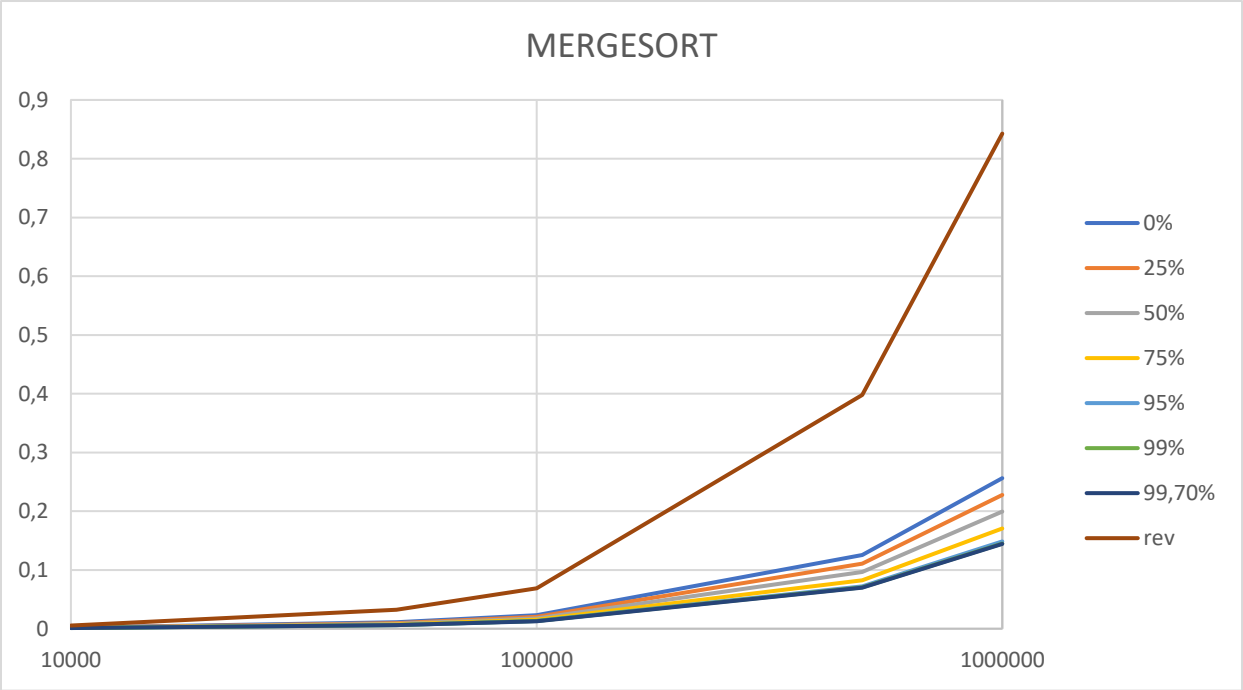
Rys. 1. Wykres zależności czasu od ilości elementów dla sortowania przez kopcowanie

3. Sortowanie przez scalanie (Merge Sort)

Sortowanie przez scalanie jest algorytmem rekurencyjnym z grupy algorytmów szybkich. Tablica w każdym kroku zostaje podzielona na dwie części (dwa mniejsze podproblemy), aż do powstania tablic jednoelementowych, które są uważane za posortowane. Następnie, algorytm porównuje „lewy podproblem" oraz „prawy podproblem" i scala je (merge) w odpowiedniej kolejności. Merge-sort w każdym wywołaniu rekurencji dzieli tablicę na pół. Dla tablicy o rozmiarze 2 algorytm „zejdzie o jeden poziom rekurencji w dół", dla 4 o dwa, dla 8 o trzy. Można więc zauważyć, że maksymalna głębokość rekurencji wynosi $\log_2 n$, gdzie n to rozmiar tablicy. Dla każdego elementu wykonywane jest porównanie i scalenie (merge), więc ostateczna złożoność obliczeniowa wynosi $O(\mathrm{nlog}_2\, n)$. Rozpatrując przypadek najgorszy można zauważyć, że zwiększa się jedynie ilość potrzebnych porównań podczas scalania elementów, więc nie powoduje to zmiany złożoności obliczeniowej. Aby móc swobodnie poruszać się w tym algorytmie po elementach potrzebujemy dodatkowej tablicy o takim samym rozmiarze jak nasza tablica do posortowania wiec złożoność pamięciowa dodatkowych struktur dla sortowania przez scalanie wynosi to O(n).

		10000	50000	100000	500000	1000000
0%	min	0,001	0,01	0,021	0,119	0,249
	max	0,003	0,014	0,026	0,139	0,268
	średnia	0,00194	0,01097	0,02278	0,12554	0,25625
25%	min	0,001	0,009	0,019	0,107	0,223
	max	0,003	0,017	0,033	0,124	0,241
	średnia	0,00173	0,0096	0,02023	0,11099	0,22768
50%	min	0,001	0,007	0,016	0,092	0,195
	max	0,002	0,01	0,022	0,101	0,208
	średnia	0,0015	0,00838	0,0176	0,09648	0,19939
75%	min	0,001	0,006	0,014	0,078	0,165
	max	0,003	0,008	0,017	0,093	0,177
	średnia	0,00133	0,00723	0,01517	0,08261	0,17062
95%	min	0,001	0,006	0,012	0,069	0,143
	max	0,004	0,008	0,015	0,078	0,154
	średnia	0,00119	0,00638	0,01332	0,07213	0,14857
99%	min	0,001	0,005	0,012	0,067	0,14
	max	0,002	0,007	0,015	0,075	0,16
	średnia	0,00118	0,00616	0,01302	0,07038	0,14496
99,70%	min	0,001	0,005	0,012	0,067	0,139
	max	0,002	0,008	0,015	0,078	0,158
	średnia	0,00114	0,00614	0,01285	0,06995	0,14431
rev	min	0,005	0,03	0,066	0,391	0,83
	max	0,006	0,042	0,073	0,432	0,899
	średnia	0,00543	0,03207	0,06872	0,39801	0,84271

Tab. 2. Wyniki pomiarów dla sortowania przez scalanie



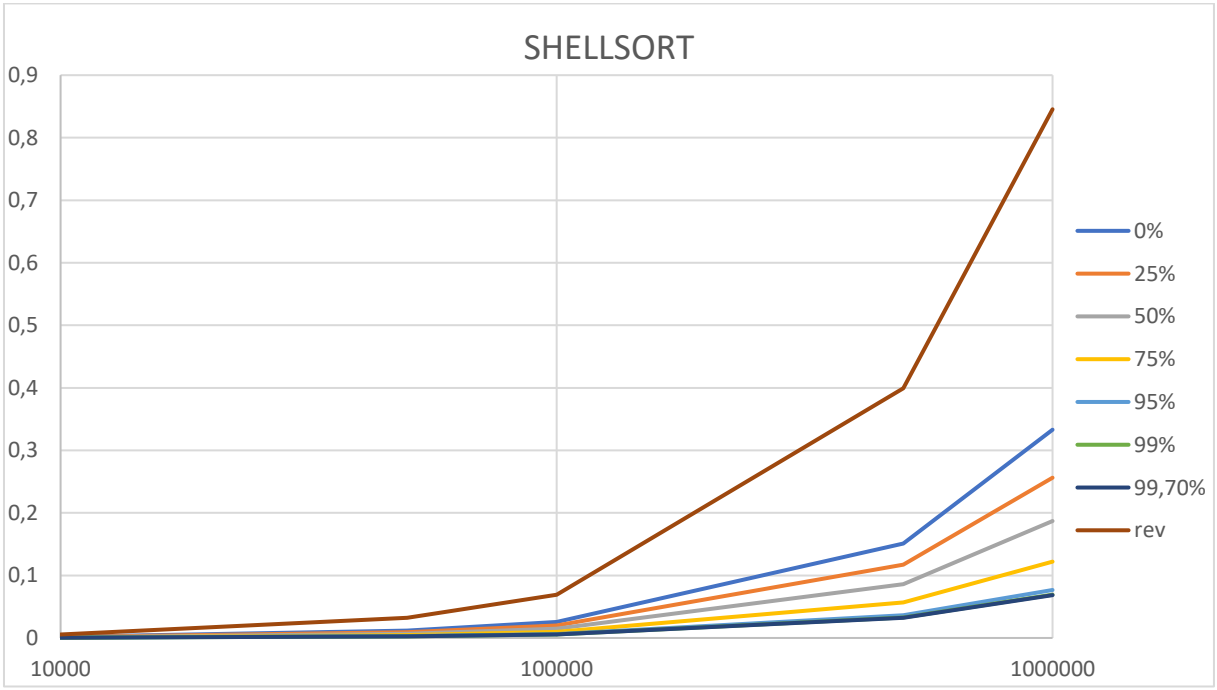
Rys. 2. Wykres zależności czasu od ilości elementów dla sortowania przez scalanie

4. Sortowanie Shella (Shell Sort)

Sortowanie Shella, inaczej nazywane sortowaniem przez wstawianie z malejącym odstępem, jeden z algorytmów sortowania działających w miejscu i korzystających z porównań elementów. Jest to tak naprawdę usprawnienie sortowania przez wstawianie. Różnica jest taka, że porównywane elementy (element i jego sąsiad) nie leżą obok siebie, lecz są oddalone o pewną odległość. Odległością zaproponowaną przez Shella jest $\frac{n}{2^k}$ gdzie n jest liczbą elementów, a k to numer kolejnej iteracji (jest to nic innego jak dzielenie dystansu przez dwa przy każdej iteracji). Odległość można też dobierać według kilku innych reguł, na przykład reguły Knutha $([3^k - 1]/2)$ czy Hibbarda $(2^k - 1)$. Po wybraniu w pierwszym etapie tego algorytmu dystansu, wykonujemy sortowanie przez wstawianie w każdej iteracji, zmniejszając odległość zgodnie z wybraną regułą aż do dystansu równego 1. Złożoność czasowa sortowania Shella zależna jest od wybranej reguły przyjmowanie odstępów. Każdy z nich charakteryzuje się inną złożonością obliczeniową. Nie da się określić dokładnie matematycznie średniej złożoności obliczeniowej Shell Sorta, jednak w zależności od wyboru dystansu może wynosić od $O(n^2)$ do $n * (\log n)^2$.

		10000	50000	100000	500000	1000000
0%	min	0,001	0,01	0,023	0,143	0,319
	max	0,003	0,014	0,029	0,158	0,359
	średnia	0,0018	0,01153	0,02538	0,1509	0,33301
25%	min	0,001	0,008	0,018	0,111	0,245
	max	0,002	0,01	0,023	0,125	0,269
	średnia	0,00153	0,00908	0,01974	0,11712	0,25626
50%	min	0,001	0,006	0,013	0,081	0,179
	max	0,002	0,009	0,016	0,092	0,198
	średnia	0,00107	0,00673	0,01465	0,08593	0,18696
75%	min	0	0,004	0,009	0,053	0,117
	max	0,001	0,006	0,011	0,061	0,127
	średnia	0,00068	0,0044	0,00964	0,05676	0,12203
95%	min	0	0,002	0,005	0,034	0,073
	max	0,001	0,004	0,008	0,042	0,081
	średnia	0,00056	0,00294	0,00621	0,03616	0,0767
99%	min	0	0,002	0,005	0,031	0,065
	max	0,001	0,004	0,007	0,037	0,074
	średnia	0,00047	0,00273	0,00576	0,03277	0,06921
99,70%	min	0	0,002	0,005	0,03	0,064
	max	0,001	0,004	0,007	0,036	0,073
	średnia	0,00043	0,00262	0,00569	0,03231	0,06846
rev	min	0,005	0,03	0,065	0,393	0,832
	max	0,007	0,036	0,074	0,415	0,864
	średnia	0,00554	0,03206	0,06909	0,39935	0,8455

Tab. 3. Wyniki pomiarów dla sortowania Shella



Rys. 3. Wykres zależności czasu od ilości elementów dla sortowania Shella

5. Specyfikacja komputera, na którym prowadzono pomiary

   Procesor: Intel Core i5-7300HQ (2.5 GHz, 3.5 GHz Turbo, 6 MB Cache)
   Ram: 8GB

6. Wnioski

   Wszystkie algorytmy są szybkie lecz z całej trójki zdecydowanie najlepiej wypadł Merge Sort, a tuż za nim Shell Sort. Zdecydowanie wolniejszy okazał się Heap Sort, lecz może to wynikać z nie do końca optymalnej implementacji jego algorytmu.
   Każdy z algorytmów sortuje tym szybciej im ta tablica jest mniejsza oraz im bardziej jest uprzednio posortowana, co jest oczekiwanym wynikiem.
   Jeśli chodzi o tablice posortowane w odwrotnej kolejności, to w tym wypadku najlepsze czasy uzyskuje Heap Sort, w pozostałych dwóch rodzajach sortowań taka konfiguracja tablicy była największym problemem.

7. Literatura

   http://www.algorytm.org/algorytmy-sortowania/

   https://pl.wikipedia.org/wiki/Sortowanie_Shella

   https://pl.wikipedia.org/wiki/Sortowanie_przez_kopcowanie

   https://pl.wikipedia.org/wiki/Sortowanie_przez_scalanie

   https://eduinf.waw.pl/inf/alg/003_sort/index.php