POP projekt

Dokumentacja kocńcowa

Kacper Maj

Łukasz Topolski

1. Opis problemu

Mamy macierz prostokątną o wymiarach $4xN$ $(N \in \mathbb{N})$. Każdy z jej elementów jest liczbą całkowitą. Oprócz niej dostępnych jest m kart. Za pomocą nich można zakryć komórki macierzy, przy czym dwa zakryte elementy nie mogą być obok siebie (zarówno w pionie, jak i w poziomie). Zadanie polega na zakryciu komórek tak, żeby suma pozostałych elementów była jak największa. Nie ma przymusu wykorzystywania wszystkich kart.

Największą możliwą sumę można osiągnąć przy usunięciu wszystkich ujemnych liczb, dlatego też problem ten sprowadza się do zakrycia m najmniejszych liczb ujemnych tak, aby warunki zadania zostały spełnione. Ponadto z treści można wywnioskować, że liczba użytych kart jest z przedziału $[0, 2N]$ (w jednej kolumnie mogą być zakryte co najwyżej 2 komórki).

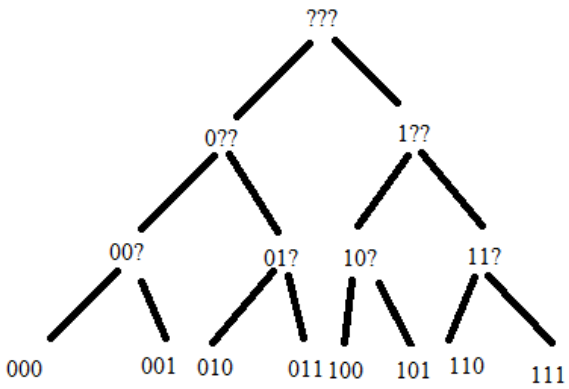
2. Propozycja rozwiązania

Do wyznaczenia najoptymalniejszego rozwiązania zostanie wykorzystany algorytm A* wraz z heurystyką. Przestrzeń przeszukiwań to wektory składające się z zer i jedynek, gdzie 1 oznacza, że i-ta liczba ujemna została zakryta, a 0 – nie została zakryta. Pierwsza wartość jest wybierana losowo, a kolejne to następne liczby z tego samego wiersza. Gdy i-ta wartość jest ostatnią z wiersza, to wartością i+1-tą jest kolejna liczba z następnego wiersza. Gdy jednak i-ta liczba jest ostatnią z ostatniego wiersza, to i+1-ta wartość to następna liczba z pierwszego wiersza. Dla przykładowej macierzy z rysunku 1. i wartości -6 jako pierwszy element wektor zer i jedynek wygląda następująco: -4 odpowiada drugiej pozycji wektora, -1 odpowiada czwartej pozycji, a -5 – ósmej.

-5	-6	8	-4
-10	10	-1	15
0	-2	1	7
2	30	-7	-8

Rys. 1.: przykładowa macierz

Przykładowe drzewo przeszukiwań zostało zaprezentowane na rys. 2. (? – nie wiemy czy liczba i-ta została wybrana)



Rys. 2.: drzewo przeszukiwań dla 3 liczb ujemnych

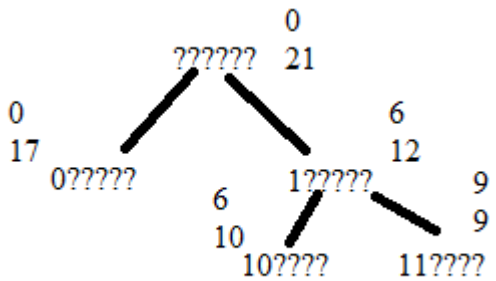
Funkcją celu jest moduł sumy (albo suma modułów – wykonujemy działania na liczbach o tym samym znaku) wybranych liczb. Funkcja heurystyczna to suma modułów co najwyżej $m_{max} - m_u$ najmniejszych (ujemnych) liczb, które potencjalnie mogą być zakryte ($m_u -$ liczba zakrytych komórek, $m_{max} -$ maksymalna możliwa liczba zakrytych komórek, $m_{max} = m$, gdy $m \leq 2N$ lub $m_{max} = 2N$, gdy $m > 2N$). W tej funkcji nie są brane pod uwagę elementy, które na pewno nie mogą być wybrane oraz te, które zostały już zakryte.

Poniżej znajduje się przykład przedstawiający kolejne kroki algorytmu:

Przykładowa plansza jest na rysunku 3., liczba kart = 4. Maksymalna możliwa liczba użytych kart wynosi 4. Pierwszym elementem niech będzie -6 (trzecia kolumna, drugi wiersz). Wartość funkcji celu wynosi 6 oraz funkcji heurystycznej – 12 (7+2+3) jeśli zakryjemy tę komórkę. Gdy nie zakryjemy, to wart. funkcji celu wynosi 0, a funkcji heurystycznej – 17 (7+5+2+3). Suma funkcji celu i funkcji heurystycznej jest większa przy wybraniu elementu -6, więc go zakrywamy. Kolejnym rozpatrywanym elementem będzie -3 (z pierwszej kolumny i ostatniego wiersza). Przy zakryciu go funkcja celu ma wartość 9, a funkcja heurystyczna – 9 (7 + 2). Natomiast gdy nie zostanie zakryty, to wart. funkcji celu wynosi 6, a funkcji heurystycznej – 10 (7 + 2 + 1). W tym przypadku element -3 zostaje zakryty. Analogicznie są wykonywane kolejne kroki aż dojdziemy do ostatniej liczby ujemnej lub gdy zakryjemy 4 liczby. Fragment drzewa przeszukiwań dla omówionego przypadku znajduje się na rysunku 4. (przy węźle drzewa pierwsza liczba oznacza wartość funkcji celu, a druga – wartość funkcji heurystycznej)

-1	8	-5	0
-7	3	-6	12
2	5	1	0
-3	7	6	-2

Rys. 3.: przykładowa tablica



Rys. 4.: Fragment drzewa przeszukiwań

3. Planowe eksperymenty numeryczne

W ramach testów planowane jest kilkukrotne uruchomienie algorytmu dla jednej macierzy i jej transpozycji z różnymi elementami startowymi i porównanie wyników (tzn. sum wszystkich elementów niewybranych tablicy).Również do porównania zostanie uwzględniona suma wszystkich wartości niezakrytych przy wybraniu co najwyżej m najmniejszych ujemnych liczb.

4. Wybrana technologia

Wybranym językiem programowania jest Python wraz z biblioteką NumPy.

5. Opis implementacji

Rozwiązanie składa się z następujących plików:

- point.py
- table.py
- A_algorithm.py
- priority_queue.py
- naive_algorithm.py

5.1. Point.py

W tym module znajduje się klasa reprezentująca element planszy. Ta reprezentacja składa się z wartości oraz określeniem pozycji w tablicy. Ponadto ta klasa zawiera metody zwracające współrzędne elementu oraz pozycje jego sąsiadów.

5.2. Table.py

Ten moduł jest odpowiedzialny za wygenerowanie nowej planszy o określonych wymiarach, zwróceniu listy ujemnych punktów (reprezentacji elementów – zob. 6.1.) oraz zwróceniu wyniku – zarówno jako sumy wszystkich elementów jak i reprezentację planszy z zakrytymi elementami (wartości do zakrycia są zastępowane zerami)

5.3. Naive_algotithm.py

Jest to implementacja prostego algorytmu, który polega na uporządkowaniu elementów rosnąco i zakryciu co najwyżej ujemnych liczb tak, aby warunki zadania zostały spełnione. Argumentami wejściowymi są: plansza wygenerowana przez moduł *table* oraz liczba dostępnych kart. Wynikiem działania algorytmu jest lista współrzędnych elementów do zakrycia

5.4. Priority_queue.py

Ten plik zawiera implementację prostej kolejki priorytetowej. Jest to zwykła lista, która jest sortowana po każdym dodaniu nowego elementu. W kolejce są krotki składające się z sumy wartości funkcji celu i funkcji heurystycznej oraz wektora binarnego określającego które elementy są zakryte.

5.5. A_algorithm.py

W tym pliku znajduje się implementacja algorytmu A* z heurystyką. Argumentami wejściowymi są: plansza (wygenerowana przez moduł *table*), indeks początkowego elementu oraz liczba dostępnych kart. W wyniku działania algorytmu otrzymywana jest lista elementów (obiektów klasy Point), które powinny być zakryte. Dodatkowo w tym pliku znajduje się funkcja przerabiająca listę elementów na listę współrzędnych.

Schemat algorytmu jest następujący:

1. Wyznacz liczby ujemne z tablicy (lista obiektów Point)
2. Zmień kolejność tych liczb w zależności od początkowego indeksu
3. Do kolejki priorytetowej dodaj krotkę składającą się z pustej listy i wartości funkcji heurystycznej przy żadnym niezakrytym elemencie
4. Dopóki kolejka nie jest pusta:
    1. Pobierz wektor binarny z kolejki.
    2. Jeśli jego długość jest równa liczbie ujemnych elementów to przerwij pętlę
    3. Jeśli element nie może być zakryty, to do wektora binarnego dodaj 0 i do kolejki dodaj zestawienie sum wartości funkcji heurystycznej i funkcji celu oraz wektora binarnego
    4. W przeciwnym przypadku wydłuż wektor binarny raz o 0 i raz o 1. Dla każdego z tych wydłużeń wyznacz sumy wartości funkcji heurystycznej i funkcji celu. Na koniec dodaj do kolejki zestawienia wektorów binarnych wraz z odpowiadającymi im sumami.

6. Działanie implementacji algorytmu heurystycznego

Poniższy rysunek przedstawia przykładową tablicę.

5	20	8	-15
3	4	7	-6
-9	-12	10	7
1	-5	3	9

Rys. 5.: Przykładowa plansza

W wyniku działania algorytmu (przy liczbie startowej -12 oraz liczbie kart=4) otrzymana tablica jest następująca (zerami oznaczone są zakryte wartości - takie oznaczenie ma taki sam wpływ na sumę wszystkich elementów jak zwyczajne odrzucenie tych liczb):

5	20	8	0
3	4	7	-6
0	-12	10	7
1	0	3	9

Rys. 5.: Przykładowa plansza

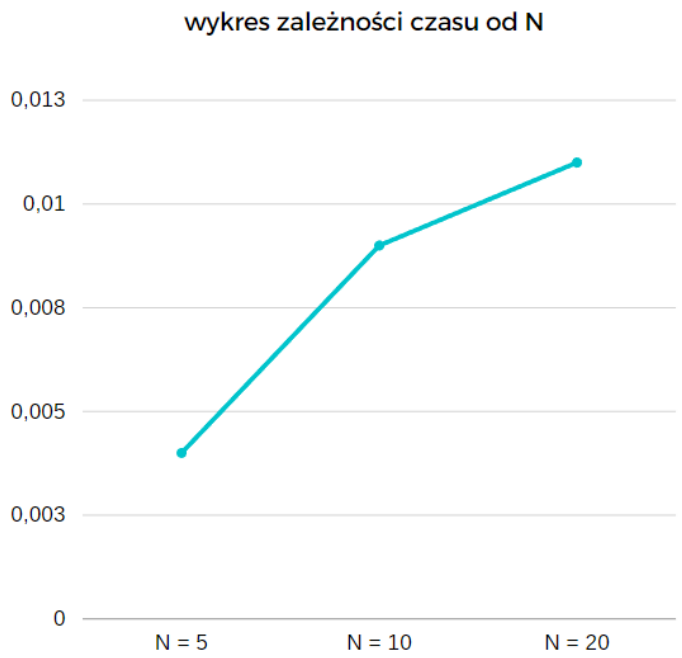
Okazuje się, że jest to najoptymalniejsze rozwiązanie. (zakrycie -12 powoduje, że nie można zakryć -9 i -5, które sumarycznie są mniejsze).

Inne przykłady prezentujące działanie algorytmu znajdują się w pliku *test_algorithms.py*.

7. Eksperymenty numeryczne

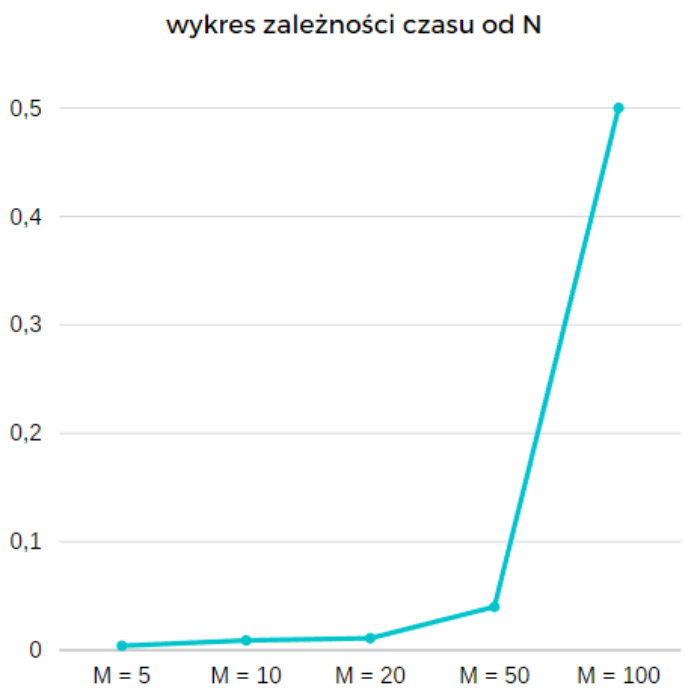
Jak wielkość tablicy i ilość kart wpływa na czas wykonania programu:

Eksperymenty zostały przeprowadzone na domyślnym ziarnie: 420, każda liczba jest średnią arytmetyczną wyciągniętą z 30 testów. Wszystkie testy są zapisane w pliku tekstowym „testy.txt" i korzystając z pliku „testy.py" i odpowiedniego ziarna mogą być w każdej chwili zreprodukowane.

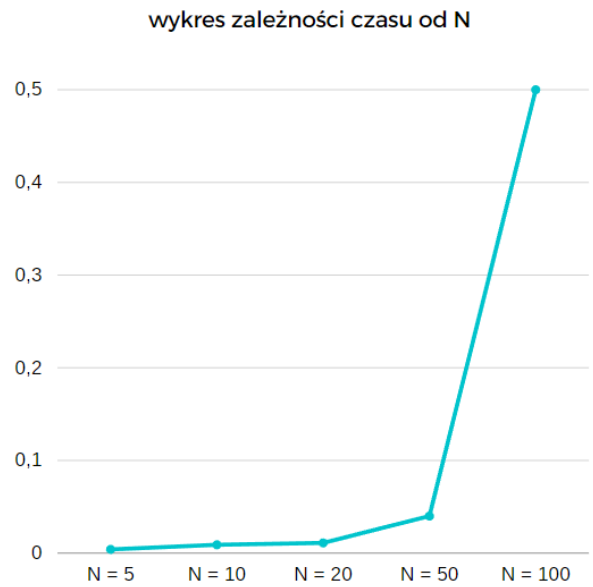


Jak widać na wykresie wielkość tablicy ma wpływ na czas wykonywania programu. Stosunek wzrostu czasu wykonania w zależności od wielkości tablicy może wydawać się liniowy, lecz późniejsze testy z większymi tablicami pokazują, że czas rośnie wykładniczo (widoczne na

wykresie poniżej). Mylny liniowy wzrost może być spowodowany bardzo szybkim czasem wykonania algorytmu lub występować u małych tablic.



Znacznie większy wpływ na czas wykonania ma natomiast ilość kart. Testy przeprowadzone na tablicy o wielkości N = 20 (widoczne na wykresie poniżej, M – ilość kart) pokazują, że czas potrzebny na wykonanie algorytmu rośnie znacząco w zależności od M. Jest to spowodowane tym, że algorytm A* stara się wyczerpująco przeszukać możliwe rozwiązania, a podwajając liczbę kart za każdym razem znacznie zwiększamy ilość teoretycznych możliwości.



Czas działania algorytmu A* w zależności od punktu startowego (liczby ujemnej):

liczba ujemna nr 0: 29.7882022857666

liczba ujemna nr 1: 16.467875957489014
liczba ujemna nr 2: 11.68854308128357
liczba ujemna nr 3: 6.549646615982056
liczba ujemna nr 4: 3.4405956268310547
liczba ujemna nr 5: 2.288400411605835
liczba ujemna nr 6: 2.8715016841888428
liczba ujemna nr 7: 2.5249404907226562
liczba ujemna nr 8: 2.0308542251586914
liczba ujemna nr 9: 1.172206163406372
liczba ujemna nr 10: 1.1461994647979736
liczba ujemna nr 11: 2.95401668548584
liczba ujemna nr 12: 1.4367499351501465
liczba ujemna nr 13: 0.9566669464111328
liczba ujemna nr 14: 0.9561672210693359
liczba ujemna nr 15: 0.8296446800231934
liczba ujemna nr 16: 4.210236072540283
liczba ujemna nr 17: 19.024822473526
liczba ujemna nr 18: 28.516480684280396
liczba ujemna nr 19: 25.868518114089966
liczba ujemna nr 20: 26.0565505027771
liczba ujemna nr 21: 23.46109628677368
liczba ujemna nr 22: 122.86445808410645
liczba ujemna nr 23: 122.15333437919617
liczba ujemna nr 24: 289.39653968811035
liczba ujemna nr 25: 222.34133005142212
liczba ujemna nr 26: 162.23883366584778
liczba ujemna nr 27: 113.65434980392456
liczba ujemna nr 28: 146.22653675079346
liczba ujemna nr 29: 83.49308037757874
liczba ujemna nr 30: 100.58556652069092
liczba ujemna nr 31: 204.3651909828186
liczba ujemna nr 32: 170.69230914115906
liczba ujemna nr 33: 130.9473683834076
liczba ujemna nr 34: 146.43307447433472
liczba ujemna nr 35: 60.92013764381409
liczba ujemna nr 36: 52.22111988067627

Jak widać czas wykonania jest bardzo zależny od przyjętego punktu startowego. Najmniejszy wynik wynosi: 0.83 dla nr 15, a największy aż 289,4 (wyniki obecne w „testy.txt).

Czy opłaca się stosować algorytm A* w porównaniu z naiwnym?

Wyniki (końcowa sumaryczna wartość tablicy) dla N = 10:

Ilość kart (M):	Algorytm A*	Algorytm naiwny
5	7	7
10	165	165
20	187	184

Wyniki dla N = 20 (test M = 20 przeprowadzony dla tylko jednej liczby w algorytmie A z powodu czasochłonności):

Ilość kart (M):	Algorytm A*	Algorytm naiwny
5	301	301
10	506	506
20	734	730
40	738	737

Jak widać w większości przypadków wyniki są identyczne. W poprzednich punktach został opisany przypadek w którym algorytm zachłanny utyka w minimum lokalnym, i testując algorytmy mamy potwierdzenie, że algorytm A* jest pod tym względem lepszy. Zwiększając liczbę kart zwiększamy liczbę możliwości, w których opisany układ liczb może wystąpić i dlatego stosowanie algorytmu A* jest tym bardziej opłacalne im więcej kart posiadamy. Co ciekawe dzięki zastosowaniu A* zamiast A wyniki w algorytmie nie są zależne od przyjętego punktu startowego (pokazane w pliku „testy_wyników.txt" możliwe do zreprodukowania przy użyciu „testy_wyników.py").

Jednak jeśli spojrzymy na czasy wykonania dla N = 10 (w algorytmie A* czasy są pokazane min – max):

Ilość kart (M):	Algorytm A*	Algorytm naiwny
5	0 - 0	0
10	0 – 0,03	0
20	0,03 – 0,2	0

I dla N = 20 (M = 40 miało tylko jeden test):

Ilość kart (M):	Algorytm A*	Algorytm naiwny
5	0 - 0	0
10	0 - 0,01	0
20	0,37 - 139,59	0
40	216,22	0

Wyniki także obecne w pliku „testy_wyników".

8. Wnioski

1. Stosowanie zaimplementowanego algorytmu A* jest korzystne jeśli chcemy osiągnąć najoptymalniejsze rozwiązanie.
2. Stosowanie algorytmu jest kosztowne czasowo i obliczeniowo.
3. Punkt startowy nie wpływa na wynik algorytmu, lecz od niego zależy czas pracy (ilość wykonanych obliczeń).
4. Nasza implementacja algorytmu nie jest najoptymalniejsza. Można ją poprawić np. stosując jako kolejkę priorytetową kopiec zamiast listy sortowanej przy każdym dodaniu elementu.