Usuń i wygraj - projekt AAL
Dokumentacja końcowa

Piotr Zmyślony, nr albumu 268833

1 Opis problemu

1.1 Treść zadania

Gra „Usuń i wygraj" polega na uzyskaniu jak największego wyniku przy sukcesywnym usuwaniu liczb z (ustalonej przed rozpoczęciem gry) listy liczb całkowitych N, dopóki nie pozostanie nam zbiór pusty. Operacja usunięcia liczby *N[i]* powoduje uzyskanie liczby punktów równej wybranej liczbie. Po usunięciu jednej liczby, wszystkie dwie sąsiednie (*N[i]-1* oraz *N[i]+1*) liczby, jeśli istnieją, również są usuwane - tym razem bez uzyskiwania jakichkolwiek punktów.

Problem polega na znalezieniu maksymalnej liczby punktów do zdobycia dla konkretnej listy.

1.2 Analiza

1.2.1 Warianty listy

Pierwszym krokiem jest posortowanie listy wstępnej rosnąco, co znacząco ułatwi dalszą analizę zagadnienia, stąd dalej przedstawione listy będą już posortowane.

Przypadkiem trywialnym problemu jest taka lista unikalnych liczb, w której żadna z liczb nie posiada sąsiadów, na przykład *N1 = [-3,-1,1,3,5,7]*. W takim wypadku kolejność wyboru liczb nie ma znaczenia, ponieważ nigdy nie „tracimy" punktów poprzez usuwanie.

Przypadkiem ogólnym jest lista, w której liczby mogą posiadać sąsiadów, na przykład *N2 = [-10,-9,-8,1,2,3,4,5,7,8,9]*. Teraz musimy rozważać rozłączne, rosnące listy liczb sąsiadujących. Lista *N2* rozkłada się na listy: *[-10,-9,-8]*, *[1,2,3,4,5]*, *[7,8,9]*. Wynikiem jest suma maksymalnych ilości punktów, jaką można uzyskać w każdej z podlist.

1.2.2 Problem duplikatów

Problem duplikatów, czyli liczb występujących w liście po klika razy, można rozwiązać sumując wszystkie wystąpienia i zastępując je tą właśnie sumą. Na przykład dla *N3 = [1,1,2,3,3,3,4,4,4,6,7,7,8]* dostajemy: *[2,2,9,12]*, *[6,14,8]*. Dla tych list, operacja usuwania zmienia się nieznacznie - usunięcie liczby *N[i]* powoduje usunięcie *N[i-1]* i *N[i+1]*.

2 Algorytm

2.1 Przetwarzanie wstępne danych

Na wstępie algorytm dostaje listę liczb całkowitych i tworzy rozłączne listy sąsiadujących i posortowanych rosnąco liczb. Na każdej z nich dokonywana jest operacja usuwania duplikatów opisana powyżej. Wynik sumaryczny właściwego algorytmu (opisanego poniżej) dla każdej z list jest odpowiedzią na to, jaki jest maksymalny możliwy do uzyskania wynik na liście początkowej.

2.2 Działanie algorytmu

Na początku algorytm dokonuje sprawdzenia, jakiego typu jest lista, czyli porównuje pierwszą (min) i ostatnią (max) liczbę w liście N

- **Typ A** - $min \geqslant 0$
- **Typ B** - $min < 0$

2.2.1 Rozwiązanie dla typu A

Jest to najprostszy z przypadków, którego rozwiązanie ma złożoność $O(n)$. Służy do tego poniższy algorytm:

Algorytm House Robber:

```
def house_robber(array):
    incl = 0
    excl = 0

    for i in array:
        new_excl = excl if excl >= incl else incl
        incl = excl + i
        excl = new_excl
    return excl if excl > incl else incl
```

2.2.2 Rozwiązanie dla typu B

W tym wypadku algorytm powyższy (nawet z modyfikacjami) nie jest w stanie policzyć największej wartości.

W pierwszym kroku, sprawdzamy czy lista posiada również wartości większe bądź równe 0. Jeśli nie, wynik daje nam rekurencyjny *Algorytm I* opisany pokrótce poniżej.

Jeżeli tak - część rozwiązania dla liczb nieujemnych możemy dostać poprzez algorytm *House Robber*. Istnieją 3 możliwe sposoby podziału takiej listy: usuwamy 0, usuwamy liczbę na lewo od 0, usuwamy liczbę na prawo od 0. Dla każdego z tych podziałów część dodatnią rozwiązujemy algorytmem *House Robber*, a część ujemną *Algorytmem I*. Z tych 3 wyników wybieramy największy jako końcowy.

Algorytm I:

```
1. wynik := 0
2. Sprawdź które liczby mają maksymalny BU*.
3. Jeśli jest tylko jedna liczba M[i] o największym BU,
   wynik += M[i], usuń M[i-1], M[i] i M[i+1] i przejdź do kroku 2.
4. Utwórz tablicę max[n], gdzie n to ilość liczb o maksymalnym BU.
5. Dla każdego i=0..(n-1) kolejno:
 5.1. z := rezultat Algorytmu I dla listy bez M[i] i sąsiadów
 5.2. max[i] := wynik + M[i] + z.
6. wynik += maksymalna wartość z tablicy max[]
7. Zwróć wynik.

*Bilans Usunięcia dla liczby N[i] równy jest N[i]-N[i-1]-N[i+1].
```

2.3 Złożoność

2.3.1 Złożoność obliczeniowa

Złożność algorytmu *House Robber* jest liniowa, wymaga jedynie sortowania w pre-processingu danych. Stąd dla samych liczb dodanich złożonosc obliczeniowa wynosi *O(nlogn)* Formuła ta pozwala na otrzymanie przybliżonej długości łuku, który łączy dwa punktu na kuli, używając jako dane wejściowe szerokości i długości geograficznych obu punktów.

$$d = 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos(\varphi_1)\cos(\varphi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

W powyższym wzorze r to promień Ziemi, φ i λ to odpowiednio szerokości i długości geograficzne.

3 Implementacja

3.1 Program

Skrypt zawierający algorytmy przyjmuje 4 argumenty :
<graf> <tryb> <początek> <cel>
przy czym tryby pracy programu są następujące:

1. alg. brutalny (przeszukiwanie wszystkich możliwych ścieżek)
2. algorytm A*
3. algorytm Dijkstry
4. A* oraz algorytm Dijkstry
5. Wszystkie 3 algorytmy
6. Tryb testowy, wymaga jedynie podania argumentów w postaci *<graf> 6 <ilość prób testowych>*. Porówuje wydajności A* oraz algorytmu Dijkstry na podstawie określonej ilości danych testowych (generowanych automatycznie).

Wynikiem działania dla wszystkich opcji są ścieżki, jej koszt (odległość) oraz czas działania. Czas działania jest mierzony tylko w momencie działania algorytmu. Funkcja heurystyczna korzysta z formuły haversine obliczającą odległość w linii prostej po powierzchni sfery pomiędzy miastami. Funkcja ta spełnia wymagania heurystyki w algorytmie A* gdyż stanowi dolne ograniczenie odległości pomiędzy miastami *[nie da się dotrzeć szybciej niż w linii prostej]*

3.2 Testy działania

Poniższa tabelka przedstawia czasy działania algorytmu dla jednej procesu wyszukiwania jednej ścieżki, uśrednione na podstawie 10000 powtórzeń dla A* i algorytmu Dijkstry.

Uśrednione wyniki		
Algorytm	średni czas dla grafu Polska	średni czas dla Germany50
Brutalny	375 μs	>15 minut
Dijkstra	19 μs	87 μs
A*	25 μs	41 μs

3.3 Analiza wyników

Algorytm Dijkstry oraz *A** są zdecydowanie szybsze od podejścia brutalnego, którego użycie dla grafów o większej ilości krawędzi i wierzchołków może prowadzić do złożoności *O(n!)*. Dodatkowo dzięki zastosowaniu algorytmu heurystycznego algorytm *A** sprawdza jedynie potencjalnie najlepsze ścieżki co ogranicza *rozprzestrzenianie* się algorytmu i zdecydowanie przyspiesza jego pracę dla odpowiednio dużych grafów. Jako, że funkcja haversine jest umiarkowanie skomplikowaną formułą to dla mapy Polski z 12 miastami i 18 krawędziami wzrost wydajności jest znikomy, ale już dla bardziej realistycznego zastosowania, w grafie dla mapy Niemiec, o 50 wierzchołkach znajduje dłuższe trasy szybciej.