Algoritmi Aproximativi

Knapsack

1. Fie S un șir de numere naturale { s1, s2, ... , sn } și K un număr na-tural, cu K ≥ si pentru orice i între 1 și n.
2. def sumaMax(S, K):  
    sumaMax = 0  
    sumePartiale = set([0])  
    for Si in S:  
    # print('\n', sumePartiale)  
    for sumaPartiala in sumePartiale.copy():  
    # print(Si, sumaPartiala)  
    if Si + sumaPartiala <= K:  
    sumaMax = max(sumaMax, Si + sumaPartiala)  
    sumePartiale.add(Si + sumaPartiala)  
    return sumaMax

Complexitate de timp – O(N\*K), unde N = len(S)

Complexitate de spatiu – O(N)

Functia sumaMax implementeaza un algoritm de programare dinamica care exploreaza toate combinatiile posibile de sume partiale pana la pasul current si le memoreaza o singura data. Pentru fiecare număr Si din lista S, se adaugă la sumele parțiale toate combinațiile posibile dintre numărul Si și sumele parțiale existente în acel moment pentru a alege suma maxima, cat mai optim.

def sumaMaxAprox(file):  
 sumaMax = 0  
 K = int(file.readline())  
 print(K)  
 while True:  
 Si = file.readline()  
 if Si == '':  
 break  
   
 Si = int(Si)  
 if Si + sumaMax <= K:  
 sumaMax += Si  
 elif sumaMax < Si:  
 sumaMax = Si  
 return sumaMax

Complexitate de timp – O(N)

Complexitate de spatiu – O(1)

Functia sumaMaxAprox calculeaza o suma maxima aproximativa, fara a memora sirul S si parcurgandu-l o singura data. Algoritmul insumeaza elementele fara a depasi valoarea K, iar In momentul in care suma depaseste valoarea K se verifica restul sirului pentru a gasi un numar mai mare decat suma curenta.

Travelling Salesman Problem

1. Considerăm o variantă a TSP, în care toate muchiile au ponderea 1 sau 2.
2. Arătați că problema rămâne NP-hard pentru aceste instanțe.

Fie graful G1(N, M), unde N este numarul de noduri si M numarul de muchii. Conform cerintei construim graful G2(N, M2) in care costul muchiilor se calculeaza astfel:

* 1, daca muchia nu exista in graful G1
* 2, daca muchia exista in graful G2

Astfel obtinem graful cerut in care toate muchiile au ponderea 1 sau 2 pe care putem aplica algoritmul TSP. Singurul caz in care solutia sa fie egala cu N (suma minima a costurilor muchiilor parcurse de voiajor = N = > parcurge doar muchii existente in G1) este atunci cand graful este un graf Hamiltonian. Atunci, problema se reduce la problema determinarii unui ciclu Hamiltonian, care este, de asemenea, o problema NP-Hard.

Deci, problema TSP ramane NP-Hard pentru un graf in care toate muchiile au ponderea 1 sau 2.

1. Arătați că aceste ponderi satisfac în continuare inegalitatea triunghiului.

Inegalitatea triunghiului - suma oricăror două laturi ale unui triunghi este întotdeauna mai mare sau cel putin egala cu lungimea celei de-a treia laturi

1. {1, 1, 1} – 1 <= 1 + 1
2. {2, 1, 1} – 2 <= 1 + 1; 1 <= 2 + 1
3. {2, 2, 1} – 2 <= 2 + 1; 1 <= 2 + 2
4. {2, 2, 2} – 2 <= 2 + 2

Intrucat se indeplinesc aceste proprietati, se satisfice inegalitatea triunghiului.

1. Algoritmul descris în curs (cursul 3, slides 18-19) oferă o aproximare de ordin 2 pentru forma generală a TSP (pentru instanțele care respectă inegalitatea triunghiului). Verificați dacă în această instanță a problemei, algoritmul din curs este 3/2-aproximativ.

ApproxTSP(G)

1: Calculam arborele partial de cost minim T pentru graful G.

2: Alegem un nod u ∈ T pe post de radacina.

3: Γ=Ø.

4: Parcurgere (u, Γ)

1: Concatenam pe u la Γ.

2: pentru fiecare v, fiu al lui u:

3: Parcurgere(v, Γ)

5:concatenam nodul u la finalul lui Γ pentru a inchide un ciclu .

6: return Γ

Algoritmul descris în pseudocodul de mai sus construiește un arbore parțial de cost minim T folosind un algoritm precum Prim sau Kruskal. Apoi, algoritmul parcurge arborele T recursiv pentru a construi un ciclu hamiltonian prin adăugarea fiecărui nod de la fiecare nivel de adâncime. În final, se returnează acest ciclu.

Fie graful complet G(N, M) si costul tuturor muchiilor 1. Arborele partial de cost minim al grafului G va contine n-1 muchii, care vor fi parcurse de 2 ori: o data cand se construieste APM-ul si o data cand acesta este parcurs pentru a construe ciclul hamiltonian.

Prin urmare putem face verificarea 2\*(n-1) <= (3/2) \* n, care este falsa pentru orice n > 4, deci algoritmul descris nu poate fi 3/2-aproximativ.

2\*(n-1) = 2n – 2

(3/2)\*n = 3n/2

* 2n -2 <= 3n/2

⬄ 4n – 4 <= 3n

⬄ n <= 4

Vertex Cover

Fie X = { x1, x2, ... , xn } o mulțime de variabile boolene. Numim formulă booleană (peste mulțimea X) în Conjunctive Normal Form (CNF) o expresie de forma C1 ∧ C2 ∧ ⋯ ∧ Cm unde fiecare predicat (clauză) Ci este o disjuncție a unui număr de variabile (este alcătuit din mai multe variabile cu simbolul „∨” - logical or - între ele).

Exemplu de astfel de expresie:

(x1 ∨ x3 ∨ x4) ∧ (x2 ∨ x3 ∨ x7) ∧ (x1 ∨ x5 ∨ x6) ∧ (x2 ∨ x5 ∨ x7)

Evident că orice astfel de expresie va fi evaluată ca true dacă toate elementele lui X iau valoarea true. Ne interesează în schimb să aflăm numărul minim de elemente din X care trebuie să aibă valoarea true astfel încât toată expresia să fie true.

Fie următorul algoritm pentru problema de mai sus în varianta în care fiecare clauză are exact trei variabile (numită 3CNF):

Greedy-3CNF

1. Fie C = { C1, ... , Cm } mulțimea de predicate, X = { x1, ... , xn } mulțimea de variabile.

2. Cât timp C ≠ ∅ execută:

(a) Alegem aleator Cj ∈ C.

(b) Fie xi una dintre variabilele din Cj.

(c) xi ← true

(d) Eliminăm din C toate predicatele care îl conțin pe xi

3. Soluția constă din variabilele pe care le-am setat ca true pe parcursul execuției algoritmului

1. Analizati factorul de aproximare (worst case) al algoritmului.

In cel mai rau caz (worst case), fiecare clauza ar putea avea variabile unice, adica toate cele n variabile sunt diferit. In acest caz, pentru a obtine evaluarea expresiei ca fiind adevarata, ar trebui sa setam fiecare dintre cele n variabile adevarate. Deci, numarul minim de variabile necesare pentru a obtine o evaluare a expresiei CNF ca adevarata poate fi de cel mult n.

Factorul de aproximare al algoritmului este n.

1. Modificați algoritmul de mai sus, astfel încât acesta să fie un algoritm 3-aproximativ pentru problema inițială (și justificați).

Algoritm modificat:

1. Fie C = { C1, ... , Cm } mulțimea de predicate, X = { x1, ... , xn } mulțimea de variabile.
2. Cât timp C ≠ ∅ execută:

(a) Alegem aleator Cj ∈ C.

(b) Fie xi una dintre variabilele din Cj.

(c) xi ← true, unde i = 1, 3

(d) Eliminăm din C toate predicatele care contin variabile din Cj

1. Soluția constă din variabilele pe care le-am setat ca true pe parcursul execuției algoritmului

In loc sa selectam o singura variabila din predicatul Cj, le selectam pe toate si stergem toate predicatele care contin una dintre cele 3 variabile ale lui Cj. Astfel, sansele de a parcurge mai multe predicate au scazut drastic. In cel mai rau caz, algoritmul selecteaza 3 elemente pentru fiecare pas.

Factorul de aproximare al algoritmului modificat este cel putin n-aproximativ.

1. Reformulați problema de mai sus sub forma unei probleme de programare liniară (cu numere reale).

Fie C = {C1, C2, ..., Cm} si X = {x1, x2, ..., xn} cu urmatoarele proprietati:

* Xi = {0, 1} // True or False
* Xi + Xj + Xk >= 1, unde Xi, Xj, Xk apartin Cj// cel putin o variabila True

Minimizeaza functia sum(x1 + x2 + ... +xn)