Изчисляване на грациозни дървета

1 Въведение

Дадено е дърво (V, E) с n върха. Нека V $\equiv J_n$, където $J_n \coloneqq \{i \in \mathbb{N} \mid 0 \leq i \leq n\}$. Тоест всеки връх ще представлява индекс в масив. Сега може да въведем стойности на върховете. Това са числата $w_0 \ldots w_{n-1}$, като стойността на върхът i е w_i. Налагаме следните ограничения върху тях:

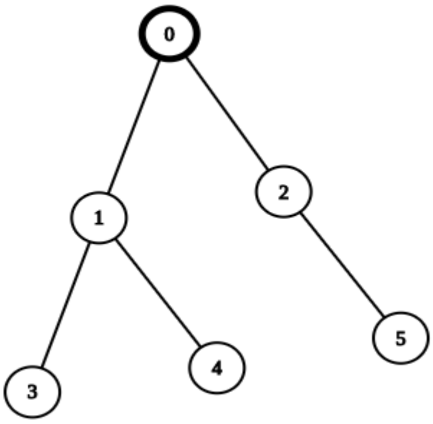
1. $\forall i \in J_n : 1 \leq w_i \leq 2n - 1$
2. $\forall i \in J_n : w_i \equiv 1 \pmod 2$
3. $\forall i, j \in J_n : i \neq j \implies w_i \neq w_j$
4. $\forall e_1, e_2 \in E : e_1 \neq e_2 \implies h(e_1) \neq h(e_2)$,
   където $h(\{i, j\}) \coloneqq \|i - j\|$.

Ако съществуват $w_0 \ldots w_{n-1}$, за която са изпълнени горните условия, казваме, че дървото е грациозно.

1.1 Четене на дървото от файл

На първия ред пише n, броя върхове. Следват $n - 1$ реда с две числа — ребрата между върховете. На следващата страница има пример. След като прочетем дървото го пазим във масив от вектори, където на позиция i стои вектор от съседите на върха i.

Фигура 1: Примерен вход



6	
0	1
0	2
1	3
1	4
2	5

2 Backtracking

За да решим проблема, правим пълно обхождане. Избираме наредба, в която ще обхождаме върховете. Последователно им присвояваме всички възможни стойности. След като приключим с всички върхове проверяваме дали изпълняват условията 1–4.

Algorithm 1 Brute-Force

```
 1: function PERMUTE(i)
 2:     if i = n then
 3:         if w_0 ... w_{n-1} is a valid assignment then
 4:             PRINT(w_0 ... w_{n-1})
 5:             exit recursion
 6:         else
 7:             return
 8:     for w ← 1, 3 ... 2n − 1 do
 9:         w_i ← w
10:         PERMUTE(i + 1)
```

Да подобрим горното търсене, като не винаги изпълняваме ред 10. В този момент имаме стойностите $w_0 \dots w_i$. Ако те нарушават условията 1–4, няма смисъл да изпълняваме ред 10. Важно е да изпълняваме тази проверка възможно най-бързо:

Algorithm 2 Backtrack

1: usedValues$[1 \dots 2n] \leftarrow$ zeroes
2: usedDiff$[1 \dots 2n] \leftarrow$ zeroes
3: **function** Backtrack(i)
4:     **if** i=n **then**
5:         Print($w_0 \dots w_{n-1}$)
6:         **exit recursion**
7:     **for** w $\leftarrow 1, 3 \dots 2n-1$ **do**
8:         d $\leftarrow \|w_{\text{Parent}(i)} - w_i\|$
9:         **if** usedValues[w] $= 0$ **and** usedDiff[d] $= 0$ **then**
10:             $w_i \leftarrow$ w
11:             usedValues[w] $\leftarrow 1$, usedDiff[d] $\leftarrow 1$
12:             Permute($i+1$)
13:             usedValues[w] $\leftarrow 0$, usedDiff[d] $\leftarrow 0$

Обръщаме внимание на ред 8. За да бъде ефикасна проверката за уникални разлики, избираме произволен връх за корен и вкореняваме дървото. Пренареждаме върховете, така че да са топологично сортирани. С други думи, ако в момента разглеждаме връх i, то задължително ще сме разгледали родителя му. Още повече, няма да сме разгледали децата му. В противен случай наредбата няма да е топологично сортирана. Така единствената нова разлика ще е между v_i и родителя му. За да е дефинирана функцията Parent в корена, правим лека модификация.

Algorithm 3 Calling Backtrack

for w $\leftarrow 1, 3 \dots 2n-1$ **do**
    $w_0 \leftarrow$ w, usedValues[w] $= 1$
    Backtrack(1)
    usedValues[w] $= 0$

Програмата действително работи по-бързо, но за далеч от желано време. Проблемът е там, че Brute Force алгоритъма е $\mathcal{O}(n^n)$. За алгоритъма 2 не може да твърдим време, по-добро от $\mathcal{O}(n!)$.

Да подобрим 2, като изберем наредба на върховете. **Most restrained variable** винаги избира най-ограничения връх, докато не изчерпа всики. Така ефекта от Backtrack се засилва най-много. За начален връх няма еднозначен победител — всеки връх може да е всяка стойност. Затова избираме този с най-голяма степен според **degree heuristic**. Добрата вест е, че понеже обхождаме върховете в топологично сортиран ред, автоматично изпълняваме MRV. За първи връх ще подредим върховете по степен и един по един ще ги използваме за корен.

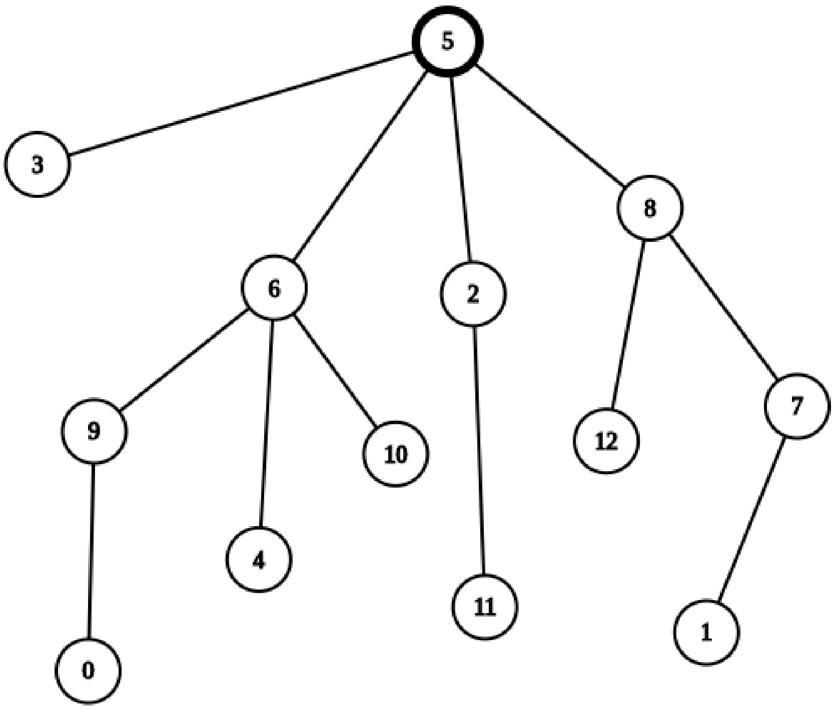
Ред 7 на Backtrack също е важен. Ако разгледаме стойностите, който е по-вероятно да стигнат до решение, ще намалим времето за търсене.

3 Експеримент

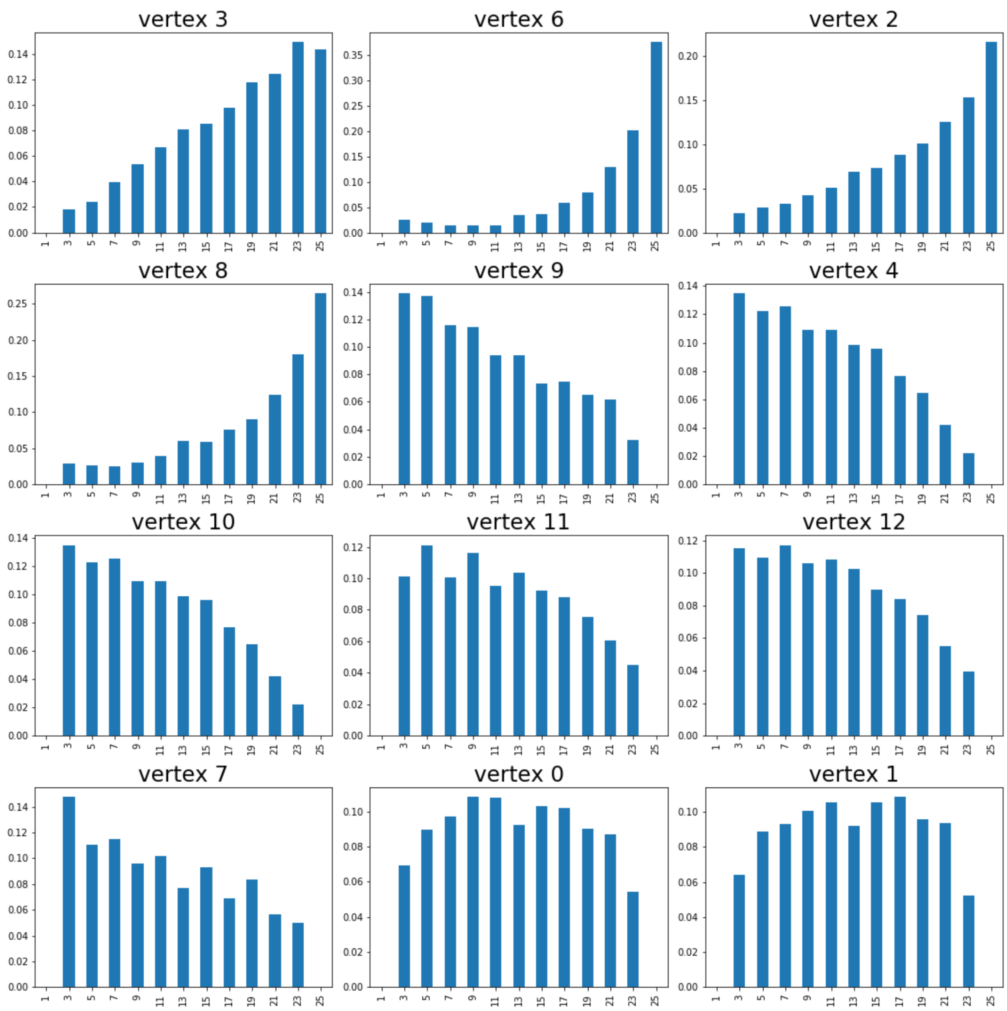
Нека разгледаме произволен връх i като случайна величина X_i. Дефинираме $\mathbb{P}(X_i = x) = \dfrac{\text{броя решения, за който } w_i = x}{\text{всички решения}}$. Всъщност на всяка стъпка в рекурсията правим избор за стойност на връх, като намаляме множеството от решения. Търсим тогава такива x за върховете i, за който $\mathbb{P}(X_i = x)$ е възможно най-голямо.

Да видим емпирично как изглежда $\mathbb{P}$ спрямо X_i. Фиг. 2 показва произволен граф с 13 върха. Фиг. 3 показва разпределението на всеки връх. За корен е избран 5, като $w_5 = 1$.

Фигура 2: Произволен граф при n=13



Фигура 3: Разпределнието на върховете



Забелязва се, че за върховете с нечетна дълбочина е по-добре да пробваме с големите стойности първо. Обратно, при другите е по-добре с ниските стойности. Също е важна подредбата на върховете. Например доста по-сигурни сме, че връх 6 трябва да е голям, отколкото връх 3. От такава гледна точка е по-добре връх 6 да е преди 3, за да има по-голям достъп до високите стойности.

Забележка Причината връх 6 да предпочита високи, а не ниски стойности, е поради избора за w_5. Ако w_5 бе $2n-1$, графиката щеше да изглежда наобратно. Произволно рехишме w_5 да е 1. Въпреки това е важно да е в един от краищата на интервлите, понеже е от висока степен. Така разликите между него и съседите му ще бъдат високи $(2n-2, 2n-4\ldots)$. Именно тези разлики най-трудно се получават.

Добрата вест е, че може да предскажем този ефект. Например, връх с много наследници е по-важен от такъв без изобщо. Емпирочно важни са следните атрибути:

- брой наследници
- дълбоча
- височина
- дали е листо или не

Тренирайки machine learning алгоритъм с тях, ще определим приоритет за всеки връх. Започвайки от корена, ще наредим върховете, така че тези с най-голям приоритет да са възможно най-напред в наредбата. Важно е да получим топологично сортиране на графа. Така общата схема на работа е:

Algorithm 4 Main Loop

G $\leftarrow$ Read tree from file
$(r_0 \ldots r_{n-1}) \leftarrow$ sort descending vertices by degree
for $r \in (s_0 \ldots s_{n-1})$ **do**
    Choose r as the root of the tree
    prio$[0 \ldots n-1] \leftarrow$ Predict node priorities
    $(v_0 \ldots v_{n-1}) \leftarrow$ OrderNodes(G, r, prio)
    $w_r \leftarrow 1$
    Alternating-Backtrack(v_1)

Algorithm 5 OrderNodes

function ORDERNODES(G, root, prio)
    q $\leftarrow \varnothing$
    ENQUE(q, root, ∞)
    order $\leftarrow \varnothing$
    while q is not empty **do**
        u $\leftarrow$ DEQUEUE(q)
        INSERTBACK(order,u)
        for v $\in$ CHILDREN(u) **do**
            ENQUE(q, v, prio[v])
    return order

Algorithm 6 Alternating-Backtrack

usedValues[$1 \dots 2n$] $\leftarrow$ zeroes
usedDiff[$1 \dots 2n$] $\leftarrow$ zeroes
function ALTERNATING-BACKTRACK(v_i)
    if v_i is one last the last vertex **then**
        PRINT($w_0 \dots w_{n-1}$)
        exit recursion
    if DEPTH(v_i) $\equiv 0 \pmod 2$ **then**
        valueOrder $\leftarrow 1, 3 \dots 2n-1$
    else
        valueOrder $\leftarrow 2n-1, 2n-3 \dots 1$
    for w **in** valueOrder **do**
        d $\leftarrow \|w_{\textsc{Parent}(v_i)} - w_i\|$
        if usedValues[w] $= 0$ **and** usedDiff[d] $= 0$ **then**
            $w_{v_i} \leftarrow$ w
            usedValues[w] $\leftarrow 1$, usedDiff[d] $\leftarrow 1$
            PERMUTE(v_{i+1})
            usedValues[w] $\leftarrow 0$, usedDiff[d] $\leftarrow 0$

4 Заключение

Каква е равносметката? ALTERNATING-BACKTRACK сам по себе си подобравя много бързодействието. Разбира се, от време на време има дървета, който отнемат твърде много време за изчисление. Например при n=50 от 100 дървета не стига време за около 6. BACKTRACK вероятно щеше да изчисли общо 6.

А дали има смисъл от machine learning? Ако направим подредба спрямо броя наследници, което е най-важния атрибут, получаваме друг резултат — 59 от 100 дървета не се изчисляват навреме. Дори ако и двата алгоритъма успеят да намерят решение, ALTERNATING-BACKTRACK ще бъде доста по-бърз. Времето, което отделяме за определяне на приоритет, се изплаща при търсенето.