VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Porovnání - Hledání nejdelší cesty v grafu

Grafové algoritmy

Bc. Marián Kapišinský (xkapis00), Bc. Filip Weigel (xweige01)

16. prosince 2021

Obsah

1 Úvod

Hledání nejkratší cesty [1] je jeden ze známých problémů hledání cesty s minimální vzdáleností mezi dvě danými vrcholy v určitém grafu. Minimální délka může být definována jako počet hran, ze kterých se cesta skládá. Jinou možností je hledat minimální součet hran v ohodnoceném grafu. Problém nejkratší cesty pro graf bez vah hran lze řešit pomocí Dijkstrova algoritmu v polynomiálním čase. Pokud mohou být hrany v grafu ohodnoceny negativní vahou, tak lze použít algoritmus Bellman-Ford.

Podobným problémem je problém nejdelší cesty [5]. Problém nejdelší cesty je podobný s problémem nejkratší cesty, pouze s rozdílem, že hledáme cestu s maximální délkou. Problém nejdelší cesty je problém, který spadá do NP-těžkých problémů. Verze problému, kde se ptáme zda-li lze najít cestu v grafu alespoň dané délky je NP-úplný. Problém nejdelší cesty lze aplikovat například při návrhu elektronických obvodů nebo pro algoritmy řídící roboty v místnosti.

Vrchol. Vrchol [6] je jedním z hlavních prvků množiny, které definují graf. Graficky se označuje kruhem, či bodem. Lze se setkat i s označením **uzel**. Z vrcholu vycházejí hrany. Pokud z vrcholu žádné hrany nevycházejí (vrchol neinciduje žádná hrana), tak jej nazýváme izolovaný vrchol. Stupeň vrcholu je počet hran, které do vrcholu zasahují. Pro orientované grafy rozlišujeme vstupní stupeň a výstupní stupeň.

Hrana. Hrana [4] je dalším z hlavních prvků množiny definující graf. Jedná se o (ne)uspořádanou dvojici vrcholů grafu. Graficky se znázorňuje jako přímka, či křivka mezi vrcholy, které spojuje. Hrany lze taktéž ohodnotit váhovou funkcí, která vyjadřuje cenu přechodu mezi danými dvěma vrcholy (reprezentuje například vzdálenost, kvalitu atp.).

Graf. Graf G [1] reprezentuje množina vrcholů V, jehož prvky jsou propojeny pomocí hran z množiny E. V případě, že pro E platí $E \subseteq \{(x, y) \mid x, y \in V\}$, tak se jedná o orientovaný graf (záleží na pořadí ve dvojici). Všeobecný orientovaný graf může obsahovat smyčky. Dále může mít graf hrany ohodnocené, které lze reprezentovat pomocí váhové funkce $w : E \mapsto \Re$, kde $\Re$ jsou reálná čísla. Úplným grafem nazýváme takový graf, jehož všechny uzly jsou vzájemně propojeny.

Cesta. Cesta v grafu [3] $G = (V, E)$ označuje posloupnost $P = (v_0, v_1, ..., v_n)$ pro kterou platí, že v grafu G existuje hrana z daného vrcholu do jeho následníka. Žádné dva vrcholy (a tedy ani hrany) se přitom neopakují.

Problém nejdelší cesty. Problém nejdelší cesty [5] lze definovat jako: orientovaný graf $G = (V, E)$, kde každá hrana je ohodnocena váhovou funkcí $w : E \mapsto \Re$, délka $l(e) \in Z$ pro každou $e \in E$, kladná hodnota K a dané vrcholy $s, t \in V$. Jak již bylo zmíněno, pro obecné grafy je problém nejdelší cesty NP-těžký. Při omezení na grafy acyklické (DAG) nebo některé typy grafů je problém řešitelný v P. Pro problém nejdelší cesty v orientovaném grafu existují dva přístupy. Exaktní algoritmus a aproximační/heuristické algoritmy. Exaktní algoritmus funguje na principu hrubé síly (brute-force). Z heuristických algoritmů lze uvést např. greedy search, generické programování, tabu search, nebo simulované žíhání [2]. Řešení problému nejdelší cesty bude omezeno na obecné orientované grafy a rozlišovat se budou pouze řídké/husté grafy.

1.1 Použití

Problém nejdelší cesty je implementován v jazyce `Python3`. Obsahuje automatické generování randomizovaného orientovaného grafu s hranami ohodnocenými váhovou funkcí [2], které mají náhodnou hodnotu od 1 do 10. Spouští se pomocí `python3 graph.py {-h -v -b -g <depth> -s <depth> -n <num> -d <num>}`. Argument h slouží k výpisu nápovědy, v pro výpis vygenerovaného grafu, b pro výpočet pomocí brute-force, g pro greedy-algoritmus, který následně vyžaduje přirozené číslo jako parametr, s pro SGL algoritmus, taktéž vyžaduje přirozené číslo jako parametr, dále argument n pro zvolenou velikost grafu (nodes) a d, který udává maximální výstupní stupeň uzlu, pomocí kterého lze ovlivnit hustotu grafu. Následně dojde k spuštění programu, vygenerování grafu a výpočtu zvolených algoritmů s případným výpisem vygenerovaného grafu. Příklad spuštění skriptu s algoritmy brute-force a greefy-search s hloubkou prohledávání 3 nad randomizovaným grafem s 30 uzly a maximálním výstupním stupněm uzlu 3 i s jeho výpisem:

```
$ python3 graph.py -v -b -g 3 -n 30 -d 3
```

2 Algoritmy

2.1 Brute-force algoritmus

Algoritmus, který využívá hrubou sílu je zapotřebí projít všechny cesty v grafu a ze všech cest vybrat právě tu, která má největší součet hran ohodnocených váhovou funkcí. Pokud projdeme všechny cesty v grafu, tak je garantované, že nalezneme skutečně nejdelší cestu na rozdíl od aproximačních/heuristických algoritmů. Hlavní nevýhoda tohoto exaktního algoritmu je krvavá daň v podobě časové náročnosti, danou nutností prozkoumat všechny cesty v grafu. Výsledkem je nejdelší cesta grafu v podobě globálního maxima.

Algorithm 1 Brute-force algoritmus

```
global P_best
function VISIT(u, P)
    path.append(u)
    for v ∈ Adj[u] do
        if v ∉ P then
            if P_best.weight < P.weight then
                P_best ⟵ P
            end if
            VISIT(v, P)
        end if
    end for
    path.remove(u)
end function
function BRUTEFORCE(G)
    for u ∈ V do
        VISIT(u, P)
    end for
    return P_best
end function
```

2.1.1 Analýza algoritmu

Algoritmus musí prozkoumat všech n uzlů. V každém dalším kroku se počet uzlů, které je třeba prozkoumat sníží o jedna. Znamená to tedy, že musí prozkoumat $O(n-1)$, poté $O(n-2)$, atd. Je zřejmé, že v nejhorším možném případě, což je úplný graf, musí algoritmus prozkoumat $n!$ možností. Z popsaného vyplývá, že teoretická složitost algoritmu je $O(n!)$.

2.2 Greedy-search algoritmus

Algoritmy z řad greedy algoritmů patří k populárním heuristickým algoritmů na řešení optimalizačních úloh. Greedy algoritmus prozkoumává k-okolí uzlu, vybere nejlepší možné lokální řešení (lokální maximum) a předpokládá, že za pomocí lokálního maxima se lze dostat ke globálnímu maximu a tedy ke správnému řešení. Algoritmus je z principu podobný algoritmu brute-force, ale omezuje svoji hloubku prohledávání na k. Z lokálního prohledávání vybere nejdelší cestu a z této cesty vybere právě jeden následující uzel, připojí jej k řešení a proces opakuje až do chvíle dokud není dostupné žádné nové, lepší řešení, které je ve většině případů suboptimální, t.j. nepřesné.

Algorithm 2 Greedy-search algoritmus

```
global P_best
function VISIT(u, P, k)
    if k = 0 then
        return
    end if
    path.append(u)
    for v ∈ Adj[u] do
        if v ∉ P then
            if P_best.weight < P.weight then
                P_best ⟵ P
            end if
            VISIT(v, P, k − 1)
        end if
    end for
    path.remove(u)
end function
function GREEDYSEARCH(G, k)
    i ⟵ 0
    for u ∈ V do
        VISIT(u, P, k)
    end for
    P ⟵ P_best[i]
    i ⟵ i + 1
    while P ≠ longest do
        VISIT(P.last, P, k)
        P.append(P_best[i])
        i ⟵ i + 1
    end while
    return P_best
end function
```

2.2.1 Analýza algoritmu

Podobně jako brute-force algoritmus i tento algoritmus musí prozkoumat všech n uzlů, v dalším kroku $O(n-1)$, atd., ale pouze do určité hloubky, která je dána hodnotou k,

z čehož vyplývá teoretická složitost algoritmu

$$n * O(n-1) * \cdots * O(n-k+1) = O\left(\frac{n!}{(n-k)!}\right) = O\left(\binom{n}{k} * k!\right).$$

2.3 k-step Greedy Lookahead (k-SGL)

Následující algoritmus *k-SGL* [2] je taktéž z řady greedy algoritmů, ale na rozdíl od předcházejícího algoritmu bere *k-SGL* algoritmus jako vstup počáteční a koncový uzel a hledá nejdelší cestu mezi nimi. Algoritmus z počátečního uzlu vyhodnotí všechny parciální cesty délky k, vybere cestu s největší váhou a do výsledné cesty přípojí druhý uzel. Tento proces se opakuje, dokud není připojen uzel koncový. Za počáteční uzel pro další iterace je vybrán poslední uzel připojený do výsledné cesty. Algoritmus nikdy nezvolí uzel takový, kterým se není možné dostat ke koncovému uzlu. Toto je zaručené použitím algoritmu *reverzního BFS* (viz příloha A), který v každém kroku algoritmu vrátí množinu uzlů, které lze připojit do aktuální cesty a existuje pro něj cesta do koncového uzlu. Aby bylo možné zmíněný algoritmus použít pro hledání nejdelší cesty v obecném orientovaném grafu je zapotřebí, aby byl algoritmus spuštěn pro všechny permutace dvojic uzlů, jejichž počet je roven $n * (n-1)$, což výrazně zhoršuje časovou složitost výpočtu.

2.3.1 Analýza algoritmu

Časová složitost procedury *kSGL-DFS* je totožná se složitostí předchozího algoritmu *Greedy-search*, což je $O(\binom{n}{k} * k!)$. S každým voláním *kSGL-DFS* dojde k volání procedury *Reverse-BFS*, jejíž složitost je $O(n^2)$. Spolu dávají tyto složitosti hodnotu $O(n^2 * \binom{n}{k} * k!)$. Procedura *kSGL-DFS* je z hlavní procedury *kSGL* volána $O(n)$-krát, což nám dává výslednou časovou složitost algoritmu $O(n^3 * \binom{n}{k} * k!)$. Vlivem potřeby testování všech dvojic uzlů pro nalezení nejdelší cesty v grafu se časová složitost algoritmu zhoršuje na

$$O\left(n^3 * \binom{n}{k} * k!\right) * O\left(n * (n-1)\right) = O\left(n^5 * \binom{n}{k} * k!\right).$$

2.3.2 Možné vylepšení složitosti algoritmu

Dle autora [2], by bylo vhodné nahradit proceduru reverzního BFS, které se volá v každým kroku *kSGL-DFS*, protože jeho složitost v této proceduře dominuje. Dále, pro řešení problému nejdelší cesty v grafu by bylo vhodné rozumně určit kandidáty na počátečné a koncové uzly a zkoumat pouze ty, aby se odstranila nutnost zkoumat všechny permutace dvojic uzlů.

Algorithm 3 kSGL algoritmus [2]

```
global P_best
function kSGL-DFS(G, t, k, curdepth, P)
    if (curdepth < k and P.last ≠ t) then
        Vn ⟵ ReverseBFS(G, P, t)
        for v ∈ Vn do
            P_next ⟵ P
            P_next.append(v)
            kSGL − DFS(G, t, k, curdepth + 1, P_next)
        end for
    else
        if P_best.weight < P.weight then
            P_best ⟵ P
        end if
    end if
end function
function kSGL(G, s, t, k)
    P ⟵ P_best ⟵ s
    while P.last ≠ t do
        kSGL − DFS(G, t, k, 0, P)
        if P.weight < P_best.weight then
            P.append(P_best.last)
        end if
    end while
    P ⟵ P_best
end function
function kSGL-permutations(G, k)
    P_globalbest
    for each (s, t) of permutations(G.V, 2) do
        kSGL(G, s, t, k)
        if P_globalbest.weight < P_best.weight then
            P_globalbest ⟵ P_best.weight
        end if
    end for
    return P_globalbest
end function
```

3 Experimenty

3.1 Metodika

Všechny testy proběhly na notebooku Lenovo Y520.
OS: Ubuntu 20.04.
CPU: čtyř-jádrový Intel i5-7300HQ @ 3,3GHz.
RAM: 16Gb DDR4.
Skript využíval vždy pouze jedno jádro procesoru, které bylo vytížené na 100%. Paměťová náročnost byla do 10Mb RAM. Skript byl vždy spuštěn samostatně a notebook vykonával pouze tuto práci.

3.2 Výsledky

V následující tabulkách jsou prezentovány výsledky experimentů prováděné na základě uvedené metodiky. Pro krátce běžící algoritmy (do 100s) byly experimenty spuštěny 5x. Následně se z hodnot pro dané algoritmy vypočítal průměr a hodnota byla zapsána do tabulky. Pro déle běžící algoritmy (100 - 1000s) se experiment opakoval 2x a pro nejdéle běžící algoritmy (>1000s) byl experiment proveden pouze jedenkrát.

V každém experimentu byl vždy proveden výpočet exaktním Brute-force algoritmem, dvě varianty heuristického Greedy algoritmu (5 a 8-lookahead) a dvě varianty heuristického SGL algoritmu (2 a 3-SGL). V záhlaví (25,3; 25,6; atd.) je vždy možno vidět počet uzlů (Nodes) a maximální výstupní stupeň každého uzlu. Naměřené hodnoty jsou uvedeny v sekundách, pokud není uvedeno jinak a jsou uvedeny ve tvaru: čas (součet vah). V případech, kdy je uvedena hodnota „>2h“ atp., byl výpočet úmyslně přerušen z důvodu vysoké časové náročnosti. Pokud jsou uvedeny otazníky, tak nebyla hodnota známa a v případě pomlčky nebyl experiment proveden.

alg./n,d	25,3	25,6	30,3	30,5	35,2
Brute-force	0.01(108)	57.97 (165)	0.23 (160)	199.59 (188)	0.01 (130)
5-GS	0.01 (78)	0.01 (93)	0.01 (123)	0.01 (129)	0.01 (89)
8-GS	0.01 (89)	0.04 (123)	0.01 (117)	0.07 (145)	0.01 (79)
2-SGL	0.234 (89)	1.48 (129)	0.73 (142)	2.33 (156)	0.53 (95)
3-SGL	0.402 (89)	4.27 (149)	1.53 (147)	5.04 (167)	0.81 (108)

Z prezentovaných tabulek pro Brute-force i relativně malý graf s $n = 35$, *outdegree 8* dokázal časovou složitost zhoršit tak výrazně, že nenalezl řešení ani za hodinu a půl. Podobně tomu

alg./n,d	35,8	75,2	75,8	100,2	100,8
Brute-force	>1.5h (???)	0.69 (245)	>2h (???)	3.53 (346)	>4h (???)
5-GS	0.03 (131)	0.01 (182)	0.09 (351)	0.01 (134)	0.12 (378)
8-GS	1.53 (153)	0.01 (139)	2.41 (403)	0.01 (165)	12.79 (343)
2-SGL	6.36 (193)	1.98 (185)	445.0 (415)	148.75 (224)	906.32 (479)
3-SGL	23.65 (203)	4.64 (201)	1472.79 (472)	351.9 (285)	3250.81 (499)

alg./n,d	200,2	300,2	500,8	1000,8
Brute-force	825.6 (532)	>10h (???)	-	-
5-GS	0.01 (248)	0.01 (342)	0.35 (2024)	1.04 (3958)
8-GS	0.01 (185)	0.01 (394)	29.94 (2126)	66.71 (4209)
2-SGL	2750.6 (351)	>2h (???)	-	-
3-SGL	10316.54 (445)	>4h (???)	-	-

bylo i při $n = 75$, *outdegree = 8* a $n = 100$, *outdegree = 8*, kdy nenalezl řešení za 2 hodiny, respektive 4 hodiny. Když byl experiment spuštěn s parametry *n=300, outdegree=2*, tak jsme byli nuceni experiment po deseti hodinách ukončit. Pro parametry *n=500, outdegree=8* a *n=1000, outdegree=8* jsme experiment nespouštěli, jelikož by to byla záležitost na několik dní a více.

Greedy algoritmus v případech 5 i 8 Greedy search pracoval velmi rychle ve všech případech. Pro *n=25, outdegree=6* zvládl heuristický výpočet za 0.01s, respektive 0.04s. V případech, kdy bylo k dispozici přesné řešení Bruteforce algoritmem se jeho úspěšnost pohybovala mezi 56% - 78%. Pozoruhodných rychlostí dosahoval i při vysokých hodnotách *n=1000, outdegree=8*, kdy výpočet provedl za 1.04s, respektive 66.71s pro 8-Greedy lookahead, kdy by výpočet hrubou silou zabral pár dní, týdnů, měsíců. Otázkou ovšem zůstává jeho přesnost při zmíněných hodnotách.

Algoritmus *SGL* vždy pracoval pomaleji než Greedy-search, ale ve všech případech (n,d) našel lepší výsledek než Greedy-search. Avšak v žádném z případů neporazil Greedy-search v časové složitosti, což se dalo dle teoretických složitostí algoritmů předpokládat. Za zmínku stojí například hodnoty *n=200, outdegree=2.* kdy byl Brute-force algoritmus výrazně rychlejší včetně přesného řešení, ale to byla jediná výjimka. Jinak se jeho úspěšnost pohybovala mezi 65% - 89%.

4 Závěr

Brute-force algoritmus. V případech, kdy graf obsahoval více hran a začal být hustší (zvyšující se hodnotou *outdegree*), tak se časová složitost výrazně zhoršila, což potvrzuje teoretickou složitost $O(n!)$. Hlavní výhodou Brute-force je bezpochyby přesné řešení. Ovšem Brute-force je použitelný pouze pro relativně malé a řídké grafy, jelikož i při 35 uzlech a maximální hodnotou výstupního stupně 8 nebyl schopen algoritmus najít řešení za hodinu a půl. Při větších grafech a hustších grafech stoupne časová složitost natolik, že Brute-force algoritmus je prakticky nepoužitelný. Variantou zrychlení by byla masová paralelizace algoritmu.

Greedy-search algoritmus. Vyniká svojí časovou složitostí. V porovnání s exaktním i druhým heuristickým algoritmem vždy našel řešení za zlomek času. Jeho hlavní nevýhodou je relativně nízká přesnost oproti oběma algoritmům, jelikož Greedy-search hledá lokální maximum. Pokud tedy potřebujeme rychlé řešení a spokojíme s nízkou přesností je Greedy-search dobrou volbou.

k-SGL algoritmus. Je relativně dobrou volbou při poměru cena/výkon. Jeho přesnost není špatná a vždy našel lepší řešení, než Greedy-search. Problém nastal u velkých grafů s nízkou hustotou, kdy našel horší řešení za podstatně delší čas než Brute-force algoritmus, jak bylo popsáno výše. V případě hustších grafů našel vždy řešení rychleji, než Brute-force algoritmus a dal nám dobrou představu o exaktním řešení, které by mělo být větší o cca. 10% - 30%.

Literatura

[1] Fieger, K. *Using graph partitioning to accelerate longest path search.* Disertační práce.

[2] Scholvin, J. K. *Approximating the longest path problem with heuristics: A survey.* Disertační práce.

[3] Wikipedia. *Cesta (graf) — Wikipedia, The Free Encyclopedia* [http://cs.wikipedia.org/w/index.php?title=Cesta%20(graf)&oldid=20373866]. 2021. [Online; accessed 04-December-2021].

[4] Wikipedia. *Hrana (graf) — Wikipedia, The Free Encyclopedia* [http://cs.wikipedia.org/w/index.php?title=Hrana%20(graf)&oldid=11484766]. 2021. [Online; accessed 04-December-2021].

[5] Wikipedia. *Longest path problem — Wikipedia, The Free Encyclopedia* [https://en.wikipedia.org/wiki/Longest_path_problem]. 2021. [Online; accessed 04-December-2021].

[6] Wikipedia. *Vrchol (graf) — Wikipedia, The Free Encyclopedia* [http://cs.wikipedia.org/w/index.php?title=Vrchol%20(graf)&oldid=20354965]. 2021. [Online; accessed 04-December-2021].

A Reverzní BFS

Algorithm 4 Reverzní BFS

```
function ReverseBFS(G, P, t)
    for u ∈ G.V do
        if u ∈ P then
            visited[u] ⟵ True
        else
            visited[u] ⟵ False
        end if
    end for
    Q ⟵ result ⟵ ∅
    ENQUEUE(Q, t)
    while Q ≠ ∅ do
        v ⟵ DEQUEUE(Q)
        if (visited[v] = False) and (v ∉ P) then
            for u ∈ G.V do
                if G.edges[u][v] > 0 and (u ≠ v) then
                    if P.last = u then
                        result ⟵ result ∪ {v}
                    else
                        ENQUEUE(Q, u)
                    end if
                end if
            end for
            visited[v] ⟵ True
        end if
    end while
    return result
end function
```