KIV/PRO

Kruskalův algoritmus

Druhá semestrální práce

Radek Švamberg

Zadání

Prostudujte pro zvolený problém existující metody řešení. Vyberte jednu z nich nebo navrhněte vlastní, implementujte a ověřte na experimentech. Postup a výsledky popište ve zprávě.

Budu řešit problém, kterým jsem se zabíral v první části mé semestrální práce, tedy hledáním minimální kostry grafu (s řešením přes alternativní implementaci Kruskalova algoritmu)

Existující metody

Využívat budu článek, který jsem zpracovával v první semestrální práci, pojednávací o implementaci Kruskalova algoritmu s využitím přihrádkového řazení, tedy hledání minimální kostry grafu. Výhoda této implementace je především zlepšení časové náročnosti Kruskalova algoritmu v případě grafu s rovnoměrně rozloženými ohodnoceními hran. Nevýhoda nastává u grafů, kde bychom měli většinu hran v podobném intervalu (např. 9 hran by spadalo do intervalu <1, 2>) a poté jen pár hran v intervalu jiném, větším (např. jedna hrana by spadala do intervalu <9,10>). Zde bychom měli několik prázdných přihrádek a kvůli tomu bychom ztratili výhodu přihrádkového řazení.

Odkaz

Pro vyhledávání minimální kostry v grafu existuje ovšem samozřejmě mnoho dalších algoritmů, mezi ně patří i základní (původní) implementace Kruskalova algoritmu, Jarníkův algoritmus, či například Borůvkův algoritmus (příklad některých např. zde).

Zvolené řešení

Alternativní implementace Kruskalova algoritmu

Vezměme si ohodnocení hran. V nich budeme mít nějakou maximální hodnotu a minimální hodnotu, max a min. Vezmeme si intervalu [min, max], který si rozdělíme do stejně dlouhých podintervalů a dáme jim indexy 1, 2, … b. Poté si vytvoříme intervaly pro „přihrádky“ – to znamená, že hrany s „podobným“ ohodnocením (neboli nacházející se ve stejném intervalu) budou ve stejné přihrádce E(j). Budeme mít tedy místo jedné velké množiny všech hran E několik menších podmnožin E = E (1) ∪ E (2) ∪ … ∪ E (b). Některé přihrádky ovšem budou prázdné a rozložení hran bude záviset na tom, jak rozložené budou ohodnocení jednotlivých hran (např. měli bychom všechny hrany ohodnoceny stejně, např. 2 a potom jednu hranu ohodnocenou 10, tak bychom měli mnoho prázdných intervalů).

Podívejme se detailněji na pseudokód tohoto algoritmu:

Function upraveny_kruskal:

Vytvořit minimální kostru stromu T pro spojový graf G(V, E) s n vrcholy

begin

Určit max a min ohodnocení hran E

Zařadit E do přihrádek E (1), E (2), ..., E(b)

T := Ø

Vložit každý vrchol do jednoprvkové množiny

j :=0, H(j) := Ø

while |T| < n – 1 begin

if H(j) = Ø then begin

Vyber j u kterého není přihrádka E(j) prázdná

Vytvoř haldu H(j) pro E(j)

end

Vyber hranu e‘ = (u, v) s nejnižší cenou c(e‘) z přihrádky E(j)

Vymaž e‘ z H(j)

if FIND(u) ≠ FIND(v) then begin

T := T ∪ {e’}

UNION(FIND(u), FIND(v))

end

end

end

Experimenty a výsledky

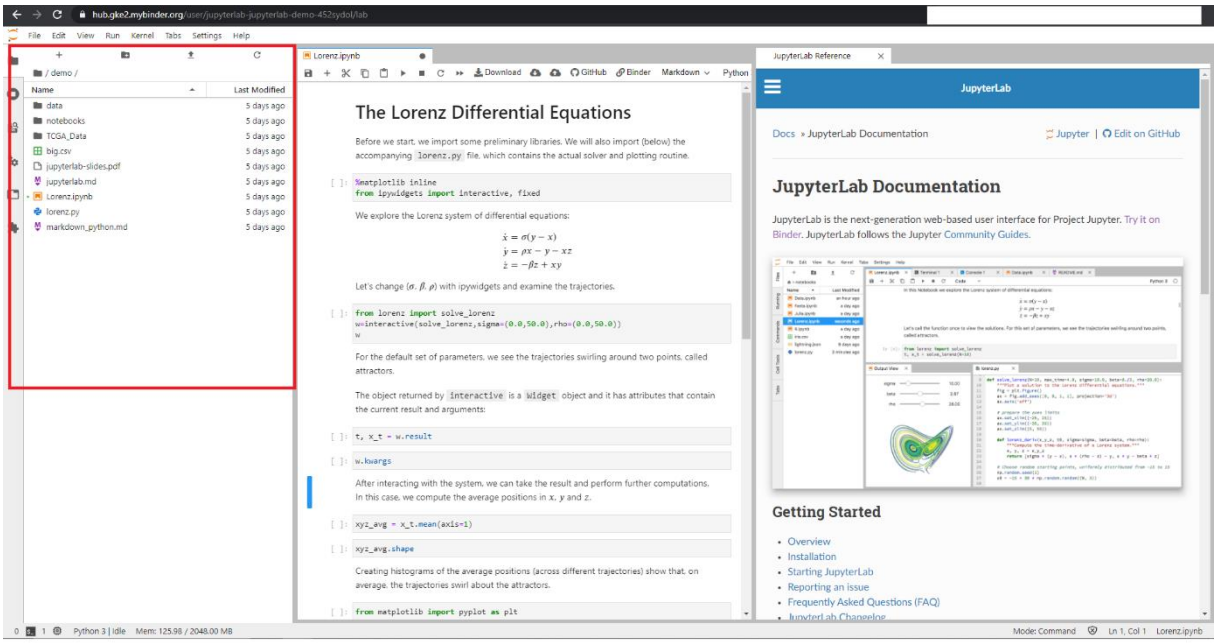
Pro tento experiment budu porovnávat, jak moc tato alternativní implementace pomůže časové náročnosti běhu oproti „obyčejné“ implementaci Kruskalova algoritmu. Očekávané výsledky budou, že v případě rovnoměrně rozložených ohodnoceních hran bude mít tato alternativní implementace lepší výsledky.

Programovat budu v jazyce Python v prostředí Visual Studio Code s využitím knihovny heapq a pandas. Knihovna heapq implementuje datový typ pro vytváření minimální haldy a manipulaci s ní. Knihovnu pandas poté využívám na vizualizaci histogramu. Prostředí Visual Studio Code umožňuje také využívat tkz. Jupyter notebooky. Nyní se podíváme na pár způsobů, jakým lze program spustit.

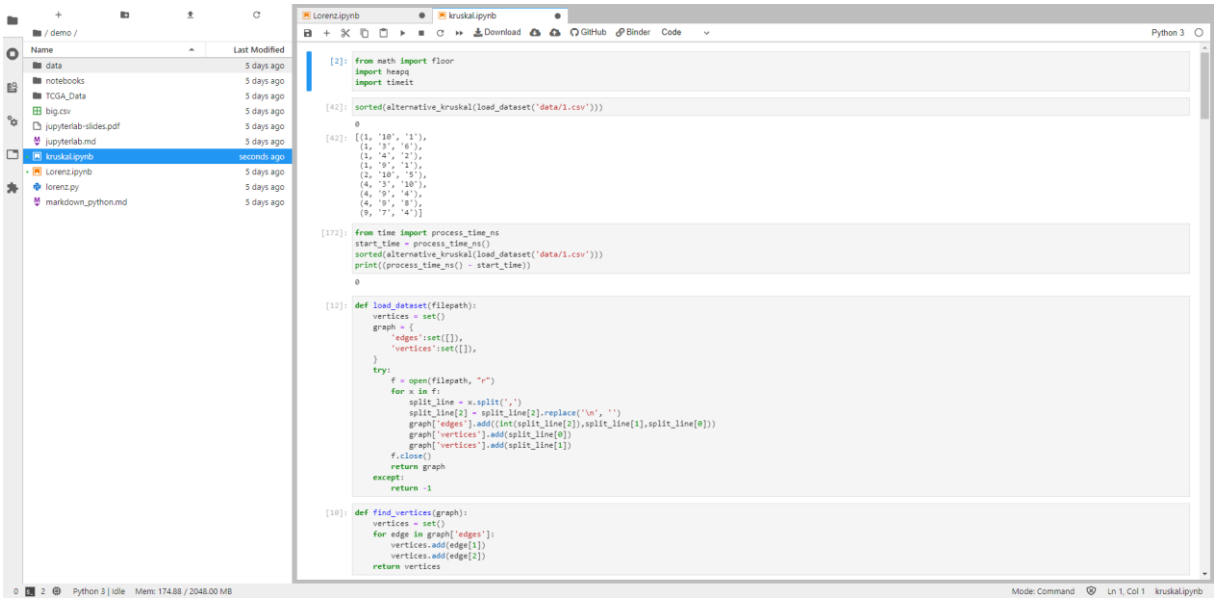
Spuštění skrz demo stránku

Pro účely kontroly / vyzkoušení kódu se dá využít demo stránka, kde běží připravené prostředí Jupyter se vším potřebným zde. Stačí pouze Jupyter soubor přetáhnout do adresáře serveru a spustit dvojitým poklepáním, viz obrázky (1) a (2). Poté do složky data/ přesunout csv soubory z /data. Výhodou je, že není nutnost se start s instalováním knihoven (obě jsou již nainstalovány). Nevýhodou je doba provádění skriptu a doba uploadu dat.

Obr. 1: adresářová struktura serveru s prostředím Jupyter



Obr. 2: otevřený souboru po přetáhnutí do adresáře /demo/



Spuštění skrz Python skript

Je připraven také skript Kruskal.py, který pouze potřebuje nainstalovanou verzi Python a pip knihovnu pandas (k tomu je potřeba spustit install_pandas.py skript). Tato verze grafy nevizualizuje rovnou, ale ukládá je jako .pdf do složky rozložení_dat/. Také využívá jinou verzi časování, která není tak přesná, jako verze ve verzi .ipynb a je zde pouze pro nějakou základní představu, nikoliv pro hlubší experimentování. Pokud není možnost pip knihovnu nainstalovat, je zde také verze bez ukládání grafů rozložení ohodnocení hran – Kruskal_no_pandas.py.

Spuštění skrz Visual Studio Code

Nejlepší možností je využít Visual Studio Code s rozšířením Jupyter. Poté stačí pouze otevřít dodanou složku s daty a zdrojovým kódem ve Visual Studio Code a spustit buňku po buňce (shift+enter).

Datasety, generace datasetů

Pro generování datasetů (neorientovaných vážených grafů) jsem využil šikovný program (knihovnu) pyrgg 0.9. V případě zájmu o vyzkoušení je přiložen .exe soubor ve složce /utils/. Tento program umožňuje generování grafů s požadavky uživatele čili jsem díky němu mohl vygenerovat několik neorientovaných vážených grafů pro účely experimentu, viz následující tabulka. Každý dataset má minimální váhu hrany 1, minimální počet hran vedoucí z vrcholu také 1 a žádná hrana neobsahuje smyčku.

Název	**Počet vrcholů**	**Maximální počet hran na vrchol**	**Maximální ohodnocení hrany**
1	10	3	10
2	100	10	100
3	1000	100	1000
4	10000	100	10000
5	100000	1000	10000
6	10000	100	10000
7	10000	100	10000

Implementace

Pro implementaci původního Kruskalova algoritmu jsem použil kód z GitHub od uživatele israelst, link zde. Tuto implementaci jsem poté upravil a na jejím základě postavil implementaci mou, podle článku z mé první semestrální práce, kde se využívá již zmíněné přihrádkové řazení.

Porovnávání implementací

Pro posouzení časové náročnosti budu využívat „magic module“ součástí IPython - %timeit, dokumentace zde. Tento šikovný modul spustí funkci, kterou chceme načasovat, tolikrát, aby mohl s dostatečnou přesností určit, jak dlouho trval její běh. Poté vypíše průměrnou dobu běhu a její směrodatnou odchylku. Díky tomu můžu přehledně porovnat obě implementace.

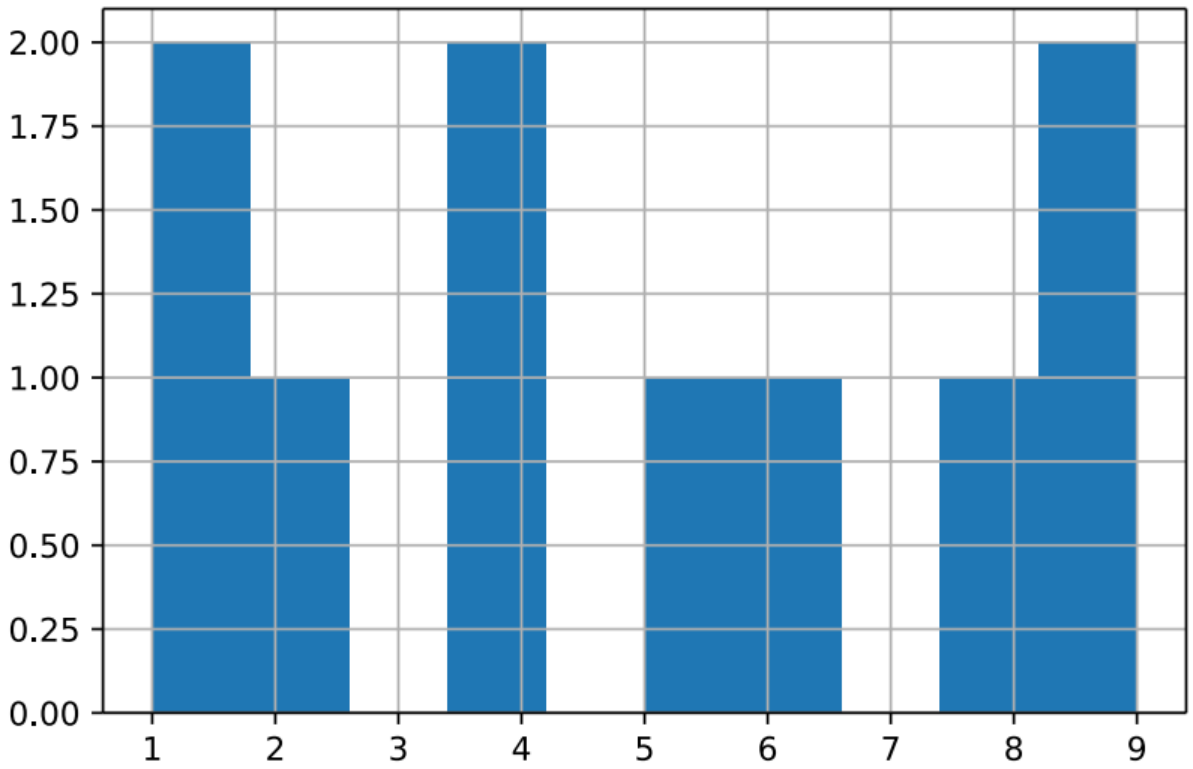
Očekávání

Očekávání u této implementace je, že si povede lépe u větších datasetů s rovnoměrným rozložením stran, díky čemuž budeme mít přihrádky rovnoměrně naplněné (nebude příliš „mezer“ mezi jednotlivými přihrádkami). Toto očekávání prověříme experimenty s časovou náročností.

Porovnání datasetů

1

Histogram rozložení ohodnocení hran



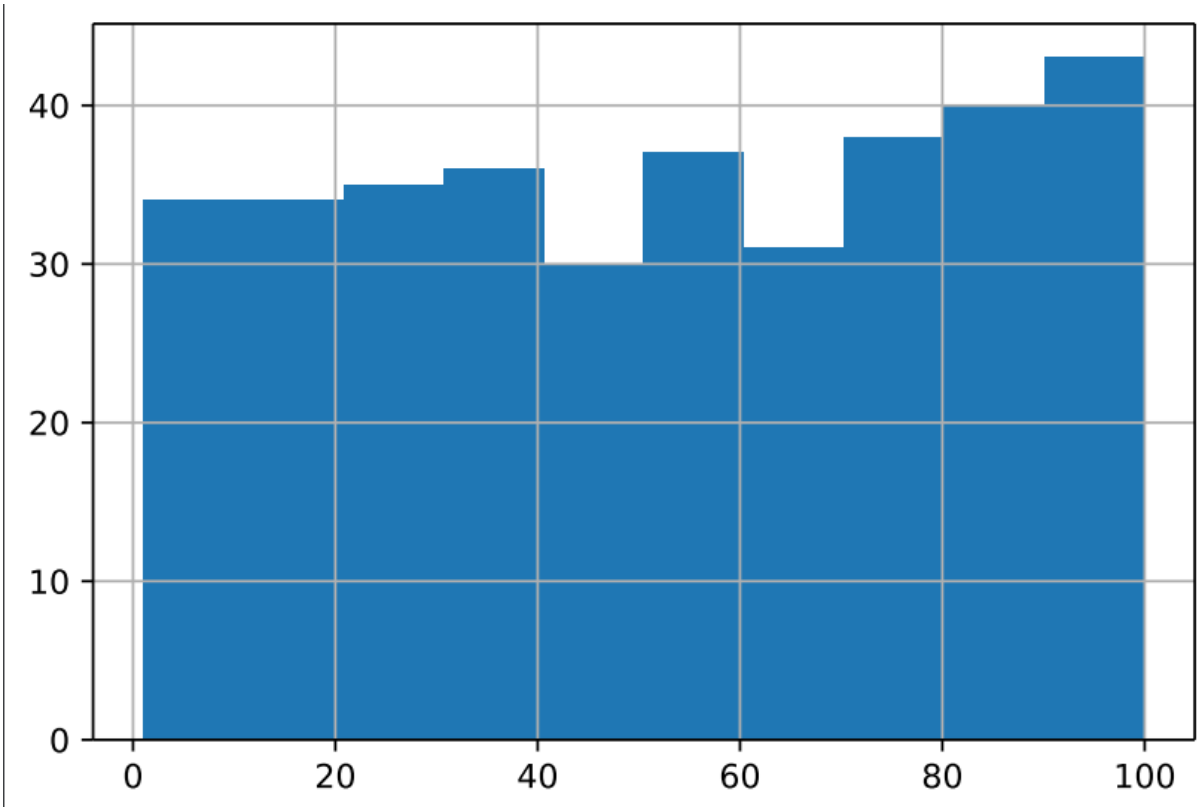
První dataset je vcelku malý, ale jak vidíme, ohodnocení hran jsou poměrně rovnoměrně rozložené. Očekával bych zde, že původní implementace bude rychlejší, ale ne o moc (pár prázdných přihrádek zde bude). Podívejme se, jak tomu doopravdy bylo.

```
[198]  M↓
    time_execution(DATASET)

Asserting minimum spanning trees...
Assert passed. Executing...

Time of execution of my kruskal implementation on the dataset 'data/1.csv'
193 µs ± 7.42 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
-----------------------------------------------------------------------
Timing of the original kruskal implementation on the dataset data/1.csv
163 µs ± 828 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Má domněnka se zde potvrdila – alternativní implementace zabrala průměrně 204 µs. Je zde ale poměrně vysoká směrodatná odchylka. Zato původní implementace zabrala 164 µs s velmi malou odchylkou 618 ns.

2

Histogram ohodnocení hran



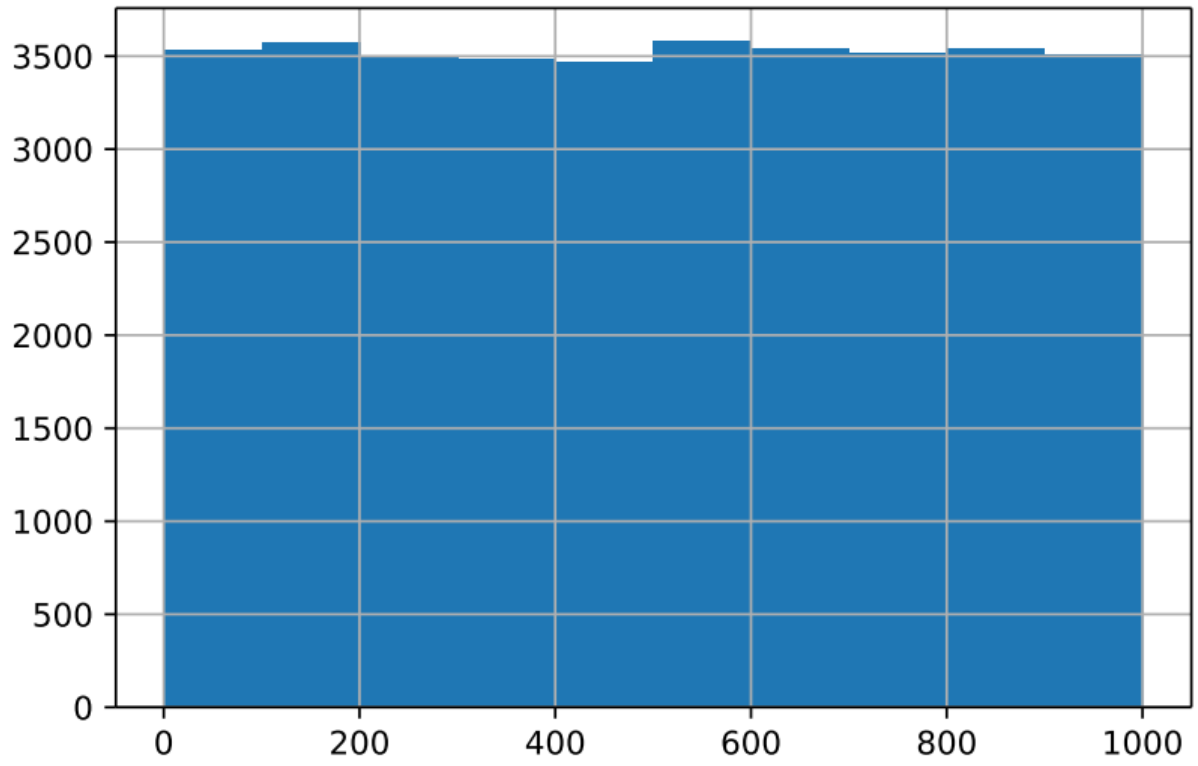
Druhý dataset je již trochu větší s poměrně velikým rozpětím ohodnocení hran. Ohodnocení jsou ovšem rovnoměrně rozložená (jak se již brzy ukáže, že tomu tak je u všech datasetů s větším počtem hran). Očekával bych tedy, že algoritmy na tom budou velmi podobně.

```
[201]
time_execution(DATASET)

Asserting minimum spanning trees...
Assert passed. Executing...

Time of execution of my kruskal implementation on the dataset 'data/2.csv'
1.55 ms ± 7.79 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
---------------------------------------------------------------------------
Timing of the original kruskal implementation on the dataset data/2.csv
1.23 ms ± 3.41 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Zde je vidět, že mé očekávané výsledky úplně neplatí. Alternativní implementace byla o 23% pomalejší. Také měla dvojnásobnou směrodatnou odchylku. To ovšem stále může být menším počtem hran, zkusme ho tedy navýšit a podívat se, jestli u většího datasetu s rovnoměrně rozloženými ohodnoceními dostaneme lepší výsledky.

3

Histogram ohodnocení hran



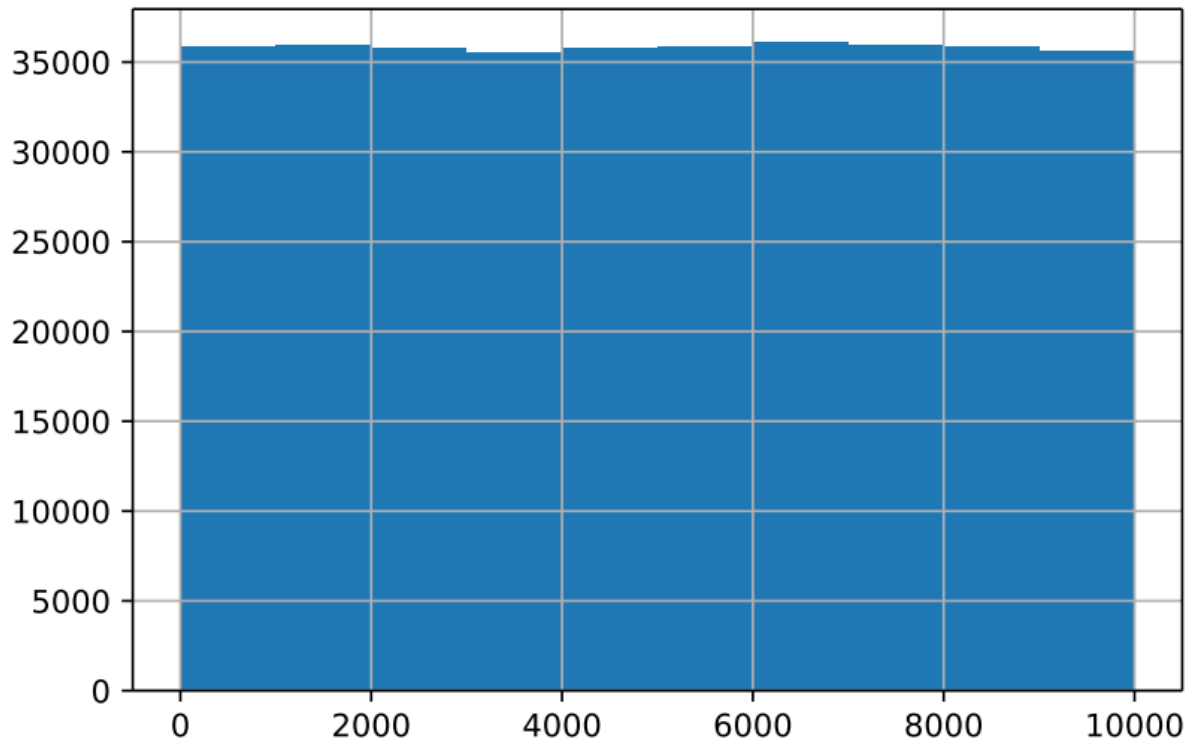
Zde jsou ohodnocení hran opět rovnoměrně rozložené a máme větší dataset. Má očekávání by byla, že pokud nějaký dataset vyhovuje alternativní implementaci (v teorii), bude to tento. Pojďme se podívat, jak tomu bylo.

```
[203]
time_execution(DATASET)

Asserting minimum spanning trees...
Assert passed. Executing...

Time of execution of my kruskal implementation on the dataset 'data/3.csv'
116 ms ± 1.3 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
---------------------------------------------------------------------------
Timing of the original kruskal implementation on the dataset data/3.csv
134 ms ± 1.74 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Zde se již začínají dostavovat klady této implementace. U většího datasetu s rovnoměrně rozloženými hranami je již alternativní implementace rychlejší o 14.4 % s menší směrodatnou odchylkou. U **4** se podíváme na ještě větší dataset – graf s 10000 vrcholy.

4

Histogram ohodnocení hran



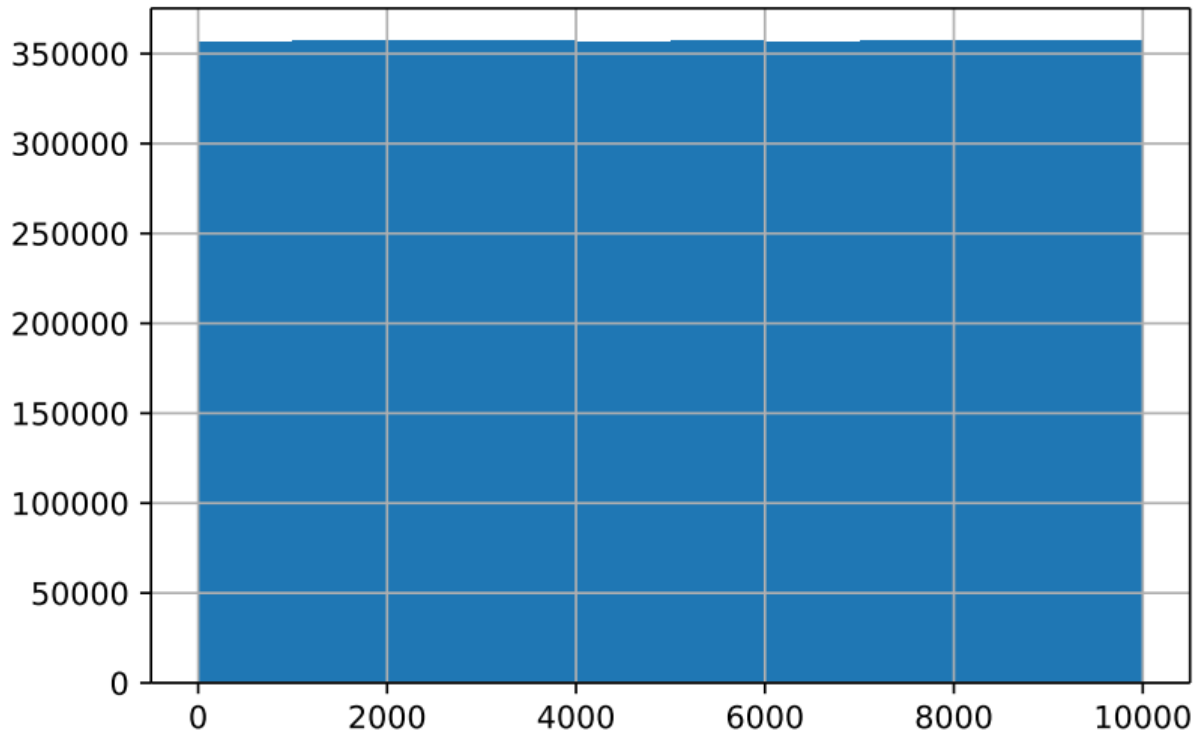
Alternativní implementace u tohoto datasetu s 10000 hranami a rovnoměrným rozložením hran by nám měl již poskytnout velmi pěkné výsledky.

```
[19]
time_execution(DATASET)

Asserting minimum spanning trees...
Assert passed. Executing...

Time of execution of my kruskal implementation on the dataset 'data/4.csv'
1.46 s ± 15.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
---------------------------------------------------------------------------
Timing of the original kruskal implementation on the dataset data/4.csv
1.88 s ± 29.5 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Jak vidíme, je tomu tak. Rozdíl už stoupl a alternativní implementace je rychlejší o 25 % s menší směrodatnou odchylkou. Vypadá to, že v naší teorii jsme měli pravdu a alternativní implementace funguje rychleji na veliké datasety s rovnoměrným rozložením ohodnocení hran. To otestujeme ve 5. datasetu – 100000 vrcholů.

5

Histogram ohodnocení hran



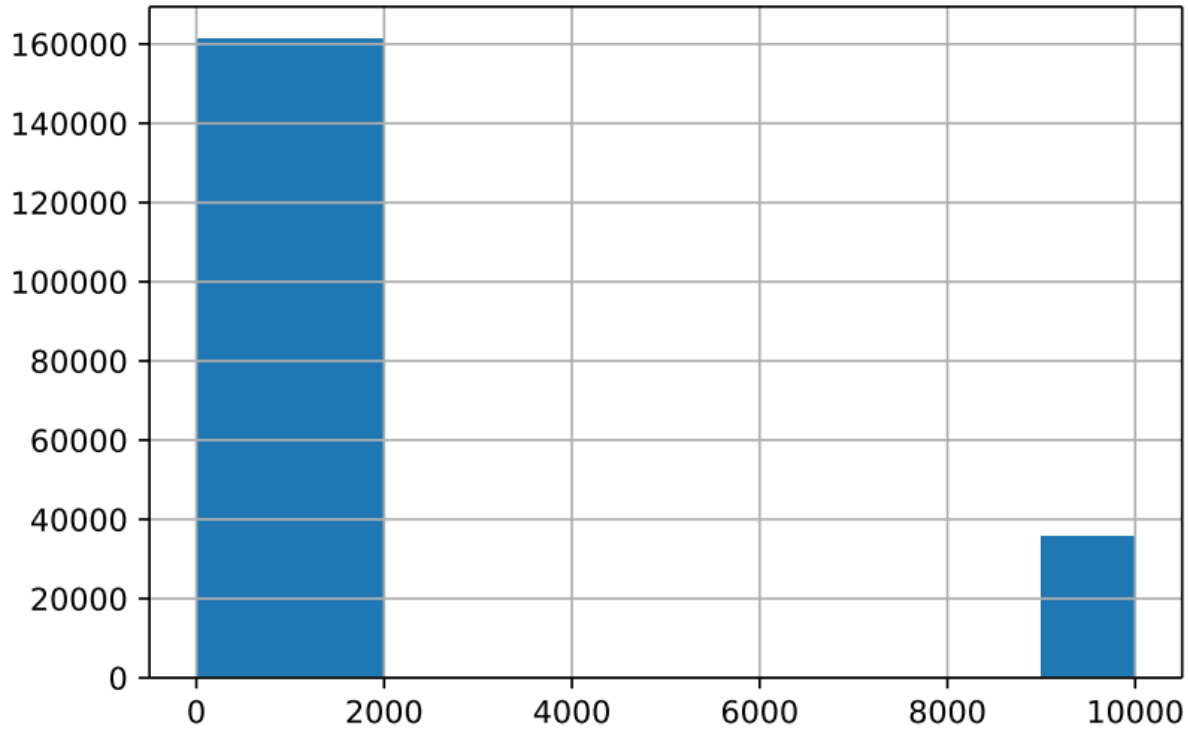
Zde již očekávám velké rozdíly v rychlosti. Dataset má 100000 vrcholů a rovnoměrné rozložení, takže je to perfektní případ pro náš algoritmus.

```
[24]
time_execution(DATASET)

Asserting minimum spanning trees...
Assert passed. Executing...

Time of execution of my kruskal implementation on the dataset 'data/5.csv'
18.9 s ± 1.13 s per loop (mean ± std. dev. of 7 runs, 1 loop each)
---------------------------------------------------------------------------
Timing of the original kruskal implementation on the dataset data/5.csv
29.3 s ± 947 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

U takto velkého grafu je má implementace rychlejší už o 43 %. Zde je vidět síla přihrádkového řazení v kombinaci s union & find v porovnání s jen union & find implementací.

6

Histogram ohodnocení hran



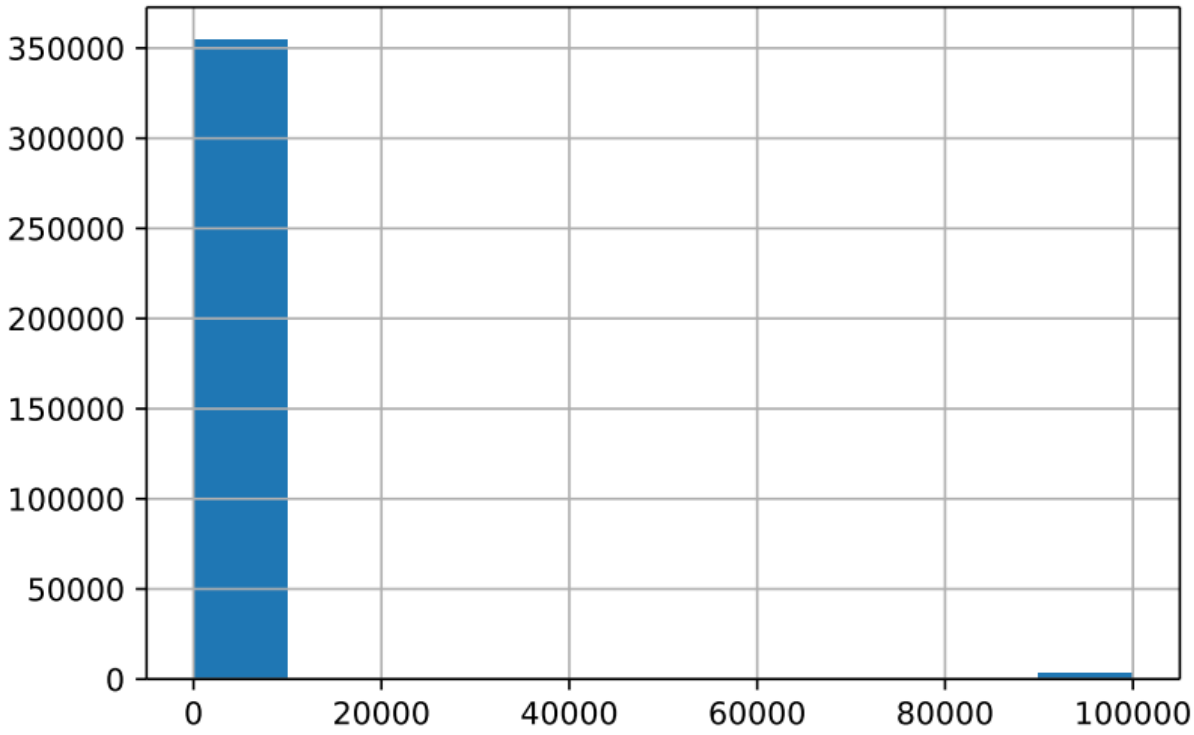
Zde jsem ohodnocení hran pozměnil, aby nebyla rovnoměrně rozložená. Tím dosáhneme hodně prázdných intervalů (přihrádek), čili náš algoritmus by měl být zpomalen.

```
[55]
    time_execution(DATASET)

Asserting minimum spanning trees...
Assert passed. Executing...

Time of execution of my kruskal implementation on the dataset 'data/6.csv'
1.62 s ± 47 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
---------------------------------------------------------------------------
Timing of the original kruskal implementation on the dataset data/6.csv
2.11 s ± 10.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Tento graf má stejný počet hran, jako graf z datasetu 4. V něm byla má implementace rychlejší o 25 %. Zde již nemáme ovšem rovnoměrné rozložení dat, a proto bych očekával, že má implementace poběží pomaleji. Vidíme ale, že opak je pravdou a má implementace je pořád rychlejší o 26 %.

7

Histogram ohodnocení hran



Zde jsem zvolil ještě extrémnější případ rozložení ohodnocení hran, díky kterému bude velmi velké množství prázdných přihrádek.

```
time_execution(DATASET)

Asserting minimum spanning trees...
Assert passed. Executing...

Time of execution of my kruskal implementation on the dataset 'data/7.csv'
1.66 s ± 174 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
---------------------------------------------------------------------
Timing of the original kruskal implementation on the dataset data/7.csv
2.08 s ± 43.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Algoritmu to přesto příliš neublížilo v porovnání s union & find implementací. Už je ovšem rychlejší o 22 %, čili poměr rychlostí už se snížil (mé implementaci tedy toto rozložení hran ublížilo).

Metody – klady a zápory

Union & find

Klady této implementace jsou především dobrý průměrný čas provedení u obecného grafu. Obzvlášť u menších grafů má implementace velmi dobré výsledky. U větších grafů ovšem začala zpomalovat oproti implementaci s přihrádkovým řazením.

Union & find + přihrádkové řazení

Klady přidání přihrádkového řazení jsou velké zrychlení u větších grafů s rovnoměrným rozložením ohodnocení hran. U 100000 vrcholů již byl algoritmus rychlejší o 43 % (přibližně 10 sekund), což je již veliký nárůst oproti základnímu union & find. Zápory jsou u grafů, které mají ohodnocení hran takové, že většina spadne do jedné přihrádky. V mé implementaci se ovšem ukázalo, že tento rozdíl není až tak drastický a má implementace dosahovala lepších výsledků i u nerovnoměrně rozloženého ohodnocení. Tuto implementaci bych určitě využil v případě, že bych pracoval s většími grafy, i kdyby neměla perfektně rozložené ohodnocení hran.

Závěr

V této práci jsem porovnával časovou náročnost dvou implementací Kruskalova algoritmu a vizualizoval výsledky. Poté jsem obě metody posoudil na základě výsledků a ohodnotil celý experiment. Vyvodil jsem, že má alternativní implementace vydává lepší výsledky u větších (počet hran > 10000) datasetů s rovnoměrně rozloženými ohodnoceními hran. Poté jsem se podíval na vliv nerovnoměrného rozložení hran a došel k závěru, že algoritmus má stále dobré výsledky i s nerovnoměrným rozložením.

Literatura

1. **J. Katajainen**, **O. Nevalainen -** An alternative for the implementation of Kruskal's minimal spanning tree algorithm https://www.sciencedirect.com/science/article/pii/0167642383900114
2. **A. Blum** – CMU 15-451/651 (Algorithms) lecture 6 (Union-Find, Minimum Spanning Trees) https://www.cs.cmu.edu/~avrim/451f13/lectures/lect0912.pdf
3. Bucket sort. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Bucket_sort
4. **R. Buckland** – Comp1927 lecture 10 https://www.youtube.com/watch?v=3mxp4JLGasE
5. **Israelt** – GitHub, Kruskal implementation https://github.com/israelst/Algorithms-Book--Python/blob/master/5-Greedy-algorithms/kruskal.py