Report – projekt NLA2

Projekt číslo 2 Ondřej Kučera KUC0436

Úvod

Cílem projektu je srovnat klasický singulární rozklad (SVD) a randomizovaný singulární rozklad (randomized SVD) na kompresi matic, a to z hlediska rychlosti výpočtu, paměťové náročnosti a přesnosti aproximace. Rovněž se zabýváme analýzou kompromisu mezi přesností a rychlostí. V tomto dokumentu si popíšeme použité postupy a projdeme si získané výsledky.

Popis použitých metod

Singulární rozklad matice M tvaru $m \times n$ je rozklad $M = U\Sigma V^*$, kde U tvaru $m \times m$ a V tvaru $n \times n$ jsou unitární matice, Σ tvaru $m \times n$ je obdélníková diagonální matice, která má nezáporné hodnoty na hlavní diagonále, kterým se říká singulární hodnoty, a * označuje hermitovsky sdruženou matici. Na rozdíl od spektrálního rozkladu, který existuje pouze pro tzv. normální matice, singulární rozklad existuje pro každou reálnou nebo komplexní matici.

Pokud se singulární rozklad neprovede úplně, ale pouze pro nejvyšších r singulárních hodnot, pak se mluví o zkráceném singulárním rozkladu (truncated SVD). Rekonstrukcí původní matice součinem $U\Sigma V^*$, kde U je tvaru $m \times r$, Σ je tvaru $r \times r$ a V^* je tvaru $r \times n$, se získá matice $\widetilde{M}$, která je pouze aproximací matice M. Konkrétně se jedná o nejlepší aproximaci původní matice maticí s nižší hodností r z hlediska Frobeniovy normy. Jelikož matice U, Σ a V obsahují pro $r \ll \min(m, n)$ mnohem méně prvků než matice M, využívá se zkrácený singulární rozklad ke kompresi dat.

Jak již bylo naznačeno v úvodu, v projektu se zabýváme analýzou časové a paměťové náročnosti výpočtu zkráceného singulárního rozkladu pro různé hodnosti matice $\widetilde{M}$, stejně jako analýzou přesnosti této matice a poměrem mezi přesností a rychlostí. Proto si musíme říct, jak se singulární rozklad provádí.

Algoritmy pro nalezení singulární rozkladu obecně vypočítají celý singulární rozklad najednou, iterativní algoritmus pro zkrácený singulární rozklad jako takový neexistuje. Proto se problém převádí na iterativní hledání vlastních čísel a vlastních vektorů matice A definované jako

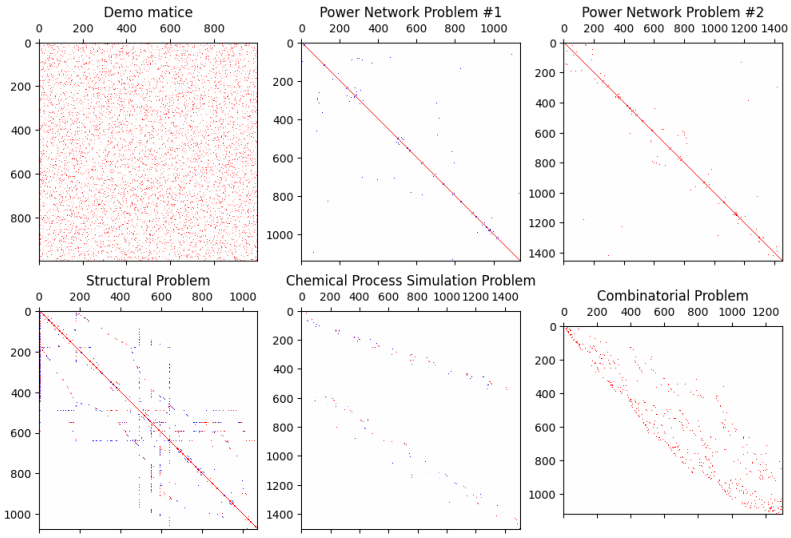
$$A = \begin{pmatrix} 0 & M \\ M^T & 0 \end{pmatrix}$$

Kromě tohoto, „naivního“, algoritmu se dá zkrácený singulární rozklad provést taky randomizovanou metodou, takovému algoritmu se pak říká randomizovaný singulární rozklad.

Randomizovaný algoritmus funguje na principu nalezení ortonormální matice Q s co nejméně sloupci tak, aby platilo $M \cong QQ^*M$ (tj. $\|M - QQ^*M\| < \varepsilon$ pro nějakou toleranci ε). Náhodný generátor se využívá právě na nalezení matice Q. Po nalezení této matice se provede singulární rozklad matice $A = Q^*M$. Jelikož matice A je menší než M, je nalezení tohoto singulárního rozkladu mnohem rychlejší. Z rozkladu matice A se potom jednoduše získá zkrácený singulární rozklad M, tedy i její aproximace $\widetilde{M}$.

V projektu jsem se rozhodl použít třídu **TruncatedSVD** z knihovny **scikit-learn**. Tahle knihovna má k dispozici oba algoritmy zkráceného singulárního rozkladu pro řídké matice – naivní algoritmus zvaný Arpack a také randomizovaný algoritmus. Umožnuje tedy jednoduché porovnání obou algoritmů.

Testování jsem se rozhodl provést na následujících maticích. Až na náhodně vygenerovanou „demo“ matici, která byla určena pro ukázku jednotlivých měření, jsem matice na testování stáhl ze **SuiteSparse Matrix Collection**. Matice mají různě uspořádané nenulové prvky a singulární hodnoty, program tedy testuje různé typy matic.



Nyní se podíváme na jednotlivé funkce měření.

Funkce na měření rychlosti algoritmů využívá knihovnu **timeit** a její stejnojmennou metodu. Metoda **timeit** opakuje výpočet několikrát pro přesnější výsledky. Algoritmus je jednoduchý, prostě změří, jak dlouho trvá výpočet pro různé hodnosti matice.

```
def measure_time(M, x):
    y_arp = []
    y_ran = []
    repeat_times = 20

    for k in x:
        svd = TruncatedSVD(n_components=k, algorithm="arpack")
        rsvd = TruncatedSVD(n_components=k, algorithm="randomized")

        time = timeit.timeit(lambda: svd.fit(M), number=repeat_times)
        y_arp.append(time / repeat_times)
        time = timeit.timeit(lambda: rsvd.fit(M), number=repeat_times)
        y_ran.append(time / repeat_times)

    return np.array(y_arp), np.array(y_ran)
```

Další měření je měření paměťové náročnosti. Za tímto účelem používám knihovnu **tracemalloc** a její metody **start**, **stop** a zejména **get_traced_memory**. Tahle metoda vrací množství alokované paměti. Rozdílem měření před a po výpočtu se tedy dá zjistit, kolik paměti bylo alokované během výpočtu, tj. jak náročný byl výpočet. Bylo potřeba postupně ukládat vytvořené instance třídy, aby nedocházelo k různým výchylkám kvůli uvolňování těchto instancí.

```python
def measure_memory(M, x):
    y_arp = np.zeros_like(x, dtype=float)
    y_ran = np.zeros_like(x, dtype=float)
    svds = []
    tracemalloc.start()

    for i, k in enumerate(x):
        mem_before, _ = tracemalloc.get_traced_memory()
        svds.append(TruncatedSVD(n_components=k, algorithm="arpack"))
        svds[-1].fit(M)
        mem_after, _ = tracemalloc.get_traced_memory()
        y_arp[i] += mem_after - mem_before

        mem_before, _ = tracemalloc.get_traced_memory()
        svds.append(TruncatedSVD(n_components=k, algorithm="randomized"))
        svds[-1].fit(M)
        mem_after, _ = tracemalloc.get_traced_memory()
        y_ran[i] += mem_after - mem_before

    tracemalloc.stop()

    return y_arp, y_ran
```

Nakonec poslední měření je měření přesnosti aproximace. Na to je potřeba zrekonstruovat matici $\widetilde{M}$, což naštěstí není složité. Relativní přesnost p aproximované matice jsem určil tímto vzorečkem:

$$p = 1 - \frac{\left\|\widetilde{M} - M\right\|}{\|M\|}$$

V něm $\|\cdot\|$ označuje Frobeniovu normu matice.

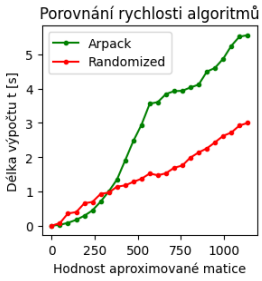
```python
def measure_accuracy(M, x):
    y_arp = []
    y_ran = []
    M_dense = M.todense("C")
    norm_exact = np.linalg.norm(M_dense)

    for k in x:
        svd = TruncatedSVD(n_components=k, algorithm="arpack")
        M_transformed = svd.fit_transform(M)
        M_tilde = M_transformed @ svd.components_
        norm_diff = np.linalg.norm(M_tilde - M_dense)
        y_arp.append(1 - norm_diff / norm_exact)

        rsvd = TruncatedSVD(n_components=k, algorithm="randomized")
        M_transformed = rsvd.fit_transform(M)
        M_tilde = M_transformed @ rsvd.components_
        norm_diff = np.linalg.norm(M_tilde - M_dense)
        y_ran.append(1 - norm_diff / norm_exact)

    return np.array(y_arp), np.array(y_ran)
```

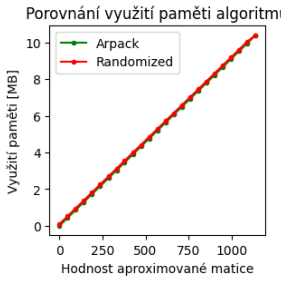
Nakonec, co se týká kompromisu mezi přesností a rychlostí výpočtu, to je jednoduché. Kompromis by měl být přímo úměrný přesnosti aproximace a nepřímo úměrný rychlosti výpočtu. Dává tedy smysl, aby velikost kompromisu byla dána poměrem přesnost/rychlost

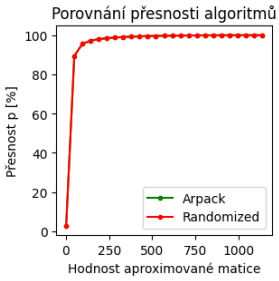
Výsledky

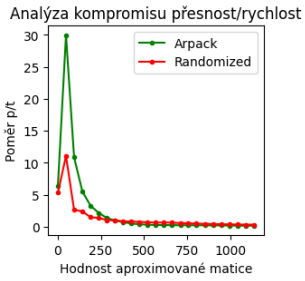
Tady jsou grafy zobrazující časovou náročnost, paměťovou náročnost, přesnost aproximace a kompromis mezi přesností a rychlostí obou algoritmů u všech pěti testovacích matic:

Power Network Problem #1

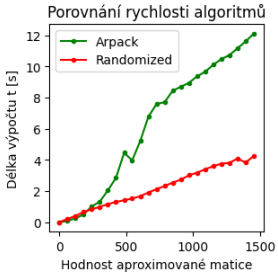


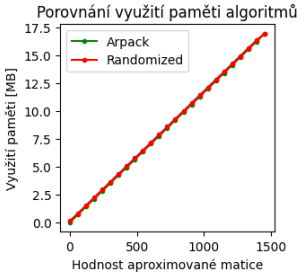


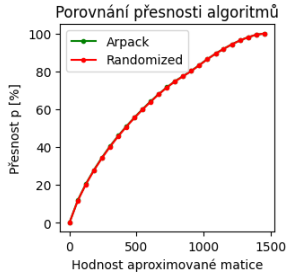


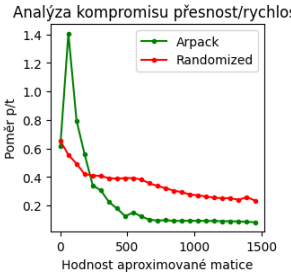


Power Network Problem #2

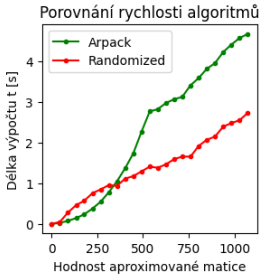


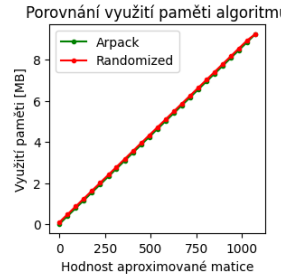


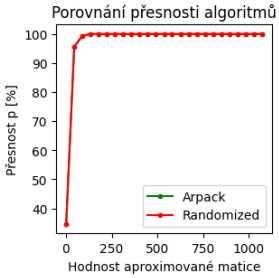


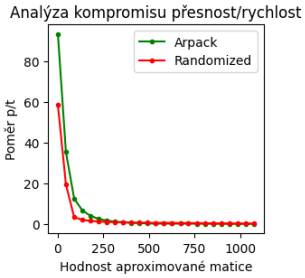


Structural Problem

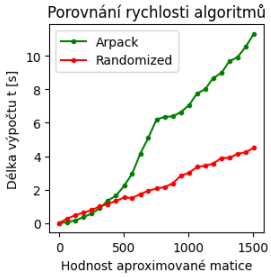


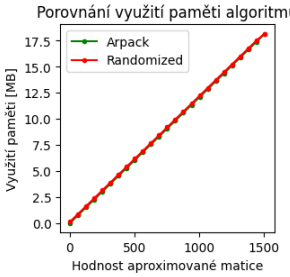


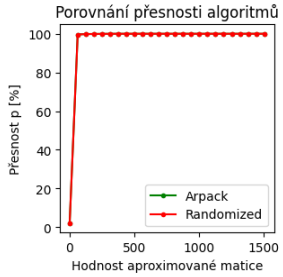


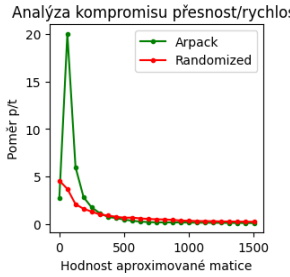


Chemical Process Simulation Problem

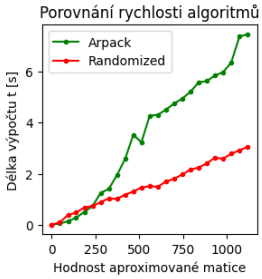


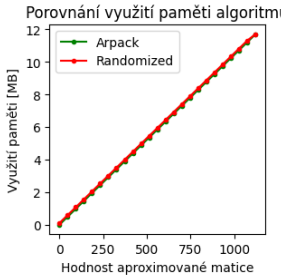


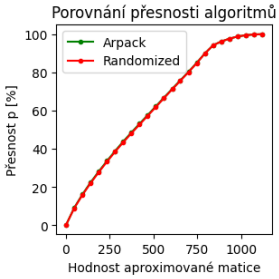


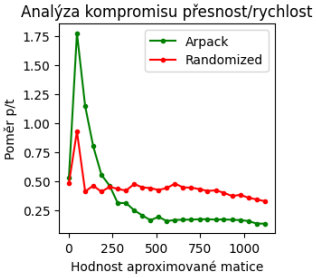


Combinatorial Problem









Proveďme nyní analýzu těchto výsledků.

Z hlediska rychlosti výpočtu je Arpack lepší pro nízké hodnosti, jež odpovídají velikostem do přibližně čtvrtině velikosti matice. Pro aproximaci maticí s vyšší hodností, než je tato hranice, je výhodnější randomizovaný algoritmus. To je očekávaný výsledek, protože randomizovaný algoritmus vznikl právě z tohoto důvodu. Na rozdíl od naivního algoritmu, jehož složitost je zřejmě exponenciální do poloviny velikosti matice a poté spíše lineární, randomizovaný algoritmus má lineární složitost pro všechny hodnoty a asymptoticky je alespoň dvakrát tak rychlý jako Arpack.

Paměťová složitost výpočtu se zvyšuje také lineárně, a to pro oba algoritmy. Lze také vidět, že randomizovaný algoritmus využívá nepatrně více paměti než Arpack. Každopádně jedná se o konstantní rozdíl, který vzhledem k lineární složitosti není nutné uvažovat.

Co se přesnosti aproximované matice týče, jsou zde vidět dva možné výsledky – buď se přesnost přiblíží 100 % už pro nízké hodnosti, anebo přesnost přibývá víceméně lineárně a pro vysokou přesnost je potřeba vysoké hodnosti. Tohle nevylučuje možnost, že by se nemohla najít taková matice, pro kterou by graf neodpovídal ani jedné z těchto možností. Přesnost aproximace totiž závisí jenom na rozložení singulárních hodnot původní matice. Dále, i když to není patrné z grafů, randomizovaný algoritmus je lehce méně přesný než Arpack. Tento rozdíl je však zanedbatelný.

Nakonec, z grafů je zřejmé, že nejlepší poměr přesnost/rychlost je pro velmi nízké hodnosti. Pro případy, kdy aproximace dosahuje vysoké přesnosti i pro nízké hodnosti, je toto zřejmé, avšak v druhém případě je to docela nesmyslné, neboť potom aproximace skoro vůbec přesná není. V některých případech vychází dokonce nejvýhodněji provést aproximaci maticí s hodností 1, jejíž přesnost se přibližuje 0. Je to dáno tím, že provést výpočet pro takto nízké hodnosti je opravdu rychlé, na rozdíl od vyšších hodností netrvá v podstatě nic.

Závěr

Při provádění zkráceného singulárního rozkladu mají oba algoritmy své výhody i nevýhody. Zatímco naivní algoritmus je rychlejší pro nízké hodnosti, asymptoticky je mnohem lepší používat randomizovaný algoritmus. Využívá sice trochu více paměti a je malilinko méně přesný, avšak často je důležité, aby výpočet netrval příliš dlouho. Pro matice, pro které na dosáhnutí vysoké přesnosti stačí nízké hodnosti matice, se vyplatí použít Arpack. Pokud by se stalo, že je možné implementovat pouze jeden z těchto algoritmů, doporučuji použít randomizovaný singulární rozklad, jelikož rychlost výpočtu pro nízké hodnosti není tak důležitá jako pro vysoké hodnosti, pro které trvá výpočet mnohem déle. Rovněž doporučuji neřídit se nejlepším kompromisem mezi rychlostí a přesností, protože to by mohlo vést k velmi nepřesným aproximacím.

Reference

(Pardon, že to není správně ocitované)

Singular value decomposition - Wikipedia

Randomized Singular Value Decomposition

What fast algorithms exist for computing truncated SVD? - Cross Validated

A review on the selection criteria for the truncated SVD in Data Science applications - ScienceDirect