

# Popis

Cílem projektu je vytvoření aplikace umožňující doporučovat uživateli produkty nad zvolenou databází produktů. Uživatel bude s aplikací interagovat pomocí webového rozhraní, ve kterém si zvolí dvojici atributů, na základě kterých aplikace nalezne zajímavé produkty. U každého atributu půjde zvolit, jestli je pro uživatele důležitější nižší nebo vyšší hodnota. Následně aplikace pomocí operátoru skyline nalezne produkty, které dominují ostatní v dané dvojici atributů.

Výstup bude realizován jako 2D graf, kde budou jednotlivé produkty reprezentovány jako body a výsledná skyline jako křivka spojující jednotlivé produkty, které dominují ostatní ve zvolených attributech. Ostatní body budou pod nebo nad skyline křivkou, podle toho, kde na osách leží optimum.

# Způsob řešení

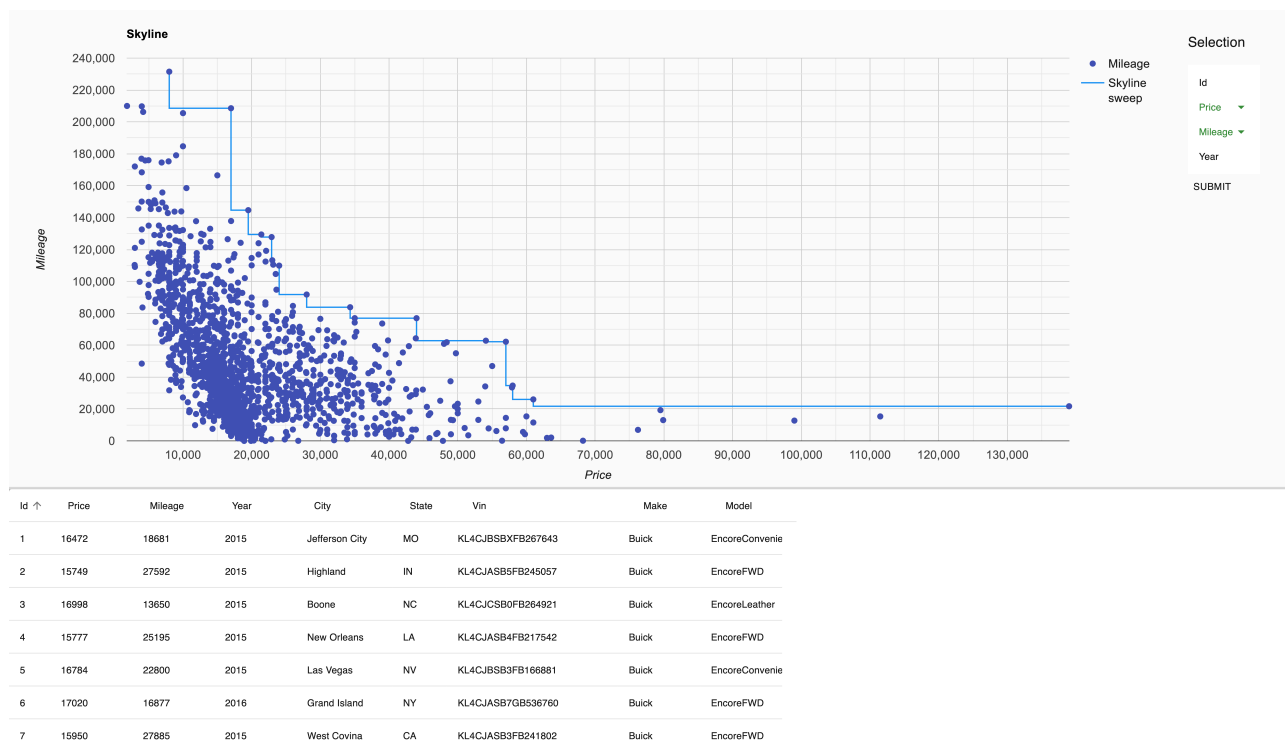
Pro implementaci jsem se rozhodl pro dvě řešení. První jako Desktop aplikaci napsanou v Pythonu a knihovnou PySide. Druhé jako webovou aplikaci v jazyce Javascript s knihovnou React. Dvě řešení proto, že jsem chtěl porovnat odlišnosti dvou jazyků a jejich vliv na samotnou implementaci jednotlivých algoritmů.

Pro výpočet skyline byly implementovány 3 algoritmy: “Brute force”, “Divide and conquer” a jako nejpokročilejší “Sweep plane”.

Řešení desktopové aplikace je díky použití pythonu cross-platform. Prvotně jsem využil pro načtení dat knihovnu pandas, ale to se ukázalo jako zbytečný over-head, protože jsem nakonec nevyužil žádné optimalizované metody, které pandas nabízí a přístup po jednotlivých řádcích (metoda iterrows) byla mnohonásobně pomalejší než nativní python list. Překvapení pro mne bylo, že ačkoliv PySide je rozhraní pro QT knihovnu v C++, tak rychlost aplikace již při 2000 prvků byla velmi nízká a pouhý re-render grafu trvá řádově vteřiny. (Ukázka rozhraní - Obr. 2)

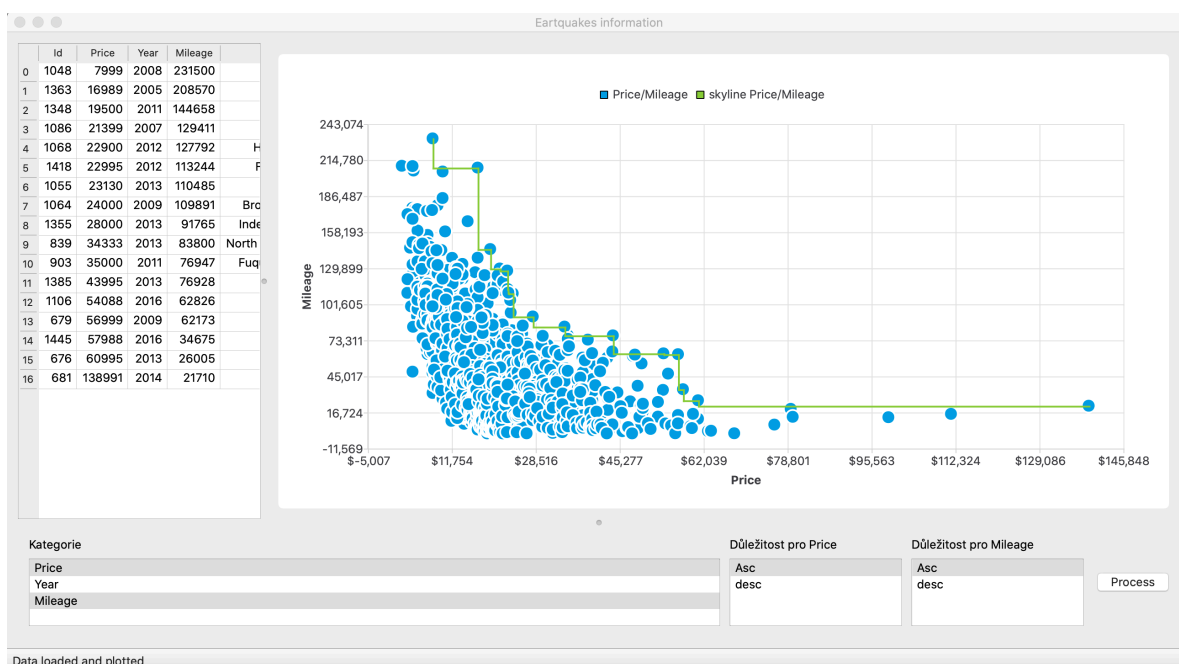
Pro řešení webové aplikace jsem využil knihovnu React a material-ui (nastylované komponenty pro React). Pro vývoj využívám standardní stack: webpack (jako builder) + babel (jako transpiler). JavaScript nemá definovaný žádný standardní řadící algoritmus a tedy se implementace velmi liší prohlížeč od prohlížeče, proto jsem využil modul “timsort” s implementací merge-sortu se složitostí  $O(n \log n)$ . Samotné uživatelské rozhraní se skládá ze tří částí. První je 2D graf se zobrazenými body a skyline křivkou. Vpravo od grafu je možnost výběru atributů a jejich důležitosti. A ve spodní části je tabulka se všemi prvky - zde jsem pro optimální výkon využil modul “react-virtualized”, který bez problému zvládne v prohlížeči pracovat s tabulkou o deseti tisících řádcích. (Ukázka rozhraní - Obr. 1)

# Příklad výstupu



Obr. 1 - Ukázka webového rozhraní

Jako vstup jsem využil staženou databázi ([odkaz](#)) o prodeji automobilů. Každý záznam obsahuje atributy: Id, cenu, rok výroby, počet najetých kilometrů, město prodeje, stát, název výrobce a model auta. Ukázka výstupu zobrazuje čistě spuštěnou aplikaci s dvěma tisíci auty a vypočtenou skyline křivku pro cenu/počet najetých kilometrů, kde důležitost je kladena na vysokou cenu a vysoký počet najetých kilometrů. V sekci “Selection” lze měnit výběr atributů a pomocí malé šipky u atributu i jeho důležitost (vyšší/nížší má být lepší). Tabulku pod grafem lze procházet posunem a řadit dle jednotlivých atributů při kliknutí na daný nadpis sloupce.



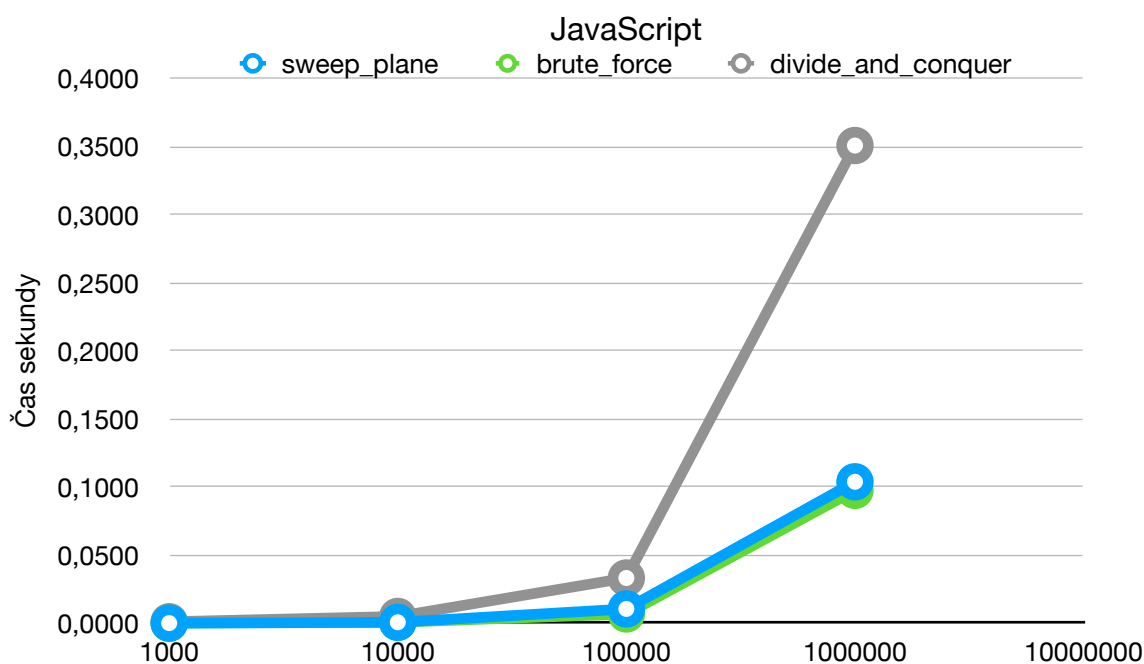
Obr. 2 - Ukázka desktopového rozhraní

## Experimentální sekce - zkoumání rychlosti a přesnosti

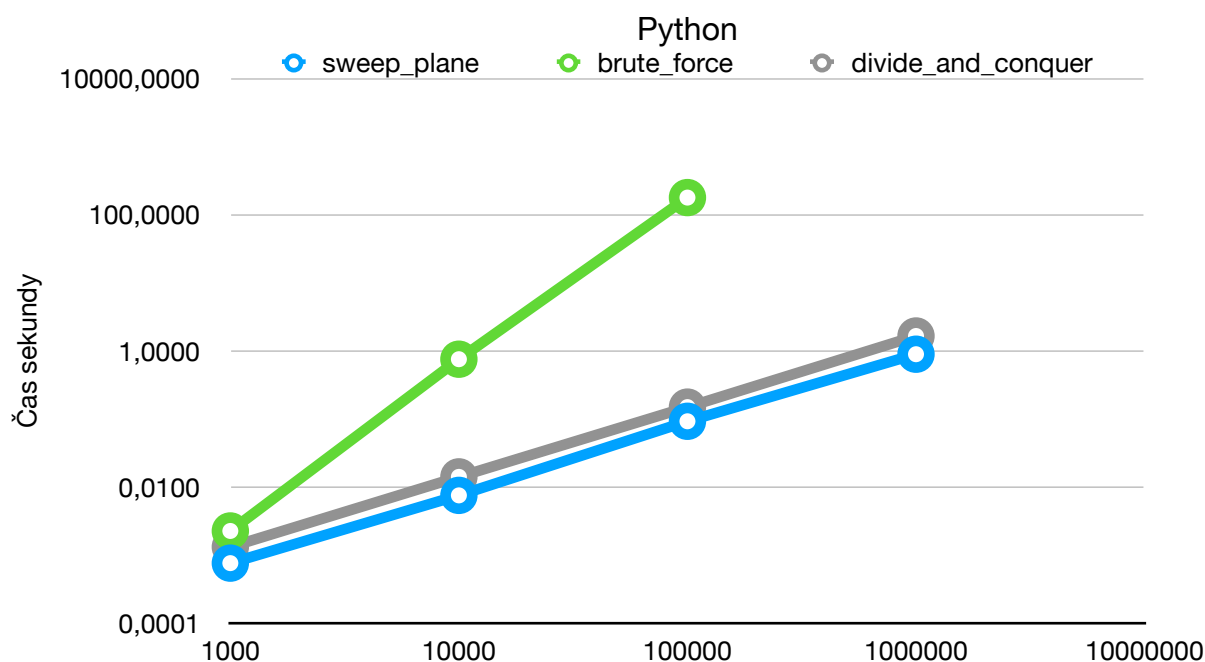
Implementovány jsou 3 algoritmy a všechny by měly spočítat stejný výsledek, protože jsou založeny na striktním porovnávání numerických hodnot. Nejpomalejší algoritmus je "Brute force" nebo-li výpočet hrubou silou. Klasický brute-force by měl složitost  $O(n^2)$ , já jsem ho trochu vylepšil, tak že jsem data nejprve seřadil a díky tomu se složitost snížila na  $O(n \cdot \log(n) + n \cdot n/2)$ .

Středně výkonný algoritmus je "Divide and conquer", který je rychlejší než brute-force, ale zase je náročnější na paměť, kvůli rekurzivnímu dělení dat na poloviny (vždy se vytvoří kopie). Jeho složitost je podle Master theoremu  $O(n \log n)$ .

Jako nejlepší řešení jsem implementoval "Sweep plane" algoritmus, který má stejnou složitost jako "Divide and conquer"  $O(n \log n)$ , ale nižší paměťovou náročnost.



Graf 1 - Výkonost JavaScriptu v závislosti na počtu

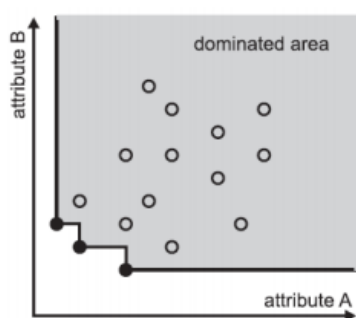


Graf 2 - Výkonost Pythonu v závislosti na počtu

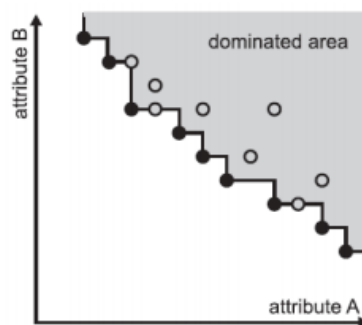
Naměřené časové údaje v případě Pythonu odpovídají předpokládané složitosti, kde u “Brute force” již při deseti tisíci prvků je čas přes 100 sekund. Ovšem u Javascript je výsledek úplně jiný, než se předpokládalo. “Brute force” je podle časové náročnosti téměř identický s algoritmem “Sweep plane”, přestože jejich odhad složitosti byl  $O(n \cdot \log(n) + n \cdot n/2)$  a  $O(n \log n)$ . Vzhledem k téměř identické implementaci algoritmů v obou jazycích, se jako nejpravděpodobnější příčina výsledného rozdílu jeví v Javascriptu engine V8 funkce JIT (Just in time compilation). Oba jazyky jsou sice interpretované, ale Javascript díky JIT dokáže za běhu vysoce optimalizovat kód, který se spouští opakovaně, což je právě případ “Brute force” řešení. JIT za běhu programu kompiluje bytecode na vysoce efektivní strojový kód, který způsobil v našem případě drastické zrychlení oproti pythonu interpreteru, který tuto funkci nemá.

## Diskuze - rozbor nedostatků a způsob řešení

U skyline operátoru může vzniknout několik problémů v závislosti na vstupních datech. Pokud vstupní data budou korelovaná, tak může vzniknout skyline pouze s několika málo body, v nejhorším případě pouze s jedním nad velkou vstupní množinou. Ve druhém extrému, kdy data budou anti-korelovaná, tak výstupní skyline může obsahovat až celou vstupní množinu. Oba případy budou problémem, v prvním uživateli doporučí málo produktů a ve druhém všechny. Problém s málo doporučenými body by se mohl vyřešit opětovným výpočtem skyline nad množinou bez předchozích doporučených produktů. V případě s moc velkou množinou doporučených bodů by se musel využít filtr na základě dalších atributů preferovaných uživatelem.



Obr. 3 - Skyline s korelovanými daty



Obr. 4 - Skyline s anti-korelovanými daty

## Závěr

V průběhu řešení byla implementována aplikace s uživatelským rozhraním, která bude uživateli doporučovat produkty z databáze produktů. Toto doporučení je realizováno pomocí 2D grafu se zobrazenou skyline křivkou. Výpočet skyline je řešen pomocí tří různých algoritmů: “Brute force”, “Divide and conquer” a “Sweep plane” u kterých jsem porovnával rychlost v závislosti na velikosti vstupu. Pro zajímavější porovnání jsem aplikaci implementoval jako webovou stránku v jazyce Javascript a potom jako desktopovou aplikaci v jazyce Python.

Při porovnání rychlostí se ukázalo, že řešení v JavaScriptu je řádově rychlejší než v Pythonu ve všech algoritmech. Velké překvapení bylo chování “Brute-force” algoritmu v JavaScriptu, který se vůči předpokladům ukázal jako nejrychlejší řešení, pravděpodobně kvůli JIT optimalizaci v JavaScript Engine.

Pro výpočet Skyline bych tedy příště použil JavaScript a algoritmus “Brute force”, protože se ukázal jako nejrychlejší řešení i při velkém množství dat.