boikomyk_knapsack_report_3

November 3, 2021

---

1 NI-KOP

Author: Mykyta Boiko

CTU login: boikomyk

1.1 Report #3

1.2 Stručný popis jednotlivých algoritmů

Dále budou následovat krátké popisy každého použitého algoritmu v experementech:

- Brute Force

- Brand & Bounds
- Dynamic programming (decomposition by weight)
- Dynamic programming (decomposition by price)
- Greedy
- Redux

Kazdy algoritmus je již do detailů popsaný a okomentovaný přímo v kódu. Tady budou uvedené pouze krátké popisy.

1.2.1 Brute Force

Jde vlastně o to, že procházíme/iterujeme přes všechny kombinace předmětů v batohu. V každém průchodu se vypočítá celková hmotnost a cena přidaných předmětů. Jakmile narážíme na vyhovující kombinaci, totiž celková hmotnost věci v batohu nebude přesahovat nosnost batohu a zároveň bude splněna podmínka na minimální cenu řešení, tím zastavujeme algoritmus. V nejhorším případě máme projít přes všechny kombinace a na konce zjistit, že kombinace věcí vyhovující podmínkám neexistuje.

Na začátku vypočítáme počet všech možných kombinací. Jelikož máme **0/1** problém batohu, jinak řečeno pokud bychom řešili konstruktivní problém, tak odpověď by měla vypadat ve tvaru posloupnosti čísel 0/1 délky n. (např. pro n=8 : 01010011). Čteme to tak, že do batohu přidáváme předměty 2,4,7 a 8. Pokud v binárním zápisu na nějaké pozice je 1 => přidáváme předmět do batohu, jinak nepřidáváme. Tím můžeme jednoduše spočítat celkový počet všech kombinaci: Pro

každý předmět: **1** - přidat, **0** - nepřidávat. Celkem máme 2 varianty. Pro n předmětů celkem máme 2^n.

Iterujeme od *combinations*:=0…2^n. V každé iteraci podle pozice všech 1 v binární reprezentaci *combinations* přidáváme předměty do batohu. Pokud je kombinace validní, t.j. nosnost batohu není překročena, tak jdeme dál a kontrolujeme podmínku na minimální cenu.

Jak již bylo uvedeno výše, v nejhorším případě máme projít přes všechny kombinace předmětů v batohu a to pokud jenom poslední kombinace je validní nebo hledaná kombinace neexistuje.

1.2.2 Branch & Bounds

Toto řešení vychází z původního Brute Force řešení. Tentokrát stavíme strom všech možných kombinací řešení a procházíme jej do hloubky s tím, že ořezáváme podstromy, které nevedou ke správnému řešení. Na to dáváme podmínku, která nám v každém kroce říká, zda maximální teoretická cena batohu, kterou při daně konfiguraci můžeme dosáhnout, je lepší, než cena již nalezeného lepšího řešení.

Pro každou z podtříd řešení se vypočítá horní mez(angl. upper bound) maximální hodnoty cenové funkce dosažené řešeními patřícími do podtřídy. Na základě těchto horních mezí se provádí další fáze větvení, vypočítávají se nové horní meze atd. Dokud se nakonec nedostaneme řešení, které má hodnotu cenové funkce větší než horní hranice všech podtříd a také vyšší než hodnoty cenové funkce pro všechna dříve získaná řešení.

Maximální teoretickou cenu dostáváme tak, že naplňujeme batoh na maximum s tím, že postupně bereme předměty seřazené podle poměru cena/váha. V případě, pokud nějaký předmět do batohu nepřidáváme, tak připočteme jeho poměrnou část ceny. Každý uzel stromu má dva syny a zpravidla levý syn je definován operaci přidávání dalšího předmětů ze seřazené posloupnosti. Pravý syn je definován kombinaci, ve které další předmět ze seřazené posloupnosti nepřidáváme.

Taky neprocházíme větve stromů, ve kterých překročíme nosnost batohu. V nejhorším případě zase budeme muset projít celý strom. Proto složitost algoritmu je exponenciální. Doba běhu je velmi závislá na tom jak dobře se povede stavový prostor prořezat.

1.2.3 Dynamic programming

Hlavní myšlenka **Dynamic Programming** v případě problémů batohu je použití tabulky kam se ukládají řešení všech podproblému (všechny možné hodnoty cen/vah, v závislosti na zvoleném typu dekompozici, které mohou vzninkout při přidání věci do batohu). Pokud v průběhu řešení znovu narážíme na již řešeny podproblém, tak stačí vzít řešení v tabulce, aniž byste podproblém museli znovu řešit.

Je třeba poznamenat, že nalezené řešení je vždy optimální, neboť algoritmus vytvoří tabulku pro celý strom všech možných výsledků. Stejně je nutné říct, že algoritmus je pseudopolynomiální, jelikož doba běhu je **O(NW)** nebo **O(N*sum_of_all_items_prices) **pro neomezený problém batohu s** N **položkami a batohem velikosti** W **(**W** však není polynomiální vůči délce vstupu). Což už klidně umožňuje Nám odpovědět na jeden z kladených dotazů, že složitost algoritmu závisí na parametru, který nemá nic společného s velikostí.

Decomposition by weight V případě dokompozice podle váhy tabulka má nalsedujci parametry: - počet řádku je stejný jako celkový počet věci daný pro danou instanci problémů

- počet sloupců je rovný kapacitě/nosnosti batohu (maximlani povolena váha)

Jinak řečeno, to znamená, že každý řádek reprezentuje řešení problémů pro počet předmětů, který se rovná indexu řádku. Stejně tak sloupce reprezentují všechny možné výsledky vah (pro všechny kombinace všech správných řešení). Což vede na to, že buňka[i,j] reprezentuje cenu takové kombinace. Finální výsledek je uloženy v posledním řádku (maximální hodnota v posledním řádku).

Tabulka se staví následovně: (Zkrácený kód..)

```
for item_index in range(items_indexes_rows_cnt + 1):
    for capacity_index in range(capacity_columns_cnt + 1):
        ...
        #  if weight of current item is lower or equal to the remaining knapsack's capacity th
        if current_item.weight <= capacity_index:
            # return the maximum of two cases:
            # - current item included or
            # - not included
            memory_table[item_index][capacity_index] = max(
                current_item.price + memory_table[item_index - 1][capacity_index - current_ite
                memory_table[item_index - 1][capacity_index]
            )
        # otherwise current proceeding item cannot be included
        else:
            memory_table[item_index][capacity_index] = memory_table[item_index - 1][capacity_i
```

Decomposition by price V podstatě přístup k řešení je skoro ten samy, avšak je nutné poznamenat další důležité rozdíly. Tabulka má nalsedujci parametry:

- počet řádku je stejný jako celkový počet věci daný pro danou instanci problémů
- počet sloupců je rovný sumě cen všech věci daně instance problému.

Finální výsledek je zase uloženy v posledním řádku (maximální hodnota, avšak je nutné aby byla splněna podmínka: *hodnota_buňky <= knapsack_capacity*)

Tabulka se staví zase následovně: (Zkrácený kód..):

```
for knapsack_current_price in range(2, items_total_sum + 1):
    for item_index in range(1, items_total_count + 1):
        ...

        index = knapsack_current_price - current_item.price
        if index < 0:
            # if index is negative, the weight remains the same
            memory_table[knapsack_current_price][item_index] = previous_weight
        else:
            # otherwise
            # resolve weight to add (zero or weight of current proceeding item)
            weight_to_add = current_item.weight if memory_table[index][item_index - 1] != infi
            # W(c, i) = min(W(c, i - 1), W(c, i - 1 ) + w_c) for any c > 0
            memory_table[knapsack_current_price][item_index] = min(previous_weight, memory_tabl
```

1.2.4 Greedy

Hlavni myslenka **Greedy** pristupu k reseni je super jednoducha: seřadit všechny veci sestupně podle jejich poměru:

$$pomr = \frac{cena}{vaha}$$

a přidávat veci postupně, dokud není batoh plný.

Je nejrychlejší metodou pro řešení batohu, její složitost závisí na složitosti použitého řazení (zpravidla je **O(nlogn)**).

Ale není zandá 100% garance, že nalezené řešení bude optimální, neboť třeba pouze jedna věc s maximální cenou a zároveň vahou může být řešením, ale kvůli nízkému poměru nejspíš bude ignorována algoritmem. (Tento případ řeší modifikace **Redux**, která bude popsána trošku dál)

U tohoto algoritmu je nutné zavést další pojem: **Relativní chyba**, která se vypočítá následovně:

$$chyba = \frac{|cena_{een} - cena_{reference}|}{max(cena_{een}, cena_{reference})}$$

1.2.5 Redux

Redux je vylepšením(/modifikace) předchozí heuristiky (vlastně řeší ten případ s jednou věci s maximální cenou a vahou).

Nejdřív pro danou instanci nalezneme řešení popsanou **Greedy** heuristikou. Pak vytvoříme další batoh s jednou nejrdazsi věci a porovnáme obě řešení. Lepší z těchto řešení (s větší cenou nebo stejnou cenou ale menší vahou) je pak řešením finálním.

1.3 Předzpracování a generování dát

V této kapitole budou provedené experimentální vyhodnocení závislosti kvality řešení a výpočetní náročnosti algoritmů na následujících parametrech instancí: - poměru kapacity batohu k sumární váze - korelaci cena/váha - rozložení vah a granularitě

Dále budou definované pomůcné funcke pro generování testovacích(vstupních) dát, řešení a vykl-resleni výsledků experimentů.

[1]:
```
# import scope
import numpy as np
import pandas as pd
from io import StringIO
import matplotlib
import seaborn as sns
import matplotlib.pyplot as plt
from typing import Dict

%matplotlib inline
```

Pro generování vstupních dát použijeme nabízeny **KG2** generátor. Pro výpočet relativní chyby taky potřebujeme nagenerovat příslušné řešení. Jako generátor řešení potřebujeme optimální aloritmus,

který generuje reseni s nulovou relativní chybou, což jsou čtyři algoritmy: **Brute Force**, **Branch & Bounds** a oba dva **Dynamic Programming**.

Vybereme **Brute Force**. (*REFERENCE_GENERATOR = 'bf'*)

[2]:
```
# common constants scope

ALGORITHMS = {
    'bf'    : 'Brute Force',
    'bb'    : 'Branch & Bounds',
    'dp_dw' : 'Dynamic Programming (decomposition by weight)',
    'dp_dp' : 'Dynamic Programming (decomposition by price)',
    'greedy': 'Greedy',
    'redux' : 'Redux'
}

REFERENCE_GENERATOR = 'bf'

# path to solver executor
SOLVER = '../../knapsack/main.py'
```

kg2 generátor povinné parametry:

```
-n počet věcí
-N počet instancí
-W max. váha věci
-C max. cena věci
```

Za velikost instanci dosadíme **10** a počet instanci bude **500**. Toho by mělo stačit, neboť testování vyžaduje spouštění každého z algoritmů.

[3]:
```
# constants scope
OUTPUTS_DIR = 'data'

# path to kg2 executor
KG2 = 'gen-1.1_mac/kg2'
# path to kg_perm executor
KG_PERM = 'gen-1.1_mac/kg_perm'

SIZE       = 10
MAX_WEIGHT = 1000
MAX_PRICE  = 1000


KG2_ARGS = {
    '-n': SIZE,
    # default count of instances (as in previous homeworks)
    '-N': 500,
    '-W': MAX_WEIGHT,
```

```
    '-C': MAX_PRICE
}
```

[4]:
```
# prepare help functions for data predprocessing
def read_and_store_buffer_input_to_df(buffer, algorithm: str, instance_size:␣
 ↪int):
    # read csv input
    df = pd.read_table(buffer, delimiter="\t", index_col=False)
    # set additinial columns
    df['INSTANCE_SIZE'] = instance_size
    df['ALGORITHM'] = algorithm
    return df
```

[56]:
```
# prepare help functions
def generate_kg2_instances_and_solve(
    parameter_with_range_to_test: Dict[str,str],
    kg2_args: Dict[str,str] = KG2_ARGS,
    log_events = True
):
    # create dir if it doesn't exist
    !test -d $OUTPUTS_DIR || mkdir $OUTPUTS_DIR

    # get valid args str
    kg2_args_str = " ".join(map(lambda arg: f"{arg[0]} {arg[1]}", kg2_args.
 ↪items()))

    # prepare storage for algorithms outputs
    algorithms_outputs = {}

    parameter_name = next(iter(parameter_with_range_to_test.keys()))
    for parameter_value in next(iter(parameter_with_range_to_test.values())):
        if log_events: print(f'\x1b[1;32m[KG2]\x1b[0m: testing parameter \x1b[1;
 ↪31m {parameter_name}={parameter_value}\x1b[0m')


        # prepare kg2 args (inlcuding param to test)
        kg2_args_with_test_param = f'{kg2_args_str} {parameter_name}␣
 ↪{parameter_value}'

        # prepare path for input and reference files:
        # - input: KG15_inst.dat
        # - ref  : KG15_sol.dat
        input_file_path = f'{OUTPUTS_DIR}/KG{kg2_args["-n"]}_inst.dat'
        reference_file_path = f'{OUTPUTS_DIR}/KG{kg2_args["-n"]}_sol.dat'
        if log_events: print(f' - Generating inputs..')
        # execute KG2 generator and generate input instance
        !$KG2 $kg2_args_with_test_param > $input_file_path
```

```
        if log_events: print(f' - Generating references.. using␣
↪"{ALGORITHMS[REFERENCE_GENERATOR]}"')
        # execute solver to generate reference
        !python3.9 $SOLVER $REFERENCE_GENERATOR -cnt=1 -in=$input_file_path -o␣
↪> $reference_file_path

        if log_events: print(f' - Solving input instance using algorithms: ',␣
↪end='')
        # solve input instances using different alogrithms
        for algorithm in ALGORITHMS.keys():
            # ala progress bar
            if log_events: print(f"{algorithm}, ", end="") if algorithm !=␣
↪list(ALGORITHMS.keys())[-1] else print(f"{algorithm}..\n", end="")

            if not algorithm in algorithms_outputs.keys():␣
↪algorithms_outputs[algorithm] = []

            # solve input instace and obtain solution
            solution = !python3.9 $SOLVER $algorithm -cnt=1␣
↪-in=$input_file_path -ref=$OUTPUTS_DIR -b

            # store solution output to dataframe
            df = read_and_store_buffer_input_to_df(
                buffer=StringIO(solution.n),
                algorithm=algorithm,
                instance_size=kg2_args["-n"]
            )
            # update dataframe with corresponding column representing testing␣
↪parameter
            df[parameter_name]=parameter_value
            algorithms_outputs[algorithm].append(df)

    # concatenate all dataframes of certain alogrithm to single dataframe
    for algorithm in ALGORITHMS.keys():
        algorithms_outputs[algorithm] = pd.
↪concat(algorithms_outputs[algorithm]).reset_index()

    return algorithms_outputs
```

[50]:
```
def plot_basic_analytics(
    algorithms_outputs: Dict[str, pd.DataFrame],
    index_column: str,
    parameter_label: str
):
    # dataset preprocessing: concatenation, renaming etc
```

```
    common_df = pd.concat(list(algorithms_outputs.values())).reset_index()
    common_df['ALGORITHM'] = common_df['ALGORITHM'].apply(lambda algorithm_key:␣
↪ALGORITHMS[algorithm_key])

    # define several pivot tables for different analytic cases
    spawn_pivot_table = lambda value, aggfunc: pd.pivot_table(
        data=common_df,
        values=value,
        index=index_column,
        columns='ALGORITHM',
        aggfunc=np.mean
    )

    # Mean for: time, steps, error
    time_mean_table = spawn_pivot_table(value='TIME[ms]', aggfunc=np.mean)
    steps_mean_table = spawn_pivot_table(value='STEPS', aggfunc=np.mean)
    error_mean_table = spawn_pivot_table(value='ERROR', aggfunc=np.mean)

    _, axes = plt.subplots(nrows=1, ncols=2, figsize=(15,7))
    # decrease size of legend
    plt.rc('legend',fontsize=9)


    ax = steps_mean_table.plot(title=f"Průměr počet kroků VS␣
↪{parameter_label}", ax=axes[0])
    ax.set_ylabel('Průměr počtu kroků')
    ax = error_mean_table.plot(title=f"Průměr relativní chyby VS␣
↪{parameter_label}", ax=axes[1])
    ax.set_ylabel('Průměr relativní chyby')
    ax = time_mean_table.plot(title=f"Průměr času v mikrosekundech VS␣
↪{parameter_label}", figsize=(15,7))
    ax.set_ylabel('Průměr času v mikrosekundech')
    plt.show()
```

1.4 Analýza a interpretace dat

1.4.1 Závislosti kvality řešení a výpočetní náročnosti algoritmů na parametrech instancí

1.4.2 poměr kapacity batohu k sumární váze

Následující test se zabývá citlivosti na poměr kapacity batohu k sumární váze, označujeme **m**. Pro testování zvolíme interval <0.1, 2.0>.

[57]:
```
kg2_args = KG2_ARGS.copy()
m_parameter = {
    # range -m <0.1, 2.0>
    '-m': [ float(f'{(x*0.1):.1f}') for x in range(1, 21)]
```

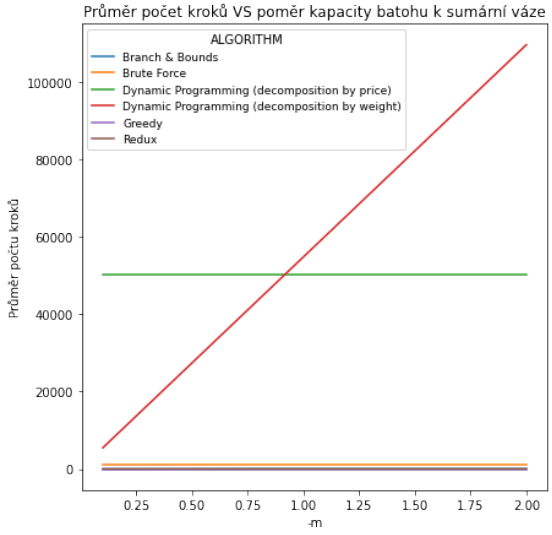
```
}
algorithms_outputs = generate_kg2_instances_and_solve(
    parameter_with_range_to_test=m_parameter,
    kg2_args=kg2_args
)
```

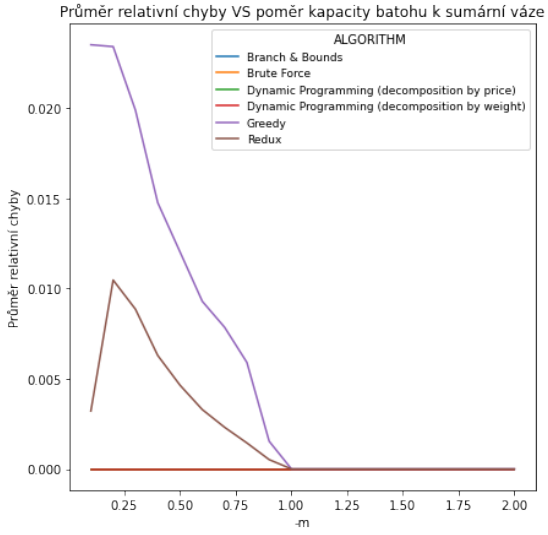
```
[KG2]: testing parameter  -m=0.1
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -m=0.2
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -m=0.3
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -m=0.4
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -m=0.5
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -m=0.6
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -m=0.7
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -m=0.8
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -m=0.9
```

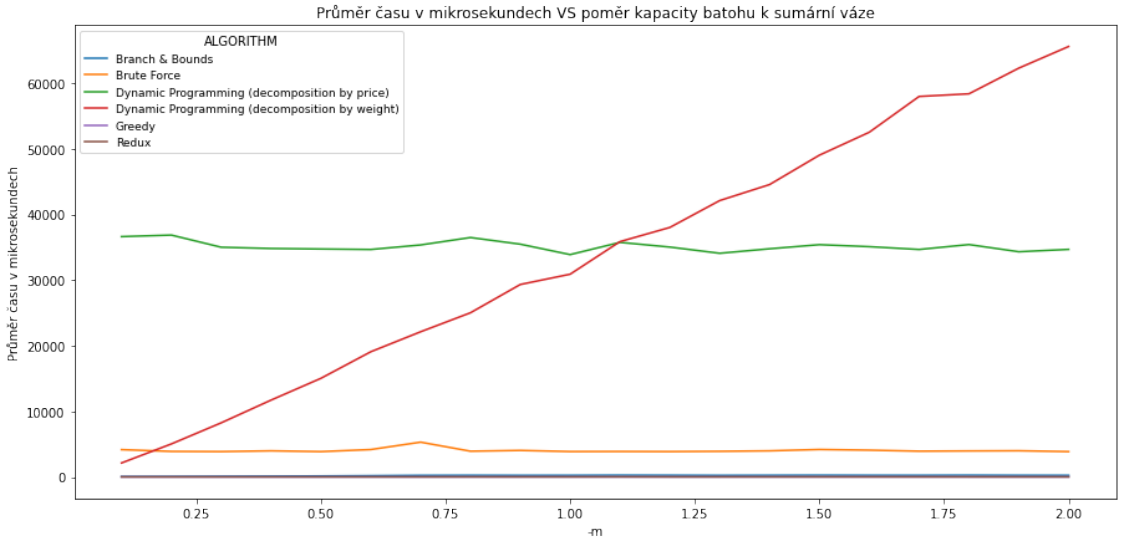
```
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -m=1.0
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -m=1.1
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -m=1.2
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -m=1.3
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -m=1.4
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -m=1.5
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -m=1.6
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -m=1.7
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -m=1.8
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
```

```
redux..
[KG2]: testing parameter  -m=1.9
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -m=2.0
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
```

[280]:
```
plot_basic_analytics(
    algorithms_outputs=algorithms_outputs,
    index_column=next(iter(m_parameter)),
    parameter_label='poměr kapacity batohu k sumární váze'
)
```





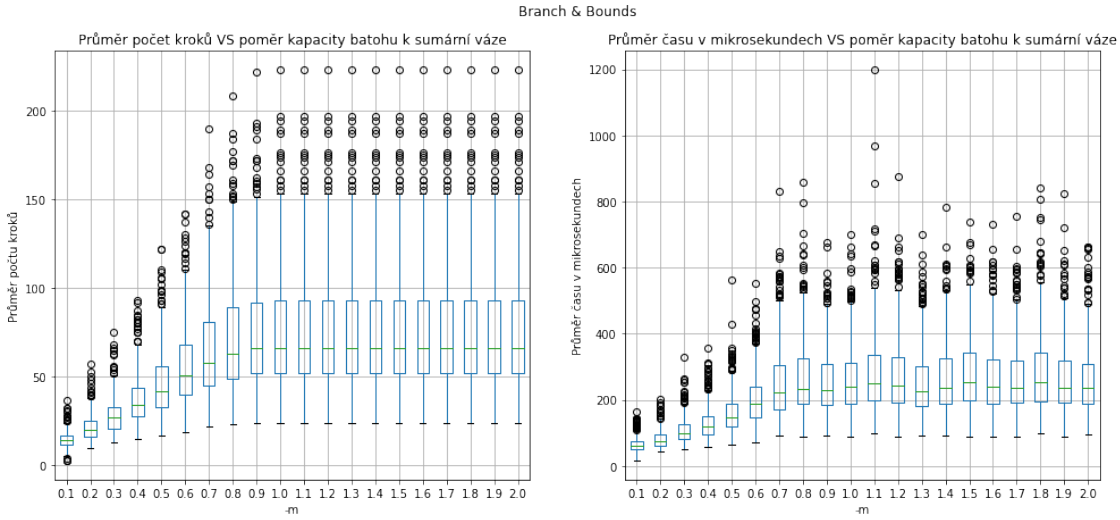


Probereme a provedeme analýzu vykreslených grafů.

_Průměr počtu kroků

Z prvního grafu *'Průměru počtu kroku'* pozorujeme několik zajímavých věci. **Dynamic programming** s dekompozici podle váhy roste lineárně, protože algoritmus je závislý na kapacity(/nosnosti batohu), neboť potřebujeme větší počet kroku pro stavbu dynamické tabulky(kde počet sloupců je rovný kapacitě/nosnosti batohu). Co se týče **Branch & Bounds**, tak kvůli velkému intervalu na ose y, není viditelný průběh vývoje přímky. Proto potřebujeme vykreslit vývoj **Branch & Bounds** do zvláštního grafu. Stejně tak vedle vyklreslime graf vývoje času, neboť že společného grafu taky ten vývoj není viditelný.

[290]:
```
_, axes = plt.subplots(nrows=1, ncols=2, figsize=(16,7))

ax = algorithms_outputs['bb'].boxplot(column='STEPS', by='-m', ax=axes[0])
ax.set_ylabel('Průměr počtu kroků')
ax.set_xlabel('-m')
ax.set_title("Průměr počet kroků VS poměr kapacity batohu k sumární váze")


ax = algorithms_outputs['bb'].boxplot(column='TIME[ms]', by='-m', ax=axes[1])
ax.set_ylabel('Průměr času v mikrosekundech')
ax.set_xlabel('-m')
ax.set_title("Průměr času v mikrosekundech VS poměr kapacity batohu k sumární␣
 ↪váze")

plt.suptitle('Branch & Bounds')
plt.show()
```



S větší kapacitou musí **Branch & Bounds** procházet větší prostor řešení, což zvyšuje počet kroků, které algoritmus provede. U grafu času také vykazuje určitou závislost, protože používá heuristiku, která je závislá na kapacitě. U obou grafu pozorujeme růst do m=1.0, pak už je konstantní průběh. **Greedy** a **Redux** mají konstantní růst, neboť počet věci je vždy 10.

_Průměr času

U **Brute Force** vidíme konstantní růst, což je logicky z důvodů iteraci přes všechny možné kombinace řešení, ale taky vidíme divnou na intervalu **m** <0.6,0.8>. Spiš bylo způsobeny přetížením počítače (v důsledku zatížení RAM nebo něco podobného). Taky vypadá, že **Brute Force** a **Branch & Bounds** jsou rychlejší než oba dva algoritmy **Dynamic Programming**, neboť velikost prostoru prohledávání je relativní menší než maximální cena a kapacita batohu. Stejně jako u grafu počtu kroku, vidíme, že růst času u **Dynamic Programming** s dekompozici podle váhy je lineární. **Greedy** a **Redux** mají konstantní růst, neboť počet věci se nemění.

_Průměr relativní chyby

Všechny algoritmy jsou optimální a vždy najdou nejlepší řešení, kromě dvou heuristických algoritmů: **Greedy** a **Redux**. Vidíme, že **Redux** nadstavba nad jednoduchým **Greedy** algoritmem docela silné znizuje relativní chybu. U obou heuristik s rostoucí kapacitou dojde ke zlepšení, protože bude možné přidavat více věci bez chyb. Od **m**=1.0 relativní chyba už je nulová (neboť do batohu už vejdou všechny věci).

1.4.3 korelaci cena/váha

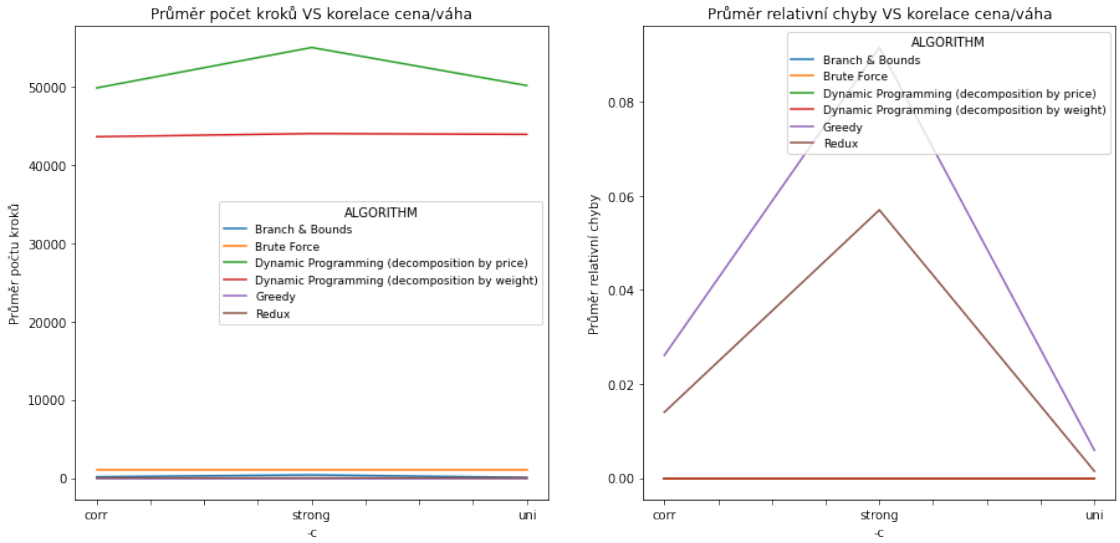
Následující test se zabývá citlivosti na korelaci mezi cenou a vahou, označujeme **c**.

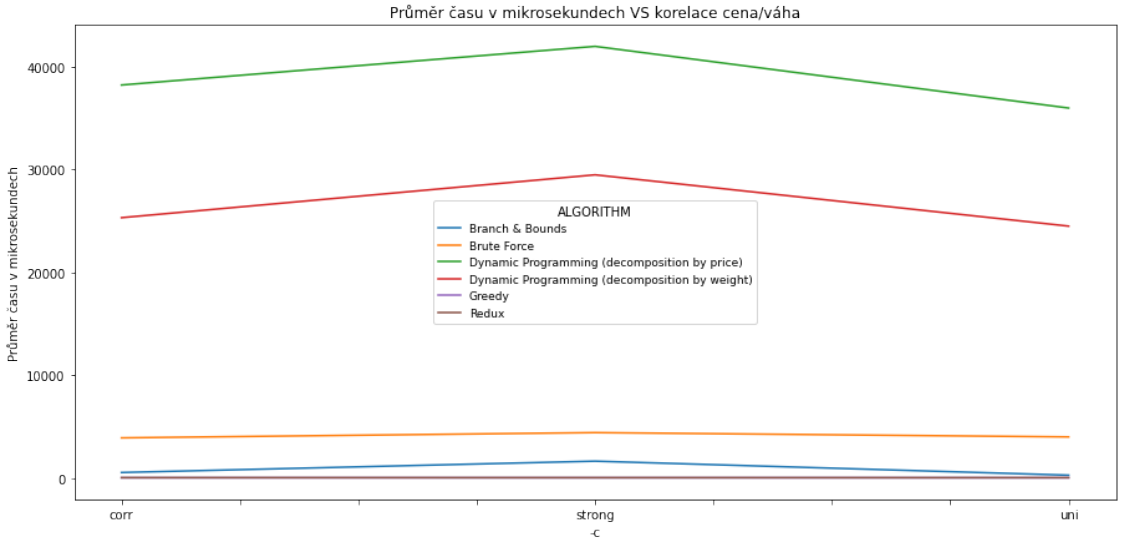
```
-c korelace s váhou
    * uni  žádná
    * corr: menší
    * strong: silná
```

[292]:
```
kg2_args = KG2_ARGS.copy()
c_parameter = {
    '-c': ['uni', 'corr', 'strong']
}
algorithms_outputs = generate_kg2_instances_and_solve(
    parameter_with_range_to_test=c_parameter,
    kg2_args=kg2_args
)
```

```
[KG2]: testing parameter  -c=uni
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -c=corr
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -c=strong
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
```

[297]:
```
plot_basic_analytics(
    algorithms_outputs=algorithms_outputs,
    index_column=next(iter(c_parameter)),
    parameter_label='korelace cena/váha'
)
```





Na těchto grafech je snadno vidět, jak korelace mezi cenou a hmotnosti přímo ovlivňují výkonnost algoritmů.

_Průměr počtu kroků

Vidíme, že u **Dynamic Programming** s dekompozici podle ceny počet kroku se zvyšuje při silné korelaci. Silná korelace znamená, poměr váhy k ceně se bude pochybovat kolem 1, a tím se nejspíš zvyšuje celková cena věci a následovně i velikost dynamické tabulky.

_Průměr času

Taky vidíme, že silná korelaci ovlivňuje oba dva algoritmy **Dynamic Programming**. Důvod je stejný jak i u grafu počtu kroku. **Branch & Bounds** taky je ovlivněny silnou korelaci, neboť používá heruistiku pro ražení věci podle jejich poměru cena/váha.

_Průměr relativní chyby

Co se týče velikosti relativní chyby, tak zase silná korelace ovlivňuje heuristické algoritmy: **Greedy** a **Redux**. Velikost chyby je výrazně větší při silné korelaci a to vlastně je způsobeny tím, že tak samé se používá ražení podle poměru váha/cena. Výsledné pořadí věci po ražení nebude moc přesně.

1.4.4 rozložení vah a granularitě

Následující test se zabývá citlivosti na rozložení vah a granularitě **w** a **k**.

```
-w převaha lehkých/těžkých věcí
    * light: lehkých
    * heavy: těžkých
-k exponent granularity
```

[301]:
```
k_parameter = {
    # interval <0.5,3.0>
    '-k': [ float(f'{(x*0.1):.1f}') for x in range(5, 31)]
}
```

převaha lehkých věcí

[302]:
```
kg2_args = KG2_ARGS.copy()
kg2_args['-w'] = 'light'

algorithms_outputs = generate_kg2_instances_and_solve(
    parameter_with_range_to_test=k_parameter,
    kg2_args=kg2_args
)
```

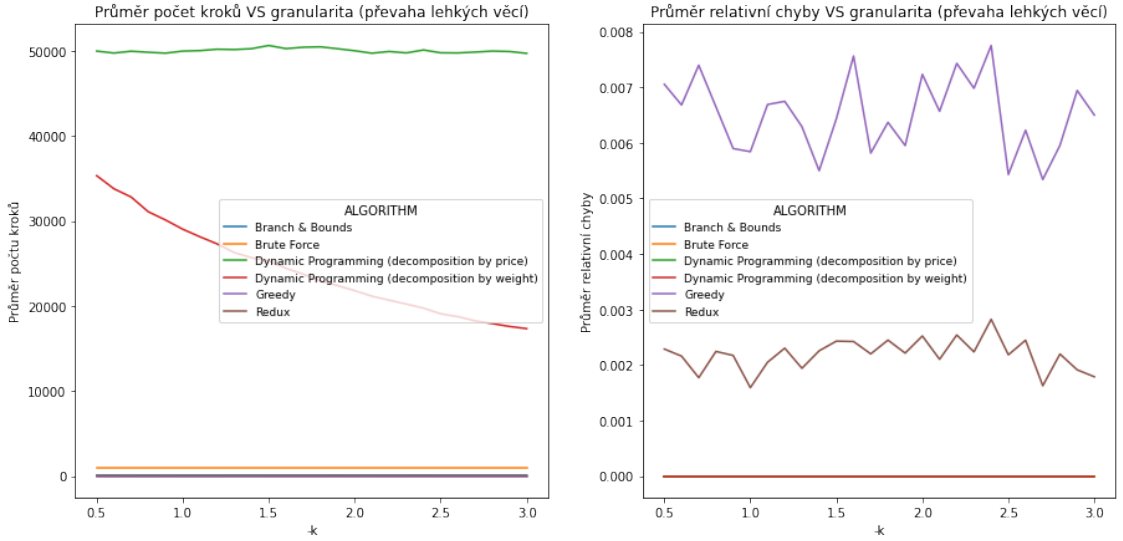
```
[KG2]: testing parameter  -k=0.5
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -k=0.6
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -k=0.7
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -k=0.8
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -k=0.9
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -k=1.0
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -k=1.1
 - Generating inputs..
 - Generating references.. using "Brute Force"
```

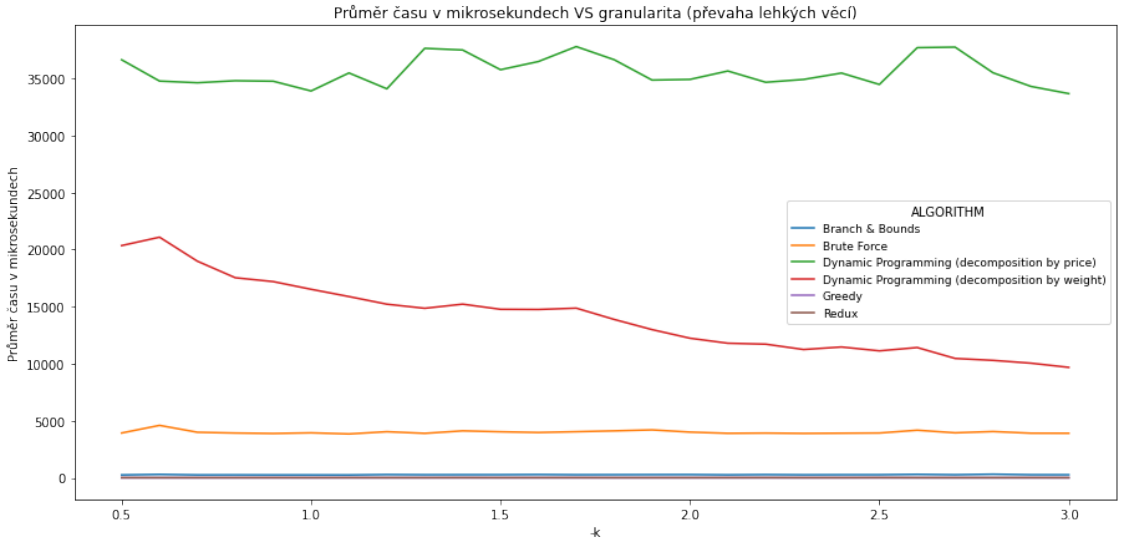
```
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -k=1.2
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -k=1.3
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -k=1.4
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -k=1.5
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -k=1.6
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -k=1.7
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -k=1.8
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -k=1.9
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -k=2.0
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -k=2.1
```

```
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -k=2.2
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -k=2.3
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -k=2.4
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -k=2.5
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -k=2.6
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -k=2.7
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -k=2.8
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -k=2.9
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
[KG2]: testing parameter  -k=3.0
 - Generating inputs..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
```

redux..

[304]:
```
plot_basic_analytics(
    algorithms_outputs=algorithms_outputs,
    index_column=next(iter(k_parameter)),
    parameter_label='granularita (převaha lehkých věcí)'
)
```





_Průměr počtu kroků

U **Dynamic Programming** s dekompozici podle váhy je vidět skoro lineární závislost poměru lehkých věci a počtu kroků. Jelikož počet kroku je skoro rovný velikosti dynamické tabulky, tak

jde říct, že celková nonost batohu s zmenšuje se zvýšením parametrů **k**.

_Průměr času

U **Dynamic Programming** s dekompozici podle váhy je vidět skoro lineární závislost poměru lehkých věci a času. Pokud je většina věcí lehká, tak se spodní část dynamické tabulky bude plnit intenzivněji a tím se ořízne více kombinace řešení. A celkový výpočet bude rychlejší. Takže větší počet lehkých věci zmenšuje dobu zpracování.

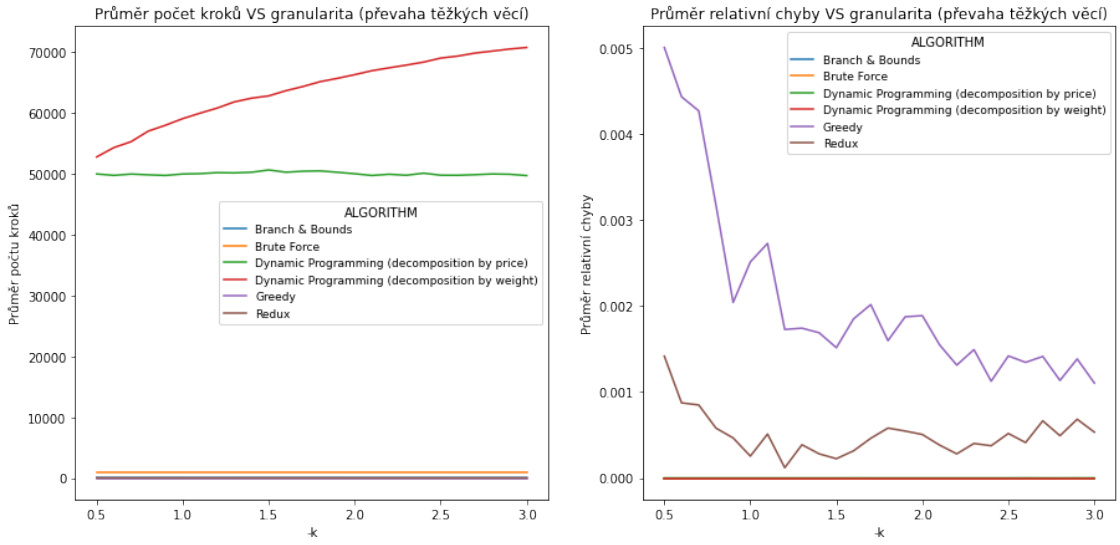
_Průměr relativní chyby

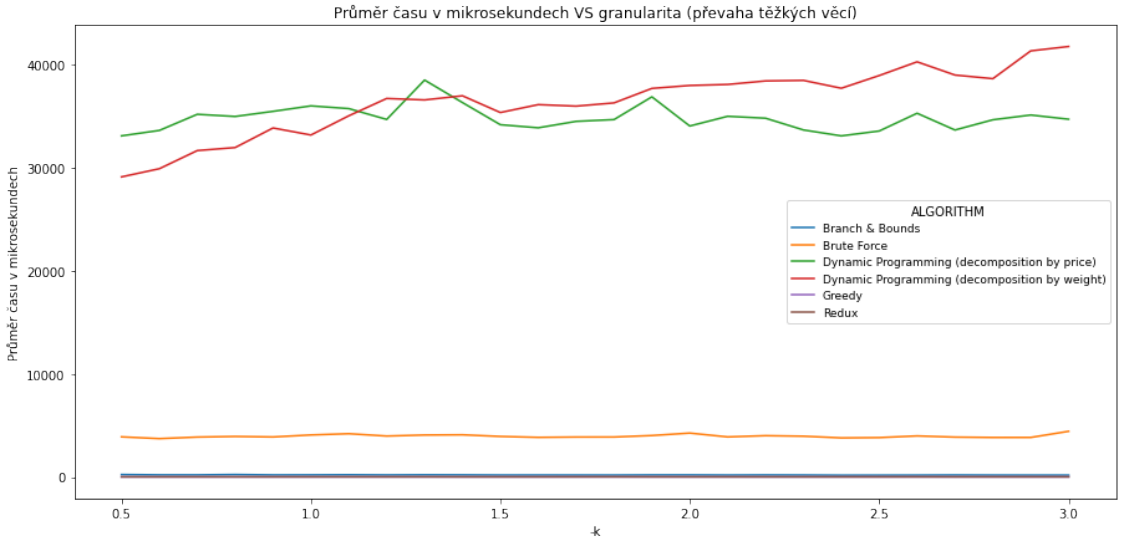
U obou heuristických algoritmů **Greedy** a **Redux** je docela velká relativní chyba.

převaha těžkých věcí

[305]:
```
kg2_args = KG2_ARGS.copy()
kg2_args['-w'] = 'heavy'

algorithms_outputs = generate_kg2_instances_and_solve(
    parameter_with_range_to_test=k_parameter,
    kg2_args=kg2_args,
    # this time without logging
    log_events=False
)
```

[307]:
```
plot_basic_analytics(
    algorithms_outputs=algorithms_outputs,
    index_column=next(iter(k_parameter)),
    parameter_label='granularita (převaha těžkých věcí)'
)
```





_Průměr počtu kroků

To samé, co i lehkých věci, ale trend je obrácený.

_Průměr času

To samé, co i lehkých věci, ale trend je obrácený. **Dynamic Programming** s dekompozici podle váhy u těžkých věci vidíme nárůst časově složitosti.

_Průměr relativní chyby

Mnohem lepší je situace u heuristických algoritmů **Greedy** a **Redux**. S větším **k** se zmenšuje relativní chyba.

1.4.5 maximální cena a váha

Taky ještě provedeme nepovinné dodatečně testy citlivosti na maximální ceně a váze (vlastně zbytek dosud neotestovaných parametrů)

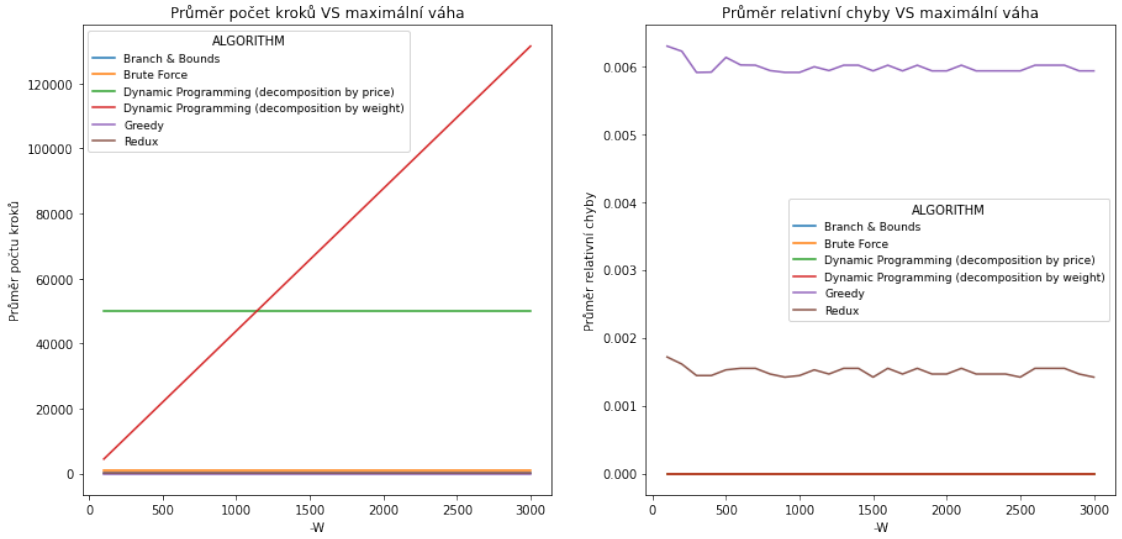
`-W max. váha věci`
`-C max. cena věci`

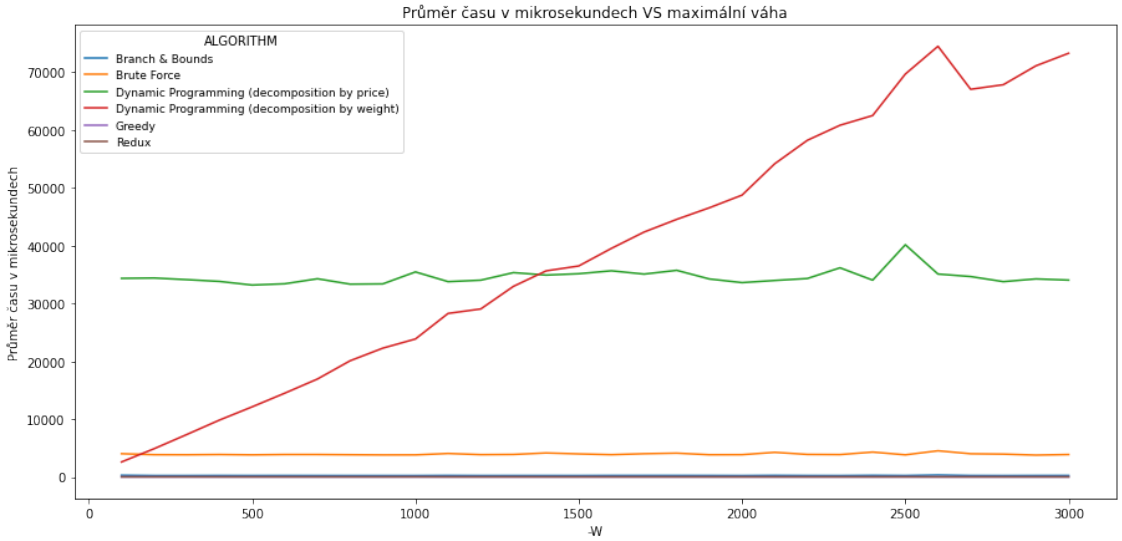
maximální váha

[312]:
```
kg2_args = KG2_ARGS.copy()
if '-W' in kg2_args: del kg2_args['-W']
W_parameter = {
    # interval <100,3000>
    '-W': [ x*100 for x in range(1, 31)]
}


algorithms_outputs = generate_kg2_instances_and_solve(
```

```
    parameter_with_range_to_test=W_parameter,
    kg2_args=kg2_args,
    # without logging
    log_events=False
)
```

[313]:
```
plot_basic_analytics(
    algorithms_outputs=algorithms_outputs,
    index_column=next(iter(W_parameter)),
    parameter_label='maximální váha'
)
```





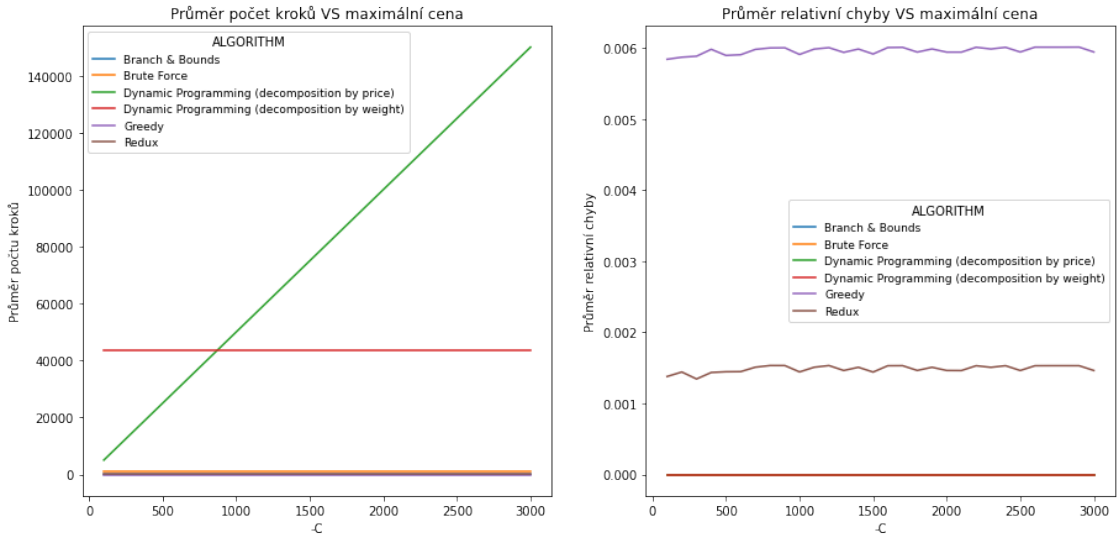
Mužů poukázat pouze na dva zajímavé body. Z grafu času a počtu kroků je vidět lineární růst u **Dynamic Programming** s dekompozici podle váhy, jelikož, jak již bylo zmíněno dříve, počet sloupců je rovný kapacitě/nosnosti batohu a s rostoucí maximální vahou roste i velikost dynamické tabulky.

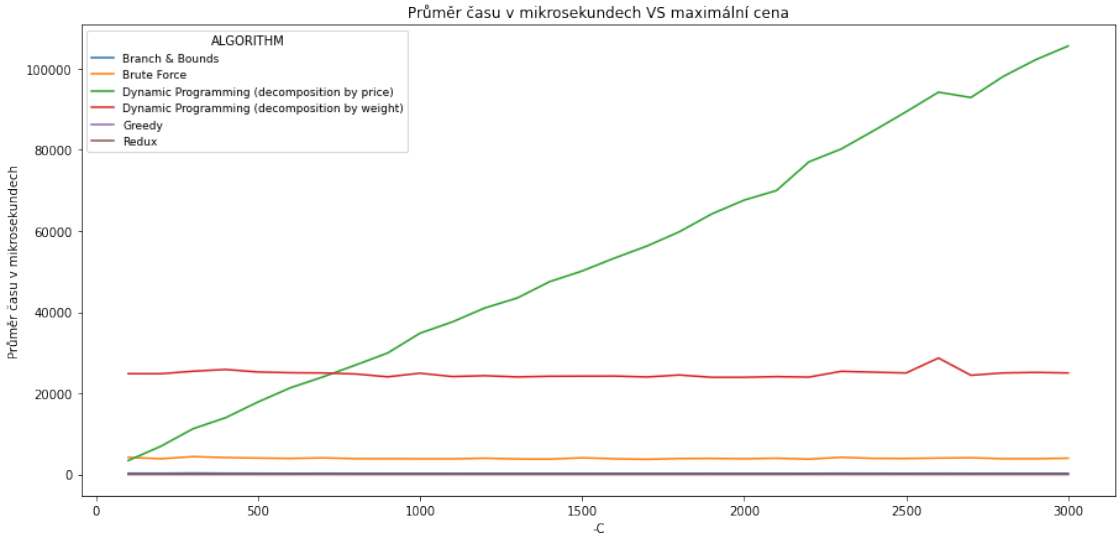
Co se týče grafu relativní chyby, tak se rosouci maximální vahou relatvni chyba se nemění u obou heuristických algoritmů.

maximální cena

[315]:
```
kg2_args = KG2_ARGS.copy()
if '-C' in kg2_args: del kg2_args['-C']
C_parameter = {
    # interval <100,3000>
    '-C': [ x*100 for x in range(1, 31)]
}


algorithms_outputs = generate_kg2_instances_and_solve(
    parameter_with_range_to_test=C_parameter,
    kg2_args=kg2_args,
    # without logging
    log_events=False
)
```

[316]:
```
plot_basic_analytics(
    algorithms_outputs=algorithms_outputs,
    index_column=next(iter(C_parameter)),
    parameter_label='maximální cena'
)
```





Platí skoro to samé, co i pro maximální váhu. Z grafu času a počtu kroků je vidět lineární růst u **Dynamic Programming** s dekompozici podle ceny a důvod je ten samy.

Relativní chyba taky se nemění s rostoucí maximální cenou.

1.4.6 Ověření robustnosti algoritmů

V této části provedeme testy robustnosti algoritmů. K testování robustnosti použijeme jednu instanci, kterou permutujeme.

Nejdřív nadefinujeme pomucnou funkci a pak provedeme příslušné experimenty.

[69]:
```
# prepare help functions
def generate_kg2_instance_and_permutate(
    number_of_permutations: int = 100,
    kg2_args: Dict[str,str] = KG2_ARGS,
    log_events = True
):
    # generate only one instance
    kg2_args['-N'] = 1

    # create dir if it doesn't exist
    !test -d $OUTPUTS_DIR || mkdir $OUTPUTS_DIR

    # get valid args str
    kg2_args_str = " ".join(map(lambda arg: f"{arg[0]} {arg[1]}", kg2_args.
 ↪items()))

    # prepare storage for algorithms outputs
```

```
    algorithms_outputs = {}

    if log_events: print(f'\x1b[1;32m[KG2]\x1b[0m: Number of permutations␣
↪\x1b[1;31m {number_of_permutations}\x1b[0m')

    # prepare path for input and reference files:
    # - input: KG15_inst.dat
    # - ref  : KG15_sol.dat
    input_file_path = f'{OUTPUTS_DIR}/KG{kg2_args["-n"]}_inst.dat'
    reference_file_path = f'{OUTPUTS_DIR}/KG{kg2_args["-n"]}_sol.dat'
    if log_events: print(f' - Generating {number_of_permutations} number of␣
↪permutations for single instance..')

#     number_of_permutations -= 1
    # execute KG2 generator and generate input instance, then redirect to␣
↪KG_PERM and generate permutations
    !$KG2 $kg2_args_str > $input_file_path
    !cat $input_file_path | $KG_PERM -d 1 -N $number_of_permutations >>␣
↪$input_file_path


    if log_events: print(f' - Generating references.. using␣
↪"{ALGORITHMS[REFERENCE_GENERATOR]}"')
    # execute solver to generate reference
    !python3.9 $SOLVER $REFERENCE_GENERATOR -cnt=1 -in=$input_file_path -o >␣
↪$reference_file_path

    if log_events: print(f' - Solving input instance using algorithms: ',␣
↪end='')
    # solve input instances using different alogrithms
    for algorithm in ALGORITHMS.keys():
        # ala progress bar
        if log_events: print(f"{algorithm}, ", end="") if algorithm !=␣
↪list(ALGORITHMS.keys())[-1] else print(f"{algorithm}..\n", end="")

        if not algorithm in algorithms_outputs.keys():␣
↪algorithms_outputs[algorithm] = []

        # solve input instace and obtain solution
        solution = !python3.9 $SOLVER $algorithm -cnt=1 -in=$input_file_path␣
↪-ref=$OUTPUTS_DIR -b

        # store solution output to dataframe
        df = read_and_store_buffer_input_to_df(
            buffer=StringIO(solution.n),
            algorithm=algorithm,
```

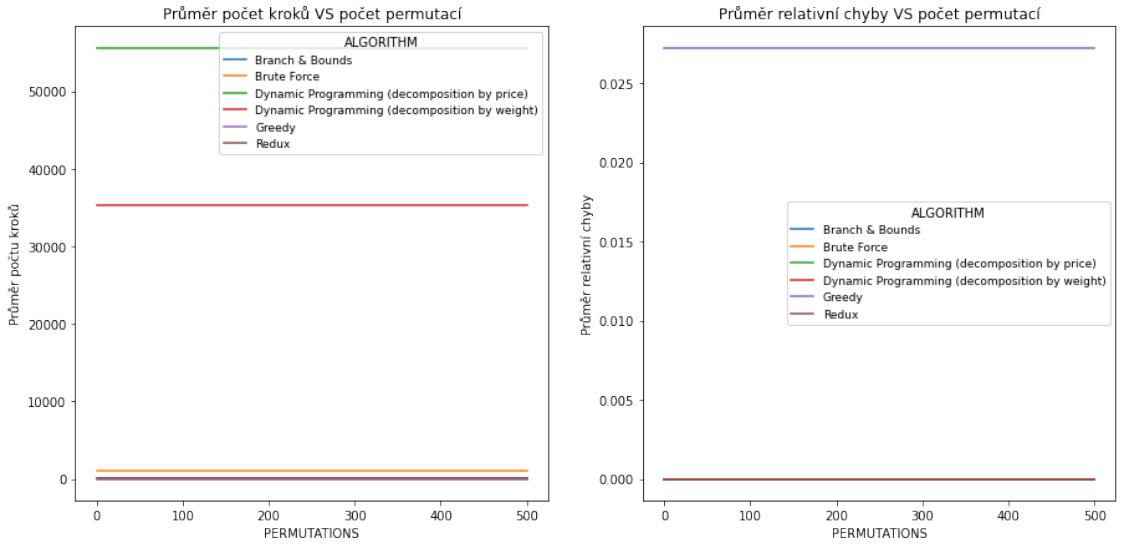
```
            instance_size=kg2_args["-n"]

        )

        # set permutations column
        df['PERMUTATIONS'] = list(range(number_of_permutations + 1))
        df.set_index('PERMUTATIONS')
        algorithms_outputs[algorithm].append(df)


    # concatenate all dataframes of certain alogrithm to single dataframe
    for algorithm in ALGORITHMS.keys():
        algorithms_outputs[algorithm] = pd.
↪concat(algorithms_outputs[algorithm]).reset_index()

    return algorithms_outputs
```

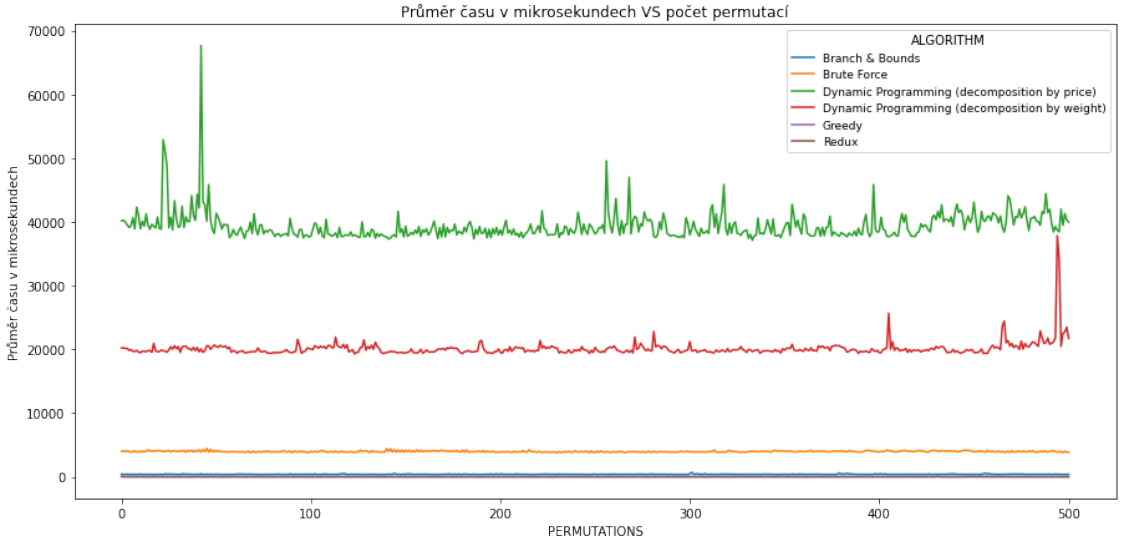
Ted už se klidně můžeme pustit do experimentů.

[70]:
```
number_of_permutations = {
    'PERMUTATIONS': 500
}

kg2_args = KG2_ARGS.copy()

algorithms_outputs = generate_kg2_instance_and_permutate(
    number_of_permutations=number_of_permutations['PERMUTATIONS'],
    kg2_args=kg2_args
)
```

```
[KG2]: Number of permutations  500
 - Generating 500 number of permutations for single instance..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
```

[71]:
```
plot_basic_analytics(
    algorithms_outputs=algorithms_outputs,
    index_column=next(iter(number_of_permutations)),
    parameter_label='počet permutací'
)
```





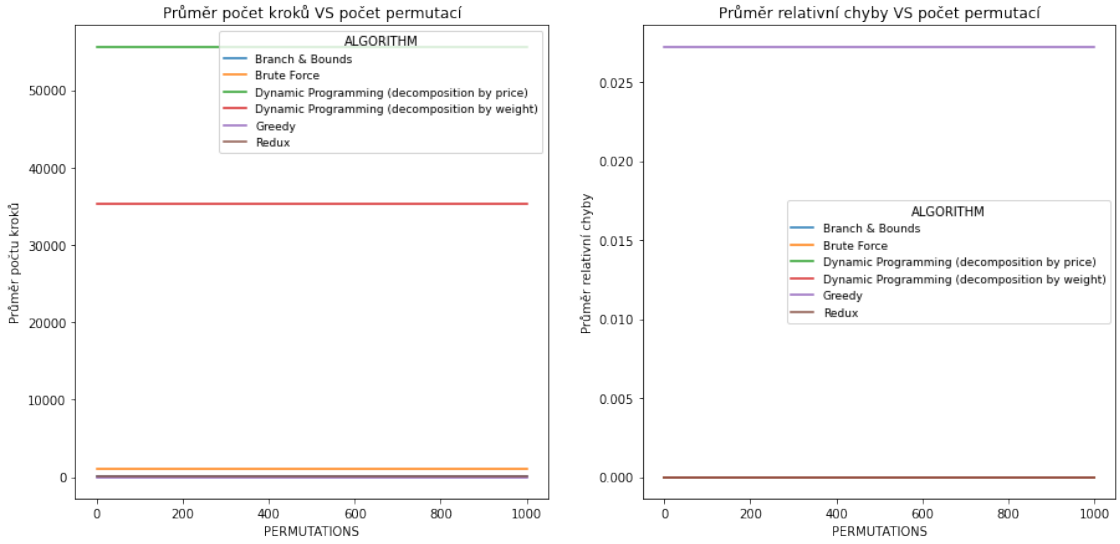
Vypadá, že algoritmy jsou robustní k permutací věci. Pro jistotu ještě zkusíme zvětšit počet permutací na 1000.

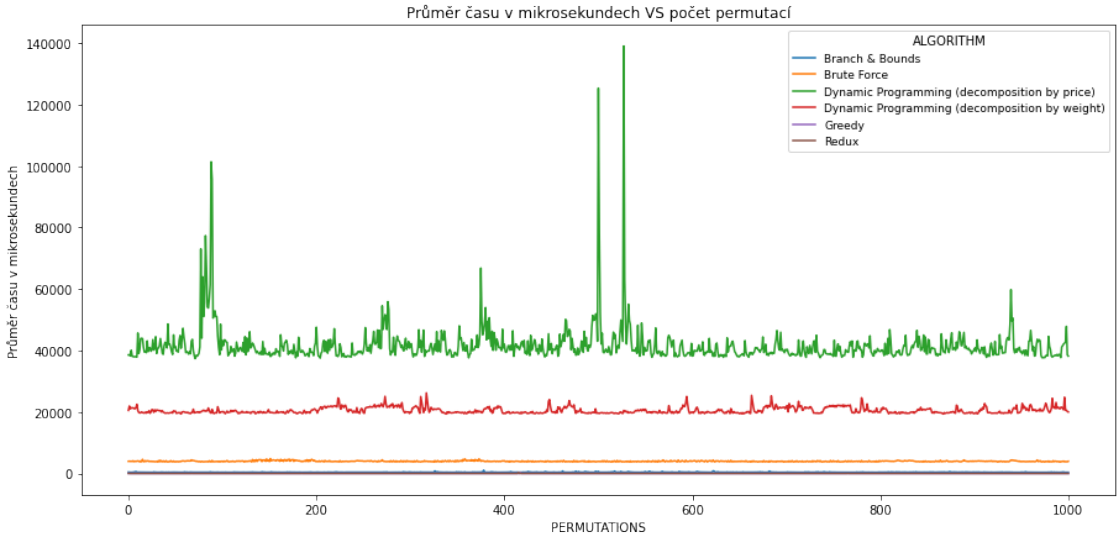
[72]:
```
number_of_permutations = {
    'PERMUTATIONS': 1000
}

kg2_args = KG2_ARGS.copy()

algorithms_outputs = generate_kg2_instance_and_permutate(
    number_of_permutations=number_of_permutations['PERMUTATIONS'],
```

```
    kg2_args=kg2_args
)
```

```
[KG2]: Number of permutations  1000
 - Generating 1000 number of permutations for single instance..
 - Generating references.. using "Brute Force"
 - Solving input instance using algorithms: bf, bb, dp_dw, dp_dp, greedy,
redux..
```

[73]:
```
plot_basic_analytics(
    algorithms_outputs=algorithms_outputs,
    index_column=next(iter(number_of_permutations)),
    parameter_label='počet permutací'
)
```





Se zvětšením počtu permutací se výsledky moc nezměnily. Stále vypadá, že všechny algoritmy jsou robustní k permutací věci.

Na grafu času je vidět několik výchylek u **Dynamic Programming** s dekompozici podle ceny. Tohle zase vypadá, jako nějaké zatížení systémovými procesy atd. Na grafu relativní chyby je vidět, že chyba **Greedy** alogritmu je relativní malá a konstantní.. co se týče **Redux**, tak on i dokonce má nulovou konstantní chybu.

1.5 Závěr

Během této práci byly provedené experimenty hodnocení kvality algoritmů na řešení problémů batohu. Byly probrané, popsané a experementalne ověřené všechny závislosti algoritmů na parametrech instancí generovaných náhodném generátorem instanci. Stejně tak všechny algorimy se ukázaly jako relativní robustní k permutací.