AN IMPROVED DYNAMIC PROGRAMMING ALGORITHM FOR COALITION STRUCTURE GENERATION

Яворський Германн

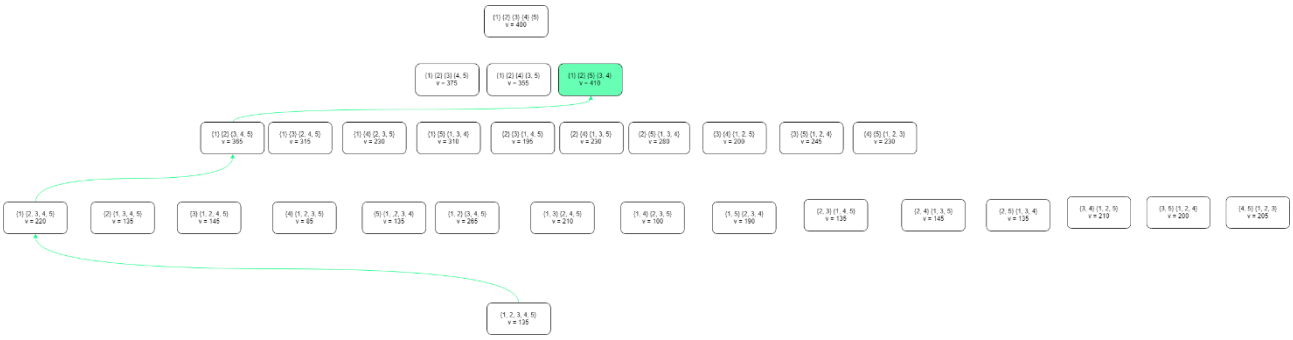
Третяк Олеся

GIT HUB

https://github.com/olesyat/ad_fontes/

ОПИС РОБОТИ

Виконуючи дану роботу, ми реалізували два алгоритми: DP (динамічний алгоритм) та IDP (покращений динамічний алгоритм) для генерації найефективніших розбиттів на коаліції.



Вузлами графа є все можливі розбиття, і DP алгоритм знаходить шлях, який надає нам найефективніше розбиття. DP – експоненціальний алгоритм, а саме $O(3^n)$.

Для реалізації DP ми використовували псевдокод наданий у статті. Крок №4 був описаний незрозуміло, на рядку 4.1 використовувалася змінна S, яка ніде до цього не була ініціалізована. Ми частково змінили цю частину коду, замінивши її на рекурсивну функцію, яка знаходить шлях у графі до розбиття, яке дає нам найвищу ефективність.

Псевдокод IDP у статті не наведений, а головною зміною повинна бути є оптимізація витрат пам'яті для роботи алгоритму.

Основні модифікації Dp, які пропонують автори наступні:

- Перебір розбиттів повинен здійснюватися лише на половині можливих розбиттів, таким чином уникаються повтори;
- Відмова від f2 (словника, де ключами є розбиття, а значеннями їхня максимальна ефективність). Натомість, замінювати дані в вхідних даних. Таким чином втрачаються оригінальні дані, проте це не впливає на знайдення оптимального розв'язку. Проте, під час тестування, ми виявили, що дана модифікація більш затратна по пам'яті, аніж з наявністю f2. Можливо, на такий результат вплинула специфіка пайтону та робота цієї мови з пам'яттю;
- Відмова від f1(словника, де ключі це розбиття, а значення – розбиття ключа, яке гарантує максимальну ефективність). Дана модифікація зменшує використання пам'яті на більше ніж 50%, проте з видаленням f1, ми втрачаємо можливість швидко відновити розв'язок (дорогу у графі, яка привела нас до певного результату).

Інтуїтивно, найкращим здається варіант з відмовою від f2 та збереження f1. При цій варіації повинен б зменшитися обсяг пам'яті та кількості оцінених розбиттів.

ДАНІ

Для кожного алгоритму вхідні дані – це словник, де ключі – все можливі розбиття для масиву довжиною n, а значення – ефективність даної коаліції.

Для порівняння роботи двох алгоритмів було згенеровано розбиття від n = 3 до 18, по 3 екземпляри на кожне значення n.

Наприклад, для n = 7 вхідний дані мають наступний вигляд:


```
{(1,): 27, (2,): 34, (3,): 14, (4,): 29, (5,): 31, (6,): 35, (7,): 36, (1, 2): 71, (1, 3): 26, (1, 4): 27, (1, 5): 39, (1, 6): 62, (1, 7): 26, (2, 3): 52, (2, 4): 35, (2, 5): 34, (2, 6): 52, (2, 7): 38, (3, 4): 64, (3, 5): 28, (3, 6): 47, (3, 7): 26, (4, 5): 61, (4, 6): 52, (4, 7): 51, (5, 6): 47, (5, 7): 61, (6, 7): 23, (1, 2, 3): 98, (1, 2, 4): 89, (1, 2, 5): 39, (1, 2, 6): 89, (1, 2, 7): 122, (1, 3, 4): 118, (1, 3, 5): 99, (1, 3, 6): 91, (1, 3, 7): 100, (1, 4, 5): 66, (1, 4, 6): 43, (1, 4, 7): 48, (1, 5, 6): 100, (1, 5, 7): 92, (1, 6, 7): 106, (2, 3, 4): 88, (2, 3, 5): 101, (2, 3, 6): 50, (2, 3, 7): 57, (2, 4, 5): 52, (2, 4, 6): 61, (2, 4, 7): 84, (2, 5, 6): 84, (2, 5, 7): 44, (2, 6, 7): 72, (3, 4, 5): 32, (3, 4, 6): 68, (3, 4, 7): 39, (3, 5, 6): 121, (3, 5, 7): 115, (3, 6, 7): 37, (4, 5, 6): 90, (4, 5, 7): 103, (4, 6, 7): 50, (5, 6, 7): 67, (1, 2, 3, 4): 165, (1, 2, 3, 5): 116, (1, 2, 3, 6): 148, (1, 2, 3, 7): 141, (1, 2, 4, 5): 130, (1, 2, 4, 6): 136, (1, 2, 4, 7): 87, (1, 2, 5, 6): 160, (1, 2, 5, 7): 75, (1, 2, 6, 7): 142, (1, 3, 4, 5): 141, (1, 3, 4, 6): 151, (1, 3, 4, 7): 103, (1, 3, 5, 6): 65, (1, 3, 5, 7): 164, (1, 3, 6, 7): 110, (1, 4, 5, 6): 85, (1, 4, 5, 7): 114, (1, 4, 6, 7): 80, (1, 5, 6, 7): 51, (2, 3, 4, 5): 104, (2, 3, 4, 6): 168, (2, 3, 4, 7): 145, (2, 3, 5, 6): 78, (2, 3, 5, 7): 104, (2, 3, 6, 7): 100, (2, 4, 5, 6): 115, (2, 4, 5, 7): 51, (2, 4, 6, 7): 109, (2, 5, 6, 7): 101, (3, 4, 5, 6): 153, (3, 4, 5, 7): 135, (3, 4, 6, 7): 80, (3, 5, 6, 7): 133, (4, 5, 6, 7): 138, (1, 2, 3, 4, 5): 111, (1, 2, 3, 4, 6): 185, (1, 2, 3, 4, 7): 77, (1, 2, 3, 5, 6): 133, (1, 2, 3, 5, 7): 115, (1, 2, 3, 6, 7): 190, (1, 2, 4, 5, 6): 171, (1, 2, 4, 5, 7): 97, (1, 2, 4, 6, 7): 186, (1, 2, 5, 6, 7): 194, (1, 3, 4, 5, 6): 79, (1, 3, 4, 5, 7): 164, (1, 3, 4, 6, 7): 103, (1, 3, 5, 6, 7): 102, (1, 4, 5, 6, 7): 78, (2, 3, 4, 5, 6): 178, (2, 3, 4, 5, 7): 57, (2, 3, 4, 6, 7): 175, (2, 3, 5, 6, 7): 76, (2, 4, 5, 6, 7): 184, (3, 4, 5, 6, 7): 119, (1, 2, 3, 4, 5, 6): 232, (1, 2, 3, 4, 5, 7): 234, (1, 2, 3, 4, 6, 7): 171, (1, 2, 3, 5, 6, 7): 144, (1, 2, 4, 5, 6, 7): 224, (1, 3, 4, 5, 6, 7): 102, (2, 3, 4, 5, 6, 7): 163, (1, 2, 3, 4, 5, 6, 7): 89}
```


Модуль для генерації даних: sample_generator.py, папка зі всіма даними: all_samples

Для знаходження всіх розбиттів на множині ми використовували вбудовану бібліотеку itertools.

Для кожного розбиття, з урахуванням його розміру, рандомно генерується ефективність.

Дані записані у файли за допомогою бібліотеки pickle (це дозволяє зручно зчитати їх зразу як об’єкт)

Функція read_samle з модуля samle_reader.py зчитує заданий файл та повертає необхідний для функції вхідний тип даних (словник).

ЕКСПЕРИМЕНТИ

NUM OF EVALUATED SPLITTINGS

Спершу, ми хотіли запускати алгоритми для n є [3, 20], проте n = 19 обраховувало відповідь більше години, тому ви вирішили зупинитися на 18.

Для пошуку оптимальних розбиттів на множині довжиною n (n є [3, 18]) проводилося по 3 запуски для кожного n із різними вхідними даними, згенерованими попередньо. Ми запускали всі 4 варіації алгоритму: DP, IDP, IDP without f2, IDP without f1 and f2.

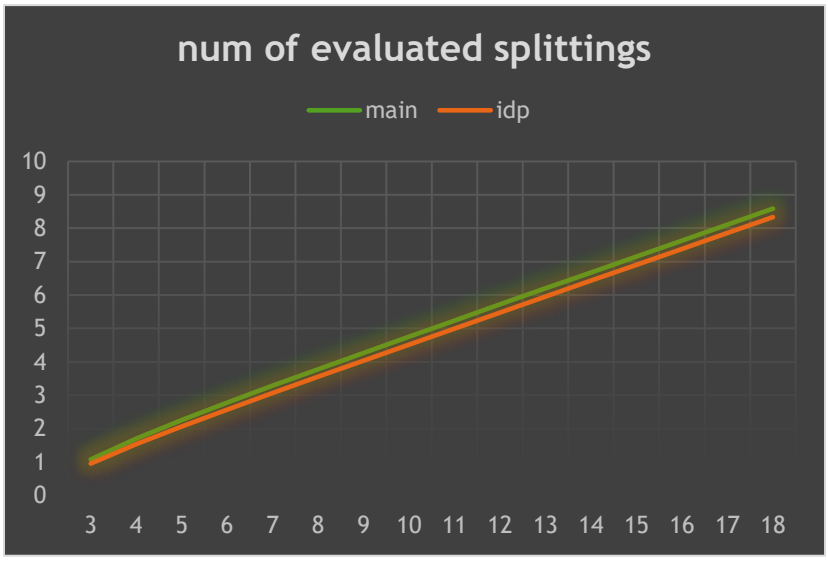
Кількість оцінений порівнянь для трьох останніх завжди однакова.

В таблиці 1 та графіку 1 (шкали - логарифмічні) наведено результати зміни кількості оцінених розбиттів для DP та IDP.

Для DP дана кількість більша на 59.73% в середньому.

	evaluations		
size	main	idp	difference
3	12	9	75.00%
4	50	34	68.00%
5	180	115	63.89%
6	602	371	61.63%
7	1932	1162	60.14%
8	6050	3578	59.14%
9	18660	10899	58.41%
10	57002	32977	57.85%
11	173052	99352	57.41%
12	523250	298519	57.05%
13	1577940	895440	56.75%
14	4750202	2683214	56.49%
15	14283372	8035489	56.26%
16	42915650	24056138	56.05%
17	128878020	72006475	55.87%
18	386896202	215524505	55.71%
		avarage=	59.73%

Таблиця 1



Графік 1

ПАМЯТЬ

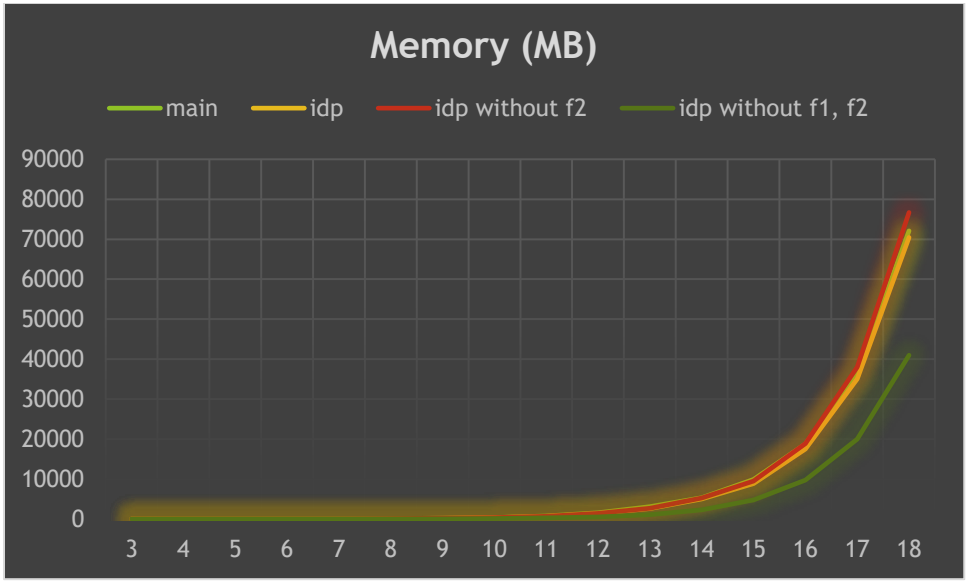
Для трекінгу затрат пам’яті ми використовували вбудовану бібліотеку від Пайтону tracemalloc.

Дана бібліотека дає інформацію про використання пам’яті програмою від моменту виклику tracemalloc.start() до tracemalloc.take_snapshot(). Обсяг використаної пам’яті, в залежності від розміру повертається у байтах, кілобайтах, мегабайтах і тд.

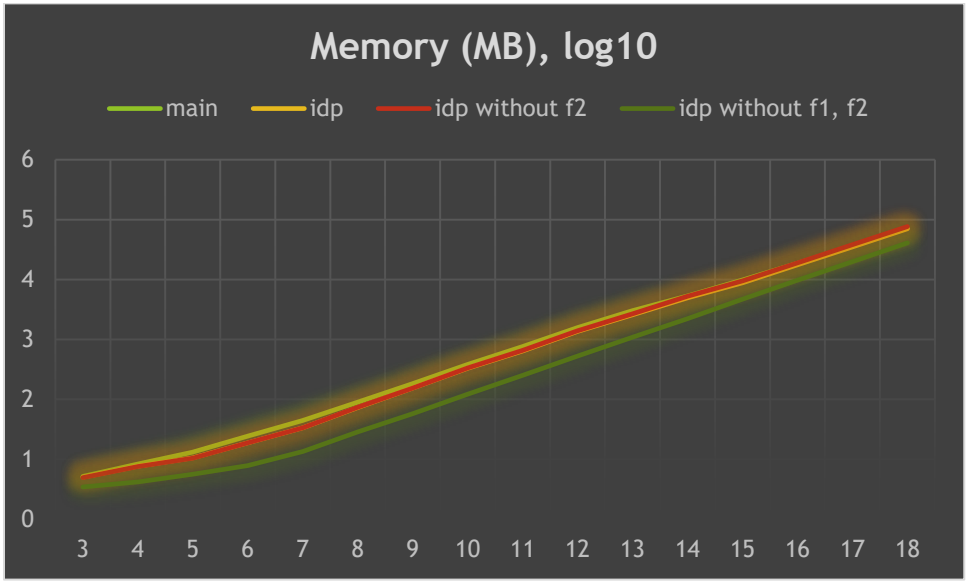
В таблиці 2 та графіку 2.1 та 2.2(логарифмічні шкали) наведена динаміка затрат пам’яті кожної модифікації (MB).

	memory (MB)			
size	main	idp	idp without f2	idp without f1, f2
3	5.203125	5.046875	4.962890625	3.470703125
4	8.421875	7.71875	7.640625	4.203125
5	13.2	10.6	10.5	5.678710938
6	24.6	18.8	18.8	7.9375
7	44.6	33.7	33.6	13.5
8	90.5	73.5	73.7	28.7
9	184	154	155	58.1
10	383	330	334	122
11	768	648	663	254
12	1579	1357	1400	526
13	3050	2578	2687	1088
14	5354	4980	5247	2241
15	9911	8863	9479	4740
16	18739.2	17510.4	18841.6	9741
17	36556.8	35020.8	37990.4	19968
18	72089.6	70451.2	76697.6	40960

Таблиця 2



2.1



2.2

Таблиця 3 показує скільки відсотків пам'яті звичайного DP алгоритму використовує кожна модифікація.

idp	idp without f2	idp without f1, f2
97.00%	95.38%	66.70%
91.65%	90.72%	49.91%
80.30%	79.55%	43.02%
76.42%	76.42%	32.27%
75.56%	75.34%	30.27%
81.22%	81.44%	31.71%
83.70%	84.24%	31.58%
86.16%	87.21%	31.85%
84.38%	86.33%	33.07%
85.94%	88.66%	33.31%
84.52%	88.10%	35.67%
93.01%	98.00%	41.86%
89.43%	95.64%	47.83%
93.44%	100.55%	51.98%
95.80%	103.92%	54.62%
97.73%	106.39%	56.82%
87.27%	89.87%	42.03%

Середнє = таблиця 3

ВИСНОВКИ

Наша гіпотеза про найоптимальнішу варіацію IDP алгоритму виявилася неправильною. Результати експерименту показують, що найкращим варіантом є IPD зі збереженням f1 та f2. Для нас це стало несподіванкою, адже IDP лише з використанням f1 здавався набагато ефективнішим – ми не зберігаємо цілий словник довжиною n у пам'яті! Причиною такого результату може бути специфіка мови Пайтон та її доступом до пам'яті.

Порівнюючи DP та IDP з f1 та f2 ("переможцем" експерименту) ми спостерігаємо зменшення кількості проітерованих розбиттів на 59.7% в середньому та оптимізацію пам'яті на 13% в середньому.

Повертаючись до алгоритму IDP без f1 та f2: даний алгоритм показав справді вражаючі результати по пам'яті. Проте, ми не можемо відтворити шлях по графу, що робить цей алгоритм непотрібним. Автори статі пропонують переоцінювати всі розбиття після знаходження цифри максимальної ефективності переоцінюючи всі попередні розбиття. Даний процес лише добавить нових оцінень сплітінгів (що ми власне намагалися оптимізувати) та буде дуже часозатратним.

Дякую за якісний та ґрунтовний звіт, а також детально прописані висновки.
Видно, що проект пропрацьований вами дуже добре.
За помилку в обрахуванні відсотків для різниці в кількості розбиттів знімається 0,5 балу.

Обрахування фінальної оцінки:
- за першу зустріч: 1 бал
- за проміжну зустріч: 3 бали
- за фінальний звіт: 5,5 балів

Разом: 1 + 3 + 5,5 = 9,5 балів.