Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

АЛГОРИТМ ДВОВИМІРНОГО ПАКУВАННЯ ПРЯМОКУТНИКІВ З МЕТАСТРУКТУРОЮ

Текстова частина до курсової роботи
за спеціальністю „Інженерія програмного забезпечення”

Керівник курсової роботи
д.т.н., доц. А. М. Глибовець
(прізвище та ініціали)
_____
(підпис)
“____” _____ 2021 р.

Виконав студент О.А. Мусіяка
_____
(прізвище та ініціали)
“____” _____ 2021 р.

Київ 2021

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Зав.кафедри інформатики,
доц., д.ф-м.н.
_____С. С. Гороховський
(підпис)
„____”_____2021 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

студенту 1-го курсу магістерської програми факультету інформатики
Мусіяці Олександру Андрійовичу
Розробити Алгоритм двовимірного пакування прямокутників з метаструктурою
Вихідні дані:

- реалізувати алгоритм, що дозволяє змінювати конфігурацію упакованих прямокутників без повторного вирішення задачі пакування з метою оптимізації додаткових параметрів розміщення
- визначити методику для виміру якості роботи алгоритму в залежності від вхідних даних
- провести тестування на різних наборах вхідних даних для визначення основних характеристик якості роботи алгоритму

Зміст ТЧ до магістерської роботи:

Зміст
Анотація
Вступ
1 Огляд існуючих методів та підходів до вирішення задач пакування
2 Вибір підходів до вирішення задачі
3 Розробка алгоритму двовимірного пакування
4 Визначення основних характеристик вхідних даних
5 Визначення методів тестування алгоритму
6 Алгоритм генерації вхідних даних
7 Результати роботи алгоритму на різних типах вхідних даних
Висновки
Список літератури
Додатки

Дата видачі „___” _____ 2021 р. Керівник _____
(підпис)
Завдання отримав _____
(підпис)

Зміст

Анотація

Алгоритми двовимірного пакування прямокутників у зону, обмежену прямокутником мають широке використання для вирішення найрізноманітніших практичних задач. Мотивація для створення нового алгоритму полягає у тому, що прямокутники це зазвичай абстракція якогось фізичного об’єкта, що має певний набір інших властивостей окрім власних розмірів. Це породжує додаткові обмеження та метрики, що досить часто потрібно оптимізувати. Метрики та обмеження є функцією взаємного положення та орієнтації об’єктів, проте при щільному пакуванні їх неможливо довільно переміщувати один відносно одного. Досить часто обчислення метрики вимагає значних витрат обчислювальних ресурсів що робить недоцільним або навіть неможливим обчислення метрики на кожному кроці процесу знаходження рішення задачі. Іншим аспектом, що утруднює оптимізацію метрики на етапі пошуку рішення з задовільною ефективністю є складність бізнес логіки програмного забезпечення, що визначає саму функцію, яка підлягає оптимізації. Інтеграція бізнес логіки, що може досить часто змінюватися з алгоритмічною частиною програми призводить до зниження якості коду у плані легкості внесення змін, обмежує можливості ефективного покриття тестами та унеможливлює розділення бізнес логіки та алгоритму на окремі модулі, що утруднює повторне використання коду.

Через зазначені вище труднощі на практиці досить часто використовують підхід до оптимізації метрик чи задоволення обмежень шляхом простого перебору альтернативних результатів роботи двовимірного пакування, та вибору найкращого варіанту, що задовольняє обмеженням. Недоліком такого підходу є той факт, що алгоритм двовимірного пакування на етапі знаходження рішення керується лише геометричними розмірами, та можливо простими метриками та обмеженнями, що можуть бути легко обчислені. Через це унаслідок певних особливостей роботи алгоритму чи статистичного розподілу вхідних даних різноманіття кінцевих результатів може бути обмеженим у плані взаємного розміщення об’єктів, що призводить до зниження якості знайденого рішення. Метою розробки даного алгоритму є реалізація можливості зміни взаємного розташування об’єктів без повторного вирішення вихідної задачі. Такі властивості знайденого рішення отримуються завдяки накладанню додаткових обмежень на властивості пакування та використанню спеціальної структури даних, що відображає певну множину рішень та може бути легко перетворена будь-яке з них.

Вступ

Задачі оптимізації взаємного розташування об’єктів включають в себе багато різновидів задач, що відрізняються за умовами постановки та за параметрами, що підлягають оптимізації. Зазвичай такі задачі мають на меті оптимізацію деякого розміщення матеріальних об’єктів, тому мають кількість вимірів у межах від 1 до 3. Одновимірний варіант задачі це класична задача про рюкзак, що є хоч і NP-повною проте у багатьох практичних випадках може бути ефективно вирішена методами динамічного програмування. Що ж до тривимірного варіанту, він навпаки досить складний у плані практичного застосування, хоч і існують алгоритми здатні вирішувати задачі тривимірного пакування, наприклад simulated annealing, проте ефективність роботи та якість рішень для довільного набору вхідних даних часто не є задовільними [1].

Задачі двовимірного пакування знаходяться як раз на межі між тривіальними одновимірними задачами та занадто складними тривимірними проблемами. Багато технологічних процесів мають двовимірну природу, наприклад розкрій листів на окремі деталі, проектування інтегральних схем чи наприклад відображення контенту на веб-сторінці. Деякі тривимірного пакування можуть бути зведені до двовимірного варіанту шляхом пошарової декомпозиції [2].

Алгоритми двовимірного пакування у загальному випадку можуть бути побудовані для довільних фігур, проте найбільш простою фігурою, що має широке практичне застосування є прямокутник, тому у даній роботі буде розглядатися саме пакування прямокутників. Задача пакування прямокутників може бути сформульована з відкритим виміром, тобто заданий набір прямокутників потрібно максимально ефективно розмістити у смугу заданої ширини та мінімізувати потрібну довжину смуги. Наприклад, оптимальний розкрій довгої смуги довільного матеріалу може бути зведений саме до такої задачі [3].

У іншому варіанті формулювання задачі задано розміри зони пакування у вигляді прямокутника, та деякий набір прямокутників, що у загальному випадку може мати більшу площу, ніж зона пакування. Метою вирішення такої задачі є максимізація загальної площі прямокутників, що розміщені у зоні пакування. Прикладом практичного застосування може бути оптимізація розрізу прямокутного листа на прямокутні частини заданого розміру.

Практичні застосування також можуть мати додаткові обмеження щодо сумісності певних об’єктів у одній зоні пакування та їх просторової орієнтації, прикладом такого виду задач є проектування топології кристалів інтегральних схем, де крім щільності розміщення потрібно також брати до уваги такі параметри як взаємні під’єднання, виділення тепла, паразитні електрорушійні сили та бажане розміщення точок підключення [4].

Існують варіанти задач, де метрикою оптимізації може виступати не лише сумарна площа прямокутників, що вдалося розмістити, але й наприклад міра кластеризації певних класів об’єктів, результуючий центр маси, моменти інерції, тощо. Одним з важливих прикладів застосування двовимірного пакування є декомпозиція задачі розміщення тривимірних об’єктів, таких як тара різного розміру у вигляді паралелепіпедів (наприклад: картонні коробки), що вертикально складена у об’єм з заданими горизонтальними розмірами (наприклад: палета). Підхід з декомпозицією дозволяє заповнювати тривимірний об’єм пошарово, при використанні такого підходу компонування кожного наступного шару предметів вимагає вирішення задачі двовимірного пакування з обмеженнями [5]. Особливістю такої задачі є саме ці додаткові обмеження, такі як умова, що предмет на наступному шарі має мати надійну опору з предметів попереднього шару, до того ж, бажано мати перекриття з декількома предметами, щоб забезпечити хорошу стійкість утвореної структури. Крім цього можуть бути також додаткові обмеження, що не дозволяють розміщувати одні типи об’єктів над іншими (наприклад: побутову хімію над продуктами харчування, або ж важкі предмети над крихкими).

Приклади різноманітних практичних задач, що наведено вище демонструють потенційну користь від алгоритму, що може створювати структуру двовимірного пакування, яка може бути розгорнута у багато варіантів конкретного розміщення об’єктів, замість того, щоб багато разів підряд виконувати обчислення, потрібні для пошуку якісного рішення. Ця структура у даній роботі називається “метаструктурою”. Ця структура служить для отримання певної множини конкретних рішень задачі, проте сама по собі не визначає конкретні позиції об’єктів [6].

Існує певний клас алгоритмів, що дають одразу багато рішень за один цикл обчислення, наприклад алгоритми типу beam search. Проте, це досить загальний клас алгоритмів, що не використовує геометричні особливості поставленої задачі. Метою даної роботи є створення алгоритму, що знаходить метаструктуру рішення, з якої будуть обчислюватись кінцеві рішення.

1 Огляд існуючих методів вирішення задачі пакування

1.1 Класифікація різновидів задач пакування за початковими умовами

Запропонований у даній роботі алгоритм є частковим випадком з великого сімейства алгоритмів, що вирішують задачу розміщень з обмеженнями. Для того, щоб огляд існуючих алгоритмів був релевантний до поставленої мети, потрібно ввести класифікацію, що дозволить визначити достатньо вузький для безпосереднього порівняння клас алгоритмів а також визначити подібні алгоритми, що належать до суміжних класів, проте вони є корисними для даного огляду.

Загальноприйнятими критеріями класифікації є кількість вимірів, з якими працює алгоритм, форма об’єктів, що розміщуються, дозволені варіанти орієнтації цих об’єктів, характеристики зони пакування та властивості утвореної структури [3]. Визначення цих характеристик для запропонованої задачі наведено у таблиці 1.1.

Таблиця 1.1 Класифікація алгоритмів пакування за параметрами

Кількість вимірів задачі	1-вимірна	**2-вимірна**	3-вимірна	багатовимірна
Форма об’єктів, що розміщуються	круг, (гіпер-) сфера	**прямокутник**	довільна форма	
Орієнтація об’єктів	довільна		**ректилінійна** (під прямим кутом)	
Обмеженість зони пакування	відкритий вимір (нескінченний простір)		**повністю задана**	
Форма зони пакування	довільна		**прямокутна**	
Властивості структури	довільна		**гільйотинована**	

Виділені комірки в таблиці 1.1 відповідають характеристикам вирішуваної задачі. Таким чином, задача, що буде вирішуватись є двовимірним пакуванням прямокутників з ректилінійною орієнтацією у просторі з повністю заданою зоною пакування прямокутної форми та можливістю гільйотинного розрізу зони пакування на вписані у неї прямокутники. У даному огляді будуть також розглядатися інші варіанти постановки задачі, що мають подібні властивості та можуть

бути вирішені аналогічним способом. Досить популярним різновидом подібних задач є пакування з відкритим виміром, де потрібно розмістити деяку множину прямокутників на стрічку заданої ширини без задання її довжини. Слід звернути увагу на те, що більшість використовуваних алгоритмів дають не гільйотиновані рішення. Це можна пояснити тим, що гільйотинована структура накладає додаткові обмеження, у наслідок чого знаходження рішення з хорошою щільністю є значно складнішим.

1.2 Класифікація алгоритмів за способом вирішення задачі

Підходи до вирішення задач такого вигляду поділяються на три основні види: точні, евристичні та гібридні. Точні методи дають гарантовано оптимальне рішення, проте більшість практичних задач не дозволяє користуватися ними через занадто велику розмірність, що у сукупності з експоненційною складністю задачі результує у неприйнятно велику кількість обчислень. Евристичні методи як правило мають поліноміальну обчислювальну складність, у наслідок чого потребують менше обчислювальних ресурсів ніж точні методи, проте на відміну від точних методів вони як правило не дають оптимального рішення. І хоча для більшості евристичних алгоритмів існують доведені нижні границі апроксимації, вони часто не є достатньо хорошими для практичних застосувань [7]. Гібридні методи є комбінацією точних методів з евристичними. Такі методи часто дозволяють знайти потрібний компроміс між якістю рішення та витратою обчислювальних ресурсів. Існує два основних типи евристичних методів вирішення задачі пакування: конструктивні та покращувальні. Конструктивні методи будують рішення з використанням заданого набору правил, тоді ж як покращувальні методи працюють шляхом ітеративного застосування невеликих змін до деякого початкового рішення.

Для наближених алгоритмів, що використовують ітеративний підхід актуальною проблемою є визначення критерію зупинки. Складність полягає у тому, що визначення моменту зупинки сильно залежить від конкретного застосування алгоритму, у той час як алгоритми намагаються створювати максимально універсальними. У якості універсального критерію може використовуватися оцінка нижньої границі цільової функції, проте досить часто її неможливо визначити належним чином. Іншим варіантом може бути середнье значення приросту якості рішення за деяку кількість ітерацій, коли пошук зупиняється якщо немає суттєвого прогресу при

виконанні ітерацій. Проте такий спосіб може хибно спрацьовувати при переміщенні по плато цільової функції, а отже не є надійним критерієм [8].

Евристичні методи можна поділити на однофазні, двофазні та метаевристики. Однофазні методи пакують об’єкти безпосередньо у зону пакування, у той час як двофазні створюють деяку проміжну структуру, зазвичай у вигляді упакованого ряду об’єктів, що відповідає вимірам зони пакування, а потім комбінують ці структури у кінцеве рішення.

Найбільш перспективними з точки зору отримання якісного рішення для досить великих екземплярів задачі двовимірного пакування є гібридні алгоритми. На відміну від евристичних, гібридні алгоритми є досить різноманітними за принципом роботи, проте вони мають спільні риси, що дозволяють згрупувати їх у окремі категорії. Слід зазначити, що ці категорії будуть перетинатися у наслідок того, що ефективні гібридні алгоритми часто використовують комбінацію декількох підходів.

Спільною рисою гібридних алгоритмів є використання методів штучного інтелекту, таких як пошук у глибину та ширину, алгоритм A*, α-β відтинання тощо. Іншою характерною рисою таких алгоритмів є використання принципу “розділяй та пануй” та методів динамічного програмування, що як правило утворює рекурсивний процес пошуку рішення [9]. Окремо слід відзначити алгоритми сімейства beam search, що самі по собі є евристичними у плані способу пошуку рішення задачі, проте попри схожу назву, їх використовують з алгоритмами двовимірного пакування, що належать до класу гібридних.

1.3 Огляд евристичних алгоритмів

Евристичні алгоритми двовимірного пакування досить широко представлені у наукових публікаціях. Цікавість таких алгоритмів з академічної точки зору полягає у їх придатності до математичного аналізу, а саме можливості доведення нижньої границі щільності пакування при застосуванні деяких припущень до вхідного набору даних. У даному розділі буде викладено огляд класичних алгоритмів без заглиблення у їх математичні властивості.

Серед евристичних алгоритмів можна виділити два досить великих класи: однокрокові та двокрокові. Прикладами однокрокових евристик можуть слугувати такі алгоритми як Next-Fit Decreasing Height (NFDH), First-Fit Decreasing Hetight (FFDH) та Best-Fit Decreasing Height (BFDH). Крім цього до однокрокових евристик належать такі алгоритми як Alternate Direction (AD), Bottom-Left Fill (BLF), Improved Lowest Gap Fill (LGFi) та Touching Perimeter (TP) [10].

До двофазних евристичних методів відносять Hybrid Next-Fit (HNF), Hybrid First-Fit (HFF), Finite Best-Strip (FBS), Knapsack Packing (KP) та Floor Ceiling (FC). У порівнянні з однофазними евристиками двофазні евристики як правило дають значно кращі результати, особливо у комбінації з алгоритмами локального пошуку. Однофазні алгоритми являють собою скоріше академічний інтерес, оскільки вони значно простіші для математичного аналізу та дають уявлення про базові способи вирішення складної задачі.

У огляді будуть розглядатися як алгоритми двовимірної упаковки для задач з відкритим виміром (Strip Packing) так і задач з заданою зоною пакування (Bin Packing). Крім цього, варто відзначити що у більшості практичних застосувань алгоритми з використанням евристики мають бути підлаштовані під специфіку конкретної задачі. Це стосується наприклад обмежень у комбінаторному пошуку для гібридних алгоритмів, адаптування вагових коефіцієнтів під статистику розподілу розміру вхідних прямокутників, урахування обмежень на поворот прямокутника на 90 градусів, тощо [11].

1.4 Евристичні однокрокові алгоритми для пакування зони з відкритим виміром

Розглянемо евристичний алгоритм Next Fit Decreasing Height (NFDH). Суть цього досить найпростішого алгоритму полягає у роботі з відсортованою множиною прямокутників, що вкладаються рядами поперек нескінченної стрічки у порядку зменшення висоти прямокутника. Якщо ширина стрічки не дозволяє розмістити наступний прямокутник у тому ж ряду, відбувається перехід на наступний ряд.

Подальшим розвитком цього підходу можна вважати алгоритм First Fit Decreasing Height (FFDH). На відміну від NFDH цей алгоритм проводить пошук прямокутника, що може вміститись у залишковий проміжок серед наступних прямокутників. Оскільки послідовність

відсортована по висоті за спаданням, то перевіряти потрібно лише ширину прямокутника [7]. Робота цих двох алгоритмів проілюстрована на малюнку рис. 1.1.

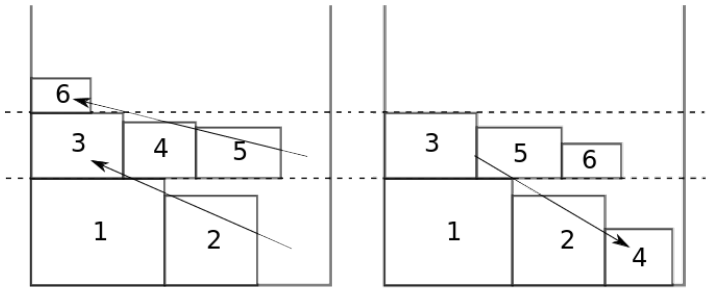

Рис.1.1 Ілюстрація роботи алгоритму NFDH (зліва) та FFDH (справа)

Робота алгоритму Best Fit Decreasing Height аналогічна FFDH з тією різницею, що вибирається не перший прямокутник, що вміщується в залишкову ширину смуги, а той, у якого залишок ширини найменший. Тобто, найширший з прямокутників, що можуть туди вміститися.

1.5 Евристичні алгоритми для пакування повністю заданої зони

Для пакування заданої зони існують аналоги алгоритмів, що використовуються для задач з відкритим виміром: Finite Next Fit (FNF) — аналогічний до NFDH, Finite First Fit (FFF) — аналогічний до FFDH, Finite Bottom Left (FBL) — аналогічний до BL. Крім них існує велика кількість інших однофазних евристичних алгоритмів, але двофазні евристики становлять більший практичний інтерес завдяки більшим можливостям щодо їх комбінації з алгоритмами оптимізації.

Серед алгоритмів, що використовують двофазну евристику заслуговує на увагу Hybrid First-Fit (HFF). Цей алгоритм по своїй суті є евристичною декомпозицією двовимірної задачі у одновимірну. На першому етапі будуються “рядки” за допомогою алгоритму FFDH, після чого застосовується алгоритм FFD, що пакує отримані “рядки” у зону пакування за висотою. Подібним чином працює алгоритм Hybrid Next Fit (HNF), який на першому етапі застосовує метод BFDH, а на другому стратегію BFD [6].

У практичних застосунках чисто евристичні методи застосовуються досить рідко, оскільки гібридні методи дають можливість збільшувати кількість варіантів пакувань, що розглядаються, і таким чином знаходити кращі рішення за рахунок використання більшої кількості обчислювальних ресурсів. З іншого боку, евристичні методи є хорошою основою для гібридних методів, що дозволяє знаходити якісні рішення без зайвого заглиблення у комбінаторний пошук, простір якого при збільшенні розмірності задачі швидко стає неосяжно великим навіть для найпотужніших комп’ютерів у наслідок його експоненційної складності [12].

1.6 Гібридні алгоритми на основі двокрокових евристичних алгоритмів

У даному розділі буде розглянуто приклад гібридного алгоритму двовимірного пакування прямокутників, в основу якого покладено декомпозицію двовимірної задачі на два кроки, у кожному з яких вирішуються одновимірні задачі оптимізації шляхом обмеженого комбінаторного пошуку.

З точки зору організації верхнього рівня алгоритм являє собою алгоритм beam search в середині якого виконується подвійний цикл. У внутрішній частині подвійного циклу проводиться пошук так званих “рядків-кандидатів”, що являють собою набір прямокутників із сумарною шириною не більше ширини зони пакування. З множини доступних на даному етапі прямокутників вибирається деяка кількість підмножин, що задовольняють умові:

$$\sum_{i \in k} w_i \leqslant W_{area}$$

Де W_{area}це ширина зони пакування, w_i- ширина і-того прямокутника, k - множина прямокутників рядка-кандидата.

Таким чином, вибирається деяка визначена кількість наборів прямокутників, кожен з яких вміщується у зону пакування у вигляді горизонтального рядка. Для вибору кращих варіантів застосовується ранжування за оціночними метриками якості. Оціночні метрики мають на меті оцінити потенційну якість рядка-кандидата у разі його фактичного додавання у пакування. Попереднье оцінювання є різновидом евристики, що зменшує обчислювальні витрати на розміщення усіх наявних рядків, що є ресурсоємним за об’ємом потрібних обчислень [4].

1.7 Використання підходу “розділяй та пануй”

Методи, що використовують розбиття великої задачі на деяку кількість малих задач мають широке застосування для вирішення обчислювальних проблем. Найбільш ефективним даний підхід є у випадку, коли результуючі задачі незалежні, а результати їх вирішення можуть бути ефективно об’єднані. У такому сприятливому випадку результуючий алгоритм вирішує задачу за лінеарифмічний або поліноміальний час. Прикладом такого застосування є широко відомі алгоритми швидкого сортування та сортування злиттям.

У випадку коли окремі підзадачі, утворені з більшого екземпляру задачі є взаємозалежними результуючий час роботи алгоритму є експоненційним. Задачі пакування утворюють саме таку структуру з взаємозалежних підзадач, у чому легко впевнитись якщо розглянути процес поділу великого екземпляру задачі на більш малі. Для того, щоб розділити задачу на підзадачі потрібно прийняти певне довільне рішення, яким саме чином цю задачу буде розділено. Це рішення впливає на множину допустимих рішень підзадач вносячи таким чином взаємозалежність, хоча самі по собі підзадачі можуть бути повністю незалежними одна від одної. Така структура задачі результує у дерево рішень, що має у якості кореня початкову задачу, а конкретні рішення є листовими вузлами цього дерева [12].

Дерево рішень породжується досить широким спектром практичних задач, і тому у рамках теорії алгоритмів та методів штучного інтелекту існують загальноприйняті способи їх розв’язання. Підходи до знаходження рішення можна поділити на два види: декомпозиційні (низхідний підхід) та конструктивні (висхідний підхід). Евристичні методи двовимірного пакування можна класифікувати як конструктивні, а методи розділяй та пануй як декомпозиційні. Окремо слід відзначити метод зустрічного пошуку, який являє собою комбінацію конструктивного та декомпозиційного підходів [13].

1.8 Динамічне програмування у якості способу пришвидшення роботи

Ідея, яка лежить у основі динамічного програмування полягає у збереженні рішень деяких підзадач для подальшого їх використання у випадку, якщо така сама або ж досить близька

підзадача виникне при подальшому розв’язанні основної задачі. З самого визначення цілком зрозуміло, що даний метод придатний лише для декомпозиційного підходу, оскільки він передбачає існування підзадач, що будуть повторюватись з певною імовірністю. У випадку використання даного метода для вирішення задач двовимірного пакування основним обмежуючим фактором є саме повторюваність підзадач та допустимість їх рішень у контексті породжуючої задачі [14]. Розглянемо ці два аспекти більш детально, оскільки їх правильна оцінка для кожного з практичних застосувань дозволяє визначити доцільність застосування методів динамічного програмування, адже збереження рішень які майже ніколи не використовуються повторно не тільки не пришвидшує роботу алгоритму, а може навіть сповільнювати його практичну реалізацію через надмірне споживання ресурсів.

При розділенні задачі двовимірного пакування на підзадачі можливі два варіанти: розділення на підзадачі, у яких обидва виміри мають довільний розмір (розбиття на квадранти) та розділення на підзадачі, у яких один із вимірів є фіксованим та відповідає виміру початкової задачі (розбиття на рядки). Для оцінки ймовірності того, що у таблиці динамічного програмування міститься рішення підзадачі придатне для повторного використання будемо вважати, що розміри підзадач є неперервна випадкова величина, а обмеженням пов’язаним з вичерпністю множини вхідних прямокутників знехтуємо. Таке припущення є занадто оптимістичним для більшості практичних випадків, проте воно дозволяє оцінити, як кількість розмірів, що змінюється у процесі поділу на підзадачі впливає на ефективність методу динамічного програмування.

Для оцінки ймовірності придатності підзадачі для повторного використання введемо оцінку ймовірності того, що розміри рішення новоутвореної підзадачі не менше обох розмірів хоча б одної вже вирішеної підзадачі, та потрапляють у деякий ε-окіл, що дозволяє обмежити втрату ефективності пакування за рахунок невикористаного простору, обмежуючи його верхню оцінку величиною ε.

$$P_{N=1} = P\left(d \in \left[s, s+\varepsilon\right]\right), \text{для одного виміру}$$

$$P_{N=2} = P\left(d_1 \in \left[s_1, s_1+\varepsilon\right]\right) * P\left(d_2 \in \left[s_2, s_2+\varepsilon\right]\right), \text{для двох вимірів}$$

Де $P_{N=1}$ та $P_{N=2}$ ймовірності потрапляння готового рішення задачі у потрібні розміри, d, d_1 та d_2 - розміри одновимірної та двовимірної підзадач, що виникли у процесі вирішення, s, s_1 та s_2 – розміри хоча б одного підходящого рішення з таблиці.

$$P_{N=2} \sim P_{N=1}^2, оскільки\ P \leqslant 1, то\ P_{N=2} \leqslant P_{N=1}$$

Як можна побачити з формул, ймовірність повторного використання рішень для підзадач з двома змінними розмірами менша, ніж для підзадач з одним змінним розміром. У випадку з двома змінними розмірами ймовірність повторного використання зменшується пропорційно до ε^2 при зменшенні величини ε [15].

Даний аналіз не є вичерпним і має на меті лише демонстрацію одного з двох важливих обмежень, які виникають при використанні динамічного програмування для вирішення задач двовимірного пакування. При рішенні практичних задач різниця між ефективністю рядкового та квадрантного поділу зони пакування значно менша, ніж слід було б очікувати з вищенаведеного аналізу. Причиною цього є інше обмеження пов’язане з вичерпністю множини прямокутників у вхідних даних. Оскільки ймовірність того, що готове рішення не можна скласти повторно з асортименту наявних прямокутників є досить високою, то більшість вже знайдених рішень є непридатними для застосування у певному конкретному контексті вирішення задачі.

2 Вибір методів вирішення задачі

2.1 Необхідні обмеження для забезпечення потрібних властивостей рішення

Метою розробки алгоритму, що пропонується у цій роботі є отримання певних корисних властивостей, що досягається шляхом введення метаструктури, яка визначає певну множину рішень задачі. Нижче буде продемонстровано, яким чином можна отримати ці властивості шляхом накладання вимоги гільйотинованості на рішення задачі двовимірного пакування прямокутників.

Гільйотинованість це властивість структури пакування, яка дозволяє розділити зону пакування на частини потрібного розміру (прямокутники, що відповідають вхідним даним) шляхом повторного застосування гільйотинного розрізу, тобто розрізу, що розділяє зону пакування на дві частини вздовж вертикальної чи горизонтальної прямої лінії, як показано на рис. 2.1.

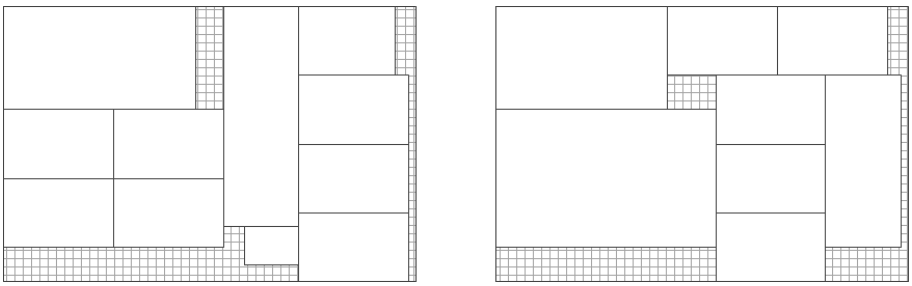

Рис. 2.1 Гільйотиноване розміщення (зліва) та не гільйотиноване (справа)

На правій та лівій частинах рис. 2.1 зображено гільйотиновану та не гільйотиновану компоновку однакових прямокутників. І хоча накладання обмеження гільйотинованості дещо ускладнює пошук рішення задачі, дана структура породжує корисну властивість такої компоновки, а саме рекурсивну структуру рішення задачі, причому на кожному кроці рекурсії виникають дві подібні підзадачі, і ці дві підзадачі мають один з розмірів рівний розміру початкової задачі. Ця особливість породжує для кожного вузла дерева рекурсії два рівноцінних з точки зору щільності упаковки способи об’єднання результатів шляхом розміщення рішень підзадач у різному порядку відносно умовної лінії розділу. Це випливає з того, як утворюються геометричні розміри підзадач з розмірів початкової задачі.

Розміри вихідної задачі співвідносяться з розмірами утворених підзадач наступним чином:

$d_1^0 = d_1^1 + d_1^2, d_2^0 = d_2^1 = d_2^2$

Де d_1^0та d_2^0це розміри вихідної задачі, $d_1^1, d_2^1, d_1^2, d_2^2$- розміри підзадач, утворених у процесі розділу. Два варіанти розміщення для кожного розбиття випливають з комутативності операції додавання, що може бути наочно проілюстровано на рис. 2.2, де показано зміну взаємного розміщення підзадач для одного з етапів розділу.

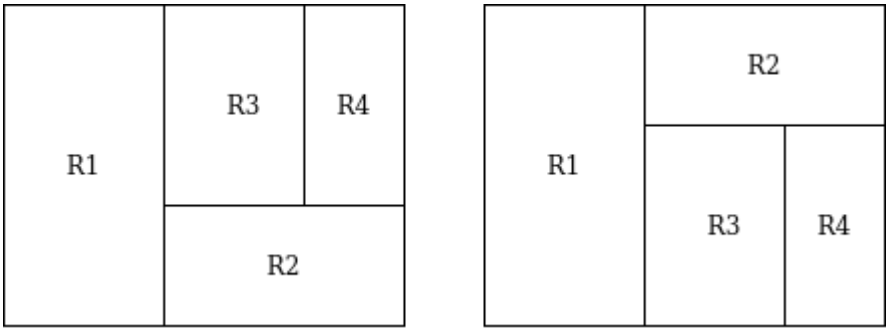

Рис. 2.2 Приклад зміни розміщення результатів вирішення підзадач

Очевидно, що такі перестановки можливі для кожного з розбиттів, та еквівалентно зміні порядку обходу вузлів дерева розбиття на підзадачі. Саме таке дерево і пропонується використовувати у якості метаструктури, що генерує кінцеві варіанти рішень шляхом довільної транспозиції його окремих вузлів.

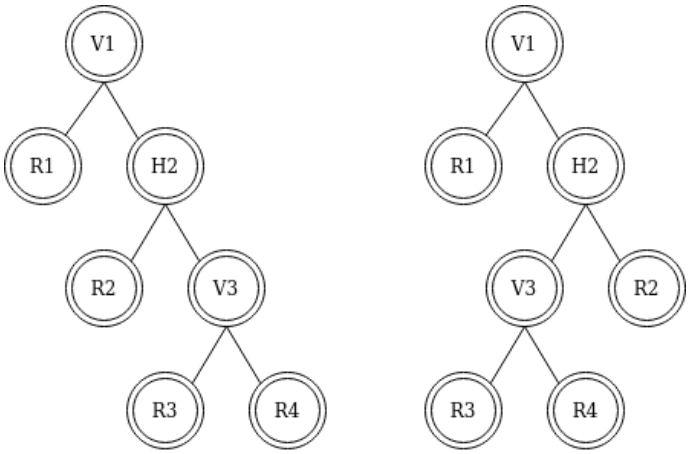

Рис. 2.3 Дерева розбиття рішень та їх транспозиції, що відповідають рис. 2.2

Як видно з ілюстрації рис. 2.3 кожному варіанту розміщення рішень підзадач відповідає транспозиція певного вузла дерева метаструктури рішення. На цьому малюнку вузли, позначені літерами R є листовими вузлами, що відповідають прямокутникам, з яких утворено пакування. Літерами H та V позначені не листові вузли, що відповідають горизонтальному та вертикальному розбиттю на підзадачі.

2.2 Оцінка потужності множини транспозиції вузлів дерева метаструктури

Кількість можливих варіантів рішень, що можна утворити з одного дерева метаструктури рішення є важливим показником придатності даного підходу до оптимізації додаткових параметрів рішення шляхом взаємної перестановки його окремих елементів. Якщо для простоти оцінки припустити, що кожен прямокутник у дереві метаструктури належить окремому класу еквівалентності, то загальна кількість транспозицій метаструктури має наступний вигляд:

$$N(k)=2^k$$

Де N це кількість можливих транспозицій а k – кількість внутрішніх вузлів дерева метаструктури. Для отримання оцінки загальної кількості варіантів потрібно виразити кількість внутрішніх вузлів дерева через кількість його листових вузлів, щоб оцінити цей параметр через наявну кількість прямокутників:

$$N_i=N_l-1, \textit{звідки } N(r)=2^{r-1}$$

Де N_i, N_l- кількість внутрішніх та листових вузлів повного бінарного дерева, N – кількість можливих транспозицій, r – кількість розміщених прямокутників.

Для простої оцінки повноти можливих перестановок розглянемо спрощений приклад, у якому деяку кількість прямокутників розміщено послідовно вздовж однієї лінії. Для такого випадку кількість усіх можливих перестановок дорівнює $r!$ Для наочної демонстрації графік функцій $r!$та 2^{r-1}зображено на рис. 2.4 з використанням лінійно-логарифмічного масштабу. Як видно з графіку, множина можливих транспозицій значно менша за множину усіх можливих перестановок, проте її зростання зі збільшенням кількості прямокутників є експоненційним, з чого можна зробити припущення, що у більшості випадків практичного застосування кількість доступних варіантів буде цілком достатньою для проведення додаткової оптимізації взаємного розміщення прямокутників.

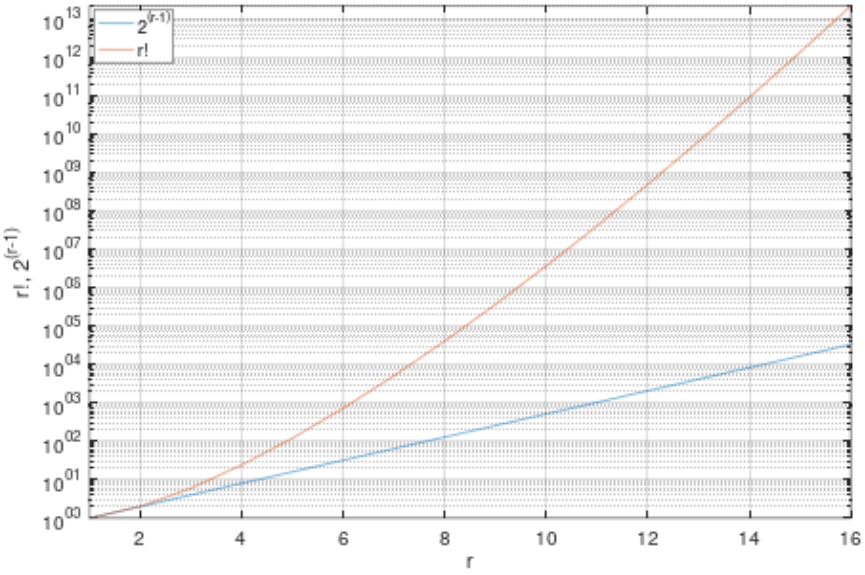

Рис 2.4 Порівняння швидкості зростання доступної та загальної кількості перестановок

2.3 Вибір алгоритмічних засобів пошуку рішення

У найпершому наближенні для знаходження рішень задачі пропонується застосувати пошук у глибину. Такий підхід дозволить отримати корисні практичні результати без введення додаткової складності в програмну реалізацію алгоритму. Наступним кроком покращення алгоритму планується реалізація комбінованого декомпозиційно-конструктивного підходу до вирішення задачі у вигляді зустрічного пошуку. У такому варіанті будуть одночасно використовуватись як пошук у глибину, так і пошук у ширину для висхідної частини алгоритму.

На практиці розбиття зони пакування відбувається не на дві, а на три підзадачі, при чому у спрощеному варіанті алгоритму у якості рішення однієї з підзадач береться один з доступних прямокутників, у той час як дві інші підзадачі обробляються пошуком у глибину. У покращеній реалізації алгоритму пропонується замість одиничного прямокутника в одній з підзадач використовувати результати конструктивного алгоритму, що реалізує висхідний підхід до вирішення задачі. Для зменшення витрат обчислювальних ресурсів пропонується чергувати ітерації висхідного алгоритму з ітераціями пошуку в глибину, проте у такому випадку необхідно вжити заходів щодо обмеження кількості комбінацій, що розглядаються.

3 Розробка алгоритму двовимірного пакування

3.1 Вибір інструментів програмної реалізації

Хоча алгоритм по своїй суті являє абстрактну послідовність кроків для вирішення певної задачі, вибір конкретної мови програмування та її парадигми відіграє суттєву роль у практичній його реалізації. Для реалізації алгоритмів комбінаторного пошуку часто використовують логічні чи функціональні мови програмування, що з одного боку спрощує саму реалізацію приховуючи складні деталі, а з іншого зменшує доступність цієї реалізації для широкого загалу користувачів та робить складним або навіть недоцільним перенесення на інші мови програмування або навіть внесення потрібних для користувача модифікацій. З цих міркувань для виконання практичної частини роботи було вирішено використовувати імперативну мову програмування.

Об’єктно-орієнтоване програмування є стандартом де-факто для розробки сучасного програмного забезпечення, тому для покращення структури коду і його придатності до повторного використання та модифікації алгоритм реалізовано на мові з підтримкою об’єктно-орієнтованого програмування. З іншого боку, у науковій спільноті існує думка, що велика кількість boilerplate-code шкодить розумінню самого алгоритму, приховуючи суть за деталями реалізації [16]. Оскільки програма, що створюється у рамках цієї роботи є демонстраційною моделлю для визначення можливостей та асимптотичної складності розробленого алгоритму, її фактична швидкодія не має великого значення. Виходячи з вищенаведених міркувань для реалізації програми було вибрано мову Python версії 3.7.

Окремої уваги заслуговує проблема документації програмного коду. Досить часто зустрічаються рекомендації, що вимагають супроводжувати програмний код великою кількістю коментарів. За задумкою це має полегшити розуміння коду, але існує і протилежна точка зору, яка вимагає щоб сам код був написаний зрозумілим для людини чином, а тривіальні коментарі, яки просто повторюють те, що можна легко зрозуміти із самого коду є шкідливими, оскільки часто можуть не відповідати дійсності через постійні модифікації коду, двозначність людської мови, чи з будь-яких інших причин [17]. З цієї причини автор даної роботи буде дотримуватись практики “чистого коду”, де першочергова роль носія корисної інформації відводиться саме програмному коду, а коментарі виконують допоміжну роль, та доповнюють програмний код там, де це є потрібним на думку автора.

3.2 Базові принципи реалізації алгоритму

При розбитті зони пакування на дві частини необхідно визначити конкретні розміри підзадач, що є частинами основної задачі. Оскільки на даному етапі рішення підзадач ще не знайдені, то є декілька варіантів реалізації цього процесу. У найпростішому випадку можна ділити вимір на дві однакові частини, або ж у довільній пропорції. Проте такий спрощений підхід вочевидь не є ефективним з точки зору знаходження якісного рішення задачі. Наступним кроком у напрямку покращення стратегії розділення на підзадачі є використання якого-небудь з доступних прямокутників у якості ключового елемента, що визначає пропорції, за якими відбувається розділ [18].

У випадку використання прямокутника у якості ключового елемента кінцевою метою є розміщення цього прямокутника у зоні пакування. Очевидно, що при розміщенні прямокутника у зоні пакування у загальному випадку буде виконано не один розділ, а одразу два, один горизонтальний і один вертикальний, при чому тут можливі два варіанти розділу на зони з різними розмірами у залежності від того, як розміщений ключовий прямокутник. Більш наочно це показано на рис. 3.1.

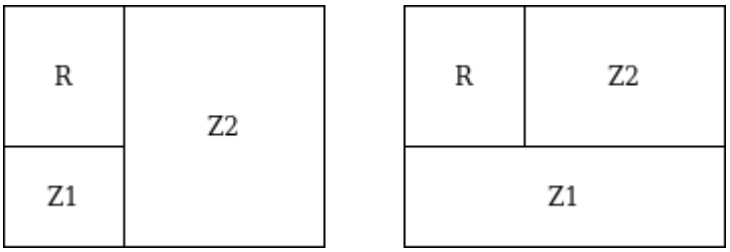

Рис. 3.1 Варіанти розділення зони пакування за допомогою ключового прямокутника

При такому розділенні існує лише два принципово відмінних варіанти, усі інші випадки з розміщеннями ключового прямокутника у різних кутах зони пакування є по суті транспозиціями цих двох ключових випадків, у чому легко впевнитись, порахувавши всі можливі транспозиції для кожного з двох наведених на малюнку 3.1 варіантів. Транспозицій існує по чотири для кожного з варіантів розділу, усього вісім можливих випадків. Оскільки розроблюваний алгоритм використовує метаструктуру для представлення результату, то на етапі вирішення задачі немає

ніякої потреби робити вибір на користь якої-небудь конкретної транспозиції оскільки вони усі потрапляють лише у два класи еквівалентності.

Інша особливість реалізації програмного коду, що випливає з наведених пояснень та ілюстрацій, це особливість будови дерева метаструктури, у якому в одному вузлі завжди комбінуються два розділи: вертикальний та горизонтальний. Крім цього, вузол метаструктури має містити ключовий елемент, який по своїй суті є листовим вузлом. Крім того, у деяких випадках одна з підзадач може виявитися замалою для того, щоб розмістити у ній будь-який з доступних прямокутників. Щоб не вводити додаткових варіантів розділу, підзадачі, що не мають не пустих рішень будуть представлені у вигляді спеціального екземпляру пустого елементу, тобто листового вузла, що не містить прямокутника. Якщо обидві підзадачі, що виникли у процесі розділу виявляться пустими, це означає, що замість внутрішнього вузла дерева метаструктури потрібно вставити листовий вузол, що містить лише даний ключовий прямокутник [19].

3.3 Дизайн ієрархії класів для представлення даних метаструктури

Для представлення даних метаструктури вирішено використати механізм поліморфізму класів шляхом створення відповідної ієрархії наслідування. Спільний інтерфейс що є предком усіх класів-нащадків має назву INode та наслідується від абстрактного базового класу ABC, що є стандартним механізмом реалізації абстрактних класів та інтерфейсів у мові Python. Для позначення абстрактних методів слугує анотація @abstractmethod, що вказується у рядку, який передує визначенню метода.

Усього передбачається чотири реалізації інтерфейсу INode, а саме: вузол з горизонтальним і потім вертикальним розділенням XSplitNode, вузол з вертикальним і потім горизонтальним розділенням YSplitNode, листовий вузол, який містить кінцевий прямокутник LeafNode та пустий листовий вузол EmptyNode. Окрім вказаних реалізацій існує додатковий абстрактний клас AbstractSplitNode, що містить спільний функціонал для XSplitNode та YSplitNode. Клас AbstractSplitNode введений для зменшення дублювання коду та покращення його придатності до модифікації, він є суперкласом для реалізацій XSplitNode та YSplitNode. Попри те, що мова Python не дає інструментів для того, щоб визначити усі поля класу незмінними, за задумкою усі

класи ієрархії є іммутабельними, тобто не передбачають зміну полів даних після створення екземпляру класу.

При перенесенні реалізації алгоритму на інші мови програмування незмінність даних може бути використана для оптимізації швидкодії, крім того, незмінні класи є хорошою практикою з точки зору якості коду, оскільки такій підхід дає можливість вважати призначені у момент створення екземпляру класу значення полів інваріантами цього екземпляру, це зменшує ймовірність помилок та повністю позбавляє проблем з синхронізацією при багатопотоковому виконанні, хоча дана властивість не є актуальною для мови Python.

Наступним кроком побудови ієрархії класів є визначення методів інтерфейсу INode. Визначення методів цього інтерфейсу по суті визначає обов’язковий функціонал для кожної його реалізації, визначаючи таким чином, які функції може виконувати метаструктура. Виходячи з принципу сегрегації інтерфейсів кожен інтерфейс має бути “тонким”, тобто містити мінімально необхідний набір методів для виконання однієї певної функції. Виходячи з семантики поняття вузла, логічним кроком стане делегація знання про геометричну структуру рішення класам-реалізаціям XSplitNode та YSplitNode. Це значно спростить взаємодію з зовнішнім кодом та дозволить уникнути дублювання коду, що виконує геометричні обчислення.

Оскільки призначення метаструктури полягає у можливості виконання транспозицій у вузлах дерева, то потрібен метод, який би приймав послідовність бажаних транспозицій та розставляв би елементи дерева у просторі відповідним чином. Тут слід зазначити, що метаструктура сама по собі ніяким чином не прив’язана до глобальної системи координат, у той час, як бажаний варіант розміщення прямокутників має використовувати конкретні координати. З цього випливає, що інтерфейс INode повинен мати метод, що дозволяє розставити вміст вузла відповідно до геометричних властивостей його конкретної реалізації. Цей метод має приймати на вхід дані що відображають потрібні транспозиції та початкові глобальні координати для розміщення його вмісту.

Метаструктура по своїй суті є деревом, що виконує рекурсивний розділ зони пакування певним чином. Виходячи з цього, найбільш природним способом розміщення вмісту метаструктури на площині є рекурсивний обхід дерева метаструктури, що у свою чергу передбачає те, що реалізації класів XSplitNode та YSplitNode, що відіграють роль внутрішніх вузлів дерева, і окрім цього будуть використовувати свої дочірні вузли через той же самий інтерфейс INode.

Для забезпечення ефективної роботи інших частин алгоритму у інтерфейс INode введені допоміжні методи, що дають змогу обчислювати параметри будь-якого піддерева. Метод get_size() обчислює компактний розмір піддерева, тобто найменший прямокутник, у який вміщується його вміст будучи складений способом, що визначає структура даного піддерева. Метод get_content() повертає послідовність унікальних ідентифікаторів, що відображають сутності прямокутників, що містяться у дереві метаструктури. Цей метод потрібен для виявлення та фільтрації еквівалентних рішень для підзадач. Крім того, вміст рішення потрібен для проріджування множини наявних прямокутників при їх використанні у підзадачах.

Для ефективної реалізації алгоритму пошуку рішення глобальної задачі потрібно мати оцінки якості рішень окремих підзадач, що виникають у процесі роботи алгоритму. Ефективністю пакування є відношення сумарної площі прямокутників до площі описаного прямокутника, але обраховувати відношення у вузлах дерева не є доцільним з точки зору правильності обчислення метрики.

Окремої уваги заслуговує абстрактний клас AbstractSplitNode, що реалізує спільну функціональність для його нащадків XSplitNode та YSplitNode. Він розширює інтерфейс INode додатковим абстрактним методом compute_bounding_boxes(), що перевизначається у класах-нащадках та служить для розрахунку розмірів підзадач для кожного з варіантів розбиття. Результуюча UML-діаграма усієї ієрархії класів метаструктури наведена на рис 3.2.

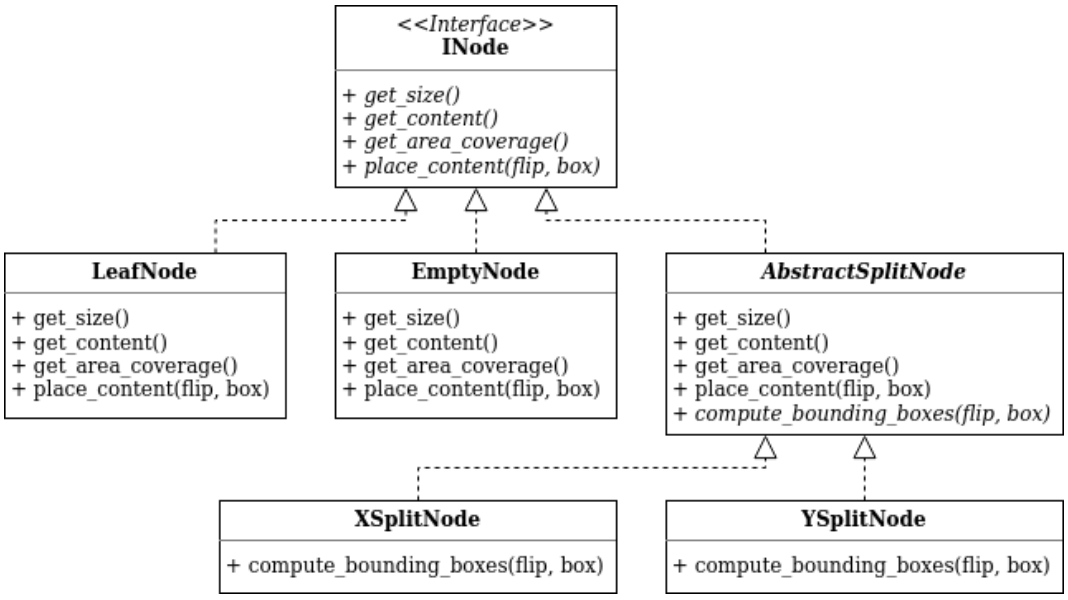

Рис. 3.2 UML діаграма інтерфейсів та класів для побудови метаструктури

3.4 Розробка класів геометричних сутностей та примітивів

У попередній частині роботи було визначено дизайн класів найвищого рівня абстракції, що репрезентують структуру кінцевого результату роботи алгоритму. З іншого боку, серед ознак хорошого об’єктно-орієнтованого дизайну програми є такі поняття як когезія та абстракція типів даних. Когезія це міра того, наскільки дані та методи, що розміщені у одному класі належать одне одному. У ідеальному випадку усі методи класу використовують усі його поля, але така ситуація не завжди можлива і більш того, не завжди доцільна. Іншим показником якості коду є рівень абстракції даних, тобто той функціонал, що відображає поведінку сутності, репрезентацією якої є екземпляр певного класу [20].

З точки зору покращення когезії, класи мають мати мінімальний функціонал, що оперує з його полями, а з точки зору абстракції даних бажано мати сутності зі складною поведінкою, яка дає потрібний інструментарій для операцій з ними. Вирішення видимого протиріччя між двома цими вимогами досягається шляхом декомпозиції складних сутностей у комбінацію більш простих, для якої вводиться додатковий функціонал [21].

Виходячи з вищенаведених міркувань, побудову класів сутностей та геометричних примітивів було вирішено проводити не у низхідному порядку, як це було зроблено з класами метаструктури, а у висхідному, шляхом створення простих сутностей з простими функціями, та комбінації їх у більш складні. При такому підході до проектування класів композиція та агрегація є більш підходящими видами відношень між окремими класами, оскільки при побудові у висхідному порядку як правило неможливо визначити спільний інтерфейс з достатнім рівнем когезії методів.

Геометричні примітиви, які потрібні для роботи з двовимірним пакуванням представлені класами Rectangle та Vector, що представляють собою відповідно абстракцію прямокутника без прив’язки до системи координат та вектор на площині. Для зручної роботи з цими типами даних було реалізовано перевизначення операторів та спеціальних методів, таких як __eq__ та __hash__, які необхідні для повноцінної роботи з колекціями. Для спрощення процесу відлагоджування програми було також перевизначено також метод __repr__, що відображає поля класу у зручному для читання вигляді.

Для представлення прямокутника, що має певне положення у системі координат використовується клас PlacedRectange, що є агрегацією класів Vector та Rectangle. Для визначення унікальних сутностей фізичних об’єктів слугує клас Item, а для визначення пари транспозицій вузла метаструктури слугує клас Flip. Як у випадку з класами метаструктури, поля класів примітивів є незмінними.

3.5 Дизайн прототипу алгоритму та допоміжних класів

Реалізація прототипу алгоритму двовимірного пакування з метаструктурою включає у себе три класи: Solver, який є власне реалізацією алгоритму та два допоміжних класи NodeCache та ContentContainer , що виконують допоміжні функції та реалізують паттерн об’єктно-орієнтованого програмування відомий як visitor. Клас NodeCache слугує для збереження готових варіантів рішення підзадачі. Клас ContentContainer виконує функцію обліку використаних прямокутників за допомогою методу get_deducted(), що повертає екземпляр класу з вилученою множиною прямокутників та має метод is_feasible() для перевірки можливості використання готового рішення підзадачі у контексті доступної множини прямокутників.

Клас Solver, що знаходить рішення двовимірного пакування складається з трьох методів, один з яких є точкою входу, а два інших рекурсивно викликають один одного, такий підхід відомий у літературі як trampolining. Обидва рекурсивних методи є генераторами мови Python, тобто виклик такого методу повертає ітератор на деяку множину елементів. Оскільки на відміну від списків генератор є лінивим, тобто виконує обчислення лише тоді, коли ітератор посувається вперед, то такий підхід є дуже зручним для перегляду великої кількості елементів без збереження їх у пам’яті, а також є незамінним, коли потрібно знайти перший елемент, що задовільняє певним вимогам у великій, або навіть нескінченній послідовності елементів.

Цікавою особливістю використання генераторів у рекурсивних викликах є можливість повертати об’єднання результатів, що отримані при ітерації по генераторам, що були рекурсивно створені в контексті даного генератора. Така структура є деревом викликів, збереженим у вигляді композиції об’єктів ітераторів, яке з точки зору користувача можна обходити як звичайний лінійний ітератор. Тобто рекурсивна композиція генераторів дозволяє реалізувати обхід дерева, причому при правильній реалізації усі операції у цьому дереві, включаючи створення його вузлів будуть відстроченими до моменту посування ітератора, тобто “лінивими”.

3.6 Опис роботи алгоритму та структура викликів

Послідовність кроків, що описують роботу алгоритму можна описати за допомогою послідовності викликів його методів. Оскільки ці методи викликаються багаторазово і рекурсивно, то вони утворюють дерево, яке незручно представляти у вигляді класичної діаграми викликів. Проте для загального уявлення про принцип роботи на рис. 3.3 дана ілюстрація послідовності роботи методів класу Solver.

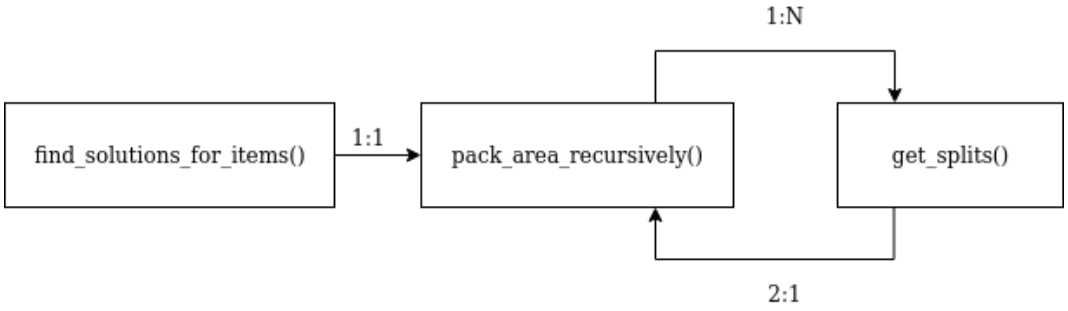

Рис. 3.3 Структурна схема рекурсивних викликів методів класу Solver

Вхідною точкою алгоритму слугує метод find_solutions() що знаходить множину рішень задачі для вхідної множини прямокутників. Вхідними даними є послідовність прямокутників та розміри зони пакування, яка також представлена окремим екземпляром прямокутника. У цьому методі створюються по одному екземпляру допоміжних об’єктів NodeCache та ContentContainer. Послідовність прямокутників передається у конструктор класу ContentContainer, після чого зі створених сутностей, що відповідають прямокутникам конструюються листові вузли LeafNode та додаються у кеш вузлів. Виклик метода pack_area_recursively() повертає ітератор, що видає послідовність усіх знайдених алгоритмом рішень задачі.

Отриманий ітератор рішень опрацьовується у звичайному циклі. Такий підхід має ряд суттєвих переваг, першою з яких слід відзначити можливість зупинити ітерацію, коли знайдено рішення, що задовольняє певним умовам, наприклад за щільністю пакування. Оскільки описана вище деревовидна структура, складена з генераторів гарантує ліниве виконання, то не потрібно застосовувати спеціальних заходів для зменшення розміру дерева пошуку. Іншою перевагою є простота підрахунку умовної кількості ітерацій, і хоча дана оцінка не є абсолютно точною через

те, що вона не враховує пусті рішення, ця метрика асимптотично зходиться з кількістю виконаних обчислень.

Для демонстрації роботи програми верхній цикл, що обробляє послідовність рішень алгоритму відстежує найкраще знайдене рішення та веде підрахунок загальної кількості знайдених рішень. Крім цього, для найкращих знайдених рішень запускається програма візуалізації, що відображає один з варіантів, що генерує метаструктура знайденого рішення. Візуалізація результатів та вхідних даних здійснюється за допомогою бібліотеки Matplotlib.

3.7 Аналіз складності роботи алгоритму

Будова даної реалізації алгоритму заснована на припущенні щодо рівномірного розподілу “якісних” рішень у дереві пошуку в глибину, яке утворюється у процесі роботи алгоритму. За припущення, що задовільне рішення з однаковою ймовірністю знаходиться у будь-якому листовому елементі дерева, середній час роботи алгоритму до знаходження першого задовільного рішення визначається лише статистикою розподілу цих рішень, визначеним рівнем “якості” та середньою кількістю прямокутників у рішенні. Проте, з теоретичної точки зору було би цікавим оцінити розмір усієї множини рішень, що генерує алгоритм як функцію розміру вхідних даних, при чому через комбінаторну природу задачі простір рішень швидко стає завеликим для повного перебору. Інший аспект, який варто мати на увазі це те, що простір рішень, що генерує алгоритм є значно менший за множину усіх можливих розміщень прямокутників, тому завжди є ймовірність, що алгоритм не знайде рішення з певним рівнем щільності упаковки, і цей факт потрібно враховувати при обробці результатів пошуку.

Для аналізу складності алгоритму зробимо спрощуюче припущення, що обчислювальна складність асимптотично пропорційна розміру дерева рекурсії. На справді, як можна побачити з реалізації коду, вартість фактичних операцій, що виконується у вузлах дерева росте лінійно зі збільшенням кількості прямокутників. Цей множник буде враховано при обчисленні асимптотичної складності алгоритму, проте оскільки для будь-якої NP-складної задачі час роботи у найгіршому випадку має експоненційну залежність від розміру вхідних даних, то поліноміальний множник впливає на асимптотичну складність лише тоді, коли він знаходиться під знаком експоненти.

Для подальшого аналізу потрібно побудувати параметризоване дерево рекурсії та дати оцінку щодо глибини вкладеності рекурсивних викликів. Це досить легко зробити для найгіршого випадку, проте для аналізу потрібно мати оцінку середнього значення. Оцінка середньої кількості елементів пакування буде взята наступного вигляду:

$$\overline{N}_R = M\left(a_1/d_1\right) * M\left(a_2/d_2\right) \sim \frac{S_a}{M\left(S_r\right)} = k\frac{S_a}{\sum S_r}$$

Де $\overline{N}_R$ - математичне очікування середньої кількості прямокутників в упаковці, $M()$ - оператор взяття оцінки математичного очікування, a_1, a_2 - розміри зони пакування, d_1, d_2 - розміри окремо взятого прямокутника, S_a- площа зони пакування, S_r - площа окремо взятого прямокутника, k – кількість прямокутників у вхідному наборі даних.

Оскільки при кожному виклику методу get_splits() відбувається зменшення розміру залишкової зони пакування на розмір ключового елемента, при чому для однієї з зон зменшується лише один розмір, а для другої зменшуються обидва розміри (див. рис. 3.1), то для однієї з гілок підходить оцінка $\sqrt{\overline{N_R}}$, а для другої $\overline{N_R}$. Але оскільки у загальному випадку відбувається чергування випадковим чином, то доцільно взяти геометричне середнє між цими двома величинами, у результаті чого отримуємо оцінку математичного очікування глибини рекурсії:

$$D_R \sim \left(\overline{N}_R\right)^{3/4} \sim \left(k\frac{S_a}{\sum S_r}\right)^{3/4}$$

Де $\overline{N}_R$ - математичне сподівання кількості прямокутників у рішенні, S_a - площа зони пакування, S_r - площа окремо взятого прямокутника, k – кількість прямокутників у вхідному наборі даних, $\overline{D}_R$- оцінка математичного сподівання глибини рекурсії.

Наступним кроком для отримання оцінки складності є оцінка математичного очікування арності вузлів дерева рекурсії, тобто параметра, що відповідає за швидкість його розширення при збільшенні глибини. Оскільки у коді реалізована рекурсія з чергуванням (trampolining), то для спрощення розрахунків одним вузлом дерева рекурсії будемо рахувати кожен послідовний виклик цих двох методів, а кількість вихідних гілок, тобто вузлів-нащадків приймемо рівною добутку кількості рекурсивних викликів в обох методах:

$$K_{total} = K_1 * K_2 = K_1 * 2 = R_{unused} * 2$$

Де K_{total}- загальна кількість наскрізних рекурсивних викликів, K_1- кількість рекурсивних викликів у методі pack_area_recursively(), K_2- кількість рекурсивних викликів у методі get_splits(), R_{unused}- кількість невикористаних елементів у кешу вузлів, що використовуються у якості ключового елемента.

Оцінка кількості невикористаних елементів в залежності від глибини рекурсії ускладнює аналіз, тому для того, щоб оцінити увесь діапазон значень цього параметра пропонується розглянути два крайні випадки: а) площа прямокутників дорівнює площі зони пакування, задача “пазл” та б) площа доступних прямокутників значно більша за площу пакування, і вичерпанням варіантів у процесі пакування можна знехтувати.

Спочатку розглянемо другий випадок, оскільки він простіший та дає більш песимістичну оцінку складності. Позначимо кількість доступних прямокутників на і-тому кроці рекурсії через R_i, при чому для другого випадку $R_i \approx k$, де k – кількість прямокутників у вхідних даних. Тоді оцінка загальної кількості наскрізних рекурсивних викликів буде виглядати так:

$$\overline{F}_c = \prod_{i=1}^{\overline{D}_R} \overline{R}_i \sim \overline{R}_i^{\overline{D}_R} \approx k^{\overline{D}_R}$$

Де $\overline{F}_c$ - кількість викликів метода, $\overline{R}_i$ - оцінка кількості доступних прямокутників на і-му кроці рекурсії, $\overline{D}_R$ - оцінка математичного сподівання кількості кроків рекурсії.

Для варіанту з вичерпанням доступних прямокутників необхідно розглянути два випадки, оскільки дерево викликів у методі get_splits() розгалужується на дві частини з різним рівнем вичерпання ресурсів. За припущення, що дерево рекурсії збалансоване, на першому виклику метода pack_area_recursively() кількість доступних прямокутників зменшується лише на одиницю за рахунок використання ключового елемента, у той час як на при другому виклику інша підзадача вже вирішена, а потрібні для неї прямокутники вилучені із множини доступних. Оскільки у другому випадку сумарна площа прямокутників дорівнює площі зони пакування, то оцінку доступної кількості прямокутників можна отримати з оцінки залишкової площі зони пакування. При чому для першої гілки це буде оцінка кількості для всієї підзадачі за винятком одного прямокутника, а для другої це буде розмір однієї з підзадач. Якщо знехтувати самим ключовим елементом, то можна вважати, що розмір однієї з підзадач відповідає початковій задачі, що можна наближено змоделювати шляхом збільшення глибини рекурсії на одиницю.

Ефективна наближена кількість прийнята за 50% від початкового значення, в результаті отримуємо:

$$\overline{F}'_c = \prod_{i=1}^{\overline{D}_R} \overline{R}_i \sim (k/2)^{\overline{D}_R+1}$$

Де $\overline{F}'_c$ - кількість викликів метода (оптимістична оцінка), $\overline{R}_i$ - оцінка кількості доступних прямокутників на і-му кроці рекурсії, $\overline{D}_R$ - оцінка математичного сподівання кількості кроків рекурсії, k-кількість прямокутників у вхідних даних. Оскільки дана оцінка дана для оптимістичного випадку, то тут слід також використати оптимістичну оцінку для значення $\overline{D}_R$, виходячи з того, що у найкращому випадку дерево рекурсії буде збалансованим, можемо визначити оптимістичну оцінку глибини рекурсії:

$$\overline{D}'_R \approx \log_2(k)$$

де k – кількість прямокутників у вхідних даних, $\overline{D}'_R$ - оцінка глибини рекурсії.

$$\overline{F}'_c \sim k^{\log_2 k+1} = k * k^{\log_2 k}$$

Де $\overline{F}'_c$ - оптимістична оцінка кількості викликів метода, k – кількість вхідних прямокутників. Для наочної демонстрації поведінки цих двох оцінок нижче наведено їх графік для випадку “пазл-задачі” у лінійно-логарифмічному масштабі (рис. 3.4).

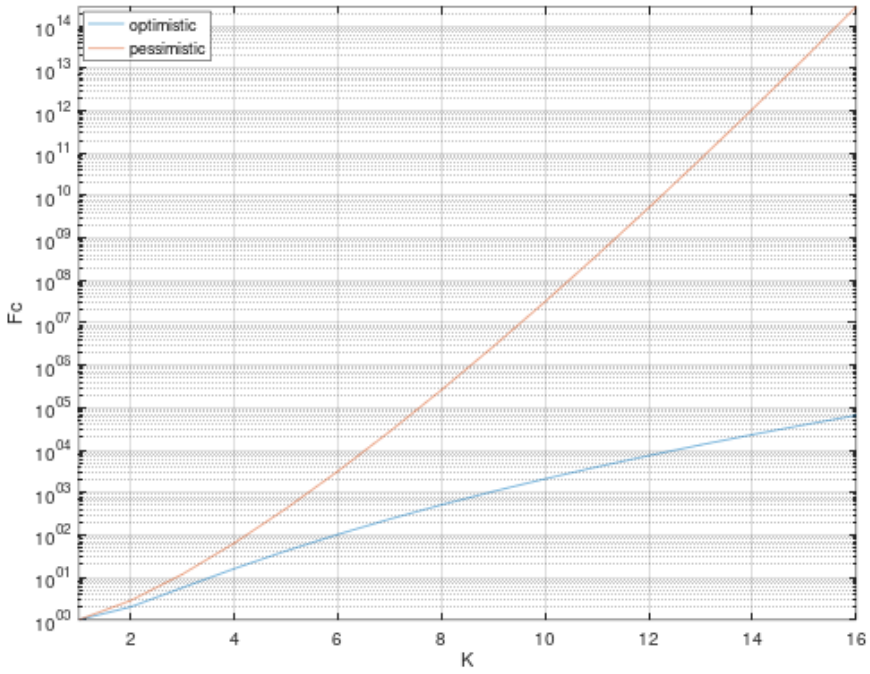

Рис. 3.4 Поведінка оптимістичної та песимістичної оцінок складності перебору усіх рішень

4 Визначення основних характеристик вхідних даних

4.1 Вплив вхідних даних на роботу алгоритму

У попередньому розділі було оцінено складність алгоритму як функцію від розміру вхідних даних, тобто кількості прямокутників. При цьому ігнорувалися параметри розподілу їх розмірів та пропорцій, дискретність розмірів, тощо. Очевидно, що ці параметри мають значний вплив як на час роботи алгоритму, так і на якість отриманого результату, проте виявити та оцінити усі можливі залежності у рамках цієї роботи не є можливим. Тому для практичних дослідів з пакування прямокутників та визначення впливу параметрів було вибрано декілька основних напрямів: а) тестування типу “пазл” (сумарна площа прямокутників дорівнює площі зони пакування) з невеликою кількістю прямокутників та дискретними значеннями величин їх розмірів, б) те саме, але розміри не дискретні, в) задача невеликого розміру, де сумарна площа прямокутників значно перевищує площу зони пакування та г) задача великого розміру, де майже будь-яке знайдене рішення буде мати задовільну щільність упаковки.

Для задач типу “пазл” з невеликою кількістю прямокутників буде шукатись рішення з ефективністю 100%, тобто буде перевірено, чи здатен алгоритм скласти пазл, “розрізаний” за певними правилами. Задача типу в) є найскладнішою з точки зору обчислювальної складності, проте це залежить від бажаного рівня якості рішення. Останній тип задачі має на меті перевірку складності обчислення одиничного рішення та “лінивості” обчислень.

4.2 Способи генерації вхідних даних

Для отримання параметрів з потрібним випадковим розподілом буде використовуватись бібліотека стандартна random мови Python версії 3.7. Для отримання повторюваних результатів перед початком роботи кожного тесту внутрішній стан генератора псевдовипадкової послідовності буде ініціалізуватися шляхом виклику функції seed() з унікальним цілочисельним значенням. Дана бібліотека дозволяє генерувати різні статистичні розподіли, проте для даної роботи буде використовуватися в рівномірний розподіл. Для уніфікації частин програми та можливості повторного використання коду не дискретні розміри будуть апроксимуватись цілочисельними значеннями, достатньо великими щоб нівелювати вплив їх дискретності.

5 Визначення методів тестування алгоритму

5.1 Перевірка складових частин алгоритму за допомогою юніт-тестів

Написання юніт-тестів є загальноприйнятою практикою, що полегшує відлагоджування коду та внесення у нього змін завдяки ранньому виявленню помилок при виконанні юніт тестів. Тому усі файли з нетривіальними класами використовують стандартну бібліотеку unittest мови Python 3.7. Дана бібліотека являє собою фреймворк для побудови юніт-тестів шляхом наслідування від базового класу TestCase, що надає такі інфраструктурні елементи як assertions, хук-методи для ініціалізації та де-ініціалізації тестового оточення та інтеграцію окремих юніт-тестів у єдиний набір.

Особливістю тестування у мові Python є динамічна типізація, що робить непотрібними такі інструменти як harmctrest matchers та спеціальні mock-класи, та значно спрощує написання самих юніт-тестів. З іншого боку, у рамках даної роботи не ставиться яких-небудь вимог до покриття юніт тестами, тому вони використовуються лише там, де на думку автора дозволять уникнути помилок та допоможуть швидко перевірити працездатність складових частин проекту.

5.2 Оцінка фактичної кількості операцій

Для того, щоб перевірити коректність теоретичних оцінок складності алгоритму та порівняти їх з фактичними вимірами, необхідно визначити методику проведення цих вимірів. У якості першого наближення пропонується використовувати таймер, лічильник кількості викликів певного метода або ж лічильник кількості результатів, що видає генератор. Для перевірки складності обчислення одиничного рішення для великого екземпляру задачі пропонується використати таймер, оскільки таке вимірювання неможливо виконати за допомогою лічильника. Усі інші види вимірювань пропонується проводити шляхом встановлення лічильника або ж підрахунку кількості виданих рішень, у теорії ці дві величини повинні мати асимптотично однакову поведінку, якщо це буде підтверджено експериментально, то усі інші виміри будуть базуватись лише на кількості отриманих рішень.

6 Алгоритм генерації вхідних даних

6.1 Алгоритм генерації вхідних даних для задачі типу “пазл”

Ідеєю, що лежить у основі задачі типу “пазл” є створення набору прямокутників, що за сумарною площею дорівнюють зоні пакування та можуть бути розміщені у ній з урахуванням обмежень, що накладені на структуру пакування, що продукує досліджуваний алгоритм. Оскільки таким обмеженням є гільйотинованість, то найбільш природним способом створення задачі “пазл” є послідовний розділ зони пакування на частини. При цьому потрібно контролювати характер розділення, а саме: контролювати мінімальний розмір прямокутника, комбінувати горизонтальний та вертикальний поділ і використовувати для керування процесом поділу генератор псевдовипадкових чисел.

Крім обмеження гільйотинованості, початкова реалізація алгоритму не використовує зустрічний пошук з висхідним конструюванням проміжних результатів, і це накладає додаткові обмеження на можливу структуру рішення, оскільки у такому спрощеному випадку ключовий елемент розбиття завжди є одиничним прямокутником, і не може складатися з комбінації декількох прямокутників. Для оцінки роботи алгоритму було би досить корисно мати алгоритм генерації даних, що може створювати дані як з урахуванням цього обмеження, що буде гарантувати можливість вирішення “пазлу” алгоритмом, так і без нього, щоб оцінити ймовірність пакування всіх прямокутників для довільного гільйоттинованого розрізу.

Алгоритм генерації вхідних даних для задачі типу “пазл” реалізовано у класі PuzzleSplitter, конструктор якого приймає на вхід параметри, що контролюють розділ прямокутника на частини. Для генерації даних з обмеженнями використовується клас ConstrainedPuzzleSplitter, що має дещо відмінну логіку розділу зони на прямокутники.

6.2 Генерація даних для великої задачі

Для генерації прямокутників довільних розмірів використовується генератор псевдовипадкових цілих чисел у заданому інтервалі з стандартної бібліотеки random.

7 Результати роботи алгоритму на різних типах вхідних даних

7.1 Результати пакування для задачі типу “пазл”

Даний тип задачі являє собою найбільш складний випадок, тому пропонується розбити його на два підвиди: а) невеликий екземпляр задачі, наприклад *R=12* зі значною дискретністю розмірів, для якого пропонується знайти рішення з якістю упаковки 100% та б) великий екземпляр задачі, для рішення якої буде виділено фіксовану кількість ітерацій, наприклад *N=10000* та оцінено якість найкращого результату для кожного з наборів вхідних даних.

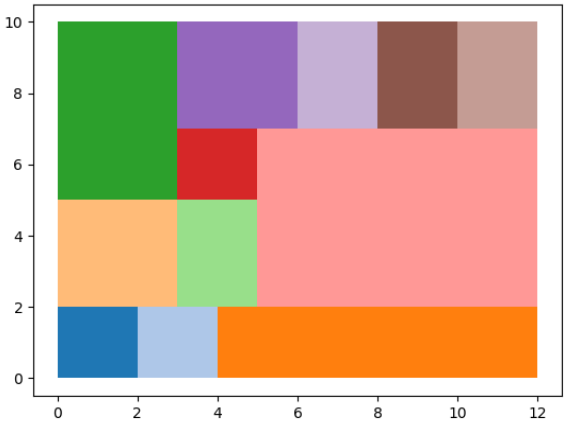

Рис. 7.1 Приклад вхідних даних для задачі типу “пазл”, N=12

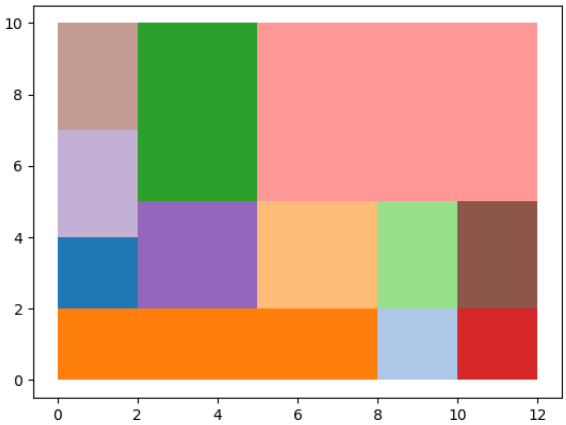

Рис. 7.2 Рішення, знайдене алгоритмом для вхідних даних з рис. 7.1

7.2 Оцінка розміру простору рішень та порівняння з теоретичними оцінками

Особливістю алгоритму генерації вхідних даних типу “пазл” є неможливість задавати точну кількість прямокутників, що буде отримано. Замість цього, випробування проводиться лише для тих екземплярів даних, що потрапляють у задані межі за кількістю прямокутників. Для утримання часу виконання експерименту у прийнятних межах максимальний розмір задачі для цього випробування було обмежено на рівні *K=12*, оскільки перебір усіх рішень є ресурсоємною операцією. Було виконано *N=70* запусків модифікованої для підрахунку кількості рішень алгоритму з випадково згенерованими вхідними даними. Модифікація знадобилася через те, що механізм який прибирає рішення підзадач, що мають однаковий вміст зменшує кількість рішень які потрапляють у результат до декількох десятків. Для даного випробування цей механізм був відключений, що дозволило отримати на виході усі можливі комбінації, що виникають у процесі вирішення задачі, та підрахувати їх кількість для кожного екземпляру задачі.

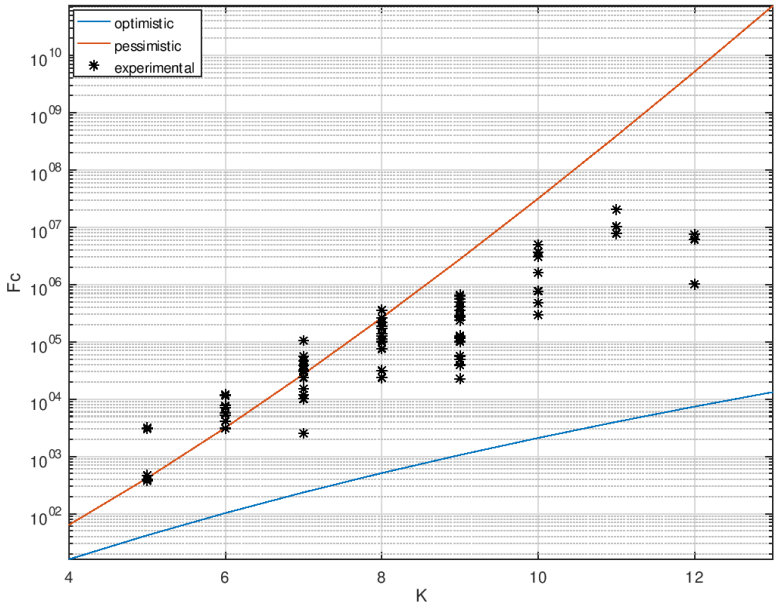

Рис 7.3 Порівняння результатів експерименту з теоретичними результатами

На рис. 7.3 приведено порівняння кількості варіантів, що отримано у результаті експерименту з теоретичною оцінкою цих значень. Теоретичні оцінки складності є асимптотичними, тобто такими, що можуть відрізнятися на деякий константний множник, та

можуть не виконуватись для деяких достатньо малих екземплярів задачі. З отриманих результатів досить складно скласти оцінку правильності асимптотичних меж, оскільки зробити експериментальний підрахунок для великих екземплярів задачі є проблематично через значну обчислювальну складність. Експериментальні дані для $K > 8$ знаходяться у межах, передбачених теоретичним аналізом, для менших екземплярів середнье значення знаходиться близько до песимістичної оцінки, але досить велика кількість випадків перевищує її, у найгіршому випадку на порядок. Проте, для експоненційного характеру зростання кількості рішень така помилка не свідчить про суттєву розбіжність, оскільки вона відповідає збільшенню задачі на один-два прямокутники.

7.3 Перевірка асимптотичної складності знаходження одного рішення

Для того щоб впевнитись, що проходження по дереву пошуку від початку до наступного знайденого рішення має у найгіршому випадку квадратичну складність від розміру вхідної задачі доведеться використати профілювання з виміром часу, оскільки підрахунок кількості кроків при роботі з колекціями не є репрезентативним. У першому наближенні можна використати тривіальну задачу, наприклад упакувати зону пакування деякою кількістю квадратів одиничного розміру. У такій задачі кожна з гілок дерева рішень дає оптимальний результат, а отже консистентну глибину рекурсії. Інший спосіб вимірювання полягає у використанні довільних прямокутників із загальною площею достатньою для покриття зони пакування. У такому випадку можна виміряти залежність часу роботи алгоритму як від кількості вхідних прямокутників, так і від кількості прямокутників, що увійшли у рішення.

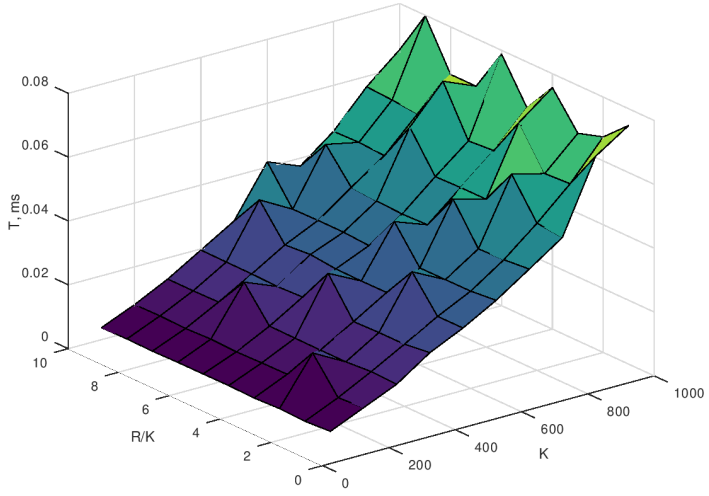

Рис. 7.4 Залежність часу знаходження одного рішення від кількості прямокутників

Як видно з рис 7.4, залежність часу знаходження одиничного рішення для тривіальної задачі практично не залежить від відношення кількості вхідних даних до кількості прямокутників у рішенні задачі (вісь *R/K*). Залежність від кількості прямокутників у рішенні близька до лінійної, для уточнення цієї залежності були проведені виміри на ширшому діапазоні вхідних значень, причому параметр *R/K* було встановлено рівним 1.

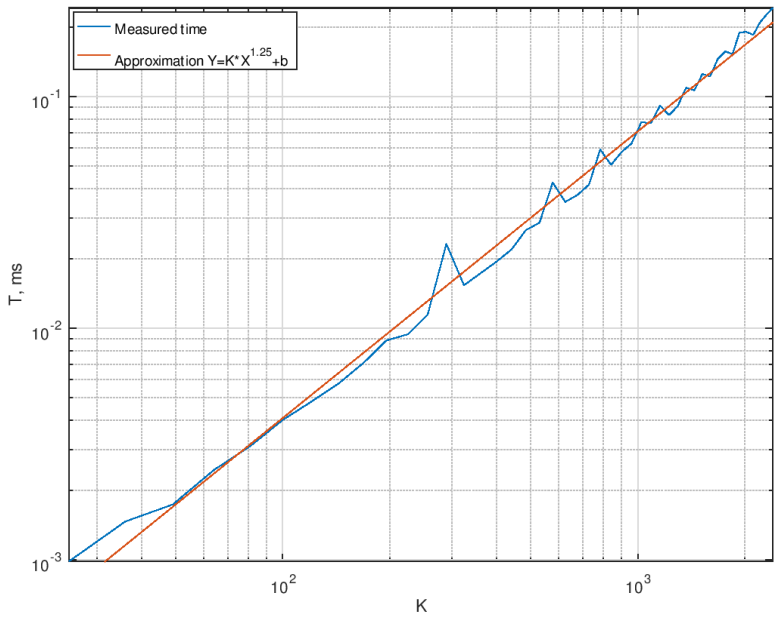

Рис. 7.5 Залежність часу роботи від кількості прямокутників у широкому діапазоні

Як видно з результатів на рис. 7.5, час роботи алгоритму має консистентну залежність від розміру вхідних даних, що у широкому діапазоні значень добре апроксимується залежністю $O(n^{1.25})$.

7.4 Час роботи алгоритму з відсіканням еквівалентних результатів

Наведені вище результати були отримані на спрощеній версії алгоритму, що не фільтрує ідентичні результати. Ця частина роботи присвячена перевірці часу роботи алгоритму у випадках близьких до практичних задач, для чого буде використовуватись повнофункціональна версія з фільтрацією рішень підзадач, що мають однаковий вміст. З одного боку це призведе до зменшення множини результатів, що видає ітератор верхнього рівня, а з іншого збільшиться час, витрачений на знаходження кожного окремого рішення. При чому сумарний час обрахунку простору рішень повинен зменшитися відносно версії з вимкненою фільтрацією.

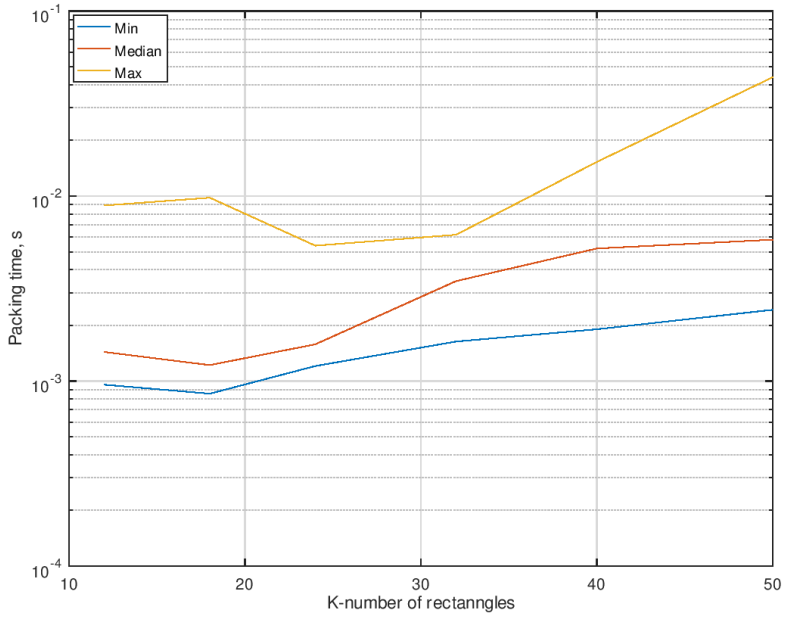

Рис. 7.6 Час роботи алгоритму з фільтрацією, знаходяться три результату

На рис. 7.6 зображено результат тестування алгоритму на випадково згенерованих даних, наближених до реальних. Відношення кількості прямокутників у вхідних даних до очікуваної

кількості прямокутників у рішенні $R/K = 3$. Різниця витрати часу між найгіршим та найкращим випадком для 10 виконаних для кожного значення K випробувань складає приблизно один порядок, що є характерним для алгоритмів двовимірного пакування, оскільки їх робота у значній мірі залежить від конкретного екземпляру вхідних даних.

Крім витраченого часу була також виміряна ефективність для кращого з перших трьох знайдених рішень. Це значення не є найкращім значенням, яке може знайти алгоритм, натомість це значення, яке може бути знайдено за визначений проміжок часу шляхом обмеження кількості знайдених рішень до перших трьох. Тестовий випадок використовує дані з великим кроком дискретності розмірів прямокутників, що робить можливим досягнення 100% ефективності, на неперервних розподілах розмірів очікувана ефективність буде нижчою, тому дані результати не призначені для порівняння з іншими алгоритмами, вони слугують лише для виявлення залежностей від кількості вхідних даних.

На рис. 7.7 зображені результати вимірів ефективності пакування для того ж випробування, з якого отримані дані на рис. 7.6. Як видно з графіку, для кожного варіанта розміру вхідних даних спостерігались рішення зі 100% ефективністю, так само як і рішення з меншою ефективністю пакування. Це може свідчити про те, що пошук більшої кількості рішень задачі може потенційно покращити щільність пакування за рахунок збільшення часу роботи програми.

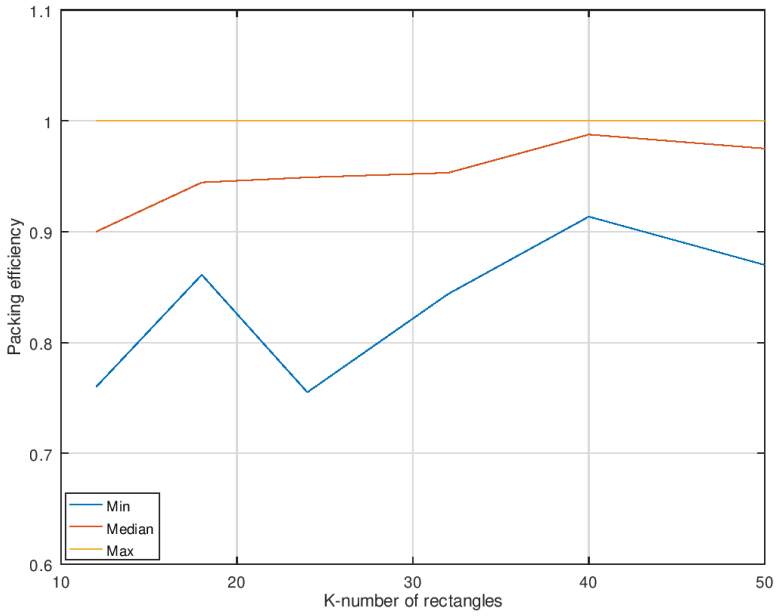

Рис. 7.7 Ефективність пакування прямокутників, розміри кожної сторони від 1 до 4

7.5 Аналіз отриманих результатів

Проведені випробування алгоритму дозволили перевірити та уточнити результати теоретичної оцінки його обчислювальної складності. Загальна асимптотична складність при послідовному розгляді усього простору рішень має асимптотичну складність $O\left(n^{n^{3/4}}\right)$, проте для окремого рішення складність знаходження склала $O\left(n^{5/4}\right)$де n це кількість прямокутників у рішенні задачі. Як і очікувалось, спостерігається сильна залежність часу роботи алгоритму від статистичного розподілу вхідних даних, при чому різниця між окремими екземплярами задачі може досягати одного порядку. Такий великий розкид часу роботи є характерним для алгоритмів пошуку, що мають експоненційну складність та демонструють високу чутливість часу роботи до вхідних даних. Залежність часу знаходження одного рішення від кількості додаткових прямокутників при їх кількостях, що не перевищують десятикратний розмір задачі при практичних вимірюваннях не виявлено. Висока дискретність розмірів прямокутників у вхідних даних призводить до покращення якості пакування як і очікувалось. Вимірювання часу роботи алгоритму підтвердили “лінивість” ітераторів дерева пошуку.

Висновки

Дана робота представляє собою розробку алгоритму двовимірного пакування прямокутників, що використовує так звану метаструктуру для забезпечення можливості створювати різноманітні варіанти одного й того ж рішення задачі за лінійний час. Розробка алгоритму була розпочата з огляду існуючих підходів, що послужили основою для розроблюваного принципу, що базується на поділі зони пакування на три частини та послідовному вирішенні підзадач меншого розміру. У роботі були дані оцінки асимптотичної складності алгоритму, була створена та описана його програмна реалізація на мові програмування Python та проведено вимірювання його основних характеристик. У завершувальній частині роботи приведені отримані результати та проведено їх зіставлення з теоретично оціненими значеннями, що демонструють адекватність теоретичних розрахунків співвідносно з отриманими результатами. Крім того, у програмній реалізації алгоритму було застосовано “ліниві” ітератори на основі динамічного дерева з генераторів, що дозволяють виконувати пошук в глибину шляхом простої ітерації по елементам генератора верхнього рівня без створення усього дерева пошуку у пам’яті.

Для оцінки переваг розробленого алгоритму потрібно його суміщення з алгоритмом, що оптимізує положення об’єктів з використанням властивостей метаструктури. Це дослідження знаходиться поза межами проблематики, що розглядається у цій роботі та пропонується як один з можливих напрямків для її подальшого розвитку. Інші можливі напрямки розвитку прототипу алгоритму двовимірного пакування включають використання динамічного програмування для повторного використання рішень підзадач та реалізацію алгоритму зустрічного пошуку, що демонструє хороші результати при застосуванні до різноманітних задач комбінаторного пошуку.

Література

1. Blum C. Solving the 2D bin packing problem by means of a hybrid evolutionary algorithm / C. Blum, S. Verena. // Procedia Computer Science. – 2013. – №18. – С. 899 – 908.

2. Karabulut K. A Hybrid Genetic Algorithm for Packing in 3D with Deepest Bottom Left with Fill Method / K. Karabulut, M. M. Inceoglu. // ADVIS. – 2004. – №3261. – С. 441–450.

3. Alvaro A. The Two-Dimensional Rectangular Strip Packing Problem : дис. докт. техн. наук / Alvaro Alvaro Luiz Junior – Porto, 2017. – 148 с.

4. Lodi A. TSpack: A Unified Tabu Search Code for Multi-Dimensional Bin Packing Problems / A. Lodi, S. Martello, D. Vigo. // Annals of Operations Research. – 2004. – №131. – С. 203–213.

5. Jansen K. An Approximation Scheme for Bin Packing with Conflicts / Klaus Jansen. // Journal of Combinatorial Optimization. – 1999. – №3. – С. 363–377.

6. Hopper E. A Review of the Application of Meta-Heuristic Algorithms to 2D Strip Packing Problems / E. Hopper, H. Turton. // Artificial Intelligence Review. – 2001. – №16. – С. 257–300.

7. Gu X. Average-Case Performance Analysis of a 2D Strip Packing Algorithm—NFDH / X. Gu, G. Chen, Y. Xu. // Journal of Combinatorial Optimization. – 2005. – №9. – С. 19–34.

8. Russel S. J. Artificial Intelligence A Modern Approach / S. J. Russel, P. Norvig. – New York: Prentice Hall, 2010. – 1132 с.

9. Cormen T. Introduction to Algorithms / T.Cormen, C. Leiserson, R. Rivest, C. Stein. – Boston: MIT Press, 2001. – 1294 с.

10. Wei L. A skyline heuristic for the 2D rectangular packing and strip packing problems / L.Wei, W. Oon, W. Zhu, A. Lim. // European Journal of Operational Research. – 2011. – №215. – С. 337–346.

11. Filippi C. An asymptotically exact algorithm for the high-multiplicity bin packing problem / C. Filippi, A. Agnetis. // Math. Program., Ser. A. – 2005. – №104. – С. 21–37.

12. Knuth D. The Art of Computer Programming, Volume 4 / Donald Knuth. – Boston: Addison-Wesley Professional, 2006. – 1315 с.

13. Sedgewick R. Algorithms / R. Sedgewick, K. Wayne. – Boston: Addison-Wesley Professional, 2011. – 976 с.

14. Friedow I. Using Contiguous 2D-Feasible 1D Cutting Patterns for the 2D Strip Packing Problem / I. Friedow, G. Scheithauer. // Operations Research Proceedings. – 2017.

15. Gu X. Performance Analysis and Improvement for Some Linear On-Line Bin-Packing Algorithms / X.Gu, J. Gu, L. Huang, Y. Jung. // Journal of Combinatorial Optimization. – 2002. – №6. – С. 455–471.

16. Matrin C. Clean Code: A Handbook of Agile Software Craftsmanship / C. Martin Robert. – New York: Pearson, 2017. – 297 с.

17. Gamma E. Design Patterns: Elements of Reusable Object-Oriented Software / E.Gamma, R. Helm, R. Johnson, J. M. Vlissides. – Boston: Addison-Wesley, 1994. – 395 с.

18. Jansen K. New Algorithmic Results for Bin Packing and Scheduling / Klaus Jansen. // CIAC. – 2017. – С. 10–15.

19. Osogami T. Local Search Algorithms for the Bin Packing Problem and Their Relationships to Various Construction Heuristics / T. Osogami, H. Okano. // Journal of Heuristics. – 2003. – №9. – С. 29–49.

20. Bugayenko Y. Elegant Objects, Volume 1 / Yegor Bugayenko., 2016. – 230 с.

21. McLaughlin B. Head First Object-Oriented Analysis and Design: A Brain Friendly Guide to OOA&D / B. McLaughlin, G. Pollice, D. West., 2006. – 412 с.