УДК 004.054+004.047

ВИКОРИСТАННЯ МОЖЛИВОСТЕЙ ТЕСТУВАННЯ ПРОГРАМ НА ВІДДАЛЕНИХ СЕРВЕРАХ ДЛЯ ПОРІВНЯННЯ ЕФЕКТИВНОСТІ МЕТОДІВ КОМБІНАТОРНОЇ ОПТИМІЗАЦІЇ
USING PROGRAM TESTING OPPORTUNITIES ON REMOTE SERVERS FOR COMPARING THE EFFICIENCY OF COMBINATORY OPTIMIZATION METHODS

Shportko A. V.[1], Mushyn M. M.[2]
[1,2]Міжнародний економіко-гуманітарний університет імені академіка Степана Дем'янчука, м. Рівне, Україна
ORCID: [1] https://orcid.org/0000-0002-4013-3057, [2]https://orcid.org/ 0009-0005-4106-1920
E-mail: [1]ITShportko@ukr.net, [2]m.mushyn@ukr.net

DOI: 10.15673/atbp.v%vi%i.2497

Анотація. *В статті обгрунтована доцільність використання можливостей тестування програм на віддалених серверах для порівняння ефективностей реалізацій різних методів розв'язання обраної задачі комбінаторної оптимізації. Описано метод поступового формування множини значень цільової функції як альтернативного методам пошуку з поверненнями та врахування змін. Пояснено механізм дії алгоритмів, які застосовують ці методи для розв'язування спрощеного варіанту класичної задачі пакування рюкзака. Наведено фрагменти програм, які реалізують дані алгоритми мовою програмування C# та проаналізовано результати їх тестування у віддаленому обчислювальному середовищі. За результатами тестування показано, що реалізація методу поступового формування множини допустимих значень кардинально зменшує час виконання програм у порівнянні з реалізаціями інших методів, що вказує на його ефективність.*

За результатами дослідження зроблено основні висновки про те, що, по-перше, для прискорення розв'язування задач комбінаторної оптимізації недостатньо оминати деякі варіанти повного перебору, а потрібно мінімізувати ще й час обчислення цільової функції для кожного варіанту, враховуючи обмеження задачі. По-друге, метод поступового формування множини допустимих значень цільової функції є дієвою альтернативою методам пошуку з поверненнями та врахування змін при розв'язуванні задач комбінаторної оптимізації, якщо область значень дискретна, а хід розв'язання подібний до методу динамічного програмування. І, по-третє, для визначення найефективнішого способу розв'язування задачі комбінаторної оптимізації недостатньо порівнювати час виконання на відомих тестових наборах, а й потрібно намагатися попередньо проаналізувати їх обчислювальну складність.

Abstract. *The article substantiates the feasibility of using the capabilities of testing programs on remote servers to compare the effectiveness of implementations of various methods of solving the selected combinatorial optimization problem. The method of gradually forming a set of values of the objective function as an alternative to the methods of search with returns and taking into account changes is described. The mechanism of action of the algorithms that use these methods to solve a simplified version of the classical problem of packing a backpack is explained. Fragments of programs that implement these algorithms in the C# programming language are presented, and the results of their testing in a remote computing environment are analyzed. According to the test results, it is shown that the implementation of the method of gradually forming a set of admissible values drastically reduces the execution time of programs in comparison with the implementation of other methods, which indicates its effectiveness.*

Based on the results of the research, the main conclusions were made. Firstly, to speed up the solution of combinatorial optimization problems, it is not enough to bypass some variants of a complete search, and it is also necessary to minimize the time of calculating the objective function for each variant, taking into account the limitations of the problem. Secondly, the method of gradually forming a set of admissible values of the objective function is an

effective alternative to the methods of search with returns and taking into account changes when solving combinatorial optimization problems, if the range of values is discrete and the solution process is similar to the method of dynamic programming. And, thirdly, to determine the most effective way of solving the combinatorial optimization problem, it is not enough to compare the execution time on known test sets, but also to try to analyze their computational complexity in advance.
Ключові слова: модульне тестування, метод поступового формування множини значень цільової функції.
Key words: unit testing, method of gradual formation of the set of target function values.

Вступ

Як відомо, комбінаторика вирішує задачі вибору та розташування елементів множини відповідно до заданих правил. Вона застосовується, наприклад, у криптографії, створенні штучних нейромереж, олімпіадному програмуванні чи при вирішенні задач із інших сфер. Хоча формули комбінаторики можуть бути простими для алгоритмізації, але їх використання для великих вхідних даних може суттєво сповільнювати роботу окремих програм та в цілому обчислювальних систем. Саме тому розвиток існуючих і створення нових методів оптимізації розв'язування комбінаторних задач є на сьогодні **важливим науковим завданням**, яке розв'язується в межах комбінаторної оптимізації [1].

Аналіз літературних даних і постановка проблеми

Комбінаторна оптимізація розглядає завдання оптимізації, в яких множина допустимих розв'язків дискретна або може бути зведене до дискретної. Метою таких завдань є відшукання деякого оптимального елемента з дискретної множини, для визначення якого за відведений час недостатньо загальновідомих способів пошуку з використанням повного перебору [1].

Традиційно для прискорення повного перебору використовують **метод врахування змін** [2], який розраховує наступний варіант розв'язку не з самого початку, а лише враховує зміни відносно попереднього варіанту розв'язку. Але найвідомішим методом для розв'язування задач комбінаторної оптимізації є **пошук з поверненнями** [3]. При його використанні часткове рішення розширюється і надалі аналізуються отримані неповні «кандидати на розв'язок». Якщо черговий «кандидат» недопустимий, то метод переходить до іншого «кандидата» чи повертається до іншого часткового рішення або продовжує пошук іншими шляхами. Згідно цього методу, усі кроки пошуку оптимального рішення фіксуються, щоб у разі змін, які не підходять під обмеження розв'язку задачі, можна було повернутись до прийнятного «кандидата» [4]. Реалізації цього методу, як правило, швидші за повний перебір, адже в процесі їх виконання недопустимі «кандидати» відкидаються і не доповнюються до повного розв'язку.

Загалом, на сьогодні при програмуванні розв'язування задач комбінаторної оптимізації основна увага приділяється відсіканню недопустимих «кандидатів» і недостатньо аналізується, на нашу думку, обчислювальна складність алгоритму [5]. А даремно, адже саме показники обчислювальної складності «дають уявлення про час виконання алгоритмів» [6, с. 57], дозволяють зрозуміти, чому один алгоритм буде працювати значно швидше від іншого, привчають майбутніх програмістів звертати увагу не лише на правильність розв'язку поставленої задачі, а й на його ефективність, не лише на час виконання програми, а й на обсяги використання пам'яті. Саме аналізуючи обчислювальну складність ми визначатимемо у цій статті доцільність застосування запропонованих алгоритмів.

Загалом, ідея застосування множини значень, пропагована у цій статті, використовуються в програмування не перший рік. Наприклад, для сортування цілочисельних масивів з невеликим діапазоном значень елементів за зростанням доцільно не порівнювати елементи між собою, а всього лише підрахувати частоти окремих значень і повторити в результуючому масиві кожне значення від найменшого до найбільшого стільки разів, скільки воно зустрічалося у вхідному масиві [7, 8]. Зрозуміло, що це доцільно робити, коли діапазон значень співмірний з розміром масиву. Ми ж застосуємо аналіз множини значень для розв'язку задач комбінаторної оптимізації.

Зрозуміло, що тестувати розроблені програми для розв'язування задач комбінаторної оптимізації можна на локальних робочих станціях а ліквідовувати їх недоліки – за допомогою популярних багтрекерних систем [9]. Але при цьому слід враховувати особливості обчислювальних середовищ і можливості пристосування програм до відомих тестових випадків. Тому ми вважаємо доцільним тестування розроблених програм у віддалених обчислювальних середовищах. Таке тестування – це спосіб розвитку логічного мислення та навиків оптимізації при вирішення нетипових задач. Одним із сайтів, який надає доступ до завдань та віддаленого обчислювального середовища, є https://www.eolymp.com. Ця робота написана з використанням його ресурсів. Переваги eolymp наступні:

- значна кількість задач з різних розділів;
- групування задах за різними рівнями складності;
- можливість порівняння свої результатів з результатами інших користувачів;
- підтримка понад 15 мов програмування для подання роз'вязків;
- наявність навчальних матеріалів;
- можливості порівняння затрат часу на виконання та обсягів використання оперативної пам'яті для окремого тесту при застосуванні різних підходів до розв'язування кожної задачі.

Формулювання задачі для апробації методів комбінаторної оптимізації

Продемонструємо застосування різних методів розв'язання задач комбінаторної оптимізації на прикладі задачі «CD» (№ 1266) із сайту https://www.e-olymp.com. Умова цієї задачі формулюється так:

У Вас попереду тривала подорож на автомобілі. На жаль, у Вас в автомобілі є лише магнітофон, а краща музика записана на компакт-дисках. У Вас є чиста магнітофона стрічка з тривалістю звучання N хвилин. Вам потрібно вибрати пісні для запису на магнітофону стрічку таким чином, щоб не використовуване на ній місце було мінімальним.

Програма повинна знайти максимально можливу довжину запису треків на стрічку зі збереженням того ж порядку треків, що і на CD.

Вхідні дані програми містять декілька рядків. У кожному рядку спочатку задано число N, далі кількість треків S і тривалість звучання кожного треку. Програма має для кожного рядка вхідних даних виводити рядок "sum:" і далі максимально можливу тривалість запису.

Обмеження на вхідні дані:

- *кількість треків S на CD не перевищує 100;*
- *жоден з треків не звучить більше N хвилин;*
- *довжина кожного треку є цілим числом;*
- *N також ціле (0 ≤ N ≤ 200).*

Обмеження в часі для виконання програмою кожного тесту – 1 с, максимальний допустимий обсяг використання оперативної пам'яті – 122.81 MB.

Мета і завдання дослідження

Метою дослідження є опис та обгрунтування доцільності застосування методу поступового формування множини значень цільової функції для розв'язування задач комбінаторної оптимізації.

Основне **завдання роботи** – створення програм, які реалізують алгоритми різних методів розв'язання однієї з типових комбінаторних задач та порівняння їх ефективності для різних тестових випадків з метою встановлення доцільності застосування кожного з розглянутих методів.

В процесі побудови алгоритмів використані загальнонаукові методи аналізу та синтезу. Оцінку складності алгоритмів проведено методами теорії алгоритмів. Для розв'язання поставленої задачі комбінаторної оптимізації використано метод пошуку з поверненнями, метод врахування змін та запропонований нами метод поступового формування множини допустимих значень.

Методи, матеріали і результати дослідження

1. Порівняння обчислювальних складностей та результатів тестування реалізацій розв'язків поставленої задачі класичними методами комбінаторної оптимізації

1.1. Використання методу повного перебору

Щоб знайти максимально можливу довжину запису треків на стрічку, яка не перевищує N, потрібно бути готовим розглянути всі можливі варіанти їх входження, адже кращою може виявитися будь-яка комбінація треків. Достроково завершити розгляд наступних варіантів можна лише тоді, коли буде знайдено варіант, при якому довжина магнітофонної стрічки буде зайнята повністю (рівна N).

За умовою задачі, кількість треків рівна S. Кожен з них може бути лише в двох станах – врахований до суми поточного варіанту або не врахований. Отже, **кожен з варіантів входження треків можна записати як S-бітове двійкове число**, а всього варіантів буде 2^S. **Запишемо довжини всіх треків справа наліво**, аналогічно розміщенню бітів двійкового числа – від молодших до старших. Для перебору всіх можливих варіантів нам знадобляться усі S-бітові двійкові числа, крім 0, адже це значення мінімально можливої суми за замовчуванням. Для знаходження суми довжин треків чергового варіанту будемо переводити номер варіанту у двійкову систему числення і аналізувати окремі біти. Тому обчислювальна складність алгоритму становитиме $2^S \times S$. В чергову суму будемо додавати лише довжини треків, для яких відповідні біти в номері варіанту рівні 1. Для розв'язку задачі серед сум варіантів, не більших за N, будемо обирати максимальну.

Врахуємо також, що якщо додавання довжини чергового треку до суми поточного варіанту робить її більшою за довжину стрічки, то це порушує умову задачі (оскільки довжини треків невід'ємні), і тому програма може достроково переходити до наступного варіанту і формування нової суми.

Для прикладу простежимо механізм формування максимально допустимої суми треків для стрічки довжини 10 ($N = 10$) при чотирьох треках ($S = 4$) з відповідними довжинами 2, 4, 8, 4 (табл. 1). Бачимо, наприклад, що встановлення першого біта справа в двійковому записі номера варіанту призводить до входження в поточну суму довжини першого треку. У розглянутому випадку пошук максимальної суми завершується до перебору всіх варіантів входжень треків ($2^4 = 16$), оскільки знайдена максимально допустима сума рівна довжині стрічки. Якби цього не сталося, програма продовжила б роботу до перебору всіх варіантів. Максимальне заповнення (10) знайдено за п'ять варіантів перебору.

Таблиця 1 - Використання двійкового запису номерів варіантів для генерування комбінацій треків для N = 10, S = 4 та довжин треків {2, 4, 8, 4}

Номер варіанту в десятковій системі	Номер варіанту в двійковій системі	Враховані треки	Сума поточного варіанту входжень треків	Відома наразі максимально допустима сума
1	0001	2	2	2
2	0010	4	4	4
3	0011	4, 2	6	6
4	0100	8	8	8
5	0101	8, 2	10	10

Наведемо текст програми для розв'язання поставленої задачі методом повного перебору мовою програмування C#, оскільки на сьогодні вона є однією з основних мов програмування прикладних додатків:

```
while(true)
{string line = Console.ReadLine(); // Зчитуємо черговий рядок з умовами задачі
 if (String.IsNullOrEmpty(line)) return; // Більше задач немає
 string[] str = line.Trim().Split(new char[] {' '});
 if (str.Length < 2)
  return; // Задача без довжини стрічки чи кількості треків не розв'язується
 int n = Convert.ToInt32(str[0]); // Загальна довжина стрічки
 int s = Convert.ToInt32(str[1]); // Кількість треків
 long i, j, sum, maxSum = 0, remainder; // Remainder – the remainder of a division
 int[] track = new int[s];
 for (i = 0; i < s; i++)
  //Зчитуємо тривалість звучання кожного треку
  track[i] = Convert.ToInt32(str[i + 2]);
•/* Переберемо всі s-бітові числа, де i-ий біт буде вказувати, враховується i-ий трек в суму (1) чи ні (0). Це
дозволить отримати кожен з можливих варіантів вибору входжень їхніх довжин */
 //Підраховуємо кількість усіх можливих варіантів вибору
 BigInteger variantsNumber = (BigInteger)1<<s; // 1<<s – теж саме, що 2^ s
 //Переведення кожного номеру варіанту входжень треків у двійковий запис,
 //обчислення відповідної суми для пошуку максимальної серед допустимих
 for (j = 1; j < variantsNumber; j++)
  {sum = 0; // Щоразу обчислюється сума для чергового варіанту вибору треків
   remainder = j; //Щоб не впливати на лічильник варіантів
   for (i = 0; i < s; i++) // Варіант вибору треків інтерпретуємо як двійкове
   //s-бітове число, довжина якого відповідає кількості треків (s)
    {if (remainder % 2 == 1) // Якщо i-ий трек до суми входить
      {sum += track[i];
       if (sum > n) break; } // Перевищення загальної довжини стрічки недопустиме
➢  remainder /= 2; } //Переходимо до наступних треків
   if (sum <= n && sum > maxSum) // Якщо знайдена сума не перевищує
   //загальну довжину стрічки та більша за кращу з попередніх
    {maxSum = sum;
     if (maxSum == n) break; }} // Вже знайшли оптимальний варіант
•Console.WriteLine("sum:" + maxSum.ToString()); }
```

Результати тестування цієї програми на сайті eolymp.com наведені на рис. 1. Бачимо, що наведена програма правильно працює на дев'яти тестах, але вичерпує ліміт часу при проходженні останнього тесту.

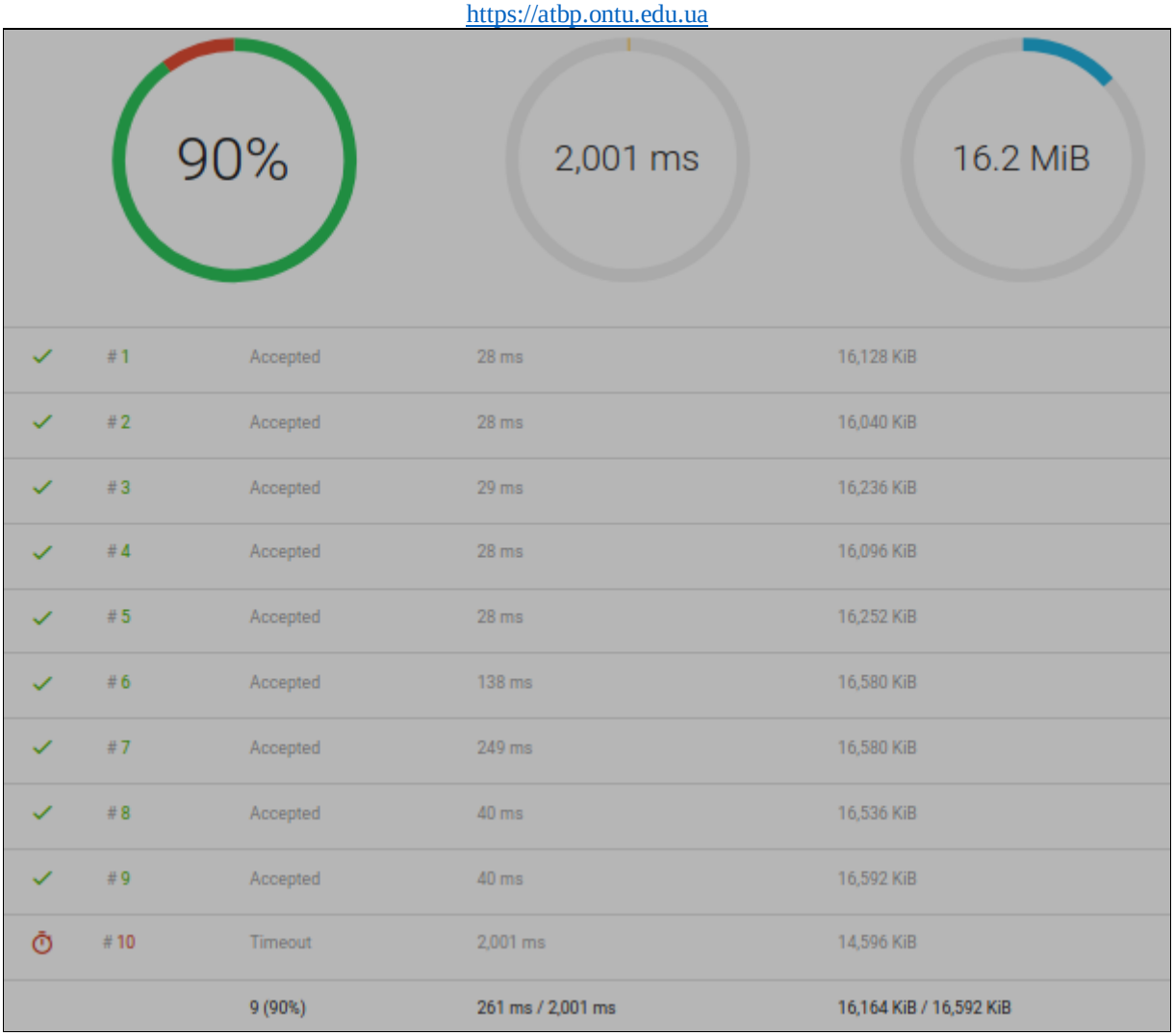

Рис. 1 - Результати тестування програми методу повного перебору

Fig. 1 - Results of testing the program that implements the complete search

1.2. Оптимізація алгоритму повного перебору

Для прискорення наведеної програми врахуємо, що цикл для переведення номеру варіанту у біти двійкового запису виконується S разів для кожного варіанту. Це призводить до зайвих операцій, так як не усі номери варіантів є s-значними двійковими числами, Взагалі кажучи, переведення числа з однієї системи числення в іншу завершується, коли частка від ділення на основу системи числення, в яку виконують переведення, стає рівною нулю. Для нашого алгоритму це означатиме, що для чергового варіанту немає більше треків зліва, які входять в поточну суму. Для реалізації цієї ідеї доповнимо код першого варіанту після рядка, позначеного маркером ➢, такою перевіркою:

```
if(remainder == 0) break; // Більше немає треків, які входять в суму
```

Ця перевірка суттєво впливає на швидкість виконання тестів з великим S при розгляді варіантів включення треків з невеликим номером (рис. 2). Бачимо що виконання останнього тесту все ще вичерпує ліміт часу. Обчислювальна складність алгоритму тепер становить $2^{S-1}\times S$, тобто зменшилася вдвічі, але реально час виконання програми суттєво не зменшився, оскільки переривання пошуку з поверненнями не впливають на збільшення проміжних сум. Якби при виконанні наведеної перевірки вдалося ще й оминути окремі варіанти розв’язків, то це було б класичною реалізацією пошуку з поверненнями.

Рис. 2 - Результати тестування програми методу повного перебору з урахуванням довжини номера варіанту в двійковій системі числення

Fig. 2 - Results of testing the program that implements the exhaustive search with taking into account a length of a variant number in the binary number representation

1.3. Застосування методу врахування змін

Реалізуємо метод урахування змін при обчисленні суми довжин треків чергового варіанту відносно суми попереднього варіанту. Поточна сума має зменшитися відносно попередньої на довжину тих треків, в двійковому представленні номера яких замість 1 став 0. І має збільшитися на довжину першого з треків, де замість 0 став 1. Порядок вибору треків, як і у попередній програмі, визначимо двійковими *S*-бітовими числами, але значення окремих бітів не будемо обчислювати щоразу, а збережемо в масиві *occurrence* з *S*+1 елемента. Це дасть змогу уникнути надлишкових ділень на два і розв'язувати задачу для будь-якого невід'ємного *S*. Перебір будемо завершувати, коли в додатковому елементі цього масиву з індексом *S* з'явиться 1, тобто будуть перебрані всі *S*-бітові двійкові числа. Для реалізації цього алгоритму замінимо в коді попередньої програми рядки між маркерами

- наступним кодом:

```
byte[] occurrence = new byte[s+1]; /* i-тий елемент масиву вказує, враховується (1) чи ні (0) i-тий трек в суму. Використання цього масиву замість двійкового запису номера варіанту дає змогу опрацьовувати довільну кількість треків. Генерація наступного варіанту вибору треків по масиву occurrence відповідає принципу збільшення числа на 1 в двійковій системі числення: додавання одиниці перетворює всі суміжні одиниці справа наліво в нулі, а перший справа нуль перетворює в одиницю. Відповідно, сума довжин треків відносно попередньої суми зменшиться на довжину треку там, де в масиві occurrence замість 1 став 0 і збільшиться на довжину треку там, де в масиві occurrence замість 0 став 1. Після створення масиву всі його елементи рівні 0, тобто перший варіантів вибору треків - той, при якому жоден з треків не враховується. Останній варіант вибору треків відповідає масиву occurrence, в якому всі елементи рівні 1, тобто всі треки враховуються. Завершення перебору варіантів вибору треків виконується при встановленні 1 в додатковій зліва позиції масиву (переповненні номера варіанту)*/
while (occurrence[s] == 0) // Поки додаткова позиція масиву не заповнена
```

```
{j = 0; //індекс треку, який змінюється в сумі
 while(occurrence[j] != 0)
  {occurrence[j] = 0; // Трек вже не враховується, тому загальна тривалість зменшується на його довжину
   sum -= track[j++]; }
 occurrence[j] = 1; // Дійшли до треку, який вже враховується,
 // тому загальна тривалість збільшується на довжину цього треку
 if (j < s) sum += track[j];
 if(sum <= n && sum > maxSum) // Знайшли кращий варіант вибору треків
  {maxSum = sum;
   if (maxSum == n) break; }}
```

Такі зміни програми прискорили її виконання для всіх тестів, але останній тест все ще не вкладається в час (рис. 3). Фактично, цей фрагмент програми в сумі треків чергового варіанту їх вибору лише враховує зміни відносно попереднього варіанту, а не перераховує цю суму повністю, але обчислювальна складність алгоритму залишається $2^S \times S$.

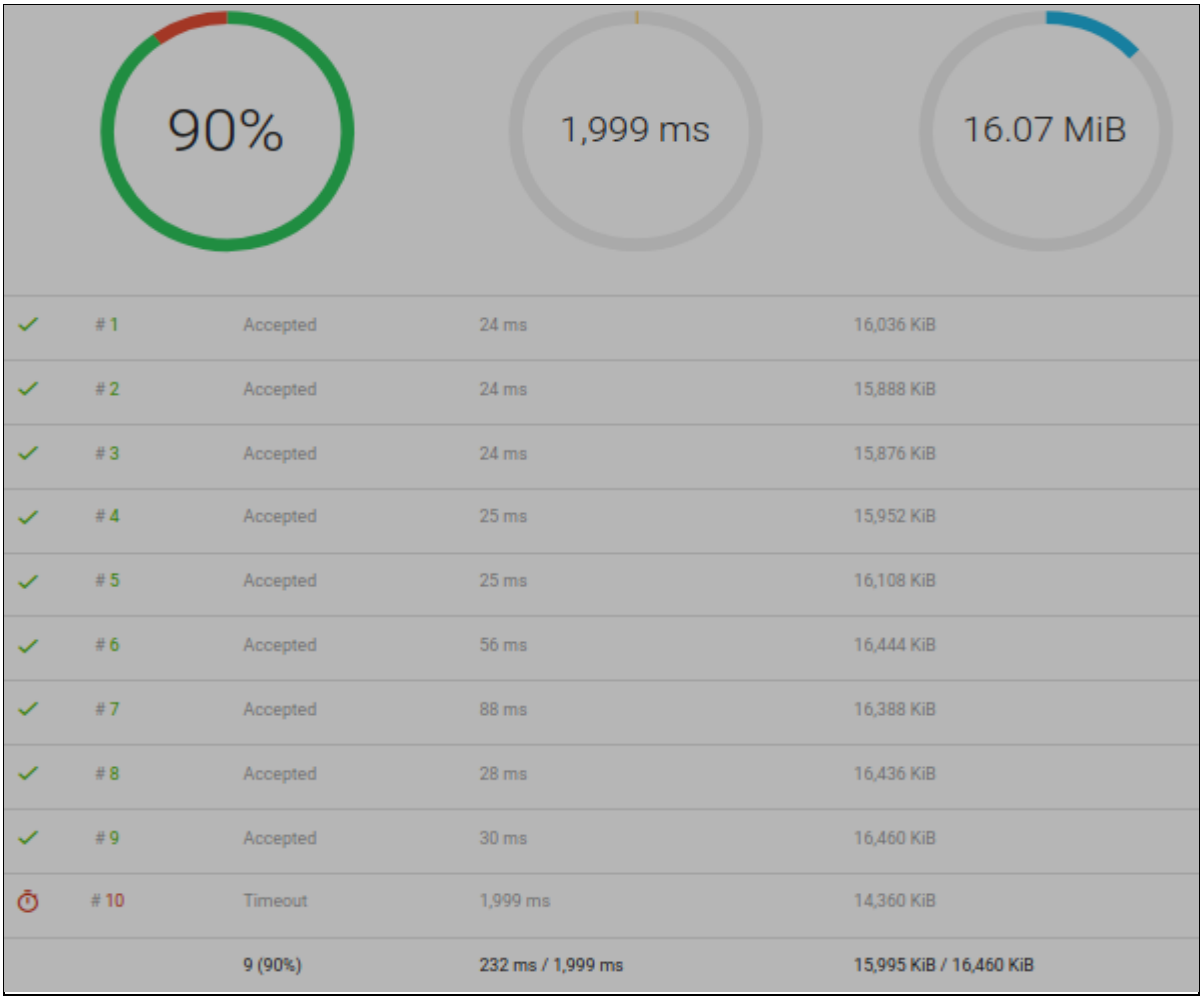

Рис. 3 - Результати тестування програми з реалізацією методу урахуванням змін

Fig. 3 - Results of testing the program that implements the taking into account changes method

1.4. Використання методу пошуку з поверненнями

Для реалізації методу пошуку з поверненнями використаємо основне обмеження розв'язку задачі – сума довжин всіх треків не повинна перевищувати N. Відповідно, для чергового варіанту вибору треків підраховуватимемо суму треків **зліва направо** і як тільки ця сума досягне N – пропустимо всі варіанти вибору з комбінацій треків, розміщених правіше, адже такі варіанти будуть недопустимими. Тобто ми пропускаємо недопустимі варіанти і відразу переходимо до наступних можливих "кандидатів" на розв'язок.

Для реалізації цього алгоритму цього методу, як і в попередньому підрозділі, замінимо в коді наведеної програми рядки між маркерами • таким фрагментом:

```
 while(occurrence[s] == 0) // Поки додаткова позиція не заповнена
  {// Переходимо до наступного номера варіанту вибору треків в двійковій системі
   j = 0;
```

```
while(occurrence[j] != 0) occurrence[j++] = 0;
occurrence[j] = 1;
// Підраховуємо суму зліва направо, доки вона допустима
sum=0; j=s-1;
while (sum<=n && j>=0)
 {if (occurrence[j]==1) sum+=track[j];
  j--; }
if(sum <= n && sum > maxSum)  // Знайшли кращий варіант вибору треків
 maxSum = sum;
if (sum>=n) // Досягнули обмеження суми треків - пропускаємо комбінації з треків, розміщених правіше
 for (i=0;i<=j;i++) occurrence[j]=1; }        .
```

Така реалізація методу пошуку з поверненнями зменшує час проходження успішно пройдених тестів в середньому на 2.5 % відносно тривалостей тестування, наведених на рис. 3, але останній тест всерівно не вкладається у відведений час. Цей алгоритм хоча й пропускає окремі недопустимі варіанти розв'язків, але суму треків щоразу підраховує спочатку і має ту саму обчислювальну складність $2^S \times S$.

2. Опис та аналіз результатів застосування методу поступового формування множини значень цільової функції

Задамося питанням: як змінюється сукупність можливих сум довжин треків після розгляду чергового треку? Іншими словами: як отримати множину можливих (допустимих після сумування) довжин треків E_i, знаючи допустимі довжини сум попередніх треків E_{i-1} та довжину чергового треку $trek_i$ (i вказує на те, що розглядаються треки від початкового до i-го включно)? Якщо черговий трек $trek_i$ не враховується, то множина допустимих сум довжин треків залишаться такою самою, тобто $E'_i=E_{i-1}$. Якщо ж черговий трек враховується то з'являться нові допустимі суми довжин, утворені збільшенням попередніх сум довжин на довжину чергового треку. Тобто, $E''_i=\{e''_{i,j} : e_{i-1,j}+trek_i\}$. Очевидно, що

$$E_i=E'_i \cup E''_i. \quad (1)$$

Якщо жоден трек не враховується, то сума довжин треків рівна 0. Якщо ж розглянути початковий трек, то допустимими є суми довжин 0 (початковий трек не враховується) та $trek_0$ (якщо початковий трек враховується). Тобто $E_0=\{0; trek_0\}$, а наступні множини допустимих сум треків обчислюються згідно (1).

Наприклад, для згаданої вище послідовності довжин треків 2, 4, 8, 4 послідовно отримаємо такі множини допустимих сум:

$E_0=\{0; 2\}$ (нулевий трек враховується або не враховується).

$E'_1=\{0; 2\}$ (перший трек не враховується), $E''_1=\{4; 6\}$ (перший трек враховується), і, згідно (1) $E_1=E'_1 \cup E''_1=\{0; 2; 4; 6\}$ (перший трек враховується або не враховується).

$E'_2=\{0; 2; 4; 6\}$ (другий трек не враховується), $E''_2=\{8; 10; 12; 14\}$ (другий трек враховується), $E_2=E'_2 \cup E''_2=\{0; 2; 4; 6; 8; 10; 12; 14\}$.

$E'_3=\{0; 2; 4; 6; 8; 10; 12; 14\}$. (третій трек не враховується), $E''_3=\{4; 6; 8; 10; 12; 14; 16; 18\}$ (третій трек враховується), $E_3=E'_3 \cup E''_3=\{0; 2; 4; 6; 8; 10; 12; 14; 16; 18\}$.

Оскільки допустимі суми щоразу зростають, то обчислювати ті з них, які більші N не має сенсу (в наведеному прикладі допустимі суми понад 10 можна було б ігнорувати). Серед залишених допустимих сум потрібно вибрати максимальну після розгляду всіх треків, або рівну N, якщо така сума трапилася раніше (в розглянутому випадку допустима сума 10 з'являється вже після розгляду третього треку). На прикладі формування третьої множини допустимих сум бачимо також, що кількість елементів множини не обов'язково щоразу зростає вдвічі, адже можливе дублювання елементів в E' та E''.

Реалізуємо тепер програмно варіант поступового формування множин допустимих сум. Множина допустимих сум E_i після розгляду чергового елемента залежить лише від множини E_{i-1}. Тому не потрібно зберігати всі множини – достатньо ітеративно, маючи попередню множину, формувати наступну множину, а для наступної ітерації присвоїти попередній множині наступну, а наступній – попередню. Цікаво, що при цьому наступну множину можна не очищати – допустимі значення будь-якої множини завжди будуть допустимими для наступних допустимих множин. Для зберігання попередньої та наступної множин допустимих сум на кожній ітерації використаємо масиви, де значення 1 в кожному елементі буде вказувати про входження його в множину допустимих значень, а 0 – на відсутність його в множині. З метою швидшого переприсвоєння масивів будемо не переприсвоювати їх елементи, а переприсвоювати **посилання** на масиви. Враховуючи ці міркування, реалізація методу поступового формування множин допустимих значень для досліджуваної задачі може бути такою:

```
byte[] nextSum = new byte[n+1], prevSum = new byte[n+1]; /* Попередні і наступні елементи множин допустимих
сум, більші N, не розглядаються */
byte[] interim;
nextSum[0] = 1; // Перед розглядом треків допустима лише сума 0
for(i = 0; i < s; i++) // Цикл по розглядуваних треках
 {/* Кожен трек може враховуватися, а може і не враховуватися в загальну суму. Відповідно, можливими
сумами довжин треків після розгляду чергового треку будуть всі попередні суми (коли поточний трек не
```

```
враховується) та попередні суми, збільшені на довжину поточного треку */
  interim = prevSum;
  /* Проміжна вказівка на масив потрібна, щоб підчас перестановки prevSum не був втрачений */
  prevSum = nextSum;
  nextSum = interim; /* При переході до наступного треку попередні можливі суми беруться з наявних наступних,
    а наступні можливі суми розраховуються. Для швидшої перестановки масивів не переприсвоюються їх
    значення, а міняються місцями посилання на них*/
  for(j = 0; j <= n; j++) /* Цикл по можливих сумах довжин запису на стрічку. Індекс масиву і є значення суми. */
   {if(prevSum[j] == 1) // Знайшли допустиму суму відносно попередніх треків
    {nextSum[j] = 1; /* Черговий трек не враховується – попередня можлива сума не змінюється */
     /* Якщо врахування чергового треку разом з можливою попередньою сумою поміститься на стрічці */
     if(j+track[i] <= n)
      nextSum[j + track[i]] = 1; }}} /* Додавання до попередніх можливих сум довжини чергового треку дозволяє
       визначати нові можливі суми, не додаючи усі довжини треків заново. */
 i = n;
 while(nextSum[i] == 0) i--; /* Знаходимо максимальну з допустимих сум треків. Пошук із кінця, тому що чим
    більший індекс, тим більша сума */
```

Результати тестування цієї програми наведено на figure 4. Бачимо, що усі тести вклалися у відведений час і при цьому використано найменше оперативної пам'яті з представлених варіантів. Прискорення розв'язання завдання у цьому випадку – не випадковість. Адже обчислювальна складність алгоритму тепер становить $S \times N$, що значно менше $2^{S-1} \times S$ з врахуванням обмежень нашої задачі ($100 \times 200 << 2^{99} * 100$). На завершення зазначимо, що метод поступового формування множин допустимих значень знаходить лише розв'язок (екстремум) цієї задачі комбінаторної оптимізації, але не визначає треки, при яких цей розв'язок отримується. Для знаходження цих треків потрібно використати інші (можливо, й розглянуті вище чи зворотного ходу методу динамічного програмування) методи, але навіть знаходження екстремуму прискорює розв'язування оптимізаційних задач в класичному випадку.

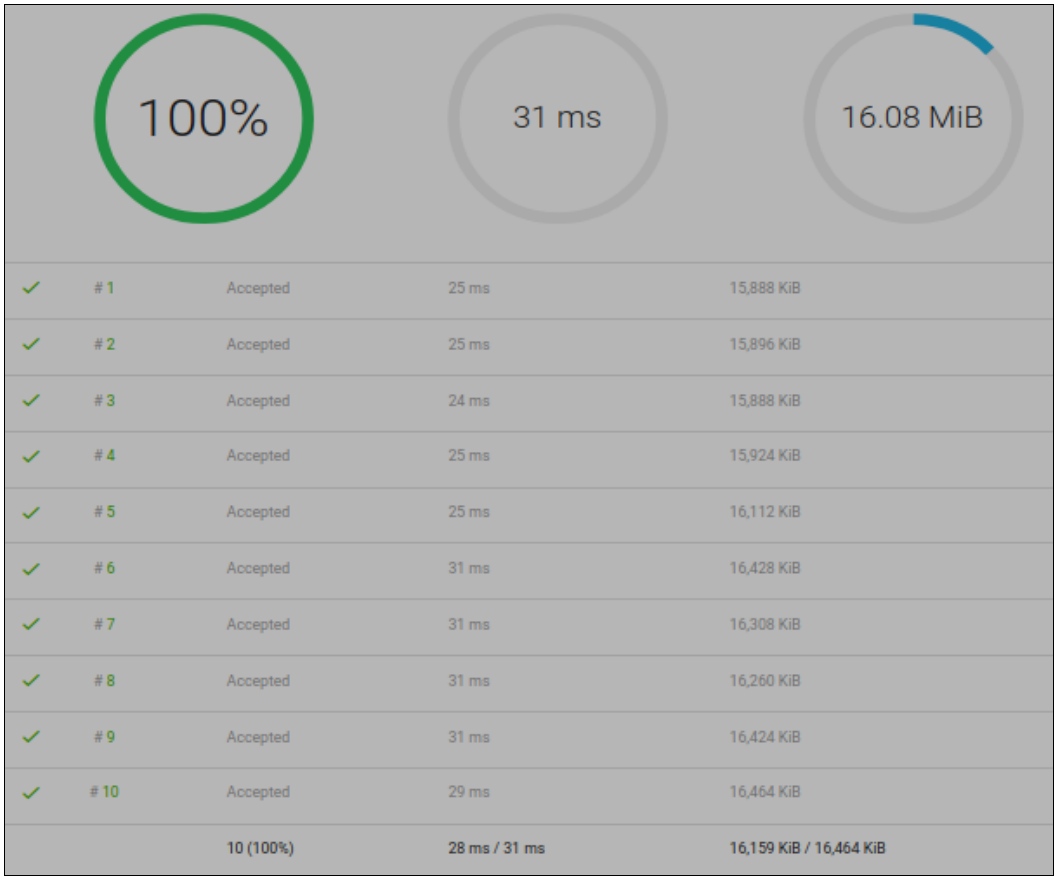

Рис. 4 - Результати тестування програми з реалізацією методу поступового формування множин допустимих значень

Fig. 4. Results of testing the program that implements the method of gradual formation of sets of admissible value

Висновки

1. Для визначення найефективнішого способу розв'язування задачі комбінаторної оптимізації недостатньо порівнювати час виконання на відомих тестових наборах, а потрібно намагатися попередньо проаналізувати їх обчислювальну складність.

2. Метод поступового формування множини допустимих значень цільової функції є дієвою альтернативою методам пошуку з поверненнями та врахування змін при розв'язуванні задач комбінаторної оптимізації, якщо область значень дискретна, а хід розв'язання подібний до методу динамічного програмування.

3. Для моделювання множин елементів в програмуванні доцільно використовувати логічні чи байтові масиви. Тоді для переприсвоєння множин достатньо переприсвоїти вказівки на масиви, а не переприсвоювати окремі елементи.

4. Для прискорення розв'язування задач комбінаторної оптимізації недостатньо оминати деякі варіанти повного перебору, а потрібно мінімізувати ще й час обчислення цільової функції для кожного варіанту, враховуючи обмеження задачі.

Список використаних джерел

[1]. Combinatorial optimization. URL: https://en.wikipedia.org/wiki/Combinatorial_optimization. (Дата звертання: 25.01.2023).
[2]. Множинний тип. URL: https://studfile.net/preview/7079855/page:12. (Дата звертання: 25.01.2023).
[3]. Пошук з вертанням. URL: https://uk.wikipedia.org/wiki/Пошук_з_вертанням. (Дата звертання: 25.01.2023).
[4]. Алгоритми з поверненням. Розв'язок задачі про рух коня URL: https://studfile.net/preview/7079855/page:20. (Дата звертання: 25.01.2023).
[5]. Обчислювальна складність - Вікіпедія. URL: https://uk.wikipedia.org/wiki/Обчислювальна складність. (Дата звернення: 25.01.2023).
[6]. Оцінка складності алгоритмів, або Що таке O(log n). URL: https://echo.lviv.ua/dev/53. (Дата звернення: 25.01.2023).
[7]. Knuth D. E. The Art of Computer Programming.Vol. 3. Sorting and Searching, 2-ed. Massachusetts: Addison Wesley Longman, 1997. p. 80.
[8]. Shportko A. V., Shportko L. V. Acceleration of large integer arrays sorting using ranges of values and frequencies of elements. *Information Extraction and Processing*. 2019, 47(123), 73-79. DOI: https://doi.org/10.15407/vidbir2019.47.073.
[9]. Шпортько О. В., Гаврилюк В. І. Сучасні багтрекерні системи відслідковування помилок: переваги та недоліки. *Матеріали Міжнародної науково-практичної конференції «Сучасні тенденції в математичному моделюванні і його програмному забезпеченні» (14.05.2020)*. Рівне: РВЦ МЕГУ ім. акад. С. Дем'янчука, 2020. С. 54-56.

References

[1]. Combinatorial optimization. URL: https://en.wikipedia.org/wiki/Combinatorial optimization. (Accessed: 25.01.2023).
[2]. Mnozhynnyi typ [in Ukranian]. URL: https://studfile.net/preview/7079855/page:12. (Accessed: 25.01.2023).
[3]. Poshuk z vertanniam [Search with returns] [in Ukranian]. https://uk.wikipedia.org/wiki/%D0%9F%D0%BE%D1%88%D1%83%D0%BA_%D0%B7_%D0%B2%D0%B5%D1%80%D1%82%D0%B0%D0%BD%D0%BD%D1%8F%D0%BC. (Accessed: 25.01.2023).
[4]. Alhorytmy z povernenniam. Rozviazok zadachi pro rukh konia [in Ukranian]. URL: https://studfile.net/preview/7079855/page:20. (Accessed: 25.01.2023).
[5]. Obchysliuvalna skladnist - Wikipedia. URL: https://uk.wikipedia.org/wiki/%D0%9E%D0%B1%D1%87%D0%B8%D1%81%D0%BB%D1%8E%D0%B2%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0_%D1%81%D0%BA%D0%BB%D0%B0%D0%B4%D0%BD%D1%96%D1%81%D1%82%D1%8C. [in Ukranian]. (Accessed: 25.01.2023).
[6]. Otsinka skladnosti alhorytmiv, abo shcho take O(log n) [in Ukranian]. URL: https://echo.lviv.ua/dev/53. (Accessed: 25.01.2023).
[7]. Knuth D. E. The Art of Computer Programming.Vol. 3. Sorting and Searching, 2-ed. Massachusetts: Addison Wesley Longman, 1997. p. 80.
[8]. Shportko A. V., Shportko L. V. Acceleration of large integer arrays sorting using ranges of values and frequencies of elements. *Information Extraction and Processing*. 2019, 47(123), 73-79. DOI: https://doi.org/10.15407/vidbir2019.47.073.
[9]. Shportko A. V., Havrilyuk V. I. Suchasni bahtrekerni systemy vidslidkovuvannia pomylok: perevahy ta nedoliky. *Materialy Mizhnarodnoi naukovo-praktychnoi konferentsii «Suchasni tendentsii v matematychnomu modeliuvanni i yoho prohramnomu zabezpechenni» (14.05.2020)*. Rivne: RPC MEGU im. akad. S. Demianchuka, 2020. C. 54-56.