ДНІПРОВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІМЕНІ ОЛЕСЯ ГОНЧАРА

Факультет прикладної математики

Кафедра обчислювальної математики та математичної кібернетики

**КУРСОВА РОБОТА**

на тему **«Алгоритмічний підхід для оцінювання значень цільової функції у задачах упорядкування»**

Перший (бакалаврський) рівень вищої освіти

Спеціальність 124 Системний аналіз

Освітня програма «Системний аналіз»

Виконавець

студентка групи ПС-18-2

Василькович Поліна Олександрівна

Керівник

канд. фіз.-мат. наук,

зав. каф. ОМ та МК

\_\_\_\_\_\_\_\_\_В.А.Турчина

Кількість балів

Оцінка за національною шкалою \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Члени комісії:

\_\_\_\_\_\_\_\_\_\_\_\_ В.А. Турчина

\_\_\_\_\_\_\_\_\_\_\_\_ Л.Л. Гарт

\_\_\_\_\_\_\_\_\_\_\_\_ А.Є. Шевельова

2021

РЕФЕРАТ

Курсова робота: 30 с., 17 рис., 0 табл., 7 джерел, 1 додаток.

*Об'єкт дослідження:* Математичні моделі задач розподілу з технологічними обмеженнями.

*Мета роботи:* розробка алгоритму розв’язання задач розподілу для спеціальних класів графів, отримання оцінок для значень цільових функцій.

*Одержані висновки та їх новизна:* Для спеціального виду графів запропоновано алгоритм розв’язання однієї з задач, який є альтернативою до відомого та дозволяє подати нову оцінку довжини упорядкування. Також, запропонована схема оцінювання знизу ресурсів, що можуть бути задіяні.

*Результати досліджень можуть бути застосовані* при викладанні дисциплін, пов’язаних з розпаралеленням обчислень, при розв’язанні проблем розподілення працівників на місцях на підприємствах.

*Перелік ключових слів:* ПРИКЛАДНІ ЗАДАЧІ РОЗПОДІЛУ, ТЕХНОЛОГІЧНІ ОБМЕЖЕННЯ, ОРІЄНТОВАНІ ГРАФИ, АЛГОРИТМИ ПОБУДОВИ УПОРЯДКУВАНЬ, АЛГОРИТМ РОЗПОДІЛУ З ОБМЕЖЕННЯМИ НА РЕСУРСИ, ОЦІНКИ ЗНАЧЕНЬ ЦІЛЬОВИХ ФУНКЦІЙ.

ЗМІСТ

[ВСТУП 4](#_Toc73826920)

[1 ПРИКЛАДНІ ЗАДАЧІ РОЗПОДІЛУ З ТЕХНОЛОГІЧНИМИ ОБМЕЖЕННЯМИ 5](#_Toc73826921)

[2 МОДЕЛЮВАННЯ ТЕХНОЛОГІЧНИХ ОБМЕЖЕНЬ 7](#_Toc73826922)

[3 МАТЕМАТИЧНІ МОДЕЛІ ЗАДАЧ РОЗПОДІЛУ 10](#_Toc73826923)

[4 ОГЛЯД МЕТОДІВ РОЗВ’ЯЗАННЯ ЗАДАЧ РОЗПОДІЛУ ТА ЇХ СКЛАДНІСТЬ 11](#_Toc73826924)

[5 АЛГОРИТМ РОЗПОДІЛУ З ОБМЕЖЕННЯМИ НА РЕСУРСИ 12](#_Toc73826925)

[6 СХЕМА ПОБУДОВИ ОЦІНКИ ДЛЯ ЧАСУ ЗАВЕРШЕННЯ ВИКОНАННЯ ЗАВДАНЬ 15](#_Toc73826926)

[7 СХЕМА ПОБУДОВИ ОЦІНКИ ДЛЯ РЕСУРСІВ В ЗАДАЧІ РОЗПОДІЛУ 18](#_Toc73826927)

[8 ПРОГРАМНИЙ ПРОДУКТ 21](#_Toc73826928)

[8.1 Опис модулів програми 21](#_Toc73826929)

[8.2 Інструкція користувача 24](#_Toc73826930)

[8.3 Тестові приклади 27](#_Toc73826931)

[ВИСНОВКИ 29](#_Toc73826932)

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 30](#_Toc73826933)

[ДОДАТОК А Код програми 31](#_Toc73826934)

ВСТУП

Оптимальна організація виробничих процесів на підприємствах сприяє організації їх конкурентноспроможності. Шляхи покращення оптимальності роботи підприємств можна удосконалювати на основі наукового підходу, тобто з використанням відповідних матиматичних моделей, їх аналізу та вибору або розробки відповідних ефективних методів. Зокрема це стосується ефективного використання ресурсів і зменшення часових затрат.

У даній роботі розглядаються моделі виробничих процесів, в яких наявні технологічні обмеження на порядок виконання завдань. Такі процеси зручно моделювати орієнтованими ациклічними графами. Графи – це дуже сучасний та актуальний інструмент для розв’язання складних математичних задач. Вони дозволяють встановити наглядні зв’язки між різними частинами складних процесів.

Моделі, що розглядаються у роботі відповідають різним типам підприємств, оскільки технологічні процесси можуть бути як лінійними, так і розподілятись на паралельні задачі. Деякі з них розглянуто в роботі, та запропоновані оцінки знизу на час та ресурси. Ці оцінки можна використовувати на тих підприємствах, де є відповідні технології виробництва.

Метою роботи є розгляд математичних моделей заснованих на реальних структурах роботи підприємства, постановка задач розподілу працівників на місця, з обмеженням на час та ресурси, розробка алгоритмів їх розв’язання та оцінок для цільових функцій, а також розробка програмного продукту, що реалізує запропонований алгоритм [1].

1 ПРИКЛАДНІ ЗАДАЧІ РОЗПОДІЛУ З ТЕХНОЛОГІЧНИМИ ОБМЕЖЕННЯМИ

У сучасному світі більшість складних продуктів не може бути створена однією людиною та потребує багатокрокового процессу обробки на одному, а іноді на декількох підприємствах, оскільки для їх виготовлення необхідні знання у різних ґалузях науки. Одним із поширених технологічних процесів є процес, що відповідає роботі конвеєра з розподіленням завдань на певні етапи, де кожен виконавець повинен виконувати свою частину роботи.

Будемо розглядати задачі призначення працівників на робочі місця наступним чином: на підприємстві є скінчена кількість завдань та працівники, що можуть їх виконувати. Треба розподілити працівників на завдання таким чином, щоб час виконання був найменшим і не порушувались технологічні обмеження.

1. Спочатку розглянемо задачу з лінійним послідовним виконанням етапів. Такими етапами можуть бути: змішування інградієнтів, підготовка тіста до випікання, формування виробів, випікання та продаж. На потужних підприємствах цей процес можна організувати з паралельним виконанням окремих операцій.
2. Тепер розглянемо більш складну задачу, наприклад, технологічний процес пов’язаний з виготовленням виробу на конвеєрі.

Нехай це підприємство, де збирають автомобілі. Спочатку потрібно зібрати внутрішні частини авто: двигун, коробка передач, осі, бензобак, внутрішні сполучення. Потім потрібно виготовити корпус автомобіля, покрасити його та покрити захисним шаром. Далі вмістити механічні частини всередину машини, підключити електроніку та зовнішні деталі: фари, колеса, захисні решітки, тощо. Після цього слідує складний процесс налагодження синхронності та врівноваження роботи деталей.

Цей процес виготовлення дуже складний та потребує багато працівників та епатів виконання. Також зауважимо, що деякі етапи можуть виконуватися одночасно. Наприклад, виготовлення двигуна не залежить від покриття корпусу. Але у результаті всі деталі збираються у один цілий продукт.

1. Наступною задачею буде задача отримання із цілого – скінченої множини складових.

Для прикладу візьмемо лісопілку на якій працюють спеціалісти однієї категорії. На вході маємо ціле дерево, а на виході різні види мітеріалу для столярних виробів: шпали, лафети, бруси і дошки. Процес розпилу може бути незалежним після початкового розпилу колоди, тому працююча бригада робочих може виконувати свої задачі одночасно.

1. На даному етапі розглянемо задачу, де з початкового однакового матеріалу отримуємо той самий кінцевий результат, але процес виготовлення розходиться на декілька паралельних епапів.

Прикладом такої задачі може бути фабрика з виготовлення кіндер сюпрпризів. Спочатку ми маємо пластик з якого буде виготовлена іграшка та шоколад для яйця. Але оскільки у кожному сюрпризі іграшка є унікальною, пластик розподіляється на різні конвеєри, що виготовляють різні іграшки. Далі іграшка потрапляє у яйце і на виході ми отримуємо однаковий продукт, що зовні не відрізняється та продається за ту саму ціну.

1. Останнім буде найскладніший технологічний процесс, при якому ми маємо багато початкових етапів та декілька кінцевих продуктів.

Маємо фабрику меблів. На ній виготовляється багато різних продуктів: столи, стільці, дивани, шафи, крісла, тумби та багато інших. До кожного виробу потрібен свій набір початкових матеріалів: різні види дерева, тканини, фурнітура. Для деяких заготівля може бути спільною, а може і не перетинатися. Подальший процес виготовлення меблів, також, може сполучатися на деяких етапах та розходитися далі. У такому випадку розподілити робочих буде найскладніше, оскільки етапи роботи можуть змінюватись в залежності від виробу, виду матеріалів, кількості необхідних меблів, сроків заказу, тощо.

2 МОДЕЛЮВАННЯ ТЕХНОЛОГІЧНИХ ОБМЕЖЕНЬ

Chart, bubble chart

Description automatically generatedЯкщо розглядати поставлені вище задачі по розподілу працівників на підприємстві, то можна побачити, що до кожної з них можна поставити у відповідність деякий конкретний вид графу.

Першій задачі буде відповідати граф, який має вигляд ланцюга, схематичний вигляд можна побачити на (рисунок 2.1)

Рисунок 2.1 – Граф, який має вид ланцюга (орієнтація зліва направо)

A picture containing device

Description automatically generatedДруга задача, що відповідає роботі конвеера, моделюється орграфом, який є деревом, що має одну вершину, з якої не виходять дуги (вона відповідає готовому виробу, зібраному на конвеєрі). Цю вершину називають коренем дерева.

Рисунок 2.2 – Вхідне дерево (орієнтація зверху вниз)

A picture containing person, outdoor, vector graphics

Description automatically generatedЯкщо всі дуги кореневого дерева орієнтовані на шлях до кореня, то таке дерево називають деревом, що сходиться до однієї вершини або **вхідним деревом** (рисунок 2.2).

У третьому випадку маємо задачу декомпозиції, тому йому буде відповідати вихідне дерево.

Рисунок 2.3 – Вихідне дерево (орієнтація зліва направо)

**Вихідним деревом**, називають кореневе дерево у якому всі дуги напрямлені шляху від кореня дерева (рисунок 2.3).

Diagram

Description automatically generatedЧетвертій задачі буде відповідати паралельно-послідовний граф, оскільки у виробництві ми маємо розподілення з одного початкового етапу на декілька паралельних задач і сходження їх у кінці до одного останьоого етапу.

Рисунок 2.4 – Паралельно-послідовний граф (орієнтація зліва направо)

**Термінальна вершина –** вершина, що не лежить між двома іншими вершинами.

**Паралельно-послідовним графом** називаеться граф з двома термінальними вершинами (джерело та стік), що був утворений двома простими операціями: паралельне та послідовне з’єднання (рисунок 2.4).

A picture containing person, outdoor, device, vector graphics

Description automatically generatedОстанньому, найскладнішому випадку буде відповідати довільний ациклічний орієнтований граф (рисунок 2.5) [2].

Рисунок 2.5 – Довільний ациклічний орієнтований граф дерево (орієнтація зверху вниз)

3 МАТЕМАТИЧНІ МОДЕЛІ ЗАДАЧ РОЗПОДІЛУ

При розв’язанні задач, сформульованих у пункті 1, в класичній постановці виходять з припущення, що будь-яке завдання може виконувати будь-який виконавець. При цьому вважається, що час виконання кожного завдання однаковий. Оскільки, технологічні обмеження на порядок виконання завдань зручно представляти орієнтованими ациклічними графами, як указано в пункті 2, то абстрагуючись від предметної області можна говорити про дві оптимізаційні задачі на графах:

Задача 1.

По заданому графу G = {V, U} і заданій величині m розподілити елементи множини V на мінімальну кількість місць, так щоб на кожному місці стояло на більше m вершин і їх порядок в даному розподілі не прорушував технологічні обмеження.

Задача 2.

По заданому графу G = {V, U} і заданій кількості місць *l* розподілити елементи множини V на ці місця, так щоб максимальна кількість елементів на цих місцях була мінімальною і не порушувалися технологічні обмеження [3].

Для побудови упорякування необхідно і достатньо, щоб орієнтований граф G був ациклічним.

Розподіли, які являються розв’язками задач 1 і 2 називають паралельними упорядкуваннями.

Паралельне упорядкування будемо називати оптимальним, якщо при ширині, що не перевищує заданої ширини, має мінімальну довжину, що відповідає першій задачі або при довжині, що не перевищує заданої довжини, має мінімальну ширину, що відповідає дугій задачі [4].

4 ОГЛЯД МЕТОДІВ РОЗВ’ЯЗАННЯ ЗАДАЧ РОЗПОДІЛУ ТА ЇХ СКЛАДНІСТЬ

Задачі 1 і 2 це задачі комбінаторної оптимізації і вони відносяться до класу NP – повних [5].

Тому для знаходження точних розв’язків цих задач не існує алгоритмів поліноміальної складності. В загальному випадку точних розв’язок цих задач можна знаходити алгоритмами, заснованими на схемах направленого перебору. Одним із них являється метод гілок та меж, який має експоненційну складність. Тому для практичних задач, які мають велику розмірність, його застосовувати проблематично. Але для задач невеликої розмірності його доцільно використовувати для того, щоб порівнювати точність результатів, отриманих за наближеними алгоритмами.

Ще одним точним методом розв’язання задач в загальному випадку є метод, що заснований на сітковому підході. Він базується на узагальненні відомої задачі про максимальний потік, для якої розроблений точний алгоритм експоненційної складності. Подальший розвиток теорії відбувається в напрямі пошуку випадків, для яких існують точні алгоритми поліноміальної складності. Доведено, що такі алгоритми існують для графів:

* що являються вхідним лісом;
* для довільних графів без транзитавних дуг та m = 2;
* для довільних графів, коли кількість виконавців не перевищує двох;
* для деяких випадків паралельно-послідовних графів.

Відомо, що в класичній теорії оптимізації можуть розглядатися дві задачі: або знайти оптимальне значення цільової функції, не цікавлячись самим розв’язком, або знайти і оптимальний розв’язок, і оптимальне значення цільової функції.

Для задач, що вивчаються в даній роботі, розробку алгоритмів можна було б удосконалити, якби вдалося знайти значення цільових функцій в задачах 1 і 2 при невідомих оптимальних розв’язках. Але в загальному випадку поки що вдається отримати лише досить тривіальні оцінки знизу для цих функцій [6].

##### 5 АЛГОРИТМ РОЗПОДІЛУ З ОБМЕЖЕННЯМИ НА РЕСУРСИ

Для розв’язання перошої задачі сформульованої у попередньому розділі для побудови упорядкування у графі типу вхідне дерево, запропоновано наступний алгоритм:

1. Нехай задано вхідне дерево T. Помічаємо вершини дерева.
   1. *i*=1.
   2. Шукаємо множину вершин, що не мають дуг, що в них входять, і приписуємо їм мітку *і*.
   3. Якщо всі вершини дерева мають мітку переходимо на 2 етап.
   4. Видаляємо помічені вершини разом з вихідними дугами із Т.
   5. *і=і+*1 і переходимо до 1.2.
2. Побудова впорядкування.
   1. *i* = 1.
   2. Якщо мітка корневої вершини дорівнює *і* переходимо до п.3.
   3. Обираємо вершини з міткою *і* та рахуємо їх кількість. Нехай вона дорівнює *r*. Якщо r ≤ m, то переходимо на крок 2.7.
   4. Якщо r > m, то обираємо (r - m) вершин у порядку спадання міток вершин у які входять дуги з вершин з міткою *i* записуємо їх у множину U.
   5. Для множини U та номера *i* виконуємо крок 2.6.
   6. Маємо деяку вхідну множину Z та мітку k.

Вершинам з множини Z, у яких дуга, що з них виходить, входить у вершину з міткою >k+1, збільшуємо мітки на 1, видиляємо їх з Z.

Якщо є інші вершини у яких дуга, що з них виходить, входить у вершину з міткою k+1, записуємо вершини в які входять дуги у множину Y. Якщо у множину Y входить коренева вершина, збільшуємо її мітку на 1 та повторюємо крок 2.6 для множини Z та номера k.

У іншому випадку повторюмо крок 2.6 для множин Z та Y і номерів k та k+1 відповідно.

* 1. *i*=*i*+1, переходимо до 2.2.

1. Ставимо вершини дерева на місця яким відповідають їх мітки.

Приклад

A picture containing text, device, scale

Description automatically generatedМаємо граф (рисунок 5.1). m = 4.

Рисунок 5.1 – Приклад вхідного дерева з пронумерованими вершинами

Розставимо мітки за першим етапом (рисунок 5.2).

Переходимо до другого етапу. Кількість вершин з міткою 1 дорівнює 9. Отже, обераємо 9-4=5 вершин і збільшуємо в них мітки.

Обчислюємо мітки для наступних вершин.

Отримуємо граф з новими мітками.

Тепер ставимо вершини на місця, що відповідають їх міткам (рисунок 5.3):

1: 1, 2, 3, 6

2: 4, 5, 7, 8

3: 9, 10, 11, 12

4: 13, 14, 15, 16

5: 17, 18, 19, 20

Diagram

Description automatically generated with medium confidence6: 21

A picture containing text, device, scale

Description automatically generatedРисунок 5.2 – Граф з мітками

Рисунок 5.3 – Граф з розставленими вершинами на місця

6 СХЕМА ПОБУДОВИ ОЦІНКИ ДЛЯ ЧАСУ ЗАВЕРШЕННЯ ВИКОНАННЯ ЗАВДАНЬ

Постановка завдання:

Заданий оріентований граф G={V, U}; |V| = n та число m.

Треба розмістити вершини графа на мінімальне число місць так, щоб:

1. на кожному місці стояло не більше m вершин;
2. якщо з вершини з номером *i* йде дуга у вершину з номером *j*, то *i* знаходиться раніше *j*.

Позначимо кількість місць, на які можна розмістити вершини через *l*. Зрозуміло, що оптимальним розв’язком даної задачі буде таке розміщення, при якому *l* приймає мінімальне значення. Очевидно, що незалежно від структури графу величина *l* буде не меншою ніж ]n/m[. Але аналіз структури графів дозволяє покращити вказану оцінку. Зупинемось на цьому більш детально.

Нехай *l0* – довжина критичного шляху в графі (де під довжиною будемо розуміти кількість вершин на критичному шляху).

Зробимо помітку вершин за наступним алгоритмом:

* k=1.
* Шукаємо множину вершин, що не мають дуг, що з них виходять. Приписуємо їм мітку k.
* Якщо всі вершини графа помічені алгоритм завершує роботу.
* Видаляємо помічені вершини та дуги, що в них входять, з графу G.
* k=k+1, G:=G переходмо до п.2.

Будемо позначати:

[l0] – множина вершин з міткою 1.

[l0-1] - множина вершин з міткою 2.

...

[1] – множину вершин з міткою *l0*.

Оцінка *l* для довільних орієнтованих графів[7]:

(1)

Оцінка (1) для кореневих лісів досяжна [7], тобто маємо:

(2)

Будемо розставляти вершини на місця за алгоритмом з попереднього розділу та позначати

S[1] – множина вершин, що стоїть на першому місці.

S[2] - множина вершин, що стоїть на другому місці.

...

Нехай ri = m - |S[i]|.

Оцінка *l* для вхідних дерев:

Приклад:

*m* = 3, *lmax* = 4, *n* = 9 (рисунок 6.1).

S[1] = 3

S[2] = 3

S[3] = 1

S[4] = 1

S[5] = 1

r1 = 0

r2 = 0

r3 = 2

r4 = 2

r5 = 2

A picture containing text, device, scale, envelope

Description automatically generated

Рисунок 6.1 – Зображення графу з номерами вершин

7 СХЕМА ПОБУДОВИ ОЦІНКИ ДЛЯ РЕСУРСІВ В ЗАДАЧІ РОЗПОДІЛУ

Постановка завдання:

Заданий оріентований граф G = {V, U}; |V| = n та число *l*.

Число *l* показує задане число місць на яких треба розташувати вершини так, щоб, якщо з вершини з номером *i* йде дуга у вершину з номером *j*, то *i* знаходиться раніше *j*.

Треба визначити мінімальне число *m*, що вказує фіксоване обмеження на кількість вершин, які можна ставити на одне місце.

Нехай *l0* – довжина критичного шляху в графі (де під довжиною будемо розуміти кількість вершин на критичному шляху).

Зробимо помітку вершин наступним чином:

Знайдемо критичний шлях у графі. Розставимо на ньому помітки, починаючи з вершини, що не має дуг, що в неї входять. Далі розставляемо помітки на інші вершини графу відносно критичного шляху за рівнями.

Будемо позначати:

[1] – множина вершин з міткою 1;

– множина вершин з міткою 2;

...

[l0-1] – множину вершин з міткою *l0* -1;

[l0] – множину вершин з міткою *l0*.

В даній роботі отримано оцінку значення m для довільного орієнтованого графу:

*.*

Приклад:

*l0* = 6, *l* = 7, *n* = 20 (рисунок 7.1)

A picture containing building, yellow

Description automatically generatedРисунок 7.1 – Зображення графу з номерами вершин та мітками

|[1]| = 2

|[2]| = 4

|[3]| = 4

|[4]| = 4

|[5]| = 3

|[6]| = 3

m ≥ max (

) =

max () =

max (3,3,3,3,2,1,3,3,3,3,2,2) = 3

Розставимо вершини на місця та побачимо чи правильна оцінка:

1: 1, 2, 3;

2: 4, 5, 6;

3: 7, 8, 9;

4: 10, 11, 12;

5: 13, 14, 16;

6: 15, 17, 19;

7: 18, 20.

Вдалося розставити вершини на 7 місць так, щоб виконувалиь умови завдання, та кількість вершин, що можна поставити на одне місце, дорівнювала 3.

8 ПРОГРАМНИЙ ПРОДУКТ

При виконанні роботи розроблено програмний продукт, що реалізує алгортм розподілу з обмеженням на ресурси, запропонований у розділі 4.

Програму написано на мові програмування С# у середовищі програмування Visual Studio з застосуванням Windows Forms. Код програми наводиться у Додатку1.

8.1 Опис модулів програми

Проект містить наступні модулі:

* Arc.cs – клас Arc
* Knot.cs – клас Knot
* Tree.cs – клас Tree
* Graph.cs – клас Graph
* Form1.cs - клас Form1 : Form

1. Клас Knot: задає об’єкт вершини графу з параметрами: номер вершини, місце вершини у фінальному розташуванні, мітка вершини, параметр, що вказує чи була відвідувана вершина на деякій ітерації, рівень на якому знаходиться вершина, координати х та у (останні три параметри потрібні для відображення вершини на екрані).
2. Клас Arc: задає об’єкт спрямованої дуги у графі, має два параметри: вершина з якої виходить дуга та вершина у яку вона спрямована.
3. Клас Tree: Задає об’єкти вхідних дерев на яких виконується алгоритм та містить функції, що реалізують сам алгоритм розподілу.

Містить наступні параметри: колекція вершин, колекція дуг графу, параметр m, номер корневої вершини, матрицю суміжності, кількість рівнів у графі, кількість вершин на кожному рівні (останні два параметри для відображення графу на екрані).

Методи класу:

* fillMatrixOfAdjacency – метод, що заповнює матрицю суміжності графу базуючись на колекціях вершин та дуг;
* putMarks – метод, що розставляє мітки у графі використовуючи матрицю суміжності;
* putPlaces – метод, що перевіряє чи не перевищує кількість вершин з однією міткою параметр m, якщо це так викликає метод chooseKnots, та розставляє вершини на місця;
* chooseKnots – приймає два параметри: мітку *і* та колекцію вершин, що мають мітку *і*. Перевіряє чи перевищує кількість вершин з міткою *i* кількість m у циклі, якщо це так, викликає метод deleteOne, та повертає колекцію вершин з міткою *і* у якій кількість вершин дорівнює m;
* deleteOne – приймає два параметри: мітку *і* та колекцію вершин, що мають мітку *і*.

Це рекурсивна функція що, шукає одну вершину з колекції, збільшує її мітку та видаляє її з колекції.

Вона робить це наступним чином: перевіряє вершини у які їдуть дуги з вершин з колекції, шукає вершину з максимальною міткою, якщо ця мітка більше ніж *i*+1, то видаляє з колекції вершину з якої їде дуга у вершину з максимальною міткою та збільшує її мітку на 1, якщо вона дорівнює *i*+1, то метод рекурсивно викликає себе для колекцій з мітками *i*+1 та *i*. Якщо вершина з максимальним номером виявилася коренем дерева, то функція збільшує його мітку на 1.

1. Клас Graph: містить функції, що працюють з об’єктами класу Tree та задають параметри тестових графів та графу, який користувач може створити самостійно:

* craeteGraph1 – метод, що задає перший тестовий граф;
* craeteGraph2 – метод, що задає другий тестовий граф;
* craeteGraph3 – метод, що задає третій тестовий граф;
* craeteGraph4 – метод, що задає четвертий тестовий граф;
* createMyGraph – метод, що створює новий граф з однією вершиною, що є коренем дерева;
* AddKnot – приймає параметр, що відповідає вхідній вершині, додає до графа, створеного користовачем, ще одну вершину та ребро, що йде з нової вершини у вхідну.

1. Клас Form1 : Form: реалізує інтерфейс програми та містить функції, що виконуються при натисканні на кнопки інтерфейсу.

Методи:

* Button1\_Click – кнопка “Граф 1” створює новий граф звертаючись до методу craeteGraph1 класу Graph та запускає виконання алгоритму.
* Button2\_Click – кнопка “Граф 2” створює новий граф звертаючись до методу craeteGraph2 класу Graph та запускає виконання алгоритму.
* Button3\_Click – кнопка “Граф 3” створює новий граф звертаючись до методу craeteGraph3 класу Graph та запускає виконання алгоритму.
* Button4\_Click – кнопка “Граф 4” створює новий граф звертаючись до методу craeteGraph4 класу Graph та запускає виконання алгоритму.
* Button5\_Click – кнопка “Свій граф” створює новий граф звертаючись до методу createMyGraph класу Graph.
* Button6\_Click – кнопка “Додати вершину та дугу” створює нову дугу у графі звертаючись до методу AddKnot класу Graph.
* Button7\_Click – запускає виконання алгоритму на графі створеному користувачем.
* showGraph – розраховує місця веришин на дуг на екрані.
* showKnot – виводить одну вершину на екран.
* showArrow – виводить одну дугу на екран.

8.2 Інструкція користувача

На Рисунку 8.1 відповідними числами позначено:

1 – зображення графу на якому виконується алгоритм;

2 – параметр m;

3 – кнопки, призначены для перегляду тестових прикладів;

4 – оптимальне розподілення вершин на місця у вигляді: “місце: номера вершин, що на ньому знаходяться”;

5 – кнопка, призначена для задання інформації про новий граф.

A picture containing diagram

Description automatically generated

Рисунок 8.1 – Інтерфейс програми з позначеннями

При виборі кнопки з пункту 5 з’являється нове меню “Компоненти графу”, де можна задати параметри нового графу.

На Рисунку 8.2:

6 – текстове поле, у якому можна задати параметр m.

За замовчанням у графі є лише одна вершина, щоб додати нову вершину та дугу треба обрати зі списку 7 – “Вхідна вершина” вершину у яку буде іти дуга та натиснути кнопку 8 – “Додати вершину та дугу”. Після цього нова вершина з’явиться на зображенні графу та її можна буже обрати у списку вхідних вершин.

Graphical user interface

Description automatically generated

9

8

7

6

Рисунок 8.2 – Інтерфейс програми з меню для створення нового графу

Після додавання всіх вершин та дуг треба натиснути кнопку 9 – “Розставити на місця” та у списку 4 з’явиться розподілення (рисунок 8.3). Після цього додавати нові вершини до графу не можна. Щоб задати новий граф треба знову натиснути кнопку 5 – “Свій Chart, radar chart

Description automatically generatedграф” (див. рис. 8.1).

Рисунок 8.3 – Приклад створення нового графу користувачем та результати роботи на ньому алгоритму

8.3 Тестові приклади

Розглянемо виконання програми на декількох графах (рисунки 8.4 – 8.7).

A picture containing diagram

Description automatically generatedЯк можна побачити розподілення є оптимальним.

Diagram

Description automatically generatedРисунок 8.4 – Тестовий приклад “Граф 1”

Рисунок 8.5 – Тестовий приклад “Граф 2”

Chart

Description automatically generatedDiagram

Description automatically generatedРисунок 8.6 – Тестовий приклад “Граф 3”

Рисунок 8.7 – Тестовий приклад “Граф 4”

ВИСНОВКИ

При виконанні даної роботи було вивчено відомий клас оптимізаційних задач на графах, який пов’язаний з тими прикладними задачами, в яких необхідно розподілити скінчену множину робіт між фіксованою кількістю виконавців з метою мінімізації часу або мінімізації кількості виконавців при умові, що час заданий. Для спеціального виду графів запропоновано алгоритм розв’язання однієї з задач, який є альтернативою до відомого та дозволяє подати оцінку довжини упорядкування в іншому вигляді. Крім того, запропонована схема оцінювання знизу кількості виконавців.

Новий наближений алгоритм побудови упорядкування мінімальної довжини при заданій ширині для вхідних лісів програмно реалізовано на мові програмування C# в середовищі Visual Studio з застосуванням Windows Forms.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Турчина В.А., Ключник М.І. Порівняльний аналіз наближених алгоритмів розв’язання задач паралельного упорядкування // Дев’ятнадцятий міжнародний науково-практичний семінар. Комбінаторні конфігурації та їх застосування., - Кропивницький 2017. - С.141-145.
2. Алексеев В.Е., Таланов. В.А. Графы. Модели вычислений структуры данных. Учебник. / Нижний Новгород. Издательство Нижегородского университета, – 2004. – 291с.
3. Hu T. C. Parallel sequencing and assembly line problems / T.C. Hu // Oper. Res. – 1961. – P.841-849.
4. Бурдюк В. Я. Алгоритмы параллельного упорядочения: Учебное пособие / В. Я. Бурдюк, В. А. Турчина. – Днепропетровск: ДГУ, 1985. – 83с.
5. Гэри М., Джонсон Д. Вычислительные машины и труднорешаемые задачи. Москва: “МИР”, 1982. - С. 31.
6. Теория расписаний и вычислительные машины / Под ред. Э. Г. Коффмана. – М.: Наука, 1984. – С. 158.
7. Турчина В. А. Дослідження оцінок довжини паралельного упорядкування вершин графу / В. А. Турчина, К. Д. Караваєв // Питання прикладної математики і математичного моделювання. – Д., 2018. – С. 186 – 195.

ДОДАТОК А Код програми

Клас вершин

public class Knot

{

public int number;

public int level;

public int place;

public int mark;

public int x;

public int y;

public bool changed;

public bool root;

public Knot() { }

public Knot(int number, int level)

{

this.number = number;

this.level = level;

}

}

Клас дуг

public class Arc

{

public Knot knotOut;

public Knot knotIn;

public Arc(Knot knotOut, Knot knotIn)

{

this.knotIn = knotIn;

this.knotOut = knotOut;

}

}

Клас дерево, що задає граф та реалізує у ньому алгоритм

public class Tree

{

public List<Knot> knots;

public List<Arc> arcs;

public int m;

public int rootNum;

public int levels;

public List<int> countLevels;

public int[,] matrixOfAdjacency;

Метод, що розраховує матрицю суміжності:

public void fillMatrixOfAdjacency()

{

matrixOfAdjacency = new int[knots.Count, knots.Count];

for (int i = 0; i < knots.Count; i++)

{

knots[i].mark = -1;

knots[i].place = -1;

knots[i].changed = false;

for (int j = 0; j < knots.Count; j++)

{

matrixOfAdjacency[i, j] = 0;

}

}

for (int i = 0; i < arcs.Count; i++)

{

matrixOfAdjacency[arcs[i].knotOut.number, arcs[i].knotIn.number] = 1;

}

}

Метод, що розставляє мітки:

public void putMarks()

{

bool isFilled = true;

bool[] putInRow = new bool[knots.Count];

bool contin = false;

int count = 0;

while (isFilled)

{

contin = false;

for (int j = 0; j < knots.Count; j++)

{

putInRow[j] = true;

if (knots[j].mark == -1)

{

for (int i = 0; i < knots.Count; i++)

{

if (knots[i].mark == -1 && matrixOfAdjacency[i, j] == 1)

{

putInRow[j] = false;

break;

}

}

}

else

{

putInRow[j] = false;

}

}

for(int i = 0; i < knots.Count; i++)

{

if (putInRow[i] == true)

{

contin = true;

knots[i].mark = count;

}

}

count++;

isFilled = contin;

}

}

Метод, що видаляє з листа одну вершину:

public void deleteOne(List<int> numbers, int place)

{

int max = place;

int num = rootNum;

for (int i = 0; i < arcs.Count; i++)

{

if (numbers.Contains(arcs[i].knotOut.number) && max < arcs[i].knotIn.mark)

{

max = arcs[i].knotIn.mark;

num = arcs[i].knotOut.number;

}

}

if (max > place + 1)

{

knots[num].mark++;

knots[num].changed = true;

}

else

{

if (num == rootNum)

{

knots[rootNum].mark++;

}

else

{

List<int> numbersNext = new List<int> { };

for (int i = 0; i < knots.Count; i++)

{

if (knots[i].mark == place + 1 && !knots[i].changed)

{

numbersNext.Add(i);

}

}

deleteOne(numbersNext, place + 1);

deleteOne(numbers, place);

}

}

}

Метод, що допомагає уникнути «зайвих» вершин у листі з вершинами з однією міткою:

public List<int> chooseKnots(List<int> numbers, int place)

{

while (numbers.Count > m)

{

deleteOne(numbers, place);

numbers = new List<int> { };

for (int i = 0; i < knots.Count; i++)

{

if (knots[i].mark == place)

{

numbers.Add(i);

}

}

}

return numbers;

}

Метод, що розставляє вершини на місця:

public void putPlaces()

{

int place = 0;

List<int> numbers;

while (knots[rootNum].place == -1)

{

numbers = new List<int> { };

for (int i = 0; i < knots.Count; i++)

{

knots[i].changed = false;

if (knots[i].mark == place)

{

numbers.Add(i);

}

}

if (numbers.Count > m)

{

numbers = chooseKnots(numbers, place);

}

for (int i = 0; i < numbers.Count; i++)

{

knots[numbers[i]].place = place;

}

place++;

}

}

}

Клас, що збірігає відомості про тестові графи та «свій граф»

class Graphs

{

public void createMyGraph(Tree tr)

{

tr.levels = 1;

tr.countLevels = new List<int> { 1 };

tr.countLevels[0] = 1;

tr.knots = new List<Knot> {new Knot(0, 1)};

tr.rootNum = 0;

tr.arcs = new List<Arc> { };

}

public void AddKnot(Tree tr, int inputKnot)

{

if (tr.knots[inputKnot - 1].level == tr.levels)

{

tr.levels++;

tr.countLevels.Add(1);

tr.knots.Add(new Knot(tr.knots.Count, tr.levels));

tr.arcs.Add(new Arc(tr.knots[tr.knots.Count - 1], tr.knots[inputKnot - 1]));

}

else

{

tr.countLevels[tr.knots[inputKnot - 1].level]++;

tr.knots.Add(new Knot(tr.knots.Count, tr.knots[inputKnot - 1].level + 1));

tr.arcs.Add(new Arc(tr.knots[tr.knots.Count - 1], tr.knots[inputKnot - 1]));

}

}

public void craeteGraph1(Tree tr);

public void craeteGraph2(Tree tr);

public void craeteGraph3(Tree tr);

public void craeteGraph4(Tree tr);