Лекція 11. Жадібні алгоритми
План.

1. Визначення.
2. Алгоритм Прима.
3. Алгоритм Крускала.
4. Алгоритм Дейкстри.
5. Дерева Хаффмана.

1.

Жадібний алгоритм — простий і прямолінійний евристичний алгоритм, який приймає найкраще рішення, виходячи з наявних на поточному етапі даних, не турбуючись про можливі наслідки, сподіваючись врешті-решт отримати оптимальне рішення. Легкий в реалізації і часто дуже ефективний за часом виконання. Багато задач не можуть бути розв'язані з його допомогою.

Наприклад, використання жадібної стратегії для задачі комівояжера породжує наступний алгоритм: «На кожному етапі вибирати найближче з невідвіданих міст».

Зазвичай, жадібний алгоритм базується на п'яти принципах:

- Набір можливих варіантів, з яких робиться вибір;
- Функція вибору, за допомогою якої знаходиться найкращий варіант;
- Функція придатності, яка визначає придатність отриманого набору;
- Функція цілі, оцінює цінність рішення, не виражена явно;
- Функція розв'язку, яка вказує на те, що знайдене кінцеве рішення.

Придатний набір варіантів — такий, що обіцяє не просто отримання рішення, а отримання оптимального рішення задачі.

На відміну від динамічного програмування, що розв'язує проблему знизу догори, жадібна стратегія робить це згори донизу, роблячи один жадібний вибір за іншим, зводячи велику задачу до малої.

Жадібний алгоритм добре розв'язує деякі задачі, а інші — ні. Більшість задач, для яких він спрацьовує добре, мають дві властивості: по-перше, до них можливо застосувати Принцип жадібного вибору, по-друге, вони мають властивість Оптимальної підструктури.

Принцип жадібного вибору.
До оптимізаційної задачі можна застосувати принцип жадібного вибору, якщо послідовність локально оптимальних виборів дає глобально оптимальний розв'язок. В типовому випадку доведення оптимальності має таку схему: спочатку доводиться, що жадібний вибір на першому етапі не унеможливлює шлях до оптимального розв'язку: для всякого розв'язку є інше, узгоджене із

жадібним і не гірше першого. Далі доводиться, що підзадача, що виникла після жадібного вибору на першому етапі, аналогічна початковій, і міркування закінчується за індукцією. Інакше кажучи, жадібний алгоритм ніколи не переглядає свої попередні вибори для здійснення наступного, на відміну від динамічного програмування.

Властивість оптимальної підструктури.
«Задача має оптимальну підструктуру, якщо оптимальне рішення задачі містить оптимальне рішення для підзадач»[1]. Інакше кажучи, задача має оптимальну підструктуру, якщо кожен наступний крок веде до оптимального розв'язку. Прикладом 'неоптимальної підструктури' може бути ситуація в шахах, коли взяття ферзя (хороший наступний крок) веде до програшу партії в цілому.

Коли алгоритми жадібного типу зазнають невдачі.
Жадібні алгоритми можна характеризувати як 'короткозорі' і 'невідновлювані'. Вони ідеальні лише для задач з 'оптимальною підструктурою'. Попри це, жадібні алгоритми найкраще підходять для простих задач. Для багатьох інших задач жадібні алгоритми зазнають невдачі у продукуванні оптимального розв'язку, і можуть навіть видати найгірший з можливих розв'язків. Один з прикладів — алгоритм найближчого сусіднього міста, описаний вище.

Також у задачі розміну монет, якщо є тільки 25, 10 і 4-центові монети, жадібний алгоритм не зможе зробити розмін 41 цента.

Приклади.
Розмін монет.
Задача. Монетна система деякої держави складається з монет вартістю. Вимагається видати суму S найменшою можливою кількістю монет.

Жадібний алгоритм розв'язання цієї задачі такий. Беремо найбільшу можливу кількість монет вартості. Так само знаходимо скільки потрібно монет меншого номіналу, і т. д.

Для цієї задачі жадібний алгоритм не завжди дає правильне рішення. Наприклад, суму в 10 копійок монетами в 1, 5 і 6 коп. жадібний алгоритм розміняє так: 6 коп. — 1 шт., 1 коп. — 4 шт., в той час як правильне рішення 2 монети по 5 копійок. Тим не менш, на всіх реальних монетних системах жадібний алгоритм видає правильну відповідь.

2.

Алгоритм Прима - алгоритм побудови мінімального кістякового дерева зваженого зв'язного неорієнтованого графа. Це жадібний алгоритм.

Побудова починається з дерева, що включає в себе одну (довільну) вершину. Протягом роботи алгоритму дерево розростається, поки не охопить всі вершини вихідного графа. На кожному кроці алгоритму до поточного дереву

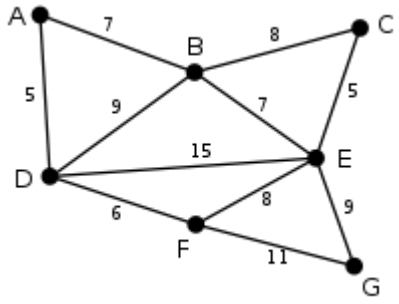
приєднується найлегше з ребер, що з'єднують вершину з побудованого дерева і вершину, що не належить дереву.

Алгоритм

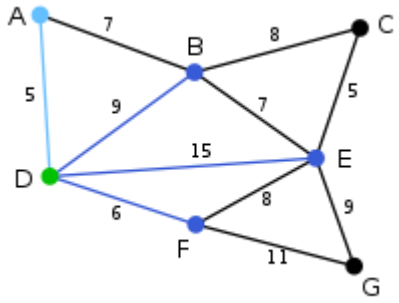
1. Спочатку ребра сортують за зростанням ваги.
2. Додають найменше ребро в дерево.
3. Зі списку ребер із найменшою вагою вибирають таке нове ребро, щоб одна з його вершин належала дереву, а інша — ні.
4. Це ребро додають у дерево і знову переходять до кроку 3.
5. Робота закінчується, коли всі вершини будуть у дереві.

Приклад виконання:

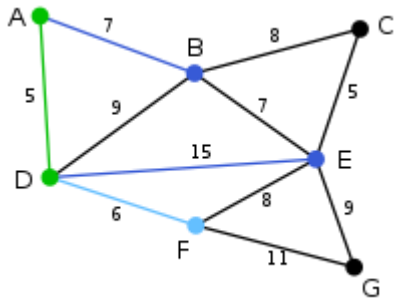
1.Вихідний зважений граф. Числа біля ребер показують їх ваги, які можна розглядати як відстані між вершинами.



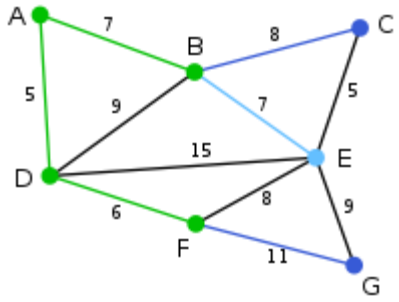
2.Як початкова довільно вибирається вершина D. Кожна з вершин A, B, E і F з'єднана з D єдиним ребром. Вершина A - найближча до D, і вибирається як друга вершина разом з ребром AD.



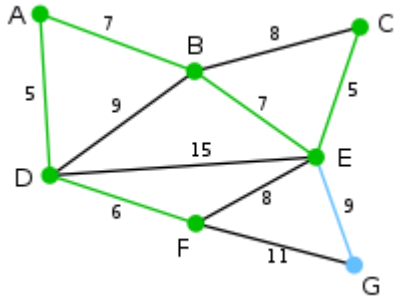
3.Наступна вершина - найближча до будь-якої з обраних вершин D або A. B віддалена від D на 9 і від A - на 7. Відстань до E дорівнює 15, а до F - 6. F є найближчою вершиною, тому вона включається в дерево F разом з ребром DF.



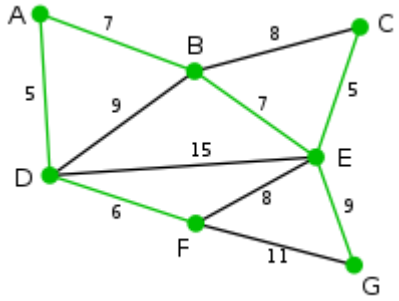
4.Аналогічними кроками приходимо до такого дерева. У цьому випадку є можливість вибрати або C, або E, або G. C віддалена від B на 8, E віддалена від B на 7, а G віддалена від F на 11. E - найближча вершина, тому вибирається E і ребро BE.



5.Єдина вершина, що залишилася - G. Відстань від F до неї одно 11, від E - 9. E ближче, тому вибирається вершина G і ребро EG.



6.Вибрані всі вершини, мінімальне кістякове дерево побудовано

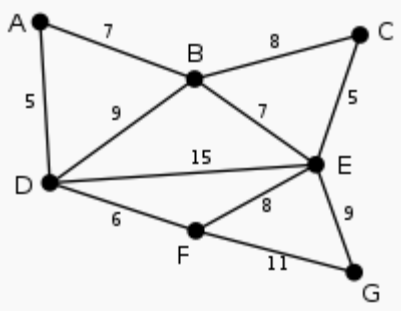
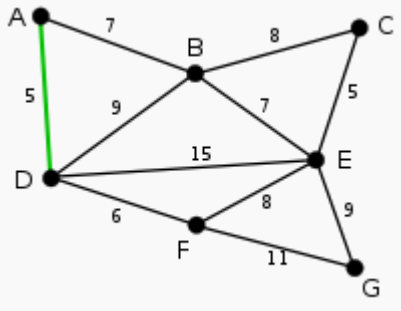
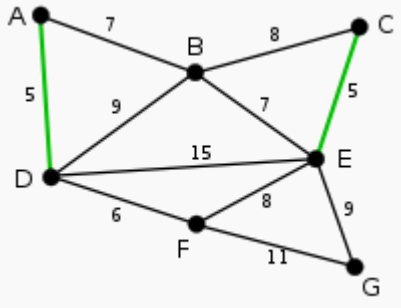


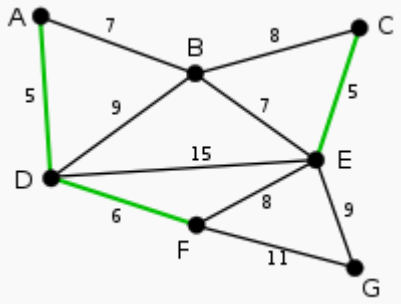
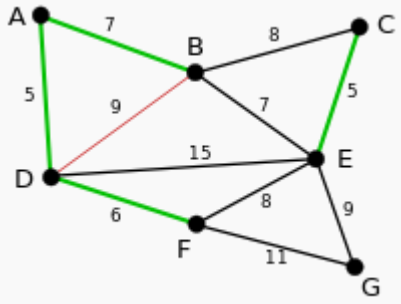
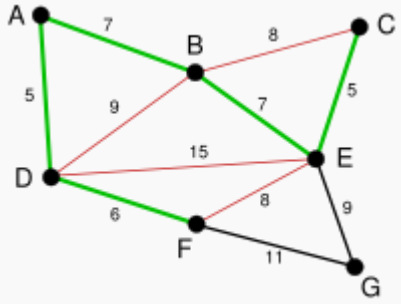
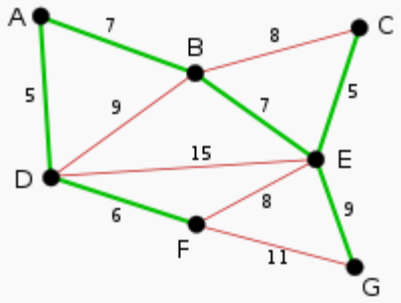
3.

Алгоритм Крускала — алгоритм побудови мінімального кістякового дерева зваженого неорієнтовного графа. Алгоритм було вперше описано Джозефом Крускалом 1956 року.

Візьмемо зважений зв'язний граф G=(V, E), де V — множина вершин, E — множина ребер, для кожного з яких задано вагу. Тоді ациклічна множина ребер, що поєднують усі вершини графа і чия загальна вага мінімальна, називається *мінімальним кістяковим деревом*.

Алгоритм Крускала починається з побудови виродженого лісу, що містить V дерев, кожне з яких складається з однієї вершини. Далі виконуються операції об'єднання двох дерев, для чого використовуються найкоротші можливі ребра, поки не утвориться єдине дерево. Це дерево і буде мінімальним кістяковим деревом.

	Початковий граф. Цифри над ребрами позначають їх вагу. Жодне з ребер не додане до кістякового дерева.
	**AD** і **CE** мають найменшу вагу 5, і **AD** вибирається з них довільно та додається до кістякового дерева.
	На цьому кроці **CE** є найлегшим ребром з вагою 5, тому воно також додається до дерева.

	Аналогічним чином обирається найлегше з недоданих ребер графа **DF** з вагою 6 і додається до кістякового дерева.
	Наступними найлегшими ребрами є **AB** і **BE**, обидва вагою 7. **AB** обирається довільно і додається до кістякового дерева. **BD** фарбується у червоний колір, оскільки воно є частиною циклу **ABD**.
	Наступним додається ребро **BE** з вагою 7. Червоним забарвлюємо ребра **BC** (цикл **BCE**), **DE** (цикл **DEBA**) і **FE** (цикл **FEBAD**).
	Додаємо ребро **EG** вагою 9 і отримуємо мінімальне кістякове дерево.

Код на C++[ред. • ред. код]

```
int cn; //число вершин
vector< vector<int> > ady; //матриця суміжності

// Повертає матрицю суміжності мінімального дерева
vector< vector<int> > Grafo :: kruskal(){
```

```
    vector< vector<int> > adyacencia = this->ady;
    vector< vector<int> > arbol(cn);
    vector<int> pertenece(cn); // позначає, чи належить дереву вершина

    for(int i = 0; i < cn; i++){
        arbol[i] = vector<int> (cn, INF);
        pertenece[i] = i;
    }

    int nodoA;
    int nodoB;
    int arcos = 1;
    while(arcos < cn){
        // Знайти найлегше ребро, що не утворює циклів і зберегти вершини і вагу.
        int min = INF;
        for(int i = 0; i < cn; i++)
            for(int j = 0; j < cn; j++)
                if(min > adyacencia[i][j] && pertenece[i] != pertenece[j]){
                    min = adyacencia[i][j];
                    nodoA = i;
                    nodoB = j;
                }

        // Якщо вершини не належать до одного дерева, додаємо ребро між ними до дерева.
        if(pertenece[nodoA] != pertenece[nodoB]){
            arbol[nodoA][nodoB] = min;
            arbol[nodoB][nodoA] = min;

            // Усі вершини дерева nodoB зараз належать до дерева nodoA.
             int temp = pertenece[nodoB];
             pertenece[nodoB] = pertenece[nodoA];
             for(int k = 0; k < cn; k++)
                        if(pertenece[k] == temp)
                                pertenece[k] = pertenece[nodoA];

            arcos++;
        }
    }
    return arbol;
}
```

Оцінка складності[ред. • ред. код]

Алгоритм Крускала (як і алгоритм Прима) є класичним алгоритмом розв'язання задачі пошуку мінімального кістякового дерева. У разі використання

найшвидших реалізацій час його роботи становить . Основна частина часу витрачається на сортування ребер за вагою.

4.

Алгоритм Дейкстри — алгоритм на графах, відкритий Дейкстрою. Знаходить найкоротший шлях від однієї вершини графа до всіх інших вершин. Класичний алгоритм Дейкстри працює тільки для графів без циклів від'ємної довжини.

Формулювання задачі.
Варіант 1. Дана мережа автомобільних доріг, що з'єднують міста Львівської області. Знайти найкоротшу відстань від Львова до кожного міста області, якщо рухатись можна тільки по дорогах.

Варіант 2. Дана карта велосипедних доріжок Латвії та Білорусі. Знайти мінімальну відстань, яку треба проїхати, щоб дістатися від Риги до Бобруйська.

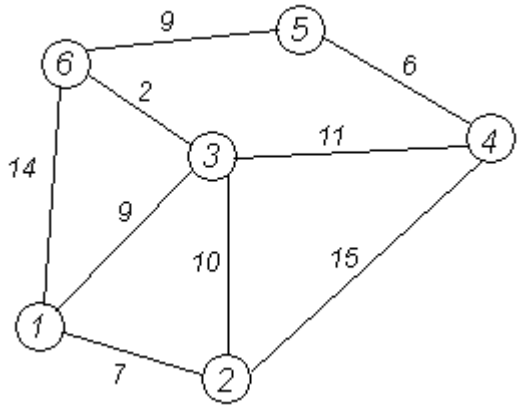
Варіант 3. Є план міста з нанесеними на нього місцями розміщення пожежних частин. Знайти найближчу до кожного дому пожежну станцію.

Абстракція.

Дано неорієнтований зв'язний граф G(V, U). Знайти відстань від вершини a до всіх інших вершин V.

Інтуїтивне пояснення.

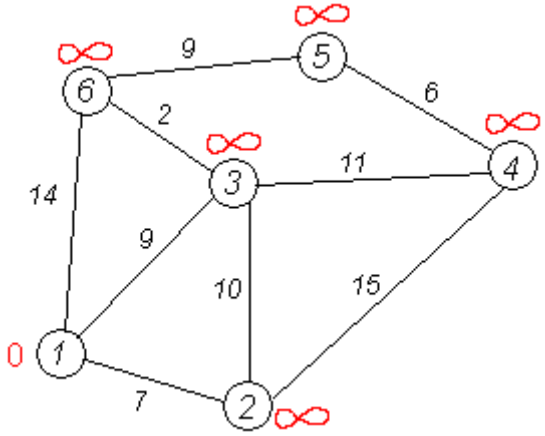
Зберігатимемо поточну мінімальну відстань до всіх вершин V (від даної вершини a) і на кожному кроці алгоритму намагатимемося зменшити цю відстань. Спочатку встановимо відстані до всіх вершин рівними нескінченості,



а до вершини a — нулю. Розглянемо виконання алгоритму на прикладі. Хай потрібно знайти відстані від 1-ої вершини до всіх інших. Кружечками позначені вершини, лініями — шляхи між ними («дуги»). Над дугами позначена їх «ціна» — довжина шляху. Надписом над кружечком позначена поточна найкоротша відстань до вершини.

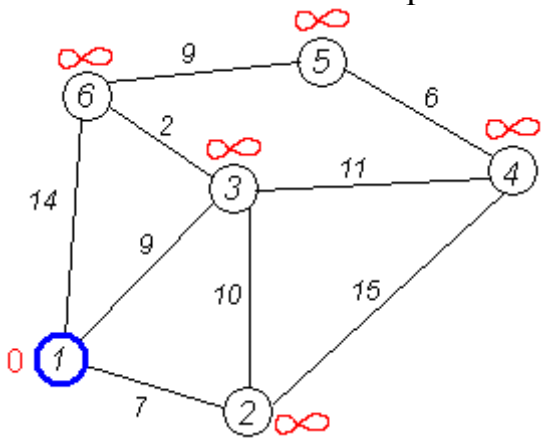
Крок 1.

Ініціалізація. Відстань до всіх вершин графа V = . Відстань до a = 0. Жодна вершина графа ще не опрацьована.



Крок 2.

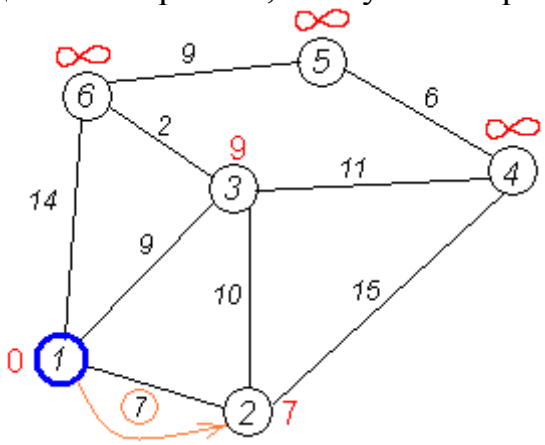
Знаходимо таку вершину (із ще не оброблених), поточна найкоротша відстань до якої мінімальна. В нашому випадку це вершина 1. Обходимо всіх її сусідів і, якщо шлях в сусідню вершину через 1 менший за поточний мінімальний шлях в цю сусідню вершину, то запам'ятовуємо цей новий, коротший шлях як поточний найкоротший шлях до сусіда.



Крок 3.

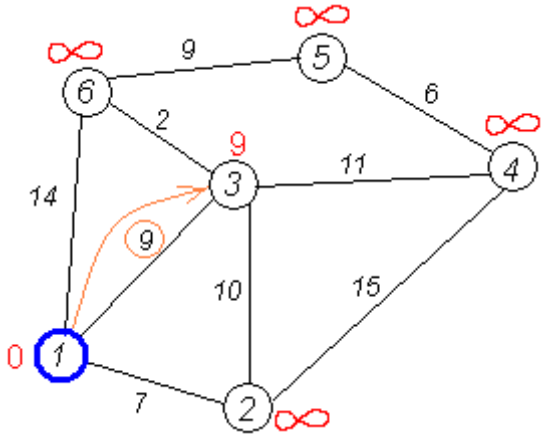
Перший по порядку сусід 1-ї вершини — 2-а вершина. Шлях до неї через 1-у вершину дорівнює найкоротшій відстані до 1-ї вершини + довжина дуги між 1-ю та 2-ю вершиною, тобто 0 + 7 = 7. Це менше поточного найкоротшого шляху

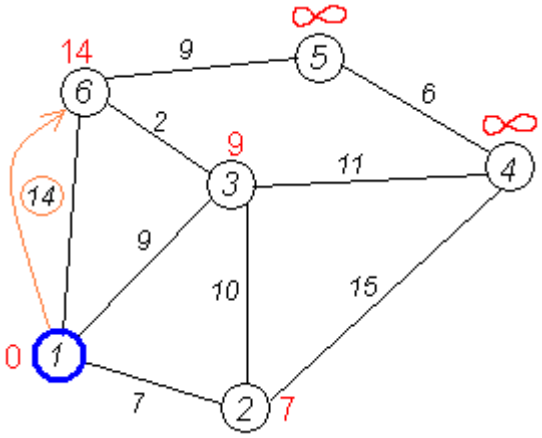
до 2-ї вершини, тому найкоротший шлях до 2-ї вершини дорівнює 7.



Кроки 4, 5.

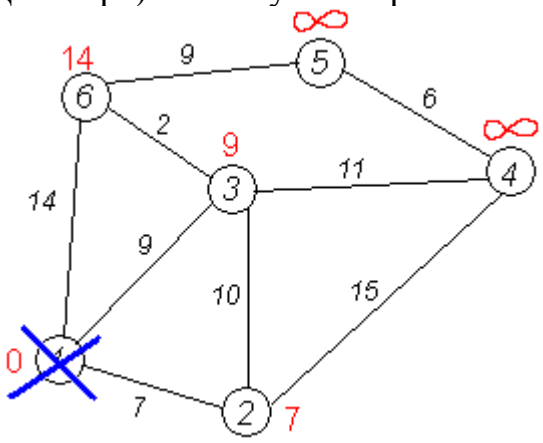
Аналогічну операцію проробляємо з двома іншими сусідами 1-ї вершини — 3-ю та 6-ю.





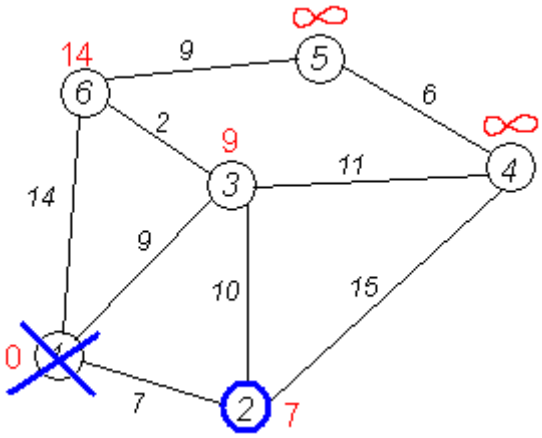
Крок 6.

Всі сусіди вершини 1 перевірені. Поточна мінімальна відстань до вершини 1 вважається остаточною і обговоренню не підлягає (те, що це дійсно так, вперше довів Дейкстра). Тому викреслимо її з графа, щоб відмітити цей факт.



Крок 7.

Практично відбувається повернення до кроку 2. Знову знаходимо «найближчу» необроблену (невикреслену) вершину. Це вершина 2 з поточною найкоротшою



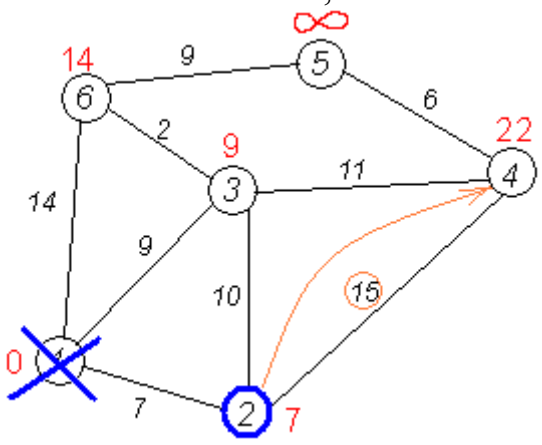
відстанню до неї = 7. І знову намагаємося зменшити відстань до всіх сусідів 2-ої вершини, намагаючись пройти в них через 2-у. Сусідами 2-ої вершини є 1, 3, 4.

Крок 8.

Перший (по порядку) сусід вершини № 2 — 1-а вершина. Але вона вже оброблена (або викреслена — див. крок 6). Тому з 1-ою вершиною нічого не робимо.

Крок 8 (з іншими вхідними данними).

Інший сусід вершини 2 — вершина 4. Якщо йти в неї через 2-у, то шлях буде = найкоротша відстань до 2-ої + відстань між 2-ою і 4-ою вершинами = 7 + 15 = 22. Оскільки 22 < ∞, встановлюємо відстань до вершини № 4 рівним

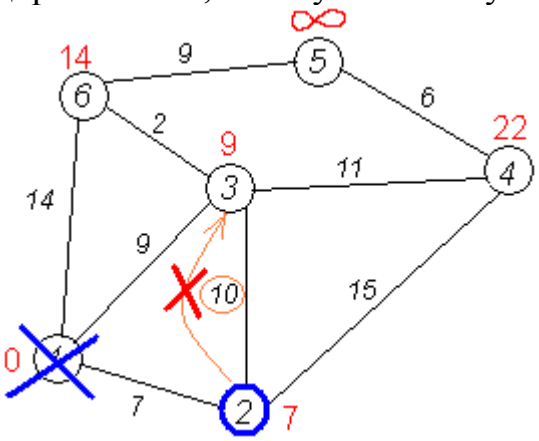


22.

Крок 9.

Ще один сусід вершини 2 — вершина 3. Якщо йти в неї через 2-у, то шлях буде = 7 + 10 = 17. Але 17 більше за відстань, що вже запам'ятали раніше до вершини

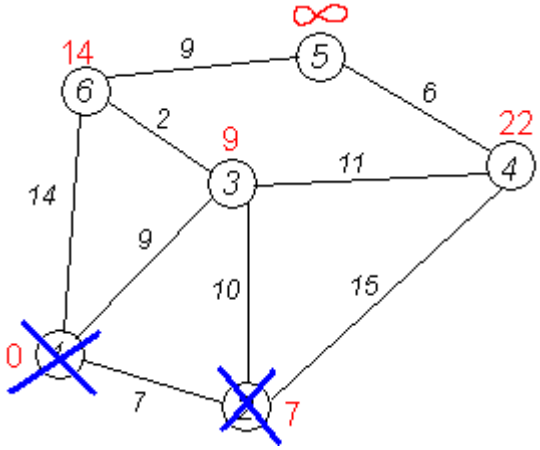
№ 3 і дорівнює 9, тому поточну відстань до 3-ої вершини не



міняємо.

Крок 10.

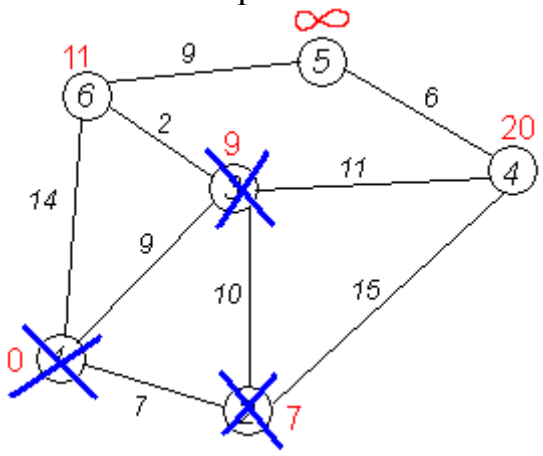
Всі сусіди вершини 2 переглянуті, заморожуємо відстань до неї і викреслюємо



її з графа.

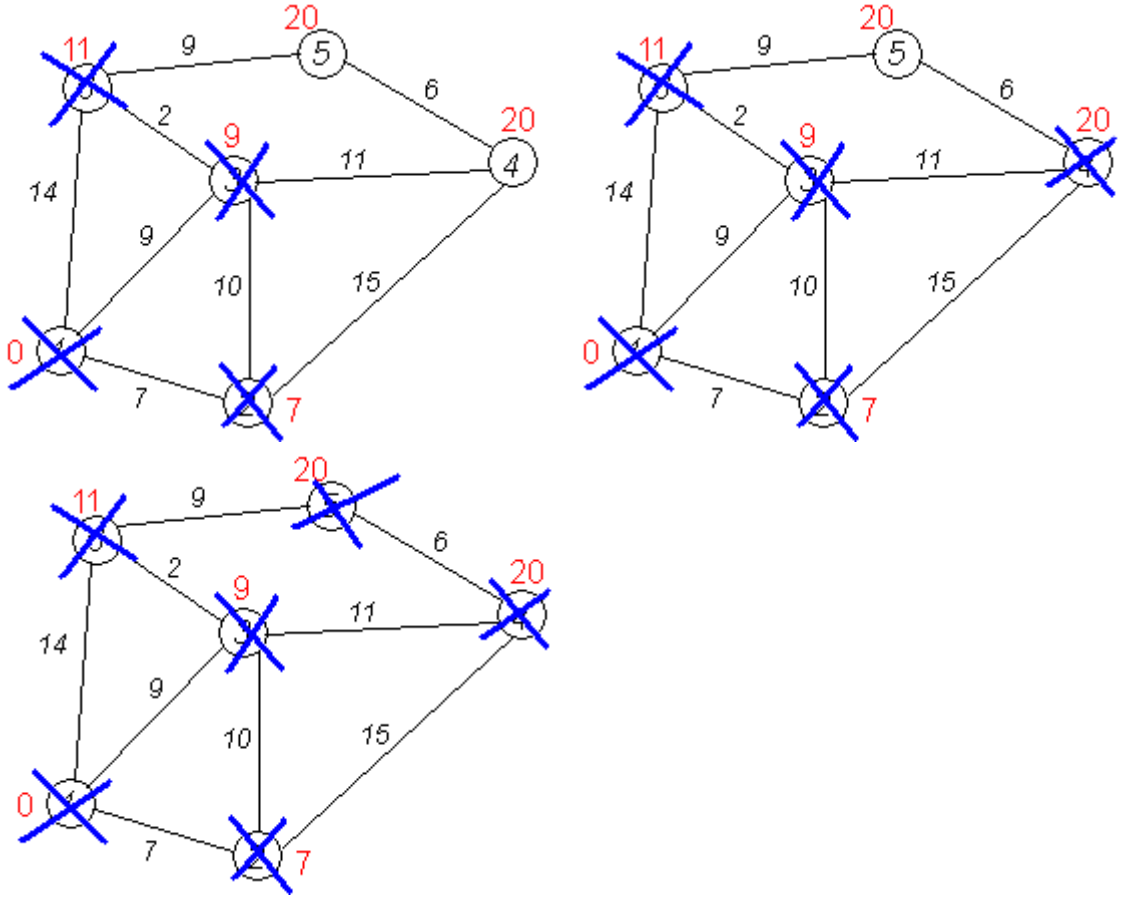
Кроки 11 — 15.

По вже «відпрацьованій» схемі повторюємо кроки 2 — 6. Тепер «найближчою» виявляється вершина № 3. Після її «обробки» отримаємо такі результати:



Наступні кроки.

Пророблюємо те саме з вершинами, що залишилися (№ по порядку: 6, 4 і 5).



Завершення виконання алгоритму.

Алгоритм закінчує роботу, коли викреслені всі вершини. Результат його роботи видно на останньому малюнку: найкоротший шлях від 1-ої вершини до 2-ої становить 7, до 3-ої — 9, до 4-ої — 20, до 5-ої — 20, до 6-ої — 11 умовних одиниць.

Найпростіша реалізація.

Найпростіша реалізація алгоритма Дейкстри потребує дій. У ній використовується масив відстаней та масив позначок. На початку алгоритму відстані заповнюються великим позитивним числом (більшим максимального можливого шляху в графі), а масив позначок заповнюється нулями. Потім відстань для початкової вершини вважається рівною нулю і запускається основний цикл.

На кожному кроці циклу ми шукаємо вершину з мінімальною відстанню і прапором рівним нулю. Потім ми встановлюємо в ній позначку 1 і перевіряємо всі сусідні з нею вершини. Якщо в ній відстань більша, ніж сума відстані до поточної вершини і довжини ребра, то зменшуємо його. Цикл завершується коли позначки всіх вершин стають рівними 1.

5.

Алгоритм Хаффмана - адаптивний жадібний алгоритм оптимального префіксного кодування алфавіту з мінімальною надмірністю. Був розроблений в 1952 році аспірантом Массачусетського технологічного інституту Девідом Хаффманом при написанні ним курсової роботи. В даний час використовується в багатьох програмах стиснення даних.

На відміну від алгоритму Шеннона - Фано, алгоритм Хаффмана залишається завжди оптимальним і для вторинних алфавітів m2 з більш ніж двома символами.

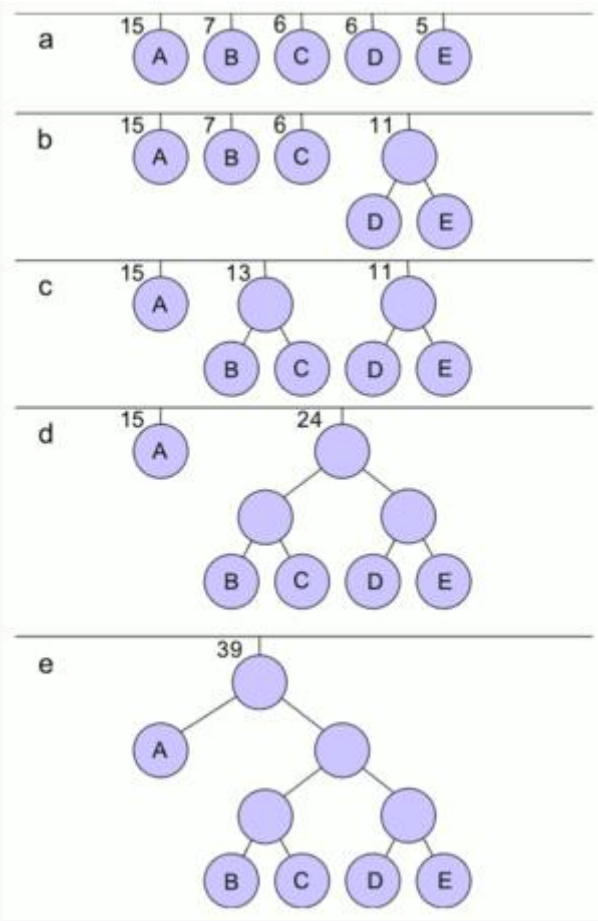
Цей метод кодування складається з двох основних етапів:

1. Побудова оптимального кодового дерева.
2. Побудова відображення коду-символів на основі побудованого дерева.

Один з перших алгоритмів ефективного кодування інформації був запропонований Д. А. Хаффманом в 1952 році. Ідея алгоритму полягає в наступному: знаючи ймовірності символів у повідомленні, можна описати процедуру побудови кодів змінної довжини, що складаються з цілої кількості бітів. Символам з більшою ймовірністю ставляться у відповідність більш короткі коди. Коди Хаффмана володіють властивістю префіксності (тобто жодне кодове слово не є префіксом іншого), що дозволяє однозначно їх декодувати.

Класичний алгоритм Хаффмана на вході отримує таблицю частот з якими зустрічаються символи у повідомленні. Далі на підставі цієї таблиці будується дерево кодування Хаффмана (Н-дерево).

1. Символи вхідного алфавіту утворюють список вільних вузлів. Кожен лист має вагу, яка може бути рівною або ймовірності, або кількості входжень символу у стиснене повідомлення.
2. Вибираються два вільних вузла дерева з найменшими вагами.
3. Створюється їх батьковий вузол з вагою, рівною їх сумарній вазі.
4. Вузол-батько додається в список вільних вузлів, а два його нащадка видаляються з цього списку.
5. Одній дузі, котра виходить з вузла батька, ставиться у відповідність біт 1, інший — біт 0.
6. Кроки, починаючи з другого, повторюються доти, поки в списку вільних вузлів не залишиться тільки один вільний вузол. Він і буде вважатися коренем дерева.



Кодування Хаффмана

Припустимо, у нас є наступна таблиця частот:

15	**7**	**6**	**6**	**5**
A	B	C	D	E

Цей процес можна представити як побудова дерева, корінь якого — символ з сумою ймовірностей об'єднаних символів, що вийшов при об'єднанні символів з останнього кроку, його n_0 нащадків — символи з попереднього кроку і т. д.

Щоб визначити код для кожного із символів, що входять в повідомлення, ми повинні пройти шлях від листа дерева, який відповідає поточному символу, до його кореня, накопичуючи біти при переміщенні по гілках дерева (перша гілка в дорозі відповідає молодшому біту). Отримана таким чином послідовність бітів є кодом даного символу, записаним у зворотному порядку.

Для даної таблиці символів коди Хаффмана будуть виглядати наступним чином.

A	**B**	**C**	**D**	**E**

0	100	101	110	111

Оскільки жоден з отриманих кодів не є префіксом іншого, вони можуть бути однозначно декодовані при читанні їх з потоку. Крім того, найбільш частий символ повідомлення A закодований найменшою кількістю біт, а найбільш рідкісний символ E — найбільшим.

При цьому загальна довжина повідомлення, що складається з наведених у таблиці символів, складе 87 біт (в середньому 2,2308 біта на символ). При використанні рівномірного кодування загальна довжина повідомлення склала б 117 біт (рівно 3 біта на символ). Зауважимо, що ентропія джерел, незалежним чином породжує символи із зазначеними частотами, складає ~ 2,1858 біта на символ, тобто надмірність побудованого для такого джерела коду Хаффмана, що розуміється, як відмінність середнього числа біт на символ від ентропії, становить менше 0,05 біт на символ.

Класичний алгоритм Хаффмана має ряд істотних недоліків. По-перше, для відновлення вмісту стиснутого повідомлення декодер повинен знати таблицю частот, якою користувався кодер. Отже, довжина стиснутого повідомлення збільшується на довжину таблиці частот, яка повинна надсилатися попереду даних, що може звести нанівець всі зусилля по стисненню повідомлення. Крім того, необхідність наявності повної частотної статистики перед початком власне кодування вимагає двох проходів по повідомленню: одного для побудови моделі повідомлення (таблиці частот і H-дерева), іншого для власне кодування. По-друге, надмірність кодування звертається в нуль лише в тих випадках, коли ймовірності кодованих символів є зворотними ступенями числа 2. По-третє, для джерела з ентропією, що не перевищує 1 (наприклад, для двійкового джерела), безпосереднє застосування коду Хаффмана безглуздо.

Адаптивне стиснення.

Адаптивне стиснення дозволяє не передавати модель повідомлення разом з ним самим і обмежитися одним проходом за повідомленням як при кодуванні, так і при декодуванні.

У створенні алгоритму адаптивного кодування Хаффмана найбільші складності виникають при розробці процедури поновлення моделі чергових символів. Теоретично можна було би просто вставити всередину цієї процедури повну побудову дерева кодування Хаффмана, однак, такий алгоритм стиснення мав би неприйнятно низьку швидкодію, так як побудова H-дерева - це занадто велика робота і робити її при обробці кожного символу нерозумно. На щастя, існує спосіб модифікувати вже існуюче H-дерево так, щоб відобразити обробку нового символу.

Оновлення дерева при зчитуванні чергового символу повідомлення складається з двох операцій.

Перша - збільшення ваги вузлів дерева. Спочатку збільшуємо вагу листа, відповідного вважають символом, за одиницю. Потім збільшуємо вагу батька,

щоб привести його у відповідність з новими значеннями ваги нащадків. Цей процес продовжується до тих пір, поки ми не доберемося до кореня дерева. Середнє число операцій збільшення ваги дорівнює середній кількістю бітів, необхідних для того, щоб закодувати символ.

Друга операція - перестановка вузлів дерева - потрібна тоді, коли збільшення ваги вузла призводить до порушення властивості впорядкованості, тобто тоді, коли збільшена вага вузла стає більшою, ніж вага наступного порядку вузла. Якщо і далі продовжувати обробляти збільшення ваги, рухаючись до кореня дерева, то дерево перестане бути деревом Хаффмана.

Щоб зберегти упорядкованість дерева кодування, алгоритм працює в такий спосіб. Нехай нова збільшена вага вузла дорівнює W+1. Тоді починаємо рухатися по списку у бік збільшення ваги, поки не знайдемо останній вузол з вагою W. Переставимо поточний і знайдений вузли між собою в списку, відновлюючи таким чином порядок в дереві (при цьому батьки кожного з вузлів теж зміняться). На цьому операція перестановки закінчується.

Після перестановки операція збільшення ваги вузлів продовжується далі. Наступний вузол, вага якого буде збільшена алгоритмом, - це новий батько вузла, збільшення ваги якого викликало перестановку.

Переповнення.

У процесі роботи алгоритму стиснення ваги вузлів в дереві кодування Хаффмана неухильно зростає. Перша проблема виникає тоді, коли вага кореня дерева починає перебільшувати місткість комірки, в якій він зберігається. Як правило, це 16-бітове значення і, отже, не може бути більше, ніж 65535. Друга проблема, яка заслуговує ще більшої уваги, може виникнути значно раніше, коли розмір найдовшого коду Хаффмана перебільшує місткість комірки, яка використовується для того, щоб передати його у вихідний потік. Декодеру все одно, якої довжини код він декодує, оскільки він рухається зверху вниз по дереву кодування, вибираючи з вхідного потоку по одному біту. Кодер ж повинен починати від листа дерева і рухатися вгору до кореня, збираючи біти, які потрібно передати. Зазвичай це відбувається зі змінною типу «цілий» і, коли довжина коду Хаффмана перевершує розмір типу «цілий» в бітах, настає переповнення.

Можна довести, що максимальну довжину код Хаффмана для повідомлень з одним і тим же вхідним алфавітом матиме, якщо частоти символів утворюють послідовність Фібоначчі. Повідомлення з частотами символів, рівними числам Фібоначчі до Fib(18), - це відмінний спосіб протестувати роботу програми стиснення по Хаффману.

Масштабування ваг вузлів дерева Хаффмана.

Беручи до уваги сказане вище, алгоритм поновлення дерева Хаффмана повинен бути змінений таким чином: при збільшенні ваги потрібно перевіряти його на досягнення допустимого максимуму. Якщо ми досягли максимуму, то необхідно «масштабувати» вагу, зазвичай розділивши вагу листка на ціле число, наприклад, 2, а потім перерахувавши вагу всіх інших вузлів.

Однак при діленні ваги навпіл виникає проблема, пов'язана з тим, що після виконання цієї операції дерево може змінити свою форму. Пояснюється це тим, що ми ділимо цілі числа і при діленні відкидаємо дробову частину.

Правильно організоване дерево Хаффмана після масштабування може мати форму, яка значно відрізняється від вихідної. Це відбувається тому, що масштабування призводить до втрати точності нашої статистики. Але зі збором нової статистики наслідки цих «помилок» практично сходять нанівець. Масштабування ваги - досить дорога операція, оскільки вона призводить до необхідності заново будувати все дерево кодування. Але так як необхідність в ній виникає відносно рідко, то з цим можна змиритися.

Виграш від масштабування

Масштабування ваги вузлів дерева через певні інтервали дає несподіваний результат. Незважаючи на те, що при масштабуванні відбувається втрата точності статистики, тести показують, що воно призводить до кращих показників стиснення, ніж якщо б масштабування відкладалося. Це можна пояснити тим, що поточні символи стисненого потоку більше «схожі» на своїх близьких попередників, ніж на тих, які зустрічалися набагато раніше. Масштабування призводить до зменшення впливу «давніх» символів на статистику і до збільшення впливу на неї «недавніх» символів. Це дуже складно виміряти кількісно, але, в принципі, масштабування робить позитивний вплив на ступінь стиснення інформації. Експерименти з масштабуванням в різних точках процесу стиснення показують, що ступінь стиснення сильно залежить від моменту масштабування ваги, але не існує правила вибору оптимального моменту масштабування для програми, орієнтованої на стиск будь-яких типів інформації.

Застосування.

Стиснення даних по Хаффману застосовується при стисненні фото-і відеозображень (JPEG, стандарти стиснення MPEG), в архіваторах (PKZIP, LZH та ін), в протоколах передачі даних MNP5 і MNP7.