University College Cork
Computer Science
Department

# Dynamic Programming (Independent study and implementation project)


Student Name: Sijie Guo

Student ID: 115107477

Supervisor: Professor Michel Schellekens

Second Reader: Dr. Steve Prestwich


B. Sc. Final Year Project Report

April 2017

# Abstract

This is an independent study and implementation project on Dynamic Programming for educational purposes, presenting a well-designed and multi-layered website. The aim is to provide users abundant, authoritative but concise and straightforward knowledge, including that of decomposed elements and components of dynamic programming algorithms. Six projects on dynamic programming, from naive to advanced, are worked out with analysis, optimization and implementation. Open source, self-implemented C++ codes provide users better learning experiences allowing them to code and test personally. Comparison between various approaches helps better understanding. Two algorithms, easily confused with Dynamic Programming— Divide and Conquer and Greedy Algorithms—are compared with Dynamic Programming via practical examples by execution time, code and space size. Greedy Algorithm and Dynamic Programming Algorithms are applied in the Shortest Path Problem, which enables users' comprehensive learning experiences of these algorithms.

Moreover, there is an investigation based on the well-known Levenshtein distance. To optimize the searching strategy and provide an accurate and unique solution, a weighted edit distance application is implemented with C# and ASP.NET. In order to give users better learning experiences, all the implementations are presented in a website with a Message Posting Board, supporting comments, suggestions and feedback.

Keywords: Educational website, Dynamic Programming, Greedy Algorithm, Divide and Conquer, Levenshtein Distance, self-learning, self-implementation, Software development, Research (Investigation).

Technologies: C++, C#, ASP. NET, HTML5, CSS3, JavaScript, JQuery, Introduction to Algorithms, Algorithm Design.

# Declaration of Originality.

In signing this declaration, you are confirming, in writing, that the submitted work is entirely your own original work, except where clearly attributed otherwise, and that it has not been submitted partly or wholly for any other educational award.

I hereby declare that:-

- this is all my own work, unless clearly indicated otherwise, with full and proper accreditation;

- with respect to my own work: none of it has been submitted at any educational institution contributing in any way towards an educational award;

- with respect to another's work: all text, diagrams, code, or ideas, whether verbatim, paraphrased or otherwise modified or adapted, have been duly attributed to the source in a scholarly manner, whether from books, papers, lecture notes or any other student's work, whether published or unpublished, electronically or in print.

Name: Sijie Guo

Signed: Sijie Guo

Date: 17/04/2017

# Acknowledgements

Heartfelt thanks to my father and mother.

Sincere appreciations to my supervisor, Professor Michel Schellekens, for his invaluable advice, constant encouragement and precise modification of such a challenging but significant project.

My gratitude to the following resources I studied and quoted in the project and in writing this thesis: Introduction to Algorithms (2nd and 3rd versions) for the most theoretical supports of algorithmic knowledge. Thanks to the technical supports from Furkanavcu and CodeProject.com for the open source code in the framework of the Levenshtein edit distance, the two templates- 2015 Free Slant and Strip Bootstrap 4.0 Theme-from cssmoban.com. Thanks to Jeff Erickson, Beck, Matthias, Geoghegan, Ross and N. J. A. Sloane, who explain the concept of Fibonacci numbers, S. K. Chang, A. Gill, Dexter Kozen and Shmuel Zaks for their research paper on Change-Making problem. Thanks to Dumitru whose publish Dynamic Programming - From Novice to Advanced inspires the structure of contents on the website.

Thanks to the Confusion matrix for spelling errors published by Stanford, which inspires the development of Manhattan weighted edit distance algorithm.

# Table Of Contents

# Part I Introduction

Dynamic Programming, widely applied in computer science, mathematics, economics and bioinformatics, is an algorithm for optimization. Dynamic Programming Algorithm solves a complex problem by breaking it down into a group of smaller subproblems and solving each of those subproblems just once, and storing their solutions with a memory-based data structure. When the same subproblem occurs again, instead of recomputing its solution, it just calls the previously computed solution. Thereby, it saves computation time at the cost of a modest expenditure in storage space. [1]

Dynamic Programming Algorithms, barely problems themselves, are much more used strategies for optimization. A Dynamic Programming Algorithm will look up the previously solved subproblems and will combine their solutions to give the optimal solution for the given problem. Therefore, based on different problems, different subproblems are constructed. Although it's working through the exponential scale of possible solutions to the given problem, it gets the optimal solution without ever examining each of them precisely. Since Dynamic Programming Algorithm balancing the effort of searching and solving subproblems delicately, it typically requests a reasonable amount of practice on this strategy to handle it expertly.

As this is an independent study and implementation project on Dynamic Programming, the most challenging part of this project is to decide what to implement and how to present the result. As mentioned above, Dynamic Programming Algorithm is used to optimize the given problems which contain the optimal substructure and overlapping subproblems, and it usually costs a lot of practice to handle this strategy. However, we expect to construct all the learning materials and implementations into one single project rather than a group of problems solved by Dynamic Programming Algorithm independently. To achieve this, an educational purpose website on

Dynamic Programming is built on self-learning and implementation materials and results. Instead of searching a substantial amount of materials on learning and losing in massive dynamic programming related exercises, we expected to comprehensively learn the Dynamic Programming Algorithm from basic concepts to challenging problem-solving tasks, until fully understood and capable of handling this strategy as a utility.

Organized into a multi-layered structure, the website is designed following the nature learning process; starts with explanations on elements and components of approaching dynamic programming strategy with decomposed steps, followed by hands-on exercises from naïve to advance. Each task consists of an introduction to the problem, analysis to break the problem into subproblems, design the optimal substructure, development of the pseudocode and implementation with C++.

As can be found in many textbooks, Dynamic Programming is always comparing with other algorithms. This is not just because there are similarities between Dynamic Programming and other algorithms, but also comparisons help learners to distinguish the differences between all these algorithms and make immediate decisions on which algorithm to choose based on the type of incoming problems. Therefore, there are also two web pages on comparisons of Dynamic Programming with Divide and Conquer and Greedy Algorithms, respectively. The comparisons are led by introduction, analysis, design, implementation and evaluation on well-known problems solved by Dynamic Programming and other algorithms.

There are quite a few questions can be placed into Dynamic Programming and Greedy Algorithm's properties. However, both algorithms present strength in different circumstances. One of the most cogent problems to apply and compare these two algorithms is Shortest Path Problem, which occupies a big portion in the website.

Apart from software development on the website, there are also two types of research on weighted edit distance base on the investigation of Levenshtein Distance. One

optimal model promises an accurate and unique solution is developed from the original model, while another one improves the original model to derive a relative accurate weight of substitution by calculating the Manhattan distance between selected pair of keys on the keyboard.

As highlighted before, this website targets for potential learners on Dynamic Programming. Thus, user's feedback counts a substantial amount of the evaluation of this website. Message Board works for posting sharable messages not only from the owner but also from users.

The outline of this thesis is as follows. In Part II we discuss the concepts of each algorithm in detail. In Part III we present the construction of the website from an overall view. In Part IV, we provide the implementation of the website as well as all the topics in four Chapters. In Chapter 1, there are five exercises on dynamic programming. In Chapter 2, there are two parts: Divide and Conquer and Greedy Algorithm compare with Dynamic programming, correspondingly. In Chapter 3, the Shortest Path Problem is derived in Single-Source and All-Pairs bases on different constraints on graphs, applied by Greedy Algorithm and Dynamic programming, separately. Finally, weighted edit distance is discussed in Chapter 4; Part V. Part VI delivers the Evaluation result of the survey. Part VII concludes this thesis and proposes the further research and implementation.

# Part II Analysis

The goal of this project is to construct an educational purpose website on Dynamic Programming. The website comprises of a main page and an elements and components page on Dynamic Programming. Inside, there are five projects for exercise, ordered by complexity: Cut Rod [2], Matrix Chain [3], Levenshtein Distance, Longest Common Sequence [4] and Optimal Binary Search Tree [5]. Each project is worked out with analysis, design, implementation and evaluation. These topics are quoted from Introduction to Algorithm [6], and the analysis and design part are paraphrased from the book. However, the code for each subject are self-implemented in C++ and evaluation of time and space complexity is also delivered bases on the execution result of the code. All above topics are separately embedded inside the website with an independent web page, and the detail of contents will be discussed further in Chapter 1.

To explore more related topics the drop-down table, which on the website, lists all the other algorithms. There are comparisons on Divide and Conquer and Greedy Algorithm with Dynamic Programming Algorithm, independently. Divide and Conquer and Dynamic Programming Algorithm will apply in solving Fibonacci sequence problem [7]. More details on analysis, implementation and evaluation of efficiency, code and space size will be provided in Chapter 2. Besides, we will solve coin problem individually by Greedy Algorithm and Dynamic Programming Algorithm and compare the code size and efficiency. However, as noticeable in particular cases greedy algorithm doesn't work, the result is also presented by testing cases in the evaluation section. Furthermore, Greedy Algorithm and Dynamic Programming are applied in Shortest Path Problem [8] on two different parts based on different properties of the graph; Single-Source Shortest Path Problem [9] on Bellman-Ford [10] and Dijkstra's Algorithms [11], and All-Pairs Source Problem [12]

on Floyd-Warshall [13] and Johnson's Algorithms [14]. This section can be found in Chapter 3.

Lastly, two weighted edit distance algorithms, which lead extra research models based on Levenshtein Distance, are implemented as project six on the main page. A supplementary application, implemented by C# and ASP.NET, was developed based on the original model [15] to evaluate the functionality of two models abreast. This application will contain flexible windows, inside which there are user type-in textboxes for original string and misspelling string and buttons for creating a table, refreshing data and computer distance. The table with grids will show the distance between each character with assigned weight, the final distance and the choice will present in the table with distinguishable color. The operations of insertion, deletion, matching and substitution are displayed in distance string textbox. Another application, developed by C++, applied Manhattan distance to find the relative accurate cost of substitution. In this algorithm, to improve transportability and confidentiality, instead of accurate distance, we would rather get a relative sense of Manhattan distance between each pair of characters in given strings. The content of these above research will be presented in Chapter 4.

Users' feedback, for information sharing, is posted on the Contact Page in Message Board, which enables users to post their comments and feedback on the website. All the posters will record locally with marked time and date, such that each time the website loaded, all the posters can be seen on the website. Survey are generated based on users' feedback. To reflect a pertinent user experience, two difference surveys are designed for Computer Science student and non-computer major students. The reason to do this is to get constructive suggestions from Computer Science students who focus much more on the practicability of the content, while comments on the user-friendliness of GUI and design from others. The original copy of surveys and users' feedback are attached in Appendix B. The statistics data was analyzed and

formed into graphs and tables, which are displayed on the Contact page in the website.

All the theories are excerpted from Introduction to Algorithm [6] for authority. Nonetheless, the content is reorganized, retrenched and demonstrated by the appropriate framework, with images, pseudocode and self-implement code. All above algorithms and topics are implemented in C++, and the implementation is exhibited as the open source code delivering inside the website to provide a synthetically learning experience.

Background and basic concepts:

Dynamic Programming Algorithm's elements and components

The common four steps [16] of approaching a dynamic programming strategy:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution with a bottom-up manner.
4. Construct an optimal solution from computed information.

Problems are applicable with Dynamic Programming Algorithm if they contain optimal substructure, which means optimal solutions to a problem exhibits optimal solutions to related subproblems, which we may solve independently. [17]

Dynamic Programming Algorithm applied to replace inefficient recursive solution as long as the problem incorporates optimal substructure. We can solve each subproblem once and store its solution for reuse when we meet the same subproblem again. In this way, instead of solving same subproblems repeatedly, we simply look up the storage table. If we have solved it before, then we call the solution directly, otherwise, we solve it and record it in a table. Dynamic Programming uses additional memory to save computation time; it serves as an example of a time-memory trade-off. [18]

There are normally two equivalent ways to implement a Dynamic Programming approach.

The first approach is top-down with memoization. [19] In this approach we write the procedure recursively, but save the solution of each subproblem. In each stage the procedure checks whether it has solved this subproblem previously. If so, recall the storage result, otherwise, compute it and save the result. We say that the recursive procedure has been memoized. [20]

The second approach is the bottom-up method. [21]

This approach typically relies on the "size" of a subproblem, such that solving any particular subproblem depends only on solving "smaller" subproblems, which known as its prerequisite subproblems. Thus, we sort the subproblems by size and solve them in increasing size order. In this case each subproblem only solves one, because we always have already solved all the smaller subproblems a particular subproblem solution depends upon.

Optimal substructure [22]

The common pattern in discovering optimal substructure:

1. Show that a solution to the problem consists of making a choice. Making this choice leaves one or more subproblems to be solved.

2. Suppose that you are given the choice that leads to an optimal solution for the given problem.

3. Given this choice, you determine which subproblems follow and how to best characterize the resulting space of subproblems.

4. Show that the solutions to the subproblems used within an optimal solution to the problem must themselves be optimal by deriving contradiction.

Various problems contain diverse optimal substructure in two ways:

1. How many subproblems an optimal solution to the original problem uses.

2. How many choices we have in determining which subproblem(s) to use in an optimal solution.

Overlapping subproblems [23]

When a recursive algorithm recomputes the same problem repeatedly we say that the optimization problem has overlapping subproblems. Dynamic Programming Algorithms typically take advantage of overlapping subproblems by solving each subproblem once and then storing the solution in a table. Then it can look up the table when needed in constant time.

Shortest Path Problem [8]

In graph theory, the shortest path problem is the problem of finding a path between two vertices in a graph such that the sum of the weights of its constituent edges is minimized.

Single-Source Shortest Path Problem [9]

Single-Source Shortest Path Problem finds shortest paths from a source vertex v to all other vertices in the graph.

All-Pairs Source Problem [12]

The all-pairs shortest path problem finds the shortest paths between every pair of vertices v, v' in the graph.

Technology:

The website is built based on existing website templates. In order to show the content systematically, the website is a combination of two templates with number of additionally developed features. One template is used for the main pages of the website while another template is used for the other algorithms comparing and associating with Dynamic Programming Algorithm. To guarantee a delightful User Experience on a concise and efficient website there are plenty of new features added

to the original template to fit in the presentation of the contents. Screenshot of each feature is provided in appendix to indicate its function.

1. Logo

Logo is important for a website. It's simple and memorable while identifying this website, impressing users and showing the professional of the website.

Therefore, a logo is designed for this website with the elements of Dynamic Programming Algorithm by Photoshop.

2. DROPDOWN Menu

Inside the dropdown menu there are the topics of other algorithms approaching: Divide and Conquer, Greedy Algorithm and Shortest Path Problems. All these topics are redirected to a new single webpage with a completely new framework implemented by CSS 3 and JavaScript, separately. Within shortest path problem, there are two parts: single-source and all-pairs shortest path problems.

3. Get Started Button

This button, which links the content note, is designed for downloading the note file from the corresponding page. In shortest path problems the buttons also redirect to new page of different algorithms inside the topics.

4. Responsive Web design

Whenever you click or place your mouse on an element in the website which is clickable, the color of the element will change.

5. All the contents and inside presentation

All the contents in the website on studying dynamic programming are designed and implemented apart from the original template.

6. Scrollbar for images

It is always straightforward and concise to present a problem or a solution by images. Therefore, for each topic, there is a scrollbar which presents as slides of images to show the expected implementation step by step.

7. Scrollbar for related topics

At the bottom of each page of the website there is a scrollbar with three topics which matches the three other algorithms in the dropdown menu to recommend the additional learning on those algorithms to users.

8. Contact page

Contact page is mainly for collecting and posting user's feedback. All the comments with time and date marks will be stored locally, which enables the website to load all the posters per time when loading the contact page.

Programming Language and templates:

The website's framework is constructed by HTML 5, CSS 3, JavaScript and JQuery. There are two templates applied in this project, "2015 Free Slant" [24] and "Strip Bootstrap 4.0 Theme" [25]. Both the templates are derived from online open source website templates. Since the goal of this project is to implement and share Dynamic Programming knowledge and code, the development on the website is mainly of front-end, which calls the CSS and JavaScript files in the templates to implement particular functions to present the content modestly. That is, to efficiently construct the educational website on Dynamic Programming, the self-learning and implementation contents are exhibited with the support of CSS and JavaScript files in the above templates.

The main implementation of the Dynamic Programming Projects was coded by C++. The code of each topic is developed on Visual Studio 2013 based on the pseudocode given in Introduction to Algorithm [6].

The Levenshtein Distance model [15] and the weighted edit distance model are implemented by C# and ASP.NET on Visual Studio in a Windows environment.

The application is built from existing online open source framework and windows. Thereby, two techniques- C# and ASP.NET- are newly learned for implementing

these two applications. The original model is developed with online open source code in CodeProject.com. The tables, windows and buttons are designed and formed in the framework. Note that the main purpose of this weighted edit distance is to analyze the property of weight of different operations- insertion, deletion and substitution- in order to give configurations to promise a unique and optimal solution. Thus, it is expediting to modify the original model to fit in our weighted model and test the correctness of our configurations.

# Part III Design

The structure of the project follows the standard learning process. Firstly, you learn the theoretical concepts of Dynamic Programming and do practical exercises to acquire the knowledge and coding skills in person. After you have the elemental sense of Dynamic Programming itself, you would like to compare it with other well-known algorithms to strike the comprehensive and eliminate confusions. Furthermore, applying this algorithm to optimize other algorithm problem strikes the learning outcomes. Moreover, when you have an adequate learning experience on Dynamic Programming Algorithm, you may be inspired to investigate on the exist algorithm to improve the accuracy and peculiarity of its solution.

The software development level design was based on the above process. The project presented as a website which enables users to find all the attractive topics, conducive learning materials and self-implemented C++ code quickly and naturally by following the guideline of the website design.

Dynamic Programming is the core of the project, which derives the pertinent topics, from naïve to advanced, Cut Rod, Matrix Chain, Levenshtein Distance, Longest Common Sequence and Optimal Binary Search. Each of the subjects is organized with an introduction of the problem, analysis to construct subproblem structure, implementation from pseudocode by C++ and evaluation of the test data. These different topics provide overall practice and challenge user to apply Dynamic Programming in various scenarios. Mainly, the Levenshtein Distance problem laid the foundation of the research model of weighted edit distance, which inherits the structure of the original model, but adds a new feature to the weight to obtain a unique and accurate solution.

To avoid confusion, there are other algorithms designed to compare with Dynamic Programming Algorithm. Divide and conquer is mainly focus on comparing the

recursive structure with top-down with memoization to present the optimal strategy Dynamic Programming issues. There compares on execution time on each algorithm and optimizations. Thereby, following libraries are involved in implementation:

<iostream>, <cstdio>, <map> // used in memoization to map the storage subproblem to the current problem

<cmath>// used in Formula calculation for Fibonacci sequence

<ctime>// used to present the execution time on each algorithm

Furthermore, Greedy Algorithm is compared with Dynamic Programming on coin problem, one of the typical NP problem. In the problem, Greedy strategy makes a quicker decision than Dynamic Programming due to its “greedy” property, but in particular cases, Greedy Algorithm fails to get the optimal solution because it only concerns local optimal regardless of the global optimum. The comparison details will expand in Chapter 2.

With a deeper understanding of Greedy Algorithm and Dynamic Programming, both algorithms are applied to the Shortest Path Problem base on different constraints of the graph. In Single-source shortest path problem, Greedy Algorithm is used in Dijkstra's algorithm, while Dynamic Programming algorithm is used in the Bellman-Ford algorithm. Moreover, Floyd-warshall algorithm as well as Johnson’s algorithm, which comprehensively combines all above algorithms, apply Dynamic Programming to solve all-pairs shortest path problems with matrice. This section not only introduces and distinguishes the different Shortest Path Problems but also profoundly compare Dynamic Programming with Greedy Algorithm in the practical Shortest Path Problem’s circumstance.

In Shortest Path Problems, there are several libraries involved:

"vector", "stack"// vector and stack are used in Floyd-warshall’s algorithm to storage the previous pair of vertices.

<limits>// use for pretreatment on constant, because Dijkstra's algorithm deals with non-negative weight. If the constant is negative, Dijkstra is not applicable.

The investigation - weighted edit distance as well as the original model- Levenshtein distance are developed into applications implemented by C# and ASP.NET. To develop an interaction application, there are several elements support it.

Flexible size windows contain all the following items:

1. User type-in Textboxes for original string and misspelling string.
2. Buttons for creating a table, refreshing data and computer distance.
3. Table with grids will show the distance between each character with assigned weight, the final distance and the choice will present in the table with distinguishable color.
4. Distance string textbox to display the four operations of insertion, deletion, matching and substitution.

Above features are implemented with these libraries:

Microsoft.CSharp, mscorlib, System,System. Core, System. Data, System. Data. DataSetExtensions, System. Deployment, System. Drawing, System. Windows. Forms,

System. Xml, System. Xml. Linq, System.Collections.Generic, System.ComponentModel, System.Text.

Commended in code, the framework of the application is generated automatically by above libraries. The weighted edit distance algorithm is built on the existing framework with above libraries by assigning a suitable weight to each operation in order to acquire pertinent configurations, which promise a unique and optimal solution.

For Manhattan weighted edit distance model, the following libraries are included in calculating the Manhattan distance of a pair of characters in given string such that issues a relatively accurate cost of substitution:

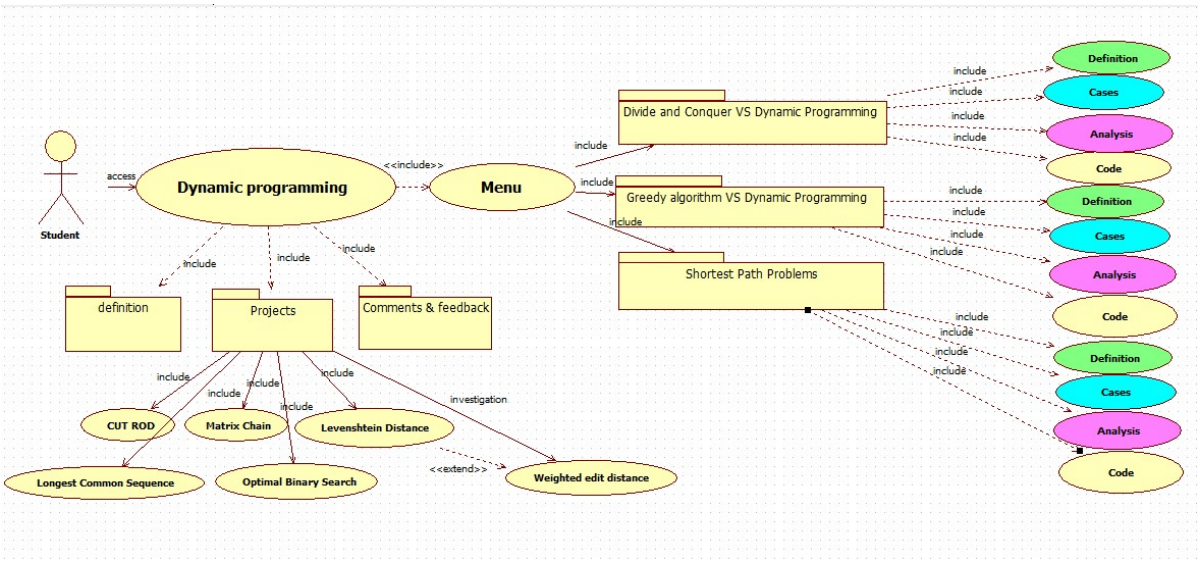
<iostream>,<stdio.h>, <stdlib.h>// to get size and get character from string

<time.h>, <string.h>, <math.h> // In order to calculate Manhattan distance between each keyword on the keyboard.

<ctype.h>// test char for category

As a final point, there is a contact page for user's feedback, which posts all previous users' comments every time loads the page. All the comments are stored locally and this is implemented by HTML5 codes embedded in the contact page.

The overall design can be presented by the UML diagram below.




Technology used in design:

Programming Language:

The website's framework is constructed by HTML 5, CSS 3, JavaScript and JQuery. The main implementation of the Dynamic Programming Projects was coded by C++. Although Java is the most popular language in the lecture teaching, labs, assignments and industry applying, C++ still has an indispensable position in Computer Science learning and research, especially algorithm designing and implementation. Although

algorithm implementation is not language based, which means there is no significant difference of implementation on an algorithm with different programming languages, undeniably, algorithms aim to optimize procedure. Compared to Java, C++ is much more a procedure-oriented language, which enables to design and explain the algorithm more accurately and much more straightforward to transform pseudocode to formal code. As C++ is much closer to a low-level language the code-generation process is simpler, optimized and faster, which is helpful to algorithm design. Meanwhile, STL makes C++ compelling in algorithm design, execution and evaluation.

There are several project implementation concerns which led me to use C++ instead of Java:

In this project, implementation and comparison of algorithms' performance play a vital role in an evaluation of time, space and efficiency. C++ code runs nearly two times faster than Java so that the assessment can be done more efficiently with C++. Furthermore, there are some online judges like SPOJ or COJ where some problems are unsolvable in Java due to time limits. Other problems are easily solvable in C++ with the correct algorithm while in Java you have to do a lot of optimization tricks to get accepted. Fibonacci sequence is another example where a problem requires algorithms hiding a big constant in complexity. They perform slowly in Java. Moreover, default stack size in Java is minuscule (between 64kb and 1024kb depending on the OS), when implementing recursive solutions you may get a Runtime Error only for this in Java this can be avoided using a Thread trick, but it is boring to code and easily forgotten. This is not a problem in C++.

One of the optimal strategies for Dynamic Programming is Memoization, and C++ has Direct Memory Access (DMA) which provides us with the capability to manage and optimize our memory usage as per our needs, and this is one of the most

important reasons why C++ is the preferred language by programmers. Also, Java provides a memory collector, which also makes it slower.

The Levenshtein Distance model and the weighted edit distance model are implemented by C# and ASP.NET on Visual Studio in a Windows environment. The application is built from existing online open source framework and windows. However, these two techniques- C# and ASP.NET- are newly learned for implementing these two applications. C# and ASP.NET developments are widely used in industry development of websites. Companies like Dell, EMC and Salesforces have many applications which are developed by C# and ASP.NET. From the real coding experience, there are several benefits from using these technologies:

1. C#, similar to Java, is a very mature and well-developed language with even more functionality. Like Java, it is a real object oriented language; only it is geared for enterprise level development.
2. The .Net framework is a robust open source framework that is actively maintained by Microsoft. Therefore, it is flexible and stable. ASP.NET dramatically reduces the overall code size required to build large applications.
3. Visual Studio is arguably the best IDE in the world, which has excellent code completion and generation capabilities.
4. Windows environment. Visual Studio /.Net are native for Windows environment. With built-in Windows authentication and per-application configuration, these applications are safe and secured.
5. .Net has enormous user base because of a long time it has been used and a lot of developers, all over the world, know how to use it.
6. The ASP.NET framework is complemented by a powerful toolbox and designers in the Visual Studio integrated development environment.

As the project is for educational purposes, a range of technologies and programming languages are used which highlights this aspect of the project, which tot only enables followers to pick up academic knowledge in wide range algorithm topics but also provides the opportunity to access, practice and compare the different uses of several popular programming languages.

# Part IV Software Development

The project presented as an educational website to share the learning materials and implemented codes on Dynamical Programming Algorithm.

The primary implementation of the website is on front-end based on an open source template. All the contents in the web pages are self-implemented. There are two temples used in this project; one is for the main website to present the relevant knowledge, self- learning materials and implementation on several topics and another is for the comparisons between Dynamic Programming Algorithm and other Algorithms such as Divide and Conquer and Greedy Algorithms.

The details of the contents of the main website will be elaborated in Chapter 1 Dynamic Programming Projects, and Chapter 3 Apply Dynamic Programming in Shortest Path Problem. The details of comparisons will be expanded in Chapter 2 Compare Dynamic Programming with other algorithmic approaches.

For a self-learning and self-implementation project on Dynamic Programming, the intricate part is to design and organize what to achieve in the project and how to present the results. As we know, Dynamic Programming Algorithm is a problem dependent algorithm. That is, for different problems, it contains a different substructure to optimize and construct an overall optimal solution based on optimal solutions of subproblems. Therefore, to get familiar with the procedure of Dynamic Programming, a significant amount of practice is inevitable. However, as discussed, if we study into a number of independent problems, we will end up with a group of projects instead of one. In order to organize the learning materials, practicing process and associated topics, we construct an educational website with above contents.

Although there are several topics accessed in teaching courses, such as Levenshtein distance, Fibonacci sequence problem, Coin problem, Bellman-Ford algorithm and Dijkstra's algorithm, other topics are freshly learned and implemented in this project.

Notably, several problems solvable by Dynamic Programming Algorithm were learned in the previous lecture, however, without systematically studying on Dynamic Programming Algorithm itself, the learning process is a significantly challenging part of this project. To master Dynamic Programming, an amount of independent learning was applied to understand the algorithm theoretically and practically. Rephrasing each topic in notes, website as well as in the report leads to a better and deeper understanding of Dynamic Programming theoretically, while coding on each topic in C++ contributes to comprehend the algorithm practically.

The website is coordinated with the natural learning process: when you load the main page, if you already know Dynamic Programming Algorithm, you can view those problems solvable by Dynamic Programming Algorithm and start to practice, or the other topics that you are interested. To start with, we assume that you are a user without any Dynamic Programming background knowledge and expect to learn from the beginning. In this case, you will approach the website in the following order.

Load the website you will see the ‘get started’ button, which will lead you to the ‘elements’ page, where you are going to learn the concepts, steps of approaching Dynamic Programming and ways to optimize the solution.

After you have acquired sufficient knowledge of Dynamic Programming itself, you may apply the algorithm to practical problems to get familiar with the way to solve those problems with Dynamic Programming. From easy to hard, there are five problems available to practice with: Cut-Rod, Matrix-Chain, Levenshtein Distance, Longest Common Sequence and Optimal Binary Search Tree problems. The details of each problem will be explored in Chapter 1. Note that, each problem is presented with an individual page where comprises of problem introduction, analysis, design, implementation and evaluation. All the contents, images and codes are self-developed. At the bottom of each page, there are three scroll bars to remind you to explore other algorithms related to Dynamic Programming.

If you are interested in other algorithms and topics, the arrow will lead you directly to the top of the page where there is a DROPDOWN table contains two links. One link guides of an individual web page of comparison between Divide and Conquer Algorithm and Dynamic Programming, while another conducting to the web page of comparing Greedy Algorithm with Dynamic Programming. All the contents, images and codes on the web pages are self-developed. The comparisons' details are covered in Chapter 2 including examples, analysis, experiential testing results and codes.

After learning Greedy Algorithms and Dynamic Programming Algorithm, we employ both algorithms to solve Shortest Path Problems who contain single-source and all-pairs two categories problems. In single-source shortest path problems, we are going to study Bellman-Ford algorithm and Dijkstra's algorithm. In all-pairs shortest path problems, we typically apply Floyd-warshall algorithm for dense graphs, while Johnson's algorithm for sparse graphs. Each topic is presented on an individual web page. We will learn the details of the Shortest Path Problem in Chapter 3. Within these web pages, all the contents, images and codes are self-developed.

Finally, you may notice that there is a contact page where you can post your comments and messages to share with other users. The messages are stored locally on the page, and all the previous posters will display each time the contact page loaded. This is accomplished with HTML code embedded in the contact page.

With the well-designed and well-organized website, a user can access to immediate information they expect to view and learn. For educational purpose, everyone can access the materials, view self-implemented code on each algorithm and share their ideas with anyone else.

# Chapter 1 Dynamic Programming Projects

## 1.1 Introduction

In Chapter one, five individual dynamic programming projects, each contains an introduction, analysis, design, implementation and evaluation, are ordered by the complexity. This section is worked for practicing the approaching procedure of dynamic programming.

All the above contents are involved in the website on the main page and Elements and Componence page.

## 1.2 Cut-Rod Problem [2]

### 1.2.1 Introduction

The cut-rod problem is to find the best solution of cutting a rod into different length to optimize the overall benefits. We will give the price of varying lengths of rods, and we want to maximize the total revenue of selling rods.

### 1.2.2 Analysis

To find the best way to cut up the rod, we need to define two variables: the price for each length of a rod as well as the length of the given rod.

In the problem, we are given the price p[i] (for i = 1, 2 …) in euro that charge of each rod of length i feet. We are also given the total length n of the rod, which is an integral number of feet.

With the information about the price and the length of a rod, we can determine the maximum revenue r[n].

Observably, if the price p[n] of the given length n of the rod is large enough, the optimal solution should be no cutting at all.

Since we have an independent option of cutting or not, at distance i feet from the left end, we can cut up a rod of length n in $2^{n-1}$ different ways. Instead of testing all the possible solutions, we can solve the above problem by Dynamic Programming.

1.2.3 Design

Dynamic Programming structure design:

1. Characterize the structure of an optimal solution:

Apparently, we can break the original problem of size n into smaller problems with same structure, then we solve the smaller but related subproblems to obtain our best solution. Practically, once we make the first cut, we treat the two pieces of rods as independent subproblems to solve in the next stage. To achieve the overall optimal solution, we maximize and incorporate revenue from each of those two related subproblems

2. Recursively define the value of an optimal solution.

Note that once we make the first cut, we attain a decomposition consists of a left-hand end of the first piece of length i, and a right-hand remainder of length n-i. The further division will apply only to the remainder. Therefore, we can recursively view each decomposition of the length-n rod as a first piece followed by some decomposition of the rest. Initially, we have the first piece with size i=n, and revenue p[n] and the remainder has size 0 with corresponding revenue r [0] =0.

Consequently, we obtain a recursive formulation of an optimal solution embodies the solution to only one related subproblem—the remainder, rather than two:

$r[n] = \max_{1 \le i \le n} (p[i] + r[n-1])$

The pseudocode of recursion structure (top-down)

```
CUT-ROD（p， n）
if n == 0
return 0
r = negative infinity
for i =1 to n
    r = max（r, p[i]+CUT-ROD(p, n-1))
return r
```

Top-down with memorization:

We usually use memorization strategy to optimize the searching space of the problem. When doing so, we store the result of each subproblem. At each stage, the procedure first checks whether it has previously solved this subproblem. If so, the method returns the saved value, saving computation time and space at this level; otherwise, it computes and saves the value in the usual manner. Memorized, as we say, it records all the previous value it has calculated in the recursive procedure.

The pseudocode of top-down with memorization:

```
MEMOIZED-CUT-ROD (p, n)
let r [0...n] be a new array
for i =0 to n
    r[i] =negative infinity
    return MEMOIZED-CUT-ROD-AUX (p, n, r)
MEMOIZED-CUT-ROD-AUX (p, n, r)
if r[n] >= 0
return r[n]
if n == 0
q=0
else q=negative infinity
```

```
for i =1 to n
    q = max.(q, p[i]+ MEMOIZED-CUT-ROD-AUX(p, n-i, r))
    r[n]=q
    return q
```

3. Compute the value of an optimal solution, typically in a bottom-up fashion.

This approach naturally depends on the "size" of a subproblem, such that solving any particular subproblem depends only on solving smaller subproblems which it relies on. We sort and resolving the subproblems by size in increasing order. Thus, when solving a particular subproblem, we have already solved and saved all of the smaller subproblems previously, which its solution depends upon. With the supports of all the prerequisites subproblems, which we have already solved, we solve each subproblem only once at the first time we see it.

The pseudocode of bottom-up fashion:

```
BOTTOM-UP-CUT-ROD (p, n)
let r[0...n] be a new array
r[0]=0
for j =1 to n
  q= negative infinity
    for i =1 to j
        q =max(q, p[i]+r[j-i]
        r[j]= q
return r[n]
```

4. Construct an optimal solution from computed information.

The Dynamic Programming solutions to the rod-cutting problem return the value of an optimal solution, rather than a list of piece sizes. We can extend the Dynamic

Programming approach to record and return not only the optimal value of each subproblem but also a choice that led to that value. To achieve this, we construct an extended version of BOTTOM-UP-CUT-ROD that computes, for each rod size j, not only the maximum revenue r[j] but also s[j], the optimal size of the first piece to cut off. In this case, we can readily print an optimal solution

The pseudocode of extended- bottom-up

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
let r[0...n] and s[0...n] be new arrays
r[0]=0
for j = 1 to n
    q = negative infinity
    for i =1 to j
        if q < p[i]+_r[j-i]
                q =p[i]+r [j-i]
                s[j]= i
     r[j]= q
return r and s

PRINT-CUT-ROD-SOLUTION (p, n)
(r; s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
while n > 0
    print s[n]
    n =n-s[n]
```

### 1.2.4 Implementation

All the above pseudocode are quoted from Introduction to Algorithms [6]. The code of the rod - cutting problem is implemented based on the pseudocodes in the design

part with C++. There are four methods: Rod, CutRod, CutRodMemoized, BottomUpCutRod and printTimes (). Rod method views all the initializations of variables and memories array space as private data. Inside, there also contains ~Rod () to release space to optimize the overall spaces. Two macro definitions: INT_MIN, the minimum integer value, and NULL, defined as zero, were referred from the library: limits. h includes <crtdefs. h>.

Three approaching strategies are involved in this problem: normal recursion, top-down with memoized and bottom up. In the standard recursion method, we select the maximum price value in each stage. To compare the current revenue, with the largest previous value, we use _max (a, b), which defines as (((a) > (b)) ? (a): (b)) in stdlib. h library. In the CutRodMemoized method, we return the optimal solution that is calculated based on previous calculation and the records of related subsections. In BottomUpCutRod, we sorted and calculated the subproblems with increasing size. Each particular subproblem, it only calculates once, calls for related subproblems it depends on. Finally, there is a printTimes() to calculate and present the calculation times in each of those three methods. The results are printed in the main function to compare the optimization of dynamic programming from the normal recursion.

1.2.5 Evaluation

Cut rod problem is to find the optimal procedure of cutting the rod to maximize the total price.

The code is implemented by recursion, top-down memorization and bottom up. There are time methods to measure execution time for each method.

In the example of “Introduction to Algorithms”, the prices of the rod were assigned to {0, 1, 5, 8, 9, 10, 17, 17, 20, 24, 30} and the rod length is 8.

| Length | i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Price | P[i] | 1 | 5 | 8 | 9 | 10 | 17 | 17 | 20 | 24 | 30 |

In the testing result, the maximize price is 22, and the length of the cut rods are two and six.

Besides, from the execution time, it is easy to tell that recursion costs the largest time, which is about 83 times than Dynamic Programming strategy, while top-down and bottom-up Dynamic Programming solutions require the same time as 36.

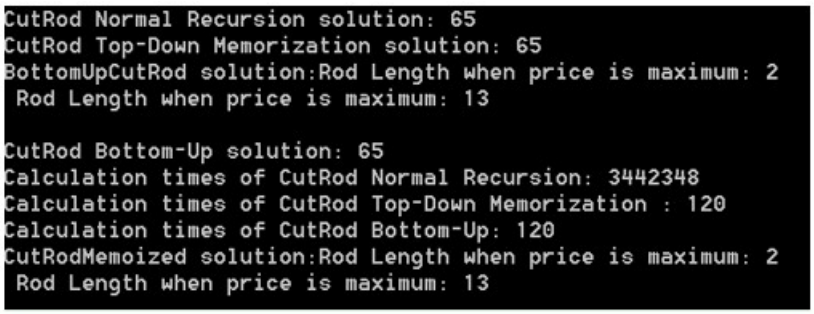
```
CutRod Normal Recursion solution: 22
CutRod Top-Down Memoized solution:22
BottomUpCutRod solution:Rod Length when price is maximum: 2
 Rod Length when price is maximum: 6

CutRod Bottom-Up solution:22
Calculation times of CutRod Normal Recursion: 2973
Calculation times of CutRod Top-Down Memoized: 36
Calculation times of CutRod Bottom-Up: 36
CutRodMemoized solution:Rod Length when price is maximum: 2
 Rod Length when price is maximum: 6
```

For a larger testing data: the prices of rod were assigned to {0, 1, 5, 8, 9, 10, 17, 17, 20, 24, 30, 35, 50, 60} and the rod length is 15.

| Length | i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Price | P[i] | 1 | 5 | 8 | 9 | 10 | 17 | 17 | 20 | 24 | 30 | 35 | 50 | 60 |

In the testing result, the maximize price is 65 and the length of the rod is two and 13.

Besides, from the execution time, it is easy to tell that recursion costs the largest time, which is about 28686 times than dynamic programming strategy, while top-down and bottom-up dynamic programming solutions cost the same time as 120.




Therefore, when the number keeps growing, dynamic programming can save time and space to an enormous extent.

Analysis of time and space complexity:

Recursive structure:

The execution time is $T(n) = 2^n$ and so the running time of the CUT-ROD is exponential in n. As for space, we can construct a recursion tree for this recursion strategy, and there will be $2^n$ nodes, and $2^{n-1}$ leaves because each cut ends up with two pieces of rods.

Top-down with memorization:

The running time of its top-down counterpart, MEMOIZED-CUT-ROD, is $\theta(n^2)$.

Extended bottom up:

The running time of its Bottom-Up counterpart, MEMOIZED-CUT-ROD, is $\theta(n^2)$.

Both top-down and bottom-up fashion leads a $\theta(n^2)$running time due to the double neat for loops.

To sum up, dynamic programming algorithm reduces the execution time of rod-cutting problem from exponential to polynomial.

## 1.3 Matrix-Chain Multiplication Problem [3]

### 1.3.1 Introduction

We state the matrix-chain multiplication problem [3]as follows: given a chain (A1, A2... An) of n matrices, where for i=1,2,...,n, matrix Ai has dimension pi-1 * pi fully

parenthesize the product A1A2... An in a way that minimizes the number of scalar multiplications.

Note that the analysis and design parts are rephrased from Introduction to Algorithms [6].

### 1.3.2 Analysis

We can multiply two matrices A and B if and only if they are compatible. That is, the number of columns of A must equal the number of rows of B. If A is a p * q matrix and B is a q* r matrix, the resulting matrix C is a p * r matrix. The time to compute C is determined by the number of scalar multiplications, which is p*q*r.

As matrix multiplication is associative, all parenthesizations yield the same structure. A product of matrices is fully parenthesized if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses.

Note that in the matrix-chain multiplication problem, our goal is not actually multiplying matrices, but only to determine an order for multiplying matrices that leads to least cost of calculation. Typically, the time devoted to determining this optimal order will pay for by the time saved later on when actually performing the matrix multiplications. That is, we find the optimal way to multiply matrices, which will lead to a time-saving multiplication process.

Counting the number of parenthesizations

Indicate the number of alternative parenthesizations of a sequence of n matrices by P(n). $P(\mathrm{n}) = \begin{cases} 1 & if\ n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if n} \geq 2. \end{cases}$

$$P(n) = \Omega\left(\frac{4^{\mathrm{n}}}{\mathrm{n}^{\left(\frac{3}{2}\right)}}\right).$$

In the above formulation, the number of solutions is thus exponential in n, and the brute-force method of exhaustion turn out to be a poor strategy when determining

how to parenthesize a matrix chain optimally. As a result, we use dynamic programming to optimize this problem.

### 1.3.3 Design

Step 1: The structure of an optimal parenthesization

By evaluating the product AiAi+1...Aj, we adopt the notation Ai ... j for the matrix, where i <= j. Note that if the problem is complicated, then to parenthesize the product AiAi+1 ... Aj, we must split the product between Ak and Ak +1, where k is in the range i <=k < j. Specifically, for some value of k, we split the product into matrices Ai...k and Ak+1...j, compute each matrix separately and then multiply them together to yield the final product Ai ...j. In this way, the overall cost of parenthesizing is the incorporation of the cost of computing the matrix Ai... Ak, the cost of computing Ak+1...Aj, and the cost of multiplying them together. The “prefix” subchain AiAi+1...Ak must be an optimal parenthesization within this optimal parenthesization of AiAi+1...Aj.

Now we can construct an optimal solution to the problem from subproblems' optimal solution. That is, we can build an optimal solution to the matrix-chain multiplication problem by dividing the problem into two subproblems (AiAi+1 … Ak and Ak+1Ak+2…Aj), finding optimal solutions to subproblems, and then combining these solutions of optimal subproblem.

Step 2: A recursive solution

For the matrix-chain multiplication problem, our subproblems are the problems of determining the minimum cost of parenthesizing AiAi+1...Aj for 1 <= i <= j <= n. Let m [i, j] be the minimum number of scalar multiplications needed to compute the matrix Ai ...j. Note that, if i = j, m[i, j]=0, if i < j, m [i, j] =m [i, k] +m [k+1, j] +pi-1*pk*pj. The procedure is based directly on the recurrence.

Pseudocode of RECURSIVE-MATRIX-CHAIN:

```
RECURSIVE-MATRIX-CHAIN (p, i, j)
if i == j
return 0
m[i, j]= negative infinity
for k = i to j - 1
    q = RECURSIVE-MATRIX-CHAIN(p, i, k) +
RECURSIVE-MATRIX-CHAIN(p, k + 1, j )+pi-1 * pk * pj
    if q < m[i, j]
        m[i, j]=q
return m[i, j]
```

Step 3: Computing the optimal costs

Note that we have a few distinct subproblems: one subproblem for each choice of i and j satisfying 1 <= i <= j <= n. or $n^2 + n = \theta(n^2)$ in total.

Instead of recursively computing the solution of recurrence, we compute the optimal cost by using a tabular, bottom-up approach in the procedure MATRIX-CHAIN-ORDER.

The procedure uses an ancillary table m [1… n, 1… n] for storing the m [i, j] costs and another accessory table s [1… n-1, 2… n] that records which index of k achieved the optimal cost in computing m [i, j]. The auxiliary table s will construct an optimal solution.

Pseudocode of matrix-chain-order:

```
MATRIX-CHAIN-ORDER (p)
n =p.length-1
let m[1... n,1... n] and s[1... n -1,2 ...n] be new tables
for i =1 to n
    m[i, i] =0
```

```
        for L=2 to n // L is the chain length
                for i = 1 to n - L + 1
                        j = i + L -1
                        m[i, j] =negative infinity
                        for k = i to j - 1
                                q = m[i, k]+m[k+1,j] + pi-1 * pk * pj
                                if q < m[i,j]
                                        m[i, j] =q
                                        s[i, j] =k
return m and s
```

Memoization often offers the efficiency of the bottom-up dynamic programming approach while maintaining a top-down strategy.

Pseudocode of MEMORIZED-MATRIX-CHAIN

```
MEMORIZED-MATRIX-CHAIN (p)
n =p.length-1
let m[1... n,1... n_]be a new tables
for i =1 to n
    for j = i to n
        m[i, j] = negative infinity
return LOOKUP-CHAIN(m, p, 1, n)

LOOKUP-CHAIN(m, p, i, j)
if m[i, j] < infinity
return m[i, j]
if i == j
```

```
m[i, j] = 0
else for k = i to j - 1
      q = LOOKUP-CHAIN(m, p, i, k)+ LOOKUP-CHAIN(m, p, k +1, j)+ pi-1 * pk *
pj
    if q < m[i,j]
          m[i, j]=q
return m[i, j]
```

Step 4: Constructing an optimal solution

The following recursive procedure prints an optimal parenthesization of (Ai, Ai+1, ..., Aj), given the s [i, j] table computed by MATRIX-CHAIN-ORDER. The initial call PRINT-OPTIMAL-PARENS(s, 1, n) prints an optimal parenthesization of (A1, A2,..., An).

```
PRINT-OPTIMAL-PARENS(s, i, j)
if i==j
     print “A”i
else print “(”
     PRINT-OPTIMAL-PARENS(s, i, s [i , j])
PRINT-OPTIMAL-PARENS(s, s [i , j] + 1, j)
     print “)”
```

### 1.3.4 Implementation

All the above pseudocode are quoted from Introduction to Algorithms [6]. The code of the Matrix Multiplication problem is implemented by C++ follows the pseudocode in Design part. Two versions of methods: Recursion with Memorization and dynamic programming strategy- Bottom Up. In the class of the Matrix, there are two

dimensions matrix m, which is the original matrix and matrix s, which is the optimal matrix of the solution. We define times_topdown and times_bottomup to accumulate the calculation time of top-down and bottom-up strategies, respectively.

In the RecursiveMatrixMulti method, we recursively insert k to each position between i and j in producing Ai…j to find the optimal solution of parenthesization of (Ai, Ai+1, ..., Aj). We use table m to record the current matrix chain and table s to record the parenthesization position of k.

In the MatrixMulti method, which applies dynamic programming bottom-up strategy, we have a similar structure as memorization approach. We also store the current matrix chain and optimal matrix chain in table m and s individually for future use. Typically, at each stage, we only have two subproblems of the matrix: Ai…k and Ak …j in produce Ai, Ai+1… Aj.

We calculate the calculation times in each method and print the solution matrix chain with parenthesis. We use Matrix () to release the space of two matrices m and s.

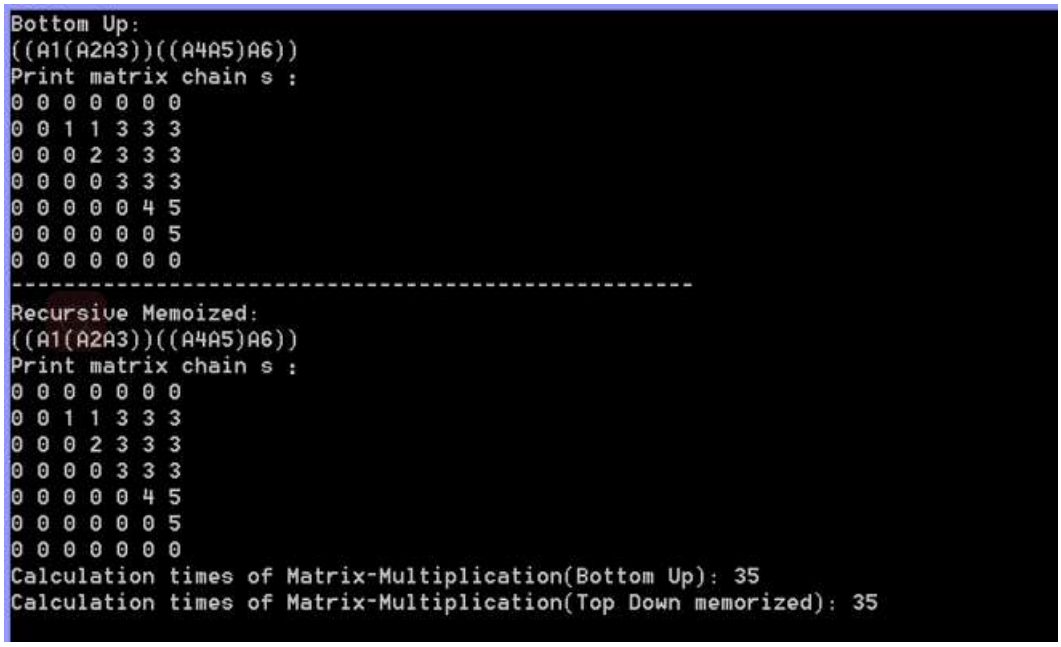
The most significant difference between the memorization and bottom-up strategies is the former one calculate store the optimal solution at each stage, while the later one narrows the number of subproblems to two and always applies optimization on current subproblems.

### 1.3.5 Evaluation

Matrix Multiplication is tending to find the optimal solution for a given matrix chain on multiplication (to decide any group of matrices in given order should multiply together).

In the example of “Introduction to Algorithms” the original matrix chain is {30, 35, 15, 5, 10, 20, 25}. The optimal solution matrix multiplication is ((A1 (A2A3)) ((A4A5) A6)).

The algorithm is implemented by dynamic programming in both bottom up and recursive memorization.




Compare recursion and dynamic programming on Overlapping subproblems:

As there will be n possible positions for parenthesization and for each position, there is an individual decision of inserting parenthesis or not. Therefore, recursion strategy will obtain $O(2^n)$ searching time.

The total amount of work performed by the call RECURSIVE-MATRIX-CHAIN (p, 1, n) is at least exponential in n. For both Memoized version of RECURSIVE-MATRIX-CHAIN and Bottom-up version of MATRIX-CHAIN-ORDER, there is triple nested for loops and each loop index (l, i, and k) takes on at most n-1 values. Therefore, the time complexity of both algorithms is $O(n^3)$, while the space complexity is $O(n^2)$ to store the two dimensions' m and s tables.

In conclusion, we can solve the matrix-chain multiplication problem by either a top-down, memoized algorithm or a bottom-up dynamic programming algorithm in

$O(n^3)$ execution time. Both methods take advantage of the overlapping-subproblems property. Totally, there are only $O(n^2)$ distinct subproblems, and either of these methods computes the solution to each subproblem only once. Without memoization, the normal recursive algorithm runs in exponential time which nearly $O(2^n)$, since subproblems are repeatedly solved.

## 1.4 Levenshtein Distance

### 1.4.1 Introduction

The edit distance between two words, also known as the Levenshtein distance, is to calculate the minimum number of letter insertions, letter deletions, and letter substitutions required to transform the source word into the target word.

Given two sequences s [1 ...m] and t [1...n] and set of transformation-operation costs, the edit distance from s to t is the cost of the least expensive operation sequence that transforms s to t. The Levenshtein Distance problem can be stated as a dynamic-programming algorithm that finds the minimum distance from s [1 ...m] to t [1...n] and prints an optimal operation sequence.

### 1.4.2 Analysis

Levenshtein distance, abbreviating as LD, is a measure of the similarity between two strings; the source string (s) and the target string (t). The distance is the accumulation of the amount of deletions, insertions, or substitutions required to transform string s into string t. For instance,

If s is "cat" and t is "cat" as well, then LD(s, t) = 0, since the strings are already identical and no transformations are required.

If s is "cat" and t is "cut", then LD(s, t) = 1, because one substitution (change "a" to "u") is sufficient to transform s into t.

The greater the Levenshtein distance, the more different the strings are.

The Levenshtein distance algorithm applies in an extensive range of fields, such as: Spell checking, Speech recognition, DNA analysis, Plagiarism detection.

### 1.4.3 Design

Step 1: Characterize the structure of an optimal solution.

To solve the Edit Distance problem by Dynamic Programming Algorithm, we need to develop a recursive definition. Our distance representation of edit sequences has a significant "optimal substructure" property. Assume that we have the distance representation table in the shortest edit sequence of two strings. If we remove the last column from the table, the remaining columns have to reproduce the shortest edit sequence for the remaining substrings.

Step 2: Recursively define the value of an optimal solution.

Once we figure out what should present in the last column, the recursion structure will instantly give us the rest of the optimal distance representation.

Thus, we recursively define the edit distance between two strings s[1..m] and t[1.. n], which we denote by Edit(s[1..m], t[1.. n]). If neither string is empty, there are three possibilities for the last column in the shortest edit sequence:

Insertion: When the last entry in the bottom row is empty, the edit distance is equal to Edit (s [1…m -1], t [1… n]) + 1. The 'plus 1' is the cost of the final insertion, and the recursive expression gives the minimum cost for the other columns.

Deletion: When the last entry in the top row is empty. In this case, the edit distance is equal to Edit (s [1...m], t [1... n - 1]) + 1. The 'plus 1' is the cost of the final deletion, and the recursive expression gives the minimum cost for the other columns.

Substitution: If both rows have characters in the last column. If the characters are the same, the substitution is free, so the edit distance is equal to Edit (s [1 ...m-1], t [1 ... n-1]), and we call this case as 'match'. If the characters are different, then the edit distance is equal to Edit (s [1 ...m-1], t [1 ... n-1]) +1.

The edit distance between s and t is the smallest of these three possibilities:

$$\text{Edit}(s[1...m], t[1...n]) = \min \begin{cases} \text{Edit}(s[1 \ldots m-1], t[1 \ldots n]) + 1 \\ \text{Edit}(s[1 \ldots m], t[1 \ldots n-1]) + 1 \\ \text{Edit}(s[1 \ldots m-1], t[1 \ldots n-1]) + 1 \end{cases}$$

This recurrence has two base cases. One is to transform the empty string into a string of n characters, which is achieved by performing n insertions. The other case is to transform a string of n characters into the empty string, which requires n deletions. Thus, if ε denotes the empty string, we have Edit (s [1 ...m], ε) = m, Edit (ε, t [1 ... n]) = n.

Both of these definitions imply the base case Edit (ε,ε) = 0.

Step 3: Compute the value of an optimal solution.

Note that the arguments to our recursive subproblems are always prefixes of the original strings s and t. Hence, we can simplify our notation by using the lengths of the prefixes. That is, we use notation Edit (i, j) to replace Edit(s[1 .. i], t[1 .. j]). This function satisfies the following recurrence:

$$\text{Edit}(i, j) = \min \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{cases} \text{Edit}(i-1, j) + 1 \\ \text{Edit}(i, j-1) + 1 \\ \text{Edit}(i-1, j-1) + 1 \end{cases} & \text{otherwise} \end{cases}$$

The edit distance between the original strings s and t is just Edit (m, n). This recurrence translates directly into a recursive algorithm; the precise running time is not evident, but it's clearly exponential in m and n. Practically, we don't care about the precise running time of the recursive algorithm. The recursive running time has none influence on our eventual dynamic programming algorithm so that we won't

compute it. Since each recursive subproblem can be identified by two indices i and j, we can memoize intermediate values in a two-dimensional array Edit [0 ...m, 0 ...n]. Each entry Edit [i, j] depends only on its three neighboring entries, Edit [i-1, j] +1, Edit [i, j-1] +1, and Edit [i -1, j - 1]. If we fill in our table row by row top-down, each row from left to right, then whenever we reach an entry in the table, the subentries it depends on have already computed.

Assembling all the configurations, we obtain the following Dynamic Programming Algorithm:

```
EDITDISTANCE (s [1...m], t [1...n]):
for j <-1 to n
    Edit (0, j) <- j
for i <-1 to m
    Edit (i, 0) <- i
    for j <- 1 to n
        if s[i] = t[j]
        Edit (i, j) <- min [Edit (i -1, j) +1, Edit (i, j -1) +1, Edit (i -1, j -1)]
        else
        Edit [i, j] <- min [Edit (i -1, j) +1, Edit (i, j -1)+1, Edit (i -1, j -1] +1]
return Edit (m, n)
```

### 1.4.4 Implementation

Firstly, we set the source and target to be the length of source string s and target string t correspondingly. If n = 0, return m and exit, while if m=0, then returns n and exit. Then, we construct a matrix: matrix_table [source + 1, target + 1] with source +1 rows and target +1 columns. Then, initialize the first row to 0 to target + 1 and initialize the first column to 0 to source +1. We examine each character of source

string s (i from 1 to source + 1) and target string t (j from 1 to target + 1). If s [i - 1] equals t [j - 1], then the cost is 0, otherwise, the cost is 1. To construct the table, we assign cell matrix_table[i, j] of the matrix equal to the minimum of the following three scenarios:

The grid immediately above plus 1: matrix_table[i-1, j] + 1.

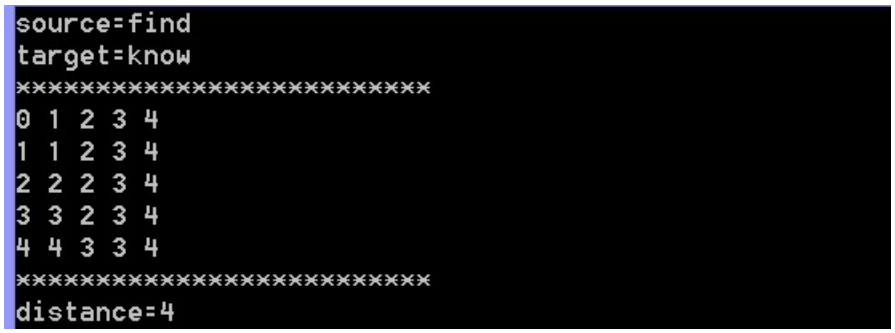
The grid immediately left plus 1: matrix_table[i, j-1] + 1.

The grid diagonally above and to the left plus 1: matrix_table[i-1, j-1] + 1.

Finally, we find and return the distance determined in cell matrix_table[i, j] and print the table on the screen.

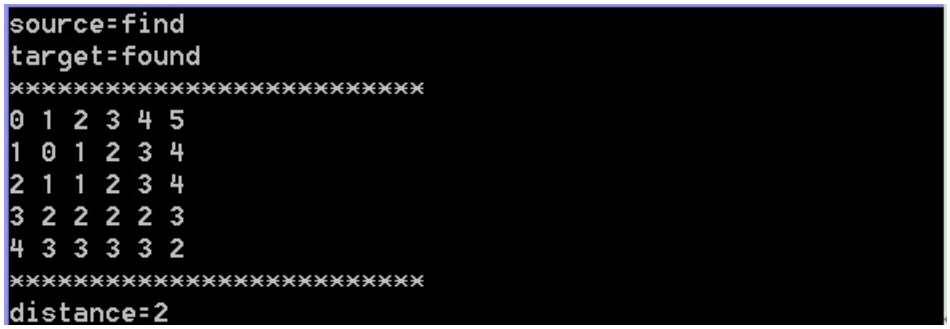
### 1.4.5 Evaluation

We test our Levenshtein Distance problem with the source string ‘find’ and target string ‘know’ which are entirely different sources so that we expect the distance between those strings to be 4, which is the cost of substitution on each character in the source string to the target string.




We test our Levenshtein Distance problem with the source string ‘find’ and target string ‘found’. In source string, there are three characters same as target string so that we expect the distance between those strings to be 2. However, there will be multiple ways to transform source string to target string. Intuitionally, for instance, to change ‘find’ to ‘found’, you can leave ‘f’, ‘n’ and ‘d’ unchanged and then delete ‘i’ and

insert ‘o’ and ‘u’. Also, you can leave ‘f’, ‘n’ and ‘d’ unchanged while substituting ‘i’ to ‘o’ (or ‘u’) and inserting another character ‘u’ ( or ‘o’).




Note that the indices source and target, which are the length of string s and string t correspondingly, ranges start at zero to adopt the base cases. Since there are (mn) entries in the table, our Dynamic Programming Algorithm uses (mn) space. As each entry in the table can be computed in (1) time if we know its predecessors, our Dynamic Programming Algorithm runs in (mn) time. Besides, backtrack time is O (m + n), which is in linear time.

As all the costs of different operations are same, we will get the final distance with differing approaches sometimes. However, sometimes we prefer particular operation(s) much more than others, in this scenario, we will introduce a new algorithm - weighted edit distance. More details on this algorithm’s design, analysis, implementation and evaluation of the original model will be presented in Part V Research.

## 1.5 Longest Common Subsequence [4]

### 1.5.1 Introduction

Given two sequences X and Y, we say that a sequence Z is a common subsequence of X and Y if Z is a subsequence of both X and Y. In the longest-common-subsequence

problem, we are given two sequences X (x1, x2, ..., xm) and Y (y1, y2, ..., yn) and wish to find a maximum length common subsequence of X and Y. [4]

Note that the analysis and design parts are rephrased from Introduction to Algorithms [6].

### 1.5.2 Analysis

A subsequence of a given sequence is just the given sequence with zero or more elements left out. Formally, given a sequence $X(x1, x2, \ldots, xm)$, another sequence $Z = (z1, z2, \ldots, zk)$ is a subsequence of X if there exists a strictly increasing sequence(i1, i2, ..., ik) of indices of X such that for all j = 1;2, …, k, we have xij = zj .

In a brute-force approach to solving the LCS problem, we would enumerate all subsequences of X and check each subsequence to see whether it is also a subsequence of Y, keeping track of the longest subsequence we find. Each subsequence of X corresponds to a subset of the indices {1, 2… m} of X. Since X has $2^m$ subsequences, this approach requires exponential time, which is impractical for applying. Therefore, we expect to optimize this solution with Dynamic Programming Algorithm. [4]

### 1.5.3 Design

Step 1: Characterizing a longest common subsequence

The LCS problem has an optimal substructure property. As we shall see, the natural classes of subproblems correspond

to pairs of “prefixes” of the two input sequences. To be precise, given a sequence $X(x1, x2, \ldots, xm)$, we define the i th prefix of X, for i = 0, 1, 2… m, as Xi = (x1, x2, ..., xi).

Let $X(x1, x2, \ldots, xm)$ and $Y = (y1, y2, \ldots, yn)$ be sequences, and let $Z = [z1, z2, \ldots, zn]$ be any LCS of X and Y.

1. $\text{If } xm = yn, \text{then } zk = xm = yn \text{ and } Zk - 1 \text{ is an LCS of } Xm - 1 \text{ and } Yn - 1.$
2. $\text{If } xm! = yn, \text{then } zk! = xm \text{ implies that Z is an LCS of } Xm - 1 \text{ and Y.}$

3. If xm! = yn, then zk! = yn implies that Z is an LCS of X and Yn − 1.

Step 2: A recursive solution

Our recursive solution to the LCS problem involves establishing a recurrence for the value of an optimal solution.

Let us define Lcs [i, j] to be the length of an LCS of the sequences Xi and Yj.

If either i = 0 or j = 0, one of the sequences has length 0, and so the LCS has length 0.

If both i and j >0, when xi =xj, we get Lcs [i, j] = Lcs [i-1, j-1] + 1, while when xi != yj, we get Lcs [i, j] = max (Lcs [i, j-1], Lcs [i-1, j] ).

The optimal substructure of the LCS problem gives the recursive formula:

$$\text{Lcs}[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ \text{Lcs}[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } xi = xj, \\ \max(\text{Lcs}[i, j-1], \text{Lcs}[i-1, j]) & \text{if } i, j > 0 \text{ and } xi \,!= xj \end{cases}$$

Step 3: Computing the length of an LCS

Procedure LCS-LENGTH Take two sequences X = (x1, x2, ..., xm) and Y = (y1, y2, ..., yn) as inputs of procedure LCS-LENGTH. It stores the Lcs [i, j] values in a table Lcs [0...m, 0...n] and it computes the entries in row-major order, which means the procedure fills in the first row of c from left to right, then the second row. The procedure also maintains the table b [1...m, 1...n] to help us construct an optimal solution. Intuitively, b [i, j] points to the table entry corresponding to the optimal subproblem solution chosen when computing Lcs [i, j]. The procedure returns the b and Lcs tables; Lcs [i, j] contains the length of an LCS of X and Y.

LCS-LENGTH(X, Y)

```
m = length(X)
n = length(Y)
let b[1...m,1...n]and Lcs [0...m,0...n]be new tables
for i = 1 to m
        do Lcs [i,0] = 0
for j = 1 to n
        do Lcs [0,j] = 0
for i = 1 to m
        for j = 1 to n
                if X[i] == Y[j]
                        then Lcs [i,j] = Lcs [i-1,j-1] + 1
                                        b[i,j] = "diag"
                else if Lcs [i-1,j] >= Lcs [i,j-1]
                        then Lcs [i,j] = Lcs [i-1,j]
                                        b[i,j] = "up"
                else Lcs [i,j] = Lcs [i,j-1]
                                        b[i,j] = "left"
return Lcs and b
```

Step 4 Constructing an LCS

The table b returned by LCS-LENGTH enables us to quickly construct an LCS of X = (x1, x2, ..., xm) and Y = (y1, y2, ..., yn). We begin at b [m, n] and trace through the table by following the arrows. Whenever we encounter a“diag”in entry b [i, j], it implies that xi = xj is an element of the LCS that LCS-LENGTH found. As we apply the search strategy by trace back, we encounter the elements of this LCS in reverse order. The following recursive procedure prints out an LCS of X and Y in the proper, forward order. The initial call is PRINT-LCS( b, X , X.length, Y.length).

```
PRINT-LCS( b, X , i, j)
if i == 0 or j == 0;
    Return
if b [i ,j] == “diag”
    PRINT-LCS( b, X , i-1, j-1)
    Print xi
elseif b [i, j] ==”up”
    PRINT-LCS( b, X , i-1, j)
else PRINT-LCS( b, X , i, j-1)
```

### 1.5.4 Implementation

All the above pseudocode are quoted from Introduction to Algorithms [6]. LCS problem is implemented follows the pseudocode above by C++. We involved library <vector> and library <string>. We use a vector to point the entry of Lcs [i, j] and each entry is depended on the other three neighbor entries. We use the ‘sizeof ’ method to get the length of string A and string B.

Firstly, we set n to be the length of X and m to be the length of Y, if n =0 return m and exit, while if m = 0, return n and exit. Construct a two dimensions’ Lcs matrix with m row and n columns and initialize the first row and column to 0. Examine each character of X (i from 1 to n) and each character of Y (j from 1 to m). Set entry b [i, j] of a matrix equals to the maximum of one of following three conditions:

1. If the character i in X equal to character j in Y, then plus 1 on the previous Lcs length
2. If Lcs [i-1,j] >= Lcs [i,j-1], then Lcs [i,j] = Lcs [i-1,j] and move up.
3. If Lcs [i,j] = Lcs [i,j-1], then Lcs[i][j] = Lcs[i][j - 1] and move left.

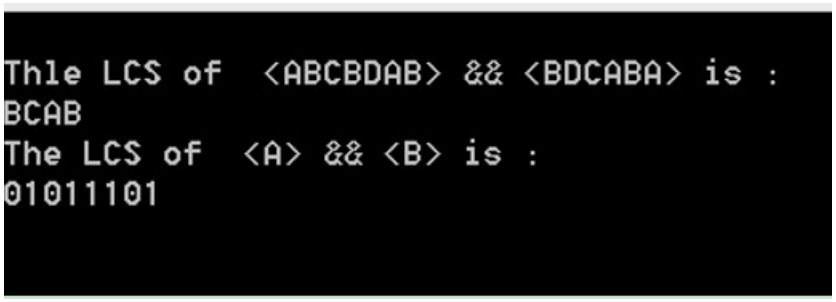
After the iteration of the above steps, we found the Longest Common Subsequence in b [i ,j] and distance in Lcs [i, j]

### 1.5.5 Evaluation

The Longest Common Subsequence problem, as its name, is to find the longest common subsequence Z of given sequence X and Y.

We test the code on the example data provides in “Introduction to Algorithms”. Take string X = “ABCBDAB” and string Y= “BDCABA” as input, apparently, our longest common subsequence shall be “BCAB”. Instead of string, we can also apply this algorithm on finding the LCS between two binary codes: assign A= {1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1 } and B = { 0, 1, 0, 1, 1, 0, 1, 1, 0 }, obviously, the LCS of A and B shall be 01011101.

Run the code we can get the expected results:




Since the LCS problem has only θ(mn) distinct subproblems, we can use dynamic programming to compute the solutions bottom up. The procedure of print the table takes time O (m+n) since it decrements at least one of i and j in each recursive call.

## 1.6 Optimal Binary Search Tree [5]

### 1.6.1 Introduction

We are given a sequence K = (k1, k2, ..., kn) of n distinct keys in an increasing sorted order (so that k1 < k2 < ...< kn), and for a given set of probabilities corresponding to

each key in sequence K, we wish to construct a binary search tree whose expected search cost is smallest. We call such a tree an optimal binary search tree. [5]

Note that the analysis and design parts are rephrased from Introduction to Algorithms [6].

### 1.6.2 Analysis

In the given sequence K = (k1, k2, ..., kn), for each key ki, we have a probability pi that a search will apply on ki. Some searches may be for values not in K, and so we also have n +1 "dummy keys" D = (d0, d1, d2,..., dn) representing values not in K. For each dummy key di, we have a probability qi that a search will be for di.

As our overall search is constructed by keys in sequence K and "dummy keys" in sequence D, the incorporation sum of the total sum of probability pi (for i = 1 to n ) and the total sum of probability qi (for i = 1 to n) shall be 1. This can be represented by the formula:$\sum_{i=1}^{n} pi + \sum_{i=0}^{n} qi = 1$

Let us give the notation of the binary search tree we are exploring as T, and suppose that the actual cost of a search equals the number of nodes examined, for instance, the depth of the node found by the search in T, plus 1 (the leaf nodes). Then the expected cost of a search in T can be displayed with the formula:

E [total search cost in T] = $\sum_{i=1}^{n}( depthT(ki) + 1) * pi + \sum_{i=0}^{n}(depthT(di) + 1) * qi = 1 + \sum_{i=1}^{n} depthT(ki) * pi + \sum_{i=0}^{n} depthT(di) * qi$

We can calculate the expected search cost node by node, however, exhaustive checking of all possibilities fails to provide an efficient algorithm. We can label the nodes of any n-node binary tree with the keys k1, k2, ..., kn to construct a binary search tree, and then add in the dummy keys as leaves. Then an obtainable solution of the n-node binary tree is $\Omega(\frac{4^{n}}{\left(n^{\frac{3}{2}}\right)})$, which means that we have to enumerate an exponential number of binary search trees in an exhaustive search. Typically, we shall solve this problem with dynamic programming.

### 1.6.3 Design

Step 1: The structure of an optimal binary search tree

If an optimal binary search tree T has a subtree T' containing keys ki ,...,kj , then the subproblem for this subtree T', with keys ki ,...,kj and dummy keys di-1,...,dj, must be optimized as well. We can form proof by applying cut-and-paste argument. If there were a subtree T'' whose expected cost is lower than that of T', then we could cut T' out of T and paste in T'', resulting in a binary search tree of lower expected cost than T, thus contradicting the optimality of T.

We need to use the optimal substructure to show that we can construct an optimal solution to the problem from optimal solutions to subproblems. Given keys ki,..., kj , we assign one of these keys, say kr (i <=r <= j ), as the root of an optimal subtree constructing with these keys. The left subtree of the root kr contains the keys ki ,...,kr-1 (and dummy keys di1, … ,dr1), and the right subtree contains the keys kr1, …,kj (and dummy keys dr, … ,dj ). As long as we examine all candidate roots kr, where i <= r <= j , and we determine all optimal binary search trees containing left subtree ki ,...,kr-1 and those containing right subtree kr+1,...,kj , we are guaranteed that we will find an optimal binary search tree.

Step 2: A recursive solution

For a subtree with keys ki,..., kj , let us denote this sum of probabilities as

$$w(i.j) = \sum_{l=i}^{j} pl + \sum_{l=i-1}^{j} ql$$

Let us define e [i，j] as the expected cost of searching an optimal binary search tree containing the keys ki,...,kj. Thus, if kr is the root of an optimal subtree containing keys ki,...,kj, we have the following equations:

e [i，j] = pr + (e [i, r-1] + w (i, r -1)) + (e [r + 1, j] + w (r + 1, j)).

Note that:

w (i, j) = w (i, r -1) + pr + w (r + 1, j).

We rewrite e [i，j]:

$$e[i, j] = e[i, r-1] + e[r+1, j] + w(i, j).$$

Assumes that we know which node kr is the root. We choose the root that gives the lowest expected search cost, and then, we can construct our recursive formulation as:

$$e[i, j] = \begin{cases} qi - 1 & if\ j = i - 1 \\ \min\limits_{i \le r \le j} \{ e[i, j] = e[i, r-1] + e[r+1, j] + w(i, j) \} & if\ i \le j \end{cases}$$

The e [i，j] values give the expected search costs in optimal binary search trees.

Step 3: Computing the expected search cost of an optimal binary search tree

Instead of computing the value of w[i, j] from scratch every time we are computing e[i, j], which would take $\theta(j - i)$additions for storing these values in a table w[1… n+1, 0…n]. For the base case, we compute w[i, i - 1] = qi-1 (for i = 1 to n + 1), while we computing w[i, j] = w[i, j - 1] + pj + qj when $i \le j$.

Thus, we can compute the $\theta(n^2)$values of w[i, j] in $\theta(1)$ time each.

The following pseudocode of OPTIMAL-BST takes inputs of the probabilities p1, …, pn andq0, …,qn and the size n, and it returns the tables e and root r.

```
OPTIMAL-BST(p,q,n)
  let e[1... n + 1, 0 ... n], w[1 ...n+1, 0 ...n],
      and root[1... n,1...n] be new tables
  for i = 1 to n + 1
      e[i, i - 1] = qi-1
      w[i, i - 1] = qi-1
  for l = 1 to n
      for i = 1 to n - l + 1
```

```
    j = i = l - 1
    e[i, j] = infinity
    w[i, j] = w[i, j - 1] + pj + qj
    for r = i to j
        t = e[i, r - 1] + e[r+1,j]+w[i, j]
        if t < e[i,j]
            e[i, j] = t
            root[i, j ] = r
return e and root
```

### 1.6.4 Implementation

The above pseudocode is quoted from Introduction to Algorithms [6]

The code of Optimal Binary Search Tree problem is implemented follows the step of pseudocode by C++.

Firstly, we define three dimension tables: e, w and root, where e stores the expected cost of searching an optimal binary search, w stores the weight of probability assigned to each node and root stores the position of the current root node. All these tables are of n+2 size since there are n nodes in the Binary Search Tree and there are two additions for head and tail of the tree.

We initialize each position in the root table to be 0. Then, we assign the probability of each dummy key ki (for i = 1 to n+1) to the corresponding position in table e and w. Then we examine through the table row by row and then column by column to assign e as infinity, while the current sum of probability w is the combination of the previous probability with the probability of current key and dummy key. We use a temp variable t to store the sum of the left subtree e [i, r-1], the right subtree e [r+1, j] and the current total probability w [i, j]. As we expect to find the minimal Optimal Binary Search Tree, we always take the minimum value between the values in tempt and table e. Once we get the minimum optimal value, we assign the corresponding root r

to our root table. After we examine through the whole Binary Search Tree, we return our optimal solution of searching cost e and root r. Finally, we print our optimal binary search tree from root r and its immediate left and right child. Then, we get down to the two children, we just found, print the new root its immediate left and right child, repeat until we print all keys in the tree.

### 1.6.5 Evaluation

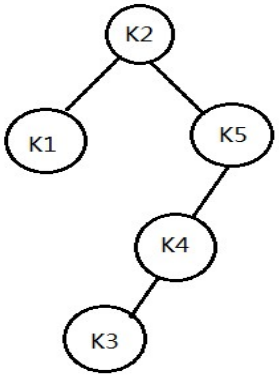
The optimal binary search tree is tending to find a binary tree, which constructs by assigning different levels of the binary tree nodes individual weights.

In the example of "Introduction to Algorithms" [6], the probability of the keys is assigned to {0, 0.15, 0.10, 0.05, 0.10, 0.20} and the probability of dummy keys is assigned to {0.05, 0.10, 0.05, 0.05, 0.05, 0.10}.

| Node | Depth | Probability | Contribution |
|---|---|---|---|
| k1 | 1 | 0.15 | 0.30 |
| k2 | 0 | 0.10 | 0.10 |
| k3 | 2 | 0.05 | 0.15 |
| k4 | 1 | 0.10 | 0.20 |
| k5 | 2 | 0.20 | 0.60 |
| d0 | 2 | 0.05 | 0.15 |
| d1 | 2 | 0.10 | 0.30 |
| d2 | 3 | 0.05 | 0.20 |
| d3 | 3 | 0.05 | 0.20 |
| d4 | 3 | 0.05 | 0.20 |
| d5 | 3 | 0.10 | 0.40 |
| Total | | | 2.80 |

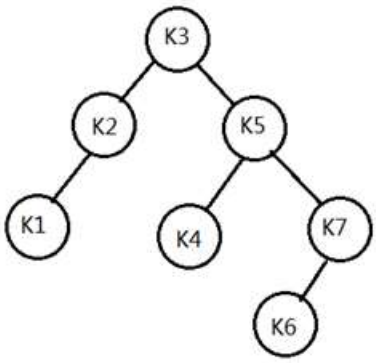
The expected result of testing is that the binary tree is printed and k2 is root. Moreover, k1 and k5 are the immediate left and right children of root k2, while k5 is the root with k4 as the left child in the second stage. In the final stage, k3 is the left child of root k4. We can print a constructed optimal binary search tree as:

```
1 1 2 2 2
  2 2 2 4
    3 4 5
      4 5
        5
k2 is root.
k1 is k2's left child
k5 is k2's right child
k4 is k5's left child
k3 is k4's left child
```




Another example, can be the keys are assigned to {0, 0.05, 0.14, 0.25, 0.02, 0, 0.02, 0.07 } and dummy keys are assigned to { 0.05, 0.10, 0.15, 0.05, 0.05, 0, 0.01, 0.01, 0.01, 0.01, 0.01}. The expected test result is that the binary tree is printed and K3 is root. In the second level, K2 and K5 are immediate left and right children of root K3. For K2, it has a left child K1, while K5 has a left child K4 and a right child K7. In the last level, K7 has a left child K6. We can print a constructed optimal binary search tree as:

```
1 2 2 3 3 3 3
  2 3 3 3 3 3
    3 3 3 3 3
      4 4 4 5
        5 5 7
          6 7
            7
k3 is root.
k2 is k3's left child
k1 is k2's left child
k5 is k3's right child
k4 is k5's left child
k7 is k5's right child
k6 is k7's left child
```




Apparently, the running time of OPTIMAL-BST procedure takes $O(n^3)$ time, since it's for loops are nested three deep and each loop index takes on at most n values. The

loop indices in OPTIMAL-BST do not have exactly the same bounds, but they are within at most 1 in all directions. Thus, the OPTIMAL-BST procedure takes $\Omega(n^3)$ time.

By now, we have completed all the practice topics on Dynamic Programming.

For any problems with optimal substructure, we can construct an optimal solution to the problem from optimal solutions to subproblems. The approaching process can always be abstract into the follows four steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

By using Dynamic Programming, we reduce the complexity of problem with exponential brute-force search down to polynomial. The usual strategies we use for optimization are recursion with memorized and bottom-up. Typically, both approaches lead us to a polynomial time solution.

In the next Chapter, we will compare dynamic programming with other well-known algorithms- divide and conquer and greedy algorithm to distinguish these algorithms and have a clear picture in mind that which algorithm works best on a particular problem.

# Chapter 2 Compare Dynamic Programming with other algorithmic approaches

A general comparison on Divide and Conquer, Dynamic Programming and Greedy Algorithm.

In general, there are four types of problems needs algorithms:

1. Decision
2. Optimization
3. Construction
4. Calculation

For Divide and Conquer, Dynamic Programming and Greedy Algorithm, they all focus on optimization.

| | Divide and Conquer | Dynamic Programming | Greedy Algorithm |
|---|---|---|---|
| Suitable Problems | General Problems | Optimization Problems | Optimization Problems |
| Substructure | Every Sub problem independent | Multiple-sub problems (Not independent) | Only one sub problem |
| Optimal substructure | No need | Must contain | Must contain |
| Sub problem(s) to solve | All sub problems | All sub problems | One sub problem |
| Sub problem in optimal solution | All | Part | Part |
| Select and solve order | Select and solve the first | Solve sub problems first | Select and solve the first |

## 2.1 Dynamic Programming vs. Divide and Conquer

### 2.1.1 Introduction

In this section, we will compare Divide and Conquer Algorithm and Dynamic Programming Algorithm on the Fibonacci number problem [7] to investigate the

difference of analysis and developing process between two strategies. Firstly, we will have a view of the Fibonacci number Problem.

The Fibonacci numbers are the numbers in the following integer sequence, called the Fibonacci sequence, and characterized by the fact that every number after the first two is the sum of the two preceding ones: 0,1,1,2,3,5,8,13,21,34... In mathematical terms, the sequence F (n) of Fibonacci numbers is defined by the recurrence relation: F (n) = F (n-1) + F (n-2), with seed values F (0) = 0, F (1) =1. Calculate the nth Fibonacci number. [26]

Note that the analysis, design and evaluation part are rephrased from Dynamic Programming Fibonacci Numbers [7].

### 2.1.2 Analysis

Divide and Conquer is an algorithm based on multi-branched recursion. A Divide and Conquer algorithm works by recursively breaking down a problem into two or more subproblems of the same or related type until they become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem. [27]

Divide the problem into some subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively. If the subproblems' size is small enough, we can solve the subproblems in a straightforward manner. That is, combine the solutions to the subproblems into the solution for the original problem.

We call recursive case as when the subproblems are large enough to solve recursively. Once the subproblems become indecomposable, we call that the base case. Sometimes, other than subproblems of the same problem, we have to solve subproblems that are not the same as the original problem.

Similar to Divide and Conquer Algorithm, Dynamic Programming Algorithm solves problems by combining the solutions to subproblems. Divide and Conquer Algorithm

break the problem into independent subproblems, solve the subproblems recursively, and then incorporate their solutions to solve the original problem. In contrast, Dynamic Programming solves overlap subproblems, when subproblems share an even smaller instance of the sub-subproblems. For this reason, Divide and Conquer Algorithm does more work than necessary due to solve the common sub-subproblems repeatedly. Dynamic Programming Algorithm solves each sub-subproblem only once and then stores its answer in an extra memory, thereby avoiding the work of recomputing the solution of each sub-subproblem that it has solved before.

We typically apply Dynamic Programming to optimize problems. Such problems can have many possible solutions. Each solution has a value, and we wish to find a solution with the optimal value. We denote such a solution an optimal solution to the problem, instead of the optimal solution, since there probably be several solutions that achieve the optimal value.

Dynamic Programming is a technique to solve the recursive problems in more efficient manner. We store the solution of these sub-problems so that we save our effort on recomputation, this is called Memoization.

To decide whether a problem can be solved by applying Dynamic Programming we check for two properties. If a problem has overlapping subproblems and optimal substructure, we can solve that problem using Dynamic Programming. Overlapping Subproblems are the name suggests the subproblems need to be repeatedly addressed. In recursion, we solve those problems every time, while in Dynamic Programming we solve these sub problems only once and store their solution for future use. If a problem can be solved by using the solutions of the subproblems, then we say that problem has an optimal substructure property.

### 2.1.3 Design

In Divide and Conquer version of the Fibonacci number problem, apply recursion on subproblems since each subproblem is independent. Then we can recursively define

Fibonacci numbers as follows: f (n) = n, when n < 2, otherwise, f (n) = f (n-1) +f (n-2).

The recursive definition of Fibonacci numbers immediately gives us a recursive algorithm for computing them:

```
int f (n)
    if (n == 0) return 0;
    if (n == 1) return 1;
    return f (n-1) +f (n-2);
```

Disadvantage: If the subproblems are not independent, the recurrence of Fibonacci numbers needs to be repeatedly solved.

To save the recomputation time, we expect to precompute all possible values. In this context, we apply Dynamic Programming (Top-down and Bottom-up) and memoization strategy on the Fibonacci number problem.

Sup-pose we need to solve the prob-lem of size N, We start solv-ing the basis prob-lem with the small-est pos-si-ble inputs and store it for future use. Then we cal-cu-late the big-ger val-ues use the stored solu-tions (solu-tion for smaller problems). Thus, whenever we encounter a problem, all its prerequisite subproblems are solved and saved. All we need to do is solve the current problem with the results of its related subproblems.

```
int f(n)
    f[0]<-0
    f[1]<-1;
    for i<-2 to n
        f[i]=f[i-1]+f[i-2];
    return f[n];
```

Typically, we use top-down with memoization to optimize the calculation space, which means we have to remember everything.

The apparent reason for the recursive algorithm's lack of speed is that it computes the same Fibonacci numbers repetitively. A single call to f (n) results in one recursive call to f (n - 1), two recursive calls to f (n - 2), three recursive calls to f (n-3), five recursive calls to f (n-4), and in general, Fk-1 (for any 0 < k < n) recursive calls to f (n- k). For each call, we're recomputing some Fibonacci number from scratch. To speed up our recursive algorithm considerably, we can just write down all the results of our recursive calls and look them up whenever we need them later.

However, in many Dynamic Programming algorithms, it is not necessary to retain all intermediate results through the whole computation. For example, we can reduce the space requirements of our algorithm significantly by maintaining only the two newest elements of the array:

```
int f(n)
    if(n<2)
        return n
    else
        if f[n] is undefined
        f[n]<-f[n-1]+f[n-2];
    return f[n];
```

We can get even more speed up by the power of the matrix$\begin{pmatrix}1 & 1\\1 & 0\end{pmatrix}$. We get the notation that Fn is the nth function we call in the Fibonacci number problem. If we calculate power n on the matrix M =$\begin{pmatrix}1 & 1\\1 & 0\end{pmatrix}$, then we get the (n+1) th Fibonacci number as the element at row and column (0, 0) in the resultant matrix. In other words, multiplying a two-dimensional vector by the matrix $\begin{pmatrix}1 & 1\\1 & 0\end{pmatrix}$ does the equivalent thing as one iteration of the inner loop of F (n) above. We can draw the conclusion

that multiplying by the matrix n times is the same as iterating the loop n times. The matrix representation gives the following closed expression for the Fibonacci numbers:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} Fn+1 & Fn \\ Fn & Fn-1 \end{pmatrix}$$

Taking determinant on both sides, we get

$$(-1)^{\mathrm{n}} = (\mathrm{Fn}+1) * (\mathrm{Fn}-1) - (\mathrm{Fn})^2$$

Moreover, since AnAm = An+m for any square matrix A, the following identities can be derived (they are obtained from two different coefficients of the matrix product)

$$FmFn + Fm-1Fn-1 = Fm+n-1$$

By putting n = n+1, we rewrite the formula as:

$$\mathrm{FmFn}+1 + \mathrm{Fm}-1\mathrm{Fn} = \mathrm{Fm}+\mathrm{n}$$

By putting m = n, we get our iteration formula:

$$\mathrm{F}(2\mathrm{n}-1) = (\mathrm{Fn})^2 + (\mathrm{Fn}-1)^2$$

$\mathrm{F}(2\mathrm{n}) = (\mathrm{F}(\mathrm{n}-1) + \mathrm{F}(\mathrm{n}+1))\mathrm{Fn} = (2\mathrm{F}(\mathrm{n}-1) + \mathrm{Fn})\mathrm{Fn}.$

Then, we discuss the problem in two follow cases.

If n is even then $\mathrm{k} = \mathrm{n}/2$, we get

$$\mathrm{F}(\mathrm{n}) = [2 * \mathrm{F}(\mathrm{k}-1) + \mathrm{F}(\mathrm{k})] * \mathrm{F}(\mathrm{k}).$$

If n is odd then k = (n + 1)/2, we get

$$\mathrm{F}(\mathrm{n}) = \mathrm{F}(\mathrm{k}) * \mathrm{F}(\mathrm{k}) + \mathrm{F}(\mathrm{k}-1) * \mathrm{F}(\mathrm{k}-1).$$

```
int f(int n)
    if (n == 0)
    return 0;
    if (n == 1 || n == 2)
    return (f[n] = 1); // Base cases
    if (f[n])       // If fib(n) is already computed
    return f[n];
    k = (n & 1)? (n+1)/2 : n/2; // Applying above formula [Note value n&1 is 1]
```

```
// if n is odd, else 0.

f[n] = (n & 1)? (fib(k)*fib(k) + fib(k-1)*fib(k-1) : (2*fib(k-1) + fib(k))*fib(k);
    return f[n];
```

We can also solve the Fibonacci number problem and check our solution above by mathematics formula: Fn $=\frac{1}{\sqrt{5}} * \left(\frac{1+\sqrt{5}}{2}\right)^{n}$.

### 2.1.4 Implementation

All the above algorithms, Recursion, Dynamic Programming, Memoization, Formula and Matrix, to solve Fibonacci number problem are implemented and compared on the time complexity and space complexity. In the implementation, we include libraries: <cstdio> for the macros traditionally defined in the Standard C library header, <map> to map the previously stored solution of subproblems in memory, <smart> to calculate 'sqrt' in mathematical formula and we also include <ctime> to method the execution time for each method to compare the efficiency of each algorithm.

Firstly, we enable users to input n, which represents the nth number in the Fibonacci sequence, and later on, we will take this n as input for each algorithm.

For normal recursion (top-down), which applied by Divide and Conquer, we return n when n < 2, otherwise, we return the sum of the previous two numbers immediately in front of the current number. We use time1 to record the calculation time of the iteration of the function.

For Dynamic Programming bottom-up, we also return n directly when n < 2, however, we calculate all the subproblems from the smallest case zero and one all the way up the nth number we require. We use fib_pre1 and fib_pre2 to store the two previous subproblems' solution, update the value in the loop every time we calculate the current value fib. We use time2 to record the calculation times of the loops.

For top-down with memoization, we also return n directly when n < 2, however, we use 'map:: iterator' to find the previously calculated solution from the storage memory tree. If we find the solutions of the two related subproblems, then we return the sum of the two values we find in the map and insert the newly calculate value into our storage tree. Otherwise, we calculate the value and save it in memory.

For the matrix, we firstly define a structure of two-dimensional matrix and a function quick_m for calling the matrix multiplication and return solution. Then we define a solution matrix to store our current solution and a 'tem' matrix to store the temperate matrix before we reach the solution matrix in produce.We initialize the solution matrix as $\begin{pmatrix}1 & 0\\0 & 1\end{pmatrix}$ tem matrix into $\begin{pmatrix}1 & 1\\1 & 0\end{pmatrix}$. Then we define a multi-function for matrix multiplication. We apply multiplication row by row and column by column, and we update the value of temp matrix each time until we return the solution matrix when we get the n-th matrix.

As we know the formula of Fibonacci number is Fn $=\frac{1}{\sqrt{5}}*\left(\frac{1+\sqrt{5}}{2}\right)^{n}$, we can also implement the mathematics formula to check our answer. Again, we return n directly when n < 2, otherwise we apply the formula by calling a function calculate, which takes input a as the base number and b as the exponential number. In this particular problem, a equal to $\left(\frac{1+\sqrt{5}}{2}\right)$ and b equal to n.

Eventually, we call clock_t in library <crime> to record to start and end time for each method to obtain the execution time of each method. We print the nth Fibonacci number return by each method with the calculation times and execution time, which equals to the result of endTime minus startTime.

### 2.1.5 Evaluation

Combine all the separate methods together, debug and test in C++, and then add a start time and end time for each method to get the execution time. We also add times to calculate the calculation times of each method.

Note that, we take the initialization of the problem as: when n=0, result =0. When n=1, result=1. We test the n-th Fibonacci number with Recursion, Dynamic Programming, Memoization, Formula and Matrix respectively as well as their execution time and calculation time in order to compare the time and space efficiency among all the above algorithms.

Test from number n = 1:

```
Please input number n
1
f_Top_Down(1)=1
f_Bottom_Up(1)=1
f_Memorization(1)=1
f_Matrix(1)=1
f_Formula(1)=1
Execution time for Recursion (Top-Down):  0second
 calculation times:  1
Execution time for Dynamic programming (Bottom-Up):  0.001second
 calculation times:  0
Execution time for Memorization:  0.001second
 and calculation times:  1
Execution time for Matrix:  0second
 and calculation times:  16
Execution time for Formula:  0.001second
```

Test when n= 5:

```
Please input number n
5
f_Top_Down(5)=5
f_Bottom_Up(5)=5
f_Memorization(5)=5
f_Matrix(5)=5
f_Formula(5)=5
Execution time for Recursion (Top-Down):  0second
 calculation times:  15
Execution time for Dynamic programming (Bottom-Up):  0.001second
 calculation times:  4
Execution time for Memorization:  0.001second
 and calculation times:  15
Execution time for Matrix:  0.001second
 and calculation times:  40
Execution time for Formula:  0second
```

Test when n=10:

Memoization uses longer time since it stores the previous solution to subproblems and searches the table to check whether or not the current subproblem is solved previously.

We can optimize the space from binary tree to list, trade-off space by time, however, there still no significant time changing.

```
Please input number n
10
f_Top_Down(10)=55
f_Bottom_Up(10)=55
f_Memorization(10)=55
f_Matrix(10)=55
f_Formula(10)=55
Execution time for Recursion (Top-Down):  0.001second
 calculation times:  177
Execution time for Dynamic programming (Bottom-Up):  0.001second
 calculation times:  9
Execution time for Memorization:  0.006second
 and calculation times:  177
Execution time for Matrix:  0.001second
 and calculation times:  48
Execution time for Formula:  0.001second
```

Test when n=15:

We notice that there still no significant time changing:

```
Please input number n
15
f_Top_Down(15)=610
f_Bottom_Up(15)=610
f_Memorization(15)=610
f_Matrix(15)=610
f_Formula(15)=610
Execution time for Recursion (Top-Down):  0.002second
 calculation times:  1973
Execution time for Dynamic programming (Bottom-Up):  0.001second
 calculation times:  14
Execution time for Memorization:  0.033second
 and calculation times:  1973
Execution time for Matrix:  0.001second
 and calculation times:  64
Execution time for Formula:  0second
```

Test when n=20:

We notice that the Recursion strategy costs significant calculation times than other algorithms since it computes the same subproblems repeatedly in each stage.

```
Please input number n
20
f_Top_Down(20)=6765
f_Bottom_Up(20)=6765
f_Memorization(20)=6765
f_Matrix(20)=6765
f_Formula(20)=6765
Execution time for Recursion (Top-Down):  0.002second
 calculation times:  21891
Execution time for Dynamic programming (Bottom-Up):  0.001second
 calculation times:  19
Execution time for Memorization:  0.367second
 and calculation times:  21891
Execution time for Matrix:  0.001second
 and calculation times:  56
Execution time for Formula:  0.001second
```

Test when n=25:

```
Please input number n
25
f_Top_Down(25)=75025
f_Bottom_Up(25)=75025
f_Memorization(25)=75025
f_Matrix(25)=75025
f_Formula(25)=75025
Execution time for Recursion (Top-Down):  0.009second
 calculation times:  242785
Execution time for Dynamic programming (Bottom-Up):  0.001second
 calculation times:  24
Execution time for Memorization:  4.841second
 and calculation times:  242785
Execution time for Matrix:  0.002second
 and calculation times:  64
Execution time for Formula:  0.001second
```

Test when n=30:

```
Please input number n
30
f_Top_Down(30)=832040
f_Bottom_Up(30)=832040
f_Memorization(30)=832040
f_Matrix(30)=832040
f_Formula(30)=832040
Execution time for Recursion (Top-Down):  0.092second
 calculation times:  2692537
Execution time for Dynamic programming (Bottom-Up):  0.001second
 calculation times:  29
Execution time for Memorization:  50.973second
 and calculation times:  2692537
Execution time for Matrix:  0.001second
 and calculation times:  72
Execution time for Formula:  0.002second
```

As top-down with memoization methods gets too slow, we drop it, and continue running other methods:

Test when n=35:

Apparently, Recursion method use significantly more time and calculation operations than other methods.

```
Please input number n
35
f_Top_Down(35)=9227465
f_Bottom_Up(35)=9227465
f_Matrix(35)=9227465
f_Formula(35)=9227465
Execution time for Recursion (Top-Down):  1.03second
 calculation times:  29860703
Execution time for Dynamic programming (Bottom-Up):  0.001second
 calculation times:  34
Execution time for Matrix:  0.001second
 and calculation times:  72
Execution time for Formula:  0.001second
```

Test when n=40:

After 40, Recursion start to get really slow, drop it, and run other methods:.

```
Please input number n
40
f_Top_Down(40)=102334155
f_Bottom_Up(40)=102334155
f_Matrix(40)=102334155
f_Formula(40)=102334155
Execution time for Recursion (Top-Down):  12.583second
 calculation times:  331160281
Execution time for Dynamic programming (Bottom-Up):  0.001second
 calculation times:  39
Execution time for Matrix:  0.004second
 and calculation times:  64
Execution time for Formula:  0.001second
```

Test when n=45:

```
Please input number n
45
f_Bottom_Up(45)=1134903170
f_Matrix(45)=1134903170
f_Formula(45)=1134903170
Execution time for Dynamic programming (Bottom-Up):  0.001second
 calculation times:  44
Execution time for Matrix:  0.001second
 and calculation times:  80
Execution time for Formula:  0.001second
```

Test when n=46:

```
Please input number n
46
f_Bottom_Up(46)=1836311903
f_Matrix(46)=1836311903
f_Formula(46)=1836311903
Execution time for Dynamic programming (Bottom-Up):  0.001second
 calculation times:  45
Execution time for Matrix:  0.001second
 and calculation times:  80
Execution time for Formula:  0.001second
```

Test when n=47:

Buffer overflow in this case. Thereby, for a 32-bit virtual machine, the largest Fibonacci number can be calculated is the 46th number in the sequence.

```
Please input number n
47
f_Bottom_Up(47)=-1323752223
f_Matrix(47)=-1323752223
f_Formula(47)=-2147483648
Execution time for Dynamic programming (Bottom-Up):  0.001second
 calculation times:  46
Execution time for Matrix:  0.003second
 and calculation times:  88
Execution time for Formula:  0.001second
```

Notes: The execution time can be slightly different in each run.

By Divide and Conquer with normal recursion algorithm, we get exponential results due to the recomputation on recurrence of subproblems.

We can get the time complexity of recursion by calculating T(n)=T(n-1)+T(n-2)+O(1), where T(n)=O([(1+√5)/2]^n), which is exponential large. As we use binary tree space for storage, our space complexity will be O (n), where n depends on depth n of the binary tree we build.

As we expected, Dynamic Programming approach performs a significant efficiency in solving Fibonacci number problem. In the bottom-up approach, the execution time reduces from exponential brute-force to linear time$O(n)$with a space complexity of $O(1)$, which only takes constant space.

For the nth calculation, if n is big enough, then time complexity will get to $O(n^2)$ and space complexity will be $O(n)$ to store every binary bit.

For matrix multiplication, if we use repeated squaring, computing the nth power of something requires only $O(\log n)$ multiplications. In this case, that means $O(\log n)$numbers of 2 * 2 matrix multiplications, each of which reduces to a constant

number of integer multiplications and additions. Thus, we can compute Fn in only $O(\log n)$ integer arithmetic operations and consume only $O(1)$ space which is the only constant number.

This is an exponential speedup over the standard iterative algorithm, which was already an exponential speedup over our original recursive algorithm.

## 2.2 Dynamic Programming vs. Greedy Algorithm

### 2.2.1 Introduction

In this Chapter, we are going to compare two well-known algorithms- Dynamic Programming and Greedy Algorithms with a real world problem - coin change problem. We are going to explore the difference between the strategies as well as the efficiencies of both algorithms. Moreover, as we know that Greedy Algorithms fail to provide a best optimal solution sometimes, due to its "greedy" property, we expect to find the appropriate constraints on the amount of coins, denominations and total value, which enables a Greedy Algorithm to find the actual optimal solution.

We can denote a coin-changing problem as to use the minimal number of the given denominations 1= d1< d2< ...< dk of coins to make change for money of amount n. For instance, we expect to get a total amount of 15 with the minimum number of coins using of the given three denominations of coins: 1, 5 and 11. If we solve this problem with Dynamic Programming Algorithm, we can provide the optimal solution with three coins of value 5 in total. However, we will get an approximately optimal solution of five coins - one of value 11 and four of value 1 - by using Greedy Algorithm. Therefore, Greedy Algorithm fails to find the real optimal solution in practice.

Note that the analysis, design and evaluation part are rephrased from Dynamic Programming – From Novice to Advanced [28]

### 2.2.2 Analysis

An algorithm has to process a sequence of steps and make a choice at each step to optimize problems. Although Dynamic Programming guarantees an optimal solution for those problems with overlapping subproblems and optimal substructure, sometimes, we prefer a much simpler and straightforward algorithm to solve those problems with the easier structure. A Greedy Algorithm always makes the best local decision bases only on the current situation without considering its influence on future decisions. Therefore, we expect this algorithm leads us to a globally optimal solution by making each locally optimal decision.

Dynamic Programming usually base on a recurrence formula and support cases. A sub-solution of the problem constructs from previously found sub-solutions in polynomial complexity.

There are lots of similarities between Greedy Algorithms and Dynamic Programming. Typically, Greedy Algorithms also applies to solve problems with optimal substructure. However, the most significant difference between Greedy Algorithms and Dynamic Programming is that Greedy Algorithms always make a “greedy” choice first and then solve the rest subproblems while Dynamic Programming is finding the optimal solutions to subproblems first and then making formal decisions.

Take the coin-changing problem as an example, to change a given amount with a minimum number of coins, with greedy strategy; we will repeatedly select the largest denomination coin that no larger than the remaining amount.

Note that Greedy Algorithms approach such problem with a very simple strategy, it can save time and space than a Dynamic Programming approach.

As Dynamic Programming solves all the subproblems before we make a formal choice to an overall problem needs the results of all solutions of those subproblems, we can always get a globally optimal solution. Greedy Algorithms do not always promise a globally optimal solution due to its “greedy” property, which is to make

each choice as the best local optimal manner. That is, sometimes, a local best choice can result in no further optimal choice but a sequence of terrible choices.

### 2.2.3 Design

Firstly, we approach coin problem with Greedy Algorithms.

We want to use the smallest number of coins to make change for amount n with denominations 1= d1< d2< ...< dk. Therefore, we can denote C[p] as the minimum number of coins needed to make change for amount p, and we denote x as the value of the first coin used in the optimal solution.

At each step, we just add the maximum value of denominations less than the remaining amount. For instance, if we have coins' denomination 1, 5 and 8 and we want to get the amount of 11. By Greedy Algorithm, we pick eight first, and we left amount 3 (11-8) to achieve, then we can only continually pick three coins with value 1. By this way, we get a solution of four coins in total.

Then We can create the following pseudocode:

```
int coin_changing_greedy (D, p) {
Let D be the set of denominations {d1, d2, ...,dk}
if p=0, then C[p]=0
x= max of D which no larger than (p - x)
While x ≠ p do
   C[p]=1+ C[p-x]
        for i= 1 to k
           C[p]= Min of C[p-di]+1 for all values of di if p>0
        endfor
return C[p]
}
```

However, an intuitive optimal solution for the above example should be two coins with value 5 and another coin of value 1.

We can easily observe that Greedy Algorithm doesn't work for certain denominations and amount.

In this case, we intend to solve this coin problem with Dynamic Programming Algorithm. To construct a Dynamic Programming solution, we need to find a sub-solution for an optimal solution and for helping us to find the optimal solution for the next sub-solution. In the coin problem, a state would be the solution for sum i, where i $\leqslant$ n (amount). A sub-sub-solution would be the solution for any sum j, where j < i, which smaller than sub-solution i. For finding a sub-solution i, we need to first find all smaller sub-solution j (j < i). Once we find the minimum number of coins which sum up to i, we can easily find the next sub-solution for i + 1.

For each coin j, where dj $\leqslant$ i, look at the minimum number of coins we have already found for the i-dj sum. Denote this number be min. If min+1 is less than the minimum number of coins already found for current sum i, then we update the current min with min + 1.

We can create the following pseudocode:

Dynamic Programming

```
int coin_changing_DP (n) {
C[0]=0;

min[0]=0;
for i=1 to n
      min[i] = infinity
      for j=1 to j
            if dj<=i then
                if(1+C[i-dj]< min) then
```

```
                    min[i] = 1 + C[i-dj]
        endfor
return  min[n]
}
```

We can apply this Dynamic Programming approaching strategy to our coin problem. Recall the example we have given above, we are given coins with values one, five, and eight and the amount n is set to be 11.

First of all, we denote that for state 0 (amount zero) we have found a solution with a minimum number of 0 coins.

In the state one, where we want to get amount one, we assign all values in the min as infinity since we haven't yet found a solution for this one yet. Then we notice that only value one is less than or equal to the current sum. We see that for sum 1-d1= 0 we have a solution with zero coins. Since we add one coin to this solution, we'll have a solution with one coin with value one for a sum of one. It's the only solution for this amount, and we store this solution.

In the state two, where we want to get amount two, the only coin which is less or equal to this amount is the first coin, having a value of one. The optimal solution found for sum one (2-1) is coin one. This one first coin plus another one will sum up to two, and thus we make a sum of two with only two coins of value one. This is the best and only solution for a sum of two. As we don’t have other value than one to choose in each state, we can only repeat the above steps for state three and state four. That is, we are going to get solutions of three coins of value one for an amount of three and four coins of value one to get an amount of four separately.

In the state five, where we now have two values to analyze- one and five. Firstly, we analysis the choice with value one. There exists a previous solution in state four for amount four (5-1), and thus we can construct a solution for sum five by adding another coin with value one to it. As the best option for sum four that we found has four coins, the update solution for sum five will have five coins. In the other case, we can take the coin with value five.

The sum for this coin needs to be added to make five, is zero. In the state 0, we already denote that sum zero is made up of 0 coins. Therefore, we can make a sum of five with only one coin with value five. Note that our current solution, which requests only one coin, is better than the previously found solution for amount three, which composes of five coins.

With the similar idea, we can get a sum of six with two coins: one and five instead of six coins of value one. We can keep going on until we achieve our given amount 11.

In this way, we have found a solution of three coins- five, five and one- which sum up to 11.

Additionally, we can backtrack what coins we used to get to a certain amount from a previous one.

For example: To get amount 11 we got by adding the coin with value one to a sum of ten. To amount ten we got from five, while five from 0. This way we find the coins used: one, five and five.

To approach in a bottom- up way, we can update the best solution for a sum i, whenever a better solution for this sum was found, the states aren't calculated sequentially. Still, consider the problem above, we start with a base case solution of 0 coins for sum 0.

Now we add the first coin with value one to the amount already found. If the solution k is constituted of fewer coins than the previous solution - we'll update the solution for it. Then we repeatedly apply this strategy for the rest of them.

For example, we first add the first coin to sum zero and get sum one, and this is the current best solution, and we note this solution as C[1]=1. By adding another same coin to sum one, we'll get sum two, thus making C[2]=2. We keep going on until the state five, where we have already got solution C[4]=4 with four coins one. After the first coin is processed, take the second coin (having a value of five). Serially try to add it to each of the sums already found. By adding it to zero, a sum five made up of one coin will result. Earlier, C[5] has been equal to 5. Thus the new solution is better than the previously found one. We update our solution C[5]=1. After adding the same coin to sum one, we'll get a sum six composed of two coins. Previously we found a

sum of six composed of six coins. Now we found a better solution, so we update C[6] to two.

The identical thing is done for next sums - each time a better solution is found, the results are updated. We keep approaching the next sum with this strategy until we reach to given amount. [28]

### 2.2.4 Implementation

The code for coin problem is composed of implementation of Greedy Algorithm approaching as well as Dynamic Programming approaching and the comparison between these two algorithms on execution time, calculation times and the optimal solution they found.

In the code, typically, we apply two the algorithms with given value 1, 5 and 11 to get the amount provided by users.

For Dynamic Programming, we have two methods- DP_Money and BestChoice- the former one is to find the optimal sum value, and the later one is to provide the optimal solution bases on the optimal sum value we found and stored previously. Therefore, before we add up to the sum of our second value 5, we return the minimum number we found as Num[i], where Num[i]=i. For example, for i = 3 we have Num[3]=3 as we can only construct the solution with value 1. For the amount Num[i], where $i \geqslant 5$, we get our solution by choosing the minimum number of coins composing with value 1, 5 and 11. We always try to fill the gap between the current sum and the given amount with the least number of coins. In BestChoice method, we partition coin problem with our given values-1,5 and 8. That is, if $i<5$, then Num[i]=i. If $5 \leqslant i<11$, we can make our optimal choice with a value2 (5) and amount-5 of value3 (1). When, $i>11$, we track through the previous amount we calculate and always choice the minimum number of coins with a contribution of all three values.

For Greedy Algorithm, we achieve the function with a Greedy method with the input of given amount and the three values we assigned. By Greedy Algorithm, we always select the largest value less than the current amount, without taking consideration of the influence of the current choice.

In the main function, we compare the execution time of two algorithms with the difference time between start and end times. We also calculate the calculation times of each method. Firstly, we print all the solutions given by Dynamic Programming and Greedy Algorithms for amount 0 to 20. Then we allow users to give an amount, and solutions will be given by both algorithms.

### 2.2.5 Evaluation

In this section, we test coin charging problem for Dynamic Programming and Greedy Algorithms. Firstly, we print all the solutions for the total money of 0 to 40.

In Dynamic Programming version we get the following result:

```
Num[0]=   0,   0,   0,   0
Num[1]=   1,   0,   0,   1
Num[2]=   2,   0,   0,   2
Num[3]=   3,   0,   0,   3
Num[4]=   4,   0,   0,   4
Num[5]=   1,   0,   1,   0
Num[6]=   2,   0,   1,   1
Num[7]=   3,   0,   1,   2
Num[8]=   4,   0,   1,   3
Num[9]=   5,   0,   1,   4
Num[10]=   2,   0,   2,   0
Num[11]=   1,   1,   0,   0
Num[12]=   2,   1,   0,   1
Num[13]=   3,   1,   0,   2
Num[14]=   4,   1,   0,   3
Num[15]=   3,   0,   3,   0
Num[16]=   2,   1,   1,   0
Num[17]=   3,   1,   1,   1
Num[18]=   4,   1,   1,   2
Num[19]=   5,   1,   1,   3
Num[20]=   4,   0,   4,   0
Num[21]=   3,   1,   2,   0
Num[22]=   2,   2,   0,   0
Num[23]=   3,   2,   0,   1
Num[24]=   4,   2,   0,   2
Num[25]=   5,   2,   0,   3
Num[26]=   4,   1,   3,   0
Num[27]=   3,   2,   1,   0
Num[28]=   4,   2,   1,   1
Num[29]=   5,   2,   1,   2
Num[30]=   6,   2,   1,   3
Num[31]=   5,   1,   4,   0
Num[32]=   4,   2,   2,   0
Num[33]=   3,   3,   0,   0
Num[34]=   4,   3,   0,   1
Num[35]=   5,   3,   0,   2
Num[36]=   6,   3,   0,   3
Num[37]=   5,   2,   3,   0
Num[38]=   4,   3,   1,   0
Num[39]=   5,   3,   1,   1
Num[40]=   6,   3,   1,   2
```

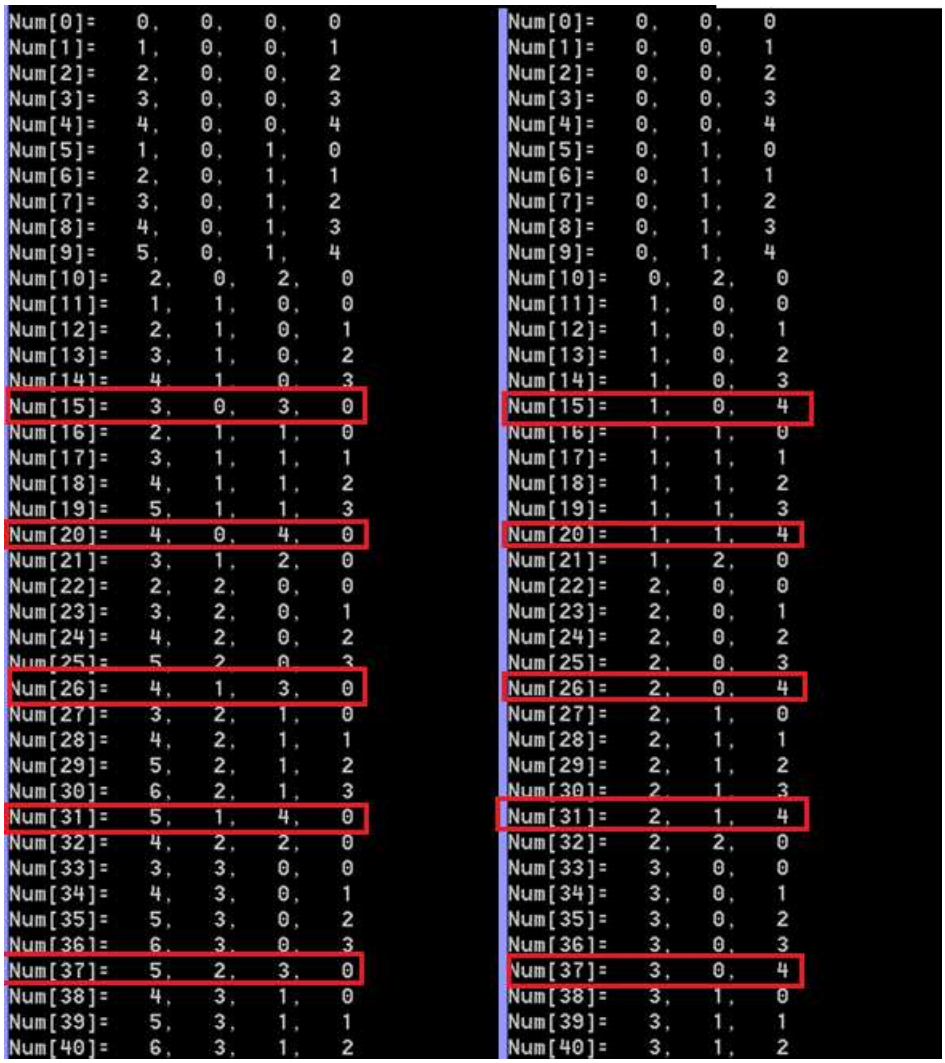
In Greedy Algorithm version we get the following result:

```
Num[0]=   0,   0,   0
Num[1]=   0,   0,   1
Num[2]=   0,   0,   2
Num[3]=   0,   0,   3
Num[4]=   0,   0,   4
Num[5]=   0,   1,   0
Num[6]=   0,   1,   1
Num[7]=   0,   1,   2
Num[8]=   0,   1,   3
Num[9]=   0,   1,   4
Num[10]=   0,   2,   0
Num[11]=   1,   0,   0
Num[12]=   1,   0,   1
Num[13]=   1,   0,   2
Num[14]=   1,   0,   3
Num[15]=   1,   0,   4
Num[16]=   1,   1,   0
Num[17]=   1,   1,   1
Num[18]=   1,   1,   2
Num[19]=   1,   1,   3
Num[20]=   1,   1,   4
Num[21]=   1,   2,   0
Num[22]=   2,   0,   0
Num[23]=   2,   0,   1
Num[24]=   2,   0,   2
Num[25]=   2,   0,   3
Num[26]=   2,   0,   4
Num[27]=   2,   1,   0
Num[28]=   2,   1,   1
Num[29]=   2,   1,   2
Num[30]=   2,   1,   3
Num[31]=   2,   1,   4
Num[32]=   2,   2,   0
Num[33]=   3,   0,   0
Num[34]=   3,   0,   1
Num[35]=   3,   0,   2
Num[36]=   3,   0,   3
Num[37]=   3,   0,   4
Num[38]=   3,   1,   0
Num[39]=   3,   1,   1
Num[40]=   3,   1,   2
```

When we compare Dynamic Programming Algorithm with Greedy Algorithm side by side, we can see from above data; the Greedy Algorithm can't give the best solution for the certain total number to charge.

The value of the coins are: 1,5,11

The comparison of Dynamic Programming and Greedy Algorithms:




The Dynamic Programming always guarantees an optimal solution, while Greedy Algorithm is failing to provide the optimal solution for some values.

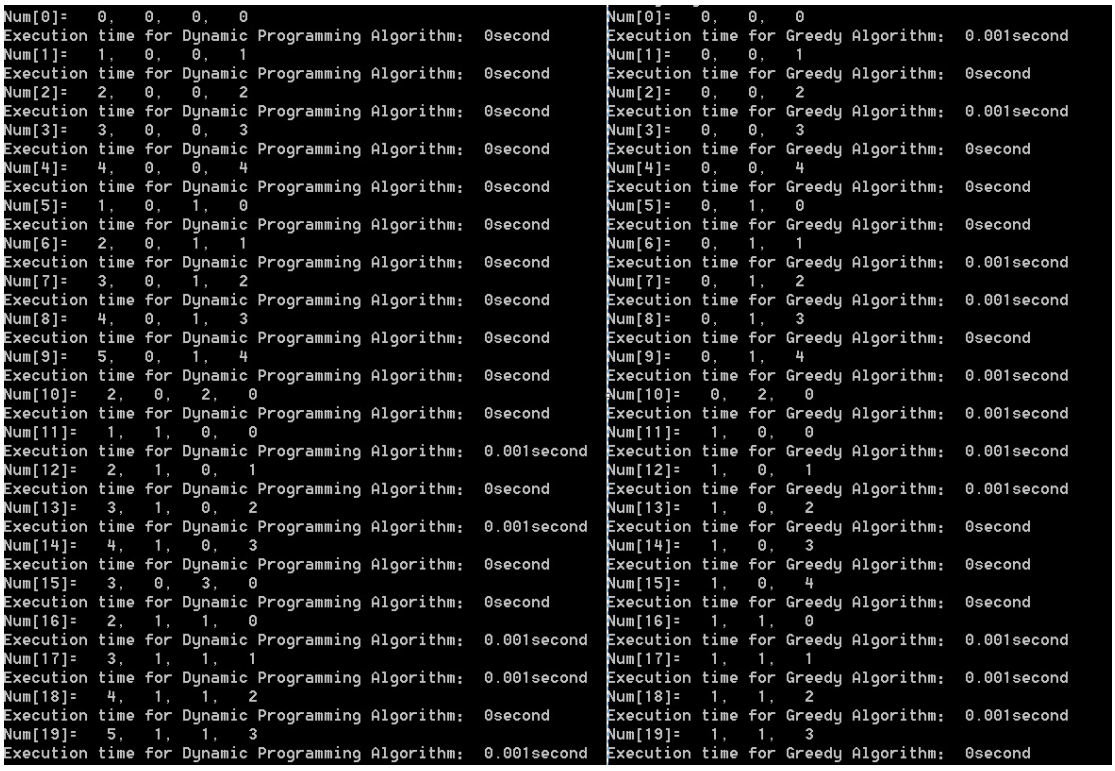
For instance, in the first 40 money charging problems, when money=15 the best solution supposes to be three coins of value 5, however, Greedy Algorithm provides five coins (one 11 and four value 1) as a solution, which is not optimization.

Similarly for the other instances, when money=20 the best solution supposes to be four coins of value 5, however, Greedy Algorithm provides six coins (one 11, one 5 and four value 1) as a solution. When money=26 the best solution supposes to be four coins (one 11 and three value 5), however, Greedy Algorithm provides six coins (two value 11 and four value 1) as a solution. When money=31 the best solution supposes to be five coins (one 11 and four value 5), however, Greedy Algorithm provides seven coins (two 11, one 5 and four value 1) as a solution. When money=37 the best

solution supposes to be five coins (two 11 and three value 5), however, Greedy Algorithm provides seven coins (three 11 and four value 1) as a solution.

Intuitionally, Greedy Algorithm always pick the largest value first and then follows the second largest value and so on. But sometimes, the max number of the greatest value can't guarantee an optimal solution. Maybe the certain number of the second largest value provide the best solution. Although Dynamic Programming ensures an optimal solution, it may cost more operations and enumeration than Greedy Algorithm. That is the reason why Greedy Algorithm is still used for particular situation and problems. Besides, Greedy Algorithm is proved to be a good way to do some tests on the NP problem, which not request for an accurate solution, but an obscure result that whether the problem is solvable.

Dynamic Programming and Greedy Algorithm on execution time:




Intuitionally, there is only one loop in Greedy Algorithms while on average there will be two loops in Dynamic Programming in total, one for optimal value and the other for the optimal solution. Therefore, Dynamic Programming supposes to take slightly longer time than Greedy Algorithms, however, both algorithms are of polynomial complexity. As the test data are relatively small, there is no significant difference

between the execution time. If the problem gets large enough, Dynamic Programming will slow down by the complex construct of subproblem and constructing the solution with sub-solutions.

```
Dynamic Programming:
Total amount:   The total number of coins :    number of each value : 11, 5, 1
Num[0] = 0 , 0 , 0 , 0
 Execution time for Dynamic Programming Algorithm:  0.001second
 The calculation times of Dynamic Programming Algorithm is: 16
Num[1] = 1 , 0 , 0 , 1
 Execution time for Dynamic Programming Algorithm:  0.001second
 The calculation times of Dynamic Programming Algorithm is: 16
Num[2] = 2 , 0 , 0 , 2
 Execution time for Dynamic Programming Algorithm:  0.001second
 The calculation times of Dynamic Programming Algorithm is: 16
Num[3] = 3 , 0 , 0 , 3
 Execution time for Dynamic Programming Algorithm:  0.001second
 The calculation times of Dynamic Programming Algorithm is: 16
Num[4] = 4 , 0 , 0 , 4
 Execution time for Dynamic Programming Algorithm:  0.001second
 The calculation times of Dynamic Programming Algorithm is: 16
Num[5] = 1 , 0 , 1 , 0
 Execution time for Dynamic Programming Algorithm:  0.001second
 The calculation times of Dynamic Programming Algorithm is: 16
Num[6] = 2 , 0 , 1 , 1
 Execution time for Dynamic Programming Algorithm:  0.002second
 The calculation times of Dynamic Programming Algorithm is: 16
Num[7] = 3 , 0 , 1 , 2
 Execution time for Dynamic Programming Algorithm:  0.002second
 The calculation times of Dynamic Programming Algorithm is: 16
Num[8] = 4 , 0 , 1 , 3
 Execution time for Dynamic Programming Algorithm:  0.001second
 The calculation times of Dynamic Programming Algorithm is: 16
Num[9] = 5 , 0 , 1 , 4
 Execution time for Dynamic Programming Algorithm:  0.001second
 The calculation times of Dynamic Programming Algorithm is: 16
Num[10] = 2 , 0 , 2 , 0
 Execution time for Dynamic Programming Algorithm:  0.001second
 The calculation times of Dynamic Programming Algorithm is: 16
Num[11] = 1 , 1 , 0 , 0
 Execution time for Dynamic Programming Algorithm:  0.003second
 The calculation times of Dynamic Programming Algorithm is: 16
Num[12] = 2 , 1 , 0 , 1
 Execution time for Dynamic Programming Algorithm:  0.001second
 The calculation times of Dynamic Programming Algorithm is: 16
Num[13] = 3 , 1 , 0 , 2
 Execution time for Dynamic Programming Algorithm:  0.001second
 The calculation times of Dynamic Programming Algorithm is: 16
Num[14] = 4 , 1 , 0 , 3
 Execution time for Dynamic Programming Algorithm:  0.002second
 The calculation times of Dynamic Programming Algorithm is: 16
```

```
Num[15] = 3 , 0 , 3 , 0
 Execution time for Dynamic Programming Algorithm:  0.002second
 The calculation times of Dynamic Programming Algorithm is: 16
Num[16] = 2 , 1 , 1 , 0
 Execution time for Dynamic Programming Algorithm:  0.002second
 The calculation times of Dynamic Programming Algorithm is: 16
Num[17] = 3 , 1 , 1 , 1
 Execution time for Dynamic Programming Algorithm:  0.001second
 The calculation times of Dynamic Programming Algorithm is: 16
Num[18] = 4 , 1 , 1 , 2
 Execution time for Dynamic Programming Algorithm:  0.002second
 The calculation times of Dynamic Programming Algorithm is: 16
Num[19] = 5 , 1 , 1 , 3
 Execution time for Dynamic Programming Algorithm:  0.001second
 The calculation times of Dynamic Programming Algorithm is: 16
Num[20] = 4 , 0 , 4 , 0
 Execution time for Dynamic Programming Algorithm:  0.002second
 The calculation times of Dynamic Programming Algorithm is: 16
```

```
Greedy Algorithm:
Total amount:  number of each value : 11, 5, 1
Num[0] = 0 , 0 , 0
Execution time for Greedy Algorithm:  0.002second
The calculation times of Greedy Algorithm is: 0
Num[1] = 0 , 0 , 1
Execution time for Greedy Algorithm:  0.004second
The calculation times of Greedy Algorithm is: 1
Num[2] = 0 , 0 , 2
Execution time for Greedy Algorithm:  0.001second
The calculation times of Greedy Algorithm is: 3
Num[3] = 0 , 0 , 3
Execution time for Greedy Algorithm:  0.001second
The calculation times of Greedy Algorithm is: 6
Num[4] = 0 , 0 , 4
Execution time for Greedy Algorithm:  0.001second
The calculation times of Greedy Algorithm is: 10
Num[5] = 0 , 1 , 0
Execution time for Greedy Algorithm:  0.002second
The calculation times of Greedy Algorithm is: 11
Num[6] = 0 , 1 , 1
Execution time for Greedy Algorithm:  0.001second
The calculation times of Greedy Algorithm is: 13
Num[7] = 0 , 1 , 2
Execution time for Greedy Algorithm:  0.001second
The calculation times of Greedy Algorithm is: 16
Num[8] = 0 , 1 , 3
Execution time for Greedy Algorithm:  0.001second
The calculation times of Greedy Algorithm is: 20
Num[9] = 0 , 1 , 4
Execution time for Greedy Algorithm:  0.001second
The calculation times of Greedy Algorithm is: 25
Num[10] = 0 , 2 , 0
Execution time for Greedy Algorithm:  0.001second
The calculation times of Greedy Algorithm is: 27
```

```
Num[11] = 1 , 0 , 0
Execution time for Greedy Algorithm:  0.001second
The calculation times of Greedy Algorithm is: 28
Num[12] = 1 , 0 , 1
Execution time for Greedy Algorithm:  0.002second
The calculation times of Greedy Algorithm is: 30
Num[13] = 1 , 0 , 2
Execution time for Greedy Algorithm:  0.001second
The calculation times of Greedy Algorithm is: 33
Num[14] = 1 , 0 , 3
Execution time for Greedy Algorithm:  0.001second
The calculation times of Greedy Algorithm is: 37
```

```
Num[15] = 1 , 0 , 4
Execution time for Greedy Algorithm:  0.001second
The calculation times of Greedy Algorithm is: 42
Num[16] = 1 , 1 , 0
Execution time for Greedy Algorithm:  0.001second
The calculation times of Greedy Algorithm is: 44
Num[17] = 1 , 1 , 1
Execution time for Greedy Algorithm:  0.001second
The calculation times of Greedy Algorithm is: 47
Num[18] = 1 , 1 , 2
Execution time for Greedy Algorithm:  0.002second
The calculation times of Greedy Algorithm is: 51
Num[19] = 1 , 1 , 3
Execution time for Greedy Algorithm:  0.002second
The calculation times of Greedy Algorithm is: 56
Num[20] = 1 , 1 , 4
Execution time for Greedy Algorithm:  0.001second
The calculation times of Greedy Algorithm is: 62
```

We notice that the calculation times of Greedy Algorithm keeps growing with the problem size. However, Dynamic Programming algorithm has a constant calculation times because Dynamic Programming calculate all the previous subproblems before the problem we are solving. That is to say, after we calculate all the potential subproblems, the only work we do is just call the certain solutions of sub-problems required by the current problem.

This is an example that user are allowed to give an amount to be changed with the three given values-11, 5 and 1- and there will be solutions given be both by Dynamic Programming and Greedy Algorithms.

```
Please input the number of money to change:
26
Algorithm :   The total number of coins is :     value : 11, 5, 1
Dynamic Programming:4,1,3,0
Greedy Algorithm:6,2,0,4
```

As we have seen in such a volume of examples that Greedy Algorithm fail to provide an optimal solution, we are wondering is there certain constraints of number of values, the given values and the given amount that influence the decision given by Greedy

Algorithm. That is to say, is it possible for us to find the given values that promise Greedy Algorithm to provide an optimal solution?Change and Gill [1] show that if the Greedy Algorithm is not always optimal for the given denominations 1= d1 <d2<…dm, then there exists a counterexample X in the range $d3 \leq X \leq \frac{dm(dm*dm-1+dm-3*dm-1)}{dm-dm}$.Recall our example above, as we only have three difference denominations-1, 5 and 11, according to the formula above, we will get counterexample in the range of $11 \leq X \leq \frac{11*(11*5+11-\ *5)}{11-5}$,which is 94. We can test the existence of such counterexample by print all the solutions given by Greedy Algorithm and compare with the optimal solutions of all X in the range above. There is another paper from Dexter Kozen[2] and Shmuel Zaks [3] presents that if a counterexample exists, then the smallest one lies in the range d3+1 < X < dm + dm-1. In our example, that is, the first counter example we will encounter exists in the range 12< X < 16. As shown above, our first counter example is when amount X=15, the Greedy Algorithm gives solution 5 while the optimal solution is 3. Therefore, this theory narrow down the bounds for us to test the existence of counter example. We only need to test from d3 +1 to dm +dm-1 to check if there exists any counterexample, if we find any counter example, we can assume that Greedy Algorithm doesn't guarantee to provide optimal solution in this given denominations, otherwise, we know that Greedy Algorithm will work and we can apply Greedy Algorithm to solve this problem since Greedy Algorithm's operation is much simpler and efficient than Dynamic Programming.

# Chapter 3 Apply Dynamic Programming in Shortest Paths Problem

## 3.1 Introduction

After comparing Dynamic Programming Algorithm with Greedy Algorithm in the last chapter, we have a better understanding of both algorithms on how to develop the algorithms as well as the constraint of what problems each algorithm is applicable. In this Chapter, we apply dynamic programming algorithms and greedy algorithm in graph problems to solve Shortest-Paths problem [8].

Shortest-paths algorithms typically hold the property that the shortest path between two vertices contains other shortest paths within it. With the optimal substructure property, we can apply Dynamic Programming and the Greedy methods. Dijkstra's algorithm employs a greedy strategy, while the Floyd-Warshall algorithm, which finds the shortest paths between all pairs of vertices, is a Dynamic Programming Algorithm. Finally, we will see Johnson's algorithm as a combination of both Greedy Algorithm and Dynamic Programming Algorithm.

In a ***shortest-paths problem*** [8], we are given a weighted, directed graph G= (V,E). Assume that each edge e (i, j) ∈ E has an associated weight wij. The weight representsa cost for going directly from node i to node j in the graph.

In this chapter, we are going to solve the following two relatedproblems.

The first type problem is for a given graph G contains negative cycles. This problem can be denoted as a directed cycle C that$\sum_{ij\in C} cij < 0$.

The other type problem is for a given graph G contains no negative cycles, and we intend to find a path p from a source node s to a destination node d with shortest total cost, which can be presented as$\sum_{ij\in p} cij$. This amount shall be the minimum cost among all the paths' cost from s to d. The problems to find this minimum cost in the given graph G is called Shortest-Path Problem.

There are several versions of Shortest-Path Problem due to the different constraints of the given graph.

There are several versions of Shortest-Path Problem due to the different constraints of the given graph.

**Single-destination shortest-paths problem**: Find the shortest path to a given destination vertex d from each vertex v. Note that this problem is just an opposing version of the single-source problem. We can solve this problem by reversing the source and destination vertices of each edge in the graph.

**Single-pair shortest-path problem** [9]: Find the shortest path from s to d for given vertices s and d. If we can solve the single-source problem with source vertex s, we can address this issue as well. Furthermore, all known algorithms for this problem have the same worst-case asymptotic running time as the best single-source algorithms.

**All-pairs shortest-paths problem** [12]: Find the shortest path from s to d for every pair of vertices s and d. Although we can solve this problem by running a single-source algorithm once for each vertex, typically, we can solve it faster because it contains optimal substructure where we can apply our dynamic programming algorithm.

Note that all the introduction, analysis and design parts of algorithms in the Shortest Path Problem are rephrased from *Introduction to Algorithm* [6].

### 3.2 Single Source Shortest Path Problems [9]

Note that all the introduction, analysis and design parts of Single Source Shortest Path Problems are rephrased from *Introduction to Algorithm* [6].

In this section, we will see two well-known algorithms: Bellman-Ford and Dijkstra algorithms. Both algorithms are implemented to single-source shortest-path problems with the technique of relaxation, which is used to get the minimum total weights of paths between the given source vertex s and the destination vertex d. , Bellman-Ford algorithm, approaching the given graph G with Dynamic Programming strategy, can handle weighted directed graphs contain negative weights. However, Dijkstra's algorithm, approaching with Greedy Algorithm, has to restrict the given weighted

directed graph without negative weights. Otherwise, it cannot promise to find an optimal solution.

In either algorithm, we need to initialize the shortest- path predecessors and estimates, which is an upper bound on the weight of the shortest path from the source vertex s to any other vertex v in given graph G (V, E), where s ∈V and v ∈V. We denote that v.p as shortest-path predecessor and v.e as the estimates. Then, we have the following $\theta(V) - \text{time}$ initialization procedure:

Initialize-single-source (G, s)

for each vertex v ∈G.V

v.e <-∞;

v.p<- ∅;

endfor

s.d = 0;

After initialization, the predecessors of all vertices are NULL, the shortest-path estimates for all vertices besides the source vertex s are assigned to infinity and the distance between source s and any other vertex v, where v ∈ V , equal to zero.

Then we will relax an edge e(u, v), where e ∈ E and u ,v ∈ V, by testing whether we can obtain a shorter path between v and u, if so, updating v.e and v.p. By relaxing, we arepossible to decrease the estimate v.e and update the predecessor v.p of vertex v. Let w (u, v) be the path cost between vertex u and v. We can form a relaxation on edge e(u, v) in O(1)-time with the following pseudocode：

Relax(u, v, w)

if v.e > u.e + w(u, v)

v.e = u.e + w(u,v);

v.p = u;

Some shortest-paths algorithms, such as Dijkstra's algorithm, where all edge weights in the given graph are nonnegative. Others, such as the Bellman-Ford algorithm, allow negative-weight edges in the given graph and produces an accurate answer as long as no negative-weight cycles are reachable from the source. Typically, if there is such a negative weighted cycle, the algorithm can detect and report its existence.

### 3.2.1 Bellman-Ford Algorithm [10]

Note that the content of Bellman-Ford Algorithmis rephrased from *Introduction to Algorithm* [6].

By applying dynamic programming, Bellman-Ford Algorithm typically finds the shortest path from source s to any vertex v in given graph G (V, E) where V is a set of vertices and s, v $\in$V, without negative circles but exists negative weights.

Given a weighted and directed graph G= (V, E) with source vertex s and weight function w: E ->R, the Bellman-Ford algorithm returns a Boolean value indicating whether there exists a negatively weighted cycle that is reachable from the source. If there exists such a cycle, the algorithm returns FALSE, which indicates that no solution exists. Otherwise, the algorithm returns the shortest paths and their weights and return TRUE. The algorithm returns TRUE if and only if the graph contains no negative-weight cycles that are reachable from the source.

Firstly, we create a distSet contains all the distances from vertex v to any other vertex in the graph and initialize all the distance to be infinity except the distance to source vertex s as zero.

The algorithm relaxes edges e (u, v), where e$\in$ E and u, v$\in$ V, sequentially decreasing an estimate v. e on the weight of a shortest path from the source s to each vertex v$\in$V . To simplify the notation, we denote the distance from vertex u as d[u], the distance from vertex v as d[v] and the weight of a path between u and v as we (u, v). Then for each edge e (u,v) in the given graph G, if d[u] adds w(u, v) is less than d[v], then we update the distance of d [v] as d[u]+ w(u, v). We traverse through all the vertices in the graph, relaxing each the edge until it achieves the actual shortest-path weight w(s, v).

Finally, by recalculating all the distances between each pair of vertices of an edge, we can check whether there exists negative weights circle. Negative weights circle exists if and only if d[v]< d[u] + w(u, v), then we return FALSE. Otherwise, we return TRUE. This algorithm can be presented with the following pseudocode:

```
BELLMAN-FORD(G, w, s)
  Initialize-single-source (G, s)
    for i <- 1 to |V[G]| - 1
          do for each edge (u, v)∈E
                do RELAX(u, v, w)
    for each edge (u, v) ∈E
          do if d[v] > d[u] + w(u, v)
                then return FALSE
    return TRUE
```

Implementation:

The code of Bellman-Ford algorithm is implemented following the pseudocode above with C++. To apply the algorithm for a given graph, we have to add more features than the above pseudocode. Additionally, we need to create a struct for edge e(u, v)∈E and vertex v ∈V. To create, operate and print the graph, we need to use links to connect the vertices from source vertex s to any other vertex v in the graph as well as to link edges from source vertex to any other edge e(u, v)∈E. We use an array to store the list of vertices and edges we create in the graph to apply further operation.

Firstly, we create our graph with method createGraph, which takes the input of a set of vertices, vertices' number and edges' number. We enable users to input the above data to create a new graph. When we have the Graph with vertices, edges and the ordering and predecessors of vertices and edges, we can apply our relaxation strategy

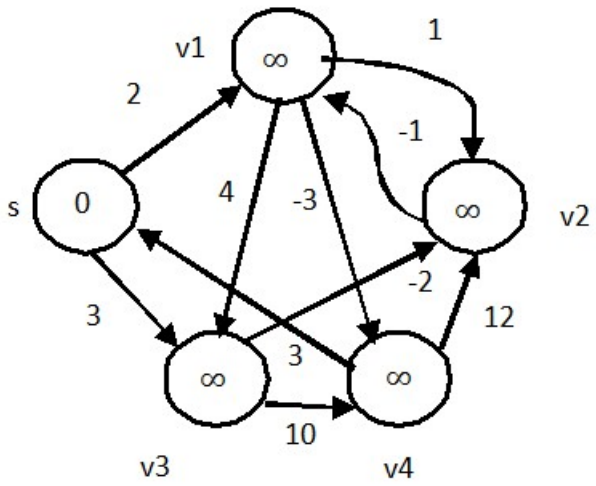
to obtain the lowest weight between any given pair of vertices u, v ∈V. Then we update the corresponding weights in the graph for each vertex in the set of vertices. After that, we check the existence of negative circle and return a Boolean to present whether there exists a solution for this given graph or not.

Finally, we create a printGraph method to print distances from the source vertex to all the other vertices in the set of vertices.

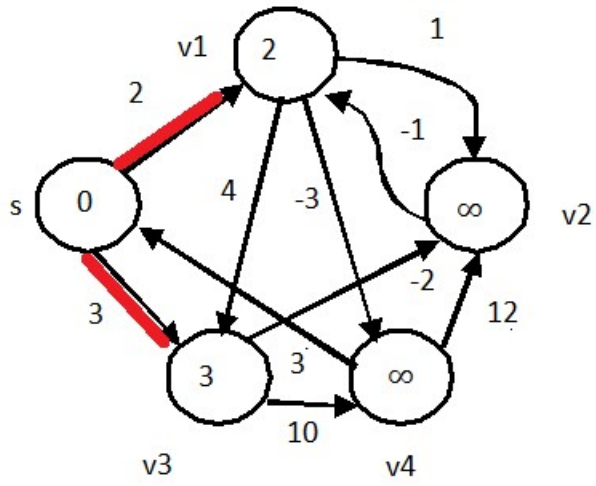
**Evaluation:**

Test the Bellman-Ford's algorithm with the experimental data in the image Bellman-Ford graph 1.

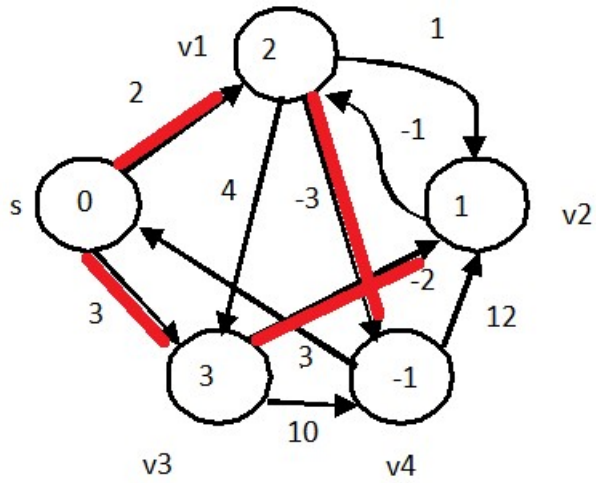
The algorithm can manage negative weights. Our approaching steps will process as in the following images from the Bellman-Ford graph 2 to Bellman-Ford graph 5. The algorithm will check through all the vertices and edges to ensure that there is no negative circle.



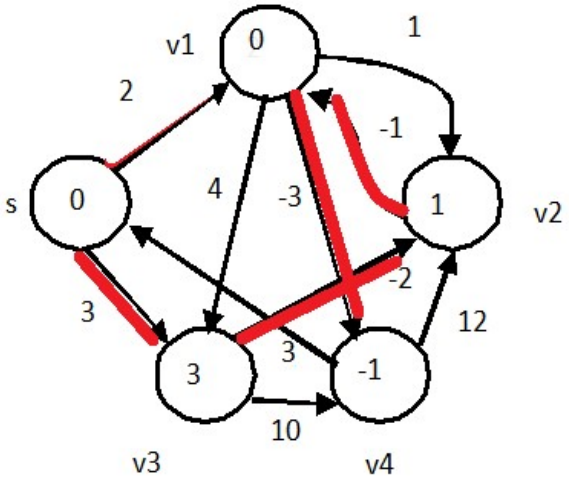

Bellman-Ford graph 1



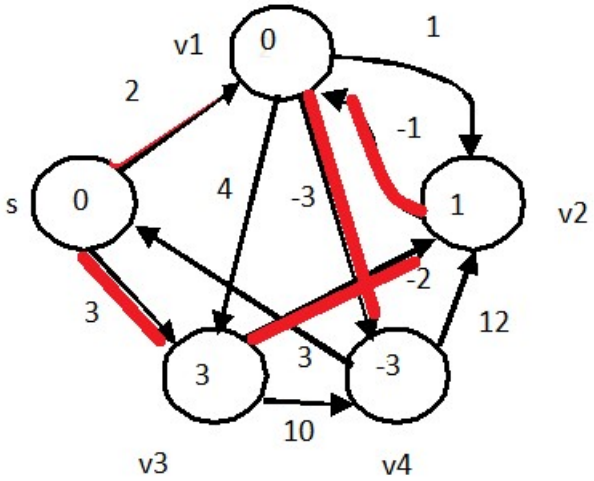

Bellman-Ford graph 2




Bellman-Ford graph 3




Bellman-Ford graph 4




Bellman-Ford graph 5

The result given by the implemented code matches the expected result in the graphs above. Particularly, we find the shortest path from source s to itself costs zero and it costs 3 to arrive v3 directly. Furthermore, it costs 1 with path (s, v3) and (v3, v2) from source s to v2, while costing 0 with path (s, v3), (v3, v2) and (v2, v1) to v1 and 3 with

path (s, v3), (v3, v2), (v2, v1) and (v1, v4) to v4.

```
Please input the number of tests: 1
Please input the number of vertexes: 5
Please input the number of edges: 10
Please input vertex 1 's No.: 1
Please input vertex 2 's No.: 2
Please input vertex 3 's No.: 3
Please input vertex 4 's No.: 4
Please input vertex 5 's No.: 5
Please input No. of edge 1 's source and destination vertexes: 1 2
Please input the weight of this edge: 2
Please input No. of edge 2 's source and destination vertexes: 1 3
Please input the weight of this edge: 3
Please input No. of edge 3 's source and destination vertexes: 2 3
Please input the weight of this edge: 4
Please input No. of edge 4 's source and destination vertexes: 2 5
Please input the weight of this edge: -3
Please input No. of edge 5 's source and destination vertexes: 2 4
Please input the weight of this edge: 1
Please input No. of edge 6 's source and destination vertexes: 3 4
Please input the weight of this edge: -2
Please input No. of edge 7 's source and destination vertexes: 4 2
Please input the weight of this edge: -1
Please input No. of edge 8 's source and destination vertexes: 3 5
Please input the weight of this edge: 10
Please input No. of edge 9 's source and destination vertexes: 5 1
Please input the weight of this edge: 3
Please input No. of edge 10 's source and destination vertexes: 5 4
Please input the weight of this edge: 12
Please input the No. of source vertex: 1
The shortest path from source vertex1to destination vertex 1is:
(1:1)
The cost of this path is: 0
The shortest path from source vertex1to destination vertex 2is:
(1:1)  (3:3)  (4:4)  (2:2)
The cost of this path is: 0
The shortest path from source vertex1to destination vertex 3is:
(1:1)  (3:3)
The cost of this path is: 3
The shortest path from source vertex1to destination vertex 4is:
(1:1)  (3:3)  (4:4)
The cost of this path is: 1
The shortest path from source vertex1to destination vertex 5is:
(1:1)  (3:3)  (4:4)  (2:2)  (5:5)
The cost of this path is: -3
```

As we search our shortest path through the graph on each vertex for all the adjacent edges, the time complexity of the Bellman-Ford algorithm is O (VE). Additionally, initialization takesθ(V) time, each of the |V | - 1 passes over the edges in the graphtake θ(E) time, and the“for” loop of finding shortest paths takes O(E) time. We can get the solution in polynomial time.

### 3.2.2 Dijkstra Algorithm [11]

3.2.1.1 Introduction

Approaching by Greedy algorithm strategy (Prim‘s algorithm), Dijkstra’s algorithm solves the single-source shortest-paths problem on a weighted, directed graph G = (V, E) where all edge weights are non-negative.

Note that the introduction, analysis and design parts of Dijkstra Algorithmis rephrased from *Introduction to Algorithms* [6].

3.2.1.2 Analysis

We shall notice that, with proper procedure, the running time of Dijkstra’s algorithm is less than that of the Bellman-Ford algorithm since Greedy Algorithm is much more efficient and simpler than dynamic programming manner.

Dijkstra’s algorithm has a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm continually selects the vertex u $\in$ V-S with the minimum shortest-path estimate, adds u to S, and relaxes all edges start from vertex u. To implement the algorithm, we shall use a min-priority queue Q of vertices, keyed by their d values.

Dijkstra’s algorithm is similar to both breadth-first search and Prim’s algorithm for computing minimum spanning trees. It is close to breadth-first search in that set S since vertices in S have their final shortest-path weights.

Dijkstra’s algorithm resembling Prim’s algorithm in that both algorithms use a min-priority queue to find the “lightest” vertex outside a given set S, add this vertex into the set, and relax the weights of the remaining vertices outside the set correspondingly. Typically, in Dijkstra’s algorithm, we always choose the “smallest” or “closest” vertex u in V - S to add to set S and keep searching minimum path cost from the selected vertex u in the remaining vertices’ set to determine our next choice.

3.2.1.3 Design

We can process this algorithm in following steps. Firstly, if there exists an edge e (s,v) from source vertex s to any vertex v, we denote that the direct edge e (s,v) as the initial shortest path, otherwise, the initial shortest –path from s to v is assigned to

infinity. Then we select any vertex u whose shortest path from s has not been determined and calculate the shortest path between s and u. Calculate all the distance from u to any other its adjacent nodes w, once we detect new paths (s,…,u)+(u,w)<(s,…,w), we replace (s,…,w) with (s,…,u,w). Repeat above steps until we obtain all the shortest paths from source vertex s to any other vertex u in the given graph G. We can describe the above procedures by the following pseudocode:

```
DIJKSTRA (G, w, s)
  INITIALIZE-SINGLE-SOURCE (G, s)
  S=∅
  Q =G.V
  while Q ≠ ∅
     u = EXTRACT-MIN (Q)
     S = S ∪ {u}
     for each vertex v ∈ G.Adjcentlist[u]
          RELAX (u, v, w)
```

3.2.1.4 Implementation

To implement Dijkstra's algorithm, we follow the steps in the above pseudocode, and we need more features to support the implementation. Additionally, we create a Node struct for vertices in vertex set and a SourceNode struct for the source vertex to start with. As described in Bellman-Ford algorithm's implementation, we generate our graph with inputs given by users. Therefore, we need to initialize our graph and storage the given information for each vertex in vertex set, the pair of ordering vertexes for each given edge as well as the given weight on each edge. Links order the vertices and edges of the graph.

To get the weight of the generated graph, we define a getWeight method to obtain the direct weight from the source vertex to reachable adjacent vertexes in our ordering vertex list. When all the precondition settled, we can implement the Dijkstra algorithm by repeatedly testing and finding the minimum weight from the source

vertex to other “known” vertexes in our min-priority queue. We always pick vertex enables the lowest cost from source vertex and add the chosen vertex to our vertex set by marking the Boolean value “isKnown” as true.

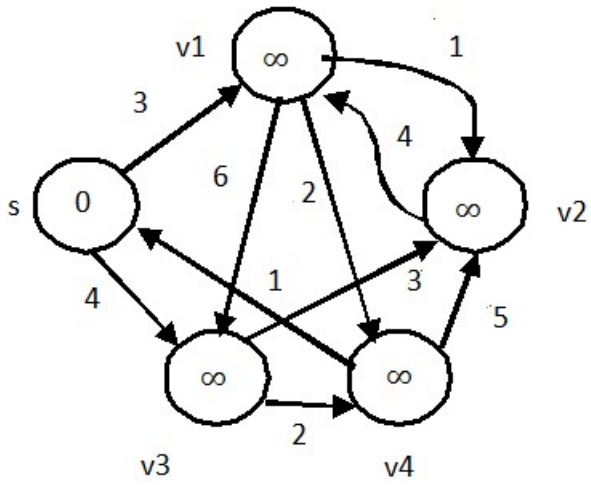
When the vertex is “known,” we apply min-heap to find the minimum weight among all its adjacent vertexes. Once we discovered a lighter path than the current path with the newly found minimum weight, we relaxed and updated the current path and weight.

Finally, we used printGraph and printPath methods to print the graph we generated from input data and the solution of shortest-path from s to each other vertex, separately.

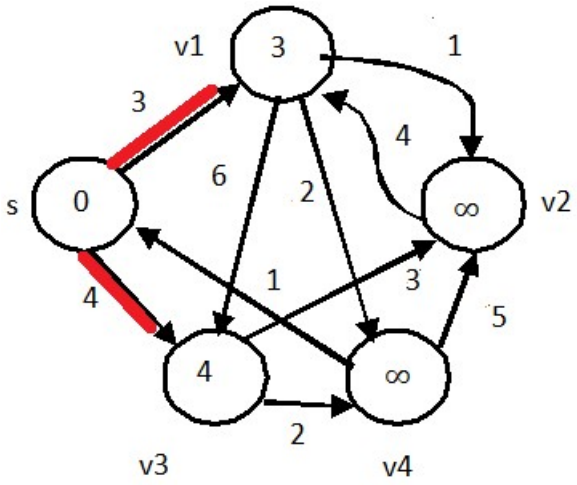
3.2.1.5 Evaluation

We test our Dijkstra’s algorithm with the following directed nonnegative weighted graph in the image Dijkstra graph 1

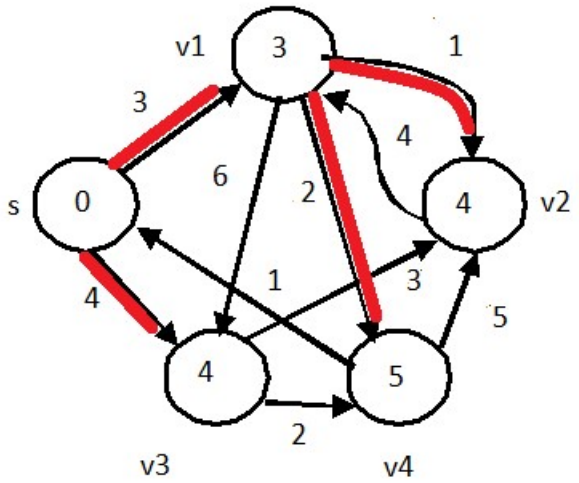
Our approaching steps will process as in the following images from Dijkstra graph 2 to Dijkstra graph 3.




Dijkstra graph 1




Dijkstra graph 2




Dijkstra graph 3

Our expected result is to find the shortest path from the source vertex to each other four vertexes with the minimum cost. Notably, we suppose to obtain a result that the shortest path from s to v1 costs three, while four to get from s to v3 directly. Furthermore, we expect to get from s to v2 with the path (s, v3) and (v3, v2) of cost four in total, and with the path(s, v3) and (v3, v4) of an overall cost five to reach v4.

Run the code of implementation, and we can get the following result:

```
Please input the number of vertexes and edges: 5 10
Build the graph(5, 10)
Please input source vertex, destination vertex and weight: 1 2 3
Please input source vertex, destination vertex and weight: 1 3 4
Please input source vertex, destination vertex and weight: 2 3 5
Please input source vertex, destination vertex and weight: 2 5 2
Please input source vertex, destination vertex and weight: 2 4 1
Please input source vertex, destination vertex and weight: 3 4 3
Please input source vertex, destination vertex and weight: 3 5 2
Please input source vertex, destination vertex and weight: 4 2 4
Please input source vertex, destination vertex and weight: 5 1 1
Please input source vertex, destination vertex and weight: 5 4 5
============================
Print the information of the Graph:
The Degree of vertex v1 is1, the edge start with this vertex is: v3(Weight:4)  v
2(Weight:3)
The Degree of vertex v2 is2, the edge start with this vertex is: v4(Weight:1)  v
5(Weight:2)  v3(Weight:5)
The Degree of vertex v3 is2, the edge start with this vertex is: v5(Weight:2)  v
4(Weight:3)
The Degree of vertex v4 is3, the edge start with this vertex is: v2(Weight:4)
The Degree of vertex v5 is2, the edge start with this vertex is: v4(Weight:5)  v
1(Weight:1)
============================
Call dijkstra alogorithm.
The current picked vertex is: v1
The current picked vertex is: v2
The current picked vertex is: v3
The current picked vertex is: v4
The current picked vertex is: v5
============================
Print the shortest path of vertexes start from v1:
The distance from v1 to v2 is 3: v1 -> v2
The distance from v1 to v3 is 4: v1 -> v3
The distance from v1 to v4 is 4: v1 -> v2 -> v4
The distance from v1 to v5 is 5: v1 -> v2 -> v5
```

The result prints in the above code is just matching our expectation where all the vertexes are picked and all the shortest- paths are found.

Note that Dijkstra's algorithm can only handle directed nonnegative weighted graph. Typically, we apply greedy algorithm (Prim's algorithm) to find minimum weight in each stage. Therefore, we can always find an optimal solution in a straightforward and efficient way with polynomial time- O ($V^2$). Additionally, we can obtain an even optimal running time of O ($E \lg V$) if edges E = o ($V^2 / \lg V$).

Since the total number of edges in the adjacency lists is |E|, for loop iterates |E| times, such that the algorithm calls DECREASE-KEY at most |E| times overall. The running time of Dijkstra's algorithm depends on how we implement the min-priority queue. Consider that we constructed the min-priority queue by taking advantage of the vertices being indexed 1 to |V|. We simply store the v.d in the vth entry of an array. Each insertion and DECREASE-KEY operation takes O (1) time, and each

EXTRACT-MIN operation takes O (V) time. Thus, we gain an overall running time of $O(V^2 + E) = O(V^2)$. However, when the graph is sufficiently sparse, particularly, $E = o(V^2/\lg V)$, we can reduce the execution time to $O(E \lg V)$ by implementing the min-priority queue with a binary min-heap.

Each EXTRACT-MIN operation then takes time O ($\lg V$). For |V| such operations, we can build our binary min-heap in O (V) time. As there are at most |E| such operations and each DECREASE-KEY operation takes time O (lg V), the total running time is, therefore, O ((V+E) lgV). If all vertices are reachable from the source, we reduce the running time to O (E lgV).

Furthermore, we can possibility achieve a better running time of O (VlgV + E) by implementing the min-priority queue with a Fibonacci heap. Note that the amortized cost of each of the |V| EXTRACT-MIN operations is O (lg V), and it takes only O (1) amortized time for at most |E| numbers of DECREASEKEY call.

### 3.3 All-Pair Source Shortest Path Problems [12]

Note that all the content of All-Pair Source Shortest Path Problems are rephrased from *Introduction to Algorithm* [6].

By running a single-source shortest-paths algorithm |V| times, once at each vertex, we can solve an all-pairs shortest-paths problem. For a directed nonnegative weighted graph, we can apply Dijkstra's algorithm with the implementation of the min-priority queue in $O(V^3 + VE) = O(V^3)$ overall running time.Moreover, we can obtain a running time of $O(VE \lg V)$ by implementing a min-priority queue with the binary min-heap. Furthermore, we canyield a running time of $O(V^2 \lg V + VE)$ by implementing the min-priority queue with a Fibonacci heap.

For directed weighted graph existing negative weight edges, we must run Bellman-Ford algorithm once for each vertex, yielding a running time of $O(V^2E)$, which is $O(V^4)$on a dense graph.

Instead of representing the graph with an adjacency-list, applying in single-source algorithms, typically, we present the graph by an adjacency matrix in all-pair source problems. We denote that the vertices are indexed 1, 2, … ,|V|, so that the input is a

n*n matrix M representing the edge weights of a directed graph G= (V, E) with n vertexes. That is, M= (wij), where

$$w(i,j) = \begin{cases} 0 & \text{if } i = j \\ \text{the weight of the directed edge } \langle i,j \rangle & \text{if } i \neq j \text{ and } \langle i,j \rangle \in E \\ \infty & \text{if } i \neq j \text{ and } \langle i,j \rangle \notin E \end{cases}$$

For the given graph G, we allow the existence of negative-weight edges, but no negative-weight cycles.

Use Dynamic programming strategy to process all-pairs shortest-paths problem on an input directed graph G = (V, E), which representing as an adjacency matrix, and we need to compute not only the shortest-path weights but also a ***predecessor matrix***.

Recall we used a matrix to speed up our execution in Chapter 2 (2.1 the Fibonacci problem). Each loop of the Dynamic Programming will invoke an operation that is very similar to matrix multiplication so that the algorithm processing as repetitive matrix multiplicationIn this case, we shallimprove its running time to $\theta(V^3 \lg V)$.

**1. The structure of a shortest path**

For the all-pairs shortest-paths problem on a graph G=(V,E) without negative-weight cycles, all sub-paths of a shortest path are shortest paths as well. Assume that we represent the graph by an adjacency matrix M(wij). Applying the Dynamic Programming strategy, the structure of this problem is following. For any two vertices u and v, if u=v, then the shortest path p from u to v costs 0. Otherwise, we decompose p into u→x→v, where p' is a path from u to x, which including at most k edges, and it is the shortest path from u to x.Consider a shortest path pijfrom vertex i to vertex j , and suppose that pijcontains at most m edges, where m is finity. If i = j, thenpijhas weight 0 and no edges. If vertices i and j are different, then $\delta(i,j) = \delta(i,k) + wkj$ and pijis constructed with at most m - 1 edges.

**2. A recursive solution to the all-pairs shortest-paths problem**

Let$w^{(m)}$ ij be the minimum weight of any path from vertex i to vertex j that contains at most m edges. We can present the above analysis with the following formulas:

$$w^{(m)}\,ij = \begin{cases} 0 & \text{if } i = j, \\ \infty & \text{if } i \neq j. \end{cases}$$

$$w^{(m)}\,ij = \min\left(w^{(m)}\,ij,\ \min_{1\le k\le n}\left\{w^{(m-1)}\,ik + wkj\right\}\right)$$

$$= \min_{1\le k\le n}\left\{w^{(m-1)}\,ik + wkj\right\}$$

$$\delta(i,j) = w^{(n-1)}\,ij = w^{(n)}\,ij = w^{(n+1)}\,ij = \dots .$$

3. **Computing the shortest-path weights bottom up**

If wesolve an all-pairs shortest-paths problemby running a single-source shortest-paths algorithm |V| times,once for each vertex. We can construct the following code:

Extend- Shortest-Paths (P, M)

$n = P.\text{rows}$

let $P' = (w'ij) be\ a\ new\ n * n\ matri$

for $i = 1$ to n

for $j = 1$ to n

$w^{(m)}\,ij = \infty$

for $k = 1$ to n

$w^{(m)}\,ij = \min\left(w^{(m)}\,ij, \min_{1\le k\le n}\left\{w^{(m-1)}\,ik + wkj\right\}\right)$

return $P'$

We can easily find its running time is $\theta(n^3)$due to the three nested **for** loops

From the pseudocode above, we can see its relation to matrix multiplication. Suppose we wish to computethe matrix product C = A * B of two n * n matrices A and B. Then, for i, j = 1, 2, … , n, we compute cij $= \sum_{k=1}^{n} aik * bkj$ .

Naturally, we shall transform our shortest path problem into a matrix multiplication format in following way; we denote our minimum weight $w^{(m-1)}$ as a, weight w as b, minimum weight$w^{(m)}asc$. Additionality, the operation of min in our shortest path problem is substituted by 'plus (+),' while operation 'plus (+)' by 'multiply (*).' Then, we get the following code:

```
MATRIX-MULTIPLY (A,B)

n ←  rows  [A]

C ← new n×n matrix

for  i ←  1 to  n

    do for  j ←  1 to  n

        do  cij  ←  ∞

            for  k ←  1 to  n

                do  cij  ←  min( cij  , aik  + bkj  )

 return  C
```

The optimal solution can be computed by calling MATRIX-MULTIPLY (A, B) for 1≤k≤n-2. We only need to run to n-2 because that will give us A(n-1). That is, it will give us all the shortest path lengths of at most n-1 edges since only n-1 edges are required to connect n vertices. SinceMATRIX-MULTIPLY is called n-2 times, the total running time is O($n^4$).

**4. Improving the running time**

The repeated squaring method:

Each M(k) matrix contains the shortest paths of at most k edges, and we denote that W as M(1). All we did in the previous solution was to find the shortest paths of at

most length k+1 with the given shortest paths of at most length k, and at most length 1. Now we are ready to improve this problem into finding the shortest paths of at most length k+k with the given shortest paths of at most length k. The repeated squaring method enables this improvement. The correctness of this approach lies in the observation that the shortest paths of at most m edges is the same as the shortest paths of at most n-1 edges for all m>n-1. Thus:

M(1) =W

M(2) = W2 =W*W

M(4) = W4 = W2* W2

…

M( 2⌈lg(n-1)⌉ ) = W( 2⌈lg(n-1)⌉ ) = W( 2⌈lg(n-1)⌉ -1) *W( 2⌈lg(n-1)⌉-1)

Using repeated squaring, we only need to runMATRIX-MULTIPLY ⌈lg(n-1)⌉ times. Hence the running time of the improved matrix multiplication is $O(n^3 \lg n)$.

The following procedure computes the above sequence of matrices by using this technique of ***repeated squaring***.

ALL-PAIRS-SHORTEST-PATHS (W)

n ← rows [W]

M(1) ← W

m ← 1

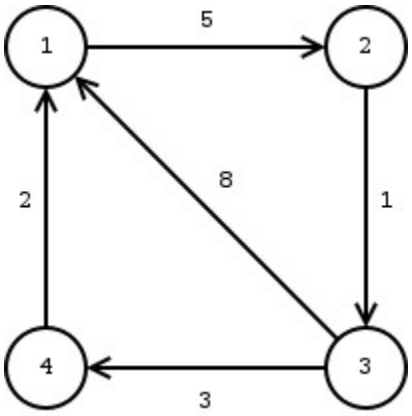
while m<n-1

do M(2m) ← MATRIX-MULTIPLY ( M(m), M(m))

m ← 2m

return M(m)

Observe that the code is tight, containing no elaborate data structures, and the constant hidden in the θ-notation is therefore small.




$$M(1) = W = \begin{pmatrix} 0 & 5 & \infty & \infty \\ \infty & 0 & 1 & \infty \\ 8 & \infty & 0 & 3 \\ 2 & \infty & \infty & 0 \end{pmatrix}$$

$$M(2) = W2 \ = W * W = \begin{pmatrix} 0 & 5 & 6 & \infty \\ 9 & 0 & 1 & 4 \\ 5 & 13 & 0 & 3 \\ 2 & 7 & \infty & 0 \end{pmatrix}$$

$$M(4) = \ \mathrm{W4} \ = \mathrm{W2} * \mathrm{W2} = \begin{pmatrix} 0 & 5 & 6 & 9 \\ 6 & 0 & 1 & 4 \\ 5 & 10 & 0 & 3 \\ 2 & 7 & 8 & 0 \end{pmatrix}$$

### 3.3.1 Floyd-Warashall Algorithm [13]

#### 3.3.1.1 Introduction

Note that all the introduction, analysis and design parts of Floyd-Warashall Algorithm is rephrased from *Introduction to Algorithm* [6].

Floyd-Warshall's algorithm is based on the observation that a path linking any two vertices u and v may have zero or more intermediate vertices. The algorithm initially

denies all the intermediate vertices. In this case, the sub-solution is simply the original weights of the graph or infinity if there is no edge.

The algorithm allows an additional intermediate vertex at each step.That is to introduce a new intermediate vertex x between any pair of vertices u and v, where $x, u, v \in V$ . In this way, the shortest path from u to v is either the minimum of the previous best estimate of $\delta(u, v)$ or the combination of the paths from $u \to x$ and $x \to v$ . We can represent this relation with following formula:

$$\delta(u, v) \leftarrow \min\big(\delta(u, v), \delta(u, x) + \delta(x, v)\big)$$

***Floyd-Warshall algorithm***, runs in$\theta(V^3)$time on a directed weighted graph G (V, E), allowing negative-weight edges but without negative-weight cycles. We follow the Dynamic Programming process to develop the algorithm and adopt a similar method to find the transitive closure of a directed graph.

### 3.3.1.2 Analysis&Design

1. **The structure of a shortest path**

The Floyd-Warshall algorithm studies the intermediate vertices of a shortest path. Note that an intermediate vertex of a simple path p=<v1, v2, … , vk> is any vertex in set p other than v1 or vk. That is, any vertex in the set p'= {v2, v3, … , v(k-1)}.

Typically, the Floyd-Warshall algorithm explores a relationship between path p and shortest paths from i to j with all intermediate vertices in the set {1, 2, … , k-1}. The relationship depends on whether there exists ak as an intermediate vertex of path p.

Therefore, we depart the problem into two individual cases:

- If k is not an intermediate vertex of path p, then all intermediate vertices of path p are in the set {1, 2, … , k-1}. In this case, a shortest path from vertex i to vertex j with all intermediate vertices in the set {1, 2, … , k-1} is a shortest path from i to j with all intermediate vertices in the set {1, 2, … , k} as well.
- If k is an intermediate vertex of path p, we can divide *p* at *k* giving two

subpaths *p*1 and *p*2, where p1 is from a shortest path vi to vk and p2 is a shortest path from vk to vj. Since vertex k is not an intermediate vertex of path p1, all intermediate vertices of p1 are in the set {1, 2, … , k-1}. Therefore, p1 is a shortest path from i to k with all intermediate vertices in the set{1, 2, … , k-1}. Correspondingly, p2 is a shortest path from vertex k to vertex j with all intermediate vertices in the set {1, 2, … , k-1}.

2. A recursive solution to the all-pairs shortest-paths problem

We define a *d*(k)ij as the minimum weight of the path from vertex *i* to vertex *j* with intermediate vertices drawn from the set {1, 2, ..., *k*}. Then the above properties give the following recursive formula:

$$\mathrm{d}^{(\mathrm{k})}\ \mathrm{ij} = \begin{cases} wij & \text{if } \mathrm{k} = 0, \\ \min\left(\mathrm{d}^{(\mathrm{k}-1)}\ \mathrm{ij}, \mathrm{d}^{(\mathrm{k}-1)}\ \mathrm{ik} + \mathrm{d}^{(\mathrm{k}-1)}\ \mathrm{kj}\right) & \text{if } \mathrm{k} \geq 1. \end{cases}$$

Thus we can represent the optimal values (when $k = n$) in a matrix as

$$\mathrm{D}^{(\mathrm{k})} = \mathrm{d}^{(\mathrm{n})}ij\ givesthefinalanswer\colon \mathrm{d}^{(\mathrm{n})}ij = \delta(\mathrm{i},\mathrm{j})\ fo\ \ all\ i,j\ \in \mathrm{V}$$

3. Computing the shortest-path weights bottom up

Compute the matrix D of shortest-path weights and then construct the predecessor matrix from the D matrix. We can give the following recursive formulation of the predecessor of vertex j,with all intermediate vertices in the set {1, 2, … , k},on a shortest path from vertex i.

$$\pi^{(0)}\ \mathrm{ij} = \begin{cases} NULL & \text{if i} = \text{j or wij} = \infty, \\ & \text{iif i} \neq \text{j and wij} < \infty. \end{cases}$$

$$\pi^{(\mathrm{k})}\ \mathrm{ij} = \begin{cases} \pi^{(\mathrm{k}-1)}\ \mathrm{ij} & \text{if } \mathrm{d}^{(\mathrm{k}-1)}\ \mathrm{ij} \leq \mathrm{d}^{(\mathrm{k}-1)}\ \mathrm{ik} + \mathrm{d}^{(\mathrm{k}-1)}\ \mathrm{kj}, \\ \pi^{(\mathrm{k}-1)}\ \mathrm{kj} & \text{if } \mathrm{d}^{(\mathrm{k}-1)}\ \mathrm{ij} > \mathrm{d}^{(\mathrm{k}-1)}\ \mathrm{ik} + \mathrm{d}^{(\mathrm{k}-1)}\ \mathrm{kj} \end{cases}$$

4. **Constructing a shortest path**

FLOYD-WARSHALL(W)

n = W.rows

$\mathrm{D}^{(0)} = \mathrm{W}$

$\Pi^{(0)} = \pi^{(0)}\ \mathrm{ij} = \mathrm{NULL},\ \text{if i} = \text{j or wij} = \infty,$

$\Pi^{(0)} = \pi^{(0)}$ ij= i, if i ≠ j and wij $< \infty$.

for k = 1 to n

letD$^{(k)}$ = d$^{(k)}$ ij be a new nxn matrix

for i = 1 to n

for j = 1 to n

$d^{(k)}$ ij = $\min(d^{(k-1)}$ ij, $d^{(k-1)}$ ik + $d^{(k-1)}$ kj$)$

if $d^{(k-1)}$ ij $\leq d^{(k-1)}$ ik + $d^{(k-1)}$ kj

$\pi^{(k)}$ ij = $\pi^{(k-1)}$ ij

else

$\pi^{(k)}$ ij = $\pi^{(k-1)}$ kj

return D(n)

The algorithm processes by repeatedly exploring paths between every pair using each vertex as an intermediate vertex.

Therefore, the running time of the Floyd-Warshall algorithm is determined by the triply nested for loops, that is, $\theta(n^3)$.

## 5. Transitive closure of a directed graph

We define the transitive closure of G as the graph G* = (V, E*), where E* = {(i, j) : there is a path from vertex i to vertex j in G}.

Floyd-Warshall algorithm can be used to determine whether or not a graph has transitive closure, which means whether or not there are paths between all vertices. To test the existence of transitive closure in $\theta(n^3)$ time, we can assign all edges in the graph to have weight =1 and then we run the Floyd-Warshall algorithm. If there is a path from vertex i to vertex j, we get dij < n. Otherwise, we get dij =∞. We have *alld*ij <*n* then we obtain a transitive closure

This procedure can implement a slightly more efficient algorithm through the use of logical operators rather than min() and '+.'

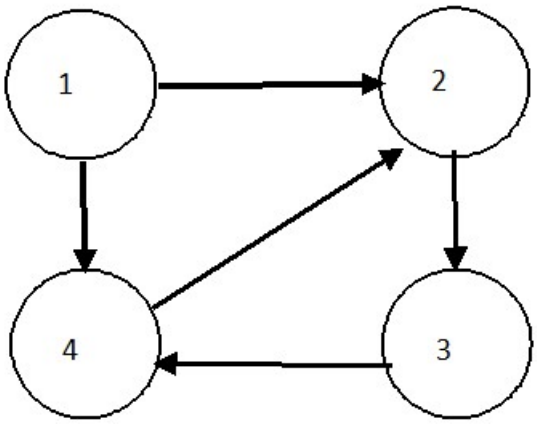
$$t^{(0)}\,ij = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i,j) \text{ not} \in E \\ 1 & \text{if } i = j \text{ or } (i,j) \in E \end{cases}$$

and for $k \geq 1$, $\quad t^{(k)}\,ij = t^{(k-1)}\,ij \vee (t^{(k-1)}\,ik \wedge t^{(k-1)}\,kj)$.

The transitive closure of a directed graph G can be determined by the following algorithm.

```
TRANSITIVE-CLOSURE (G)
n ← |V|
Let T(0) = t(0) ij be a new n*n matrix
for i ← 1 to n
    do for j ← 1 to n
        do if i = j or (i, j) ∈ E
            then t(0) ij ← 1
            else t(0) ij ← 0
for k ← 1 to n
    Let T(k) = ( t(k) ij) be a new n*n matrix
    do for i ← 1 to n
        do for j ← 1 to n
            do t(k) ij = t(k−1) ij ∨ (t(k−1) ik ∧ t(k−1) kj)
Return T(n)
```

We can present the above pseudocode in the following graph by gradually approaches:




$$T^{(0)} = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

$$T^{(1)} = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

$$T^{(2)} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}$$

$$T^{(4)} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}$$

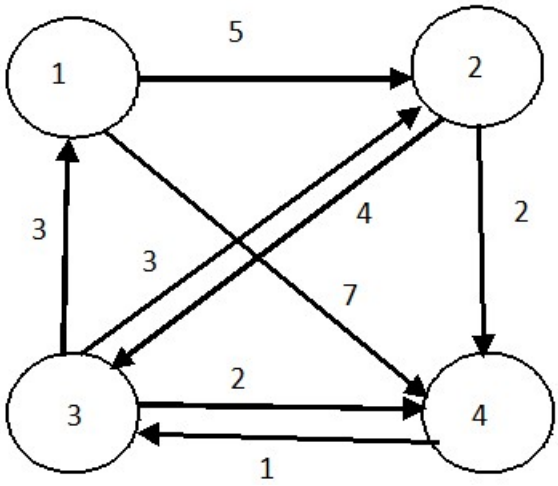
### 3.3.1.3 Implementation

Not covering in any past lecture, and with the introduction of Matrix Multiplication, the implementation of Floyd algorithm is not trivial. Although we construct the code in a quite similar way as we did in single-source shortest-path algorithms, we have to figure out an efficient way to implement the representation of our graph- matrix. With the prior learning and coding experience in Chapter 2 - 2.1, where we speed up the implementation of Fibonacci number problem with Matrix Multiplication method, we can solve this issue with a stack. Different from inputting vertices and weights, we construct a graph matrix- arrMatrix- and denote arbitrary integer as the weight of an edge between the corresponding indexes of vertices, 10000 as no edge between the two vertexes and 0 as the vertex itself. For instance, if arrMatrix[3][1]=1, that is, there exists an edge between vertex 3 and vertex 1. Then, we apply the Floyd algorithm as we analyzed above, searching the shortest path through the matrix with three nested for loops. To print our graph in correct order, typically, we apply stack-verticeSet with a first in last out order. As we are not interested in vertices on the diagonal, we only use the stack to store and pop up vertices other than those on the diagonal. We push all vertices in our stack, and we pop up the vertex, which not appears on diagonal, on top of our stack each time. In this way, we finally get our shortest paths among each pair of vertexes.

#### 3.3.1.4 Evaluation

Note that Floyd algorithm is able to handle directed weighted graph without negative circles but allowing negative weights.

We test our Floyd algorithm with the following graph.



s

$$D^{(0)}=\begin{bmatrix}0 & 5 & \infty & 7\\ \infty & 0 & 4 & 2\\ 3 & 3 & 0 & 2\\ \infty & \infty & 1 & 0\end{bmatrix}\ \Pi^{(0)}=\begin{bmatrix}-1 & -1 & -1 & -1\\ -1 & -1 & -1 & -1\\ -1 & -1 & -1 & -1\\ -1 & -1 & -1 & -1\end{bmatrix}$$

$$D^{(1)}=\begin{bmatrix}0 & 5 & 9 & 7\\ \infty & 0 & 4 & 2\\ 3 & 3 & 0 & 2\\ \infty & \infty & 1 & 0\end{bmatrix}\ \Pi^{(1)}=\begin{bmatrix}-1 & -1 & -1 & -1\\ -1 & -1 & -1 & -1\\ -1 & -1 & -1 & -1\\ -1 & -1 & -1 & -1\end{bmatrix}$$

$$D^{(2)}=\begin{bmatrix}0 & 5 & 9 & 7\\ 7 & 0 & 4 & 2\\ 3 & 3 & 0 & 2\\ 4 & 4 & 1 & 0\end{bmatrix}\ \Pi^{(2)}=\begin{bmatrix}-1 & -1 & -1 & -1\\ 2 & -1 & -1 & -1\\ -1 & -1 & -1 & -1\\ 2 & 2 & -1 & -1\end{bmatrix}$$

$$D^{(3)}=\begin{bmatrix}0 & 5 & 8 & 7\\ 6 & 0 & 3 & 2\\ 3 & 3 & 0 & 2\\ 4 & 4 & 1 & 0\end{bmatrix}\ \Pi^{(3)}=\begin{bmatrix}-1 & -1 & 3 & -1\\ 3 & -1 & 3 & -1\\ -1 & -1 & -1 & -1\\ 2 & 2 & -1 & -1\end{bmatrix}$$

```
Please input the number of vertexes and edges: 4 8
Please input row and col of adjacent matrix:
0 5 10000 7
10000 0 4 2
3 3 0 2
10000 10000 1 0
Source -> Dest   Distance    Path
1 -> 2          5               1 -> 2
1 -> 3          8               1 -> 4 -> 3
1 -> 4          7               1 -> 4
2 -> 1          6               2 -> 4 -> 3 -> 1
2 -> 3          3               2 -> 4 -> 3
2 -> 4          2               2 -> 4
3 -> 1          3               3 -> 1
3 -> 2          3               3 -> 2
3 -> 4          2               3 -> 4
4 -> 1          4               4 -> 3 -> 1
4 -> 2          4               4 -> 3 -> 2
4 -> 3          1               4 -> 3
```

We print the results row by row, column by column.

As we expected, the Floyd algorithm provides the solution of shortest path and weight between each pair of vertexes. In detail, we can get from vertex 1 to vertexes 2 and 4 directly with the distance of five and seven separately. Pass through intermediate vertex 4, from vertex 1 to vertex 3 with an overall distance of eight. Similarly for other vertices. We note that matrix represents the input simpler and more straightforward than we did before by inputting source vertex, destination vertex and weight of each edge in the graph.

### 3.3.2 Johnson's Algorithm [14]

#### 3.3.2.1 Introduction

Note that all the introduction, analysis and design parts of Johnson's Algorithm [14] are rephrased from *Introduction to Algorithm* [6].

Johnson's algorithm works best for sparse graphs.

Johnson's algorithm is an algorithm that applies to solve the all pairs shortest path problem on a graph G(V,E) and it outputs the shortest path between each pair of vertices in that graph. Johnson's algorithm is very analogous to the Floyd-Warshall algorithm. Nevertheless, Floyd-Warshall's algorithm is most effective for dense

graphs (many edges), while Johnson's algorithm is more effective for sparse graphs (few edges). Johnson's algorithm works better for sparse graphs since its time complexity relies on the number of edges in the graph, while Floyd-Warshall's algorithm does not.

Johnson's algorithm runs in $O(V^2 \lg V + V * E)$ time. For sparse graphs, it is significantly faster than either repeated squaring of matrices or the Floyd-Warshall algorithm. Besides, we take Johnson's algorithm as a combination of all previous shortest path algorithms we have seen before because it uses two other shortest path algorithms as subroutines. It uses Bellman-Ford to eliminate negative edges and detect negative cycles in the input graph. Take the altered graph without negative weights as new input, it then uses Dijkstra's algorithm to calculate the shortest path between all pairs of vertices.

The algorithm either returns a matrix of shortest-path weights for all pairs of vertices in the original graph or reports that the input graph contains a negative-weight cycle.

#### 3.3.2.2 Analysis

Johnson's algorithm re-weight all edges and make them all positive, then apply Dijkstra's algorithm for every vertex. The technique of ***reweighting*** works as follows.

If given graph G=(V, E) contain no negative weight edges, performingDijkstra's algorithmonce from each vertex to find shortestpaths between all pairs of vertices in $O(V^2 \lg V + V * E)$time.

If, however, G has negative-weight edges but no negative-weight cycles, we compute a new set of nonnegative edge weights. We denote the new set of edge weights as w so that the following constraint holds. For all pairs of vertices u, v in the graph, weight (u, v) must be non-negative, and if a path is the shortest path between those vertices before reweighting, it must be the shortest path between those vertices after reweighting as well.

As we search through the whole graph on each vertex and its joint edges, we can determine the new weight function w in O(VE) time.

1. **Preserving shortest paths by reweighting**

Johnson's algorithm initially assigns a weight to every vertex.

Given a weighted, directed graph G = (V, E) with weight function W and denote h as any function mapping vertices to real numbers. For each edge (u, v) ∈V, we define that W (u, v) = w (u, v) + h(u) − h(v). Let p = <v0, v1, … , vk> be any path from vertex $v0$ to vertex $vk$. Then p is a shortest path from $v0$ to $vk$ with weight function w only if it is a shortest path with weight function W.W(p) $= \sum_{i=1}^{k} W(vi - 1, vi)$

$$= \sum_{i=1}^{k} \big(w(vi-1, vi) + h(vi-1) - h(\text{vi})\big)$$

$$= \sum_{i=1}^{k} w(vi-1, vi) + h(v0) - h\big((vk)\big)$$

$$= w(p) + h(v0) - h(vk)$$

So, any path from v0 to vk that is a shortest path using the original weight function $w(p)$ is also a shortest path using the reweighting function $(p)$ . This is because $h(v0)$ and $(vk)$ not depend on the path.

Finally, we show that G has a negative-weight cycle using weight function w if and only if G has a negative-weight cycle using weight function W. So we can detect negative weight cycles after the transformation

Considering any cycle $c = < v0, v1, \ldots, vk >$where v0 = vk and what we just proved in the step above,

We have:

$$W(c) = w(c) + h(v0) - h(vk) = w(c).$$

So, any cycle c that has a negative weight value w before the transformation will also have one W after reweighting.

2. **All reweighted edges must have a non-negative value**

Given a weighted, directed graph G = (V, E)with weight function W,we want W (u, v) to be nonnegative for all edges (u, v) ∈ E. Therefore, we make a new graph G' = (V', E'), where V'=V ∪ {s} for that new vertex s not inV and E' =E∪ {(s, v): v ∈ V}.Then, we extend the weight function w so that w (s, v) = 0 for all v∈ V. Moreover, G'has no negative-weight cycles if and only if G has no negative-weight cycles.

Now suppose that both G and G' have no negative-weight cycles. Define h (v) = $\text{distance}(s, v)$ for all v $\in V'$.By the triangle inequality, we have h (v)$\leq$ $h(u) + w(u, v)$ for all edges (u, v) ∈ E'. Thus, if we define the new weights W by reweighting, we have W (u, v) = w (u, v) + h (u) – h (v) $\geq$0. In this case, we held our definition that all reweighted edges must have a non-negative value

### 3. **Computing all-pairs shortest paths**

Johnson's algorithm to compute all-pairs shortest paths uses the Bellman-Ford algorithm and Dijkstra's algorithm as subroutines.By using Bellman-Ford to help reweight the edges, Johnson's algorithm is able to detect negative weight cycles, a property of Bellman-Ford. Then, Dijkstra's algorithm is run on all vertices to find the shortest path. This is possible because the weights have been transformed into non-negative weights. Finally, we have to convert these path weights back into the original path weights so that we can return an accurate path weight as output. This is achieved by reversing the reweighting process, and we return a data structure like a matrix. At every cell d(u,v),where d(u,v)=$\text{distance}(u, v)$ of this matrix D is the shortest path from vertex u to vertex v. We typically assume that the vertices are numbered from 1 to |V|. Then, the algorithm returns the usual |V| * |V| matrix Dor it reports that the input graph contains a negative-weight cycle.

#### 3.3.2.3 Design

Johnson's algorithm works on directed, weighted graphs G(V, E). It does allow edges to have negative weights, but there can be no negative weight cycles.

Johnson's algorithm re-weight all edges with Bellman-Ford's algorithmto make them all positive, then apply Dijkstra's algorithm for every vertex.

There are mainly four steps to apply in Johnson's algorithm:

1. Add a new vertex to the graph, and edges of zero weight connect it to all other

vertices in the graph.

2. Run Bellman-Ford algorithm on G' on the source s. If we find a negative weight cycle, then return the Boolean value. We would not create a negative weight cycle from new vertex s since there is no edge to s.
3. Re-weight all edges in the original graph to eliminate negative weight edges. For each edge (u, v), assign the new weight as W(u, v) = weight(u, v) + h(u) - h(v)
4. Remove the added vertex in step 1 and Dijkstra's algorithm is run on every vertex in the graph.

With the analysis above we can create following pseudocode:

```
Johnson(G, w)
    1.
    create G' where G'.V = G.V + {s},
        G'.E = G.E + ((s, u) for u in G.V), and
        weight(s, u) = 0 for u in G.V
    2.
    if Bellman-Ford(G) == false
        return "The input graph has a negative weight cycle"
    3.
    else:
        for vertex v in G'.V:
            h(v) = distance(s, v) computed by Bellman-Ford
        for edge (u, v) in G'.E:
            W(u, v) = weight(u, v) + h(u) - h(v)
```

4.

```
D = new matrix of distances initialized to infinity
for vertex u in G'.V:
    run Dijkstra(G, W, u) to compute distance'(u, v) for all v in G.V
    for each vertex v in G.V:
        d(u, v) =distance'(u, v) + h(v) - h(u)
return D
```

### 3.3.2.4 Implementation

Johnson's algorithm is relatively hard to implement. This algorithm was not introduced in any previous lecture, so it is completely new knowledge. Besides, the structure of this algorithm is complicated as it composes with an amount of other algorithms and technical methods. Specifically, as analyzed above, we need to detect negative weight circle and re-weight edges with Bellman-Ford Algorithm. Then, search shortest paths among each pair of vertexes with Dijkstra's algorithm. Particularly, we apply Dijkstra's algorithm with the support of a Fibonacci heap as its minimum priority queue, which is the most efficient data structure of the algorithm. Have to note that Fibonacci heap is also not covered in previous lectures. It requests additional learning in this technic manner. Similar to our implementation of Floyd-Warshall algorithm, Johnson's algorithm represents the graph with two-dimensional graph matrix, which implicates the introduction of adjacency list to store the list of our vertices. With the above pre-requirements in mind, we can implement Johnson's algorithm.

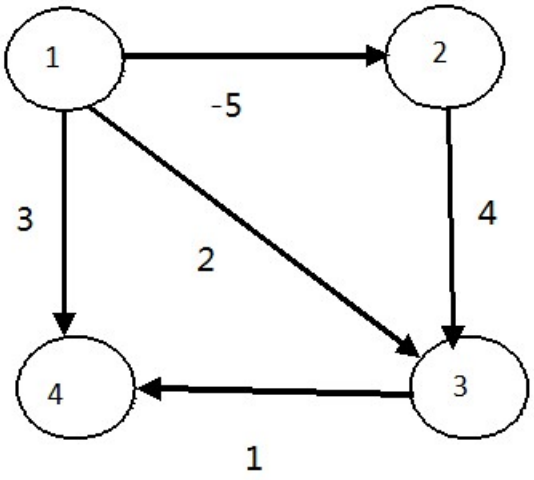
As we did for all short-path problems before, we define three structs for edges, vertices and priority queue separately. We define an adjacent list-adjlist- to store all vertices in the graph. Then, we can create a graph with the user's input. As in step 1 in our pseudocode, we add an extra vertex with index 0. Connect with each other vertex

in the graph with newly added vertex 0 and assign all the edges from vertex 0 to other vertices to weight 0.

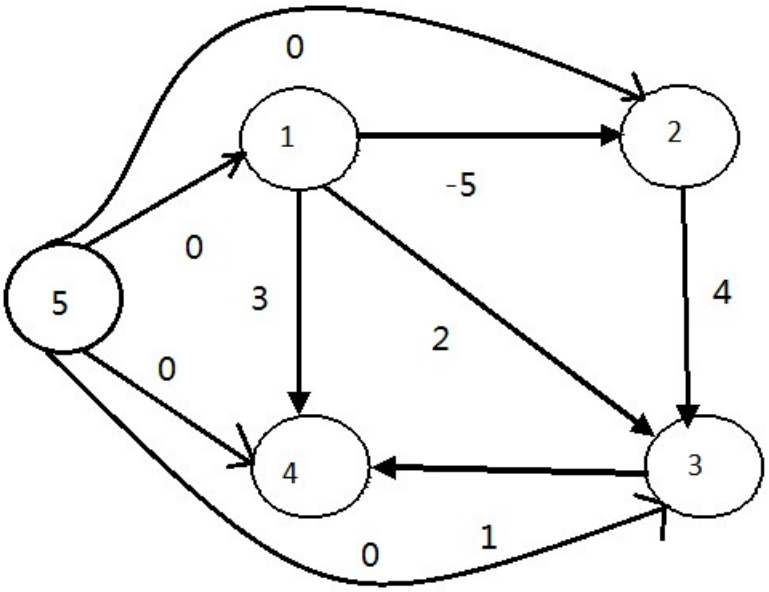
As we solved in section 3.2.1 Bellman-Ford's algorithm, we apply this algorithm to our given graph to find negative weight by calculating minimum weight from vertex 0 to all other vertices in the graph. We present our searching result with a Boolean value; we return false if and only if the input graph includes a negative weight cycle. Then, we can re-weight all nonnegative wedges. As we want to use the Fibonacci heap [31] to implement our priority queue, we need to create three associated methods of the Fibonacci heap: keep_fibheap(), insent_fibheap() and fibheap_extract_min(). Function keep_fibheap() saves all the vertices in the priority queue with a binary tree structure presenting the priority order of vertices, and we take the source vertex as root. And then, find the minimum in the tree with pointers in linked list. Functions insent_fibheap() and fibheap_extract_min() are for merging, which is achieved by combining two lists containing the tree roots. Compare the roots of the two heaps to be merged, and the smaller one becomes the new root of the newly combined heap. The other tree is added as a subtree to this root. This process can be done in constant time. With the Fibonacci heap, we can implement the Dijkstra's algorithm on graph G' which we created by reweighting edges in graph G. Finally, we print the result in the main function.

#### 3.3.2.5 Evaluation

We can test our implemented code of Johnson's algorithm on the following graph.

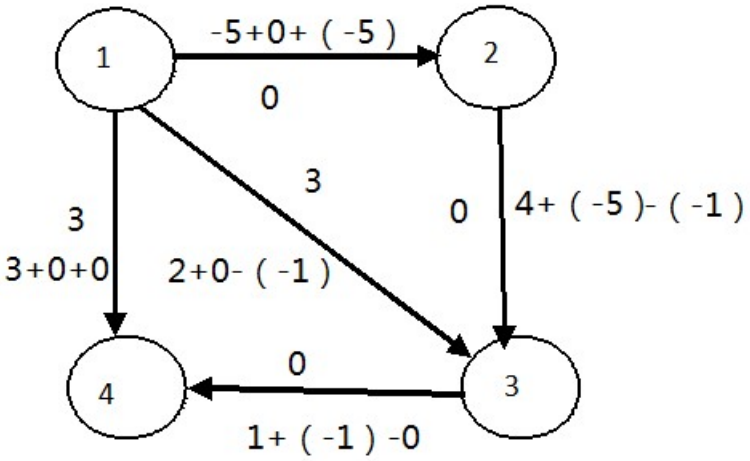


Johnson image 1




Johnson image 2

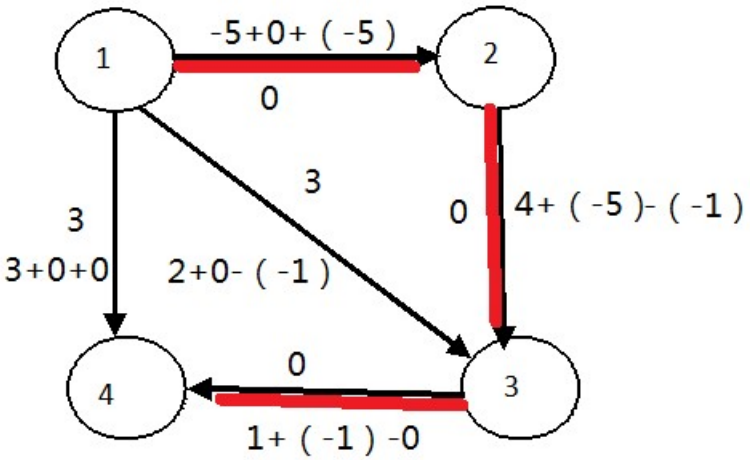
In this image, we present the approach of adding a source vertex 5 and adding edges from vertex5 to all vertices of the original graph.




Johnson image 3

In this image, we present the calculation of the shortest distance from vertex 5 to all other vertices using the Bellman-Ford algorithm. The shortest distance from vertex 5 to vertex1, 2 and 3 are 1, -5, -1 and 0 correspondingly. Once we get the above distances, we remove the source vertex 5 and reweight all the edges using following formula:

$$W(u,v) = \text{weight}(u,v) + h(u) - h(v)$$




Johnson image 4

Since all weights are positive now, we can run Dijkstra's algorithm for every vertex as a source in image 4.

Finally, we match the shortest path we have found in the new graph back to the original graph. Run code on the above images, we can get the following result:

```
Please input the number test cases: 1
Please input the number of vertexes: 4
Please input the number of edges: 5
Please input the vertex information:
The index of vertex 1 is : 1
The index of vertex 2 is : 2
The index of vertex 3 is : 3
The index of vertex 4 is : 4
Please input the edge information:
The source and destination vertexes of edge1 is : 1 2
Please input the weight of this edge: -5
The source and destination vertexes of edge2 is : 1 3
Please input the weight of this edge: 2
The source and destination vertexes of edge3 is : 1 4
Please input the weight of this edge: 3
The source and destination vertexes of edge4 is : 2 3
Please input the weight of this edge: 4
The source and destination vertexes of edge5 is : 1 3
Please input the weight of this edge: 2
```

```
The shortest distance of each pair of vertexes:
From source vertex(1:1) to destination vertex(1:1) is : 0
From source vertex(1:1) to destination vertex(2:2) is : -5
From source vertex(1:1) to destination vertex(3:3) is : -1
From source vertex(1:1) to destination vertex(4:4) is : 3
From source vertex(2:2) to destination vertex(1:1) is : 65540
From source vertex(2:2) to destination vertex(2:2) is : 0
From source vertex(2:2) to destination vertex(3:3) is : 4
From source vertex(2:2) to destination vertex(4:4) is : 65540
From source vertex(3:3) to destination vertex(1:1) is : 65536
From source vertex(3:3) to destination vertex(2:2) is : 65531
From source vertex(3:3) to destination vertex(3:3) is : 0
From source vertex(3:3) to destination vertex(4:4) is : 65536
From source vertex(4:4) to destination vertex(1:1) is : 65535
From source vertex(4:4) to destination vertex(2:2) is : 65530
From source vertex(4:4) to destination vertex(3:3) is : 65534
From source vertex(4:4) to destination vertex(4:4) is : 0
```

In order to get a clear and straightforward view, we can construct the raw data we obtained above into the following matrix:

$$D = \begin{pmatrix} 0 & -5 & 2 & 3 \\ \infty & 0 & 4 & \infty \\ \infty & \infty & 0 & 1 \\ \infty & \infty & \infty & 0 \end{pmatrix}.$$

This code performs as we specified.

The first step in the algorithm of adding vertex 0 runs just in O (V) time because it makes a new edge to all vertices in the graph. The second step of the algorithm, Bellman-Ford, runs in $O(VE)$ time. With Bellman-Ford algorithm, the process of reweighting the graph runs at the same time.

The last step of the algorithm is to apply Dijkstra's algorithm to all V vertices. With a Fibonacci heap, each of these iterations takes the time to complete for a total complexity of $O(V^2 \log(V) + VE)$.

The simpler binary minimum heap implementation yields a running time of $O(VE\log(V))$, which is still faster than the Floyd-Warshall algorithm if the graph is sparse.

# Part V Research (investigation)

## Chapter 4 Weighted edit distance

### 4.1 Introduction

Weighted edit distance is an improvement of the well-known Levenshtein distance, which we have analyzed and implemented in Chapter 1 - 1.4 Levenshtein distance. Levenshtein distance, using for preprocessing raw data, has been applied to a various range of fields. However, the Levenshtein distance is not adequate to solve all real world problems due to its property. The Levenshtein distance is used to measure the similarity of two given strings with minimal cost of possible operations of deletions, insertions, or substitutions, where the algorithm treats all operations as equivalent weight. However, in different conditions of problems to meet particular predilections, we shall undertake analysis and assign the appropriate weight in our original algorithm. This idea invokes the improved version of Levenshtein distance - Weighted edit distance. As we note that for the different background of problems, there can be entirely different ways to assign the weight to meet the requirement and there not exists one perfect assignment of the weight suitable for all problems. For example, if we apply weighted edit distance to measure the similarity of DNA chains, we need to analyze the entry for each base. Then, we will assign weight to the specific positions depend on the preference and rules of constructing DNA chains. However, in information retrieval, we tend to analyze the similarity between the given keyword and matching words in an article such that we give appropriate weight to those words similar to the keyword. From the above examples, we also observe that for a particular problem, whether we can build a precise and reliable weighted edit distance algorithm largely depend on the analysis of relative data. In this section, instead of the using a quantity of experimental data, we want to develop a basic weighted edit distance algorithm for relatively general cases by analyzing its operation properties in the keyboard typing scenario. Furthermore, another model aims to find the probability of misspelling a word to another, specifically, by analyzing the Manhattan distances on the keyboard from character to character. Therefore, the weighted edit distance

algorithm we built in this section is an immediate improvement of the original model, which solves the same type of problem- similarity of strings.

### 4.2 Analysis

Recall that Levenshtein distance is a measure of the similarity between two strings: the source string (s) and the target string (t). The distance is the minimal amount of deletions, insertions, or substitutions required to transform s into t. Note that each operation (‘delete,' ‘insert,' or ‘substitute’) costs weight 1 in our original model.

In this section, we expect to build two applications of weighted edit distance.

One base on analysis of the operations properties of the well-known Levenshtein distance on keyboard typing. We call it **Constraint weighted edit distance**. The advantage of this approach is that eliminates the risk of obtaining several possible solutions in the original model due to the equivalent weight for each operation. For example, to transform the string ‘ cat’ to ‘act,' we can either delete the first ‘c’ and add it between ‘a’ and ‘t’ or replace ‘c’ with ‘a’ and ‘a’ with ‘c,' both manners yield two operations in total. However, with observation, in the keyboard scenario, ‘delete’ costs only one particular action of pressing the ‘Delete ’or ‘Backspace’ key and ‘add’ costs the searching time of the correct key on the keyboard and pressing the key. Moreover, action ‘substitute’ is a combination of both ‘delete’ action and ‘add’ action. With the above analysis, as we always expect the minimum costs of operation, we suppose to assign the weight of ‘delete’ less than ‘add,' of which less than ‘substitute.' That is, among all operations, we prefer ‘delete’ most, while ‘substitute’ least. To achieve this, we built our application base on the existing Levenshtein distance application by fixing the weight of each operation to produce a unique and optimal solution. We implement this application with C# and ASP.NET to obtain an explicit view of the difference between the original model and the weighted model. Keep testing the result we collected from different weight assignment so that we can get our optimal solution.

Another application also analyzes the difference between strings but applies weights of the various characters by calculating the Manhattan distance of keyboard characters. We call this approach **Manhattan weighted edit distance**. The previous approach

focuses on three operations of ‘delete,' ‘add’ and ‘substitute,' however, this method focuses on the similarity of characters inside the string itself. That is, we expect to get the relative accuracy cost of replacing one character with another character on the keyboard. Typically, we are much more likely to mistype the key around the correct key instead of those keys far away from it on the keyboard. There is a confusion matrix for spelling errors [32] (the content of the matrix displays in Appendix A) published by Stanford, which indicates the accurate experimental data on the possibility of misspelling each character to another character on the keyboard. To implement this, we have to build a 26*26 size matrix containing all the probability for misspelling each character to any other characters. Although it is painful to create this matrix table at the beginning, once built, it is once for all.

If we assign the probability in the matrix as weight, in this case, for the source strings and target string t, we have to look through the matrix on each character in the corresponding position of s and t to find the weight between the two characters. For example, if s=’cat’ and t=’act,' then, for the first position, we need to get the weight between ‘a’ and ‘c’ from the given confusion matrix which is 6. And we can apply this methodology character by character on both strings.

Despite this heuristic promises the accuracy of the weight since it comes from statistics, it is relatively complicated, and it hurts the transportability since all the weight assignment dependent on the data in confusion matrix. That means, whenever we want to implement this weighted edit distance we have to import the matrix.

For that reason, we want to find a simpler way to represent the weights among each pair of characters on the keyboard. Alternatively, we can assign the weight as the Manhattan distance between each pair of characters on the keyboard. The reason that we are not going to use Euclidean distance is that we only concern the relative distance to get the sense that whether or not the character is close to another character instead of to get an accurate value.

To apply Manhattan distance, we create a two-dimensional matrix 'dict' to store all the character keys on the keyboard from ‘a’ to ‘z’ in the corresponding location of matrix 'dict.' We assign coordinate (row, col), where row from 0 to 2 and col from 0 to 9, to each key on the keyboard in the corresponding entry of the matrix.

Note that Manhattan distance[33] is the distance between two points in a grid based on a strictly horizontal and/or vertical path (that is, along the grid lines). The Manhattan distance is the simple sum of the horizontal and vertical components.

More formally, we denote the Manhattan distance between the certain position (i,j) in source string s and position (k,l) in target string t as dist (s[i][j],t[k][l]), where i and k are from 0 to 2 and j and l are from 0 to 9, then we have the formula:

$$\text{dist}\,(s[i][j], t[k][l]) = \text{abs}(s[i][j] - t[k][l])$$

Manhattan distance is not only straightforward to calculate but also suitable for applying on the keyboard and associating with the other two operation weights which are integers as well.

In our **Manhattan weighted edit distance** algorithm, we calculate the Manhattan distance, row by row and column by column, between each key to any other key in our matrix 'dict.' We store each the Manhattan distance of key in matrix 'dist' entry by entry. Instead of a confusion matrix, we record all the Manhattan distance we calculated for each pair of keys on the keyboard. With all the Manhattan distance in our matrix, we can perform **Manhattan weighted edit distance**. We implement this algorithm in Visual Studio with C++.

### 4.3 Design

Our weighted edit distance, either ***Constraint***versionor ***Manhattan*** version, is developed from the original Levenshtein distance algorithm by adding weights. So we recall the structure of Levenshtein distance, where we use notation Edit(i, j) to replace Edit(s[1 .. i], t[1 .. j]). This function satisfies the following recurrence:

$$\text{Edit}\,(i,j) = \min \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{cases} \text{Edit}\,(i-1,j) + 1 & \text{insertion} \\ \text{Edit}\,(i,j-1) + 1 & \text{deletion} \\ \text{Edit}\,(i-1,j-1) + 1 & \text{substitution} \end{cases} \end{cases}$$

Note that in the above function Edit(i,j) assigns weight 1 to each operation of insertion, deletion and substitution.

1. Constraint weighted edit distance

For constraint weighted edit distance, as we analyzed before, we assign various weights on different operations respectively to gain a unique and optimal solution of transforming misspelling string t to original string s. The result will present as a sequence of operations lead to minimum cost with assigned weights.

As deletion requires only one action- deleting, we can assign the lowest weight to the operation ‘delete. ’ We denote the weight of 'delete' as wd. As insertion costs two actions - searching the match and adding the key- we denote the weight of operation ‘insert’ as wi, which is larger than our basic weight. Define the weight of operation ‘substitute’ as ws, which is larger than both wd and wi, due to the two operations it calls: ‘delete’ and ‘insert,' where ‘insert’ contains two actions. To be comparable, we want to assign the total weight of our weighted edit distance algorithm the same as the original one, in which is 3 (1+1+1).

Note that, if the sum of wd and wi smaller or equal to ws, we will replace ‘substitute’ by two joint operations- ‘delete’ and ‘insert,' in this case, we can never apply operation ‘substitute.' To prevent this, we declare that weight wd, wi and ws satisfy triangle inequality

$$\mathrm{wd} + \mathrm{wi} > ws.$$

From the above, we get all our properties of weights in constraint weighted edit distance model as follows:

$$\mathrm{wd} + \mathrm{wi} + \mathrm{ws} = 3$$

$$\mathrm{wd} < wi < ws$$

$$\mathrm{wd} + \mathrm{wi} > ws$$

We also have to take the initialization into consideration. Assume that we have original string s with length n and misspelling string t with length m. We assign initial weight coefficient of original string s as $\alpha$ and coefficient of misspelling string t as $\beta$. Denote the initial weight for each character in string s as vi, where we calculate vi by $vi = i * \alpha$ where i from 1 to n. Also denote the initial weight for each character in

string t as vj, where we calculate vi by vj = j *β, where j from 1 to m. To simplify our problem, we can view vi and vj impartially, that is, we let $\alpha = \beta$. Keep our goal in mind that we shall ensure that the initial weight will not tamper our optimal solution and not result in several possible solutions.

We can test different value range of coefficient α and β, where $\alpha = \beta$. Typically, there are four possible value ranges:

1. $\alpha = \beta < wd$
2. $\text{wd} \leq \alpha = \beta < wi$
3. $\text{wi} \leq \alpha = \beta < ws$
4. $\text{ws} \leq \alpha = \beta$

By testing all the values on coefficient α and β in each range, we notice that with thespecified value of coefficient α and β, we fail to obtain an optimal solution. For example, to transform t =’af’ to string s = ‘asdf,' if we assign coefficient α and β=0.2 and wd =0.4, wi =1 and ws = 1.6, the optimal solution supposed to be that add ‘s’ and ‘d’ between ‘a’ and ‘f.' However, the solution we get is to add ‘a’ and ‘s,' replace ‘d’ to ‘a.' Intuitively, the solution we obtain has the minimum cost with the assigned weight, however, fails to provide the optimal operations. To prevent this problem, we can assign the value of coefficient α and β larger than the maximum value of the difference between wi and wd and the difference between ws and wi. Formally, we have $\alpha = \beta > \max(\text{wi} - \text{wd}, \text{ws} - \text{wi})$. That means, the coefficientα and β and weight wd, wi and ws all satisfy following triangle inequalities:

$$\text{wd} + \alpha(\text{ or } \beta) > wi$$

$$\text{wi} + \alpha(\text{ or } \beta) > ws$$

Compare the following two image:

| | | a | f | | | a | f |
|---|---|---|---|---|---|---|---|
| | 0 | 0.2 | 0.4 | | 0 | 0.4 | 0.8 |
| a | 0.2 | 0 | 0.2 | a | 0.4 | 0 | 0.2 |
| s | 0.4 | 0.5 | 0.7 | s | 0.8 | 0.5 | 0.7 |
| d | 0.6 | 0.8 | 1 | d | 1.2 | 1 | 1.2 |
| f | 0.8 | 1 | 0.8 | f | 1.6 | 1.5 | 1 |
| | | | | | | | |

With the same input and operation weights, in the left image, we assigned coefficient $\alpha = \beta = 0.2$, while in the right image, we assigned coefficient $\alpha = \beta = 0.4$. As analyzed above, the left one fails to provide the optimal operations due to the violation of our triangle inequality.

If we use LitWeiEdit (i,j) to denote that Constraint Weighted Edit Distance we can construct the following recurrence:

$$\text{LitWeiEdit (i,j)} = \min \begin{cases} i * \alpha & \text{if } j = 0 \\ j * \beta & \text{if } i = 0 \\ \min \begin{cases} \text{LitWeiEdit}(i-1, j) + \text{wi} & \text{insertion} \\ \text{LitWeiEdit}(i, j-1) + \text{wd} & \text{deletion} \\ \text{LitWeiEdit}(i-1, j-1) + \text{ws} & \text{substitution} \end{cases} \end{cases}$$

With following configurations:

$$\text{wd} + \text{wi} + \text{ws} = 3$$

$$\text{wd} < wi < ws$$

$$\text{wd} + \text{wi} > ws$$

$$\alpha = \beta > \max(\text{wi} - \text{wd}, \text{ws} - \text{wi}),$$

$$which\ enable\ the\ two\ following\ \text{triangle inequalities}\ \ hold$$

$$\text{wd} + \alpha(\text{ or } \beta) > wi$$

$$\text{wi} + \alpha(\text{ or } \beta) > ws$$

With the above recurrence, we can create the following pseudocode:

CONSTRAINTWEIGHTEDEDIT (s [1...n], t [1...m]):

```
for j <-1 to m
    LitWeiEdit (0, j) <-  j ∗ β
for i <-1 to n
    LitWeiEdit (i, 0) <- i ∗ α
    for j <- 1 to n
        if s[i] = t[j]
            LitWeiEdit (i,j) <- min [Edit (i -1, j) + wi, Edit (i, j -1) + wd, Edit (i -1, j -1)]
        else
            LitWeiEdit (i,j) <- min [Edit (i -1, j) + wi, Edit (i, j -1) + wd, Edit (i -1, j -1) + ws]
return LitWeiEdit (n, m)
```

Constraint weighted edit distance yields a unique solution with a set of optimal operations since it always applies dynamic programming to select the operations, who hold the minimum weight, which is much more likely to result in a unique and optimal solution without violating any above configurations.

## 2. Manhattan weighted edit distance

In the keyboard-typing scenario, with a similar idea but a different goal, we developed Manhattan weighted edit distance algorithm to analyze the relative accuracy of substitution from character to character on the keyboard. That is, when we apply the operation of ‘substitute,' we expect to get the distance from the misspelling character to the correct character on the keyboard. As mentioned above, “confusion matrix for spelling errors” provides the most accurate difference between each character to any other characters on the keyboard. However, with an also significant shortage on data transportability and confidentiality, we expect to find an algorithm to replace the “confusion matrix.” That is, we neither want to hard code the whole matrix in our code nor make our weighted edit distance model only depend on the data in that

matrix. Therefore, we introduce Manhattan weighted edit distance, which focuses on characters' location on the keyboard without relying on the statistical data. Intuitively, we are much more likely to misspell the original character into the characters closer to it rather than characters far from it. For example, on the keyboard, 'a' is much closer to 's' than 'p.' Therefore, we can calculate all Manhattan distance from character to character, and we give a threshold weight ρ to substitution operation. Then, we assign the Manhattan distance as the weight of operation 'substitute.'

In Manhattan weighted distance model, as we only want to analyze the role of character distance to apply 'substitute,' we need to dump the interruption of 'insert' and 'delete.' That is, we will assign the weights of 'insert' and 'delete' large enough so that we can only consider substituting characters. In this way, besides the case that two characters are too far from each other, we expect only to apply substitution between the pair of characters. To achieve this, we have to assign the threshold weight ρ to be large enough.

To choose an appropriate ρ, we recall the triangle inequality we applied before, we still denote the weight of operation 'delete', 'insert' and 'substitute' as wd, wi and ws independently, the weights have to satisfy the following triangle inequality: To choose an appropriate ρ, we recall the triangle inequality we applied before, we still denote the weight of operation 'delete', 'insert' and 'substitute' as wd, wi and ws independently, the weights have to satisfy the following triangle inequality:

$$\mathrm{wd} + \mathrm{wi} > ws,$$

and we also have:

$$\mathrm{ws} \leq \rho .$$

Combine the above two inequalities we can get our general range for wd and wi, which is:

$$\mathrm{wd} + \mathrm{wi} > \rho .$$

Note that we will calculate the Manhattan distance between characters, so the results will promise to be integers. In this case, we also assign all the weights of operations as integers as well. To make it simpler, we give equal weight to wd and wi, because in this algorithm we ignore the slight difference between operation 'delete' and 'insert'

and principally focus on the weight of 'substitute' given be the Manhattan distance between characters on the keyboard.

In string scenario, we only consider letters. Therefore, we can create a two-dimensional array with three rows and ten columns to carry those characters. Especially, we add one '*' in the row starts with 'a' and two '*' in the row starts with 'z' to occupy the positions.

Apparently, among all the characters, the largest distance on the keyboard is the distance between 'p' and "z," where the Manhattan distance is 12 (calculated by the sum of ten horizontal grids and two vertical grids). Hence, we have our threshold weight ρ = 12, and we can assign the weight of 'delete' and 'insert' to be seven for each. Then we have:

$$\mathrm{wd} = \mathrm{wi} \geq 7$$

where the triangle inequality $\mathrm{wd} + \mathrm{wi} > \rho$ hold.

For given original string s and misspelling string t, we calculate the Manhattan distance between each character in string s and each character in string t. Then we assign the Manhattan distance value to be the weight of 'substitute' ws on corresponding pairs of characters.

We denote that p is the character locates at the dict[i][j] position while q is found at the dict[k][l] position in matrix dict, where i and k are from 0 to 2 while j and l are from 0 to 9. Note that the location of each character reserves in matrix dict matching the position it appears on the keyboard.

Then we have the formula:

$$\mathrm{ws}(\mathrm{p}, \mathrm{q}) = \mathrm{dist}(\mathrm{dict[i][j]}, \mathrm{dict[k][l]}) = \mathrm{abs}(\mathrm{dict[i][j]} - \mathrm{dict[k][l]})$$

If we use MhtWeiEdit (i,j) to symbolize Manhattan Weighted Edit Distance we can construct the following recurrence:

$$\text{MhtWeiEdit}(i,j) = \min \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{cases} \text{MhtWeiEdit}(i-1,j) + \text{wi} & \text{insertion} \\ \text{MhtWeiEdit}(i,j-1) + \text{wd} & \text{deletion} \\ \text{MhtWeiEdit}(i-1,j-1) + \text{ws} & \text{substitution} \end{cases} \end{cases}$$

With following configurations:

$$\text{wd} + \text{wi} > \rho \, .$$

$$\rho = 12$$

$$\text{wd} = \text{wi} \geq 7$$

$$\text{ws}(p,q) = \text{dist}(\text{dict}[i][j], \text{dict}[k][l]) = \text{abs}(\text{dict}[i][j] - \text{dict}[k][l])$$

With the above recurrence, we can create the following pseudocode:

MANHATTANWEIGHTEDEDIT (s [1...n], t [1...m]):

Create matrix dict[][] to store the characters on the keyboard at the corresponding position.

//Calculate the Manhattan distance between each character to any other characters in dict [][] and store the Manhattan distance on each corresponding position in matrix dist[][]..

```
for row <- 0 to 2
   for col <- 0 to 10
      for x <- 0 to 2
         for y <- 0 to 10
            if (dict[x][y] != '*')
               dist[dict[row][col] - 'a'][dict[x][y] - 'a'] = abs(row - x) + abs(col - y);
for j <-1 to m
     MhtWeiEdit (0, j) <-  j
for i <-1 to n
```

```
MhtWeiEdit (i, 0) <- i

for j <- 1 to n

ws(i,j)= dist[i][j]

    if s[i] = t[j]

        MhtWeiEdit (i,j) <- min [Edit (i -1, j) + wi, Edit (i, j -1) + wd, Edit (i -1,
    j -1)]

    else

        MhtWeiEdit (i,j) <- min [Edit (i -1, j) + wi, Edit (i, j -1) + wd, Edit (i -1,
    j -1) + ws(i-1, j-1)]

return MhtWeiEdit (n, m)
```

Manhattan weighted edit distance yields a unique and optimal solution with the relative accurate weight of substituting a character to another. Since it always applies Dynamic Programming to get the minimum weight, we achieve a unique and optimal solution without violating any above configurations. As we avoid hard code and dependency of data in confusion matrix, we promised the confidentiality and transportability of our Manhattan weighted edit distance.

### 4.4 Implementation

The implemented applications are created to test the correctness of the above two algorithms. The toughest part of this research project is insufficient of supporting materials in the research field. Although there are plenty of papers online study weighted edit distance algorithms, they have different focuses. As we mentioned before, weighted edit distance algorithms are problem dependent. Especially, they are applied to subjects and research fields other than Computer Science, such as image analysis, DNA chain analysis and stock data preprocessing. However, as we know weighted edit distance is an improvement of the Levenshtein edit distance, there are few studies on the general property of weighted edit distance itself to improve the functionality of edit distance. Therefore, based on some discoveries from experimental data and compared with other existed weighted edit distance models,

Constraint weighted edit distance and Manhattan weighted edit distance algorithms are developed to improve the performance of the original model.

1. Constraint weighted edit distance application

Constraint weighted edit distance application is developed with C# and ASP.NET, where the framework and windows are imported from open source code [15]. That is, the application is developed from the existing framework, and the implementation of the Constraint weight edit distance is made on the Levenshtein Distance Algorithm.cs file.As Constraint weighted edit distance algorithm is comparable to Levenshtein edit distance, the approach of Dynamic Programming search for the minimum distance is the same as the code we implanted in 1.4 Levenshtein distance. Apart from that, we implemented the weights assigned to each operation of ‘delete,' ‘insert’ and ‘substitute’ according to the pseudocode and constraints we have given in design part. Moreover, we can print each operation we applied to obtain our minimum cost in reverse order from searching. Note that Dynamic Programming finds an optimal solution bases on the optimal solutions to its subproblems. That is, in each step we have already calculated all the required sub-solution for the current problem. Therefore, when we find the solution, we will backtrack from end to the start position to find all the operations we applied. However, we want to print forward the operations, from start position, we used to find our optimal solution. Thus, we have to print the operations we collected from backtracking inversely.

In C#, we use an array to store the set of operations leading to the optimal solution and print the array in reverse order by applying function Array. Reverse() to the collected string. Therefore, we can check the operations we printed to check whether the algorithm provides the expected result.

As Constraint weighted edit distance algorithm is aimed to analyze the influence of different assignments on weights of operations, it is essential to print the operations that we used to get the optimal solution. To examine whether we collect an optimal solution, we need to check whether the solution has the minimum cost of operations and whether the operations we applied to achieve the solution are optimal. As we implement weighted model based on the original one by adding appropriate weights,

the running time of Constraint weighted edit distance is the same as the original edit distance algorithm, which is $\theta(mn)$.

2. Manhattan weighted edit distance application

Manhattan weighted edit distance application is built according to the above analysis and pseudocode we designed. Taking advantage of the calculation speed of C++, we can calculate the Manhattan distance between characters in the original string and misspelling string rapidly. For each position in the original string and misspelling string, we calculate the Manhattan distance value and assign it as the weight of 'substitute' operation. That is, for two given strings where the original string has length n, and misspelling string has length m, we will call calculation of Manhattan distance n*m times. Although there are four nested for loop, since we size of calculation range is fixed, which is the distance between corresponding positions in two matrices with three rows and ten columns, we can calculate our Manhattan distance between two characters in constant time O(1). Therefore, the weighted edit distance algorithm has the same time complexity as the original algorithm which is $\theta(mn)$.

### 4.5 Evaluation

Levenshtein distance works for finding the minimum distance between two strings.

1. Constraint weighted edit distance algorithm

There are four operations in total: match, substitute, delete and insert.

However, the four operations not of the same complexity. As it costs nothing for matching cases, the weight of matching is assigned to zero. Deleting on the keyboard only requests for one action- press the 'Backspace' key, in this way, deleting should be allocated the minimum weight among all the operations. Then, inserting requires to find the inserting key on the keyboard which is slightly more complicated than deleting, so inserting supposes to have the second minimum weight. The most costly weight should be assigned to substitution because it requires two actions; removing the existing key and adding the new key.

In the Levenshtein edit distance, the weight of every operation is assigned to one, so the total potentially available weight will be three (insert, delete and substitute). However, according to the analysis above, the order of our preference of operations should be deletion followed by insertion and substitution. The initialization weight matters as well, for instance, if the initial data is 1, while the weights of deleting, inserting and substituting are 0.4, 1 and 1.6. Deletion will certainly be preferred than other operations, and a certain amount of accumulated calculations lead to a unique solution.

In the testing data, there are following constraints:

1. Weight of deletion less than insertion less than substitution
2. The total weight will be three (the same as the total weight in the original model to compare the optimization of the weighted model)
3. The initialization value is changeable

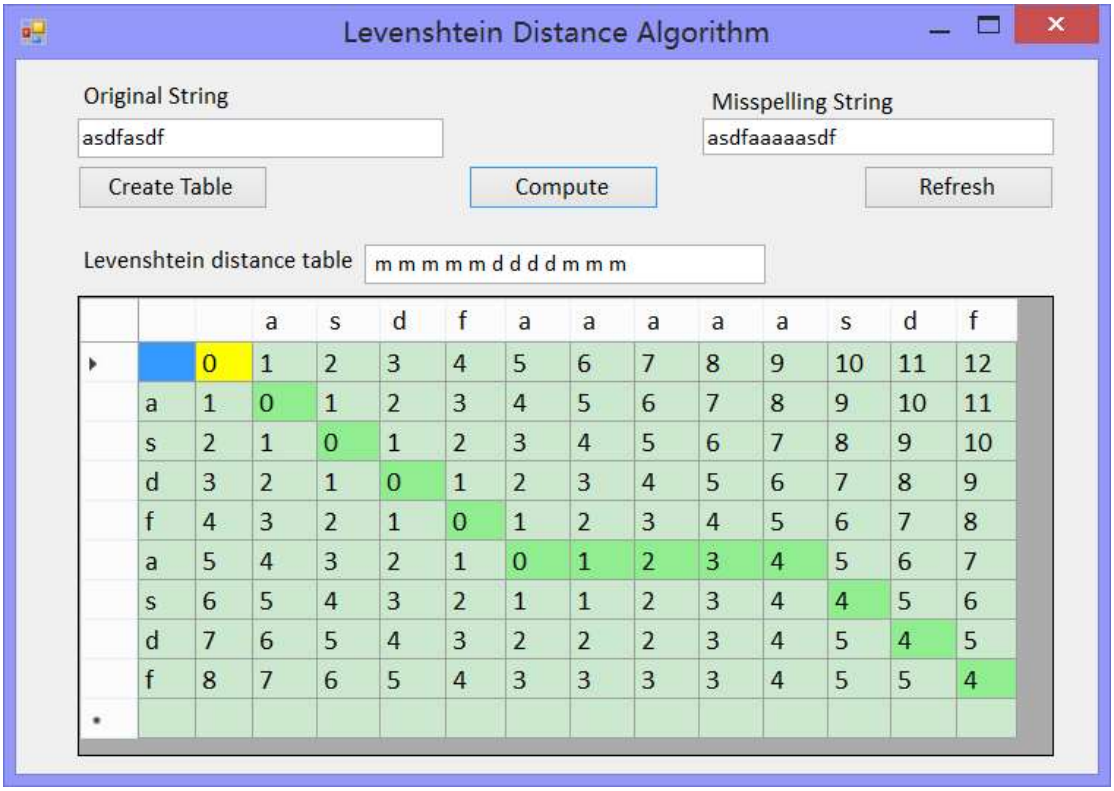
The expect results:

1. The model will pick the operation according to our preference.
2. The total weight in the weighted model should be less than in the original algorithm.
3. The solution for the weighted model should be unique (as we have a particular preference there shouldn't be other solution), while the original model may have more than one solution.

We test the original string and misspelling string in the original model and weighted model respectively and compare the returned result in the following images.

Test string: Original string: asdfasdf Misspelling string: asdfaaaaasdf

Original model:




| | | a | s | d | f | a | a | a | a | a | s | d | f |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| a | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| s | 2 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| d | 3 | 2 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| f | 4 | 3 | 2 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| a | 5 | 4 | 3 | 2 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| s | 6 | 5 | 4 | 3 | 2 | 1 | 1 | 2 | 3 | 4 | 4 | 5 | 6 |
| d | 7 | 6 | 5 | 4 | 3 | 2 | 2 | 2 | 3 | 4 | 5 | 4 | 5 |
| f | 8 | 7 | 6 | 5 | 4 | 3 | 3 | 3 | 3 | 4 | 5 | 5 | 4 |

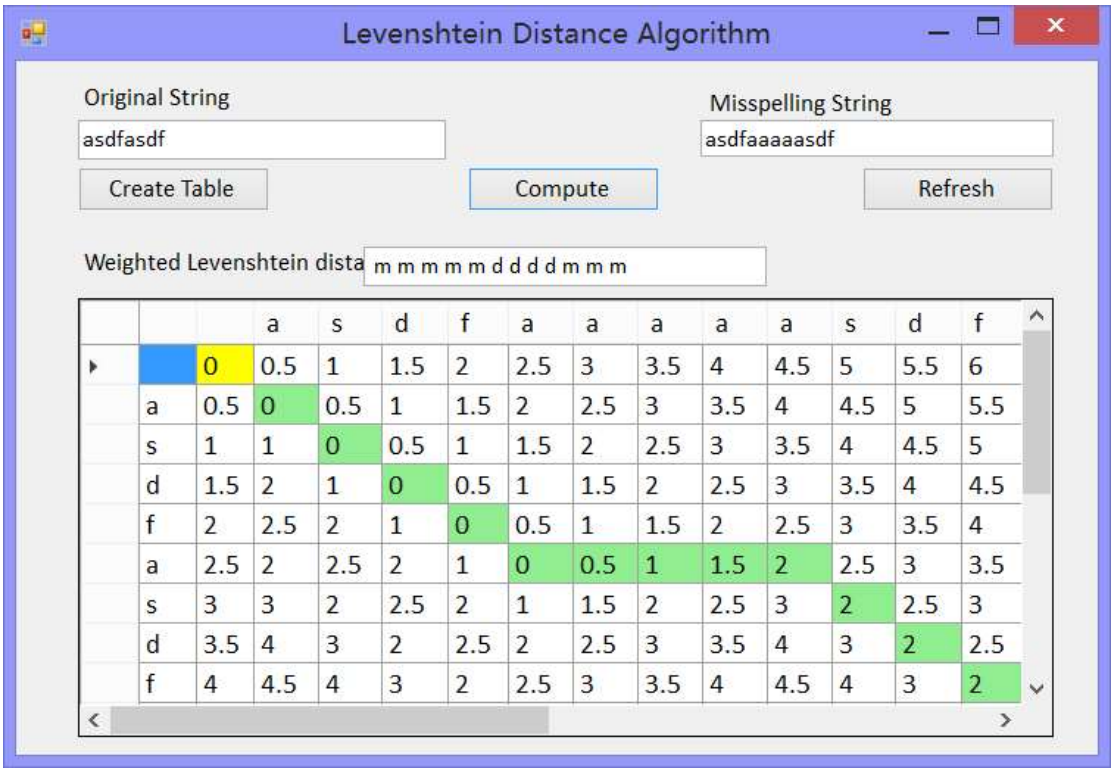
Test data:

Group 1:

Initialization: 0.5

Weight of deletion=0.5, insertion= 1.0, substitution=1.5

Weighted model:



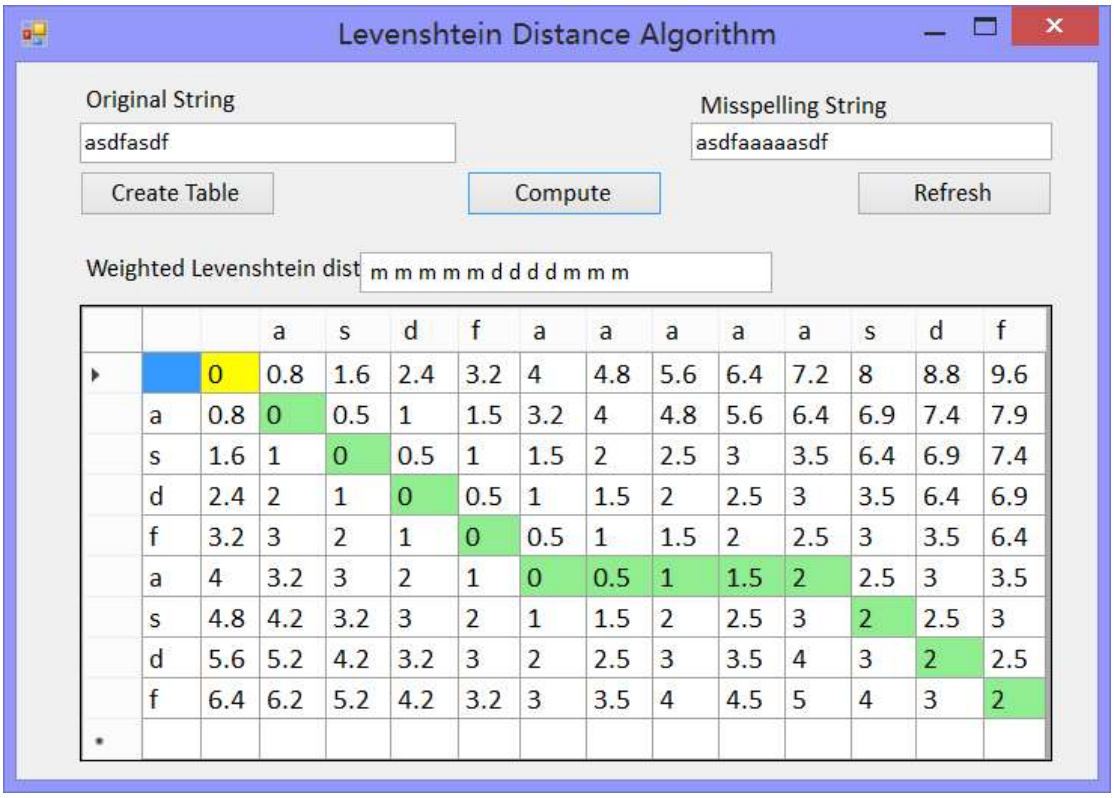

| | | a | s | d | f | a | a | a | a | a | s | d | f |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0.5 | 1 | 1.5 | 2 | 2.5 | 3 | 3.5 | 4 | 4.5 | 5 | 5.5 | 6 |
| a | 0.5 | 0 | 0.5 | 1 | 1.5 | 2 | 2.5 | 3 | 3.5 | 4 | 4.5 | 5 | 5.5 |
| s | 1 | 1 | 0 | 0.5 | 1 | 1.5 | 2 | 2.5 | 3 | 3.5 | 4 | 4.5 | 5 |
| d | 1.5 | 2 | 1 | 0 | 0.5 | 1 | 1.5 | 2 | 2.5 | 3 | 3.5 | 4 | 4.5 |
| f | 2 | 2.5 | 2 | 1 | 0 | 0.5 | 1 | 1.5 | 2 | 2.5 | 3 | 3.5 | 4 |
| a | 2.5 | 2 | 2.5 | 2 | 1 | 0 | 0.5 | 1 | 1.5 | 2 | 2.5 | 3 | 3.5 |
| s | 3 | 3 | 2 | 2.5 | 2 | 1 | 1.5 | 2 | 2.5 | 3 | 2 | 2.5 | 3 |
| d | 3.5 | 4 | 3 | 2 | 2.5 | 2 | 2.5 | 3 | 3.5 | 4 | 3 | 2 | 2.5 |
| f | 4 | 4.5 | 4 | 3 | 2 | 2.5 | 3 | 3.5 | 4 | 4.5 | 4 | 3 | 2 |

Group 2:

Initialization: 0.8

Weight of deletion=0.5, insertion= 1.0, substitution=1.5

Weighted model:



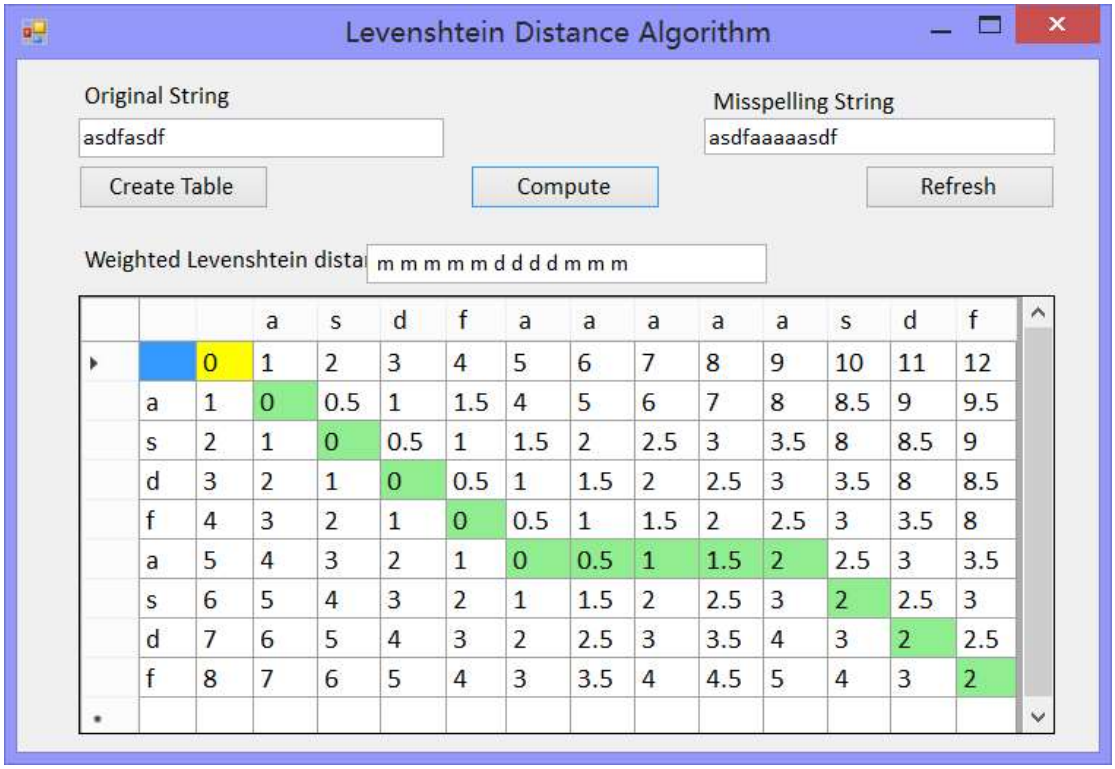

| | | | a | s | d | f | a | a | a | a | a | s | d | f |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ▸ | | 0 | 0.8 | 1.6 | 2.4 | 3.2 | 4 | 4.8 | 5.6 | 6.4 | 7.2 | 8 | 8.8 | 9.6 |
| | a | 0.8 | 0 | 0.5 | 1 | 1.5 | 3.2 | 4 | 4.8 | 5.6 | 6.4 | 6.9 | 7.4 | 7.9 |
| | s | 1.6 | 1 | 0 | 0.5 | 1 | 1.5 | 2 | 2.5 | 3 | 3.5 | 6.4 | 6.9 | 7.4 |
| | d | 2.4 | 2 | 1 | 0 | 0.5 | 1 | 1.5 | 2 | 2.5 | 3 | 3.5 | 6.4 | 6.9 |
| | f | 3.2 | 3 | 2 | 1 | 0 | 0.5 | 1 | 1.5 | 2 | 2.5 | 3 | 3.5 | 6.4 |
| | a | 4 | 3.2 | 3 | 2 | 1 | 0 | 0.5 | 1 | 1.5 | 2 | 2.5 | 3 | 3.5 |
| | s | 4.8 | 4.2 | 3.2 | 3 | 2 | 1 | 1.5 | 2 | 2.5 | 3 | 2 | 2.5 | 3 |
| | d | 5.6 | 5.2 | 4.2 | 3.2 | 3 | 2 | 2.5 | 3 | 3.5 | 4 | 3 | 2 | 2.5 |
| | f | 6.4 | 6.2 | 5.2 | 4.2 | 3.2 | 3 | 3.5 | 4 | 4.5 | 5 | 4 | 3 | 2 |
| * | | | | | | | | | | | | | | |

Group 3:

Initialization: 1

Weight of deletion=0.5, insertion= 1.0, substitution=1.5

Weighted model:

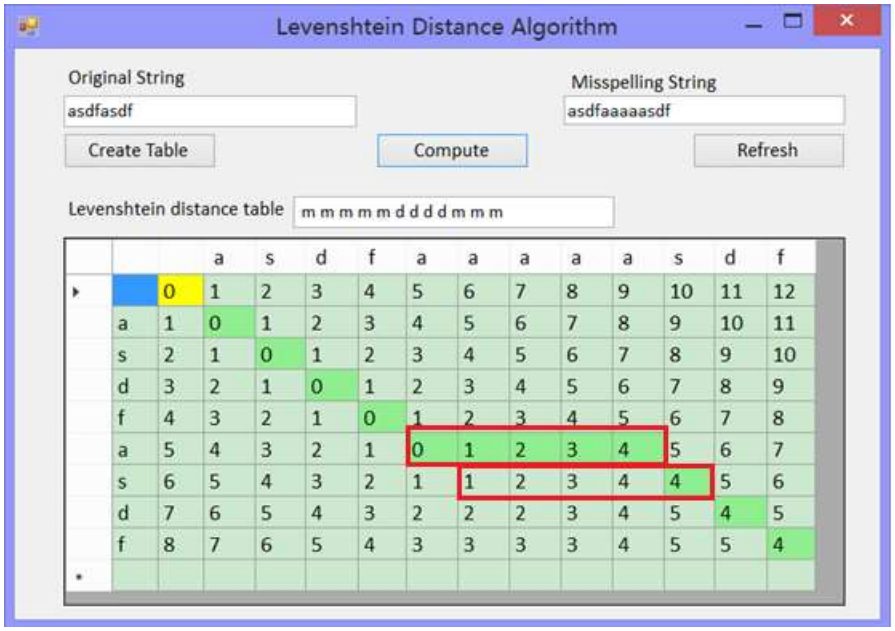



| | | | a | s | d | f | a | a | a | a | a | s | d | f |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ▸ | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| | a | 1 | 0 | 0.5 | 1 | 1.5 | 4 | 5 | 6 | 7 | 8 | 8.5 | 9 | 9.5 |
| | s | 2 | 1 | 0 | 0.5 | 1 | 1.5 | 2 | 2.5 | 3 | 3.5 | 8 | 8.5 | 9 |
| | d | 3 | 2 | 1 | 0 | 0.5 | 1 | 1.5 | 2 | 2.5 | 3 | 3.5 | 8 | 8.5 |
| | f | 4 | 3 | 2 | 1 | 0 | 0.5 | 1 | 1.5 | 2 | 2.5 | 3 | 3.5 | 8 |
| | a | 5 | 4 | 3 | 2 | 1 | 0 | 0.5 | 1 | 1.5 | 2 | 2.5 | 3 | 3.5 |
| | s | 6 | 5 | 4 | 3 | 2 | 1 | 1.5 | 2 | 2.5 | 3 | 2 | 2.5 | 3 |
| | d | 7 | 6 | 5 | 4 | 3 | 2 | 2.5 | 3 | 3.5 | 4 | 3 | 2 | 2.5 |
| | f | 8 | 7 | 6 | 5 | 4 | 3 | 3.5 | 4 | 4.5 | 5 | 4 | 3 | 2 |
| * | | | | | | | | | | | | | | |

Conclusion：

In these cases, initialization value influences the creation of table values, but won't impact the final decision and result in edit distance since there is no violation of any configurations of weights' assignment.

Besides, the total weight of four operations is three which the same as the original model, but as the weighted model prefer deletion, therefore, the total weight reduces from four to two.




| | | a | s | d | f | a | a | a | a | a | s | d | f |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| a | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| s | 2 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| d | 3 | 2 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| f | 4 | 3 | 2 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| a | 5 | 4 | 3 | 2 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| s | 6 | 5 | 4 | 3 | 2 | 1 | 1 | 2 | 3 | 4 | 4 | 5 | 6 |
| d | 7 | 6 | 5 | 4 | 3 | 2 | 2 | 2 | 3 | 4 | 5 | 4 | 5 |
| f | 8 | 7 | 6 | 5 | 4 | 3 | 3 | 3 | 3 | 4 | 5 | 5 | 4 |

In the original model, there are totally two possible solutions:

D1: asdfaaaaasdf -> asdfasdf by mmmmmdddmmm

D2: asdfaaaaasdf -> asdfasdf by mmmmmsddmmm

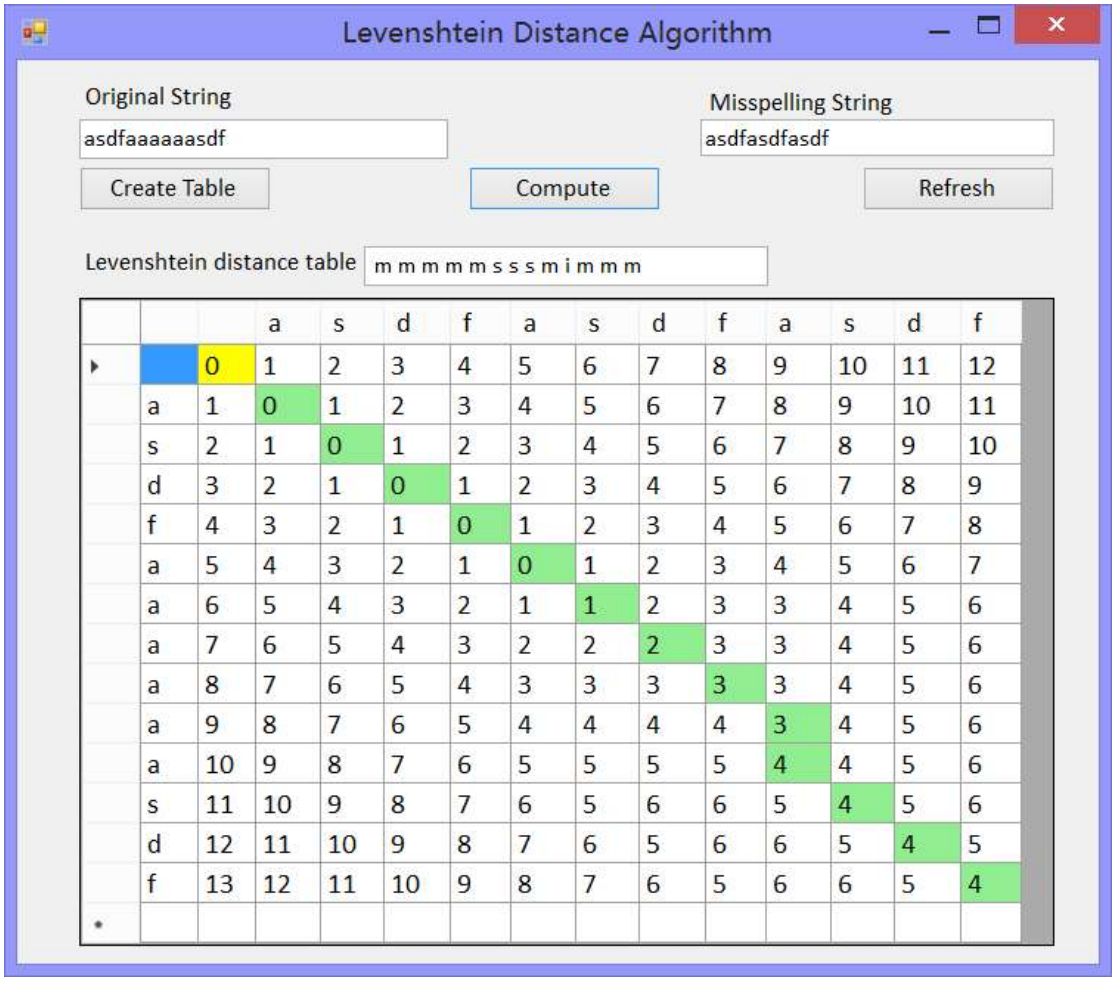
Instead, in the weighted model, there can only be one unique and optimal solutionaccording to our assignment of weights:

D1: asdfaaaaasdf -> asdfasdf by mmmmmdddmmm

Test data for other weights assignments and strings:

Original String: asdfaaaaasdf. Misspelling String: asdfasdfasdf

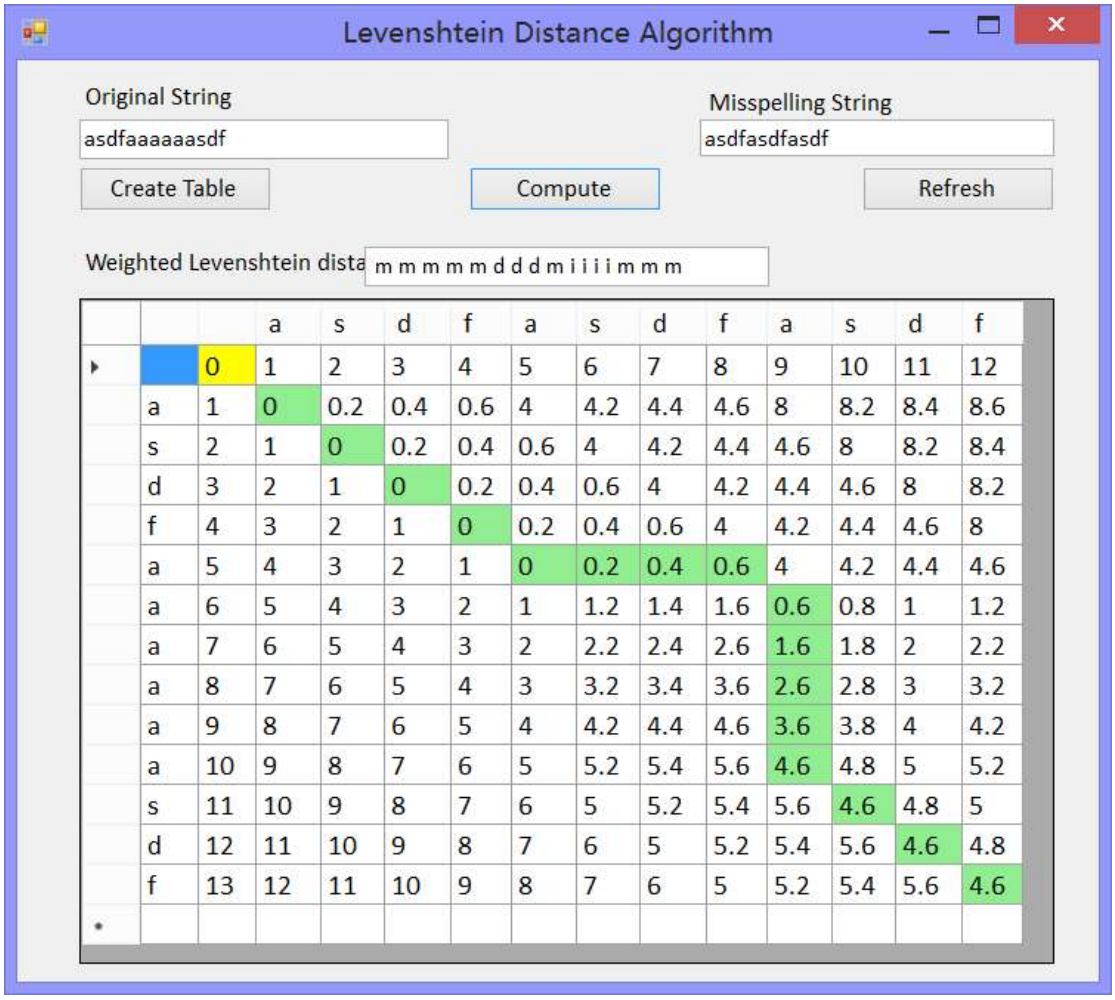
Original model:




| | | | a | s | d | f | a | s | d | f | a | s | d | f |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ▸ | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| | a | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| | s | 2 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| | d | 3 | 2 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| | f | 4 | 3 | 2 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| | a | 5 | 4 | 3 | 2 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| | a | 6 | 5 | 4 | 3 | 2 | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 6 |
| | a | 7 | 6 | 5 | 4 | 3 | 2 | 2 | 2 | 3 | 3 | 4 | 5 | 6 |
| | a | 8 | 7 | 6 | 5 | 4 | 3 | 3 | 3 | 3 | 3 | 4 | 5 | 6 |
| | a | 9 | 8 | 7 | 6 | 5 | 4 | 4 | 4 | 4 | 3 | 4 | 5 | 6 |
| | a | 10 | 9 | 8 | 7 | 6 | 5 | 5 | 5 | 5 | 4 | 4 | 5 | 6 |
| | s | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 6 | 6 | 5 | 4 | 5 | 6 |
| | d | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 6 | 6 | 5 | 4 | 5 |
| | f | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 6 | 6 | 5 | 4 |
| * | | | | | | | | | | | | | | |

Initialization: 1

Weight of deletion=0.2, insertion= 1.0, substitution=1.8




| | | | a | s | d | f | a | s | d | f | a | s | d | f |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ▸ | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| | a | 1 | 0 | 0.2 | 0.4 | 0.6 | 4 | 4.2 | 4.4 | 4.6 | 8 | 8.2 | 8.4 | 8.6 |
| | s | 2 | 1 | 0 | 0.2 | 0.4 | 0.6 | 4 | 4.2 | 4.4 | 4.6 | 8 | 8.2 | 8.4 |
| | d | 3 | 2 | 1 | 0 | 0.2 | 0.4 | 0.6 | 4 | 4.2 | 4.4 | 4.6 | 8 | 8.2 |
| | f | 4 | 3 | 2 | 1 | 0 | 0.2 | 0.4 | 0.6 | 4 | 4.2 | 4.4 | 4.6 | 8 |
| | a | 5 | 4 | 3 | 2 | 1 | 0 | 0.2 | 0.4 | 0.6 | 4 | 4.2 | 4.4 | 4.6 |
| | a | 6 | 5 | 4 | 3 | 2 | 1 | 1.2 | 1.4 | 1.6 | 0.6 | 0.8 | 1 | 1.2 |
| | a | 7 | 6 | 5 | 4 | 3 | 2 | 2.2 | 2.4 | 2.6 | 1.6 | 1.8 | 2 | 2.2 |
| | a | 8 | 7 | 6 | 5 | 4 | 3 | 3.2 | 3.4 | 3.6 | 2.6 | 2.8 | 3 | 3.2 |
| | a | 9 | 8 | 7 | 6 | 5 | 4 | 4.2 | 4.4 | 4.6 | 3.6 | 3.8 | 4 | 4.2 |
| | a | 10 | 9 | 8 | 7 | 6 | 5 | 5.2 | 5.4 | 5.6 | 4.6 | 4.8 | 5 | 5.2 |
| | s | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 5.2 | 5.4 | 5.6 | 4.6 | 4.8 | 5 |
| | d | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 5.2 | 5.4 | 5.6 | 4.6 | 4.8 |
| | f | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 5.2 | 5.4 | 5.6 | 4.6 |
| * | | | | | | | | | | | | | | |

In this case, Constraintweighted edit distance doesn’tprovide an optimal solution, due to the violation of the triangle inequality:

$$\mathrm{wd} + \mathrm{wi} > ws$$

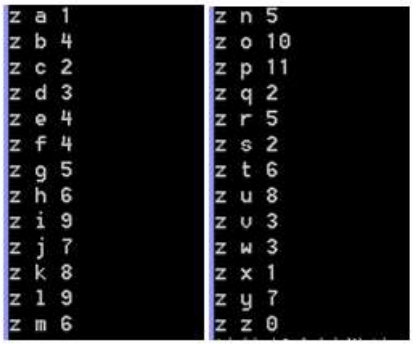
The expected operations supposed to be ‘mmmmmsssmmmm,' however, the solution we collected is ‘mmmmmdddmiiiimmm,' which contains the unnecessarily redundant operations of ‘dddmiiii’ which should be replaced by ‘sss.'

To sum up, by testing Constraint edit distance in different conditions with different assigned weighs, we can always obtain a unique and optimal solution as long as we do not violate any configuration.

2. Manhattan weighted edit distance algorithm

In weighted edit distance, firstly, calculated all the characters distances (Manhattan distance) on the keyboard. Then, initialized the table with the updated weight from characters to characters.

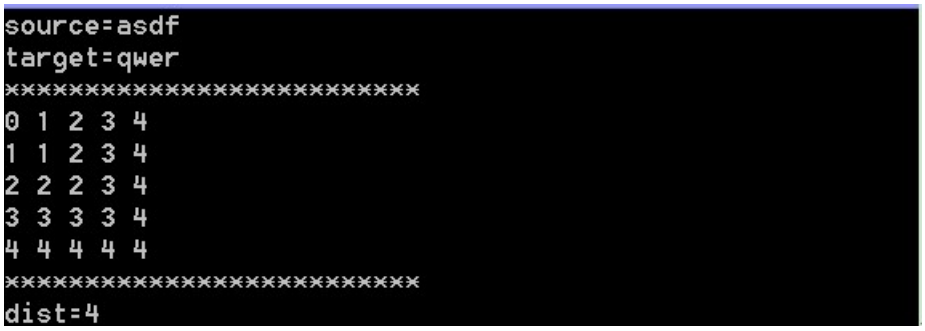
Take character ‘z’ as an example, the Manhattan distance from ‘z’ to any other characters on the keyboard is presented as following:




Assign the weight of both ‘insert’ and ‘delete’ as 7.

Compare the edit distance from the original model and the weighted model with strings: ‘asdf’ and ‘qwer’:

In original model:




In weighted model:

```
Please input original string and misspelling string:
source: asdf
target: qwer
**************************
0 7 14 21 28
7 1 8 15 22
14 8 2 9 16
21 15 9 3 10
28 22 16 10 4
**************************
The weighted edit distance between asdf and qwer is 4
```

Notice that although all the characters in source string different from the target string, however, as the positions of each pair of characters on the keyboard are very close where the Manhattan distance is only one, we obtain a relatively small overall cost by substitution.

Compare the edit distance from the original model and the weighted model with strings: ‘asdf’ and ‘qrwe’:

In original model:

```
source=asdf
target=qrwe
**************************
0 1 2 3 4
1 1 2 3 4
2 2 2 3 4
3 3 3 3 4
4 4 4 4 4
**************************
dist=4
```

In weighted model:

```
Please input original string and misspelling string:
source: asdf
target: qrwe
**************************
0 7 14 21 28
7 1 8 15 22
14 8 4 9 16
21 15 10 6 10
28 22 16 13 8
**************************
The weighted edit distance between asdf and qrwe is 8
```

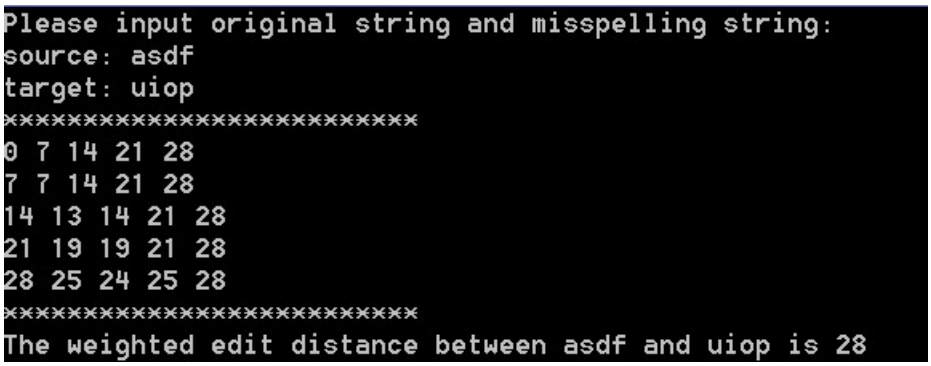
Notice that all characters in the source string are still entirely different from the target string. However, as the positions of each pair of characters on the keyboard are slightly farther from each other, where the Manhattan distance is also increased, we obtain a moderately larger cost by substitution.

Compare the edit distance from the original model and the weighted model with strings: ‘asdf’ and ‘uiop’:

In original model:

```
source=asdf
target=uiop
**************************
0 1 2 3 4
1 1 2 3 4
2 2 2 3 4
3 3 3 3 4
4 4 4 4 4
**************************
distance=4
```

In weighted model:




In this case, although all the characters in source string still completely different from the target string, as the positions of each pair of characters on the keyboard are extremely far from each other, where the Manhattan distance is getting larger as well, we obtain a relatively enormous cost by substitution.

Conclusion：

For the same length of strings, in the original model, the overall cost appears to be the same by substitution. However, in the real world, ‘asdf’ is much closer to ‘qwer’ than ‘uiop.' Therefore, in Manhattan weighted edit distance, the overall cost supposes to be higher due to the assigned weight of substitution. In this way, we can get a unique and much more accurate value resulting from the weights of substitutions based on the Manhattan distance instead of importing Confusion matrix for spelling errors.

# Part VI Evaluation

In this project, an educational website is built to present the self-learning and implementation results on Dynamic Programming Algorithm. In the website, all the features and contents are implemented as expected, it is fluid to use, and it is user-friendly.

On the main page, you will be guided directly to 'element' page to learn Dynamic Programming Algorithm theoretically, or you may decide to explore other parts of the website. After learning the elemental knowledge of Dynamic Programming you may practice on six given projects from easy to challenging to get familiar with the approach of Dynamic Programming Algorithm in real world problems. For each project, you can access to analysis, design, self-implemented C++ code and experimental evaluation. There are images helps to learn as well. Furthermore, you can explore the comparisons between Dynamic Programming Algorithm and other algorithms- Divide and Conquer and Greedy Algorithms- from the drop-down menu. As discussed in Chapter 2 already, by experimental approaching and testing, Dynamic Programming is much more efficient than Divide and Conquer and guarantees to provide an optimal solution. Note that Greedy Algorithms fail to give an optimal solution to problem optimization sometimes, however, by testing a specified range of data, we may draw a conclusion that whether Greedy Algorithms work or not. These comparison parts are displayed with another framework, and you will view the materials on separate web pages. However, these individual web pages connected seamlessly with the main website. Moreover, both Greedy Algorithms and Dynamic Programming strategies are applied to solve Shortest Path Problems. The analysis, design, implementation and testing of the four algorithms developed in Shortest Path Problems have been expanded elaborated in Chapter 3. Specifically, we use Bellman-Ford and Dijkstra's Algorithm to solve single-source shortest path problems. Note that Bellman-Ford solves graph allowing negative weight edge, but without negative circles with Dynamic Programming strategy while Dijkstra's algorithm solves graph with non-negative weight edges by Greedy Algorithms. Besides, Floyd-Warshall's algorithm and Johnson's algorithm are typically applied to solve

all-pair shortest path problems, where Floyd-Warshall works well for dense graphs while Johnson's algorithm efficient for sparse graphs.

All the above algorithms are well examined on graphs with the matching constraints, and they all meet the expected results.

As an investigation part in the project, two weighted edit distance algorithms- Literature weighted edit distance algorithm, and Manhattan weighted edit distance algorithm are introduced, analyzed, designed, implemented and evaluated in the previous section. By refinedly testing on all the possible weights under configurations, we declare that the Constraint weighted edit distance algorithm guarantees a unique and optimal solution are associating with optimal operations. Furthermore, by rigorous design and testing, Manhattan weighted edit distance algorithm applied in keyboard-typing scenario successfully discovers the relative accurate cost of substitution on different pairs of keys with the various distances on the keyboard without importing the confusion matrix given by Stanford.

As we emphasized, this website is developed for educational purpose of Dynamic Programming Algorithm. One of the indispensable measurement of the quality of this website is users' feedback. There is a contact page in the website, users can post any comments on the website, and all the previous comments are published according to time order whenever the page loaded since the data are stored locally. This function works as expected.

Moreover, surveys are collected from users with different majors: Computer Science and Non-Computer Science, different years: Second year, Third year and Final year.

There are 25 users participated in fulfilling their surveys in total. Specifically, 18 junior year students, who major in Computer Science without background knowledge of Dynamic Programming Algorithm, consists of our primary target users of this project. Additionally, there also contains four senior Non-Computer Science major students, one senior Computer Science student who learned Dynamic Programming Algorithm before and two Non-Computer Science sophomore. To be equality, different surveys are designed for Computer Science students and Non-Computer Science Students respectively. For Computer Science Students, the content of the

survey is much focus on the value of the content and structure, while for other students, the survey is much emphasis on the user experience and design.

After collecting the raw data, we created following tables present the feedbacks we obtained:

User distribution:

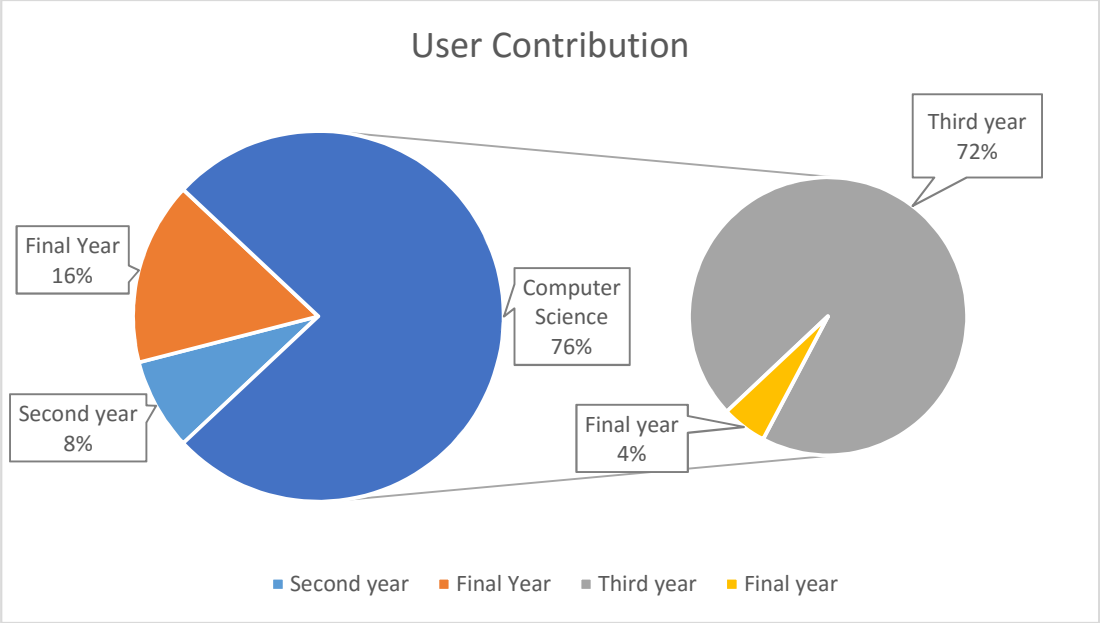
| | |
|---|---|
| User number | 25 |
| Second year | 2 |
| Third year | 18 |
| Four year | 5 |

| | |
|---|---|
| Users' major | |
| Computer Science | 19 |
| Non-Computer Science | 6 |

Combine the above two tables we obtain a table for user distribution:

| | |
|---|---|
| Non-Computer Science | 6 |
| Second year | 2 |
| Final Year | 4 |
| Computer Science | 19 |
| Third year | 18 |
| Final year | 1 |

The user distribution can also be displayed with the following graph:

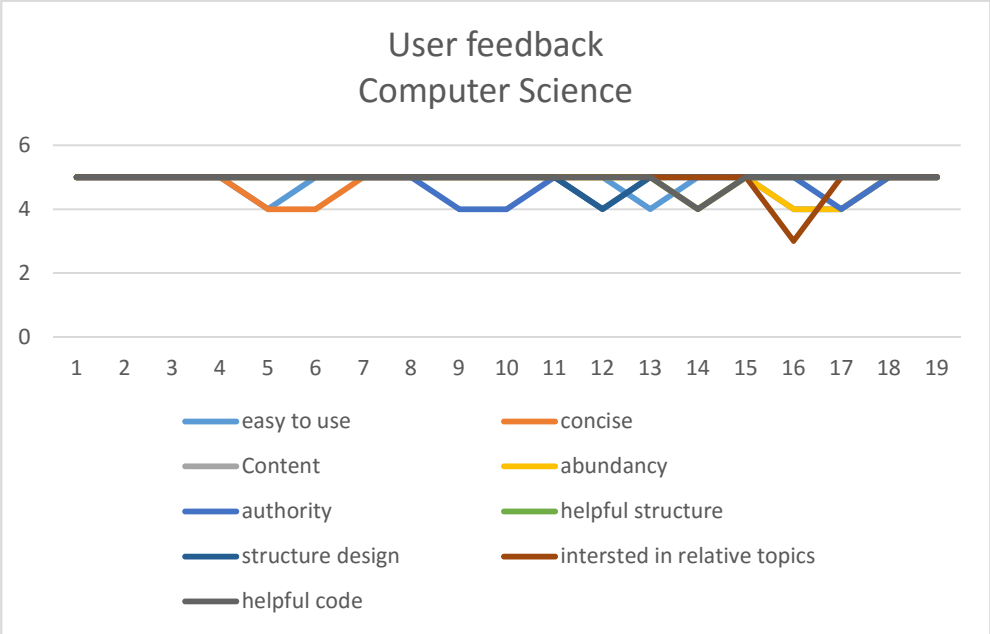



Firstly, we present Computer Science students' feedback with the following tables:

| User feedback | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Computer Science | | | | | | | | | | | | | | | | | | | |
| Excellent - 5 Good - 4 Fine - 3 Bad - 2 Very Bad -1 | | | | | | | | | | | | | | | | | | | |
| User No. | User 1 | User 2 | User 3 | User 4 | User 5 | User 6 | User 7 | User 8 | User 9 | User 10 | User 11 | User 12 | User 13 | User 14 | User 15 | User 16 | User 17 | User 18 | User 19 |
| GUI | | | | | | | | | | | | | | | | | | | |
| easy to use | 5 | 5 | 5 | 5 | 4 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 4 | 5 | 5 | 4 | 4 | 5 | 5 |
| concise | 5 | 5 | 5 | 5 | 4 | 4 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| Content | | | | | | | | | | | | | | | | | | | |
| abundancy | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 4 | 5 | 4 | 4 | 5 | 5 |
| authority | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 4 | 4 | 5 | 4 | 5 | 5 | 5 | 5 | 4 | 5 | 5 |
| helpful structure | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 4 | 5 | 5 | 5 | 5 | 5 |
| structure design | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 4 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| intersted in relative topics | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 3 | 5 | 5 | 5 |
| helpful code | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 4 | 5 | 5 | 5 | 5 | 5 |

| User feedback | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Computer Science | | | | | | | | |
| Excellent - 5 Good - 4 Fine - 3 Bad - 2 Very Bad -1 | | | | | | | | |
| | GUI | | Content | | | | | |
| User No. | easy to use | concise | abundancy | authority | helpful structure | structure design | inspiration | helpful code |
| User 1 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| User 2 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| User 3 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| User 4 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| User 5 | 4 | 4 | 5 | 5 | 5 | 5 | 5 | 5 |
| User 6 | 5 | 4 | 5 | 5 | 5 | 5 | 5 | 5 |
| User 7 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| User 8 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| User 9 | 5 | 5 | 5 | 4 | 5 | 5 | 5 | 5 |
| User 10 | 5 | 5 | 5 | 4 | 5 | 5 | 5 | 5 |
| User 11 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| User 12 | 5 | 5 | 5 | 4 | 5 | 4 | 5 | 5 |
| User 13 | 4 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| User 14 | 5 | 5 | 4 | 5 | 4 | 5 | 5 | 4 |
| User 15 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| User 16 | 4 | 5 | 5 | 5 | 5 | 5 | 3 | 5 |
| User 17 | 4 | 4 | 5 | 5 | 5 | 5 | 5 | 5 |
| User 18 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| User 19 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |

Based on the tables above, we can construct the following graph to present the trend of feedback on both GUI and contents:

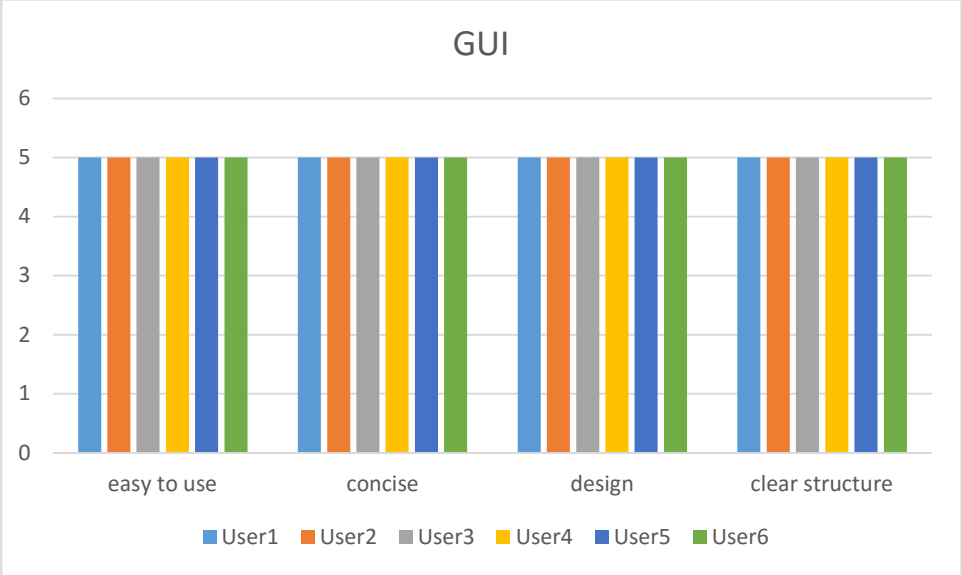


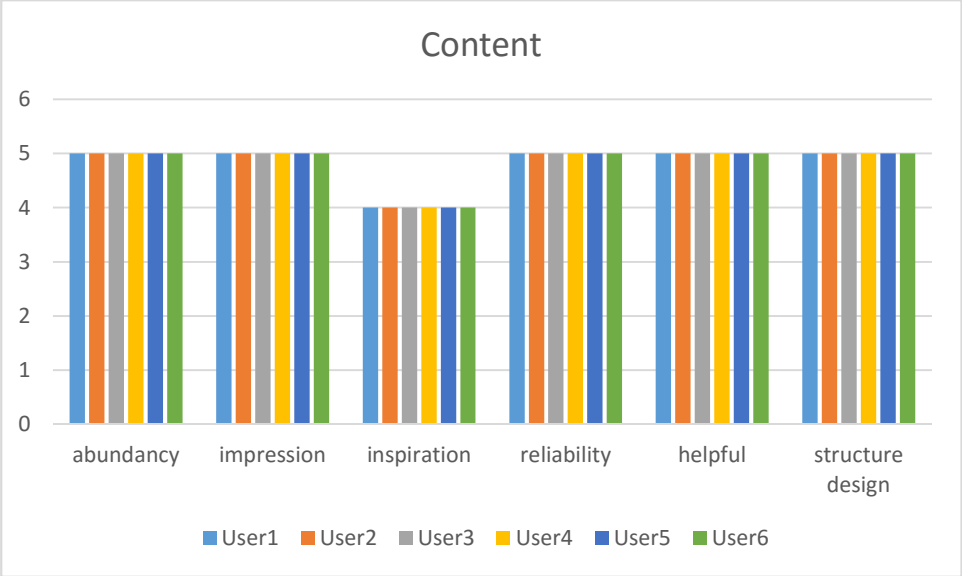

In the survey, we investigate users' feedback on the above nine factors of the website: user-friendly, concise, content, abundance, authority, structure, design, inspiration and code. We use Mark 1 to 5 to symbolize the quality of the above factors, where '5' is 'excellent,' '4' is 'good,' '3' is fine, '2' is bad and '1' is awful. Most users responded with 'excellent' on most factors, and there are just rare '4' which means 'good.' Therefore, to make an improvement on the website, the website could be more straightforward to use, the contents could be more concise and precise. However, the only one '3' is given on 'inspiration,' where it is reasonable that not everyone interested in learning more related algorithms. From users' general feedback, we can see almost every user appreciate the design and structure of the website and believe the contents and codes on the website are helpful in their learning.

We also collect six surveys from Non-Computer Science users. The feedback is presented in the following table on GUI and content:

The same as above, 5 is ‘excellent,' ‘4’ is ‘good,' ‘3’ is fine, ‘2’ is bad and ‘1’ is awful.

| Non-Computer Science User feedback | | | | | | |
|---|---|---|---|---|---|---|
| GUI | User1 | User2 | User3 | User4 | User5 | User6 |
| easy to use | 5 | 5 | 5 | 5 | 5 | 5 |
| concise | 5 | 5 | 5 | 5 | 5 | 5 |
| design | 5 | 5 | 5 | 5 | 5 | 5 |
| clear structure | 5 | 5 | 5 | 5 | 5 | 5 |
| Content | User1 | User2 | User3 | User4 | User5 | User6 |
| abundancy | 5 | 5 | 5 | 5 | 5 | 5 |
| impression | 5 | 5 | 5 | 5 | 5 | 5 |
| inspiration | 4 | 4 | 4 | 4 | 4 | 4 |
| reliability | 5 | 5 | 5 | 5 | 5 | 5 |
| helpful | 5 | 5 | 5 | 5 | 5 | 5 |
| structure design | 5 | 5 | 5 | 5 | 5 | 5 |






In the survey, we investigate users' feedback on the above ten factors of the website, where four on the GUI: user-friendly, concise, design and structure and six on contents: abundance, impression, inspiration, reliability, helpfulness and structure design. The same as before, we use Mark 1 to 5 to symbolize the quality of the above factors, where '5' is 'excellent,' '4' is 'good,' '3' is fine, '2' is bad and '1' is awful. Apparently, although users appreciated the design, content and structure of the website, they still not intense passion for the content of Dynamic Programming Algorithm itself due to the major gap. However, it proves the website is easy and accessible for a large range of user to use even without Computer Science background.

In conclusion, the website successful achieves the educational purpose of Dynamic Programming Algorithm and users responded with excellent feedback on the GUI, design and contents of the website, especially the design of the structure and self-implemented code.

The website is valuable and comprehensive for users who are eager to learn Dynamic Programming. All the users have great user-experience on the website no matter their major in Computer Science or not.

# Part VII Conclusion

In this project, a well-designed and multi-layered website is developed on Dynamic Programming for educational purposes. This website successfully provides users sufficient, authoritative, but concise and straightforward knowledge, including that of elemental concepts and knowledge of dynamic programming algorithms. To practice with real-world problems, six projects on dynamic programming are presented with analysis, optimization and implementation of self-implemented C++ codes. Furthermore, comparisons between various approaches help better understanding. Divide and Conquer and Greedy Algorithms are compared with Dynamic Programming via practical examples by execution time, code and space size. Greedy Algorithm and Dynamic Programming Algorithms are applied in the Shortest Path Problem, which enables users' comprehensive learning experiences of these algorithms. Notably, single-source shortest path problems are solved by Bellman-Ford and Dijkstra's Algorithm with Dynamic Programming and Greedy Algorithms respectively, while all-pair shortest path problems are solved by Floyd-Warshall algorithm and Johnson's algorithm for dense graphs and sparse graphs respectively. Typically, both algorithms are represented by the matrix. All the above works on algorithms are implemented by C++ and well-tested in multiple cases.

Moreover, we developed two weighted edit distance algorithms- Constraint weighted edit distance algorithm and Manhattan weighted edit distance algorithm based on the well-known Levenshtein distance. To optimize the searching strategy and provide an accurate and unique solution, we developed and precisely tested a Literature weighted edit distance algorithm. Furthermore, we implemented a weighted edit distance application based on this algorithm with C# and ASP.NET. To get a relatively accurate cost of substitution, we developed Manhattan weighted edit distance algorithm to calculate the grid distance between each pair of characters in the original string and misspelling string rather than importing a hard-code matrix.

Last but not least, to give users better learning experiences, all the implementations are presented on a website with a Message Posting Board, where any user can post comments, suggestions and feedback. Note that all the previous posters display based on time order once the contact page loaded.

All the above contents are successfully achieved in this project. As the website is for educational purpose, users' feedback is vital for evaluating the quality and value of this project. Therefore, surveys collected from 25 users' are invested as a sample to give comments and feedbacks on the website. With different knowledge background, almost every user appreciate the value of this website on Dynamic Programming, especially on design, contents and code.

This project can help users who intend to learn Dynamic Programming since this website provides comprehensive learning materials and meets users' diverse expectations. It can also benefit lecturers who teach Dynamic Programming since the website follows the natural learning process and the projects presented in the website are valuable practicing materials.

In future, there can be other weighted edit distance algorithms developed for different problems in a various range of research fields.

# Part VIII Appendices

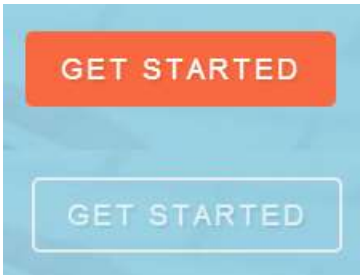
## A Features in GUI

1. Logo


2. DROPDOWN Menu


3. Get Started Button


4. Responsive Web design



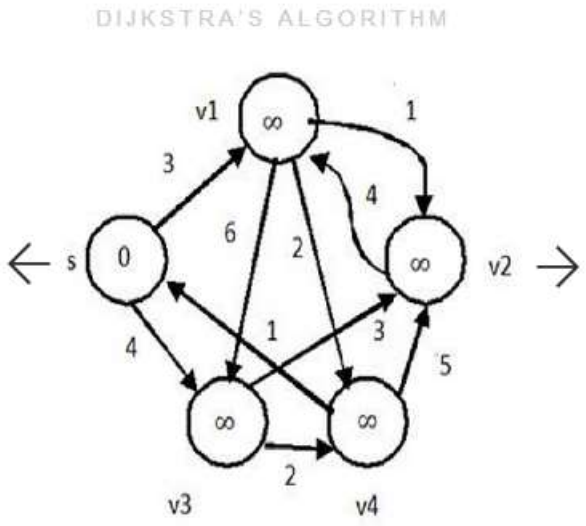

5. All the contents and inside presentation

   For instance: part of the elements page


Subproblems of the same form often arise when we make each choice. Dynamic programming is effective when a given subproblem may come from more than one partial set of choices; the key technique is to store the solution to each such subproblem for further use. This idea can sometimes transform exponential-time algorithms into polynomial-time algorithms.

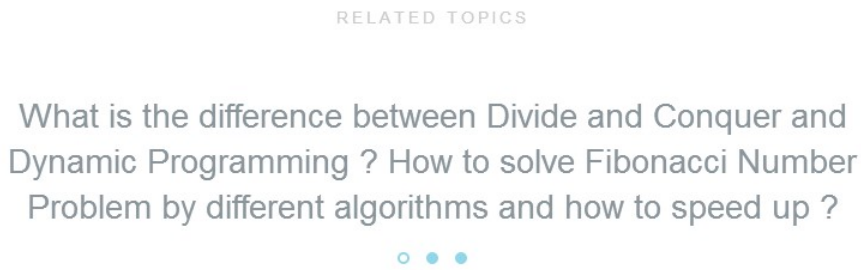
6. Scrollbar for images

   It is always straightforward and concise to present a problem or a solution by images. Therefore, for each topic, there is a scrollbar which presents as slides of images to show the expected implementation step by step.



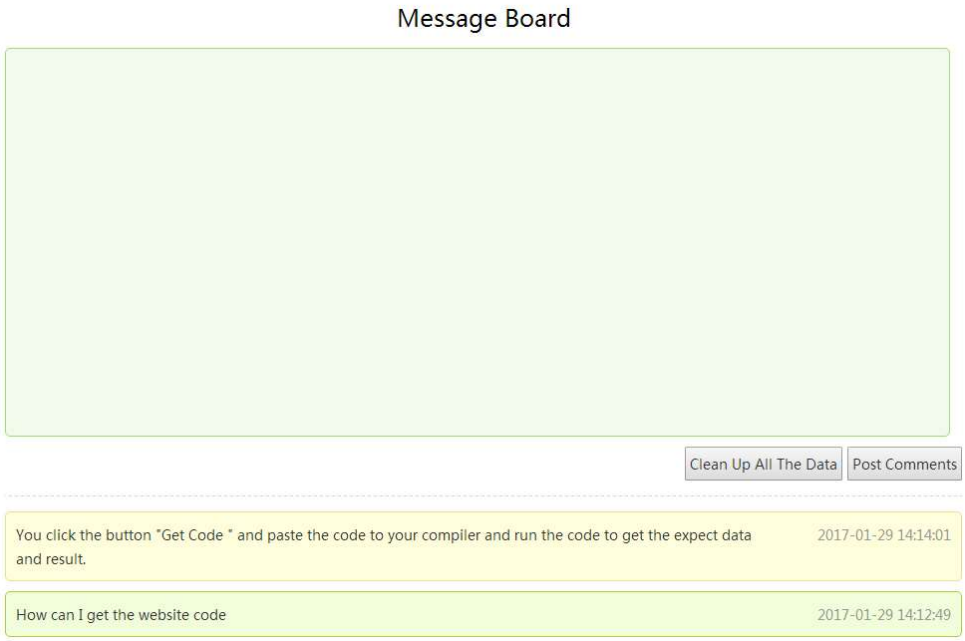

7. Scrollbar for related topics

   At the bottom of each page of the website there is a scrollbar with three topics which matches the three other algorithms in the dropdown menu to recommend the additional learning on those algorithms to users.

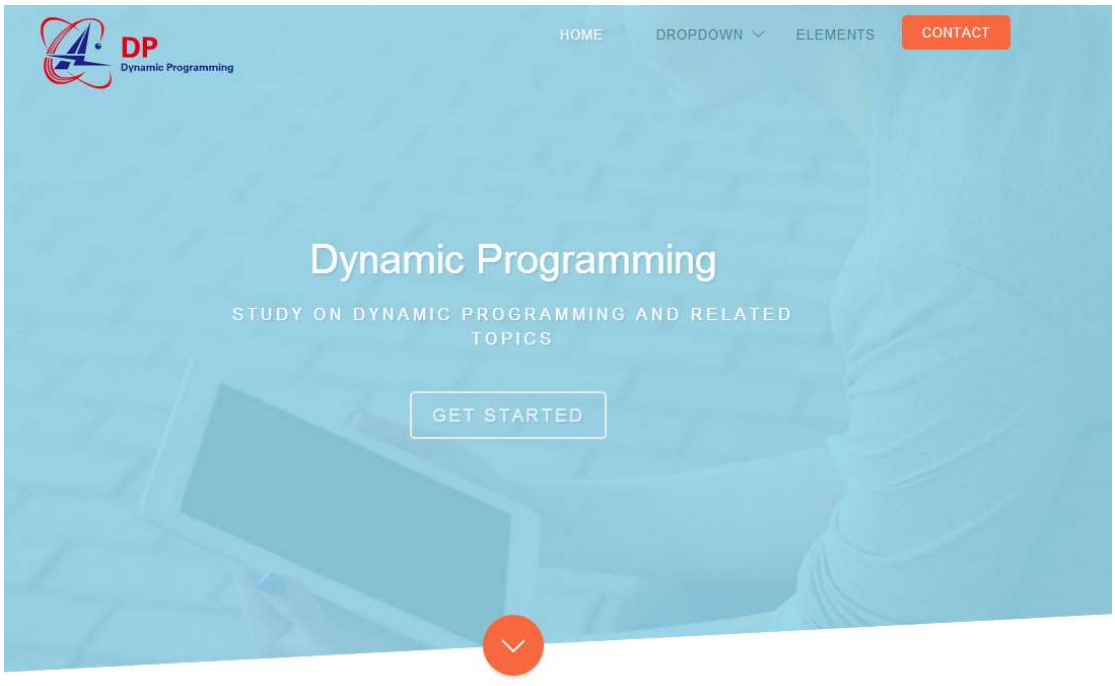



8. Contact page

   Contact page is mainly for collecting and posting user's feedback. All the comments with time and date marks will be stored locally, which enables the website to load all the posters per time when loading the contact page.

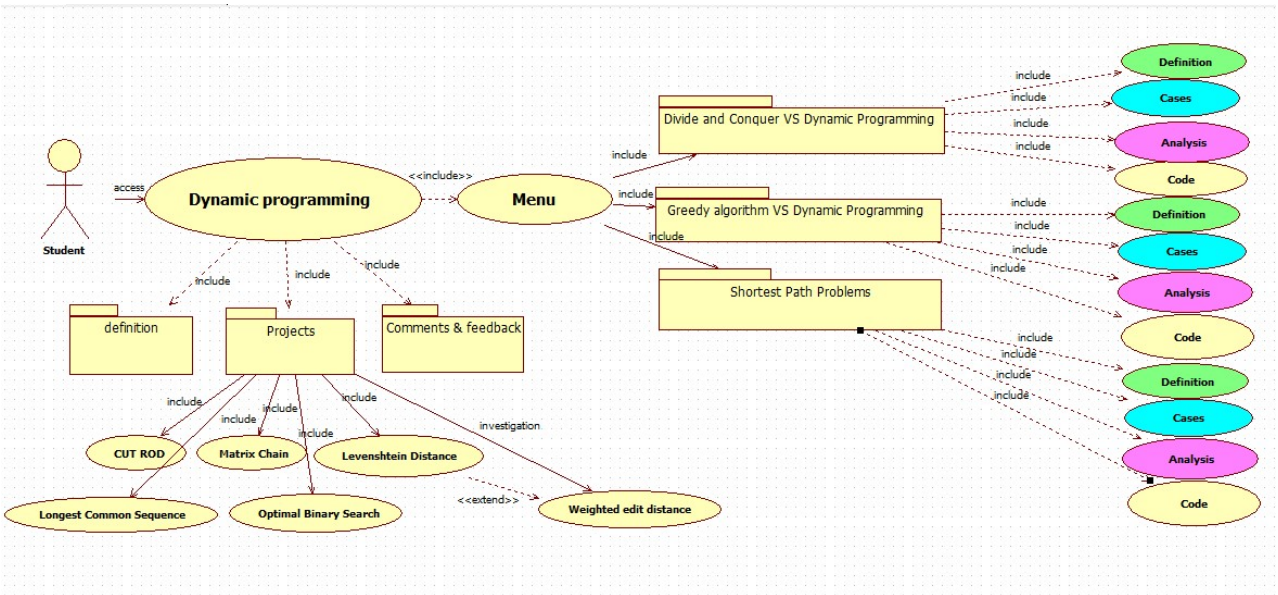



Website with template of 2015 Free Slant:




Website with template of Strip Bootstrap 4.0 Theme:


# UML




# Confusion matrix for spelling errors

Dan Jurafsky


# Confusion matrix for spelling errors

sub[X, Y] = Substitution of X (incorrect) for Y (correct)

| X \ Y (correct) | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | 0 | 0 | 7 | 1 | 342 | 0 | 0 | 2 | 118 | 0 | 1 | 0 | 0 | 3 | 76 | 0 | 0 | 1 | 35 | 9 | 9 | 0 | 1 | 0 | 5 | 0 |
| b | 0 | 0 | 9 | 9 | 2 | 2 | 3 | 1 | 0 | 0 | 0 | 5 | 11 | 5 | 0 | 10 | 0 | 0 | 2 | 1 | 0 | 0 | 8 | 0 | 0 | 0 |
| c | 6 | 5 | 0 | 16 | 0 | 9 | 5 | 0 | 0 | 0 | 1 | 0 | 7 | 9 | 1 | 10 | 2 | 5 | 39 | 40 | 1 | 3 | 7 | 1 | 1 | 0 |
| d | 1 | 10 | 13 | 0 | 12 | 0 | 5 | 5 | 0 | 0 | 2 | 3 | 7 | 3 | 0 | 1 | 0 | 43 | 30 | 22 | 0 | 0 | 4 | 0 | 2 | 0 |
| e | 388 | 0 | 3 | 11 | 0 | 2 | 2 | 0 | 89 | 0 | 0 | 3 | 0 | 5 | 93 | 0 | 0 | 14 | 12 | 6 | 15 | 0 | 1 | 0 | 18 | 0 |
| f | 0 | 15 | 0 | 3 | 1 | 0 | 5 | 2 | 0 | 0 | 0 | 3 | 4 | 1 | 0 | 0 | 0 | 6 | 4 | 12 | 0 | 0 | 2 | 0 | 0 | 0 |
| g | 4 | 1 | 11 | 11 | 9 | 2 | 0 | 0 | 0 | 1 | 1 | 3 | 0 | 0 | 2 | 1 | 3 | 5 | 13 | 21 | 0 | 0 | 1 | 0 | 3 | 0 |
| h | 1 | 8 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 12 | 14 | 2 | 3 | 0 | 3 | 1 | 11 | 0 | 0 | 2 | 0 | 0 | 0 |
| i | 103 | 0 | 0 | 0 | 146 | 0 | 1 | 0 | 0 | 0 | 0 | 6 | 0 | 0 | 49 | 0 | 0 | 0 | 2 | 1 | 47 | 0 | 2 | 1 | 15 | 0 |
| j | 0 | 1 | 1 | 9 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| k | 1 | 2 | 8 | 4 | 1 | 1 | 2 | 5 | 0 | 0 | 0 | 0 | 5 | 0 | 2 | 0 | 0 | 0 | 6 | 0 | 0 | 0 | 4 | 0 | 0 | 3 |
| l | 2 | 10 | 1 | 4 | 0 | 4 | 5 | 6 | 13 | 0 | 1 | 0 | 0 | 14 | 2 | 5 | 0 | 11 | 10 | 2 | 0 | 0 | 0 | 0 | 0 | 0 |
| m | 1 | 3 | 7 | 8 | 0 | 2 | 0 | 6 | 0 | 0 | 4 | 4 | 0 | 180 | 0 | 6 | 0 | 0 | 9 | 15 | 13 | 3 | 2 | 2 | 3 | 0 |
| n | 2 | 7 | 6 | 5 | 3 | 0 | 1 | 19 | 1 | 0 | 4 | 35 | 78 | 0 | 0 | 7 | 0 | 28 | 5 | 7 | 0 | 0 | 1 | 2 | 0 | 2 |
| o | 91 | 1 | 1 | 3 | 116 | 0 | 0 | 0 | 25 | 0 | 2 | 0 | 0 | 0 | 0 | 14 | 0 | 2 | 4 | 14 | 39 | 0 | 0 | 0 | 18 | 0 |
| p | 0 | 11 | 1 | 2 | 0 | 6 | 5 | 0 | 2 | 9 | 0 | 2 | 7 | 6 | 15 | 0 | 0 | 1 | 3 | 6 | 0 | 4 | 1 | 0 | 0 | 0 |
| q | 0 | 0 | 1 | 0 | 0 | 0 | 27 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| r | 0 | 14 | 0 | 30 | 12 | 2 | 2 | 8 | 2 | 0 | 5 | 8 | 4 | 20 | 1 | 14 | 0 | 0 | 12 | 22 | 4 | 0 | 0 | 1 | 0 | 0 |
| s | 11 | 8 | 27 | 33 | 35 | 4 | 0 | 1 | 0 | 1 | 0 | 27 | 0 | 6 | 1 | 7 | 0 | 14 | 0 | 15 | 0 | 0 | 5 | 3 | 20 | 1 |
| t | 3 | 4 | 9 | 42 | 7 | 5 | 19 | 5 | 0 | 1 | 0 | 14 | 9 | 5 | 5 | 6 | 0 | 11 | 37 | 0 | 0 | 2 | 19 | 0 | 7 | 6 |
| u | 20 | 0 | 0 | 0 | 44 | 0 | 0 | 0 | 64 | 0 | 0 | 0 | 0 | 2 | 43 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 2 | 0 | 8 | 0 |
| v | 0 | 0 | 7 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 8 | 3 | 0 | 0 | 0 | 0 | 0 | 0 |
| w | 2 | 2 | 1 | 0 | 1 | 0 | 0 | 2 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 7 | 0 | 6 | 3 | 3 | 1 | 0 | 0 | 0 | 0 | 0 |
| x | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| y | 0 | 0 | 2 | 0 | 15 | 0 | 1 | 7 | 15 | 0 | 0 | 0 | 2 | 0 | 6 | 1 | 0 | 7 | 36 | 8 | 5 | 0 | 0 | 1 | 0 | 0 |
| z | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 7 | 5 | 0 | 0 | 0 | 0 | 2 | 21 | 3 | 0 | 0 | 0 | 0 | 3 | 0 |

# B Surveys

Survey on FYP

(Computer Science)

Project Topic: Dynamic Programming

User name:

Student ID:

Questions (Please give marks from 1 to 5 on each question and leave your feedback at the end, thank you for cooperation!)

GUI:

1. Is the website easy enough to use?
2. Is the web page concise and straightforward?
3. Do you have any problem on using the website?

Content:

1. Does the contents of the website abundant enough?
2. Does the contents of the website authoritative enough?
3. Does the structure of website help you to learn better and deeper?
4. Do you like the design of the structure?
5. Do you like to learn the relative topics (Divide and Conquer, Greedy Algorithm and Shortest Path) with Dynamic Programing?
6. Do you think the code help you to learn better of Dynamic Programming?
7. Is there any other topics you are interested in related to Dynamic Programming?

General feedback:

Survey on FYP

(Non-computer major)

Project Topic: Dynamic Programming

User name:

Student ID:

Major:

Questions (Please give marks from 1 to 5 on each question and leave your feedback at the end, thank you for cooperation!)

GUI:

1. Is the website easy enough to use?
2. Is the web page concise and straightforward?
3. Do you like the design of webpages?
4. Do you find it is easy follow the structure of the website?

Content:

1. Is the content of the website abundant enough?
2. To what extent are you impressed by the website?
3. Do you think this website would invoke your interest in learning Dynamic Programming?
4. If you were going to learn Dynamic Programming Algorithm, do you think the website would be a good source?
5. Does the structure of website help you to learn?
6. Do you like the design of the structure?

General feedback:

# Part IX References

[1]Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C. (2001), Introduction to Algorithms (2nd ed.), MIT Press & McGraw–Hill, ISBN 0-262-03293-7 . pp. 327–8.

[2] Thomas.H.Cormen.Ronald.L.Rivest.Charles.E.Leiserson.Clifford.Stein. Introduction to Algorithms. 3rd. Massachusetts Institute of Technology, 2009, 15, 360-369. ISBN 978-0-262-53305-8 (pbk. : alk. paper).

[3]Thomas.H.Cormen.Ronald.L.Rivest.Charles.E.Leiserson.Clifford.Stein. Introduction to Algorithms. 3rd. Massachusetts Institute of Technology, 2009, 15, 370-377. ISBN 978-0-262-53305-8 (pbk. : alk. paper).

[4]Thomas.H.Cormen.Ronald.L.Rivest.Charles.E.Leiserson.Clifford.Stein. Introduction to Algorithms. 3rd. Massachusetts Institute of Technology, 2009, 15, 390-396. ISBN 978-0-262-53305-8 (pbk. : alk. paper).

[5]Thomas.H.Cormen.Ronald.L.Rivest.Charles.E.Leiserson.Clifford.Stein. Introduction to Algorithms. 3rd. Massachusetts Institute of Technology, 2009, 15, 397-403. ISBN 978-0-262-53305-8 (pbk. : alk. paper).

[6]Thomas.H.Cormen.Ronald.L.Rivest.Charles.E.Leiserson.Clifford.Stein. Introduction to Algorithms. 3rd. Massachusetts Institute of Technology, 2009. ISBN 978-0-262-53305-8 (pbk. : alk. paper).

[7]Copyright 2010 Jeff Erickson. Released under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 License (http://creativecommons.org/licenses/by-nc-sa/3.0/).

[8]Thomas.H.Cormen.Ronald.L.Rivest.Charles.E.Leiserson.Clifford.Stein. Introduction to Algorithms. 3rd. Massachusetts Institute of Technology, 2009, 24, 643. ISBN 978-0-262-53305-8 (pbk. : alk. paper).

[9]Thomas.H.Cormen.Ronald.L.Rivest.Charles.E.Leiserson.Clifford.Stein. Introduction to Algorithms. 3rd. Massachusetts Institute of Technology, 2009, 24, 643-650. ISBN 978-0-262-53305-8 (pbk. : alk. paper).

[10]Thomas.H.Cormen.Ronald.L.Rivest.Charles.E.Leiserson.Clifford.Stein. Introduction to Algorithms. 3rd. Massachusetts Institute of Technology, 2009, 24, 651-654. ISBN 978-0-262-53305-8 (pbk. : alk. paper).

[11]Thomas.H.Cormen.Ronald.L.Rivest.Charles.E.Leiserson.Clifford.Stein. Introduction to Algorithms. 3rd. Massachusetts Institute of Technology, 2009, 24, 658-662. ISBN 978-0-262-53305-8 (pbk. : alk. paper).

[12]Thomas.H.Cormen.Ronald.L.Rivest.Charles.E.Leiserson.Clifford.Stein. Introduction to Algorithms. 3rd. Massachusetts Institute of Technology, 2009, 25, 684-691. ISBN 978-0-262-53305-8 (pbk. : alk. paper).

[13]Thomas.H.Cormen.Ronald.L.Rivest.Charles.E.Leiserson.Clifford.Stein. Introduction to Algorithms. 3rd. Massachusetts Institute of Technology, 2009, 25, 693-699. ISBN 978-0-262-53305-8 (pbk. : alk. paper).

[14]Thomas.H.Cormen.Ronald.L.Rivest.Charles.E.Leiserson.Clifford.Stein. Introduction to Algorithms. 3rd. Massachusetts Institute of Technology, 2009, 25, 700-704. ISBN 978-0-262-53305-8 (pbk. : alk. paper).

[15] Furkanavcu, Student, University Of Kocaeli, Turkey. Copyright © CodeProject, 1999-2017. CodeProject.com.

[16]Thomas.H.Cormen.Ronald.L.Rivest.Charles.E.Leiserson.Clifford.Stein. Introduction to Algorithms. 3rd. Massachusetts Institute of Technology, 2009, 15, 359. ISBN 978-0-262-53305-8 (pbk. : alk. paper).

[17]Thomas.H.Cormen.Ronald.L.Rivest.Charles.E.Leiserson.Clifford.Stein. Introduction to Algorithms. 3rd. Massachusetts Institute of Technology, 2009, 15, 362. ISBN 978-0-262-53305-8 (pbk. : alk. paper).

[18] [19] [20] [21]
Thomas.H.Cormen.Ronald.L.Rivest.Charles.E.Leiserson.Clifford.Stein. Introduction to Algorithms. 3rd. Massachusetts Institute of Technology, 2009, 15, 365. ISBN 978-0-262-53305-8 (pbk. : alk. paper).

[22]Thomas.H.Cormen.Ronald.L.Rivest.Charles.E.Leiserson.Clifford.Stein. Introduction to Algorithms. 3rd. Massachusetts Institute of Technology, 2009, 15, 379-380. ISBN 978-0-262-53305-8 (pbk. : alk. paper).

[23]Thomas.H.Cormen.Ronald.L.Rivest.Charles.E.Leiserson.Clifford.Stein. Introduction to Algorithms. 3rd. Massachusetts Institute of Technology, 2009, 15, 384. ISBN 978-0-262-53305-8 (pbk. : alk. paper).

[24]Free HTML5 Template by FREEHTML5.CO. 2015 Free Slant. URL: http://www.cssmoban.com/

[25]Strip Bootstrap 4.0 Theme URL: http://www.cssmoban.com/

The website template quote following files:

bootstrap.min.js

Bootstrap v4.0.0-alpha.2 (http://getbootstrap.com)，Copyright 2011-2015 Twitter, Inc.Licensed under MIT (https://github.com/twbs/bootstrap/blob/master/LICENSE). normalize.css v3.0.3 | MIT License | github.com/necolas/normalize.csssourceMappingURL=bootstrap.min.css.map

jquery.singlePageNav.min.js

Single Page Nav Plugin, Copyright (c) 2014 Chris Wojcik <hello@chriswojcik.net>, Dual licensed under MIT and GPL.@author Chris Wojcik. @version 1.2.0.

jquery-1.11.3.min.js

jQuery v1.11.3 | (c) 2005, 2015 jQuery Foundation, Inc. | jquery.org/license

[26] Beck, Matthias; Geoghegan, Ross (2010), *The Art of Proof: Basic Training for Deeper Mathematics, New York: Springer.*

Bóna, Miklós *(2011), A Walk Through Combinatorics (3rd ed.), New Jersey: World Scientific*.

Fibonacci numbers, N.J.A.Sloane, THE ON-LINE ENCYCLOPEDIA OF INTEGER SEQUENCES, 1964. URL：https://oeis.org/A000045

[27] Thomas.H.Cormen.Ronald.L.Rivest.Charles.E.Leiserson.Clifford.Stein. Introduction to Algorithms. 3rd. Massachusetts Institute of Technology, 2009, 4, 65. ISBN 978-0-262-53305-8 (pbk. : alk. paper).

[28]Dumitru,Dynamic Programming – From Novice to Advanced, URL: https://www.topcoder.com/community/data-science/data-science-tutorials/dynamic-programming-from-novice-to-advanced/

[29] S. K. Chang and A. Gill. Algorithmic solution of the change-making problem. J. Assoc. Comput. Math., 17(1): 113-122, January, 1970

[30] Dexter Kozen and Shmuel Zaks. Optimal Bounds for the Change-Making Problem. Computer Science Department, Cornell University Ithaca, New York 148553, USA. Computer Science Department, Technion Haifa, Israel.

[31] Cormen, Thomas H.*;* Leiserson, Charles E.*;* Rivest, Ronald L.*;* Stein, Clifford*(2001) [1990]. "Chapter 20: Fibonacci Heaps".* Introduction to Algorithms *(2nd ed.). MIT Press and McGraw-Hill. pp. 476–497.* ISBN 0-262-03293-7. Third edition p. 518.

Fredman, Michael Lawrence; Tarjan, Robert E. (July 1987). "Fibonacci heaps and their uses in improved network optimization algorithms" (PDF). Journal of the Association for Computing Machinery. 34(3): 596–615. doi:10.1145/28869.28874.

[32] Confusion matrix for spelling errors

Minimum Edit Distance

Online slide- https://web.stanford.edu/class/cs124/lec/med.pdf

[33]"Manhattan distance." *Definitions.net.* STANDS4 LLC, 2017. Web. 16 Apr. 2017. <http://www.definitions.net/definition/Manhattan distance>.