UNIVERSITAT DE BARCELONA

Facultat de Matemàtiques i Informàtica

GRAU DE MATEMÀTIQUES

Treball final de grau

DYNAMIC PROGRAMMING AND DNA SEQUENCE ALIGNMENT ALGORITHMS

Autor: Marc Jordà Mascaró

Director: Dr. Miquel Bosch Gual

Realitzat a: Departament de Matemàtiques i Informàtica

Barcelona, 19 de juny de 2019

Contents

Abstract

The goal of this work is to present dynamic programming, which is a mathematical field that solves optimisation problems based on multistage decision-making processes. First, its mathematical foundations are gradually built up based on the theorem of optimality, the functional equation and the principle of optimality. Next, the basic elements of the computational procedure are presented, including its most remarkable advantages and drawbacks in comparison to other more exhaustive computational methods. Finally, several up-to-date problems in bioinformatics are introduced in order to compare DNA and protein sequences, which is useful to find out unknown gene functions and compare the genome of different species. Computer algorithms to solve these problems have been written and attached to this work.

2010 Mathematics Subject Classification. 49L20, 90C35, 90C39, 90C90, 92D20

Acknowledgements

To my advisor Dr. Miquel Bosch, for having accepted to supervise a work about the topic I wanted, for concrete suggestions about the content and, especially, for having shown such a big degree of confidence in my abilities.

To Dr. Yi Li, who introduced me in the field of the operations research in the University of Sheffield, appreciated my work there and suggested me dynamic programming as a topic for this research project.

To my parents, whose first priority was always their children's education. Going through with their support was so easy that anything I achieve in my life is as theirs as mine.

To my sister Sara, my guiding light, the first opinion I need to hear in any situation.

To my whole family and to the ones who are not here anymore.

To Andrea. To Anna. For having given so much to me these last years that I will never be able to give it back in return.

To my town friends. Because I could not think in a better atmosphere, even when difficulties showed up, to feel the luckiest person on Earth.

To all the friends I made at University, whose undeniable brightness definitely made me a better mathematician, a better physicist and, above all, a better person.

To the friends I made last year in Sheffield, who changed my life entirely.

To my current workmates, for their interest and laughs which made this last effort much easier to deal with.

Thank you very much.

Introduction

Dynamic programming is a field of mathematics highly related to operations research which deals with optimisation problems by giving particular approaches which are able to easily solve some complex problems which would be unfeasible in almost any other way.

The basic idea behind dynamic programming is to consider problems in which several decisions must be consecutively made to minimise a cost or maximise a profit. Next, the situation is split into single-decision optimisation subproblems, which are much easier to solve. Then, some iteration is defined to match the solutions of these subproblems so as to construct the answer for the larger one. What is more, computations are organized in such a way to avoid recalculating already known values and then saving so much computing time.

First of all, the first chapter provides a gradual theoretical construction, accounting for the appropriate hypothesis and restrictions and carefully justifying every step, in order to build up a suitable framework which can properly justify later results.

In any case, dynamic programming is essentially intended to be used by computer programs. So, the second chapter describes the basic aspects of the computational procedure. It does not only show its advantages against other more common exhaustive methods, but also clearly states its limitations.

In addition, the most essential aspect of this optimisation field is its high degree of applicability. Namely, the third chapter applies the dynamic programming method to the alignment of DNA and protein sequences, which is an up-to-date bioinformatics application really useful to discover unknown gene functions, find out causes of diseases or look for evolutionary similarities between different species. With this goal in mind, I enrolled in a massive open online course (MOOC) offered by the University of California in the edX website called *Dynamic Programming: Applications In Machine Learning and Genomics* [10], whose first half contents are reflected in this work.

Eventually, the computer algorithms to solve the different problems introduced in the third chapter have been written as suggested and assessed activities for the mentioned online course. Actually, they are attached in the appendix, in *Python* programming language, which I also specifically learned to do this work.

Chapter 1

Dynamic programming fundamentals

This first chapter gives the basic theory required to understand the dynamic programming fundamentals in order to thereafter explain its computational methods and some of its applications.

First of all, multistage decision processes are introduced, because dynamic programming can be basically regarded as the optimisation of such processes. Later on, this part of the work sheds light on the basic approach of dynamic programming to solve problems, introducing the theorem of optimality and the functional equation from basic principles as the key points to build on a proper theoretical structure. Afterwards, the principle of optimality is stated and its relation with the previous results is discussed. Eventually, the basic procedure to recover optimal decisions and optimal trajectories from an optimal solution is also described.

The development of the content of this chapter is based on a rearrangement of different elements of the references [1], [3] and [4].

1.1 Multistage decision processes

The theory of dynamic programming provides a framework to deal with the optimisation of multistage decision processes, which are described in detail in this first section.

1.1.1 Multistage processes

We will start by defining the basic concepts to build up a multistage process.

Definition 1.1. *States*:

- A *state space* is a nonempty set X which contains all the possible configurations of a system. In our case, X will be a nonempty subset of $\mathbb{R}^n$.
- A *state* is a specific point in the state space X. The *state vector* x is the vector of the n coordinates of a state in the set X which uniquely specify it in $\mathbb{R}^n$:

$$x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

The n components of the state vector x are called *state variables*.

We consider now a transformation $g : X \to X$ which produces the sequence of states $[x(0), x(1), x(2), \ldots]$ as it follows:

$$\begin{aligned} x(1) &= g(x(0)) \\ x(2) &= g(x(1)) = g^2(x(0)) \\ &\vdots \\ x(k+1) &= g(x(k)) = g^{k+1}(x(0)) \\ &\vdots \end{aligned} \tag{1.1}$$

Definition 1.2. The *stage variable* k is the index of the sequence and it determines the order in which events occur in the system.

Definition 1.3. A *multistage process* is the set of elements $[x(0), x(1), x(2), \ldots]$. Equivalently, it is the pair $[x(0), g]$, because both items uniquely specify the previous sequence. We will call the finite segment $[x(0), \ldots, x(N)]$ an *N-stage process*.

We will work with deterministic systems accomplishing the property of causality.

Definition 1.4. A system is *deterministic* if no randomness is involved in determining its future states.

Definition 1.5. A system is *causal* if the output at stage k only depends on values of the input at stages $k' \leq k$.

The property of causality is a requirement for a system to be feasible in reality and it is accomplished by most physical systems. As for multistage processes, we can infer from equations (1.1) that, given the state $x(k)$, all later elements are

determined by $x(k)$ and the transformation g. In other words, the future states are only determined by the current state and the transformation law, and not by the particular path followed in reaching the current state.
So far, it has been assumed that the transformation g was the same in every application, but it could be considered stage-dependent:

$$x(k+1) = g(k, x(k)) \tag{1.2}$$

In addition, it can be mentioned that state variables might be constrained in a way depending on the stage of the system. This is why we introduce the following sets:

Definition 1.6. $X_i \subset X$ is denoted as the *set of all the feasible states belonging to stage i*. Namely, X_0 is a non-empty subset of X whose elements are called *initial states*.

1.1.2 Decisions

The concept of decision-making will be introduced so as to obtain a far-reaching generalization of multistage processes. The purpose of these decisions will be to obtain optimal values for certain functions.

We begin with the discrete, deterministic N-stage process $[x(0), g]$. Now, the transformation g also depends on a *decision* u which is at our disposal:

$$g = g(k, x(k), u(k)) \tag{1.3}$$

Definition 1.7. *Decisions*:

- A *decision* u is a value which can be chosen directly to affect the state variables in some prescribed way.
- The *decision space* U is a subset of $\mathbb{R}^m$ where the decision u is selected from.
- A *decision vector* is a m-dimensional vector where the coordinates of u are arranged:

$$u = \begin{bmatrix} u_1 \\ \vdots \\ u_m \end{bmatrix}$$

  The components of u are called *decision variables*.

The multistage process is now:

$$x(k+1) = g(k, x(k), u(k)), \qquad k = 0, \dots, N-1, \tag{1.4}$$

where $x(0)$ is a given initial state. These relations are called *system equations* and define how the state variables at stage $k+1$ are related to the state and decision variables at stage k.
So, our N-stage decision process is determined by the following elements:

$$x(0), \dots, x(N); u(0), \dots, u(N)$$

Definition 1.8. The *decision sequence* is the sequence of elements $\mathcal{U} = [u(0), \dots, u(N)]$.

In any multistage process, the goal of making decisions is to find the optimal value for certain functions:

Definition 1.9. The *criterion function* is a scalar function J generated by the states and decisions $J = J(x(0), \dots, x(N); u(0), \dots, u(N))$ to be maximised or minimised and provides an evaluation of a given decision sequence $\mathcal{U}$. If the criterion function is maximised, it is called *objective function*. If it is minimised, it is called *cost function* instead.

Take note that $x(0)$ is a given initial condition and that it can be seen from equations (1.4) that later states are worked out based on the state and decision at their immediately previous stage. Then, J can be only expressed in terms of the decision sequence:

$$J = J(u(0), \dots, u(N))$$

The values of the decision variables are usually constrained differently depending on the state and stage of the system. This is why we introduce the following sets:

Definition 1.10. The set $U_{k,x(k)} \subset U$ is considered the *set of feasible decisions at stage k given the state* $x(k)$.

In consequence, we can introduce the following recurrence from definition 1.6 about the way states are constrained:

$$X_{k+1} := \{g(k, x(k), u(k)) : x(k) \in X_k,\ u(k) \in U_{k,x(k)}\}, \qquad 0 \le k < N \tag{1.5}$$

One of the main purposes of dynamic programming is to identify the rules behind decisions. This is why we introduce the concept of policies:

Definition 1.11. *Policies*:

- *Policies* refer to the rules that prescribe which decisions are made.
- *Feasible policies* refer to rules according to feasible sequences of decisions.
- *Optimal policies* determine an application of feasible rules which provides an optimal sequence of decisions which minimise (or maximise) the criterion function J.

There are numerous problems of great practical interest which can be approached using dynamic programming ideas about making decisions one stage at a time [2]. However, we need some restrictions to make J analytic and computational headway. For example, we can firstly assume that the current decision is a function only of the current state and current stage. Formally:

Definition 1.12. *Markovian policies* accomplish that to determine the decision at stage k only it is required information about the state currently observed at this stage, i.e.,

$$u(k) = u(k, x(k)), \tag{1.6}$$

and that $u(k, x(k)) \in U_{k,x(k)}$ for any pair $(k, x(k))$ for which the set $U_{k,x(k)}$ is not empty.

All in all, we have set up a suitable mathematical framework as a base to build on the basic theory of dynamic programming.

1.2 Theoretical foundations

The goal of this section is to develop an analytical structure to study multistage decision processes using dynamic programming ideas.
We will consider from now on the following minimisation problem:

Problem 1.13.

$$\min_{(u(0),\ldots,u(N))\in\bar{\mathcal{S}}} J(u(0),\ldots,u(N)) \tag{1.7}$$

where J is a real function of the variables $u(0),\ldots,u(N)$ and $\bar{\mathcal{S}} \subset \mathbb{R}^{N+1}$ is the *set of all the feasible solutions to this problem*. Following the framework stated in previous sections, system equations (1.4) hold, $x(0) = c$ is a given initial state and constraints $x(k) \in X_k$ and $u(k, x(k)) \in U_{k,x(k)}$ must be satisfied $\forall k = 0,\ldots,N$.
On the other hand, J is considered to be a continuous function of all their arguments. States and decisions are ranged over closed, bounded subsets of $\mathbb{R}^n$ and $\mathbb{R}^m$, respectively. By *the Weierstrass theorem* [7], a minimising policy exists.

The same reasoning that will be followed here can be obviously applied to study maximisation problems in an analogous way.

The basic idea is that dynamic programming approaches this problem considering it as a member of a class of similar problems. Namely, we want to find an embedding[1] for the problem such that:

[1]An *embedding*, or *imbedding*, generally refers to the representation of a mathematical object in a certain space in such a way that its properties are preserved [14].

(a) One member of the family of problems has a relatively simple solution.

(b) Relations are obtained linking members of the family of problems in a certain way.

The goal, then, is to obtain the solution of the problem, which may be difficult to solve at first, starting with the solution of the simple problem and then using the relations between the members of the family to solve the rest of them.

First, the relations specified in the step **(b)** of the embedding will be attempted by means of the introduction of the *theorem of optimality* and *the functional equation*. Then, the simple member of the family described in step **(a)** will be found straightforward as a particular case of the functional equation.
The basic idea behind this procedure is to transform our complex problem of $N+1$ variables into a set of $N+1$ much simpler problems of a single variable. This way of proceeding can be regarded as a *decomposition method*.

1.2.1 Theorem of optimality

The first phase of the decomposition method will be to split the problem 1.13 of $N+1$ variables into two different problems of less variables each. This is why we will introduce the optimality theorem. Before that, we need to make some additional assumptions about J. To handle easier notation, we set $u(k) \equiv u_k \ \forall k$ and $\vec{u}$ as the N-vector of components $u_1, \dots, u_N$.

Definition 1.14. Our criterion function J depending on $N+1$ variables $u_0, u_1, \dots u_N$ is said to be *separable* if it can be expressible in terms of other $N+1$ scalar functions $J_0, J_1, \dots, J_N$ such that:

$$\begin{aligned} J[u_0, \dots, u_N] &= J_0\{u_0, \overline{J}_1[u_1, \dots, u_N]\} \\ \overline{J}_1[u_1, \dots, u_N] &= J_1\{u_1, \overline{J}_2[u_2, \dots, u_N]\} \\ \overline{J}_2[u_2, \dots, u_N] &= J_2\{u_2, \overline{J}_3[u_3, \dots, u_N]\} \\ &\vdots \\ \overline{J}_N[u_N] &= J_N[u_N] \end{aligned} \tag{1.8}$$

Definition 1.15. Our criterion function J depending on variables u_0 and $\vec{u}$ is *decomposable* into two other scalar functions J_0 and J_1 if J is separable, i.e.:

$$J(u_0, \vec{u}) = J_0(u_0, J_1(\vec{u})), \tag{1.9}$$

and, moreover, the function J_0 is monotone non-decreasing relative to its second argument.

On the other hand, given an initial state $x(0)$, for every $u_0 \in U_{0,x(0)}$ we define:

$$\bar{\mathcal{S}}_{u_0} = \{\vec{u} : \vec{u} \in \mathbb{R}^N; (u_0, \vec{u}) \in \bar{\mathcal{S}}\} \tag{1.10}$$

Since it could happen that $\bar{\mathcal{S}}_{u_0} = \emptyset$ for some values of u_0, we agree beforehand that

$$\min_{\vec{u} \in \bar{\mathcal{S}}_{u_0}} \{J_1(\vec{u})\} = +\infty, \qquad \text{if} \qquad \bar{\mathcal{S}}_{u_0} = \emptyset$$

and also

$$\forall u_0 \in U_{0,x(0)} : \qquad J_1(u_0, +\infty) = +\infty, \qquad J_1(u_0, -\infty) = -\infty$$

Now, we can finally state the *optimality theorem*.

Theorem 1.16. *Let J be a real function of $u_0 \equiv u(0)$ and $\vec{u} = (u(1), \ldots, u(N))$. If J is decomposable with $J(u_0, \vec{u}) = J_0(u_0, J_1(\vec{u}))$, then:*

$$\min_{(u_0,\vec{u}) \in \bar{\mathcal{S}}} \{J(u_0, \vec{u})\} = \min_{u_0 \in U_{0,x(0)}} \left\{ J_0 \left(u_0, \min_{\vec{u} \in \bar{\mathcal{S}}_{u_0}} \{J_1(\vec{u})\} \right) \right\} \tag{1.11}$$

Proof. By the definition of a minimum for $J(u_0, \vec{u}) = J_0(u_0, J_1(\vec{u}))$, we can state:

$$\forall\ u_0' \in U_{0,x(0)}, \forall\ \vec{u}' \in \bar{\mathcal{S}}_{u_0} : \qquad \min_{(u_0,\vec{u}) \in \bar{\mathcal{S}}} J_0(u_0, J_1(\vec{u})) \leq J_0(u_0', J_1(\vec{u}'))$$

Particularly, taking $\vec{u}' = \vec{u}''$ such that $J_1(\vec{u}'') = \min_{\vec{u} \in \bar{\mathcal{S}}_{u_0}} \{J_1(\vec{u})\}$, we get:

$$\forall\ u_0' \in U_{0,x(0)} : \qquad \min_{(u_0,\vec{u}) \in \bar{\mathcal{S}}} J_0(u_0, J_1(\vec{u})) \leq J_0(u_0', J_1(\vec{u}''))$$

This latest inequality is still true when choosing for u_0' the value which minimises the right-hand side, so:

$$\min_{(u_0,\vec{u}) \in \bar{\mathcal{S}}} J_0(u_0, J_1(\vec{u})) \leq \min_{u_0 \in U_{0,x(0)}} \left\{ J_0 \left(u_0, \min_{\vec{u} \in \bar{\mathcal{S}}_{u_0}} \{J_1(\vec{u})\} \right) \right\}$$

This shows one direction of the inequality. As for the opposite one, we can use that J_0 is monotone non-decreasing relative to its second argument:

$$\forall\ u_0' \in U_{0,x(0)}, \forall\ \vec{u}' \in \bar{\mathcal{S}}_{u_0} : \qquad J_0 \left(u_0', \min_{\vec{u} \in \bar{\mathcal{S}}_{u_0}} \{J_1(\vec{u})\} \right) \leq J_0(u_0', J_1(\vec{u}'))$$

This relation still holds by taking the value of $\vec{u}'$ which minimises the right-hand side:

$$\forall\ u_0' \in U_{0,x(0)} : \qquad J_0 \left(u_0', \min_{\vec{u} \in \mathcal{S}_{u_0}} \{J_1(\vec{u})\} \right) \leq \min_{\vec{u} \in \mathcal{S}_{u_0}} J_0(u_0', J_1(\vec{u}))$$

We then take for u_0' the value u_0'' which minimises on u_0 the right-hand side. Hence:

$$\min_{u_0 \in U_{0,x(0)}} \left\{ J_0 \left(u_0, \min_{\vec{u} \in \bar{\mathcal{S}}_{u_0}} \{J_1(\vec{u})\} \right) \right\} \leq J_0 \left(u_0'', \min_{\vec{u} \in \mathcal{S}_{u_0}} \{J_1(\vec{u})\} \right) \leq \min_{(u_0,\vec{u}) \in \bar{\mathcal{S}}} \{J(u_0, \vec{u})\}$$

And this completes the proof. □

This theorem achieves to convert the optimisation of a function depending on $N+1$ variables into two different optimisation problems, the first one with a function depending on a single variable and the second one with another function depending on N variables.

1.2.2 Functional equation of dynamic programming

The optimality theorem is essential since it can be the root to build on a system of solution methods for dynamic programming problems. The key point is to regard the domain of feasible solutions $\bar{\mathcal{S}}$ of the original problem as an element of a larger family of domains parameterized by the stage and the state variables:

$$\mathcal{S}_{k,x(k)} := \{(u(k), \ldots, u(N)) : u(j) \in U_{j,x(j)}, \qquad k \leq j \leq N\} \tag{1.12}$$

By construction, for a given $x(0) \in X_0$, we easily see that $\mathcal{S}_{0,x(0)} \equiv \bar{\mathcal{S}}$.

We start by considering a general constrained optimisation problem in variables $u(0), \ldots, u(N)$ described by:

Problem 1.17. Consider the previous problem 1.13:

$$I^* = \min_{(u(0),\ldots,u(N)) \in \bar{\mathcal{S}}} J[u(0), \ldots, u(N)] \tag{1.13}$$

But now, moreover, J also satisfyes the hypotheses of theorem 1.16.

The basic idea of the procedure involves replacing the solution of the problem 1.17, which consists of $N+1$ variables, by the solution of a whole family of optimisation problems only containing N variables:

Problem 1.18. Consider J_1 as the function specified in (1.9) resulting from the separability of the function J defined in the problem 1.17. Let $x(1) \in X$ be a state. It must be solved:

$$I(1, x(1)) = \min_{(u(1),\ldots,u(N)) \in \mathcal{S}_{1,x(1)}} J_1[u(1), \ldots, u(N)] \tag{1.14}$$

for $x(1)$ running through X.

We can notice, though, that $x(0)$ is the given initial state and that $u(0) \in U_{0,x(0)}$ and determines $x(1)$ based on

$$x(1) = g(0, x(0), u(0)), \tag{1.15}$$

so expressions (1.10) for $u(0)$ and (1.12) for $x(1)$ actually refer to the same set. This is why we can state that

$$\bar{\mathcal{S}}_{u(0)} = \mathcal{S}_{1,x(1)} \tag{1.16}$$

On the other hand, we agree to set

$$I(1, x(1)) = +\infty \qquad \text{when} \qquad \bar{\mathcal{S}}_{u(0)} = \emptyset \tag{1.17}$$

Alternatively, we could also have considered X_1 as the set of feasible states in problem 1.18 and have simply run $x(1)$ through X_1.

From the theorem 1.16, the function defined in the problem 1.17 can be stated as:

$$I^* = \min_{u(0)\in U_{0,x(0)}} \left\{ J_0 \left(u(0), \min_{(u(1),\ldots,u(N))\in\mathcal{S}_{1,x(1)}} \{J_1(u(1),\ldots,u(N))\} \right) \right\}$$

By definition of the problem 1.18 and the relation (1.15),

$$I[1, g(0, x(0), u(0))] = \min_{(u(1),\ldots,u(N))\in\mathcal{S}_{1,x(1)}} \{J_1(u(1),\ldots,u(N))\}$$

In consequence, we are ready now to get to the heart of the matter. When the problem 1.18 is solved $\forall x(1) \in X_1$, then the previous problem 1.17 becomes a minimisation problem in a single variable $u(0)$:

$$I^* = \min_{u(0)\in U_{0,x(0)}} \{J_0 \left(u(0), I[1, g(0, x(0), u(0))]\right)\} \tag{1.18}$$

By applying this result recursively, an optimisation problem in $N + 1$ variables can be solved in $N + 1$ stages, where in each stage a certain number of optimisation problems in a single variable are solved.

Generally, at stage k of the recursivity, for $1 \leq k < N$, by assuming that the criterion function J is decomposable such that it accomplishes the equations of separability (1.8), the minimisation problems of $I(k, x(k))$ can be defined $\forall x(k) \in X_k$ by the relation

$$I(k, x(k)) = \min_{u(k)\in U_{k,x(k)}} \{J_k \left(u(k), I[k+1, g(k, x(k), u(k))]\right)\} \tag{1.19}$$

which is called the *functional equation of dynamic programming*. It allows to solve optimisation problems of $I(k, x(k))$ for all states in X_k from the knowledge of

$I(k+1, x(k+1))$ for all states in X_{k+1}.

It is clear that relations between problems of the same family have just been defined by the functional equation, so step **(b)** of the required embedding is actually accomplished. On the other hand, we need to satisfy step **(a)** as well, i.e., to find one member of the family with a simple enough solution. In fact, this can be easily achieved by considering the last stage of the process, where $k = N$:

$$I(N, x(N)) = \min_{u(N) \in U_{N,x(N)}} \{J_N(u(N))\} \tag{1.20}$$

which is a minimisation over the single variable $u(N)$.

To sum up, starting by solving the equation (1.20) for all $x(N) \in X_N$ and then applying backwards the recursivity in equation (1.19) from $k = N-1$ until $k = 0$, we achieve both **(a)** and **(b)** and we can find the solution of the original problem 1.17.

1.2.3 Restrictions

This backward procedure of dynamic programming cannot be applied, though, to all situations. It is restricted to cases in which:

- The state space $X \subset \mathbb{R}^n$ is finite, with small cardinality.
- The state space $X \subset \mathbb{R}^n$ contains an infinite number of elements, but the dimension n of the state vector is small enough, so that it is possible to approximate the criterion function on X by discretizing states to a finite number of points. The computations of the rest of states can be performed by interpolation methods.

1.2.4 Example

Example 1.19. In practical terms, we will usually consider separable functions which are expressible as the addition of terms that depend only on the state and decision at a single stage. For instance:

$$J = \sum_{k=0}^{N} L(k, x(k), u(k)) \tag{1.21}$$

where $L(i, x(i), u(i))$ is a positive function of i, $x(i)$ and $u(i)$. All conditions assumed for J in this section so far apply here as well.

We first attempt the step **(b)** of the embedding. We define the cost function $I(k,x)$ as the minimum cost that can be obtained by using an admissible decision sequence for the remainder of the process starting from an arbitrary stage k, $0 \leq k \leq N$, and an admissible state $x = x(k) \in X_k$. Hence, we consider the family of the problems about the minimisation of

$$I(k,x) = \min_{u(k),\ldots,u(N)\in\mathcal{S}_{k,x(k)}} \left\{ \sum_{j=k}^{N} L[j,x(j),u(j)] \right\} \tag{1.22}$$

We proceed to find the functional equation using the principle of causality:

$$\begin{aligned}
I(k,x) &= \min_{u(k),\ldots,u(N)\in\mathcal{S}_{k,x(k)}} \left\{ L[k,x,u(k)] + \sum_{j=k+1}^{N} L[j,x(j),u(j)] \right\} = \\
&= \min_{u(k)\in U_{k,x(k)}} \min_{u(k+1),\ldots,u(N)\in\mathcal{S}_{k+1,x(k+1)}} \left\{ L[k,x,u(k)] + \sum_{j=k+1}^{N} L[j,x(j),u(j)] \right\} = \\
&= \min_{u(k)\in U_{k,x(k)}} \{L[k,x,u(k)]\} + \min_{u(k)\in U_{k,x(k)}} \min_{u(k+1),\ldots,u(N)\in\mathcal{S}_{k+1,x(k+1)}} \left\{ \sum_{j=k+1}^{N} L[j,x(j),u(j)] \right\} = \\
&= \min_{u(k)\in U_{k,x(k)}} \{L[k,x,u(k)]\} + \min_{u(k)\in U_{k,x(k)}} \{I[k+1,g(k,x,u(k))]\}
\end{aligned}$$

So, finally:

$$I(k,x) = \min_{u(k)\in U_{k,x(k)}} \{L[k,x,u(k)] + I[k+1,g(k,x,u(k))]\} \tag{1.23}$$

We need to satisfy the step **(a)** of the embedding as well. When $k = N$, we obtain:

$$I(N,x) = \min_{u(N)\in U_{N,x}} \{L(N,x,u(N))\}, \tag{1.24}$$

which is a minimisation over the single variable $u(N)$.
If we look at the recurrence relation (1.23), the minimum cost from state x and stage k is found by minimising the sum of the current single-stage cost $L(k,x,u)$ plus the minimum cost of going to the end of the process from the resulting next stage $g(k,x,u(k))$. Thus, by carrying the minimum cost function backward one stage at a time, we can find the minimum cost at any state and stage by calculating one decision at a time.

1.3 Bellman's principle of optimality

The development of the functional equation of dynamic programming (1.19) can be understood as a way to construct an optimal solution to an initial problem

from partial optimal solutions, which involve just a portion of the total stages of the procedure.
The optimal solution obtained, then, is related to a very important concept defined by Richard Bellman in 1954 which is called *principle of optimality*:

Definition 1.20. The *principle of optimality* states than an optimal policy has the property that, whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision [8].

In other words, all the portions of an optimal solution are, themselves, optimal solutions as well. We can give an easy example to visually understand this statement by imagining an optimal trajectory from certain point A to another point C. Then, the path from any intermediate point B to point C must be the optimal trajectory from B to C too.

However, there is not always an equivalence between the principle of optimality and the functional equation (1.19), because the conditions assumed to validate the functional equation -essentially, separability and monotonocity-, are not sufficient to ensure the validity of the optimality principle. This will be illustrated using the following example [9]:

Example 1.21. Imagine we are looking for the shortest path in the graph $G = (V, E)$ in figure 1.1, in which the length of a path is defined as the product of the lengths of the edges conforming that path. $V = \{1, 2, 3, 4\}$ is the set of vertices and E is the set of edges e_{ij} from vertex i to vertex j with associated cost or length c_{ij}, $1 \leq i, j \leq 4$. L is a real number such that $L > 1$.

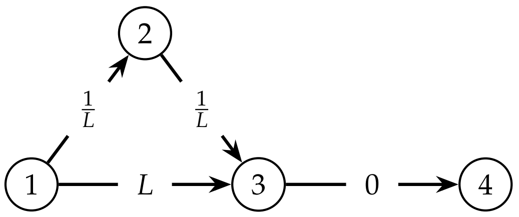

Figure 1.1: Example of an optimal solution of the functional equation which does not satisfy the principle of optimality.

The functional equations

$$J(1) = 1$$
$$J(j) = \min_i c_{ij} \cdot J(i), \qquad e_{ij} \in E, \qquad j = 2, \ldots, N$$

satisfy both separability and monotonicity conditions. However, the optimal path $\{1,3,4\}$ of cost 0 given by those equations contains a partial path $\{1,3\}$ which is not optimal between 1 and 3.

This counterexample makes clear, actually, that functional equations are not simply a mathematical transliteration of the principle of optimality.
To ensure the validity of the principle of optimality we need to introduce a slightly stronger assumption:

Definition 1.22. Our criterion function J depending on variables u_0 and $\vec{u}$ is *decomposable in the strict sense* into two other scalar functions J_0 and J_1 if J is separable such that

$$J(u_0, \vec{u}) = J_0(u_0, J_1(\vec{u}))$$

and J_0 is a *strictly monotone non-decreasing* function of its second argument.

We state now the following theorem:

Theorem 1.23. Let J be a decomposable in the strict sense function such that it satisfies $J(u_0, \vec{u}) = J_0(u_0, J_1(\vec{u}))$. Let $(u_0, \vec{u})$ be an optimal policy which determines a path from the state x_0 to the state x_N, where g represents the system equations such that $g(0, x_0, u_0) = x_1$. Let $\vec{u} = (u_1, \ldots, u_N)$ be a feasible policy which determines a path from x_1 to x_N. Then, $\vec{u}$ is an optimal policy from x_1 to x_N.

Proof. We will proof this result by contradiction. Assume that $\vec{u}$ is not the optimal policy for the trajectory to x_1 to x_N, i.e., $\exists$ a feasible policy $\vec{u}' = (u'_1, \ldots, u'_N)$ from x_1 to x_N such that

$$J_1(\vec{u}') < J_1(\vec{u})$$

First, since $g(0, x_0, u_0) = x_1$, then $(u_0, \vec{u}')$ is a feasible policy from x_0 to x_N.
By the assumption of strict non-decreasing monotonicity, we have:

$$J_1(\vec{u}') < J_1(\vec{u}) \Rightarrow J_0(u_0, J_1(\vec{u}')) < J_0(u_0, J_1(\vec{u})) \Rightarrow J(u_0, \vec{u}') < J(u_0, \vec{u})$$

This means that $(u_0, \vec{u})$ is not an optimal policy from x_0 to x_N, which leads to a contradiction and, in consequence, to the end of this proof. □

Both this theorem and proof can be applied in an analogous way to show that any optimal policy consists of subobtimal policies by assuming that the criterion function is decomposable in the strict sense. Fortunately, additive and multiplicative[2] criterion functions accomplish this property.

[2]Additive criterion functions must consist of non-negative terms and multiplicative functions must contain strictly positive factors.

What is more, it can be shown that if it is just assumed that the criterion function is decomposable, the functional equation of dynamic programming will recover an optimal solution but it may not recover all of them. However, it can also be shown that if the criterion function is assumed to be decomposable in the strict sense, then the resulting functional equation will recover all the optimal solutions. This topic is widely discussed in the 10th chapter of the reference [4].

1.4 The optimum decision policy

Once the recurrence relations for the criterion function are established, the next goal is to determine the optimum sequence of decisions to achieve this optimal solution. Dynamic programming provides an easy way to get this optimal decision policy, because when the recurrence relation (1.19) is performed, it is simply a matter of keeping track the values of the decisions $\hat{u}(k,x)$, for every stage and state, which actually achieve this minimisation. Formally:

$$\hat{u}(k,x) = \arg \min_{u(k) \in U_{k,x(k)}} \{J_k\left(u(k), I[k+1, g(k,x(k),u(k))]\right)\} \tag{1.25}$$

The optimal decision policy allows us to determine the optimum decision sequence from any state and stage. This means that, instead of just solving the problem for the initial stage and state $x(0)$, we have found the optimal solution from any state and stage of the system, which is a huge advantage about using dynamic programming.

In addition, the optimal decision policy also determines the optimal trajectory in state space:

Definition 1.24. The *optimal trajectory in state space* or *optimal trajectory* is the optimum sequence of states, $\hat{x}(i)$, $i = 0, \ldots, N$, where $\hat{x}(0) = c$ is the given initial state.

To explain it in a simple way, the basic dynamic programming method involves two sweeps through the stage variable. In the first sweep, we work backwards computing $I(k,x)$ in terms of $I(k+1,x)$ and getting the optimal decision policy meanwhile. In the second sweep, we work forward, beginning at the initial state c at stage 0. At every stage, we use the optimal decision policy function to recover the next optimum decision in our sequence $\hat{u}(k, \hat{x}(k))$. Then, the next state in the optimal trajectory $\hat{x}(k)$ is found by forward iteration of

$$\hat{x}(k+1) = g[k, \hat{x}(k), \hat{u}(k, \hat{x}(k))], \tag{1.26}$$

until we reach the final state at the final stage N.
The procedure can also be applied when the initial state is not specified but must be determined by minimising a function over all possible initial states: the initial state is first selected by performing the minimisation, and the optimum decision sequence and optimal trajectory are recovered by usual.

Chapter 2

Dynamic programming computational procedure

The aim of this chapter is to discuss, based on the theoretical results previously shown, the basic computational procedure of dynamic programming. Most concepts are based on arguments from [1] and will be introduced through an additive separable cost function to simplify the general reasoning.

We start by reviewing the basic problem formulation used in this chapter. Then, we introduce the brute-force enumeration method to compare it to dynamic programming and show that the latter requires much less computing time. Later on, we explain the main elements of the computational procedure: constraints and quantization of states, the starting procedure and two sweeps through the stage variable to recover the complete solution of the problem.
Afterwards, the main properties of dynamic programming methods are listed so as to show that its implementation is essentially beneficial. Furthermore, main computational issues are discussed to evaluate the limits and implications of the dynamic programming procedure. Finally, an analogous forward procedure is introduced to broaden the range of problems which can be approached by our computational method.

2.1 Problem reformulation

We have already checked the way dynamic programming approaches the optimisation of multistage decision processes. The basic general problem formulation applied in this method is reviewed here so as to be subsequently used by computational methods.

On the one hand, stage variable usually takes discrete values $k = 0, \ldots, N$. In addition, we might even consider continuous problems by quantizing the stage variable into uniform increments. For instance, we could define a continuous stage variable t over a range $t_o \le t \le t_f$ with increment size Δt, so stages would be indexed in the following way:

$$t = t_0 + j \cdot \Delta t, \qquad j = 0, \ldots, N$$

On the other hand, it will be assumed in this chapter that for any given problem a set of n independent state variables $x_1, \ldots, x_n$ can be specified. Besides, there are generally m decision variables $u_1, \ldots, u_m$. The values of both types of variables are usually restricted by *constraints*:

$$x(k) \in X_k \subset X \subset \mathbb{R}^n, \qquad u(k, x(k)) \in U_{k,x(k)} \subset U \subset \mathbb{R}^m \tag{2.1}$$

The system equations are:

$$x(k+1) = g(k, x(k), u(k)), \qquad k = 0, \ldots, N-1, \tag{2.2}$$

The computational procedure can be applied to the minimisation (or maximisation) of any criterion function satisfying the conditions of the general problem 1.17. However, this second chapter is not meant to be as formal as the first one, because the presentation of the main features of the computational procedure is likely best suited for a not so formal treatment. This is why we will restrict to additive separable cost functions:

$$J = \sum_{i=0}^{N} L[i, x(i), u(i)], \tag{2.3}$$

where $L[i, x(i), u(i)]$ is a positive function of i, $x(i)$ and $u(i)$.
We can already specify which problems will be considered all along this chapter:

Problem 2.1. Given:

(i) An N-stage decision process described by system equations (2.2).

(ii) Constraints on the states and decisions (2.1).

(iii) An initial state $x(0) = c$.

Find the decision sequence $u(0), \ldots, u(N)$ which minimises (2.3) while satisfying the constraints.

2.2 Dynamic programming vs brute-force enumeration

2.2.1 Optimisation via brute-force enumeration

Before solving problem 2.1 by using dynamic programming, it is quite useful to briefly look at a a brute-force enumeration procedure in order to discuss its limitations and show why it is beneficial to apply a dynamic programming method instead.

In a brute-force enumeration process, the set of admissible decisions is quantized to a finite number of values:

$$U = \{u^{(1)}, \dots, u^{(M)}\},$$

where both the number M of admissible decisions and their values $u^{(q)}$, $q = 1, \dots, M$, can vary with the state and the stage.
Steps of the enumeration procedure are based on a classical decision tree model:

- At the given state x(0) and stage $k = 0$, every admissible decision $u \in U$ is applied. For each of these decisions, the next state is computed from:

  $$x(1) = g[0, x(0), u]$$

  If $x(1) \in X_1$, then the associated cost is:

  $$\Omega[1, x(1)] = L[0, x(0), u]$$

  If $x(1) \notin X_1$, no further consideration is given.

- In general, when a set of states $x(k) \in X_k$ ($k = 1, \dots, N-1$) is defined, a new set of states $x(k+1) \in X_{k+1}$ is defined by applying all of $u \in U$ at all of the $x(k) \in X_k$, computing the system equations and evaluating the following cost:

  $$\Omega[k+1, x(k+1)] = L[k, x(k), u] + \Omega[k, x(k)]$$

- The process continues until $k = N$ is reached.

The minimum cost is evaluated computing $\Omega[x(N), N]$ for the admissible states $x(N) \in X_N$ and choosing the minimum value. We can recover from that, backwards, the optimal decision sequence and the optimal decision cost. Unfortunately, assuming constraints are not violated, the number of trajectories is given by M^N, so the total number of computed evaluations -scoring the costs, comparing them to find the minimum and tracing back along the decision tree to find the optimal decision sequence and optimal trajectory- follows $O(M^N)$, which is not computationally feasible for moderately large values of M and N.

2.2.2 Computing time comparison

We will check now whether the dynamic programming iterative function improves the brute-force enumeration procedure.
We assume the quantization of X to some finite number of values:

$$X = \{x^{(1)}, \dots, x^{(\Theta)}\} \tag{2.4}$$

where both Θ and $x(j)$, for $j = 1, \dots, \Theta$, can vary with the stage k. We also assume the quantization of $U = \{u^{(1)}, \dots, u^{(M)}\}$.
We obtained in the example 1.19 the following iterative functional equation:

$$I(k,x) = \min_{u \in U} \{L[k,x,u] + I[k+1, g(k,x,u)]\}, \qquad k = 0, \dots, N-1 \tag{2.5}$$

starting by

$$I(N,x) = \min_{u \in U} \{L(N,x,u)\}, \tag{2.6}$$

If we work backwards from (2.6), given a particular state x at stage k, we only need to carry the minimum cost function (2.5) and the optimum decision associated to it. Namely, for each $u \in U$, we evaluate:

1. The cost of the present stage $L(k,x,u)$.
2. Next state $g(k,x,u)$.
3. The minimum cost at the next state $I(k+1, g(k,x,u))$.

Finally, we compare $L[k,x,u] + I[k+1, g(k,x,u)]\ \forall u \in U$.

These calculations are computed once per quantized decision for each state and for each stage. In this case, if the number of quantized decisions, quantized states and stages are M, Θ and N, respectively, the total number of evaluations for this dynamic programming method follows $O(M \cdot \Theta \cdot N)$. This linear growth is much less than the exponential growth needed in the direct enumeration procedure, which was ruled by $O(M^N)$.

Hence, in all but the very simplest examples, dynamic programming requires orders of magnitude less computing time than direct enumeration, so its implementation is quite preferable.

2.3 Details of the computational procedure

In this section, some different aspects affecting the implementation of dynamic programming mehods are considered.

2.3.1 Constraints and quantization

Dynamic programming -like direct enumeration- is very flexible when handling constraints, which actually reduces the computational effort. The set of admissible states X is defined based on inequality constraints which restrict the range of the state variables by

$$\beta_i^-(k) \leq x_i(k) \leq \beta_i^+(k), \qquad i = 1, \dots, n, \qquad k = 0, \dots, N \tag{2.7}$$

Analogously, the set of admissible decisions U uses inequality constraints that restrict the range of the decision variables by

$$\alpha_j^-(k,x) \leq u_j(k,x) \leq \alpha_j^+(k,x), \qquad j = 1, \dots, m, \qquad k = 0, \dots, N \tag{2.8}$$

On the other hand, we specified in last chapter that dynamic programming requires a finite number of admissible states. If necessary, state variables x_i could be quantized in the range described by (2.7) assuming non-uniform increments $\Delta x_i(k, j_i)$:

$$x_i(k, j_i) = \beta_i^-(k) + \sum_{j=0}^{j_i} \Delta x_i(k, j), \qquad j_i = 0, \dots, \Theta_i, \qquad k = 0, \dots, N \tag{2.9}$$

such that

$$\sum_{j=0}^{\Theta_i} \Delta x_i(k, j) = \beta_i^+(k) - \beta_i^-(k), \qquad \forall\, k = 0, \dots, N \tag{2.10}$$

Anyway, for simplicity, for the remainder of this chapter α_j^-, α_j^+, β_i^- and β_i^+ will be assumed constant. In addition, we will also assume that $X = \{x^{(1)}, \dots, x^{(\Theta)}\}$ and $U = \{u^{(1)}, \dots, u^{(M)}\}$.

2.3.2 Starting procedure

Since the iterative functional equation (2.5) expresses the minimum cost at stage k depending on the minimum cost at stage $k+1$, we need to specify a set of boundary conditions at the final stage N. So, the quantity to be determined is $I(N, x)$ for every quantized state $x \in X$ from equation (2.6).
Besides, if no decision was made at the final stage $k = N$, then we could simply write:

$$I(N, x) = L(N, x) \tag{2.11}$$

2.3.3 First sweep: calculation of optimal decisions

Once we have the values of the boundary conditions $I(N,x)\ \forall x \in X$, the functional equation (2.5) is used at first to compute the minimum cost $I(N-1,x)$ and the optimal decision $\hat{u}(N-1,x)$ at state x and stage $N-1$. Then, the same equation is iteratively applied backwards through all stages until $k=0$.

The general iterative procedure assumes that, at stage k, $I(k+1,x)$ is known $\forall\ x \in X$. Given one of these quantized states $x \in X$, at stage k each of admissible decisions $u^{(q)}$ is applied and then the cost at current stage is:

$$L^{(q)} = L[k,x,u^{(q)}], \qquad q = 1,\dots,M \tag{2.12}$$

On the other hand, next state at stage $k+1$ is:

$$x^{(q)}(k+1) = g[k,x,u^{(q)}], \qquad q = 1,\dots,M \tag{2.13}$$

If $x^{(q)}(k+1)$ lies outside of the range of admissible states X, the decision $u^{(q)}$ is rejected. If system equations lead to obtain a state $x^{(q)}(k+1)$ within the range of allowable states but not being a quantized value, we need to use some interpolation procedure to evaluate the criterion function at this point by expressing it in terms of the actual values at quantized states[1].

Reasoning from equation (2.5), we need to compute:

$$I[k,x] = \min_{u^{(q)} \in U} \left\{ L[k,x,u^{(q)}] + I[k+1,x^{(q)}(k+1)] \right\} \tag{2.14}$$

The optimal decision at this state and stage, $\hat{u}(k,x)$, is the value $u^{(q)}$ for which the minimum in equation (2.14) is actually taken on.

This procedure is repeated again for all $x \in X$. Later on, the whole process is developed again at every stage, backwards, until $\hat{u}(0,x)$ and $I(0,x)$ have been computed.

2.3.4 Second sweep: recovery of an optimal trajectory

We have seen that dynamic programming consists in solving a whole family of related problems. This is the reason why a dynamic programming solution does not only give a simple optimal value. Actually, a complete solution specifies

[1]For practical reasons only the simplest interpolation techniques are extensively used in dynamic programming, basically low-order polynomial approximations over a small region. These methods are described in detail in reference [1].

$\hat{u}(k,x)$ and $I(k,x)$ for all quantized states $x \in X$ and for all stages $k = 0,\ldots,N$ instead. However, we were originally asked about to determine an optimal sequence of decisions from a given initial state $x(0)$.

Once the optimal decisions and minimum costs have already been computed and also stored for every state and stage, it is really easy to explicitly find the optimal sequence of decisions associated to the considered initial state. Starting from the given $x(0)$, first decision is evaluated as $\hat{u}(0) = \hat{u}(0, x(0))$. Afterwards, next state in the optimal trajectory is found by $\hat{x}(1) = g[0, x(0), \hat{u}(0)]$. If $\hat{x}(1)$ is quantized, then the next decision is directly $\hat{u}(1) = \hat{u}(1, \hat{x}(1))$. Otherwise, we need to interpolate from values $\hat{u}(1,x)$ for quantized states in X. The recovery procedure continues forward through all stages until $\hat{u}(N)$ and $\hat{x}(N)$ have been obtained.

2.4 Properties of the computational procedure

At this point, it is clear that dynamic programming offers some interesting properties to be applied in optimisation problems of multistage decision processes. In this section, main properties of the dynamic programming computational procedure will be listed so as to outline its desirability in the application of these problems.

Some of the main dynamic programming features are actually shared with the brute-force enumeration method:

- The main assumptions required for system equations g and the single-stage cost functions L are a rule for determining values of these functions at quantized values of the stage, state and decision variables and a procedure to interpolate them between quantized values.
- It easily handles constraints.
- It always determines an absolute minimum (or absolute maximum), since all states are considered for every stage and all decisions are considered for every state and stage.
- It is a quite simple procedure, since it only steps forward the system equation g, looks up and/or interpolates the minimum cost function at the next stage and compares scalar quantities.

In addition, there are some other properties which involve a clear improvement respect to the brute-force enumeration method:

- Running time requirements are drastically reduced.
- Solutions are obtained for an entire family of problems, not for just a single problem. This property was called by Bellman as *invariant imbedding*.
- A *feedback control* or *decision policy* is obtained, which means that the optimal decision is simplified as a function of both stage and state. So, if there are deviations from the original optimal trajectory, a truly optimal decision can be found for the remaining stages.

Nevertheless, as in any other computational method, some aspects must be carefully taken into account when implementing dynamic programming. These possible drawbacks will be presented in the next section.

2.5 Computational requirements

In this section, we will analyse the computational requirements involved in the dynamic programming method we have been discussing all along this chapter.

2.5.1 The curse of dimensionality

Despite dynamic programming presents beneficial properties, computational requirements of the procedure grow rapidly with the number of state and decision variables. This feature was called by Bellman as *curse of dimensionality*. Actually, dimensionality problems affect many different computational methods, but they are specially significant for discrete optimisation problems, which are the main target for dynamic programming.
Nowadays, computers allow the resolution of problems which were not affordable several decades ago. Despite these advances, the curse of dimensionality is still an obstacle. Yet, dynamic programming is still more efficient than brute-force enumeration in terms of execution time. In addition, several types of problems, like many graph and network problems, are not essentially affected by this size problem.
The curse of dimensionality is often faced using heuristic methods, which try to find good approximations to the actual exact solutions of the dynamic programming functional equation. Unfortunately, the absence of exact solutions makes often difficult to determine if these results are good enough.
The best idea is probably analysing problems on a case by case basis in order to find a functional equation which eludes dimensionality issues as much as possible. In fact, a general approach seems to have no chance against the curse of dimensionality.

2.5.2 Memory

Despite dynamic programming highly advantages brute-force enumeration in terms of running time, it involves much more important memory issues which must be taken under consideration.
To begin with, the first barrier is the number of locations in the high-speed memory which must be available during computations. It requires that enough data should be stored to specify $I(k,x)\ \forall x \in X$ at every stage k, generally by keeping one value of $I(k,x)$ per every quantized state $x \in X$ and by using a simple interpolation formula.
Generally, the required number of locations is:

$$N_n = \prod_{i=1}^{n} N_i,$$

where N_i is the number of quantized values of the ith state variable and n the total number of state variables. Since at stage $k-1$ we need both $I(k-1,x)$ and $I(k,x)$, we actually require $2N_n$ locations.

A second storage issue involves saving the results of computation. If there are N_n quantized states at each stage and N stages, the number of values of $\hat{u}(k,x)$ and $I(k,x)$ computed is $N_c = N_n \cdot N$ each. On the other hand, the number of discrete values of u that must be tried at each of these N_c points is:

$$N_d = \prod_{j=1}^{m} M_j,$$

where M_j is the number of quantized values of jth decision variable and m is the number of decision variables.

In fact, the memory requirements can be directly related to the computing time. If, at a given state x and stage k, we call Δt_c the time, in seconds, needed by a computer to work out the quantity

$$\{L(k,x,u) + I(k+1, g(k,x,u))\}$$

and to compare it with other values, the total running time becomes:

$$t_c = N_c \cdot N_d \cdot \Delta t_c, \tag{2.15}$$

which seems a quite reasonable running time for most available computers.

To sum up, we can actually conclude that dynamic programming provides an extremely powerful conceptual point of view for structuring approaches to complex problems and, at the same time, it can also be a practical computational tool for obtaining numerical results in these problems.

2.6 Forward dynamic programming

Despite we have presented a basic procedure under some assumptions, the general method can actually accept modifications in order to be applied to a wider range of problems. For example, dynamic programming can be applied in problems where there is not an explicit stage variable and the end of the process depends on some condition rather than a concrete number of applied steps. It can be used as well in problems where the number of stages becomes infinite. Both situations are widely discussed in reference [1].

In this section, we will focus on problems where the basic recursive equations are not solved backwards, because there exist certain cases where it is preferable to reverse the directions of the sweeps in the stage variable.

Definition 2.2. *Forward dynamic programming* involves procedures in which the basic recursive relation is operated forward in the stage variable.

Analogously, we will call *backward dynamic programming* to the processes we were studying so far.
There are two ways to interpret the concept of forward dynamic programming:

1. We can simply relabel the stage index as $\ell = N - k$, for $k = 0, \dots, N$. Then, the stage variable represents the number of stages to go, rather than the index of the current stage. So, the relations (2.5) and (2.6) become:

$$I(\ell, x) = \min_{u \in U}\{L(N-\ell, x, u) + I(\ell - 1, g(N-\ell, x, u))\}, \qquad \ell = 1, \dots, N \tag{2.16}$$

$$I(0, x) = \min_{u \in U}\{L(N, x, u)\} \tag{2.17}$$

If we momentarily consider the stage-invariant case $L(k, x, u) \equiv L(x, u)$ and $g(k, x, u) \equiv g(x, u)$:

$$I(\ell, x) = \min_{u \in U}\{L(x, u) + I(\ell - 1, g(x, u))\}, \qquad \ell = 1, \dots, N \tag{2.18}$$

$$I(0, x) = \min_{u \in U}\{L(x, u)\} \tag{2.19}$$

These equations lead to a more meaningful interpretation, because the expression in (2.18) states that a process with ℓ stages is given recursively in terms of a process with $(\ell - 1)$ stages. Hence, we start from the initial stage but we allow the length of the process to increase one stage at a time. However, this interpretation does not work for a stage-varying problem.

2. Alternatively, we can define the minimum cost $I'(k,x)$ as the one that can be achieved starting from one admissible initial state and arriving at state x at stage k:

$$I'(k,x) = \min_{u(j)\in U,\ 0\leq j\leq k-1} \left\{ \sum_{j=0}^{k-1} L(j,x(j),u(j)) \right\}, \tag{2.20}$$

where, as usual, $g[k-1,x(k-1),u(k-1)] = x$.
To obtain the iterative equation, we assume the injectivity [6] of the system equation g and we define its inverse $g^{-1} : X \to X$, where $g^{-1}[k,x(k+1),u(k)]$ is the state $x(k)$ from which the state $x(k+1)$ is reached at stage k by applying the decision $u(k)$. So, based on this definition:

$$g\{k,g^{-1}[k,x(k+1),u(k)],u(k)\} = g\{k,x(k),u(k)\} = x(k+1) \tag{2.21}$$

Then, the iterative equation becomes:

$$I'(k,x) = \min_{u\in U}\{L[k-1,g^{-1}(k-1,x,u),u] + I'[k-1,g^{-1}(k-1,x,u)]\} \tag{2.22}$$

The fact that $I'(k,x)$ is determined in terms of $I'(k-1,x)$ implies that calculations proceed forward in the stage variable.

As for the initial condition to begin with, we must indicate the minimum cost function at the initial state. If an initial state $x(0) = c$ is specified:

$$I'(0,x) = \begin{cases} 0, & x = c \\ \infty, & x \neq c \end{cases} \tag{2.23}$$

Alternatively, the initial state may not specified but an initial cost function $\psi(x(0))$ could be given instead.

The corresponding optimum policy function $\hat{u}'(k,x)$ consists of the values of u for which the minimum in (2.22) is reached and specifies what decision should have been applied at stage $k-1$ to reach state x at stage k.

The iteration proceeds forward until $I'(N, x)$ and $\hat{u}(N, x)$ are obtained. From this point, the optimum policy function allows to recover the optimal trajectory going backwards one stage at a time until $x(0)$ is reached.

2.6.1 Advantages and disadvantages of forward dynamic programming

Forward dynamic programming provides several useful properties to be applied in solving some different problems. Its basic advantages are listed below:

- Great flexibility in the terminal cost function and/or terminal constraints. A terminal cost function or constraint can be added after all computations have been performed or it is possible to asses the effects of using different ones without repeating all the calculations.
- The initial conditions can easily be constrained to only one state. So, when we need an optimal trajectory from a concrete given initial state, the use of forward dynamic programming can save computing time.
- There is a straightforward interpolation procedure for problems in which the next states do not occur exactly at quantized values, as it is explained in reference [1].

On the other hand, some disadvantages must be taken into account as well:

- The feedback control property is not retained: if a deviation occurs from the selected optimal trajectory, a new optimal trajectory cannot be easily found.
- We cannot ensure beforehand that a decision $u(k-1)$ is an admissible decision at the state $x(k-1) = g^{-1}(k, x, u(k-1))$.
- It might be difficult to compute g^{-1} if g is a nonlinear, time-varying function.

To sum up, forward dynamic programming suits problems where the initial state is specified, flexibility in choosing the terminal state is desirable, and computation of a new optimal trajectory is feasible if deviations occur from the original trajectory. Much further considerations about this topic are extensively considered in the 14th chapter of reference [4].

Chapter 3

DNA sequence alignment

After having built up the dynamic programming theoretical structure and having introduced its computational procedure, we can now care about its applications. Dynamic programming generally involves an interesting approach to typical operations research problems like allocation problems, scheduling problems, inventory problems or routing problems, as it is explained in reference [2]. In this work, though, we will focus on an up-to-date bioinformatics application consisting in the comparison of DNA sequences to find out gene functions.

According to this purpose, some context is initially provided and main biological concepts are briefly described. All along the next sections, we will try to explain the best methods to compare sequences algorithmically by introducing some different problems whose computer algorithms are attached in the appendix. At first, a basic *change problem* is introduced and recursive and dynamic programming approaches are compared. Afterwards, both the *Manhattan tourist problem* and *the longest path in a direct acyclic graph problem* are useful to define the appropriate framework to consider DNA sequence comparison methods, which are mainly based on two optimisation problems called *the alignment game* and *the longest common subsequence problem*.
Later on, the basic alignment procedure has been increasingly complicated to take into account some relevant biological aspects. In this sense, problems like *global alignment*, *local alignment*, *affine penalties* and *multiple alignment* are also regarded in this chapter. Finally, further questions about the topic are considered as well.

All theoretical contents in this chapter are based on the references [5] and [10]. However, the computer programs containing the algorithms are essentially mine.

3.1 The power of DNA sequence comparison

One remarkable application of dynamic programming is the development of algorithms which can find similarities between different DNA sequences. Actually, DNA sequence comparison algorithms are widely used in biology to discover gene functions. After finding a newly sequenced gene, its function is usually unknown. A common approach to discover it is to compare this new DNA sequence against sequence databases of genes of already known function.

One outstanding example of this procedure took place in 1984, when scientists used a simple computational technique to compare a cancer-causing gene called *v-sis* with all known genes at that time. It surprisingly matched a normal gene responsible for growth and development called *platelet-derived growth factor* (*PDGF*). This similarity allowed scientists to conclude that cancer could be caused by a normal growth gene being activated at the wrong time. Afterwards, many applications of sequence comparison algorithms quickly followed [5].

3.1.1 A short biological introduction

Since this is still a work about mathematics, we will just briefly give the basic biological notions needed to understand the upcoming sections.
DNA is formed over four elements or *nucleotides* called *adenine* (A), *cytosine* (C), *guanine* (G) and *thymine* (T), so in this work DNA sequences will be represented by strings containing elements of a 4-letter alphabet $\{A, C, G, T\}$. DNA can be transformed into RNA simply replacing all ocurrences of *thymine* (T) to *uracil* (U) in a process called *transcription*. So, RNA is analogously represented by strings with elements of a 4-letter alphabet $\{A, C, G, U\}$.
In addition, RNA transcripts are partitioned into non-overlapping substrings of length 3 called *codons*. Each codon converts into one of 20 amino acids via the genetic code[1] in a process called *translation*, except for three stop codons which halt this mechanism. Since protein sequences consist of amino acids, they will be considered in this work as strings over a 20-letter alphabet.

[1]In 1961, Marshall Nirenberg decoded the ribosomal genetic code by associating how amino acids are translated from triplets of nucleotides of RNA [16].

3.2 The change problem

First of all, we will begin by introducing a very simple problem which will be useful to compare the dynamic programming approach to a common recursive solution:

Problem 3.1. *The change problem*: Find the minimum number of coins needed to change a given amount of money.

- **Input**: An amount of cents M and coins from denominations $c = (c_1, c_2, \ldots, c_d)$.
- **Output**: The minimum number of coins $c = (c_1, c_2, \ldots, c_d)$ to change M cents.

First, if minNumCoins_M is the smallest number of coins required to change M cents, we can introduce the following recurrence relation:

$$\text{minNumCoins}_M = \min \begin{cases} \text{minNumCoins}_{M-c_1} + 1 \\ \vdots \\ \text{minNumCoins}_{M-c_d} + 1 \end{cases} \tag{3.1}$$

A simple recursive change algorithm could be designed straightforward based on the previous recurrence. Every time the amount of coins needed to change M cents is computed, it implies d recursive calls to find out the minimum number of coins needed to change $M - \mathrm{c}_i$, with $i = 1, \ldots, d$.

Algorithm 3.2. RECURSIVE_CHANGE(M, c, d)

```
if $M = 0$ then
    return 0
end if
$minNumCoins \leftarrow \infty$
for $i \leftarrow 1$ to $d$ do
    if $M \geq c_i$ then
        numCoins $\leftarrow$ RECURSIVE_CHANGE$(M - c_i, c, d)$
        if numCoins $+ 1 <$ minNumCoins then
            minNumCoins $\leftarrow$ numCoins $+ 1$
        end if
    end if
end for
return minNumCoins
```

Despite being strictly correct, this algorithm becomes unfeasible because it takes too much time ($O(M^d)$) to solve the problem, since it recalculates the optimal solution for a given amount of money repeatedly [5]. The *Python* script to solve the change problem using a recursive method can be found in the appendix A.1.1.

Luckily, dynamic programming can improve the former algorithm with a different approach. Since the optimal solution for M cents relies on optimal solutions for $M - c_i$, with $i = 1, \ldots, d$, we can reverse the execution order and solve the problem for each increasing amount of money from 0 to M. It could seem we are facing a more complex task at first sight, but at every stage the algorithm is analysing d precomputed values instead of recomputing them. This procedure is represented by the following faster algorithm, with running time $O(M \cdot d)$:

Algorithm 3.3. DPCHANGE(M, c, d)

```
minNumCoins(0) $\leftarrow$ 0
**for** $m \leftarrow$ 1 to $M$ **do**
    minNumCoins($m$) $\leftarrow \infty$
    **for** $i \leftarrow$ 1 to $d$ **do**
        **if** $m \geq c_i$ **then**
            **if** minNumCoins($m - c_i$) $+ 1 <$ minNumCoins($m$) **then**
                minNumCoins($m$) $\leftarrow$ minNumCoins($m - c_i$) $+ 1$
            **end if**
        **end if**
    **end for**
**end for**
**return** $minNumCoins(M)$
```

The *Python* code to solve the change problem using dynamic programming can be found in the appendix A.1.2.

Example 3.4. We introduce a concrete example of the change problem so as to compare the running time of the recursive algorithm (appendix A.1.1) and the dynamic programming algorithm (appendix A.1.2). For the following input:

$$M = 49 \qquad \text{and} \qquad c = (11, 9, 7, 3, 1),$$

the output in both cases gives the same optimal solution (5 coins), as expected. The interesting point here is that we used the *Python* library *timeit* to record the running time of each program. Actually, we executed each program several times and we considered the average running time of all executions in each case to give more statistical significance to the results. The dynamic programming algorithm lasted $9,02 \cdot 10^{-5}$ seconds to find the answer, whereas the recursive algorithm lasted $1,86 \cdot 10^{3}$ seconds $\simeq 31$ minutes. This means that, in this case, the dynamic programming approach improves the running time requirement in 8 orders of magnitude.

3.3 The Manhattan tourist problem

Dynamic Programming can be well illustrated by a graph theory problem called *the Manhattan tourist problem*, which will be surprisingly useful as a base to build thereafter the DNA sequence alignment problem.

Imagine we are on a sightseeing tour in Manhattan. We must go from one corner to another, with restricted movements to south or east, and our goal is to visit as many attractions as possible along this path. We can represent Manhattan as a grid-like graph, where each city intersection is a vertex or node and every edge has associated a weight indicating the number of attractions along its path. Vertical and horizontal edges can only be travelled downward or rightward, respectively. We must decide among all possible paths between the northwesternmost point (the *source vertex*) and the southeasternmost point (the *sink vertex*). A path is a continuous concatenation of edges and its length is defined by the sum of the weights of all these edges. Hence, the aim of this problem will be to find the longest path in the graph.

Problem 3.5. *Manhattan Tourist Problem*: find a longest path in a rectangular city grid.

- Input: a weighted $n \times m$ rectangular grid.
- Output: a longest path $s_{n,m}$ from the source $(0,0)$ to the sink (n,m) in the grid.

Brute force approach to this problem -searching among all paths in the graph- cannot be developed in a feasible time for moderately large grids. *Greedy strategies*[2] might be used as well, but short-term optimal decisions will not usually lead to long-term optimal solutions.
Dynamic programming approach involves solving a more general problem. Instead of finding the longest path from source to sink, it finds the longest path $s_{i,j}$ from source to every vertex (i,j) with $0 \leq i \leq n$, $0 \leq j \leq m$. A first glance, it seems

[2]Greedy algorithms choose the local optimal decision at every step so as to try to find a global optimum in a reasonable time scale.

we are facing a more complicated task, because $n \times m$ problems must be solved now. However, we are actually defining an embedding for the original problem in which all the problems of the family are easily related and there are also so simple problems among all of them. The fact that solving the whole family of problems is as easy as solving the original one is the basis of dynamic programming.

In practical terms, we can directly find optimal paths on the first row of the grid, i.e., $s_{0,j}$ for $0 \leq j \leq m$, since movement is restricted to the east, so $s_{0,j}$ is the sum of weights of the first j edges of the grid's first row. Similarly, $s_{i,0}$ is also easy to compute for $0 \leq i \leq n$, because in this case movement is forced to the south.
Once $s_{0,1}$ and $s_{1,0}$ are already computed, $s_{1,1}$ can be then obtained. The vertex $(1,1)$ can only be reached travelling south from $(0,1)$ or east from $(1,0)$. The value of the first path is $s_{0,1}+$ *weight of the edge between* $(0,1)$ *and* $(1,1)$ and the value of the second one is $s_{1,0}+$ *weight of the edge between* $(1,0)$ *and* $(1,1)$. The longest path to $(1,1)$ will be, then, the larger amount between these two quantities.
This method can be easily generalised, because the only way to get to the vertex (i,j) is either by going southward from node $(i-1,j)$ or by going eastward from node $(i,j-1)$. We obtain, in consequence, the following recurrence:

$$s_{i,j} = \max \begin{cases} s_{i-1,j} + \text{ weight of the edge between } (i-1,j) \text{ and } (i,j) \\ s_{i,j-1} + \text{ weight of the edge between } (i,j-1) \text{ and } (i,j) \end{cases} \tag{3.2}$$

This expression can therefore compute every score $s_{i,j}$ in a single sweep of the grid as it is shown in the following algorithm called ManhattanTourist. Let $\overset{\downarrow}{w}$ and $\vec{w}$ be two-dimensional matrices with the weights of the grid's southward and eastward edges, respectively. Namely, $\overset{\downarrow}{w}_{i,j}$ is the weight of the edge between $(i-1,j)$ and (i,j) and $\vec{w}_{i,j}$ is the weight of the edge between $(i,j-1)$ and (i,j). So:

Algorithm 3.6. ManhattanTourist($\overset{\downarrow}{w}, \vec{w}, n, m$)

```
$s_{0,0} \leftarrow 0$
for $i \leftarrow 1$ to $n$ do
    $s_{i,0} \leftarrow s_{i-1,0} + \overset{\downarrow}{w}_{i,0}$
end for
for $j \leftarrow 1$ to $m$ do
    $s_{0,j} \leftarrow s_{0,j-1} + \overset{\rightarrow}{w}_{0,j}$
end for
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow 1$ to $m$ do
        $s_{i,j} \leftarrow \max\{s_{i-1,j} + \overset{\downarrow}{w}_{i,j}, s_{i,j-1} + \overset{\rightarrow}{w}_{i,j}\}$
    end for
end for
return $s_{n,m}$
```

The *Python* programming code to solve the Manhattan tourist problem can be found in the appendix A.2.

The ManhattanTourist algorithm returns the length of the longest path in the grid, but does not give the path itself. The algorithm can be slightly modified by keeping track of the edges selected to reach every vertex -called *backtracing pointers*- and by keeping in memory the value of longest paths for every node as well. Once the value of the longest path is already calculated for every node in the grid, we start from the sink going backwards through backtracing pointers selected in the process until the source is reached.
It should be mentioned that, when the two alternatives to reach a vertex have an equal length, both edges should be kept in memory, because there might be several different paths of optimal length. Proceding in this way, we can get all the optimal paths in the graph. For simplicity, though, the computer algorithm is written so as to give just one solution.

Another point we could have in mind is that a city is not generally a perfect rectangular grid. So, we can transform this grid into an arbitrary graph where edges can go from whatever vertex to whatever node. This generalization of the Manhattan tourist problem is called *longest path in a directed acyclic graph problem.*

3.4 Longest path in a DAG problem

In this section, the method used to find the longest path in a perfect rectangular grid will be applied to more general graphs. Before that, we should mention that one of the requirements to implement a dynamic programming algorithm in a graph is that we must decide beforehand in what order the nodes must be explored. Logically, when a certain node is analysed, all its incoming edges are involved in the calculations. Hence, the initial vertices of these edges -called *predecessors*- must have been already computed. However, graphs containing directed cycles would never satisfy this requirement, because the directed cycle could be indefinitely travelled. So, we introduce some new concepts:

Definition 3.7. A *directed graph* is a graph whose edges have an associated direction. A *directed acyclic graph (DAG)* is a finite directed graph which does not contain any directed cycles.

Definition 3.8. A *topological sorting* or *topological ordering* is a linear ordering of the nodes of a graph such that for all directed edges (v_i, v_j) from vertex v_i to vertex v_j, v_i comes before v_j in the ordering.

The next stated result will be quite useful:

Proposition 3.9. A directed nonempty graph G has a topological ordering if and only if G is a DAG [12], [13].

Proof.

- $\Rightarrow$:
  We will prove the first implication by contradicition.
  We assume that G is not acyclic, so there is a directed cycle C in G with edges $(v_{i0}, v_{i1}), (v_{i1}, v_{i2}), \dots, (v_{ik}, v_{i0})$. Since by hypothesis G has a topological ordering, from edges in the cycle C we have the following inequality about the ordering:
  $$v_{i0} < v_{i1} < v_{i2} < \dots < v_{ik} < v_{i0},$$
  which is impossible. We reached a contradicition and therefore the initial assumption about G not being acyclic is false. We conclude then that G is acyclic instead.

- $\Leftarrow$:
  As for the second implication, we will first show that if G is a DAG, it has at least one source node. We will prove it, again, by contradiction.
  We suppose that every vertex in G contains at least one incoming edge. We start a vertices sequence from an arbitrary node $v_1 \in G$ and we continue picking nodes tracing edges backwards, giving a sequence $v_1, v_2, v_3 \dots$. Since there are only finitely many nodes in G, this process will finally revisit a node v_i. However, we will have a subsequence $v_i, v_{i+1}, v_{i+2}, \dots, v_i$ which is a cycle in G traced in reverse order. But it contradicts the fact that G is a DAG. So our initial assumption was wrong and G must contain at least one source node.

  Now, we use this result to prove that G has a topological ordering. Let $v_1 \in G$ be a source node -which we have proved to exist. We then remove v_1 from G, obtaining $G_1 = G - \{v_1\}$. The graph G_1 is a DAG as well because we cannot add cycles to a graph by removing edges. Since G_1 is a DAG, it contains a source node v_2, as we proved above. We repeat the same process and we remove v_2 from G_1, so $G_2 = G - \{v_1, v_2\}$, which is a DAG as well. We continue this method repetadly until every vertex is removed. The resulting numbering of nodes is a **topological order** because an edge (v_i, v_j) must be deleted before the removal of vertex v_j and, therefore, by the same rule the vertex v_i must be deleted before the vertex v_j. In consequence, $i < j$ holds for every edge (v_i, v_j), which is the definition of topological ordering.

□

Now, we will give a pseudocode to be able to define a topological ordering for a graph, based on the proof of the proposition above:

Algorithm 3.10. TopologicalOrdering(G)

```
i = 1
while $G \neq \varnothing$ do
    Let $x$ = vertex in $G$ with indegree$(x) = 0$ ▷ indegree(x) = #predecessors of $x$
    Label $x$ with $i$
    Delete all edges $(x, z)$ from $G$
    i++
end while
```

Once the topological sorting is found, it will indicate the order to visit the vertices of the DAG during the algorithm. Naturally, several topological orderings can be valid for the same DAG. Now, we can introduce the following problem:

Problem 3.11. *Longest path in a directed acyclic graph problem*: find a longest path between two nodes in an edge-weighted DAG.

- Input: an edge-weighted DAG with source and sink nodes.
- Output: a longest path from the source to the sink in the DAG.

This problem can also be solved by dynamic programming, which involves finding out the longest path from the source to each vertex of the graph. If s_b refers to the length of the longest path from the source to node b, we can use a simple recurrence:

$$s_a = \max_{\text{all predecessors } b \text{ of node } a} \{s_b + \text{weight of edge from } b \text{ to } a\} \tag{3.3}$$

This expression actually works as a functional equation, relating the family of problems which will be solved. A pseudocode for the original problem can be described, in which nodes are updated following certain topological sorting:

Algorithm 3.12. LongestPath(Graph,source,sink)

```
for each node $a$ in Graph do
    $s_a \leftarrow -\infty$
end for
```

```
$s_{\text{source}} \leftarrow 0$
TopologicalOrdering(Graph)
**for** each node $a$ **do**                    ▷ from source to sink in the topological order
    $s_a \leftarrow \max_{\text{all predecessors } b \text{ of node } a}\{s_b + \text{weight of edge from } b \text{ to } a\}$
**end for**
**return** $s_{\text{sink}}$
```

Since this dynamic programming method uses every edge just once to work out the value of the length of its final vertex, the runtime of this algorithm is proportional to the number of edges of the graph. The *Python* programming code to solve the longest path in a DAG problem with a given topological ordering can be found in the appendix A.3.

3.5 The alignment game

DNA sequence comparison problems are efficiently solved by dynamic programming algorithms based on appropriate Manhattan-like grids which model the features of particular biological problems. In this section, we will explain how to compare different DNA sequences by introducing a quite simple procedure called *the alignment game*. First, we want to specify more in detail the meaning of *similarity* or *distance* between two DNA sequences. We introduce a useful concept from information theory [11]:

Definition 3.13. The *Hamming distance* between two strings of equal length is the amount of positions at which their corresponding symbols are different.

The Hamming distance necessarily assumes that the ith symbol of one sequence is aligned against the ith symbol of the other one. However, since DNA replication is subject to errors such as substitutions, insertions and deletions, it is hardly ever known whether DNA sequences are aligned. It will be then helpful to consider the following alternative distance introduced in 1966 by Vladimir Levenshtein:

Definition 3.14. The *edit distance* between two strings is the minimum number of editing operations- i.e., insertions, deletions and substitutions of symbols- required to transform one string into another

Edit distance obviously allows comparisons between strings of different lengths. In bioinformatics, the basic notion of sequence alignment is represented by *the alignment game*, which constructs the alignment of two sequences.

Assume we have one string v with n characters and another string w with m characters. *The alignment of v and w* is determined by a two-row matrix whose first row contains the ordered characters of v and whose second row contains the ordered characters of w. Both rows can be interspersed by spaces (represented by the symbol "-"), so characters are not necessarily adjacently written. It should be stated that spaces in both rows for the same column are not allowed, so the matrix length is at most $n + m$. We can classify the columns of the *alignment matrix* as follows:

- Match: column containing two identical letters.
- Mismatch: column containing two different letters.
- Indel: column with a space symbol on it.
    - Insertion: column where the first row contains a space symbol.
    - Deletion: column where the second row contains a space symbol.

As for the actual procedure, given two sequences the alignment game allows the following decisions at every stage of the process, with their corresponding rewards defined by a *scoring function*:

1. Remove the first symbol from both sequences:
    (a) If they are the same (*match*), we get 1 point.
    (b) If they are different (*mismatch*), we get 0 points.
2. Remove the first symbol of one of the sequences (*insertion* or *deletion*), and we get 0 points.

The removed characters are used to construct the alignment matrix, column by column. These decisions are made until there are no symbols left.

It should be mentioned that the substitution operation is not allowed in the alignment game, so now the edit distance between v and w is regarded as the minimum amount of insertions and deletions needed to transform v into w. The total score is given by the addition of all individual column scores. So, based on the previous scoring function, the maximum score in the alignment game will be given by the maximum number of matches in the alignment matrix. Take note that an initial comparison of both strings might give very few matches at first sight, but the introduction of insertions and deletions allows shifts in the sequences -by introducing spaces- which rearrange their elements so as to construct an alignment matrix in which will likely increase the number of initial apparent matches.

Since we are actually looking for matches between both sequences, we have transformed the alignment game into finding the longest common subsequence between both strings.

3.6 The longest common subsequence problem

Before stating the longest common subsequence (LCS) problem, which is based on the alignment game, we will give two definitions:

Definition 3.15. A *subsequence* of a string v is an ordered not necessarily consecutive sequence of characters from v.

Definition 3.16. A *common subsequence of two strings* $v = v_1 \dots v_n$ and $w = w_1 \dots w_n$ is a sequence of positions in v, $1 \leq i_1 < \cdots < i_k \leq n$, and a sequence of positions in w, $1 \leq j_1 < \cdots < j_k \leq m$, such that $v_{i_t} = w_{j_t}$ for $1 \leq t \leq k$.

It can be seen straightforward that a common subsequence in the alignment of two strings is formed by their set of matches. We can introduce now the LCS problem:

Problem 3.17. *Longest common subsequence problem*: Find the longest common subsequence of two strings.

- **Input**: Two strings, v and w.
- **Output**: The longest common subsequence of v and w.

The key point in this part of the work is to be able to find out a connection between the Manhattan tourist problem and the LCS problem so as to apply an analogous dynamic programming algorithm on it.
The alignment matrix columns can actually be regarded as coordinates in a bidimensional $n \times m$ grid, similar to the grid we used to represent Manhattan. We will basically build a kind of Manhattan grid for the alignment game, called *edit graph* or *alignment graph*, now including diagonal edges as well.
Namely, the symbols of the first sequence v are used to index the rows of the grid and the symbols of the second sequence w are used to index its columns. Besides, every column of the alignment matrix is converted to an edge translated into the new grid as follows:

- Matches correspond to southeastward diagonal edges which imply equal symbols on their pertinent rows and columns in the grid and have weight or score 1.

- Mismatches correspond to southeastward diagonal edges with different symbols on their pertinent rows and columns in the grid and have weight 0.
- Insertions correspond to eastward horizontal edges and have weight 0.
- Deletions correspond to southward vertical edges and have weight 0.

Paths in this edit graph correspond to an alignment of both sequences, since edges in the graph are actually columns in the alignment matrix. Analogously, optimally travelling through this graph means constructing longest common subsequences, and the length of the longest path will correspond to the maximum score in the alignment game, which is the edit distance between the two sequences as well. The figure 3.1 shows an example of edit graph based on the previous indications.

We apply a dynamic programming method to solve the alignment problem be-

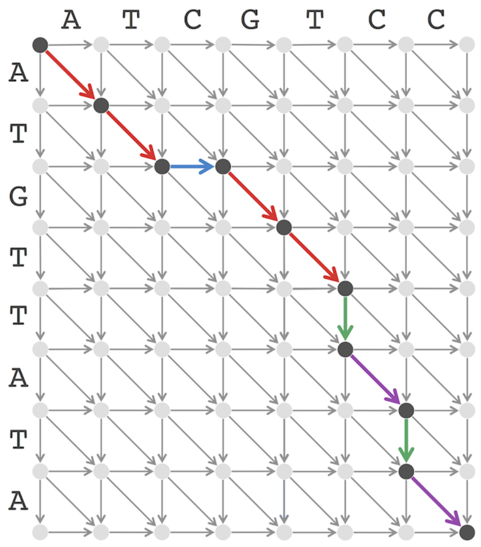

Figure 3.1: Edit graph representing the alignment between two DNA strings. Its longest path is outlined so that matches are shown in red, mismatches in purple, insertions in blue and deletions in green.

tween v and w, so it means that we are in fact solving $n \times m$ problems: the length of a LCS between $v_1 \dots v_i$ and $w_1 \dots w_j$ -i.e., the longest path from the source to the node (i, j)-, represented by $s_{i,j}$, for $0 \leq i \leq n$ and $0 \leq i \leq m$. At every stage of the algorithm, three decisions can be made to reach node (i, j): moving diagonally, corresponding to an edge of weight $\overset{\searrow}{w_{i,j}} = 1$ when both symbols are equal (match) or weight $\overset{\searrow}{w_{i,j}} = 0$ when both symbols are different (mismatch); moving rightward, corresponding to an edge of weight $\overset{\rightarrow}{w_{i,j}} = 0$ (insertion); or moving downward, corresponding to an edge of weight $\overset{\downarrow}{w}_{i,j} = 0$ (deletion).

It is easily seen that $s_{i,0} = s_{0,j} = 0$ for all $1 \leq i \leq n$ and $1 \leq j \leq m$. As for the general recurrence, it is very similar to the one used in the Manhattan tourist problem:

$$s_{i,j} = \max \begin{cases} s_{i-1,j} + \overset{\downarrow}{w}_{i,j} = s_{i-1,j} \\ s_{i,j-1} + \vec{w}_{i,j} = s_{i,j-1} \\ s_{i-1,j-1} + \overset{\searrow}{w}_{i,j} = s_{i-1,j-1} + 1 \quad \text{if} \quad v_i = w_j \\ s_{i-1,j-1} + \overset{\searrow}{w}_{i,j} = s_{i-1,j-1} \quad \text{if} \quad v_i \neq w_j \end{cases} \tag{3.4}$$

Just as in the Manhattan problem, backtracking pointers are preserved so as to reconstruct the concrete LCS for the alignment:

$$\text{backtrack}_{i,j} = \begin{cases} \overset{\downarrow}{w}_{i,j}, \quad \text{if} \quad s_{i,j} = s_{i-1,j} \\ \vec{w}_{i,j}, \quad \text{if} \quad s_{i,j} = s_{i,j-1} \\ \overset{\searrow}{w}_{i,j}, \quad \text{if} \quad s_{i,j} = s_{i-1,j-1} + 1 \text{ and } v_i = w_j \\ \overset{\searrow}{w}_{i,j}, \quad \text{if} \quad s_{i,j} = s_{i-1,j-1} \text{ and } v_i \neq w_j \end{cases} \tag{3.5}$$

We can use this information to move from the sink backwards until arriving to the initial node, and all traced edges represent the solution to the longest common subsequence problem. Obviously, there could be more than one optimal path in the edit graph and, ideally, all of them should be recorded.
The dynamic programming algorithm is shown below. It computes the optimal length of the LCS and, for simplicity, gives only one concrete longest common subsequence, by keeping in a bidimensional vector b an incoming selected edge for every node (either $\vec{w}_{i,j}$, $\overset{\downarrow}{w}_{i,j}$ or $\overset{\searrow}{w}_{i,j}$).

Algorithm 3.18. $LCS(v, w)$

```
  for $i \leftarrow 0$ to $n$ do
      $s_{i,0} \leftarrow 0$
  end for
  for $j \leftarrow 1$ to $m$ do
      $s_{0,j} \leftarrow 0$
  end for
  for $i \leftarrow 1$ to $n$ do
      for $j \leftarrow 1$ to $m$ do
```

```
        $s_{i,j} = \max \begin{cases} s_{i-1,j} \\ s_{i,j-1} \\ s_{i-1,j-1} + 1 & \text{if} \quad v_i = w_j \\ s_{i-1,j-1} & \text{if} \quad v_i \neq w_j \end{cases}$
        $b_{i,j} = \begin{cases} \overset{\downarrow}{w}_{i,j}, & \text{if} \quad s_{i,j} = s_{i-1,j} \\ \overset{\rightarrow}{w}_{i,j}, & \text{if} \quad s_{i,j} = s_{i,j-1} \\ \overset{\searrow}{w}_{i,j}, & \text{if} \quad s_{i,j} = s_{i-1,j-1} + 1 \text{ and } v_i = w_j \\ \overset{\searrow}{w}_{i,j}, & \text{if} \quad s_{i,j} = s_{i-1,j-1} \text{ and } v_i \neq w_j \end{cases}$
    end for
end for
return $(s_{n,m}, b)$
```

The *Python* programming code to solve the LCS problem can be found in the appendix A.4.

3.7 The global alignment problem

During the next sections, we will introduce several modifications to the alignment game and the longest common subsequence problem so as to give more biological relevance to the problems. According to this, the weights of our graphs will usually reflect the cost of mutations from one DNA sequence to another.

Unfortunately, the longest common subsequence problem is not a realistic approach to the alignment problem. Its restrictive scoring function awards 1 point for matches but it does not penalize indels, so there is no limit to introduce insertions or deletions to find a LCS. One alignment with an excessive amount of indels might not be biologically meaningful. Hence, we will deal with a generalization of the LCS where the scoring method will be defined by a *scoring matrix* which does not only reward matched symbols but also penalizes indels and mismatches. We consider the k-letter alphabet $\mathbb{A}$ and we extend it including the space symbol '-', where k is typically 4 or 20 depending on the type of sequence (DNA or proteins) is being analysed. We build a $(k+1) \times (k+1)$ scoring matrix which includes the score of aligning every pair of symbols. An initial approach still gives 1 point for matches, but it also considers a penalty for mismatches by some positive constant μ and another penalty for indels by some positive constant σ. The scoring matrix for this scoring system in DNA sequences comparison $(k = 4)$ would be as follows:

$$\left(\begin{array}{cccccc} & \mathbf{A} & \mathbf{C} & \mathbf{G} & \mathbf{T} & \mathbf{-} \\ \mathbf{A} & +1 & -\mu & -\mu & -\mu & -\sigma \\ \mathbf{C} & -\mu & +1 & -\mu & -\mu & -\sigma \\ \mathbf{G} & -\mu & -\mu & +1 & -\mu & -\sigma \\ \mathbf{T} & -\mu & -\mu & -\mu & +1 & -\sigma \\ \mathbf{-} & -\sigma & -\sigma & -\sigma & -\sigma & \end{array}\right)$$

We remark that the LCS problem considered the parameters $\mu = 0$ and $\sigma = 0$. The score of an alignment is then updated now to:

$$\#\text{matches} - \mu \cdot \#\text{mismatches} - \sigma \cdot \#\text{indels} \tag{3.6}$$

Actually, whatever matrices can be defined, mainly when dealing with the amino acid alphabet. Since some mutations can be more likely than others, biologists try to design scoring matrices to reflect the mutation propensity of amino acids into another amino acids[3]. So, the (i, j) coefficient in the scoring matrix indicates how often amino acid i substitutes amino acid j in the alignments of evolutionarily related sequences. It should be stated that this system can imply that optimal biologically meaningful alignments of amino acid sequences could result in very few matches, because it could be preferable to avoid heavy penalties rather than positively scoring because of matches.

Let *Score* be whatever suitable scoring matrix for the current sequences comparison. Taking it as an input, we can solve a generalized version of the alignment problem:

Problem 3.19. *Global Alignment Problem*: Find a highest-scoring alignment of two strings as defined by a scoring matrix.

- **Input**: Two strings v and w and a scoring matrix *Score*.
- **Output**: An alignment of the strings whose alignment score (as defined by *Score*) is maximised over all alignments of v and w.

The values in the scoring matrix are reflected in the weights of the edges of the alignment graph. Recalling that matches and mismatches correspond to diagonal edges, deletions to vertical ones and insertions to horizontal ones, a recurrence $s_{i,j}$

[3]These scoring matrices are usually obtained firstly aligning very similar sequences, which can be even achieved without scoring matrices. As a result, a biological database of related sequences is built and then it can be counted how many times the amino acid i is aligned with the amino acid j. These pieces of information can be used to iteratively compute a scoring matrix to deal with less and less obvious alignments [5].

to compute the length of a longest path from $(0,0)$ to (i,j) is given by:

$$s_{i,j} = \max \begin{cases} s_{i-1,j} + \text{Score}(v_i, -) \\ s_{i,j-1} + \text{Score}(-, w_j) \\ s_{i-1,j-1} + \text{Score}(v_i, w_j) \end{cases} \tag{3.7}$$

The *Python* programming code to solve the global alignment problem with given concrete μ and σ penalties can be found in the appendix A.5.

3.8 The local alignment problem

Even though global alignment is a good model for some biological sequences comparison problems -for instance, when the similarity between the strings extends over their entire length-, it can turn out to be a bad model for some others. For example, two genes in different species could be only similar over short conserved regions but very different over all the rest of the DNA[4]. So, global alignment algorithm might not find this small region containing biologically relevant common sequences with a really high score on its own so as not to be highly penalized by indels over the rest of the region.

Example 3.20. Imagine we have the following two DNA sequences:

- GCC$\underline{\text{CAGTTATGTCAG}}$GGGGCACGAGCATGCACA
- GCCGCCGTCGTTTTCAG$\underline{\text{CAGTTATGTCAG}}$

where we can see, underlined in both cases, a common subsequence which is likely to be biologically meaningful. Nevertheless, if we use the algorithm for the global alignment problem attached in the appendix A.5, scoring 1 for matches and penalizing $\mu = 3$ for mismatches and $\sigma = 2$ for indels, we obtain a global maximum score of -16 and the following alignment:

- G C C - C A G T - - T A T G T C A G G G G G C A C G - - A - G - C A T G C
- G C C G C C G T C G T T T - T C A G - - - - C A - G T T A T G T C A - G -

This is an alignment which has a higher score from the perspective of global alignment but therefore hides the biologically relevant alignment, which incurs heavy indel penalties.

[4]For example, *homeobox genes* -which regulate embryonic development-, despite greatly differing between different species, highly conserve an approximately 60 amino acid-long region in each gene, the *homeodomain* [10].

In 1981, Temple Smith and Michael Waterman modified the global alignment dynamic programming algorithm to find small conserved areas while ignoring other areas with little similarity. The *local alignment problem* looks for short similar segments within a bigger sequence and it is formulated as follows:

Problem 3.21. *Local alignment problem*: Find the highest-scoring local alignment between two strings.

- **Input**: Strings v and w and a scoring matrix *Score*.
- **Output**: Substrings[5] of v and w whose global alignment score (defined by *Score*) is maximised among all global alignments of all substrings of v and w.

This problem could be directly solved finding the longest path connecting every pair of nodes -rather than just between the source and the sink- and then selecting the path having maximum weight over all these longest paths.
However, a faster approach is to consider the addition of any extra edges of weight zero from the source to any other node, as well as edges from every node to the sink, with the purpose of tracing the common substrings without having heavy penalties in the remaining path[6]. This will result in a DAG suitable for solving the Local Alignment Problem, because thanks to the extra edges it will be enough to find the longest path from source to sink, rather than between every pair of vertices.
The number of edges in our graph still remains quadratic ($O(|v| \cdot |w|)$), so the problem will still be solvable in a feasible time. Our former dynamic programming recurrence changes for the local alignment by adding a forth possibility to reach every node, which is the weight of an edge from $(0, 0)$ to (i, j), which is zero:

$$s_{i,j} = \max \begin{cases} \text{weight from } (0,0) \text{ to } (i,j) = 0 \\ s_{i-1,j} + \text{Score}(v_i, -) \\ s_{i,j-1} + \text{Score}(-, w_j) \\ s_{i-1,j-1} + \text{Score}(v_i, w_j) \end{cases} \tag{3.8}$$

In the case of the *sink*, where all other nodes are its predecessors:

$$s_{n,m} = \max_{0 \le i \le n, 0 \le j \le m} s_{i,j} \tag{3.9}$$

[5]The difference between a subsequence and a substring is that a substring consists only of consecutive characters from a string, while a subsequence may pick and choose any characters as long as their order is preserved.

[6]Using the context of the Manhattan tourist problem, these freely added edges are usually called *free taxi rides*.

This will make our local alignment algorithm practical and fast. The *Python* programming code to solve the local alignment problem can be found in the appendix A.6.

Example 3.22. We will use the local alignment algorithm in appendix A.6 to solve the example 3.20, which was not appropriately approached with the global alignment method. Keeping the same input as before, we get now as an output a global score of 12 and the following alignment:

- C A G T T A T G T C A G
- C A G T T A T G T C A G

which is the conserved underlined region for both original sequences and, in consequence, the result we really expected from the very beginning.

3.9 Alignment with affine gap penalties problem

In this section, we will take into account the introduction of biologically adequate insertion and deletion penalties in sequence alignment.

Mutations are usually produced by DNA replication errors which insert or delete an entire substring of several nucleotides. In fact, we should consider an insertion or deletion of k nucleotides as a single event rather than k independent mutations. Hence, starting a *gap* -a contiguous sequence of spaces in one of the rows of the alignment matrix- should be penalized much more than extending an already started gap, because it is likely to be a part of the same mutation. Since in our former model the penalty for the insertion or deletion of a k-symbol gap was rather excessive ($k \cdot \sigma$), we will now define a softer approach to gap penalties called *affine penalty*: σ penalty to open a gap and ϵ penalty to extend it, with $\sigma > \epsilon > 0$. Hence, now the penalty for a gap of length k will be $\sigma + (k-1) \cdot \epsilon < k \cdot \sigma$.
We can state now the following problem:

Problem 3.23. *Alignment with affine gap penalties problem*: Construct a highest-scoring global alignment between two strings (with affine gap penalties).

- **Input**: Two strings v and w, a scoring matrix *Score* and numbers σ and ϵ.
- **Output**: A highest-scoring global alignment between these strings, as defined by the scoring matrix *Score* and by the gap opening and extension penalties σ and ϵ.

We could represent gaps in the alignment graph as long vertical (deletion) and horizontal (insertion) edges to account for every possible gap. Namely, we could add edges connecting the (i, j) node to both $(i + k, j)$ and $(i, j + k)$ for all possible gap lengths k. Using this procedure, yet, for two sequences of length n the number of edges would increase from $O(n^2)$ to $O(n^3)$, as well as the running time.

To solve this issue, we can increase the number of nodes instead, by splitting our alignment graph into three levels: for every vertex (i, j) we build three distinct ones: $(i, j)_{lower}$, $(i, j)_{middle}$ and $(i, j)_{upper}$, as it is shown in figure 3.2. The middle level consists of diagonal edges of weight Score(v_i, w_j) representing matches and mismatches; the lower level has vertical edges of weight $-\epsilon$ to represent gap extensions in v; and upper level presents horizontal edges weighting $-\epsilon$ as gap extensions in w. To move between these grid layers we add edges from $(i, j)_{middle}$ to $(i + 1, j)_{lower}$ or $(i, j + 1)_{upper}$ to open a gap at cost $-\sigma$, or from $(i, j)_{lower}$ or $(i, j)_{upper}$ to $(i, j)_{middle}$ to close a gap at free cost.
It can be easily seen that the number of edges remains below $7 \times |v| \times |w|$, so we

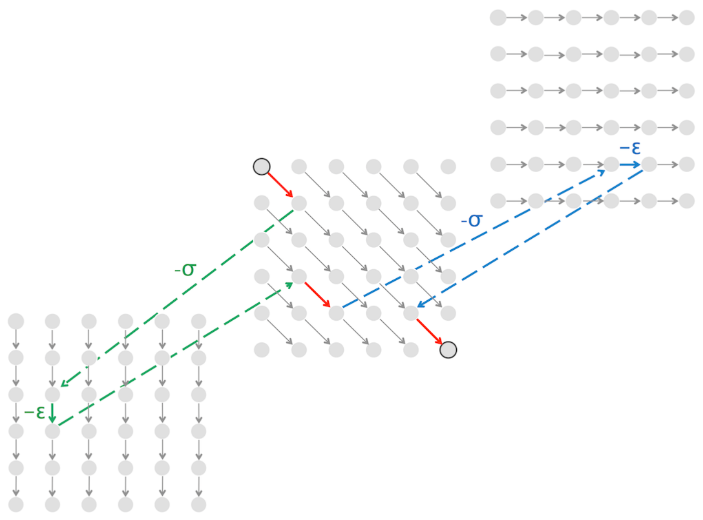

Figure 3.2: The three layers for the alignment graph in the affine penalties problem.

keep an affordable running time. It is still required to find a path from the top left corner to the bottom right corner, either moving along the grid or between their layers.
The algorithm is based on the three following recurrence relations, where lower$_{i,j}$, upper$_{i,j}$ and middle$_{i,j}$ are the lengths of the longest paths from the source node to

$(i,j)_{\text{lower}}$, $(i,j)_{\text{upper}}$ and $(i,j)_{\text{middle}}$, respectively:

$$\text{lower}_{i,j} = \max \begin{cases} \text{lower}_{i-1,j} - \epsilon \\ \text{middle}_{i-1,j} - \sigma \end{cases}$$

$$\text{upper}_{i,j} = \max \begin{cases} \text{upper}_{i,j-1} - \epsilon \\ \text{middle}_{i,j-1} - \sigma \end{cases} \tag{3.10}$$

$$\text{middle}_{i,j} = \max \begin{cases} \text{lower}_{i,j} \\ \text{middle}_{i-1,j-1} + \text{score}(v_i, w_j) \\ \text{upper}_{i,j} \end{cases}$$

All in all, despite having created a much more complicated graph, the alignment algorithm also works with it. The *Python* programming code to solve the alignment with affine gap penalties problem can be found in the appendix A.7.

3.10 Multiple sequence alignment

We have studied *pairwise alignment* so far, which consists of comparing two sequences to find out structural or functional similarities among proteins or DNA sequences. Although it is hard to know whether certain similarities are really meaningful or just by chance, they can become more relevant if they are found in many other sequences as well. Hence, pairwise alignment can be generalized to *multiple alignment*, in which many sequences are simultaneously compared.

Let $v_1, \ldots, v_t$ be t strings of length $n_1, \ldots, n_t$ over an alphabet $\mathcal{A}$, whereas $\mathcal{A}'$ is the extended alphabet $\mathcal{A} \cup \{-\}$, including the space character. A *multiple alignment* of $v_1, \ldots, v_t$ is specified by a $t \times n$ matrix A, where $n \geq \max_{1 \leq i \leq t} n_i$. Its coefficients are elements of $\mathcal{A}'$ and each row i contains the ordered characters of v_i, interspersed with $n - n_i$ spaces. Columns are assumed not to contain only spaces. The score of a multiple alignment is given by the sum of scores of the columns (or weights of edges in the alignment path). A very general scoring function will be used, a t-dimensional matrix *Score* of size $|\mathcal{A}'| \times \ldots \times |\mathcal{A}'|$ describing scores of all possible combinations of t symbols from $\mathcal{A}'$.

Problem 3.24. *Multiple Alignment Problem*: Find the highest-scoring alignment between multiple strings under a given scoring matrix.

- **Input**: A collection of t strings and a t-dimensional matrix *Score*.
- **Output**: A multiple alignment of these strings whose score (as defined by the matrix *Score*) is maximised among all possible alignments of these strings.

This problem is solved by a straightforward application of the dynamic programming alignment algorithm for a t-dimensional alignment graph. For example, applying the same logic as in two dimensions, the alignment graph for three sequences is a grid of cubes, where every node has up to seven incoming edges, as it is shown in figure 3.3.

For three particular sequences v, w and u, $s_{i,j,k}$ is defined as the length of the

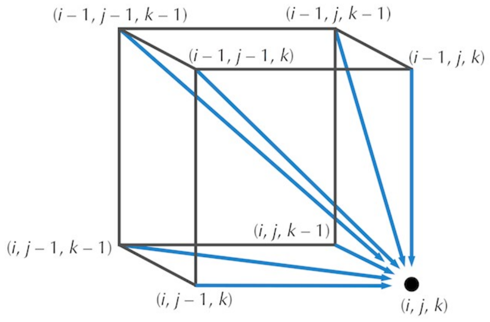

Figure 3.3: Representation of a cell in a 3D aligment graph.

longest path from the source $(0,0,0)$ to node (i,j,k) in the alignment graph:

$$s_{I,j,k} = \max \begin{cases} s_{i-1,j,k} + \text{Score}(v_i, -, -) \\ s_{i,j-1,k} + \text{Score}(-, w_j, -) \\ s_{i,j,k-1} + \text{Score}(-, -, u_k) \\ s_{i-1,j-1,k} + \text{Score}(v_i, w_j, -) \\ s_{i-1,j,k-1} + \text{Score}(v_i, -, u_k) \\ s_{i,j-1,k-1} + \text{Score}(-, w_j, u_k) \\ s_{i-1,j-1,k-1} + \text{Score}(v_i, w_j, u_k) \end{cases} \tag{3.11}$$

The *Python* programming code to solve the multiple alignment problem for three sequences can be found in the appendix A.8.

Describing the multiple alignment problem as a generalization of the more basic pairwise alignment problem implies some problems when dealing with t sequences of length n for larger parameters t and n. On the one hand, it may not be really handy to work with a t-dimensional scoring matrix. On the other hand, the alignment graph would consist of approximately n^t nodes, each one with up to $2^t - 1$ incoming edges[7], involving a runtime of $O(n^t \cdot 2^t)$, so the dynamic programming algorithm would not work in a feasible time for many long sequences.

Biologists usually face this problem using greedy strategies by comparing all pairwise alignments between all sequences and choosing the most similar pair of sequences as a root to build up the multiple alignment. Then, at each step, the string having the maximum score against the current alignment is selected so as to build a new growing multiple alignment. Despite this procedure can be computed in a reasonable time, it does not guarantee optimal solutions, because short-term optimal decisions may not involve long-term optimal decisions as well. Alternatively, there exists a dynamic programming method for the multiple alignment problem involving a score function based on the concept of entropy. Both approaches are explained in detail in the 6th chapter of the reference [5].

3.11 Further questions

In this last chapter, we introduced several techniques to align DNA or protein sequences by using dynamic programming approaches while taking into account some biologically relevant aspects about the matter.

However, in practical terms, we should wonder whether these methods can indeed find subtle similarities between billion-nucleotide long sequences from highly diverged species, because alignment algorithms with quadratic time may be unpractical when dealing with entire genomes.

All these limitations can be treated by developing a new computational framework called Hidden Markov Models (HMM), a machine learning approach which finds some unknown probabilistic parameters in the context of dynamic programming by considering scoring approaches varying at each column of the alignment matrix and edit graphs with probabilistic weights for the edges. This topic is out of the scope of this work, unfortunately, but the readers who are interested in keeping learning about it can find information in references [5] and [10].

[7]The number of vertices in a hypercube of dimension t is 2^t [15].

Conclusions

The variety of topics treated in this work allows a very general perspective about dynamic programming.

First of all, despite it is commonly regarded as a field closer to algorithmics, the truth is that dynamic programming has also been shown to be characterised by strong mathematical foundations.

In addition, one of the most relevant aspects of dynamic programming is its easiness to be implemented by computer programs. As it happens with all computational procedures, it presents different advantages and drawbacks but, weighing up pros and cons, its beneficial properties are worth enough to consider its utilisation.

Another point which should be mentioned is that dynamic programming has a really wide range of application. On the one hand, this flexibility makes impossible to define a general functional equation to solve all possible problems in the same way. Then, every problem must be analysed on a case by case basis to find a suitable functional equation. On the other hand, though, this flexibility can be regarded as positive as well because it means that the procedure can be applied in so many different situations.

As for the application in DNA sequence alignment, it should be outlined how apparently simple procedures can be translated into so high achievements. By introducing, bit by bit, some complications in the initial graph theory, we were able to account for biologically relevant aspects in our statements. In addition, the results obtained through simulations of the written algorithms strongly supported the ideas presented in the last chapter.

Unfortunately, as dynamic programming is a really extensive field, many interesting aspects about it had to remain out of the scope of this work. Some of them are mentioned all along the text with corresponding related references for the readers who are interested in them.

Appendix A

Algorithms

This appendix contains all the computer algorithms refered to the problems presented in the third chapter of this work. Actually, most of them correspond to problems proposed by the massive open online course (MOOC) *Dynamic Programming: Applications In Machine Learning and Genomics* offered by The University of California in the edX website [10]. In addition, these codes were also assessed in that course in order to ensure their validity. *Python* programming language was chosen as one of the options offered by the course and due to its simplicity when dealing with dynamic memory.

A.1 The change problem

Find the minimum number of coins needed to make change.

- **Input**: An integer *money* and an integer array $Coins = (coin_1, ..., coin_d)$.
- **Input Format**: The first line of the input contains the positive integer *money*. The second line contains a comma-delimited list of positive integers *Coins*.
- **Output**: The minimum number of coins with denominations *Coins* that changes *money*.

A.1.1 Recursive method

- **Answer**:

```
import sys

def find_change(money, coins):
    if money == 0:
        return 0
    min_num_coins = float("inf")
    for i in range(0, len(coins)):
        if money >= coins[i]:
            num_coins = find_change(money-coins[i],coins)
            if num_coins + 1 < min_num_coins:
                min_num_coins = num_coins + 1
    return min_num_coins

if __name__ == "__main__":
    money = int(sys.stdin.readline().strip())
    coins = list(map(int, sys.stdin.readline().strip().split(',')))
    print(find_change(money, coins))
```

A.1.2 Dynamic programming method

- **Answer**:

```
import sys

def find_change(money, coins):
    min_num_coins = []
    min_num_coins. append(0)
    for m in range(1, money + 1):
        min_num_coins.append(float("inf"))
        for i in range(0, len(coins)):
            if m >= coins[i]:
                if min_num_coins[m - coins[i]] + 1 < min_num_coins[m]:
                    min_num_coins[m] = min_num_coins[m - coins[i]] + 1
    return min_num_coins[money]

if __name__ == "__main__":
    money = int(sys.stdin.readline().strip())
    coins = list(map(int, sys.stdin.readline().strip().split(',')))
    print(find_change(money, coins))
```

A.2 The Manhattan tourist problem

Find the length of a longest path in a rectangular grid.

- **Input**: Integers n and m, an $n \times (m+1)$ matrix *Down*, and an $(n+1) \times m$ matrix *Right*.
- **Input Format**: The first line of the input contains the integers n and m (separated by a space). The next n lines (each with $m+1$ space-delimited numbers) represent the matrix *Down*. The next line is a "-" symbol. The last $n+1$ lines (each with m space-delimited numbers) represent the matrix *Right*.
- **Output**: The length of a longest path from source $(0,0)$ to sink (n,m) in the $n \times m$ rectangular grid whose edge weights are defined by the matrices *Down* and *Right*.
- **Answer**:

```
import sys

def longest_path(n,m,down,right):
    longest_path = [0] * (n + 1)
    for k in range(n + 1):
        longest_path[k] = [0] * (m + 1)
    longest_path[0][0] = 0
    for k in range (1, n + 1):
        longest_path[k][0] = longest_path[k - 1][0] + down[k - 1][0]
    for k in range (1, m + 1):
        longest_path[0][k] = longest_path[0][k - 1] + right[0][k - 1]
    for i in range (1, n + 1):
        for j in range(1, m + 1):
            path1 = longest_path[i - 1][j] + down[i - 1][j]
            path2 = longest_path[i][j - 1] + right[i][j - 1]
            if path1 >= path2:
                longest_path[i][j] = path1
            else:
                longest_path[i][j] = path2
    return longest_path[n][m]

if __name__ == "__main__":
    n,m = map(int, sys.stdin.readline().strip().split())
    down = [list(map(int, sys.stdin.readline().strip().split()))
            for _ in range(n)]
    sys.stdin.readline()
    right = [list(map(int, sys.stdin.readline().strip().split()))
```

```
                for _ in range(n+1)]

        print(longest_path(n,m,down,right))
```

A.3 The longest path in a DAG problem

Find a longest path between two nodes in an edge-weighted DAG.

- **Input**: An edge-weighted graph, a source node *source*, and a sink node *sink*.
- **Input Format**: The first line of the input contains an integer -with the smallest label- representing *source*. The second line of the input contains an integer -with the largest lebel- representing *sink*. Each of the remaining lines represents an edge in the graph $G(V, E)$ with node-set V and edge-set E, is in the format $u \rightarrow v : w$ denoting an edge from node u to node v with weight w.
- **Output**: The length of the longest path from *source* to *sink*, followed by a longest path.
- **Output Format**: The first line of the output should contain a number representing the length of the longest path from source to sink. The second line of the output should be a longest path in the format *source*$\rightarrow a \rightarrow b \rightarrow c \rightarrow \ldots \rightarrow$ *sink*, where these elements are nodes in G. (If multiple longest paths exist, you may return any one.)
- **Answer**:

```
import sys

def longest_path(source, sink, edges):
    edge = [0] * len(edges)
    for k in range(len(edges)):
        edge[k] = [0] * 3
    for i in range(0, len(edges)):
        temp1 = str(edges[i])
        j = 0
        temp2 = []
        while temp1[j] != "-":
            temp2.append(temp1[j])
```

```
                j += 1
            temp2_as_string = ''.join(temp2)
            edge[i][0] = int(temp2_as_string)
            j += 2
            temp3 = []
            while temp1[j] != ":":
                temp3.append(temp1[j])
                j += 1
            temp3_as_string = ''.join(temp3)
            edge[i][1] = int(temp3_as_string)
            j += 1
            temp4 = []
            while j != len(temp1):
                temp4.append(temp1[j])
                j += 1
            temp4_as_string = ''.join(temp4)
            edge[i][2] = int(temp4_as_string)
        weight = []
        path = []
        weight.append(0)
        path.append(str(source))
        for k in range(source + 1, sink + 1):
            weight.append(float("-inf"))
            path.append(str(source))
        for i in range(source, sink + 1):
            for j in range(0, len(edges)):
                if edge[j][1] == i and weight[edge[j][0] - source] + edge[j][2] >
                weight[i - source]:
                    weight[i - source] = weight[edge[j][0] - source] + edge[j][2]
                    path[i - source] = path[edge[j][0] - source] + "->" + str(i)
        result = str(weight[sink - source]) + "\n" + str(path[sink - source])
        return result

    if __name__ == "__main__":
        source = int(sys.stdin.readline().strip())
        sink = int(sys.stdin.readline().strip())
        edges = []
        i = 0
        while True:
            edges.append(sys.stdin.readline().strip())
            if edges[i] == "":
                edges.remove("")
                break
            i += 1
        print(longest_path(source, sink, edges))
```

A.4 The longest common subsequence problem

Find a longest common subsequence of two strings.

- **Input**: Strings *s* and *t*.
- **Input Format**: The first line of the input contains a string *s*, and the second line of the input contains a string *t*.
- **Output**: A longest common subsequence of *s* and *t*.
- **Output Format**: A longest common subsequence of *s* and *t*. (Note: you can output any of the solutions.)
- **Answer**:

```
import sys

def LCS(s,t):
    n = len(t)
    m = len(s)
    longest_path = [0] * (n + 1)
    backtrack = [" "] * (n + 1)
    for k in range(n + 1):
        longest_path[k] = [0] * (m + 1)
        backtrack[k] = [" "] * (m + 1)
    longest_path[0][0] = 0
    for k in range (1, n + 1):
        longest_path[k][0] = longest_path[k - 1][0]
        backtrack[k][0] = "down"
    for k in range (1, m + 1):
        longest_path[0][k] = longest_path[0][k - 1]
        backtrack[0][k] = "right"
    for i in range (1, n + 1):
        for j in range(1, m + 1):
            path1 = longest_path[i - 1][j]
            path2 = longest_path[i][j - 1]
            if s[j - 1] == t[i - 1]:
                path3 = longest_path[i-1][j-1] + 1
            else:
                path3 = longest_path[i-1][j-1]
            if path1 >= path2 and path1 >= path3:
                longest_path[i][j] = path1
                backtrack[i][j] = "down"
            elif path2 >= path1 and path2 >= path3:
                longest_path[i][j] = path2
                backtrack[i][j] = "right"
            else:
                longest_path[i][j] = path3
                backtrack[i][j] = "diagonal"
    LCS = []
    i = n
    j = m
    while i > 0 and j > 0:
        if backtrack[i][j] == "down":
            i -= 1
        elif backtrack[i][j] == "right":
            j -= 1;
        else:
            if s[j - 1] == t[i - 1]:
                LCS.append(s[j - 1])
            j -= 1;
            i -= 1;
    LCS.reverse()
    LCS_as_string = ''.join(LCS)
    return LCS_as_string

if __name__ == "__main__":
    s = sys.stdin.readline().strip()
    t = sys.stdin.readline().strip()
    print(LCS(s,t))
```

A.5 The global alignment problem

Find a highest-scoring alignment between two strings.

- **Input**: A match score m, a mismatch penalty μ, a gap penalty σ and two DNA strings s and t.
- **Input Format**: The first line contains m, μ and σ separated by spaces. The second and third lines contain DNA strings s and t, respectively.
- **Output**: The maximum alignment score of s and t and an actual alignment achieving this maximum score.
- **Output Format**: The first line of the output should contain the maximum score of an alignment between s and t, and the second and third lines should contain an alignment of s and t, respectively, with gaps placed appropriately, achieving this maximum score.
- **Answer**:

```
import sys

def align(m,mu,sigma,s,t):
    nt = len(t)
    ns = len(s)
    longest_path = [0] * (nt + 1)
    backtrack = [" "] * (nt + 1)
    for k in range(nt + 1):
        longest_path[k] = [0] * (ns + 1)
        backtrack[k] = [" "] * (ns + 1)
    longest_path[0][0] = 0
    for k in range (1, nt + 1):
        longest_path[k][0] = longest_path[k - 1][0] - sigma
        backtrack[k][0] = "down"
    for k in range (1, ns + 1):
        longest_path[0][k] = longest_path[0][k - 1] - sigma
        backtrack[0][k] = "right"
    for i in range (1, nt + 1):
        for j in range(1, ns + 1):
            path1 = longest_path[i - 1][j] - sigma
            path2 = longest_path[i][j - 1] - sigma
            if s[j - 1] == t[i - 1]:
                path3 = longest_path[i-1][j-1] + m
            else:
                path3 = longest_path[i-1][j-1] - mu
            if path1 >= path2 and path1 >= path3:
                longest_path[i][j] = path1
                backtrack[i][j] = "down"
            elif path2 >= path1 and path2 >= path3:
                longest_path[i][j] = path2
                backtrack[i][j] = "right"
            else:
                longest_path[i][j] = path3
                backtrack[i][j] = "diagonal"
    t_aligned = []
    s_aligned = []
    i = nt
    j = ns
    while i > 0 or j > 0:
        if backtrack[i][j] == "down":
            s_aligned.append("-")
            t_aligned.append(t[i-1])
            i -= 1
        elif backtrack[i][j] == "right":
            s_aligned.append(s[j-1])
            t_aligned.append("-")
            j -= 1
        else:
            s_aligned.append(s[j-1])
            t_aligned.append(t[i-1])
            j -= 1
            i -= 1
    s_aligned.reverse()
    t_aligned.reverse()
    s_aligned_as_string = ''.join(s_aligned)
    t_aligned_as_string = ''.join(t_aligned)
    result = str(longest_path[nt][ns]) + "\n" + s_aligned_as_string
    + "\n" + t_aligned_as_string
    return result

if __name__ == "__main__":
    m,mu,sigma = map(int,sys.stdin.readline().strip().split())
    s,t = [sys.stdin.readline().strip() for _ in range(2)]
    print(align(m,mu,sigma,s,t))
```

A.6 The local alignment problem

Find a highest-scoring local alignment between two strings using a scoring matrix.

- **Input**: A match score m, a mismatch penalty μ, a gap penalty σ and two DNA strings s and t.
- **Input Format**: The first line contains m, μ and σ separated by spaces. The second and third lines contain DNA strings s and t, respectively.
- **Output**: The maximum alignment score of a local alignment between s and t and an actual local alignment achieving this maximum score.
- **Output Format**: The first line should contain the score of an optimal local alignment between s and t. The second and third lines should contain an alignment of s and t, respectively, with gaps placed appropriately, achieving this maximum score.
- **Answer**:

```
import sys

def align(m,mu,sigma,s,t):
    nt = len(t)
    ns = len(s)
    longest_path = [0] * (nt + 1)
    backtrack = [" "] * (nt + 1)
    for k in range(nt + 1):
        longest_path[k] = [0] * (ns + 1)
        backtrack[k] = [" "] * (ns + 1)
    longest_path[0][0] = 0
    for k in range (1, nt + 1):
        longest_path[k][0] = 0
        backtrack[k][0] = "free_ride"
    for k in range (1, ns + 1):
        longest_path[0][k] = 0
        backtrack[0][k] = "free_ride"
    for i in range (1, nt + 1):
        for j in range(1, ns + 1):
            path0 = 0
            path1 = longest_path[i - 1][j] - sigma
            path2 = longest_path[i][j - 1] - sigma
            if s[j - 1] == t[i - 1]:
                path3 = longest_path[i-1][j-1] + m
            else:
                path3 = longest_path[i-1][j-1] - mu
            if path0 >= path1 and path0 >= path2 and path0 >= path3:
                longest_path[i][j] = path0
                backtrack[i][j] = "free_ride"
            elif path1 >= path2 and path1 >= path3:
                longest_path[i][j] = path1
                backtrack[i][j] = "down"
            elif path2 >= path3:
                longest_path[i][j] = path2
                backtrack[i][j] = "right"
            else:
                longest_path[i][j] = path3
                backtrack[i][j] = "diagonal"
    max_path = float("-inf")
    for i in range (0, nt + 1):
        for j in range(0, ns + 1):
            if longest_path[i][j] > max_path:
                max_path = longest_path[i][j]
                max_i = i
                max_j = j
    t_aligned = []
    s_aligned = []
    i = max_i
    j = max_j
    longest_path[nt][ns] = max_path
    while backtrack[i][j] != "free_ride" and (i > 0 or j > 0):
        if backtrack[i][j] == "down":
            s_aligned.append("-")
            t_aligned.append(t[i-1])
```

```
                i -= 1
            elif backtrack[i][j] == "right":
                s_aligned.append(s[j-1])
                t_aligned.append("-")
                j -= 1
            else:
                s_aligned.append(s[j-1])
                t_aligned.append(t[i-1])
                j -= 1
                i -= 1
        s_aligned.reverse()
        t_aligned.reverse()
        s_aligned_as_string = ''.join(s_aligned)
        t_aligned_as_string = ''.join(t_aligned)
        result = str(longest_path[nt][ns]) + "\n" + s_aligned_as_string
        + "\n" + t_aligned_as_string
        return result


    if __name__ == "__main__":
        m,mu,sigma = map(int,sys.stdin.readline().strip().split())
        s,t = [sys.stdin.readline().strip() for _ in range(2)]
        print(align(m,mu,sigma,s,t))
```

A.7 The alignment with affine gap penalties problem

Find a highest-scoring global alignment between two strings (with affine gap penalties).

- **Input**: A match score m, a mismatch penalty μ, a gap opening penalty σ, a gap extension penalty ϵ and two DNA strings s and t.
- **Input Format**: The first line contains m, μ, σ and ϵ separated by spaces. Second and third lines contain DNA strings s and t, respectively.
- **Output**: The maximum alignment score of an alignment between s and t (using affine gap penalties) followed by an alignment achieving this maximum score.
- **Output Format**: The first line should contain the maximum score of an alignment between s and t using affine gap penalties. The second and third lines

should contain an alignment of *s* and *t*, respectively, with gaps placed appropriately, achieving this maximum score.

- **Answer**:

```
import sys

def align(m,mu,sigma,eps,s,t):
    nt = len(t)
    ns = len(s)
    lower = [0] * (nt + 1)
    upper = [0] * (nt + 1)
    middle = [0] * (nt + 1)
    backtrack_lower = [" "] * (nt + 1)
    backtrack_upper = [" "] * (nt + 1)
    backtrack_middle = [" "] * (nt + 1)
    for k in range(nt + 1):
        lower[k] = [0] * (ns + 1)
        upper[k] = [0] * (ns + 1)
        middle[k] = [0] * (ns + 1)
        backtrack_lower[k] = [" "] * (ns + 1)
        backtrack_upper[k] = [" "] * (ns + 1)
        backtrack_middle[k] = [" "] * (ns + 1)
    middle[0][0] = 0
    lower[0][0] = float("-inf")
    upper[0][0] = float("-inf")
    for k in range (1, nt + 1):
        upper[k][0] = float("-inf")
        backtrack_upper[k][0] = "midtoup"
    for k in range (1, ns + 1):
        lower[0][k] = float("-inf")
        backtrack_lower[0][k] = "midtolow"
    for k in range (1, nt + 1):
        path1 = lower[k-1][0] - eps
        path2 = middle[k-1][0] - sigma
        if path1 >= path2:
            lower[k][0] = path1
            backtrack_lower[k][0] = "lowtolow"
        else:
            lower[k][0] = path2
            backtrack_lower[k][0] = "midtolow"
        middle[k][0] = lower[k][0]
        backtrack_middle[k][0] = "lowtomid"
    for k in range (1, ns + 1):
        path1 = upper[0][k-1] - eps
        path2 = middle[0][k-1] - sigma
        if path1 >= path2:
            upper[0][k] = path1
            backtrack_upper[0][k] = "uptoup"
        else:
            upper[0][k] = path2
            backtrack_upper[0][k] = "midtoup"
        middle[0][k] = upper[0][k]
        backtrack_middle[0][k] = "uptomid"
    for i in range (1, nt + 1):
        for j in range(1, ns + 1):
            path1 = lower[i-1][j] - eps
            path2 = middle[i-1][j] - sigma
            if path1 >= path2:
                lower[i][j] = path1
                backtrack_lower[i][j] = "lowtolow"
            else:
                lower[i][j] = path2
                backtrack_lower[i][j] = "midtolow"
            path1 = upper[i][j-1] - eps
            path2 = middle[i][j-1] - sigma
            if path1 >= path2:
                upper[i][j] = path1
                backtrack_upper[i][j] = "uptoup"
            else:
                upper[i][j] = path2
                backtrack_upper[i][j] = "midtoup"
            path1 = lower[i][j]
            if s[j - 1] == t[i - 1]:
                path2 = middle[i-1][j-1] + m
            else:
                path2 = middle[i-1][j-1] - mu
            path3 = upper[i][j]
            if path1 >= path2 and path1 >= path3:
                middle[i][j] = path1
                backtrack_middle[i][j] = "lowtomid"
            elif path3 >= path2:
                middle[i][j] = path3
                backtrack_middle[i][j] = "uptomid"
            else:
                middle[i][j] = path2
                backtrack_middle[i][j] = "midtomid"
    t_aligned = []
    s_aligned = []
    i = nt
    j = ns
    floor = "middle"
    while i > 0 or j > 0:
        if floor == "middle":
            if backtrack_middle[i][j] == "lowtomid":
                floor = "lower"
            elif backtrack_middle[i][j] == "midtomid":
                s_aligned.append(s[j-1])
                t_aligned.append(t[i-1])
                j -= 1
                i -= 1
            else:
                floor = "upper"
        elif floor == "lower":
            if backtrack_lower[i][j] == "lowtolow":
                s_aligned.append("-")
                t_aligned.append(t[i-1])
                i -= 1
            else:
                s_aligned.append("-")
                t_aligned.append(t[i-1])
                i -= 1
                floor = "middle"
        else:
            if backtrack_upper[i][j] == "uptoup":
                s_aligned.append(s[j-1])
                t_aligned.append("-")
                j -= 1
            else:
                s_aligned.append(s[j-1])
                t_aligned.append("-")
                j -= 1
                floor = "middle"
    s_aligned.reverse()
    t_aligned.reverse()
    s_aligned_as_string = ''.join(s_aligned)
    t_aligned_as_string = ''.join(t_aligned)
    result = str(middle[nt][ns]) + "\n" + s_aligned_as_string
    + "\n" + t_aligned_as_string
    return result

if __name__ == "__main__":
    m,mu,sigma,eps = map(int,sys.stdin.readline().strip().split())
    s,t = [sys.stdin.readline().strip() for _ in range(2)]
    print(align(m,mu,sigma,eps,s,t))
```

A.8 The multiple alignment problem

Find an alignment of three strings.

- **Input**: Strings r, s and t.
- **Input Format**: The first, second and third lines contain strings r, s and t, respectively.
- **Output**: The maximum score of a multiple alignment of these three strings and an actual multiple alignment of the three strings achieving this maximum using a scoring function in which the score of an alignment column is 1 if all three symbols are identical and 0 otherwise.
- **Output Format**: The first line should contain the maximum score of an alignment between the three input strings. The second, third and fourth lines should contain an alignment of r, s and t, respectively, with gaps placed appropriately, achieving this maximum score.
- **Answer**:

```
import sys
```

```
def align(r,s,t):
    # Despite it would be easier to take advantage of the simple
    # scoring function to simplify the algorithm,
    # we will construct something easily adaptable
    # to more complicated scoring systems
    nt = len(t)
    ns = len(s)
    nr = len(r)
    longest_path = [0] * (nt + 1)
    backtrack = [" "] * (nt + 1)
    for i in range(nt + 1):
        longest_path[i] = [0] * (ns + 1)
        backtrack[i] = [" "] * (ns + 1)
        for j in range(ns + 1):
            longest_path[i][j] = [0] * (nr + 1)
            backtrack[i][j] = [0] * (nr + 1)
    longest_path[0][0][0] = 0
    for k in range (1, nt + 1):
        longest_path[k][0][0] = longest_path[k - 1][0][0]
        backtrack[k][0][0] = "down"
    for k in range (1, ns + 1):
        longest_path[0][k][0] = longest_path[0][k - 1][0]
        backtrack[0][k][0] = "right"
    for k in range (1, nr + 1):
        longest_path[0][0][k] = longest_path[0][0][k - 1]
        backtrack[0][0][k] = "front"
    for i in range(1, nt + 1):
        for j in range(1, ns + 1):
            path1 = longest_path[i - 1][j][0]
            path2 = longest_path[i][j - 1][0]
            path3 = longest_path[i - 1][j - 1][0]
            if path1 >= path2 and path1 >= path3:
                longest_path[i][j][0] = path1
                backtrack[i][j][0] = "down"
            elif path2 >= path3:
                longest_path[i][j][0] = path2
                backtrack[i][j][0] = "right"
            else:
                longest_path[i][j][0] = path3
                backtrack[i][j][0] = "down-right"
    for j in range(1, ns + 1):
        for k in range(1, nr + 1):
            path1 = longest_path[0][j - 1][k]
            path2 = longest_path[0][j][k - 1]
            path3 = longest_path[0][j - 1][k - 1]
            if path1 >= path2 and path1 >= path3:
                longest_path[0][j][k] = path1
                backtrack[0][j][k] = "right"
            elif path2 >= path3:
                longest_path[0][j][k] = path2
                backtrack[0][j][k] = "front"
            else:
                longest_path[0][j][k] = path3
                backtrack[0][j][k] = "right-front"
    for i in range(1, nt + 1):
        for k in range(1, nr + 1):
            path1 = longest_path[i - 1][0][k]
            path2 = longest_path[i][0][k - 1]
            path3 = longest_path[i - 1][0][k - 1]
            if path1 >= path2 and path1 >= path3:
                longest_path[i][0][k] = path1
                backtrack[i][0][k] = "down"
            elif path2 >= path3:
                longest_path[i][0][k] = path2
                backtrack[i][0][k] = "front"
            else:
                longest_path[i][0][k] = path3
                backtrack[i][0][k] = "down-front"
    for i in range (1, nt + 1):
        for j in range(1, ns + 1):
            for k in range(1, nr + 1):
                path1 = longest_path[i - 1][j][k]
                path2 = longest_path[i][j - 1][k]
                path3 = longest_path[i][j][k - 1]
                path4 = longest_path[i - 1][j - 1][k]
                path5 = longest_path[i - 1][j][k - 1]
                path6 = longest_path[i][j - 1][k - 1]
                if r[k - 1] == s[j - 1] and r[k - 1] == t[i - 1]
                and s[j - 1] == t[i - 1]:
                    path7 = longest_path[i - 1][j - 1][k - 1] + 1
                else:
                    path7 = longest_path[i - 1][j - 1][k - 1]
                paths = [path1, path2, path3, path4, path5, path6, path7]
                if max(paths) == path1:
                    longest_path[i][j][k] = path1
                    backtrack[i][j][k] = "down"
                elif max(paths) == path2:
                    longest_path[i][j][k] = path2
                    backtrack[i][j][k] = "right"
                elif max(paths) == path3:
                    longest_path[i][j][k] = path3
                    backtrack[i][j][k] = "front"
                elif max(paths) == path4:
                    longest_path[i][j][k] = path4
                    backtrack[i][j][k] = "down-right"
                elif max(paths) == path5:
                    longest_path[i][j][k] = path5
                    backtrack[i][j][k] = "down-front"
                elif max(paths) == path6:
                    longest_path[i][j][k] = path6
                    backtrack[i][j][k] = "right-front"
                else:
                    longest_path[i][j][k] = path7
                    backtrack[i][j][k] = "diagonal"
    t_aligned = []
    s_aligned = []
    r_aligned = []
    i = nt
    j = ns
    k = nr
    while i > 0 or j > 0 or k > 0:
        if backtrack[i][j][k] == "down":
            r_aligned.append("-")
            s_aligned.append("-")
            t_aligned.append(t[i - 1])
            i -= 1
        elif backtrack[i][j][k] == "right":
            r_aligned.append("-")
            s_aligned.append(s[j - 1])
            t_aligned.append("-")
            j -= 1
        elif backtrack[i][j][k] == "front":
            r_aligned.append(r[k - 1])
            s_aligned.append("-")
            t_aligned.append("-")
            k -= 1
        elif backtrack[i][j][k] == "down-right":
            r_aligned.append("-")
            s_aligned.append(s[j - 1])
            t_aligned.append(t[i - 1])
            i -= 1
            j -= 1
        elif backtrack[i][j][k] == "down-front":
            r_aligned.append(r[k - 1])
            s_aligned.append("-")
            t_aligned.append(t[i - 1])
            i -= 1
            k -= 1
        elif backtrack[i][j][k] == "right-front":
            r_aligned.append(r[k - 1])
            s_aligned.append(s[j - 1])
            t_aligned.append("-")
            j -= 1
            k -= 1
        else:
            r_aligned.append(r[k - 1])
            s_aligned.append(s[j - 1])
            t_aligned.append(t[i - 1])
            j -= 1
            i -= 1
            k -= 1
    r_aligned.reverse()
    s_aligned.reverse()
    t_aligned.reverse()
    r_aligned_as_string = ''.join(r_aligned)
    s_aligned_as_string = ''.join(s_aligned)
```

```
    t_aligned_as_string = ''.join(t_aligned)
    result = str(longest_path[nt][ns][nr]) + "\n" + r_aligned_as_string
    + "\n" + s_aligned_as_string + "\n" + t_aligned_as_string
    return result

if __name__ == "__main__":
    r,s,t = [sys.stdin.readline().strip() for _ in range(3)]
    print(align(r,s,t))
```

Bibliography

[1] R.E. Larson and J.L. Casti, *Principles of Dynamic Programming, Part I: Basic Analytic and Computational Methods*, (Marcel Dekker, Inc., New York, NY, USA, 1978).

[2] R.E. Larson and J.L. Casti, *Principles of Dynamic Programming, Part II: Advanced Theory and Applications*, (Marcel Dekker, Inc., New York City, New York, USA, 1982).

[3] M. Minoux, *Mathematical programming: Theory and alglrithms*, (John Wiley and Sons, Chichester, UK, 1986).

[4] M. Sniedovich, *Dynamic Programming: Foundations and Principles*, (CRC Press, Taylor & Francis Group, Boca Ratón, Florida, USA, 2011).

[5] N.C. Jones and P.A. Pevzner, *An Introduction to Bioinformatics Algorithms*, (The MIT Press, Cambridge, Massachusetts, USA, 2004).

[6] D. Pestana, J.M. Rodríguez, E. Romera, E. Touris, V. Álvarez and A. Portilla, *Curso práctico de Cálculo y Precálculo*, (Editorial Ariel, S.A., Barcelona, Spain, 2007).

[7] J.E. Marsden and A.J. Tromba, *Cálculo Vectorial*, (Addison-Wesley Iberoamericana, S.A., Wilmington, Delaware, USA, 1991).

[8] R. Bellman, *The theory of dynamic programming*, Bull. Amer. Math. Soc., **60**, no. 6, (1954), 503-515.

[9] T.L. Morin, *Monotonicity and the Principle of Optimality*, Journal of Mathematical Analysis and Applications **86**, (1982), 665-674.

[10] P. Compeau and P. Pevzner, *Dynamic Programming: Applications In Machine Learning and Genomics*, (MOOC offered by The University of California (San Diego, California, USA) in edX, 2018-2020). Retrieved from:
https://courses.edx.org/courses/course-v1:UCSanDiegoX+ALGS205x+1T2017/course/

[11] T.K. Carne, *Codes and Cryptography*, (Department of Pure Mathematics and Mathematical Statistics, University of Cambridge, Cambridge, UK). Retrieved from:
https://www.dpmms.cam.ac.uk/ tkc/CodesandCryptography/CodesandCryptography.pdf

[12] S.Y. Cheung, *Directed Acyclic Graph (DAG): topological ordering*, (CS323: Data Structures and Algorithms, Department of Computer Science, Emory University, Atlanta, Georgia, USA). Retrieved from:
http://www.mathcs.emory.edu/ cheung/Courses/323/Syllabus/Graph/DAG.html

[13] K. Schwarz, *Fundamental Graph Algorithms, Part II*, (CS161: Design and Analysis of Algorithms, Department of Computer Science, Stanford University, Stanford, California, USA). Retrieved from:
https://web.stanford.edu/class/archive/cs/cs161/cs161.1138/lectures/02/Small02.pdf

[14] M. Insall, T. Rowland and E.W. Weisstein, *Embedding*, (MathWorld - A Wolfram Web Resource). Retrieved from:
http://mathworld.wolfram.com/Embedding.html

[15] E.W. Weisstein, *Hypercube*, (MathWorld - A Wolfram Web Resource). Retrieved from:
http://mathworld.wolfram.com/Hypercube.html

[16] B.J. Culliton, E.R. Winstead, K. Ruder, C.S. Silver and B. Reinert, *1961: Marshall Nirenberg (1927-) cracks the genetic code*, (Genome News Network, J. Craig Venter Institute, 2000 - 2004). Retrieved from:
http://www.genomenewsnetwork.org/resources/timeline/1961_Nirenberg.php