University of Alberta

Library Release Form

Name of Author: Matthew Paul McNaughton

Title of Thesis: Exact Multiple Sequence Alignment

Degree: Master of Science

Year this Degree Granted: 2003

Matthew Paul McNaughton
Apt. 1802, 10140 - 113 Street
Edmonton, AB
Canada, T5K 2H6

Date: ____________________

One of the symptoms of an approaching nervous breakdown is the belief that one's work is terribly important.

Bertrand Russell

University of Alberta

Exact Multiple Sequence Alignment

by

Matthew Paul McNaughton

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Spring 2003

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Exact Multiple Sequence Alignment** submitted by Matthew Paul McNaughton in partial fulfillment of the requirements for the degree of **Master of Science**.

Duane Szafron
Supervisor

Jonathan Schaeffer
Supervisor

Paul Lu

Warren Gallin
External Examiner

Date:

Abstract

We introduce a novel heuristic for optimal multiple sequence alignment using A* search. We construct the heuristic by first selecting three sequences and computing tables of values representing the best alignments among all combinations of suffixes of these three sequences. Three-way alignments produce better heuristics, but are not used in practice due to their large space requirements. We present a novel technique that uses an octree to reduce space requirements by storing only small portions of the table and recomputing omitted portions as-needed. The octree-supported three-way heuristics result in such a substantial reduction to the A* open list, that they offset the additional space and time requirements for the three-way alignments. The resulting search using the octree heuristic is both faster and uses less memory overall than using A* with traditional pair-wise heuristics.

Acknowledgements

Thanks go to

- My parents for their advice and stories about their experiences gaining Master's degrees.
- My supervisors, Jonathan Schaeffer and Duane Szafron, for their generous financial support and capable collaboration.
- Paul Lu for his discussion and collaboration.
- Warren Gallin for his test data and insights into biology.
- The undergraduate laboratory administrators for looking the other way when I commandeered their facilities.

Contents

List of Tables

List of Figures

List of Symbols

- $\mathbb{R}$ The set of real numbers
- $\mathbb{Z}$ The set of integers
- $\mathbb{Z}^+$ The set of integers strictly greater than zero
- h_{p} A* heuristic composed of pairwise dynamic programming tables
- s,t,u Nodes of a search space
- h^* The ideal heuristic
- $\mathcal{A}$ An alignment
- $\bar{A}$ An optimal alignment
- S_i A sequence
- S A set of sequences
- K The size of a set of sequences
- c A scoring function
- - The artificial gap alignment character

Chapter 1

Introduction

This dissertation has two aims. First, to present background knowledge and an overview of recent work in exact and approximate multiple sequence alignment so that the reader may appreciate the research described in this dissertation. Second, to describe a novel technique based upon the octree data structure the author has developed to enhance exact multiple sequence alignment.

Chapter one motivates and describes the sequence alignment problem to a reader with background neither in biology nor in computing science. Chapter two describes the problem formally and discusses fundamental algorithms for aligning sequences, for the reader somewhat familiar with notations for computer programming. Chapter three surveys a broad range of literature on sequence alignment. Chapter four describes the octree method for storing heuristics developed by the author, followed by results and discussion. Chapter five discusses future research directions.

1.1 Introduction to Biological Sequences

All living creatures, from the smallest bacterium to the largest whale have something in common: they are all made of proteins. Your fingernails and hair contain a hard, fibrous protein called keratin, your muscles have moving proteins called actin and myosin, and a light-sensitive protein called rhodopsin in your eyes gives you the ability to see. Proteins are very large molecules built from chains of smaller molecules called **amino acids**. There are many different kinds of amino acids, but only twenty kinds (Table 1.1) are used to make all the different kinds of proteins found in living things. The basic structure of any protein can be described by its sequence of amino acids. Biologists often write down a protein as just the sequence of letters standing for each amino acid. For example, Figure 1.1 [34] shows a part of a protein called hemoglobin that carries oxygen in your blood, and makes it red. A hemoglobin molecule is actually made with two copies each of this, the alpha type, and one other kind, the beta type, but we only care about each kind of chain by itself.

```
>HAHU hemoglobin alpha chain - human
MVLSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTYFPHFDLSHGSAQVKGHGKKVADALTNAV
AHVDDMPNALSALSDLHAHKLRVDPVNFKLLSHCLLVTLAAHLPAEFTPAVHASLDKFLASVSTVLTSKYR
```

Figure 1.1: The human hemoglobin α-chain protein "HAHU"

How does your body know the amino acids, or letters, that make up that protein? And how does it make sure that they get put together in that order? The answer is DNA. That's short for deoxyribonucleic acid, and it is another very long linear strand of basic molecules. But it is not a chain of amino acids, instead it is a chain of just four kinds of chemicals, or **bases**: adenine, cytosine, guanine, and thymine. They are usually represented with letters too – "A", "C", "G", and"T", respectively. All of the proteins in your body are encoded in a single very long strand of DNA - a human's DNA strand is approximately three billion bases long, and almost all of your cells have a copy. Three billion bases is long enough that you could see it if it were unraveled, if only it weren't so thin. You have trillions of cells, so that is a lot of DNA!

There are many different kinds of information encoded in your DNA, and not all of its purpose is understood, but for our purposes we can make the simplifying assumption that each protein has a devoted section of DNA that describes its structure.

The DNA molecule is long and fragile, and it sits in the centre of the cell, in a special bubble called the **nucleus**, safe from all of the harsh chemicals and activity in the rest of the cell. When a new protein is needed outside the nucleus, special machinery finds the part of the DNA that describes that protein, and copies the sequence in the DNA to the protein's amino acid sequence. DNA uses a very simple code to describe the protein - a group of three bases in the DNA corresponds to each amino acid, in a simple correspondence described in any textbook on genetics or biochemistry.

1.1.1 Protein Structure

We know that there are only twenty kinds of amino acids that make all of the proteins in your body, and it happens that all proteins include a few of nearly each kind of amino acid. For example, even a small protein like HAHU (Figure 1.1) contains 19 of the 20 kinds of amino acids, missing only isoleucine. So if they're all made of the same stuff, what is it about each protein that makes it different from another? The answer is shape. Proteins don't float around in a long straight chain. They fold up into a special shape, and it is that shape, and a few chemical attachments, that determines the way that protein behaves. Think of dogs. There are many kinds of dogs, large and small. Some kinds of dogs are good for pulling sleds, some for fetching dead birds, some for sitting in your lap, and some for barking really loud. They're all made of the same stuff: fur, muscles and slobber, but it's the length of their legs, and the size of their ears, and the sharpness of their teeth that make

	Name	Three-letter code	One-letter-code
1	Alanine	Ala	A
2	Cysteine	Cys	C
3	Aspartic Acid	Asp	D
4	Glutamic Acid	Glu	E
5	Phenylalanine	Phe	F
6	Glycine	Gly	G
7	Histidine	His	H
8	Isoleucine	Ile	I
9	Lysine	Lys	K
10	Leucine	Leu	L
11	Methionine	Met	M
12	Asparagine	Asn	N
13	Proline	Pro	P
14	Glutamine	Gln	Q
15	Arginine	Arg	R
16	Serine	Ser	S
17	Threonine	Thr	T
18	Valine	Val	V
19	Tryptophan	Trp	W
20	Tyrosine	Tyr	Y

Table 1.1: The twenty amino acids found in proteins

them different from each other.

Biologists talk about four kinds of protein structure: **primary structure**, **secondary structure**, **tertiary structure** and **quaternary structure**. Those are just fancy names for “first”, “second”, “third”, and “fourth”. The primary structure of a protein is simply its unique sequence of amino acids. This is the level of structure we are interested in when we do sequence alignment. At a slightly larger scale, the amino acids might twist into little helices like a phone cord, or stay stretched in a long strand like dental floss, to make the secondary structure. At a yet larger scale, the protein can pack into another shape, as if you took the helical (secondary structure) phone cord and mushed it into a ball. That shape is the tertiary structure. Protein chains can then finally join together to make a whole active protein, like hemoglobin with its four parts of two HAHU and two HBHU. This is the quaternary structure.

How the primary structure determines the overall protein folding, and how the cell forces the protein into one configuration among many chemically stable alternatives is not well understood but all we care about is the fact that very often, *proteins with a similar primary sequence have a similar folding.*

As a species evolves, its DNA changes, or mutates. If the DNA changes, then the proteins that it encodes should change along with it. For many reasons, certain parts of a protein are more likely to change than others. One reason might be that the given portion is very important to the function of the protein and that it constitutes a special part of

```
>HACZ hemoglobin alpha chain - chimpanzee
MVLSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTYFPHFDLSHGSAQVKGHGKKVADALTNAV
AHVDDMPNALSALSDLHAHKLRVDPVNFKLLSHCLLVTLAAHLPAEFTPAVHASLDKFLASVSTVLTSKYR
```

Figure 1.2: The chimpanzee hemoglobin α-chain protein "HACZ"

```
>HAGSI  hemoglobin alpha-A chain - bar-headed goose
VLSAADKTNVKGVFSKISGHAEEYGAETLERMFTAYPQTKTYFPHFDLQHGSAQIKAHGKKVVAALVEAVN
HIDDIAGALSKLSDLHAQKLRVDPVNFKFLGHCFLVVVAIHHPSALTAEVHASLDKFLCAVGTVLTAKYR
```

Figure 1.3: A goose hemoglobin protein known as "HAGSI"

the protein that does a special task that can only be performed by one special arrangement of amino acids. Such a part might not change readily. Other portions may not be so important, padding out the protein to fill space in a way that could be accomplished with many alternative sequences of amino acids. So, if we wanted to know how a protein worked, we could take similar proteins from many related organisms, and look for spots where they were all the same.

Let's take our HAHU(human hemoglobin alpha chain) protein for instance. All mammals have hemoglobin, and in particular, all primates. What if we compared the alpha hemoglobin of many primate species? What would we find?

Let's start with the chimpanzee. Its hemoglobin is called HACZ, and the sequence is in Figure 1.2. Oh, no. This is exactly the same as human, down to the last amino acid. But we already knew that chimpanzees are very closely related to humans, didn't we? So let's move farther afield. Figure 1.3 shows one from a goose. That's a little better. That's definitely different from the human. Let's try to line them up in Figure 1.4 and see just how different they are.

The punctuation beneath each line summarizes the degree of similarity of bases in that column and is described in Table 1.2. How do we know how similar amino acids are? We take it as a given that some clever biologists have come up with a matrix of numbers that accurately indicates the likelihood that a mutation from one amino acid to another one is accepted, *i.e.* that the protein still functions properly, and doesn't kill the organism. There are many alternative matrices to choose from that are more appropriate at various evolutionary distances, or are more appropriate for certain organisms' prevailing chemical

```
HAHU     MVLSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTYFPHFDLSHGSAQVKGHGKKVADALTNAV
HAGSI    -VLSAADKTNVKGVFSKISGHAEEYGAETLERMFTAYPQTKTYFPHFDLQHGSAQIKAHGKKVVAALVEAV
          ***.*******..:.*:..** *****:***** ::* **********.*****:*.*****. **.:**
HAHU     AHVDDMPNALSALSDLHAHKLRVDPVNFKLLSHCLLVTLAAHLPAEFTPAVHASLDKFLASVSTVLTSKYR
HAGSI    NHIDDIAGALSKLSDLHAQKLRVDPVNFKFLGHCFLVVVAIHHPSALTAEVHASLDKFLCAVGTVLTAKYR
          *:**:..*** ******:**********:*.**:**.:* * *: :*. *********.:*.****:***
```

Figure 1.4: An alignment of the human (HAHU) and goose (HAGSI) hemoglobin proteins

Symbol	Meaning
*	Amino acids are identical.
:	Amino acids are of very similar types.
.	Amino acids are of somewhat similar types.
(blank)	Amino acids are not similar.

Table 1.2: Column similarity markup for alignments

environment, or scheme for translating triples of DNA bases to amino acid (which varies). Appendix A shows some examples in Tables A.1 and A.2.

What's that '-' character at the beginning of HAGSI? That's a gap. That means, either something has been *inserted* in the first sequence, or *deleted* from the second sequence (an **indel**), so that there is nothing from the other sequence that lines up well with that portion. The likelihood that this happened is indicated by the **gap parameter**, or **gap insertion cost**. There are many schemes for specifying the gap parameter, which we will discuss later.

So this gives a hint as to what parts are important. Let's look at another one. It's big, so I've put it in Appendix A, Figure A.1. It ranges from hemoglobin-like proteins found in yeast all the way to our human alpha hemoglobin! That is as vast an evolutionary distance as you could hope for – from primordial ooze all the way to the Prime Minister of Canada, and yet, the proteins still have identical portions among them. Those parts must be very important!

1.1.2 Evolutionary Trees

Which are more closely related – an ostrich (which makes an excellent meal) and a turkey, or an ostrich and a pigeon? Why don't we line up their proteins and see which ones are more similar? If an ostrich and a turkey share similarities that neither share with a pigeon, that might give us a clue that those species diverged more recently in evolution.

Let's look at hemoglobin again. In Figure 1.5 we can see that all three share many similarities, and each pair of sequences shares a few similarities that the remaining sequence does not. In the majority of columns where one sequence is the odd man out, that sequence is the pigeon. It happens 5 times for turkey, 7 times for ostrich, and 15 times for pigeon. So we could hypothesize that ostriches and turkeys share a common ancestor more recently in the evolutionary tree than either does with the pigeon. That tree might look something like Figure 1.6. Comparing other proteins would strengthen (or weaken) this hypothesis.

How do we construct alignments like the one in Figure A.1 of the Appendix? A very careful and patient person with an amino acid similarity matrix close at hand and a lot of time might be able to line them all up so as to capture those few columns where all the acids are the same or almost all the same. But look at the excerpted portion in Figure 1.7 around column 50 – those few isolated gaps would be very hard to get just right. Sadly,

```
S56102  -VLSAADKNNVKGIFTKIAGHAEEYGAETLERMFITYPPTKTYFPHFDLSHGSAQIKGHGKKVVAALIEAA
HAOS    -VLSGTDKTNVKGIFSKISSHAEEYGAETLERMFITYPQTKTYFPHFDLHHGSAQIKAHGKKVANALIEAV
JC5514  MVLSANDKSNVKAVFGKIGGQAGDLGGEALERLFITYPQTKTYFPHFDLSHGSAQIKGHGKKVAEALVEAA
         ***. **.***.:* **..:* : *.*:***:***** ********** *******.*****. **:**.
S56102  NHIDDIAGTLSKLSDLHAHKLRVDPVNFKLLGQCFLVVVAIHHPAALTPEVHASLDKFLCAVGTVLTAKYR
HAOS    NHIDDISGALSKLSDLHAQKLRVDPVNFKLLGQCFLVVVAIHHPSALTPEVHASLDKFLCAVGAVLTAKYR
JC5514  NHIDDIAGALSKLSDLHAQKLRVDPVNFKLLGHCFLVVVAVHFPSLLTPEVHASLDKFVCAVGTVLTAKYR
       ******:*:*********:*************:*******:*.*: ************:****:*******
```

Figure 1.5: Alignment of turkey (S56102), ostrich (HAOS), and pigeon (JC5514) hemoglobin alpha-A chain proteins. There are 15 columns where the turkey and ostrich proteins have the same amino acid but are different from the pigeon(boxes). The turkey and pigeon differ from the ostrich in 7 columns, and the ostrich and pigeon share a unique similarity in only 5 columns.

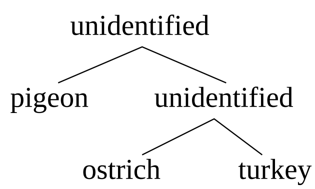


Figure 1.6: A possible evolutionary tree for ostrich, pigeon, and turkey

```
                  50
HAHU    ...HF-DLSH-----G...
HBHU    ...SFGDLSTPDAVMG...
S15768  ...FLKN---SAEVQD...
JT0423  ...QFA-GKDLASIKD...
JC1516  ...GAEKYTA-DDVKK...
A53396  ...M-----------A...
A39601  ...RLGDVSA-ENIPY...
```

Figure 1.7: Fragment of a hemoglobin alignment excerpted from Figure A.1

the primary sequence together with the similarity matrix and gap insertion cost does not give us enough information to correctly align such a group of sequences. For that task we typically require the higher-order(secondary and tertiary) structures. But it is possible to come close.

1.2 Computing Alignments

1.2.1 Two Sequences

Let's try to align some simple sequences by hand using a simple model of amino acid similarity and gap insertion cost. For simplicity, we will allow only 4 different letters to appear: A(alanine), C(cysteine), G(glycine), and T(threonine). Coincidentally, these also happen to be the letters used for the four nucleotides(adenine, cytosine, guanine, thymine) of DNA. The procedures for aligning protein strings and DNA strings are the same. Mathematically, when we're aligning sequences, what the letters actually represent is completely irrelevant!

We'll take $s_1 = \texttt{GATAGTC}$ and $s_2 = \texttt{AGGTGCA}$. For simplicity, we'll count -2 when we get an exact match in a column of an alignment, $+1$ when we have a mismatch, and $+2$ when we insert a gap. We'll try to get the lowest cost possible. So, here is an alignment:

```
-GATAGTC
AGGT-GCA
```

The cost is $+2-2+1-2+2-2+1+1=1$. Is this the best (lowest) possible cost for this pair of sequences and cost scheme? It seems like it might be. But how could we prove it? We could generate all possibilities, but there are 8989 of them so that would take much too long. Observe that we can divide the alignment along a column and vary each half independently:

-GATA	GTC	-GATA	GTC-
AGGT-	GCA	AGGT-	G-CA
$2-2+1-2+2=1$	$-2+1+1=0$	$2-2+1-2+2=1$	$-2+2-2+2=0$

We kept the left half the same, and changed the right half. The change we made to the right half didn't affect the score for the left half. This makes sense, because the scheme we're using to compute the score for the alignment works on a column-by-column basis. So, it should be possible to divide the problem into pieces, solve the pieces, and then combine them. Let's start by just trying to solve the problem in the last column. That should be easy!

-	C	C
A	-	A
2	2	1

So we have three possibilities for the last column. The best alignment must end with one of these columns. We just have to find the best alignment for the first parts of the sequences in each case, and add the scores for those alignments to the scores for the last column. Then, the lowest total score will be the best alignment for the whole sequence.

In the first case we have to find the best score c for an alignment of the first seven characters of the first sequence with the first six characters of the second sequence. Let's call this value $c = p(7,6)$. In the second case we have to find the best score for an alignment of the first six characters of the first sequence with the first seven characters of the second sequence. Call this value $p(6,7)$. In the third case we have to optimally align the first six characters of the first sequence with the first six characters of the the second sequence. Its value is $p(6,6)$. Continuing in this vein, we can give the name $p(7,7)$ to the best score for our original problem of lining up `GATAGTC` with `AGGTGCA`. So if we can just find the best costs for $p(6,6)$, $p(6,7)$, and $p(7,6)$, we'll know the best cost for $p(7,7)$, according to the following formula:

$$p(7,7) = min \begin{pmatrix} 2 & + & p(7,6), \\ 2 & + & p(6,7), \\ 1 & + & p(6,6) \end{pmatrix}$$

We'll worry about how to reconstruct the actual best alignment later. Let's just hope that the number of subproblems doesn't explode as we proceed.

For $p(7,6)$, we want to find the best cost to align `GATAGTC` with `AGGTGC`. Again, we'll start with just the last column, for which there are three possibilities.

$$\boxed{\begin{array}{c}\texttt{-}\\ \texttt{C}\end{array}} \; : 2 + p(7,5) \qquad \boxed{\begin{array}{c}\texttt{C}\\ \texttt{-}\end{array}} \; : 2 + p(6,6) \qquad \boxed{\begin{array}{c}\texttt{C}\\ \texttt{C}\end{array}} \; : -2 + p(6,5)$$

That's interesting. To solve $p(7,6)$ we need to know $p(6,6)$, but we also need to know it for $p(7,7)$. We can save some work and only solve $p(6,6)$ once.

Next we can do $p(6,7)$:

$$p(6,7) = min(\quad \overset{\boxed{\begin{array}{c}-\\ \texttt{A}\end{array}}}{2 + p(6,6)}, \quad \overset{\boxed{\begin{array}{c}\texttt{T}\\ -\end{array}}}{2 + p(5,7)}, \quad \overset{\boxed{\begin{array}{c}\texttt{T}\\ \texttt{A}\end{array}}}{1 + p(5,6)} \quad)$$

We have $p(6,6)$ again. Let's do that next. For $p(6,6)$ we have to align `GATAGT` with `AGGTGC`.

$$p(6,6) = min(\quad \overset{\boxed{\begin{array}{c}-\\ \texttt{C}\end{array}}}{2 + p(6,5)}, \quad \overset{\boxed{\begin{array}{c}\texttt{T}\\ -\end{array}}}{2 + p(5,6)}, \quad \overset{\boxed{\begin{array}{c}\texttt{T}\\ \texttt{C}\end{array}}}{1 + p(5,5)} \quad)$$

We're finding duplicate subproblems all over the place. Let's skip to the first column and work back the other way. $p(0,0)$ is the problem of lining up the first 0 characters of the first sequence with the first 0 characters of the second sequence. That's easy to solve. The alignment is completely empty, so it seems reasonable to define $p(0,0) = 0$. $p(1,0)$ and $p(0,1)$ are both easy to solve, too. $p(1,0) = 2$ because it puts a `G` above a `-`. $p(0,1)$ is also 2 because it puts a `-` above a `A`. $p(1,1)$ is

$$p(1,1) = min \left(\begin{array}{ccc} \boxed{\begin{array}{c}\texttt{-G}\\ \texttt{A-}\end{array}} & \boxed{\begin{array}{c}\texttt{G-}\\ \texttt{-A}\end{array}} & \boxed{\begin{array}{c}\texttt{G}\\ \texttt{A}\end{array}} \\ p(0,1)+2, & p(1,0)+2, & p(0,0)+1 \end{array} \right) \quad (1.1)$$

$$= min(2+2, 2+2, 1+0) = 1$$

Look at that. We've proved that the optimal alignment of G with A using our scoring scheme is

$$\boxed{\begin{array}{c}\texttt{G}\\ \texttt{A}\end{array}}$$

And its cost is 1.

This notation is getting unwieldy. Let's make a shorter way of writing the score for a column. We'll continue to use the same scoring scheme, but we'll use a shorter way to write it. Define

$$\begin{array}{rcrcr} & & c(\texttt{X},\texttt{Y}) & = & 1 \\ c(-,\texttt{X}) & = & c(\texttt{X},-) & = & 2 \\ & & c(\texttt{X},\texttt{X}) & = & -2 \\ & & c(-,-) & = & 0 \end{array}$$

Where X and Y are letters for an amino acid, and $-$ is a gap. Why do we bother defining $c(-,-) = 0$ when two gaps are never allowed in the same column? It will make things convenient later.

Let's call $s_1 = \texttt{GATAGTC}$ and $s_2 = \texttt{AGGTGCA}$. Then we can say that $s_1[j]$ is the jth character of s_1, starting at 1. For example, $s_1[1] = \texttt{G}$, $s_1[2] = \texttt{A}$, and $s_1[3] = \texttt{T}$.

Now that we have these new notations, let's go back and fix things up. We can rewrite equation 1.1 as

$$\begin{array}{rclclcl} p(0,0) & = & 0 & & & & \\ p(0,1) & = & c(-,s_2[1]) + p(0,0) & = & c(-,\texttt{A}) & = & 2 \\ p(1,0) & = & c(s_1[1],-) + p(0,0) & = & c(\texttt{G},-) & = & 2 \end{array} \quad (1.2)$$

$$\begin{aligned} p(1,1) &= min \left(\begin{array}{l} p(0,0) + c(s_1[1], s_2[1]), \\ p(0,1) + c(s_1[1], -), \\ p(1,0) + c(-, s_2[1]) \end{array} \right) \\ &= min \left(\begin{array}{l} p(0,0) + c(\texttt{G},\texttt{A}), \\ p(0,1) + c(\texttt{G}, -), \\ p(1,0) + c(-,\texttt{A}) \end{array} \right) \\ &= min(0+1, 2+2, 2+2) = 1 \end{aligned} \quad (1.3)$$

Now we have all the tools we need to succinctly define $p(x,y)$ for any x and y on any pair of sequences s_1 and s_2.

$$p(x,y) = min \left(\begin{array}{l} c(s_1[x], s_2[y]) + p(x-1, y-1), \\ c(s_1[x], -) + p(x-1, y), \\ c(-, s_2[y]) + p(x, y-1) \end{array} \right) \quad \text{when } x > 0 \text{ and } y > 0$$

$$
\begin{aligned}
p(0,y) &= c(-,s_2[y]) + p(0,y-1) \quad \text{when } y > 0 \\
p(x,0) &= c(s_1[x],-) + p(x-1,y) \quad \text{when } x > 0 \\
p(0,0) &= 0
\end{aligned}
\tag{1.4}
$$

Now it's easy to write down all of the entries in a table T.

$p(x,y)$	x	0	1	2	3	4	5	6	7
y			G	A	T	A	G	T	C
0		**0**	2	4	6	8	10	12	14
1	A	**2**	1	0	2	4	6	8	10
2	G	4	**0**	2	1	3	2	4	6
3	G	6	2	**1**	3	2	1	3	5
4	T	8	4	3	**-1**	**1**	3	-1	1
5	G	10	6	5	1	0	**-1**	1	0
6	C	12	8	7	3	2	1	**0**	-1
7	A	14	10	6	5	1	3	2	**1**

(1.5)

So, our initial guess was correct. The best alignment score, listed in the bottom right corner, is 1. That means our initial alignment was (one of possibly several) optimal. But suppose we hadn't already guessed an alignment that turned out to have an optimal score? How could we use this table to find the optimal alignment? Observe that at each entry, the "winning" predecessor, the one that had the lowest value among those considered in the $min()$ expression, represents the best column to concatenate to get the optimal alignment corresponding to the portion of the sequences up to those coordinates. The digits in bold represent the entries along one of the optimal paths from the beginning of the sequences to the end. Then we just follow the path back, prepending columns as we go. We start with $p(7,7)$, which can be reached equally well from $p(6,6)$ (by mismatching `C` with `A`) or from $p(7,6)$ (by putting a gap above `A`). We'll choose $p(6,6)$, because alignments with fewer gaps are often considered better.

```
      -        G        A        T        A        G        T        C
      A        G        G        T        -        G        C        A
```
$p(0,0) \rightarrow p(0,1) \rightarrow p(1,2) \rightarrow p(2,3) \rightarrow p(3,4) \rightarrow p(4,4) \rightarrow p(5,5) \rightarrow p(6,6) \rightarrow p(7,7)$

This technique of building a table of all possible intermediate results to compute the optimal alignment is used in many optimization problems throughout computing science, and is known generally as **dynamic programming**. The application of the dynamic programming approach to sequence alignment was first described by Needleman and Wunsch [40] for the special case of two sequences.

Complexity Analysis

Computing scientists like to categorize algorithms by their **computational complexity**, which expresses the memory and time it consumes as a function of the size of the problem.

For example, the dynamic programming algorithm for sequence alignment requires a table with $s \times t$ entries to align two sequences of lengths s and t. So we say that its space requirement is in the category $O(s \times t)$. To compute each of those entries it must look at 3 neighbouring table entries, so its time requirements are $3s \times t$, neglecting the entries in the first row and column. So, its memory requirements are also in the category $O(s \times t)$, since we also neglect constant factors when categorizing algorithms.

1.2.2 Aligning Several Sequences

The dynamic programming technique can easily be extended to compute alignments for more than two sequences. We simply add parameters to $p(x, y)$. To align three sequences with dynamic programming, we build a table with entries $p(x, y, z)$. If their lengths are s, t, and u, we would need a table with $s \times t \times u$ entries. For four sequences, we would have a table with $s \times t \times u \times v$ entries named $p(w, x, y, z)$. And so on. There are several formulas one could use to compute the value of each cell from the values of its predecessors. We will discuss them in the next chapter. For now, we will merely note that in order to compute the value of $p(x, y, z)$, we must consider:

$$\begin{array}{l}
p(x, y, z-1) \\
p(x, y-1, z) \\
p(x, y-1, z-1) \\
p(x-1, y, z) \\
p(x-1, y, z-1) \\
p(x-1, y-1, z) \\
p(x-1, y-1, z-1)
\end{array}$$

That is 7 predecessors. When computing $p(w, x, y, z)$ we must consider each of:

$$\begin{array}{ll}
 & p(w-1, w, y, z) \\
p(w, x, y, z-1) & p(w-1, x, y, z-1) \\
p(w, x, y-1, z) & p(w-1, x, y-1, z) \\
p(w, x, y-1, z-1) & p(w-1, x, y-1, z-1) \\
p(w, x-1, y, z) & p(w-1, x-1, y, z) \\
p(w, x-1, y, z-1) & p(w-1, x-1, y, z-1) \\
p(w, x-1, y-1, z) & p(w-1, x-1, y-1, z) \\
p(w, x-1, y-1, z-1) & p(w-1, x-1, y-1, z-1)
\end{array}$$

That's a total of 15 predecessors. The pattern that is emerging is that we have to consider $2^m - 1$ predecessors at each entry, where m is the number of sequences we are aligning. Let's look again at Figure A.1. This is an alignment of 16 sequences, each of approximately 140 letters in length. So the table would have to contain approximately $140^{16} = 2177953337809371136000000000000000 \approx 2 \times 10^{33}$ entries. Given that the Earth weighs approximately 6×10^{24} kilograms, one might be able to pull it off if the entire planet were one giant memory chip. Further, at each entry we would have to consider $2^{16} - 1 = 65535$ predecessors. The memory complexity for this dynamic programming algorithm is $O(n^m)$, where n is the average length of the sequences, and its time complexity is $O(2^m \times n^m)$.

Clearly, dynamic programming is completely infeasible for more than a few sequences.

1.3 Contributions

Now that we have learned a little bit about sequence alignment from both the biological and computational perspectives, we can begin to discuss the contributions of this thesis. It turns out that biologists are mainly interested in finding alignments for sequences that are similar to each other, since the purpose of a sequence alignment is to identify subtle traces of preserved structure among related sequences. It is not really meaningful in a biological sense to compare completely unrelated and dissimilar sequences. When the sequences to be aligned are similar to each other, as compared to the typical similarity between sequences generated randomly, large portions of the dynamic programming table are irrelevant. That is, many entries are nowhere near any optimal path. Consider, for example, the top right corner entry of the table in equation 1.5. This represents the partial alignment:

```
GATAGTC
-------
```

which can be extended only to a complete alignment of

```
GATAGTC-------
-------AGGTGCA
```

For some sequences and scoring schemes, such an alignment may actually be the best. In other situations, including all of those of interest to biologists, it is a waste of time. To reduce the cost of computing the table, one could simply refuse to compute the entries far away from the likely location of the optimal path, and treat them as having some extremely undesirable score. However, our goal is to develop an algorithm that provides alignments that are guaranteed to be optimal with no tolerance for error. There always exists the possibility, however small, that an ignored cell could have been on the optimal path, so this technique is inappropriate, though it is used by alignment algorithms that only claim to provide approximate results.

There is a technique broadly used in the field of artificial intelligence(AI) known as **A* search**. Briefly, it keeps a list of "good" partial solutions to a problem, and uses a **heuristic** function to select one to extend. If this heuristic satisfies certain requirements, then eventually a complete solution will be reached, and it will be guaranteed to be optimal. The author's contribution is a heuristic function for sequence alignment based upon an **octree** that is efficient in space and time usage.

1.4 Summary

This chapter has introduced and motivated the sequence alignment problem in biology, and the reader should now have the conceptual tools required to think about the problem in mathematical terms. We will spend the next chapter formalizing the sequence alignment problem and discussing fundamental algorithms for computing alignments based solely on the primary sequence.

Chapter 2

Notation and Fundamental Algorithms

2.1 Notation

Take a set of K sequences $S = \{S_1, \ldots, S_K\}$ consisting of letters drawn from an alphabet Σ that does not include the **gap** (also called **blank**) character '$-$'. An **alignment** of S is a matrix, $\mathcal{A}$, with K rows, ω columns, and entries from $\Sigma' = \Sigma \cup \{-\}$. Each row of $\mathcal{A}$ contains one sequence from S, such that row i of $\mathcal{A}$, called $\mathcal{A}_i$, would be equal to S_i if the blanks were removed. Each row may have any number of blanks (including zero), but all rows must be of the same length and no column may consist entirely of blanks. This means that ω, the number of columns in $\mathcal{A}$, must fall in the range

$$\max_{1\leq i\leq K}\{\ell_i\} \leq \omega \leq \sum_{1\leq i\leq K} \ell_i,$$

where ℓ_i is the length of sequence S_i.

We can define a score $c(\mathcal{A})$ for $\mathcal{A}$. This score is defined in terms of primitive score functions c_{letter} and c_{gap} which deal with pairs of rows of $\mathcal{A}$.

$$\sum_{1\leq i<j\leq K} w_{ij}(c_{\text{letter}}(\mathcal{A}_i, \mathcal{A}_j) + c_{\text{gap}}(\mathcal{A}_i, \mathcal{A}_j))$$

The numbers $\{w_{ij}\} \subset \mathbb{R}$ are called **weights**. They are mentioned here for completeness with respect to previous work, but they are a superficial embellishment. Therefore, for the purposes of clarity, they are set to 1 and not mentioned again in this work.

The function c_{letter} is in turn defined in terms of a simple function $c : \Sigma' \times \Sigma' \to \mathbb{Z}$ called the **substitution matrix**. We re-use the letter c here, but which c is meant should be clear from the context. Sometimes in theoretical discussions $\mathbb{R}$ is used instead of $\mathbb{Z}$ but practical implementations always use $\mathbb{Z}$ for efficiency.

$$c_{\text{letter}}(\mathcal{A}_i, \mathcal{A}_j) = \sum_{1\leq k\leq \omega} c(a_{ik}, a_{jk})$$

The substitution matrix must satisfy $c(x,-) = c(-,x) = c(-,-) = 0$ when $x \in \Sigma$ (i.e. when x is anything but the blank character).

The constituent function c_{gap}, too, is defined on pairs of rows. It may be chosen from among many alternatives. We will be concerned mainly with the simplest of these, the linear gap cost c_{linear}:

$$c_{\text{linear}}(\mathcal{A}_i, \mathcal{A}_j) = \sum_{1 \le k \le \omega} c_g(a_{ik}, a_{jk}),$$

where c_g is yet another auxiliary function parameterized by $g \in \mathbb{R}$:

$$\begin{aligned} c_g(x,y) &= 0 \quad \text{for } x = y = - \\ &= g \quad \text{for } x \in \Sigma,\ y = - \\ &= g \quad \text{for } x = -,\ y \in \Sigma \\ &= 0 \quad \text{for } x, y \in \Sigma. \end{aligned}$$

2.2 Sequence Alignment Algorithms

We now consider algorithms for computing optimal alignments. An optimal alignment $\bar{\mathcal{A}}$ for a set of sequences $\{S_1, \ldots, S_K\}$ is defined with respect to a substitution matrix c and score function c_{gap}.

The substitution matrix c and the gap scoring function c_{gap} may be formulated such that the meaningfully optimal alignment could be taken as the maximum score over all possible alignments $\mathcal{A}$ of a set of sequences, or as the minimum. Unfortunately the biology community has chosen to use the maximum and the computing science community has chosen to use the minimum. The two treatments are isomophic. We will use the minimum in this work, and so the score is often referred to as the cost.

2.2.1 Aligning Two Sequences

We begin with the simplest possible case, that of aligning two sequences using the linear gap function c_{linear} with gap cost g. There are two major ways to do this.

Recursive Method

We start with a recursive formulation in Figure 2.1. For brevity S_1, S_2, g, and c are taken to be implicit parameters to DP2Linear-Rec.

Proof of optimality:

Abbreviate "DP2Linear-Rec" by d. The trivial cases $d(0,0) = 0$, $d(0,x) = xg$, and $d(x,0) = xg$ are dealt with by lines 1, 2, 3 of the algorithm. Proceed by mathematical induction. Assume that if $i' < i$ and $j' \le j$, or $i' \le i$ and $j' < j$, that $d(i',j')$ is the value of an optimal alignment $\bar{\mathcal{A}} = \bar{\mathcal{A}}(S_1[1 \ldots i], S_2[1 \ldots j])$ with $\bar{\omega}$ columns. Then the first $\bar{\omega} - 1$ columns of $\bar{\mathcal{A}}$ must be an alignment $\widehat{\mathcal{A}} = \bar{\mathcal{A}}_{1 \ldots \bar{\omega}-1}$ of the first $(i-1, j)$, $(i, j-1)$, or $(i-1, j-1)$ characters of S_1, S_2. The score for $\bar{\mathcal{A}}$ is formed by adding $c(\widehat{\mathcal{A}})$ to the score for

```
function DP2Linear-Rec(i,j): integer
     if i = 0 and j = 0 then DP2Linear-Rec ← 0
else if i = 0 then DP2Linear-Rec ← DP2Linear-Rec(i,j − 1) +g
else if j = 0 then DP2Linear-Rec ← DP2Linear-Rec(i − 1,j) +g
else DP2Linear-Rec ← min( DP2Linear-Rec(i − 1,j − 1) + c(S1[i], S2[j]),
                          DP2Linear-Rec(i,j − 1) + g,
                          DP2Linear-Rec(i − 1,j) + g )
```

Figure 2.1: Recursive algorithm to align two sequences using linear gap cost

the last column, $\bar{\omega}$, which is independent of any before it. If $\bar{\mathcal{A}}$ is optimal then the alignment $\widehat{\mathcal{A}}$ must be optimal. If it were not, we could replace $\widehat{\mathcal{A}}$ in $\bar{\mathcal{A}}$ with an optimal alignment to result in a new alignment $\widehat{\mathcal{A}'}$ having a better score, contradicting the optimality of $\widehat{\mathcal{A}}$. The last column of $\bar{\mathcal{A}}$ must look like one of

$$\text{(a)}\ \begin{array}{c} S_1[i] \\ S_2[j] \end{array} \qquad \text{(b)}\ \begin{array}{c} - \\ S_2[j] \end{array} \qquad \text{(c)}\ \begin{array}{c} S_1[i] \\ - \end{array} \qquad .$$

In case (a), $\widehat{\mathcal{A}} = \bar{A}(S_1[1\ldots i-1], S_2[1\ldots j-1])$ and the score for the last column is $c(S_1[i], S_2[j])$, making the overall score

$$c(\bar{\mathcal{A}}) = d(i-1, j-1) + c(S_1[i], S_2[j]).$$

For case (b), $\widehat{\mathcal{A}} = \bar{A}(S_1[1\ldots i][1\ldots j])$ and the score for the last column is g, so

$$c(\bar{\mathcal{A}}) = d(i, j-1) + g.$$

Case (c) is similar:

$$c(\bar{\mathcal{A}}) = d(i-1, j) + g.$$

So the optimal score is:

$$\begin{array}{rcll} d(i,j) & = & \min(& d(i-1,j-1) + c(S_1[i], S_2[j]), \\ & & & d(i-1,j) + g, \\ & & & d(i,j-1) + g). \end{array}$$

This recursive formula does not translate directly into an efficient implementation in a procedural language, nor does it offer a way to recover $\mathcal{A}$ from the result of $d(\ell_1, \ell_2)$.

Iterative Method

Figure 2.2 illustrates DP2Linear-Iter, an algorithm for constructing the entire dynamic programming matrix using explicit loops. DP2Linear-Recover (Figure 2.3) may then be employed to recover the alignment from the dynamic programming table, much like the example from Chapter 1.

There may be many optimal alignments $\bar{\mathcal{A}}$. DP2Linear-Recover may be modified to list them all. We call the sequence of cells traversed by this loop through M the “path” of the alignment $\mathcal{A}$.

DP2Linear-Iter(S_1,S_2) : array[. . .][. . .]
M ←array[$0 \ldots \ell_1$,$0 \ldots \ell_2$]
for i ← 0 **to** ℓ_1 **do** $M[i,0] \leftarrow i * g$
for j ← 0 **to** ℓ_2 **do** $M[0,j] \leftarrow j * g$
for i ← 1 **to** ℓ_1 **do begin**
for j ← 1 **to** ℓ_2 **do begin**
$M[i,j]$← min($M[i-1,j-1] + c(S_1[i], S_2[j])$,
$M[i-1,j] + g$,
$M[i, \quad j-1] + g$)
end
end
DP2Linear-Iter ← M // *score is in* $M[\ell_1, \ell_2]$
end

Figure 2.2: Iterative solution to aligning two sequences with linear gap cost. The time and space complexities of this algorithm are both $O(\ell_1 \times \ell_2)$.

DP2Linear-Recover(S_1, S_2, M) : result is alignment as array
$\mathcal{A}$ ← array[$2, \omega(M)$]
$i \leftarrow \ell_1$; $j \leftarrow \ell_2$; Acol ← 1
while $i <> 0$ **or** $j <> 0$ **do begin**
if $i > 1$ **and** $j > 1$ **and** $M[i,j] - c(S_1[i], S_2[j]) = M[i-1, j-1]$ **then begin**
$\mathcal{A}$[1,Acol] ← $S_1[i]$
$\mathcal{A}$[2,Acol] ← $S_2[j]$
$i \leftarrow i - 1$; $j \leftarrow j - 1$
end else if $i > 0$ and $M[i,j] - g = M[i-1, j]$ **then begin**
$\mathcal{A}$[1,Acol] ← $S_1[i]$
$\mathcal{A}$[2,Acol] ← '-'
$i \leftarrow i - 1$
end else if $j > 0$ and $M[i,j] - g = M[i, j-1]$ **then begin**
$\mathcal{A}$[1,Acol] ← '-'
$\mathcal{A}$[2,Acol] ← $S_2[j]$
$j \leftarrow j - 1$
end // *There are no other possibilities.*
Acol ← Acol + 1
end
$\mathcal{A}$ ← reverse-order-of-columns($\mathcal{A}$)
DP2Linear-Recover ← $\mathcal{A}$
end

Figure 2.3: Algorithm to recover the alignment from the matrix M produced by DP2Linear-Iter (Figure 2.2). The time and space complexities of this algorithm are both $O(\ell_1 + \ell_2)$.

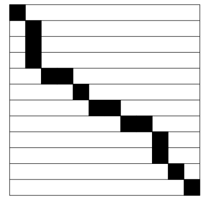


Figure 2.4: The optimal path must pass through each row

Hirschberg's Algorithm

For very long sequences the table M may be too large to fit in memory. Hirschberg [21] described a **divide-and-conquer** algorithm to align two sequences in $O(\ell)$ space and $O(\ell_1 \times \ell_2) = O(\ell^2)$ time. Hirschberg's algorithm works by splitting the sequences in two, recursively aligning the halves, and then joining the alignments together to yield the final answer. As discussed in Chapter 1, if $M[i,j]$ is on the optimal path then we can split S_1 into two portions, $S_1^a = S_1[1 \ldots i]$ and $S_1^z = S_1[i+1 \ldots \ell_1]$. Likewise $S_2^a = S_2[1 \ldots j]$ and $S_2^z = S_2[j+1 \ldots \ell_2]$. The optimal alignment $\bar{A}(S_1^a, S_2^a)$ of the two first halves and the optimal alignment $\bar{A}(S_1^z, S_2^z)$ of the two second halves can then be concatenated to obtain the optimal alignment $\bar{A}(S_1, S_2)$. Since the optimal path must pass through each row (Figure 2.4) we can pick the middle row (any would do, but the middle typically makes the total size of the subproblems smaller) which fixes i at $\ell_1/2$, and find a column j so that $M[i,j]$ is on the optimal path. Then we can break the sequences at (i,j) and recurse, using DP2Linear-Iter if M would fit in the available space, or Hirschberg's algorithm again.

How do we find an entry $M[i,j]$ on the optimal path? It's not enough just to compute the row $M[i]$ because the minimum entry on that row won't necessarily extend to the lowest-cost path overall. In order to know whether (i,j) is on the optimal path we also have to know the minimum cost $E[i,j]$ from (i,j) to (ℓ_1, ℓ_2) – the *end* of the sequences. That is, given i we need to find j so that $M[i,j] + E[i,j]$ is minimal. How do we get the row $E[i]$? Simple – by reversing the sequences and computing $M[\ell_1 - i]$. HirschRow (Figure 2.6) takes care of computing the ith row of M in a space-efficient way. The main code for Hirschberg (Figure 2.5) computes the two middle rows needed, searches for the optimal crossing, and recurses.

Charter, Schaeffer, and Szafron [7] generalized Hirschberg's algorithm with their FastLSA algorithm, which makes better use of available space and is also readily parallelizable [11].

2.2.2 Aligning More Than Two Sequences

Hirschberg's algorithm is readily generalized to dynamic programming on any number of sequences. For three sequences the matrix is three-dimensional, and the path is guaranteed to pass through any given plane, so Hirschberg's algorithm can be used to fix a position along

Hirschberg(S_1,S_2) : alignment
    if $\ell_1 * \ell_2 <$ AvailableMemory **then**
        // Base case.
        Hirschberg $\leftarrow$ DP2Linear-Iter(S_1,S_2)
    else begin
        $i \leftarrow \ell_1/2$
        row1 $\leftarrow$ HirschRow(S_1,S_2,i)
        row2 $\leftarrow$ HirschRow(reverse(S_1),reverse(S_2), $\ell_1 - i$)
        minscore $\leftarrow \infty$
        for $j \leftarrow 1$ **to** ℓ_2 **do begin**
            score $\leftarrow$ row1[j] + row2[$\ell_2 - j$]
            if score < minscore **then begin**
                minscore $\leftarrow$ score
                minj $\leftarrow$ j
            end
        end
        Hirschberg $\leftarrow$ concatenate(Hirschberg($S_1[1 \ldots i], S_2[1 \ldots \text{minj}]$),
                        Hirschberg($S_1[i+1 \ldots \ell_1], S_2[\text{minj}+1 \ldots \ell_2]$))
    end

Figure 2.5: Hirschberg's main algorithm

// Compute row "row" of M from DP2Linear-Iter using only two rows of storage
HirschRow(S_1,S_2,row) : array
    $R \leftarrow$ array$[0 \ldots \ell_2]$; $R' \leftarrow$ array$[0 \ldots \ell_2]$

    // First row of M: align empty string against S_2.
    for $j \leftarrow 0$ **to** ℓ_2 **do** $R[j] \leftarrow g * i$

    // Remaining rows up to row.
    for $i \leftarrow 1$ **to** row **do begin**
        $R'[0] \leftarrow R[0] + g$

        // Compute row $M[i]$ into R' using the
        // previous row $M[i-1]$, stored in R.
        for $j \leftarrow 1$ **to** ℓ_2 **do begin**
a: $R'[j] \leftarrow \min($ $R'[j-1] + g$,
b: $R[j-1] + c(S_1[i], S_2[j])$,
c: $R[j] + g$)
        end
        copy R' to R // Current row becomes previous.
    end
    HirschRow $\leftarrow R$

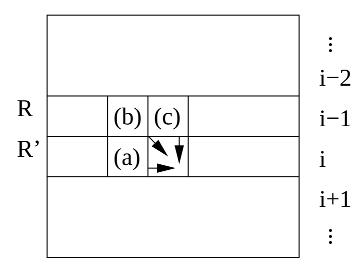


Figure 2.6: The auxiliary Hirschberg function HirschRow. A more efficient implementation would dispense with copying and use only one row of storage, plus one additional cell.

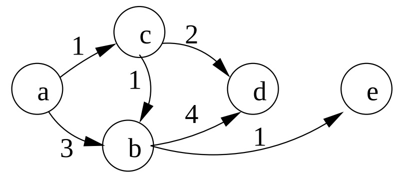


Figure 2.7: The concept of the weighted, directed graph is easy to grasp intuitively. **Nodes** (also called **vertices**) are connected by one-way **edges** with associated **weights** (also **costs**). In this example, the circles labelled a,b,c,d,e are nodes, and the arrows pointing between them are edges. The **shortest path** from node a to node e goes through nodes c and b and has a cost of 3. Note that the "shortest path" is not the one with the fewest edges, but the lowest cost. Node d is a dead end. The **successors** of node c are nodes b and d. The **predecessors** of node b are nodes a and c.

one sequence, which specifies a plane. That plane may then be computed and the minimum-valued entry found in it to retrieve the optimal split position for the other two sequences. Likewise for four sequences, we may fix a position along one sequence and then compute the 3D block for the other three sequences. The emerging pattern is that Hirschberg's algorithm allows a space savings of only one factor of ℓ, taking the overall space complexity (computing science jargon for "usage") from $O(\ell^K)$ to $O(\ell^{K-1})$.

Dynamic programming requires too much storage. We need to try a different approach.

The A* search algorithm

A* is an algorithm used to solve a vast range of problems in search. So far we have looked at algorithms that compute the value of a cell in the space of optimal alignment costs of prefixes by "looking back" at the values of preceding cells. Since we do not know ahead of time which value will "win" at any given cell, we have had to compute the entire table to ensure that the values of all directly or indirectly preceding cells are available. A* allows us to work forward and completely ignore portions of the table which can be proven to be irrelevant to the final result.

A* is applicable for optimization problems which can be formulated as a search for a shortest path through a type of space called a weighted directed graph(Figure 2.7).

Briefly, A* maintains a growing list (the **open list**) of nodes in the graph. With each node u on the list it records the best known path (with **cost** g) to reach it, and a heuristic estimate (that is, a guess) h as to the cost of the best path from it to the goal node. It then picks the node on the open list with lowest value $f = g + h$, moves it from the open list to the **closed list** and adds its successors to the open list (**expands** it). This step is repeated until the goal node is on "top" of the open list. If h always underestimates the true value h^* of the cost of the best path from u to the goal (h is **admissible**), then when the goal node reaches the top of the open list, its cost g is guaranteed to be that of the optimal path. For simplicity, our discussion and sample implementation of A* assume a stronger condition

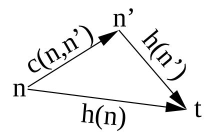


Figure 2.8: A **consistent** heuristic satisfies the **triangle inequality**.

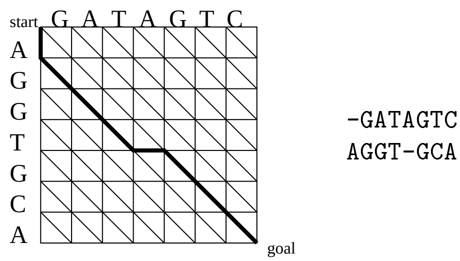


Figure 2.9: The example of Chapter 1 depicted as a lattice. Here, the corners, rather than the interiors of the cells, are the occupiable states, or nodes, and the lines are the edges connecting them. Arrowheads are not drawn: edges are pointing downwards and rightwards.

called **consistency**. If n' is a successor of n, then consistency requires that

$$h(n) \leq c(n \to n') + h(n').$$

Figure 2.8 illustrates this concept. Consistency implies admissibility, but not vice versa. However, a consistent heuristic can be constructed from an admissible one, which allows our simplification. A deeper discussion of how admissibility and consistency relate to A* search can be found in any introductory text on artificial intelligence [47].

Sequence alignment is readily framed in terms of a search for the shortest path through a graph [23]. Throughout the following, we will use the linear gap scoring function.

Because of its regular structure, the sequence alignment graph is also called a **lattice** (Figure 2.9). Call this K-dimensional lattice $L(S_1, \ldots, S_K)$, or just L. A node $u = (u_1, \ldots, u_K) \in L$ represents the partial alignment of $S_1[1 \ldots u_1], \ldots, S_K[1 \ldots u_K]$. A successor $v = (v_1, \ldots, v_K)$ of u has the property that $v_i = u_i$ or $v_i = u_i + 1$, so that a move from u to v equates to adding one column to the alignment represented by u. The cost of that edge is the function c applied to that column. A path through L corresponds to an alignment, and the shortest (lowest-cost) path corresponds to the optimal alignment.

The admissible heuristic h required by A* must provide a lower-bound estimate of the length of the shortest path h^* from a node u to the goal node t. For sequence alignment this is equivalent to the cost of the optimal alignment $c(\bar{A}(S_1[u_1+1 \ldots \ell_1], \ldots, S_K[u_K+1 \ldots \ell_K]))$.

How can we construct an admissible heuristic for sequence alignment? Suppose we have aligned S_1, S_2, and S_3 optimally like so:

$$\bar{A}(\mathtt{ACGT}, \mathtt{ACT}, \mathtt{AGT}) = \begin{array}{cccc} \mathtt{A} & \mathtt{C} & \mathtt{G} & \mathtt{T} \\ \mathtt{A} & \mathtt{C} & - & \mathtt{T} \\ \mathtt{A} & - & \mathtt{G} & \mathtt{T} \end{array}.$$

		T	G	C	A
	0	2	4	6	8
T	2	-2	0	2	4
C	4	0	-1	-2	0
A	6	2	1	0	-4

		T	G	C	A
	0	2	4	6	8
T	2	-2	0	2	4
G	4	0	-4	-2	0
A	6	2	-2	-3	-4

		T	C	A
	0	2	4	6
T	2	-2	0	2
G	4	0	-1	1
A	6	2	1	-3

Table 2.1: The h_p tables for ACGT, ACT, and AGT.

We will use the scoring function for the main example of Chapter 1: -2 for a match, 1 for a mismatch, and 2 for a gap. The cost of the alignment is

$$c\left(\begin{array}{c}\mathtt{ACGT}\\ \mathtt{AC-T}\end{array}\right)+c\left(\begin{array}{c}\mathtt{ACGT}\\ \mathtt{A-GT}\end{array}\right)+c\left(\begin{array}{c}\mathtt{AC-T}\\ \mathtt{A-GT}\end{array}\right)=-4-4+0=-8.$$

Notice that each of these terms is the cost of an alignment of a pair of sequences, possibly optimal, possibly not. The optimal alignments and scores of each pair are

$$c\left(\begin{array}{c}\mathtt{ACGT}\\ \mathtt{AC-T}\end{array}\right)+c\left(\begin{array}{c}\mathtt{ACGT}\\ \mathtt{A-GT}\end{array}\right)+c\left(\begin{array}{c}\mathtt{ACT}\\ \mathtt{AGT}\end{array}\right)=-4-4-3=-11.$$

It is no accident that this total is lower. It is guaranteed, since the optimal score of the alignment for three sequences is computed as the sum over three two-way alignments. But since those alignments are themselves optimal at best (how could they be better?) there is no way their scores could exceed those of the corresponding optimal pair-wise alignments. This means that any time we want a lower-bound estimate for

$$h^* = c(\bar{A}(S_1, \ldots, S_K))$$

we can use

$$h_p = \sum_{1 \le i < j \le K} c(\bar{A}(S_i, S_j)). \tag{2.1}$$

and always be guaranteed that $h_p \le h^*$, as required by A*. We call h_p the **pairwise heuristic** because it is constructed from alignments of pairs of sequences. This heuristic is also consistent, though we will not prove it. Other heuristics are conceivable. We will talk about them in Chapter 4.

An understanding of the A* algorithm and how it applies to sequence alignment is important enough that we present a fully worked example. We align the three sequences $S_1 =$ ACGT, $S_2 =$ ACT, and $S_3 =$ AGT, again using -2 for a match, 1 for a mismatch, and 2 for a gap.

The first thing we will need is the set of tables for h_p. We need h_p to estimate the cost from *any* node to the goal node. We could use M from DP2Linear-Iter, but it provides the cost from the *start* node to any node. We can still use it, however: much as we did with Hirschberg's algorithm (Section 2.2.1), we can reverse the sequences and compute M from DP2Linear-Iter just once to get the needed values for any node – see Table 2.1.

A* keeps two lists of lattice nodes under consideration, called "OPEN" and "CLOSED". OPEN contains nodes that have been identified as possibly worth expanding, and CLOSED contains nodes that have been expanded. Expanding a node consists of moving it from OPEN to CLOSED and putting its successors into OPEN. We start with just one node in OPEN, the start node $(0,0,0)$ representing the empty alignment of the first zero characters of each of S_1, S_2, and S_3. CLOSED is empty.

The search begins:

$$\text{OPEN} = \{\ \{(0,0,0), f = g + h = 0 + (-4 + -4 + -3) = -11, \mathit{predecessor} = \textbf{NIL}\}\ \}$$

$$\text{CLOSED} = \{\ \ \}$$

We add a *predecessor* pointer (hereinafter abbreviated p) to each node so we can recover the alignment once we reach the goal node. The start node has no predecessors, a fact we denote by **NIL**.

The node $(0,0,0)$ has value $g = 0$ because it represents the empty alignment. To see how $h = (-4 + -4 + -3) = -11$ is obtained, recall from Equation 2.1 and Table 2.1 that:

$$\begin{aligned}
h_p(0,0,0) &= c(\bar{A}(S_1[0\ldots\ell_1], S_2[0\ldots\ell_2]))+ \\
&\quad\ c(\bar{A}(S_1[0\ldots\ell_1], S_3[0\ldots\ell_3]))+ \\
&\quad\ c(\bar{A}(S_2[0\ldots\ell_2], S_3[0\ldots\ell_3])) \\
&= c(\bar{A}(\texttt{ACGT},\texttt{ACT})) + c(\bar{A}(\texttt{ACGT},\texttt{AGT})) + c(\bar{A}(\texttt{ACT},\texttt{AGT})) \\
&= c(\bar{A}(\texttt{TGCA},\texttt{TCA})) + c(\bar{A}(\texttt{TGCA},\texttt{TGA})) + c(\bar{A}(\texttt{TCA},\texttt{TGA})) \\
&= -4 + -4 + -3
\end{aligned}$$

We now proceed to take the node with lowest f off the open list. It happens to be the only node on the open list right now. First we check whether it is the goal: its coordinates are $(0,0,0)$, and the goal has coordinates $(4,3,3)$, so it is not. We put it on the closed list, and add its successors to the open list. One of these successors, $(1,0,0)$, is shown in more detail than the others. Its g is shown in two components. The first, 0, is the cost to reach its predecessor. The second, $(2+2+0)$, is the cost to reach it from its predecessor. Its predecessor is $(0,0,0)$, representing the empty alignment with value 0. The cost to reach $(1,0,0)$ from $(0,0,0)$ is the cost of adding the column

$$\begin{array}{c}\texttt{A}\\ -\\ -\end{array}$$

which has cost $c(\texttt{A},-) + c(\texttt{A},-) + c(-,-) = 2 + 2 + 0 = 4$.

For h:

$$\begin{aligned}
h_p(1,0,0) &= c(\bar{A}(S_1[1\ldots\ell_1], S_2[0\ldots\ell_2]))+ \\
&\quad\ c(\bar{A}(S_1[1\ldots\ell_1], S_3[0\ldots\ell_3]))+ \\
&\quad\ c(\bar{A}(S_2[0\ldots\ell_2], S_3[0\ldots\ell_3])) \\
&= c(\bar{A}(\texttt{CGT},\texttt{ACT})) + c(\bar{A}(\texttt{CGT},\texttt{AGT})) + c(\bar{A}(\texttt{ACT},\texttt{AGT})) \\
&= c(\bar{A}(\texttt{TGCA},\texttt{TCA})) + c(\bar{A}(\texttt{TGCA},\texttt{TGA})) + c(\bar{A}(\texttt{TCA},\texttt{TGA})) \\
&= 0 + -3 + -3
\end{aligned}$$

$$\text{OPEN} = \left\{ \begin{array}{lll} \{(1,1,1), & f = -6 + -5 = -11, & p = (0,0,0)\} \\ \{(1,1,0), & f = 2 - 4 = -2, & p = (0,0,0)\} \\ \{(1,0,0), & f = (0 + (2+2+0)) + (0 + -3 + -3) = -2, & p = (0,0,0)\} \\ \{(1,0,1), & f = 2 - 1 = 1, & p = (0,0,0)\} \\ \{(0,1,1), & f = 2 - 1 = 1, & p = (0,0,0)\} \\ \{(0,0,1), & f = 4 - 3 = 1, & p = (0,0,0)\} \\ \{(0,1,0), & f = 4 - 3 = 1, & p = (0,0,0)\} \end{array} \right\}$$

$$\text{CLOSED} = \left\{ \ \{(0,0,0), \quad f = 0 - 11 = -11, \quad p = \mathbf{NIL}\} \ \right\}$$

OPEN is sorted from lowest f to highest. This completes the first A* loop. We repeat. The node with lowest f in OPEN is $(1,1,1)$. It is not the goal node, so we take it off OPEN and put it in CLOSED, then add its successors to OPEN. New entrants are in bold.

$$\text{OPEN} = \left\{ \begin{array}{lll} \boldsymbol{\{(2,2,2),} & \boldsymbol{f = -6 - 2 = -8,} & \boldsymbol{p = (1,1,1)\}} \\ \boldsymbol{\{(2,2,1),} & \boldsymbol{f = -4 - 4 = -8,} & \boldsymbol{p = (1,1,1)\}} \\ \boldsymbol{\{(2,1,1),} & \boldsymbol{f = -2 - 6 = -8,} & \boldsymbol{p = (1,1,1)\}} \\ \boldsymbol{\{(2,1,2),} & \boldsymbol{f = -1 - 1 = -2,} & \boldsymbol{p = (1,1,1)\}} \\ \boldsymbol{\{(1,2,1),} & \boldsymbol{f = -2 + 0 = -2,} & \boldsymbol{p = (1,1,1)\}} \\ \boldsymbol{\{(1,1,2),} & \boldsymbol{f = -2 + 0 = -2,} & \boldsymbol{p = (1,1,1)\}} \\ \{(1,1,0), & f = 2 - 4 = -2, & p = (0,0,0)\} \\ \{(1,0,0), & f = 4 - 6 = -2, & p = (0,0,0)\} \\ \boldsymbol{\{(1,2,2),} & \boldsymbol{f = -1 + 2 = 1,} & \boldsymbol{p = (1,1,1)\}} \\ \{(0,1,1), & f = 2 - 1 = 1, & p = (0,0,0)\} \\ \{(1,0,1), & f = 2 - 1 = 1, & p = (0,0,0)\} \\ \{(0,1,0), & f = 4 - 3 = 1, & p = (0,0,0)\} \\ \{(0,0,1), & f = 4 - 3 = 1, & p = (0,0,0)\} \end{array} \right\}$$

$$\text{CLOSED} = \left\{ \begin{array}{lll} \boldsymbol{\{(1,1,1),} & \boldsymbol{f = -6 - 5 = -11,} & \boldsymbol{p = (0,0,0)\}} \\ \{(0,0,0), & f = 0 - 11 = -11, & p = \mathbf{NIL}\} \end{array} \right\}$$

Let's look again in detail at the procedure of adding one successor of $(1,1,1)$ to OPEN. We'll use $(2,2,2)$ for this example. Our first task is to find g and h, adding them to get f. The value g represents the total cost to reach $(2,2,2)$, which is simply the total cost to reach its predecessor $(1,1,1)$, plus the cost to reach $(2,2,2)$ from there. The former value is -6, and the latter is the score for the column formed by moving from position 1 to position 2 of S_1, position 1 to position 2 of S_2, and from 1 to 2 in S_3. This move forms the column

```
C
C
G
```

which has score $-2 + 1 + 1 = 0$. That makes $g = -6 + 0 = -6$. Next, we need h, which is an estimate of the remaining distance, equal to $c(\bar{A}(S_1[3 \ldots \ell_1], S_2[3 \ldots \ell_2], S_3[3 \ldots \ell_3])$ $= c(\bar{A}(\mathtt{GT}, \mathtt{T}, \mathtt{T}))$. We have chosen the heuristic h_p, equal to $c(\bar{A}(\mathtt{GT}, \mathtt{T})) + c(\bar{A}(\mathtt{GT}, \mathtt{T})) + c(\bar{A}(\mathtt{T}, \mathtt{T}))$. The tables in Table 2.1 give this as $h = 0 + 0 - 2 = -2$.

No doubt the reader has noticed that OPEN has grown quite quickly. With each node we moved to CLOSED, up to seven more took its place. However, we are undaunted, and soldier bravely on. We have our pick of three nodes tied for first place on OPEN. We can choose any of them, so we'll choose the one that is furthest along the lattice, $(2, 2, 2)$ with a score of -8.

$$
\text{OPEN} = \left\{ \begin{array}{lll}
\boldsymbol{\{(3,2,2),} & \boldsymbol{f = -2 - 6 = -8,} & \boldsymbol{p = (2,2,2)\}} \\
\{(2,2,1), & f = -4 - 4 = -8, & p = (1,1,1)\} \\
\{(2,1,1), & f = -2 - 6 = -8, & p = (1,1,1)\} \\
\boldsymbol{\{(3,3,3),} & \boldsymbol{f = -6 + 4 = -2,} & \boldsymbol{p = (2,2,2)\}} \\
\{(2,1,2), & f = -1 - 1 = -2, & p = (1,1,1)\} \\
\{(1,2,1), & f = -2 + 0 = -2, & p = (1,1,1)\} \\
\{(1,1,2), & f = -2 + 0 = -2, & p = (1,1,1)\} \\
\{(1,1,0), & f = 2 - 4 = -2, & p = (0,0,0)\} \\
\{(1,0,0), & f = 4 - 6 = -2, & p = (0,0,0)\} \\
\boldsymbol{\{(3,2,3),} & \boldsymbol{f = -1 + 2 = 1,} & \boldsymbol{p = (2,2,2)\}} \\
\boldsymbol{\{(3,3,2),} & \boldsymbol{f = -1 + 2 = 1,} & \boldsymbol{p = (2,2,2)\}} \\
\{(1,2,2), & f = -1 + 2 = 1, & p = (1,1,1)\} \\
\{(1,0,1), & f = 2 - 1 = 1, & p = (0,0,0)\} \\
\{(0,1,1), & f = 2 - 1 = 1, & p = (0,0,0)\} \\
\{(0,0,1), & f = 4 - 3 = 1, & p = (0,0,0)\} \\
\{(0,1,0), & f = 4 - 3 = 1, & p = (0,0,0)\} \\
\boldsymbol{\{(2,3,3),} & \boldsymbol{f = -4 + 8 = 4,} & \boldsymbol{p = (2,2,2)\}} \\
\boldsymbol{\{(2,2,3),} & \boldsymbol{f = -2 + 6 = 4,} & \boldsymbol{p = (2,2,2)\}} \\
\boldsymbol{\{(2,3,2),} & \boldsymbol{f = -2 + 6 = 4,} & \boldsymbol{p = (2,2,2)\}}
\end{array} \right\}
$$

$$
\text{CLOSED} = \left\{ \begin{array}{lll}
\boldsymbol{\{(2,2,2),} & \boldsymbol{f = -6 - 2 = -8,} & \boldsymbol{p = (1,1,1)\}} \\
\{(1,1,1), & f = -6 - 5 = -11, & p = (0,0,0)\} \\
\{(0,0,0), & f = 0 - 11 = -11, & p = \textbf{NIL}\}
\end{array} \right\}
$$

Several of the entries added in this round are quite poor, occupying the middle or even the very bottom of the list. We will find a way to mitigate this problem in the next section.

The top entry on OPEN, $(3, 2, 2)$, is still not the goal node, so we repeat the A* loop: move it to CLOSED and expand it. Continuing our habit of writing down every entry in OPEN is becoming burdensome, however, so we'll abbreviate our presentation, showing only the lattice coordinates of nodes with poor f-values:

$$
\text{OPEN} = \left\{ \begin{array}{l}
\begin{array}{lll}
\boldsymbol{\{(4,3,3),} & \boldsymbol{f = -8 + 0 = -8,} & \boldsymbol{p = (3,2,2)\}} \\
\{(2,2,1), & f = -4 - 4 = -8, & p = (1,1,1)\} \\
\{(2,1,1), & f = -2 - 6 = -8, & p = (1,1,1)\} \\
\{(3,3,3), & f = -6 + 4 = -2, & p = (2,2,2)\}
\end{array} \\
\begin{array}{cccccc}
(2,1,2) & (1,2,1) & (1,1,2) & (1,1,0) & (1,0,0) & \boldsymbol{(3,3,2)} \\
\boldsymbol{(3,2,3)} & (1,2,2) & (0,1,1) & (1,0,1) & (0,1,0) & (0,0,1) \\
\boldsymbol{(4,2,3)} & \boldsymbol{(4,3,2)} & \boldsymbol{(2,3,3)} & \boldsymbol{(4,2,2)} & \boldsymbol{(2,2,3)} & \boldsymbol{(2,3,2)}
\end{array}
\end{array} \right\}
$$

$$
\text{CLOSED} = \left\{ \begin{array}{lll}
\boldsymbol{\{(3,2,2),} & \boldsymbol{f = -2 - 6 = -8,} & \boldsymbol{p = (2,2,2)\}} \\
\{(2,2,2), & f = -6 - 2 = -8, & p = (1,1,1)\} \\
\{(1,1,1), & f = -6 - 5 = -11, & p = (0,0,0)\} \\
\{(0,0,0), & f = 0 - 11 = -11, & p = \textbf{NIL}\}
\end{array} \right\}
$$

We've reached the goal: $(4, 3, 3)$. It is on top of OPEN so we may safely conclude (with the guarantee A* gives us in return for providing a consistent heuristic) that we have found the shortest path to it. Reconstructing the optimal alignment is easy, we simply follow

```
AStarAlign(S_1, ..., S_K) : alignment
    for i ← 1 to K − 1 do
        for j ← i + 1 to K do
            M_ij ← DP2Align-Iter( reverse(S_i), reverse(S_j) )
    s.node ← (0, ..., 0)   // K zeroes
    s.g ← 0; s.h ← h({M_ij}, s.node); s.f = s.g + s.h
    OPEN ← {s}
    CLOSED ← ∅
    while OPEN ≠ ∅ do begin
        u ← min arg(u.f = u.g + u.h) ∈ OPEN
        if u is goal then AStarAlign ← u.alignment; exit
        OPEN ← OPEN \ {u}
        for each successor v of u do begin
            if v ∉ OPEN ∪ CLOSED then begin
                v.g ← u.g + c(u, v); v.h ← h({M_ij}, v.node)
                OPEN ← OPEN ∪ {v}
            end else if v ∈ OPEN and u.g + c(u, v) < v.g then begin
                v.g ← u.g + c(u, v)
            end
        end
        CLOSED ← CLOSED ∪ {u}
    end
    // There is always a path through the sequence alignment lattice,
    // so this cannot happen.
    AStarAlign ← no solution
end
```

Figure 2.10: The canonical A* sequence alignment algorithm for linear gap cost and pairwise heuristic. $h(u) = \sum_{i,j} M_{ij}[\ell_i - u.\text{node}[i], \ell_j - u.\text{node}[j]]$.

predecessor pointers back up to the start node.

$$\begin{array}{ccccc} \texttt{T} & & \texttt{G} & \texttt{C} & \texttt{A} \\ \texttt{T} & & - & \texttt{C} & \texttt{A} \\ \texttt{T} & & - & \texttt{G} & \texttt{A} \end{array}$$

$$(4,3,3) \to (3,2,2) \to (2,2,2) \to (1,1,1) \to (0,0,0)$$

Notice that many entries in the lattice are found neither in OPEN nor CLOSED, for example $(3,1,2)$. Combined they contain 26 entries out of a total possible of $5 \times 4 \times 4 = 80$. That's quite a savings! With longer sequences the relative savings improves.

Partial-Expansion A*

We saw in the previous section that each time we expanded a node we were forced to put several nodes onto OPEN that were clearly of no interest. However, it is not safe to simply discard them, since we cannot be sure that they will not rise slowly to the top and redeem themselves. We call the number b of successors of each node the **branching factor**, and for sequence alignment $b = 2^K - 1$, from whence we get the 7 successors in our example of the previous section. Realistic alignment problems typically involve at least 5 sequences, forcing

$b = 31$. This is quite large among applications of A*. Yoshizumi et al.[62] found a way to keep these nodes out of OPEN without sacrificing our treasured guarantee of optimality.

Associated with each node u in a Partial-Expansion A* (PEA*) search is a stored copy of the f-value called F, which may change as the search progresses. When u is expanded, each successor node v's F is initialized to $f = g + h$. If $u.\mathrm{F} + C \leq v.\mathrm{F}$, then v is discarded, and u is returned to OPEN with its F-value adjusted to the lowest (best) among all discarded successors v. C is a parameter to the search that controls how aggressively successors are discarded. When $C = 0$, PEA* adds only the best of the remaining successors of u to OPEN. When $C = \infty$, it adds all successors, just as A* does. This demonstrates that PEA* is a generalization of A*.

The degree of savings is affected by the ability of the heuristic h to discriminate between successor nodes that are on the optimal path to the goal and those that are not. In the best case, when $h = h^*$, PEA* can reduce storage used by the open list by a factor of up to b [62]. In real-world cases the savings are significantly less. Chapter 4 discusses experimental results in more detail.

2.3 Other Gap Models

Affine Gap

The linear gap function assigns to a gap the score $f(x) = gx$ where x is the length of the gap. The "affine gap" model c_{affine} assigns a score of $f(x) = a + bx$, where a is the **gap opening parameter** and b the **gap extension parameter**. Note that when $a = 0$ we have the linear gap model. More complex functions f are imaginable but tend to be computationally infeasible, as in fact is the affine gap model applied to multiple sequences [2].

Quasi-natural Gap

The "quasi-natural gap"[2] scoring method approximates the affine gap method using less effort. It is impossible to compute the affine gap score for an alignment for each column independently, as with the linear gap scoring method, since the score for a column depends on the length of the run of gaps it is a part of. The quasi-natural gap scoring method approximates the affine gap score, forming the score for each column by looking only at the previous column. For example, the following illustrates the distinct patterns of gap and letter that the quasi-natural gap scheme distinguishes between.

... `--` `-X`	... `X-` `XX`	... `X-` `-X`
start new gap	start new gap	start new gap
... `--` `XX`	... `?-` `?-`	... `?X` `?X`
extend gap	no gap	no gap

DP2Quasi-Iter(S_1,S_2,gi,ge,c)

$M \leftarrow$ special array$[0 \ldots \ell_1, 0 \ldots \ell_2, (\substack{-\\ \mathtt{x}}, \substack{\mathtt{x}\\ -}, \substack{\mathtt{x}\\ \mathtt{x}})]$

$M[0,0,\substack{-\\ \mathtt{x}}] \leftarrow \infty$; $M[0,0,\substack{\mathtt{x}\\ -}] \leftarrow \infty$; $M[0,0,\substack{\mathtt{x}\\ \mathtt{x}}] \leftarrow 0$

$M[0,1,(\substack{-\\ \mathtt{x}}, \substack{\mathtt{x}\\ -}, \substack{\mathtt{x}\\ \mathtt{x}})] \leftarrow M[1,0,(\substack{-\\ \mathtt{x}}, \substack{\mathtt{x}\\ -}, \substack{\mathtt{x}\\ \mathtt{x}})] \leftarrow (\infty, \infty, \mathrm{gi} + \mathrm{ge})$

for $i \leftarrow 0$ **to** ℓ_1 **do begin**

for $j \leftarrow 0$ **to** ℓ_2 **do begin**

if $i = 0$ **and** $j = 0$ **then** next loop

if $i > 0$ **then**

$$M[i,j,\substack{\mathtt{x}\\ -}] = \min(M[i-1,j,\substack{\mathtt{x}\\ -}] + \mathrm{ge}, M[i-1,j,\substack{-\\ \mathtt{x}}] + \mathrm{gi}, M[i-1,j,\substack{\mathtt{x}\\ \mathtt{x}}] + \mathrm{gi})$$

else

$M[i,j,\substack{\mathtt{x}\\ -}] \leftarrow \infty$

if $j > 0$ **then**

$$M[i,j,\substack{-\\ \mathtt{x}}] = \min(M[i,j-1,\substack{\mathtt{x}\\ -}] + \mathrm{gi}, M[i,j-1,\substack{-\\ \mathtt{x}}] + \mathrm{ge}, M[i,j-1,\substack{\mathtt{x}\\ \mathtt{x}}] + \mathrm{gi})$$

else

$M[i,j,\substack{-\\ \mathtt{x}}] \leftarrow \infty$

if $i > 0$ **and** $j > 0$ **then**

$$M[i,j,\substack{\mathtt{x}\\ \mathtt{x}}] = \min(M[i-1,j-1,\substack{\mathtt{x}\\ -}] + c(S_1[i], S_2[j]), M[i-1,j-1,\substack{-\\ \mathtt{x}}] + c(S_1[i], S_2[j]), M[i-1,j-1,\substack{\mathtt{x}\\ \mathtt{x}}] + c(S_1[i], S_2[j]))$$

else

$M[i,j,\substack{\mathtt{x}\\ \mathtt{x}}] \leftarrow \infty$

end

end

DP2Quasi-Iter $\leftarrow M$

Figure 2.11: An dynamic programming alignment algorithm for two sequences using the quasi-natural gap scoring scheme.

Two gap costs are used. When a column initiates a new gap, the cost assessed is gi. When an existing gap is extended through a column, the cost is ge.

The quasi-natural gap scoring method seems to be the least expensive one that can yield biologically acceptable results, though no study has conclusively compared the balance between speed and quality for these functions. For example, linear gap scoring formally induces a smaller search space for dynamic programming and A* search than quasi-natural, but in practice a search using quasi-natural gaps frequently searches a smaller space.

The quasi-natural gap is more difficult to program and was set aside for this research, the aim of which was to develop new heuristic and search techniques that would be applicable to both models. However, future research should be done using the quasi-natural gap, so it is introduced here. Figure 2.11 illustrates the more complex dynamic programming algorithm required to compute the optimal alignment of two sequences under the quasi-natural gap model.

The value assigned to a column containing a gap by the quasi-natural gap-scoring scheme depends upon the state of the previous column. However, we cannot know the state of the column that will ultimately be chosen. So, it is insufficient to store in $M[i,j]$ just the best score to align S_1 and S_2 up to through the first i, j letters. In order to extend the represented alignment, it is necessary to store the best score for each form the last column may take. So $M[i,j,\substack{\texttt{-}\\\texttt{X}}]$ contains the best possible score for an alignment of the first i letters of S_1 against the first j letters of S_2, such that the last column has a gap in the first row and a letter in the second row. That is, the alignment ending $\cdots \substack{\texttt{-}\\\texttt{X}}$. $M[i,j,\substack{\texttt{X}\\\texttt{-}}]$ and $M[i,j,\substack{\texttt{X}\\\texttt{X}}]$ are defined similarly. Since any legal alignment must end in one of these patterns, one of these must be the best-scoring alignment of $S_1[1\ldots i]$ and $S_2[1\ldots j]$ regardless of the form of the last column. Once M has been constructed, it is necessary to determine which of $M[\ell_1,\ell_2,\substack{\texttt{-}\\\texttt{X}}], M[\ell_1,\ell_2,\substack{\texttt{X}\\\texttt{-}}], M[\ell_1\,\ell_2,\substack{\texttt{X}\\\texttt{X}}]$ has the highest score overall and use a procedure similar to DP2Linear-Recover (Figure 2.3), starting from that cell. This clearly adds a factor of 2^K-1 space requirement to the dynamic programming table.

Leading and Trailing Gaps

When fragments from different but overlapping parts of a sequence are aligned, it is undesirable to impose a cost on portions at the end of one sequence that have no corresponding region in another, because this can distort the alignment of those portions which do correspond. An important feature of a practical aligner is that it can assign a 0 cost to gaps at the beginning (leading) and end (trailing) of a sequence. This feature was not vital to the aim of this work and was not implemented.

2.4 Summary

In this chapter we have introduced a mathematical notation for discussing sequence alignment in precise terms, and discussed the fundamental algorithms for exact alignment of biological sequences. The next chapter presents a broad survey of related work in sequence alignment.

Chapter 3

Related Work

A broad range of preceding work in sequence alignment is described, from the earliest papers to the most recent developments, including work both on exact sequence alignment and interesting ideas in approximate alignment.

3.1 Seminal Work

Fitch [12] first proposed a mathematical measure of the similarity between two amino acids based on the number of nucleotides in common (0, 1, 2, or 3) and a computer method for identifying regions of homology between two proteins, which was simply to compare all amino acid subsequences of length 30 from each sequence, without gaps and to count the number of mutations required.

3.1.1 Pairwise Alignment

Needleman & Wunsch

Needleman & Wunsch published the first paper describing a mathematical model of aligning two sequences that defined a score for each possible alignment and a procedure for finding the one with the highest score [40]. Their method used dynamic programming with a linear gap cost, and was basically that of DP2Linear-Iter from Chapter 2. It was first given mathematical treatment by Sellers [48]

Affine Gaps

Gotoh [15] recognized the need for a gap scoring function that would tend to minimize the number of gaps inserted into a pairwise alignment, but was less computationally demanding than the $O(\ell^3)$ algorithm of Smith & Waterman. He introduced a special-purpose $O(\ell^2)$-time algorithm for computing the affine gap cost, $g(k) = ak + b$ where k is the length of the gap, and $a, b \geq 0$ are the gap weights. In some literature this is called a linear gap weighting function, but this terminology is inappropriate. Properly speaking, a linear function $f(k)$

satisfies $f(k_1 + k_2) = f(k_1) + f(k_2)$ for all k_1, k_2. The affine gap weighting fails this test:

$$\begin{aligned} g(k_1 + k_2) &= a(k_1 + k_2) + b = ak_1 + ak_2 + b \\ g(k_1) + g(k_2) &= ak_1 + b + ak_2 + b \end{aligned}$$

Efficient Concave Gaps

Waterman [59] responded with an algorithm that he conjectured had effective complexity of $O(\ell^2)$ for most practical alignments using a concave downward scoring function, but was unable to prove its formal complexity. A concave downward function f is one which is strictly increasing, and the difference $f(x+1) - f(x)$ is never larger than $f(x) - f(x-1)$, intuitively meaning that the functions grows ever more slowly as it increases. Both the linear and affine gap score functions have this property. As far as the author is aware, no work extending the general result to more than two sequences has been successfully undertaken.

3.1.2 Multiple Alignment

Carrillo and Lipman

Carrillo and Lipman [5] recognized that the sums of scores of optimal alignments of covering subsets of sequences forms a lower bound on the cost of the optimal MSA. They showed that this bound allows one to ignore regions of the dynamic programming matrix with scores higher than the bound. This became known as the "Carrillo-Lipman bound".

The alignment program called simply `MSA` [35, 19] finds optimal alignments by pruning the dynamic table according to the Carrillo-Lipman bound. However, it is usually not efficient enough to find the optimal solution, and bounds the search using a very tight heuristic upper bound on the cost of the optimal alignment.

A* Algorithm

Ikeda and Imai [23] applied the well-known A* search technique from the field of artificial intelligence to the sequence alignment problem, using the pairwise heuristic. They showed that it examines fewer nodes than dynamic programming using the Carrillo-Lipman bound.

Kobayashi and Imai [26] formulated several schemes for combining optimal alignments of subsets of sequences to form an A* heuristic. They further developed a method that employs a heuristic upper bound to guide the computation and storage of only those cells necessary for the search. Finally they propose a procedure to compute the heuristic function using a recursively-invoked A* search.

Lermen and Reinert [43] implemented an A*-based aligner and described heuristics that effectively limit the size of the search space by confining the search to stay nearby the optimal paths in the pairwise alignment tables. This sacrifices the guarantee of optimality. They further showed that using a three-way alignment (like pairwise but with dynamic programming tables over three sequences rather than two) in the A* heuristic can considerably improve its search performance, though they fail to propose a compact representation of the table that is efficient in both space and time while also being useful in searches that guarantee optimal results. This failing is corrected in the current work.

3.2 General Search Algorithms

Iterative-Deepening A*

Iterative-Deepening A* (IDA*) search [27] is similar to A*, in that it yields optimal results given an admissible heuristic, but uses neither closed nor open lists. It sets a cutoff C (typically $h(\text{root})$) and searches depth-first from the root until it finds a solution, refusing to expand any node u with $f(u) > C$. If it finds no solution, it adjusts C upwards and starts over. By refraining from storing open and closed lists, IDA* is capable of solving problems using linear space. However, IDA* is not useful for sequence alignment because it is too slow. Each node in the search lattice can be reached by an exponential number of paths, and since it does not record visited nodes in the open or closed lists, IDA* is unable to avoid traversing the exponential number of paths having $g \leq C$.

Stochastic Node Caching

Miura and Ishida introduced Stochastic Node Caching(SNC) search [38] and evaluated its performance on multiple sequence alignment. SNC takes advantage of the memory that IDA* with its small memory requirements frequently leaves available by augmenting it with a cache of visited nodes. If a node v is in the cache of visited nodes, then the search can refrain from traversing its children again. If there is space available to store only a small proportion of nodes, then SNC is unable to sufficiently mitigate the number of nodes revisited by IDA*, so its use is limited. Researchers in computer games had already described an equivalent approach under the name of "transposition table".

Linear-Space Best-First Search

Korf described the linear-space best-first search (RBFS) [28] technique, which uses space linear in the solution path length, at the expense of repeated reevaluation of nodes. There is little mention in the literature of the performance of RBFS on multiple sequence alignment. It could be expected to be poor due to excessive node revisiting, like IDA*, but it is conceivable that techniques like SNC could improve it. Miura and Ishida mention that RBFS is indeed compatible with SNC, but did not run experiments.

Robnik-Šikonja [45] described an enhancement for RBFS that uses a more sophisticated approach than SNC to taking advantage of available memory. Results of its application to the MSA problem are absent from the literature, and it would be worth examining.

Divide-and-Conquer Frontier Search

Divide-and-Conquer Frontier Search (DCFS) was introduced by Korf & Zhang [32] for the MSA problem, and has a similar character to Hirschberg's algorithm. DCFS modifies A* search by dispensing with the closed list but keeping the open list. Node revisitation is not a problem if a consistent heuristic is used. With just the open list of A*, it is impossible to reconstruct the optimal path from the start node to goal node. To reconstruct the path, DCFS designates a midpoint cutting the search space, and stores with each node the point where its predecessor crossed that midpoint. Once the goal is reached, the search space is divided in half at the goal's midpoint, and a search is run recursively on each half. However, for MSA problem instances, the closed list is rarely larger than the open list, and so DCFS offers weak performance improvements by dispensing with the closed list.

Partial-Expansion A*

Since the open list can be much larger than the closed list, Yoshizumi, Miura, and Ishida [62] recognized that there should be more profit in reducing the size of the open list. Partial-Expansion A* (PEA*) discards a child node v of a parent u if they satisfy $f(v) > f(u) + C$, where C is a tunable parameter. The parent node's f-value is modified to that of the lowest (best) f-value over all children that are discarded, and it is put back on the open list. The time and space that would be spent putting unpromising nodes on the open list is saved, and correctness is maintained since the value of an unpromising node is represented by its parent. PEA* offers significant space savings on problem instances containing large numbers of highly **homologous** (similar) sequences, but its ability to save space is tied to the accuracy of the heuristic used, and it becomes much less useful as the problem increases in difficulty. Yoshizumi et al. report problem instances where PEA* stores only 4.7% of the nodes that would be stored by regular A*, but our experience indicates that 30% is more typical for problem instances that are too hard to compute by hand. Results are presented in Chapter 4.

3.3 MSA-Specific Exact Search Algorithms

3.3.1 Three Sequences

Algorithms for aligning three sequences were developed as computers became powerful enough to make them feasible.

Gotoh[16] described an interesting procedure for computing an exact three-sequence MSA. He computes and thens discard the dynamic programming matrix, storing only the previous and current planes. The information required to construct the alignment is represented in a special graph G. Vertices in G correspond to cells of the table that are at the beginning or end of a run of insertions/deletions or matches.

3.4 Evaluations and Comparisons

Altschul [2] compared and evaluated existing gap cost schemes for multiple sequence alignment, with attention to the minimum time and space complexities required by an exact algorithm that implemented them. He then introduced the quasi-natural gap cost, which provides a good balance between computational cost and biological usefulness for exact alignment algorithms.

McClure et al.[37] composed a small collection of reference alignments and used it to test several alignment programs. This reference set was used by later authors, including Lermen and Reinert [43], to evaluate their own work against competing alignment strategies. This reference set's disadvantage is its small size.

Vingron and Argos [56] proposed a method for determining the biological significance of an aligned region between two proteins. This is an important issue when making comparative evaluations of sequence alignment methods.

Thompson et al. have provided a large series[54] of test cases called BAliBASE, composed of sets of expertly aligned sequences, using proteins with known secondary structure and even three-dimensional conformation. In order to test the ability of different aligners to cope well with varying conditions, they created five types of reference alignments designed to exhibit a wide variety of characteristics, with alignments of varying difficulty in each set. Each reference alignment also indicates regions that are not reliably aligned. Thompson et al. evaluated many popular sequence alignment tools against their reference alignments in order to characterize their strengths and weaknesses, and have continued to grow the reference test set since their initial publication. Unfortunately, detailed results of running times and the exact alignments produced by each program on their set are not available in the literature, though PRRP [17], CLUSTALW [53], and SAGA [41] led the ranks of alignment quality.

3.5 Complexity Analyses

The MSA problem is NP-complete for a special case of the scoring matrix, according to the results of Wang and Jiang [57]. Their result was generalized by Just [25] to show that it is NP-hard on a class of scoring matrices that includes those actually used in biological applications.

3.6 Approximation Algorithms

3.6.1 Progressive Alignment

The progressive alignment approach begins by constructing a **phylogenetic tree** representing the closeness of the relationships between the sequences. Constructing a phylogenetic tree is a challenging research problem in itself. One simple method is to align each pair of sequences to obtain a measure of their similarity and to build a tree by making sibling leaves out of related sequences. However the tree is constructed, the process continues by choosing sibling leaves, aligning them, then replacing them in the tree by a pseudo-sequence representing the aligned pair. The procedure continues until only one node is left in the tree. The most prominent implementation of this approach is CLUSTALW [53]. The PRRP method mentioned above [17] is another variant. A major disadvantage of progressive alignment is that errors in early alignments cannot be corrected in later stages. T-Coffee [42] attempts to mitigate this disadvantage by using information from other global and local alignments to guide the progressive alignment.

3.6.2 Divide-and-Conquer Alignment

Divide-and-Conquer (DCA) alignment [51, 52, 44, 55] is an interesting approach to approximate alignment reminiscent of Hirschberg's algorithm. DCA guesses a point in the lattice on the optimal path, splits the sequences at that point, and recurses. On sequence fragments that are short enough, it invokes an exact aligment algorithm. Once all fragments are aligned, they are concatenated to yield an approximate alignment. This method finds the split point by placing it at the halfway mark of the longest sequence, then searching a special form of the pairwise-alignment dynamic programming tables for a set of coordinates with minimal score. DCA does not take direct advantage of heuristic procedures for finding the split point that use information such as the presence of strong motifs in subsets of the input sequences.

Generalized Dynamic Programming (GDP) [18] is similar to DCA. The difference is that GDP does not commit to only one set of split points. It uses local and global approximate alignments to generate a large number of plausible "anchor points" aimed at the optimal path through the lattice. It then performs dynamic programming over the anchor points themselves using a progressive alignment to compute the score to move from one anchor point to the next. Gracy and Sallantin [18] claim successful alignment of up to 18 sequences with GDP, with results superior to CLUSTALW. We are not aware of any publications extending this work.

3.6.3 Other Approaches

Traveling Salesman Problem

Gonnet et al. [33, 14] propose an approximate method that quickly finds an MSA by a traveling salesman tour of an unknown phylogenetic tree, with a guarantee that the resulting MSA will have a score within a factor of $\frac{K}{K-1}$ of optimal.

Genetic Algorithm

Notredame and Higgins [41] proposed and implemented an MSA using a genetic algorithm that mutates and breeds a population of alignments. The claimed advantage of this approach is that it can be used to find good approximate alignments for any conceivable type of scoring function, where it was possible previously only to optimize with respect to simple, unrealistic functions such as the linear gap cost. This implies that it will be a useful tool for experimenting with new scoring functions. The major disadvantage of the genetic algorithm is that the intermediate alignment can become caught in a "local-minimum" and fail to find the true global minimum.

Composition of Optimal Alignments

Gusfield [20] was the first to describe an algorithm that computed an approximate alignment with score guaranteed to be within a factor of 2 of the true optimal score, with a high probability that it will do much better. He used pairwise optimal alignments, merged in a star pattern.

Bafna et al. [3] generalized this result. They showed that given a set of K sequences and all optimally aligned subsets Q of size $L < K$, it it is possible to compose Q, using a star pattern, into an alignment that has a score within a factor of $2 - L/K$ of optimal, with performance polynomial in ℓ and K, given fixed L.

3.7 Summary

We have examined a broad range of previous work in sequence alignment. The next chapter describes the author's novel contribution to the field: the octree heuristic.

Chapter 4

The Octree Heuristic

We start this chapter by discussing how a heuristic function h for an A* sequence alignment search may be composed from a collection of dynamic programming tables for subsets of sequences. We then introduce the main result of the thesis, which is the octree, a structure for compactly representing three-dimensional dynamic programming tables for use in the heuristic. We discuss how to tune the octree parameters and how to select the sequences for use with the octree. Finally, we discuss the performance tradeoffs of using the octree versus using only two-dimensional dynamic programming tables, or storing entire three-dimensional dynamic programming tables.

4.1 Introduction

4.1.1 Aim of research

The aim of this research was to develop new heuristic search techniques for optimal multiple sequence alignment (MSA) that could be incorporated into a practical software tool of use to researchers in the biological sciences. Effort was aimed particularly at developing novel admissible heuristics for exact alignment with A* search. Attention was not focussed on developing generally-applicable search techniques in the vein of DCFS [29] or PEA* [62].

4.1.2 Implementation

A full-featured exact alignment program was written in C++ using partial-expansion A*, a wide variety of diagnostic features, facilities to visualize the progress of the search, and rich facilities to support experimenting with heuristics. The work approached 9000 lines of C++ code.

4.1.3 Motivation: Using Three-Dimensional Tables

We found a practical way to construct a heuristic from dynamic programming tables for three sequences. The number of nodes examined and stored by A* search is almost completely determined by the heuristic. Minor but negligible variations may be due to the order in which nodes with the same f-value are expanded. A heuristic composed of 3D tables will perform no worse than a heuristic made only with 2D tables, and may perform much better. The difficulty is that the 3D tables can be prohibitively large. For sequences with a typical length of 500, a single such table would be $500^3 \times 4 \approx 500$ megabytes, assuming 4-byte integers. This is essentially all of the storage available on a typical well-equipped PC. Some way of compressing these tables needed to be found.

It is conceivable that a heuristic could be constructed using dynamic programming tables of four sequences or more, but the problem of mitigating these tables' even larger storage demands is not addressed in this work.

That a heuristic made of 3D tables can do no worse than a heuristic of 2D tables was observed by Carrillo and Lipman [5]. They observed that in the general case, a lower bound on the cost of any alignment $\mathcal{A} = \bar{A}(S)$, where $S = \{S_1, \ldots, S_K\}$, is

$$c(\mathcal{A}) \geq \sum_{T_i \in T = \{T_1, \ldots, T_H\}} c(\bar{A}(T_i)) \tag{4.1}$$

where $T_i \subseteq S$ and for each pair of distinct sequences S_j, S_k, there is *exactly one* T_i such that $\{S_j, S_k\} \subseteq T_i$. H is the number of sets of sequences, and depends on their sizes. In the extreme case that each set contains only two sequences, then $H = (K^2 - K)/2$. At the other extreme where each set contains all K sequences, there is only set, $H = 1$.

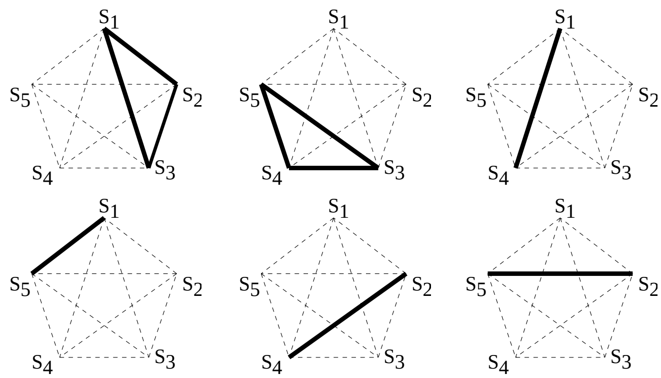
This means that we can estimate the optimal score for the alignment of a set of sequences S by taking the sum of scores of alignments of subsets T_i of S where each pair of sequences appears together in the same T_i exactly once. Figure 4.1 shows an example. Students of graph theory will recognize that this amounts to generating a covering of a complete graph with cliques.

As discussed previously, for each node u the A* search examines, it requires a lower-bound estimate of the cost of the shortest path to reach the goal node t from u. As we have seen, a cell of the dynamic-programming matrix contains the cost of the shortest path from the source s to u for all cells u. Therefore, the dynamic-programming matrix for the reversed sequences provides the required cost from u to t. In Chapter 2, Figure 2.10, we saw how to construct a heuristic from several 2-dimensional dynamic programming tables. Equation 4.1 tells us that we could use 3-dimensional or higher dynamic programming tables as well. In the next section we examine the octree, a data structure for storing a 3-dimensional dynamic programming table for use in a sequence alignment heuristic.

4.2 The Octree

The octree is a data structure that can compactly represent a 3-dimensional dynamic programming table, storing only those portions that are needed by the search, and efficiently

$T_1 = \{S_1, S_2, S_3\}$ $T_2 = \{S_3, S_4, S_5\}$ $T_3 = \{S_1, S_4\}$



$T_4 = \{S_1, S_5\}$ $T_5 = \{S_2, S_4\}$ $T_6 = \{S_2, S_5\}$

Figure 4.1: A composition of $S = \{S_1, S_2, S_3, S_4, S_5\}$ into heuristic structure $T = \{T_1, T_2, T_3, T_4, T_5, T_6\}$. Each pair of sequences is represented exactly once.

generating omitted portions as-needed. Kobayashi and Imai [26] described a scheme for generating in an efficient way only those table cells demanded by the search. However, their method required an accurate upper bound on the cost to reach the goal in order to be effective. The octree method does not require an upper bound.

4.2.1 Basic Structure

For sequences of significant length, the dynamic programming matrix is small enough for modern computers to generate quickly, but requires too much storage ($O(\ell^3)$ for strings of length ℓ). Our goal is to create an effective heuristic to use in the A* search, but at the same time we would like to reserve as much space as possible for the open list. The octree efficiently represents the three-dimensional (3D) dynamic programming tables of 3-way alignments. It stores only those portions of the table that are needed by the search, and efficiently regenerates omitted portions as they are needed. Octrees are typically used in CAD, GIS, fluid dynamics simulations, and other 3D image processing and simulation applications to compress grid representations of space containing large uniform volumes.

The octree is a tree data structure that represents a 3D space using rectangular blocks (Figure 4.2). The octree is constructed from three types of nodes:

Internal node Contains no dynamic programming table cells in itself. Has eight children, each of which represents one octant (corner) of its parent's space in more detail.

Full leaf node Contains all of the values of the dynamic programming matrix for the portion of the space it represents.

Empty (face) leaf node Contains only the values over the surface of the volume, so that the contents may be regenerated as needed.

Obviously, the root node represents the entire table. The pseudo-code for octree construction is in Figure 4.3.

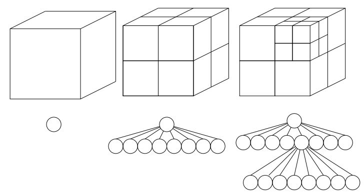


Figure 4.2: Octree Data Structure.

At the start of the A* search, the octree contains only internal nodes and 'empty' leaf nodes. As table values are requested by the search, 'empty' leaves containing the queried position are converted to internal nodes with new children. Children for areas beneath a threshold volume τ are created as 'full' nodes, and those above, as 'empty' nodes. 'Full' children can answer queries without further computation.

The octree structure can be generalized to any number of dimensions. However, the two-dimensional case, known as a **quadtree**, is of little use because the space used by pairwise DP tables is low to begin with. Higher dimensional versions are excessively large to compute and store. The four-dimensional analogue of the octree would have to store full 3-dimensional DP tables over the 'surfaces' of its empty leaf nodes.

4.2.2 Quadtree Example

I won't offer a worked example of an octree because it is awkward to write down three-dimensional tables of numbers on paper while preserving a sense of their spatial relationships.

Instead, I use the two-dimensional analogue of the octree, called the quadtree. Conceptually the two are identical. While reading this example, refer to the octree code in Figure 4.3.

Refer to the table T in Equation 1.5 of Chapter 1. T is the dynamic programming table for the alignment of sequences s_1 = `GATAGTC` and s_2 = `AGGTGCA`, scored using -2 for a match, $+1$ for a mismatch, and $+2$ for a gap. Recall from Chapter 2 that T could form part of the heuristic (Figure 2.1) used by an A* search for an alignment of reverse(s_1) and reverse(s_2) with some other sequences. The example will show what T looks like in quadtree format, and how its structure evolves to satisfy a request for a table entry from the A* search. We will use a full leaf threshold τ of 1 to make the example clear, though in practice 1 is far too small. A later section discusses how to choose τ for best performance.

At the beginning, in Figure 4.4, T consists of just one empty leaf node. The left part shows the table structure of T. Only the first row and first column are present. The rest

```
    Lookup(node,x,y,z) : integer
1:      if node is Internal then
2:          // Look in the child that contains (x,y,z)
3:          octant o ← octantFor(node.bounds,x,y,z)
4:          return Lookup(node.child[o],x,y,z)
5:      else if node is Full Leaf then
6:          return node.table[x][y][z]
7:      else // node is Empty Leaf
8:          Node n ← new Internal(node.bounds)
9:          addChildren(n)
10:         Fill n.child[].table using node.surface
11:         Replace node with n
12:         return Lookup(node,x,y,z)
        end if
    end Lookup

    addChildren(node):
13:     foreach o of node.bounds
14:         if volume(o) > τ then
15:             node.child[o] ← new EmptyLeaf(o)
16:         else
17:             node.child[o] ← new FullLeaf(o)
            end if
        end foreach
    end addChildren
```

Figure 4.3: Pseudo-code for Octree Construction

of the table is empty. The tree structure of T is on the right. It consists of only one node *node*.

Now suppose that the A* search using T for its heuristic requires the value at $(3,3)$. The "0" in the top left corner is at $(0,0)$, so $(3,3)$ is the entry underneath the "6" in the top row and to the right of the "6" in the left column. The request results in a call to Lookup($node, 3, 3$) in Figure 4.3. Since *node* is an empty leaf, control moves to line 8, where a new internal node a is allocated to take its place, and its children are constructed by a call to addChildren(a). The volume of each child is 16, which is greater than $\tau = 1$, so all children are created as empty leaf nodes. Their first rows and columns are filled in back at line 10 after addChildren returns. This requires generating the full contents of T by dynamic programming, and all table cells unneeded by the children are discarded. Finally, *node* is discarded and replaced by a.

Figure 4.5 shows the result. The dotted square around the four empty tables on the left represents the root internal node a, also drawn as a dotted square, on the right.

None of the empty leaf nodes yet contains the value for $(3,3)$. It would have to have been available when the contents of T were regenerated to furnish the values for the four

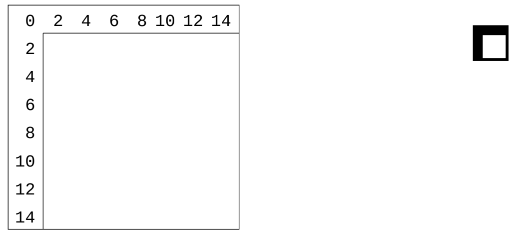


Figure 4.4: Quadtree Example 1. The left part shows the table structure, and the right part shows the tree structure.

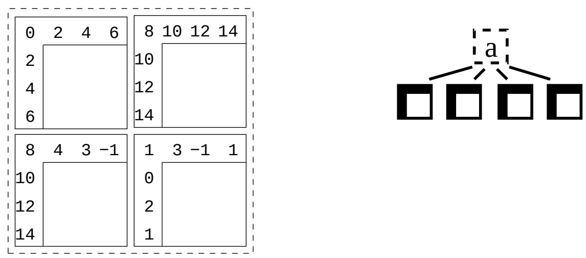


Figure 4.5: Quadtree Example 2

empty leaf nodes in the figure above and could have been returned at that time. However, the goal of the quadtree (and octree) is to store pieces of T that are near past requests, with the expectation that future requests for values will be nearby.

Control moves to line 12, where Lookup$(a, 3, 3)$ is called recursively. In this invocation of Lookup, the variable *node* (with value a) is an internal node, so control moves to line 3. The top-left table of the above figure contains the coordinate $(3, 3)$. For this example, the corresponding node in the tree representation is the leftmost of the four empty leaf nodes for this example. Lookup is recursively called on this node. Since it is an empty node, and its childrens' volumes each have volume 4, it is replaced with a new internal node b having four empty leaf node children. Figure 4.6 shows the result.

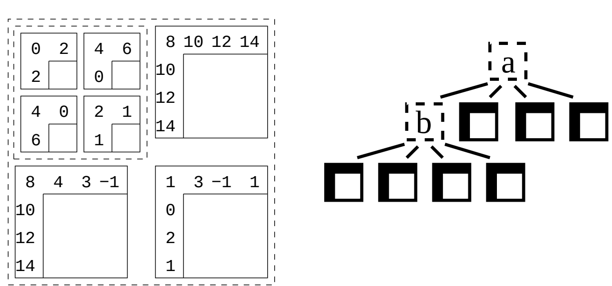


Figure 4.6: Quadtree Example 3

Control resumes with a recursive call to Lookup$(b, 3, 3)$. The coordinate $(3, 3)$ falls into the bottom-right table of b, its right-most child in the tree representation.

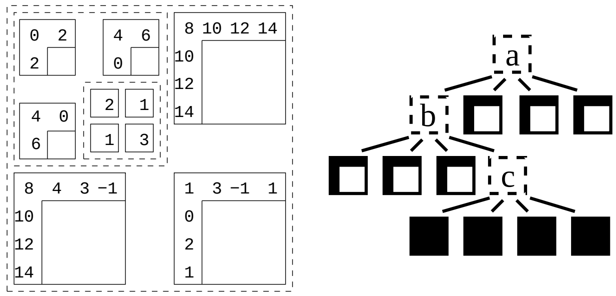


Figure 4.7: Quadtree Example 4

This time, the call to addChildren$(c, 3, 3)$ finds that the volume of each child of c is only 1 cell. The comparison on line 14 is therefore false, and the children of c are created as full leaves on line 17. The tree reaches its final form for the this request in Figure 4.7. Lookup is called on the rightmost child of c. Since it is a full leaf, control moves to line 6 and the value 3 is returned from its table. The stack of recursive calls to Lookup unwinds, and finally the value 3 is returned to the A* search. If the A* search should later require the values at cells $(2, 2)$, $(2, 3)$, $(3, 2)$, they are immediately available.

A remark on the implementation of the octree for this dissertation is necessary. In this presentation, portions of T were regenerated and discarded several times to satisfy one initial call to Lookup. This simplification of the code was for expository purposes. My implementation builds the entire tree structure at the beginning of a request, and fills in the new leaf nodes at all levels with one regeneration of T.

4.3 Experimental Data Sets

In this work we experimented with the threshold parameter τ of the octree, examined the effect of the cutoff parameter C with partial-expansion A* search, and examined the heuristic selection procedure. We also compared the overall performance of searches using the octree with other heuristics.

For all experiments we used sequences drawn from a set of 74 potassium ion channel sequences (K^+) provided by Dr. Warren Gallin at the University of Alberta. The advantage of this set is that the sequences have widely varying degrees of homology, so that it is easy to compose a subset exhibiting nearly any desired degree of alignment difficulty. The 74 K^+ sequences range in length from 377 to 955 characters long, and are named Ch01 through Ch76, with Ch05 and Ch72 not present. Alignments on this sequence set are biologically interesting because the proteins all have similar functions.

For experiments related to tuning the performance of the octree, only a few distinct sets of sequences drawn from the K^+ set were used. The three sets were $E_1 = \{$Ch01, Ch02, Ch03, Ch04$\}$, $E_2 = \{$Ch01, Ch02, Ch03, Ch04, Ch06$\}$, $E_3 = \{$Ch01[1 . . . 300], Ch02[1 . . . 300], Ch03[1 . . . 300], Ch04[1 . . . 300], Ch06[1 . . . 300]$\}$. That is, E_3 uses only the first 300 characters of the sequences from E_2, because of the difficulty of aligning their full lengths using weaker heuristics. The sequences of E_2 have lengths 580, 603, 659, 376, 835, respectively.

All experiments were run on "ohaton", a Sun E/420 with 4 GB of RAM and 4 CPUs running at 450 MHz.

4.4 Selecting the Leaf Type Threshold

The octree stores leaves as empty when they are above a certain threshold volume τ, and as full when they are smaller. It was found that the threshold is not critical to performance. There is a very wide range of values that perform equally well. However, some are slightly better than others.

Experiments were run on several sets of sequences. Each alignment was repeated several times using the best octree heuristic (Section 4.5). Values of τ tested were 10, 100, 500, 1000, $10,000$, $100,000$ and $1,000,000$. All the sequence sets I examined displayed similar behavior, so I only describe the results for E_2.

We wanted to answer the following questions:

1. How many cells are stored in each type of node at the end of the search?

2. How many distinct cells are queried in the octree?

3. An empty leaf, even though it is hollow, can satisfy queries that land on one of its three planes. So, how many queries to the octree are satisfied by each type of node over the course of the search?

4. How many cells does the octree compute, and recompute, over the course of the search. How much time does it spend doing this?

5. Not all space allocated for an octree is taken up by cells of the DP table. How many bytes are actually allocated?

The number of cells that are actually accessed is a function purely of the search space and is independent of τ. It is an interesting quantity because it tends to be much lower than the number of cells stored for any value of τ, indicating that even more space-efficient data structures could be developed.

Figures 4.9 and 4.8 show the number of each type of octree cells stored at the end of the search, and the proportion that each type of node satisfied, answering questions 1, 2,

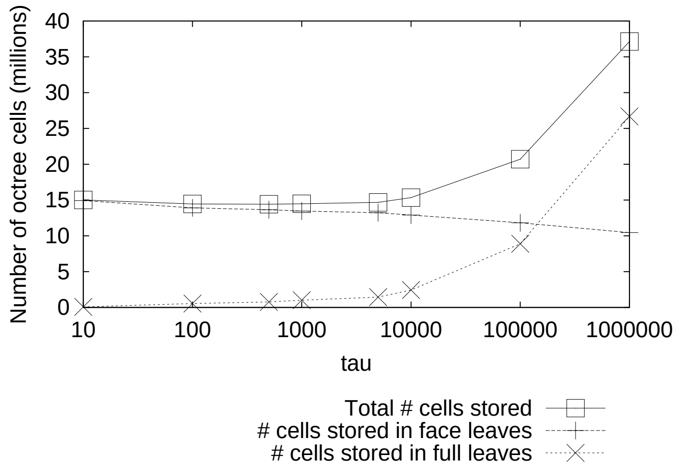


Figure 4.8: Total number of cells stored in octrees during the alignment search on E_2.

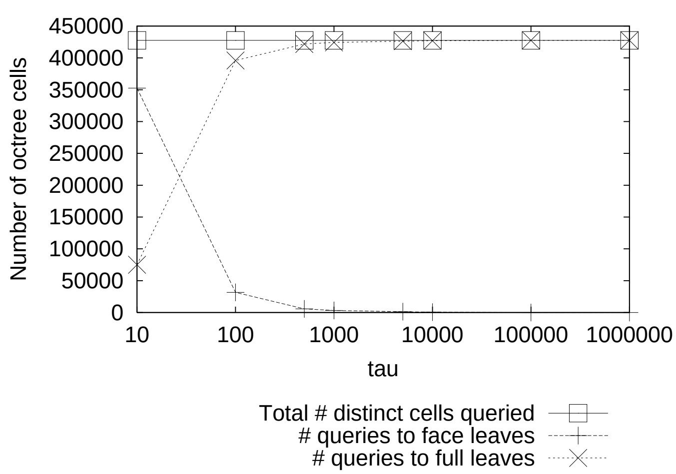


Figure 4.9: Number of distinct octree cells accessed over the course of the alignment search for E_2, and how they were satisfied.

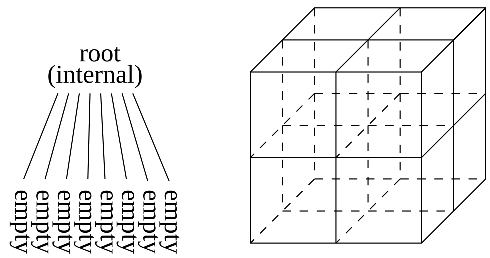


Figure 4.10: A one level deep octree with empty leaves

and 3. We can see in Figure 4.8 that for most values of τ, the vast majority of cells stored are actually in empty nodes. The proportion stored in full nodes begins to really increase only above $\tau = 100\ 000$, and only does so at the expense of a drastic increase in the total number of nodes stored. Figure 4.9 shows that the total number of nodes actually accessed is vanishingly small compared to the number stored. This is because the number of cells on the originating surface of the dynamic programming table is approximately $3\ell(S)^2$, and the number stored will tend to increase as the tree deepens. For example, if the structure of the tree is as in Figure 4.10, the total number of cells stored on the faces of the empty leaves is $8 \times 3 \times \ell(S/2)^2 = 6\ell(S)^2$. For $l(S) \approx 500$ this is 1500000. Adding one level to the tree doubles the number of cells stored in empty leaves.

This large difference in the space taken by each type of node marks a sharp contrast to their usefulness to the search. The vast majority of accesses are satisfied by full leaf nodes, except for very small values of τ. When $\tau = 10$, corresponding to a cube with side lengths around 2 or 3, at which size a full cube contains barely more cells than its surface. Figure 4.9 shows that when $\tau = 10$, most accesses are satisfied by the surfaces of empty leaves.

The experiments show that empty leaves account for most of the memory used by the octree, but that most queries are satisfied by full leaves. Further, most stored cells are never accessed at all. This pattern is consistently followed by other sequence sets, regardless of their alignment difficulty.

Questions 1, 2, and 3 are revealing of the weaknesses of the octree and the potential for its improvement, but to select τ requires attention mainly to questions 4 and 5: how much space and time is used? Larger values of τ will tend to decrease the amount of time spent recomputing portions of the octree, at the expense of increasing memory usage, while also decreasing the amount of time spent traversing the tree to reach the leaf nodes. Very small values of τ will also tend to increase memory usage as a result of overhead from many very small nodes. Therefore the optimal τ should be moderate. Figure 4.11 shows the number of bytes allocated for the octree data structures at the end of the search. We can see that it dips to a minimum between τ values of 1000 and 5000. Figure 4.12 shows that the total time taken by the search varies surprisingly little, and the reason is that recomputation of

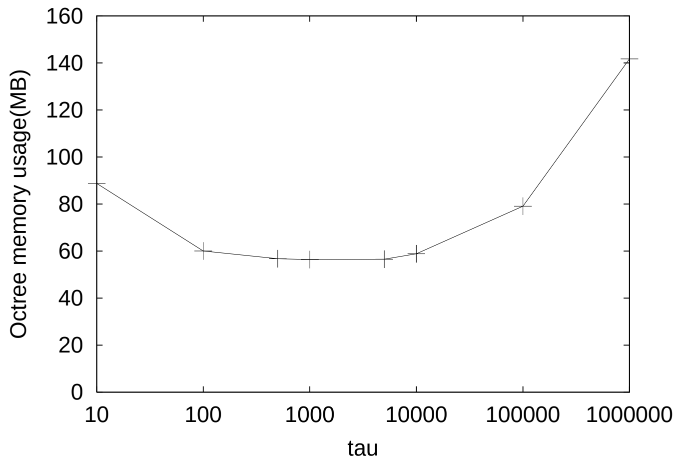


Figure 4.11: Total number of bytes allocated in octree data structures during the alignment search on E_2 for various values of τ.

small volumes takes very little time compared to the initial computation of the dynamic programming table. The extra time taken to access leaves deep in the tree is also negligible compared to the many other tasks the search must undertake when generating a node.

The total number of nodes initially computed is constant w.r.t. τ, but Figure 4.13 shows that the number of them that are recomputed over the course of the search is also fairly uniform. The reason is that most of the time is taken by the recomputation of large volumes at the beginning of the search. The additional recomputations forced by small values of τ are in very small areas, and as we saw from Figure 4.9, these are few in number.

Some experiments described in this chapter were run using a value of $\tau = 1000$ ($10 \times 10 \times 10$ cube), and some with $\tau = 10,000$ ($\approx 22 \times 22 \times 22$ cube) . Preliminary experiments indicated that the latter value was good, and on this information many time-consuming experiments were run. Later, the preceding experiments showed that $\tau = 1000$ was slightly better. The choice makes only a small difference to the performance, so it was decided not to re-rerun those experiments that had used $\tau = 10,000$.

There are other attributes of the octree that could be examined: the actual number of empty and full leaves, as opposed to just the number of cells they contained in aggregate. We have not discussed how to decide what the initial depth of the octree should be at the beginning of the search.

4.5 Heuristic Selection

The value provided by the heuristic must be composed of values corresponding to alignments including each pair of sequences exactly once (Equation 4.1, Figure 4.1). If we are using only pairwise alignments to construct the heuristic, there is only way to do it. With octrees,

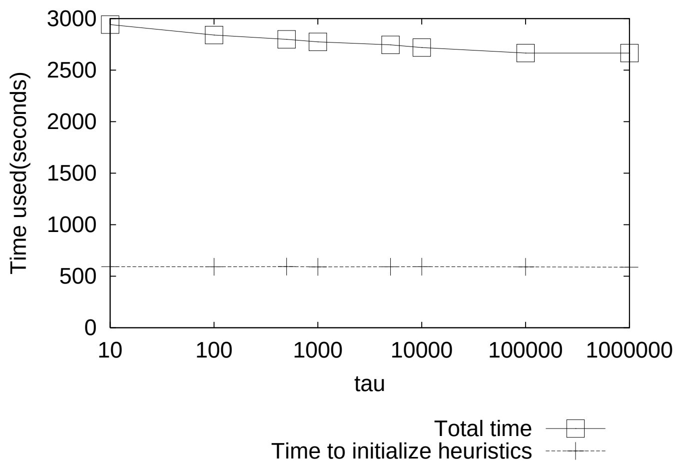


Figure 4.12: The top line is the total CPU time taken by the search on E_2 for various values of τ. On the bottom is the amount of time taken at the beginning of the search to construct the octrees. This line is expected to be flat because the same octree is constructed in each case.

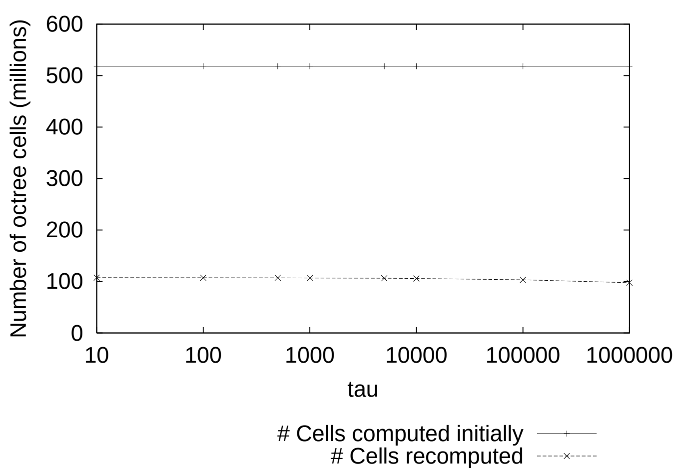


Figure 4.13: Number of cells computed and recomputed during the search on E_2 for various values of τ.

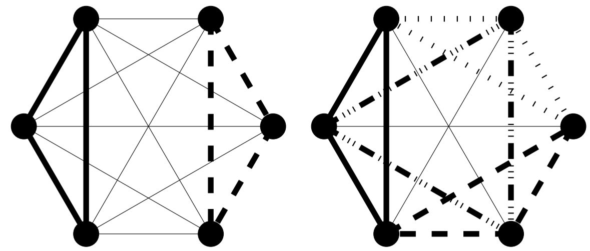


Figure 4.14: The two 3-cliques on the left form a **maximal** compatible set since no more 3-cliques can be added, but the set of four on the right is **maximum**, since no set of compatible 3-cliques is larger.

we must select triplets of sequences that do not have an overlapping pair between them, and there may be several choices. Some choices are better than others. We wish to choose well and quickly. Lermen and Reinert [43] discussed this problem but did not verify the effectiveness of their proposed solution with experiments. It is simply stated: if several heuristics h are available, choose the one with a maximum value $h(s)$.

For example, a heuristic for a four-sequence problem instance $S = \{S_1, S_2, S_3, S_4\}$ can be formed using the triplet $\{S_1, S_2, S_3\}$ and the remaining pairs $\{S_1, S_4\}$, $\{S_2, S_4\}$, and $\{S_3, S_4\}$. Clearly there are four distinct ways of constructing a heuristic for S using optimal 2-way and 3-way alignments: there are four choices of the sequence that will be excluded from the triplet, and this determines the rest. For five sequences there are 15 **maximal** (Figure 4.14) combinations. To see this, note that since there are only five sequences and we must form two sets of three that don't share a pair in common, exactly one sequence must participate in both. Once that sequence is chosen, there are $\binom{4}{2}/2 = 3$ ways to divide the remaining sequences into 2 groups, for a total of 15 unique combinations.

The number of ways of decomposing K sequences into sets of size 3 and 2 satisfying the restriction of Equation 4.1 explodes combinatorially with K, but for the relatively small values of K we are concerned with it is feasible to consider them all. The problem reduces to finding all maximal cliques in a certain undirected graph with $\binom{K}{3}$ nodes. The structure of this graph is $G = (V, E)$ where V is the set $\{v : v \subseteq S, |v| = 3\}$ and $E = \{(v_1, v_2) : |v_1 \cap v_2| \leq 1\}$. That is, a pair of vertices is connected by an edge if the corresponding set of three sequences shares no more than one sequence in common. For example, Figure 4.15 shows such a graph for 5 sequences.

A clique in this graph represents a set of triplets that are pairwise compatible, i.e. no two of them share two sequences. We are only concerned with finding maximal cliques since it never hurts the heuristic to add another if there is room. To find the best maximal clique we must consider them all, and the number of them grows rapidly with the number of sequences. For up to eight sequences, it is tractable to consider them all. In order to determine which combination of triplets is best, it is necessary to first evaluate $c(\bar{A}(\{S_i, S_j, S_k\}))$ for each

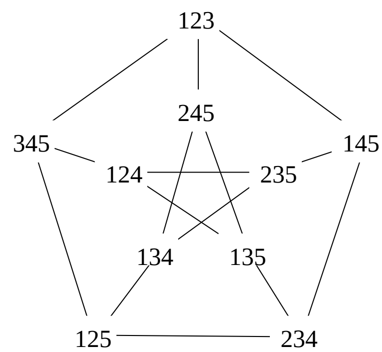


Figure 4.15: The graph of compatible triplets from 5 sequences.

$K = \|S\|$	$\binom{K}{3}$	# cliques	# Maximal cliques	Size of maximum clique	Time to compute
3	1	2	1	1	fast
4	4	5	4	1	fast
5	10	26	15	2	fast
6	20	271	40	4	fast
7	35	5596	450	7	fast
8	56	231577	16200	8	6 seconds
9	84	21286940	834960	12	17 minutes

Table 4.1: The growth of the selection problem with K. Times quoted are for an Intel Pentium-III 500MHz with a naïve backtracking program to count all cliques.

set of triplets $\{S_i, S_j, S_k\} \in S$. Table 4.1 lists the number of these as $\binom{K}{3}$. We also need $c(\bar{A}(\{S_i, S_j\}))$ for each pair $\{S_i, S_j\} \in S$. Finding the best combination of triplets then requires adding up the sums of the corresponding $c(\bar{A})$ for each maximal combination of triplets. Table 4.1 shows the number of triplets there are to choose from, and how many valid combinations of them there are for a given number of sequences.

For our purposes it is plausible to evaluate all 16200 maximal combinations of triplets for 8 sequences, but for larger K, a heuristic method to select the heuristic would have to be developed.

The simplest way of selecting the heuristic h is to identify the one which yields the highest score at the root node s, that is, the one that has maximum $h(s)$. Note that where not stated otherwise, all experiments in this chapter use this procedure.

This makes sense because the heuristic h_{p} that uses only pairwise alignments must have the lowest score among all possible heuristics due to Equation 4.1. Intuitively, it should also be the worst because it encodes the least information about the constraints between the sequences. Conversely, the highest-scoring h ought to be the best, since, by Equation 4.1 again, the very best possible heuristic gives the true value for the cost from u to t. In this section we experimentally verify this conjecture by running a search for each possible heuristic structure T, and compare performance.

Experiments described in this section were performed on set E_3. It was not possible to use E_2 because some of the weaker heuristics could not align it in the space available on the test machine.

As Table 4.1 shows for $|S| = 5$, there are 26 distinct heuristics, including maximal sets of two triplets and non-maximal sets involving only one or zero triplets. They are presented here without distinction between maximal and non-maximal.

The score of a heuristic h at the root node is simply equal to $\sum_{T_i} c(\bar{A}(T_i))$. Note that it is sufficient to find the scores for these alignments. It is not necessary to find the actual alignment, let alone the DP tables that will be needed once the heuristic is actually selected. However, the simplest way to find the score for an alignment of two sequences is in fact to compute the DP table, and in any case they are needed to compute the scores for the triplets, which is done with A* using the pairwise heuristics. The amount of time required to compute the $h(s)$ scores of all 10 (Table 4.1) triplets is negligibly small compared to the time required to compute the optimal score for the whole set, as Figure 4.16 shows. The line showing the amount of time it takes to evaluate $h(s)$ for all heuristics h is flush with the x-axis, demonstrating that it is a completely negligible quantity. The next line is the amount of time spent preparing the heuristic tables. Its height corresponds to the number of octrees in the chosen heuristic: where the line is higher, there are two, and where it is lower there is one, and in the lowest case, there are zero octrees, corresponding to the one heuristic involving only pairwise tables and no octrees. The highest line shows the total amount of time used for the search, including time taken to select and prepare heuristics. The sequence set used for this experiment was E_3. Recall that these sequences are the first 300 characters of E_2. E_3 was used because the alignment search used too much memory for the weaker heuristics.

Figure 4.17 supports the notion that the most effective heuristics at restricting the search space tend to have high $h(s)$. Intuitively, they capture more of the constraints among the sequences. We note that the highest one is not necessarily the absolute best, but it often is, and if it is not, it is at least very good.

4.6 Analysis

The octree conserves memory by storing leaves as full tables only once a table cell from that leaf's volume is demanded by the search. If the search were to demand cells from every region of the table, the octree would be forced to store the entire table, and to re-compute it many times. For simplicity, assume all three sequences have identical length $\ell = 2^a$ and $\tau = 2^{3b}$, where $a, b \in \mathbb{Z}$. In practice we must have $a > b$ or else the octree degenerates to full matrix, so we restrict our attention to this case.

The side length of the root node is 2^a, and its volume is 2^{3a}. Each of its children has

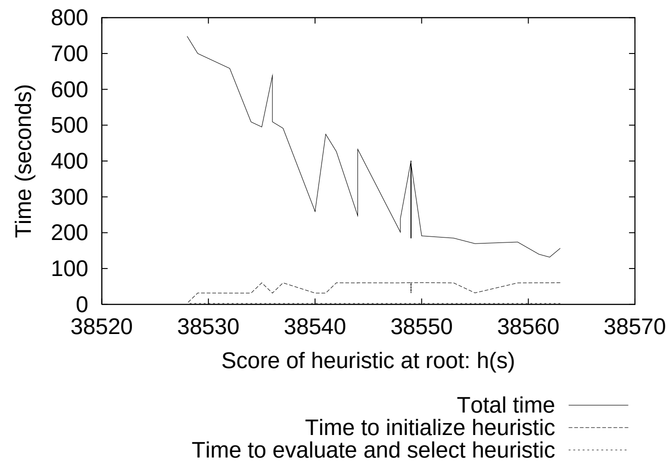


Figure 4.16: The time taken to evaluate all possible octree heuristics for E_3 and select the best is barely distinguishable from the x-axis. The time taken to initialize the chosen heuristic is also low, but since it is roughly constant it consumes a larger portion of the total runtime as the heuristic becomes stronger. The top line is the total runtime of the search, which is dominated by the A* search phase.

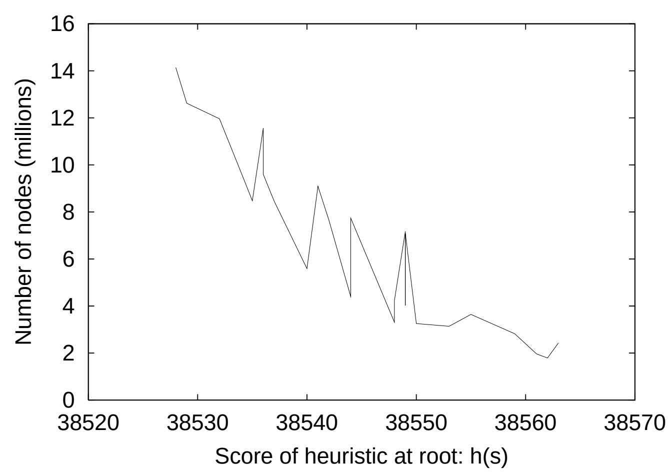


Figure 4.17: Total number of open and closed nodes stored in the search using a heuristic with $h(s) = x$. The shape of the curve is very similar to that of Figure 4.16. This is expected, since the time taken by an A* search is proportional to the number of nodes it examines.

side length 2^{a-1} and has volume 2^{3a-3}. The nodes in the bottom layer have side length 2^b, and have volume $\tau = 2^{3b}$. So, there are $a - b + 1$ layers in the tree. Each time an empty leaf suffers an access to its interior, it recomputes its contents once. In so doing, it replaces itself with an internal node and adds a subtree with 7 empty leaves and one internal node per level, down to the bottom level with 8 full leaves. If each region of volume τ is accessed once, the tree will fill out to include all possible full leaf nodes, at a cost of recomputing seven-eighths of T for each level of the tree, for $7/8\ell^3(a-b+1) = 7/8\ell^3(\log_2 \ell - \log_2 \sqrt[3]{\tau}+1)$ total table cells computed. Compare this to ℓ^3 cells computed if all of T is stored at the outset. For example, if $\ell = 256$ and $\tau = 512$, we have $256^3 = 16,777,216$ cells computed for a full table, and $(7/8)256^3(\log_2(256) - \log_2 \sqrt[3]{512} + 1) = 88,080,384$ recomputed cells in the worst case.

4.7 Benefits of Partial-Expansion A*

Partial-expansion A* search (PEA*) [62] puts a node back on the open list if some of its children have a very poor score. The authors claim [62] that they were able to reduce the number of entries in the open list by up to 95% in some cases, and decrease the time used as well. We examined these claims and found that the degree of savings rises with the ease of the problem.

We tested both E_1 and E_3 with varying values of C for both the octree heuristic and the pairwise heuristic used by the authors of PEA* [62]. E_2 was, again, too difficult to align using the pairwise heuristic. Recall from Chapter 3 that when PEA* expands a node n, it does not put a child n' on the OPEN list if $f(n') - f(n) > C$.

Figure 4.18 shows the sizes of the open and closed lists, and the sum of their sizes over a wide range of values of C for each of the heuristics tested. The value $C = 100$ is roughly equivalent on this set to a value of $C = \infty$ because a child exceeded the cutoff only 887 times over the course of the search for the pairwise, which is negligible. For the octree heuristic it never occurred at $C = 100$. The graph shows that for the octree heuristic PEA* shaved off at best 60% of the space used. With the pairwise heuristic it reduced space usage by only about 40%. This is our first indication that PEA* increases in power with the strength of the heuristic.

The graphs in Figure 4.19 show the times taken by the corresponding searches. As C shrinks linearly, the time taken by the search seems to rise exponentially. When $C = 0$, the search takes up to six or seven times as long as when $C = 100$ (equivalent to plain A*). At $C = 5$, the space savings is still close to its peak, and the time cost has decreased to a factor of 2.

These results serve to establish that PEA* does not necessarily provide a substantial benefit. However, they do not adequately characterize the situations where it is beneficial.

To do this it would be necessary to run many alignments, varying the degree of homology, lengths of sequences, number of sequences, and the strength of heuristic.

The sequence set on which Ishida et al. reported 95% space savings had 8 sequences, meaning each node had 255 children. The search rarely needed to revisit nodes that exceeded their cutoff, so net time was saved by not adding them to the open list. Nodes in E_2 by contrast were often re-expanded because omitted nodes were later found to have been necessary after all, nullifying the time savings. The same savings in time were not achieved by the octree because the sequences were very easy to align. The search using h_{p} could be completed in less time that it would take to construct the tables for h_{o}.

Diminishing Returns

This section describes experiments to see how the amount of memory saved by PEA* is affected by the difficulty of the problem. A collection E_4 of thirteen sets of five sequences distributed over a wide range of problem difficulty was drawn from K^+. Each sequence was truncated to 300 characters. The problem difficulty is defined as the number of nodes on OPEN and CLOSED at the end of a normal A* alignment search. 161 sets of sequences were drawn at random and run on ohaton using h_{p}. Sequence sets that took more than 2GB of RAM were cancelled and the sets discarded. Of the sets that could be aligned in the available space, a subset representing the full range of problem difficulties that could be aligned was chosen. On each of these sets, PEA* was run seven times, with C set to 1, 5, 10, 20, 30, 40, and 50.

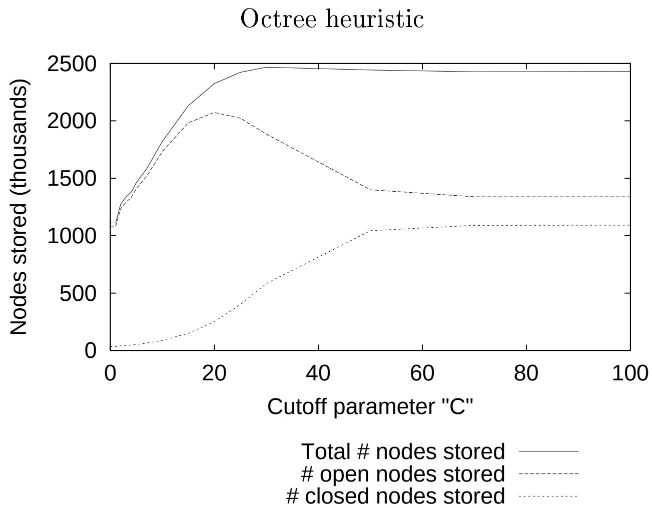
Figure 4.20 shows that as the problem becomes more difficult, the effectiveness of PEA* relative to plain A* decreases. With the easy problems, the ratio is around 0.5, much like Figure 4.18. This is expected since E_2 is also a relatively easy set. As the difficulty increases, the amount of memory used by PEA* with $C = 1$ increases to a factor of 0.8 of normal A*. Figure 4.21 shows more clearly than Figure 4.19 that the amount of time taken by PEA* can increase substantially with no significant corresponding gains in memory performance. So, if PEA* is used, C must be selected very carefully. If the problem is difficult, it may not be worth using PEA*.

4.8 Performance Benefits of the Octree

In the previous sections we have been concerned with justifying choices of ancillary parameters of the octree. We now turn to justifying its usefulness.

4.8.1 Data Sets and Parameters

Experiments used a normalized Dayhoff PAM250 [10] scoring matrix (Figure A.1, Appendix) with a linear gap cost of 8.



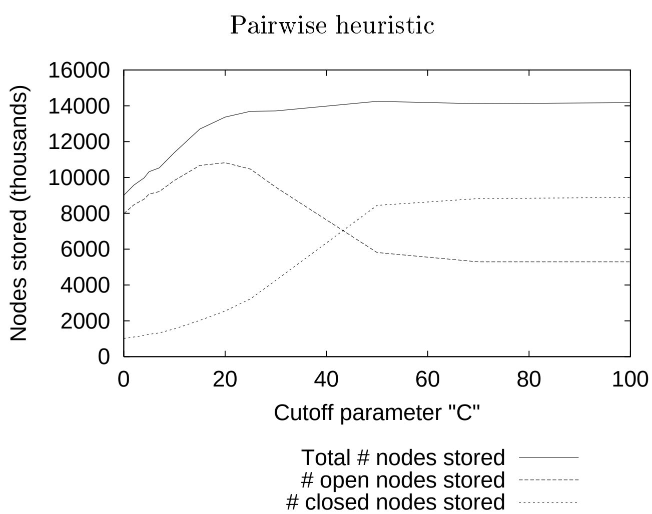


Figure 4.18: The total number of nodes stored at the end of the PEA* search on set E_2, using the octree heuristic (top) and the pairwise heuristic (bottom), broken down by population on open and closed lists, for various values of the cutoff parameter C.

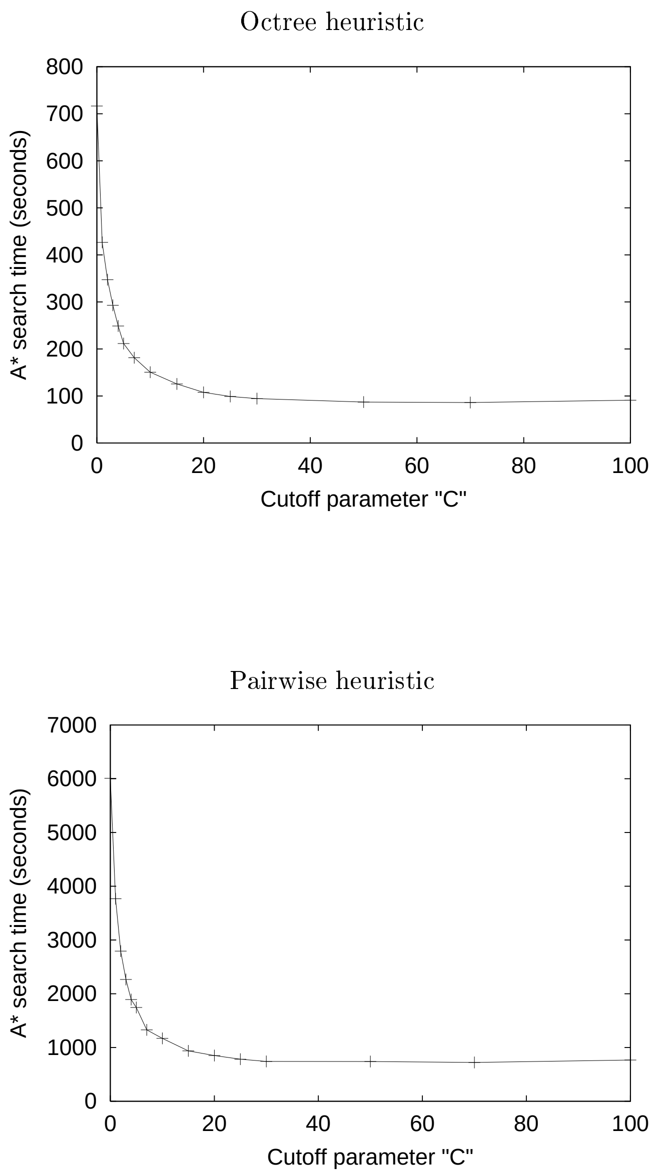


Figure 4.19: Time taken by the PEA* search on the set E_2 using the octree heuristic (top) and the pairwise heuristic (bottom). These quantities exclude all other times taken by the alignment, including both the time taken to initialize and to recompute octree heuristics. Note the different scales on the y-axis.

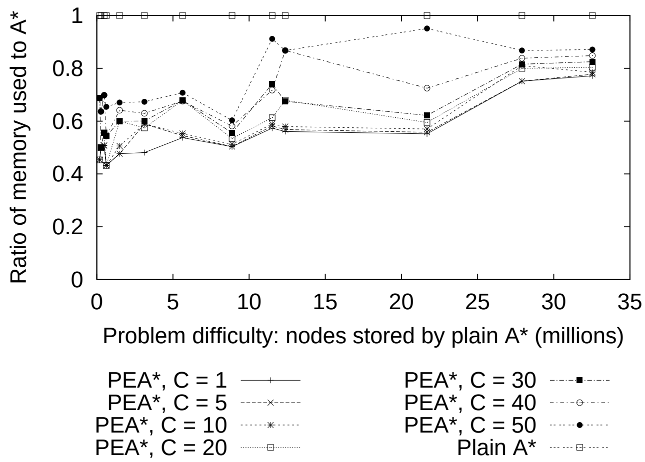


Figure 4.20: The ratio of the peak amount of memory used by each run of PEA* on the sets of E_4 to the amount used by plain A*, against the problem difficulty.

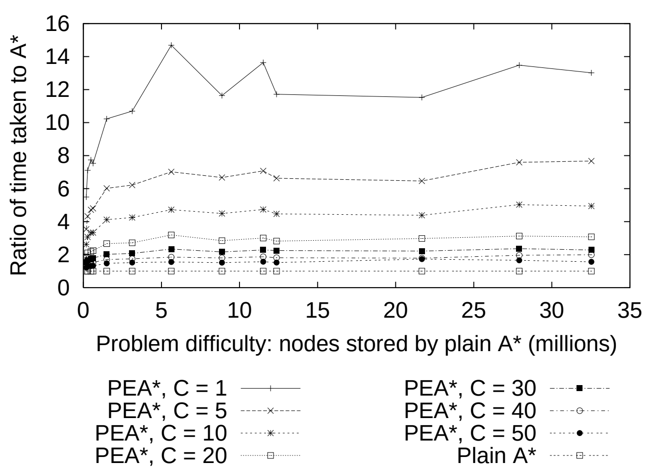


Figure 4.21: The ratio of time taken by each run of PEA* on the sets of E_4 to the amount used by plain A*, against problem difficulty.

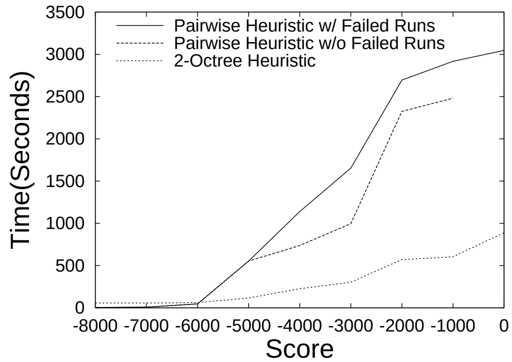


Figure 4.22: 5-way MSA Time

The experiments in this section were performed using $\tau = 10\ 000$. This value was later found (Section 4.4) to be inferior to $\tau = 1000$, but the difference is minor and the number of experiments that would have to be re-run was onerous. The PEA* cutoff parameter was set at $C = \infty$, equivalent to plain A*.

The full K^+ sequence set was used, with all sequences truncated to 300 characters. A variety of heuristics were used for space and time usage comparisons. These were: the best octree heuristic using two octrees (abbreviated "2 Octree" or h_{2o}), the best octree heuristic using only one octree ("1 Octree", h_{1o}), the best heuristic using two full 3-dimensional DP tables ("2 Full", h_{2f}), the best using only one 3-dimensional DP table ("1 Full", h_{1f}), and the plain pairwise DP tables heuristic ("Pairwise", h_p).

The experiments were run on ohaton. Not all alignment runs could complete in the available space. Runs that consumed more than 2 GB of RAM were killed.

Sets of 5 sequences were selected at random and aligned using h_{2o}. If the alignment completed, then the same set was attempted using h_p. Sets of size 5 were used because 4 is too small to justify using the octree for the vast majority of sets of this size, and 6 is too large for the search to complete using the pairwise heuristic on most sets, frustrating the comparison.

A total of 161 5-way alignments were attempted. Of these, 41 were solved by both h_{2o} and h_p, 27 were solved only by h_{2o}, and 93 were solved by neither, given the 2GB memory limitation. Note that there was no instance where a pairwise heuristic solved an alignment that the octree did not. Some alignments were then solved using the remaining heuristics h_{1o}, h_{2f}, and h_{1f} to provide finer comparisons of time and memory usage.

Figure 4.22 shows the time used by the various A* searches as a function of the alignment score; Figure 4.23 shows the memory usage. Failed runs were excluded completely from the averages for "w/o Failed Runs" and optimistically assumed to use exactly 2 GB of RAM and 3500 seconds of CPU time for "w/Failed Runs".

The more negative the score is, the more homologous the sequences and the easier the

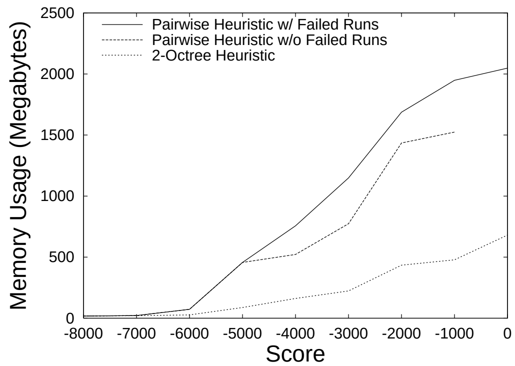


Figure 4.23: 5-way MSA Memory

alignment. The data has been grouped into buckets of 1000 and averaged. For example, all multiple alignments with a score between -8000 and -7000 were averaged into a single data point at -8000.

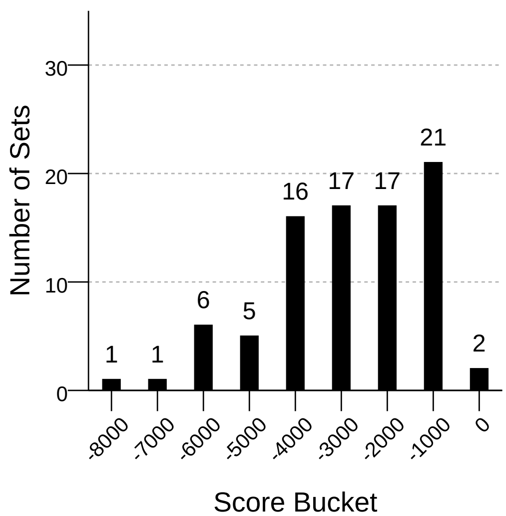


Figure 4.24: Distribution of sequence sets into score buckets

Figure 4.24 shows the number of sequence sets that fell into each score bucket. There were very few easy alignments (those with lower scores) and a much large number of more difficult alignments. The number of difficult alignments is understated because it is the difficult alignments that failed to complete and do not show up on this graph. For example, the data point at 0 shows that only two sequence sets with scores in the range $(0, 1000)$ were successfully aligned. The actual values of these two alignments were 10 and 50, on the easy edge of the bucket.

Figure 4.22 shows a positive correlation between the optimal alignment score and the ease of computation. For relatively easier alignments, h_{p} approximates h^* closely enough that the search space is small. In these cases h_{p} is faster and uses less space than $h_{2\mathrm{o}}$. As the score increases, h_{p} diverges from h^*. At -6000 the performance gap begins to widen and continues to become more exaggerated. At -3000 the average $h_{2\mathrm{o}}$ is three times faster than the h_{p} heuristics that completed. The numbers for $h_{2\mathrm{o}}$ include those for which the h_{p} counterpart failed, which decreases the apparent performance difference. The scores for $h_{2\mathrm{o}}$ are four times better than h_{p} including the lower bound estimate on the searches that did not complete.

For difficult problem instances, both the time and space taken to construct the octrees are more than compensated for by the reduced size of the search. A smaller search uses less space for the open and closed lists, and takes less time. Extrapolating from the curves of Figures 4.22 and 4.23 suggests an exponential increase in growth. This growth pattern appears to taper off for the h_{p} but this is illusory since we can see that at the (-1000,0) bucket almost all runs are hitting the memory limit. The $h_{2\mathrm{o}}$ shows the pattern, though its growth is much slower, because the exponent of the search is lower.

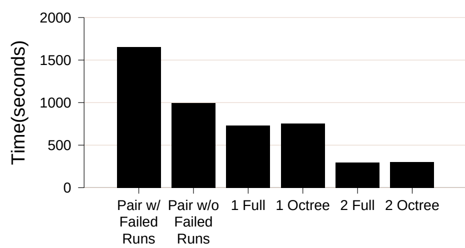


Figure 4.25: Time (-3000 bucket)

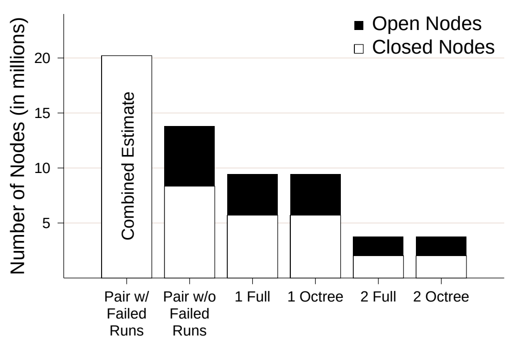


Figure 4.26: Search Space (-3000 bucket)

Figures 4.25, 4.26 and 4.27 show a detailed analysis of the (-3000,-2000) bucket. This bucket had the most data points. For sequence sets in this group, the search was re-run using $h_{1\mathrm{f}}$, $h_{1\mathrm{o}}$, and $h_{2\mathrm{f}}$. The left bar of Figure 4.26 is an estimate since runs that failed to complete did not report this information, so the estimate of the combined sizes of the open and closed lists is derived from the 2 GB memory limit.

The searches with $h_{2\mathrm{o}}$ average 4-6 times faster than h_{p}. Memory usage is similar. In all

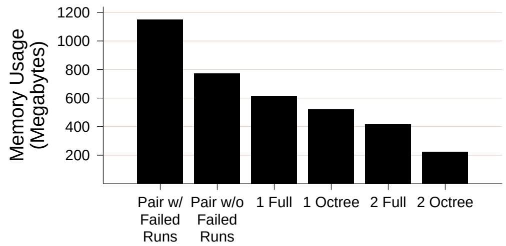


Figure 4.27: Memory (-3000 bucket)

cases the octree runs as fast or faster and uses less space than its competitors. The full matrix is occasionally slightly faster than the octree. However, the octree uses a more compact memory representation that has better cache behavior, mostly offsetting the disadvantage it has in deep tree accesses and recomputation time. The full matrix could also use the same memory organization to improve its time performance. Note that both $h_{2\mathrm{o}}$ and $h_{2\mathrm{f}}$ use less overall memory and time than either of their counterparts $h_{1\mathrm{o}}$ and $h_{1\mathrm{o}}$. Obviously they use more memory to store their heuristics but this memory usage is more than offset by the smaller search space.

4.9 Summary

The octree shows a clear performance improvement over the typical pairwise heuristic in A* sequence alignment search, for sufficiently difficult problem instances. Among the interesting problem instances, difficult ones are far more plentiful than easy ones. The optimal leaf-type threshold parameter τ can be chosen from a wide range of values. When several alternative heuristics exist, their values at the root node are a reliable indicator of their relative efficacy. Partial-expansion A* performs more poorly as problem difficulty increases.

Chapter 5

Discussion and Future Work

5.1 Enhancements to the Octree

As discussed in the previous chapter, the octree as presented could be forced in the worst case to recompute the dynamic programming table many times. It is designed to take advantage of the fact that nodes in the lattice that are examined by the search tend to occur in clusters. For the vast majority of sequence sets, the alignment path and the searched areas of the lattice tends to run down the diagonal of the space. If this does not hold, then the octree will be at a disadvantage. However, there is a mitigating characteristic of the search's behavior that the octree does not take advantage of: no node is considered in the A* search without a predecessor node having been considered previously. We did not take advantage of this in order that the octree would be flexible enough for experimental purposes to accomodate other access patterns.

The octree as described did not take advantage of upper bound estimates to limit the volume of the DP table that it computed and stored. We saw that most of its space usage is accounted for by the surfaces of empty leaves. It should be possible not only to limit this space usage but to decrease the time to generate the octree by safely ignoring portions of the table, much like LBD-Align [9].

5.2 Improved Evaluation

It has been difficult to compare the performance of search techniques for sequence alignment that have come out of the AI community because researchers have made up their own data sets rather than using a standard benchmark of sequences. Future evaluation of alignment techniques should be against BAliBASE [54], a benchmark set of alignments that covers a broad range of lengths and degrees of similarity of interest to biologists, with hand-verified reference alignments. Initial results indicate that the linear gap cost commonly used by AI researchers [62, 32] does not typically produce high-quality optimal alignments. The octree should be enhanced to use the quasi-natural affine gap model [2] which formally induces a larger search space but according to initial tests does not necessarily search more nodes than the linear gap cost. The same tests indicate that it should produce better-quality alignments, according to BAliBASE.

We did not use BAliBASE because the purpose of this thesis was to tune the octree heuristic and to evaluate its performance against the pairwise heuristic. For this task any set of sequences presenting a variety of alignment difficulty would do, and the K^+ set served well. Additionally, most sequences sets in BAliBASE are too difficult to align exactly, either because the sets contain too many sequences or have very low homology. We would have had to adapt BAliBASE by trimming the sets, which would have defeated its major purpose, which is to provide a standard set of sequences for comparing alignment programs.

5.3 Other Heuristics

Before the octree was discovered, many heuristics were examined and discarded as ineffectual. Others require more study.

5.3.1 Three-Way DP Tables

The octree stores only needed portions of a DP table for three sequences. Other methods of storing less than the full table were attempted.

Antidiagonal Plane

The antidiagonal plane p in the DP table t is the set of coordinates $\{(i,j,k) : i+j+k=c\}$ for some constant c unique to p. Heuristic h_{diag} stored only the minimum value in each antidiagonal. This requires space only $O(\ell)$, but was less effective than h_{p}.

Subcube downsampling

The DP table can be reduce by a constant factor of $\textit{step}^3$ by only storing $t(i,j,k)$ if $i,j,k \equiv 0 \bmod \textit{step}$. I.e. divide the table into cubes of side length *step* and store only one corner of each cube. However, it is not possible to reconstruct admissible values for discarded nodes given this information. This is resolved by storing three cells per cube: one for each of the perpendicular faces farthest from the origin. The shortest path from a discarded cell c inside the cube to the goal node t must pass through one of these faces. Assume a shortest path with perfect matches from c to each face, and take the minimum cost of such a path over all faces. This guarantees $h(c) \leq h^*(c)$. This method was not effective.

5.3.2 A* subsearch

Kobayashi and Imai [26] showed how an A* search over subsets of sequences could efficiently provide a heuristic estimate. By forcing the closed list to grow even after the goal node of the subsearch is reached it can be formed into a pool of values useful as heuristics. We independently discovered this idea but did not develop it to the extent that the cited paper had. Having already moved on to the octree, we did not revisit it nor compare its performance to the octree, but it does merit further investigation.

5.3.3 Alternation and Aggregation

The degree of homology of sequences may vary by location. It may be profitable to maintain a few heuristic functions and consult several of them, picking the maximum at each node. Preliminary experiments showed that the reduction in search effort was not outweighed by the increase in time to prepare octrees. However, if octrees or other complex heuristics become cheaper to evaluate, or, if processors that would otherwise be idle are available, it may be profitable to combine several overlapping heuristics. This is also discussed by Kobayashi and Imai [26].

5.4 Future Work

5.4.1 Parallelization

General-purpose parallel search [46] allows many computers to share the task of expanding nodes on the open list but suffers from high communication overhead in coordinating work and avoiding duplicate effort. Results of applying such methods to sequence alignment are absent from the literature. It would seem more promising in any case to assign responsibility for the open list to one computer, and distribute the evaluation of heuristics. This could be a particularly large win for sequence alignment for several reasons:

- Each node has many children, which can be evaluated in parallel.
- Each heuristic can be composed from many independent functions, which can be evaluated in parallel.
- Several overlapping heuristics may be evaluated in parallel (Section 5.3.3).

Results using these types of parallelization are also absent from the literature.

5.4.2 Opportunistic Recursive Best-First Search

Korf [28] introduced the recursive best-first search, which finds an optimal path using space proportional to the length of the path and the branching factor. It can therefore search using a very limited amount of space. However, it does not make use of additional memory when it is available. Robnik-Šikonja [45] presented Opportunistic RBFS (ORBFS), an adaptation of Korf's algorithm that can be restricted to the same amount of memory, but it uses additional memory to avoid recomputation. A naïve and incorrect version of ORBFS was initially implemented for sequence alignment, and appeared promising. It should be implemented properly and examined for parallel search as well.

5.4.3 Approximate Alignment

Provably exact alignments will probably never be feasible for sets of K sequences when K is large, simply due to the fact that the branching factor of the search, that is, the number of children at each node, is $b = 2^K - 1$. Provably exact alignment will probably never even be feasible for small numbers of highly dissimilar sequences. For the latter case, however, it should be feasible to create exact alignments very close to optimal. Divide-and-Conquer (DCA) [51, 52, 44, 55] represents one attempt. It breaks all sequences along a hypothesized point on the optimal path through the lattice and recursively aligns each half. By doing so it reduces the search space by a factor of 2^{K-1} each time it recurses (keeping only the two corners of a K-dimensional block cut down the middle). Howver, this method may fail to split the points anywhere near the true optimal if the sequences are highly dissimilar. However, it often happens that a few of the sequences share a common **motif** (sequence of letters) along which it is obvious that they should align, and it would be nice to be able to force them to do so without sacrificing accuracy elsewhere. To achieve this, the A* sequence alignment search could be modified to accept a list of *partially-specified* coordinates through which the search must pass. E.g. for four sequences such a coordinate might be $(52, \times, 50, \times)$, specifying that the 52nd character of the first sequence must line up with the 50th character of the third sequence, and that the other two sequences are not constrained. If, for example, the node $(51, 32, 51, 50)$ was encountered in the search it could be discarded. This has not been described in the literature and would be worth investigating. If the benefit is sufficient it would strike a nice balance between all-or-nothing approaches like DCA and progressive alignment methods like CLUSTALW [53].

5.5 Relevance of Exact MSA

Initial results on BAliBASE show that the linear gap model used for all experiments in this thesis does not induce good exact alignments. It may be that it could induce good alignments if the gap parameters and substitution matrix were judiciously tuned, but this seems unlikely. The quasi-natural gap model was incompletely implemented in this work, but a serviceable (yet technically inadmissible) A* version was constructed. It was tested against a correct dynamic programming version and found to deviate rarely from the true quasi-natural alignment. On that basis we tested it against the BAliBASE reference alignments and found that while it was markedly better than the linear gap cost, it too was not perfect, and in particular, not as good as results of approximate alignments published by the BAliBASE authors [54].

The biological validity of exact alignment is limited by the gap model. The quasi-natural gap model is the most accurate one that can be computed exactly for sequence sets large enough to be of interest. But is it accurate enough? Is the *quantitatively* optimal exact alignment *qualitatively* superior to a near-optimal approximate alignment? That is, approximation techniques exist that give a near-optimal alignment with high confidence. Is the difference in score worth the extra cost? Second, and more fundamental, is whether the alignment that is quantitatively optimal with respect to a scoring function also qualitatively optimal? If not, there is no point in finding it. These questions have not been conclusively settled. Therefore, further work on making exact multiple sequence alignment faster should be contingent on it being shown to be effective.

Bibliography

[1] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *J. Mol. Biol.*, 215:403–410, 1990.

[2] Stephen F. Altschul. Gap costs for multiple sequence alignment. *J. Theor. Biol.*, 138:297–309, 1989.

[3] Vineet Bafna, Eugene L. Lawler, and Pavel A. Pevzner. Approximation algorithms for multiple sequence alignment. *Theoretical Computer Science*, 182:233–244, 1997.

[4] T. H. Byers and Michael S. Waterman. Determining all optimal and near-optimal solutions when solving shortest path problems by dynamic programming. *Oper. Res.*, 32:1381–1384, 1984.

[5] Humberto Carrillo and David Lipman. The multiple sequence alignment problem in biology. *SIAM J. Appl. Math.*, 48(5):1073–1082, 1988.

[6] K. Chao, R. Hardison, and W. Miller. Recent developments in linear-space alignment methods: A survey. *J. Comp. Biol.*, 1(4):271–291, 1994.

[7] Kevin Charter, Jonathan Schaeffer, and Duane Szafron. Sequence alignment using FastLSA. In *Proceedings of The 2000 International Conference on Mathematics and Engineering Techniques in Medicine and Biological Sciences (METMBS'2000)*, pages 239–245, June 2000.

[8] Joe Culberson and Jonathan Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.

[9] Aaron Davidson. A fast pruning algorithm for optimal sequence alignment. In *Proceedings of The 2nd IEEE International Symposium on Bioinformatics & Bioengineering (BIBE'2001)*, November 2001.

[10] M. O. Dayhoff, R. M. Schwartz, and B. C. Orcutt. A model of evolutionary change in proteins. In M. O. Dayhoff, editor, *Atlas of Protein Structure*, volume 5(Suppl. 3), pages 345–352. National Biomedical Research Foundataion, Silver Spring, Md., 1979.

[11] Adrian Driga. Parallel FastLSA: A parallel algorithm for pairwise sequence alignment. Master's thesis, University of Alberta, 2001.

[12] Walter M. Fitch. An improved method of testing for evolutionary homology. *J. Mol. Biol.*, 16:9–16, 1966.

[13] Warren Gallin. Potassium channel sequences. Private communication. `http://www.cs.ualberta.ca/~bioinfo/sequences/pchannel/`.

[14] Gaston H. Gonnet, Chantal Korostensky, and Steve Benner. Evaluation measures of multiple sequence alignments. *J. Comp. Biol.*, 7(1,2):261–76, 2000.

[15] Osamu Gotoh. An improved algorithm for matching biological sequences. *J. Mol. Biol.*, 162:705–708, 1982.

[16] Osamu Gotoh. Alignment of three biological sequences with an efficient traceback procedure. *J. Theor. Biol.*, 121:327–337, 1986.

[17] Osamu Gotoh. Significant improvement in accuracy of multiple protein sequence alignments by iterative refinement as assessed by reference to structural alignments. *J. Mol. Biol.*, pages 823–838, 1996.

[18] Jerome Gracy and Jean Sallantin. Multiple sequence alignment using anchor points through generalized dynamic programming. In S. Miyano, T. Akutsu, H. Imai, O. Gotoh, and T. Takagi, editors, *Proceedings of Genome Informatics Workshop 1994*. Universal Academy Press, Tokyo, 1994, 1994.

[19] Sandeep K. Gupta, John D. Kececioglu, and Alejandro A. Schäffer. Improving the practical space and time efficiency of the shortest-paths approach to sum-of-pairs multiple sequence alignment. *J. Comp. Biol.*, 2(3):459–472, 1995.

[20] Dan Gusfield. Efficient methods for multiple sequence alignment with guaranteed error bounds. *Bulletin of Mathematical Biology*, 55(1):141–154, 1993.

[21] D. Hirschberg. A linear space algorithm for computing maximal common subexpressions. *CACM*, 18(6):341–343, 1975.

[22] Robert Holte. Hierarchical A*: Searching abstraction hierarchies efficiently. *AAAI*, pages 530–535, 1996.

[23] Takahiro Ikeda and Hiroshi Imai. Fast A* algorithms for multiple sequence alignment. In S. Miyano, T. Akutsu, H. Imai, O. Gotoh, and T. Takagi, editors, *Proceedings of Genome Informatics Workshop*, pages 90–99. Universal Academy Press, Tokyo, 1994.

[24] Andreas Junghanns and Jonathan Schaeffer. Enhancing single-agent search using domain knowledge. *Artificial Intelligence*, 129(1-2):219–251, 2001.

[25] Winfried Just. Computation complexity of multiple sequence alignment with sp-score. *J. Comp. Biol.*, 8(2):615–623, 2001.

[26] Hirotada Kobayashi and Hiroshi Imai. Improvement of the A* algorithm for multiple sequence alignment. In *The Ninth Workshop on Genome Informatics*, 1998.

[27] Richard E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[28] Richard E. Korf. Linear-space best-first search. *Artificial Intelligence*, 62:41–78, 1993.

[29] Richard E. Korf. Divide-and-conquer bidirectional search: First results. *IJCAI*, pages 1184–1189, 1999.

[30] Richard E. Korf. Recent progress in the design and analysis of admissible heuristic functions. *AAAI*, pages 1165–1170, 2000.

[31] Richard E. Korf and M. Reid. Complexity analysis of admissible heuristic search. *AAAI*, pages 305–310, 1998.

[32] Richard E. Korf and Weixiong Zhang. Divide-and-conquer frontier search applied to optimal sequence alignment. *AAAI*, pages 910–916, 2000.

[33] Chantal Korostensky and Gaston H. Gonnet. Near optimal multiple sequence alignments using a traveling salesman problem approach. In *SPIRE/CRIWG*, pages 105–114, 1999.

[34] Stephen A. Liebhaber, Michel J. Goossens, and Yuet Wai Kan. Cloning and complete nucleotide sequence of human $5'$–α–globin gene. *Proc. Natl. Acad. Sci. USA*, 77(12):7054–7058, December 1980.

[35] David. J. Lipman, Stephen. F. Altschul, and John. D. Kececioglu. A tool for multiple sequence alignment. *Proc. Natl. Acad. Sci. USA.*, 86:4412–4415, June 1989.

[36] G. Manzini. BIDA*: An improved perimeter search algorithm. *Artificial Intelligence*, 75:347–360, 1995.

[37] Marcella A. McClure, Taha K. Vasi, and Walter M. Fitch. Comparative analysis of multiple protein-sequence alignment methods. *Mol. Biol. Evol.*, 11(4):571–592, 1994.

[38] Teruhisa Miura and Toru Ishida. Stochastic node caching for memory-bounded search. In *AAAI/IAAI*, pages 450–456, 1998.

[39] E. Myers and W. Miller. Optimal alignments in linear space. *CABIOS*, 4(1):11–17, 1988.

[40] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *J. Mol. Biol.*, 48:443–453, 1970.

[41] Cédric Notredame and Desmond G. Higgins. SAGA - sequence alignment by genetic algorithm. *Nucleic Acids Research*, 24(8):1515–1524, 1996.

[42] Cédric Notredame, Desmond G. Higgins, and Jaap Heringa. T-coffee: A novel method for fast and accurate multiple sequence alignment. *J. Mol. Biol.*, 302:205–217, 2000.

[43] Knut Reinert and Martin Lermen. The practical use of the A* algorithm for exact multiple sequence alignment. *J. Comp. Biol.*, 7(5):655–671, 2000.

[44] Knut Reinert, Jens Stoye, and Torsten Will. An iterative method for faster sum-of-pairs multiple sequence alignment. *Bioinformatics*, 16(9):808–814, 2000.

[45] Marko Robnik-Šikonja. Effective use of memory in linear space best first search. Technical report, University of Ljubljana, 1996.

[46] John W. Romein, Aske Plaat, Henri E. Bal, and Jonathan Schaeffer. Transposition table driven work scheduling in distributed search. In *AAAI/IAAI*, pages 725–731, 1999.

[47] Stuart Russell and Peter Norvig. *Artificial Intelligence: A modern approach*. Prentice-Hall, Inc., 1995.

[48] Peter H. Sellers. On the theory and computation of evolutionary distances. *SIAM J. Appl. Math.*, 26(4):787–793, 1974.

[49] T. F. Smith, Michael S. Waterman, and Walter M. Fitch. Comparative biosequence metrics. *J. Mol. Evol.*, 18:38–46, 1981.

[50] John L. Spouge. Speeding up dynamic programming algorithms for finding optimal lattice paths. *SIAM J. Appl. Math.*, 49(5):1552–1566, 1989.

[51] Jens Stoye. Multiple sequence alignment with the divide-and-conquer method. *Gene*, 211:GC45–GC56, 1998.

[52] Jens Stoye, Sören W. Perrey, and Andreas W. M. Dress. Improving the divide-and-conquer approach to sum-of-pairs multiple sequence alignment. *Appl. Math. Lett.*, 10(2):67–73, 1997.

[53] J. Thompson, D. Higgins, and T. Gibson. CLUSTAL W: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Research*, 22:4673–4680, 1994.

[54] Julie D. Thompson., Frédéric Plewniak, and Olivier Poch. A comprehensive comparison of multiple sequence alignment programs. *Nucleic Acids Research*, 27(13):2682–2690, 1999.

[55] Udo Tönges, Sören W. Perrey, Jens Stoye, and Andreas W.M. Dress. A general method for fast multiple sequence alignment. *Gene*, 172:GC33–GC41, 1996.

[56] Martin Vingron and Patrick Argos. Determination of reliable regions in protein sequence alignments. *Protein Engineering*, 3(7):565–569, 1990.

[57] Lusheng Wang and Tao Jiang. On the complexity of multiple sequence alignment. *J. Comp. Biol.*, 1(4):337–348, 1994.

[58] Michael. S. Waterman. Sequence alignments in the neighborhood of the optimum with general application to dynamic programming. *Proc. Natl. Acad. Sci. USA*, 80:3123–3124, May 1983.

[59] Michael S. Waterman. Efficient sequence alignment algorithms. *J. Theor. Biol.*, 108:333–337, 1984.

[60] Michael S. Waterman. General methods of sequence comparison. *Bulletin of Mathematical Biology*, 46(4):473–500, 1984.

[61] Michael S. Waterman, T. F. Smith, and W. A. Beyer. Some biological sequence metrics. *Advances in Mathematics*, 20:367–387, 1976.

[62] Takayuki Yoshizumi, Teruhisa Miura, and Toru Ishida. A* with partial expansion for large branching factor problems. *AAAI*, pages 923–929, 2000.

Appendix A

Example Alignments and Scoring Matrices

Figure A.1: A big hemoglobin alignment. See Figure A.2 for descriptions of these sequences.

```
                10        20        30        40        50        60        70
HAHU    LSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTYFPHF-DLSH-----GSAQVKGHGKKVADALTNAVA
HBHU    LTPEEKSAVTALWGKV--NVDEVGGEALGRLLVVYPWTQRFFESFGDLSTPDAVMGNPKVKAHGKKVLGAFSDGLA
MYHU    LSDGEWQLVLNVWGKVEADIPGHGQEVLIRLFKGHPETLEKFDKFKHLKSEDEMKASEDLKKHGATVLTALGGILK
S15768  FTDKQEALVNSSWEAFKQNLPRYSVFFYTVVLEKAPAAKGLFSFLKN---SAEVQDSPQLQAHAEKVFGLVRDSAV
JT0423  LSADEAAIVKSSWDQVKHN----EVDILAAVFAAYPDIQAKFPQFA-GKDLASIKDTAAFATHATRIVSFFTEVIS
A39023  LSGAEADLLAKSWAPVFANKDANGDNFLIALFEAFPDSANFFGDFK-GKSIADIRASPKLRSVSSRIVNRLNDFVG
JC1516  SDLCVKSLE-GRMVGTEAQNIENGNAFFRYFFTNFPDLRVYFKGAEKYTA-DDVKKSERFDKQGQRILLACHLLAN
A53396  LTQKTKDIVKATAPVLAEHGYDIIKCFYQRMFEAHPELKNVFNM-----------AHQEQGQQQQALARAVYAYAE
S57699  LAEKTRSIIKATVPVLEQQGTVITRTFYKNMLTEHTELLNIFNR-----------TNQKVGAQPNALATTVLAAAK
GGZLB   LDQQTINIIKATVPVLKEHGVTITTTFYKNLFAKHPEVRPLFDM-----------GRQESLEQPKALAMTVLAAAQ
S44277  LDQQTIATIKSTIPLLAETGPALTAHFYQRMFHHNPELKDIFNM-----------SNQRNGDQREALFNAICAYAT
S26964  LTPTEINFLQSLAPVVKEHGVTVTSTMYKYMFQTYPEVRSYFNM-----------TNQKTGRQPKVLAFSLYQYIL
A39601  TQPANKNLIRSTWNMMVGDRGN-GVELMGLLFQRAPDSKIDFKRLGDVSA-ENIPYNRKLNGHGITLWYALMNFVD
A39601  TKPANKGLIRETWNMIAGDRKN-GVELMALLFEMAPDSKKDFRRLGDVSP-SNIPNNRKLNGHGITLWYALMNFVD
A47183  RELCMKSLEHAKVDTSNEARQD-GIDLYKHMFENYPPLRKYFKNREEYTA-EDVQNDPFFAKQGQKILLACHVLCA
A47183  RHQCMRSLQHIDIGHSETAKQN-GIDLYKHMFENYPSMREAFKDRENYTA-EDVQKDPFFVKQGQRILLACHLLCA
        .      .    .          .   .   .   :     *                    ..     .
          80        90       100       110       120       130       140       150
HAHU    HVD------DMPNALSALSDLHAHKLRVDPVNFKLLSHCLLVTLAAHLPAEFTPAVHASLDKFLASVSTVLTSKYR
HBHU    HLD------NLKGTFATLSELHCDKLHVDPENFRLLGNVLVCVLAHHFGKEFTPPVQAAYQKVVAGVANALAHKYH
MYHU    KKG------HHEAEIKPLAQSHATKHKIPVKYLEFISECIIQVLQSKHPGDFGADAQGAMNKALELFRKDMASNYK
S15768  QLR---ATGGVVLGDATLGAIHVRKGVVDPHFVVVKEALLKTIK-EAAGDKWSEELNTAWEVAYDALATAIKKAMS
JT0423  LSGNQANLSAVYALVSKLGVDHKARGISAAQFGEFRTALVSYLQ---AHVSWGDNVAAAWNHALDNTYAVALKSLE
A39023  NAA---DAGKMAGMLDQFSKEHVGFGVGSQQFENVRSMFPGFVS---SVAAPPAGADAAWGKLFGLIIDALKKAGK
JC1516  VYT---NEEVFKGYVRETINRHRIYKM-DPALWMAFFTVFTGYL--ESVGCLNDQQKAAWMALGKEFNAESQTHLK
A53396  NIE---DPNSLMAVLKNIANKHASLGVKPEQYPIVGEHLLAAIK-EVLGNAATDDIISAWAQAYGNLADVLMGMES
S57699  NID---DLSVLMDHVKQIGHKHRALQIKPEHYPIVGEYLLKAIK-EVLGDAATPEIINAWGEAYQAIADIFITVEK
GGZLB   NIE---NLPAILPAVKKIAVKHCQAGVAAAHYPIVGQELLGAIK-EVLGDAATDDILDAWGKAYGVIADVFIQVEA
S44277  HIE---NLPALLPAVERIAQKHASFNIQPEQYQIVGTHLLATLE-EMF--QPGQAVLDAWGKRYGVLANVFIQRES
S26964  HLN---DLTPISGFVNQIVLKHCGLGIKPDQYPVVGESLVQAFK-MVLGEAADEHFVEVFKKAYGNLAQTLIDAEA
A39601  QLD---SKKDLEDVCRKFAVNHVIRGVLDVKFGWIKEPMAELLR-RKCGNDC-DDAIQAWWKLIDVICAVLKESKK
A39601  QLD---NKIDLEDVCRKFAVNHVNRGVLDVKFAWIKEPLAELLR-RKCGQTCTDQHIQAWWKLIDVVCAVLEEKK-
A47183  TYD---DRETFNAYTRELLDRHARDHVHMPPEVWTDFWKLFEEY-LGKKTTLDEPTKQAWHEIGREFAKEINKHGR
A47183  SYD---DEETFHMYVHELMERHERLGVQLPDQHWTDFWKLFEEF-LEKKSHLCEHTKHAWAVIGKEFAYEATRHGK
                             *   ..           ..        .         :. .     .
```

Sequence ID	Full sequence name
HAHU	hemoglobin alpha chain [validated] - human
HBHU	hemoglobin beta chain [validated] - human
MYHU	myoglobin [validated] - human
S15768	leghemoglobin - alfalfa
JT0423	hemoglobin VIIB-7 precursor - midge (Chironomus thummi thummi)
A39023	globin - ragged sea hare
JC1516	globin - Caenorhabditis elegans
A53396	flavohemoglobin - Alcaligenes eutrophus
S57699	flavohemoglobin - yeast (Saccharomyces cerevisiae)
GGZLB	bacterial hemoglobin - Vitreoscilla sp.
S44277	flavohemoglobin hmpX - Erwinia chrysanthemi
S26964	flavohemoglobin - yeast (Pichia norvegensis)
A39601	hemoglobin 35K chain - ark shell (Barbatia reeveana)
A47183	hemoglobin precursor [validated] - pig roundworm

Figure A.2: Sequence descriptions

C	-12																			
S	0	-2																		
T	2	-1	-3																	
P	3	-1	0	-6																
A	2	-1	-1	-1	-2															
G	3	-1	0	1	-1	-5														
N	4	-1	0	1	0	0	-2													
D	5	0	0	1	0	-1	-2	-4												
E	5	0	0	1	0	0	-1	-3	-4											
Q	5	1	1	0	0	1	-1	-2	-2	-4										
H	3	1	1	0	1	2	-2	-1	-1	-3	-6									
R	4	0	1	0	2	3	0	1	1	-1	-2	-6								
K	5	0	0	1	1	2	-1	0	0	-1	0	-3	-5							
M	5	2	1	2	1	3	2	3	2	1	2	0	0	-6						
I	2	1	0	2	1	3	2	2	2	2	2	2	2	-2	-5					
L	6	3	2	3	2	4	3	4	3	2	2	3	3	-4	-2	-6				
V	2	1	0	1	0	1	2	2	2	2	2	2	2	-2	-4	-2	-4			
F	4	3	3	5	4	5	4	6	5	5	2	4	5	0	-1	-2	1	-9		
Y	0	3	3	5	3	5	2	4	4	4	0	4	4	2	1	1	2	-7	-10	
W	8	2	5	6	6	7	4	7	7	5	3	-2	3	4	5	2	6	0	0	-17
	C	S	T	P	A	G	N	D	E	Q	H	R	K	M	I	L	V	F	Y	W

Table A.1: The substitution matrix [62] used for comparing the performance of octree heuristic to pairwise heuristic

A	15																			
C	19	5																		
D	17	22	13																	
E	17	22	14	13																
F	21	21	23	22	8															
G	16	20	16	17	22	12														
H	18	20	16	16	19	19	11													
I	18	19	19	19	16	20	19	12												
K	18	22	17	17	22	19	17	19	12											
L	19	23	21	20	15	21	19	15	20	11										
M	18	22	20	19	17	20	19	15	17	13	11									
N	17	21	15	16	21	17	15	19	16	20	19	15								
P	16	20	18	18	22	18	17	19	18	20	19	18	11							
Q	17	22	15	15	22	18	14	19	16	19	18	16	17	13						
R	19	21	18	18	21	20	15	19	14	20	17	17	17	16	11					
S	16	17	17	17	20	16	18	18	17	20	19	16	16	18	17	15				
T	16	19	17	17	20	17	18	17	17	19	18	17	17	18	18	16	14			
V	17	19	19	19	18	18	19	13	19	15	15	19	18	19	19	18	17	13		
W	23	25	24	24	17	24	20	22	20	19	21	21	23	22	15	19	22	23	0	
Y	20	17	21	21	10	22	17	18	21	18	19	19	22	21	21	20	20	19	17	7
	A	C	D	E	F	G	H	I	K	L	M	N	P	Q	R	S	T	V	W	Y

Table A.2: The substitution matrix [35] used for studying octree tuning parameters