Acknowledgments

Wise sayings often fall on barren ground; but a kind word is never thrown away.

Sir Arthur Helps

First of all I would like to thank my adviser Søren Larsen for his advice and patience. Then I will thank Henrik Gordon Petersen for being my second supervisor for a short while and for being the first to believe in this project. Thanks to Vibeke Pierson and Lars Jacobsen for reading the whole lot and point out the errors and misspellings. A special thanks to James Lundon for maintaining the FAQ for the *rec.puzzles.crosswords* newsgroup. Without his effort I had never found the appropriate references to papers and books. Also a big thanks to Kevin McCann for setting up the Cruciverb-L discussion forum from which I have been inspired with many a good idea. A grateful thought to all the people at this forum. Thanks to Doug McIlroy at Bell Laboratories for sending me the Word Ways papers. A big thanks to Margit Christiansen, our librarian, for finding the remaining papers.

Thanks goes to the following people for thoughts and comments: Andres Rodriguez at MacGuffin Corporation, Boston University, Toke Eskildsen at Aarhus University, Doug McIlroy at Bell Laboratories, Lee Daniel Quinn (a retired communications consultant), Peter Jarosi, John Connett, Lee McGinty, Alec McHoul at Murdoch University, Australia, Navin Kashyap, Frank Longo, Ross Beresford, Ronan O'Riain, William Tunstall-Pedoe, Wei-Hwa Huang, Dan Tilque, and Ole Storm Pedersen.

To all the rest I have forgotten: Thanks a lot!

Sik Cambon Jensen, 2nd January 1997.

Contents

Preface

The beginning is the most important part of the work.
Plato

0.1 Introduction

> **There are two kinds of people, those who finish what they start and so on.**
> *Robert Byrne*

0.1.1 What made *me* start?

The idea to create artificial crossword puzzles using computers was initiated by an article in the newspaper *Ingeniøren* January 21st 1994. The article describes a program capable of **solving** crossword puzzles - that is - it can assist a human solver in a more advanced way than various other dictionaries are capable of. This gave me the idea ...

0.1.2 Questions to be raised

It soon became apparent that the crossword puzzle problem raises a lot of interesting computer scientific questions.

1. **Heuristics**
   What type of algorithm will perform best on individual problems, and will it also perform as the best algorithm overall?

2. **Correctness**
   Given a crossword generating program can we prove that all possible solutions will be found?

3. **Number of solutions**
   Given a word list and a puzzle geometry how many solutions do there exist?

4. **Word list size**
   Is it possible to design two word lists W_A and W_B, so that W_A is half the size of W_B, but given an arbitrary puzzle geometry both word lists will result in the same number of solution?

5. **Level control**
   Is it possible to adjust the level of a puzzle? Is is possible to make themed puzzles?

6. **Degree of difficulty**
   Is it possible to define a (simple) metric that unambiguously measures how hard a given puzzle geometry is to fill in?

7. **Human nature**
   Is it possible to generate inconsistent puzzles using human like style and with sensible clues not dull but with human wit and genius?

8. **Other applications**
   Do there exist other problems but the crossword puzzle problem which can be solved with a *crossword compiler*? And, will a *crossword compiler* perform better than previous solutions to such problems?

These questions and other topics will be discussed and answered. Crossword puzzles will never be the same again once you are through reading this. Enjoy.

0.2 Overview

> **A wise man can see more from the bottom of a well than a fool can from a mountain top.**
> *Unknown*

This thesis consists of eight chapters

1. **Chapter 1** Three introductions to crossword puzzles will be given: **(1)** a historical introduction, **(2)** a computational introduction, and **(3)** a definition of the crossword puzzle problem(s).

2. **Chapter 2** The dictionary will be described. Words will be defined, created, selected, matched, and organized. Organizing words include, among other things, a compaction scheme (Huffman codes and digital search trees will be discussed). Finally, we will design and implement two types of dictionaries: **(1)** a prefix supported dictionary and **(2)** a template supported dictionary. Results and possible improvements will end the chapter.

3. **Chapter 3** This is the largest chapter. We start of with a discussion of man vs. machine. Next we define the crossword puzzle problem as a combinatoric problem followed by references to previous work in the crossword compiler field. Grid walks are then defined and five predefined grid walk heuristics are proposed [1]. A dynamic grid walk will be discussed too. Using grid walks we redesign previous work - we use grid walks to simulate earlier approaches to the problem.
   Prefix walks and non-prefix walks are introduced and predefined walks vs. dynamic walks are discussed. Then we prove the correctness of predefined prefix grid walks. The dynamic prefix grid walk was an exceedingly hard problem to break and how

[1] Grid walks and walk heuristics are defined on page 40 and page 41 respectively.

we solved it is described in the next section followed by another correctness proof. A discussion on grid walks and search trees follows and towards the end of the chapter the implementation of the general crossword data structures and grid walk heuristics are presented, and, finally, a display of computer generated crossword puzzles is given.

4. **Chapter 4** In this chapter two benchmark schemes will be used on the developed programs. The tests will verify the correctness of the programs, manifest the worst case runs, and the average case ditto. Comparisons with previous work and between our own programs will be done.

5. **Chapter 5** Estimating the output when given a well defined input will be examined here. A possible extension of an existing analytical formula will be discussed and the possible prediction of puzzles, containing words from natural languages, will be looked at.

6. **Chapter 6** Some solution sets are hard to find - simply because they are so rare. We will look at **(1)** *square puzzles*, **(2)** *cubic puzzles*, **(3)** *Crozzles*, and **(4)** *number puzzles*.

7. **Chapter 7** We will briefly mention clue generation. This area represents a project in itself, but it is included here in order to complete the crossword puzzle compilation problem. English cryptic clues are the main clue type we will look at, but a small selection of Danish clues will finish of the chapter.

8. **Chapter 8** Finally we will sum up the knowledge we have gained and questions raised earlier will be answered.

There are 8 appendices

1. **Appendix A** A list of on-line dictionaries.

2. **Appendix B** The Prefix Dictionary Source Code.

3. **Appendix C** The Template Dictionary Source Code.

4. **Appendix D** The Cross Module Source Code.

5. **Appendix E** The Walk Heuristics Source Code.

6. **Appendix F** The Back Heuristics Source Code.

7. **Appendix G** The Timer Module Source Code.

8. **Appendix H** The Main Program Source Code.

Chapter 1

Introducing crossword puzzles

People recognize what they already have a model for.

Plato

1.1 Historical introduction

> **This is a puzzling world**
>
> *George Eliot*

Crossword puzzles are said to be the most popular and widespread word game in the world, yet have a short history. The first crosswords appeared in England during the 19th century. They were of an elementary kind, apparently derived from the word square, a group of words arranged so the letters read alike vertically and horizontally, and printed in children's puzzle books and various periodicals. In the United States, however, the puzzle developed into a serious adult pastime.

The first known published crossword puzzle was created by a journalist named Arthur Wynne from Liverpool, and he is usually credited as the inventor of the popular word game. December 21, 1913 was the date and it appeared in a Sunday newspaper, the *New York World*. Wynne's puzzle differed from today's crosswords in that it was diamond shaped and contained no internal black squares. During the early 1920's other newspapers picked up the newly discovered pastime and within a decade crossword puzzles were featured in almost all American newspapers. It was in this period crosswords began to assume their familiar form. Ten years after its rebirth in the States it crossed the Atlantic and re-conquered Europe.

The first appearance of a crossword in a British publication was in *Pearson's Magazine* in February 1922, and the first *Times* crossword appeared on February 1 1930. British puzzles quickly developed their own style, being considerably more difficult than the American variety. In particular the cryptic crossword became established and rapidly gained popularity. The generally considered governing rules for cryptic puzzles were laid down by A. F. Ritchie and D. S. Macnutt.

These people, gifted with the ability to *see* words puzzled together in given geometrical patterns and capable of twisting and turning words into word plays dancing on the wit of human minds, have since constructed millions of puzzles by hand and each of these puzzlers has developed personal styles known and loved by his fans. These people, known as human crossword compilers, have set the standard of what to expect from a quality crossword puzzle.

In the mid seventies L. J. Mazlack was the first to create simple crossword puzzles using a computer. His work was continued by other computer scientists and several approaches have been developed since for the purpose of creating crossword puzzles constituting the same high quality level as those made by hand. A lot of computational tasks met in the conquest have been examined and solved.

In the mid eighties the first computer constructed crossword puzzles were found good enough for publication. These computer generated puzzles were for many indistinguishable from those created by people. For professionals, however, there was and still is grave concern that technology will produce bland, standardized puzzles, without any of the wit or nuance that made crosswords the favorite pastime for millions.

Today crossword puzzles have been computerized even further - crossword puzzles are now on display and ready to solve on-line on the Internet and now reaches more people than ever before.

1.2 Computational introduction

> **Puzzlers, take note: the biggest clue to doing crosswords in the '90s may be a nine-letter word that starts with a "C" – computers**
> *New York Times*

Computerized crossword compilation involves many areas of computer science. It addresses known techniques in new contexts and at the same time asks for development of new terminologies and methods. Different successful (and unsuccessful) approaches have shown that the creation of crossword puzzles involves aspects from *database design, artificial intelligence, computational linguistics, logic, linear/integer programming, combinatorics*, and *mathematical analysis* among others.

Database design Independent of the chosen approach a set of words must be at hand. This set is usually quite extensive in order to ensure the existence of a puzzle solution. Saving space and time is often required necessitating the word list to be compressed, but in such a way it is still readily accessible.

Artificial intelligence A large part of crossword compilation touches elements taken from artificial intelligence. Often given a scenario the computer must be able to choose between an optional set of possible continuations and among these choose an 'optimal' continuation. Since it is not always possible to define an optimal continuation we have to tell the computer to choose what *seems* to be the best way - and let it keep this decision in mind for later usage.

The main reason for a computer compiled crossword puzzle to fail publication is the inconsistencies it often possesses. It simply lacks style and 'human touch'. Artificial intelligence can be used to soften the *mechanical* process and add human ideas and aesthetics.

Computational linguistics Clue processing asks for even more advanced structures and bindings between words within the word database. Here word hierarchies known from computational linguistics and speech recognition are possible areas of interest. Grammars for cryptic type puzzles have been searched for and developed.

Logic, linear/integer programming and combinatorics The crossword puzzle problem can be formulated in many ways and it has been found solvable with well known approaches such as logical frameworks, linear programming and combinatorics.

Mathematical analysis Given a word list and a grid configuration we could ask the following questions: how many different solutions can we expect to find? And how long will it take to find them all? The first question is independent of method and machine whereas the second is not. Using mathematical estimates it has been possible to predict answers to both questions.

1.3 Defining the problem

> **An undefined problem has an infinite number of solutions.**
>
> *Robert A. Humphrey*

We will first define the crossword puzzle and later on the *crossword puzzle problem* (**CPP**).

Definition 1.1 (Grid configuration) A crossword puzzle is defined upon an $m \times n$ grid where most, if not all, of the cells are destined to be filled in with characters which comprise words along horizontal and vertical axes.

Definition 1.2 (Open and closed cells) An *open cell* is a blank box destined to contain a character in the final solution of the entire puzzle. A *closed cell* appears as a solid box, does not contain any character and is not actually part of the puzzle but indicates an internal border.

Definition 1.3 (Word slot) Contiguous open cells read from left to right or from top to bottom constitute words, and these contiguous cells are referred to as *word slots.*

Puzzles may be described by at least three characteristics: *geometry*, *density* and *degree of interlocking.*

Definition 1.4 (Geometry) The *geometry* of the puzzle is defined by the size and shape of the grid and the distribution of open and closed cells.

Definition 1.5 (Density) The *density* of a puzzle refers to the percentage of open cells in the puzzle. If no closed cells exist the puzzle is called a *full puzzle.*

Definition 1.6 (Degree of interlocking) The *degree of interlocking* in a puzzle is the percentage of shared cells. A *shared cell* is an open cell belonging to both a vertical and a horizontal word slot. The cell in which two word slots intersect is called an *orthogonal intercept.* If all open cells in a puzzle are shared, the puzzle is *completely interlocked.*

Crossword compilation refers to the stages in the creation of crossword puzzles, and with the terminology above, it can be separated into 6 different stages [Ber87, HB89]:

(1) **creation of the host grid,**
(2) **determination of the overall design,**
(3) **specification of word slots,**
(4) **identification of shared cells,**
(5) **construction of one or more solution sets, and**
(6) **composition of a clue set for each solution set.**

The word list The words we insert into the word slots are strings of symbols from the alphabet and the strings constitute words that are recognized in acknowledged dictionaries. Dictionaries could be (and will be) used as a word list without any further preparation, but normally they need some trimming here and growing there.

We are now ready to define

Definition 1.7 (The crossword puzzle problem, CPP) Given a word list and a grid configuration a crossword compiler, man or machine, should find one or more solutions. A solution in this context is a filling of the grid with words all belonging to the specified word list.

Step (5) above *is* solving the CPP.

It is possible to sub-categorize the CPPs

(1) **Unconstrained CPP:** the position of the black squares are decided during run time.

(2) **Protected CPP:** some words are defined before run time and are part of the final solution(s).

1.3.1 Crossword puzzle research areas

Crossword compilation within a computer aided context includes at least six research areas:

(1) **Dictionary:** word list management
(2) **Construction:** construction of solution sets
(3) **Benchmark:** verifying correctness and comparing algorithms
(4) **Estimation:** estimating the number of solutions
(5) **Search:** searching for rare solution sets
(6) **Clue generation**: developing clues

The following chapters will, in turn, discuss these research areas and hereby introduce the readers to the various aspects of crossword puzzle compilation research.

Chapter 2

The dictionary

I was reading the dictionary. I thought it was a poem about everything.
Steven Wright

2.1 Defining words

> **"When I use a word," Humpty Dumpty said, in a rather scornful tone, "it means just what I choose it to mean – neither more nor less."**
> *Lewis Carroll [Through the Looking Glass]*

Theoretically, words do not have to be words as we know them from crossword puzzles, and theoretically, crossword puzzles do not have to contain words as we know them neither. We will now define what words are within this context.

Definition 2.1 (Alphabet) An *alphabet* V is a finite set of elementary objects called symbols.

Examples of alphabets are the alphabet of letters $\{a, b, c, ..., z\}$, the alphabet of decimal digits $\{0, 1, 2, .., 9\}$, the alphabet of binary digits $\{0, 1\}$, the alphabet of DNA bases $\{A, T, G, C\}$, or some other set of symbols.

Definition 2.2 (String) A *string* over an alphabet V is a finite sequence of symbols from V. The set of all strings over V is denoted V^*.

Definition 2.3 (Language) A *language* L over an alphabet V is a subset of V^*.

Now, words that we will insert into crossword puzzles belong to a word list W, where W is a subset of a language L over an alphabet V. These subsets we will term *natural languages.* The natural language we will use is the English or the Danish language.

2.2 Creating word lists

> **The most valuable of all talents is that of never using two words when one will do.**
> *Thomas Jefferson*

The total number of different strings over the English alphabet $E = \{a, b, c, ..., z\}$ where $|E| = 26$ is

$$\sum_{i=1}^{n} 26^i$$

The longest word in the Oxford English Dictionary is *Floccinaucinihilipilification* which is 29 letters long and assuming no other words exist being longer we can set $n = 29$ in the above formula. E^* with $n = 29$ we write $E^{*(29)}$ and it contains all existing English words (as well as most Danish, German, French, ...) and **a lot** of rubbish. This is because of the size of $E^{*(29)}$ ($E^{*(29)}$ is **large**, of order 10^{41}!) and never will (or can) be created. For benchmark purposes [HB90a, L.J90] its within reason to artificially create **small**, complete subsets of $E^{*(29)}$. Real word lists used in crossword compilation, however, can only be created by hand.

Ever since man began to communicate orally, new words have emerged developing the natural languages as we know them today. This development is fueled by the elements of chaos within natural languages; natural languages are historically unstable and subject to unpredictable changes. Later man began to write (and read) and among lyrics, essays and poems he also made records (word lists) of the words he knew and of those new words he invented. These records have blueprinted and stabilized the natural languages ever since.

Today all sorts of records in the forms of dictionaries, spelling lists, names' lists, etc. are printed in books - and even better: exist in electronic medias. As a crossword compiler programmer it is, therefore, not a big deal to get hold of **hand-picked** words; the real problem is to *select* the words that will produce **good** crossword puzzles.

2.3 Selecting words

> **If you kept out *all* words that might offend someone, somewhere, you'd be left with nothing!**
>
> *Frank Longo*

L. J. Mazlack [Maz76a, Maz76b] showed that a word list of less than 2000 words was insufficient to compile even small puzzles and he suggested that a larger word list would increase the probability of success. This was verified by **(1)** C. Long [C.L92] and by **(2)** G. H. Harris and J. J. H. Forster [G.H90b] who both made estimates for the number of solutions to crossword puzzles. Intuitively this comes as no surprise, but at the same time we may wonder why adding two sets containing 100 words each will not increase the probability for success equally well. The explanation is the non-uniform distribution of letters in natural languages which was mentioned in both papers but not, however, incorporated into any of the estimates they made.

A few questions may be put forward: **1)** is it possible to calculate the exact increase in performance when adding one particular set of words to a known word list, **2)** is it possible to find and remove words which for some reason are not part of any solution (or very few) and **3)** what is the best [*number of solutions/dictionary size*] - ratio. We could perform the following tests and find the answers.

Tests

1. Measure the success increase when using a small word list with selected words ([*vowel/consonant*] characterized) and then add different types [1] of word sets.

2. Count the number of times a word is used in a solution. Examine the words with small counts.

3. Define

$$\delta = \frac{number\ of\ solutions}{word\ list\ size}$$

As we see it, a large δ will result in a better dictionary. Maximizing δ can be done by removing words found in **2)** and adding good words found in **1)**. These tests can be used to produce a measure for the *weaving* property of a word list based on size and [*vowel/consonant*] characteristics.
This will not, however, please no one but a computer scientist. A good word list with a high success rate can not necessarily produce aesthetically good puzzles (usually good puzzles do not allow abbreviations, *s*-words [2] , etc.). Man, once again, needs to hand pick words and this time the words must be selected among words which are common, interesting and personal [3] . Such lists do exist [4] but none have yet been approved simply because words have personal meanings and may change characteristics in time.

2.4 Matching words

If a word in the dictionary were misspelled, how would we know?
Steven Wright

Dictionaries are word lists with added structures enabling fast accesses to all words with certain properties and/or with certain statistical characteristics regarding the current state of the dictionary. Often it is not possible to separate dictionary design issues from the design of the chosen crossword compiler algorithm; the dictionary must be more or less structured according to which queries it may have to answer.

Definition 2.4 (Prefix) A string x is called a *prefix* of y if there exists a string $z \in V$ so that $y = xz$. If z is different from the empty string x is called a *proper prefix* of y.

Definition 2.5 (Pattern) Let $s = s_0 s_1 ... s_m$ and $p = p_0 p_1 ... p_n$, $m \geq n$, be strings. p is a *pattern* of s if $\exists k, 0 \leq k \leq (m - n)$ so that $p_j = s_{j+k}$ for $j = 0, 1, ..., n$.

Definition 2.6 (Overlap) Let $s = s_0 s_1 ... s_m$ be a string. Two patterns $p^1 = p^1_0 p^1_1 ... p^1_u$ and $p^2 = p^2_0 p^2_1 ... p^2_v$ *overlap* in s if

[1] 'Types' refer to the origin of the words - are they taken from fiction or non-fiction.

[2] *s*-words are fx. the car*s*, the man*s*, or the programmer*s*.

[3] Phrases can be allowed too in word lists. Allowing phrases in lists will produce unexpected letter sequences in the grid which may confuse the solver but in a good way.

[4] E.g. The UK Advanced Cryptic Dictionary (UKACD), a comprehensive list of English words and phrases, maintained by Ross Beresford at *ftp://ftp.simtel.net/pub/simtelnet/win3/homeent/ukacd14.zip*.

$\exists k, k_1, k_2, 0 \leq k_1 \leq u, 0 \leq k_2 \leq v : s_{k+i} = p^1_{k_1+i} = p^2_i, 0 \leq i \leq k_2$ or $s_{k+i} = p^2_{k_2+i} = p^1_i, 0 \leq i \leq k_1$.

Definition 2.7 (Template) Let $s = s_0 s_1 ... s_m$ be a string and let Φ be a set of strings. Φ form a *template* of s if

all $\phi \in \Phi$ are patterns of s, and

no two strings $\phi_1, \phi_2 \in \Phi$ overlap.

Notice that if two patterns ϕ_i and ϕ_j overlap they can be replaced by a new pattern ϕ_{ij} which is a merge of the two overlapping patterns.

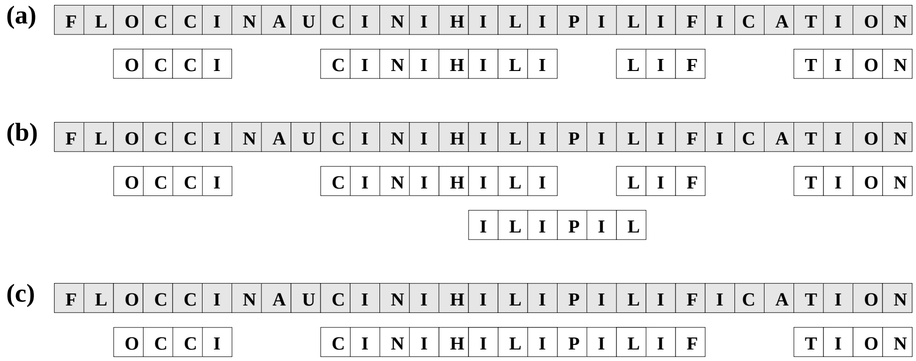

(a)
F L O C C I N A U C I N I H I L I P I L I F I C A T I O N
O C C I
C I N I H I L I
L I F
T I O N
(b)
F L O C C I N A U C I N I H I L I P I L I F I C A T I O N
O C C I
C I N I H I L I
L I F
T I O N
I L I P I L
(c)
F L O C C I N A U C I N I H I L I P I L I F I C A T I O N
O C C I
C I N I H I L I P I L I F
T I O N

Figure 2.1: **(a)** *Four patterns with no overlaps,* **(b)** *five patterns with two overlaps and* **(c)** *the patterns from (b) transformed into a set of non-overlapping patterns.*

2.5 Organizing words

Don't agonize. Organize.

Florynce Kennedy

A word list, we have seen, must contain a reasonable number of words in order to ensure a reasonable number of crossword puzzle solutions. Main memory was sparse and precious when the first computer compilations of crossword puzzles were attempted and word lists had to be kept small in order to avoid and/or limit time consuming accesses to secondary storage (in those days: magnetic tape). This is why the first selected word lists were small (Mazlack used 2000 words [Maz76a, Maz76b], Feger 6000 [O.F75]) whereas today's word lists easily reaches 100.000 words or more. Machine power and memory capacity have improved considerably over the years but two key questions still persist: **(1)** data compression and **(2)** fast data access. These two operations seem to exclude one another but we will now argue that this does not have to be the case.

2.5.1 Huffman encoding

Various data compression schemes exist most of which are used on data files for the purpose of minimizing the storage needs on secondary medias. For a brief moment we will look at one of these file reduction schemes and see how it works.

Huffman encoding is a widely used and very effective technique for compressing data; savings of 20% to 90% are typical, depending on the characteristics of the file being compressed. Letters are converted into a *binary character code* which is either *fixed-length* or *variable-length.* The first type uses the same bit length for all encodings whereas the last does not.

Example 1: Imagine we have an artificial word list containing, say, 20000 words of length 5, that is a total of 100.000 letters. If the alphabet contains 6 symbols, {`a`,`b`,`c`,`d`,`e`,`f`}, then a fixed-length code needs 3 bits to represent the symbols. In all the method requires 300.000 bits to code the entire word list.

A variable-length code uses the frequencies of occurrence of each character to build up an optimal way of representing each character.

Definition 2.8 (Prefix code) Codes in which no codeword is also a prefix of some other codeword are called *prefix codes.*

Optimal data compression can always be achieved with a prefix code, and an optimal code for a file is always represented by a *full* binary tree, T. The total number of bits required to encode a file is

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

where $f(c)$ denotes the frequency of the letter c and $d_T(c)$ is the depth of the letter c in the tree T. $B(T)$ is called the *cost* of the tree T. Huffman invented a greedy algorithm capable of constructing an optimal prefix code known as the *Huffman code.* The algorithm will not be examined here.

Example 2: If the frequencies are: `a`= 45000, `b`= 13000, `c`= 12000, `d`= 16000, `e`= 9000 and `f`= 5000, a variable-length code reduces the bit count to 224.000 bits.

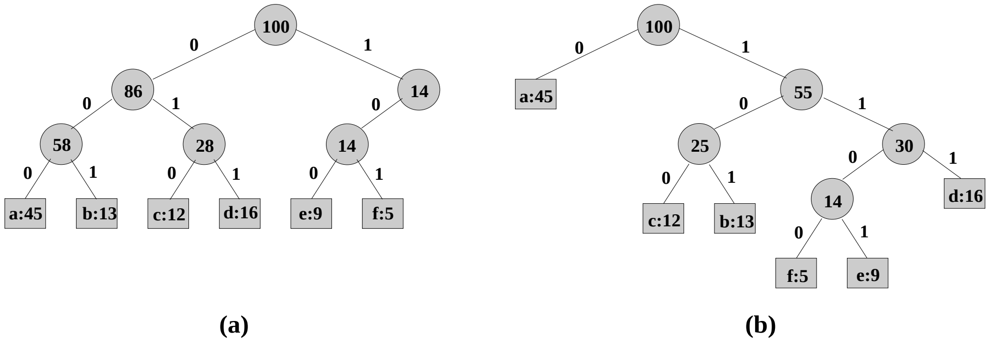


Figure 2.2: *Trees corresponding to the two coding schemes:* (a) *fixed-length and* (b) *variable-length.*

File reductions on `Combi.txt`		
Compression scheme	Original size	Reduced size
Huffman	400942	214498
Adaptive Lempel-Ziv coding	400942	174064
Zip	400942	118309

Table 2.1: *Table showing the reduction of a file using three different file reduction schemes.*

Most frequently appearing letters overall		
1	Webster's Second Unabridged[5]	`eiaorn tslcup mdhygb fvkwzx qj`
2	Several gigabytes of Usenet traffic[6]	`etaoin srhldc umpfgy wbvkxj qz`
3	The UK Advanced Cryptic Dictionary[7]	`esiart nolcud pgmhby fvkwzx qj`

Table 2.2: *Table showing the most frequently appearing letters in different kinds of text.*

The Huffman code, we have seen above, can reduce memory requirements considerably when used on data files. The Huffman file reduction scheme reduces each character in the file separately and therefore we have a 1-1 relation between the actual character and the equivalent encoding. Can we use these codes internally in the main memory and save memory space? The answer is unfortunately no. We will soon argue that saving bits costs bytes ...

2.5.2 Digital search trees

Huffman coding was not a solution to the data compression scheme needed. Now, we will look at digital search trees instead and show that they solve both the storage problem and the fast access problem in one solution.

Definition 2.9 (Binary tree) A *binary tree*, T, is a structure defined on a finite set of nodes that either

1. contains no nodes, or
2. is comprised of three disjoint sets of nodes: a *root*, a binary tree called its *left subtree* and a binary tree called its *right subtree.*

In the following a *search key* is defined as

1. the empty string, or
2. the string $s = s_0 s_1 ... s_n$ where n is finite

and each letter, s_i, in the string $s = s_0 s_1 ... s_n$ is stored in a node in such a way that if $i > j$ then s_i belongs to one of the subtrees to s_j and if $j > i$ then s_j belong to one of the subtrees to s_i.

Definition 2.10 (Binary search tree) A *binary search tree* is a binary tree stored in a such way that: If x is a node in the search tree and if $\{y_0, y_1\}$ is the two binary subtrees of x, then $key[y_i] \leq key[x]$, $i = 0, 1$. If $key[y_0] < key[y_1]$ then y_0 is the left sibling to y_1, and if $key[y_1] < key[y_0]$ then y_0 is the right sibling to y_1.

[5] **Source:** The rec.puzzles archive at *http://einstein.et.tudelft.nl/ arlet/puzzles/sol.cgi/language/english/frequency*

[6] **Source:** The rec.puzzles archive at *http://einstein.et.tudelft.nl/ arlet/puzzles/sol.cgi/language/english/frequency*

[7] **Source:** *ftp://ftp.simtel.net/pub/simtelnet/win3/homeent/ukacd14.zip*

Definition 2.11 (Lexicographical) Given two strings $a = a_0a_1, ..., a_p$ and $b = b_0b_1, ..., b_q$ over the same alphabet, V, we say that a is *lexicographically less than* b if either

1. there exist an integer j, $0 \leq j \leq \min(p, q)$, so that $a_i = b_i$ for all $i = 0, 1, ..., j - 1$ and $a_j \leq b_j$, or
2. $p < q$ and $a_i = b_i$ for all $i = 0, 1, ..., p$.

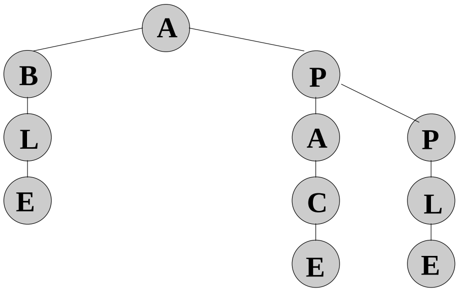
Definition 2.12 (Digital tree) A *digital tree*, T, is a structure defined on a finite set of nodes that either

1. contains no nodes, or
2. is comprised of three or more disjoint sets of nodes: a *root* and two or more *digital subtrees.*

Digital trees are also known as *tries.*

Definition 2.13 (Digital search tree) A *digital search tree* is a tree stored so that: If x is a node in the search tree and if $\{y_0, y_1, ..., y_n\}$ is the set of digital subtrees of x, then $key[y_i] \leq key[x]$, $i = 0, 1, ..., n$. If $key[y_i] \leq key[y_j]$ then y_i is a left sibling to y_j and if $key[y_j] \leq key[y_i]$ then y_j is a right sibling to y_i.

Example: Depicted to the right is the lexicographically sorted set of strings {`able`, `apace`, `apple`} stored in a digital search tree.



Placing the words in a digital search tree reduces memory requirements in that all shared prefixes are only stored once.

2.6 Previous work and ours

> **Rem tene, verba sequntur "Keep to the subject, and the words will follow"**
>
> *Cato the Censor*

The words used by a crossword puzzle compiler are stored in a word list and with added structures, references and statistical information it is called a dictionary. If the word list is complete (all possible strings are contained in the word list) we will refer to it as a (totally enumerated) lexicon.

Definition 2.14 (N-Gram Analysis) N-gram analysis is a method of matching text based on the statistical similarity of occurrences of n-grams (n-length combinations of letters) in the text.

N-gram analysis is used in research areas such as *speech recognition*, *database interfacing*, *network communication* and *crossword puzzle dictionary optimization.*

H. Berghel, D. Roach and J. Talburt

> *Approximate String Matching and the Automation of Word Games* [H.B90b]
>
> Berghel et al. describe how N-gram analysis can improve the search for words. N-gram analysis requires that we generate all N-grams of length N, e.g. the word "word" and $N = 3$ produce two trigrams: *"wor"* and *"ord"*. For a word slot: `- O - - -` we must find the list of words of length five having the letter `O` on the second letter position. Most such searches will fail because of the non-uniform distribution of characters within word positions. Filtering dictionary look-ups in advance will prevent some of the failures. If none of the trigrams `- O -` or `O - -` exist we stop because no word(s) exists in the dictionary. N-gram analysis is useful because of the non-uniform distribution of characters within the word positions. □

In the following we will consider two types of dictionary designs

1. **Prefix supported design:** The dictionary is capable of returning words matching a given prefix, if such words exist in the dictionary.
2. **Template supported design:** The dictionary is capable of returning words matching a given template, if such words exist in the dictionary.

A prefix is of course also a template, and, being so, template supported design includes prefix supported design.

2.7 Prefix supported design

> **"Where shall I begin, please your Majesty?" he asked. "Begin at the beginning," the King said, gravely, "and go on till you come to the end: then stop."**
>
> *Alice's Adventures in Wonderland, Lewis Carroll*

Maintaining the word list in a digital search tree has not only solved potential storage problems it has also envisaged a fast access scheme to all words in the dictionary. This is, admittedly, only if we work on prefixes; a given prefix can easily be extended in that the prefix itself is the *key* describing the way down the digital search tree and where it ends all nodes below the last encountered node contain all possible extensions to the given prefix or none if the prefix fails.

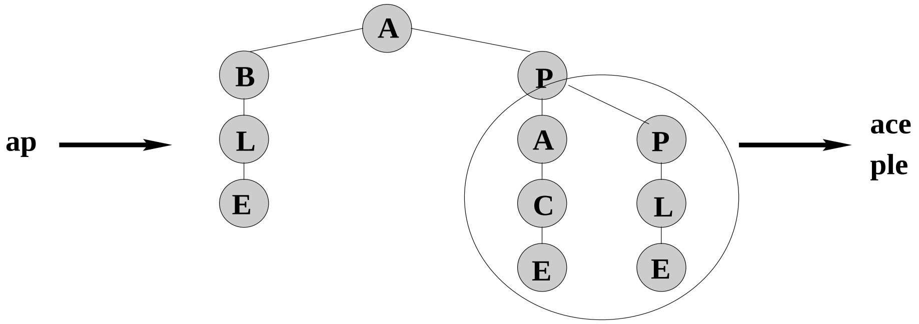


Figure 2.3: *The prefix* **ap** *can be given the suffixes* **ace** *and* **ple***.*

Obtaining fast access to any query other than prefix extensions will require additional dictionary structures.

2.8 Template supported design

> **"Now here, you see, it takes all the running you can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that."**
> *The Red Queen, from "Through the Looking Glass" (Lewis Carroll)*

The structure above showed how extensions to given prefixes were easily found - but what if we are given a template other than a prefix? Can we still obtain fast access to all matches in the word list? The answer is of course confirmative. A template is, as we defined it above, a set of patterns to which we now want to find *a* matching word or even better *a set* of matching words.

A pattern describes the letter configurations for words in that it: **(1)** defines a letter, **(2)** in a given letter position **(3)** within a word of a certain length. That is, we need three parameters to describe the above situation: **(1)** the position of the wanted letter in the alphabet, **(2)** the position of the letter within the word, and **(3)** the length of the word in which the letter is to be found.

If we have the word configuration, `- - j - -` , this can be written as $[10, 3, 5]$. This tuple is then used as a reference to a subset of the word list which fulfills the description. If we have more than one letter, `- - j - n` , the list of words we are looking for is then the intersection of the references $[10, 3, 5]$ and $[14, 5, 5]$.

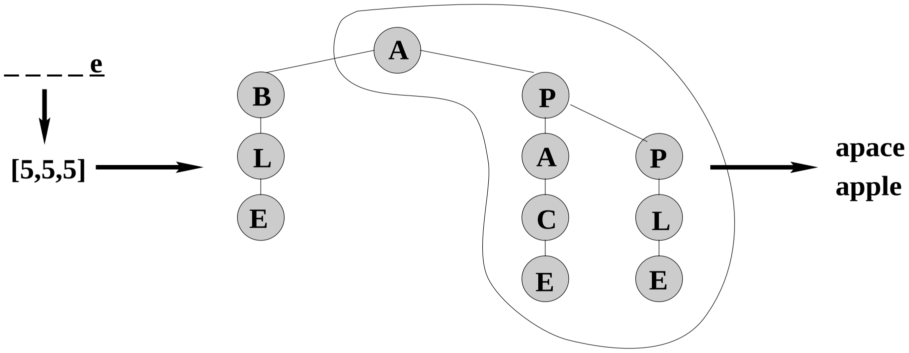


Figure 2.4: *The pattern* `- - - - e` *transforms into the reference* $[5, 5, 5]$ *and the words* **apace** *and* **apple** *are candidates.*

Please, notice that such a dictionary is in fact a crossword solvers favorite tool in that it can suggest matches to any given template - and templates, they occur all the time when solving crossword puzzles.

2.9 Implementing the dictionary

> **As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.**
>
> *Maurice Wilkes discovers debugging, 1949*

In this section we will take a look at how the dictionaries used in this project have been implemented. All implementation has been done using *standard ansi* **C**.

Two designs are present, but first a preliminary discussion on data compression.

2.9.1 Storing the word list in main memory

The memory capacity problems endured in the 'old days' are not so urgently present today. The word lists used in our work have not, but in a few cases, exceeded the capacity of the machinery available and therefore have not given us reason to spend large resources in the quest for compressed word list design implementations. Huffman codes and digital search trees have been considered and digital search trees have been used in practice.

Huffman codes reduce files 20% to 90% where the extreme upper limit is achieved on files represented in small alphabets and with a high degree of character repetition. Natural languages, however, have semi large alphabets and little character repetition giving hope for only small file reductions. Earlier we argued that Huffman file reduction schemes theoretically could be used internally within dictionary designs for the representation of letters. This, however, turns out to be technically unprepossessing since the extra data structures used to administrate Huffman representation internally in the program will cost more (memory space) than if we did not use Huffman code representation. Needless to say, Huffman code representation was rejected.

Digital search trees were on the other hand much more favorable. Starting out as an idea to store the words readily at hand, it turned out that sharing prefixes did save not a few bytes here and there but a lot of bytes almost everywhere. Counting nodes in the search trees have shown that up to 71% could be saved. This was considered good enough - no further data compression was investigated. Maintaining the word list in *one* digital search tree will result in an optimal saving scheme. This, however, makes it difficult to tell whether a certain prefix also is a word. Searching for a three letter word starting with an `e` might result in `equ` (since `equ` is a prefix of `equation`) but `equ` should not be valid. In order to avoid these kinds of mismatches it was chosen to maintain a digital search tree for each word length found in the word list. By measuring the savings it was found that even with 26 digital search trees instead of just one the savings [8] were still as high as 37%. We can therefore conclude that digital search trees are good data compression schemes while at the same time enabling easy (prefix) accesses to all words in the word list.

[8] Using the UKACD word list containing 177642 words.

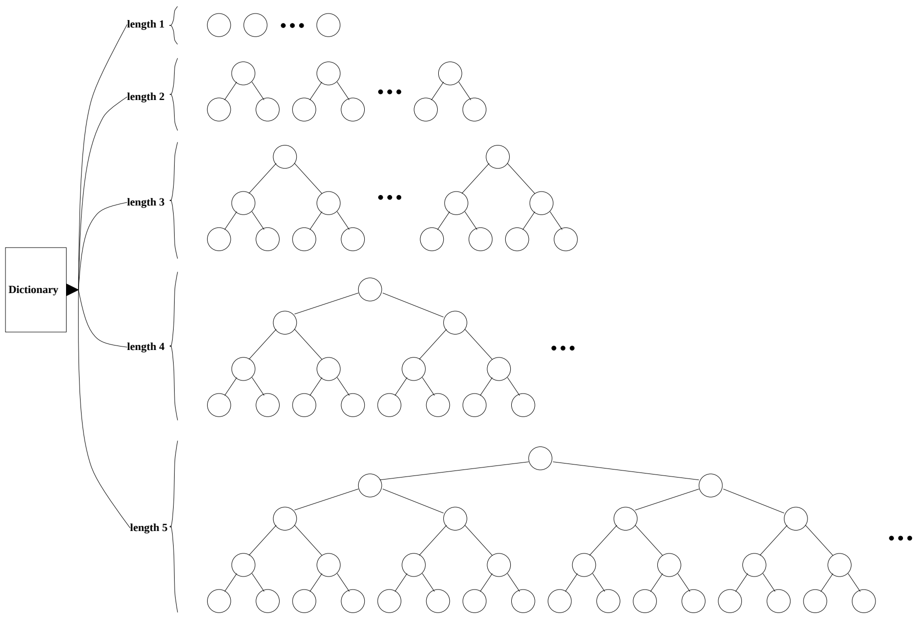


Figure 2.5: *Words of length 1 to 5 are stored in separate digital search trees.*

2.9.2 Data structures

In this section we will discuss how the digital search tree is implemented and how words are inserted and found.

The dictionary has been designed to accommodate words of length 1 to words of length 30. Words longer than 30 letters are very rare and those that might exist are very unsuitable for crossword puzzles anyway. An array of 30 pointers has therefore been made, making it easy to start the search for a word of any length (1-30).

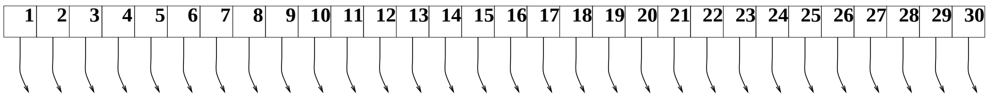


Figure 2.6: *An array of pointers pointing to the 30 digital search trees containing the words of length 1 to length 30.*

The nodes in the search trees can be implemented two ways: **(1)** maintain 26 pointers in the node; one pointer for each of the potential children, or **(2)** maintain a pointer to a sorted list containing the actual children and a pointer to the left sibling and a pointer to the right sibling.

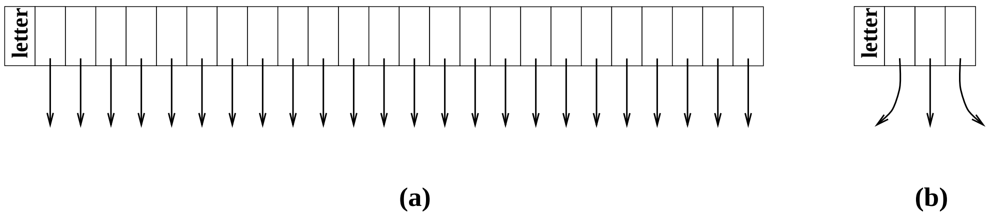


Figure 2.7: **(a)** *A node containing a 'letter' and 26 pointers to the 26 potential children and* **(b)** *a node with a 'letter' and a pointer to the left and right sibling plus a pointer to a sorted list of children.*

Method number 1 is easily seen to be a waste of pointers which the following example illustrates

Example: For a word list with 4101 words of length 7 it has been shown [9] that 15387 nodes were needed to create the digital tree. Method one above uses 106626 pointers whereas method 2 only uses 46161, a saving of 57%.

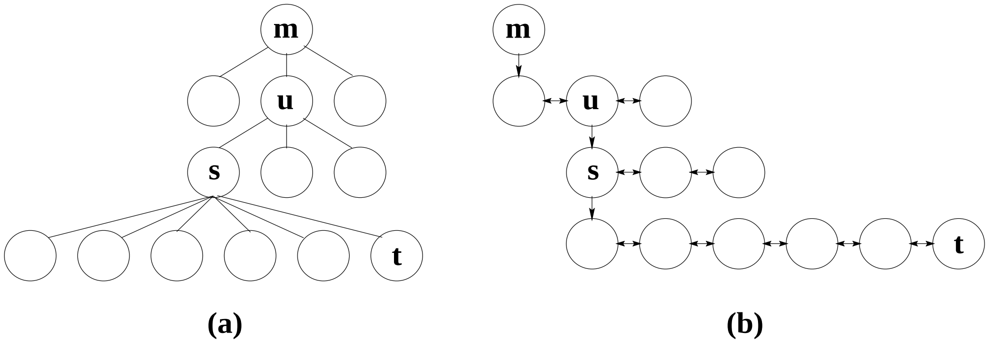


Figure 2.8: *The digital search tree without lists* (**a**) *and with lists* (**b**).

Insertion of a word from the word list in the dictionary can now be done as depicted in the following figure

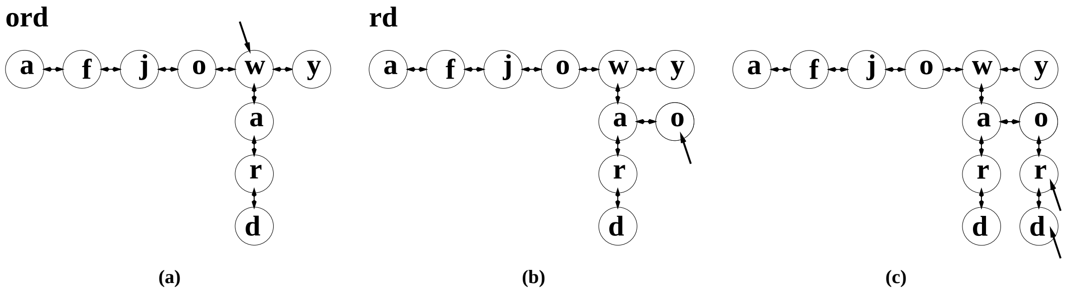


Figure 2.9: *The word* `word` *is to be inserted in the digital search tree. In* (**a**) `w` *is found in the first list, in* (**b**) `o` *does not exist in the second list and a new node is therefore inserted, and in* (**c**) *the nodes with* `r` *and* `d` *are appended. (Note that the tree only shows the nodes involved - the tree is not shown in its entirety.*

So far we have only considered the positive aspect of using lists (method 2 above) but it has got a negative side too. Using method 1 the number of nodes visited during a search will maximum equal the number of letters in the word whereas the maximum number in method 2 is $n \times 26$, where n is the length of the word. The reason for this is that we may have to search all the way through the whole list before finding the right letters - but will this ever happen? Theoretically yes and practically no. If we should hit maximum then the word we are looking for must be **zzz...z** and this will only happen for $n = 1, 2$ [10] . Of course using lists increase the number of nodes traversed but the overhead will be within acceptable limits. The overhead can be reduced if we sort the lists dynamically, that is we make sure that the most wanted letters are up front and the least wanted down below. We have, however, settled on alphabetically sorted lists which have shown good results.

[9] See the table in the section on results.

[10] Three and more identical letters are rarely desirable words.

The data structures so far are only geared to support a prefix dictionary search, so in order to support a template dictionary search we need additional structures.

Template supported dictionary design: Given the *length*, the *letter position* and the *letter position in the alphabet* enable us to implement a data structure that supports templates. Each word length (1 to 30) is given an array of size equal the length. Each array entry is set to point at another array of size 26 which is the number of letters in the alphabet, and, finally, each of these last 26 array entries points at lists of matching words, if any. Within each word list all words are of the same length and have the same letter in the same letter position. Words are still stored in a digital search tree but now all words are kept in just one tree. When inserting a word each letter in the word results in a word list entry.

Example The word `must` belong to four lists: the list of words of length 4 and with an `m` in the first letter position, the list of words of length 4 and with a `u` in the second letter position, etc.

The words are identified unambiguously in the digital search as they are read from the bottom and up i.e. backwards. This has made it necessary to add pointers to each node which is set to point at the parent node. The problem we had before, when specifying whether a prefix also was a word, no longer exist.
When matching a template it is now possible to look up one or more lists (one list for each given letter in the template) and then select those words belonging to all of the lists.

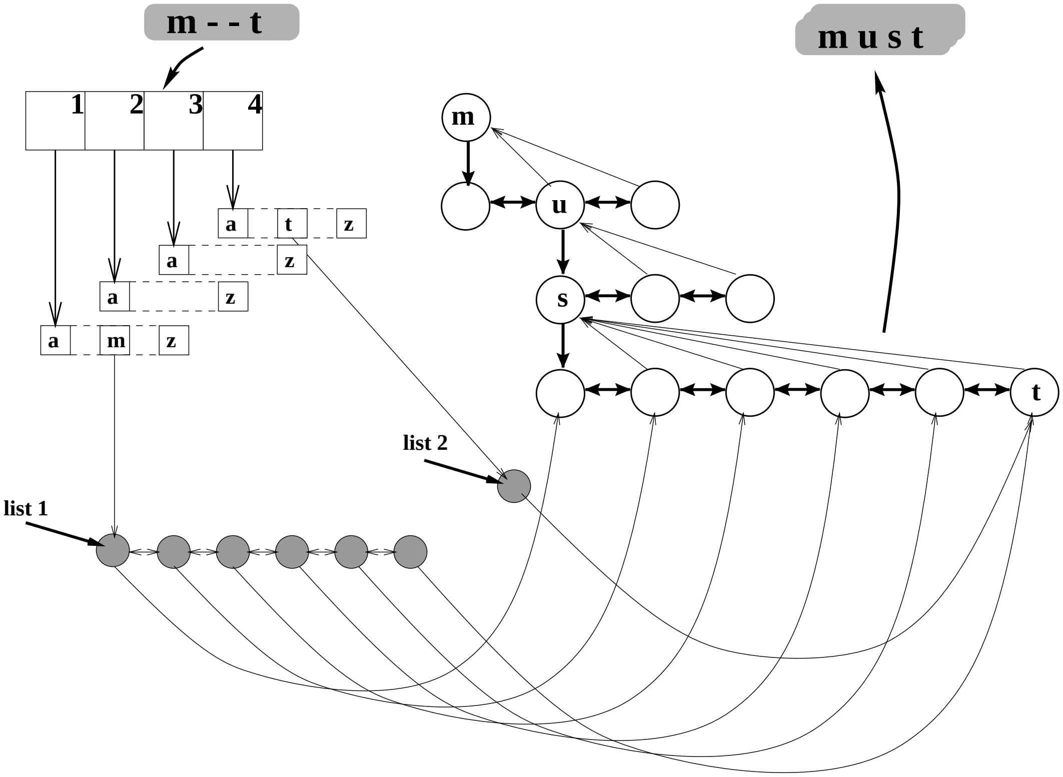


Figure 2.10: *Given the template* `m - - t` *the following happens: (1) the list containing the words with an* `m` *in the first letter position,* **list 1**, *and the list containing the words with a* `t` *in the fourth letter position,* **list 2**, *are found, (2) the list of words belonging to both lists matching the template is created.*

2.10 Results

> **Results are what you expect, and consequences are what you get.**
>
> *Unknown*

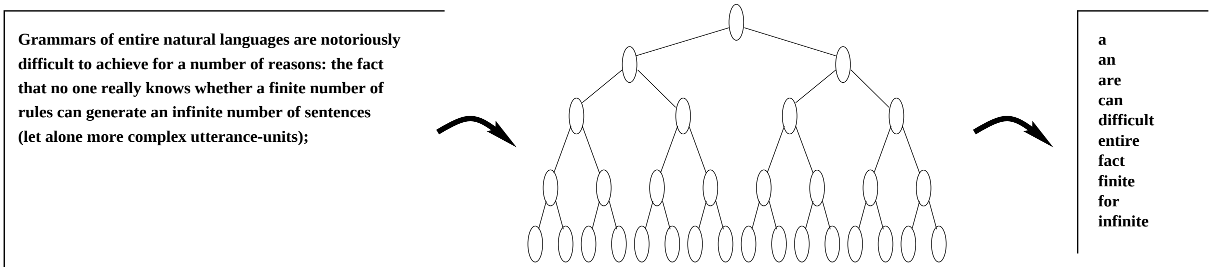
In this section we will look at how word lists have been created/chosen and how well they have been compressed.

2.10.1 Creating word lists

Making the word lists can be done in at least three ways: **(1)** by typing in the words, **(2)** by extracting words from electronic texts, or **(3)** by using existing word lists. We have tried all three methods with varying success.

Typing in the words by hand is no doubt a big job to do especially if you need several thousands of them. The advantage is, however, that you decide which words should be part of the list. We tried (merely for fun) to type in a couple of hundred words which had all been used in published crossword puzzles [11] hereby creating a quality crossword puzzle word list.

Extracting words from electronic texts were done using the same digital tree structure as the one described above. Scanning the electronic text the words in it are inserted into the digital search tree one after the other but each different word only once. When finished the words can be extracted alphabetically from the tree and written into a word list.



Grammars of entire natural languages are notoriously difficult to achieve for a number of reasons: the fact that no one really knows whether a finite number of rules can generate an infinite number of sentences (let alone more complex utterance-units);

a
an
are
can
difficult
entire
fact
finite
for
infinite

Figure 2.11: *The words are inserted in the digital tree. Words are only stored once and an alphabetical printout is easy to make.*

Electronic texts in many forms can be found on the Internet. The books in table 2.3 have been found at The Book Wire Reading Room [12]

[11] Weekendavisens expert krydsord (A danish crossword puzzle).

[12] http://www.bookwire.com/links/readingroom/readingroom.html

Fiction				
Author	Title	#words	#different	%diff
Jane Austen	Sense and sensibility	119449	6508	94%
James Matthew Barrie	Peter Pan	47796	5145	87%
George Byron	Don Juan	126889	13808	89%
Lewis Carroll	Alice in wonderland	28195	3063	89%
Charles Dickens	A Christmas carol	28351	4431	84%
Conan Doyle	Hound of Baskervilles	59611	5980	90%
Rudyard Kipling	The Jungle Book	51440	4889	90%
Herman Melville	Moby Dick	212095	17691	92%
Bram Stoker	Dracula	160434	10284	94%
Leo Tolstoy	Anna Karenina	351511	13959	96%
Sun Tzu	The art of war	55728	6662	88%
TOTAL		1241499	30351	97%

Table 2.3: *The difference between the number of words and the number of different words in fiction books.*

Non-Fiction				
Author	Title	#words	#different	%diff
Hakim Bey	Temporary Autonomous Zone	41374	9334	77%
Norman Coombs	The Black Experience in America	40270	5876	85%
A. Morita & S. Ishihara	The Japan That Can Say No	11399	2493	79%
Ann & Alex Shulgin	PiHKAL: A Chemical Love Story	33369	4133	88%
Lysander Spooner	No Treason	19542	2120	89%
H. Wasserman & N. Solomon	Killing Our Own	162069	13033	92%
TOTAL		308023	22337	93%

Table 2.4: *The difference between the number of words and the number of different words in non-fiction books.*

When extracting words from an electronic text, how good is the resulting word list? The two tables above show how many words a collection of texts contain and how many different words there are in the resulting word lists. It comes as no surprise that the number of different words in a text is much smaller than the total number of words but it has surprised us that the differences are as high as 90% or more. A quick glimpse at the tables above shows that the differences are almost identical for all text files, and that a large text file does not necessarily result in a large word list. There is a slight difference between text files containing fiction and those containing non-fiction but the difference may also be due to smaller non-fiction text files.

It is quite obvious that combining two word lists does not result in a word list equal to the accumulated size, but how much extra do we get then? It can vary a lot. Combining word lists extracted from identical sources (e.g. both fiction) will not result in large new word lists whereas combining word lists extracted from different sources (e.g. fiction and non-fiction) may. The total number of words in the combined fiction and nonfiction word lists is 52688 and the resulting combined word list size is 40858; a reduction of only 22% (compared to 90% when sources are identical).

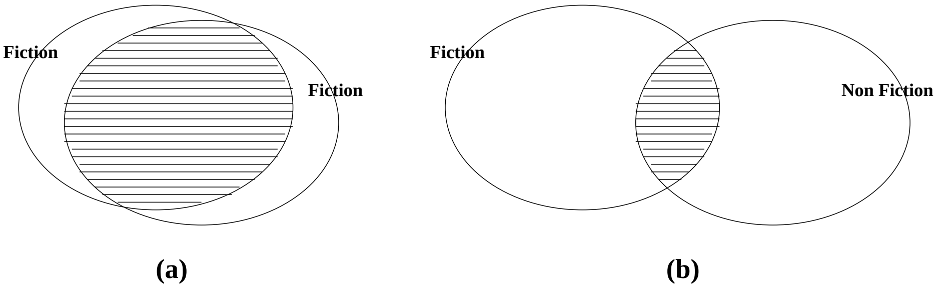


Figure 2.12: *The intersecting set between two word lists both derived from the same source* (**a**) *is large and will thereby cause the resulting word list to gain a little whereas the intersecting set between two word lists derived from different sources* (**b**) *is small causing the resulting list to gain a lot.*

Existing word lists Finally, the easiest method is using already existing files containing pre-compiled word lists. Several such lists exist locally (at IMADA) and on the Internet [13]. We have combined two such lists: **(1)** Locally we have a Danish word list used to check for spelling errors. It contains 25485 words. **(2)** On the Internet we found a Danish word list at Oxford [14] containing 27109 words. Merging the two lists resulted in 37969 words. For the English crossword puzzles the UK Advanced Cryptic Dictionary was used [15] Below is a table showing the distribution of word lengths in these word lists.

Word length	#Local Dict.	#Oxford Dict.	#Combined	#UKACD
1	29	28	29	24
2	165	150	189	152
3	628	628	755	1209
4	1173	1263	1512	4869
5	1964	2080	2609	10438
6	2637	2563	3542	17119
7	2868	3004	4101	23679
8	2839	3075	4224	26773
9	2786	2947	4185	26080
10	2452	2608	3731	22864
11	2012	2085	3100	17091
12	1554	1549	2386	11758
13	1130	1202	1826	7481
14	836	956	1411	4306
15	626	772	1101	2252
16	498	571	858	886
17	388	466	690	396
18	251	344	482	149
19	215	300	416	68
20	143	197	292	27
21	110	128	205	11
22	73	88	135	2
23	49	47	85	2
24	30	25	47	0
25	11	12	22	1
26	11	13	21	0
27	4	5	9	0
28	3	3	6	1
29	0	0	0	1
30	0	0	0	0
TOTAL	25485	27109	37969	177642

[13] See Appendix A

[14] *ftp://ftp.ox.ac.uk:/pub/wordlists/*

[15] *ftp://ftp.simtel.net/pub/simtelnet/win3/homeent/ukacd14.zip*

Artificial totally enumerated word lists have been created too for benchmarking purposes - more about these lists in the section on benchmarking (chapter 4).

2.10.2 Compressing word lists.

The only compaction scheme we have used was digital search trees in which inserted words share prefixes. Each word length has been assigned a separate digital tree and therefore an optimal saving has not been achieved but still the savings have been high. Below is shown a table depicting the file size and the savings for the combined word list.

Combined word list.				
Length	#Words	Uncompressed	Compressed	Reduction
2	189	378	213	44%
3	755	2265	1056	53%
4	1512	6048	2881	52%
5	2609	13035	6487	50%
6	3542	21252	10810	50%
7	4101	28707	15387	46%
8	4224	33792	18813	44%
9	4185	37665	21454	43%
10	3731	37310	22102	41%
11	3100	34100	20897	39%
12	2386	28632	18496	35%
13	1826	23738	15681	34%
14	1411	19754	13409	32%
15	1101	16515	11590	30%
16	858	13728	9931	28%
17	690	11730	8398	28%
18	482	8676	6371	27%
19	416	7904	5757	27%
20	292	5840	4365	25%
21	205	4305	3173	26%
22	135	2970	2249	24%
23	85	1955	1525	22%
24	47	1128	1003	11%
25	22	550	496	10%
26	21	546	519	5%
27	9	243	204	16%
28	6	168	166	1%
TOTAL (28 trees)	37969	363246	223433	38%
TOTAL (1 tree)	37969	363246	106309	71%

It can be seen above that the savings are high when the words are short (length 3 to 6) which is caused by the length of the shared prefixes. The length of the shared prefixes for short words is *relatively* longer than those exhibited by longer words. Long words may share prefixes but the lengths of these are short compared to the total length of the words. The table above shows that the saving is as high as 38% even when we use a separate search tree for each word length (it becomes 71% when stored in one tree).

2.11 Improving performance

> **For a list of all the ways technology has failed to improve the quality of life, please press three.**
>
> *Alice Kahn*

If the dictionary can not be stored in the main memory it is desirable to reduce traffic between main memory and secondary memory. Mazlack [Maz76a, Maz76b] used statistical data to create artificial words which, when completed, were verified in the dictionary hereby eliminating the storage of the dictionary in the main memory altogether.

Berghel et al. [H.B90b] showed how N-gram analysis can be used in the dictionary design. Maintaining all N-grams in main memory instead of the actual words can also reduce traffic.

Improving crossword puzzle quality, level, and theme control can too be done at the dictionary design level. First of all, as mentioned earlier, we can assemble word lists which contain only 'quality' words. Ranking words using qualitative, difficult and thematic measures could be done making it easier to adjust output.

These issues, among other things, were not examined further.

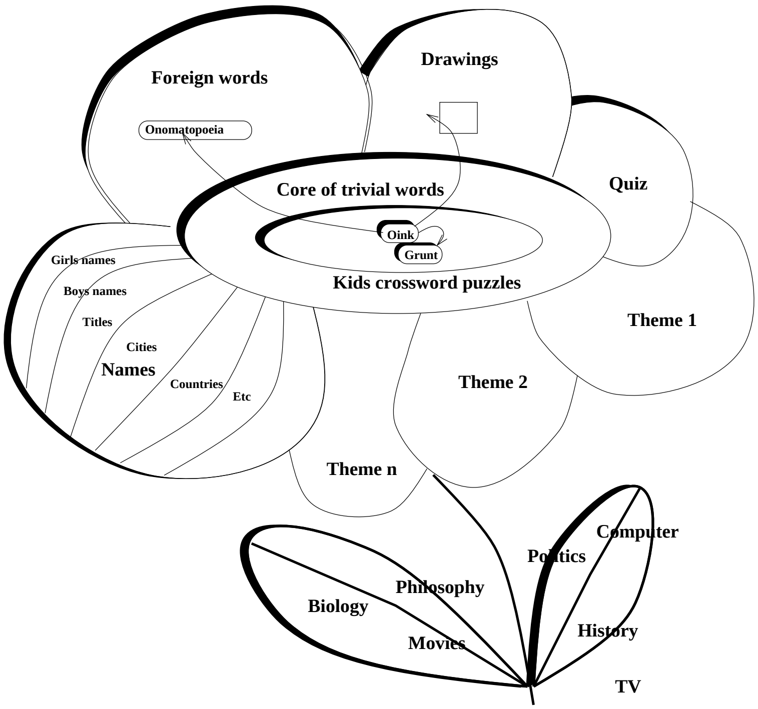


Figure 2.13: *Words can be sorted and separated into several topics improving the flexibility of the crossword puzzle compiler output.*

Chapter 3

Construction

The shortest distance between two points is under construction.

Noelie Altito

This chapter is devoted to crossword compiler algorithm designs and all the various aspects of the creation of the algorithms used, why they are designed the way they are, what they can do, and the ideas inspiring their creation. Despite the variegation in earlier designs we have found that an underlying technique exists which makes it possible to join together a large portion of these techniques into one single design.

First we will discuss the differences between humans and computers in the context of crossword puzzle compilations, then we will look at previous work in this area of crossword puzzle compilation which then lead us to the methods and designs we have developed and tested. Towards the end of this chapter we will discuss cycles and how to prevent them and finally we will end of the chapter with a presentation of computer compiled puzzles.

3.1 Man vs. Machine

> **It may be irrational of me, but human beings are quite my favorite species**
>
> *Doctor Who*

In the past few decades many games and pastimes have been subject to some form of implementation on the computer. In games such as Chess and Scrabble it has been for the purpose of challenging a human player with a skilled opponent, and this has in many cases been so successful that the computer has shown to be much superior to the human player, mainly due to its superior information processing capabilities. There has been and still is an ongoing quest for the creation of systems capable of imitating the intelligence and performance of humans in better ways and in any task domain.

The crossword puzzles domain is different from the games domain, humans and machines are challenging each other on two separate grounds neither of which is interactive (like Chess and Scrabble) and qualitative measures are hard to define and give unambiguously. The two fronts are

(1) **Construction:** computer crossword compilers challenge human crossword compilers, and

(2) **Solving:** computer constructed crossword puzzles challenge human crossword solvers.

In some narrow fields of crossword puzzle construction (like the search for rare solution sets, see chapter 6) computers manage quite well. In fact they are superior. When it comes to *real* crossword puzzles, however, the qualitative measures are not given on grounds based on how well solvers *manage* to solve them or on how *fast* the puzzles are solved. Instead judgment measures concentrate on how well the solvers are *entertained* partly based on the selection of *words used* and the *clues given*. In the crossword puzzle domain computers are **very** good at producing quantity, **much less** good at producing quality. If, however, crossword puzzle quality could be strictly defined (mathematically/grammatically/ etc.) down to the finest detail and without missing anything out, perhaps computers would manage the fine art of quality crossword compilation but this is probably not feasible. Words and grammars derived from natural languages are subject to an ever ongoing un-predictable evolution and implicitly so is crossword puzzle compilation. Artificial intelligence can to some extent achieve human-like results but fails too often within the crossword puzzle fields on obvious decisions which, on the other hand are not at all confusing to humans.

3.2 The human approach

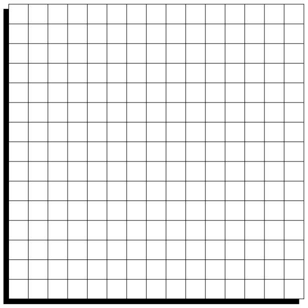
> **The human mind treats a new idea the same way the body treats a strange protein; it rejects it.**
>
> *P. B. Medawar*

We all know crossword puzzles from the solvers' point of view, that is we meet them in newspapers and magazines and sometimes perhaps they make us wonder how they were made. And well, people actually earn money making them and the best have been made the old fashioned way: with pencil, paper, eraser, dictionary, and plenty of spare time. So when you want to computerize the process it would be natural first to look at the methods used by humans.

If you have ever tried to cross words on a piece of paper you have probably shortly thereafter realized how hard it is to make even small crossword constructions. Compiling a puzzle is not as easy as one would expect it to be. Constructing crossword puzzles by hand require talent and practice. Style and method vary **a lot** from constructor to constructor and it would be very hard if not impossible to give a detailed description on how to produce a crossword puzzle by hand.

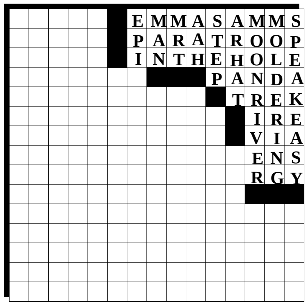
Some rough guidelines are available though. You should first settle on a grid geometry and then start inserting words. This, however, does not prevent you from altering the grid at a later stage. When the grid configuration is settled on you begin to insert words and the words used often belong to special categories hereby giving the puzzle a theme or a feel of completeness. Words are also carefully picked in order to adjust difficulty levels, in order to exhibit actuality, in order to satisfy a strict crossword puzzle editor, and in order of a lot of other things ... Also, it is not unusual in the process that when choosing a particular word, the constructor already has a special clue in mind. In other words, the human compiler merges the compilation steps involving construction and clue generation. In order to get an idea of the human mind constructing a puzzle we will quote Frank Longo [1] a long term cruciverbalist and one of the few making a living on the construction market (his puzzles can often be seen in *The New York Times*).

[1] In the CRUCIVERB-L construction forum on the Internet on Wed, 16 Oct 1996.

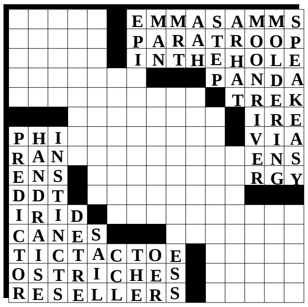


Well, I started this particular puzzle much as I would any other wide-open, theme-less. Since there are no theme entries to deal with, I always first try to get a nice, "eye-popping" corner which I can build from. For this puzzle, I decided, for a change, to have the wide-open corner in the upper *right* instead of the upper left. For absolutely no other reason than "it just popped into my mind", the first entry I placed in the grid was EMMA SAMMS (of course, there were no black squares placed at this point). Once this was in place, I thought I'd like to have not only three stacked 9's horizontally, but also *vertically* in the same corner.

So, after much fussing, I came up with a satisfactory corner. I liked it because three of the six 9-letter entries were phrases (EMMA SAMMS, IN THE HOLE and MOON RIVER), and of the other three, two were interesting (PARATROOP, SPEAKEASY) and only one was bland (MOLDERING). The resulting three, four and five-letter entries were all nice as well: **(a)**
I knew I could build from this, too, because many words and phrases start with EPI- and MAN-, and end with -ENS and -RGY. But before investigating those possibilities, my better judgment and experience told me that I'd better get a good opposing corner *first*, so I then tackled the lower left corner, which now had to have the symmetrically- opposite shape as the above corner.



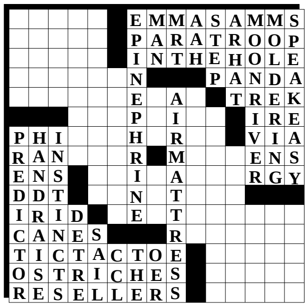
(a)



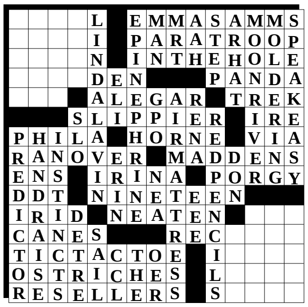
(b)

For a good 9-letter phrase, I chose TIC-TAC-TOE and placed it first. Actually, HAND-RAISE *was* in my vocabulary. Since I own many small pets (guinea pigs, mice, rats, hamsters) and have often had to hand-raise them, I have seen and used the phrase quite often; thus I actually *chose* the phrase, thinking it was a good non-dictionary entry. I knew that this corner also had to be expandable. In the end, after lots of trying, I settled for a fill which, admittedly, isn't wonderful. It's probably the weakest area of the grid, but I was stubborn in that I absolutely would not budge the phrase TIC-TAC-TOE from its position. In retrospect, the corner probably could have been improved if I'd have been willing to throw it out and try something else. I ended up with two blah 9-letter words, PREDICTOR and RESELLERS, and the worst entry of the corner (probably of the whole puzzle) was IRID, which crossed another weak partial, D'ETRE: **(b)**.

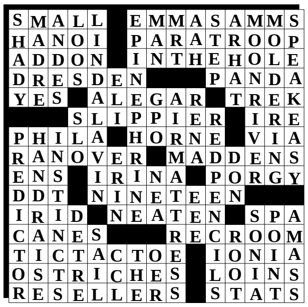
Now, with these corners fixed, I could go about the task of expanding them. I always go for phrases, if possible, but there just aren't many good ones starting with EPI-. So I chose the long and interesting word EPINEPHRINE. For the symmetrically-opposed word, having thousands of possibilities to choose from with the pattern ———ESS, I went with AIR MATTRESS. So now I had: **(c)**.
I noticed that, if I placed a black square in the dead center of the grid, I'd have the two seven-letter templates RAN—R and M—ENS. I liked this, so I did so and went with RAN OVER and MADDENS. Then, while fiddling the center of the puzzle, I always kept in mind that I wanted at least two more long vertical phrases which would extend into the upper-left and lower-right corners.



(c)



(d)



(e)

So, I suppose I got a bit lucky in that I was able to get a good fill for the middle which included two such phrases, LINDA LAVIN and RED-PENCILS. I was only unhappy about one entry, ALEGAR, but I thought it was worth it considering the rest of the fill: **(d)**.
Now all that was left were two easy corners! I was able to cram another nice phrase, REC ROOM, into the lower right, and fill the corners with all very common entries, since they were only 5x5's: **(e)**.
Note that I was successful in my goal not to use any "cheater" squares in this puzzle. I usually end up with a few, but I really resisted it with this construction. Now that you're all yawning from boredom, I'll shut up. Thanks for listening! –Frank

And with this insight into how a hand made crossword puzzle is created we will turn to computers ...

3.3 The computational approach

> **Artificial Intelligence is no match for natural stupidity.**
> *Unknown*

Computers are superior when it comes to information processing and whenever computers outdo humans it is when the jobs are more or less a question of processing (and only processing) large data sets. When computers loose it is because they do more work when less work could do the same. Now, humans are clever (and lazy) and thereof good at realizing when and how work can be reduced without lowering the quality of the result (on the contrary). In some situations computers can be taught to do the same and apparently imitate human intelligence, but not always and far from optimal in all cases. The approach Frank Longo used above can never be fully imitated by a computer - maybe not even close - instead we must relay on the powers that computers are in possession of and develop construction methods more on their terms (the computers).

Before we look at the CPP some definitions are required.

Definition 3.1 (Solution types) When all open cells in a puzzle geometry is filled with symbols from the supporting alphabet we call the solution an *enumerated solution.* If all word slots contain templates (possibly empty) verifiable by the dictionary we call the solution a *partial solution.* And finally the partial solution is a *legal solution* when all word slots are filled with verifiable words.

The CPP [2] can now be rephrased

Definition 3.2 (The crossword puzzle problem, CPP) Given a puzzle geometry and a word list find all legal solutions.

3.3.1 Solving a combinatoric problem

The CPP is a combinatoric problem; given a set of symbols we are asked to place them in a grid in such a way that the resulting strings read horizontally and vertically belong to a specified word list. Finding the legal combinations among all the enumerated combinations can be done with more or less intelligence.

A brute force method

(1) produce all enumerated solutions, and

(2) examine each enumerated solution and check if it is a legal solution

Even for small puzzles this approach is out of range with the capacity of today's computers. A square grid with, say, 16 open cells has got $26^{16} = 4.4 * 10^{22}$ enumerated solutions that have to be checked. Even if we had a computer capable of checking 100 billion enumerated solutions each second the search will still last 13.828.242 years!

A search among partial solutions

If we instead of checking all enumerated solutions limited the search to partial solutions things would turn to the better. Using the 4×4 square above and a word list with, say 1500 words of length 4 one type of filling in the grid is to insert legal words in the horizontal direction and hereby create partial solutions. The total number of such partial solutions is $1500 * 1499 * 1498 * 1497 = 5.0 * 10^{12}$ (1500 possible words in the first row, 1499 possible words in the second, etc). Among these partial solutions some of them are legal solutions (solutions in which words read vertically turn out to be legal too) and with the same computer power as above the search will last only 1.4 hours.

A search using look ahead

When two words have been inserted only a small subset of these will generate actual legal prefixes in the four existing vertical word slots. Above, a third word will be tested with **all** two-word combinations; it will be tested against two-word combinations which we already have rejected, i.e. a waste of work. So among the $1500*1499 = 2.2*10^6$ two-word combinations only a small subset should be tested against the third word. This argument can be used again on the insertion of the fourth word but this time the argument is even stronger. Above, the fourth and last inserted word was tested against $1500*1499*1498 = 3.4 * 10^9$ three-word combinations but only a very small fraction of these three-word combinations were valid.

Now, if we have an $n \times n$ grid without any closed cells and a word list containing N words

[2] First defined in chapter 2.

of length n the number of two-word combination tests to perform is $n_2 = N * (N-1)$. Now, what is the number of *legal* three-word combinations? The number of *legal* four-word combinations? The number of *legal* n-word combinations? We can only guess. An estimate, $E(i)$, must be recursive and contain a reduction factor function, $R(i)$, where i is the number of words in the grid at the time of insertion. Let

$$E(1) = N \quad \text{and} \quad E(i) = E(i-1) * (N-i+1) * R(i)$$

We have not defined the reduction factor function, $R(i)$, but if defined it would depend on

N the number of letters in the word list,
n the length of the words in the word list, and
i the number of words in the grid.

Optimal search

Further refinements: instead of testing **all** partial solutions we could use the information at hand better and possibly reject foul sets of partial solutions *without* actually testing them. This is possible because the search space can be traversed in a search tree structure - and search trees can be pruned. The methods for pruning are among others *multiple look ahead* and *intelligent backtracking.*

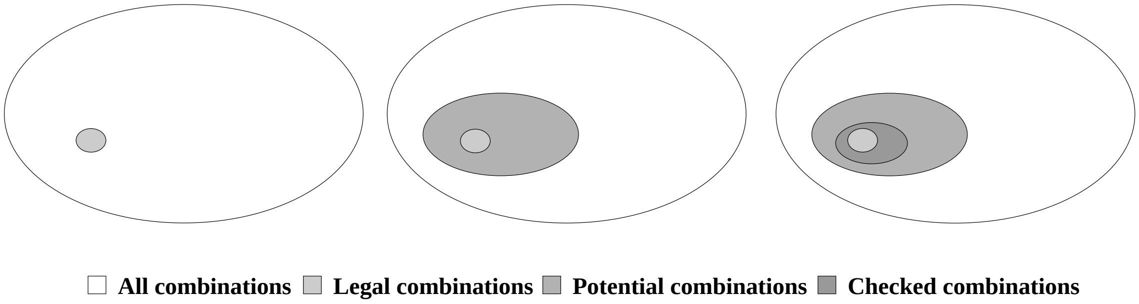


Figure 3.1: *Going from hard to smart.*

If we define

$$C = |checked\ solutions|$$
$$L = |legal\ solutions|$$
$$O = C - L$$

an optimal algorithm would result in $O = 0$. This, however, is almost certainly impossible and what we should try instead is to minimize O. In other words minimize the search overhead. The question is just how low can we get? How effectively can we *prune* the tree? How well can we avoid *repeating* searches ? And by the way what is the size of the legal set L after all ?

3.4 Construction - Previous work

God is not dead but alive and well and working on a much less ambitious project.

Anonymous Graffito

The work done on crossword compiler construction is without doubt the most interesting research area when discussing computational aspects of crossword compilation. The research can be divided into procedural and declarative approaches both of which can be subdivided into node evaluation and labyrinthic methods. The best results have been achieved with procedural knowledge and labyrinthic insertion [3].

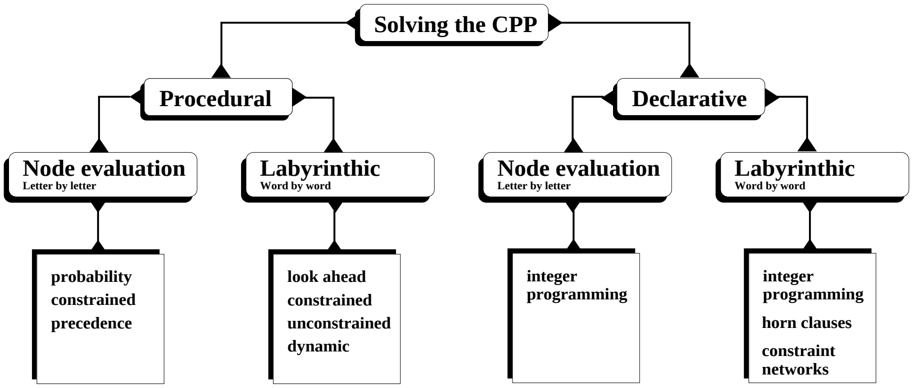


Figure 3.2: *Methods used to solve the CPP and how they relate.*

3.4.1 Preliminary definitions

Before we get on to the main problem in question here we must look at some definitions and look at some initiating steps. The Crossword Puzzle Problem (**CPP**) can be solved in two ways: **(1)** with procedural knowledge, or **(2)** with declarative knowledge.

Definition 3.3 (Procedural knowledge) Problems with *procedural representations* encode how to do some task; achieve a particular result.

Definition 3.4 (Declarative knowledge) Problems with *declarative representations* have knowledge in a format that may be manipulated, decomposed and analyzed by the reasoning engine independent of its content.

In short, the two types of knowledge can be described as knowing *how* and knowing *that* respectively. They also differ in that procedural knowledge often leads to faster solutions where declarative knowledge has the ability to use knowledge in ways that the programmer did not foresee.

In chapter 2 we defined what conventional CPP compilation is about but other methods exist. What we defined previously was how to solve a constrained CPP in which the geometry of the puzzle is defined before runtime. When geometry is not predefined we deal with an unconstrained CPP.

[3] See the definitions in the next subsection.

Definition 3.5 (Constrained crossword puzzle) A *constrained crossword puzzle* is one in which the grid configuration and the word list specifications are known before runtime.

Definition 3.6 (Unconstrained crossword puzzle) An *unconstrained crossword puzzle* is one in which only the puzzle dimensions and some word list specifications are known before runtime.

With declarative knowledge it is possible to solve the CPP using logic; Horn Clauses, Integer Programming and Constraint Networks have been used in previous works.

Definition 3.7 (First-order logic) The language describing the truth of mathematical formulas. Formulas describe properties of terms and have a truth value.

A subset of first-order logic is

Definition 3.8 (Horn Clause) Also known as a definite clause. A *Horn clause* is defined as a type of rule which only has one conclusion.

The logician Alfred Horn studied this form of rule and discovered that all propositional statements can be expressed in clauses that contain at most one conclusion. Thus, clauses with either no conclusion or one conclusion are now known as Horn clauses. A Horn clause with no conclusion is called a fact.

Definition 3.9 (Linear programming) In the general linear-programming problem, we are given an $m \times n$ matrix, A, an m-vector b, and an objective function $\sum_{i=1}^{n} c_i x_i$ subject to the m constraints given by $Ax \leq b$.

Definition 3.10 (Integer programming) An integer programming problem in which all variables are required to be integer is called a *pure integer programming problem.* If some variables are restricted to be integer and some are not then the problem is a *mixed integer programming problem.* The case where the integer variables are restricted to be 0 or 1 is called *pure (mixed) 0-1 programming problems* or *pure (mixed) binary integer programming problems.*

3.4.2 Procedural approaches - node evaluation

In this section a couple of methods using procedural knowledge and node evaluation will be examined. They all insert one letter at the time in the grid giving the method its name.

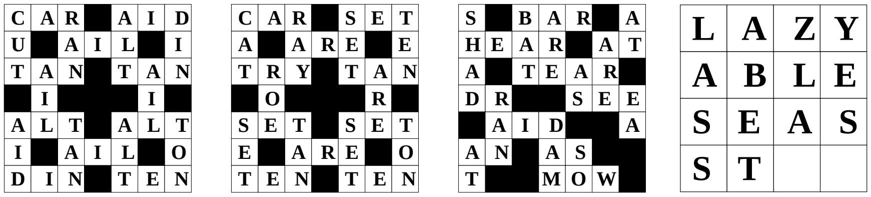
E. S. Spiegenthal

> *Redundancy Exploitation in the Computer Construction of Double-Crostics*
>
> Acrostics, a special variant of crossword puzzles are short verse composition; initial letters of the lines taken consecutively form words. Double acrostics are constructions in which not only the initial letters of the lines but in some cases also the middle or last letters form words. Back in 1960 E. S. Spiegenthal developed a puzzle solver for the solution of double acrostics and he is probably the first ever to have solved a word construction problem using a computer. □

L. J. Mazlack

Computer Construction of Crossword Puzzles Using Precedence Relationships [Maz76a]
Machine Selection of Elements in Crossword Puzzles [Maz76b]

L. J. Mazlack described two methods: **(1)** word by word and **(2)** letter by letter insertion; the labyrinthic method and the node evaluation method respectively. With (2) and using precedence relationships Mazlack had success though the puzzles were very simple due to system limitations; the word list only had 2000 words (max length 4) and it was not accommodated in main memory. Probability measures were used to minimize disc accesses. Mazlack used a precedence stack in which potential cells were kept sorted by weight. Cells were defined potential if they were *visible* (they belonged to a word slot in which at least one other cell was filled) and the assigned weight was given by *degrees of freedom + number of open cells directly connected.* When a cell was chosen the letter to be filled in was chosen using occurrence ratios.



C	A	R	■	A	I	D
U	■	A	I	L	■	I
T	A	N	■	T	A	N
■	I	■	■	■	I	■
A	L	T	■	A	L	T
I	■	A	I	L	■	O
D	I	N	■	T	E	N

C	A	R	■	S	E	T
A	■	A	R	E	■	E
T	R	Y	■	T	A	N
■	O	■	■	■	R	■
S	E	T	■	S	E	T
E	■	A	R	E	■	O
T	E	N	■	T	E	N

S	■	B	A	R	■	A
H	E	A	R	■	A	T
A	■	T	E	A	R	■
D	R	■	■	S	E	E
■	A	I	D	■	■	A
A	N	■	A	S	■	■
T	■	■	M	O	W	■

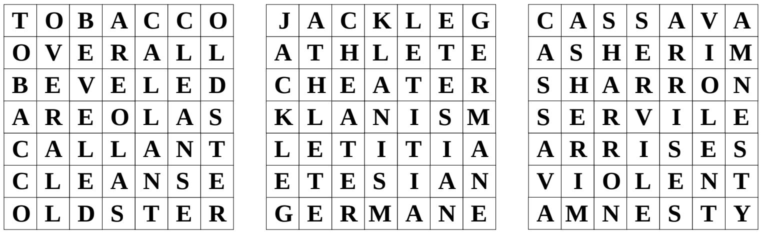
L	A	Z	Y
A	B	L	E
S	E	A	S
S	T		

Figure 3.3: *Some puzzles Mazlack managed to compile and a puzzle he couldn't finish*

Mazlack failed to solve square puzzles with no closed cells of size 4. □

M. D. McIlroy

M. D. McIlroy, in the quarterly journal Word Ways published no less than 52 7 × 7 puzzles containing no closed cells using a dictionary with 9663 seven letter words. He did search for 4×4 puzzles too and concluded that possibly millions of these existed. Presumably McIlroy did not know of the work by Mazlack (and vice versa).



T	O	B	A	C	C	O
O	V	E	R	A	L	L
B	E	V	E	L	E	D
A	R	E	O	L	A	S
C	A	L	L	A	N	T
C	L	E	A	N	S	E
O	L	D	S	T	E	R

J	A	C	K	L	E	G
A	T	H	L	E	T	E
C	H	E	A	T	E	R
K	L	A	N	I	S	M
L	E	T	I	T	I	A
E	T	E	S	I	A	N
G	E	R	M	A	N	E

C	A	S	S	A	V	A
A	S	H	E	R	I	M
S	H	A	R	R	O	N
S	E	R	V	I	L	E
A	R	R	I	S	E	S
V	I	O	L	E	N	T
A	M	N	E	S	T	Y

Figure 3.4: *Puzzles McIlroy found*

McIlroy did not describe the algorithm he used but it must have been simpler since it was only designed to find square puzzles. More about square puzzles in chapter 5 and chapter 6. □

3.4.3 Procedural approaches - labyrinthic

In this section methods using procedural knowledge and labyrinthic insertion will be examined. They all insert one word at a time in the grid.

O. Feger

Ein Program zur Konstruktion von Kreuzworträtseln [O.F75]

O. Feger's crossword compiler algorithm: **(1)** Determine the filling order of the open cells, **(2)** calculate cell guideline values, **(3)** insert words using the guideline values and if no word exist: backtrack, and finally **(4)** add clues. When no closed cells exist the order in (1) is determined by the order in which open cells occur when alternately filling a horizontal and a vertical word slot. When closed cells exist the order is less trivial (see figure 3.5 below).

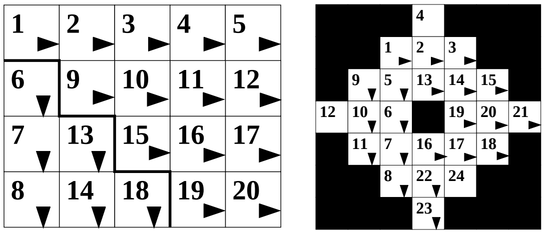


Figure 3.5: *The order in which cells in a puzzle with no closed cells are filled and the order when closed cells are introduced.*

The guideline values in (2) include among others: (i) the length of the current word, (ii) the number of open cells on which the current cell depends, (iii) the letter position in current word and (iv) the number of the cell where to backtracking should proceed. The puzzles produced by Feger were better than those of Mazlack, but Feger had a larger word list. Feger found square puzzles of size 4 and 5, but gave up on larger puzzles and, wrongly, claimed that square puzzles larger than 5 were very unlikely to exist. □

P. D. Smith and S. Y. Steen

A prototype crossword compiler [SS81]

Using a labyrinthic method P. D. Smith and S. Y. Steen produced the first crossword compiler capable of compiling publishable crossword puzzles. They were the first people to use the terms *crossword compiler*, *word slot* and the definitions on *geometry*, *density* and *degree of interlocking*. Smith and Steen's algorithm: **(1)** Associate to each word slot the exact/estimated number of words in the dictionary of the right length, **(2)** fill the word slot having the lowest estimate if possible otherwise backtrack, **(3)** update estimates of word slots intersecting the newly filled word slot. The first estimates in (1) are the exact numbers whereas those in step (3) are not. Instead a formula is used to calculate the estimates. The formula proposed by Smith and Steen was: $((C_n)^{1/n})^{n-k}$, where C_n is the number of n-letter words in the word list and k is the number of filled cells in the word slot. In step (2) a search may fail because the estimates are not exact. □

M. L. Ginsberg et al.

Search Lessons Learned from Crossword Puzzles [ea90]

Ginsberg et al., working in the field of artificial intelligence (AI) in 1990 took up the crossword puzzle search challenge and described in their paper some approaches to the problem. Having recognized the CPP as a non-trivial search problem Ginsberg et al. solved the CPP using selected AI search heuristics. Their work is to be considered a quest for a good search technique more than a technique to solve the CPP. AI techniques used:

Cheapest-first always expand the most difficult conjunct to solve.

Cheapest-first in use always choose the word slot with the fewest possible words available. This will minimize the number of backtracks and it will force them to happen as high up in the search tree as possible. An estimate was used to calculate the number of available words $\omega(n)/26^k$, where k is the number of already filled cells. Cheapest first was also used by Smith and Steen [SS81].

Connectivity solve a conjunct that involves a variable that was bound by the previous conjunct.

Connectivity in use connectivity tries to make an intelligent backtrack move. A simple backtrack would choose the previous conjunct, not taking into account that the current problem could happen again (the backtrack conjunct may not be directly connected to the current conjunct). Using connectivity we force the program to backtrack to a conjunct that is directly connected and thereby ensures that the current problem is dealt with. Connectivity can be viewed as a cheap version of dependency-directed backtracking (dependency-directed backtracking does not require maintenance of dependency information during the search).

Adjacency obtain an optimal ordering of the conjuncts.

Adjacency in use Adjacency is optimal in situations where the two other heuristics are clearly suboptimal. There is just one drawback: the orderings are too large. Even a puzzle as small as 5×5 have more than 50000 orderings.

Finally, Ginsberg et al. tailored some search heuristics aimed directly at the CPP:

Look ahead When inserting a new word those word slots which may be influenced are checked for possible word candidates. If any of these tests fail the new word is rejected. Look ahead to a greater depth is also possible.

Intelligent instantiation Among the potential words to be used in a word slot some are better suited than others. As an example we will assume that one word has got a Q in a shared position. Such a word is clearly not as good as a word with an E in the very same position. □

G. Harris

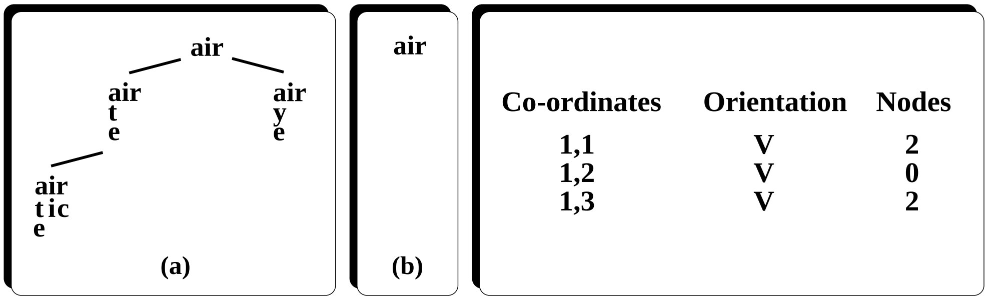
Generation of Solution Sets for Unconstrained Crossword Puzzles [G.H90a]

Harris was the first person to propose an algorithm for unconstrained crossword puzzles. The positions of the word slots were now to be determined during run time. A brute force algorithm: **(1)** Generate all possible grid configurations, and **(2)** use an ordinary crossword compiler to find all solutions to each puzzle configuration generated in (1). The number of grids in (1) for an $m \times n$ grid is 2^{mn} indicating the weakness of the method. A large number of the possible puzzle configurations do not have solutions either, because of trivial puzzle configurations or because of a limited word list, and many configurations are a contained part of other configurations. Harris proposed a scheme that avoided the generation of unnecessary grid configurations. He generated the grid configurations as the words were inserted and hereby at all times made sure that the final grid configurations had at least one solution. Harris developed a dynamic slot table mechanism which at all times contained the word slots defined in the grid. Harris algorithm: **(1)** Insert an initiating word in the grid, **(2)** compute all new word slots, and **(3)** find a word that intersects the previously inserted word if possible otherwise backtrack. □

G. H. Harris, D. Roach, P. D. Smith, and H. Berghel

Dynamic Crossword Slot Table Implementation [G.H90c]

Smith and Steen [SS81] introduced and implemented a static version of the slot table design. Years later Harris et al. proposed a dynamic approach with good results. The algorithm chooses as the next slot to be filled the one that minimizes the number of branches at the current conjunct. Consider a 3 by 3 puzzle and the word list: `air ate aye bet eye nee net ran reb tie`.



Co-ordinates	Orientation	Nodes
1,1	V	2
1,2	V	0
1,3	V	2

Figure 3.6: *Part of the search tree without dynamic crossword tables (a) and with (b). To the right the slot table .*

In (a) above is shown part of the search tree grown when using the static word slot design and in (b) the same tree when using the dynamic scheme proposed by Harris et al. The insertion of `air` can be rejected right away since there are no words in the word list beginning with an 'i'. By calculating the number of potential words for each word slot influenced by the insertion of the word `air` it can be seen that word slot 1,2 has zero continuations. Harris et al. tested their program using the two proposed benchmarks developed by Berghel and Rankin [HB90a] and Spring et al. [L.J90]. For further description see chapter 4. □

I. Berker and A. C. Cem Say

A Crossword Generator For Turkish

Presumably unaware of the existing work I. Berker and A. C. Cem Say proposed various methods for solving the unconstrained version of the CPP. Four different approaches were discussed:

Random placing: place a random word at a random position in the grid. This method requires a very sophisticated backtracking scheme.

Divide and conquer: fill the middle row and the middle column and hereby divide the grid into four smaller 'similar' problems. These problems, however, are not independent.

Circular fill: first, fill in the words along the border of the grid, then in a circular manner continue towards the center.

Row-column fill: a row and a column are successively filled.

Berker and Cem Say settled on the last method mainly because this method resembles how human compilers proceed. They were then capable of producing puzzles of various sizes but did not, however, try any exhausting searches over any complete search space. This was presumably too time consuming. The solutions they found for ten by ten puzzles varied a lot in cpu time. It could take anything from a few minutes to around 40 hours (15 percent less than an hour). □

3.4.4 Declarative approaches

In this section methods using declarative knowledge will be examined.

H. Berghel (C. Yi)

Crossword Compilation with Horn Clauses [Ber87]
Crossword Compiler Compilation [HB89]

Berghel reformulated the CPP within a logical framework. To every grid configuration and word list a set of Horn Clauses can be defined and implemented in Prolog. First, Berghel defined a simple approach which resulted in a search of the whole of the search space in a left-right depth-first manner. Like in earlier approaches to the CPP Berghel saw that the problem core was hidden in the search strategy and therefore added further logical constraints. In a whole word insertion process it can be advantageous to test the intersecting word slots before insertion and only if all substrings are found positive (they are in the word list) the word is inserted. This heuristic also avoided naive backtracking, because it made the program realize its mistakes as high up in the search tree as possible. A great deal of work in subtrees was prevented this way. Together with C. Yi Berghel extended his work [HB89]. Together they proposed a Crossword Compiler-Compilation scheme in which a grid configuration was transformed into a Prolog program like the one mentioned above. □

J. M. Wilson

Crossword Compilation Using Integer Programming [Wil89]

Wilson too saw the logical structure in the CPP and tried integer programming as the key towards solutions. Wilson formulated two pure 0-1 integer models: one for the word by word insertion method and one for the letter by letter insertion method. Both approaches showed good run time results on small puzzles, but suffered one serious set-back as the grid and the word list grew: NP-completeness. Wilson did, however, point out that the number of variables in both integer problems were proportional with the number of words in the word list. In word lists containing several thousands of words, an integer programming approach to the crossword puzzle problem simply turns into a mission impossible. Wilson only implicitly explained why the integer programming approach was impossible, but later (?) studies have shown that pure 0-1 programming problems are NP-complete. □

3.4.5 Who made what?

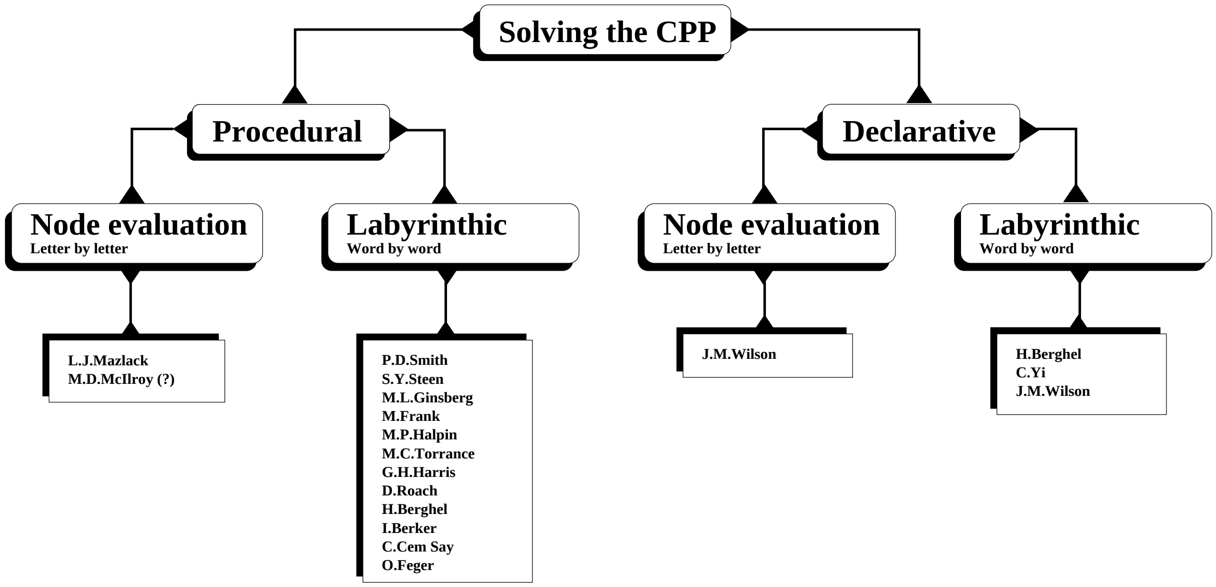


Figure 3.7: *Methods used to solve the CPP and who developed them.*

3.5 Our work - origin of our species

> **My theory of evolution is that Darwin was adopted.**
>
> *Steven Wright*

Declarative knowledge methods have been successful but severely handicapped by their strong ties to dictionary sizes as well as grid sizes. Procedural approaches on the other hand have much more convincingly exhibited good and better results.

Mazlack [Maz76a, Maz76b] was the first to make a procedural knowledge based compiler. He first attempted a word by word scheme but gave up and succeeded with a letter by letter scheme. Strangely enough, or perhaps that is the reason why, all later researches within the procedural knowledge area have been labyrinthic (dynamic word slots [G.H90c], unconstrained setting [G.H90a] among a few).

Also, up until now in the literature node evaluation and labyrinthic methods have been two separate areas of research, and implementation designs have been very distinct. This, however, is quite peculiar because the two methods are very much alike on the implementation design level.

The algorithms and designs we have developed do belong to the procedural knowledge branch in the crossword compiler family tree and they will soon be known as grid walks.

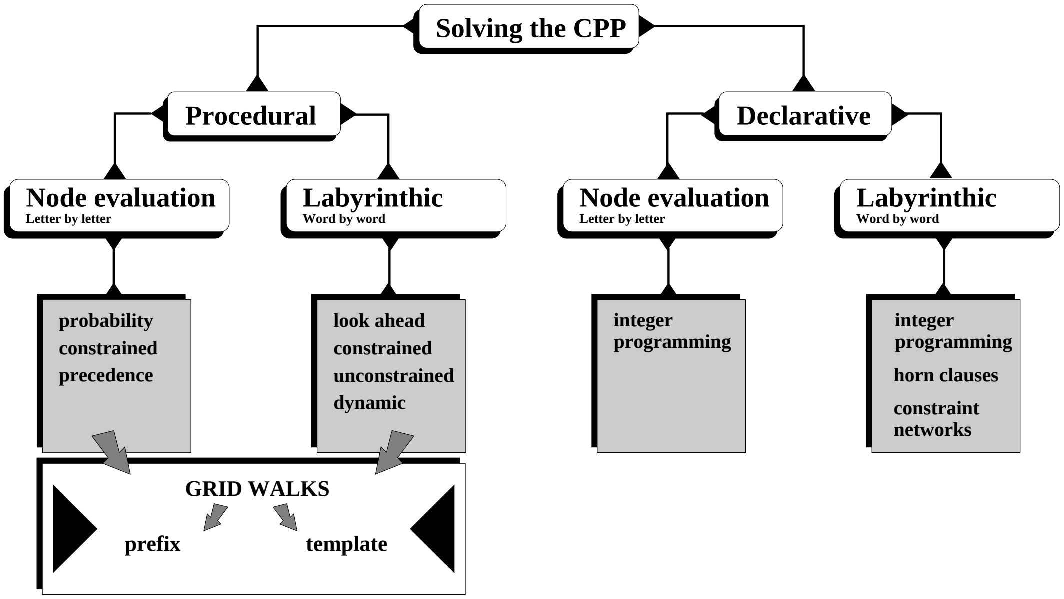


Figure 3.8: *Previous methods and ours.*

3.5.1 The making of a hybrid approach

The procedural knowledge-based crossword compilers all progress in the following manner

(1) find a cell or a word slot to fill next,

(2) find a letter or a word that fits, and

(3) verify that the chosen letter or word is legal.

Now, if we for a moment forget about letters and words but combine these into an abstract object we will call a *chunk*, then the above steps will be

(1) find a *chunk* to fill next,

(2) find a set of letters that fit, and

(3) verify that the chosen letters are legal.

Up until now we have only seen two possible insertion type objects, a letter and a word, but why not include *substrings* and *partial grid fillings*? This would surely increase design flexibility. And talking about design, will adding two new insertion schemes not lead to two new design schemes? Not necessarily.

Assume we have two crossword compilers: a **(1)** letter by letter crossword compiler and a **(2)** word by word crossword compiler. Which one is the most versatile? We think it is number (1) and the reason is that (1) can simulate (2). When (2) inserts a word (1) can be told to do the same namely by filling the cells belonging to the word slot in question and doing it cell by cell. So when (2) chooses a word slot, (1) should choose a set of contiguous cells. The key idea that pops into mind (or at least into the mind of the author) is a *fill order*. If we take a letter by letter crossword compiler and add a specific *fill order* it can simulate any insertion scheme. In other words it is not the sizes of the insertion chunks (letters, words, etc) that determine which type a crossword compiler is, it is the associated *fill order*.

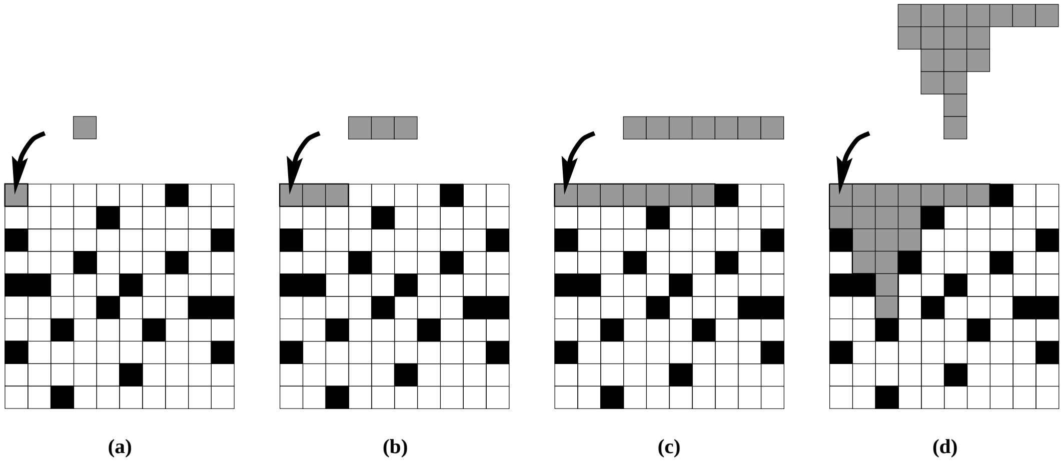


Figure 3.9: *Going from small steps to large steps:* (**a**) *inserting one letter at a time,* (**b**) *inserting one substring at a time,* (**c**) *inserting one word at a time, and finally* (**d**) *inserting a partial solution at a time.*

3.5.2 Walking the grid

So now, with our eyes focused on fill orders rather than grid chunk sizes, we will define what a fill order is.

Definition 3.11 (Fill order) A *fill order* is a specified order in which open cells are filled.

The fill order specifies the order in which the cells in a grid are filled and it can be pictured as an unbroken line that visits each cell in the grid exactly once. Such a line we will call a *walk*, and when we follow the line we say that we *walk the grid.*
The walks can and should be categorized. The main type is the *template walk*, which allows any order of the cells in the walk. Then, as a subset, comes *prefix walks* which are restricted walks, in that each cell in the fill order is to the right and below either an earlier cell in the walk or the grid border. A walk can also be specified as a

(1) *static walk*: the walk is defined **before** runtime, or

(2) *dynamic walk*: the walk is defined **during** runtime.

and as a

(3) *random walk*: the walk is defined at random, or

(4) *weighted walk*: the walk is defined using a weight function.

A walk can be a mix of all these, a *prefix walk* defined *randomly* before runtime (making it a *static walk*).
The following figures, 3.10 and 3.11, illustrate various types of walks

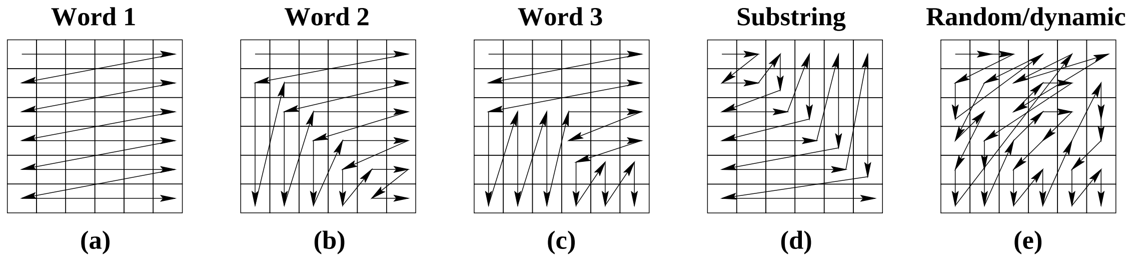


Figure 3.10: *Prefix walks.*

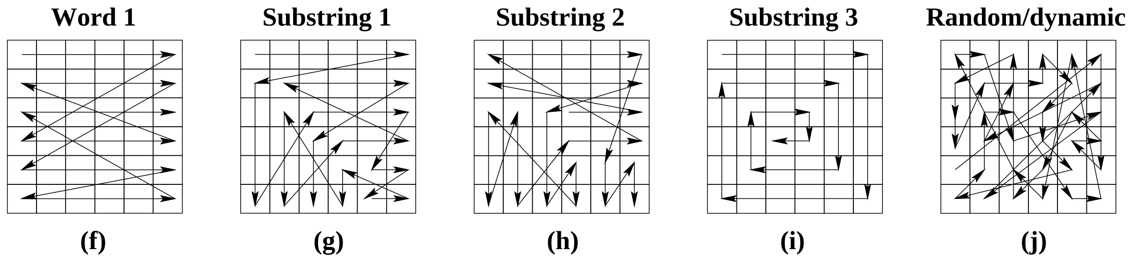


Figure 3.11: *Template walks.*

The walks defined above can be recognized as

> **word by word insertion:** (a), (b), (c), and (f),
>
> **substring insertion:** (d), (g), (h), and (i),
>
> **true letter by letter insertion:** (e) and (j).

3.6 Walk heuristics

> **Everywhere is walking distance if you have the time.**
> *Stephen Wright*

We can walk the grid in many different ways and with many different results. One type of walk will be good with one type of grid and terrible with another grid. One type of walk is easy to generate and another is very hard to generate. A walk tells us where to go next, and it is also capable of telling us where to go if we need to walk back (back track), but with the walk alone we are only able to perform naive backtracking, i.e. turn back to the previous cell in the walk. This is, however, far from optimal, and we are therefore asked to define 'optimal' back walks.

Definition 3.12 (Back Walk) A *back walk* specifies an order in which back tracking can proceed.

A back walk can be pictured as set of lines where each cell is visited by one line only and only once. These lines we will call *back tracks* and when following such a line we are *backtracking*. And when is backtracking necessary? That is when a cell is chosen and no letters are available.

3.6.1 The walk module

A walk module can, when given a grid and a cell, return the cell that is next in the walk.

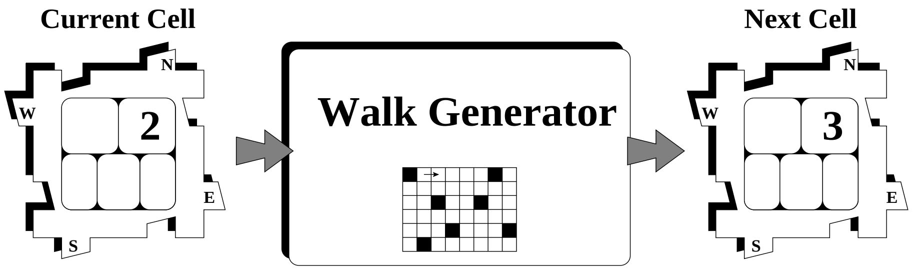


Figure 3.12: *When given a cell the walk module can, using the grid, find the next cell to be filled.*

3.6.2 The backtrack module

Again and again during the construction of a puzzle we need to go back to a previous cell because the walk ahead has ended blind. Like we have the walk modules we have for each walk module a backtrack module. These modules will when given a cell return the cell to which we will resume the construction.

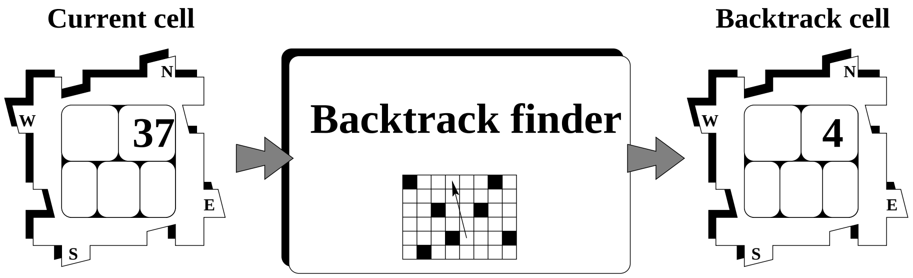
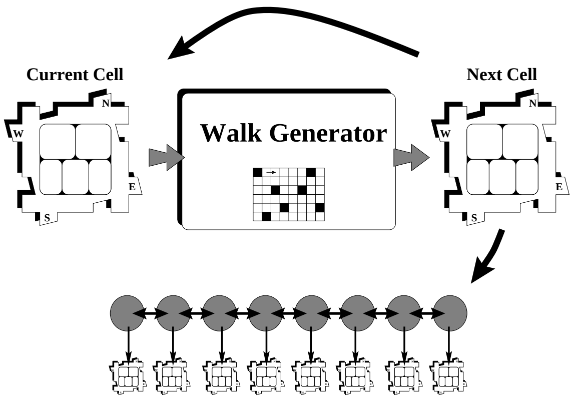


Figure 3.13: *When given a cell the backtrack module can by using the grid find a backtrack cell.*

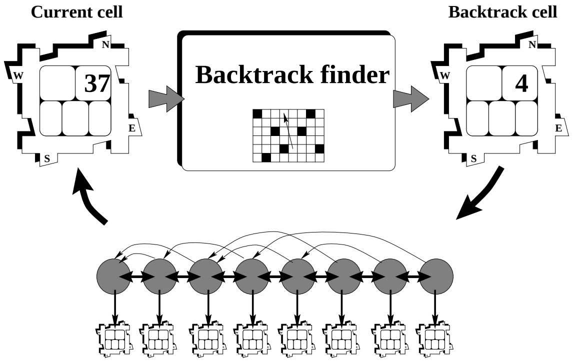
3.6.3 Generating a pre-walk and some pre-backtracks

Except for the *dynamic walks* a walk can be generated before the actual search for crossword puzzle solutions. Generating the walks before the search will reduce the number of calls to the walk generator from a lot to the number of cells in the grid. Notice too that only open cells need to be added to the walk description.



```
List ← PreWalk (cell)

while (Cells left) do
    cell ← WALKGENERATOR (cell)
    List ← ADD (cell,List)
```

When the walk has been generated the backtracks can with advantage be pre-defined too.



```
List ← PreBack (List)

while (Cells left) do
    cell ← BACKGENERATOR (cell)
    cell[BACK] ← FINDBACK (cell,List)
```

3.6.4 Inserting a letter in the cell

When a cell is due to have a letter inserted we must first **(1)** find those letters that are available and then **(2)** choose a letter among these. The available letters in (2) are what we will call *legal* letters.

Definition 3.13 (Legal letter) If a letter, l, can be inserted into a cell and still preserve the templates (prefixes) in the word slots involved l is a *legal letter*.

Definition 3.14 (Letter supply) The set of legal letters associated to an open cell is called the *letter supply*.

If the letter supply is empty backtracking must be performed, otherwise a letter is

random: chosen at random,

weight: chosen by using a weight function

probability: chosen by using statistical information extracted from the dictionary.

3.6.5 Proposed walk heuristics

In this section we will design a set of walk modules and to each module a possible backtrack scheme. Ginsberg et al. [ea90] discussed various search heuristics and with specially designed walks and backtracks some of these heuristics will show up as a natural part of the design.

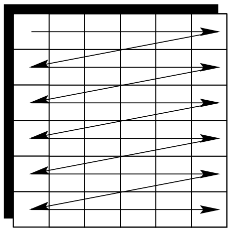
The walks in general

Cheapest first and *adjacency* can with advantage be invoked when designing the walks. *Look ahead*, and *Intelligent instantiation* are automatically used because we insert one *legal* letter in each insertion step.

The back tracks in general

When using static walks naive backtracking to the last visited cell is easy to perform. This approach, however, is not optimal since the last visited cell might be far from the current cell and therefore not directly involved in the problems right at hand. The answer to this when moving back is to use *connectivity* (Ginsberg et al. [ea90]).

3.6.6 The Prefix Straight Walk Heuristic



Straight walk

```
Cell ← Straight Walk (cell)

if cell[East] exists then
    Cell ← cell[East]
else
    if cell[South] exists then
        Cell ← MOVEWest(cell)
return Cell
```

MOVEWest(cell) returns the cell to the far left of the grid in the row below the current cell. (Source code can be seen on page 149 to the right)

3.6.7 The Prefix Straight Backtrack Heuristic

The naive backtracking scheme mentioned above will be OK in most cases as in figure 3.14 **(a)** below but in some cases like in **(b)** it can and should be optimized. In **(b)** the reason for the blind end is not the previous cell in the walk but the cell right above the current cell. It is therefore possible to improve the backtrack. Is this multiple cell backtrack always possible ? The answer is no. In **(c)** we backtrack up to the cell above but ignore the fact that the current cell depends on two cells (not just one as before). Backtracking up will delete all current settings in the other cells in the same row as the troubled cell but that would be wrong. In the search tree it is the equivalent to move up two or more nodes and not examine the unexplored subtrees at the same level. So when a cell depends on more than one cell we must always choose the one which is the most previous of the two in the walk.

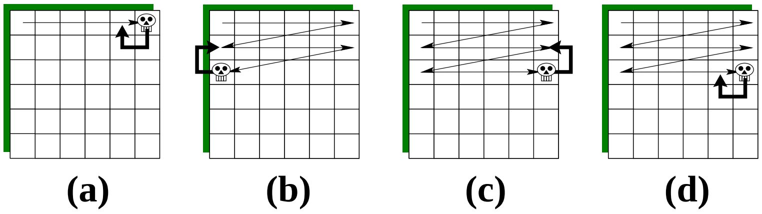
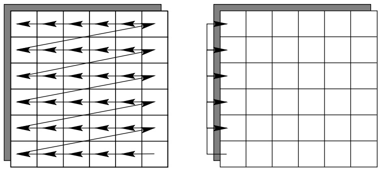


Figure 3.14: **(a)** *simple backtrack,* **(b)** *optimized backtrack,* **(c)** *illegal backtrack, and* **(d)** *correct backtrack.*

When more than one consecutive backtrack is necessary **only the first** move can be optimal - the following must be naive. If this last condition is not observed we may risk losing combinations.

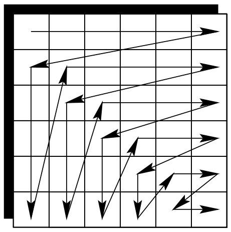


Straight back – naive and optimal

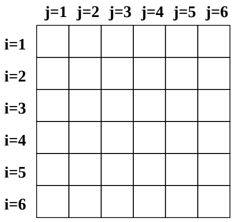
```
Cell ← Straight Back (cell)

(1)  if cell[West] exists then
(2)      Cell ← cell[West]
(3)  else
(4)      Cell ← PrevCell(cell)
(5)  return Cell
```

PrevCell(cell) returns the cell before the current cell in the walk. In line 4 above **PrevCell** should be **Cell** ← **cell[North]** if optimal backtrack is possible. (Source code can be seen on page 153 to the left)

3.6.8 The Prefix Switch Walk Heuristic



Switch walk



Grid co-ordinates

```
Cell ← Switch Walk (cell)

i = (cell[number]−1) mod Width +1
j = (cell[number]−1) div Width +1
if (j≤ i) then
    if cell[East] exists then
        Cell ← cell[East]
    else
        Cell ← MOVEmptyWest(cell)
else
    if cell[South] exists then
        Cell ← cell[South]
    else
        Cell ← MOVEmptyNorth(cell)
return Cell
```

MOVEmptyWest(cell) returns the empty cell farthest to the left in the row below the current cell. **MOVEmptyNorth(cell)** is the vertical equivalent. (Source code can be seen on page 150 to the left)

3.6.9 The Prefix Switch Backtrack Heuristic

In the Prefix Switch Walk Heuristic we must, as long as we are not on the diagonal going from top-left to right-bottom, perform naive backtracking. Are we on the other hand on the diagonal, an optimal backtrack could lead us to the cell above or to the cell to the left. Again, in order not to prune the search tree too hard, we must choose the one of the two which is the most previous in the walk.

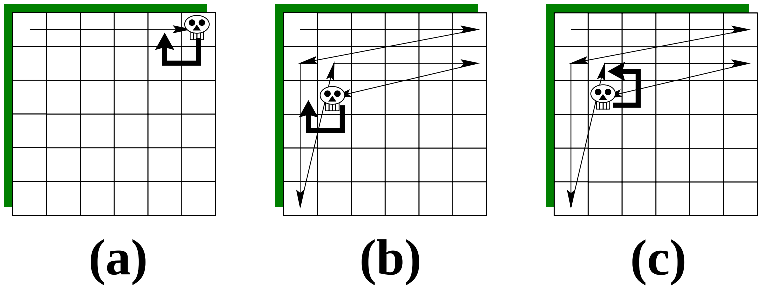
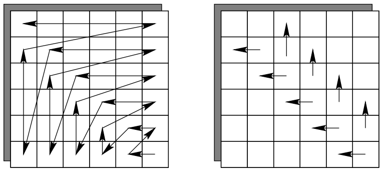


Figure 3.15: **(a)** *naive backtrack,* **(b)** *illegal backtrack, and* **(c)** *correct backtrack.*

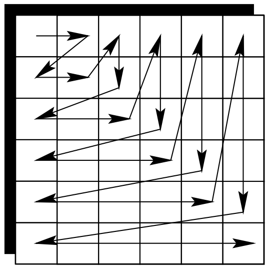


Switch back – naive and optimal

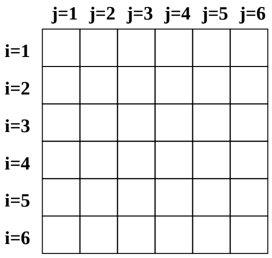
```
Cell ← Switch Back (cell)

i = (cell[number]−1) mod Width +1
j = (cell[number]−1) div Width +1
if i = j then
  Cell ← cell[West]
else if i = (j−1) then
  Cell ← cell[North]
else
    Cell ← PrevCell(cell)
return Cell
```

PrevCell(cell) returns the cell before the current cell in the walk. (Source code can be seen on page 153 to the right)

3.6.10 The Prefix Snake Walk Heuristic



Snake walk



Grid co-ordinates

```
Cell ← Snake Walk (cell)

i = (cell[number]−1) mod width +1
j = (cell[number]−1) div Width +1
if (j≤ i) then
  if (j = i) then
    if cell[East] exists then
      Cell ← MOVENorth(cell)
    else
      Cell ← MOVEWest(cell)
  else
    Cell ← cell[East]
else
  if (i = j-1) then
    if cell[South] exists then
      Cell ← MOVEWest(cell)
    else
      Cell ← MOVENorth(cell)
  else
    Cell ← cell[South]
return Cell
```

Source code can be seen on page 149 to the right.

3.6.11 The Prefix Snake Walk Backtrack Heuristic

For the Prefix Snake Walk naive backtracking is again optional but when the cell is next to the upper border or next to left border of the grid an optimization of the simple backtrack is possible. These border cells only depend on the cell above or to the cell to the left and **not** on the previous cell in the walk. In figure 3.16 below is shown in **(a)** when a simple backtrack is performed, and in **(c)** and **(d)** when an optimization is possible. Notice that the further along the walk we get the more effective the optimized backtracks get (simply because the border cells and their optimized backtrack cells are further and further apart).

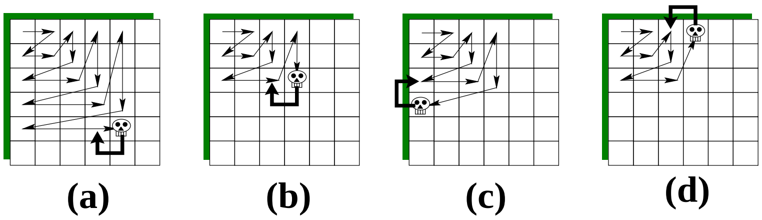
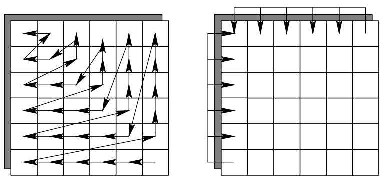


Figure 3.16: **(a)** *naive backtrack,* **(b)** *illegal backtrack, and* **(c)+(d)** *optimized backtracks.*



Snake back – naive and optimal

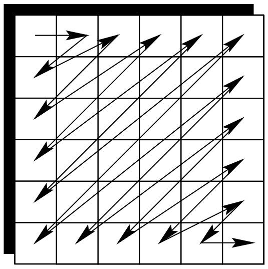
```
Cell ← Snake Back (cell)

if cell[West] exists then
    Cell ← StraightBack(cell)
else
    if cell[North] exists then
        Cell ← PrevCell(cell)
    else
        Cell ← cell[East]
return Cell
```

PrevCell(cell) returns the cell before the current cell in the walk. (Source code can be seen on page 153 to the right)

3.6.12 The Prefix Diagonal Walk Heuristics

There are two types of static walks we have not mentioned earlier. They are the Diagonal Walks. Later (in section 3.9.5) we will define in greater detail what we understand when we talk about grid diagonals, but for now we will just present the walks and the associated backtrack moves.

The Prefix Sik Sak Walk Heuristic

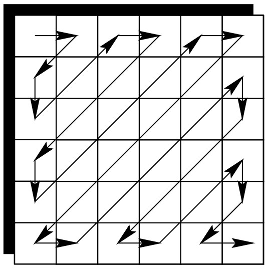


Sik Sak walk

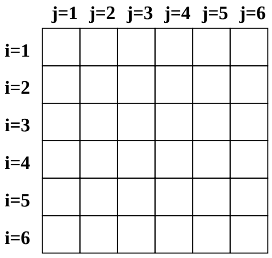
```
Cell ← Sik Sak Walk (cell)

if (cell[WEST] exists AND cell[SOUTH] exists) then
    cell ← cell[SOUTH][WEST]
else
    cell ← cell[EAST]
    cell ← BORDER (cell)
return cell
```

BORDER(cell) repeatedly look-up the cell north-east until the grid border is reached. (Source code can be seen on page 150 to the left)

The Prefix Slalom Walk Heuristic



Slalom walk



Grid co-ordinates

```
Cell ← Slalom Walk (cell)

i = (cell[NUMBER]−1) mod Width +1
j = (cell[NUMBER]−1) div Width +1
if (EVEN(i+j)) then
  if (j=1) then
    if (i<Width) then
      Cell ← cell[EAST]
    else
      Cell ← cell[SOUTH]
  else
    if (i<Width) then
      Cell ← cell[EAST][NORTH]
    else
      Cell ← cell[SOUTH]
else
  if (i=1) then
    if (j<Height) then
      Cell ← cell[SOUTH]
    else
      Cell ← cell[EAST]
  else
    if (j<Height) then
      Cell ← cell[WEST][SOUTH]
    else
      Cell ← cell[EAST]
```

(Source code can be seen on page 150 to the right)

The Prefix Sik Sak Backtrack Heuristic

Diagonal walks have a very nicely built-in backtracking property. Simple backtracks can be used but they are not necessary; from any cell it is possible to perform an optimal backtrack in which we jump back more than one cell in the walk. When we are stuck at one cell we must backtrack either to the cell above or to the cell to the left (if they exist). The cells between the current cell and the first seen of these backtrack candidates in the walk can then safely be cleared because they are all dependent on the chosen backtrack cell. In figure 3.17 **(c)** below, the cell to the left is the first cell we meet when we move back along the predefined walk, and the cells we pass on the way all depend on the back cell. Later we will define in more detail what cell dependency is (section 3.8).

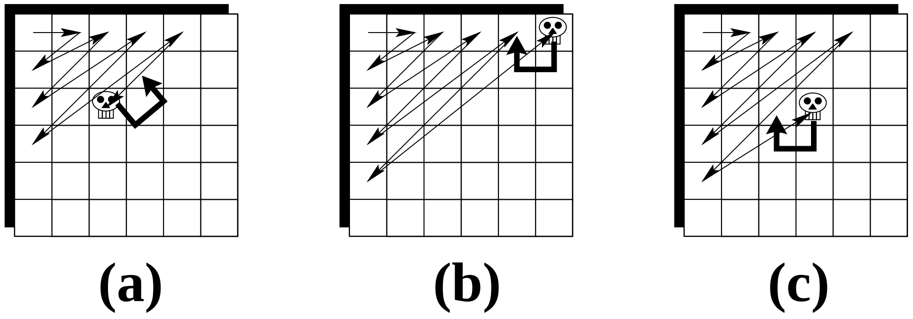
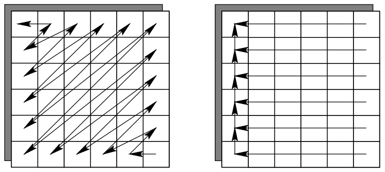


Figure 3.17: **(a)** *naive backtrack,* **(b)** *and* **(c)** *optimized backtrack.*



Sik Sak back – naive and optimal

```
Cell ← Sik Sak Back (cell)

if (cell[WEST] exists then
   cell ← cell[West]
else
   cell ← cell[North]
return cell
```

(Source code can be seen on page 153 to the right)

The Prefix Slalom Backtrack Heuristic

Like the Sik Sak backtrack, naive backtracking can be used but it is not necessary unless sometimes when the cell is next to the border. So again, if we are stuck at one cell we must backtrack to either one of the cells above or to the left whichever is the most previous in the walk. The intermediate cells in the walk between the current cell and the backtrack cell can safely be cleared because they all depend on the backtrack cell.

In figure 3.18 **(a)** below we see a naive backtrack and in **(b)** and **(c)** two optimal backtracks.

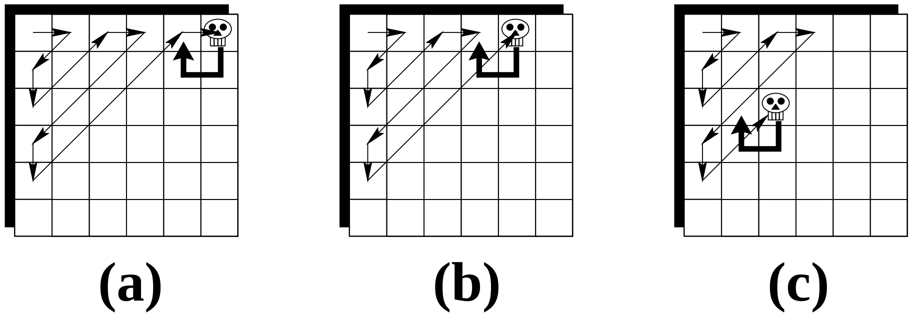
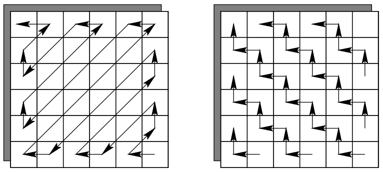


Figure 3.18: **(a)** *naive backtrack,* **(b)** *and* **(c)** *optimal backtrack.*

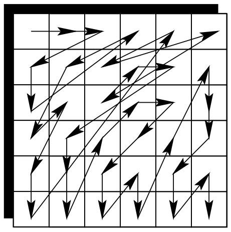


Slalom back – naive and optimal

```
Cell ← Slalom Back (cell)

i = (cell[number]−1) mod Width +1
j = (cell[number]−1) div Width +1
if (Even(i)) then
  if (Even(j)) then
    cell ← cell[West]
  else
    cell ← cell[North]
else
  if (Even(j)) then
    cell ← cell[North]
  else
    cell ← cell[West]
return cell
```

(Source code can be seen on page 154 to the left)

3.6.13 The Prefix Dynamic Walk Heuristic



Dynamic walk

```
Cell ← Dynamic Walk (List)

List ← INITIALIZE (grid)
Cell ← CHOOSE(List)
if (Cell[East] exists) then
  List ← ADD (Cell[East],List)
if (Cell[South] exists) then
  List ← ADD (Cell[South],List)
```

INITIALIZE (grid) produces a list of cells in which all members are the initial cell in the word slot they belong to. **CHOOSE (List)** returns an open cell from the **List** (chosen randomly or by using weights). **ADD (Cell,List)** adds **Cell** to **List**. (Source code can be seen on page 151 to the left)

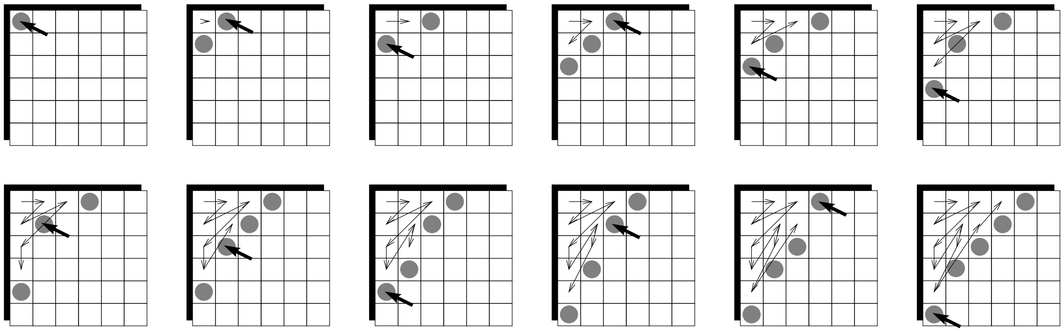


Figure 3.19: *An example of a dynamic walk in progress. The grey circles indicate potential cells. The circle indicated with an arrow is the one we choose. Each time a cell is filled the set of potential cells is updated.*

The Prefix Dynamic Backtrack Heuristic

The backtracking schemes in the previous walks were quite simple - at least not too complicated. In the dynamic walk heuristic there are no simple backtracks (some backtrack moves may look like a simple backtrack move but that is coincidental). Without the deterministic walk description backtracking becomes exceedingly hard.

We have tested a simple backtrack method: When stuck at one cell choose either of the cells above or to the left (using weights or random choices). Add the chosen backtrack cell, c_b, to the **List** and remove those cells already in the **List** and which are dependent on c_b.

This scheme seems to be rather simple and without complications but we will soon see that it is wrong and that there are some very nasty problems hidden in correcting it. More on this in section 3.9 ...

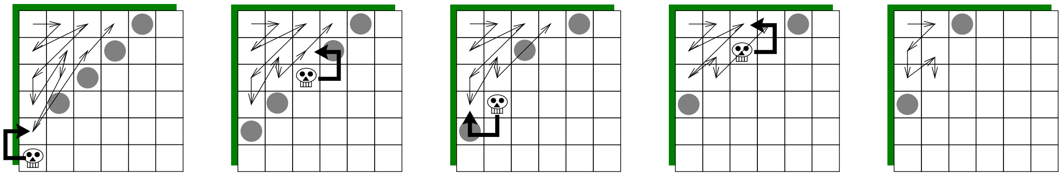


Figure 3.20: *A couple of contiguous backtrack moves.*

3.7 Redesigning previous works

> **Everything should be made as simple as possible, but not simpler.**
>
> *Albert Einstein*

Before we continue we will in this section see how previous work can be redesigned with the use of grid walks.

L. J. Mazlack [Maz76a, Maz76b]

Mazlack used a letter by letter approach, dynamic cell ordering (precedence stack) and probability measures to choose letters.

Redesign

Use a variant of the dynamic walk heuristic. Instead of prefix cells we maintain a list of *visible* cells sorted by weight. Letters in the cell are chosen using probability (or better statistical) measures. □

O. Feger [O.F75]

Feger defines a word by word filling order before run time.

Redesign

Define the equivalent grid walk and the equivalent back walks. □

P. D. Smith and S. Y. Steen [SS81]

Smith and Steen defined a dynamic word by word order determined by using estimation.

Redesign

Maintain a sorted list of remaining word slots sorted by the number of potential letters available for the first cell in these. Choose the word slot with the smallest set of potential letters and fill in the word slot letter by letter. □

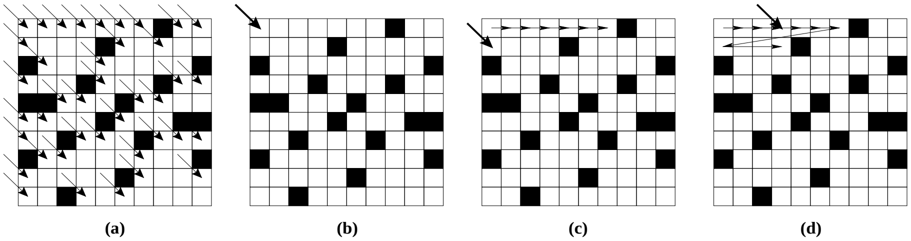


Figure 3.21: **(a)** *All word slots.* **(b)** *A word slot is chosen,* **(c)** *the word slot is completed letter by letter, the next word slot found, and* **(d)** *the next chosen word slot is filled, the next word slot chosen and so forth.*

Ginsberg et al. [ea90]

Ginsberg et al. proposed a set of heuristics

Possible designs using these

Cheapest-first
Like Smith and Steen above.

Connectivity
Static and dynamic solutions are possible and, depending on how we define the conjuncts, different connectivity filling orders can be defined. If the conjuncts are defined as word insertions **(a)**, **(b)**, **(c)**, and **(f)** in figure 41 and figure 3.11 on page 41 qualify, and if the conjuncts are defined as sub-string insertions or as letter insertions **(i)** qualify. The rest do not exhibit connectivity.

Adjacency
If an optimal ordering could be defined a similar walk could be made.

Look ahead
Inserting a legal letter at a time is nothing but look ahead at depth 1.

Intelligent instantiation
In a shared cell only letters belonging to the *letter supply* are selected - in other words intelligent instantiation, if we keep the list sorted by letter frequency. □

Harris et al. [G.H90c]

Harris et al. made a dynamic word slot implementation in which a word only fitted in if all shared cells involved remained legal after the insertion. That is made sure that the word did not create dead ends.

Redesign

This scheme, however, is nothing else but part of the look ahead heuristic scheme mentioned above. □

I. Berker and A. C. Cem Say

Berker and Cem Say defined four heuristics

Redesign

Random placing
See the grid walks in **(e)** and **(j)** in figure 3.10 and figure 3.11 above.

Divide and conquer
Walk the middle row and the middle column and hereby divide the grid into four 'similar' problems. □

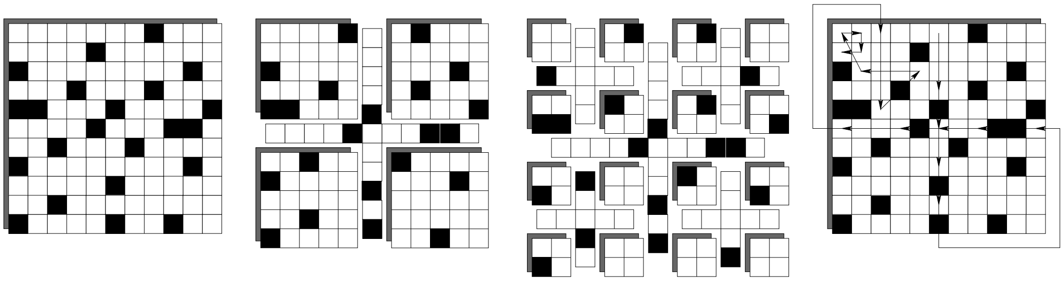


Figure 3.22: *The puzzle to the left is divided twice and to the right is shown how a filling order can simulate the same procedure (only the beginning is shown).*

Circular fill
Already shown in figure **(i)** in 3.11 on page 41.

Row-column fill
Already shown in figure **(b)** on page 41. □

3.8 Correctness of the static walk design

I never said it was possible. I only said it was true.

Charles Richet, Nobel Laureate in Physiology

When we say that the static walk design is correct we mean the following: When given an arbitrary word list and an arbitrary puzzle geometry, the static walk design will find **all** existing solutions.

Lemma 3.1 Let c_k be cell number k in the walk and let l be the number of cells in the walk that have been filled after the cell c_k was visited last. If a *trivial backtrack* traverses the cell c_k, then all combinations between the cells $\{c_{k+1}, ..., c_{k+l}\}$ have been tested.

Proof The proof goes by induction:

l = 1: If we traverse c_{k+1} it's because all letters in c_{k+1} have been tried.

l = 2: The cell c_{k+1} stops further backtrack as long as it has an unused letter in its letter supply. Each time this happens the cell c_{k+2} is visited next and all available letters in c_{k+2} are tried before we return to cell c_{k+1} again.

l > 2: Assume that for $l = n$ the assertion holds true. Now, we will show that it is true for $l = n + 1$ too.
Again the cell c_{k+1} stops further backtrack as long as it can insert an unused letter. Each time this happens all new combinations between the cells $\{c_{k+1}, ..., c_{k+l}\}$ are according to the assumption thereafter tested. ■

In the proof above assume that we perform naive backtracking, but what if optimal backtracking is introduced? Will the lemma still hold true true? The answer is yes, but, before we continue, here are two definitions about dependency:

Definition 3.15 (Fill dependency) A cell, c_d, is *fill dependent* with another cell, c_p, if

(1) they are *closely fill dependent*: c_d and c_p belong to the same word slot and the letter position of c_d comes after the letter position of c_p, or

(2) or there exist a sequence of cells, c_i, $i = 1, ..., n$, n finite, so that c_p and c_1, c_i and c_{i+1}, and c_n and c_d all are closely dependent.

Definition 3.16 (Combinatoric dependency) A cell, c_d, is *combinatoric dependent* with another cell, c_p, if

(1) they are *closely combinatoric dependent*: there exists an open cell, c_{cp}, so that (i) c_p and c_{cp}, and (ii) c_d and c_{cp} are closely fill dependent neighboor cells.

(2) there exists an open cell, c_{cp}, so that (i) c_p and c_{cp}, and (ii) c_d and c_{cp} are fill dependent cells.

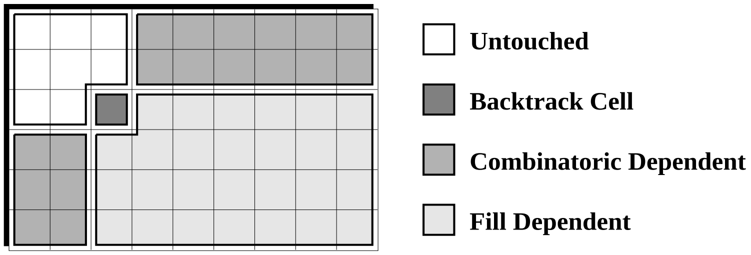


Figure 3.23: *When a cell is chosen as a backtrack cell this figure illustrates the type of dependency of the surrounding cells with the backtrack cell.*

Now, let us look at the difference the optimal backtracking cause and examine whether it introduces loss of solutions or not.
The main issue in question here is *connectivity*. Recall that when using *connectivity* the

backtrack always leads us back to a cell belonging to a word slot common with the cell in which we are stuck. In order to jump back to a cell other than the previous cell in the walk we must make sure that the cells in between the current cell and the backtrack cell safely can be cleared.

Definition 3.17 (Walk sequence) A sequence of cells, not necessarily all possible cells, taken in the order specified by the grid walk is called a *walk sequence*.

Lemma 3.2 Let the cell c_a precede the cell c_b in a given grid walk and let c_a and c_b be *closely fill dependent*. Assume that no cell in the *walk sequence*, W, starting at c_a and ending at c_b is *closely fill dependent* with c_b, then all cells in W can be safely cleared.

Proof If we are stuck at the cell c_b and if c_b is *closely fill dependent* with another cell c_a then any letter change in *non-fill dependent* cells will not change the problem in c_b. Any *non-fill dependent* cell belonging to the *walk sequence* starting at c_a and ending in c_b can not solve the blind end at c_b - they can be safely cleared. ■

We can reformulate lemma 3.1.

Lemma 3.3 Let c_k be cell number k in the walk and let l be the number of different cells in the walk that have been filled after the cell c_k was visited last. If a *trivial* or *optimal backtrack* traverses cell c_k, then all *useful* combinations between the cells $\{c_{k+1}, ..., c_{k+l}\}$ have been tested.

The term *useful* in lemma 3.3 indicate that not all combinations are tested, but those left out would not result in solutions.

Theorem 3.1 The static walk design is correct.

Proof Use lemma 3.1 and set $k = 1$ and set l equal the number of open cells in the grid. ■

3.9 The cycle problem and a possible solution

> **I put contact lenses in my dog's eyes. They had little pictures of cats on them. Then I took one out and he ran around in circles.**
> *Steven Wright*

Going from static walks to dynamic walks was a much larger step than first expected and a great amount of work has been used on the design and implementation of the dynamic walk/backtrack algorithm. The first attempt was a straight forward implementation as will be described in section 3.12.2 and 3.12.3. The method, however, did not work! Not a single solution was found and the program did not terminate. We were, no doubt, caught in a loop somewhere in the process but where and why?

The loop we search is a cycle of some kind and a cycle in this context is

Definition 3.18 (A cycle) A *cycle* is a sequence of cell fillings and backtrack moves that is repeated an infinite number of times.

In the following we will report how we first designed a simple dynamic walk and backtrack scheme, then how we found the origin of the cycles, how we designed a scheme to avoid the cycles, and, finally, we show the correctness of the new design.

3.9.1 Chasing a cycle

In order to solve the problem we needed to find a cycle and then examine the steps constituting it. This is what we did:

1. A 6 × 6 grid and a word list was given as input.
2. Each time the input resulted in a program termination failure we reduced the input (the word list) and tried again.

Using this approach we finally ended up with a small and easy word list which would result in a cycle. The word list (Danish words!) is here

```
ansete      eugene      renset      øjnene
blevet      letter      resten
dagens      nedbør      rester
emalje      neural      ugunst
```

Figure 3.24 below shows the 13 backtrack scenarios that are involved in the cycle we found. Notice that the figure only shows us the backtrack scenarios and **not** the intervening filling steps.

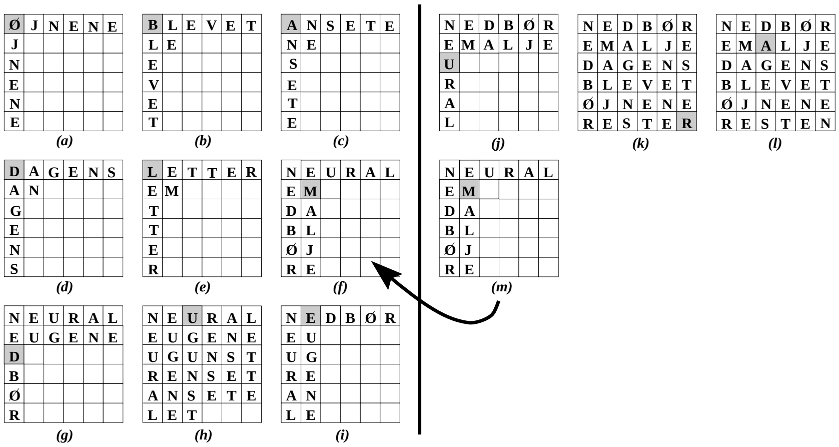


Figure 3.24: *The figure shows 13 contiguous backtrack moves. The optimal backtrack cell in each move is shaded grey. Two out of three possible solutions are found (***(k)+(l)***) then a cycle is initiated. The backtrack scenario in* **(m)** *is identical with the scenario in* **(f)** *and the steps* **(f)** - **(m)** *appear to be a cycle.*

Now, with an example at hand, we will try and explain why it appeared.

3.9.2 Explaining cycles

This is how letters are inserted and removed

1. when a letter, l_1 is inserted in a cell, c_1, it is removed from the letter supply of c_1,
2. but when a new letter, l_2 in a *closely combinatoric dependent* cell, c_2 is inserted l_1 must be re-engaged in c_1 - that is - a new letter in c_2 resets the letter supply in c_1.

This scheme is OK if it did not work the other way around too. c_1 and c_2 can change role and since we do not keep track on how many times a letter is reset we may risk cycles. This happens in figure 3.25 below.

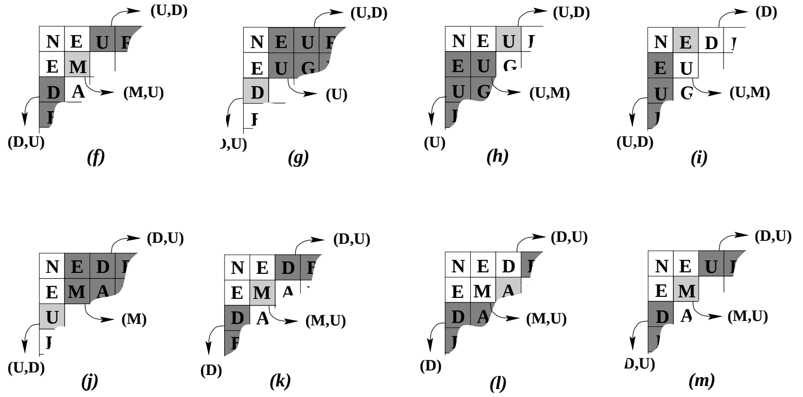


Figure 3.25: *The steps from figure 3.24 constituting the cycle are revisited. The optimal backtrack cell is shaded lightly and combinatoric dependent cells are shaded dark. Three cells are interesting and have their supply of letters printed. Each of the three cells has two letters in their supply; that is there are* $2^3 = 8$ *combinations to check, but only six of these are tested before the cycle starts in figure* **(k)**.

Figure 3.25 above shows us a closer look at the cycle we found. The three cells of interest are shown along with their letter supplies and this reveals the errors: illegal letter supply *resets*.

Now, let us simplify the problem further. The smallest crossword puzzle, apart from the trivial example, is the 2×2 puzzle. Will cycles appear here? The answer is yes. Let two cells c_1 and c_2 be *closely combinatoric dependent* and give each a letter supply of, say, two letters. Four combinations must be tested; if more tests occur there is a cycle underway.

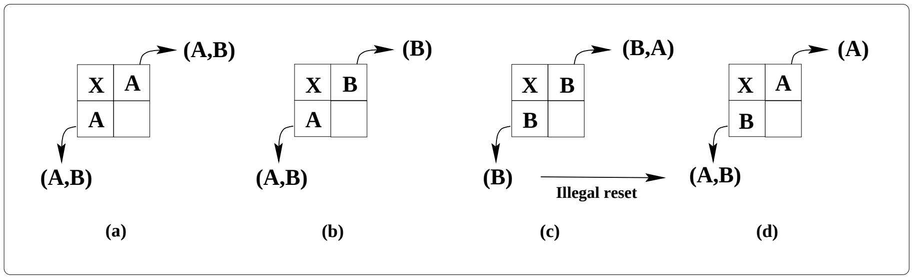


Figure 3.26: **(a)** *Start,* **(b)** B *is inserted and* A *removed from the letter supply of the cell to the north-east,* **(c)** B *is inserted and* A *removed from the letter supply of the cell to the south-west AND the letter supply in the cell to the north-east is reset, and* **(d)** A *is inserted and* B *removed from the letter supply of the cell to the north-east AND the supply in the cell south-west is reset. This last reset initiates a cycle.*

Illegal resets initiate the cycles, in order to prevent cycles illegal resets must be prevented. Can we prevent the illegal resets?

3.9.3 Solving the 2 × 2 cycle problem

We will now look at a scheme that can prevent the cycle problem in the simple example above. The cycle appeared because a letter combination between two *closely combinatoric dependent* cells was illegally reinstated. In order to make the program realize this we introduce letter counters. Each letter in a cells letter supply can at most be used a number of times equaling the size of the letter supply in the *closely combinatoric dependent* cells. This is what we do

Each letter in the letter supply is given two letter counters

- **one to be used with the closely combinatoric dependent north-east cell, and**
- **one to be used with the closely combinatoric dependent south-west cell.**
- □ **if the cell north-east/south-west is closed or non-existent the letter counter is set to zero**

A letter counter is incremented when

- **the letter is inserted in the cell, and when**
- **a letter is inserted in a closely combinatoric cell.**

A letter counter is reset when

- **the cell is cleared, or when**
- **a closely combinatoric cell is cleared.**

Choosing a letter:

- □ **select a letter with a counter value smaller than the current maximum counter value, if possible,**
- □ **choose randomly if all counter values are equal, and**
- □ **choose none if all counters have reached maximum.**

Backtracking must be done when

- **all letter counters in all prefix extending cells have reached maximum values.**

Using the above scheme results in the following sequence of events depicted in figure 3.27.

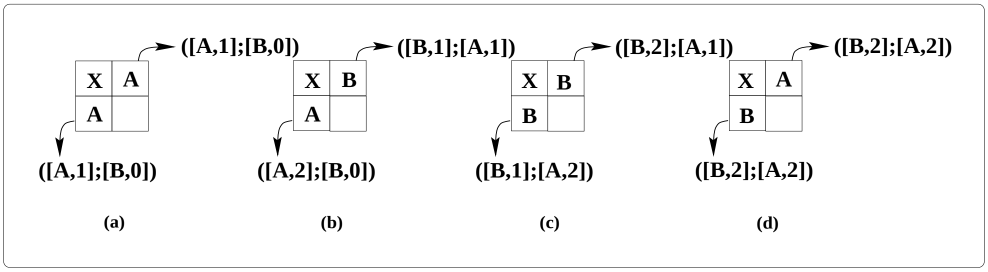


Figure 3.27: *The* 2 × 2 *puzzle from before is revisited. In* (d) *we see that all letter counters have reached their maximum values which enables the program to realize it must backtrack instead of starting over again.*

Going from infinite cycles to finite cycles

At first glance The scheme above looked like a winner but solutions disappeared and/or mysterious **finite cycles** showed up. Examining the problem further revealed the following unexpected scenario.

When a letter to a cell is picked it is chosen randomly among the letters in the letter supply. This will, if we are lucky, work, but most of the time it will not. Infinite cycles as we saw them before are gone, but *finite* cycles persist. Does this mean that the letter counter scheme is inadequate or what? No, the reason is to be found elsewhere.

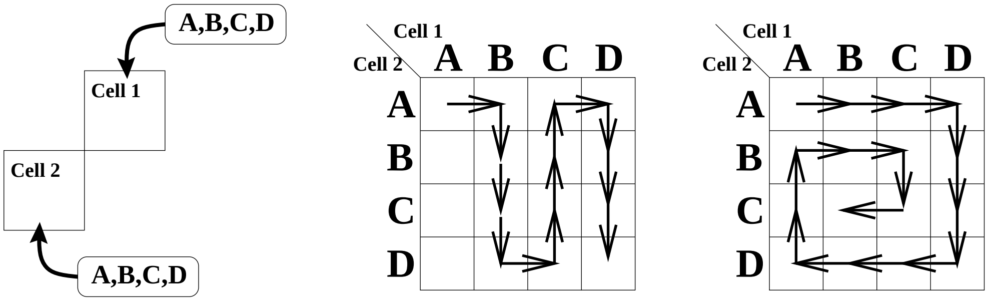


Figure 3.28: *Two combinatoric dependent cells,* **cell 1** *and* **cell 2** *both having the letter supply* {A,B,C,D} *are shown to the left. The two other figures show two different combination orders.*

In figure 3.28 the middle figure shows the order in which the letters were tested against each other. First (A,A), then (B,A), (B,B), ... ,(D,C), and (D,D). Three combinations still remain, but in order to test these we must re-engage the combinations (C,D) and (B,D) which will exceed the allowed counter values for these letters. Since we do not allow any counter to grow beyond the number of letters in the combinatoric dependent cells, the three missing combinations were never tested.
If the maximum letter counter values were raised the missing combinations were tested too, but the price was small finite cycles of additional combination tests.

Removing finite cycles

The finite cycles can be avoided if we either introduce a scheme in which two cells can change letter at the same time, or if we force the combination order to follow an order like the one shown to the right in figure 3.28. The latter was chosen because we do not want to change the single cell fill scheme to allow double cell fills. Forcing the combination order was done by changing the backtracking scheme.

Using the old method we find a potential backtrack cell. We then examine the letter counter for the currently inserted letter one more time and compare it with the equivalent letter counters in *closely combinatoric dependent* cells, if any. Then we settle on the final backtrack cell by asking

> if the two letter counter both are 1 **then** we can choose any of the two cells,
>
> if one letter counter has reached a maximum **then** the other cell is chosen,
>
> if both letter counters have reached a maximum **then** we must backtrack further,
>
> if none of the above situation **then** we choose the cell containing the letter with the highest letter counter value.

This scheme will lead to a combination fill order like the one shown to the right in figure 3.28.

3.9.4 Solving the 2 × m and n × 2 cycle problems

Within the $2 \times m$ and $n \times 2$ puzzles there are $(n - 1)$ and $(m - 1)$ nested 2×2 puzzles respectively. Using the letter counter scheme defined above we showed that filling each of these 2×2 puzzles will not create any cycles that is we are sure that the program will terminate. The question is now: are we losing any solutions? The following theorem will give an answer.

Theorem 3.2 The $2 \times m$ and $n \times 2$ puzzles are tested without any solution being lost when using letter counters.

Proof In a $2 \times m$ or $n \times 2$ puzzle all cells but the very first and the very last cell can be paired as *closely combinatoric dependent* cells. In the 2×2 case we have shown that letter counters ensure that **all** combinations between two combinatoric dependent cells are correctly tested. When all combinations between two such cells are tried we must backtrack which in the $2 \times m$ and $n \times 2$ cases is very simple because only two cells can be chosen. Choosing either one brings us back to another pair of *closely combinatoric dependent* cells or back to the very first cell which in neither case will result in the loss of solutions. ■

3.9.5 Solving the 3 × 3 cycle problem

In the $2 \times m$ and $m \times 2$ puzzles a cell can at most have one other cell to which it is closely combinatoric dependent. In the 3×3 puzzles and larger puzzles, cells can have two such cells making things a bit more complicated.
Using the scheme devised above without modifications on a 3×3 puzzle will fail and the reason is the three cell diagonal.
Now, before we continue we need to define

Definition 3.19 (Tightness) If all letter counters in one direction have reached their maximum legal value we say that the cell is *tight* in that direction. If the cell is tight in both directions the cell is *double tight.* If not tight at all it is *non-tight.* If the cell is closely combinatoric dependent to just one cell or to no cell at all (the cell is at the border or next to a closed cell) we will say that the cell is *(double) tight* too.

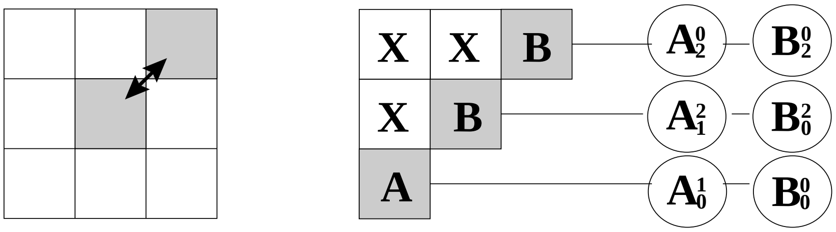


Figure 3.29: *In the figure to the left two cells in the diagonal are tight. To the right is shown the same diagonal but with letter supplies and letter counters added.*

Why the 3 × 3 puzzle fails

Definition 3.20 (Grid diagonal) A grid diagonal is a sequence of cells running diagonally **(1)** from the top border of the grid to the left border or **(2)** from the right border to bottom border of the grid ignoring closed cells. In an $n \times m$ grid the longest grid diagonal has got length $min(n, m)$ and the number of grid diagonal is at most $n+m-1$. Diagonals constituting of only one cell is called a *trivial diagonal.*

In our letter counter scheme developed above we work on a local basis, that is we only keep track of what have happened between the current cell and the associated closely combinatoric dependent cell(s). When diagonals become longer than 2 this local monitoring scheme will fail globally. If, say, the two cells in the upper right part of the diagonal in figure 3.29 are tight we will know that all combinations between the two were tested and that any tested combinations will not reappear. This mechanism was invented in order to avoid cycles but it does more than that - it **avoids** potential combinations to be tested on an overall basis. When a cell becomes double tight it is locked and will not be unlocked unless we backtrack beyond the diagonal it belongs to. Locking a cell is OK when the diagonal has got length 2 whereas higher lengths will give trouble. In figure 3.29 above two cells have been locked but a third cell in the same diagonal is not tested through. In order to do that we have to unlock the diagonal cells (in the example 8 combinations exist but only 5 are tested, see figure 3.30). Unlocking cells, however, must be done with care otherwise we will re-introduce cycles.

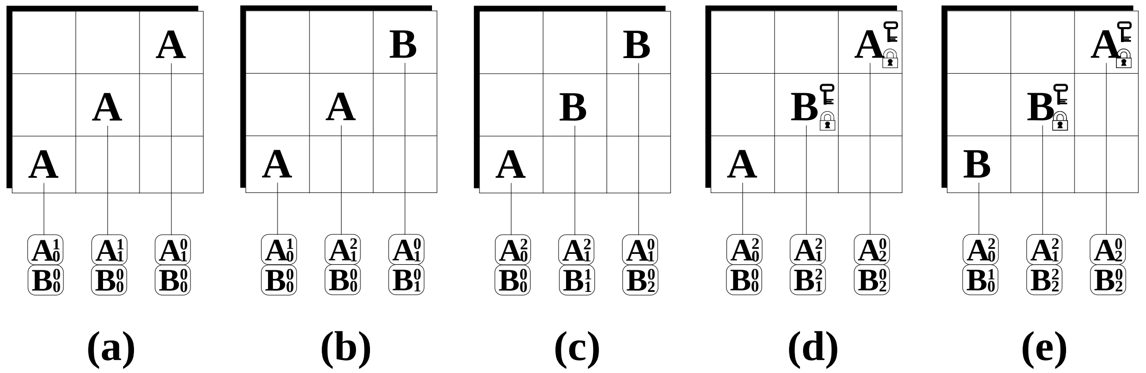


Figure 3.30: *The diagonal cells are shown with their supplies added. All three cells have supply size two. When the combinations* `AAA`, `BAA`, `BBA`, `ABA` *are found then the two upper left cells in the diagonal are locked. The last cell in the diagonal can make one more letter change creating* `ABB`, *but the remaining three combinations are overlooked.*

Solving the 3 × 3 puzzle

If we want the scheme to work on a global basis it seems that we must add a global dimension to the combination check. Fortunately, it is enough to look at grid diagonals. In the 2 × 2 puzzle there was only one non-trivial grid diagonal and all (both) cells were considered in each combinatoric insertion. In the 3 × 3 puzzle case we must extend this grid diagonal check. As we have already discussed, and as seen in figure 3.30, the reason we stopped prematurely was because cells were locked. In the example above we must move the lock (not remove it). In figure 3.31 below we see how the lock is moved from the two upper right cells in the diagonal to the lower left cell in the diagonal (Notice that the steps in figure 3.31 continue the steps shown in figure 3.30!!).

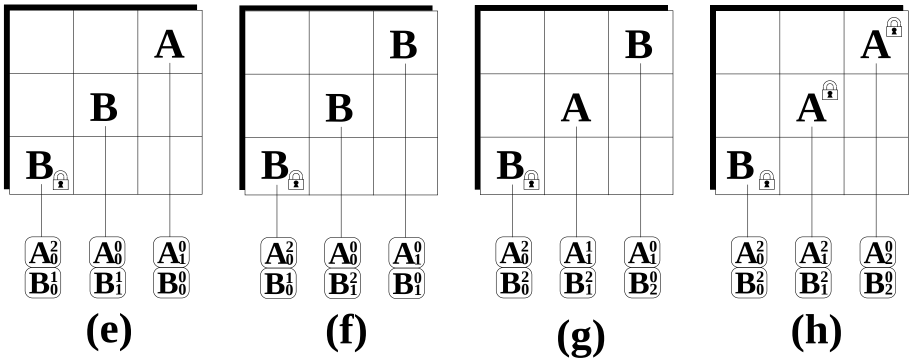


Figure 3.31: *The three remaining combinations are freed by moving the locks from the two upper left cells to the lower right cell in the diagonal.*

The locked cell in figure 3.31 has got two letters in its supply. That means we can lock the cell for good. If the supply had been larger we would need to unlock it, choose a new letter in the supply, lock it again and unlock the other two cells in the diagonal. The number of such unlock steps is $\mid c_{ll} \mid -1$ (the supply size of the lower left cell minus one).

Solving the 3 × n and m × 3

These puzzles contain three types of diagonals: **(1)** trivial diagonals, **(2)** diagonals of length 2, and diagonals of length 3. All these diagonal types, we have shown, can be tested without overlooking combinations.

3.9.6 Solving the general m × n cycle problems

In the puzzles with grid diagonals longer than 3 the lock and unlock design from earlier must be extended. Instead of just a lock/unlock scheme we need a lock and key-hierarchy.

The Lock

The lock can be associated with either a cell or a partial diagonal of length two (i.e two cells). Associating a lock to more than one cell at the time is new.

The Key

To each lock is associated a key which can unlock the cell or a partial diagonal. When all combinations in the cell(s) in the locked area have been tested, the key is thrown away and the lock remains locked until the cell(s) is/are either cleared or another key (see below) is found.

The Key Hierarchy

Keys belonging to the same grid diagonal are placed in a *lock hierarchy* in which keys at level n can unlock all locks associated to the keys at levels lower than n. The first key is the most powerful key, and when the first lock is locked for good i.e the first key is thrown away all the combinations in the grid diagonal have been tested.

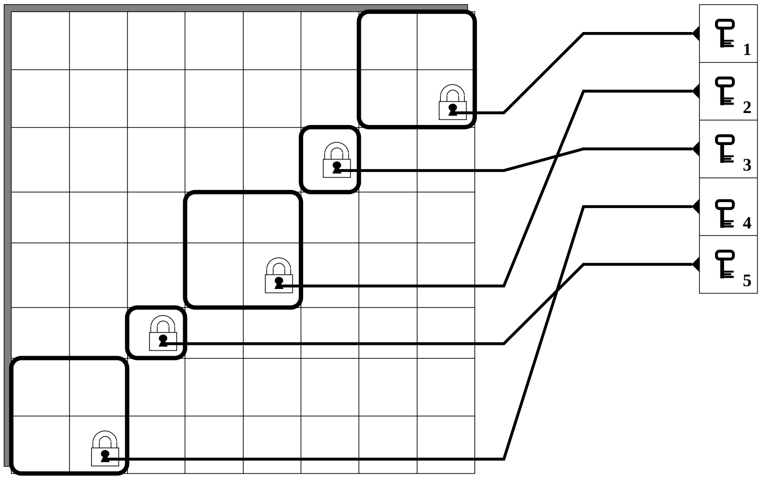


Figure 3.32: *A lock and key hierarchy with five levels.*

Definition 3.21 (Lock Order) When all the cells in a grid diagonal are locked, the order in which the locks were placed is called the *lock order* and is denoted by L.

Note that when all cells in a grid diagonal are locked, the lock order is equal to the order in which the associated keys appear in the key hierarchy.

Lemma 3.4 Let $L_1 = *[c_1] ... * [c_i] * [c_j] * ... [c_{n-1}] * [c_n]$ and $L_2 = *[c_1] ... * [c_j] * [c_i] ... * [c_{n-1}] * [c_n]$ be two lock orders, then the number of combinations checked in both lock orders is the same, $\mid L_1 \mid = \mid L_2 \mid$.

Proof The key with the lowest priority, $*[c_n]$, is used $\mid c_n \mid$ times before it is thrown away by the key $*[c_{n-1}]$ but as long as $*[c_{n-1}]$ exists the lock and key associated to c_n are re-installed. This happens exactly $\mid c_{n-1} \mid$ times. This argument is repeated for the keys $*[c_{n-1}]$ and $*[c_{n-2}]$ and again for $*[c_{n-2}]$ and $*[c_{n-3}]$ etc. until we reach the first key $*[c_1]$. The total number of times the diagonal is completely filled in with letters (and each time with a new and not previously seen combination) is equal the number of times the lock $[c_n]$ is locked. This number is

$$\mid L \mid = \mid c_1 \mid * ... * \mid c_n \mid$$

Since multiplication is associative the order of the elements does not matter. ■

Theorem 3.3 Using the lock and key hierarchy no cycles appear and no solutions are lost.

Proof The total number of letter combinations contained in a grid diagonal of length n is

$$G_d = \mid c_1 \mid * ... * \mid c_n \mid$$

where $\mid c_i \mid$ is the size of the letter supply for cell c_i. We will now show that using the lock and key hierarchy we test each of these combinations exactly once.

The lock and key hierarchy scheme chooses a cell c_1 as the first cell in the diagonal and associates with this cell a lock, $[c_1]$, and a key, $*[c_1]$. This first key is used when it is the only key left and when it is executed it unlocks and eliminates all locks with a lower key priority in the diagonal. Since the supply of letters in c_1 is $\mid c_1 \mid$ this will happen exactly $\mid c_1 \mid$ times. The order in which the rest of the diagonal cells are locked is done differently from time to time, determined by the dynamic algorithm used but according to the lemma above we can, however, assume that the order always is the same. Let the diagonal cell lock order be

$$L = *[c_1] * [c_2] * [c_3] ... * [c_{n-1}] * [c_n]$$

The lemma above also gives us that

$$\mid L \mid = \mid c_1 \mid * ... * \mid c_n \mid$$

I.e. $\mid L \mid = G_d$, and since we have equality no solutions are lost and since we also are sure that the algorithm terminates no cycles appear. ■

3.10 The Connection between Grid Walks and Search Trees

> **Words are the leaves of the tree of language, of which, if some fall away, a new succession takes their place.** *Field Marshall John French*

In section 3.3.1 we mentioned that search trees are in fact the ordered result of the search in the vast space of enumerated combinations. In the huge combinatoric space associated with even small puzzles and small word lists we will grow the trees on which solutions might appear. Now, is the time in which we will picture the trees associated implicitly created by grid walks.

3.10.1 The total search tree

The search tree can supposedly be depicted in various ways; here is how we see it:

- Let the root be at level zero. Then the number of children for each node on an even level, l_e (including the root), is $n - (l_e/2)$, where n is the number of open cells in the grid.
- The number of children for a node on an odd level, l_o, is the number of letters in the supply, n_{supply}, of the particular cell at this instance in the tree.

The depth of this tree is $2n$, where n is the number of open cells in the grid and the maximum width of the tree is $26^n n!$ in other words the maximum width of the search tree is **large** even for small problems.

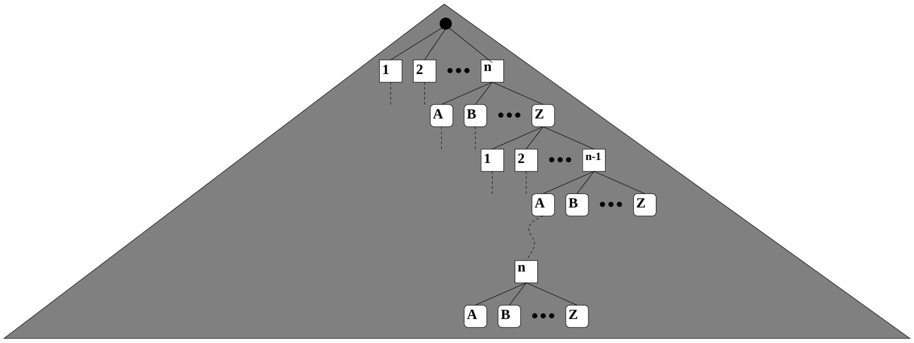


Figure 3.33: *The total search tree.*

We have earlier defined two types of grid walks: static walks and dynamic walks. The associated search trees do not look alike.

3.10.2 The static search tree

Static walks alter the *total search tree* a bit. At all even levels in the *total search tree* the number of children is reduced from $n - (l_e/2)$ to just 1 (static walks determine the fill order of the open cells before runtime). The *total search tree* can therefore be transformed into a new tree with height n and with maximum width 26^n. The maximum width is only reached when using an artificial enumerated word list, i.e. maximum width is never realized on proper word lists. But a rule of thumb is that the larger the word list we have the wider is the width of the associated search tree.

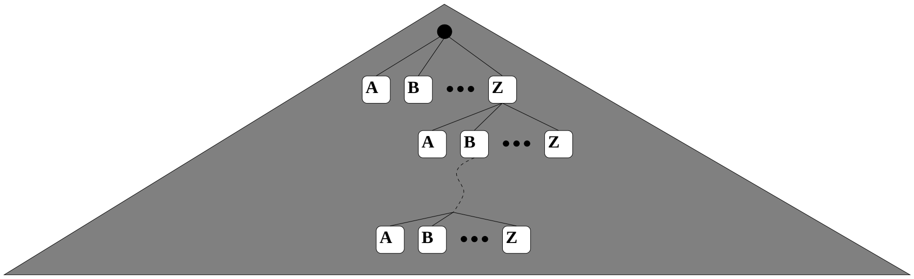


Figure 3.34: *The static search tree.*

3.10.3 The dynamic search tree

The dynamic search tree does not differ much from the static search tree in that the dynamic search tree too only have one node at each odd level in the tree. The difference is in the node-cell relationships. In the static search tree all nodes at an odd level would refer to the **same** open cell in the grid. In the dynamic search tree this is no longer true. The transformation of the search tree performed on the static search trees is not possible here so the depth of the dynamic search tree remain $2n$, but the maximum width is 26^n like for the static search tree.

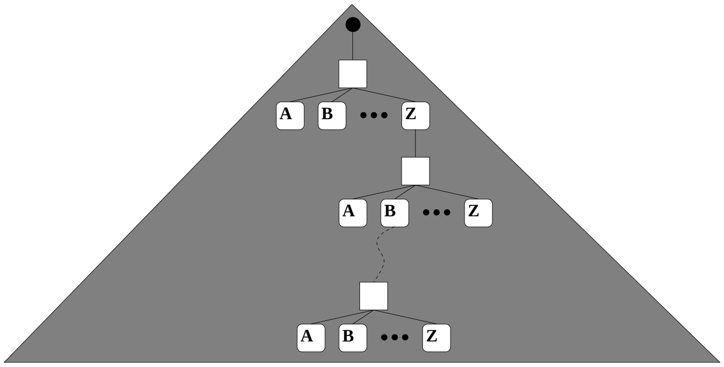


Figure 3.35: *The dynamic search tree.*

3.10.4 Backtracking revisited

Static search trees

When using static grid walks the search trees mentioned above will be searched in a *depth first* manner in which *naive backtracking* always brings us back to the parent of the node in which the blind end was encountered. Introducing *optimized backtracking* changes this; instead of moving just one level up in the search tree we move p levels up, where potentially $p = 1, 2, ..., n$. A significant number of subtrees are pruned of the tree unsearched.

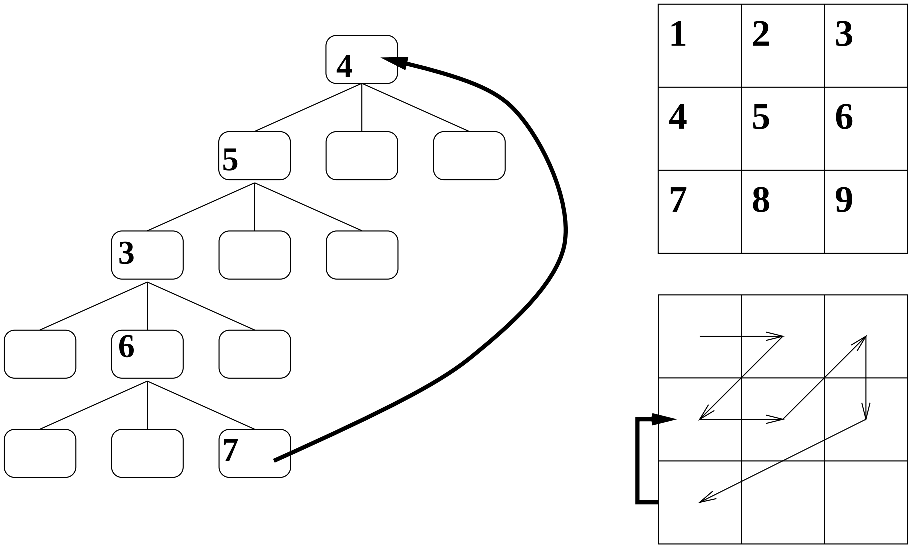


Figure 3.36: **To the right:** *A* 3 × 3 *grid is shown with cell numbers and a static prefix snakewalk is drawn showing the optimal backtrack from cell* 7 *to cell* 4. **To the left:** *Part of the search tree with indication of the multi level backtrack move.*

Dynamic search trees

The initial dynamic search tree is the same as the *total search tree* but as the order of the cells are chosen and backtracking is performed it is pruned down dramatically. A backtrack move here will always go up to a higher level in the search tree but not necessarily to an ancestor in the same subtree. When the backtrack node is found pruning will be performed on all nodes having the same grandparent as the backtrack node. What the pruning imply is:

- Only a node representing a cell can be chosen as a backtrack node.
- The grandparent has got, say, 6 children and each of these has got the same number of children, namely the number of empty open cells in the grid. When a cell-node is chosen the equivalent cousin nodes must be removed.

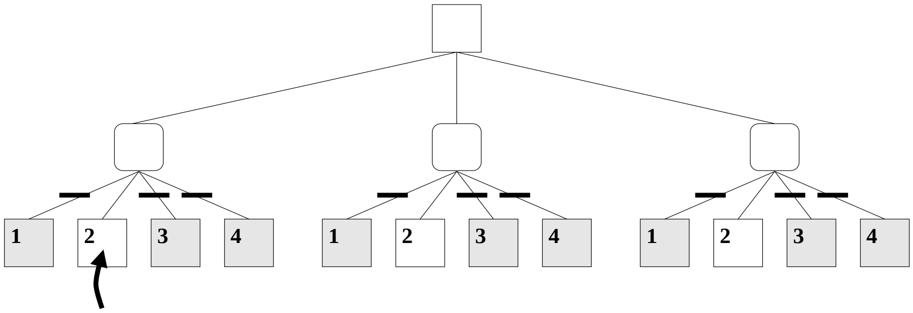


Figure 3.37: *A backtrack node is picked all siblings are removed. The equivalent preservation and removal among cousins in the search tree is also done.*

The reason why equivalent cousins are removed is that the remaining problem at this point is the same no matter which branch of the tree we have followed. Take as an example the

3×3 puzzle: filling the cells in the following two different orders, $\{1, 2, 4, 5\}$ or $\{1, 4, 2, 5\}$ will result in two different parts of the tree but the remaining problems are identical.

When the dynamic search is finished the nodes that are left represents a tree like the one in figure 3.35.

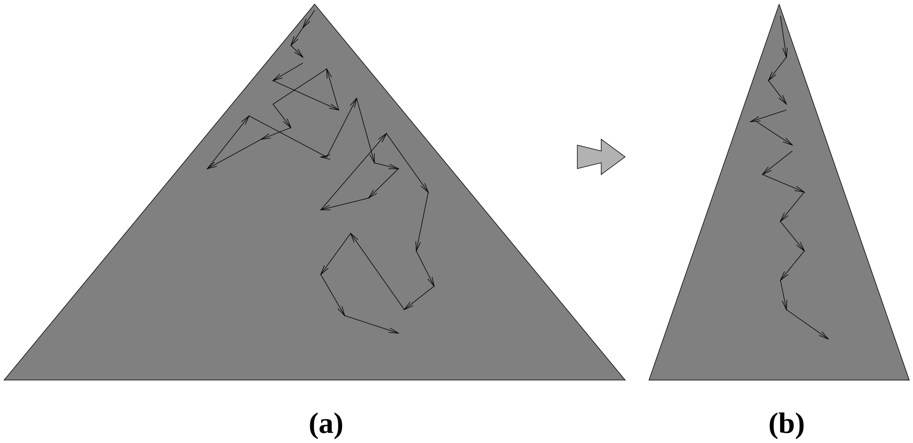


Figure 3.38: *The search through the tree as it may look in* (a) *the total tree and in* (b) *the dynamic search tree equivalent.*

3.10.5 Translation

We have just seen that filling a crossword puzzle grid is in fact a search in a search as a tree. Now, would it be suitable to view an arbitrary search tree as a crossword puzzle problem? Could we gain new information about the search if viewed as a CPP instead of tree? Maybe or maybe not. The transformation of a problem would certainly not be straight forward.

3.11 Design and Implementation of the Crossword Puzzle Tools

> **If the only tool you have is a hammer, you tend to see every problem as a nail.**
> *Abraham Maslow*

We will now define the structures that contain the physical crossword and the tools applied on these for data manipulation purposes.

3.11.1 The cells and the grid

Cells are for crossword puzzles what bricks are for houses - the time has come where we must define cells from a programmer's point of view. Apart from the actual letter, what other information are needed along with the cell? Surely we need to specify the neighbor cells of which a cell can have from 1 to 4. If the cell is a shared cell we need to specify the direction of the two words, that is we need two references to a list of letters that will preserve legality of the word slot when used in either of the two directions. Since we want both word slots to remain legal after insertion we will store a list of letters containing the intersection of the two lists just mentioned. More data could be stored but this will do for now.

Once the cells are defined we can start to build the grids. Given the height, the width and the positions of closed cells the puzzle is built. We assume here that all puzzle geometries are squares but stars, diamonds, etc. are legal puzzle geometries too (all these can, by the way, be enclosed in a square though).

The figures 3.39 and 3.40 depict how we view open and closed cells and how they are assembled to build a grid structure.

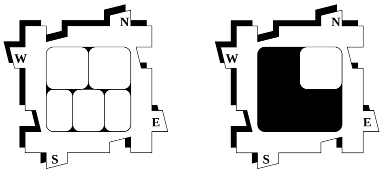
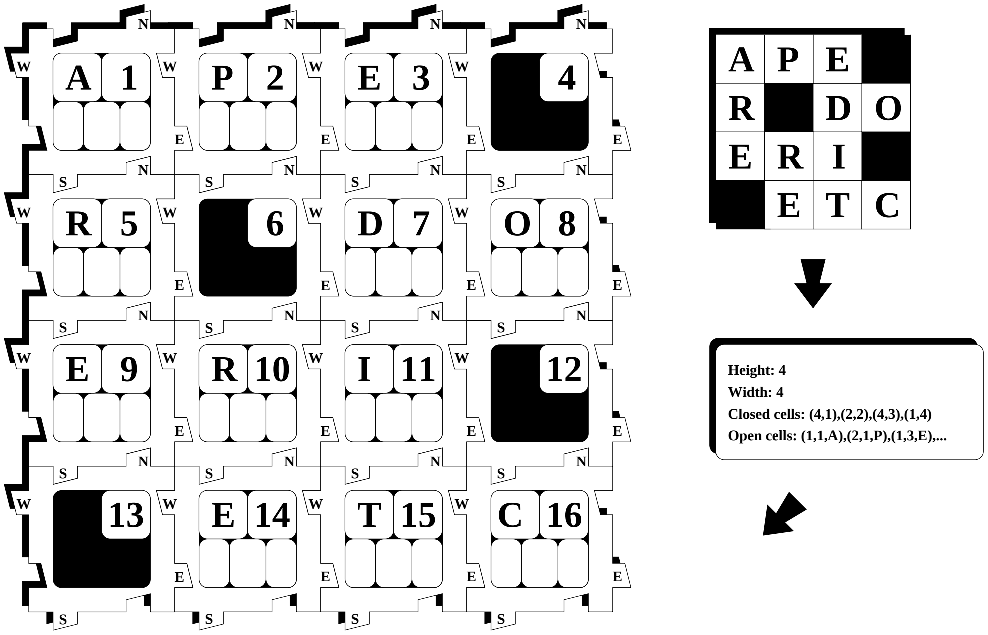


Figure 3.39: *The open cell and the closed cell.*



Height: 4
Width: 4
Closed cells: (4,1),(2,2),(4,3),(1,4)
Open cells: (1,1,A),(2,1,P),(1,3,E),...

Figure 3.40: *On the left is shown a small* 4×4 *puzzle and on the right is the equivalent data structure.*

3.11.2 Performing dictionary look-up

Assume we have an empty word slot. By making a series of look-ups in the dictionary we can assign a list of potential letters to each cell, e.g. at letter position 3 in a word of length 7 the possible letters could be: {a,g,e,r,j,u,y}. When inserting a letter in one of the word slots it will reflect the others in that each of the remaining empty cells must have their supply of potential letters updated, that is, for each empty cell we must make a new look-up in the dictionary. A word of length n will in other words require $\sum_{k=1}^{n} k$ look-ups. This number, however, can be reduced to n because it is a waste of work to update an empty cell if it is not to be filled next and when using prefix insertion a lot of empty cells can be ignored.

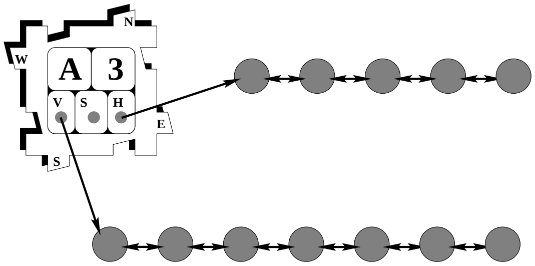


Figure 3.41: *References to the dictionary, horizontal and vertical directions.*

Prefix look-up: Remember the dictionary section on prefix supported design? References to the dictionary are easily made: the prefix in the word slot is the search key which defines the path down the digital search tree in the dictionary. The potential letters for the current cell are then the list pointed to at the end of the search.

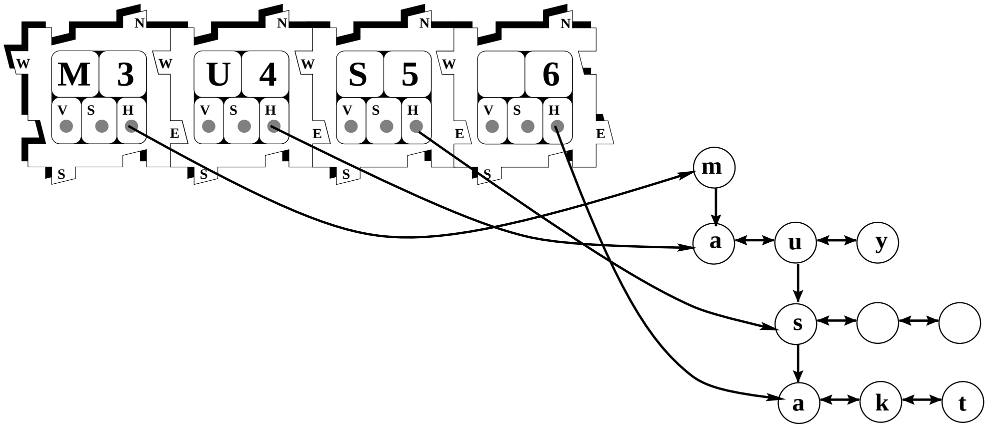


Figure 3.42: *Here we have a horizontal word slot containing four cells. The first cell has been given the letter* M*, the second cell the letter* U*, the third cell the letter* S*, and, finally, the fourth cell is empty. To the right is shown where in the digital search tree the different cells point and it can be seen that the letters* A, K *and* T *are legal for the last cell.*

Template look-up: If we have a template supported dictionary the dictionary, references will look a bit different. Instead of maintaining a list of possible letters in the cell we maintain a list of possible words. As the word slot is filled the list of continuing template matching words will become smaller and smaller. Another difference is that each filled cell in the same word slot in the end references a separate list of words (see figure 3.44).

3.11.3 Finding the legal letter supply

Up until now we have only discussed how the legal letters were found to a non-shared cell. If, on the other hand, the cell is shared (it belongs to a horizontal and a vertical word) we need to find the intersection of available letters in the two directions (see figure 3.43 and 3.45).

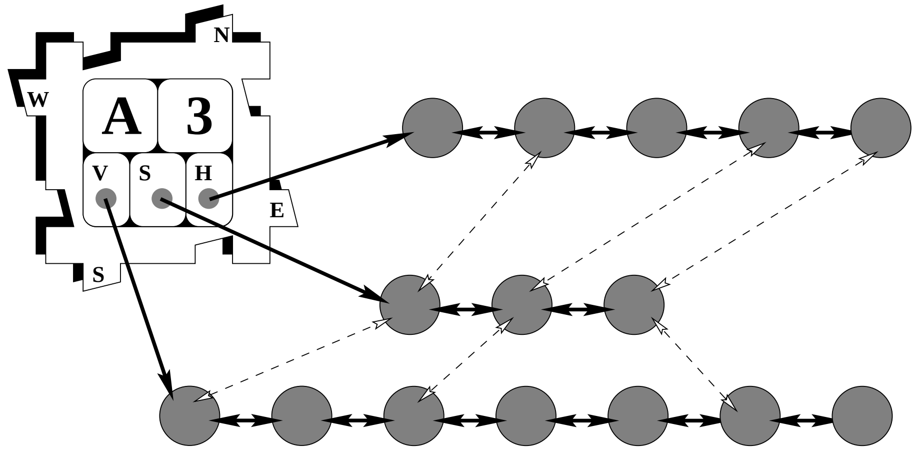


Figure 3.43: **The prefix dictionary:** *Letters found in both lists are maintained in a shared list - in the letter supply.*

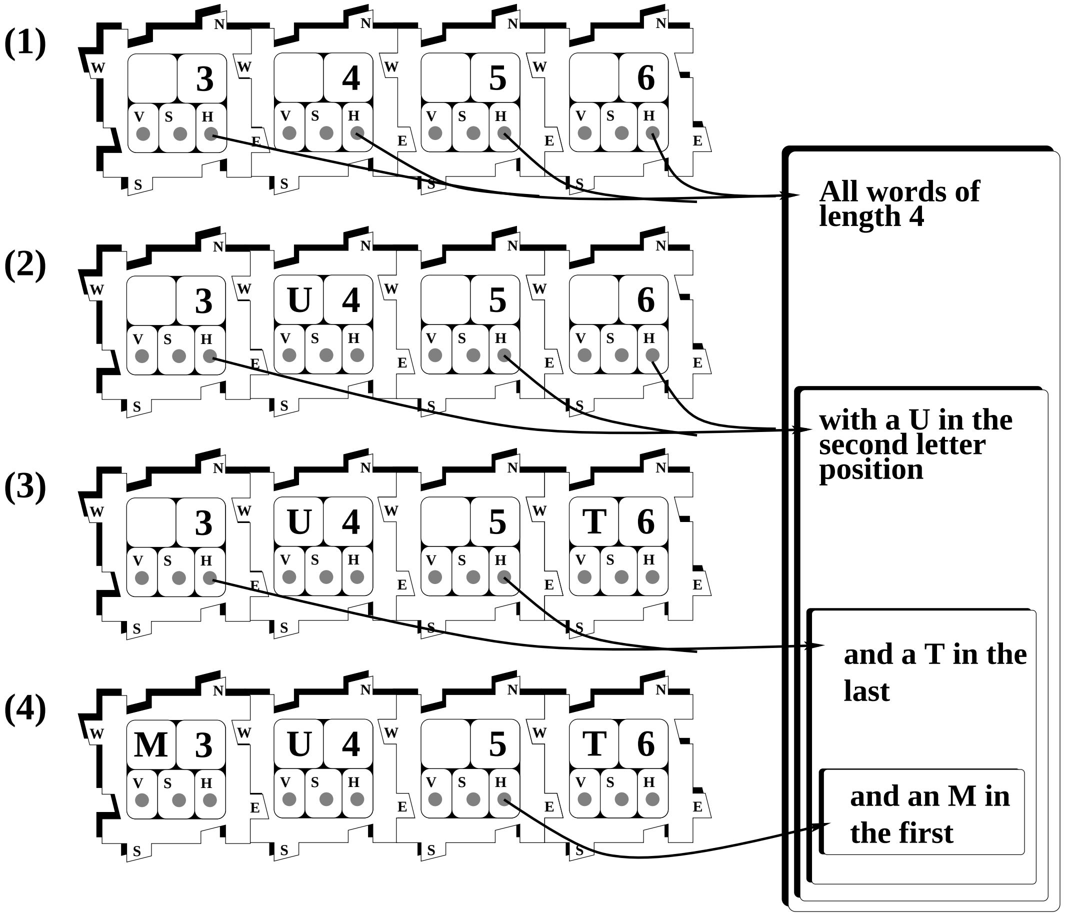


Figure 3.44: *When the word slot is empty all cells refer to the same list namely the list containing all words in the word list of the right length. When one cell is filled all the remaining empty cells refer to a subset of the original list where all words fulfill the current word slot configuration.*

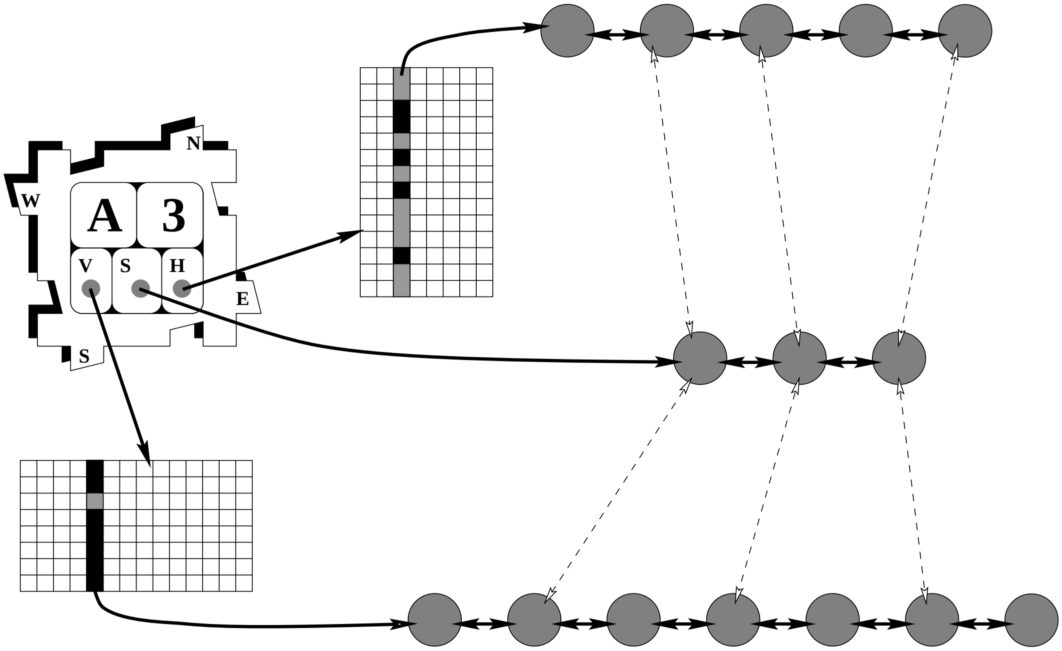


Figure 3.45: **The template dictionary:** *The two word lists are searched and a list of letters is found in which a letter is only present once. Finally, the intersection of these two lists is found.*

3.12 Implementing the crossword compiler parts

> **Goto, n.: A programming tool that exists to allow structured programmers to complain about unstructured programmers.**
>
> *Ray Simard*

¡BR¿ The various procedures and structures have been divided into five different modules, *the Dictionary module, the Cross Module, the Walk Module*, *the Back Module*, *the Timer Module*, and the main structure using the contents of these is found in the main file. The Dictionary Module was explained in the previous chapter and the rest will be discussed here. All implementation has been done using *standard ansi* **C**.

3.12.1 The Cross Module

The Cross Module contains the data structures and the procedures which relate to the physical crossword grid.

Data structure / The cell

The implementation of the cell has got

- one `char` to hold the letter,
- four `pointers` to the four possible neighbors,
- two `pointers` to the dictionary,
- a `pointer` to the letter supply, and

- an int to count the number of times the cell has been reset.

The closed cell is implemented in the same way and identified by a special character in the letter field. In a crossword puzzle the number of closed cells is low and it has therefore not been considered necessary to implement a special data structure for a closed cell.

Data structure / The Pit

The Pits are data structures used to contain the pre-generated walks. A pit has got

- a pointer to a cell,
- three pointers; one to the next pit, one to the previous pit and one pointer to a temporary pit (used when removing pits from the walk), and
- an int containing the number of pits in the walk after the current pit.

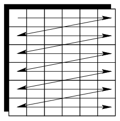
Procedures

Procedures to perform the following tasks have been implemented

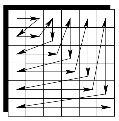
- Check grid size
- Find dictionary references
- Insert closed cells (and predefined letters)
- Merge two lists
- Choose, remove and insert a letter in a list
- Move around the grid
- Print the grid

3.12.2 The Walk Module

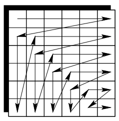
The Walk Module contains the procedures that create the walks. Among the many types of walks we have implemented six all of which can be supported by a prefix designed dictionary.



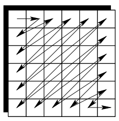
Straight



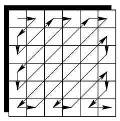
Snake



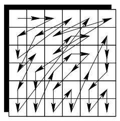
Switch



Sik Sak



Slalom



Dynamic

The first five walks are predefined and stored in a list before runtime, whereas the last one is dynamically stored in a sorted list which will change during runtime.

Defining a dynamic walk

Before we can define a dynamic walk we need to define

Definition 3.22 (Prefix extension) If o is an open cell then o is a *prefix extension* if the previous cell (above or to the left) is either **(1)** filled, **(2)** closed, or **(3)** not there. If o is a prefix extension in two directions, we say that o is a *double prefix extension.*

Lemma 3.5 Let O be the set of open cells in a grid and let U be the subset of O of unfilled open cells. If $U \neq \emptyset \Rightarrow \exists$ a set $u \subseteq U : \forall c \in u$ c is a double prefix extension.

Proof We have that $U \neq \emptyset$. Assume that $c_1 \in U$ and c_1 is not a double prefix extension. Since c_1 is not a double prefix extension an open cell $c_2 \in U$ exists situated above or to the left of c_1. If c_2 is not a double prefix extension either then yet another cell $c_3 \in U$ exists situated above or to the left of c_2. Since the grid has got a finite geometry $\exists n$ so that $c_n \in U$ and c_n is a double prefix extension. That is $c_n \in u$ and we therefore have $u \neq \emptyset$. ■

What the lemma above tells us is that as long as there are empty cells, we can always find and fill a cell and preserve the prefixes in all word slots.

The dynamic walk we will define uses this property. Every time we insert a letter we will choose to fill a prefix extending cell which we have just proven is always possible to find. The next question is which of the prefix extending cells (if more than one exists) should we choose. We can either choose it randomly or we could use weights. We have used both (weights became, during the process more or less, randomized).

Weights can be defined in many ways, but the weights we have defined depend on the sizes of the supplies in the involved cells. Each cell has got a supply of possible letters to be inserted (otherwise backtracking must be done) and by choosing the cell with the smallest supply we have in a sense chosen the hardest cell. Choosing the hardest cell ensures that failures will be provoked to happen as early as possible.

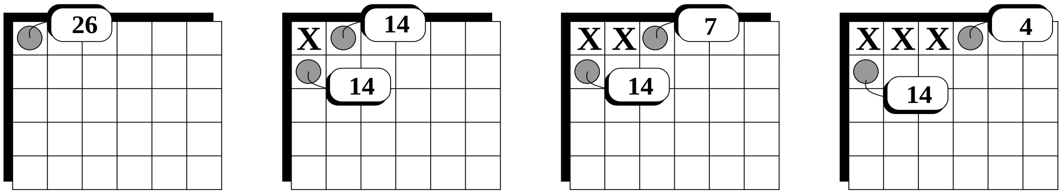


Figure 3.46: *The first four steps in the dynamic walk on a* 6×5 *grid. The numbers indicate the letter supply sizes.*

The example in figure 3.46 shows that the dynamic walk using supply weights progresses a lot like the Switch Walk and this would also be the case if we did not encounter supply weights equal to zero (i.e. no backtracks necessary). The backtracking associated with the dynamic walk is by nature dynamic too and after a few backtracks the walk will differ a lot from the Switch Walk.

This was a discussion on the prefix supported walks we have implemented, but walks not supported by prefix designed dictionaries have been considered, yet not implemented.

3.12.3 The Back Module

Static backtracking

The Back Module contains the backtrack procedures discussed so far associated to each of the walks defined in the Walk Module. The implementation for the static walks has been straight forward as explained in section 3.6.5, and both naive and optimized backtracking have been used:

```
back ← BACKCELL (cell)
    if (optimized backtracking is possible) then
        back ← OPT(cell)
    else
        back ← cell[PREVIOUS]

cell ← BACKTRACK (cell)
    back ← BACKCELL (cell)
    while (cell ≠ back) do
        Clear (cell)
        cell ← cell[PREVIOUS]
```

```
Clear (cell)
    delete current letter from supply
    remove letter from cell
    if (cell[EAST] exists) then
        clear cell[EAST][horizontal dict]
    if (cell[SOUTH] exists) then
        clear cell[SOUTH][vertical dict]
```

Determining if optimized backtracking is possible we have to look at the position of the cell. If the current cell is the first cell in at least one of the (two) word slots it belongs to, optimization is possible. If so, we may move backwards in the walk until we meet either the cell above the current cell or the cell to the left of the current cell.

Dynamic backtracking

Dynamic backtracking is required when all cells in the set, u, of double prefix extension cells have empty letter supplies. Then we need to determine two things: **(1)** which cells can we backtrack to without losing solutions, and **(2)** which of these is the best to choose.

Definition 3.23 (Optimal backtrack cell) A cell, c is a possible, *optimal backtrack cell* if c is an open cell, c has got a non-empty supply of letters, and all other cells in the largest possible rectangle with c as upper left corner have empty supplies. In the set C of potential optimal cells the *optimal backtrack cell*, c_{opt}, is the cell with the smallest supply of letters. If two cells are equal one is chosen randomly.

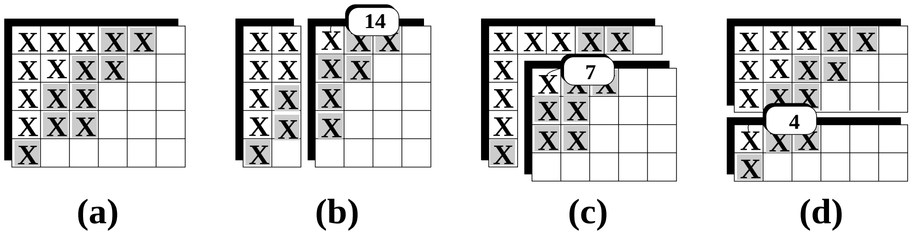


Figure 3.47: *In* **(a)** *a backtrack situation is shown. Cells with empty letter supplies are painted grey.* **(b)**, **(c)**, *and* **(d)** *show the three possible backtrack cells and the rectangles they belong to. The backtrack cell in* **(d)** *has got the lowest supply of letters and is therefore chosen.*

Lemma 3.6 If we backtrack to a potential backtrack cell we will not lose any solutions.

Proof The only way we can lose solutions is if we do not examine all possible combinations. Since we backtrack to a cell to which **all** later cells have empty supplies all combinations have been examined - and no solutions are lost. ■

This scheme has been implemented using a recursive scheme

```
back ← BackFind (List)
  if (List is empty) then
    back ← Failed
  else
    for (cell ← List[CELL]) do
      if (cell[NORTH] exists) do
        BackList ← Add (cell[NORTH], BackList)
      if (cell[EAST] exists) do
        BackList ← Add (cell[EAST], BackList)
    BackList ← DeleteDoublets (BackList)
    BackList ← DeleteFalse (BackList)
    back ← Min (BackList)
    if (back = NONE) then
      back ← BackFind (BackList)
  return back
```

The first **BackList** is the list of prefix extending cells. **DeleteDoublets (BackList)** removes double occurrences of the same cell in the **BackList**. **DeleteFalse (BackList)** removes cells from **BackList** which are not legal. If two cells belong to the same word slot we can not backtrack to the foremost cell because we thereby may lose solutions; these are not legal. **Min (BackList)** returns the cell in **BackList** which has the smallest supply of letters. If all cells in **BackList** have zero supply a 'NONE' value is returned.

Source code can be seen on page 154 to the left.

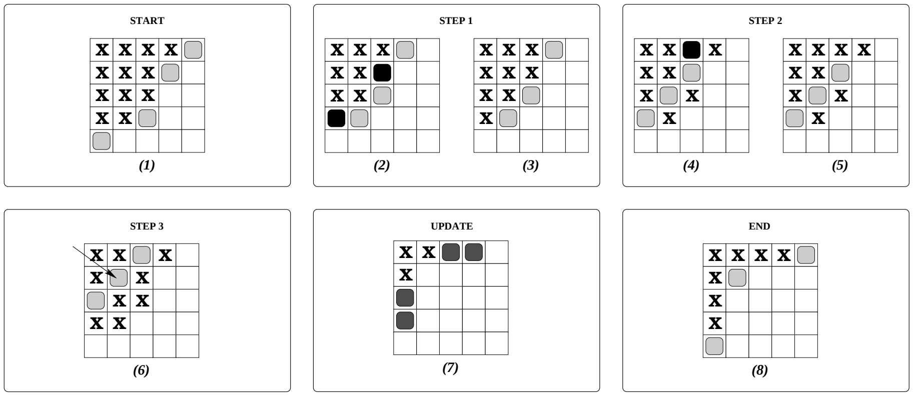


Figure 3.48: *The figure shows a backtracking example in which three backtrack lists were produced before an optimal cell was found.* **START (1)** *the cells in the prefix extending list is marked with grey boxes.* **STEP 1 (2)** *the first backtracking list is found . False cells are marked with black boxes.* **(3)** *false cells are removed.* **STEP 2 (4)** *the second backtrack list is found and one false cell is marked.* **(5)** *the false cell is removed.* **STEP 3 (6)** *the third backtrack list is found with no false cells but with an optimal backtracking cell.* **Update (7)** *the grid is updated.* **END (8)** *the grid after backtrack. The three new prefix extending cells are marked.*

Resets

Updating the grid involves two types of resets: **(1)** clear the cell, and **(2)** reset the cell supply. Cells which are fill dependent with the chosen backtrack cell must be cleared and those cells which are combinatoric dependent must have their letter supply reset.

3.12.4 The Timer Module

The Timer Module contains one procedure which measures CPU time. It is used to measure the time used by various operations (measured in micro seconds). This module will be used extensively in the next chapter (on benchmarking).

3.13 Assembling the pieces

> **If A equals success, then the formula is: A = X + Y + Z. X is work. Y is play. Z is keep your mouth shut.**
>
> *Albert Einstein*

With what we have now, we are able to define a crossword puzzle compiler - all we need is the motor which in this context is the specification of the filling sequence and the definition of how to backtrack. Say, we know that, let us assemble the pieces and see how it will work. Recall from chapter 2 the four first stages in crossword puzzle compilation:

(1) Creation of the host grid: Given the height and width of the puzzle we can define the internal representation of the grid.

(2) Determination of the overall design: Given a specification of where to place the closed cells these are placed into the grid.

(3) Specification of word slots: Add references from open cells to the dictionary.

(4) Identification of shared cells: For the cells belonging to two word slots we find the intersecting set of letters legal for both word slots.

The two stages

(5) Construction of one or more solution will be discussed in the next section.

(6) Composition of a clue set for each solution set will be discussed in chapter 7.

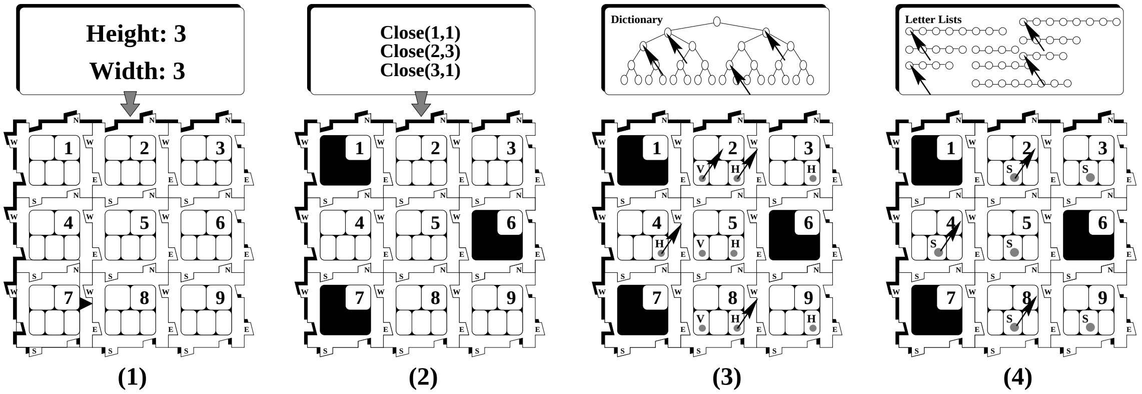


Figure 3.49: **(1)** *Creation of the host grid.* **(2)** *Determination of the overall design.* **(3)** *Specification of word slots.* **(4)** *Identification of shared cells.*

3.13.1 The Crossword Compiler

The crossword compiler can now be implemented.

```
Crossword Compiler (tag)

walk ← Prewalk (tag)
back ← Preback (walk)
while (tag[Supply] not empty) do
  cell ← SetDict (cell)
  cell[supply] ← GetStock (cell)
  if (cell[supply] is empty) then
    cell ← BackTrack (cell)
  else
    cell[Letter] ← SetLetter (cell[Supply])
    cell ← NextCell (cell, walk)
```

Source code can be seen in Appendix H.

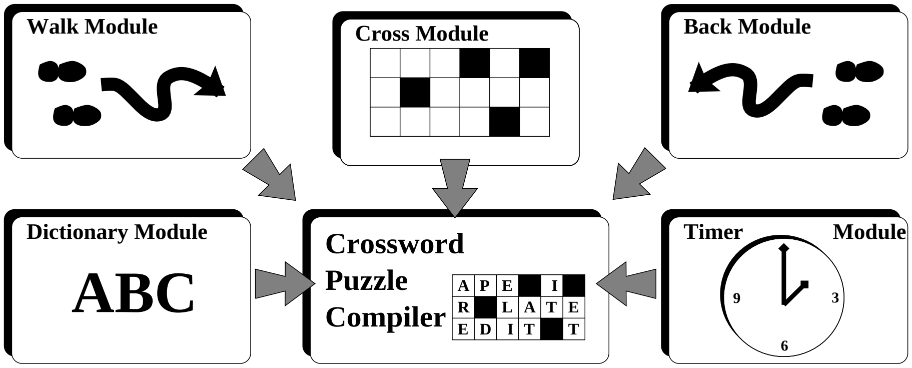


Figure 3.50: *The crossword puzzle compiler and the modules.*

3.14 Implementing the lock and key hierarchy scheme

> **"Who are you and how did you get in here?" "I'm a locksmith. And, I'm a locksmith."**
>
> *Leslie Nielsen as Lieutenant Frank Drebin,"Police Squad"*

The locks and keys have been designed and prepared for implementation but not yet made real (implemented) for the following two reasons:

- Too much time has been spent on implementing the designs - reaching even further into the time schedule with what seems to be endless steps forwards and backwards were given lower priority than the work on comparisons and estimation (see the next two chapters).

- Though in a static fashion, the two *diagonal grid walks* [4] do employ a diagonal lock scheme. Implementation of these instead were considered a fair replacement but in future work we would of course include an implementation and comparison in which a test on static versus dynamic diagonal lock schemes were performed.

Finally, if we look at the how the static and the dynamic diagonal grid walks progresses we will not find the big differences. It is true that the dynamic version allow an undulation like movement back and forth whereas instead the static version moves back and forth in zigzag course. This does not, however, change the fact that the area to be filled next is the lower right triangular area of the grid. A selected triangle is completely finished in the static version whereas in the dynamic it is visited several times before completion [5]. Where we probably will see a difference is of course in the order solutions are found and in the speed by which the first solution is found. Dynamic implementations in our work has been designed bearing in mind that a solution could be found within reasonable time - not to speed up the completion of the entire search space. Only pruning a trimming would help in that case.

Extra data structures

The following is a list of extra data structures needed, augmentations of others, and operations on them.

[4] See section 3.6.12.

[5] *Completion* is here the test of all letter combinations within the triangular area.

- The cell was defined in the beginning of this section. An extra lock indicator is added now so as to distinguish locked and unlocked cells.

- Each diagonal will have a list of keys sorted by importance as depicted in figure 3.51.

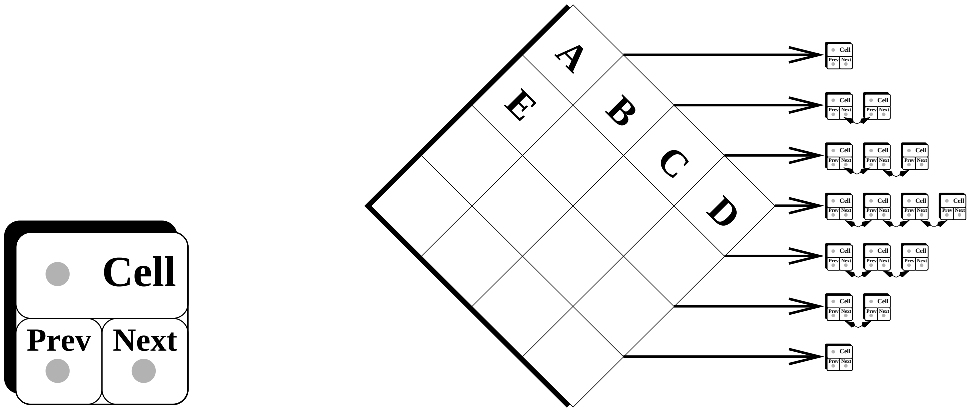


Figure 3.51: **To the left:** *A key as viewed by the programmer.* **To the right:** *A* 4 × 4 *grid with the* 5 *key hierarchies associated to it.*

And operations

- `Create` and `Destroy` a key.

- `Insert` and `Delete` a key.
  Insertion is always in the back of the key list, i.e. the most important key first. A key is deleted by the cell it belongs to.

- `Find` a key.
  When a cell must be unlocked the key with the lowest possible priority is found.

3.15 Results - Puzzles on display

> **There is nothing more exhilarating than to be shot at without result.**
>
> *Winston Churchill (1874-1965)*

This section contains a sample of puzzles produced with the programs described in this chapter. The puzzle geometries we have chosen are some of those used by Mazlack [Maz76a, Maz76b], Smith and Steen [SS81], Ginsberg et al. [ea90], and Berghel and Yi [Ber87].

3.15.1 Mazlack's puzzle geometry

Mazlack produced many small puzzles (he had only words of length 2, 3, and 4) all with low *density* and low *degree of interlocking.* Here is the largest puzzle geometry he used and one of our solutions to it.

D	Y	N	E	■	H	■	U	T	■	Q	U	A
E	W	E	■	■	E	■	N	■	■	I	D	S
V	I	T	A	■	W	H	I	P	■	■	O	K
I	S	■	Y	O	■	E	T	A	■	A	S	S
■	■	P	S	I	S	■	■	L	I	S	■	■
U	R	E	■	K	I	W	I	■	L	A	K	H
D	A	■	■	S	T	E	N	■	■	■	■	Y
O	T	I	C	■	E	D	D	O	■	I	K	E
■	■	B	O	N	■	■	O	U	R	N	■	■
P	R	O	■	U	D	S	■	I	O	N	S	■
R	I	■	■	B	R	N	O	■	O	■	Y	A
O	P	T	■	■	A	I	N	■	■	■	P	I
G	E	O	■	E	G	G	S	■	C	R	E	D

Geometry:	13 × 13
Density:	70%
Degree of interlock:	92%

Solutions were produced 'just like that' and in a few minutes hundreds of solutions were found. If we look at the grid will see that it is divided into five areas: three parallel diagonals running from top left to bottom right of the grid, and two triangular areas above and below the diagonals. Each of these areas are connected by just one open cell. This separation of the grid into subareas lowers the *fill dependency* and the *combinatoric dependency* considerably and along with a low density explain why solutions are found 'just like that'.

3.15.2 Smith and Steen's puzzle geometry

Smith and Steen used a symmetric puzzle geometry ....

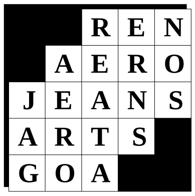
J	U	M	E	L	L	E	■	O	■	J	■
I	■	B	■	U	■	O	O	N	S	E	S
M	O	I	S	T	■	N	■	L	■	T	■
C	■	R	■	E	■	I	R	O	N	S	■
R	H	A	B	D	U	S	■	O	■	■	I
A	■	■	A	■	■	M	E	K	O	N	G
C	Y	C	L	E	S	■	■	E	■	■	N
K	■	■	L	■	E	M	E	R	I	T	I
■	J	N	A	N	A	■	L	■	L	■	T
■	Y	■	S	■	S	■	V	O	L	T	E
I	N	I	T	I	O	■	E	■	E	■	R
■	X	■	S	■	N	A	S	A	R	D	S

Geometry:	12 × 12
Density:	71%
Degree of interlock:	71%

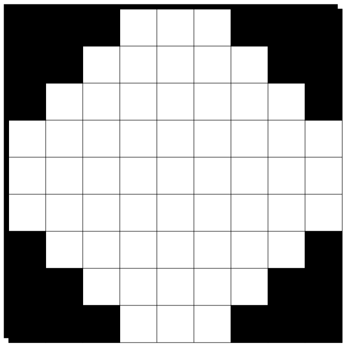
Solutions to this grid were easily found too. The low toughness level of this grid can be explained by the low density and the low degree of interlocking.

3.15.3 Ginsberg's puzzle geometries

Ginsberg et al. tested their programs on different puzzle geometries with varying difficulty levels.

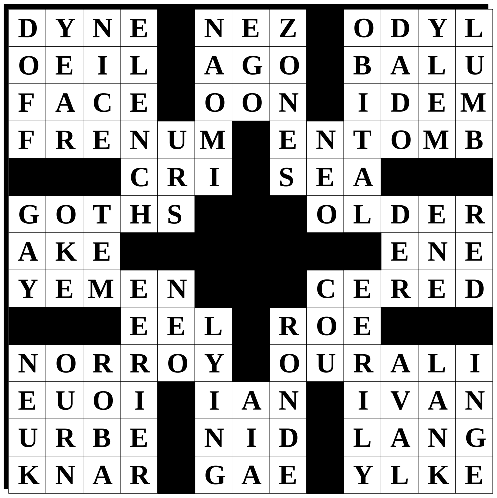


		R	E	N
	A	E	R	O
J	E	A	N	S
A	R	T	S	
G	O	A		

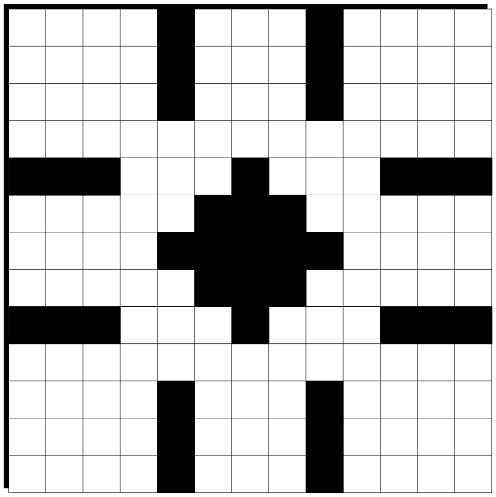


Geometry:	5 × 5	**Geometry:**	9 × 9
Density:	76%	**Density:**	64%
Degree of interlock:	Complete	**Degree of interlock:**	Complete

The small 5 × 5 puzzle is simple to solve and needs no further discussion. The 9 × 9 on the other hand is much (much) tougher. The density is low but the their position in the grid do not affect the difficulty level much (since they are positioned along the borders). The degree of interlocking is complete, i.e. all cells are shared cells. The geometry is also what we call a wide open puzzle (it has got an wide center with three 9-letter words stacked on top of each other and another three 9-letter words (stacked) crossing these). This grid configuration maximizes the *fill dependency* and the *combinatoric dependency.* No solutions were found within reasonable time (more than 24 hours) with any of our crossword compilers.



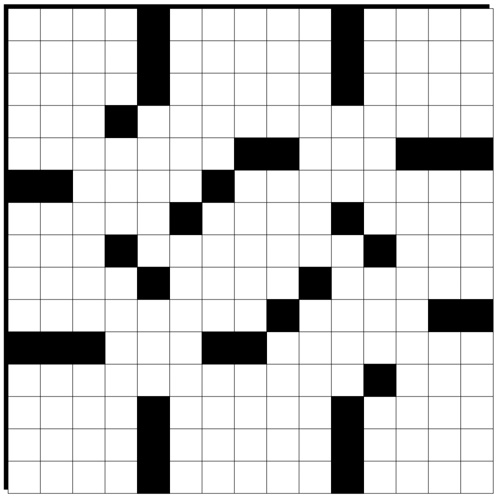
D	Y	N	E		N	E	Z		O	D	Y	L
O	E	I	L		A	G	O		B	A	L	U
F	A	C	E		O	O	N		I	D	E	M
F	R	E	N	U	M		E	N	T	O	M	B
			C	R	I		S	E	A			
G	O	T	H	S				O	L	D	E	R
A	K	E								E	N	E
Y	E	M	E	N				C	E	R	E	D
			E	E	L		R	O	E			
N	O	R	R	O	Y		O	U	R	A	L	I
E	U	O	I		I	A	N		I	V	A	N
U	R	B	E		N	I	D		L	A	N	G
K	N	A	R		G	A	E		Y	L	K	E



Geometry:	13 × 13	**Geometry:**	13 × 13
Density:	76%	**Density:**	78%
Degree of interlock:	Complete	**Degree of interlock:**	Complete

These two puzzles above look quite alike but the small differences they exhibit has a dramatic change in the number of solutions we can expect to find. The densities are 76% and 78% respectively but the four 'missing' closed cells in the grid geometry to the right introduces four 13-letter word entries and apart from being introduced to the grid they do also cross. The left grid was easy to find solutions to but the right was given up on (24 hours after search initiation). The lesson to be taught here is the density in some cases reflects the toughness of a puzzle geometry but far from always.

3.15.4 Berghel and Yi's geometry



Geometry:	15 × 15
Density:	87%
Degree of interlock:	Complete

Berghel and Yi tested their program on various easy grid geometries (not shown here) and as an example of a *real* grid geometry they presented this grid. They did not, however, present a solution to it. We tried to solve it and did not find a solution neither within the time limit (24 hours).

3.15.5 Other geometries

Various so called fantasy grids or 'eye-popping' geometries can be defined and the constructors (man or machine) who solve one are sure to have their name quoted among crossword fanatics thereafter. Here are three examples

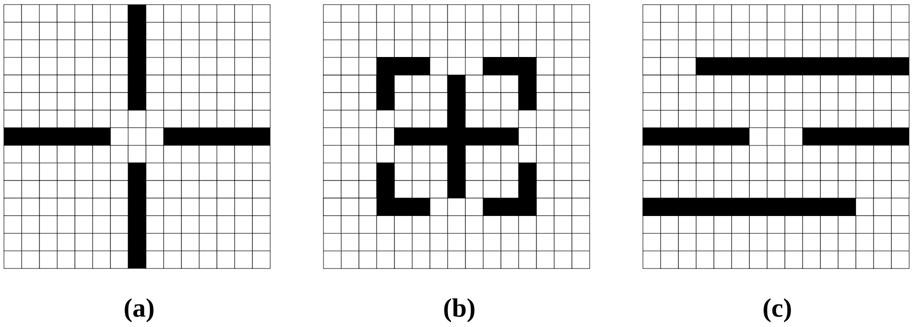
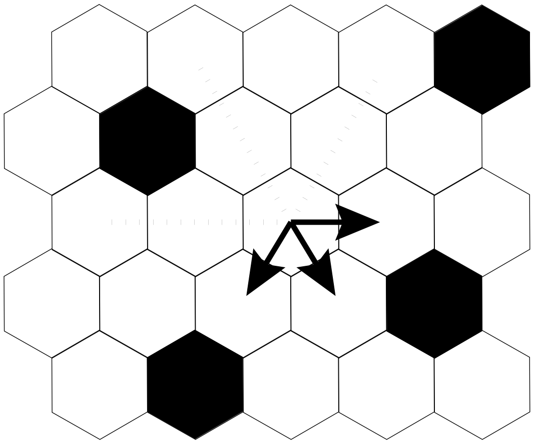
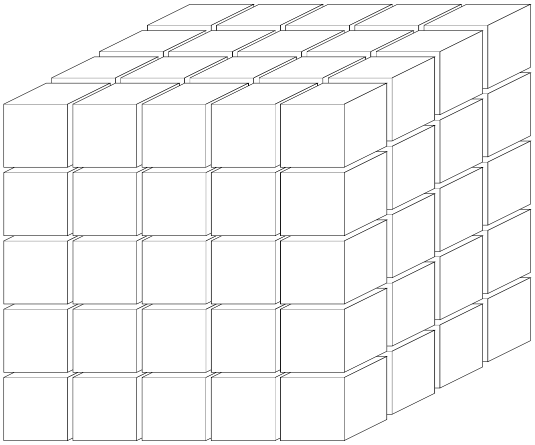


Figure 3.52: *Three 'eye-poppers'.*

Another group of specially tough puzzle geometries are square puzzles which are square geometries with no closed cells at all. These type of puzzles have been given great attention over the years and puzzles up to 9 × 9 have been solved (found using a computer). In chapter 6 we will take a closer look at this type of puzzle.

If we for a moment let our imagination free we can extend puzzle geometries to include honeycomb shaped open cells or what about 3D puzzles? Both types allow three words to cross in one single cell.





Both types of puzzles have been mentioned before, but never solved (not that it would be too hard): **Honeycomb** Jarosi Peter, Medical University of Pecs, Hungary. **3D puzzles** "Three-Dimensional Crossword Puzzles in Hebrew", H. Bluhme, 306–309, Information and Control, sep, 1963", Vol 6, number 3.

Other geometries in 3D can be defined - somehow it is only the imagination that sets the limit here ...

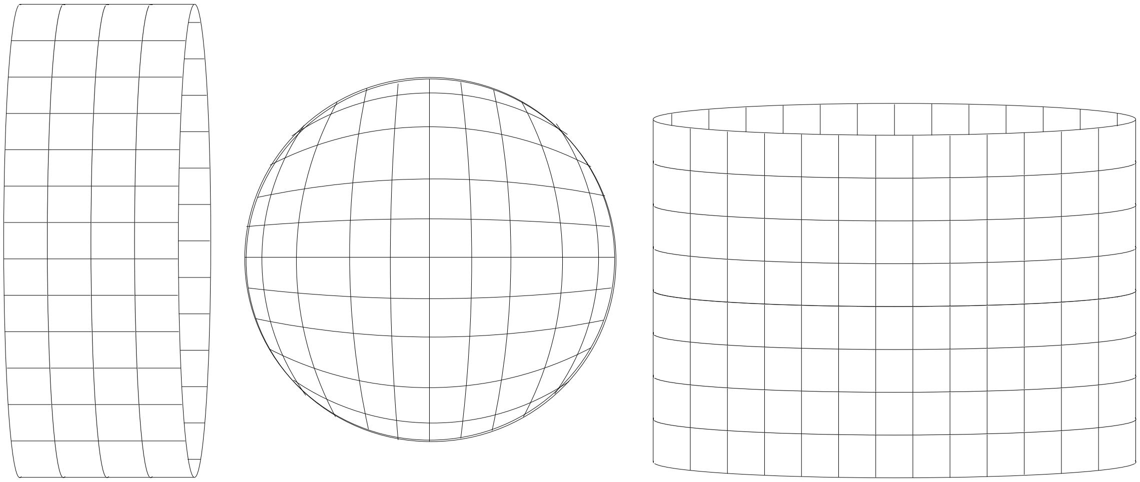


Figure 3.53: *Three dimensional grid geometries.*

Chapter 4

Benchmarking

To compare is not to improve.

Field Marshall John French

In the last chapter we discussed a line of methods capable of solving the CPP, and we saw how a general scheme could be defined which was capable of embracing, not all, but most earlier procedural knowledge approaches. By using *grid walks* we were able to define various methods some good, some bad, but all of them correct. Both static and dynamic prefix walks were implemented and run, and most problems could be solved with these methods and those that could not were recognized as particular tough problems (some which it is uncertain whether there do exist solutions at all). Time has come when we must define a model in which previous work and our own work can be compared among each other and against each other on an equal basis independent on hardware and compilers. We will in the following use schemes in which only algorithmic designs matters. What we need is a benchmark scheme.

4.1 Preliminary comments

The best way to keep one's word is not to give it.

Napoleon Bonaparte (1769-1821)

Over the years many different algorithm designs have been suggested - all different and all using different constructions, but what holds true for all of them is the fact that given an unambiguously stated problem they (should) all produce an equivalent unambiguously stated output solution set. Comparisons between compilers are done using tests which in a well defined way extract measures telling us how well a certain method perform; how many words are tested, how many times is a set of words re-used etc. Counting words and/or letters is in this context an implicit measure on how large a part of the search tree we actually visit and/or how many times. The best method will therefore be the one that minimizes the number of nodes visited in the search tree. Or stated combinatorically: the best method is the one that tests the fewest (partial) combinations without missing solutions.

4.2 An unambiguous input design

> **Do you know that doing your best is not good enough? First you must know what to do.**
> *W. Edwards Deming*

There are two types of inputs that must be present: **(1)** a puzzle geometry and **(2)** a word list specification. So in order to define a benchmark design specification these inputs must be settled on first. A set of grids must be chosen which will force the crossword compiler into all possible situations, and the word list should likewise posses this ability. It has in previous work been useful to construct artificially produced word lists (enumerated word lists, see definition below) and thereby keep this input constraint under control. What is left is now just a matter of monitoring the chosen compiler and see how it will perform - will it produce the correct (expected output) and how 'fast' and how well?

Definition 4.1 (Totally enumerated lexicons) A *totally enumerated lexicon* consists all possible permutations of a set characters.

A benchmark using a totally enumerated word list is not dependent on specific characteristic of any natural language.

4.3 Previous work - Benchmarking

> **One machine can do the work of fifty ordinary men. No machine can do the work of one extraordinary man.**
> *Elbert Hubbard*

Two papers have been published on the subject:

4.3.1 H. Berghel and R. Rankin

A Proposed Standard for Measuring Crossword Compilation Efficiency [HB90a]

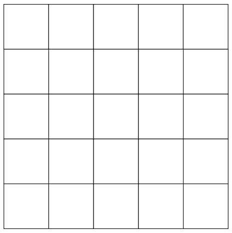
Berghel and Rankin developed a benchmark test capable of testing the average run performance as well as the worst case performance. Their philosophy was to make the test simple, yet hard.

INPUT: As input Berghel and Rankin settled on a 5×5 grid with no blanks and a word list with 134 words. The puzzle geometry should maximize the number of fail points in the search space and with an open grid this was achieved. Setting $m = n = 5$ the problem was simple yet not trivial and with $m = n$ the word list could be kept small (only 5 letter words). These input are easy to regenerate, they are predictable (24 solutions exist, not counting transposes) and the use of secondary storage can be avoided.

OUTPUT: No duplicate words were allowed in any particular solution whereas transposes on the other hand were recommended since the overall performance would be weakened if during search these should be found and rejected. Also in order not to spend time and space solutions were counted but not stored.

Average performance: Finding all 24 solutions (not counting transposes).

Worst case performance: All 24 solutions use the word AARON and by eliminating this word from the word list reduces the number of solutions to zero. A run with this amputated word list will henceforth reflect a worst case behavior.



```
-----------------------------------------------------------
AARON   ALBAN   AROMA   BREVA   LAYER   NODAL   RABAT   RERUN
ABASE   ALLOW   ARULO   BROMA   MANNY   NONET   RACON   RETAN
ABBAS   ALOSA   AWARD   BROSE   MARAL   NONYL   RAMED   RIATA
ABDAL   ALULA   BACON   BURET   MARKA   NOTAL   RANGE   ROBIN
ABEAM   AMPLY   BADON   CLINE   METAL   NOTER   RASPY   RODGE
ABELE   ANANA   BATON   CLITE   MONAL   OBESE   RATED   ROPER
ABNER   ANELE   BEFIT   CRORE   NAMER   OCHNA   RATER   ROTAL
ABRAM   ANENT   BENIN   DEMIT   NANNY   OMINA   REBUD   RUDAS
ABRIM   ANKLE   BETIS   DENIM   NASAL   ONSET   REBUT   SASAL
ACANA   ARACA   BLORE   EMEND   NATAL   ORAON   RECON   SALAY
ACARA   ARAIN   BOSUN   ENATE   NATHE   ORIEL   REDAN   SANDY
ACATE   ARAMU   BRACE   EURUS   NEEDS   OSELA   REFEL   SAYAL
ACOIN   ARARA   BRAVA   IDIOT   NEELD   OTTAR   REGES   SKEET
ADDIE   ARDEB   BRAVO   INERT   NEESE   OVILE   RENES   SOWEL
AEGLE   AREEK   BRAZE   ITALA   NENTA   OVOID   RENKY   TATES
ALADA   ARENA   BREBA   LADER   NETTY   OVULA   REPEW
ALATA   ARETE   BREME   LASER   NEWEL   OZARK   REROW
-----------------------------------------------------------
```

With a small problem as the one defined, Berghel and Rankin to some extend, avoided to measure whatever memory management and backtracking characteristics which the executable program might inherit from the compiler. □

4.3.2 L. J. Spring et al.

A Proposed Benchmark for Testing Implementations of Crossword Puzzle Algorithms [L.J90]

The benchmark by Berghel et al. had very narrow input constraints; only 5 by 5 puzzle geometries were supported and the word list was small and somewhat fortuitous - made for the occasion without knowing whether it was representative or not. Spring et al. introduced a benchmark in which these constraints were softened a bit. Puzzle geometries with closed cells and totally enumerated lexicons were used.
In the following a node is defined as an attempt to insert a word into a word slot, and, in the resulting tree, each complete leaf will then represent a solution. The benchmark consisted of three parts:

(1) Correctness of the algorithm: the first test was designed so that every leaf in the search space represented a unique solution, i.e. no trimming of the search space can be done. Should the program wrongly perform a trim an incorrectness has been revealed.
(2) Worst case behavior: the second test is developed so that no leaf represents a solution.
(3) Average case behavior: Some leaves represent solutions and some don't.

Run time is measured and the number of nodes are counted in each of these tests. The benchmarks proposed above rely completely on the grid and the word lists given; special designed grids and word lists were therefore necessary. The grids used by Spring et al. are depicted in figure 4.1 below.

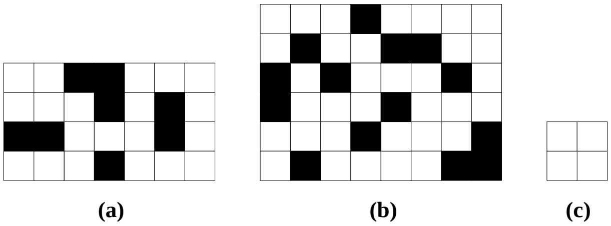


Figure 4.1: *The three grids used in the benchmark.*

Using the symbols $\{a, b\}$ the totally enumerated lexicon consisting of words of length two, three and four was created and used in **(a)**. Adding all four letter words constructed using the set $\{d, e, f\}$ to the word list used above will made solving **(b)** impossible. And finally **(c)** is solved with $k = 2, ..26$, where k is the number of symbols used to create the totally enumerated lexicon consisting of words of length 2. An analytical formula for the 2 by 2 puzzle exists and is given by

$$S(k, n = 2) = (k + 2)k(k - 1)(k - 2)$$

where k is the number of different symbols and n the size of the square grid. In **(a)** and **(b)** duplicate words are allowed in a solution and in **(c)** duplicate words are not (hereby only some of the leaves in the search tree constitute solutions). As in the benchmark by Berghel et al. solutions are counted but not stored. □

4.3.3 Previous benchmarks used on grid walks

The two benchmarks presented above can and will be used on the grid walk approaches - compiler algorithm design differences will, however, necessitate a minor change in the benchmark procedures. Input and output constraints will remain unchanged without any further adaptation, whereas we need to modify the data measures collected during the search. Both Berghel et al. and Spring et al. assumed that the search tree had nodes for every word insertion - this can no longer hold true since we insert single letters. Instead, as we already have done in the previous section, we will consider a letter insertion as a node. A tree leaf at the deepest possible level is, however, still a solution to the grid.

We will first collect data from the two benchmarks and then at the end of the chapter, in a separate section, analyze the data.

4.4 Testing algorithm correctness

> **The opposite of a correct statement is a false statement. But the opposite of a profound truth may well be another profound truth.**
> *Niels Bohr*

In the previous chapter we proved that the grid walk approach was correct; we will now use the two benchmarks to verify this.

4.4.1 The benchmark by Berghel et al.

Berghel et al. [HB90a] used a word list with 134 five letter words and a 5×5 open grid as input and showed that the expected output would be 24 solutions. The 24 solutions were produced so that no word would appear more than once in each puzzle, and so that no solution would appear as a transpose of any earlier solution. This requirements will be relaxed a bit. We will allow both double (triple or more) word entries in a solution and we will not remove transposes. The reasons are: **(1)** it will require extra work by the crossword compiler to recognize transposes, and **(2)** double entries are from our point of view solutions too, and should not be removed.
The total number of solutions is therefore not 24 but 72.

Berghel's 24 solutions (48 counting transposes):

```
aaron   aaron   aaron   aaron   aaron   aaron   aaron   aaron   aaron
crore   cline   clite   clite   blore   brava   brava   bravo   brose
addie   abase   alate   alate   rabat   rebut   bacon   benin   repew
reges   tates   rotal   notal   idiot   itala   acoin   aegle   anele
abele   enate   award   award   manny   metal   sandy   skeet   maral

aaron   aaron   aaron   aaron   aaron   aaron   aaron   aaron   aaron
bravo   bravo   braze   bravo   broma   breme   breba   breva   breva
rebud   rebut   retan   bosun   betis   befit   buret   badon   baton
itala   itala   inert   amply   anana   anent   alosa   arain   arain
metal   metal   marka   sayal   salal   salay   sowel   sandy   sandy

aaron   aaron   aaron   aaron   aaron   aaron
breva   brace   breva   bravo   breva   breva
bacon   nathe   eurus   demit   denim   denim
acoin   emend   alula   anele   ankle   anele
sandy   rudas   monal   lader   layer   laser
```

The 24 regular squares [1]:

```
maral   maral   maral   maral   maral   maral   maral   maral   nasal
anana   anana   arena   arena   arena   arena   acana   acana   alada
rated   ramed   reges   renes   renes   renky   rated   ramed   sandy
anele   anele   anele   ankle   anele   anele   anele   anele   addie
lader   lader   laser   layer   laser   laser   lader   lader   layer

maral   maral   maral   maral   maral   maral   nasal   natal   natal
arena   arara   arara   alula   acara   acara   araca   arara   acara
renky   rated   ramed   rudas   ramed   rated   salay   tates   tates
ankle   arete   arete   alate   arete   arete   acate   arete   arete
layer   lader   lader   laser   lader   lader   layer   laser   laser

natal   natal   arara   arara   abram   abram
acana   anana   recon   racon   brace   brace
tates   tates   acate   acate   rabat   rabat
anele   anele   rotal   rotal   acana   acara
laser   laser   anele   anele   metal   metal
```

Heuristic	#Solutions	Run Time (Sec)
Straight Walk	72	0.41
Snake Walk	72	0.94
Switch Walk	72	0.98
Sik Sak Walk	72	1.15
Slalom Walk	72	1.81

Table 4.1: *Run time results for the five grid walks.*

System	#Solutions	Run Time (Sec)
IBM PC/XT	72	319.6
IBM PC/AT	72	110.1
10 MHz AT Clone	72	64.7
Model 70 PS-2	72	15.6

Table 4.2: *Run times by Berghel et al. on four different machines.*

4.4.2 The benchmark by Spring et al.

The second benchmark, suggested by Spring et al., uses a totally enumerated lexicon with words of length 2 to 4 using the characters $\{a, b\}$ on (a) in figure 4.1. Using this list all leaf nodes will represent a solution and the number is easily predicted: with 20 open cells and 2 symbols the number of solutions is $2^{20} = 1048576$. An extra remark: Spring et al. uses a word by word insertion scheme and need not to worry about non-shared cells (there are 7 non-shared cells). In our work we insert a letter at a time and we do not treat non-shared cells any different than the rest of the the open cell, so in order to find solutions we must add the two words of length 1, $\{a, b\}$, to the word list.

[1] Regular squares and other squares are defined in chapter 6 page 111.

Run times

Heuristic	#Solutions	Run Time (Sec)
Straight Walk	1048576	68.56
Snake Walk	1048576	73.42
Switch Walk	1048576	77.00
SikSak Walk	1048576	69.93
Slalom Walk	1048576	76.63

Table 4.3: *Run time results for the five grid walks.*

System	#Solutions	Run Time (Sec)
XT(P)	1048576	2478.5
XT(C)	1048576	1802.3
AT(P)	1048576	704.14
AT(C)	1048576	551.83
386(P)	1048576	182.80
386(C)	1048576	137.59

Table 4.4: *Spring et al. implementations in* **C** *and* **P***ascal on three different machines.*

Node counts

Each letter insertion will in the following be considered a node in the search tree.

Heuristic	#Nodes
Straight Walk	1572862
Snake Walk	1572862
Switch Walk	1572862
SikSak Walk	1572862
Slalom Walk	1572862
Spring	2444012

Table 4.5: *Node counts for the five grid walks and for Spring et al. (Notice: the count for Spring et al. is for whole word insertions!)*

4.5 Worst case behaviour

Mankind's worst enemy is fear of work.
Author Unknown

In both of the proposed benchmarks a worst case scenario is set up and that being the one search in which no solutions can be found. How well will the program manage? How well will it prune the search tree?

4.5.1 The benchmark by Berghel et al.

In order to test a worst case scenario Berghel et al. removed the word aaron from the word list. This word was part of all of Berghel's 24 solutions. Leaving it out would result in zero solutions and a search through the entire search space would be necessary. If we additionally remove the words `maral`, `nasal`, `natal`, `arara`, and `abram` we eliminate all regular squares, and then we get a word list with no solutions too.

Heuristic	#Solutions	Run Time (Sec)
Straight Walk	0	0.20
Snake Walk	0	0.63
Switch Walk	0	0.60
SikSak Walk	0	0.65
Slalom Walk	0	1.10

Table 4.6: *Grid walk run time results.*

System	#Solutions	Run Time (Sec)
IBM PC/XT	0	250.1
IBM PC/AT	0	83.6
10 MHz AT Clone	0	50.6
Model 70 PS-2	0	12.3

Table 4.7: *Berghel et al. run time results.*

4.5.2 The benchmark by Spring et al.

Spring et al. too developed an input that would result in no solutions.

Grid: The grid depicted in (b) in figure 4.1 was used.

Word list: All enumerated words of length 2 and 3 were constructed from the characters $\{a, b, c\}$ and all enumerated words length 4 were constructed from the characters $\{d, e, f\}$.

This grid and this word list have no solutions.

Heuristic	#Solutions	Run Time (Sec)
Straight Walk	0	0.02
Snake Walk	0	**Bad**
Switch Walk	0	0.02
SikSak Walk	0	83.00
Slalom Walk	0	141.78

Table 4.8: *Grid walk run times (Snake Walk exceeded the time limit of 24 hours).*

System	#Solutions	Run Time (Sec)
XT(P)	0	35.70
XT(C)	0	26.75
AT(P)	0	10.11
AT(C)	0	7.69
386(P)	0	2.52
386(C)	0	1.87

Table 4.9: *Spring et al. run times (using* **C** *and* **P***ascal implementations).*

Node counts

Heuristic	#Nodes
Straight Walk	363
Snake Walk	**Bad**
Switch Walk	363
SikSak Walk	974307
Slalom Walk	1505748
Spring	44721

Table 4.10: *Node counts for the five grid walks and for Spring et al. (Snake Walk exceeded the time limit of 24 hours). Notice: the count for Spring et al. is for whole word insertions!*

4.6 Average case behaviour

> **I feel like a fugitive from the law of averages.**
> *William H. Mauldin*

Only Spring et al. proposed a test for the average performance.

4.6.1 The benchmark by Spring et al.

In the 'average' performance test, Spring et al. designed a test in which some (and therefore not all) leaves in the search tree represented solutions. They used a 2×2 grid and 25 word lists. The word lists were constructed from 2,3,...,26 different characters. Run times and node counts were measured on the various systems they had at their disposal.

We will not perform this test for various reasons:

1. The five grid walks we have designed are identical on a 2×2 grid making comparisons worthless.
2. Even if the grid were altered from a 2×2 grid to a $n \times m$ grid the number of nodes would remain the same for all grid walks since the word lists are totally enumerated.
3. Spring et al. do not allow a word to be used more than once.

What we will do instead is to count nodes using the grid and word list presented by Berghel et al.

Heuristic	#Nodes1	#Nodes2
Straight Walk	9471	4730
Snake Walk	20416	14176
Switch Walk	20294	11851
SikSak Walk	26529	15249
Slalom Walk	32210	19819

The number of nodes visited when using the word list resulting in 72 solutions, **Nodes1**, *and when using the word list resulting in zero solutions,* **Nodes2**.

4.7 Optimization effects

> **"I think there should be something in science called the "reindeer effect." I don't know what it would be, but I think it'd be good to hear someone say, "Gentlemen, what we have here is a terrifying example of the reindeer effect."**
> *Jack Handey*

In the previous chapter we discussed the possibility to optimize the backtracking process by carefully examining the grid walks. We would expect that the optimized backtracks did speed up the search - now we will examine if this is in fact so.

In the following **Run Time 1 + #NodesA** is the optimized version and **Run Time 2 + #NodesB** the naive version.

4.7.1 The Berghel et al. benchmark

Correctness

Heuristic	#Solutions	Run Time 1 (Sec)	Run Time 2 (Sec)	#NodesA	#NodesB
Straight Walk	72	0.41	0.36	9471	9501
Snake Walk	72	0.94	0.86	14176	21721
Switch Walk	72	0.98	0.80	11851	20981
SikSak Walk	72	1.15	1.47	15249	40380
Slalom Walk	72	1.81	1.51	19819	40380

Table 4.11:

Worst case

Heuristic	#Solutions	Run Time 1 (Sec)	Run Time 2 (Sec)	#NodesA	#NodesB
Straight Walk	0	0.20	0.18	4730	4750
Snake Walk	0	0.63	0.54	14176	14317
Switch Walk	0	0.60	0.49	11851	12326
SikSak Walk	0	0.65	0.90	15249	24472
Slalom Walk	0	1.10	0.88	19819	24472

Table 4.12:

4.7.2 The Spring et al. benchmark

Correctness

Heuristic	#Solutions	Run Time 1 (Sec)	Run Time 2 (Sec)	#NodesA	#NodesB
Straight Walk	1048576	68.56	61.78	1572862	1572862
Snake Walk	1048576	73.42	62.31	1572862	1572862
Switch Walk	1048576	77.00	61.66	1572862	1572862
SikSak Walk	1048576	69.93	61.93	1572862	1572862
Slalom Walk	1048576	76.63	61.54	1572862	1572862

Table 4.13:

Worst case

Heuristic	#Solutions	Run Time 1 (Sec)	Run Time 2 (Sec)	#NodesA	#NodesB
Straight Walk	0	0.02	0.01	363	363
Snake Walk	0	**Bad**	**Bad**	**Bad**	**Bad**
Switch Walk	0	0.02	0.01	363	363
SikSak Walk	0	83.00	727.69	974307	21523359
Slalom Walk	0	141.78	**Bad**	1505748	581130732

Table 4.14:

4.8 Reflecting on the results

An idealist is one who, on noticing that roses smell better than a cabbage, concludes that it will also make better soup.
H. L. Mencken (1880-1956)

In this section we will look at the results and comment on them.

4.8.1 Correctness

All five grid walk heuristics passed the test - all produced the expected output given the input defined by the two benchmark protocols.

General comparisons

Run times were considerably better than those found by both Berghel et al. and Spring et al. but this is probably due to the considerably better hardware we have access to:

```
Host Name           : wagner
Host Aliases        : imada.ou.dk loghost
Host Address(es)    : 130.225.128.15
Host ID             : 7140110c
Serial Number       :
Manufacturer        : Sun (Sun Microsystems Incorporated)
System Model        : SPARCsystem 600 (4/6x0)
Main Memory         : 64 MB
Virtual Memory      : 288 MB
ROM Version         : 2.5
Number of CPUs      : 2
CPU Type            : sparc
App Architecture    : sun4
Kernel Architecture : sun4m
OS Name             : SunOS
OS Version          : 4.1.2
Kernel Version      : SunOS Release 4.1.2 (WAGNER) #2: Wed Jan 29 13:23:17 MET 1992
```

Comparisons between their algorithms and ours is therefore **not** possible based only on run time measures. What we **can** do, however, is to compare grid walk run times.

Grid walk comparisons

The two correctness test we have used differ slightly in that Berghel et al. uses a complete puzzle (no closed cells) and only a small word list whereas Spring et al. feed the crossword compiler with a larger input set (and therefore asks for more 'muscles') with closed cells and bigger word lists. The run time results produced may be given the following remarks:

1. Straight Walk is best (fastest) in both tests.

2. Slalom Walk is slowest in the Berghel et al. bench test whereas Switch Walk is slowest in the Spring et al. bench test.

3. The variation in run time differ most in the Berghel et al. bench test whereas in the Spring et al. they are very close.

4. Sik Sak Walk is fourth in the Berghel et al. bench test but advances to second position in the Spring et al. bench text indicating that grids with closed cells should be solved with a diagonal walk heuristic.

Spring et al. also devised a node count scheme and it came as no surprise that all five grid walks resulted in the same number of nodes. It is expected since all leaves in the search tree represents solutions and therefore must be visited. A good property of all the grid walks, however, is the seemingly non-repeating sub-tree searches (see table 4.5).

4.8.2 Worst case

Again all our five grid walk heuristics produced the expected results which also provided us with an interesting (read unexpected) benchmark behavior.

General comparisons

The run times for the Berghel et al. bench test went well and the order of the heuristics are the same as for the Berghel et al. correctness test. If we turn to the other worst case bench test by Spring et al. we see big differences.

Grid walk comparisons

Straight Walk and Switch Walk show excellent run times whereas Sik Sak Walk and Slalom Walk fall far behind not to mention Snake Walk which is lost beyond comparisons.
If we look at node counts we see that the number of nodes for the two winners are identical and very low whereas for the rest the number is high (but not higher than the number of nodes visited in the correctness test).

Why the Snake fail

These results do not look good at all - but they might not be as bad as they seem. The grid used by Spring et al. has three built in fail points: three of the word slots of length four, together with the word list which has no compatible four letter words, create an impossible grid area. When one of these word slots are encountered we must backtrack and preferably a backtrack should happen as high up in the search tree as possible. Both Straight Walk and Switch Walk realizes quickly that there is a problem in the upper right corner of the grid whereas the rest concentrate on filling other grid areas first not realizing they are doing wasted work.

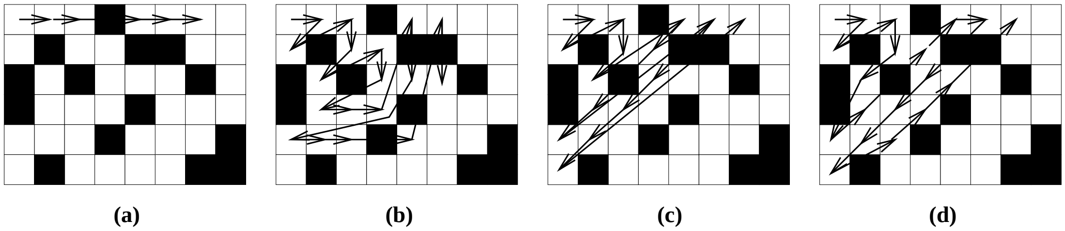


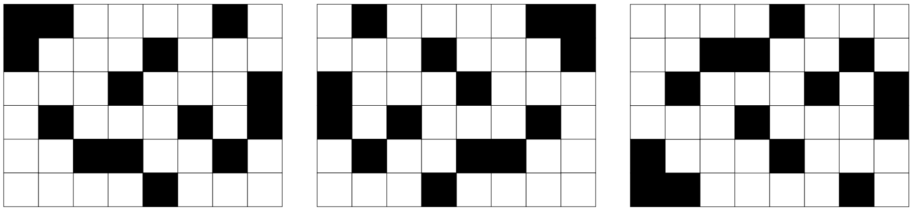
Figure 4.2: *The figure depicts how far in the walk we must go before an error is encountered when using* (**a**) *Straight Walk and Switch Walk,* (**b**) *Snake Walk,* (**c**) *Sik Sak Walk, and* (**d**) *Slalom Walk*

The figure above illustrate the situation. Both Straight Walk and Switch Walk need only to traverse five open cells before the problem occur whereas the other walks require between 15 and 18 visited open cells before trouble appear. The number of combinations tested before termination for the five grid walks is depicted in the following table

Heuristic	#Number
Straight Walk	$3^5 = 243$
Snake Walk	$3^{18} = 3.87 * 10^8$
Switch Walk	$3^5 = 243$
SikSak Walk	$3^{15} = 1.4 * 10^7$
Slalom Walk	$3^{18} = 3.87 * 10^8$

Changing the grid

How the grid design, proposed by Spring et al. to test worst case performance, has been made is not mentioned in their paper - but it did result in very diverse run time performances. If the grid in some way could be said to represent a 'homogeneous' grid geometry (e.g. symmetric) this run time diversity could be used as a comparative measure on the efficiency of the algorithms tested. This, however, is not so since a little alternation of the grid would result in very different run time performances.
In order to show this we will test the algorithms again but this time use three mirrored versions of the original grid.



Grid 1

Grid 2

Grid 3

Figure 4.3: *The three mirrored grids.*

Heuristic	Original	Grid 1	Grid 2	Grid 3
Straight Walk	0.02	0.00	0.00	0.00
Snake Walk	**Bad**	0.00	0.00	0.00
Switch Walk	0.02	0.00	0.00	0.00
SikSak Walk	83.00	0.00	0.00	0.00
Slalom Walk	141.78	0.00	0.01	0.00

Table 4.15: *Run time results for the mirrored versions of the Spring et al. grid. Most of the run times are lower than* 0.005 *which results in* 0.00 *sec. in the table.*

Heuristic	Original	Grid 1	Grid 2	Grid 3
Straight Walk	363	3	3	3
Snake Walk	**Bad**	12	39	3
Switch Walk	363	3	3	3
SikSak Walk	974307	12	12	3
Slalom Walk	1505748	0	39	3

Table 4.16: *Node count results for the mirrored versions of the Spring et al. grid.*

These last results tells us that choosing a grid for benchmarking purposes should be done with great care. When using prefix grid walks it is crucial when the fail points happen: if late in the walk - a lot of work is wasted, if early in the walk - a lot of work is avoided. Prefix grid walks can not be sure to establish a fail point early in the walk whereas a template walk could. Unfortunately, we have not implemented any such walk.

4.8.3 Backtrack optimization

The optimization of the backtrack by securing *connectivity* was added in order to increase performance. The implementations, however, did not exactly support the expected results.

If we look at table 4.11 and table 4.13 we see an *increase* in run times for all grid walks (except for Sik Sak Walk tested on the Berghel et al. grid) instead of an expected decrease in run times. Why it that? In the correctness tests all nodes must be visited and therefore no clever backtracking is needed (we can at most move up one level in the search tree). The extra time used by the optimized grid walk versions must originate from the extra work it takes to examine whether an optimized backtrack is possible or not.
In table 4.12 we see that the increase in run time is even bigger. That is because, we not only examine whether an optimized backtrack is possible, we also use extra time when updating the data structures after an actual backtrack move.

If we define

$b_{overhead}$	Backtrack overhead
$u_{overhead}$	Update overhead
$t_{overhead}$	Total overhead
n_{time}	Naive backtrack time
o_{time}	Optimal backtrack time
d_{time}	Saved time

We then get

1. $d_{time} = n_{time} - o_{time}$

2. $t_{overhead} = b_{overhead} + u_{overhead}$

3. $d_{time} < t_{overhead}$

The inequality in **3)** is an '<' but if we look at table 4.14 (See the Sik Sak Walk) we see that it can also be an '>' . This indicates that the unfortunate backtrack overhead only matters if the search problem is small - when a bigger problem (including closed cells) is chosen we see that optimal backtracking is indeed an advantage.
If we look at the Sik Sak Walk performance in table 4.11 and 4.12 we see that this walk is the only walk that actually do exhibit a better run time **with** optimal backtracking (optimal backtracking can be done in every backtrack session, see table 4.17).

Heuristic	Naive total	Optimal total	Optimal	Percentage
Straight Walk	2206	2194	6	0.2
Snake Walk	5708	5631	24	0.4
Switch Walk	4849	4625	290	6.2
SikSak Walk	7635	3042	3042	100
Slalom Walk	7635	5377	2888	53.7

Table 4.17: *The number of times the backtracking has been necessary when solving Berghel's* 5×5 *grid with the 'no solution word list'. The first column contain the total counts for the naive backtracks, the second column the total number of counts for the optimized backtracks, the third column the number of optimized backtrack moves, and finally the percentage of optimized backtracks.*

The backtrack counts in table 4.17 once again brings Straight Walk up in front with the lowest backtrack count (and only 6 of them being optimal). Slalom Walk and Sik Sak Walk uses optimized backtracking the most but their backtrack count is still very high.
If we look at the percentage of optimal backtrack moves we must conclude that optimal backtracking is not worth it for the first three grid walks but absolutely necessary for the Slalom walk and almost indispensable for the Sik Sak walk.

Chapter 5

Estimation

Why are our days numbered and not, say, lettered?
Woody Allen

We have in the previous chapters discussed and defined algorithms which were capable of constructing crossword puzzles given any word list and any grid. In the Benchmark chapter we saw how run times varied from very fast to very very slow and it became quite evidently that some input configurations would result in millions of solutions while other configurations probably would not result in any solutions at all (no matter how long we searched). In order to save time it will therefore be quite useful to have a method in which we can predict the outcome: **(1)** run time, and **(2)** number of solutions.
Two papers (probably more) exists in which estimates have been developed and tested using Bayesian Estimation. So far only one exact estimate has been produced (it predicts the number of solutions to the 2×2 grid in which no word repetition is allowed). More general estimates, however, suffer from either not being very general at all or to deviate too much from the actual results.
We begin with a discussion of this previous work and then turn to a combinatoric approach.

5.1 Previous work - Estimation

> **If you are distressed by anything external, the pain is not due to the thing itself but to yourown estimate of it; and this you have the power to revoke at any moment.**
> *Marcus Aurelius*

Two papers have been published on the subject both using Baysian Estimates

Theorem 5.1 Baye's theorem

Let $\{B_1, B_2, ..., B_n\}$ be a mutually exclusive and exhaustive set of events, and let A be an observed event. The probability that the event B_j is the causal event giving rise to A is given by

$$P(B_j \mid A) = \frac{P(B_j)P(A \mid B_j)}{\sum P(B_i)P(A \mid B_i)}$$

5.1.1 Chris Long

Mathematics of Square Construction [C.L92]

Ross Eckler (editor of the quarterly journal Word Ways) in the May 1992 issue of Word Ways raised the question of the minimum number of words needed before you'd expect to be able to form a square puzzle (puzzles with no closed cells) and termed this number the *support*. Chris Long proposed an estimate based on Bayes' theorem in which he assumed that the frequencies for each letter in each word were probabilistically independent, and that letter frequencies were positionally independent. The estimate found by Long quite clearly explained why square puzzles larger than 10 × 10 haven't been seen (and probably never will be).

Definition 5.1 (Regular/Double n-square) A regular n-square is a complete puzzle with equal sides. The inserted words are the same across and down. A double n-square is like the regular n-square except all words are different.

C. Long found an estimate for the support for both types of squares and discovered a very nice relation between them. Let $f(*)$ denote the frequency of the letter $*$ in the word list, define $p = f(a) + ... + f(z)$ and let W be the number of n letter words. Applying Bayes' theorem the expected number of fills for a grid F is approximately

$$E(F) = W^{2n}p^{n^2}$$

Long found that $p \sim 1/15,8$ with very little variation and by setting $E(F) = 1$ he found that $W = p^{-n/2}$ so the support $S(F)$ for the grid F is given by

$$S(F) = W = 15,8^{n/2}$$

For the regular n-square we must not include redundancies so

$$E(F) = W^n p^{n(n-1)/2}$$

and therefore

$$S(F) = 15,8^{(n-1)/2}$$

In other words it's just as hard to find a double $(n-1)$-square as it is to find a regular n-square.

n	$S(F_1)$	$S(F_2)$
4	63	250
5	250	992
6	992	3944
7	3944	15678
8	15678	62320
9	62320	247718
10	247718	984658
11	984658	3913938
12	3913938	15557597

Table 5.1: *The support for regular (F_1) and double (F_2) squares of size 4 to 12.*

□

5.1.2 G. H. Harris and J. J. H. Forster

On the Number of Solutions $S(k,n)$ to a Crossword Puzzle
On the Bayesian Estimation and Computation of the Number of Solutions to Crossword Puzzles [G.H90b]

G. H. Harris and J. J. H. Forster went even further. Also using Bayesian estimation they formulated an upper and lower estimate on the number of solutions to a given word list to any grid configuration, not just square puzzles. The estimate was, among other tests, tested on small square puzzles and did not manage as well as Chris' estimate. This was probably due to the more general formulation of the estimate designed to cover all types of puzzle configurations.

Harris and Forster in the paper "On the Number of Solutions $S(k,n)$ to a Crossword Puzzle" formulated and upper and lower estimate for the number of solutions to square puzzles when the words used are extracted from an totally enumerated dictionary.

The proposed estimate for the lower limit for a square of size n:

$$S_{LL}(k,n) = \binom{k^n}{2n}\left(\frac{k^{n-1}-1}{k^n-1}\right)^n \left(\frac{1}{k}\right)^{n^2-n}$$

k is the number of symbols in the alphabet and therefore k^n is the number of available words. $2n$ is the number of used words in the puzzle. So the permutation

$$\binom{k^p}{2n}$$

corresponds to words being used in the solution without replacement.

The $j'th$ vertical word and the $j'th$ horizontal word in the square puzzle, where $j = 1,..,n$, are legitimate words if and only if the $j'th$ letter in both words is the same. This gives rise to the term

$$\left(\frac{k^{n-1}-1}{k^n-1}\right)^n$$

The term

$$\left(\frac{1}{k}\right)^{n^2-n}$$

represents the $n^2 - n$ off-diagonal elements.

Harris and Forster continued this work in the paper "On the Bayesian Estimation and Computation of the Number of Solutions to Crossword Puzzles" [G.H90b] in which they tried to generalize the formula to cover any grid geometry and any dictionary. Define $P_n(a,i)$ as the probability of an n letter word in the dictionary having the $a'th$ letter of the alphabet in the $i'th$ position within the word ($i \leq n$). Let n_i be the number of words of length in the dictionary. The size, N, of the dictionary is then $N = n_1 + n_2 + ...$ Let w_i be the number of word slots in the puzzle of length i. The total number of word slots, W, is then $W = w_1 + w_2 + ...$ Let there be m intersections in the puzzle geometry with the $j'th$ intersection consisting of one word of length n_1 and the other of length n_2, occurring between the i'_1th letter of the first word and the i'_2th letter of the second word. The number of solutions is estimated:

$$\left(\prod_i \binom{n_i}{w_i}\right) \times \left(\prod_{j=1}^{m} \sum_a P_{n_1^{(j)}}(a,i_1) \cdot P_{n_2^{(j)}}(a,i_2)\right)$$

The first term represents the number of permutations

$$\binom{n_i}{w_i}$$

in which the slots of a specific length, i, can be filled from the given dictionary. These permutations multiplied together form the overall measure.

Grid Size	Dict Size	Computed Solutions	Bayesian Estimate	Relative Error
2	100	182	176	-0.03
	200	3372	3211	-0.05
	300	16834	16226	-0.04
	400	53010	51089	-0.04
	500	130004	125072	-0.04
3	100	2	2	0.08
	200	162	168	0.04
	300	742	1243	0.68
	400	7996	8342	0.04
	500	15972	23292	0.46
4	250	0	0	0
	500	2	10	4.05
	750	148	307	1.07
	1000	1244	2497	1.01
	1250	9014	16652	0.85
5	500	0	0	0
	1000	0	0	0

Table 5.2: *A comparison of computed and Bayesian estimates of solutions to totally interlocking puzzles.*

□

5.2 Predicting the results

The best way to predict the future is to invent it.

Alan Kay

We will now present a method capable of predicting the output for a well defined input. The following method is developed in an *ad hoc* manner and .....

5.2.1 The search tree

Our estimate has it's point of origin among the search trees. In chapter 3 section 3.10 we explained the connection between crossword puzzles and search trees and if we recollect what was said then we remember that all even levels in the search tree contain open cell references and all odd levels contain letter supplies.

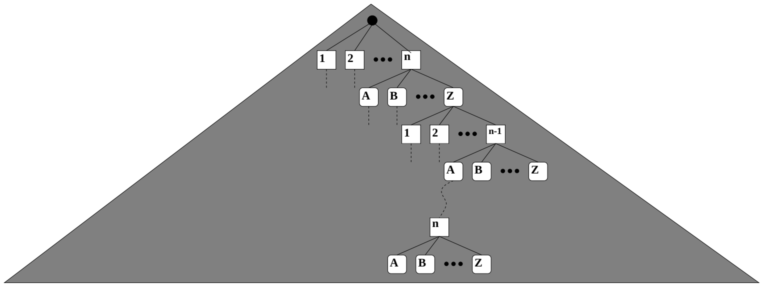


Figure 5.1: *The total search tree.*

Each leave in the search tree at level $2n$ represents a solution to the grid (n is the number of open cells in the grid). So the real problem here is to estimate the number of leaves at this level. When inserting words from a totally enumerated dictionary produced from k characters in a grid with n open cells we can calculate the number of leaves at level $2n$: #solutions is k^{n^2}.

This, of course, will not lead us anywhere if we

1. reject the usage of duplicate words,

2. and use a word lists derived from natural languages.

Is it possible to make an estimate predicting any grid and and word list outcome? We start with squares and see where it may lead us ...

5.2.2 Predicting squares - using enumerate word lists

Harris and Forster have made the only analytical formula to calculate the exact number of solutions to the 2×2 grid using k letters and no duplicate word entries:

$$S(k, n = 2) \quad = \quad (k + 2)k(k - 1)(k - 2)$$

The 2×2 square

By looking at search trees we will now derive this formula:

First look at figure 5.2 and 5.3 below and notice that only $1/k$ of the search tree need to be shown (the other $(k - 1)/k$ sub-trees have the same size and structure).

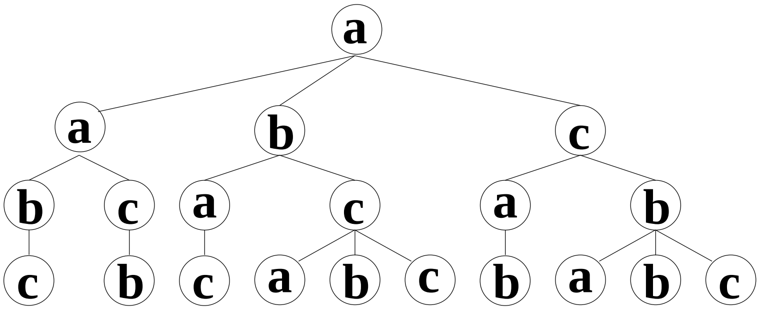


Figure 5.2: *One third of the search tree for the 2×2 grid with $k = 3$ symbols.*

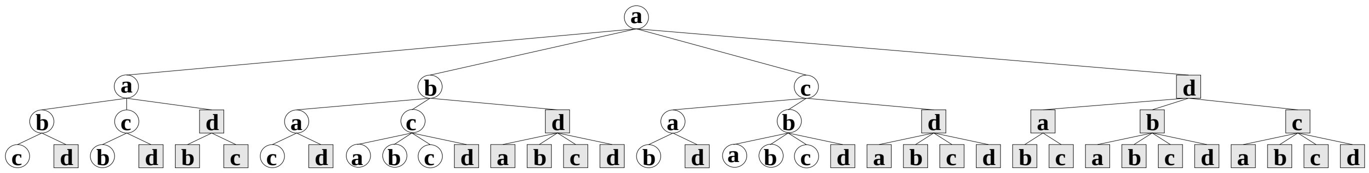


Figure 5.3: *One fourth of the search tree for the 2×2 grid with $k = 4$ symbols. The grey boxes are added when the letter 'd' is introduced.*

Now, look at the nodes at level 3 (the second deepest level) and notice that the number of children is either one of two possible values; in figure 5.2 the two values are 1 and 3, and in figure 5.3 the values are 2 and 4).

This can be generalized: let $s(k)$ denote the first value and let $l(k)$ denote the other (in the above we had: $s(3) = 1$, $s(4) = 2$, $l(3) = 3$, and $l(4) = 4$). Each time we increase k by one we can verify [1] that $s(k+1) = s(k) + 1$ and $l(k+1) = l(k) + 1$, and, finally, end up with $s(k) = k - 2$ and $l(k) = k$.

Now that we know how many children the last parent nodes have we need to find the number of each type of such parents. We will denote these functions, $Small(k)$ and $Large(k)$.
In figure 5.2 there are 4 $s(3)$-sibling nodes and 2 $l(3)$-sibling nodes and in figure 5.3 the two numbers are both 6. If we let $Small(k)$ be the number of nodes with the smallest number of children and if we let $Large(k)$ be the number of nodes with the largest number of children it can be found (by counting) that:

k	$s(k)$	$Small(k)$	$l(k)$	$Large(k)$	#Solutions
3	1	4	3	2	30
4	2	6	4	6	144
5	3	8	5	12	420
6	4	10	6	20	960
7	5	12	7	30	1890
8	6	14	8	42	3360
9	7	16	9	56	5544
10	8	18	10	72	8640
11	9	20	11	90	12870
12	10	22	12	110	18480
13	11	24	13	132	25740
14	12	26	14	156	34944
15	13	28	15	182	46410
16	14	30	16	210	60480
17	15	32	17	340	77520
18	16	34	18	372	97920
19	17	36	19	406	122094
20	18	38	20	442	150480
21	19	40	21	480	183540
22	20	42	22	520	221760
23	21	44	23	562	265650
24	22	46	24	606	315744
25	23	48	25	652	372600
26	24	50	26	700	436800

By examining the numbers in the table above it looks like as if $Small(k)$ and $Large(k)$ are defined this way:

$$\begin{aligned} Small(k) &= 2(k-1) \\ Large(k) &= 2\sum_{1}^{k-2} p \end{aligned}$$

[1] Verification has been done by comparing this with actual data.

The total number of solutions to the 2×2 grid with k symbols is then

$$\begin{aligned} k * [s(k)Small(k) + l(k)Large(k)] &= k[2(k-1)(k-2) + 2k\sum_{1}^{k-2} p] \\ &= k[2(k-1)(k-2) + 2k\frac{1}{2}(k-2)(k-1)] \\ &= k(k+2)(k-2)(k-1) \\ &= S(k, n=2) \end{aligned}$$

Harris and Forster's analytical formula is hereby re-discovered. This, however, taught us more than just how that formula can be derived it gave us a preliminary method which can be generalized and possibly be used to derive analytical formulas for larger squares.

The 3×3 square

We will assume that the formula for the 3×3 square has the same structure as the 2×2 square. We must therefore be looking for functions similar to $s(k)$, $l(k)$, $Small(k)$, and $Large(k)$. Now look at figure 5.4 below:

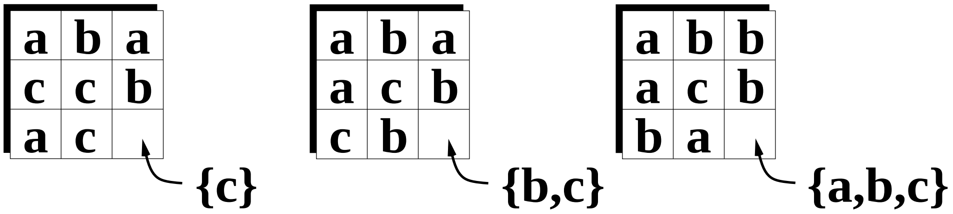


Figure 5.4: *Three 3×3 grid fills in which we have $k = 3$. The first fill only allow one letter (a 'c') to be inserted in the last cell, the second fill has two letters available for the last cell and finally the last fill has three letters for the last cell.*

The mere fact that there exists fills like those depicted in figure 5.4 indicates that there must exist there types of nodes at the last parent level in the search tree: **(1)** nodes having just one child, **(2)** nodes having two children, and **(3)** nodes with three children. Above we had $s(k) = k - 2$ and $l(k) = k$, now we can define three similar functions: $size_1(k) = k - 2$, $size_2(k) = k - 1$, and $size_3(k) = k$. The big question is now how many nodes are there with one child, two children, and three children? We are searching three functions, $num_1(k)$, $num_2(k)$, and $num_3(k)$.

An obvious way to find these numbers is to let our program do the counting but (and there is a 'but') this have been rejected [2] and instead some finger counting and search tree drawings have been done. The following is what we discovered ...

If we draw the search tree (we drew some of it) for the 3×3 grid with $k = 3$ symbols we see that some sub-structures in the search tree are repeated again and again. The total tree consists of 3 identical sub-tree structures and each of these sub-tree structures consist of smaller repeated sub-structures.

[2] This would require that our program did **not** allow duplicate words which it does at the moment. Changing this has been given low priority considering the time we had at disposal.

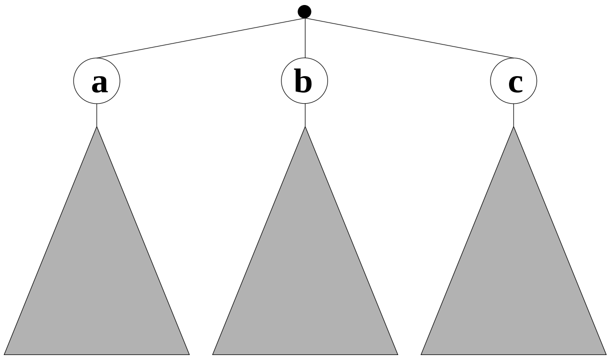
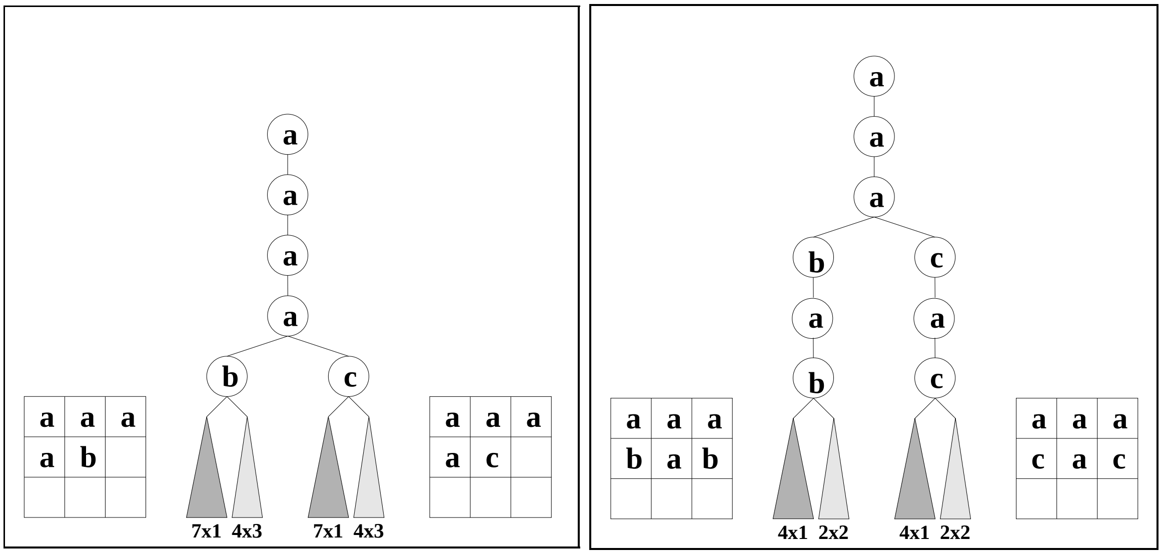


Figure 5.5: *The total search tree consists of three identical sub-tree structures.*



If the total search tree was drawn all sub-structures could be identified and the number of one-child nodes, two-child nodes, and three-child nodes could be counted. In other words we would have found the values $num_1(3)$, $num_2(3)$, and $num_3(3)$.
When raising k from 3 to 4 we would need to examine each sub-structure and identify by how much it would grow and count the extra one-child nodes, two-child nodes, and three-child nodes. When done the values $num_1(4)$, $num_2(4)$, and $num_3(4)$ would be found. Likewise could $num_i(5)$, $num_i(6)$, ... , be found and we would then be able to guess the functions like we did in the 2×2 problem above. We then get

$$S(k, n = 3) \quad = \quad size_1(k)num_1(k) + size_2(k)num_2(k) + size_3(k)num_3(k)$$

The $n \times n$ square

For general squares of size n the formulas will probably look something like this:

$$S(k, n) \quad = \quad \sum_{j=1}^{n} size_j(k)num_j(k)$$

5.2.3 Predicting squares - using natural language word lists

Restricting the word lists to words from natural languages makes it much harder (if not impossible) to predict the precise outcome from a given input. We have tried but **not** found any estimate better than those already proposed by C. Long [C.L92] and Harris and Forster [G.H90b]. We do, however, believe that some fairly good estimates based on the exact analytical formulas from the previous section could be produced if we incorporate letter frequencies and letter distributions into the formulas.

5.2.4 Predicting any grid geometry

Given a grid geometry with, say p, open cells and allowing k different symbols gives rise to k^p enumerated solutions. If we invoke the non-repetition of words the analytical formulas from earlier can not be used. A square puzzle require words of the same lengths in all word slots and this is not the case for any other grid geometry. A formula must therefore take into account the existence of at least two (probably many more) different word lengths. One should also realize that two puzzles with the same distribution of word lengths need not have the same number of solutions. The number of solutions is highly dependent on the distribution of closed cells.

Using the approach used earlier to produce analytical functions can be done for individual puzzle geometries but producing these functions is surely a much bigger job than just letting the computer search for a solution (a grid of size $n \times m$ have 2^{m+n} grid geometries!).

What we will suggest instead is to scan the grid and look for problem areas. A problem area can be an open area in the grid in which many long words cross each other. Estimates for such areas may be possible to produce and an *ad hoc* answer whether a solution to the grid exists or not can then be given.

Chapter 6

Search

Just because you understand what something should look like doesn't mean you know how to build it.

karl@neosoft.com

In this chapter we will define various types of rare crossword puzzle solutions sets. Certain puzzles can be categorized into families and within these families some members seems to be all over the place while others are never or rarely seen (or have never existed - i.e. 'family myths').

6.1 The square puzzle family

> **Always and never are two words you should always remember never to use.**
>
> *Wendell Johnson*

Square puzzles are puzzles containing no closed cells at all i.e. the connectivity is as high as can be. Normally square puzzles are 'squares' (rectangles and squares) but other polygons can be considered as well. Square puzzles are hard to find and it takes more than average skill to be able to produce them by hand. 'The handmade' ones often tend to include words which only a chosen few know the meaning of. Using a computer exclude the need of highly developed skills and furthermore results in squares with 'normal' words (you decide which word the computer should know of).

Definition 6.1 (Square puzzles) An n-square is a complete puzzle with equal sides. Square puzzles can be divided into at least three categories:

1a. A *Regular square* is symmetrical i.e. it uses the same word down and across.

1b. A *Palindromic square* is a regular square in which the first word is a reversal of the last word, the second word is a reversal of the second last word, etc. If the size, n, is odd the center word is palindromic.

2. A *Double square* contains words that are all different.

3. A *Polyglot square* contains words from more than one natural language.

Squares are compared **(1)** by their sizes (squares for $n = 2, 3, .., 9$ have been made/found) and **(2)** by how common the words used in the squares are.

Chris Long [C.L92] estimated the *support*[1] for an n-square and showed that the toughness for the puzzles grow exponentially with n.

Our squares

The following is a collection of squares we have found

Regular squares

a	w
w	e

j	a	m
a	n	a
m	a	d

d	y	n	e
y	l	e	m
n	e	u	m
e	m	m	a

d	u	x	e	s
u	l	e	r	i
x	e	n	o	n
e	r	o	s	e
s	i	n	e	s

d	u	t	i	e	d
u	n	e	d	g	e
t	e	l	l	e	n
i	d	l	e	s	t
e	g	e	s	t	a
d	e	n	t	a	l

d	e	c	a	m	p	s
e	t	e	s	i	a	n
c	e	l	e	s	t	a
a	s	e	p	s	i	s
m	i	s	s	e	n	t
p	a	t	i	n	a	e
s	n	a	s	t	e	s

Figure 6.1: *Regular squares of sizes* $n = 2, 3, 4, 5, 6,$ *and* 7.

Double squares

a	n
t	o

j	u	d
a	n	a
w	i	n

d	y	n	e
o	e	i	l
l	a	c	s
t	r	e	e

d	u	f	f	s
o	d	e	o	n
n	a	c	r	e
e	l	i	t	e
e	s	t	e	r

d	u	t	i	e	d
e	r	i	n	g	o
t	a	l	l	e	t
e	n	l	i	s	t
n	i	e	n	t	e
u	n	r	e	a	d

Figure 6.2: *Double squares of sizes* $n = 2, 3, 4, 5,$ *and* 6

The double 7-square and regular 8-square are according to Chris Long equally hard to find and either of these were within reach of our capability. We made a search on a Silicon Graphic machine using the Straight Walk Heuristic but killed the process 4 days later. A solution was probably to be found since we had 31475 words available (and the support is *only* 15679) but the cpu power was needed for something (more useful?).

We could then ask ourselves if this poor performance is due to a poor algorithm? The answer is partly yes partly no. Our algorithms are designed to solve all types of puzzles (and not only square puzzles) whereas a design aimed directly at the square puzzle problem can make time saving short cuts. Fx if we search for a regular square we should take advantage of the symmetry of the puzzle and just solve half the puzzle. This is something that our designs do not use.
Closer examination of the Straight Walk Heuristic shows us that this walk is much better suited for double square searches. The Straight Walk inserts an across word and keeps this

[1] See chapter 5, page 102

word in the puzzle until all combinations below has been tested. Among these a regular square will pop up once in a while but more frequently a double square will show it's physical appearance. So in other words our regular 8-square search was a *double* 8-square search and therefore comparable to a regular 9-square search! No wonder a solution was not found. Among our heuristics the Switch Walk would probably have been much better.

Other squares

The following is a collection of squares others have found

Regular squares

T	O	B	A	C	C	O
O	V	E	R	A	L	L
B	E	V	E	L	E	D
A	R	E	O	L	A	S
C	A	L	L	A	N	T
C	L	E	A	N	S	E
O	L	D	S	T	E	R

J	A	C	K	L	E	G
A	T	H	L	E	T	E
C	H	E	A	T	E	R
K	L	A	N	I	S	M
L	E	T	I	T	I	A
E	T	E	S	I	A	N
G	E	R	M	A	N	E

C	A	S	S	A	V	A
A	S	H	E	R	I	M
S	H	A	R	R	O	N
S	E	R	V	I	L	E
A	R	R	I	S	E	S
V	I	O	L	E	N	T
A	M	N	E	S	T	Y

Figure 6.3: *Three regular 7-squares found by McIlroy, 1975*

S	A	P	P	E	R
A	D	I	A	T	E
M	A	R	T	H	A
U	N	I	T	E	R
E	C	T	E	N	E
L	E	E	R	E	D

P	R	I	M	A	L
R	E	C	I	P	E
O	T	I	T	I	S
S	E	C	R	E	T
E	N	L	A	C	E
R	E	E	L	E	R

J	I	G	S	A	W
U	N	L	A	C	E
S	T	A	N	C	E
T	E	D	D	E	D
U	N	L	A	D	E
S	T	Y	L	E	R

Figure 6.4: *Three double 6-squares found by McIlroy, 1976*

S	A	T	O	R
A	R	E	P	O
T	E	N	E	T
O	P	E	R	A
R	O	T	A	S

C	A	R	E	S
A	N	E	L	E
R	E	F	E	R
E	L	E	N	A
S	E	R	A	C

Figure 6.5: *Two palindromic 5-squares found by McIlroy, 1976*

T	R	A	T	T	L	E	D
H	E	M	E	R	I	N	E
A	P	O	T	O	M	E	S
M	E	T	A	P	O	R	E
N	A	I	L	I	N	G	S
A	L	O	I	S	I	A	S
T	E	N	T	M	A	T	E
A	S	S	E	S	S	E	D

N	E	C	E	S	S	I	S	M
E	X	I	S	T	E	N	C	E
C	I	R	C	U	M	F	E	R
E	S	C	A	R	P	I	N	G
S	T	U	R	N	I	D	A	E
S	E	M	P	I	T	E	R	N
I	N	F	I	D	E	L	I	C
S	C	E	N	A	R	I	Z	E
M	E	R	G	E	N	C	E	S

Figure 6.6: *A double 8-square by Chris Long and a regular 9-square by Eric Albert.*

D	E	T	A	S	S	E	L	E	D
E	X	E	R	C	I	T	A	T	E
T	E	C	T	O	N	I	C	A	L
A	R	T	H	R	O	L	I	T	E
S	C	O	R	P	I	O	N	I	S
S	I	N	O	I	P	R	O	C	S
E	T	I	L	O	R	H	T	R	A
L	A	C	I	N	O	T	C	E	T
E	T	A	T	I	C	R	E	X	E
D	E	L	E	S	S	A	T	E	D

Figure 6.7: *A palindromic* 10*-square by Chris Long*

f	S	i	n	l	A	M	e	U	E	y	l
S	u	n	T	e	R	a	I	s	V	e	E
i	n	G	A	s	a	n	D	G	e	E	T
n	T	A	i	B	n	o	n	L	E	s	T
l	e	s	B	s	R	E	E	e	n	S	E
A	R	a	n	R	g	i	A	n	N	a	T
M	a	n	o	E	i	n	T	I	t	E	R
e	I	D	n	E	A	T	t	o	E	m	b
U	s	G	L	e	n	I	o	E	r	R	s
E	V	e	E	n	N	t	E	r	R	e	e
y	e	E	s	S	T	E	m	R	e	S	n
L	E	T	T	E	a	R	b	s	e	n	t

Figure 6.8: *An* **Artist** 12*-square (or a regular* 12 *−* *square). Each word entry is a merge of two words - the lower case letters constitute one word and the upper case letters another. Source: The Enigma, The National Puzzlers' League (NPL).*

				z				
			f	i	b			
		t	i	g	e	r		
	f	i	z	z	l	e	d	
z	i	g	z	a	g	g	e	d
	b	e	l	g	i	a	n	
		r	e	g	a	l		
			d	e	n			
				d				

Figure 6.9: *A different kind of square puzzle - a much easier puzzle because words lengths 3 to 9 are used. Source: The Enigma, The National Puzzlers' League (NPL).*

6.2 The cube puzzle family

The impossible is often the untried.

Jim Goodwin

Back in 1963 in the journal *Information and Control* H. Bluhme presented a paper *Three-Dimensional Crossword Puzzles in Hebrew.* We haven't read the paper but it apparently deals with the construction of crosswords in three dimensions. The definitions for the square puzzles in the previous section can with some modifications be adapted to cover cubic puzzles as well.

Definition 6.2 (Cubic puzzles) An n-cube is a complete puzzle in three dimensions with equal sides. Cubic puzzles can be divided into at least three categories:

1a. A *Regular cube* is symmetrical i.e. they use the same word triple.

1b. A *Palindromic cube* is a regular cube in which at each story in the cube the first word is a reversal of the last word, the second word is a reversal of the second last word, etc. If the size, n, is odd the center word is palindromic.

2. A *Triple cube* contains words all different.

3. A *Polyglot cube* contains words from more than one natural language.

We haven't implemented any program to produce cubes but grid walks as we presented them in chapter 3 can be redefined to cover three dimensions. Legal letter insertion would then require verification in three directions instead of only two.

A regular cube

Even though we haven't got a program it does not mean we can't present a regular 3-cube. The following 3-cube was produced by hand.

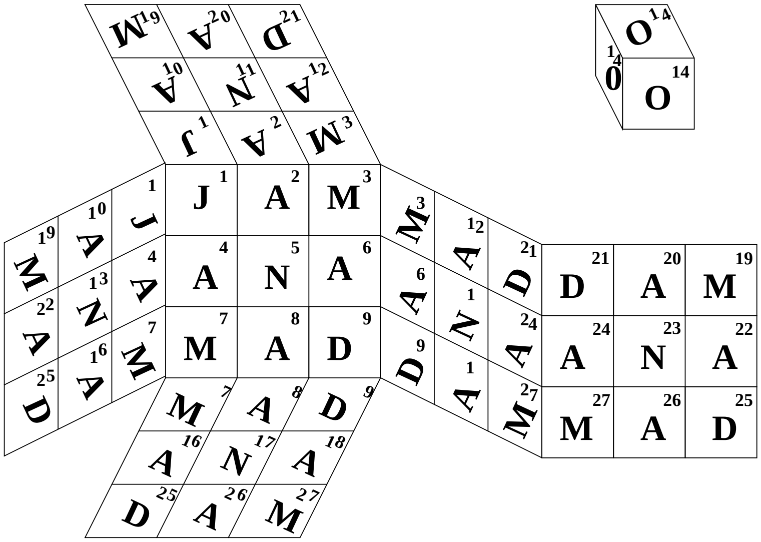


Figure 6.10: *A regular 3-cube.*

6.3 Crozzles

> **The difference between the right word and a similar word is the difference between lightning and a lightning bug.**
> *Mark Twain*

Crozzles are a special type of crossword puzzle and for some reason only found in Australian newspapers.

6.3.1 The problem

Given a word list (usually containing 100-200 words) and a grid configuration (only dimensions provided) insert as many of the words from the word list into the grid so that no word are used more than once and so that all words are connected.

Solutions are ranked using a rating system like for instance:

> *The score for a board is ten points per word formed, plus bonus points for each letter that is a ”cross” piece, i.e. being used by both a horizontal and a vertical word.*

6.3.2 Previous work

In the paper ’*The Crozzle - a Problem for Automation*’ J.J.H. Forster, G.H. Harris, and P.D. Smith propose a method to solve the Crozzle problem and their work is a continuation of G. Harris [G.H90a].

6.3.3 Our work

Solving the Crozzle problem can not be done combinatorically simply because the number of combinations even for very small grid sizes is astronomical. Using any of our grid walk heuristics would fail unless we add a new ’letter’ to the alphabet: ’■’. Here is an algorithm which can solve the Crozzle problem using grid walks.

```
solution ← Crozzle (wordlist)

while (grid not full) do
  cell ← Next(cell)
  if (cell[Supply] not empty) then
    cell[Letter] ← InsertLetter(cell[supply])
  else
    OK ← CheckWord(cell)
    if (OK) then
      cell[Letter] ← ■
    else
       cell ← Backtrack(cell)
```

CheckWord(cell) examine the other word that might be ended and it returns true if the other word is OK.

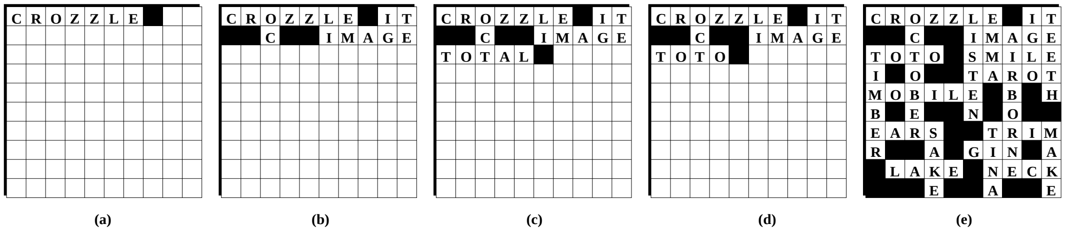


Figure 6.11: *Using the Straight Walk Heuristic the first steps in the search for a solution is shown. Whenever a letter insertion fail a ■ is inserted unless it creates an illegal word in the other direction (this happens in* **(c)** *because the word* `LI` *is illegal.*

This approach do not try to optimize scoring (maximize word connections and number of words inserted).

In the figure above the Straight Walk heuristic was used, but we guess that when it comes to unconstrained puzzles this is not the best heuristic - it will probably be better to work in both dimensions instead of 'straight lines'.

6.4 Number puzzles

There is no safety in numbers, or in anything else.

James Thurber

Instead of word lists we could use number lists. If any number would do no word list would be required but if we restrict the numbers to, say, prime numbers it could be interesting to see how many, say, square puzzles there are.

The first 100.000 primes

A 'word list' containing the first 100.000 prime numbers was found on the Internet [2]. Among these we find **all** primes of length 2 to 6 and in figure 6.12 below is shown examples of squares containing these primes.

4	3
1	7

9	1	1
7	6	9
1	3	1

6	6	1	9
5	5	2	1
4	5	2	3
7	1	9	3

3	6	5	5	9
3	5	7	2	9
4	4	5	7	9
7	4	8	2	7
9	7	7	7	1

8	4	3	4	4	9
8	2	9	2	1	1
2	4	9	1	3	3
8	7	5	6	5	9
8	2	4	9	3	3
1	9	1	7	7	3

Figure 6.12: *Squares of sizes 2 to 6 containing prime numbers.*

[2] At ftp://nptn.org/pub/e.texts/gutenberg/etext93/prime10.txt

n	#primes	#Solutions
2	21	106
3	143	38461
4	1061	**A LOT**
5	8363	**A LOT**
6	39222	**A LOT**

Table 6.1: *The number of prime numbers of length 2 to 6 is shown along with the number of prime square solutions for the squares of sizes 2 and 3. The numbers of squares of higher dimensions are too large to be found within reasonable time limits.*

In table 6.1 above the number of primes and the number of solutions are shown. Only the number of solutions for prime squares of sizes 2 and 3 are found, larger prime squares require much longer run times (several hours) before the entire search space has been exhausted. But the total number of solutions when using primes will be larger than the total number of solutions when using natural language words. This will become more and more evidently when the length of the 'words' increases. This is due to one very distinctive property between primes and natural words: as n grows the number of primes of length n increases whereas the number of natural words of length n decreases.

Definition 6.3 (Dictionary richness)

$$rich \quad = \quad \frac{\#avail\ words}{\#total\ words} * 100\%$$

We see that the dictionary richness for the primes word list is larger than for the natural words word list.

Prime numbers			
n	#total	#avail.	%-rich.
2	100	21	21
3	1000	143	14
4	10000	1061	11
5	100000	8363	8
6	1000000	39222	4

The ukacd dictionary			
n	#total	#avail.	%-rich.
2	26^2	152	22
3	26^3	1209	7
4	26^4	4869	1
5	26^5	10438	0.09
6	26^6	17119	0.006

Table 6.2: *These two tables show the second columns total number of 'words' of length 2 to 6 when using* **(1)** *digits and* **(2)** *the English alphabet. In the third columns the available number of 'words' is shown (in the right table the* **ukacd dictionary** *compiled by Ross Beresford is used). The last columns the dictionary richness is found.*

Chapter 7

Clue Generation

Y'know, every once in a while, when you look off into the distance, you can see a shimmering in the atmosphere's coefficient of refraction, as someone, someone small and slow on the horizon, gets a clue.
Blair Houghton, who's yet to get one

Clue generation is the final step (5) in the complete construction of crossword puzzles.

Well beyond the scope of this project, we would like, anyway, to give an introduction to this area - brief as it should be. Clue generation involve many interesting computer scientific aspects which has been and further should be treated more thoroughly in other projects. So for the sake of completion this will constitute the final chapter in the quest for computer generated crossword puzzles.

Crossword puzzle clues differ a great deal in style and presentation. The standard American (and Danish) style crossword puzzle deal with straight forward clues, a fact or synonym of a crossword whereas an English style cryptic crossword, you deal with a whole set of word play to lead you to an answer.

7.1 Solving Cryptic Crosswords

> **There is no human problem which could not be solved if people would simply do as I advise.**
> *Gore Vidal*

For non-experienced solvers cryptic crosswords look more like a word mess than an artistic word stunt. Fortunately, these weird sentences observe well defined rules and when known by the solver a true contest between clue constructor and clue solver has begun. The following explanations are partly taken from the paper *Cryptic crossword clue interpreter* by M. Hart and R.H. Davis [MH92].

The basic rule

Every cryptic clue has two parts: **(1)** a normal definition answer and **(2)** another way of arriving at the answer. These two parts are put next to each other and part of the fun is to find where one part ends and another starts. Sometimes the two parts are separated by an equality indicator such as `'IS A'`, or `'CONSTITUTE'`.

The definition

The definition part of the clue is either at the beginning of the clue or at the end (never in the middle). A definition can end with a question mark warning the solver to think laterally and not literally.

The word play

The other part of the clue is a hint that involves a word play of which eight different types exist. Each of these have their own recognizable signal leading the solver in the right direction. The signal we are looking for is a keyword or clue pointer indicating the type of word play. Associated with each type is a frame like fx:

□ **a broken** □ *or* □ **inside** □ **is** □

Here are the eight types:

1. A **Double-synonym clue** has no keyword, instead the two parts are synonyms of the same clue.
   An example: **Stone jar.**
   The answer is **rock**.

2. An **Anagram clue** is a re-arrangements of letters. Keywords are *broken, ruined, rebuilt, deranged, disorder, ...*
   An example: **Soil a broken heart**
   Answer: **Soil** is the definition and **earth** the answer (an anagram of **heart**).

3. A **Reversal clue** is a reversal of one or more words in the clue.
   An example: **Wolf goes up a stream**
   Answer: The definition is **a stream**, the key pointer is **goes up**, and the answer is **flow** (**wolf** reversed).

4. An **Enclosure clue** involves a merging of two or more words so that one set of words encloses the other to form a solution, which will be a synonym of the remainder of the sentence.
   An example: **Ford put certain points about publicity.**
   Answer: **about** it the keyword, **certain points** are **w** and **e**, and a synonym for **publicity** is **ad**. So the answer is **wade**!

5. An **Insertion clue** is similar to an enclosure clue except the keyword indicate that one word should be inserted into another (or synonym if this).
   An example: **Writing material for Greek character in fireplace.**
   Answer: the definition is **writing material**, **Greek character** is **phi**, a synonym for **fireplace** is **grate**, and the keyword is **in**. So the answer is **graphite**!

6. A **Hidden-word clue** is a clue in which the answer should be found among possible letter sequences in the sentence.
   An example: **Smack which appears in East Anglian ports.**
   Answer: the answer is **tang** which means **smack** (in the sense of taste) and it is hidden in **easT ANGlian ports**.

7. A **Juxtaposition clue** involves the concatenation of synonyms, words, parts of words, and letters to form the solution.
   An example: **To strike hotel messenger constitutes violent behavior.**
   Answer: **violent behavior** is the definition, **constitutes** is the equality indicator, and synonyms for **to strike** and **hotel messenger** are **ram** and **page** respectively. So the answer is **rampage**!

8. A **Compound clue** contain combinations of the above clue types. If the other clue types were hard compound clues are even harder ...

These clues have been both produced and solved using computers (one may ask what are **we** then supposed to do?).

7.2 Previous work - Clue generation

You cannot depend on your eyes when your imagination is out of focus.
Mark Twain

A computer can easily produce clues to American style crossword puzzles, not so easy, is it to computerize the clue generation of cryptics.

7.2.1 G. W. Smith and J. B. H. Du Boulay

The Generation of Cryptic Crossword Clues [GS86]

Most cryptic clues adhere to a certain structure; a solver must scan the clue for keywords and key phrases associated with the clue in order to solve it. Smith and Boulay reverse the solving process - that is they build a clue around the keywords. Given a word used in a crossword puzzle we progress in the following manner:

1. **Word splitting:** the word is split into two or three sub-words, rearranged and verified (the produced sub-words must be in the dictionary).
2. **Definition gathering**: the sub-words are further identified; is a word a noun, a verb or something else. This is done in order to produce grammatically correct clues.
3. **Keywords**: associated with each of the basic types is a set of frames containing keywords and slots to be filled. The keywords indicate the kind of clue it is to be solved. Among the possible frames one is chosen randomly.
4. **Building the clue**: sub-words are inserted into the vacant slots in the randomly chosen frame.

□

7.2.2 P. D. Smith

XENO: Computer-Assisted Compilation of Crossword Puzzles [Smi83]

A word used in a puzzle can have many different clues associated to it. Smith describes in his paper a method in which these possible clues are maintained and chosen. This method can easily be adapted to work on American style crosswords too. The following is a description of a part of a package called XENO. XENO is an aid to puzzle compilation rather than a crossword puzzle compiler. The method:

Each entry for the dictionary points to a list of clues and these have associated with them three types of information

(i) a difficulty rating

(ii) a classification for the clue

(iii) the clue text

Clues in (ii) are classified into eight different categories: Quotations, Hidden words, Anagrams, Second definitions, Charades, Homophones, Other general knowledge and other cryptics. Note that a clue can be in more than one category. Picking the clues to all word slots are done in six phases:

1. screening and forming lists
2. ordering of table entries
3. computing cumulative totals
4. random list reordering
5. tree searching
6. selecting randomly from the clue lists.

Step (1) to (5) tries to establish a difficulty level and in step (6) the set of clues to be used are selected. □

7.2.3 P. W. Williams and D. Woodhead

Computer Assisted Analysis of Cryptic Crosswords [PW79]

'Computer languages have one characteristic that clearly distinguishes them from natural languages' - computer languages have unique meanings and can be analyzed by a computer program. Williams and Woodhead in their paper 'Computer assisted analysis of cryptic crosswords' [PW79] try to develop a grammar for the English cryptic clues. This is theoretically possible since crossword puzzle clues are a restricted form of English in which 'the syntactic and semantic ambiguities in the language are deliberately exploited to give a variety of possible parsings for a clue'. They name the restricted language LACROSS (LAnguage for CROSSword puzzles). Williams and Woodhead defines the language using Backus Naur form, BNF (some computer languages are defined this way too), and describes the language LACROSS and a computer program which can assist a human solve cryptic crossword puzzles. □

7.3 Cryptic Grammars

Grammar and logic free language from being at the mercy of the tone of voice. Grammar protects us against misunderstanding the sound of an uttered name; logic protects us against what we say have double meaning.

Rosenstock-Huessy

Grammars for entire natural languages is difficult (if not impossible) to define because **(1)** we do not know if a finite number of rules can generate an infinite number of sentences and because **(2)** natural languages has a built in element of chaos. If we, however, restrict ourselves to sub-languages of natural languages grammars may be feasible. Cryptic English used in English cryptic clues is an example of a sub-language. Research on this area have been done by at least two groups

1. *Cryptic crossword clue interpreter* by M. Hart and R.H. Davis [C.L92], and
2. *Does he analyze a tragic stammer? (14): towards a grammar of cryptic English* by Alec McHoul [McH].

7.4 Danish Clues

It's like pissing your pants to keep yourself warm.

Disparaging Danish engineering proverb describing short-term solutions

Danish clues are like the American clues of the straight forward kind, a fact or synonym. There is, however, a sub-group of clues that could have little resemblance with English Cryptics. In some clues there is a question mark at the end indicating (like in the English clues) that we should think creatively instead of rationally. Here is a small list of such clues

Clue	Answer	Source
Omskæring	Boycut	Politikens bagsideredaktion.
Vokalensemble	Ø-gruppe	Krydstillægget i ugebladet Søndag
Tilbageslag	Returkamp	Krydstillægget i ugebladet Søndag
Kammerherre	Rummand	Krydstillægget i ugebladet Søndag
Atletikdyst	Hammerslag	Krydstillægget i ugebladet Søndag
Ungpigelogi	Jomfruhummer	Krydstillægget i ugebladet Søndag
Æbleflæsk	Frugtkød	Krydstillægget i ugebladet Søndag
Kontaktmand	Elinstallatør	Krydstillægget i ugebladet Søndag
Våbenkapløb	Kanonspurt	Krydstillægget i ugebladet Søndag
Anretning for vegetarer	Salatfad	Krydstillægget i ugebladet Søndag
Sameleder	Overlap	Gyden
Graviditetstegn for grise	So-ve	JyllandsPosten Søndag
Mulddyr	Orme	Krydstillægget i ugebladet Søndag
Sparebøsse	Cent-rum	JyllandsPosten Søndag
Onomatopoetikon	Øf	Ukendt

Table 7.1: *Danish clues found in various newspapers and magazines.*

And this completes the chapter on clue generation.

Chapter 8

Conclusion

A conclusion is the place where you got tired of thinking.

Martin H. Fischer

The crossword puzzle has by now been discussed and examined in many a different way and here follows a final discussion on the subject.

8.1 Questions to be answered

> **It is better to know some of the questions than all of the answers.**
>
> *James Thurber*

In the preface we raised a few questions - now is the time to answer.

Q1) *What type of algorithm will perform best on individual problems, and will it also perform as the best algorithm overall?*

A1) If we look at previous work and previous algorithms it is apparent that *declarative approaches* do well on single problems whereas *procedural approaches* are better on an overall basis.
Previous designs are almost all, with very few exceptions, whole word insertion schemes and in our work, we too, have discovered an advantage about this method.

We used a procedural approach that could be adjusted to work on different insertion levels: **(1)** letter insertion, **(2)** substring insertion, and **(3)** whole word insertion and we defined a concept which we named *grid walks* [1]. Six different walks were designed and implemented. The behavior of these varied a lot mainly because of the different fail points and the updating following the backtracks.
Each grid walk defined a search tree which was traversed in a depth first manner. We saw that fail points should happen as high up in the search tree as possible in order to prevent too much wasted work. Certain grid geometries could, however, as we saw in chapter 4 (on benchmarking), cheat a grid walk and make it do much more work than required.
Using *connectivity* in the backtracking process reduced this overhead to a large

[1] Grid walks are orderings and these are briefly mentioned in [ea90].

extent but not well enough. And worse, adding *connectivity* to the backtracking schemes required more work than it saved (at least this was the case on small problems).
We tried to make a walk (*Dynamic Walk*) that would use a *Cheapest-first* approach (fill the most difficult cell next), but we were to tied up by the *prefix condition* and the backtracking required much more extra work than it saved.
The walk that performed best was the simplest of the six defined: *Straight Walk* which is also a 'whole word insertion' method. Fail points seems to be discovered early and when backtracking is needed it is very simple to backtrack to a connected cell (i.e no connectivity overhead).

All in all we believe that a whole word insertion scheme is the best method and words should only be inserted in one dimension (only vertically or horizontally, not both). If the *prefix condition* can be removed a *Cheapest-first* scheme should be added and in order to perform well on most grids a dynamic element must be added too, enabling the walk to change during run time and thereby restore filled areas not directly connected to the problem (possibly separated by closed cells). This will tune it to work well on most general grids but not, however, very well on *square puzzles.* These types of puzzles require specially designed walks which can take advantage of the symmetry found in *regular n-squares.*

Q2) *Given a crossword generating program can we prove that all possible solutions will be found?*

A2) If the given program searches the search space systematically and we can prove that all possible combinations will be tested the answer is yes. In previous work we have not seen any strict prove that the designed programs did in fact find all solutions, merely they were content with the fact that solutions were found at all. We have, on the other hand, proved that our programs find all solutions for any given input.

Q3) *Given a word list and a puzzle geometry how many solutions do there exist?*

A3) For small problems we can let a (correct) program count the solutions but for larger problems it is desirable to calculate an estimate instead.
One analytical formula exists which can predict the precise answer to the question: $S(k, n = 2)$ gives us the exact number of solutions to a 2×2 grid using k symbols. We believe that similar analytical formulas exist for larger values of n too and we have devised a possible method to find these. We did not, however, succeed.
Estimates to general puzzle structures exist but none are very good and we doubt that it is possible to make a general estimate which is capable of predicting any puzzle geometry. We do believe that it is possible to design good estimates to classes of grid geometries.

Q4) *Is it possible to design two word lists W_A and W_B, so that W_A is half the size of W_B, but given an arbitrary puzzle geometry both word lists will result in the same number of solution?*

A4) If we use artificial word lists it may be possible but with words extracted from a natural language it may not be possible. In the question there is hidden a much more interesting question: 'How small is the *best* word list?'. It is obvious that a large word list may be better than a small one but could the opposite happen? A small word list is better than a large one?

This we can not answer but we would like to emphasize the following observation. Say, we want X words in our word list and we compile two such lists. The first list contain prime numbers and the other English words. Which list will produce the largest number of solutions? The first will, because the first list contain 'words' made from only 10 symbols whereas the second word list uses 26 symbols. Since we have X words in both lists the dictionary richness is highest in the first list.

The lesson to be taught is this: the higher the dictionary richness the more solutions we will find.

Q5) *Is it possible to adjust the level of a puzzle? Is is possible to make themed puzzles?*

A5) This is very much a question on the words contained in the word list. Word lists has been made in which words are ranked but since words have different meanings to different people these rankings can not be given unambiguously. We do believe, however, that level control is more tightly bound to clue generation. You can in most cases phrase the clue in such a way that more or less information is given to the solver. It is a well known fact that difficult parts of a puzzle may contain 'give-aways' - that is an easy clue which will give the solver a kick and help him get on with the rest of the puzzle.

Q6) *Is it possible to define a (simple) metric that unambiguously measures how hard a given puzzle geometry is to fill in?*

A6) *Density* and *degree of interlocking* is two metrics used but they are not unambiguous. That is there exist puzzles with high density and high degree of interlocking but they are still, however, easy to fill in.

In chapter 3 (about construction) we defined *fill dependency* (page 55). If we when given a grid calculate for each open cell the number of cells on which it fill dependent and sum these numbers we have a measure on how hard the puzzle is to be filled. This measure can be given unambiguous.

Q7) *Is it possible to generate inconsistent puzzles using human like style and with sensible clues not dull but with human wit and genius?*

A7) **NO!**. Computers can be taught to produce good 'mainstream' puzzles not quality puzzles like those made by professional human compilers. Computers can on the other hand be an invaluable tool (like computers are in so many other areas) to a human compiler. First of all a grid editor and associated tools can help the constructor to keep the construction tidy and easy to handle (no more paper, pencil, and eraser). Secondly, a crossword compiler can be used to suggest solutions to tough grid areas and the human compiler can then choose among given suggestions. Thirdly, when constructing the clues various electronic dictionaries, encyclopedias, names' lists, etc. can be a good tool to verify and/or create ideas.

These concepts can take out the dull mechanical parts of crossword compilation but still we need the human brain to lead the way.

Q8) *Do there exist other problems but the crossword puzzle problem which can be solved with a crossword compiler? And, will a crossword compiler perform better than previous solutions to such problems?*

A8) Any problem that require elements to be puzzled together in 2 or 3 dimensions can be solved using a crossword compiler. We have from the very beginning of this project tried to think of such problems but have not, however, found any obvious case so the second part of the question is a bit hard to answer.
We do still believe that alternative 'crossword puzzle' problems exist and that it is only a question of translation.

Bibliography

[Ber87] H. Berghel. Crossword compilation with horn clauses. *The Computer Journal*, 30(2):183–188, 1987.

[C.L92] C.Long. Mathematics of square constructions. *Word Ways*, 26(1), 1992.

[ea90] Matt L. Ginsberg et al. Search lessons learned from crossword puzzles. Proceedings of AAAI 90, 1990. http://medg.lcs.mit/mpf/papers/Ginsberg/Ginsberg-et-al-90.html.

[G.H90a] G.Harris. Generation of solution sets for unconstrained crossword puzzles. Symposium on Applied Computry. TH037-9/90/0000/0214 Copyright 1990 IEEE, April 5-6 1990.

[G.H90b] J.J.H.Forster G.H.Harris. On the bayasian estimation and computation of the number of solutions to crossword puzzles. Symposium on Applied Computry. TH0307-9/90/0000/0220 Copyright 1990 IEEE, April 5-6 1990.

[G.H90c] P.D.Smith H.Berghel G.H.Harris, D.Roach. Dynamic crossword slot table implementation. Symposium on Applied Computry. Copyright 1992 ACM 0-89791-502-X/92/0095, 1990.

[GS86] J.B.H. Du Boulay G.W. Smith. The generation of cryptic crossword clues. *The Computer Journal*, 29(3):282–283, 1986.

[HB89] C. Yi H. Berghel. Crossword compiler-compilation. *The Computer Journal*, 32(3):276–280, 1989.

[HB90a] R. Rankin H. Berghel. A proposed standard for measuring crossword compilation efficiency. *The Computer Journal*, 33(2):181–184, 1990. rrankin@mcs213k.cs.umr.edu.

[H.B90b] J.Talburt H.Berghel, D.Roach. Approximate string matching and the automation of word games. Symposium on Applied Computry. TH0307-9/90/0000/0209 Copyright 1990 IEEE, April 5-6 1990.

[L.J90] G.H.Harris J.J.H.Forster L.J.Spring, H.Berghel. A proposed benchmark for testing implementations of crossword puzzle algorithms. volume VolI of *Symposium on Applied Computry.* Copyright 1991 ACM 0-89791-502-x/92/0002/0099, 1990.

[Maz76a] L.J. Mazlack. Computer construction of crossword puzzles using precedence relationships. *Artificial Intelligence*, 7(1):1–19, 1976.

[Maz76b] L.J. Mazlack. Machine selection of elements in crossword puzzles. *SIAM Journal of Computing*, 5(1):51–72, March 1976.

[McH] Alec McHoul. Does he analyse a tragic stammer? (14): towards a grammar of cryptic english.

[MH92] R.H. Davis M. Hart. Cryptic crossword clue interpreter. *Information and Software Technology*, 34(1):16–27, January 1992.

[O.F75] O.Feger. A program for the construction of crossword puzzles. *Data information*, 17(5):189–195, 1975. Angewandte Informatik.

[PW79] D. Woodhead P.W. Williams. Computer assisted analysis of cryptic crosswords. *The Computer Journal*, 22(1):67–70, 1979.

[Smi83] P.D. Smith. Xeno: Computer-assisted compilation of crossword puzzl. *The Computer Journal*, 26(4):296–302, 1983.

[SS81] P.D. Smith and S.Y. Steen. A prototype crossword compiler. *The Computer Journal*, 24(2):107–111, 1981.

[Wil89] J.M. Wilson. Crossword compilation using integer programming. *The Computer Journal*, 32(3):273–275, 1989.

Appendix A

On-line dictionaries

There are quite a few locations where complete on-line dictionaries are to be found on the Internet. Many thanks must go to Ross Beresford for the following list.

```
        File name(s)   : ukacd11.zip,teadac11.zip
        File size(s)   : 543330,530953
        Site(s)        : gatekeeper.dec.com (crossword archive)
        Directory      : /pub/micro/msdos/misc/crossword-archive
        Origin         : UK Advanced Cryptics Dictionary
        Entries        : 185582
        Inflected Forms: yes
        Phrases        : yes
        Mixed case     : yes
        Comments       : Wordlist specifically for crosswords maintained
                         by Ross Beresford (ross@bryson.demon.co.uk).
                         ukacd11.zip is in plain ASCII; teadac11.zip is
                         in TEA format (see The Electronic Alveary).
\
        File name(s)   : web2.Z
        File size(s)   : 1038775
        Site(s)        : many sites (for example wuarchive.wustl.edu)
        Directory      : (for example /mirrors4/4.3bsd-reno/share/dict)
        Origin         : Websters 2nd Edition words (cf web2a.Z)
        Entries        : 234932
        Inflected Forms: no
        Phrases        : no
        Mixed case     : yes
        Comments       :

        File name(s)   : web2a.Z
        File size(s)   : 434291
        Site(s)        : many sites (for example wuarchive.wustl.edu)
        Directory      : (for example /mirrors4/4.3bsd-reno/share/dict)
        Origin         : Websters 2nd Edition phrases (cf web2.Z)
        Entries        : 76205
        Inflected Forms: no
        Phrases        : yes
        Mixed case     : yes
        Comments       :

        File name(s)   : OSPD.shar.Z
        File size(s)   : 472885
        Site(s)        : ftp.cs.cornell.edu
        directory      : /pub/turney
        Origin         : U.S. Official Scrabble Player's Dictionary
        Entries        : 113901
        Inflected Forms: yes
        Phrases        : no
        Mixed case     : no
        Comments       :
```

```
File name(s)   : mrc2.dct
File size(s)   : 11179399
Site(s)        : black.ox.ac.uk
Directory      : /ota/dicts/1054
Origin         : Shorter Oxford English Dictionary
Entries        : 119888
Inflected Forms: yes
Phrases        : no
Mixed case     : yes
Comments       : Maintained by the Oxford Text Archive.

File name(s)   : words[1234].zip
File size(s)   : 95306,74597,99024,84500
Site(s)        : wuarchive.wustl.edu
Directory      : /mirrors/msdos/linguistics
Origin         : Uncertain (see read.me file)
Entries        : 109582
Inflected Forms: yes
Phrases        : no
Mixed case     : no
Comments       : This list has also been seen split into zip
                 files as evanwrd[1234].zip

File name(s)   : words.english.Z
File size(s)   : 288385
Site(s)        : sparta.nmsu.edu,haywire.nmsu.edu
Directory      : /pub/lexicals/word-lists
Origin         : Unknown
Entries        : 69964
Inflected Forms: yes
Phrases        : no
Mixed case     : yes
Comments       :

File name(s)   : Unabr.dict.Z
File size(s)   : 951951
Site(s)        : arthur.cs.purdue.edu,
                 ftp.denet.dk
Directory      : /pub/pcert/dict/misc/black.ox.ac.uk,
                 /pub/wordlists/dictionaries
Origin         : Unknown
Entries        : 213557
Inflected Forms: no
Phrases        : no
Mixed case     : no
Comments       :

File name(s)   : unabrd.dic.Z
File size(s)   : 1041512
Site(s)        : world.std.com
Directory      : /obi/WordLists/English
Origin         : Unknown
Entries        : 235544
Inflected Forms: no
Phrases        : no
Mixed case     : yes
Comments       :

File name(s)   : pocket.dic.Z
File size(s)   : 85821
Site(s)        : ftp.uu.net
Directory      : /doc/literary/obi/WordLists/English
Origin         : Unknown
Entries        : 21111
Inflected Forms: no
Phrases        : no
Mixed case     : no
Comments       :

File name(s)   : w130794.Z
File size(s)   : 522533
```

```
Site(s)        : ftp.uu.net
Directory      : /doc/literary/obi/WordLists/English
Origin         : Unknown
Entries        : 130794
Inflected Forms: yes
Phrases        : no
Mixed case     : no
Comments       :

File name(s)   : ispell-3.0.09.tar.z
File size(s)   : 467745
Site(s)        : prep.ai.mit.edu
Directory      : /pub/gnu
Origin         : Uncertain (see README files)
Entries        : ca. 50000
Inflected Forms: yes
Phrases        : no
Mixed case     : yes
Comments       : This is the GNU ispell package which could undergo
                 quite frequent releases. Hence the file name and
                 size could change.

File name(s)   : roget13a.zip
File size(s)   : 643011
Site(s)        : mrcnext.cso.uiuc.edu
Directory      : /gutenberg/etext91
Origin         : Roget's Thesaurus, 1911
Entries        :
Inflected Forms: yes
Phrases        : yes
Mixed case     : yes
Comments       : Since this edition is out of copyright, it appears
                 in several different forms on the net. The one
                 above is maintained by Project Gutenberg.

File name(s)   : dictionaries.tar.Z
File size(s)   : 485521
Site(s)        : guardian.cs.psu.edu
Directory      : /pub
Origin         : Unknown
Entries        : 53091
Inflected Forms: no
Phrases        : no
Mixed case     : yes
Comments       : A collection of specialised word lists, primarily
                 intended for password screening.
```

If you cannot find the file in the directory specified, consult your local archie to find an FTP site which does have the file in question.

There are some commercial outfits willing to sell you large word/phrase lists. We advise you to think very carefully before deciding to buy any such lists as the files above will suffice in most circumstances, if you are looking for single word combinations only.

Appendix B

The Prefix Dictionary Code

An ounce of prevention is worth a ton of code.

an anonymous programmer

B.1 Dict.h

```
/* This file contain procedures used to build up the dictionary. */

/* Checks if this file have been compiled before. */
#ifndef DICT

#include <stdio.h>
#include <stdlib.h>

/* Maximum length of a word in the dictonary. */
#define MAXLENGTH 40
/* Boolean variables. */
#define WORDIN 1
#define WORDOUT 0

/* Create a 'tegn'. */

  typedef struct tegn {
    char letter;
    int last;
    struct tegn *next;
    struct tegn *prev;
    struct tegn *down;
    struct tegn *up;
  } tegn;


/* Define a pointer to a pointer to tegn. */

  typedef tegn *Ptegn;


/* Create a 'bogstav'. */

  typedef struct bogstav {
    char letter;
    struct bogstav *next;
    struct bogstav *prev;
    struct bogstav *old;
  } bogstav;


/* Define a pointer to a pointer to bogstav. */

  typedef bogstav *Pbogstav;


/* Frees memory space previously occupied by a 'bogstav'. */

  void bfree(bogstav *head);


/* Create and initialize a 'tegn'. */

  struct tegn *maketegn(void);


/* Create and initialize a 'bogstav'. */

  struct bogstav *makebogstav(void);

/* Given a wordlist makedict create the dictionary. */

  Ptegn *makedict(char Dictname[]);

#define DICT
#endif
```

B.2 Dict.c

```
/* This file contain procedures used to build up the dictionary. */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "Dict.h"
#include "Cross.h"

/* talloc allocate memory to a 'tegn'. tegn contain one letter.      */
/* It is used to build the radix tree in which the words are kept. */

struct tegn *talloc(void) {
  return (struct tegn *) malloc(sizeof(struct tegn));
}

/* Create and initialize a 'tegn'. */

struct tegn *maketegn(void) {
  tegn *new;

  new = talloc();
  new→letter = '0';
  new→next = NULL;
  new→prev = NULL;
  new→down = NULL;
  new→up = NULL;
  return new;
}

/* inserttegn inserts a 'tegn'. */

void inserttegn(tegn *before, tegn *new, tegn *parent) {
  if (before→next ≠ NULL) {
    before→next→prev = new;
  }
  new→next = before→next;
  before→next = new;
  new→prev = before;
  if (parent ≠ NULL) {
    new→up = parent;
  }
}

/* Given a word and a radix search tree putword places the */
/* word at the position in the radix search tree.           */

tegn *putword(tegn *tag, char word[]) {
  tegn *start, *temp, *prev, *new, *parent;
  int state, bogstav;
  char c, compare;
  tegn *retur;

  prev = temp = start = tag;
  retur = tag;
  bogstav = 0;
  while (word[bogstav] ≠ '\0') {
    c = word[bogstav];
    if (temp→letter == '0') {
      temp→letter = c;
      temp→down = maketegn();
      prev = temp;
      temp = temp→down;
      temp→up = prev;
    }
    else {
      state = WORDIN;
      parent = temp→up;
      while ((c > temp→letter) && state) {
        if (temp→next == NULL) {
          state = WORDOUT;
        }
        else {
          prev = temp;
          temp = temp→next;
        }
      }
      compare = temp→letter;
      if (c ≠ compare) {
        new = maketegn();
        new→letter = c;
        if (c < compare) {
          if (temp == start) {
            new→next = start;
            start = new;
            if (bogstav == 0) {
              retur = new;
            }
          }
          else {
            if (prev→down == temp) {
              new→next = temp;
              prev→down = new;
            }
            else {
              inserttegn(prev,new,parent);
            }
          }
        }
        else {
          inserttegn(temp,new,parent);
        }
        new→down = maketegn();
        new→down→up = new;
        temp = new;
      }
      prev = temp;
      temp = temp→down;
    }
    bogstav++;
  }
  return retur;
}

/* Given a wordlist makedict create the dictionary. */

Ptegn *makedict(char Dictname[]) {
  FILE *inputfile;
  char c;
  int i, ch;
  Ptegn *wordsize;
  char ord[MAXLENGTH+1];

  inputfile = fopen(Dictname,"r");
  wordsize = (Ptegn *) malloc(sizeof(Ptegn)*(MAXLENGTH+1));
  for (i=1;i≤MAXLENGTH;i++) {
    wordsize[i] = maketegn();
  }
  i = 0;
  while ((c = getc(inputfile)) ≠ EOF) {
    ch = c;
    if ((ch > 96 && c < 123) || c == -27 || c == -8 || c == -26 ||
(ch > 47 && ch < 58)) {
      if (i>30) {
        printf("\nEt ord langere end 30 tegn ignoreret.");
        i = 0;
      }
      else {
        ord[i] = c;
      }
      i++;
    }
    else {
      ord[i] = '\0';
      wordsize[i] = putword(wordsize[i],ord);
      i = 0;
    }
  }
  return wordsize;
}
```

Appendix C

The Template Dictionary Code

> **In the face of entropy and nothingness, you have to kind of pretend it's not there if you want to keep writing good code.**
>
> *Karl Lehenbauer*

C.1 Dict.h

```
/* This file contain procedures used to build up the dictionary. */

#ifndef DICT
/* Checks if this file have been compiled before. */

/* INCLUDE LIBRARIES */

#include <stdio.h>
#include <stdlib.h>

/* CONSTANTS */

#define MAXLENGTH 30
/* Maximum length of a word in the dictonary. */
#define WORDIN 1
/* Boolean variables. */
#define WORDOUT 0

/* Create a 'tegn'. */

  typedef struct tegn {
    char letter;
    int last;
    struct tegn *next;
    struct tegn *prev;
    struct tegn *down;
    struct tegn *up;
  } tegn;

/* Create a word_ref container */

  typedef struct word_ref {
    struct word_ref *next;
    struct word_ref *prev;
    tegn *word;
  } word_ref;

/* Define a pointer to a pointer to tegn. */

  typedef tegn *Ptegn;

/* Create and initialize a 'tegn'. */

  struct tegn *maketegn(void);

/* Create and initialize a 'bogstav'. */

  struct bogstav *makebogstav(void);

#define WORD_LENGTH 30
#define LENGTH 30
#define SYMBOLS 30

word_ref *meta_dict[WORD_LENGTH][LENGTH][SYMBOLS];

typedef word_ref *Pword_ref;

word_ref *combine(word_ref *old, word_ref *new);

/* Given a wordlist makedict create the dictionary. */

  tegn *makedict(char Dictname[]);

#define DICT
#endif
```

C.2 Dict.c

```
/* This file contain procedures used to build up the dictionary. */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "Dict.h"

int NUM = 0;

void printNUM() {

  printf("\nAntal knuder:  %i", NUM);
}


/* talloc allocate memory to a 'tegn'. tegn contain one letter. */
/* It is used to build the radix tree in which the words are kept. */

struct tegn *talloc(void) {
  return (struct tegn *) malloc(sizeof(struct tegn));
}

/* walloc allocate memory to a 'word_ref'. */

struct word_ref *walloc(void) {
  return (struct word_ref *) malloc(sizeof(struct word_ref));
}

/* Create and initialize a 'tegn'. */

struct tegn *maketegn(void) {
  tegn *new;

NUM++;

  new = talloc();
  new→letter = '0';
  new→next = NULL;
  new→prev = NULL;
  new→down = NULL;
  new→up = NULL;
  return new;
}

/* Create and initialize a 'word_ref'. */

struct word_ref *makeword_ref(void) {
  word_ref *new;

  new = walloc();
  new→next = NULL;
  new→prev = NULL;
  new→word = NULL;
  return new;
}

/* num returns the position of a given letter in the alphabet. */
```

```
int num(char c) {

  int res;

  if (c > 96 && c < 123) {
    res = c - 97;
  }
  else if (c == -27) {
    res = 26;
  }
  else if (c == -8) {
    res = 27;
  }
  else if (c == -26) {
    res = 28;
  }
  else {
    printf("\nnum:  Illegal char value");
  }
  return res;
}


/* inserttegn inserts a 'tegn'. */

void inserttegn(tegn *before, tegn *new, tegn *parent) {
  if (before→next ≠ NULL) {
    before→next→prev = new;
  }
  new→next = before→next;
  before→next = new;
  new→prev = before;
  if (parent ≠ NULL) {
    new→up = parent;
  }
}

/* Given a word and a radix search tree putword places */
/* the word at the position in the radix search tree. */

tegn *putword(tegn *tag, char word[], word_ref *ref) {
  tegn *start, *temp, *prev, *new, *parent;
  int state, bogstav;
  char c, compare;
  tegn *retur;

  prev = temp = start = tag;
  retur = tag;
  bogstav = 0;
  while (word[bogstav] ≠ '\0') {
    c = word[bogstav];
    if (temp→letter == '0') {
      temp→letter = c;
      temp→down = maketegn();
      prev = temp;
      temp = temp→down;
      temp→up = prev;
    }
    else {
      state = WORDIN;
      parent = temp→up;
      while ((c > temp→letter) && state) {
        if (temp→next == NULL) {
          state = WORDOUT;
        }
        else {
          prev = temp;
          temp = temp→next;
        }
      }
      compare = temp→letter;
      if (c ≠ compare) {
        new = maketegn();
        new→letter = c;
        if (c < compare) {
          if (temp == start) {
            new→next = start;
            start = new;
            if (bogstav == 0) {
              retur = new;
            }
          }
          else {
            if (prev→down == temp) {
              new→next = temp;
              prev→down = new;
              new→up = parent;
            }
            else {
              inserttegn(prev,new,parent);
            }
          }

        }
        else
          {
            inserttegn(temp,new,parent);
          }
        new→down = maketegn();
        new→down→up = new;
        temp = new;
      }
      prev = temp;
      temp = temp→down;
    }
    bogstav++;
  }
  ref→word = temp;
  return retur;
}

word_ref *copy_ref(word_ref *org) {

  word_ref *copy;

  copy = makeword_ref();
  copy→word = org→word;
  return copy;
}

word_ref *insert_ref(word_ref *entry, word_ref *here) {

  if (here == NULL) {
    here = entry;
  }
  else {
    here→prev = entry;
    entry→next = here;
    here = entry;
  }

  return here;
}

word_ref *delete_ref(word_ref *bad, word_ref *old) {

  if (bad→prev == NULL) {
    old = bad→next;
    if (old ≠ NULL) {
      old→prev = NULL;
    }
  }
  else {
    bad→prev→next = bad→next;
    if (bad→next ≠ NULL) {
      bad→next→prev = bad→prev;
    }
  }
  free(bad);

  return old;
}

int exist_ref(word_ref *old, word_ref *new) {

  int res;
  word_ref *tmp;

  res = 0;
  tmp = new;

  while (tmp ≠ NULL) {
    if (old→word == tmp→word) {
      res = 1;
    }
    tmp = tmp→next;
  }
  return res;
}

word_ref *combine(word_ref *old, word_ref *new) {

  word_ref *tmp, *tag;

  if (new ≠ NULL) {
  if (old == NULL) {
    tmp = new;
    while (tmp→next ≠ NULL) {
      old = insert_ref(copy_ref(tmp), old);
      tmp = tmp→next;
    }
    old = insert_ref(copy_ref(tmp), old);
  }
  else {
    tmp = old;
    while (tmp ≠ NULL) {
      if (!exist_ref(tmp, new)) {
        tag = tmp;
        tmp = tmp→next;
        old = delete_ref(tag, old);
      }
      else {
        tmp = tmp→next;
      }
    }
  }
  }
  return old;
}

/* Given a wordlist makedict create the dictionary. */

tegn *makedict(char Dictname[]) {
  FILE *inputfile;
  char c;
  int i, j, ch;
  tegn *wordsize;
  char ord[MAXLENGTH+1];
  word_ref *ref, *entry, *here;

  inputfile = fopen(Dictname,"r");
  wordsize = malloc(sizeof(tegn));
  for (i=1;i≤MAXLENGTH;i++) {
    wordsize = maketegn();
  }
  i = 0;
  while ((c = getc(inputfile)) ≠ EOF) {
    ch = c;
    if ((ch > 96 && c < 123) || c == -27 || c == -8 || c == -26) {
      if (i>30) {
        printf("\nEt ord længere end 30 tegn ignoreret.");
        i = 0;
      }
      else {
        ord[i] = c;
      }
      i++;
    }
    else {
      ord[i] = '\0';
      ref = makeword_ref();
      wordsize = putword(wordsize, ord, ref);
      for (j=0;j<i;j++) {
        here = meta_dict[i][j][num(ord[j])];
        entry = makeword_ref();
        entry→word = ref→word;
        here = insert_ref(entry, here);
        meta_dict[i][j][num(ord[j])] = here;
      }
      free(ref);
      i = 0;
    }
  }
  return wordsize;
}
```

C.3 Ord.c

```
/* INCLUDE LIBRARIES ————————————————————
 */

#include <stdio.h>
#include <stdlib.h>

#include "timebackup.h"
#include "Dict.h"

/* ORD PROCEDURES ————————————————————
 */

void print_ord(tegn *last) {

  int i, up;
  tegn *tmp;
  char buffer[30];

  tmp = last;
  i = 0;
  up = 1;
  while(up) {
    if (tmp→up ≠ NULL) {
      buffer[i++] = tmp→letter;
      tmp = tmp→up;
    }
    else {
      buffer[i++] = tmp→letter;
      up = 0;
    }
  }
  while (i>1) {
    printf("%c", buffer[--i]);
  }
}

/* MAIN ————————————————————————*/

void main() {

/* DEFINE VARIABLES ————————————————————-
 */

  int length, i;

  char *dictionary, streng[30];    /* The name of the dictionary */

  tegn *dict;                          /* Pointer to the dictionary */

  long start, stop,                         /* Start and stop times */
        diff;                                        /* Running time */

  word_ref *ref, *new_ref;

/* SET VARIABLES  ———————————————————— */

/* The dictionary ................................................. */

  dictionary = "../Wordlist/ukcad.8";       /*Textsources/all.txt"; */

  start = measuretime();                        /* Start stop watch. */
  dict = makedict(dictionary);                /* Set up dictionary. */
  stop = measuretime();                          /* Stop stop watch. */
  diff = stop - start;        /* Calculate dictionary creation time */

  printf("\nTime :  %ld", diff);                      /* Time result. */

  printNUM();

  while(1) {
  printf("\nIndtast delstreng:  ");
  scanf("%s", streng);

  length = 0;
  while (streng[length] ≠ '\000') {length++;}
  ref = NULL;

  for (i=0;i<length;i++) {
    if (streng[i] ≠ '0') {
      new_ref = meta_dict[length][i][num(streng[i])];
      ref = combine(ref, new_ref);
      if (ref == NULL) { i = length; }
    }
  }

  if (ref ≠ NULL) {
    while (ref→next ≠ NULL) {
      printf("\n");
      print_ord(ref→word);
      ref = ref→next;
    }
    printf("\n");
    print_ord(ref→word);
  }
  else {
    printf("\nNo words matching the substring");
  }
  }
}
```

Appendix D

The Cross Module Code

> **Sometimes it pays to stay in bed on Monday, rather than spending the rest of the week debugging Monday's code.**
> *Dan Salomon*

D.1 Cross.h

```
/* This file contains procedures to build the puzzle and */
/* maintain the parameters for each cell in the puzzle. */
/* Procedures to print the grid are also available. */

/* Checks if this file have been compiled before. */
#ifndef CROSS

#include <stdio.h>
#include <stdlib.h>

#include <time.h>

#include "Head.h"
#include "Dict.h"

#define MAXH 30                  /* Maximim height of the grid.   */
#define MAXB 30                  /* Maximim width of the grid.    */

/* Definition of a cell in the grid. */

  typedef struct felt {
    char letter;                  /* The letter if any.            */
    bogstav *supply;              /* The letters in stock.         */
    struct felt *N;               /* The neighbor above,           */
    struct felt *S;               /* below,                        */
    struct felt *E;               /* to the right and              */
    struct felt *W;               /* to the left.                  */
    tegn *hdict;                  /* List of horisontal words.     */
    tegn *vdict;                  /* List of vartical words.       */
    int hpart;                    /* Length of current hori.fill.  */
    int htotal;                   /* Total length of hori. word.   */
    int vpart;                    /* Length of current vert.fill.  */
    int vtotal;                   /* Total length of vert. word.   */
    int number;                   /* Cell index number.            */
    int reset_num;                /* Number of resets made.        */
  } felt;


/* Definition af a pit in the potential cell list. */

  typedef struct pit {
    struct felt *square;          /* Pointer to a cell.            */
    int antal;                    /* Number of elements.           */
    struct pit *next;             /* The element after and         */
    struct pit *prev;             /* The element before.           */
    struct pit *old;              /* Temp. pointer keep.           */
} pit;


/* makefelt create a cell and initialize it. */

  struct felt *makefelt(void);


/* makepit create a pit and initialize it. */

  struct pit *makepit(void);


/* setdict sets the dictionary entries for a given cell. */

  void setdict(felt **act);


/* setletter finds a letter from the stock for a given cell */
/* in the predefined order used and moves to the next cell */
/* in that order. */

  void setletter(felt **actual, pit **walking);


/* Insert a given letter into a given cell. */

  void insertletter(felt *receiver, char letter);


/* THE FOLLOWING PROCEDURES ARE USED TO CHECK   */
/* THE CORRECT SIZE OF THE GRID. */

/* Procedure used to check correct user input; height of the */
/* grid. */

  int checkheight(int height);


/* Procedure used to check correct user input; width of the */
/* grid. */

  int checkwidth(int width);


/* THE FOLLOWING PROCEDURES ARE USED TO PLACE */
/* LETTERS AND BLANKS IN THE GRID AT CHOSEN    */
/* POSITIONS. */

/* Places a letter in the cell at position (xpos, ypos).  */

  void putletter(char c, felt *tag, int xpos, int ypos);


/* Marks the cell at position (xpos, ypos) as blank. */

  void putblank(felt *tag, int xpos, int ypos);


/* Insert a letter in the given cell. */

void insert_letter(felt *act);


/* THE FOLLOWING PROCEDURE BUILDS THE THE GRID. */


  struct felt *makegrid(int height, int width);


/* THE FOLLOWING PROCEDURES ARE USED TO PRINT */
/* THE GRID. */

/* Print the grid. */

  void printgrid(felt *tag);


/* THE FOLLOWING PROCEDURE is USED TO INITIALIZE */
/* DICTIONARY ENTRIES. */

/* Initialize horizontal word. */

  void markwords(felt *tag, Ptegn *dict);


/* Given two lists of letters getstock return the list */
/* containing the letters found i both input lists. */
```

```
  bogstav *getstock(tegn *h, tegn *v);


/* THE FOLLOWING PROCEDURES WORK ON */
/* THE STOCK LIST. */


/* Inserts a cell position to the list of potential next */
/* cells used in the dynamic walk of PREHARD WALK. */

void insertpit(pit *old, pit *new);


/* jump jumps to a specifyed element in the stock list. */

  tegn *jump(char fix, tegn *in);


/* count finds the number of letters in the stock list. */

  int count(bogstav *act);


/* Initialization of the random number generator. */

  void InitRand();


/* pickletter finds the letter in the stock list at a */
/* given position. */

  char pickletter(int random, bogstav *act);


/* choose chooses a letter from the stock list by random. */

  char choose(bogstav *supply);


/* removeletter removes a letter from the stock list. */

  bogstav *removeletter(char bad, bogstav *supply);

/* bogcounter finds the number of 'bogstav'-elements */
/* in use. */

int bogcounter(pit *walk);

/* THE FOLLOWING PROCEDURES ARE USED TO */
/* MOVE AROUND THE GRID. */

/* Move up. */

felt *walknorth(felt *tag);

/* Move down */

felt *walksouth(felt *tag);

/* Move to the right. */

felt *walkeast(felt *tag);

/* Move to the left. */

felt *walkwest(felt *tag);

/* Move to the top of the grid. Stay in the same column. */

felt *walkup(felt *tag);

/* Move to the bottom of the grid. Stay in the same column. */

felt *walkdown(felt *tag);

/* Move to the far right of the grid. Stay in the same row. */

felt *walkright(felt *tag);

/* Move to the far left of the grid. Stay in the same row. */

felt *walkleft(felt *tag);

#define CROSS
#endif
```

D.2 Cross.c

```
/* This file contains procedures to build the puzzle and */
/* maintain the parameters for each cell in the puzzle. */
/* Procedures to print the grid are also available. */

#include "Cross.h"
#include "Dict.h"

/* falloc allocate memory space for a cell.  */

struct felt *falloc(void) {
  return (struct felt *) malloc(sizeof(struct felt));
}

/* makefelt create a cell and initialize it. */

struct felt *makefelt(void) {
  felt *new;

  new = falloc();
  new→letter = '0';            /* Empty cell.                    */
  new→supply = NULL;            /* No letters in stock.          */
  new→N = NULL;                /* No neighbors above,            */
  new→S = NULL;               /* below,                          */
  new→E = NULL;                /* to the right                   */
  new→W = NULL;                 /* or to the left.               */
  new→hdict = NULL;             /* No possible horizontal        */
  new→vdict = NULL;             /* or vertical words.            */
  new→hpart = 0;                  /* Length of current hori.fill.*/
  new→htotal = 0;                /* Total length of hori. word. */
  new→vpart = 0;                   /* Length of current vert.fill.*/
  new→vtotal = 0;                 /* Total length of vert. word. */
  new→reset_num = 0;
  return new;
}

/* balloc allocate memory to a 'bogstav'. bogstav contain one */
/* letter. It is used to build the stock list for each cell.  */

struct bogstav *balloc(void) {
  return (struct bogstav *) malloc(sizeof(struct bogstav));
}

/* Frees memory space previously occupied by a 'bogstav'. */

void bfree(bogstav *head) {
  bogstav *p, *q;

  for (p = head; p ≠ NULL; p = q) {
    q = p→next;
    free(p);
  }
}

/* palloc  allocate memory space for a pit .Elements used to */
/* hold the predefined order used by STRAIGHT WALK and */
/* SNAKE WALK. */

struct pit *palloc(void) {
  return (struct pit *) malloc(sizeof(struct pit));
}

/* makepit create a pit and initialize it. */

struct pit *makepit(void) {
  pit *new;

  new = palloc();
  new→square = NULL;                  /* Empty pit.             */
  new→antal = 0;                      /* Number of elements. */
  new→next = NULL;                    /* No element after      */
  new→prev = NULL;                     /* or before.            */
  new→old = NULL;                      /* Temp. pointer keep. */
  return new;
}

/* Create and initialize a 'bogstav'. */

struct bogstav *makebogstav(void) {
  bogstav *new;

  new = balloc();
  new→letter = '0';
  new→next = NULL;
  new→prev = NULL;
  new→old = NULL;
  return new;
}
/* inserts a bogstav in the stock list. */

bogstav *insertbogstav(bogstav *out, bogstav *new) {
  bogstav *act;

  new→next = out;
  if (new→next ≠ NULL) {
    new→next→prev = new;
  }
  act = new;
  return act;
}
```

```
/* Given two lists of letters getstock return the list */
/* containing the letters found i both input lists. */

bogstav *getstock(tegn *h, tegn *v) {
  int finished, stop;
  tegn *temp1, *temp2, *small, *big;
  bogstav *new, *out;

  out = NULL;
  stop = finished = 0;

  temp1 = h;
  temp2 = v;
  while (!finished) {
    if (temp1 ≠ NULL && temp2 ≠ NULL) {
      if (temp1→letter ≤ temp2→letter) {
        small = temp1;
        big = temp2;
      }
      else {
        small = temp2;
        big = temp1;
      }
      while (!stop) {
        if (small→letter ≥ big→letter) {
          stop = 1;
        }
        else {
          if (small→next ≠ NULL) {
            small = small→next;
          }
          else {
            finished = 1;
            stop = 1;
          }
        }
      }
      stop = 0;
      if (!finished) {
        while (!stop) {
          if (small→letter == big→letter) {
            new = makebogstav();
            new→letter = big→letter;
            out = insertbogstav(out, new);
            if (small→next ≠ NULL && big→next ≠ NULL) {
              small = small→next;
              big = big→next;
            }
            else {
              finished = 1;
              stop = 1;
            }
          }
          else {
            stop = 1;
          }
        }
        temp1 = small;
        temp2 = big;
      }
      stop = 0;
    }
    else {
      finished = 1;
    }
  }
  return out;
}

/* THE FOLLOWING PROCEDURES WORK ON */
/* THE STOCK LIST. */

/* jump jumps to a specifyed element in the stock list. */

tegn *jump(char fix, tegn *in) {

  while (in ≠ NULL && in→letter ≠ fix) {
    in = in→next;
  }
  if (in == NULL) {
    return NULL;
  }
  else {
    return in→down;
  }
}

/* count finds the number of letters in the stock list. */

int count(bogstav *act) {
  int ant, cont;

  ant = 0;
  cont = 1;
```

```
  if (act ≠ NULL) {
    while (cont) {
      ant++;
      if (act→next ≠ NULL) {
        act = act→next;
      }
      else {
        cont = 0;
      }
    }
  }
  return ant;
}

/* Initialization of the random number generator. */

void InitRand() {
  time_t tid;

  time(&tid);
  srand(tid);
  srand(clock());
}

/* pickletter finds the letter in the stock list at a */
/* given position. */

char pickletter(int random, bogstav *act) {
  int ant;

  for (ant=1;ant≤random;ant++) {
    act = act→next;
  }
  return act→letter;
}

/* choose chooses a letter from the stock list by random. */

char choose(bogstav *supply) {
  int ant, random;
  bogstav *act;
  char c;

  c = '0';
  act = supply;
  ant = count(act);
  if (ant > 0) {
    random = rand() % ant;
    /* random = srand(clock()) % ant; */
    c = pickletter(random, act);
  }
  return c;
}

bogstav *removeletter(char bad, bogstav *supply) {

  int found, cont;

  found = 0; cont = 1;

  while (supply ≠ NULL && !found) {
    /* Find the bad letter if it exists */
    if (bad == supply→letter) {
      found = 1;
    }
    if (!found) {
      supply = supply→next;
    }
  }
  if (found) {
    /* Bad letter found - now remove it */
    if (supply→prev == NULL) {
      /* Bad letter is head of the list */
      if (supply→next == NULL) {
        /* Bad letter is also tail of the list */
        free(supply);
        /* Free element */
        supply = NULL;
      }
      else {
        /* Bad letter is head of the list - but not tail */
        supply = supply→next;
        supply→prev→next = NULL;
        supply→old = supply→prev;
        supply→prev = NULL;
        free(supply→old);
        supply→old = NULL;
      }
    }
    else {
      /* Bad letter is not head of the list */
      if (supply→next == NULL) {
        /* Bad letter is tail of the list */
        supply = supply→prev;
        supply→next→prev = NULL;
        supply→old = supply→next;
        supply→next = NULL;
        free(supply→old);
        supply→old = NULL;
      }
      else {
        /* Bad letter is neither head nor tail */
        supply→prev→next = supply→next;
        /* prev points to next */
        supply = supply→next;
        /* Move left */
        supply→old = supply→prev;
        /* Keep track of old bad letter */
        supply→prev = supply→prev→prev;
        /* next points to prev */
        supply→old→prev = NULL;
        supply→old→next = NULL;
        free(supply→old);
        supply→old = NULL;
      }
    }
  }
  if (supply ≠ NULL) {
    while (cont) {
      /* Goto head of list */
      if (supply→prev ≠ NULL) {
        supply = supply→prev;
      }
      else {
        cont = 0;
      }
    }
  }
  return supply;
}

/* removeletter removes a letter from the stock list. */

bogstav *removeletterOld(char bad, bogstav *supply) {
  int stop, found;

  stop = 0; found = 0;
  if (supply ≠ NULL) {
    while (!stop && !found && supply ≠ NULL) {
      if (supply→letter == bad) {
        found = 1;
      }
      if (supply ≠ NULL) {
        if (!found) {
          supply = supply→next;
        }
      }
      else {
        stop = 1;
      }
    }
    if (found && supply ≠ NULL && supply→prev ≠ NULL) {
      supply→prev→next = supply→next;
      if (supply→next ≠ NULL) {
        supply = supply→next;
        supply→old = supply→prev;
        supply→prev = supply→prev→prev;
        free(supply→old);
        supply→old = NULL;
      }
      while (supply→prev ≠ NULL) {
        supply = supply→prev;
      }
    }
    else {
      if (found && supply ≠ NULL && supply→next ≠ NULL) {
        supply = supply→next;
        free(supply→prev);
        supply→prev = NULL;
      }
      else {
        free(supply);
        supply = NULL;
      }
    }
  }
  return supply;
}

/* setdict sets the dictionary entries for a given cell. */

void setdict(felt **act) {

  if ((*act)→letter ≠ '/') {
    if ((*act)→hdict == NULL) {
      (*act)→hdict = jump((*act)→W→letter, (*act)→W→hdict);
    }
    if ((*act)→vdict == NULL) {
      (*act)→vdict = jump((*act)→N→letter, (*act)→N→vdict);
    }
  }
}
```

```
/* setletter finds a letter from the stock for a given cell */
/* in the predefined order used and moves to the next cell */
/* in that order. */

void setletter(felt **actual, pit **walking) {
  felt *act;
  pit *walk;

  act = *actual;
  walk = *walking;
  if ((*actual)→letter ≠ '/') {
    (*actual)→letter = choose((*actual)→supply);
  }
  if ((*walking)→next ≠ NULL) {
    *walking = (*walking)→next;
    *actual = (*walking)→square;
  }
}

/* Insert a given letter into a given cell. */

void insertletter(felt *receiver, char letter) {
  receiver→letter = letter;
}

/* THE FOLLOWING ARE PROCEDURES USED */
/* TO BUILD THE GRID. */

/* A cell new cell is added above the actual cell. */

void addnorth(felt *actual, felt *new) {

  if (actual→N ≠ NULL) {
    printf("\nWarning:  addnorth:  nyt felt sletter eksisterende
felt.");
    actual→N→S = NULL;
  }
  actual→N = new;
  new→S = actual;
}

/* A cell new cell is added below the actual cell. */

void addsouth(felt *actual, felt *new) {

  if (actual→S ≠ NULL) {
    printf("\nWarning:  addsouth:  nyt felt sletter eksisterende
felt.");
    actual→S→N = NULL;
  }
  actual→S = new;
  new→N = actual;
}

/* A cell new cell is added to the right of the actual cell. */

void addeast(felt *actual, felt *new) {

  if (actual→E ≠ NULL) {
    printf("\nWarning:  addeast:  nyt felt sletter eksisterende
felt.");
    actual→E→W = NULL;
  }
  actual→E = new;
  new→W = actual;
}

/* A cell new cell is added to the left of the actual cell. */

void addwest(felt *actual, felt *new) {

  if (actual→W ≠ NULL) {
    printf("\nWarning:  addwest:  nyt felt sletter eksisterende
felt.");
    actual→W→E = NULL;
  }
  actual→W = new;
  new→E = actual;
}

/* Removes the actual cell. */

/* void remove(felt *actual) {

  if (actual->N != NULL) {
    actual->N->S = NULL;
  }
  if (actual->S != NULL) {
    actual->S->N = NULL;
  }
  if (actual->E != NULL) {
    actual->E->W = NULL;
  }
  if (actual->W != NULL) {
    actual->W->E = NULL;
  }
} */

/* THE FOLLOWING PROCEDURES ARE USED TO */
/* CHECK THE CORRECT SIZE OF THE GRID. */

/* Procedure used to check correct user input; height of */
/* the grid. */

int checkheight(int height) {
  int res;

  res = 1;
  if (height<0 || height>MAXH) {
    res = 0;
  }
  return res;
}

/* Procedure used to check correct user input; width of the */
/* grid. */

int checkwidth(int width) {
  int res;

  res = 1;
  if (width<0 || width>MAXH) {
    res = 0;
  }
  return res;
}

/* THE FOLLOWING PROCEDURES ARE USED */
/* TO PLACE LETTERS AND BLANKS IN THE */
/* GRID AT CHOSEN POSITIONS. */

/* Places a letter in the cell at position (xpos, ypos). */

void putletter(char c, felt *tag, int xpos, int ypos) {
  felt *here;
  int h, w;

  here = tag;
  for (w=1;w<xpos;w++) {
    here = here→E;
  }
  for (h=1;h<ypos;h++) {
    here = here→S;
  }
  here→letter = c;
}

/* Marks the cell at position (xpos, ypos) as blank. */

void putblank(felt *tag, int xpos, int ypos) {

  putletter('/',tag,xpos,ypos);
}

/* Insert a letter in the given cell. */

void insert_letter(felt *act) {

  if (act→supply ≠ NULL) {
    act→letter = choose(act→supply);
  }
}


/* THE FOLLOWING PROCEDURES ARE USED */
/* TO MOVE AROUND THE GRID. */

/* Move up. */

felt *walknorth(felt *tag) {
  felt *act, *res;

  act = tag;
  if (act→N ≠ NULL) {
    res = act→N;
  }
  else {
    printf("\nWarning:  walknorth:  ulovlig walk, felt eksisterer
ikke.");
  }
  return res;
}

/* Move down */

felt *walksouth(felt *tag) {
  felt *act, *res;

  act = tag;
  if (act→S ≠ NULL) {
    res = act→S;
```

```
  }
  else {
    printf("\nWarning:  walksouth:  ulovlig walk, felt eksisterer
ikke.");
  }
  return res;
}

/* Move to the right. */

felt *walkeast(felt *tag) {
  felt *act, *res;

  act = tag;
  if (act→E ≠ NULL) {
    res = act→E;
  }
  else {
    printf("\nWarning:  walkeast:  ulovlig walk, felt eksisterer
ikke.");
  }
  return res;
}

/* Move to the left. */

felt *walkwest(felt *tag) {
  felt *act, *res;

  act = tag;
  if (act→W ≠ NULL) {
    res = act→W;
  }
  else {
    printf("\nWarning:  walkwest:  ulovlig walk, felt eksisterer
ikke.");
  }
  return res;
}

/* Move to the top of the grid. Stay in the same column. */

felt *walkup(felt *tag) {
  felt *act;

  act = tag;
  while (act→N ≠ NULL) {
    act = walknorth(act);
  }
  return act;
}

/* Move to the bottom of the grid. Stay in the same column. */

felt *walkdown(felt *tag) {
  felt *act;

  act = tag;
  while (act→S ≠ NULL) {
    act = walksouth(act);
  }
  return act;
}

/* Move to the far right of the grid. Stay in the same row. */

felt *walkright(felt *tag) {
  felt *act;

  act = tag;
  while (act→E ≠ NULL) {
    act = walkeast(act);
  }
  return act;
}

/* Move to the far left of the grid. Stay in the same row. */

felt *walkleft(felt *tag) {
  felt *act;

  act = tag;
  while (act→W ≠ NULL) {
    act = walkwest(act);
  }
  return act;
}

/* THE FOLLOWING PROCEDURE BUILDS THE */
/* GRID. */

struct felt *makegrid(int height, int width) {
  felt *start, *act, *left, *above, *new;
  int w, h, number;

  number = 1;
  for (h=1;h≤height;h++) {
    for (w=1;w≤width;w++) {
      new = makefelt();
      new→number = number;
      number++;
      if (w == 1) {
        if (h == 1) {
          act = left = above = start = new;
        }
        else {
          addsouth(left, new);
          act = new;
          above = left;
          left = left→S;
        }
      }
      else {
        addeast(act, new);
        act = new;
        if (h>1) {
          above = above→E;
          addnorth(act, above);
        }
      }
    }
  }
  return start;
}

/* THE FOLLOWING PROCEDURES ARE */
/* USED TO PRINT THE GRID. */

/* Print a row. */

void printrow(felt *tag) {
  felt *act;

  act = tag;
  printf(" ");
  while (act→E ≠ NULL) {
    printf("%c", act→letter);
    act = act→E;
  }
  printf("%c", act→letter);
  printf("\n");
}

/* Print the grid. */

void printgrid(felt *tag) {
  felt *left;

  left = tag;
  printf("\n");
  while (left→S ≠ NULL) {
    printrow(left);
    left = left→S;
  }
  printrow(left);
  printf("\n");
}

/* THE FOLLOWING PROCEDURES ARE */
/* USED TO DETERMINE THE LENGTH */
/* OF A WORD. */

/* Length of horizontal word. */

int nexthorisontal(felt *tag) {
  felt *ac;
  int length, next;

  ac = tag;
  length = 0;
  next = 1;
  while (next) {
    if (ac→letter ≠ '/') {
      length++;
    }
    else {
      next = 0;
    }
    if (ac→E ≠ NULL) {
      ac = walkeast(ac);
    }
    else {
      next = 0;
    }
  }
  return length;
}

/* Length of vertical word. */

int nextvertical(felt *tag) {
  felt *ac;
```

```
  int length, next;

  ac = tag;
  length = 0;
  next = 1;
  while (next) {
    if (ac→letter ≠ '/') {
      length++;
    }
    else {
      next = 0;
    }
    if (ac→S ≠ NULL) {
      ac = walksouth(ac);
    }
    else {
      next = 0;
    }
  }
  return length;
}

/* THE FOLLOWING PROCEDURES ARE */
/* USED TO INITIALIZE DICTIONARY */
/* ENTRIES. */

/* Initialize horizontal word. */

void horisontalwords(felt *tag, Ptegn *dict) {
  felt *ac, *tracker;
  int in, length, cont;

  tracker = ac = tag;
  cont = in = length = 1;
  while (cont) {
    if (ac→E == NULL) {
      if (ac→W→letter == '/') {
        ac→hdict = dict[1];
      }
      if (ac→S ≠ NULL) {
        ac = walkleft(walksouth(ac));
      }
      else {
        cont = 0;
      }
    }
    else {
      if (ac→letter == '/') {
        ac = walkeast(ac);
      }
      else {
        length = nexthorisontal(ac);
        ac→hdict = dict[length];
        for (in=0;in<length;in++) {
          ac→hpart = in;
          ac→htotal = length;
          if (ac→E ≠ NULL) {
            ac = walkeast(ac);
          }
        }
      }
    }
  }
}

/* Initialize vertical word. */

void verticalwords(felt *tag, Ptegn *dict) {
  felt *ac, *tracker;
  int in, length, cont;

  tracker = ac = tag;
  cont = in = length = 1;
  while (cont) {
    if (ac→S == NULL) {
      if (ac→N→letter == '/') {
        ac→vdict = dict[1];
      }
      if (ac→E ≠ NULL) {
        ac = walkup(walkeast(ac));
      }
      else {
        cont = 0;
      }
    }
    else {
      if (ac→letter == '/') {
        ac = walksouth(ac);
      }
      else {
        length = nextvertical(ac);
        ac→vdict = dict[length];
        for (in=0;in<length;in++) {
          ac→vpart = in;
          ac→vtotal = length;
          if (ac→S ≠ NULL) {
                  ac = walksouth(ac);
               }
            }
         }
      }
   }
}

/* Initialize words. */

void markwords(felt *tag, Ptegn *dict) {

   horisontalwords(tag, dict);
   verticalwords(tag, dict);
}

/* Inserts a cell position to the list of potential */
/* next cells used in the dynamic walk */
/* of PREHARD WALK. */

void insertpit(pit *old, pit *new) {

   if (old == new) {
      old→next = new;
      new→prev = old;
   }
   else {
      new→next = old→next;
         if (old→next ≠ NULL) {
            old→next→prev = new;
         }
         old→next = new;
         new→prev = old;
   }
}

/* bogcounter finds the number of 'bogstav'-elements in use. */

int bogcounter(pit *walk) {
   int res;

   res = 0;
   if (walk ≠ NULL) {
      while (walk→prev ≠ NULL) {
         walk = walk→prev;
      }
   }
   while (walk ≠ NULL) {
      res = res + count(walk→square→supply);
      walk = walk→next;
   }
   return res;
}
```

Appendix E

The Walk Heuristics Code

> **Don't get suckered in by the comments – they can be terribly misleading. Debug only code.**
>
> *Dave Storer*

E.1 walkheurestics.h

```
/* walkheurestics contain procedures that define the order in */
/* which to fill the grid. */

#ifndef WALK
/* Checks if this file have been compiled before. */

#include "Head.h"

#include "Cross.h"
#include "Dict.h"


/* makewalk is told either to produce the predefined used */
/* either by  STRAIGHT WALK or SNAKE WALK. */

  pit *makewalk(felt *tag, int program);


/* This procedure makes the initial list of cells used by */
/* PREHARD WALK. */

  pit *walkhardinit(felt *tag);


/* This procedure defines the walk for PREHARD WALK. */
/* The walk is created dynamically during run time. */

  struct felt *Walkhard(pit *tag);


/* addpit adds a pit to the list of potential cells. */
/* The list is sorted with the smallest first. */


pit *addpit(pit *head, felt *tegn);


/* subtractpit delete bad from head. */

pit *subtractpit(pit *head, felt *bad);


/* This procedure checks if a given cell is a legal prefix cell. */

int hardcheck(felt *tegn);

#define WALK
#endif
```

E.2 walkheurestics.c

```
/* walkheurestics contain procedures that define the order */
/* in which to fill the grid. */

#include "walkheurestics.h"
#include "Head.h"

#include "Cross.h"
#include "Dict.h"


/* THE FOLLOWING PROCEDURES ARE USED */
/* BY STRAIGHT WALK and SNAKE WALK. */

/* This procedure is used to produces the predefined walk /*
/* used by STRAIGHT WALK. Given a cell position */
/* it returns the next cell in the walk. */

felt *nextstraightpit(felt *tag) {

  felt *ac;

  ac = tag;

  if (ac→E == NULL && ac→S == NULL) {
    ac = NULL;
  }
  else {
    if (ac→E ≠ NULL) {
      ac = walkeast(ac);
    }
    else {
      if (ac→S ≠ NULL) {
        ac = walksouth(walkleft(ac));
      }
      else {
        ac = NULL;
      }
    }
  }

  return ac;
}

/* This procedure is used to produce the predefined walk */
/* used by SNAKEWALK. Given a cell position it */
/* returns the next cell in the walk. */

felt *nextsnakepit(felt *tag) {

  static felt *corner;
  felt *ac;
  static int right;

  ac = tag;

  if (ac→E == NULL && ac→S == NULL) {
    ac = NULL;
  }
  else {
    if (ac→W == NULL && ac→N == NULL) {
      corner = ac;
      right = 1;
      if (ac→E ≠ NULL) {
        ac = walkeast(ac);
      }
      else {
        ac = walksouth(ac);
      }
    }
    else if (ac→W == corner) {
      corner = ac;
      if (ac→S ≠ NULL) {
        ac = walkleft(walksouth(ac));
        right = 1;
      }
      else {
        ac = walkup(walkeast(ac));
```

```
            right = 0;
         }
      }
      else if (ac→N == corner) {
         corner = ac;
         if (ac→E ≠ NULL) {
            ac = walkup(walkeast(ac));
            right = 0;
         }
         else {
            ac = walkleft(walksouth(ac));
            right = 1;
         }
      }
      else {
         if (right) {
            ac = walkeast(ac);
         }
         else {
            ac = walksouth(ac);
         }
      }
   }
   return ac;
}


/* This procedure is used to produce the predefined walk */
/* used by SWITCHWALK. Given a cell position */
/* it returns the next cell in the walk. */

felt *nextswitchpit(felt *tag) {

   static felt *corner;
   felt *ac;
   int i,j;

   ac = tag;

   if (ac→E == NULL && ac→S == NULL) {
      ac = NULL;
   }
   else {
      i = ((ac→number - 1) % bredde) + 1;
      j = ((ac→number - 1) / bredde) + 1;
      if (ac→W == NULL && ac→N == NULL) {
         corner = ac;
      }
      if (j≤i) {
         if (ac→E ≠ NULL) {
            ac = ac→E;
         }
         else {
            ac = walksouth(corner);
            if (corner→S ≠ NULL) {
               corner = corner→S;
            }
         }
      }
      else {
         if (ac→S ≠ NULL) {
            ac = ac→S;
         }
         else {
            ac = walkeast(corner);
            if (corner→E ≠ NULL) {
               corner = corner→E;
            }
         }
      }
   }
   return ac;
}

/* This procedure is used to produce the predefined */
/* walk used by SIKSAKWALK. Given a cell */
/* position it returns the next cell in the walk. */

felt *nextsiksakpit(felt *tag) {

   felt *ac;
   int dummy;

   ac = tag;
   dummy = 1;
   if (ac→S == NULL && ac→E == NULL) {
      ac = NULL;
   }
   else {
      if (ac→S ≠ NULL && ac→W ≠ NULL) {
         ac = walksouth(walkwest(ac));
      }
      else {
         ac = walkeast(ac);
         while (dummy) {
            if (ac→N ≠ NULL && ac→E ≠ NULL) {
               ac = walknorth(walkeast(ac));
            }
            else {
               dummy = 0;
            }
         }
         dummy = 1;
      }
   }
   return ac;
}

/* This procedure is used to produce the predefined */
/* walk used by SLALOMWALK. Given a cell */
/* position it returns the next cell in the walk. */

felt *nextslalompit(felt *tag) {

   felt *ac;
   int i,j;

   ac = tag;

   if (ac→E == NULL && ac→S == NULL) {
      ac = NULL;
   }
   else {
      i = ((ac→number - 1) % bredde) + 1;
      j = ((ac→number - 1) / bredde) + 1;
      if (((i+j) % 2) == 0) {
         if (j == 1) {
            if (i<bredde) {
               ac = walkeast(ac);
            }
            else {
               ac = walksouth(ac);
            }
         }
         else {
            if (i<bredde) {
               ac = walkeast(walknorth(ac));
            }
            else {
               ac = walksouth(ac);
            }
         }
      }
      else {
         if (i == 1) {
            if (j<hojde) {
               ac = walksouth(ac);
            }
            else {
               ac = walkeast(ac);
            }
         }
         else {
            if (j<hojde) {
               ac = walkwest(walksouth(ac));
            }
            else {
               ac = walkeast(ac);
            }
         }
      }
   }
   return ac;
}

/* makewalk is told either to produce the predefined */
/* used either by STRAIGHT WALK, */
/* SNAKE WALK or SWITCH WALK. */

pit *makewalk(felt *tag, int program) {
   pit *act, *prevact;
   felt *tegn;

   prevact = act = NULL;
   tegn = tag;
   if (tegn→letter ≠ '/') {
      prevact = act = makepit();
      act→square = tegn;
      insertpit(prevact, act);
   }
   switch (program) {
   case SNAKEWALK:
      tegn = nextsnakepit(tegn);
      break;
   case STRAIGHTWALK:
      tegn = nextstraightpit(tegn);
      break;
   case SWITCHWALK:
      tegn = nextswitchpit(tegn);
      break;
   case SIKSAKWALK:
      tegn = nextsiksakpit(tegn);
      break;
    case SLALOMWALK:
      tegn = nextslalompit(tegn);
      break;

    default:
      printf("\nmakewalk: forkert programtype!");
    }
    while (tegn ≠ NULL) {
      if (tegn→letter ≠ '/') {
        act = makepit();
        act→square = tegn;
        if (prevact ≠ NULL) {
          insertpit(prevact, act);
        }
        else {
          act→next = act;
          act→prev = act;
        }
        prevact = act;
      }
      switch (program) {
      case SNAKEWALK:
        tegn = nextsnakepit(tegn);
        break;
      case STRAIGHTWALK:
        tegn = nextstraightpit(tegn);
        break;
      case SWITCHWALK:
        tegn = nextswitchpit(tegn);
        break;
      case SIKSAKWALK:
        tegn = nextsiksakpit(tegn);
        break;
      case SLALOMWALK:
        tegn = nextslalompit(tegn);
        break;

      default:
        printf("\nmakewalk: forkert programtype!");
      }
    }
    if (act→next ≠ NULL) {
      act = act→next;
      act→prev→next = NULL;
      act→prev = NULL;
    }
    return act;
}

/* THE FOLLOWING PROCEDURES ARE */
/* USED BY PREHARD WALK. */

/* This procedure checks if a given cell is a legal */
/* prefix cell. */

int hardcheck(felt *tegn) {
    int res, top, left;

    res = top = left = 0;

    if (tegn→N == NULL) { top = 1; }
    if (tegn→W == NULL) { left = 1; }
    if (tegn→N ≠ NULL && !empty(tegn→N→letter)) { top = 1; }
    if (tegn→W ≠ NULL && !empty(tegn→W→letter)) { left = 1; }
    if (top && left) {
      res = 1;
    }

    return res;
}

/* This procedure makes the initial list of cells */
/* used by PREHARD WALK. */

pit *walkhardinit(felt *tag) {

    felt *tegn;
    pit *head;

    head = NULL;
    tegn = tag;

    if (tegn→letter == '0') {
      head = addpit(head, tegn);
    }
    while ((tegn = nextstraightpit(tegn)) ≠ NULL) {
      if (hardcheck(tegn)) {
        head = addpit(head, tegn);
      }
    }

    return head;
}

/* double return true if index is found in the list head. */
```

```
int twice(pit *head, int index) {

    pit *temp;
    int res;

    res = 0;
    temp = head;
    if (temp ≠ NULL) {
      if (temp→square→number == index) {
        res = 1;
      }
      while(temp→next ≠ NULL) {
        temp = temp→next;
        if (temp→square→number == index) {
          res = 1;
        }
      }
    }
    return res;
}


/* addpit adds a pit to the list of potential cells. */
/* The list is sorted with the smallest first. */

pit *addpit(pit *head, felt *tegn) {

    pit *new;
    int dublet;

    dublet = twice(head, tegn→number);

    if (!dublet) {

      new = makepit();

      if (tegn→supply == NULL) {
        setdict(&tegn);
        tegn→supply = getstock(tegn→hdict, tegn→vdict);
        if (tegn→supply ≠ NULL) {
          new→antal = count(tegn→supply);
        }
        else {
          new→antal = 0;
        }
      }
      else {
        new→antal = count(tegn→supply);
      }

      new→square = tegn;

      while (head ≠ NULL && head→next ≠ NULL && new→antal
< head→antal) {
        head = head→next;
      }

      if (head == NULL) {
        head = new;
      }
      else {
        new→next = head;
        if (head→prev ≠ NULL) {
          head→prev→next = new;
          new→prev = head→prev;
        }
        head→prev = new;
      }

      while (head→prev ≠ NULL) {
        head = head→prev;
      }
    }

    return head;
}

/* subtractpit delete bad from head. */

pit *subtractpit(pit *head, felt *bad) {

    if (head ≠ NULL) {
      while (head→next ≠ NULL && bad ≠ head→square) {
        head = head→next;
      }
      if (head→square == bad) {
        /* bad found in head */
        if (head→prev == NULL) {
          /* Bad pit is head of the list */
          if (head→next == NULL) {
            /* Bad pit is also tail of the list */
            free(head);
            /* Free element */
            head = NULL;
          }
```

```
            else {
               /* Bad pit is head of the list - but not tail */
               head = head→next;
               head→prev→next = NULL;
               head→old = head→prev;
               head→prev = NULL;
               free(head→old);
               head→old = NULL;
            }
         }
         else {
            /* Bad pit is not head of the list */
            if (head→next == NULL) {
               /* Bad pit is tail of the list */
               head = head→prev;
               head→next→prev = NULL;
               head→old = head→next;
               head→next = NULL;
               free(head→old);
               head→old = NULL;
            }
            else {
               /* Bad pit is neither head nor tail */
               head→prev→next = head→next;
               /* prev points to next */
               head = head→next;
               /* Move left */
               head→old = head→prev;
               /* Keep track of old bad pit */
               head→prev = head→prev→prev;
               /* next points to prev */
               head→old→prev = NULL;
               head→old→next = NULL;
               free(head→old);
               head→old = NULL;
            }
         }
      }
   }
   if (head ≠ NULL) {
      while (head→prev ≠ NULL) {
         head = head→prev;
      }
   }
   return head;
}

/* This procedure defines the walk for PREHARD WALK. */
/* The walk is created dynamically during run time.  */

felt *Walkhard(pit *head) {

   pit *new;
   felt *tegn;

   new = head;

   while (new ≠ NULL && new→next ≠ NULL && new→antal
== 0) {
      new = new→next;
   }
   if (new→antal == 0) {
      tegn = NULL;
   }
   else {
      tegn = new→square;
   }

   return tegn;
}
```

Appendix F

The Back Heuristics Code

> **It won't be covered in the book. The source code has to be useful for something, after all ... :-)**
>
> *Larry Wall*

F.1 backtrack.h

```
/* This file include procedures used in the */
/* backtracking sessions of STRAIGHT WALK, */
/* SNAKE WALK and PREHARD WALK. */

#ifndef BACK
/* Checks if this file have been compiled before. */

#include "Cross.h"
#include "Dict.h"


/* This procedure is called when the static walks */
/* need to backtrack. */

  int Straight_backtrack(felt **actual, pit **walk);
  int Snake_backtrack(felt **actual, pit **walk);
  int Switch_backtrack(felt **actual, pit **walk);
  int Siksak_backtrack(felt **actual, pit **walk);
  int Slalom_backtrack(felt **actual, pit **walk);

/* This procedure is called when PREHARD WALK */
/* needs to backtrack. */

  pit *hard_backtrack(pit *head, pit *first_head);

/* The list containing the potential cells is updated */
/* by updatehead. */

pit *updatehead(pit *head, felt *old);

/* empty returns true if the char c is '0'. */

int empty(char c);

felt *update_reset(felt *cell);

#define BACK
#endif
```

F.2 backtrack.c

```
/* This file include procedures used in the */
/* backtracking sessions */

#include "backtrack.h"

#include "Head.h"

#include "Cross.h"
#include "Dict.h"
#include "walkheurestics.h"

/* THE FOLLOWING PROCEDURES ARE */
/* USED IN THE BACKTRACKING PROCESSES. */

felt *find_straight_dest(felt *origin, pit *walk, int multi) {

  felt *dest;

  if (!multi && origin→W == NULL) {
    dest = origin→N;
  }
  else {
    dest = walk→prev→square;
  }
  return dest;
}

felt *find_snake_dest(felt *origin, pit *walk, int multi) {

  felt *dest;

  if (origin→S ≠ NULL && (origin→S→letter == '0'
||origin→S→letter == '/')) {
    dest = find_straight_dest(origin, walk, multi);
  }
  else {
    if ((multi && origin→S == NULL) || origin→N ≠ NULL) {
      dest = walk→prev→square;
    }
    else {
      dest = origin→E;
    }
  }

  return dest;
}

felt *find_switch_dest(felt *origin, pit *walk, int multi) {

  felt *dest;
  int i,j;

  i = ((origin→number - 1) % bredde) + 1;
  j = ((origin→number - 1) / bredde) + 1;
  if (multi) {
    dest = walk→prev→square;
  }
  else {
    if (i == j) {
      dest = origin→W;
    }
    else if (i == (j-1)) {
      dest = origin→N;
    }
    else {
      dest = walk→prev→square;
    }
  }
  return dest;
}

felt *find_siksak_dest(felt *origin, pit *walk, int multi) {

  felt *dest;

  if (multi) {
    dest = walk→prev→square;
  }
  else {
    if (origin→W ≠ NULL && origin→W→letter ≠ '/') {
      dest = origin→W;
    }
    else if (origin→N ≠ NULL && origin→N→letter ≠ '/') {
      dest = origin→N;
    }
    else {
      dest = walk→prev→square;
    }
  }
```

```
   return dest;
}

felt *find_slalom_dest(felt *origin, pit *walk, int multi) {

   felt *dest;
   int i,j;

   i = ((origin→number - 1) % bredde) + 1;
   j = ((origin→number - 1) / bredde) + 1;
   if (multi) {
      dest = walk→prev→square;
   }
   else {
      if ((i % 2) == 0) {
         if ((j % 2) == 0 || origin→N == NULL) {
            if (origin→W→letter ≠ '/') {
               dest = origin→W;
            }
            else {
               dest = walk→prev→square;
            }
         }
         else {
            if (origin→N→letter ≠ '/') {
               dest = origin→N;
            }
            else {
               dest = walk→prev→square;
            }
         }
      }
      else {
         if ((j % 2) == 0 || origin→W == NULL) {
            if (origin→N→letter ≠ '/') {
               dest = origin→N;
            }
            else {
               dest = walk→prev→square;
            }
         }
         else {
            if (origin→W→letter ≠ '/') {
               dest = origin→W;
            }
            else {
               dest = walk→prev→square;
            }
         }
      }
   }
   return dest;
}

int test_up (felt *origin, felt *dest) {

   int res;

   res = 0;
   if (origin→N ≠ NULL && origin→N == dest) {
      res = 1;
   }

   return res;
}

/* This procedure is called when STRAIGHT WALK   */
/* need to backtrack. */

int Backtrack(felt **actual, pit **walk, int program) {

   int res, up, multi;
   felt *origin, *dest;

   res = 1;
   up = 0;
   multi = 0;
   origin = *actual;
   /* Remember where we started */

   while ((*actual)→supply == NULL && res) {
      if ((*walk)→prev == NULL) {
         res = 0;
      }
      else {

         switch (program) {
         case STRAIGHTWALK:
            dest = find_straight_dest(origin, *walk, multi);
         break;
         case SNAKEWALK:
            dest = find_snake_dest(origin, *walk, multi);
         break;
         case SWITCHWALK:
            dest = find_switch_dest(origin, *walk, multi);
         break;
```

```
         case SIKSAKWALK:
            dest = find_siksak_dest(origin, *walk, multi);
         break;
         case SLALOMWALK:
            dest = find_slalom_dest(origin, *walk, multi);
         break;

         default:
            printf("\nMain:  Ukendt programtype!");
         }
         if (test_up(origin, dest) || program == SWITCHWALK ||
program == SIKSAKWALK || program == SLALOMWALK ||
program == SNAKEWALK) {
            up = 1;
         }

         while ((*walk)→prev→square ≠ dest && up) {
            *walk = (*walk)→prev;
            /* Backtrack more than one step */
            (*actual) = (*walk)→square;
            while ((*actual)→supply ≠ NULL) {
               /* Drain the list */
               (*actual)→supply =
                  removeletter((*actual)→supply→letter,(*actual)→supply);
            }
            if ((*actual)→E ≠ NULL) {
               /* No hdict */
               (*actual)→E→hdict = NULL;
            }
            if ((*actual)→S ≠ NULL) {
               /* No vdict */
               (*actual)→S→vdict = NULL;
            }
            (*actual)→letter = '0';
            origin = *actual;
         }
         up = 0;
         *walk = (*walk)→prev;
         (*actual) = (*walk)→square;
         (*actual)→supply = removeletter((*actual)→letter,
(*actual)→supply);
         (*actual)→letter = '0';
         origin = *actual;
         if ((*actual)→E ≠ NULL) {
            /* No hdict */
            (*actual)→E→hdict = NULL;
         }
         if ((*actual)→S ≠ NULL) {
            /* No vdict */
            (*actual)→S→vdict = NULL;
         }
      }
      multi = 1;
   }
   return res;
}


int Snake_backtrack(felt **actual, pit **walk) {

   int res, up, multi;
   felt *origin, *dest;

   res = 1;
   up = 0;
   multi = 0;
   origin = *actual;
   /* Remember where we started */

   while ((*actual)→supply == NULL && res) {
      if ((*walk)→prev == NULL) {
         res = 0;
      }
      else {
         dest = find_snake_dest(origin, *walk, multi);
         if (test_up(origin, dest)) {
            up = 1;
         }
         while ((*walk)→prev→square ≠ dest && up) {
            *walk = (*walk)→prev;
            /* Backtrack more than one step */
            (*actual) = (*walk)→square;
            while ((*actual)→supply ≠ NULL) {
               /* Drain the list */
               (*actual)→supply =
                  removeletter((*actual)→supply→letter,(*actual)→supply);
            }
            if ((*actual)→E ≠ NULL) {
               /* No hdict */
               (*actual)→E→hdict = NULL;
            }
            if ((*actual)→S ≠ NULL) {
               /* No vdict */
               (*actual)→S→vdict = NULL;
            }
            (*actual)→letter = '0';
```

```
        origin = *actual;
      }
      up = 0;
      *walk = (*walk)→prev;
      (*actual) = (*walk)→square;
      (*actual)→supply = removeletter((*actual)→letter,
(*actual)→supply);
      (*actual)→letter = '0';
      if ((*actual)→E ≠ NULL) {
        /* No hdict */
        (*actual)→E→hdict = NULL;
      }
      if ((*actual)→S ≠ NULL) {
        /* No vdict */
        (*actual)→S→vdict = NULL;
      }
    }
    multi = 1;
  }
  return res;
}


/* THE FOLLOWING PROCEDURES ARE USED */
/* IN THE BACKTRACKING PROCES DONE BY  */
/* PREHARD WALK. */

/* The list containing the potential cells is updated */
/* by updatehead. */

pit *updatehead(pit *head, felt *old) {

  head = subtractpit(head, old);
  /* Remove pit */

  if (old→S ≠ NULL && hardcheck(old→S)) {
    /* Add pits */
    head = addpit(head, old→S);
  }
  if (old→E ≠ NULL && hardcheck(old→E)) {
    head = addpit(head, old→E);
  }
  return head;
}

/* empty returns true if the char c is '0'. */

int empty(char c) {

  int res;

  res = 0;
  if (c == '0') {
    res = 1;
  }
  return res;
}

/* dumb_pred removes illegal elements from list, */
/* so being when cell have been added. */

pit *dumb_pred(pit *list, felt *cell) {

  int finished;
  felt *north, *east, *south, *west, *cmp;
  pit *temp;

  finished = 0;
  temp = list;

  while (!finished) {
    if (temp == NULL) {
      finished = 1;
    }
    else {
      north = cell→N;
      east = cell→E;
      south = cell→S;
      west = cell→W;
      cmp = temp→square;
      if (north == cmp) {
        list = subtractpit(list, cmp);
      }
      if (west == cmp) {
        list = subtractpit(list, cmp);
      }
      if (south ≠ NULL && south == cmp) {
        list = subtractpit(list, south);
      }
      if (east ≠ NULL && east == cmp) {
        list = subtractpit(list, east);
      }
      temp = temp→next;
    }
  }
  return list;
}

/* new_cell return true if cell exists in list, */
/* false otherwise. */

int new_cell(pit *list, felt *cell) {

  int res, finished;
  pit *temp;

  finished = 0;
  res = 1;
  temp = list;

  if (temp == NULL) {
    res = 1;
  }
  else {
    while (!finished) {
      if (temp == NULL) {
        finished = 1;
      }
      else {
        if (temp→square == cell) {
          finished = 0;
        }
        temp = temp→next;
      }
    }
  }

  return res;
}

/* submit inserts cell into list if not already there. */
/* It also removes cells from list that are neighbors */
/* above or to the left of cell. */

pit *submit(pit *list, felt *cell) {

  if (new_cell(list, cell)) {
    list = addpit(list, cell);
    list = dumb_pred(list, cell);
  }

  return list;
}

/* destinations returns a list of all possible */
/* backtracking cells given a list of potential cells. */

pit *destinations(pit *list) {

  int finished;
  felt *north, *west;
  pit *dest, *temp;

  finished = 0;
  dest = NULL;
  temp = list;

  while (!finished) {
    if (temp == NULL) {
      finished = 1;
    }
    else {
      north = temp→square→N;
      west = temp→square→W;
      if (north ≠ NULL) {
        dest = submit(dest, north);
      }
      if (west ≠ NULL) {
        dest = submit(dest, west);
      }
      temp = temp→next;
    }
  }

  return dest;
}

/* find_element find the cell with the smallest supply */
/* which is not empty. */

felt *find_element(pit *good) {

  int finished, low, num;
  felt *elem, *current;
  pit *temp;

  low = 29;
  temp = good;
  elem = NULL;
  finished = 0;
  if (temp ≠ NULL) {
    while (!finished) {
      current = temp→square;
      num = temp→antal;
      if (num > 1 && num < low) {
        elem = current;
        low = num;
      }
      if (temp→next ≠ NULL) {
        temp = temp→next;
      }
      else {
        finished = 1;
      }
    }
  }

  return elem;
}

/* free_list deallocate memory previously occupied by */
/* list elements. */

pit *free_list(pit *list) {

  while (list ≠ NULL) {
    if (list→next == NULL) {
      free(list);
      list = NULL;
    }
    else {
      list = list→next;
      list→prev→prev = NULL;
      list→prev→next = NULL;
      list→old = list→prev;
      free(list→old);
      list→prev = NULL;
      list→old = NULL;
    }
  }

  return list;
}

/* clean_cell initializes the cell parameters. */

felt *clean_cell(felt *cell) {

  while (cell→supply ≠ NULL) {
    cell→supply = removeletter(cell→supply→letter, cell→supply);
  }
  cell→letter = '0';
  if (cell→E ≠ NULL) {
    cell→E→hdict = NULL;
  }
  if (cell→N ≠ NULL && empty(cell→N→letter)) {
    cell→vdict = NULL;
  }
  cell→reset_num = 0;

  return cell;
}

int prefix_cell(felt *cell) {

  int res, up, left, emp;

  up = left = emp = res = 0;

  if (empty(cell→letter)) { emp = 1; }
  if (cell→W == NULL) { left = 1; }
  if (cell→W ≠ NULL && !empty(cell→W→letter)) { left = 1; }

  /*
  if (cell->N == NULL) { up = 1; }
  if (cell->N != NULL && !empty(cell->N->letter)) { up = 1; }
  */

  up = 1;
  /* Deleting cells from above. */

  if (up && left && emp) { res = 1; }

  return res;
}

/* clean_column clear the actual column below temp. */

pit *clean_column(pit *head, felt *org) {

  int bottom;
  felt *temp;

  bottom = 0;
  temp = org;

  while (!bottom) {
    if (temp ≠ NULL) {
      if (!empty(temp→letter)) {
        temp = clean_cell(temp);
        temp = temp→S;
      }
      else {
        if (temp→W == NULL ||
            (temp→W ≠ NULL && !empty(temp→W→letter))) {
          head = subtractpit(head, temp);
        }
        temp = clean_cell(temp);
        bottom = 1;
      }
    }
    else {
      bottom = 1;
    }
  }

  return head;
}

/* next_update updates cell. Letter is removed from */
/* supply, blank symbol is inserted and vdict and */
/* hdict are set the appropiate places. */

felt *next_update(felt *cell) {

  cell→supply = removeletter(cell→letter, cell→supply);
  cell→letter = '0';
  if (cell→E ≠ NULL) {
    cell→E→hdict = NULL;
  }
  if (cell→N ≠ NULL && empty(cell→N→letter)) {
    cell→vdict = NULL;
  }

  return cell;
}

/* reset_column_suppply resets the supplies below the */
/* cell org. */

felt *reset_column_supply(felt *org) {

  felt *cell;

  cell = org;

  while (cell ≠ NULL && !empty(cell→letter)) {
    cell→supply = getstock(cell→hdict, cell→vdict);
    cell = cell→S;
  }

  return org;
}

/* reset_column_supplies resets supplies in the South-West */
/* region. */

felt *reset_column_supplies(felt *org) {

  felt *cell;

  cell = org;
  cell→reset_num++;

  while (cell ≠ NULL) {
    cell = reset_column_supply(cell);
    cell = cell→W;
  }

  return org;
}

/* reset_row_supply resets supplies in the cells to the right */
/* of org. */

felt *reset_row_supply(felt *org) {

  felt *cell;

  cell = org;

  while (cell ≠ NULL && !empty(cell→letter)) {
    cell→supply = getstock(cell→hdict, cell→vdict);
    cell = cell→E;
  }

  return org;
}

/* reset_row_supplies resets the supplies in the North-East */
/* region. */
```

```
felt *reset_row_supplies(felt *org) {

   felt *cell;

   cell = org;
   cell→reset_num++;

   while (cell ≠ NULL) {
      cell = reset_row_supply(cell);
      cell = cell→N;
   }

   return org;
}

/* reset_supplies resets the supply North-East and South-West */
/* of origin. */

felt *reset_supplies(felt *origin) {

   felt *cell;
   int supply_num, reset_num;

   cell = origin;
   supply_num = count(cell→supply);

   if (cell→S ≠ NULL && cell→W ≠ NULL) {
      reset_num = cell→S→W→reset_num;
      if (supply_num ≥ reset_num) {
         cell = reset_column_supplies(cell→S→W);
      }
   }
   if (cell→N ≠ NULL && cell→E ≠ NULL) {
      reset_num = cell→N→E→reset_num;
      if (supply_num ≥ reset_num) {
         cell = reset_row_supplies(cell→N→E);
      }
   }

   return origin;
}

/* update_reset */

felt *update_reset(felt *cell) {

   if (cell→N ≠ NULL && cell→E ≠ NULL) {
      cell→N→E→reset_num = 0;
   }
   if (cell→S ≠ NULL && cell→W ≠ NULL) {
      cell→S→W→reset_num = 0;
   }

   return cell;

}

/* clean_update updates the grid. Removes letters etc. */

pit *clean_update(pit *head, felt *anchor, felt *origin) {

   int finished;
   felt *cell;

   finished = 0;
   cell = origin;

   while (!finished) {
      if (anchor == origin) {
         origin = next_update(origin);
         if (cell→number ≠ 1 || (cell→number == 1 && cell→supply
≠ NULL)) {
            head = addpit(head, cell);
         }
         if (anchor→S ≠ NULL) {
            head = clean_column(head, anchor→S);
         }
      }
      else {
         head = clean_column(head, anchor);
      }
      if (origin == anchor) {
         finished = 1;
      }
      if (anchor→W ≠ NULL) {
         anchor = anchor→W;
      }
   }

   return head;
}

felt *move_right(felt *anchor) {

   int finished;
```

```
   finished = 0;

   while (!finished) {
      if (anchor→E == NULL || empty(anchor→letter)) {
         finished = 1;
      }
      else {
         anchor = anchor→E;
      }
   }

   return anchor;

}

/* clean_up updates the head list and clears the grid */
/* patterns to be removed. */

pit *clean_up(pit *head, felt *cell) {

   felt *anchor, *origin;

   origin = anchor = cell;

   anchor = move_right(anchor);

   head = clean_update(head, anchor, origin);

   origin = reset_supplies(origin);

   return head;

}

/* This procedure is called when PREHARD WALK needs to */
/* backtrack. */

pit *hard_backtrack(pit *head, pit *first_head) {

   pit *dest_list;
   felt *dest;
   int multi;

   multi = 0;

   dest_list = destinations(head);
   if (dest_list == NULL) {
      dest = head→square;
   }
   else {
      dest = find_element(dest_list);
      if (dest == NULL) {
         multi = 1;
         head = hard_backtrack(dest_list, first_head);
      }
   }
   if (!multi) {
      head = clean_up(first_head, dest);
   }
   if (dest_list ≠ NULL) {
      dest_list = free_list(dest_list);
   }

   return head;
}
```

Appendix G

The Timer Module

> **Time flies like an arrow. Fruit flies like a banana.**
>
> *Lisa Grossman*

G.1 timebackup.h

```
/* This module contain a function to time cpu usage */

unsigned long measuretime();
```

G.2 timebackup.c

```
#include <sys/time.h>
#include <sys/resource.h>

#include "timebackup.h"

unsigned long measuretime() {

#ifdef RUSAGE_SELF
  struct rusage rusagev;

  getrusage(RUSAGE_SELF,&rusagev);
  return
rusagev.ru_utime.tv_sec*1000000+rusagev.ru_utime.tv_usec;

#else

  return clock()*1000000/CLOCKS_PER_SEC;

#endif
}
```

Appendix H

The Main Program

> **Real programmers can write assembly code in any language. :-)**
>
> *Larry Wall*

H.1

```
/* This file contain the main program. By changing */
/* the 'program' variable it is possible to switch */
/* between STRAIGHT WALK, SNAKE */
/* WALK, SWITCH WALK, SIKSAK WALK, */
/* and PREHARD WALK. */

#include <stdio.h>
#include <stdlib.h>

#include "Head.h"

#include "timebackup.h"
#include "Cross.h"
#include "Dict.h"
#include "backtrack.h"
#include "walkheurestics.h"

/* If user input should be used set 'userinput 1'. */

#define userinput 0

void main() {
  int height, width, back, finished, felttal, program,
        cont, more, num;
  long NumberOfSolutions, Nodes=0;
  felt *tag, *act;
  char dictionary[25];
  Ptegn *dict;
  pit *walk, *prewalk;
  long start, stop, diff;
  pit *head;

  strcpy(dictionary,"../../../Wordlist/prime.txt");

  NumberOfSolutions = 0;

  num = 0;

  /* Switch program here.      */

  program = STRAIGHTWALK;
  /* program = SNAKEWALK;       */
  /* program = SWITCHWALK;     */
  /* program = SIKSAKWALK;     */
  /* program = SLALOMWALK;     */

  /* program = PREHARDWALK;   */

  switch (program) {

  case STRAIGHTWALK:
    printf("\nWalkheuristic:  STRAIGHTWALK");
  break;
  case SNAKEWALK:
    printf("\nWalkheuristic:  SNAKEWALK");
  break;
  case PREHARDWALK:
    printf("\nWalkheuristic:  PREHARDWALK");
  break;
  case SWITCHWALK:
    printf("\nWalkheuristic:  SWITCHWALK");
  break;
  case SIKSAKWALK:
    printf("\nWalkheuristic:  SIKSAKWALK");
  break;
  case SLALOMWALK:
    printf("\nWalkheuristic:  SLALOMWALK");
  break;

  default:
    printf("\nMain:  Ukendt programtype!");
  }

/* Initialization and startup. */

  printf("\nDictionary:  %s",dictionary);
  printf("\nDictionary status:  building");
  start = measuretime();
  /* Start stop watch. */
  dict = makedict(dictionary);
  /* Set up dictionary. */
  stop = measuretime();
  /* Stop stop watch. */
  diff = stop - start;
  printf(" and done!");
  printf("\nDictionary creation time:  %ld", diff);

  /* Grid size is defined here. */

  if (userinput) {
    printf("\nAngiv højde (max 30) :  ");
    scanf("%i", &height);
    printf("\n\nAngiv bredde (max 30):  ");
    scanf("%i", &width);
  }
  else {
    height = hojde;
    width = bredde;
  }

  printf("\nGridsize:  %ix%i\n\n",height,width);

  felttal = 0;

  /* Main */

  if (checkheight(height) && checkwidth(width)) {
    act = tag = makegrid(height,width);

    /* Insert grid pattern here */

    /* include "../../Grids/grid3" */

    /* ----------*/

    printf("Start grid konfig:  \n\n");
    printgrid(tag);
    /* Print empty grid. */

    printf("\nMarking wordslots");
    markwords(tag, dict);
    /* All wordslots are set. */
    printf(" Done!");

    printf("\nStarting stop watch\n");
    start = measuretime();
    /* Start stop watch. */

    switch (program) {

    case SNAKEWALK: case STRAIGHTWALK: case
SWITCHWALK: case SIKSAKWALK: case SLALOMWALK:
      /* Predefined walk is worked out. */
      finished = 0;
      back = 1;
      cont = 0;
      if (program == SNAKEWALK) {
        walk = makewalk(act, SNAKEWALK);
      }
      if (program == STRAIGHTWALK) {
        walk = makewalk(act, STRAIGHTWALK);
      }
      if (program == SWITCHWALK) {
        walk = makewalk(act, SWITCHWALK);
      }
      if (program == SIKSAKWALK) {
        walk = makewalk(act, SIKSAKWALK);
      }
      if (program == SLALOMWALK) {
        walk = makewalk(act, SLALOMWALK);
      }
      act = walk→square;

      /* Main loop. */

      while (!finished) {
        /* printgrid(tag); */
        setdict(&act);
        if (!cont) {
          act→supply = getstock(act→hdict, act→vdict);
        }
        cont = 0;
        if (act→supply == NULL && act→letter ≠ '/') {
          /*  printgrid(tag);  */
          back = Backtrack(&act, &walk, program);
        }
        if (!back) {
          finished = 1;
        }
        else {
          prewalk = walk;
          setletter(&act, &walk);
        }

        if (prewalk→next == NULL) {

          if (act→supply == NULL) {
            printgrid(tag);
          }
          else {
            while (act→supply ≠ NULL) {
              printgrid(tag);
              NumberOfSolutions++;
              act→supply = removeletter(act→letter,
act→supply);
              act→letter = choose(act→supply);
              cont = 1;
            }
          }
          if (act→letter ≠ '/') {
            act→letter = '0';
          }
        }
      }

      break;

    case PREHARDWALK:

      finished = 0;
      more = 1;

      head = walkhardinit(tag);
      /* Potential cells are found. */

      /* Main loop. */

      num = 0;

      while (more) {

        while (!finished) {
          if (head == NULL) {
            finished = 1;
            if (tag→supply == NULL) {
              more = 0;
            }
          }
          else {
            act = Walkhard(head);
            /* act is NULL if no optimal cell exist */
            if (act == NULL) {
              printgrid(tag);
              head = hard_backtrack(head, head);
            }
            else {
              insert_letter(act);
              act = update_reset(act);
              head = updatehead(head, act);
            }
          }
          num++;
        }

        finished = 0;

        if (more) {
          head = addpit(head, act);
          head→antal = 0;
          while (act→supply ≠ NULL) {
            act→supply = removeletter(act→letter, act→supply);
            act→letter = choose(act→supply);
            NumberOfSolutions++;
            printgrid(tag);
          }
        }

        if (act ≠ NULL) {
          act→letter = '0';
        }
        if (num > 100) {
          more = 0;
        }
      }

      break;

    default:
      printf("\nMain:  Ukendt programtype!");
    }
  }
  else {
    printf("Det maksimale grid måler %ix%i!", MAXH, MAXB);
  }

  stop = measuretime();
  /* Stop stop watch. */
  printf("\nStop watch stopped\n\n");

  diff = stop - start;
  /* Calculate cpu time. */

  printf("\nTime :  %ld", diff);
  /* Time result. */

  printf("\nNodes:  %i", Nodes);

  printf("\n\nNumber of solutions:  %i", NumberOfSolutions);
  printf("\nAvarage building time:  %ld\n\n",
diff/NumberOfSolutions);
}
```