A SURVEY ON DICHOTOMOUS SEARCH AND RELATED PROBLEMS

Thesis submitted towards the degree of Master of Science in Operations Research

Anna Sarid

June 2013

Supervisor:
Prof. Rafi Hassin

Tel-Aviv University
The Raymond and Beverly Sackler Faculty of Exact Sciences
The Department of Statistics and Operations Research

A SURVEY ON DICHOTOMOUS SEARCH AND RELATED PROBLEMS

ANNA SARID

Abstract. An object is searched for in an integer interval of length N. Queries for the object are sequentially conducted. Each query reveals whether the object lies to the left or to the right of the searched location. The objective is to find the object within minimal expected number of queries. This problem is called the "dichotomous search" problem and has various versions and formulations. In this paper a survey of dichotomous search problems is conducted.

Acknowledgements. I would like to thank Prof. Rafi Hassin for his great help on all aspects of research and writing.

Dedication. I would like to dedicate this thesis to my dear husband Adi and my sweet son Eitan.

Date: June 16, 2013.

Table of Contents

1. Introduction

We consider a family of search problems called "dichotomous search problems". The simplest form of a dichotomous search problem can be formulated as follows: An object to be searched for lies at location x and is represented by an integer in an interval of length N. Queries for the object are sequentially conducted. Each query returns whether or not the object lies to the left of the point. Using this information the search converges to the location of the object. The objective is to find x after minimal expected number of queries.

The basic problem and various versions of it have been widely studied in the last fifty years. At first those problems were investigated out of pure mathematical interest (see [25, 130, 101, 102]). Later the problem was reformulated in terms of alphabetic binary search trees to be used in computer science for efficiently retrieving information from storage places. The first article to do that was [87] (1971). Simultaneously the basic problem was again reformulated in the language of game theory, applied to optimization of wage bargaining and gathering commercial information from inventories (see [7, 8, 9, 114, 116, 117]) and provided a basis for optimizing quality control of production lines ([60, 111] etc.). The problem has many application in coding theory, where it is equivalent to prefix-free alphabetic code problem.

The literature on the topic of this survey comes from several disciplines - computer science, applied mathematics, operations research, statistics, industrial engineering and economics. Figure 1[1] illustrates the distribution of the articles on this subject over the different disciplines. The research on dichotomous search problems began in the 50'th and continues to this day, as can be seen from figure 2. The objective of this paper is to provide a comprehensive survey of publications in the area of dichotomous search, on its different variations and applications.

The survey is organized as follows:

§2 provides the necessary definitions to introduce the dichotomous search problem in terms of graph theory, such that a search policy resembles building a binary alphabetic tree with minimal weighted path length; and in terms of coding theory as an alphabetic prefix-free code. Throughout the survey we freely switch between the formulations.

A comprehensive survey of the basic alphabetic binary tree problem with minimal path length is the subject of §3. We describe in general terms the basic ideas of some fundamental results, starting from an $O(N^3)$ (in terms of computational complexity) algorithm of Knuth, through $O(N \log N)$-time algorithms of Hu-Tucker and of Garsia-Wachs, to linear time algorithms for special cases.

[1]Next to each discipline, it is shown how many publications from it are included in this survey, by an absolute number and by a percentage of all articles in the survey.

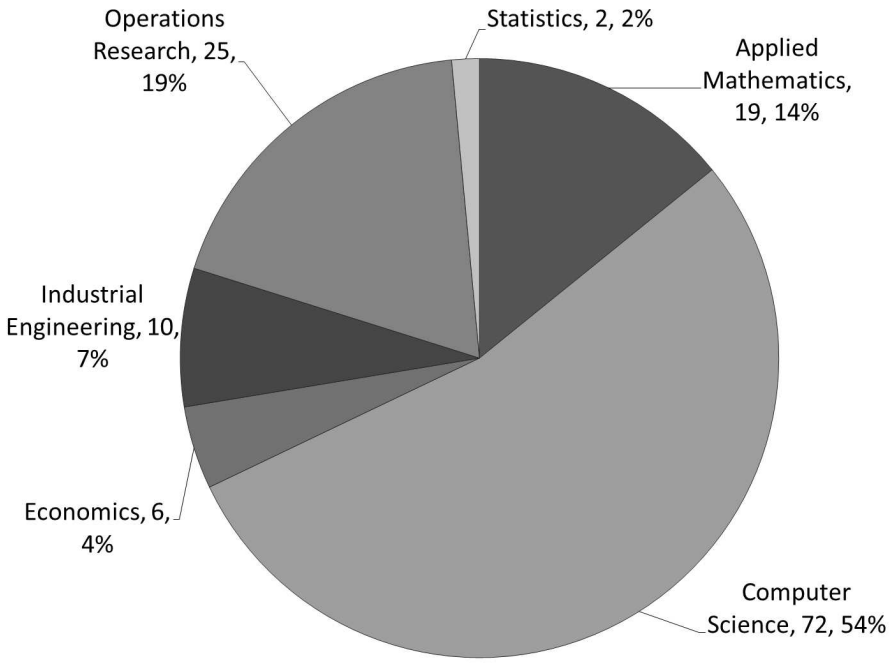

FIGURE 1. The number of publications on dichotomous search in different disciplines.

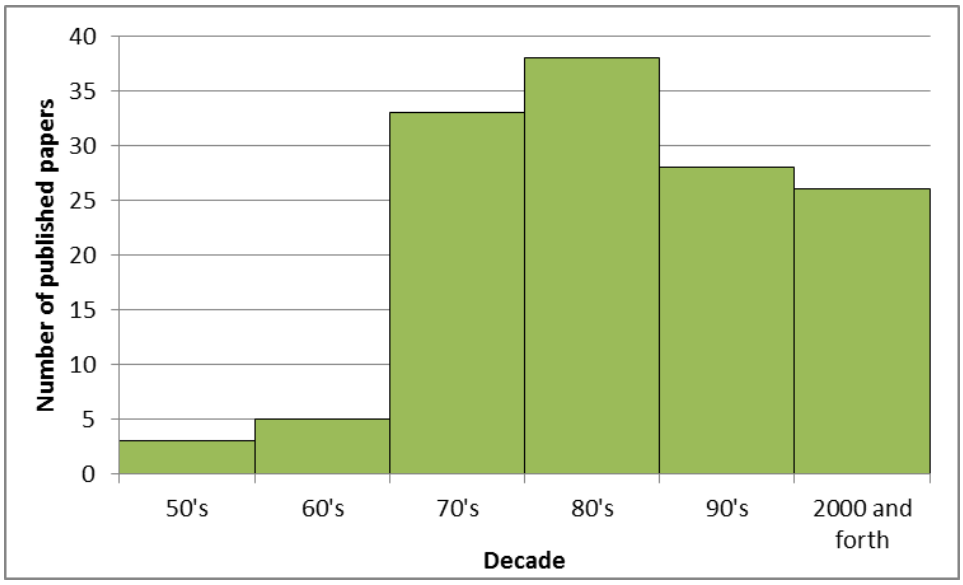

FIGURE 2. The number of publications on dichotomous search over decades.

In some variations of the alphabetic binary tree problem the computational complexity can be cut using what is called a monotonicity property or

"Knuth method", first introduced by Knuth [87]. §4 delimits the search problems, to which this property is applicable.

When an a priori distribution of the location of the object is known, additional solution methods are available. §5 explores the dichotomous search problem with a uniform a priori distribution and a more general distribution $p_k = ck^{-a}$.

Sometimes real life situations arise variations on the basic dichotomous search problem. For example, travel time between two subsequent queries may also have its cost, thus encouraging the searcher to travel smaller distances. At another example, traveling to the right costs more than traveling to the left, as it happens when heavy equipment, looking for an appropriate ground to build a tunnel, has more expenses moving uphill than downhill. Those situations can be formulated as dichotomous search problems with appropriate corrections in the objective function and costs. They are the topic of §6.

§7 surveys the literature on the restricted dichotomous search problem, where the number of queries is a priori limited, or equivalently the alphabetic tree's depth must be limited and the objective remains to minimize the expected cost of the search within this limitation.

§8 describes optimal policies in a dichotomous search problem where answers are unreliable - the searcher may be told that the object searched for is to his left, while in fact it is to his right. Such misleading information may cost the searcher additional resources, but being a priori aware of its possibility allows the searcher to build an optimal policy.

The dichotomous search problem has been generalized to various "multiplicity" problems. §9 discusses the problems of searching for several objects at a time; searching for one object, but by several parallel queries; and searching for a vector among lexicographically ordered set of vectors.

In §10 we shall see several generalized search problems, of which both the alphabetic tree problem and the Huffman tree problem are special cases.

§11, describes a specific problem of searching through rationals rather than natural numbers.

Dichotomous search problems can also be viewed as games of two players: a searcher and a hider. The hider wishes to hide the object well enough to enlarge the searcher's effort to find it, while the searcher wishes to find the object with minimal effort. Such a formulation allows the application of tools from game theory to solving dichotomous search problems. That is the topic of §12.

In §13 the following problem is discussed: Suppose a machine that produces items sequentially in a production line is subject to random failures. Once a failure occurred while producing a certain item, that item is flawed and so are all the following items until the failure is discovered by quality control. We are given a batch of produced items, such that the first is satisfactory and the last is flawed. The objective is to find the first flawed item, whose

location has a known distribution which is commonly geometric, using minimal expected number of queries, in order to dispose of all the unsatisfactory production.

§14 discusses two applications of dichotomous search games to the world of economics: "How to make an optimal wage request while applying for a new job?" and "How to optimize the scope of production by gathering information from inventories?".

Throughout the survey $\log N$ denotes logarithm with the basis of 2, unless stated otherwise.

2. Definitions and formulations of the dichotomous search problem

2.1 Formulation of dichotomous search in terms of binary trees

A t-ary tree is a rooted tree in which every node has t or zero edges emanating downward from it and every node, except the root, is met by an edge from an upper node. A 2-tree is also called a *binary tree*. Nodes having no edges emanating downward from them are called *terminal nodes* or *leaves* or *external nodes*. A terminal node has no children, while the other nodes, called *internal nodes*, have two nodes as their children. The number of internal nodes is always one less than the number of terminal nodes.

The *path length* of a node is the length (number of arcs) of the unique path from the root to that node. The path length of the node is sometimes also called *the depth of the node* or the *level of the node* and is denoted by $l(j)$, where $l(root) = 0$. The *depth of a tree* is the maximal depth of its node, among all nodes.

One can easily prove that in a 2-tree with N terminal nodes with depths $l(1), \ldots, l(N)$,

$$\sum_{j=1}^{N} \left(\frac{1}{2}\right)^{l(j)} = 1. \tag{2.1}$$

A node j is called an *ancestor* of a node i at a higher level if the path from the root to i passes through j. In many problems the terminal node j has associated with it a positive weight w_j. The *weighted path length of a tree* T, denoted by $|T|$, is defined to be $\sum_{i=1}^{N} w_j l(j)$. In problems where there are no weights associated with the terminal nodes, the *path length of the tree* is defined as $\sum l(j)$.

Binary alphabetic trees. In some problems we are interested in binary trees where the terminal nodes $1, 2, ..., N$ appear from left to right consecutively. Such binary trees are called *alphabetic trees*. A binary alphabetic tree are of interest, because such a tree represents a strategy for the solution of the dichotomous search problem. Say we are searching for a point among $1, 2, ..., 5$. The tree in figure 3 represents the following strategy: First search at 3. If the object lies among $1, 2, 3$ then search at 2. Otherwise search at

4. After at most three stages the object will be found. The path length of node x is the number of queries needed to find the object using that search policy, given that the object lies in x. Thus the depth of the binary tree corresponds to the maximal number of questions needed to find the object (three in our example) and the (weighted) path length of the tree is N times the (weighted) average number of queries needed to find the object wherever it is hidden.

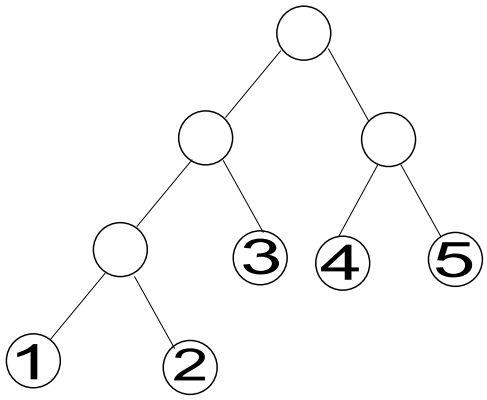

FIGURE 3. A 2-tree is a search strategy

A binary alphabetic tree with minimum weighted path length is an *optimal binary alphabetic tree*. The weight w_i of a terminal node is associated with the a priori probability $p_i = \frac{w_i}{\sum_{i=1}^{N} w_i}$ of the object to be located at the corresponding location i. An optimal binary tree is therefore associated with an optimal search strategy given the a priori location distribution $p_1, p_2, ..., p_N$.

Binary Huffman trees. A binary Huffman tree is also a binary tree with weights $w_1, w_2, \ldots, w_N$ attached to leaves $1, 2, \ldots, N$, only that in contrast to alphabetic binary trees, the leaves do not necessarily appear in order $1, 2, \ldots, N$ from left to right. An *optimal binary Huffman tree* is a binary Huffman tree with minimal weighted path length. A simple algorithm for the optimization problem was found by **Huffman** [74] **(1952)**: First find two nodes with smallest weights, say w_1 and w_2. Then combine these nodes (they have a common parent). Next replace the subtree formed by the two nodes and their parent by a new terminal node having weight $w_1 + w_2$ and repeat the same procedure on the reduced problem of $N - 1$ terminal nodes with weights $w_1 + w_2, w_3, ..., w_N$. The process of letting two nodes have a common parent will from now on be reffered to as *combining* the two nodes. Continue until the tree is formed.

Binary search trees. A binary search tree is a data structure for retrieving objects associated with keys. We are given n names $A_1, \ldots, A_n$ and $2n + 1$ frequencies $\beta_1, \ldots, \beta_n, \alpha_0, \alpha_1, \ldots, \alpha_n$ with $\sum \beta_i + \sum \alpha_i = 1$. β_i is the frequency of encountering A_i, and α_j is the frequency of encountering a name which lies between A_j and A_{j+1} for $j = 1, \ldots, n - 1$. α_0 (α_n) is the frequency of encountering a name which lies before A_1 (after A_n) A

binary search tree for the names $A_1, \ldots, A_n$ is a tree with n interior nodes labeled A_i in increasing order from left to right and $n+1$ leaves labeled by intervals (A_j, A_{j+1}) in increasing order from left to right. Let b_i be the distance (number of arcs) of interior node A_i to the root, and let a_j be the distance of leaf (A_j, A_{j+1}) to the root. The *weighted path length* of T, i.e., the expected number of comparisons needed to locate a name is $\sum_{i=1}^{n} \beta_i(b_i+1) + \sum_{j=0}^{n} \alpha_j a_j$. A binary search tree is optimal if it has minimal weighted path length. Note that the optimal alphabetic binary tree problem is just a special case of the optimal binary search tree problem, letting $\beta_1 = \beta_2 = \cdots = \beta_n = 0$.

Throughout the survey we will be interested in alphabetic binary trees and binary search trees, as results about binary search trees directly imply results on alphabetic binary trees. Search for the optimal Huffman tree is beyond the scope of this survey (except for the elegant Huffman algorithm which is described below).

2.2 A prefix-free code formulation of the dichotomous search problem

Let $\Sigma = \{\sigma_1, \ldots, \sigma_t\}$ be an *encoding alphabet*. Let Σ^* represent all finite words written using Σ. Word $v \in \Sigma^*$ is a *prefix* of word $v' \in \Sigma^*$ if $v' = vu$ where $v' \neq v$. A *code* over Σ is a collection of words $C = \{v_1, \ldots, v_n\}$. Code C is *prefix-free* if for all $i \neq j$ v_j is not a prefix of v_i. Let $cost(v)$ be the *length* or number of characters in v. Given a set of associated probabilities $p_1, \ldots, p_n$, the cost of the code is $\sum_{i=1}^{n} p_i cost(v_i)$. The *prefix-free coding problem*, sometimes known as the *Huffman encoding problem* is to find a prefix-free code over Σ of minimum cost. There is equivalence of the Huffman encoding problem to the minimum weighted path-length Huffman t-ary tree problem on n leaves, where $cost(v_i)$ is the path length of node v_i, p_i is the weight attached to node v_i and the objective is to find a minimum weighted path length t-ary tree. For $t = 2$ this is the Huffman binary tree problem. The letters $\{\sigma_1, \ldots, \sigma_t\}$ can be thought of as t children of the root. Then, every internal node also has children named $\{\sigma_1, \ldots, \sigma_t\}$. Traveling from the root to a leaf v_i is equivalent to collecting letters from Σ to build the word v_i. The prefix-free property means, in terms of t-ary trees, that the words are located at the leaves only, thus the combinations of letters created in the internal nodes are non-words.

Alphabetic coding is the same problem with an additional constraint: the codewords must be chosen in increasing alphabetic order (with respect to the words to be encoded). This corresponds to the problem of constructing optimal (with respect to average search time) search trees for items with the given access probabilities or frequencies. Thus the alphabetic coding problem is equivalent to the alphabetic t-ary tree problem.

3. The optimal alphabetic binary tree problem

3.1 Towards an $O(N \log N)$ algorithm

As stated in §2, the alphabetic binary tree is a special case of the binary search tree. Let N be the number of terminal nodes. The first algorithm running in time polynomial in N for this special case is given by **Gilbert and Moore** [48] **(1958)**. It requires $O(N^3)$ time and $O(N^2)$ space and uses dynamic programming. This method is extended by **Knuth** [87] **(1971)** to include the more general case where the successful and unsuccessful search probabilities are both taken into account. Knuth's extension also runs in time $O(N^3)$, but he also shows that by refining the method the running time can be cut to order $O(N^2)$. Dynamic programming is applicable to this problem since all subtrees of an optimal tree are optimal. We do not elaborate on Gilbert and Moore's algorithm, since the algorithm of Knuth, described below, (without the cut of running time) includes it.

Recall the definitions of $A_1, ..., A_N,\ \alpha_0, \alpha_1, ..., \alpha_N,\ \beta_1, ..., \beta_N$ from §2. Let $|T|_{i,j}$ denote the weighted path length of an optimal search tree for all words lying between A_i and A_{j+1} (including A_i and A_{j+1}), when $i \leq j$; let $W_{i,j} = \alpha_i + \beta_{i+1} + \alpha_{i+1} + \beta_{i+2} + \cdots + \beta_{j-1} + \alpha_j$; and let R_{ij} denote the label of the root of this tree, where $i < j$. The following formulas determine the algorithm:

$$\begin{aligned} &|T|_{i,i} = W_{i,i} = \alpha_i \text{ for } 0 \leq i \leq n, \\ &|T|_{i,j} = W_{i,j} + \min_{i<k\leq j}(|T|_{i,k-1} + |T|_{k,j}) \text{ for } 0 \leq i < j \leq n. \end{aligned} \tag{3.1}$$

The complexity of the algorithm defined by (3.1) is $O(N^3)$. Refinement of the algorithm is achieved in [87] using the result that there is always a solution to (3.1) satisfying $R_{i,j-1} \leq R_{i,j}$ and $R_{i,j} \leq R_{i+1,j}$ for $0 \leq i < j - 1 < n$. This result allows to fasten the algorithm, since we usually will not have to search the entire range $i < k \leq j$ when determining $R_{i,j}$. In fact, only $R_{i+1,j} - R_{i,j-1} + 1$ values need to be examined when $R_{i,j}$ is calculated, and the calculations are conducted in growing order of $j - i$. Summing for fixed $j - i$ gives a telescoping series, therefore the total amount of work is $O(N^2)$. This method of reduction the complexity of dynamic programming solution is called *"the telescopic method"* or *"Knuth method"*.

In [71] **Hu and Tucker (1971)** present an $O(N \log N)$ algorithm for constructing an optimal alphabetic tree. Any alphabetic tree can be built as follows: Start with an initial sequence of terminal nodes, also called *square nodes*, $V_1, ..., V_N$; Combine some adjacent pair of nodes V_i, V_{i+1} and form a new sequence: $V_1, V_2, ..., V_{i-1}, v_{(i,i+1)}, V_{i+2}, ..., V_N$, where $v_{(i,i+1)}$ is an internal node, also called *a round node*, and the other nodes are still terminal; now combine some adjacent pair in this new sequence and replace the combined pair by a new internal node in the sequence representing their parent in the tree which the algorithm constructs; and so on. The intermediate sequences and the initial sequence are called *construction sequences*. To

form an alphabetic tree $N-1$ combinations are needed. When combining two nonadjacent nodes in a construction sequence, let the parent take the position of its left child in the resultant construction sequence (and of course the right child no longer appears). Two nodes in a given construction sequence are called *tenative-connecting* (T-C for short) if the sequence of nodes between the two nodes is either empty or consists entirely of round nodes.

The T-C algorithm, presented in [71] has two parts: The first part constructs a tree T' which does not satisfy the ordering restriction: In each successive construction sequence (starting with the initial sequence), it combines the pair of T-C nodes with the minimum sum of weights (that pair of nodes is called a *lmcp - local minimum compatible pair*). Figure 4 illustrates the first part of the T-C algorithm (the initial weights are in the square nodes). The algorithm starts by combining the lmcp 1 and 3, the rightmost pair of square nodes, to get a new round node with weight 4. It proceeds to get round nodes with weights $5, 9, 12, 13, 17, 25, 37, 62$. The parents are placed over their left children to indicate their position in the successive construction sequences. In figure 4, the left part illustrates the construction of tree T', and in the right part T' is presented in its proper way to emphasize that the ordering property is not preserved through the construction.

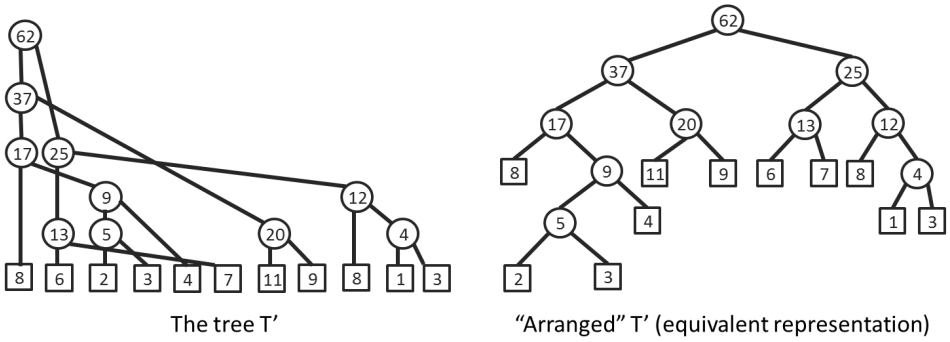

FIGURE 4. T' and "arranged" T'

In the second part T' is converted into an alphabetic tree T'_N with the same cost. For that the following definitions are established:

- A *T-C tree* built on a given initial sequence is a tree which can be built up in successive construction sequences such that each successively combined pair of nodes is T-C in its construction sequence. T' is a T-C tree.
- For a given T-C tree, a *T-C level-by-level construction* of it combines all nodes on the highest level (the farthest level from the root) first, then all nodes on the next-to-highest level, and so on. For example, if the T-C tree is as shown in figure 4, a T-C level-by-level construction would create the internal nodes in the following order: $5, 9, 4, 17, 13, 20, 12, 37, 25$. Those T-C trees for which the T-C level-by-level construction is possible are called *T-C level-by-level trees*.

- A *T-C forest* and *T-C level-by-level forest* are similarly defined.

Let $C(S)$ be the class of all level-by-level forests. Obviously all alphabetic forests are in $C(S)$. It is shown that for every forest T in $C(S)$, there is an alphabetic forest T_N of the same cost. Then by proving that $T' \in C(S)$ and that it is optimal in this class, there exists an alphabetic tree T'_N of the same cost as T'. T_N is obtained from T as follows: Let level q be the highest level of a leaf in T. There is an even number of nodes, say $2k$, on this level. Reassign the parent-child relationships between the k parents on level $q-1$ and the $2k$ children on level q so that the leftmost parent has as children the two leftmost nodes on level q, according to the initial ordering of the nodes and so on. The key fact is that this reassignment does not change the path length of any terminal node. By repeating the reassignment on successively lower levels of T, the tree T_N is obtained.

Figure 5 illustrates the alphabetic tree T'_N resulting from the tree T', presented in figure 4, by the second part of the algorithm. Note that T'_N is alphabetic and the level of each leaf of T'_N equals its level in T'. Thus the cost of T'_N equals the cost of T'.

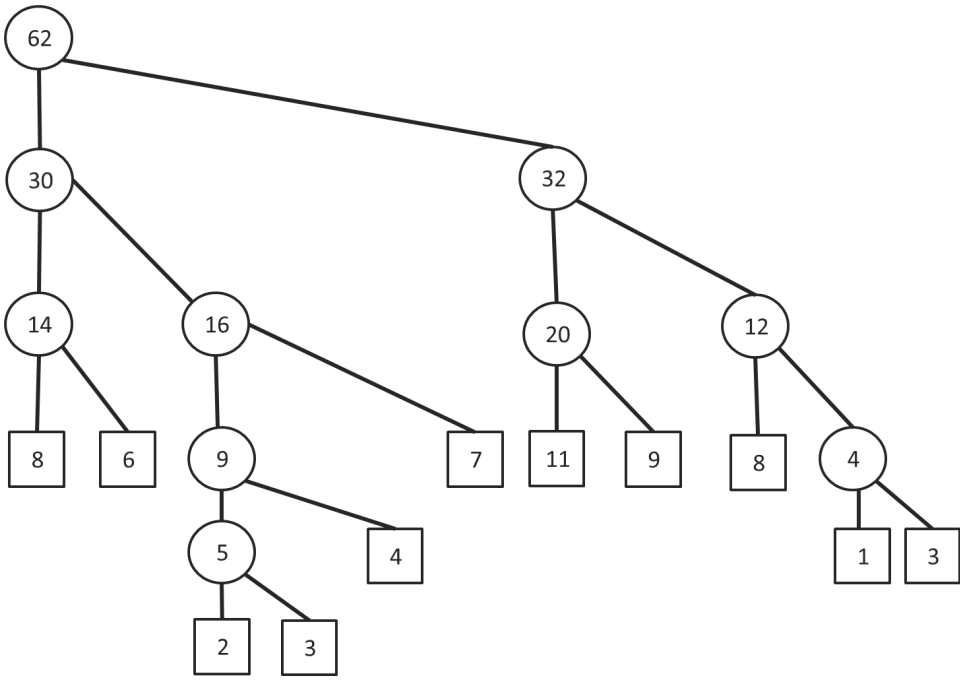

FIGURE 5. The tree T'_N

The tree T'_N is an optimal alphabetic binary tree. The T-C algorithm needs $O(N \log N)$ operations. This result is the best known for arbitrary weights (i.e., arbitrary probability distribution of the location of the object).

In a subsequent work [70] **Hu and Tan (1972)** show that when the terminal nodes have monotonically increasing weights from left to right, the T-C algorithm for optimal alphabetic binary trees actually coincides with the Huffman's algorithm for optimal binary non-alphabetic trees. Further, in [66]**, Hu (1973)** provides another, simpler, proof of the T-C algorithm.

In **1977 Garsia and Wachs** [45] propose an algorithm, closely related to the Hu and Tucker algorithm in [71], providing a way to compute an optimal alphabetic binary tree. Both the complexity and the algorithm are similar to that at [71], therefore it is not covered thoroughly here. **Kingston** [82] **(1988)** provides another proof of correctness of the Garsia-Wachs algorithm.

In [80] **(1996) Karpinski, Larmore and Rytter** present several new observations which lead to new correctness proofs of the the two known algorithms (Hu-Tucker and Garsia-Wachs) for construction of optimal alphabetic trees. A generalized version of the Garsia-Wachs algorithm is given, that permits any non-negative weights, as opposed to strictly positive weights required in the original Garsia-Wachs algorithm. We do not elaborate on this article.

3.2 $o(N \log N)$ algorithms for special cases

In [84] **(1995) Klawe and Mumey** make an attempt to improve the $O(N \log N)$ complexity of the optimal alphabetic binary tree problem. They show that any natural method employing either the idea of construction *lmcp-tree* (see definition below), as Hu-Tucker and Garsia-Wachs, or the idea of region processing, presented in their paper, may force us to sort a substantial amount of input. Thus the basic question of whether there is a general $o(N \log N)$-algorithm for finding optimal alphabetic binary trees remains open. Yet they introduce an idea for finding optimal alphabetic trees which they refer to as *region processing* and use this method to produce $O(N)$-time algorithms when all weights $\{w_1, \ldots, w_N\}$ are within a constant factor of one another (i.e., $\max\left\{\frac{w_i}{w_j}\right\} < \sigma$ for some constant σ) and when they are exponentially separated - an input weight sequence $w_1, w_2, \ldots, w_N$ is defined to be *exponentially separated* if there exists a constant C such that $|\{i \,|\lfloor \log w_i = k \rfloor\}| < C$ for all $k \in \mathbb{Z}$.

Recall the Hu-Tucker algorithm and definitions: the Hu-Tucker algorithm begins with a list of leaf nodes containing the weights $w_1, \ldots, w_N$ in that order. This list is called the *the initial construction sequence* and is used to determine how the nodes combine to form the intermediate tree. Nodes in the worklist are designated either as *square* or *round*. This affects the way the nodes may combine. Initially all nodes are square. The parent resulting from the combination of two nodes is called round and assigned a weight that equals the sum of the weights of its children. The combined nodes are removed from the worklist and the new parent node occupies the former position of its left child. The authors repeat the definition of lmcp from the Hu-Tucker algorithm using new terms: A compatible (also called

"tenatively-connected") pair of nodes v_a, v_b is said to be *local minimum compatible pair (lmcp)* if:

(1) $w_b \leq w_x$ for all nodes v_x compatible with node v_a.

(2) $w_a \leq w_y$ for all nodes v_y compatible with node v_b.

To obtain the *lmcp tree*, keep combining the local minimum compatible pairs. The result is the intermediate tree (T' in [71]) which is not alphabetic, but whose leaves have the same levels as the optimal alphabetic tree.

In [84], the authors classify the input weights w_i according to their order of magnitude, base 2. The *category* of a node of weight w is $\lfloor \log w \rfloor$. A maximal length sequence in the worklist of weights with the same category is called a *region*. By keeping a stack of regions and considering only regions whose adjacent regions have a higher category, the attention is restricted only to the node combinations occurring within these regions. This is called *region-processing*.

Using region-processing the authors prove that there is a linear-time algorithm for finding an optimal alphabetic tree on a sequence of input weights which differ at most by a constant factor. The authors note that for $\sigma = 2$ Garsia and Wachs [45] also gave a linear-time algorithm.

When the weights are exponentially separated, an $O(N)$ algorithm is also given for constructing an optimal alphabetic tree, again using the idea of region processing. It is observed that there are at most $2C$ nodes (recall C from the definition of exponentionally separated weights) in any region processed. It is shown that every region of size r can be processed in $O(r)$ time and use their region processing method to construct the lmcp tree in $O(N)$ time. Once the lmcp tree is constructed, the second part of the H-T algorithm constructs the optimal alphabetic tree in $O(N)$ time, thus the overall complexity is $O(N)$.

Hu and Morgenthaler [69] **(1996)** discuss special conditions on the weights of the leaves where the alphabetic binary tree can be built in linear time. They divide the Hu-Tucker algorithm into three steps:

(1) Combination. Keep combining lmcp until a tree T is obtained.

(2) Level Assignment. Find the level number of every leaf in tree T, say $l(1), \ldots, l(N)$.

(3) Reconstruction. Use a stack algorithm to construct an alphabetic binary tree based on $l(1), \ldots, l(N)$.

Steps 2 and 3 take linear time, while step 1 takes $O(N \log N)$ time. Several definitions are needed to describe the main results of [69]:

- A weight sequence is *increasing* if $w_1 \leq w_2 \leq \cdots \leq w_N$.
- A weight sequence is *decreasing* if $w_1 \geq w_2 \geq \cdots \geq w_N$.
- A weight sequence is a *valley* if $w_1 \leq w_2 \cdots \leq w_i \geq w_{i+1} \geq \ldots w_N$.

- A weight sequence is *bimonotonal increasing* if $w_1 + w_2 \leq w_2 + w_3 \leq \cdots \leq w_{N_1} + w_N$.
- A weight sequence is *bimonotonal decreasing* if $w_1 + w_2 \geq w_2 + w_3 \geq \cdots \geq w_{N_1} + w_N$.
- A weight sequence is a *bimonotonal valley* if it contains a single lmcp.

The authors prove that if the leaves form an increasing sequence or a decreasing sequence, the combination phase of the Hu-Tucker algorithm can be performed in linear time by queuing the round nodes as they are created. Compatible round nodes are placed at the end of a sorted queue as they are formed. Only the bottom two nodes of this queue need to be examined to determine the next lmcp.

For the bimonotal increasing sequence the combination phase can also be performed in linear time, since the bimonotal condition guarantees that the leftmost square (external) node will always be smaller than the node two to the right. For a bimonotonal valley sequence consisting of a bimonotonal decreasing sequence followed by a bimonotonal increasing sequence the combination phase is still linear, since we are guaranteed a single queue of round nodes by the presence of a single initial lmcp.

Any weight sequence can be broken into a set of bimonotonal valley sequences in linear time. The only thing that prevents the modified algorithm from running in linear time for an arbitrary weight sequence is the need to "merge" two queues when the mountaintop (a notion defined in the paper) between their valleys combines. Each merge requires linear time, but as $O(N)$ queues could be merged at one time, the merging may require $O(N \log N)$ time.

In [95] **(1998) Larmore and Przytycka** study the complexity of the optimal alphabetic binary tree problem relative to the complexity of sorting and priority queue operations. They present an $O(N \log P(k))$-time algorithm for the general optimal alphabetic binary tree problem and an $O(N\sqrt{\log k})$-time algorithm for the integer optimal alphabetic binary tree problem (in the integer optimal alphabetic binary tree problem, the weights $\{w_i\}$ are all integers), where k is a number bounded above by the minimal leaf weight and $P(k)$ is the time complexity of priority queue insert/delete - min operation. This, depending on the computational model, gives an $O(N \log k)$ or $O(N \log \log N)$-time algorithm. For the integer case, this algorithm combined with the Johnson integer priority queue data structure leads to an $O(N \log \log W)$-time algorithm for the integer optimal alphabetic binary tree problem, where $W = \sum_{i=1}^{N} w_i$.

The authors also relate the complexity of the optimal alphabetic binary tree problem to the complexity of sorting. They show that on the standard comparison model the problem reduces to $O(\sqrt{\log k})$ instances of sorting. This, for the integer case leads to $O(N\sqrt{\log k})$ optimal alphabetic binary tree algorithm. For an arbitrary k this is worse than the $O(N \log \log N)$-time priority queue approach, but it provides an improvement for small values of k.

Further progress in the field of efficient computation of optimal alphabetic trees we find in [68] **(2005) by Hu, Larmore and Morgenthaler**. They improve the implementation of the Hu-Tucker algorithm [71] for the optimal alphabetic binary tree problem, using an additional assumption that weights can be sorted in linear time, for example if weights are all integers in a small range. Their result is an $O(N)$ time algorithm for that special case. Thus, the overall complexity of the integer algorithm given in [95] is reduced from $O(N\sqrt{\log N})$ to linear.

The original Hu- Tucker algorithm takes $O(N \log N)$ to construct the lmcp tree (T'), because it requires $O(\log N)$ time to find the minimum compatible pair and update the structure, and this action must be done $N-1$ times. The authors in [68] improve the $O(\log N)$ complexity of finding an lmcp to $O(1)$, provided all weights are integers in the range $0, \ldots, N^{O(1)}$. We do not elaborate here on the implementation details.

We shall see in §13 that in the special case when the leaves have weights equal to the geometric probabilities (i.e., the dichotomous search problem with a proiri geometric distribution), that there exists an $O(N)$ algorithms that finds an optimal solution [54].

3.3 Applications of optimal alphabetic trees

Garey and Hwang [44] **(1974)** investigate group-testing procedures that isolate a single defective item within a finite set of items $I = \{I_1, \ldots, I_N\}$ with a minimum expected number of tests. They prove that an optimal procedure can be found by constructing an optimal alphabetic tree. The item I_i is defective with an a priori probability p_i. A "group test" is a test of any set of items, which either determines that all members of that set are non defective or that the set contains at least one defective item.

Any group-testing procedure which isolates a single defective from I can be specified with a labeled binary tree called a *binary testing tree*, which has the following structure:

(1) "Each nonterminal node is labeled with a subset of I, denoting a particular group test, and has its left arc labeled D for "defective" and its right arc labeled N for "non defective".

(2) Each terminal node is labeled with a single item from I which has the property that whenever the sequence of group tests on the path from the initial node (or "root") to that terminal node is performed, with outcome identical to the sequence of arc labels on that path, then that item is certainly defective."

The testing procedure described by such a binary testing tree is as follows: perform the test labeling the initial node, apply next the group test labeling the node obtained by following the arc corresponding to the most recent test result, stop upon reaching a terminal node. The item labeling that terminal node is isolated as defective. The path length of a terminal node in a binary testing tree is simply the number of arcs occurring in the path from

the initial node to that terminal node. Given a binary testing tree, if $l(i)$ denotes the path length of the ith terminal node and if w_i is the probability of reaching that node when applying the test procedure, then the expected number of tests required by the procedure is given by the total weighted path length, $\sum_i l(i)w_i$.

The authors show that an optimal testing procedure for the problem can be obtained by ordering the items so that $p_1 \geq p_2 \geq \cdots \geq p_N$ and constructing an optimal alphabetic binary tree for the sequence of weights $w_1, w_2, \ldots, w_N$ with $w_i = \left(1 - \prod_{j=1}^{N} [1 - p_j]\right)^{-1} p_i \prod_{j=1}^{i-1} [1 - p_j]$.

Anily and Hassin [11] **(1989)** investigate the problem of ranking a K-best binary tree (i.e., the best binary tree excluding $K-1$ better binary trees), both alphabetic and Huffman. To fit the scope of this survey, we elaborate on the ranking of alphabetic trees only. The problem arises when we want to construct the best tree satisfying certain constraints, and no efficient algorithm is known to find this tree. We can then rank the best trees ignoring these additional constraints starting from the best to the next best until the best tree obeying the constraints is reached.

Two algorithms are developed for this problem - an $O(KN^3)$-time and an $O(KN^4)$-time. We elaborate on the more efficient one. A binary tree T is uniquely identified by the sequence of levels of its leaves $T = (l(1), \ldots, l(N))$. The objective is to find a binary tree with the K-best weighted path length (the K-lowest cost). Let T_{ij}^k and C_{ij}^k denote the k-best tree and its cost for a problem consisting of the weights $w_i, w_{i+1}, \ldots, w_j$, and define $W_{ij} = \sum_{r=1}^{j} w_r$. Then $C_{ii}^1 = 0$ for $i = 1, \ldots, N$ and

$$C_{ij}^1 = \min_{i \leq r \leq j-1} \left\{ C_{ir}^1 + C_{r+1,j}^1 \right\} + W_{ij} \text{ for } i < j. \tag{3.2}$$

For $k > 1$, C_{ij}^k is given by $C_{ir}^u + C_{r+1,j}^v + W_{ij}$ for some $i \leq r < j$ and $u, v \leq k$. Thus C_{ij}^k is fully characterized by the triple (r, u, v) and we denote $T_{ij}^k = (r_{ij}^k, u_{ij}^k, v_{ij}^k)$. Let

$$U(i, j, r, K) = \max \left\{ u \,\middle|\, r_{ij}^k = r \text{ and } u_{ij}^k = u, \text{ for some } k = 1, \ldots, K-1 \right\}, \tag{3.3}$$

$$LAST(i, j, r, K, u) = \max \left\{ v \,\middle|\, v_{ij}^k = v, u_{ij}^k = u, r_{ij}^k = r \text{ for some } k = 1, \ldots, K-1 \right\}.$$

Let $M_N = \frac{1}{N}\binom{2N-2}{N-1}$ be the number of distinct alphabetic binary trees with N leaves [120]. For $K \leq M_N$ the proposed algorithm is the following:

Algorithm 1.
Compute C_{ij}^1 *for all* $1 \le i \le j \le N$ *using recursion (3.2).*
For $m = 1, \ldots, N-1$ *do begin*
For $i = 1, \ldots, N-m$ *do begin*
For $k = 2, \ldots, \min(K, M_{m+1})$ *do*

$$C_{i,i+m}^k = W_{i,i+m} + \min_{i \le r < i+m} \left\{ \min_{1 \le u \le U(i,i+m,r,k)+1} \left[C_{ir}^u + C_{r+1,i+m}^{LAST(i,i+m,r,k,u)+1} \right] \right\}. \tag{3.4}$$

end
end
end

Maintaining the appropriate data structure of

$$\left\{ C_{ir}^u + C_{r+1,i+m}^{LAST(i,i+m,r,K,u)+1} \,|\, r = i, \ldots, i+m-1, u = 1, \ldots, K \right\}$$

for each pair (i, m), the $O(K)$ minimizations in the inner loop require $O(N + K\log(N+K))$-time. Since $K \le \log M_N = O(N)$, the overall complexity of the algorithm is $O(KN^3)$.

Normally, a search through a search tree requires a three-way comparison: is the present key equal to, less than, or greater than the search key. **Andersson** [10] describes a method by which it is possible to perform a single two-way comparison at each internal node, with one final comparison when an external node is reached. To do that the author uses a pointer that maps every external node onto an internal node (recall that in a full binary tree the number of external nodes is equal to the number of internal nodes plus one, hence we can associate the external nodes one-to-one with the internal nodes, with one leftover. By dropping the first external node, we take the rest in order (left to right) and pair them up with the internal nodes, also taken in order). Now, suppose that we start at the root of the search tree and go left at every internal node whose key is $<$ the search key, and right otherwise (i.e., when the nodes key is $\ge$ the search key). When we get to an external node, the pointer points back to the last node at which a right branch was taken. We then make an equality comparison with the key at that node: if not equal, then the key is not in the tree[2].

Hu and Tucker [72] **(1998)** introduce an application of alphabetic binary trees to accelerate the search in binary search trees. The authors show that the application of the two-way search procedure introduced in [10] fits exactly the alphabetic tree model, which speeds the complexity of finding an optimal binary search tree from $O(N^2)$ to $O(N\log N)$.

[2]The description of Andersson's method is taken from Hu and Tucker [72], rather than from the original article.

3.4 Verification of optimality

Ramanan [108] **(1992)** shows that, in some special cases, a given alphabetic tree on a sequence of weights can be checked for optimality in $O(N)$ time . Using the correctness of the Hu-Tucker algorithm, the author derives necessary and sufficient conditions on the weight sequence for a given tree to be optimal and concludes that the optimality of very skewed trees (a *skewed* tree is a tree whose maximum number of nodes in any level is bounded by some constant) and well balanced trees (i.e., trees in which the maximum difference between the levels of any two leaves is bounded by some constant) can be tested in linear time.

To do so, the author explores the geometric properties of alphabetic trees. There are C_{N-1} binary trees on N leaves, where C_k is the k-th catalan number $C_k = \frac{1}{(k+1)\binom{2k}{k}} = 2^{\Theta(k)}$. Since the weighted path length is a linear function of $w_1, \ldots, w_N$, the conditions for the optimality of T are given by the C_{N-1} linear inequalities of the form $|T| \leq |T'|$, where T' is any other alphabetic tree with N leaves. The region determined by these inequalities is an unbounded convex closed polytope *poly(T)* in $\mathbb{R}^N$. The author's motivation to study *poly(T)* is as follows: In the problem of constructing an optimal alphabetic tree, the weight sequence $w = \{w_1, \ldots, w_n\}$ can be thought of as a point in the first orthant of $\mathbb{R}^N$. This orthant is partitioned into convex polytopes, each polytope being a *poly(T)* for some T. We are required to find a T such that $w \in$ *Poly(T)*. The complexity of this task depends on the number of *poly(T)s* as well as on the structural complexity of the individual *poly(T)s*.

3.5 Keeping optimality while merging optimal alphabetic trees and inserting nodes into them

Belal, Selim and Arafat [20] **(2002)** introduce an $O(N)$-time algorithm to merge two optimal alphabetic trees with n_1, n_2 nodes into an $N = n_1 + n_2$ nodes optimal alphabetic tree. Given two sequences of lmcps[3] (i.e., sequences of pairs of nodes, which were chosen as lmcps) generated for two weight sequences of lengths n_1, n_2, it is required to find the corresponding sequence of lmcps after concatenating the two weight sequences into one weight sequence of length $n_1 + n_2$.

The sequence of lmcps for an old tree is called the *old tree list*. A general form of an *old lmcp* is two compatible nodes that constituted the local minimum compatible (or tenative-connected, as denoted in the original Hu-Tucker paper) pair for its old tree. The set of nodes that needs to be examined to determine each new lmcp is called *the working sequence*. When the two nodes of an old lmcp are examined in the working sequence and one of them combines to form the new lmcp, the old lmcp must be deleted from all further appearances in the old tree list since it is no longer a valid entry for the new tree.

[3]Recall the definition of lmcp from [71]

The idea of the algorithm is to emulate the effect of rebuilding the optimal N-node tree using the Hu-Tucker algorithm and making use of the information already obtained during the process of building the two previous trees. The processing starts when the two boundary nodes, the rightmost node of the left tree and the leftmost node of the right tree, appear in their corresponding lmcp lists. As long as these two nodes are external, nodes from one tree cannot combine with nodes from the other tree. Thus, old lmcps formed in both trees before those two boundary nodes appear remain valid lmcps for the new tree. When the new lmcp list is completely formed, these entries that were originally valid lmcps for the old trees, although sorted amongst themselves, will cause the new list of lmcps to be unsorted. A final merging phase is required to merge three lists, the list of valid lmcps from each old tree, and the list of newly formed lmcps.

In [21] **(2002) Belal, Selim and Arafat** show how the merging algorithm described in [20] can be applied successively in order to recursively construct an optimal alphabetic tree. Given a weight sequence of N nodes, it is required to find the lmcp list for the optimal alphabetic tree of that weight sequence. The algorithm is as follows: The set of weights is divided into subsets of length 2, where the lmcp list contains a single entry, the existing pair of nodes with the node with smaller index on the left (i.e. an lmcp entry reserves the relative order between its two nodes). The $\frac{N}{2}$ sublists are then grouped into $\frac{N}{4}$ pairs, where each pair contains two adjacent sublists. The two lmcp lists of each group are merged, using the merging algorithm from [20], giving the lmcp list for the subtree of the 4 nodes in the group. The process is repeated, at each step the existing $\frac{N}{k}$ sublists each containing k nodes are merged into $\frac{N}{2k}$ sublists each containing $2k$ nodes. In the last step, two trees are merged, each of length $\frac{N}{2}$, to get the final tree. The algorithm has complexity of $O(N \log N)$ as the known Hu-Tucker or Garsia-Wachs algorithms.

Belal, Selim and Arafat [22] **(2004)** give an $O(N)$-time algorithm to insert an element into an optimal alphabetic tree with N external nodes, keeping the resulting $(N+1)$-leaves tree optimal. The idea is to emulate the effect of rebuilding the optimal $N+1$ leaf tree using the Hu-Tucker algorithm and making use of the information already obtained during the process of building the previous N-leaf tree. We do not elaborate on the inserting algorithm in this survey. Deleting a node q between nodes l, r is accomplished by inserting the negative weight node $(-q)$ between them. This works fine as long as the two nodes $q, -q$ are guaranteed to be combined in the final tree thus forming a node of weight zero. The required tree is obtained by removing the node of weight zero and its two children and decrementing by one the level of the node that formed a lmcp with the zero node.

3.6 Bounds on the cost of an optimal binary alphabetic tree

Melhorn [99] **(1975)** discusses the following "rule of thumb" for constructing nearly optimal binary search trees (recall this model and the relevant

notations from §2): choose the root so as to balance the total weight of the left and right subtrees as much as possible, then proceed recursively[4]. The weighted path length of a tree constructed according to this rule is bounded above by $2 + 1.44 \cdot H$, where $H = \sum \beta_i \log \frac{1}{\beta_i} + \sum \alpha_i \log \frac{1}{\alpha_i}$ is the entropy of the frequency distribution. Bayer [18] (1975) improves this bound to $2 + H$.

In a later work [100] (1977) Melhorn proves an upper bound of $1 + \sum \alpha_j + H$ for the weighted path length, and that this bound is best possible. The approximation algorithm, that creates the upper bound, uses bisection on the set

$$\left\{ s_i \,\middle|\, s_i = \sum_{p=0}^{i-1} (\alpha_p + \beta_p) + \beta_i + \frac{\alpha_i}{2} \text{ and } 0 \le i \le n \right\}, \tag{3.5}$$

i.e., the root k is determined such that $s_{k-1} \le \frac{1}{2}$ and $s_k \ge \frac{1}{2}$. It then proceeds recursively on the subsets $\{s_i \,|i \le k-1\}$ and $\{s_i \,|i \ge k\}$. This rule can be implemented to work in linear time and space.

Yeung [132] **(1991)** proves that the cost $|T|_{opt}$ of an optimal alphabetic tree (i.e., $\beta_1 = \beta_2 = \cdots = \beta_n = 0$ in the binary search problem) is upper bounded by

$$|T|_{opt} < H + 2 - \alpha_0 - \alpha_n. \tag{3.6}$$

Actually, Yeung proposes a linear time algorithm to construct an alphabetical binary tree, whose cost is $\le H + 2 - f(\alpha_0) - f(\alpha_n)$ where $f(x) = x(2 - \log x - \lceil -\log x \rceil)$.

In [34] **De Prisco and De Santis** propose a linear time heuristic that constructs binary search trees whose cost is near to optimal. In fact they prove that the cost $|T|$ of the binary search trees constructed by their algorithm satisfies

$$|T|_{opt} \le |T| < H + 1 - \alpha_0 - \alpha_n + \alpha_{max}, \tag{3.7}$$

where α_{max} is the maximum value among $\alpha_1, \ldots, \alpha_n$. This provides an upper bound on the optimal cost which improves on [99] and also improves on [132] when restricted to alphabetic trees.

Kleitman and Saks [85] **(1981)** obtain an upper bound on the cost of alphabetic binary trees by finding the "worst possible" order of the leaf weights. The authors raise the following question: given a leaf set $E = \{e_1, \ldots, e_N\}$, with w_i the weight of e_i, with the elements listed from least to greatest weight, what linear order of E maximizes the minimum

[4]In [97] Lepala studies the same heuristic for the special case of the alphabetic tree problem

cost of an alphabetic tree? They show that the most expensive order is $e_1, e_N, e_2, e_{N-1}, \ldots$ (they call it a "sawtooth order"). The bound implied by the above solution is as follows: let $H(\{w_1, \ldots, w_N\})$ be the cost of the optimal Huffman (unrestricted) tree on the set. The cost of the optimal alphabetic tree for the worst case ordering is then

$$H\left(\left\{w_i + w_{N-i+1} \,\middle|\, i = 1, \ldots, \frac{N}{2}\right\}\right) + \sum_{i=1}^{N} w_i \tag{3.8}$$

for even N, and

$$H\left(\left\{\frac{w_{N+1}}{2}\right\} \bigcup \left\{w_i + w_{N-i+1} \,\middle|\, i = 1, \ldots, \frac{N-1}{2}\right\}\right) + \sum_{i=1}^{N} w_i - w_{\frac{N+1}{2}} \tag{3.9}$$

for odd N.

Chu [26] **(1985)** extends the result of Kleitman and Saks [85] to the restricted alphabetic binary trees. The author shows that the "sawtooth" sequence is also the most expensive sequence for the L-restricted alphabetic binary trees where L is the maximum depth of the leaves and $\lceil \log N + 1 \rceil \leq L \leq N$.

3.7 Parallel construction of binary alphabetic trees

The computational complexity of an algorithm that seeks an optimal alphabetic tree can be reduced if several computer processors are available. The binary search tree problem can be solved using parallel dynamic programming in $O(\log^2 N)$ time with roughly N^4 processors, using a concave matrix multiplication algorithm (a result by Atallah, Rao Kosaraju, Larmore, Miller and Teng, [13], 1989). An $O(\log^2 N)$-time, N^2-processor algorithm that uses dynamic programming to compute an approximately optimal binary search tree was also given in [13]. The optimal alphabetic tree problem is a special case of the optimum binary search tree problem, thus those algorithms can be used to obtain an approximately optimal alphabetic tree. On the other hand, the best currently known sequential algorithm to construct an optimal alphabetic tree does not use dynamic programming. Thus one can ask whether, if several computer processors are applied, one can also find a non-dynamic programming algorithm whose complexity is less than the complexity of parallel known optimum binary search tree algorithms.

In [93] **Larmore and Przytycka (1991)** provide a partial answer to this question. They present an $O(K \log N)$-time N-processor CREW PRAM[5] algorithm which constructs an optimal alphabetic tree with height restriction K. As a consequence, they obtain an $O(l \log^2 N)$-time, N-processor algorithm that constructs an almost optimal alphabetic tree (more precisely a tree whose cost differs by at most $\frac{1}{N^l}$ from the cost of an optimal tree). Furthermore, if the input sequence of weights does not contain two consecutive elements of weight less than $\frac{1}{N^l}$, the algorithm produces an optimal tree. Their algorithm is based on the Package Merge technique (see further details in §7) and is not described here.

In [96] **(1993) Larmore, Przytycka and Rytter** present a way to parallelize the algorithm of Hu and Tucker [71] (or the equivalent version of Garsia and Wachs [45]). Their algorithm constructs an optimal alphabetic tree in $O(\log^3 N)$ time with $N^2 \log N$ CREW PRAM processors. In this survey we don't provide the necessary background from the theory of parallel algorithms, thus we suffice with a general overview of the algorithm. The algorithm has the same two phases as the Hu-Tucker and the Garsia-Wachs algorithms. In the first phase, a certain tree called the l-tree (short for "level tree") is constructed. This is a binary tree whose leaves are the nodes in the list, and whose internal nodes are what Hu and Tucker call "round nodes", and are called packages in [93]. In the second phase, the alphabetic tree is found where each item is at the same level as in the l-tree. This is the optimal alphabetic tree.

Arafat [12] introduces a simple parallel algorithm that builds an optimal alphabetic tree using p ($p < N$) processors in $O(\frac{N}{p} \log N + N)$ time. The algorithm is an extension of a sequential divide and conquer algorithm for the problem, published by **Belal, Selim and Arafat** in [21]. The sequential algorithm builds the optimal alphabetic binary tree by repeated merging of subtrees, using a previous result [20] that showed how optimal alphabetic binary trees can be merged in linear time: The initial sequence of nodes is divided into sublists of two elements each, the optimal alphabetic binary tree for each pair is trivial to find. Each two adjacent optimal alphabetic binary trees are merged into one optimal alphabetic binary tree. This step is repeated till all nodes are merged into the final optimal alphabetic tree ($\log N$ steps).

The parallel proposed algorithm, given N items and p processors (where p is a power of two) has two phases:

(1) "Each processor is assigned a contiguous list of no more than $\frac{N}{p}$ elements. All processors build an optimal alphabetic tree of their

[5]In computer science, a parallel random-access machine (PRAM) is a shared-memory abstract machine. As its name indicates, the PRAM was intended as the parallel-computing analogy to the random-access machine (RAM). In the same way, that the RAM is used by sequential-algorithm designers to model algorithmic performance, the PRAM is used by parallel-algorithm designers to model parallel algorithmic performance. Read/write conflicts in accessing the same shared memory location simultaneously are resolved by the CREW (Concurrent Read Exclusive Write) strategy: multiple processors can read a memory cell but only one can write at a time.

lists in parallel using any sequential algorithm of time complexity $O(N \log N)$ (the divide and conquer algorithm or the Hu-Tucker algorithm for example), then all processors synchronize.

(2) At this stage, there are p optimal alphabetic trees, each with $\frac{N}{p}$ nodes in it. These optimal alphabetic binary trees need to be merged together into one optimal alphabetic binary tree. Merging is done successfully in $\log p$ steps, where subtrees are merged two by two in parallel at each step. Each merge lasts $O(\frac{N}{p})$ time according to [20]."

3.8 Approximation algorithms for the optimal alphabetic binary trees.

Hwang and Tsai [76] **(2003)** derive asymptotic approximations for the sequence $f(n)$ defined recursively by $f(n) = \min_{1 \le j < n} \{f(j) + f(n-j)\} + g(n)$, when the asymptotic behavior of $g(n)$ is known. Following are some of their results:

(1) If $g(n)$ is *rapidly growing*, i.e., $g(n)$ satisfies $\frac{g(n)}{\max\{g(\lfloor \frac{n}{2} \rfloor), g(\lceil \frac{n}{2} \rceil)\}} \longrightarrow \infty$ as $n \longrightarrow \infty$, then $f(n) \approx g(n)$.

(2) If $g(n) \approx n^a l(n)$, where $a > 1$ and $l(n)$ is *slowly varying*, i.e $\frac{l(\lfloor bn \rfloor)}{l(n)} \approx 1$ for all $b > 0$ and $n \longrightarrow \infty$, then $f(n) \approx \frac{n^a l(n)}{1-2^{1-a}}$.

The results are also applicable to problems where the function $g = g(j, n-j)$ is dependent on both j and n. The general pattern of such a recurrence is $f(n) = \min_{1 \le j < n} \{f(j) + f(n-j) + g(j, n-j)\}$ with $f(1)$ given. Functions $g(x,y) = ax + by$ and $g(x,y) = ax(x+y) + b(x+y)$ appear in dichotomous search problems, thus the results of [76] are of an interest in this survey. For example, the function $g(n) = nf(n)$ with $f(n)$ defined in (6.10) has a structure that admits the model of [76].

4. The monotonicity property - conditions and benefits for the various costs problems

Hassin and Henig (1993) address in [56] a general formulation of various cost search problems and explore the limits of "the Knuth method" of reducing the computation effort of solution. An object is searched in an interval $1, \ldots, N$. Let p_i be the probability that the object lies in i, for $i = 1, \ldots, N$. A query at k reveals whether the object lies in $\{1, \ldots, k\}$ or in $\{k+1, \ldots, N\}$. Define "Problem (m,n)" the instance where an object is known to be located in the interval $\{\min(m,n), \max(m,n)\}$.

Two types of costs involved in the search are considered:

- $D^l(m,n,k)$ for placing the lth query at k in Problem (m,n).
- C_{li} if the object is found in i after l queries.

The problem can be solved by applying the following recursive equations: Denote $p_{ij} = p_i + \cdots + p_j$. For $l = 0, \ldots, N-1$, $m = 1, \ldots, N$

$$F^l(m,m) = C_{lm}, \tag{4.1}$$

for $m < n$ and $l = 1, \ldots, N - (n-m)$

$$F^{l-1}(m,n) = \min_{m \le k < n} \left\{ D^l(m,n,k) + \frac{p_{mk}}{p_{mn}} F^l(k,m) + \frac{p_{k+1,n}}{p_{mn}} F^l(k+1,n) \right\}, \tag{4.2}$$

and for $m > n$ and $l = 1, \ldots, N - (m-n)$

$$F^{l-1}(m,n) = \min_{n \le k < m} \left\{ D^l(m,n,k) + \frac{p_{nk}}{p_{nm}} F^l(k,n) + \frac{p_{k+1,m}}{p_{nm}} F^l(k+1,m) \right\}. \tag{4.3}$$

$F^l(m,n)$ is the minimum expected cost involved with locating the object in Problem (m,n) when l queries have already been placed. Let k be a minimizing value for the right-hand side of (4.2) or (4.3), then F is said to be attained at k. The minimum total cost of the search is $F^0(1,N)$. The complexity of the algorithm is $O(N^4)$. It can be reduced to $O(N^3)$ by applying ideas from [87] in cases when the following *"monotonicity property"* holds:

The monotonicity property: If $F^l(m,n-1)$ is attained at k', then for some $k \ge k'$, $F^l(m,n)$ is attained at k.

Knuth in [87] proved that the monotonicity property holds when C is linear in l and constant in i, and D is constant. The authors in [56] prove that the monotonicity property holds also under more general conditions.

Define $G^l(m,n) = p_{mn} F^l(m,n)$ if $m \le n$ and $G^l(m,n) = p_{nm} F^l(m,n)$ if $m > n$. Let $d^l(m,n,k)$ be equal to $D^l(m,n,k) p_{mn}$ for $m \le n$ and to $D^l(m,n,k) p_{nm}$ for $n < m$. Let $c_{lm} = C_{lm} p_m$. Rewriting the dynamic programming equations (4.1)-(4.3) for G, it can be seen that $G^l(m,n)$ is attained at the same values as $F^l(m,n)$.

The main result of [56] states that the following "submodular" property of G is sufficient for the monotonicity property to hold, and thus for the reduction of the complexity of the dichotomous search to $O(N^3)$:

"For $l = 0, \ldots, N-1$, let d^l be defined over the lattice $\Phi = \{(m,n,k) : 1 \le m, n < N, \min\{m,n\} \le k <$
Let c_{ln} be defined over $\{(l,n) : l = 1, \ldots N-1, n = 1, \ldots N\}$. We make the following assumptions:

- d^l is submodular for every fixed l, i.e., for every pair of points (m_i, n_i, k_i) $i = 1, 2$ conditions (4.4),(4.5),(4.6) hold:

$$d^l(m_1,n_1,k_1)+d^l(m_2,n_2,k_2)\geq \tag{4.4}$$

$$d^l\left(\min\{m_1,m_2\},\min\{n_1,n_2\},\min\{k_1,k_2\}\right)+$$

$$d^l\left(\max\{m_1,m_2\},\max\{n_1,n_2\},\max\{k_1,k_2\}\right),$$

$$d^l(n-1,n,n-1)+d^l(n,n+1,n)\leq \tag{4.5}$$
$$\min\left\{d^l(n-1,n+1,n-1)+d^{l+1}(n,n+1,n),d^l(n-1,n+1,n)+d^{l+1}(n,n-1,n-1)\right\},$$

$$d^l(n,n-1,n-1)+d^l(n+1,n,n)\leq \tag{4.6}$$

$$\min\left\{d^l(n+1,n-1,n-1)+d^{l+1}(n+1,n,n),d^l(n+1,n-1,n)+d^{l+1}(n-1,n,n-1)\right\}.$$

- c is convex in l for every fixed n, i.e., $2c_{ln}\leq c_{l+1,n}+c_{l-1,n}$ for $l=2,\ldots,N-2$.
- c is nonnegative and nondecreasing in l for every fixed n.

Under these assumptions G^l is submodular, i.e., for $1\leq m\leq m+a\leq N$, $1\leq n\leq n+b\leq N$ and $2\leq l\leq N-\max\{|m-n-b|,|m-n+a|\}-1$:

$$G^l(m,n)+G^l(m+a,n+b)\leq G^l(m,n+b)+G^l(m+a,n). \tag{4.7}$$

Also, the monotonicity property holds under these assumptions."

The authors provide some applications of the theorem to reduction of complexity of various problems. For example, it is applicable to dichotomous search with *position-dependent query costs*, a version of which (search for a record on a tape) was discussed in [125]. Here the cost of placing a query varies according to the position of the query. Let $d^l(m,n,k)=d^l(k)$. It is easy to check that the necessary conditions hold.

Hinderer and Stieglitz [62] **(2000)** continue the work of Hassin and Henig and weaken the requirement on submodularity of function G^l from set $\mathbb{Z}^2$ to a smaller set.

The authors consider a family of problems SPD_K of dichotomous search of at most K queries in the interval of integers $[1,N]$. Their main result is a method for the problem SPD_K that derives natural conditions under which at each stage k, $1\leq k\leq K$, the smallest optimal place for search in $[i,j]$ is increasing both in i and in j.

As in [56], the crucial point is to find conditions under which the l-stage transformed minimal cost function G^l is submodular on the momentary state space $S_l\subset\mathbb{Z}^2$ (we do not delve into the precise definition of spaces S_l). The authors inherit this property from G_{l-1} to G_l based on two ideas:

(1) They make systematic use of the work of **Topkis (1978)** [122]. This allows, under the natural assumptions that G_0 and also the expected one-stage search cost functions are submodular, to inherit submodularity of G_{l-1} on S_{l-1} to submodularity of C_l on a certain "large" sublattice S_l' of S_l.

(2) G_l turns out to be submodular on the whole set S_l if in addition to assumptions in (1) we require that G_l is submodular on a certain "small" sublattice S_l'' of S_l such that $S_l' \bigcup S_l'' = S_l$, provided that "S_l' and S_l'' overlap sufficiently". This completes the analysis since that additional assumption can be expressed in a straightforward manner as a system of inequalities in terms of the data of the search problem considered.

5. Search for an object distributed uniformly or as $p_k = ck^{-a}$

In this section we consider the dichotomous search problem with an a priori assumption on the distribution of the object. We start with the assumption of uniform distribution, and further proceed with a more general assumption - distribution $p_k = ck^{-a}$.

Dichotomous search with uniform distribution is an alphabetic binary tree problem in which the weights attached to the leaves are all equal. An integer x is searched for through the set of integers $1, ..., N$. The a priori distribution of the location of x is uniform, meaning that the probability that $x = i$ equals $\frac{1}{N}$ for all $i = 1, 2, ..., N$. In a comparison of x against x_i two possible outcomes exist, namely $x > x_i$ or $x \leq x_i$. The goal is to minimize the expected number of comparisons per successful search.

Ferguson [37] **(1960)** analyzes the performance of *Fibbonaci search* for this problem when N is a Fibonacci number. The Fibonacci numbers are defined by:

$$\begin{array}{ll} u_i = i & 0 \leq i \leq 2 \\ u_i = u_{i-1} + u_{i-2} & i \geq 2 \end{array} \tag{5.1}$$

The Fibbonachi search is defined as follows: Suppose the interval to be searched through is of size u_k for some k (at start $u_k = N$ in case N is a Fibonacci number. If N is not a Fibonacci number, the algorithms fails), and the location of the item searched for is uniformly distributed in it. If at some point in the process the item has been isolated to an interval of size u_i, beginning at A, compare the value of the argument to $A + u_{i-1}$. If the item is to the left, then it is now isolated to an interval of size u_{i-1}, otherwise it is in an interval of size u_{i-2}.

The expected number of comparisons while searching a list of N elements (N is some Fibonacci number) is $O(\log_2 N)$, and the maximum searching time is $O(\log_\varphi N)$ where $\varphi = \frac{1+\sqrt{5}}{2}$ is the "golden section" (note that the expected and the maximum search times are asymptotically identical). The

original motivation of the author to explore the Fibonacci search is not to reduce the number of computations relative to choosing the middle item each tome (the classic binary search), but to replace the mathematical operations needed from divisions to substractions.

Overholt [105] **(1973)** holds a more exact analysis of the Fibonacci search performance. He shows that this method has a mean search length some 4% greater than the ordinary binary search and also a much greater maximum search length and standard deviation. In contrast to the ordinary binary search, where the greatest search length is one or two tests longer than the mean search length, the Fibonacci maximum search length is nearly 40% greater than the mean.

A search procedure based on Fibonacci numbers can be used in approximating the maximum of a function as **Hassin (1981)** shows in [53]. A function f is called *unimodal* on $[a, b]$ if there exists $a \leq x \leq b$ such that $f(y)$ is strictly increasing for $a \leq y \leq x$ and strictly decreasing for $x \leq y \leq b$. The maximum point x is the search argument. Suppose that only N f-evaluations are available. The property of unimodality enables, after two evaluations of f, to obtain a smaller interval of uncertainty regarding the location of x. The Fibonacci method divides the initial interval (whose length itself is a Fibonacci number) to two intervals, such that the proportion of their lengths is the proportion of sequential Fibonacci numbers. Earlier, it has been shown at [14] and [81] that the Fibonacci search method guarantees the smallest final interval of uncertainty among all methods requiring a fixed number of function evaluations. In [53] the author shows that Fibonacci search guarantees the smallest final interval of uncertainty of the maximum also when f is not unimodal. The author also suggests a refinement of the algorithm for the case that the initial interval's length is not a Fibonacci number.

Wong [130] **(1964)** develops optimal solutions for the search problem with a uniform a priori distribution. The author assumes that in a comparison of x against x_i three possible outcomes exist (and not two, as considered earlier), namely $x > x_i, x < x_i$ or $x = x_i$. Let $n^*(N)$ be the first step of an optimal strategy (of course $n^*(N)$ need not be unique). The objective is to minimize the expected number of comparisons per successful search. The author depicts the complete set of optimal strategies:

For $N = 2^{k+1}+2m$ if $m < 2^{k-1}$ then $n^*(N) = \{2^k, 2^k + 1, \ldots, 2^k + 2m + 1\}$.
If $m \geq 2^{k-1}$ then $n^*(N) = \{2^k + 2m + 1, 2^k + 2m + 2, \ldots, 2^{k+1}\}$.

For $N = 2^{k+1}+2m-1$ if $m \leq 2^{k-1}$ then $n^*(N) = \{2^k, 2^k + 2, \ldots, 2^k + 2m\}$.
If $m > 2^{k-1}$ then $n^*(N) = \{2m, 2m + 1, \ldots, 2^{k+1}\}$.

For example, consider $N = 2^4 + 9 = 25$. Then $n^*(N) = \{9, \ldots, 16\}$. This is a surprising result since, intuitively, one would expect that the optimal solution $n^*(N)$ should divide the remaining $N - 1$ cells into nearly equal subsets. Thus, the large multiplicity of solutions is not expected. The author also computes the optimal value of the problem, produced by applying the optimal strategies described above.

In order to construct a lower bound for the expected number of comparisons required to sort a table of N items, **Morris** [101] **(1969)** solves the following binary search problem with uniform a priori distribution: Suppose that $(N-1)$ items $A_1, ..., A_{N-1}$ have already been sorted into linear order. The goal is to compute $F(N)$, the minimal expected number of comparisons required to insert the next object A_N into the linear order.

Via the dynamic programming equation (5.2) and the convexity property of the function $G(N) = NF(N)$, the author proves for the first time the lower bound of $\lceil \log_2 N \rceil$ for $F(N)$. Moreover, Morris proves that if N is a power of 2 then $F(N) = \log_2 N$.

$$\begin{aligned} &F(1) = 0, \\ &F(N) = 1 + \min_{k=1,..,N-1} \left\{ \tfrac{k}{N} F(k) + \tfrac{(N-k)}{N} F(N-k) \right\}. \end{aligned} \tag{5.2}$$

For N that is not a power of 2, an explicit description of $G(N)$ is provided: the function $G(N)$ is linear in any interval $2^l \leq n \leq 2^{l+1}$ and coincides with the function $h(n) = n \log_2 n$ at the points n which are powers of 2.

Several other papers on this subject are available with the assumption of a uniform a priori distribution. **Gal (1974)** [42] considers the problem as a zero-sum two-person game in which a searcher tries to identify an integer that has been chosen by a hider from $1, ..., N$. This paper will be covered in §12.

Another class of papers considers the dichotomous search problem with an a priori distribution assumption but also with various cost and goal formulations. Examples of such are [102, 65, 61, 25], which will be discussed in §6.

Now, suppose that the a priori distribution of the object in the interval of integers $[1, 2, \ldots, N]$ is known to be $p_k = ck^{-a}$. **Lepala** [97] **(1979)** studies the *generalized binary search* heuristic of searching for the object at the location that divides the set $[1, \ldots, N]$ to two parts with equal probability of containing the object. More formally, to search the interval of integers $[l, u]$, divide it into two parts by determining index $i = \operatorname{argmin}_{l \leq i \leq u} \left| \sum_{k=1}^{i-1} p_k - \sum_{k=i+1}^{u} p_k \right|$. The same heuristic was previously defined and considered in [99] (1975) for the more general problem of binary search trees. In case of the uniform distribution, this method coincides with the standard binary search. Yet when other distributions are taken into account, this heuristic has better performance than the ordinary binary search.

Let $p_k = ck^{-a}$ be the a priori probability that the object lies at k for $(1 \leq k \leq N)$, where $a \geq 0$ and $c = \frac{1}{\sum_{i=1}^{N} i^{-a}}$ is a normalization constant. For $a = 0$ the uniform distribution is obtained, for $a = 1$ the Zipfian distribution. The objective is to minimize the expected number of comparisons.

By the definition of p_k, and by approximating the series $\sum \frac{1}{k}$ by the integral $\int x^{-a}$ we get:

$$i = \begin{cases} \left\lfloor \sqrt{lu} \right\rfloor & a = 1 \\ \left(\frac{l^{1-a} + u^{1-a}}{2} \right)^{\frac{1}{1-a}} & a \neq 1. \end{cases} \tag{5.3}$$

For $a = 0$, the bisection rule of binary search is obtained, and for the Zipfian distribution ($a = 1$), we have the geometric mean of l and u instead of the arithmetic mean of the regular binary search.

For $0 \leq a < 1$ the expected number of comparisons using the generalized binary search is:

$$C_N \approx \frac{2 \log N}{\frac{2}{1-a} - \log(2^{\frac{1}{1-a}} - 1)} \tag{5.4}$$

It is of the same order as the corresponding result $C_N \approx \log N$ of binary search, but the coefficient is smaller than 1 for all $0 < a < 1$.

In the case $a = 1$ of Zipfian distribution, the expected number of comparisons is $C_N \approx 2 \log \log N$, an order of magnitude smaller than the result of the ordinary binary search.

When $a > 1$, the expected number of comparisons using the generalized binary searched is constant, thus by far superior to the ordinary binary search.

6. Different objective and costs formulations

This section is a survey of the literature on dichotomous search problems, in which the costs and objective functions are different from the basic version discussed earlier. In particular we describe problems where the cost for traveling right is different from the cost of traveling left ("asymmetric error costs") and where the travel itself has a cost proportional to the distance traveled ("search with travel costs"). As for the objective function, here we encounter goals different from minimizing the expected sum of search costs, such as minimizing the overall traveled distance ("minimizing the sum of errors") and maximizing the probability of finding the object within a given number of queries.

6.1 Asymmetric error costs

The earliest work on the search problem with asymmetric error costs is that of **Cameron and Narayanamurthy** [25] **(1964)**. An event is presented by a point in an interval and the distribution of its location is uniform on the interval (the distribution is continuous). It is desired to locate the event. The only available test is to select a point and find out whether the event lies to its left or to its right. The cost of the test is 1 unit if the event lies

to the left of the test point and k units (it is assumed that $k > 1$ [6]) if it is to the right. The search is terminated if the event is located within an interval of length equal to or smaller than 1. The objective is to derive a search policy that minimizes the expected cost.

Any policy can be represented by a function $g(x)$, $x > 1$, such that if the interval is of length x_0 the point to be tested divides it in the ratio $g(x_0) : 1-g(x_0)$. Each test locates the event within a smaller interval, and repeated application of the policy to the diminishing intervals constitutes the search procedure. Intuitively, it is seen that for large x the optimal $g(x)$ must approach asymptotically a constant value α that depends only on k.

Let $f(x)$ be the *expected* cost of the search procedure using an optimal policy for the interval $[0, x]$. Then

$$f(x) = \begin{cases} 0 & 0 \le x \le 1, \\ \min_{g(x)} \left[1 + g(x)f(xg(x))], [k + f(x(1-g(x)))\right] & x > 1. \end{cases} \tag{6.1}$$

Considering only functions $g(x)$ that approach a constant value r asymptotically, for large x we have the functional equation:

$$f(x) = \min_r \left[r[1 + f(rx)] + (1-r)[k + f((1-r)x)]\right]. \tag{6.2}$$

The optimal value α of r must satisfy the equation obtained by setting to zero the derivative of the quantity within brackets in (6.2). By solving that equation it is proven that the optimal value of $f(x)$ is given by $f(x) = p\log(x) + c$ with p calculated by (6.3):

$$p\log(1-\alpha^{1-k}) = k, \tag{6.3}$$

and α uniquely determined by (6.4):

$$\alpha^k + \alpha = 1. \tag{6.4}$$

There is a unique positive p that satisfies equation (6.3) and lies in the range $0 < p < \frac{k}{\log 2}$. Equation (6.3) does not determine the constant c. The resulting policy of testing at the point that divides the interval in the ratio $\alpha : 1-\alpha$ until the event is located within a unit interval is not optimal because only constant ratio policies are considered. The authors call this policy *the suboptimal policy*. Via (6.1) the exact optimal solution is computed for special cases. Comparison leads to the conclusion that the suboptimal policy is likely to be satisfactory for most applications.

Murakami [102] **(1971)** continues the work of Cameron and Narayanamurthy [25] by deriving an explicit representation of the constant c for the

[6]Note that it is not assumed that k is an integer

function $f(x)$ discussed above, thus completing the *suboptimal solution* of [25]. Moreover, Murakami presents an optimal search strategy for this problem. Notable is the fact that the author does not use dynamic programming:

The author formulates an equivalent problem to that in [25]: the search is for an object located uniformly in an interval of length 1 and the objective is to find it in an interval of length $\frac{1}{n}$ for a given n. The cost of a test is 1 unit if the event lies to the left of the test point and k units if it lies to the right of the test point, with k being a positive integer[7]. $E(n,x)$ is defined to be the *expected cost* of the search procedure required when we select first a test point x and thereafter use an optimal policy. The objective function is therefore defined as $f(n) = \min_{0\leq x\leq n} E(n,x)$. For the **exact optimal solution** two series $N(i)$ and $g(n)$ are computed:

$$N(i) = \begin{cases} 1 & i = 2-k, 3-k, \ldots, 0, 1. \\ N(i-1) + N(i-k) & i = 2, 3, 4 \ldots \end{cases} \tag{6.5}$$

$$g(n) = \begin{cases} 1+k & n = 1 \\ g(n-1) + \varphi(n) & n = 2, 3, 4, ... \end{cases} \tag{6.6}$$

where $\varphi(n) = 1$ if there exists an integer j satisfying the equation $n = N(j)$ and $\varphi(n) = 0$ otherwise.

The values of $N(i)$[8] and $g(n)$ are computable recursively. First compute $g(n)$, afterwards compute $N(g(n))$. An explicit formulation of $f(n)$ is presented:

$$f(n) = g(n) - \frac{N(g(n))}{n}, \quad \text{for } n = 1, 2, ... \tag{6.7}$$

Moreover, the author presents the set $x^*(n)$ of optimal test points for any fixed n. The algorithm for computing $x^*(n)$ and $f(n)$ is constructive, but the recursive computation is rather complicated. Therefore, an approximate expression of the optimal solution is also derived, thus completing the previous work of Cameron and Narayanamurthy. Recall the asymptotic result in [25]

$$f(n) = p\ln(n) + c, \tag{6.8}$$

where p is a function of k. The author finds an exact expression for the constant c:

$$c = 2 - \frac{1}{1-\alpha} + p\ln(\alpha + k(1-\alpha)), \tag{6.9}$$

[7]Note that this assumption is not needed in the work of Cameron and Narayanamurthy

[8]The formula for $N(i)$ makes use of the assumption that k is a positive integer.

where α is the optimal value for the division ratio r, calculated in [25].

Hinderer [61] **(1990)** expands the results of Murakami to a more general case, in which the costs for tests to the left and to the right of the current test point can also be rational (as opposed to natural in [102]) . Let $c > 0$ be the cost for moving right to the previous location, and $c' > 0$ the cost of moving left (c and c' are rational numbers). Let $f(s)$ be the the minimal expected search cost for an object initially hidden in $N_s = \{1, 2, \ldots, s\}$, $s \geq 2$. Then $f(1) = 0$ and

$$\begin{aligned} f(s) &= \min_{a \in N_{s-1}} \left\{ (c + f(a)) \cdot \frac{a}{s} + (c' + f(s-a)) \cdot \left(1 - \frac{a}{s}\right) \right\} \\ &:= \min \{ W(s,a) \,|a \in N_{s-1} \}. \end{aligned} \tag{6.10}$$

Let $D^*(s)$ denote the set of minimum points of $W(s,a)$. Assume that $\frac{c}{c'} = \frac{m}{k}$ for positive integers m and k. In the simplest case $m = k = 1$ an explicit solution was obtained by Morris [101]. If $m = 1$ and $k \in \mathbb{N}$, Murakami [102] provided an explicit solution. Hinderer provides an explicit solution also for the more general case, in which m, k are any natural numbers.

For m, k natural numbers let $(N(i), i \in \mathbb{N})$ be the solution of:

$$\begin{array}{ll} N(i) = N(i-m) + N(i-k) & i \geq 2 \\ N(i) = 1 & 2 - \max(m,k) \leq i \leq 1. \end{array} \tag{6.11}$$

Define $H(s) = sf(s)$, since the analysis of H is simpler then that of f. Let $S_i = \{ s \in \mathbb{N} \,| N(i) \leq s < N(i+1) \}$ and $\bar{S}_i = \{ s \in \mathbb{N} \,| N(i) \leq s \leq N(i+1) \}$.

The main theorem of [61] states:

(1) "If $i \geq 2$ and $S_i \neq \emptyset$, then $H(s+1) - H(s) = i + m + k - 1$ for $s \in S_i$.

(2) Moreover, $D^*(s) = \{ s \in \mathbb{N} \,| L(s) \leq s \leq M(s) \}$, where for $i \geq 2$ and $s \in S_i$

$$\begin{array}{l} L(s) = \max(N(i-m), s - N(i-k+1)), \\ M(s) = \min(N(i-m+1), s - N(i-k)).'' \end{array} \tag{6.12}$$

The author also analyzes the asymptotics of the function $f(s)$, seeking for a good approximation $A(s)$. Recall that Murakami [102] proved that for the case $m = 1$, for explicitly given $p = p(k)$ and $b = b(k)$ statements (6.9,6.8) hold, which can be rewritten for $i \to \infty$ as:

$$f(N(i)) - p \ln[N(i)] - b \to 0. \tag{6.13}$$

The approximation (6.13) is excellent. However, computing $f(s)$ for a given s (not just $s = N(i)$) by means of (6.13) requires first to calculate $j_s = \max \{ i \in \mathbb{N} \,| N(i) \leq s \}$ and $N(i)$. Then for $A(s) := p \ln(s) + b$:

$$f(s) = f(N(i))\frac{N(i)}{s} + \left(1 - \frac{N(i)}{s}\right)(i+k) \tag{6.14}$$

$$\approx A(N(i))\frac{N(i)}{s} + \left(1 - \frac{N(i)}{s}\right)(i+k).$$

On the other hand, the *exact* solution given in (6.12) requires not much more, namely the computation of $N(i+k)$. Thus (6.14) seems to be preferable to (6.12) only if k is large. The approximate computation of $f(s)$ could have been possible without computing j_s and $N(i)$ if the statement $f(s) = p\ln(s) + b + o(1), s \to \infty$ was true. The question whether it is true remained open by Murakami in [102] but receives an answer in [61].

The main approximation result of this paper shows that for arbitrary m and k (assume that g.c.d$(m,k) = 1$) and appropriate constants $p = p(m,k)$ and $b = b(m,k)$ (defined in the paper) the following holds:

(1) The approximation (6.13) holds and $f(N(i)) - (i+m+k-1-\frac{1}{1-\alpha}) \to 0$ for $\alpha \in (0,1)$ determined by $\alpha^m + \alpha^k = 1$.

(2) The minimum and maximum of $f(\cdot) - A(\cdot)$ on S_i tend for $i \to \infty$ to zero and to some explicitly given positive constant $M = M(m,k)$, respectively. Thus $f(\cdot)$ can be approximated by $A(\cdot)$. In particular $f(s) \neq p\ln(s) + b + o(1)$ but $f(s) = p\ln(s) + b + O(1)$. From this we derive that $f(s)$ cannot be calculated without calculating j_s first.

Itay [77] **(1976)** discusses the problem of asymmetric error costs on σ-ary alphabetic trees. We elaborate on this article below in §9.2.1.

Abigadol and Ben-Tal [1] **(1985)** consider the following problem: k components compose a system. Each component is subject to failure if temperature is above an unknown critical level. The system as a whole fails when at least one component fails. If z_i is the critical temperature of the ith component then $z^* = \min\{z_i \mid i = 1,2,\ldots,k\}$ is defined as the critical level of the system. It is given that $z_1,\ldots,z_k \in (0,1]$. No assumption on the distribution of $z_1,\ldots,z_k$ is taken[9]. The components are tested individually at different temperature levels. If the temperature is below the critical level the cost is 1, otherwise the test is destructive and the cost is $m > 1$. A *search procedure* is a strategy specifying at each iteration which component to test and at what level, according to information on previous tests. The tests continue until the initial budget of size n is exhausted (n and m are positive integers, inputs of the problem). The *relevant interval* of z_i, denoted by L_i is a subinterval of $(0,1]$ such that $z_i \notin L_i \Longrightarrow z_i \neq z^*$. The authors illustrate the problem by the case of two components, z_1 and z_2, at an intermediate stage of the search, where both have the relevant interval $[a,b]$: "Suppose that the next observation is on z_1 at $x \in (a,b)$. If the outcome is that z_1 is

[9]In a simulation study of the performance of the algorithm, several distributions of the points are examined, among them a uniform distribution and a normal distribution truncated to the interval $[0,1]$

to the left of x, then the relevant interval of both points is $[a, x]$, while, if the result is that z_1 is to the right of x, the relevant interval of z_1 is $[x, b]$ and the one of z_2 remains $[a, b]$." At each stage of the search, the *error* ϵ is defined as the length of the maximal relevant interval. $s_k(n)$ is a search procedure with k components and initial budget n, and $v[s_k(n)]$ the value of $s_k(n)$, i.e., the maximum possible error induced by it. The objective is to develop a search procedure that minimizes the maximum possible final error $v[s_k(n)]$ over all possible $z_1, \ldots, z_k \in (0, 1)$. If after a query is conducted, it appears that the remaining budget is not enough to pay for it, than that last query is canceled and the error of the search is the length of the last relevant interval before the query was conducted.

For $k = 1$ the solution is calculated using dynamic programming: "Let $v(n)$ be the maximal length of the relevant interval resulting from an optimal minmax search, given initially a budget of size n ($n \geq m$). Then

$$v(n) = \begin{cases} 1 & 0 \leq n < m \\ \min_{0<x<1} \max\{xv(n-m), (1-x)v(n-1)\} & n \geq m. \end{cases} \tag{6.15}$$

Let $x = x_n$ be the minimizer of (6.15). Then clearly

$$v(n) = x_n v(n-m) = (1-x_n)v(n-1), \tag{6.16}$$

and hence

$$x_n = \frac{v(n-1)}{v(n-1) + v(n-m)}. \tag{6.17}$$

Substituting (6.17) into (6.16) we get

$$v(n) = \begin{cases} 1 & 0 \leq n < m \\ \frac{v(n-1)\cdot v(n-m)}{v(n-1)+v(n-m)} & n \geq m. \end{cases} \tag{6.18}$$

In another form, letting $\lambda_n = \frac{1}{v(n)}$, equations (6.17,6.18) become $x_n = \frac{\lambda_{n-m}}{\lambda_n}$,

$$\lambda_n = \begin{cases} 1 & 0 \leq n < m \\ \lambda_{n-m} + \lambda_{n-1} & n \geq m. \end{cases} \tag{6.19}$$

The sequence λ_n is the solution of a linear difference equation. Note that for $m = 2$ it is the Fibonacci sequence."

For $k = 2$, the optimal search procedure is the following: Starting with relevant intervals both of size one, one proceeds sequentially to test the location of one of the components, to obtain smaller relevant intervals. To specify the search procedure two questions must be answered: (a) which component (z_1 or z_2) to test, and (b) where to place the observation point

x. As for the first question there is an optimal answer - at every stage test the component with the largest relevant interval. To get a precise answer to the second question, dynamic programming equations must be solved.

For $k = 2$ (and for larger k also) the dynamic programming is very costly. Thus the authors develop a computationally feasible upper bound to the final error for $k = 2$. Let $E(a_l, a_2; n)$ be the length of the maximal relevant interval resulting from an optimal (minmax) search given a budget n and current relevant intervals of lengths a_1, a_2 (it is shown that each stage is fully characterized only by the length of the relevant intervals and not by their exact positioning). Due to the homogeneity property of E ($E(a_1, a_2; n) = a_1 E(1, \frac{a_2}{a_1; n})$) one can always look at a problem where the longest relevant interval is of size one. Denote $H(t, n) = E(1, t; n)$. The main theorem of the research states that $H(t, n) \leq \alpha_n(1 + \beta_n t)$ where

$$\alpha_n = v(n), \tag{6.20}$$

and

$$\beta_n = \begin{cases} 1 & n = 0, \ldots, m-1 \\ \frac{(\alpha_{n-1}+\alpha_{n-m})(\beta_{n-m}+1+\beta_{n-1}\beta_{n-m})}{\alpha_{n-1}\beta_{n-1}+\alpha_{n-m}\beta_{n-m}+\alpha_{n-m}} & n \geq m. \end{cases} \tag{6.21}$$

This upper-bound error is attained by the following algorithm, reffered to as *the linear bound algorithm*: test at distance x^* from the right-hand side of largest relevant interval, where x^* is given by:

$$x^* = \begin{cases} \frac{\alpha_{n-m}-\alpha_{n-1}\beta_{n-1}t}{\alpha_{n-1}+\alpha_{n-m}} & t \geq t^* \\ \frac{\alpha_{n-m}+(\alpha_{n-m}\beta_{n-m}-\alpha_{n-1})t}{\alpha_{n-1}\beta_{n-1}+\alpha_{n-m}\beta_{n-m}+\alpha_{n-m}} & t \leq t^*, \end{cases} \tag{6.22}$$

where

$$t^* = \frac{\alpha_{n-m}}{\alpha_{n-m} + \alpha_{n-1} + \alpha_{n-1}\beta_{n-1}}. \tag{6.23}$$

This algorithm is shown to be *asymptotically optimal*, i.e.: A possible performance measure of a search procedure is $r_k = \lim_{n\to\infty} \left\{ v[s_k(n)]^{\frac{1}{n}} \right\}$. It measures the asymptotic reduction factor of the error per observation, i.e., if for budget of size n the error is ϵ_n, and n is large, then the error after the next observation is approximately $r_k \epsilon_n$. A lower bound for r_2 is computed, and it is shown that the linear bound algorithm achieves that lower bound, meaning that it is asymptotically optimal.

This "linear bound algorithm" is extended to the case $k > 2$, but the asymptotical optimality is not shown to be preserved. The resulting complexity is $O(nk \log k)$.

Efraimidis [36] **(2010)** generalizes the Fibonacci search to fit the dichotomous search problem with asymmetric error costs. The (a, b) *binary search problem* is defined as follows: An array V of N items sorted in increasing order is given. Each item of the array can be accessed in time $O(1)$. For any value x and any item v_k of V the cost of the comparison of x to v_k is a if $x \leq v_k$ and b otherwise. $a, b \in \mathbb{N}$. The cost of a search is equal to the sum of the costs of all comparison operations performed during the search. The objective is to minimize the worst-case cost of the search.

Given integers $a \geq 1$ and $b \geq 1$, the (a, b) *Fibonacci sequence* is given by the recurrence relation:

$$g(k) = g(k - a) + g(k - b), \tag{6.24}$$

with initial values $g(k) = 0$, for $k < 0$, and $g(0) = 1$.

The (a, b) *decision tree* is defined as follows: "The root node of the tree has level 0. Let v be a node at level d_v of the tree. Then node v is either a leaf of the tree, or it has a left child at level $(d_v + a)$ and a right child at level $(d_v + b)$. Note that the term level is used to represent the weighted depth of the tree nodes. Each level of the tree corresponds to a particular cost value. The *level of an* (a, b) *decision tree* is the maximum level of any of its nodes. The *depth of a node* is the common depth of tree nodes, i.e., the number of links from the root to the node. The *depth of an* (a, b) *decision tree* is the maximum depth of any of its nodes. The distinguishing property of (a, b) decision trees is that nodes which have the same parent may be located at different tree levels."

To obtain a lower bound for the worst case cost of the search, the author defines a sequence $G(k)$: given integers $a, b \geq 0$ and $l = \min\{a, b\}$ let:

$$G(k) = \sum_{i=1}^{l} g(k + 1 - i), \tag{6.25}$$

where $g(k)$ is the (a, b) Fibonacci sequence.

The number of nodes at level k of a complete (a, b) decision tree is the value of term $g(k)$ of corresponding (a, b) Fibonacci sequence of equation (6.24). The number of leaf nodes of a complete (a, b) decision tree of level k is $G(k)$, where $G(k)$ is the (a, b) Fibonacci sequence of equation (6.25). It turns out that the $G(k)$ sequence is also an (a, b) Fibonacci sequence but with its own initial values. The worst-case cost for an (a, b) binary search problem with N items is at least k, where k is the minimum index such that $G(k) \geq N$.

Based on that lower-bound, the following (a, b) Fibonacci search algorithm is presented:

Algorithm 2.
***"Input**: A sorted array V with n items and a requested item x in V. The*

items in V are indexed from 0 to $n-1$.
Step 0: *$(a,b)* **Fibonacci numbers**. Prepare the (a,b) Fibonacci numbers up to index k such that $G(k) \geq n$.*
Step 1: Initializations. *Let $z = k-a$, $left = 0$, $right = n-1$.*
Step 2: Search Loop.
While $(left < right)$
(a) $index = left + G(z)$; if $(index > right)$ then $index = right$;
(b) $value = v[index]$;
(c) Compare $(x \geq value)$:
true: $right = index$; $z = z - a$;
false: $left = index + 1$; $z = z - b$."

The algorithm approximates (this is not an optimization algorithm) the worst case cost of the asymmetric costs search problem and the corresponding (a,b) decision tree in time $O(\log N)$. The level of the implicit (a,b) decision tree obtained by the algorithm is at most $\max\{a,b\}\lceil \log N \rceil$.

6.2 Search with travel costs

Among many problems encountered in real life, there are cases where travel costs are not negligible and need to be considered. Commonly discussed versions of the dichotomous search problem are those in which the cost of the search depends not only on the number of queries, but also on the traveled distance during the search operation or on the direction of travel between each two sequential queries.

For example suppose an underground communication line is found to be cut off. A technician wants to locate a segment of the line, of unit length, that contains the cut off point, and replace it. Assuming that the line is cut off in exactly one point, it follows that from each point on it, it is possible to communicate with exactly one of its ends. This provides a mechanism for testing whether the cut off point is "before" or "after" the inspected point. Two costs are involved with the search: a travel cost, proportional to the traveled distance, and a query cost, proportional to the number of points in which a test is performed. The technician's objective is to locate the object, within a segment of unit length, with minimum expected cost (the example is taken from [57]).

A search is conducted in the interval of integers $[1,\ldots,N]$ (the initial *uncertainty interval*) and starts at the left endpoint 1. ax is the travel cost required for a searcher to move distance x at any stage of the search, to any direction. Each query costs b. a, b, N are all non-negative real numbers. After each query the search continues from the point of the last query, whether it is the left or the right end of the new uncertainty interval.

Murakami [103] **(1976)** views the problem of determining a sequence of queries so as to minimize the **maximum cost** required to diminish the existing interval of length N to unit length. The author constructs an explicit formulation of the optimal strategy and calculates the value of the appropriate objective function.

Let $h(N)$ denote the maximum cost of the search over all possible locations of the target. The following equation (6.26) is governing $h(N)$:

$$h(N) = \begin{cases} 0 & 1 \geq N \geq 0 \\ \min_{0 \leq x \leq n}[ax + b + \max(h(x), h(N-x))] & N > 1. \end{cases} \tag{6.26}$$

For a real number m define $g(m)$ as the unique integer such that $2^{g(m)} < m \leq 2^{g(m)+1}$. Then the solution of equation (6.26), i.e., the value of the objective function for the optimal policy is obtained by the following equation:

$$h(N) = \begin{cases} 0 & 0 < N \leq 1 \\ (N-1)\alpha + (g(N)+1)b & n > 1 \end{cases} \tag{6.27}$$

The optimal policy is described as follows: Having diminished the initial interval $[1, \ldots, N]$ to an interval of size n, execute your next search at point $S(n)$ which is given by:

When $n > 2$:

$$S(n) = \begin{cases} \{x | n - 2^{g(n)} \leq x \leq n/2\} & a, b > 0 \\ \{x | 0 \leq x \leq n/2\} & a > 0, b = 0 \\ \{x | n - 2^{g(n)} \leq x \leq 2^{g(n)}\} & a = 0, b > 0 \end{cases} \tag{6.28}$$

When $2 \geq n > 1$,

$$S(n) = \begin{cases} \{x | x = n - 1\} & a, b > 0 \\ \{x | 0 \leq x \leq n - 1\} & a > 0, b = 0 \\ \{x | n - 1 \leq x \leq 1\} & a = 0, b > 0 \end{cases} \tag{6.29}$$

A.J. Hu [65] **(1986)** develops a heuristic to solve a dichotomous search problem with travel costs. N, a and b are defined as earlier. The objective function discussed here is to minimize the expected (as opposed to minmax in [103]) total cost of the search. The a priori distribution of the object's location is assumed to be uniform. Hu limits the discussion to one family of search procedures - uniform partition search, thus conceding finding the optimal solution, but only the best solution out of the examined class. A uniform partition search is one which consists of recursively dividing the list $[1, \ldots, N]$ into sublists of uniform size (a sublist can be thought of as an interval of integers), traveling among sublists from left to right and conducting queries as "Does the desirable record lie in sublist x?". When we reach a first point of a sublist, we pay the appropriate price for traveling there and the cost of a query and are told if the object lies in that sublist. If it does, we narrow our search to that sublist, dividing it again. If it doesn't, we travel to the first point of the next sublist, pay the additional travel fee

and the cost of a query, conduct the following query and so on. It takes one read to discover that the desired record lies in the first sublist, two reads, in the second sublist and so forth. Travel costs are proportional to the traveled distance, i.e., if the searcher needs to move through sublists $(1, \ldots, k)$, each of size $\frac{N}{n}$, he shall pay $\frac{ak}{n}$ for the travel expenses. After finding the proper sublist, it is again partitioned and the search problem is diminished.

Some definitions are required: The integer p, $2 \leq p \leq N$, determines the search algorithm ranging from the traditional binary search, when $p = 2$, to the sequential search, when $p = N$, i.e., p denotes the number of parts the list is divided to. Also, the word "level" is used to denote partitioning the list into p parts, for example, one level of a search with $p = 7$ entails partitioning the list into seven sublists and finding the sublist which contains the desired record (the search among sublists is done one by one, starting with the first sublist); the next level consists of repeating this partitioning process on the sublist.

Since search cost is the sum of travel costs and read costs, we have

$$f(p) = br(p) + at(p). \tag{6.30}$$

where

- $f(p)$= the expected cost function
- $r(p)$= the expected number of reads needed to complete the search
- $t(p)$= the expected distance (in records) traveled to complete the search

$r(p)$ is computed the following way: Searching a list of N records will require $\lceil \log_p N \rceil$ levels, since each level produces a sublist which is $\frac{1}{p}$ the size of the list. One read is needed to discover that the desired record is in the first sublist; 2 reads - that it's in the second and so forth. However, if the desired record is in the p-th sublist, we will know it after $p - 1$ reads. Via the sum of arithmetic series, we get the expected number of reads:

$$r(p) = \lceil \log_p N \rceil \, \frac{(p+2)(p-1)}{2p}. \tag{6.31}$$

$\frac{1}{p}$ of the list is traveled through for each read. On the first level, the list is N records long, so $\frac{1}{p}$ of the list is $\frac{N}{p}$. Therefore the expected distance traveled in the first level is $\frac{N(p+2)(p-1)}{2p^2}$. Each successive level will have the same expected travel, except that it will be $\frac{1}{p}$ as much. The expected travel $t(p)$ is the sum of geometric series, i.e.,

$$t(p) = \frac{N(p+2)}{2p}. \tag{6.32}$$

Combining equations (6.31)-(6.32), the author finds $f(p)$:

$$f(p) = b\left\lceil \log_p N \right\rceil \frac{(p+2)(p-1)}{2p} + a\frac{N(p+2)}{2p}. \tag{6.33}$$

The cost function is the objective function, that we wish to minimize. By differentiating (6.33) and setting it equal to zero the author discovers an interesting relationship (6.34) which allows p to be computed easily from a, b, N.

$$\frac{p^2}{\ln(p)} = \left(\frac{a}{b}\right)\frac{(\sqrt{N})^2}{\ln(\sqrt{N})}. \tag{6.34}$$

Note that p is dependent of the initial N. After one iteration the list size is smaller than N, but p remains the same as earlier and is not calculated again for the new list size.

Hu and Wachs [73] **(1987)** introduce an $O(N)$ constructive algorithm for computing an optimal binary tree that minimizes the expected number of comparisons and movements for the case $a = b = 1$.

The authors characterize the optimal trees by giving explicit expressions for the sizes $\{m_1, ...m_k\}$ (for some k) of the subtrees that hang from the right-most path of the tree, as demonstrated in figure 6.

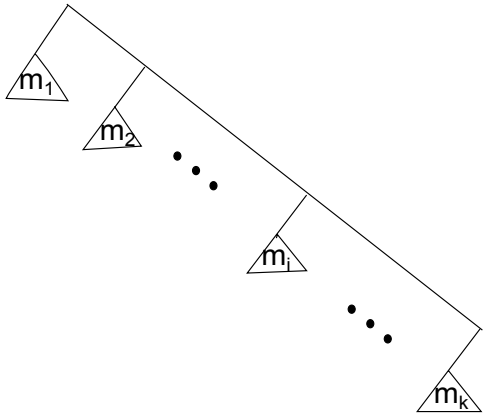

FIGURE 6. The tree $\langle m_1, \ldots m_k \rangle$.

For binary trees A and B, $A \wedge B$ denotes the tree whose left subtree is A and whose right subtree is B; $\bar{A}$ denotes the binary tree obtained from A by interchanging the left and right subtree of each node; $|A|$ denotes the number of nodes of A. Suppose $A = A_1 \wedge A_2$ has N nodes. For a binary tree A, let $T(A)$ denote the sum of the total expected number of movements needed to search for every node of A and the total expected number of comparisons. Then:

$$T(A_1 \wedge A_2) = \begin{cases} (|A_1|+2)N + T(A_1) + T(A_2) & \text{if } |A| = N > 0 \\ 0 & \text{if } |A| = 0 \end{cases} \tag{6.35}$$

A binary tree is defined to be *optimal* if $T(A) \leq T(A')$ for all A' such that $|A| = |A'|$. The *rightmost optimal* tree with N nodes is an optimal tree A such that if $A = A_1 \wedge A_2$ then A_1 and A_2 are rightmost optimal and if $B_1 \wedge B_2$ is optimal with N nodes, then $|A_2| > |B_2|$. The tree $[m_1] \wedge ([m_2 \wedge ([m_3] \wedge \ldots ([m_k] \wedge [0]) \ldots))$, where $[n]$ denotes the rightmost optimal tree with n nodes, will be denoted by $\langle m_1, \ldots m_k \rangle$ (see figure 6).

It is shown that if $\langle m_1, \ldots m_k \rangle$ is optimal then the m_i are non-increasing. Trees $\langle m_1, \ldots m_k \rangle$ for which $m_1 \geq m_2 \geq \cdots \geq m_k$ are called *trimmed trees*. For $m \geq 0$ and $a_i \geq 0$, $i = 0, \ldots, m$, let $(a_m, a_{m-1}, \ldots, a_0)$ denote the trimmed tree
$\langle m, m, \ldots, m, m-1, \ldots, m-1, \ldots, 0, 0, \ldots, 0 \rangle$ where each i appears a_i times. If $a_i > 0$ $\forall i = 0, \ldots, m$, the tree is called a *proper trimmed tree*.

It is proven that every optimal tree is equivalent to a unique proper trimmed tree $(a_m, a_{m-1}, \ldots, a_0)$. From now on the sequence of integers $A = (a_m, \ldots, a_0)$ is uniquely identified with a proper trimmed tree A.

Let $A = (a_m, \ldots, a_0)$ be a sequence of integers. We define a growing operator G:

$$G_i(A) = \begin{cases} (a_m, a_{m-1}, \ldots, a_i + 1, a_{i-1} - 1, \ldots, a_0) & \text{for } i = 0, \ldots\ m \\ (a_m, a_{m-1}, \ldots, a_0 + 1) & \text{for } i = 0 \\ (1, a_m - 1, a_{m-1}, \ldots, a_0) & \text{for } i = m+1 \end{cases} \tag{6.36}$$

Let $S = \{i_1, \ldots, i_j\} \subseteq \{0, \ldots, m+1\}$ and denote $G_S(A) := G_{i_1} G_{i_2} \ldots G_{i_j}(A)$.

Finally, the solution is the following: $w_0, w_1, w_2, \ldots$ are defined recursively in the article. For each $m \geq 0$, let $N_m = \sum_{i=0}^{m} (i+1) w_i$. Let $N_{m-1} \leq N < N_m$ and let d and r be determined by $N - N_{m-1} = d(m+1) + r$, where $d \geq 0$, $0 \leq r < m+1$. Then a tree with N nodes is optimal if and only if it is equivalent to $G_S(d, w_{m-1}, w_{m-2}, \ldots, w_0)$ for some $S \subset \{0, 1, \ldots, N\}$ with $|S| = r$. The rightmost optimal tree with N nodes is $(d, w_{m-1}, \ldots, w_{r-1} + 1, \ldots, w_0)$ if $r > 0$ and is $(d, w_{m-1}, \ldots, w_0)$ if $r = 0$.

Moreover, the cost of an optimal tree can be expressed in terms of w_i:

$$T(N) = \sum_{j=0}^{N-1} (w_j - 1)(N - j) = (N+1) \sum_{j=0}^{N-1} w_j - N_{N-1} - \frac{N(N+1)}{2}. \tag{6.37}$$

It follows that the cost of an optimal tree lies between $\frac{N(N+1)}{2}$ and $N(N+1)$. Since $N(N+1)$ is the cost of sequential search, we see that the optimal strategy is at best twice as fast as sequential search.

Moreover the authors show that in a more general search problem with travel costs, where each movement costs a and each comparison costs b, optimal trees can also be grown in a greedy fashion: If A is an optimal tree with N nodes, then an optimal tree with $N+1$ nodes can be obtained by attaching a leaf to A. This gives an $O(N)$ algorithm for finding optimal trees in the general case.

Nishihara and Nishino [104] **(1987)** consider the problem of dichotomous search with travel costs, where $a = 1$ and $b = 0$, i.e., moving one unit of distance costs 1, while conducting a query is free. The objective is to minimize the total expected cost of the search. The authors compare the performance of three algorithms: the ordinary binary search (BS), Fibonacci search (FS[10], given an interval of length k, the algorithm moves to the $(1-\varphi^{-1})k$-th key, where φ is the "golden ratio", and compares it to the search key.), and movement-minimizing Fibonacci search (mFS) - the latter algorithm is introduced in the article for the first time, while the two others are commonly known.

The FS algorithm builds a binary tree in which at each point of decision the sizes (number of nodes) of the two resulting subtrees form two successive Fibonacci numbers. The mFS algorithm modifies the basic FS algorithm, in a way that the amount of movement is kept as small as possible, by moving "lazily", trying to check a key placed close to the present head position.

Chung, Chen and Lin [30] **(1992)** analyze the same special case of the dichotomous search problem with travel costs as [104]. They study the expected costs of four search algorithms: the three introduced in [104] and sequential search. Assume that the sorted file that we search through contains $F_n - 1$ records, where F_n is the nth Fibonacci number. The authors show that the expected costs of the sequential search, BS, FS, and mFS are asymptotically equal to $0.5F_n$, F_n, $0.882F_n$ and $0.809F_n$ respectively.

Hornick, Madilla, Mucke, Rosenberg, Sol Skiena and Tollins [64] **(1990)** bring to our attention that the "simpler" problem discussed in [104] and [30] (where $a = 1$ and $b = 0$), to which the performances of FS and mFS algorithms are analyzed, has in fact an obvious optimal solution - the linear search. While the mFS algorithm has less expected head movement than the conventional binary search ($\approx 0.809N - o(N)$ versus $N - o(N)$, where N is the number of keys in the list), the linear search achieves an expected cost of $\frac{N}{2}$.

The authors investigate heuristics for the dichotomous search problem with travel costs where N is the length of the sorted list, $a, b \geq 0$, and the objective is to minimize the expected traveled distance and the cost of queries.

[10] see [37]

Recall that in [65] a family of heuristics for this the problem was examined. The objective function here in [64] is (similar to (6.30) in [65]):

$$\bar{S}_A(N, a, b) := a\bar{M}_A(N) + b\bar{C}_A(N). \tag{6.38}$$

where $\bar{M}_A(N)$ is the expected traveled distance and $\bar{C}_A(N)$ is the expected number of queries made by algorithm A.

The authors examine two classes of algorithms, fixed-ratio search and block searching, and point out the optimal solutions from each class.

A *fixed-ratio search* is parametrized by a fixed ratio $r \in (0, 1)$ (recall discussion of this family of search policies for the asymmetric search cost problem in [25]). Given an interval of length k, the algorithm moves to the rk-th key and compares it to the search key. For binary search $r = 0.5$, for the Fibonacci search in [104], $r = 1 - \varphi^{-1}$, where φ is the "golden ratio". It is shown that the expected total cost for a fixed-ratio search with parameter r is:

$$\bar{S}_{FR(r)}(N) = \frac{aN}{2(1-r)} + \frac{b \log N}{-r \log r - (1-r)\log(1-r)}. \tag{6.39}$$

By taking the derivative of (6.39) with respect to r one can find an asymptotically optimal value r^* for r is $r^* \approx 2\sqrt{\frac{b}{aN}}$. The value of the optimal fixed-ratio algorithm is:

$$\bar{S}_{FR(r)}(N) = a(N/2) + 2\sqrt{baN} + o(\sqrt{N}). \tag{6.40}$$

In a *block search* algorithm with parameter r we partition the data into blocks of size rN, and sequentially compare the last element of each block to the search key. Once we have found the block containing the key, any further search procedure is possible, and ones discussed here are the *block binary search (BB)* (i.e., subsequent partitioning of the relevant block into halves) and *block linear search (BL)*(i.e., searching the search keys in the relevant block one after another from left to right).

The optimal r for the BB asymptotically approaches $r^* \approx \sqrt{\frac{b}{3aN}}$ and the expected cost is given in (6.41), which is somewhat better than the optimal fixed ratio search.

$$\bar{S}_{BB(r^*)}(N) = a(N/2) + \sqrt{3baN} + o(\sqrt{N}). \tag{6.41}$$

The optimal r for the BL is $r^* = \sqrt{\frac{b}{N(2a+b)}}$ and the optimal expected cost is:

$$\bar{S}_{BL(r^*)}(N) = a(N/2) + \sqrt{b(2a+b)N} + \frac{a}{2} + b. \tag{6.42}$$

The BL search is asymptotically better than BB search if and only if $b < a$, but it can never achieve expected total time less than $a\frac{N}{2} + \sqrt{2abN}$.

Hassin and Hotovely [57] **(1992)** develop heuristics to the search problem on the continuous interval $[0, N]$ with travel costs. Each query costs b and travel per unit distance costs a. The problem is to determine a sequence of queries that minimizes the expected sum of query and travel costs required to locate the object within an interval of unit length. Let $F(N)$ be the expected cost function. $F_\pi(N)$ is the expected cost associated with searching an interval of length N while using policy π at each decision point. A policy π is called *asymptotically optimal* if $\lim_{N\to\infty} \frac{F_\pi(N)}{F(N)} = 1$. If an arbitrary function $g(N)$ is an approximation for another function $f(N)$, then the *relative error* of this approximation is defined as $\left|\frac{f(N)-g(N)}{f(N)}\right|$. The authors analyze the performance of several simple approximations to the optimal policy: fixed-step policies, fixed-ratio policies and myopic policies.

A *fixed step policy* is one where the searcher advances by a fixed distance as long as the direction of his movement is fixed. The authors prove that a step of size $\sqrt{\frac{Nb}{a}}$ is asymptotically optimal for this class of policies, and its relative error converges to 0 as fast as $\frac{1}{\sqrt{N}}$. The expected cost of this policy, $h(N)$, satisfies

$$h(N) = 0.5aN + \sqrt{Nba} + o\left(\sqrt{N}\right). \tag{6.43}$$

A *fixed ratio policy* is characterized by a number $p \in (0, 1)$. Given an interval of size $N > 1$ that contains the object, the next query is placed at a distance of Np from the endpoint in which the searcher is located. The fixed-ratio policy with $p = \frac{2}{\sqrt{N}}\sqrt{\frac{b}{a}}$ is asymptotically optimal and its cost $h(N)$ also satisfies (6.43). In [64] the same result is achieved (see (6.40)). The proof of this result here in [57] uses the notion of *binary entropy* $H(p) = -p\log p - (1-p)\log(1-p)$. It is shown that the expected number of queries for a fixed ratio policy with parameter p, $S(N, p)$, admits the following approximation for $p < \frac{1}{3}$:

$$\left|S(N,p) - \frac{\log N}{H(p)}\right| < \frac{1}{pH(p)} \log\frac{4}{p}. \tag{6.44}$$

As a corollary, the relative error of the approximation $\frac{\log N}{H(p)}$ to $S(N, p)$ converges to 0 at least as fast as $\frac{1}{\log N}$.

Let $D(N,p)$ be the expected travel distance under a fixed ratio policy with parameter p. It is shown that the relative error of the function $d(N,p) = \frac{N-1}{2(1-p)}$ to $D(N,p)$ converges to 0 at least as fast as $\frac{\log N}{N}$.

The authors define $\widehat{F}(N,p) = b\frac{\log N}{H(p)} + ad(N,p)$, take a derivative with respect to p, equate to zero and let $N \longrightarrow \infty$, thus obtaining $p = \frac{2}{\sqrt{N}}\sqrt{\frac{b}{a}}$ and $\widehat{F}(N,p) \approx \sqrt{Nab} + \frac{aN}{2}$, which is asymptotically optimal.

In fact, [57] discover that by a fixed-step policy one can get a result with is asymptotically similar to the best fixed-ratio policy.

Myopic policies intend to maximize the reduction in the interval's length per unit of cost associated with the first query. By investing $b+xa$ we reduce the size of the interval to x with probability $\frac{x}{N}$, and to $N-x$ with probability $\frac{N-x}{N}$. A myopic policy is one that maximizes the expected reduction in the size of the interval $\frac{x}{N}(N-x) + \frac{N-x}{N}x$ per unit of cost. The optimal result for this class of policies is achieved by a first step of $\left\lfloor \sqrt{\frac{Nb}{a}} \right\rfloor$. The cost of this policy is of order $aN + 2\sqrt{Nba}$.

Wachs [125] **(1989)** considers the following problem: A search is held for an object at locations $\{\alpha_1, \ldots, \alpha_N\}$, $0 \le \alpha_1 \le \alpha_2 \le \cdots \le \alpha_N$ and it can be found at one of those locations or between two subsequent locations. Let p_i be the probability that the object lies at α_i and q_i the probability that the search is unsuccessful with the search argument lying between α_i and α_{i+1} (with corresponding definitions for q_0 and q_N). In order to make a query at a particular location we must first move from the last location at which a query was made to the given location. Each query reveals if the object lies at the searched location, to its left or to its right. The travel cost for a unit distance is $a > 0$, i.e., the cost of traveling from α_i to α_j is $a\,|\alpha_i - \alpha_j|$. Each comparison costs b. The goal is to find a search strategy with expected minimal cost.

Let $u(i,j) = a(\alpha_j - \alpha_i) + b$. The cost of moving from α_i to α_j and conducting a query at α_j is $u(i,j)$ if $\alpha_i < \alpha_j$ and $u(j,i)$ if $\alpha_i > \alpha_j$. The cost of the first query at α_k is $u(0,k)$. Let $c'(i,j)$ be the cost of an optimal search strategy for the reduced problem on set of locations $\{\alpha_i, \ldots, \alpha_j\}$, $i < j$, such that the search begins at α_i. Let $d'(i,j)$ be the cost of an optimal search strategy for the reduced problem on set of locations $\{\alpha_i, \ldots, \alpha_j\}$, $i < j$, such that the search begins at α_j. Let $w(i,j) = p_{i+1} + \cdots + p_{j-1} + q_i + \cdots + q_{j-1}$ for $0 \le i < j \le N+1$.

The dynamic programming recurrence relations can be expressed as:

$$
\begin{array}{ll}
c'(i,j) = \min_{i<k<j}\left[u(i,k) + \frac{w(i,k)}{w(i,j)}d'(i,k) + \frac{w(k,j)}{w(i,j)}c'(k,j)\right] & \text{for } i < j-1 \\
c'(i,j) = 0 & \text{for } i = j-1 \\
d'(i,j) = \min_{i<k<j}\left[u(k,j) + \frac{w(i,k)}{w(i,j)}d'(i,k) + \frac{w(k,j)}{w(i,j)}c'(k,j)\right] & \text{for } i < j-1 \\
d'(i,j) = 0 & \text{for } i = j-1
\end{array}
$$

(6.45)

By setting $c(i,j) = w(i,j)c'(i,j)$ and $d(i,j) = w(i,j)d'(i,j)$ we get

$$\begin{array}{ll} c(i,j) = \min_{i<k<j} [u(i,k)w(i,j) + d(i,k) + c(k,j)] & \text{for } i < j-1 \\ c(i,j) = 0 & \text{for } i = j-1 \\ d(i,j) = \min_{i<k<j} [u(k,j)w(i,j) + d(i,k) + c(k,j)] & \text{for } i < j-1 \\ d(i,j) = 0 & \text{for } i = j-1 \end{array} \tag{6.46}$$

Indeed, k minimizes (6.45) for c' or d' if and only if α_k is the first query of an optimal search strategy for the reduced problem on $\{\alpha_i, \ldots, \alpha_j\}$. If the object lies to the left of α_k, the additional cost is $d'(i,k)$, while if it lies to the right of α_k it is $c'(k,j)$, and the cost of the first query is $u(i,k)$ for c and $u(k,j)$ for d. The straightforward dynamic programming algorithm determines the minimizing k for both cost functions c and d and all i,j and therefore constructs the optimal tree in time $O(N^3)$.

Some more definitions are needed in order to explain the technique of reduction of this complexity to $O(N^2)$:

- "A real valued function $f(i,j)$, where $0 \le i \le j \le n+1$, is said to satisfy the *quadrangle inequalities* if $f(i,j) + f(i',j') \le f(i,j') + f(i',j)$ for $i \le i' \le j \le j'$. Such a function is also called supermodular.
- The function $f(i,j)$ is said to be *monotone* if $f(i',j) \le f(i,j')$ for $i \le i' \le j \le j'$.
- $f(i,j)$ is said to be *standard-monotone* if $f(i,j) \le f(i',j')$ for $i \le i'$ and $j \le j'$."

Yao [131] **(1980)** proved the following general result, which was applied to the problem of binary search trees among numerous other examples. Let the following recurrence equations describe a solution to an optimization problem:

$$\begin{array}{l} c(i,i) = 0 \\ c(i,j) = w(i,j) + \min_{i<k\le j} \{c(i,k-1) + c(k,j)\} \quad \text{for } i < j, \end{array} \tag{6.47}$$

for $1 \le i,j \le n$. If the increment function w is monotone and satisfies the quadrangle inequalities, then the function c defined by (6.47) can be computed in $O(n^2)$.

In the dichotomous search problem, discussed above, where $u(i,j)$ is identically 1 ($a = 0, b = 1$) and $w(i,j)$ defined as earlier, the increment function w is monotone and satisfies the quadrangle inequalities (in fact as equalities). For $u(i,j) \equiv 1$ equation (6.46) has the form of equation (6.47), thus Yao's result can be applied to this problem.

Wachs extends this idea to a problem where $u(i,j)$ is not identically 1. In order to do that he proves the next two results:

Result 1: "Let $u(i,j)$ and $w(i,j)$, $0 \le i \le j \le n+1$, be monotone functions that satisfy the quadrangle inequalities. If $u(i,i) \ge 0$ for all i and $w(i,j) \ge 0$ for $i < j$ then the functions $c(i,j)$ and $d(i,j)$ satisfy the *strong quadrangle inequalities* for $i \le i' < j \le j'$:

$$c(i,j')+c(i',j)-c(i',j')-c(i,j) \ge \big(u(i,i') - u(i',i')\big)\big(w(j,j') - w(j,j)\big)\,, \tag{6.48}$$

$$d(i,j')+d(i',j)-d(i',j')-d(i,j) \ge \big(u(j,j') - u(j,j)\big)\big(w(i,i') - w(i',i')\big)\,. \tag{6.49}$$

It is shown that the assumptions of Result 1 hold for the cost functions c' and d'."

Result 2: "Let $k_c(i,j)$ and $k_d(i,j)$ be the roots of the optimal left and right subtrees: $k_c(i,j) = \max\{k \,|c(i,j) = u(i,k)w(i,j) + d(i,k) + c(k,j)\}$, $k_d(i,j) = \max\{k \,|d(i,j) = u(k,j)w(i,j) + d(i,k) + c(k,j)\}$. Under the conditions of Result 1, the functions $k_c(i,j), k_d(i,j)$ are standard-monotone. Moreover, the functions $k_c(i,j), k_d(i,j), c(i,j), d(i,j)$ can be computed in $O(N^2)$ time."

6.3 Different costs for search at different points

Knight [86] **(1988)** analyzes the following problem: An item is searched for in the set of locations $\{1, 2, \ldots, N\}$, and it is equally likely to be found in any of them or not to be found at all (each possibility has probability $\frac{1}{N+1}$). The cost of search at point k is $P(k)$. Let T_N be the binary search tree[11] corresponding to a unique search strategy: if the answer is yes, go left; if the answer is no, go right; stop only when the location of the item is identified, which happens when a query occurs at a parent of a leaf or at a leaf. There are N internal nodes, at which the item may be located, and $N+1$ leaves. The situation that the last query occurs at a leaf corresponds to an unsuccessful search (otherwise the last query occurs at a parent of a leaf). $W_k(T_N)$ denotes the number of internal nodes in the subtree of T_N rooted at k. The expected cost of the search using T_N is $\frac{1}{N+1}\sum_{k=1}^{N} P(k)W_k(T_N) + \frac{1}{N+1}\sum_{k=1}^{N} P(k)$. For computational convenience the constant factor $\frac{1}{N+1}$ and the constant term $\sum_{k=1}^{N} P(k)$ are discarded, and the remaining sum is called the *search cost of* T_N and denoted $S_{P(k)}(T_N)$:

[11] Recall the definition of a binary search tree from §2

$$S_{P(k)}(T_N) = \sum_{i=1}^{N} P(i) W_i(T_N). \tag{6.50}$$

The author calculates **upper and lower bounds** for $S_{P(k)}(T_N)$ for various functions $P(k)$. For $P(k) \equiv 1$, the lower bound for every search tree T_N is given by (6.51):

$$S_1(T_N) \geq (N+1)\left[\log(N+1) - 1\right] + 1. \tag{6.51}$$

For linear inspection cost function $P(k) = k$, the lower bound is given by (6.52):

$$S_k(T_N) \geq \frac{1}{2}(N+1)^2 \left[\log(N+1) - 1.070\right] + \frac{1}{2}(N+1). \tag{6.52}$$

For any linear inspection cost function $P(k) = \alpha k + \beta$ $(\alpha, \beta \geq 0)$, a lower bound is (6.53):

$$S_{\alpha k+\beta}(T_N) \geq \frac{\alpha}{2}(N+1)^2 \left[\log(N+1) - 1.070\right] +$$

$$\frac{\alpha}{2}(N+1) + \beta\left[(N+1)\log(N+1) - N\right]. \tag{6.53}$$

When $P(k)$ is linear, ordinary binary search gives near-optimal results (the near-optimality is shown by extensive computer calculations): Let $B(N)$ be the tree corresponding to the ordinary binary search strategy. Then for all $(\alpha, \beta \geq 0)$ we have:

$$S_{\alpha k+\beta}(B_N) \leq \frac{\alpha}{2}(N+1)^2 \left[\log(N+1) - 0.9138\right] + \frac{\alpha}{2}(N+1) +$$

$$+\beta\left[(N+1)(\lfloor\log(N+1)\rfloor + 1) - 2^{\lfloor\log(N+1)\rfloor+1} + 1\right]. \tag{6.54}$$

For a nonlinear polynomial function $P(k) = k^p$ where p is a positive integer:

$$S_{k^p}(T_N) \geq \frac{1}{p+1}(N+1)^{p+1}\log(N+1) - (N+1)^{p+1}. \tag{6.55}$$

Again, the ordinary binary search (with the corresponding tree $B(N)$) is nearly optimal:

$$S_{k^p}(B_N) \leq \frac{1}{p+1}(N+1)^{p+1}\left[\log(N+1) - 0.9138\right] + \frac{1}{2}(N+1)^p. \tag{6.56}$$

6.4 Minimizing the sum of errors

Yet another version of a search cost is discussed in [17] by Baston and Bostock and in [5] by Alpern. A searcher wishes to locate point $H \in [0,1]$ by successive guesses $g_1, g_2, ...$, each with the knowledge of whether the previous guess was high or low. When putting a guess, the searcher knows the results of all previous guesses. The cost function is the "sum of errors" $c = \sum_{i=1}^{\infty} |g_i - H|$. The discussed approach considers the problem as a two-person (a hider and a searcher) zero-sum game, with cost as a payoff. The searcher's goal is to **minimize the maximum cost** of the search, while the hider's goal is to **maximize the minimum payoff**.

Baston and Bostock [17] **(1985)** found a pure search strategy that guaranteed that the searcher's cost does not exceed 0.628. Moreover they showed that the hider can always ensure an expectation of at least 0.6. They did not prove that such a game always has a value, but only that if such value exists then it lies in an interval of length $0.028 = 0.628 - 0.6$.

The main result of **Alpern** [5] **(1985)** is the proof that the game has a value and the construction of the approximately optimal search strategy. The optimal minimax strategy has a simple description in terms of a sequence $\{\lambda_k\}_{k=-\infty}^{\infty}$ of constants with $\frac{1}{2} = \lambda_0 < \lambda_1 < \cdots < 1$ and $\lambda_{-k} = 1 - \lambda_k$. Suppose (a,b) is the interval of uncertainty after the guesses $g_1, \ldots, g_n$ have been made. Suppose that of these n numbers, k more of them have been to the left of (a,b) than to the right. Then $g_{n+1} = (1-\lambda_k)a + \lambda_k b$. Approximate values of the λ_i are also constructed.

Such a search game has been adapted to suit economic models of production and of wage bargaining by Alpern and Snower, and both of these models have been extended by Reyniers (see §14).

6.5 Maximazing the probability of finding a hidden object

Berry and Mensch [24] **(1986)** consider the following problem: The object is hidden in one of N cells ($0 < N \leq \infty$). p_i is the probability that the object is in cell i; The act of searching a cell for the object is called a "look".

A strategy σ is an infinite sequence of positive integers that indicates the order to search the cells. The probability of finding the object within n looks when following strategy σ is called the n-return of σ and written $r_n(\sigma)$. The maximal n-return is written r_n^* and a strategy σ is *n-optimal* if $r_n(\sigma) = r_n^*$.

The author introduces a class of bisection strategies. Define the *n-bisection region* as the set of all cells if $2^n - 1 > N$, and as the set of cells numbered 2^{n-1} to $N - (2^{n-1} - 1)$ if $2^n - 1 \leq N$. Define an *n-bisection strategy* as one that always allocates the next look to any site in the current bisection region and define the *n-coverage* of a strategy σ as the set of sites such that an object in one of the sites would be found within n looks with probability 1. Let $s_n(\sigma)$ be the number of sites in the n-coverage of strategy σ. If σ is any n-bisection strategy then $s_n(\sigma) = \min\{N, 2^n - 1\}$.

The main result states the following: Suppose n looks are available. Then a strategy is optimal if and only if it is an n-bisection strategy on a set of sites with the $2^n - 1$ largest probabilities, and r_n^* is the sum of $2^n - 1$ largest probabilities. For example if $p_i = 1/N$ for $i = 1, \ldots, N$ (uniform distribution), then any n-bisection strategy on any set containing $\min\{N, 2^n - 1\}$ sites is optimal and $r_n^* = \min\left\{1, \frac{2^n-1}{N}\right\}$. For a geometric case $p_i = (1-p)p^{i-1}$, the sites with $2^n - 1$ largest probabilities are $\{1, 2, \ldots, 2^n - 1\}$, and so $r_n^* = 1 - p^{2^n-1}$.

6.6 Alphabetic trees with non-constant leaf costs

Fujiwara and Jacobs [40] **(2010)** analyze the optimal alphabetic tree problem, where the cost of each leaf is not constant, but rather dependent on its depth. Assume that the cost of leaf v_i, having distance $l(i)$ from the root is determined by $f_i(l(i))$, where $f_i : \mathbb{N}_0 \longrightarrow \mathbb{R}_+^0$ is an arbitrary function. There are N cost functions $f_1, \ldots, f_N$, one for each leaf. The *General Cost Alphabetic Tree Problem* (GAT) is defined as follows: Given N arbitrary functions $f_1, \ldots, f_N$ the objective of GAT is to determine a binary tree T whose leaves in left-to-right order are $v_1, \ldots, v_N$ such that $\sum_{i=1}^{N} f_i(l(i))$ is minimized.

The authors show that an extension of the Gilbert-Moore Algorithm, based on dynamic programming, solves the general cost alphabetic tree problem in time $O(N^4)$ and space $O(N^3)$, regardless of the cost functions. Speedup of the algorithm by a factor of N is possible if the cost function fulfills one of the following properties: *subtree optimality* or *structural continuity*.

- *Structural Continuity.* The property of structural continuity was proven by Knuth [87] to hold for the classical alphabetic tree problem[12]. It roughly states that the root of an optimal alphabetic tree can only move left when the interval under consideration is extended to the left.
- *Subtree Optimality.* This property states that an optimal tree for $f_1 \ldots, f_N$ is a combination of optimal trees for $f_1, \ldots, f_i$ and $f_{i+1}, \ldots, f_N$ for some $i \in \{1, \ldots, N\}$. By induction it follows that for any internal node v in an optimal alphabetic tree, the subtree under v is an optimal alphabetic tree for the sequence of leaves that are descendants of v.

[12] In §4 Hassin and Henig provide more conditions for the existence of the monotonicity property in the classical alphabetic tree problem

The speedups are independent from each other, so problem instances whose cost functions satisfy both properties admit a $O(N^2)$ time optimal algorithm. The authors prove that if the cost functions are nondecreasing and convex, then the property of structural continuity is satisfied. Therefore, this case can be solved in time $O(N^3)$.

6.7 Alphabetic minmax trees

In this subsection we present results concerning alphabetic minmax trees, rather than alphabetic trees with minimal path length. By an alphabetic minmax tree we mean a binary (or t-ary) alphabetic tree for which the maximum value of an objective function of the path lengths and the weights of the leaves is minimized. Several objective functions have been considered in the literature, as $\max_i w_i 2^{l(i)}$ or $\max_{1\le i\le N}\{w_i + l(i)\}$. Note that the minimal value of $\max_i l(i)$ is $\lceil \log N \rceil$. Surprisingly, we found no literature that explores the minimum, of $\max_i w_i l(i)$.

Hu, Kleitman and Tamaki [67] **(1979)** investigate the problem of finding an alphabetic binary tree that minimizes $f = \max_i w_i 2^{l(i)}$. They find that the Hu-Tucker [71] algorithm for the minimal weighted path length alphabetic tree problem can be modified to minimize the other objective function and thus obtain a $O(N \log N)$- time algorithm for the alphabetic minmax problem. The authors note that the modification of the Hu-Tucker algorithm for the alphabetic binary minmax problem works also for finding the alphabetic ternary minmax tree.

Due to the complexity of the algorithm on the one hand and to similarity to the original Hu-Tucker algorithm on the other hand we do not elaborate on it here. Yet, though out of the scope of our survey, we note that for the non-alphabetic version of the minmax problem the modification of the famous Huffman algorithm is rather simple: Given a set of weights $w_1, \ldots, w_N$ construct a binary tree such that $\max_i w_i 2^{l(i)}$ is minimized. Among the N weights $w_1, \ldots, w_N$ find the two smallest weights w_1 and w_2, say. Replace the two nodes by one single node having weight $t \cdot \max\{w_1, w_2\}$ and two children with weights w_1 and w_2. Do this recursively for the $N-1$ weights $\{t \cdot \max\{w_1, w_2\}, w_3, \ldots, w_N\}$. The final single node is then the root of the binary tree. Thus the only difference between this algorithm and the original Huffman algorithm lies in the nature of the replacement step that assigns a new weight to the merged node.

From now on to the end of this subsection, we relate to the following problem (*t-ary Problem A*): Given vertices $v_l, \ldots, v_N$ with weights $w_1, \ldots, w_N$, construct a t-ary tree with leaves $v_l, \ldots, v_N$ in left to right order, such that if $l(i)$ denotes the length of the path from v_i to the root for each i, the function $f(w_1, \ldots, w_N) = \max_{1\le i\le N}\{w_i + l(i)\}$ is minimized. An equivalent version of the problem is to consider weighted t-ary trees in which the weight of each internal vertex is 1+ the maximum of the weights of its children. In this formulation we are trying to construct a weighted t-ary tree with leaves $v_l, \ldots, v_N$ in left to right order, such that the weight of the root is minimized.

Kirkpatrick and Klawe [83] **(1985)** present a linear time algorithm for a t-ary Problem A, for the case where all the weights are integers, and this is used to obtain an $O(N \log N)$ algorithm for general weights. Moreover it is shown that the minmax value obtained is bounded above by $2+\log_t(\sum t^{w_i})$. Also, if desired, by solving k versions of the integer problem, with overall complexity $O(kN)$, one can approximate the solution to a general alphabetic problem with error at most $\frac{1}{2^{k-1}}$.

Coppersmith, Klawe and Pippenger [32] **(1986)** relax the constraint on the degrees of internal vertices of the tree - while in [83] all internal nodes need to be of degree t, in [32] all internal nodes must have a degree no larger than t. As in [83] the authors obtain a linear algorithm for the case of integer weights, a $O(N \log N)$ algorithm for the case of real weights and prove a tight upper bound on $f(w_1, \ldots, w_N)$ in terms of $w_1, \ldots, w_N$:

$$f(w_1, \ldots, w_N) < 1 + \log_t 2 + \log_t \left(\sum_{i=1}^{N} t^{w_i} \right). \tag{6.57}$$

Gagie [41] **(2009)** develops an $O(Nd \log\log N)$-time algorithm for t-ary Problem A with real weights, where d is the cardinality of $\{ \lfloor w_i \rfloor \mid i = 1, \ldots, N \}$. This is an improvement of [83] when d is small. The algorithm uses a data structure, developed especially for it, that enables to avoid some sorting operations, and thus improve the $O(N \log N)$ result of [83]. We do not elaborate on this algorithm here.

The author brings an example for the motivation to investigate the min-max alphabetic tree problem - a problem concerning alphabetic prefix codes (recall the definition from §2): "Suppose we want to build an alphabetic prefix code with which to compress a file (or, equivalently, a leaf-oriented binary search tree with which to sort it), but we are given only a sample of its characters. Let $P = p_1, \ldots, p_N$ be the distribution of characters in the file, let $Q = q_1, \ldots, q_N$ be the distribution of characters in the sample, and suppose our codewords are $C = c_1, \ldots, c_N$. An ideal code for Q assigns the ith character a codeword of length $\log\left(\frac{1}{q_i}\right)$ (which may not be an integer), and the average codeword's length using such a code is $H(P) + D(P \parallel Q)$, where $H(P) = \sum_i p_i \log \frac{1}{p_i}$ is the entropy of P and $D(P \parallel Q) = \sum_i p_i \log \frac{p_i}{q_i}$ is the relative entropy between P and Q. Consider the best worst-case bound we can achieve on how much the average codeword's length exceeds $H(P)+D(P \parallel Q)$. As long as $q_i > 0$ whenever $p_i > 0$, the average codewords length is:

$$\begin{aligned} \sum_i p_i |c_i| &= \sum_i p_i \left(\log \frac{1}{p_i} + \log \frac{p_i}{q_i} + \log q_i + |c_i| \right) = \\ &= H(P) + D(P \parallel Q) + \sum_i p_i \left(\log q_i + |c_i| \right). \end{aligned} \tag{6.58}$$

where the first equation is true since $\log \frac{1}{p_i} + \log \frac{p_i}{q_i} + \log q_i = 1$ (if $q_i = 0$

but $p_i > 0$ for some i, then the formula is undefined). Notice that each $|c_i|$ is the length of the ith branch in the tree for C. Therefore, the best bound we can achieve is

$$\begin{aligned} \min_C \max_P \left\{ \sum_i p_i \left(\log q_i + |c_i| \right) \right\} &= \min_C \max_i \left\{ \log q_i + |c_i| \right\} = \\ &= f\left(\log q_1, \ldots, \log q_N \right). \end{aligned} \tag{6.59}$$

and we achieve it when the tree for C is an alphabetic minmax tree for $\log q_1, \ldots, \log q_N$." To understand the last equation recall that $f(w_1, \ldots, w_N) = \max_{1 \le i \le N} \{w_i + l(i)\}$ and note that $c(i)$ has in fact the same interpretation as $l(i)$ in terms of trees.

Gawrychowski [46] **(2012)** develops a linear time algorithm for 2-ary Problem A, which improves the previously known $O(N \log N)$ algorithms of [83] and [32] and the $O(Nd \log\log N)$ algorithm of [41].

The author starts with a simple linear time algorithm for the case when all w_i are integers and then develops an $O(Nd)$ time algorithm for the more general case when w_i are arbitrary real numbers, with d being the cardinality of $\{\lfloor w_i \rfloor \mid i = 1, \ldots, N\}$. Then the complexity is improved to linear using the word RAM model [13].

6.8 Alphabetic trees with exponential costs (alphabetic minsum trees)

In this section we describe a variation on the optimal alphabetic tree problem, where the objective function is a non-linear function of the path lengths ($l(i)$'s), rather than a linear combination of them. We consider the problem of finding an alphabetic tree that minimizes $\log_a \sum_{i=1}^N w_i a^{l(i)}$ rather than $\sum_{i=1}^N w_i l(i)$.

Three papers ([67], [107], [75]) independently consider this problem for $a > 1$ for non alphabetic trees, the solution of which is very similar to that of Huffmans algorithm. [67] further notes that an algorithm similar to Hu and Tucker's solves the alphabetically constrained version of this problem. Baer [15] shows that it is not always correct.

Baer [15] (2010) presents an $O(N^3)$ time and $O(N^2)$ space algorithm for the alphabetic version that is somewhat similar to Gilbert and Moore's method [48] for the alphabetic weighted path length problem. The algorithm is based on dynamic programming formulation of the problem. Let $W_{j,k}$ be the maximum tree weight for items j through k. The algorithm starts with

[13] By the word RAM model we mean that we are allowed to perform operations on integers consisting of $\log N$ bits in constant time. No specific encoding of the real numbers given in the input is assumed, but we do require that their representation allows performing a few basic operations (comparing, subtracting, extracting the fractional part and rounding to an integer if its value does not exceed N) in constant time.

$W_{j,j} = w_j$ and finds $W_{j,k}$ for each value of $k - j$ from 0 to $N - 1$ (in order), by computing inductively:

$$W_{j,k} = a \max_{s \in \{j+1, j+2, \ldots, k\}} [W_{j,s-1} + W_{s,k}] . \tag{6.60}$$

The author shows that the Knuth's method for speeding up dynamic programming fails for $a < 1$ and provides the following counter-example: "Define the *splitting point* of an internal node (or the corresponding subtree) as the smallest index among the leaves of the right subtree. Knuth uses the fact that the splitting point of an optimal tree of size N must be between the splitting points of the two optimal subtrees of size $N - 1$. With the discussed problem this no longer holds. Consider $a = 0.6$ with input weights w = (8, 1, 9, 6). The splitting point of $(8, 1, 9)$ is $s = 3$ ($w(s) = w(3) = 9$, yielding subtrees with $(8, 1)$ and (9)), and the splitting point of $(1, 9, 6)$ is $s = 4$ ($w(s) = 6$). However, the optimal splitting point of $(8, 1, 9, 6)$ is $s = 2$ ($w(s) = 1$)."

The author also shows, using a counter-example, that the Hu-Tucker-like method, that is optimal for $a > 1$ does not work for $a < 1$.

Finally, approximation algorithms are built, related to those for the linear problem, which find suboptimal solutions in $O(N)$ and $O(N \log N)$, leading to simple bounds for both these solutions and the optimal ones.

7. The depth restricted problem

In some applications an additional restriction is necessary to binary alphabetic trees of minimum weighted path length problem. In this subsection we consider the optimal binary alphabetic tree problem with the restriction that no depth of a leaf is permitted to exceed a given bound K. In terms of dichotomous search, a restricted depth limitation means that after at most K queries the object must be found.

Garey [44] attended this problem in 1974 for non-alphabetic (Huffman) trees. He developed an algorithm that requires $O(KN^2)$ operations to find such an optimal tree, thus improving earlier results of Karp, Gilbert, Hu and Tan [70]. An $O(KN^3)$ algorithm easily results from a dynamic programming formulation of the problem, but the author's innovation is an improvement of the application to $O(KN^2)$. This result was further improved in 1987 by Larmore [89], who cut the running time for the non-alphabetic case to $O(N^{1.5}K\sqrt{\log N})$.

Since our focus in this paper is on the *optimal alphabetic binary tree* problem, we do not describe Garey's and Larmore's solution in detail. Moreover, we do not conduct the appropriate research to find out whether Larmore's result was further improved. Yet we note that in his paper Garey posed an "open problem" - the constructing of an optimal K-restricted alphabetic binary tree. He noted that the basic $O(KN^3)$ algorithm can be easily adapted

to the alphabetic case and posed a question whether it can be further improved.

Itay (1976) provides such an improvement in [77] - an $O(KN^2)$ algorithm for the depth restricted alphabetic version. Recall the notation $|T|$ of the weighted path length (cost) of a tree T. Let T be an optimal tree, v an internal node of T of depth $l(v)$. Then the subtree of T with root v is an optimal $(K - l(v))$-restricted alphabetic tree for its own sequence of weights.

The basic $O(KN^3)$ algorithm, introduced by Garey and by Itay is the following: Let $[i, j, k]$, $1 \le i \le j \le n$ and $0 \le k \le K$, denote the subproblem of finding an optimal tree of depth at most k for $(w_i, w_{i-1}, ..., w_j)$. Let $T[i, j, k]$ be the optimal alphabetic tree for the restricted problem $[i, j, k]$. If $k < \lfloor \log_2(j - i + 1) \rfloor$, then no solution exists, and we set $|T[i, j, k]| = \infty$. Otherwise set:

$$\begin{array}{l} |T[i,i,k]| = 0 \quad i = 1, 2, \ldots, n \\ |T[i,j,k]| = \sum_{r=i}^{j} w_r + \min_{i \le b < j} \{|T[i,b,k-1]| + |T[b+1,j,k-1]|\}, \\ i = 1, \ldots, n-1, \quad j = i+1, \ldots, n \end{array} \tag{7.1}$$

The value b_0, for which (7.1) produces the minimum, is the last node (in postorder) of the left subtree and is called the *breakpoint*. If b_0 is a breakpoint for $[i, j, k]$, we can construct an optimal tree for $[i, j, k]$ by building an optimal tree for $[i, b_0, k-1]$ and $[b_0+1, j, k-1]$ and combining the two trees. Solving $[i, j, k]$ for all i, j, k results in an optimal tree $T[1, n, K]$ for the original problem. The resulting algorithm requires execution time $O(KN^3)$.

For a given K, the necessary conditions for implying the monotonicity property (the "Knuth method" described in §2) hold. This enables to reduce complexity to $O(KN^2)$. The Knuth method is widely explored in [56], which is covered in §4.

In [129] **Wessner (1976)** gives an algorithm for optimal binary search trees (recall the definition of a binary search tree from §2). The algorithm is a modification to optimal binary search trees of Garey's [44] algorithm for an optimal non-alphabetic binary trees, and runs in time $O(KN^3)$ (here the total number of nodes is $2N + 1$). The author shows that the "Knuth method" can be applied to the optimal binary search tree problem, thus cutting the running time to $O(KN^2)$. The methods in [129] and [77] are quite similar - the articles were published simultaneously and are products of independent work.

Larmore and Przytycka [92] **(1994)** use the *Package Merge* algorithm to reduce the computational complexity of the restricted alphabetic tree problem to $O(NK \log N)$. In [91] and [90] the Package Merge algorithm is introduced for length-limited Huffman (non-alphabetic) binary trees. A radical departure from the dynamic programming methods, the $O(NK)$-time Package Merge algorithm returns to the original greedy approach of

the Huffman algorithm. The alphabetic version of the Package Merge algorithm, an $O(NK \log N)$ procedure that is the algorithm of [92], is quite simple to describe, but appears hard to prove. The algorithm first appeared in [91] (1988) by Larmore and Hirschberg but without proof of correctness.[91] consists of two parts: A full solution for the non-alphabetic version of the problem, including proof of correctness, and a suited solution for the alphabetic problem. For simplicity of description, we don't elaborate on Larmore and Hirschberg's paper, but rather describe the Package Merge algorithm as part of [92].

In [92] the authors describe the Package Merge algorithm for a more general problem, which they call the *weighted binary tree* problem, including proof of correctness.

An instance of the weighted binary tree problem consists of a *weight matrix* $w_{i,l}$, for $i = 1, \ldots, N$ and $l = 0, \ldots, K$ for some given $K \geq \lceil \log N \rceil$, such that

(1) $w_{i,0} = 0$

(2) $w_{i,l+1} \geq w_{i,l}$ (monotonicity)

(3) $2w_{i,l} \leq w_{i,l-1} + w_{i,l+1}$ (concavity)

The *cost* of a binary tree T with respect to a weight matrix $\{w_{i,l}\}$ is defined as

$$|T| = \sum_{i=1}^{N} w_{i,l(i)},$$

where $l(1), \ldots, l(N)$ is the *leaf sequence* of T, i.e., the list of leaf depths. The problem is to find the tree T with minimal cost. For example, the depth-restricted by K alphabetic binary tree problem, discussed in §7 , reduces to the weighted binary tree problem by letting $w_{i,l} = lw_i$ for all $0 \leq l \leq K$.

Define a *tile* as an ordered pair of integers (i, l) such that $i \in [1, N]$ and $l \in [0, K]$. The tile (i, l) is said to have *index* i, *level* l and *width* 2^{-l}. We define the *weight* of (i, l) to be $w_{i,l} - w_{i,l-1}$ for $l > 0$, and zero if $l = 0$. If A is a set of tiles, define its weight as the sum of the weights of its members; The width of A is defined to be the sum of the widths of its members; the index of A is defined as the minimum of the indices of its members. If T is any binary tree, with leaf sequence $l(1), \ldots, l(n)$, we define the *associated set of tiles* to be $skyline(T) = \{(i, l) : l \in [0, l(i)]\}$ whose weight is defined as $cost(T)$. The Package Merge algorithm operates by finding $skyline(T)$, the minimal weight set of tiles which has width $(2n - 1)$, subject to the condition that it is a *geometric tree* (this term is defined in the article, but we don't define it here). The set of tiles generated by the Package Merge algorithm will always satisfy this condition. T can then be recovered from $skyline(T)$.

Recall the Hu-Tucker algorithm for constructing a minimal alphabetic binary tree. Initially, there is a list of N "square nodes" (terminal nodes) each of

which has a weight, and during each step, two square nodes are combined to form a new "round" node, whose weight is equal to the sum of the weights of its children. The Package Merge algorithm has many features in common with the Hu-Tucker algorithm. "Square" nodes of Hu-Tucker correspond to tiles in the Package Merge algorithm, and "round" nodes correspond to packages. A *level-l package*, where l is a nonnegative integer, is a certain kind of tile set that has at least two members, and whose width is 2^{-l}. Packages and tiles are together called items. F^l is the list of all level-l tiles, ordered by index, for $l = 0, \ldots, K$. A pair of items in F^l is *tenatively connected* if there is no tile which is strictly between them in the list. Starting with $l = K - 1$ and ending with $l = 0$, optimal packages of level l are formed by combining "tenatively-connected" items in F^{l+1}. These optimal level-l packages are then merged with the list of level-l tiles to form F^l, a list ordered by index, where a tile is always in front of a package of the same index. The set $skyline(T)$ is simply the union of the items in F^0, from which T can be easily recovered.

The authors suggest using a list of mergeable priority queues to represent F^{l+1}. A pair of items will be tenatively connected if and only if they are both in the same priority queue. Using this data structure, it takes $O(\log N)$ time to select the minimal weight, tentatively connected pair. Since that selection must occur at most $(N - 1)$ times on each of K levels, the time complexity of the Package Merge algorithm is $O(NK \log N)$.

In [94] **(1996) Larmore and Przytycka** develop a $O(K \log N)$-time N-processor parallel algorithm for construction of an optimal alphabetic binary tree with depth restricted to K. Thus the order of the total work of their algorithm is the same as that of the best known sequential time. The algorithm provides a parallel implementation of the Package Merge procedure ([91], [92]). The authors also obtain an $O(l \log^2 N)$-time N processor algorithm that constructs an alphabetic binary tree whose cost differs by at most $\frac{1}{N^l}$ from the cost of an optimal tree. If the input sequence of weights does not contain two consecutive elements of weight less than $\frac{1}{N^l}$, then this approximation algorithm produces an optimal tree. We don't dive into the precise description of the algorithms in this survey.

8. Dichotomous search with unreliable answers

Sometimes while conducting a dichotomous search, erroneous information may be received. **Rivest, Meyer, Kleitman and Spencer** [118] **(1977)** consider such a problem. A search is held for $x \in \{1, 2, \ldots, N\}$ via questions of sort "Is $x \leq a$?". It is assumed that up to k of the answers we receive may be incorrect. The erroneous answers are detected only after the object is found. It is proven that in the worst case $\log(N) + k \log\log(N) + O(k)$ questions are sufficient, and a search strategy is demonstrated to achieve this upper bound. A lower bound of number of questions required for a fixed k is $\log(N) + k \log\log(N) + O(1)$, meaning that the upper bound is within an additive constant of optimal. The lower bound $\log(N) + k \log\log(N) +$

$O(1)$ holds also when more general questions of sort "Is $x \in T$" for $T \subset \{1, 2, \ldots, N\}$ are allowed to be asked.

Rivest, Meyer, Kleitman and Winklmann [119] **(1980)** continue the work of [118]. They show that there is a unique optimal strategy for the continuous problem with only comparison questions allowed: Given $\epsilon > 0$, how many comparison questions about an unknown $x \in (0, 1]$ are necessary in the worst case to determine a subset A of $(0, 1]$ of size less or equal to ϵ which contains x, when up to k answers may be erroneous? Note that A is not required to be an interval. The authors show that the number of questions needed is $q = \min\left\{q' \,\middle|\, \epsilon \geq 2^{-q'} \cdot \sum_{i=0}^{k} \binom{q'}{i}\right\}$. This defines q not only for k being a constant, but also for k being a function of ϵ. The optimal comparison strategy is described.

This comparison strategy for the continuous case does not translate into a strategy for the discrete problem just by setting $\epsilon = \frac{1}{N}$, since the region A which is found to contain x may not be an interval. Yet, it is shown that k extra comparisons are always sufficient to cut A down to an interval. This then yields a comparison strategy for the discrete problem using no more than $\log N + k \log\log N + O(k \log k)$ questions in the worst case, which is optimal even among strategies using arbitrary "Yes-No" questions about x.

Cicalese and Vaccaro [31] **(2003)** discuss a binary search problem where answers to questions are received with a delay of d time units and up to c of the answers may not be received at all. A number is hidden in $\{1, \ldots, N\}$. Questions of the form "Is $x \leq a$?" are sequentially asked (at times $i = 1, 2, 3, \ldots$). The ith question is the question formulated in time i. The answer to the ith question is delivered only with a delay of d, i.e., at time $i + d$. This problem is called "the (N, d, c) problem".

Let $A_d^{(c)}(t)$ be the largest integer N such that the (N, d, c) problem is solved with t questions. The main result of the paper determines recursively the value of $A_d^{(c)}(t)$ for $c = 1$: For any integers $t, d \geq 0$

$$A_d^{(1)}(t) = \begin{cases} \lfloor 0.5t \rfloor + 1 & \text{if } t \leq d + 1 \\ A_d^{(1)}(t-1) + A_d^{(1)}(t-d-1) & \text{if } t \geq d + 2. \end{cases} \tag{8.1}$$

As a corollary, let k_d be the largest positive real root of $x^{d+1} = x^d + 1$. Then $\log_{k_d}(N+1) + O(1)$ questions are necessary and sufficient to solve the $(N, d, 1)$ problem.

For the general case $c \geq 1$ a lower bound on $A_d^{(c)}(t)$ is established. It is shown that $A_d^{(c)}(t) \geq B_d^{(c)}(t)$, where

$$B_d^{(c)}(t) = \begin{cases} 1 & \text{for } t \leq 0 \\ \sum_{i=1}^{t+1} G_i^{(c)}(t, d) & \text{otherwise,} \end{cases} \tag{8.2}$$

where, for all $i = 1, 2, \ldots, t + 1$

$$G_i^{(c)}(t,d) = \min_{0\le j\le \min\{i-1,c\}} \left\{ B_d^{(c-j)}(t-d-i+j) - \sum_{k=1}^{j} G_{i-k}^{(c)}(t,d) \right\}, \tag{8.3}$$

and $G_i^{(c)}(t,d) = 0$ for $i \le 0$. The question of an upper bound remains open in this paper.

Sheu et al. [121] (2003), Wang [126] (2007), Chun [27] (2007) and Tzimerman and Herer [124] (2009) consider the case where unreliable answers might be accepted during the inspection of a production batch, where the location of the first non conforming object has a geometric distribution. That subject is covered in §13.5.

9. Search for multiple objects, by multiple searchers or in high dimensions

In this section we discuss three generalizations of the dichotomous search problem:

(1) Where not only one object is searched for, but several objects;

(2) Where one object is searched for, but several queries may be conducted in an instant;

(3) Where the search is conducted in a multidimensional space.

9.1 Search for several objects

In [58] **Hassin and Megiddo (1985)** formulate the following problem: Let $f : \{0,1,...,N\} \longmapsto \{0,1,\ldots,K\}$ be a monotone nondecreasing step function satisfying $f(0) = 0, f(N) = K$. The objective is to find a policy that locates all the jumps of f using minimal number of f-evaluations in the worst case. Obviously, by performing K binary searches one may recognize $f(i)\,\forall i$, so that $K\lceil \log_2 N\rceil$ f-evaluations should suffice.

When $K \ge N$, $N-1$ f-evaluations are sufficient and may also be necessary in the worst case. For the general case the authors find that the exact upper bound on the number of f-evaluations required for the recognition of all the jumps of f is $K\left\lfloor \log\left(\frac{N}{K}\right)\right\rfloor + \left\lfloor (N-1)\,2^{-\log\left(\frac{N}{K}\right)}\right\rfloor$, where $\log(x) = \max(0, \log_2 x)$.

The authors also state an optimal strategy that reaches the bound: "Place your first search at $i = 2^m$ such that $m = \left\lfloor \log\left(\frac{N}{K}\right)\right\rfloor$. Suppose that $f(i) = K_1$. Proceed recursively with the two resulting problems, namely finding all the K_1 jumps of f over the set $\{0,1,...,i\}$ (if $K_1 > 0$) and $K - K_1$ jumps over $\{i+1,...,N\}$ (if $K_1 < K$)".

In [55] **(1984) Hassin and Henig** explore the problem of finding all the jumps of an integer function in $\{0, 1, \ldots, N\}$ using the minimum expected number of queries (rather than minimizing the worst case scenario as in [58]). Let I be the *information function* defined on subintervals of $[0, N]$. Initially, an information $I(0, N)$ is known and an a priori joint distribution of the jumps is given. At stage $m = 1, 2 \ldots$ a list of points $0 = t_0 < t_1 < \cdots < t_m = N$ is given, the information $I(t_0, \ldots, t_m)$ is obtained and the joint distribution of the jumps is updated. If all the jumps were identified, the search terminates, otherwise we split an interval (t_{i-1}, t_i) for some $i = 1, \ldots, m$ by selecting $t_{i-1} < t^* < t_i$. A *selection strategy* is a set of rules telling us at each stage, based on the given information, which point to select. An interval (t_{i-1}, t_i) is said to be *resolved* at stage m if $I(t_0, \ldots, t_m)$ identifies all the jumps in the interval. A selection strategy is called *optimal* if it minimizes the expected number of stages until $(0, N]$ is resolved.

The main result of the paper suggests a selection strategy which is proven to be optimal under the following conditions: Let $R(t_{i-1}, t_i, I(t_0, \ldots, t_m)) = 1$ if $I(t_0, \ldots, t_m)$ indicates that $(t_{i-1}, t_i]$ is unresolved, and 0 otherwise.

- Condition A: "$R(t_{i-1}, t_i, I(t_0, \ldots, t_m)) = R(t_{i-1}, t_i, I(t_0, \ldots, t^*, \ldots, t_m))$ for every $t_{j-1} < t^* < t_j$, $j \neq i$. This condition means that selection of points outside the interval $(t_{i-1}, t_i]$ does not affect the value of R. This condition implies a common value $R(a, b) := R(a, b, I(t_0, \ldots, t_m))$ for every set of points $t_0 < t_1, \cdots < t_m$ such that $a = t_{i-1}, b = t_i$ for some $1 \leq i \leq m$".

- Condition B: "$\Pr[R(a, a + d, I(t_0, \ldots, t_m))] = 1$ is independent of a and monotone increasing and concave in d for every list $t_0 < t_1, \cdots < t_m$ such that $t_{i-1} < a < t_i - d$ for some $i = 1, \ldots, m$. This condition states that the probability function that an interval $(a, a + d] \subseteq (t_{i-1}, t_i]$ is unresolved given $I(t_0, \ldots, t_m)$ depends only on its length d and is monotone increasing and concave."

If both conditions hold then the following strategy is optimal: In each stage arbitrarily select an unresolved interval $(t_{i-1}, t_i]$ and split it at $t_{i-1} + 2^{t(d)}$ or at $t_i - 2^{t(d)}$ where $d = t_i - t_{i-1}$ and $t(d)$ is the unique integer satisfying $3 \cdot 2^{t(d)-1} \leq d < 3 \cdot 2^{t(d)}$.

For example[14] suppose that several objects are independently uniformly distributed over the interval $(0, N]$. Let $f(t)$ be the number of objects in $[0, t]$. At stage m of the search the information obtained is whether $(t_{i-1}, t_i]$, $i = 1, \ldots, m$, is empty (contains no objects) or not. Thus, $(t_{i-1}, t_i]$ is resolved if it is either empty or $t_i - t_{i-1} = 1$. Condition A is trivially satisfied since splitting at $t^* \notin (t_{i-1}, t_i]$ cannot change the fact that $(t_{i-1}, t_i]$ is empty or nonempty. Let $q(d \,|k)$ be the probability that that the interval $(a, a + d] \subseteq (t_{i-1}, t_i]$, $d > 1$, is empty, given that $(t_{i-1}, t_i]$ contains k empty points. Clearly the empty points are uniformly distributed over $(t_{i-1}, t_i]$ and therefore

[14]The example is quoted from [55].

$$q(d\,|k) = \prod_{j=0}^{d-1} \left[\frac{k-j}{t_i - t_{i-1} - j} \right] \text{ for } d \le k, \text{ and} \tag{9.1}$$

$$q(d\,|k) = 0 \text{ for } d > k$$

It is easy to verify that $q(d\,|k)$ is monotone decreasing and convex in d. Therefore, $\sum_{k=0}^{t_i - t_{i-1}} q(d\,|k) \Pr\{(t_{i-1}, t_i] \text{contains } k \text{ empty points}\}$, which is the probability that $(a, a+d]$ is empty, is monotone decreasing and convex in d, which implies condition B.

In [79], **Karp (1993)** finds more solutions for the problem discussed in [58]. While [58] finds an optimal algorithm for this problem, [79] also states that there is usually more than one solution. In fact each $i \in [0, ..., N]$ such that i or $N - i$ is a multiple of $2^{\lfloor \log(\frac{K}{N}) \rfloor}$ is a first step of an optimal algorithm.

Damaschke [33] **(1997)** considers the following problem: An $n \times n$ matrix is given, whose entries are from the set $\{0, 1\}$ and are subject to the following condition: The right and the lower neighbor of an entry 0 are also 0; the left and the upper neighbor of an entry 1 are also 1. We may query any matrix entry to determine its contents. The aim is to determine the matrix completely by such queries in an efficient way. The problem is equivalent to the search for a discrete monotone nondecreasing function f by threshold queries ($f(i)$ can be thought of as the number of 1's in column i). That means, we have an unknown function f from the set of integers $\{0, 1, \ldots, n\}$ into the same set, with the only condition that $x < z$ always implies $f(x) < f(z)$, and for any pair (x, y) we may ask whether $f(x) \ge y$. Note that this problem is different from that discussed earlier in [58], where a query at argument x returned $f(x)$.

The main result of the paper states that an unknown discrete monotone function with domain and range of size n can be found optimally in $O(\log n)$ time using $O(n)$ independent[15] threshold queries and arithmetic operations on an EREW PRAM.

9.2 Parallel search

In this section we explore a variation of dichotomous search, where more than one query may be made at once.

In [77] **(1976) Itay** analyzes **alphabetic trees of degree** $\sigma > 2$, also called *σ-ary trees*. The objective is to find optimal alphabetic *σ-ary trees*, i.e., *σ-ary trees*, with minimum weighted path length (recall that definition

[15] A set of threshold queries asked during the search process is called *independent* if simultaneous queries (asked in the same unit of time) always call mutually distinct x and mutually distinct y, respectively; or in the matrix formulation - simultaneous queries must involve entries in mutually distinct rows and columns, respectively.

from §2). This model is equivalent to search in which $\sigma - 1$ queries are made simultaneously.

Let an s-forest be a sequence of s trees. The cost of an s-forest is the sum of the costs of its trees. Denote a minimal cost (optimal) alphabetic s-forest on weights $(w_i, ..., w_j)$ by $F_s[i, j]$. The cost of the s-forest, $s > 1$, is

$$WF_s[i,j] = \min_{i \le b < j} \{WF_{s'}[i,b] + WF_{s-s'}[b+1,j]\}, \tag{9.2}$$

for any s' such that $1 \le s' < s$. We search for $WF_1[1, N]$, and the initial conditions are $WF_1[k, k] = w_k \ \forall k \in \{i, \ldots, j\}$. The cost of a σ-ary tree is $f[i, j] = W_{ij} + WF_\sigma[i, j]$, where $W_{ij} = \sum_{r=i}^{j} w_r$. Using these equations the dynamic programming solution can be extended to find optimal trees and forests. When $\delta = j - i \ge \sigma$, there is always an optimal σ-ary tree for $(w_i, \ldots, w_j)$ with exactly σ subtrees. For each $\delta = 1, \ldots, N - 1$, the author finds optimal σ-trees and s-forests for weights $(w_i, \ldots, w_j)$, $j = i + \delta$. By choosing s' in an economic way at each stage (for example $s'_i = 2^{\lfloor \log s'_{i-1} \rfloor} - 1$), the resulting complexity is $O(N^3 \log \sigma)$. The author uses the "Knuth method" (see §2) to cut the running time to $O(N^2 \log \sigma)$, but later, in [51], Gotlieb and Wood explain why in fact the Knuth method is not applicable to alphabetic σ-ary trees.

Gotlieb [50] **(1981)** considers the following problem: We are given keys $K_1 < K_2 < \cdots < K_N$, a page capacity $m \ge 1$, and nonnegative weights $\{p_1, \ldots, p_N, q_0, \ldots, q_N\}$. $W := p_1 + \cdots + p_N + q_0 + \cdots + q_N$. $\frac{p_i}{W}$ is the probability that K_i is the search argument, and $\frac{q_j}{W}$ is the probability of the search argument to be between K_j and K_{j+1}. The problem is to construct an $m+1$-ary search tree which minimizes the cost $\sum_{i=1}^{N} p_i l(p_i) + \sum_{j=0}^{N} q_j (l(q_j) - 1)$ (recall that $l(j)$ is defined as the level of the node j).

The basic dynamic programming solution is presented and it runs in time $O(N^3 m)$. In case that the q's (the missing-key weights) are all zero[16], the author shows an adaptation of the Knuth monotonicity property which allows to cut the running time to $O(N^2 m)$.

The running time in both cases can be further cut to $O(N^3 \log m)$, or $O(N^2 \log m)$ when the q's are all zero using a technique employed in Itay [77].

Gotlieb and Wood [51] **(1981)** discuss the applicability of the monotonicity principle (also called the "Knuth" method) to the m-ary search tree problem. They show, by a counterexample, that the monotonicity principle, when using the dynamic programming construction, cannot be extended to m-ary optimal alphabetic code trees, i.e., trees where the weights appear only at the leaves, contrary to what is claimed by Itay [77]. Yet, the monotonicity principle can be applied to the case where the q's are all zero, and the running time can be reduced to $O(N^2 \log m)$ in that case. In general,

[16]The reader should pay attention that an m-ary ($m > 2$) search tree with the q's all zero is not an alphabetic m-ary tree, as opposed to the case of $m = 2$.

the $O(N^3 \log m)$ time bound is best possible for the dynamic programming method, for both alphabetic and Huffman trees.

Herer and Raz [60] **(2000)** use the information theoretic approach to construct a heuristic for the search problem, where several units are allowed to be inspected simultaneously. The problem is formulated as follows: A batch is produced on a production line, that is subject to failures under a known probability distribution function (not necessarily geometric). Suppose it is known that the last unit of the batch is nonconforming. Also it is assumed that once one unit was produced with a defect, so must be all the next units in the batch. By means of inspection of units the goal is to find the first nonconforming unit (FNU) by minimal expected cost. Two cost parameters are involved: a fixed cost of performing an inspection round (independent of the number of units inspected, can be thought of as the waiting time through the round) and the cost of each inspection. Let M be the number of simultaneous inspections carried out during each inspection round. Let N be the batch size, and let p_i be the probability that i is the FNU. $P := (p_1, \ldots, p_N)$.

Let $K = (k_1, \ldots, k_M)$ be the inspection vector, used to denote all the units inspected during that round (also define $k_0 = 0$, $k_{M+1} = N$, and call unit 0 conforming). Let $a(k, N, P)$ denote the probability of shifting to the abnormal state no later than unit k. $U_0(N, P) = -\sum_{i=1}^{N} p_i \log p_i$ is the uncertainty before the inspections start. When M units in parallel are inspected there are $M + 1$ possible outcomes, and the uncertainty (entropy) regarding the FNU after inspecting according to K is

$$U_k(N, P) = \tag{9.3}$$

$$\left[\sum_{i=1}^{M+1} (a(k_i, N, P) - a(k_{i-1}, N, P)) \cdot U_0\left(k_i - k_{i-1}, \frac{p_{k_{i-1}+1}, \ldots, p_{k_i}}{\sum_{i=k_{i-1}+1}^{k_i} p_i}\right)\right].$$

The main results in [60] state that:

(1) "If M units are to be inspected in parallel, then the amount of remaining uncertainty regarding the location of the FNU is minimized by inspection according to the inspection vector that is closest to dividing the batch into $M + 1$ segments that have equal probability of containing the FNU," i.e., by the choice of an inspection vector $K = \operatorname{argmin}_K \sum_{i=1}^{M+1} \left| a(k_i, N, P) - a(k_{i-1}, N, P) - \frac{1}{M+1} \right|$.

(2) "The maximum reduction in uncertainty achieved by M inspections in parallel is $\log(M + 1)$. Thus, by dividing the initial amount of uncertainty by $\log(M+1)$ a lower bound is obtained on the expected number of inspection rounds required to precisely locate the FNU."

By computational tests it is shown that the proposed heuristic has better performance than the $(M + 1)$-ary search (i.e., dividing the batch to $M + 1$

segments of equal size), the natural extension of the binary search. Also an algorithm is provided for calculating the most cost-effective number of units that should be inspected in parallel at each inspection round.

9.2.1. Unequal inspection costs in parallel search Recall that in the optimal σ-ary alphabetic tree problem the level of a leaf is the number of edges connecting it to the root. Equivalently, suppose that each traversing each edge costs 1. A level of a leaf is the cost of traversing through all the edges from the root to it. An interesting problem arises when we consider structuring an optimal alphabetic tree with unequal costs of the edges. Let the *cost of an edge* (a, b) of a σ-ary tree be c_i, where b is the i-th child of a, $1 \le i \le \sigma$. In general, the σ-ary tree need not be full, i.e., not every node has σ children (for $\sigma = 5$ a node may have a second and forth child while lacking the first, third and fifth children). The *cost of a node* is the sum of the costs of the edges of the unique path from the root to that node. The *cost of a alphabetic tree* is $F[1, N] = \sum_{i=1}^{N} p_i w_i$ where w_i is the weight of leaf i and p_i is the cost of the node associated with it. The objective is to find, for given weights, an alphabetic tree of minimum cost.

Itay [77] **(1976)** formulates the dynamic programing equations for this problem: Let $F_{\alpha,\beta}[i, j]$ be the cost of an optimal tree for weights $(w_i, \ldots, w_j)$ in which the root has no child smaller than α and none greater than β, $1 \le \alpha \le \beta \le \sigma$, $1 \le i \le j \le N$.

$$F_{\alpha,\alpha}[i, i] = c_\alpha w_i \text{ for } 1 \le i \le N, \tag{9.4}$$

$$F_{\alpha,\beta}[i, i] = \min_{\alpha \le \gamma \le \beta} \{F_{\gamma,\gamma}[i, i]\} \text{ for } 1 \le i \le N, \tag{9.5}$$

For $i < j$ and $1 \le \alpha \le \sigma$:

$$F_{\alpha,\alpha}[i, j] = c_\alpha W_{ij} + \min_{\substack{i \le b < j \\ 1 \le \gamma < \sigma}} \{F_{\gamma,\gamma}[i, b] + F_{\gamma+1,\sigma}[b + 1, j]\}, \tag{9.6}$$

and for $i < j$, $\alpha < \beta$:

$$F_{\alpha,\beta}[i, j] = \min \left\{ F_{\alpha+1,\beta}[i, j], \min_{i \le b < j} \{F_{\alpha,\alpha}[i, b] + F_{\alpha+1,\beta}[b + 1, j]\}, F_{\alpha,\alpha}[i, j] \right\}. \tag{9.7}$$

The computation time can be reduced by the Knuth method by a factor N to $O(\sigma^2 N^2)$.

The unequal inspection costs problem in parallel search can be also reformulated in terms of coding theory: recall the formulation of the dichotomous search problem as an alphabetic prefix free coding problem from §2. One well studied generalization of this problem is to let the encoding letters have

different costs, which is equivalent to unequal costs of the edges in σ-ary binary trees. Let Σ be the coding alphabet and let $\sigma_i \in \Sigma$ have associated cost c_i. The cost of codeword $w = \{\sigma_{i_1}, \ldots, \sigma_{i_l}\}$ is $cost(w) = \sum_{k=1}^{l} c_{i_k}$, i.e., the sum of the costs of its letters (rather than the length of the codeword) with the cost of the code still being defined as $Cost(w) = \sum_{i=1}^{n} cost(w_i)p_i$ with this new cost function.

Golin and Lin [49] **(2008)** [17] write the research history on the unequal letter cost prefix-free coding, which we find important to include in this survey. The unequal letter cost coding problem was originally motivated by coding problems in which different characters have different transmission times or storage costs. One example is the telegraph channel in which $\Sigma = \{\cdot, -\}$ and $c_1 = 1$, $c_2 = 2$, i.e., in which dashes are twice as long as dots. Another is the (a, b) *run-length-limited* codes used in magnetic and optical storage, in which the codewords are binary and constrained so that each 1 must be preceded by at least a, and at most b, 0's. This example is the unequal-cost letter problem with an encoding alphabet of $r = b - a + 1$ characters $\{0^k 1 : k = a, a + 1, \ldots, b\}$ with associated costs $\{c_i = a + i - 1\}$.

The alphabetic coding problem has a polynomial $O(tN^3)$ time algorithm [77] that we described earlier in this section.

9.3 Multidimensional search

The multidimensional search problem deals with the complexity of finding a given vector among an ordered (lexicographically or otherwise sorted) list of vectors.

Generalization of the dichotomous search to high dimensions opens a hatch to problems from computational geometry. We do not elaborate on that subject in this survey, but describe only one example by **Dobkin and Lipton** [35] **(1976)**: Given a point (x, y) in $\mathbb{R}^2$ and a collection of m lines $L_1, \ldots, L_m$ we are to find whether the point lies on any of the lines. The main result is that the determination can be done in $O(\log m)$ steps given that the lines have been previously preconditioned (put in a convenient order). The preconditioning condition is a strong request but no stronger than in the ordinary binary search problem, where the items are assumed to be sorted.

The basic algorithm: Let the intersections of the lines be given by the points $z_1, \ldots, z_n$ $n \leq m(m-1)/2$ and let the projections of these points onto the x-axis be given by $p_1, \ldots, p_n$ (the preconditioning of the lines ensures that $p_1, \ldots, p_n$ are ordered). These points define a set of intervals $I_1, \ldots, I_{n+1}$ such that within the slice of the plane defined by each of these intervals no two lines intersect. Define a relation $<_i$ $(1 \leq i \leq n + 1)$: $L_j <_i L_k$ iff $\forall x \in \mathbb{R}$ if $p_i \leq x \leq p_{i+1}$ then $L_j(x) \leq L_k(x)$. Thus we can define for each i $(1 \leq i \leq n + 1)$ a permutation $\pi(i, 1), \ldots, \pi(i, m)$ of $1, 2, \ldots, m$ such that $L_{\pi(i,1)} <_i \cdots <_i L_{\pi(i,m)}$.

[17] The article of Golin and Lin [49] concerns mainly with the non-alphabetic coding problem and not the alphabetic one, thus falls beyond the scope of this survey.

To discover whether (x, y) lies at any of the lines, first find the interval I_i such that $x \in I_i$. This can be done by a dichotomous search with questions of the form "Is $x > z_j$". Once the interval I_i is discovered, conduct a dichotomous search for y by queries of the form "Is $y > L_j(x)$". An algorithm consisting of a dichotomous search into a set of at most $m(m-1)/2+2$ objects (the points p_i) and a dichotomous search into a set of m objects (the lines $L_{\pi(i,1)}, \ldots, L_{\pi(i,m)}$ for the proper choice of i) requires at most $g(m) + g(m(m-1)/2+2)$ steps where $g(m) = \lfloor \log m \rfloor + 1$, and since g is a monotonically increasing function with $g(m^2) \leq 2g(m)$, this quantity is at most $3g(m)$.

This algorithm is further extended to $\mathbb{R}^n$. As a result, for any set of lines $L_1, \ldots, L_m$ in $\mathbb{R}^n$, in can be determined whether a point $x \in \mathbb{R}^n$ is on any of the L_i's in $O((n+1)g(m))$, given that the lines have been preconditioned. Actually, for any set of linear varieties $a_1, \ldots, a_m$ of dimension k in $\mathbb{R}^n$, there is an algorithm that determines whether x is on any of the a_i's in time $O((3 \cdot 2^{k-1} + (n-2))g(m))$ for any $x \in \mathbb{R}^n$, given that the a_i's have been preconditioned.

Now consider another version of multidimensional search- the problem of finding a given vector among a lexicographically ordered list of vectors. Given is a k-vector $x = (a_1, \ldots, a_k)$ and n k-vectors $y_i = (b_{i,1}, \ldots, b_{i,k})$ for $i = 1, ..., n$ such that $y_1 < y_2 < \cdots < y_n$ by lexicographic order. Comparison of an a_i with a $b_{j,k}$ returns $a_i \geq b_{j,k}$ or $a_i < b_{j,k}$ and is counted as a step. The goal is to find the worst case time complexity for the search problem. **Hirschberg** [63] **(1980)** showed that $\lfloor \log(n) \rfloor + k$ comparisons are necessary and, for $n \geq 4k$, $k \left\lceil \log(\frac{n}{k}) \right\rceil + 2k - 1$ comparisons are sufficient.

In **1980 Rao Kosaraju** [109] improved both upper and lower bounds for this problem known at that time: The upper bound was improved in [109] to $\log(n) + \sqrt{2}(k-1)\sqrt{\log(n)} + 2k + O(1)$ and the lower bound of [63] was improved to $\log(n) + \frac{1}{2}\sqrt{k \log(n)} - h(k)$ where h is a function of k only.

Belal, Ahmed and Arafat [19] **(1998)** explore the problem of obtaining an optimal alphabetic tree for the $M \cdot N$ two-dimensional array of weights. This problem is in fact equivalent to the problem of minimizing the cost of sequentially cutting the array into individual cells using vertical and horizontal cuts. In terms of dichotomous search, if we search for an object (x, y) in the two dimensional array, then each horizontal cut is equivalent to a query of the form "Is $y > j$", while each vertical cut means "Is $x > i$". The root of a tree corresponds to a cut. This two dimensional problem can be solved in polynomial time $O(N^4)$ when $M = O(N)$ using dynamic programming. The authors use a measure of goodness for each cut to limit the number of feasible solutions to the problem and thus reduce the number of possible choices for the root of the optimal tree and improve the computational complexity. The complexity of the new method is not computed in the article, but experimental results show improvement in the run time relative to the dynamic programming algorithm.

First, the authors define a lower bound on the cost of the optimal alphabetic tree: If $T(R_i)$ and $T(C_j)$ denote the cost of optimal trees for row i and

column j respectively, the cost of the optimal tree for the two dimensional array is bounded by

$$T_{optimal} \geq \text{Bound} = \sum_{i=1}^{M} T(R_i) + \sum_{j=1}^{N} T(C_j). \tag{9.8}$$

This bound is achieved when all rows, or all the columns, have the same optimal tree. Define limit L as $L = T_{approx} - \text{Bound}$, where T_{approx} is obtained using a fast greedy algorithm for finding an approximate optimal alphabetic tree, given in [4]. Suppose that a vertical cut divides an $M \cdot N$ array into a left array and a right array. If each of the resulting arrays can be solved such that their respective lower bounds are achieved, then the cost of the tree of left part equals

$$c(left) = \sum_i T(R'_i) + \sum_j T(R'_j),$$

and the cost of the tree of right part equals

$$c(right) = \sum_i T(R''_i) + \sum_j T(R''_j),$$

where R', C' are the rows and columns of the left part, and R'', C'' are the respective ones of the right part. The resulting tree obtained by making this vertical cut will have the cost $C(left) + C(right) + W$, where W is the sum of the weights of the original array.

The *expense of a cut* (Ex) is defined as the deviation of its cost from the bound, and is computed as follows:

$$Ex = \sum_i T(R'_i) + \sum_j T(C'_j) + \sum_i T(R''_i) + \sum_j T(C''_j) + W - \sum_i T(R_i) - \sum_j T(C_j) \tag{9.9}$$

giving

$$Ex(vertical\ cut) = \sum_i T(R'_i) + \sum_i T(R''_i) + W - \sum_i T(R_i) \tag{9.10}$$

and

$$Ex(horizontal\ cut) = \sum_j T(C'_j) + \sum_j T(C''_j) + W - \sum_j T(C_j) \tag{9.11}$$

As an example, also given in the article, for the array with two rows R_1, R_2 and N columns the horizontal cut has zero expense, using equation (9.11) and noting that in this case $T(C'_j) = T(C''_j) = 0$ for $j = 1, 2$ and $\sum T(C_j) = W$. The optimal tree in this case has the lower bound of equation (9.8) and

its cost. This is expected since in the $2 \cdot N$ arrays, all columns have the same optimal tree.

The algorithm is as follows: "given an $M \cdot N$ array of weights, the limit L is computed, and then the expenses of all possible $(M-1)+(N-1)$ vertical and horizontal cuts are computed. Only the cuts with expense less than L are retained as candidates to an optimal solution. Then for a given candidate with expense $Ex < L$, the two parts generated by the cut are explored for candidate cuts by looking for a pair of cuts, one for each part, whose sum of expenses is less than the new limit $(L-Ex)$. Finally, when the size along one of the dimensions reaches the value 4, no further exploration is needed and dynamic programming can be used to obtain the optimal tree for this array."

10. Generalizations of both the alphabetic tree problem and the Huffman tree problem

Some literature exists on problems that generalize the alphabetic tree problem. The problems we describe here are general enough to have the Huffman tree problem as a special case also.

Rao Kosaraju, Przytycka and Borgstrom [110] **(1999)** introduce the optimal split tree problem, that generalizes both the Huffman tree problem and the alphabetic tree problem. Consider a set $A = \{a_1, \ldots, a_N\}$, with each element a_i having an associated weight $w(a_i) > 0$. A partition of A into two sets $B, A - B$ is called a *split*. A set S of splits such that for any pair $a, b \in A$ there exists a split $\{B, A - B\}$ in S such that $a \in A$ and $b \in B$ is called a *complete set of splits*. A *split tree* for a set A with a complete set of splits S is a rooted tree T in which the leaves are labeled with elements of A and internal nodes correspond to splits in S. More formally, for any node v of a leaf labeled tree T, let $L(v)$ be the set of labels of the leaves of the subtree rooted at v. Then a split tree is a full binary tree such that for any internal node v with children v_1, v_2 there exists a split $\{B, B'\} \in S$ such that $B \cap L(v) = L(v_1)$ and $B' \cap L(v) = L(v_2)$. Note that such a split tree is guaranteed to exist when the set of splits is complete. Then the cost $c(T)$ of a split tree T is defined in the standard way: $c(T) = \sum_{i=1}^{N} l(i)w(a_i)$ where $l(i)$ is the length of the path from the root to leaf a_i. The optimal split tree problem is to compute for a given (A, S) a minimum cost split tree. The problem is a generalization of the classic Huffman coding problem, in which the set of splits S contains all possible splits of A.

A split tree problem can be viewed as a problem of constructing a search tree, where the elements we search for are located in the leaves. Each split corresponds to a property that partitions the input set into two subsets. If the weight corresponds to the probability of accessing a given element, then an optimal split tree optimizes expected length of a search path. In this context, the split tree problem generalizes the alphabetic tree problem: That is, if we assume that the input set $A = \{a_1, \ldots, a_N\}$, is linearly ordered and define the set of splits to be $N - 1$ splits $S_1, \ldots, S_{N-1}$ where $S_i =$

$(LE_i, A - LE_i)$ and $LE_i = \{a \in A \,|a \leq a_i\}$, the optimal split tree problem reduces to the classic optimal alphabetic tree problem.

The authors show that the optimal split tree problem is NP-complete. They demonstrate that a modification of the greedy algorithm which always chooses a best balanced split guarantees $O(\log N)$ approximation ratio.

Barkan and Kaplan [16] **(2006)** introduce a different generalization - the *partial alphabetic tree problem* (PAT). In the partial alphabetic tree problem a set of non-negative weights $W = \{w_1, \ldots, w_N\}$ is given, partitioned into $m \leq N$ blocks $B_1, \ldots, B_m$. The objective is to find a binary tree T where the elements of W reside in its leaves such that if we traverse the leaves from left to right then all leaves of B_i precede all leaves of B_j for every $i < j$. The order of the items within each block is arbitrary. Furthermore, among all such trees, T has to minimize $\sum_{i=1}^{N} w_i l(i)$, where $l(i)$ is the depth of w_i in T.

The Huffman problem is a special case of the PAT problem where there is only one block. The minimal cost alphabetic tree problem is also a special case of the PAT problem when there are $m = N$ blocks, and block i contains the single element w_i. Partial alphabetic trees are useful when we want to code a set of items with known frequencies subject to an alphabetic restriction on some of the codewords.

The main result is an algorithm for the partial alphabetic tree problem which runs in time $O(\left[\frac{W_{sum}}{W_{min}}\right]^{2\alpha} \log \frac{W_{sum}N^2}{W_{min}})$ where $W_{sum} = \sum_{i=1}^{N} w_i$, $W_{min} = \min_{1 \leq i \leq N} \{w_i\}$ and $\alpha = \frac{1}{\log \varphi} \approx 1.44$ (φ is the golden ratio). In particular the running time is polynomial in case the weights are bounded by a polynomial function of N.

The proposed algorithm for the PAT problem relies on a solution to what is called *the layered Huffman forest problem* which is of independent interest. In the layered Huffman forest problem we are given a sequence of weights $w_1, \ldots, w_N$ and a sequence of depths $d_1, \ldots, d_k$. The goal is to find a forest F of k binary trees that minimizes $\sum_{i=1}^{N} w_i d_F(w_i)$ where $d_F(w_i) = d_{T_j}(w_i) + d_j$ if $w_i \in T_j$. The authors build an algorithm for this problem that runs in $O(kN^2)$ time.

Barkan and Kaplan also develop an algorithm for another problem, that is also a generalization of the alphabetic tree and the Huffman tree problem. That problem, which the authors call the *parallel alphabetic tree* problem was first introduced by Abrahams (1997) in [2]. In the parallel alphabetic tree problem the weights are also partitioned into disjoint sets, but this time in the resulting tree the weights in the same set should be ordered while there is no restriction on the order of weights from different sets. This model is useful for encoding an alphabet that has several groups of characters such as numbers, letters, and punctuation marks, and the alphabetic order of characters in each group should be maintained. The authors [16] show that the parallel alphabetic tree problem can be solved in polynomial time when the number of sets is constant.

11. Search for rationals

How many queries of the form "is $x \leq \frac{p}{q}$" (for $p, q \in \mathbb{N}$) are needed to determine a positive rational number x where the denominator and numerator are integers bounded by an integer $M > 0$? An immediate solution to this problem is to list all $\Theta(M^2)$ possible rational numbers in an array, sort them and perform a binary search on the sorted array. The maximum number of queries needed in this solution matches the lower bound of $2 \log_2 M$, but a preprocessing phase is needed that requires $\Theta(M^2)$ time and space.

Efficient $\Theta(\log M)$ - time algorithms that do not require a "preprocessing phase" have been proposed by Papadimitriou [106] and Reiss [112].

Papadimitiou [106] **(1979)** shows that x can be determined exactly by $O(\log M)$ queries of the form "is $x \leq \frac{p}{q}$", where $p, q \leq M$ and only $O(\log M)$ other operations and comparisons of integers of size up to $2M$. The proof is based on Farey series, and we don't elaborate on it here. Naturally, the achieved bound is asymptotically optimal, since it takes $O(\log M)$ queries just to distinguish among $\frac{0}{M}, \frac{1}{M}, \frac{2}{M}, \ldots, \frac{M}{M}$.

Reiss [112] **(1979)** uses continued fractions to solve the problem. The author shows that it is sufficient to use binary search, or any other standard technique, to find an approximate solution, $\frac{x}{y}$, such that $\left|\frac{p}{q} - \frac{x}{y}\right| < \epsilon$ where $\epsilon < \frac{1}{2M^2}$. This is true because the best continued fraction approximation to $\frac{x}{y}$ with denominator bounded by M must be $\frac{p}{q}$. Since this approximation can be found in $O(\log M)$ arithmetic operations and since we can find an appropriate $\frac{x}{y}$ using $\lceil \log(2M^3) \rceil = O(\log M)$ queries, we can determine $\frac{p}{q}$ in $O(\log M)$ steps (without the wasteful preprocessing phase of sorting the array).

Zemel [133] **(1981)** notices that both methods, that of Papadimitrio and that of Reiss, lend themselves easily to practical implementations. The author discusses two problems in which finding a rational number in a bounded set of rationals is useful: minimization of ratio functions and the weighted p-center problem on a tree. Here we elaborate on the first problem only.

Consider the problem (PR):

$$s^* = \min \frac{c_0 + cx}{d_0 + dx} \quad \text{subject to } x \in F,$$

where $c_0, d_0 \in \mathbb{Z}$, $c, d \in \mathbb{Z}^n$ and F is a set of $0-1$ vectors in $\mathbb{R}^n$. Consider the linear version (PL):

$$s^* = \max cx \quad \text{subject to } x \in F.$$

Obviously, any algorithm for (PR) can solve (PL) as well. The author recalls previous results of Megiddo: Megiddo [98] proved that if problem (PL) can be solved within $O(p(n))$ comparisons and $O(q(n))$ additions, then problem

(PR) can be solved in time $O(p(n)[q(n)+p(n)])$. However, Megiddo's result falls short of asserting that (PR) is solved in polynomial time if (PL) is. This is due to the limitation on the type of operations allowed by the algorithm for (PL), namely additions and comparisons only. Zemel, using the result on search for rationals, proves that (PR) is indeed solvable in polynomial time iff (PL) is.

The author shows several examples for the use of the result on minimization of ratio functions. For example, let G be a perfect graph. It is shown in [52] how to find, in polynomial time, a subset of vertices of G which is independent (i.e., no two vertices in the subset are connected with an edge) and which maximizes a linear objective function. Thus, we can solve in polynomial time the problem $s^* = \max \frac{c_0+cx}{d_0+dx}$, where x is the incidence vector of an independent set G, and where c, d are vectors of integers with $d_0 + dx > 0$ for every feasible x.

Kwek and Melhorn [88] **(2003)** present an algorithm that requires only $2\log M + O(1)$ queries, which matches the lower bound for this problem. First express x as $\lfloor x \rfloor + \frac{a}{b}$ where a and b are relatively prime and $a < b$. Searching for the integer part combines exponential search with binary search: first compare x with 2^k for $k = 0, 1, 2 \ldots$ until $x \leq 2^k$ and then use binary search to locate x in the interval $[2^{k-1}, 2^k]$. The number of comparisons required so far is $2\log \lfloor x \rfloor + O(1)$. To determine $\frac{a}{b}$ efficiently, the authors first determine the fraction in some interval of form $\left[\frac{\mu}{2T^2}, \frac{\mu+1}{2T^2}\right]$ for $T := \left\lfloor \frac{M}{\lfloor x \rfloor} \right\rfloor$ and some μ in $2\log M - 2\log \lfloor x \rfloor + O(1)$ queries, and then prove that this fraction is unique in that interval.

12. Dichotomous search games

The dichotomous search problem appears also in game theory, considered as a game between a hider and a searcher. In general, the hider makes the first move, in which he hides the object in the "search space", and the searcher must make an effort to find its location. The search proceeds in discrete steps, and after each step a feedback is provided.

To fit the focus of this survey, we only consider the search games in which:

(1) The hider H chooses the initial location of the object y and never changes it during the entire game.

(2) The searcher S looks in the set of integers $1, 2, ..., N$ ("the search space"). The game will be referred to as G_N.

(3) No restriction on the order of the searcher's queries are made.

(4) After each query x_i the searcher receives a feedback whether $y \leq x_i$ or $y > x_i$. The game proceeds until the searcher locates y.

(5) The payoff to the hider is the expected number of queries required to locate y. The *value of the game* G_N is denoted by $v(N)$ and defined

as the maximal payoff that can be assured by the hider (i.e., the minimal price the searcher must pay).

Gilbert [47] **(1962)** explores the game G_N via two other games: G'_N and G''_N. In G'_N, H may change y before each guess of S (as opposed to game G_N, in which changes at the object's location can never be made); in G''_N, H may change y after each guess of S, before providing an answer, The changes in G'_N and G''_N need to be in line with previous answers. The author uses the two games G'_N and G''_N to provide bounds on G_N.

The game G''_N is easy to explore: in G''_N, H can pick a y at each step, which gives S bounds as far apart as possible. S should pick each x_i to bisect, as nearly as possible, the range in which y is known to lie. The value of the game is $v''(N) = 1 + \log N$. In the game G'_N, whose value is $v'(N)$, optimal strategies for S and H must pick $1 \le x_1 \le N$ and $1 \le y \le N$ so as to solve the game with $N \times N$ payoff matrix:

$$M_N(y, x_1) = \begin{cases} 1 + v'(x_1 - 1) & x_1 < y \\ 1 & x_1 = y \\ 1 + v'(N - x_1) & y < x_1 \end{cases} \tag{12.1}$$

The symmetry $M_N(N+1-y, N+1-x) = M_N(y, x)$ simplifies solving this game. After the first guess of S in G'_N, H can change y within the diminished interval obtained by the first query, thus by changing it (or leaving it in place) there begins a new game G_{x_1-1} or G_{N-x_1}. The argument which treats the first game of G'_N as a separate game does not apply to G_N.

Values $v'(N)$ for $N \le 6$ are computed, and also the according optimal S strategies are stated. No values for $N \ge 7$ are presented since they require fast computation, which was not available at that time. The values $v'(A)$ for small A are found useful for bounding $v'(N)$ for large N: write $N = 2^n A + x$ such that $A = 3, \ldots, 6$ and $x = 0, \ldots, 2^n - 1$ to get the bound $v'(N) \le v'(A) + n$.

Several upper bounds for the game G_N are derived. A simple upper bound is:

$$v(N) \le v'(N) \le v''(N) = 1 + \log N. \tag{12.2}$$

A more accurate upper bound is $v(N) \le v'(N) \le v'(A) + n$ as shown above. A lower bound for $v(N)$ is

$$v(N) \ge a - 1 + \frac{2w + a}{N}, \tag{12.3}$$

where $a = \log(N+1)$ and $w = N + 1 - 2^a$. This bound is obtained as the smallest expected payoff S can give to H in G_N when H picks $y = 1, 2, \ldots, N$ with probabilities $1/N$ each.

Johnson [78] **(1964)** proves necessary conditions for optimality of a strategy for G_N, which greatly reduce the computational complexity of computing the value. The hider plays $\{p_k\}$ for $k = 1, \ldots, N$ and the searcher plays a strategy $S_i = \{S_{ij}\}$ where S_{ij} is the number of the guess when j is tried. S_i is chosen by the searcher with probability t_i. The value of the game is:

$$v(N) = \min_{t_i} \max_{p_j} \sum_i t_i \sum_j S_{ij} p_j. \tag{12.4}$$

The author shows that in every optimal hider strategy, $p_j = p_{N-j}$ and $p_1 \geq p_2$ hold. Moreover for $N \geq 5$ $p_1 > p_2$. As for the searcher - suppose that at a given stage the searcher, playing S_i, has located his guess on the interval $k \leq j \leq m$, and that S_i calls for next playing at a, left of the median of the hider's frequent distribution on this interval, and if a is too small, next playing at b to the right of a. Then a necessary condition for optimality of S_i against p_j is that

$$\sum_{k \leq j \leq a} p_j \geq \sum_{b \leq j \leq m} p_j. \tag{12.5}$$

For example, if the searcher's first try at 3 is too small, his second try must be $\geq N - 2$. Moreover, at each stage the searcher should make his guess inside the middle third of the hider's probability distribution on the current interval of uncertainty.

Gal [42] **(1974)** considers G_N and presents the optimal strategies for both players. Let Q be a mixed strategy of S, and let q be a mixed strategy of H. $V_N(q, Q)$ is defined as the expected number of queries used by the searcher to locate y. Further definitions:

$$I = \lceil \log_2 N \rceil, \tag{12.6}$$

$$J = \left\lceil \frac{N}{2} \right\rceil, \tag{12.7}$$

$$t_N = I + \frac{2N - 2^{I+1}}{N}, \tag{12.8}$$

$$v(N) = \sup_q \inf_Q V_N(q, Q) = \inf_Q \sup_q V_N(q, Q). \tag{12.9}$$

The main result of this paper is a formulation of optimal strategies of the hider. It is shown that for $N = 2J$, the value $v(N)$ of G_N is equal to t_N. An optimal strategy of the hider must be one of the following:

- "z- the mixed strategy of H, according to which he chooses each integer $1 \leq y \leq N$ with probability $1/N$.
- z_1- the mixed strategy of H defined only for even N, $N = 2J$ according to which he chooses each odd integer with probability $1/J$.
- z_2- the mixed strategy of H defined only for even N, $N = 2J$ according to which he chooses each even integer with probability $1/J$.

If $N = 2J + 1$, then $V_N = I + \frac{2N-2^{I+1}}{N-1}$ and the following strategy is optimal:

- z_3- the mixed strategy of H defined for $N = 2J + 1$ according to which he chooses each even integer with probability $1/J$."

An optimal strategy Q_N of S is also constructed. If $N = 2J$ Q_N is defined as a mixture of J pure strategies, that are constructed in the paper, each of them chosen with probability $1/J$. For odd N an optimal strategy is also presented. The value of the game using Q_N is of course t_N. An interesting remark is that the "natural" bisection strategy is not optimal for S. For example, if $N = 6$, then the value of the game is proven to be $V(6) = t_6 = 2\frac{2}{3}$. On the other hand, if S always chooses 3 first, then H can choose y to be 2 and the payoff would be 3.

Gal [43] **(1978)** studies the game of locating the searched object in an interval as small as possible in spite that the information that the searcher obtains using dichotomous queries may be wrong. The hider H chooses the point $e \in [a, b\ )$ and the searcher S tries to locate it. In order to do so he can obtain information by making a fixed number n of sequential queries of the form "Is x greater than e?. For $a \leq t < b$ let the independent random variables Y_t have the distribution given by:

$$\begin{array}{llll} \Pr(Y_t = 1) = 1 - \alpha, & \Pr(Y_t = 0) = \alpha, & \text{for } t \leq e \\ \Pr(Y_t = 1) = \beta, & \Pr(Y_t = 0) = 1 - \beta, & \text{for } t > e \end{array} \tag{12.10}$$

where $\beta > 1 - \alpha$. $\Pr(Y_t = 1)$ is the probability of an affirmative answer to the query "Is t greater than e"? The searcher looks at location t: If $t \leq e$, i.e., the searcher's guess is too low, the probability of obtaining a wrong indication of the location of e is equal to $1 - \alpha$, while if $t > e$, i.e., the searcher's guess is too high, the probability of a false indication is equal to $1 - \beta$.

A pure strategy of S consists of choosing the first point x_1, and observing the value Y_{x_1}, then choosing x_2 and observing Y_{x_2} etc., where $x_i = g(x_1, Y_{x_1}, x_2, Y_{x_2}, \ldots x_{i-1}, Y_{x_{i-1}})$ are predetermined functions (that form the search policy). After making these n observations, S chooses a set $E = E(x_1, Y_{x_1}, \ldots x_n, Y_{x_n})$ and receives $\frac{1}{\mu(E)}$, where $\mu(E)$ is the Lebesque measure of E, if $e \in E$ and 0 if $e \notin E$. Thus S wishes to find a set which is small and contains e with high probability. H, on the other hand, wishes to a distribution function of the location of e in a way that minimizes $\frac{1}{\mu(E)}$.

Define recursively the pure strategy $D_n(a', b')$ of S as follows: $D_0(a', b')$ consists of choosing the segment $[a', b')$ as the set E. For $n \geq 1$ the strategy $D_n(a', b')$ consists of choosing the first point of observation x_1 according to

$$x_1 = a' + \frac{\beta}{\alpha + \beta}(b' - a'). \tag{12.11}$$

If $Y_{x_1} = 0$ proceed using $D_{n-1}(x_1, b')$ while if $Y_{x_1} = 1$ use $D_{n-1}(a', x_1)$.

The main result is the following:

(1) "The uniform strategy $u(a, b)$ (i.e., choosing the point e according to the uniform distribution on $[a, b)$ is the unique optimal strategy of H).

(2) $D_n(a, b)$ is an optimal strategy of S.

(3) The value of the search game is equal to $\frac{(\alpha+\beta)^n}{b-a}$."

This result implies that by using the strategy $D_n(a, b)$, S can ensure a pay-off of at least $\frac{(\alpha+\beta)^n}{b-a}$; while on the other hand H can ensure that the expected pay-off to S will not exceed this sum, by using strategy $u(a, b)$. That means that this is a zero-sum game. Note that both optimal strategies are independent of n. $D_n(a, b)$ guarantees a payoff that increases exponentially with n at a rate of $\alpha + \beta$ (which is greater that 1). This is achieved by two contradicting effects: For any $e \in [a, b)$ the probability that the final interval E actually covers the point e lies between $(\min(\alpha, \beta))^n$ and $(\max(\alpha, \beta))^n$ so that it decreases exponentially. However, the length of E lies between $\left\{\frac{\min(\alpha,\beta)}{\alpha+\beta}\right\}^n (b-a)$ and $\left\{\frac{\max(\alpha,\beta)}{\alpha+\beta}\right\}^n (b-a)$ so that it decreases exponentially at an even faster rate.

Ferguson [38] **(1996)** considers the following simple game: The hider chooses a number y in the interval $[-1, 1]$. The searcher chooses a number x in that interval and is informed whether $x < y$, $x = y$ or $x > y$. The searcher can choose a number x only once, after which he estimates the value of y by z. The payoff given by the searcher to the hider is $(y - z)^2$. The objective of the hider is to maximize the minimum possible payoff and the objective of the searcher is to minimize it.

The author proves that this game has a minimax value $v = \frac{1}{2e}$. The unique optimal strategy for the hider is to choose $y \in [-1, 1]$ according to the distribution $F(y)$ that has a mass $\frac{1}{e+1}$ at $y = -1$, a mass $\frac{1}{e+1}$ at $y = 1$ and density (12.12):

$$f(y) = \frac{1}{2e(e^{-1} + y^2)^{1.5}} \quad \text{for } |a| < 0.5(1 - e^{-1}). \tag{12.12}$$

The unique optimal strategy for the searcher is to choose $x \in [-1, 1]$ according to the distribution $G(x)$ that has a mass 0.25 at $x = 0.5(1 - e^{-1})$, a mass 0.25 at $x = -0.5(1 - e^{-1})$ and density (12.13):

$$g(x) = \frac{1}{2(e^{-1} + x^2)^{0.5}} \quad \text{for } |x| < 0.5(1 - e^{-1}). \tag{12.13}$$

If $y < x$, choose $z = x - \sqrt{e^{-1} + x^2}$; if $y > x$, choose $z = x + \sqrt{e^{-1} + x^2}$; if $y = x$ choose $z = x$.

Additional literature on search games is covered in this survey: see [17, 5, 43]. Applications of search games to economics are described in §14.

13. Search for a state transition point in production processes with geometric or arbitrary failure rate

Suppose that an item produced by a certain machine is found to be defective. It is known to be the Nth item produced since the machine was last inspected and found to be operating properly. The location of the first defective item is known to have a certain distribution. All items produced after the first defective item are also flawed. The producer's goal is to minimize the expected number of inspections required to locate the first defective item. That is an instance of a dichotomous search problem, where a query at an item reveals whether it is defective. If it is not defective, then the first defective item is produced later. If it is flawed, then the first defective item has been already produced.

In many applications the failure rate is assumed to be constant, i.e., the location of the first non-conforming object is assumed to have a geometric distribution. Yet, some results hold not only for the geometric distribution, but for any a priori distribution. Throughout this section we note the assumptions at the basis of each article.

13.1 Algorithms and heuristics for the basic problem

Consider another example (introduced by Hassin [54]): A communication system consists of $N - 1$ transmitting stations. A message is sent from the source to the first station, then to the second and so forth, until it is sent from the $N - 1$-th station to the final destination. The number of messages a station transmits until it fails is geometrically distributed. Given that a message has been sent from the source and has not arrived at the destination, the goal is to locate the defective transmitter using minimal expected number of queries. A query at a transmitting station reveals whether the message has arrived to it. If it has, then the defective transmitter is on the way from the checked transmitter to the destination. If it hasn't, then the defective transmitter lies on the way from the source to the checked transmitter.

Hassin [54] **(1984)** shows that the $O(N \log N)$ complexity of search derived at [71] can be reduced to just $O(N)$.

The problem is formulated as follows:

Given is a 2-state binary stochastic process $\{I_j\}, j = 0, 1, ...,$ with initial state $I_0 = 0$. Once in state 1 the system remains there. If the system is in state 0 at time t, it stays there with probability $p < 1$. State 0 is also called the *normal state*, and state 1 the *abnormal state* Suppose that it is given that $I_N = 1$. The cost of each query is 1 and it reveals whether the first defective item lies before or after the searched location. The objective is to find the unique time period $t \in \{1, ...N\}$ such that $I_0 = ... = I_{t-1} = 0$ and $I_t = I_{t+1} = ... = I_N = 1$, with minimum expected cost. That unique time period will also be referred to as FNU (first nonconforming unit).

The problem can be solved via dynamic programming: The probability that we observe state 0 after j transitions is $\frac{p^j(1-p^{N-j})}{1-p^N}$. Let $f(n)$ denote the expected cost of search under the optimal strategy in a problem of length n. Then:

$$f(1) = 0, \tag{13.1}$$

$$f(n) = 1 + \min_{x=1,...,n-1} \left\{ \frac{p^x(1-p^{n-x})}{1-p^n} f(n-x) + \frac{1-p^x}{1-p^n} f(x) \right\}. \tag{13.2}$$

Let x_n^* denote the argument that minimizes the right-hand side of (13.2).

The problem can be reformulated via the 2-tree formulation in the following way: Find a 2-tree that minimizes $\sum_{k=1}^{N} l(k)p^{k-1}$ (the weighted path length), where $l(t)$, the level of node t, is the cost of reaching the tth node when the search starts from the root. Such a tree is called optimal. §2 elaborates on the terms discussed here.

A tree satisfying $l(1) \leq l(2) \leq \cdots \leq l(N)$ is said to be *nondecreasing*. It is easy to see that there is a one-to-one correspondence between nondecreasing sequences of integers satisfying (2.1) and nondecreasing alphabetic 2-trees. It is shown that the optimal tree is nondecreasing and the levels $l(j)$ of its terminal nodes solve the following problem:

$$\begin{array}{ll} \text{Minimize} & \sum_{k=1}^{N} l(k)p^{k-1} \\ \text{subject to:} & \\ & \sum_{k=1}^{N} \left(\frac{1}{2}\right)^{l(k)} = 1 \\ & l(1), ..., l(N) \text{ are integers.} \end{array} \tag{13.3}$$

An optimal strategy is constructed in linear time complexity $O(N)$. The efficient computation is enabled due to the major result:

$$x^*_{n+1} \in \{x^*_n, x^*_n + 1\} \quad \text{for } n \in \{1, \ldots, N\}\,. \tag{13.4}$$

As a corollary, (13.2) can be modified as follows: Let $F(n) = (1 - p^n)f(n)$. Then

$$F(N) = (1 - p^N) + \min_{x = x^*_{N-1}, x^*_{N-1}+1} \{p^x F(N - x) + F(x)\}\,. \tag{13.5}$$

Only x^*_{N-1} and $x^*_{N-1} + 1$ need to be considered as candidates to be optimal, and thus only $O(N)$ operations are necessary to construct the optimal strategy.

Apart from the optimal strategy, Hassin also suggests an approximate solution to the posed problem applying the Lagrange approximation technique based on relaxation of (13.3), thus treating $l(k)$ as a continuous variable. The approximate value $\widehat{F}(N)$ of $F(N)$ is given as a closed formula (13.6). If $x^*_N = \log_p\left(\frac{1+p^N}{2}\right)$ is between $k - \frac{1}{2}$ and $k + \frac{1}{2}$ for an integer k, the approximate strategy is k.

$$\widehat{F}(N) = (1-p^N)\log_2\left(\frac{p - p^{N+1}}{1-p}\right) - \left(\frac{1 - p^{N+1}}{1-p} - (N+1)p^N\right)\log_2 p. \tag{13.6}$$

Straightforward and approximate results for the optimal values are compared in [54], and the difference is found not to exceed 0.5%.

A subsequent paper [59] of **He, Gerchak and Grosfeld-Nir (1996)** deals with a version of the problem in [54], different only by the fact that at the last time period $t = N$ the state of the stochastic process is not a priori assumed to be 1.

A simple modification to Hassin's algorithm solves this problem: Check the last state - if it is 0, then $I_j = 0$ for all $j = 0, ..., N$. Otherwise the problem is reduced to Hassin's version. The complexity of the search is $O(N)$, and the expected number of queries differs from the optimal solution by at most one query.

The authors of [59] use dynamic programming to solve the case where I_N is not necessarily 1. It is given that $I_0 = 0$. Let $g(n)$ be the expected number of inspections needed to find the state at which the switch is performed among n numbers if the first number inspected is optimal. $f(n)$ is the expected number of inspections when I_n is known to be 1 and the first place inspected is optimal. Note that $g(1) = 1$ and $f(1) = 0$.

Then we have the following two recursive equations:

$$g(n) = \min_{1 \le k \le n} \left\{1 + p^k g(n-k) + (1 - p^k)f(k)\right\}, \tag{13.7}$$

$$f(n) = \min_{1 \le k < n} \left\{ 1 + \frac{p^k(1-p^{n-k})}{1-p^n} f(n-k) + \frac{1-p^k}{1-p^n} f(k) \right\}. \qquad (13.8)$$

Note that in [54] it was assumed that $I_N = 1$, so only recursion (13.8), which is equivalent to equation (13.2) was needed.

The optimal location to inspect first is proved to have a limiting value of $\log_p(0.5)$ when $N \to \infty$. Moreover, the optimal location to inspect first converges to the same limiting value if it is known that $I_N = 1$. This result suggests the following search heuristic:

(1) "If $N \ge \log_p(0.5)$, inspect unit $\lfloor \log_p(0.5) \rfloor$

(2) If $N < \log_p(0.5)$, and quality of unit N is unknown, inspect unit N.

(3) If the last unit is defective, inspect unit $N/2$."

Empirical comparison of the heuristic to the optimal solution over various values of N and of p shows that the heuristic achieves expected number of inspections very close to optimal.

Herer and Raz [60] **(2000)** apply Shannon's information theoretic approach to find the FNU in a batch. Their work actually explores a wide variety of search techniques, including both serial and parallel inspection. In this paragraph we state the main results for serial inspection procedure (meaning that the queries are made one after another, rather than in parallel), while in §9.2 the results of parallel inspection will be introduced. The following results can be applied to any probability distribution of the FNU's location, and the geometric distribution is presented as an example.

Let N be the batch size, and let p_i be the probability that i is the FNU. $P := (p_1, \ldots, p_N)$ is called the *probability vector*. The problem of minimizing the expected number of inspections required to find the FNU appears to be closely related to the problem of determining which units, when inspected, minimize the uncertainty regarding the location of the FNU. This uncertainty can be measured by the notion of *entropy* $U_0(N, P) = -\sum_{i=1}^{N} p_i \log p_i$.

Let $a(k, n, P)$ denote the probability of shifting to the abnormal state no later than unit k in a batch of n units when the process probability vector is P. $a(k, n, P)$ can be computed for each probability vector P. The uncertainty regarding the FNU after inspecting unit k in the batch becomes:

$$\begin{aligned} U_k(n, P) &= (1 - a(k, n, P)) U_0 \left(n-k, \frac{(p_{k+1}, \ldots, p_n)}{\sum_{i=k+1}^{n} p_i} \right) \\ &+ a(k, n, P) U_0 \left(k, \frac{(p_1, \ldots, p_k)}{\sum_{i=1}^{k} p_i} \right). \end{aligned} \qquad (13.9)$$

The authors prove that if only one inspection is available, then it is optimal, with regard to minimizing the amount of remaining uncertainty about the

location of the FNU, to inspect the unit that is closest to dividing the batch into two segments that have equal probability of containing the FNU. Formally, the optimal unit to inspect is $\bar{k} = \arg\min_{k\in 1,\ldots,N} |0.5 - a(k,N,P)|$.

For the geometric case denote p the probability that the process remains in the normal state while producing a unit, i.e., for this case $p_i = \frac{p^{i-1}(1-p)}{1-p^N}$. Thus the unit to inspect first in order to minimize the uncertainty regarding the location of the FNU is $\bar{k} = \left\lfloor \max(1, \log_p(0.5) + \log_p(p^N + 1)) + 0.5 \right\rfloor$. The very same heuristic for the geometric case was proposed in [54] discussed earlier, though it was developed by a completely different method of continuous relaxation of an integer programming problem. The authors in [60] were aware of the result in [54], and in fact generalized it for an arbitrary distribution of the location of the FNU.

Note that the proposed heuristic minimizes the uncertainty, but not necessarily the number of inspections needed to locate the FNU. Each inspection reduces the uncertainty by exactly one unit, ignoring the fact that inspections are conducted on integers. This relaxation means that the initial amount of uncertainty is precisely equal to the expected number of inspections required to identify the FNU. Hence, $D(N,P)$ - the expected number of inspections in the optimal policy is higher than the uncertainty, i.e., $U_0(N,P) \leq D(N,P)$ for all N and P.

Numerical comparisons presented in [60] show that for the geometric case this lower bound is very tight - an average deviation from the optimum is, in most cases, less than one percent. Moreover, the proposed heuristic yields better results (closer to optimum that is computed by dynamic programming) than both the simple binary search and the heuristic developed in [59].

13.2 Various cost formulations- perfect information, zero defects and economic optimization.

Herer, Raz and Grosfeld-Nir (2000) explore in [111] a wider economical aspect of finding the FNU in a production batch. Sometimes it is not economically wise to inspect many items and find the exact place where the items start to be non-conforming (get *"perfect information"* about each item). An alternative policy may be of *"zero defects"*, which does not allow non-conforming units to reach the costumer, but may allow conforming units to be scrapped. Yet another policy is the cost-minimizing policy (*"economic optimization"*), where errors of both kinds are acceptable and the cost of errors is taken into account.

Let p be defined as earlier (the length of time up to the FNU is geometric). Let C_I be the inspection cost per unit; C_P - the penalty of incorrect acceptance: and C_S - the penalty of incorrect rejection.

In the development of the optimal inspection/disposition policy, the authors first find the optimal disposition policy if no inspections are performed at all:

(1) If the quality of the last unit is unknown accept the first

$$j^* = \left\lfloor \frac{\log\left(\frac{C_P}{C_S+C_P}\right)}{\log p} \right\rfloor \tag{13.10}$$

units, and reject the rest. Apparently, the break-even point does not depend on the batch size. $j^* = 0$ means that all units should be rejected. When $j^* > N$ we set $j^* = N$. The minimum expected total cost for a batch of size N with no inspections and the quality of the last unit is unknown is $V^0(N) = C_p[j^* - p\frac{1-p^{j^*}}{1-p}] + C_S\frac{p^{j^*+1}-p^{N+1}}{1-p}$.

(2) If the last unit is known to be non conforming accept the first

$$j' = \left\lfloor \frac{\log\left(\frac{C_P+p^N C_S}{C_S+C_P}\right)}{\log p} \right\rfloor \tag{13.11}$$

units, and reject the rest. As expected, $j' < N$. The minimum expected total cost for a batch of size N with no inspections and the quality of the last unit is known as non conforming is

$$G^0(N) = C_p\frac{j' - p\left(\frac{1-p^{j'}}{1-p}\right)}{1-p^N} + C_S\left[\frac{p^{j'+1}-p^{N+1}}{(1-p)(1-p^N)} - \frac{(N-j')p^N}{(1-p^N)}\right].$$

In case that N is very large, p^N is approximately 0, and the optimal no-inspection policies with or without the knowledge of the last unit's condition coincide.

In order to find the optimal inspection/disposition policy, the following recursive equations are developed:

$$V(k) = \min\left\{\min_{1\le j\le k}\left\{C_I + \mathrm{P}[I_j = 1]G(j) + \mathrm{P}[I_j = 0]V(k-j)\right\}, V^0(k)\right\} \tag{13.12}$$

$$G(k) = \min\left\{\min_{1\le j\le k-1}\left\{C_I + \mathrm{P}[I_j = 1|x_k = 0]G(j) + \mathrm{P}[I_j = 0|x_k = 0]G(k-j)\right\}, G^0(k)\right\}.$$

where:

- $I_j = 0$ if j conforms to specifications and $I_j = 1$ otherwise.
- $V(k)$ is the cost of the optimal inspection/disposition policy, given that the batch size is k and the quality of k is unknown.
- $G(k)$ is the cost of the optimal inspection/disposition policy, given that the batch size is k and unit k is non-conforming.

Boundary conditions are $G(1) = 1$ and $V(0) = 0$, and the computational complexity is $O(N^2)$.

If we wish to use the perfect information approach, we should just set C_S and C_P to be very large. To implement a zero-defect policy, just set C_p to be very large.

The expected number of inspections is calculated for each case: for example for batch size 100 and $p = 0.99$, for perfect information the expected number of comparisons is 5.19, while for the zero-defect approach it is 4.17.

Chun [29] **(2010)** explores the following problem: An item is produced on a high-speed mass production line which is subject to random failures with geometric rate. Every defective item must be located and salvaged (as opposed to [111], where accepting a defective item is possible and bound to a fee). A non-defective item can be accepted or salvaged, in which case a fee must be payed. The problem explored is not only about finding the first defective item among a given lot, but also:

(1) "How often to conduct a regular inspection given a known inspection cost c and an estimated failure rate from past data?

(2) Once a non-conforming item is found, how to conduct an inspection in the last batch and when to stop it? The optimal policy may be to salvage every item in this last batch, or on the contrary - to search for the first defective item until it is detected, all depends on the costs involved."

The resolution of the second question is methodologically similar to that in [111] and is not described here in detail.

To the first question, how often to perform a regular inspection, the author approaches from a Bayesian point of view. Since we assume that the failure process follows a geometric distribution with parameter p, the number of regular inspections k taken until we detect a defective item is also a geometric distribution with parameter p^n (n is the inspection interval): for $k = 1, 2, \cdots, \infty$ $P[k] = p^{n(k-1)}(1 - p^n)$. Let c be the cost of one inspection. The expected profit per a produced item π is:

$$\pi(n) = p^n v_a + (1 - p^n)\frac{EV^*(n)}{n} - \frac{c}{n}. \tag{13.13}$$

where $EV^*(n)$ is the maximum expected total profit obtained for a sequence of n items, when the nth item is known to be flawed.

The optimal inspection interval n^* is n that minimizes $\pi(n)$. The author also suggests a methodology to estimate p. If inspection data are available, we may use one of the statistical estimation methods such as the method of *moments* or the method of *maximum likelihood*. The author chooses to use a Bayes based method, where the prior knowledge of the unknown geometric variable p is expressed as a prior density function of a beta distribution. The estimate for p is developed, but we do not elaborate on it here.

13.3 Search for a state transition point with ability of process recovery after failure

Finkelshtein, Herer, Raz and Ben-Gal [39] **(2005)** extend the work of Raz et al. [111] by taking into account the ability of the production process to recover after failure. Their model is also based on economic optimization - the objective is to define the inspection/disposition procedure that minimizes the sum of the inspection cost and the penalty costs for incorrect disposition decisions. The status of the system during the production of a batch is modeled as a discrete-time two-state (IN and OUT) Markov process. The process typically starts in the IN state, it can switch to OUT state at some point during the production of the batch and it can also switch back to IN state later and so on. The probabilities of IN-OUT switch and of OUT-IN switch are constant and known, p_c and p_n respectively.

Three costs are considered by the model: the cost of one inspection C_I, the cost of incorrect acceptance C_P, and the cost of incorrect rejection C_S. Let S_b and S_e be the status of the system before the start of the batch and at the end of the batch respectively. Each of these variables can have one of three possible values: c (conforming), n (non-conforming) and u (unknown).

Let $P_i^{S_b}$ be the probability that unit i is conforming given that the initial condition of the batch was S_b. Using an induction argument on i one can show that for all $i \geq 0$ $P_i^{S_b}$ is expressible in terms of p_c and p_n:

$$P_i^c = \frac{(1-p_n-p_c)^i p_c + p_n}{p_n + p_c}, \tag{13.14}$$

$$P_i^n = 1 - \frac{(1-p_n-p_c)^i p_n + p_c}{p_n + p_c}. \tag{13.15}$$

Let $a_i^{S_b S_e}(K)$ be the probability that unit i is conforming in a batch of size K, given that before (after) the batch started (completed) the process was in the S_b (S_e) state. Then also $a_i^{S_b S_e}(K)$ can be computed in terms of p_c and p_n:

$$a_i^{S_b S_e} = \begin{cases} \frac{P_i^c P_{K-i}^c}{P_K^c} & (S_b, S_e) = (c,c) \\ \frac{P_i^c (1-P_{K-i}^c)}{1-P_K^c} & (S_b, S_e) = (c,n) \\ \frac{P_i^n (1-P_{K-i}^c)}{1-P_K^n} & (S_b, S_e) = (n,n) \\ \frac{P_i^n P_{K-i}^c}{P_K^n} & (S_b, S_e) = (n,c). \end{cases} \tag{13.16}$$

In order to determine whether inspection is economically justified we must consider the optimal policy if no inspections are performed. Let $W^{S_b S_e}(K)$ be the minimal expected cost of classifying all the units in a batch of size K without inspection, given initial state S_b and final state S_e. Then:

$$W^{S_bS_e}(K) = \sum_{i=1}^{K} \min\left(a_i^{S_bS_e}(K)C_S, \left[1 - a_i^{S_bS_e}(K)\right] C_P\right). \qquad (13.17)$$

The minimal expected cost $G^{S_bS_e}(K)$ of classifying all the units in a batch of size K can be calculated using the following recursion:

$$G^{S_bS_e}(K) = \min\left[W^{S_bS_e}(K), \min_{1\le j\le K} G_j^{S_bS_e}(K)\right]. \qquad (13.18)$$

where $G_j^{S_bS_e}(K)$ is the minimal cost if classifying all the units in a batch of size K, given initial state S_b, final state S_e and that unit j is to be inspected:

$$\begin{aligned} G_j^{S_bS_e}(K) &= C_I + a_j^{S_bS_e}(K)\left(G^{S_bc}(j) + G^{cS_e}(K-j)\right) \\ &+ \left(1 - a_j^{S_bS_e}(K)\right)\left(G^{S_bn}(j) + G^{nS_e}(K-j)\right). \end{aligned} \qquad (13.19)$$

The computational time required for implementing this recursion for a batch of N units is $O(N^2)$.

The authors also provide an easy to implement heuristic to this problem: Take a batch of N units, divide it into sub-batches of l units each for some $l = 1, \ldots, N$ and inspect the last unit of each batch. The expected cost is equal to the sum of expected costs of each of the sub-batches. Evaluate the expected cost for each $l = 1, \ldots, N$ and choose the l with the lowest cost. The performance of the heuristic relative to the optimal solution improves with higher values of failure and recovery probabilities, p_c and p_n.

13.4 Search for a state transition point with process' ability to conduct rework on non-conforming units.

W. Wang, Sheu, Chen and Horng [128] **(2009)** add another layer to economic optimization considerations, the option to conduct rework of non conforming items instead of scrapping them. It is assumed that the proportion of defective units that can be reworked and repaired is constant, given, and denoted by δ. For example, if unit j is found to be non conforming, then a portion δ of units $j+1$ through N is assumed to be repairable while the other portion is considered to be scrap. The reworking cost per unit is C_r, and it is paid only for the units that are repairable. The objective is economic optimization, as defined in [111].

Let P_j denote the probability that the FNU is larger than $j-1$. Note that it is not assumed that the distribution of the FNU is geometric, but the results can definitely be applied to that special case.

The optimal solution is given by the following recursive formulas:

$$ER_V(N) = \max\left\{\max_{1\le j\le N}\left[ER_V^1(N,j)\right], ER_V^0(N)\right\}, \tag{13.20}$$

$$ER_G(N) = \max\left\{\max_{1\le j\le N-1}\left[ER_G^1(N,j)\right], ER_G^0(N)\right\}, \tag{13.21}$$

where $ER_V(N)$ is the expected profit from implementing the optimal inspection policy when the batch size is N, while $ER_G(N)$ is the same value when the last unit is known to be non-conforming. $ER_V^1(N,j)$ is the expected profit from implementing the optimal inspection policy given that the j th unit will be inspected first. $ER_G^1(N,j)$ is the same value when the last unit is known to be non-conforming. $ER_V^0(N)$ is the expected profit from implementing the optimal no-inspection policy when the batch size is N, $ER_G^0(N)$ is the same value when the last unit is known to be non-conforming.

The functions $ER_V^1(N,j)$ and $ER_G^1(N,j)$ are computed recursively and include the rework factor δ:

$$ER_G^1(N,j) = \Pr(FNU \le j \,|FNU \le N)\,\{ER_G(j) + (N-j)\delta(U - C_r)\} +$$

$$\Pr(FNU > j \,|FNU \le N)\,\{ER_G(N-j) + U\cdot j\} - C_I, \tag{13.22}$$

$$ER_V^1(N,j) = \Pr(FNU \le j)\,\{ER_G(j) + (N-j)\delta(U - C_r)\} +$$

$$\Pr(FNU > j)\,\{ER_V(N-j) + U\cdot j\} - C_I. \tag{13.23}$$

where $ER_G(j)$ is the expected profit from implementing the optimal policy to the first j units in the batch given that the jth unit is known to be non conforming, $ER_V(j)$ is the same value if the state of the j's unit is not known and U- the expected profit from the sale of one conforming unit.

The functions $ER_V^0(N)$ and $ER_G^0(N)$ are explicitly computed in manner similar to that in [111], accompanied by a numerical study.

Tsai and C. Wang [123] **(2011)** claim that the results of W. Wang, Sheu, Chen and Horng [128] are correct only if the distribution of the FNU is geometric. Moreover, when an inspection policy is explored, not only reworking the identified nonconforming units but also their rejection should be considered. Finally, the boundary conditions given in the solution procedure of [128] should be corrected; more precisely, when a sub-batch has only one nonconforming unit, the expected payoff is possible to be positive instead of zero as used in [128] since rework is under consideration.

13.5 Search for a state transition point with unreliable answers

In some real life problems, the procedure of inspection for the FNU is not free of errors: conforming units can be mistakenly classified as non conforming and vice versa. Let α denote the type 1 inspection error, that is, the probability of misclassifying a conforming unit; β is the probability of the type 2 inspection error. A common assumption to the articles described below is that each item can be tested only once during the search.

Sheu, Chen, W. Wang and Shin [121] **(2003)** use the same notations and definitions as in [111] to explore search procedures for a geometrically distributed object taking into account inspection errors that might occur during the inspection process. Let x_j receive the values 1 and 0 if unit j is judged to be conforming or non-conforming respectively. Since α and β are assumed to be known, the analysis is conducted in a same manner as in [111], only with $\mathrm{P}[x_j = 1] = p^j(1-\alpha) + (1-p^j)\beta$. Recursion equations are developed and numerical analysis is conducted for the three inspection policies defined in [111]: zero-defects, perfect information and the economic policy.

C. Wang [126] **(2007)** noticed a flaw in the solution suggested in [121]- since the process state is always unknown, the process's Markovian properly cannot be applied, i.e. after one inspection the distribution of the object's location ceases to be geometric. Moreover, the model suggested by [121] underestimates the penalty costs associated with incorrect acceptances and incorrect rejections - not only C_P and C_S should be considered, but also additional loss caused by mistakenly accepting or discarding of several items. For example, if unit j is found to be conforming then according to their model units 1 to j are accepted and no longer searched through, and if unit j turns out non-conforming, only penalty of C_P is payed, while several other items were also misclassified. That additional cost to the firm is ignored in [121].

C. Wang reformulates the model of [111] for the two types of inspection errors. Let $(1-p)$ $(0 < p < 1)$ be the shift probability of the production system. The distribution of the shift location is geometric. For a non-inspection policy the *break-even point (BEP)* is found, i.e., a point such that the expected penalty cost of rejecting all units after it and accepting all units before it is minimal, provided that no inspections are done (in particular the state of the last unit is unknown). The BEP for the non-inspection policy is given in (13.10) in [111].

If the last unit in the batch (of size N) is inspected to be non-conforming, the BEP moves to:

$$j_g = \left\lfloor \frac{\log\left(\frac{C_P + C_S p^N (1-\delta_1)}{C_P + C_S}\right)}{\log(p)} \right\rfloor, \tag{13.24}$$

where $\delta_1 = \frac{\alpha}{1-\beta}$ is defined as the probability ratio for "inspection is wrong" to "inspection is right". j_g is increasing with δ_1. The BEP values are also

calculated for case where the first unit of the batch has been inspected to be conforming and for the case that the first and the last unit in the batch have been inspected to be conforming.

A recursive algorithm for computing the optimal inspection policy is based on the following equation:

$$V^1(k,j) = C_I + [p^j\alpha + (1-p^j)(1-\beta)]G(1,j)+$$

$$[p^j(1-\alpha) + (1-p^j)\beta]H(j,k-j+1) + C_S\alpha\left(\frac{p^{1+j}-p^{1+k}}{1-p}\right)+$$

$$C_P\beta\left(j-1-\frac{p-p^j}{1-p}\right), \tag{13.25}$$

where $G(f,k)$ is the cost of the optimal policy when a batch is of size k, the first unit is f and the last unit has been inspected to be non-conforming, and $H(f,k)$ is the cost of the optimal policy when a batch is of size k, the first unit is f and the first unit has been inspected to be conforming. The complexity of the algorithm is $O(N^3)$.

To determine the batches for which the no inspection policy performs better than the other inspection policies, the author investigates the *threshold batch size* (TBS), i.e., the batch size at which inspection becomes economically justified. If no inspection errors exist, the results are the same as in [111]. Under inspection errors the TBS increases with process reliability (it has been checked for $\alpha = \beta = 0$, $\alpha = \beta = 0.05$, $\alpha = \beta = 0.1$) when $(1-p)$ is relatively small, since the first non-conforming product is produced later. For large shift probabilities ($(1-p) > 0.1$) no results are shown. Also, the TBS is non-decreasing when α and β increase.

Another result is that for economic optimization[18] the expected number of inspections is non-decreasing when inspection errors increase. This is because when inspection errors increase, there will be a preference for choosing the inspected unit as close as possible to the tail or the beginning of the batch. This corrects the mistake in [121] that the expected cost is decreasing with inspection errors.

Chun [27] **(2007)**[19] argues that the problem of dichotomous search with inspection errors is mistreated not only by Chen, Sheu et.al [121] but also by C. Wang [126]. The basic argument is the following: In a model without inspection errors with an a priori geometric distribution for the place of the transition unit, the posteriori distribution (after one inspection) is also geometric. This property is the one that makes dynamic programming applicable to this model. When the basic model is complicated by introducing

[18] recall from [111] that the three policies - economic optimization, perfect information and zero-defects, are formulated as different combinations of the values of cost parameters C_I, C_P and C_S

[19] unpublished

inspection errors into it the posteriori distribution is no longer geometric. Moreover, an inspection does not narrow the interval of uncertainty, since inspection errors are possible. Chun does not offer a new way to solve this problem, but rather shows that it is much harder than its counterpart which has no inspection errors.

Chun [28] **(2008)** publishes a viewpoint summarizing the flaws in the paper of Sheu et.al. [121], one of which is that discussed in [27]. In that viewpoint Chun provides a reformulation of the equations to overcome those mistakes. In a reply to Chun's viewpoint Sheu et.al. accept their critique.

Tzimerman and Herer [124] **(2009)** also consider the problem of inspection errors in quality control over a production line, as do [121] and [126]. The main difference is that previous analyzes wish to classify all items with minimal cost (the economic approach), while the present paper deals with finding the transition unit with a confidence level γ using minimal number of inspections. Accordingly, the previous analyzes assume that after a unit is inspected and found to be conforming (non-conforming), all units preceding (following) the inspected unit are accepted (rejected). Tzimerman and Herer, on the other hand, found that for the different objective a better policy is possible if one considers all the inspection results before accepting or rejecting units.

The location of the transition unit has an arbitrary distribution, in particular it is not assumed to be distributed geometrically. As we shall see, the complexity of the solution for an arbitrary distribution rises and is no longer polynomial.

The method for finding the optimal solution is again dynamic programming. To describe it notations must be introduced: α and β are type 1 and type 2 errors respectively. X_j is 1 if unit j is conforming and -1 otherwise; I_j is 1 if the inspection result indicates that unit j is conforming and -1 otherwise.

Let t_{ji} be the status of unit j by iteration i: 0 if not-inspected, 1 if inspected and found conforming, -1 if inspected and found non-conforming. $T_i := (t_{1i}, \ldots, t_{Ni})$. T_i^{j+} (T_i^{j-}) is the state vector corresponding to iteration i and the subsequent inspection of unit j, which identified the unit as conforming (non-conforming). $f(T_i, j)$ is the expected number of inspections remaining at iteration i if unit j is inspected next; j^* is the optimal unit to be inspected. $f^*(T_i)$ is the minimum expected number of inspections remaining at iteration i (with state vector T_i), given that we inspect the optimal unit j^*. Thus, $f^*(T_j) \equiv f(T_i, j^*)$.

The expected number of inspections remaining at iteration i, if unit $j \neq 0$ is inspected next is equal to one plus the expected number of inspections remaining in the next iteration:

$$f(T_i, j) = 1 + \Pr[I_j = 1 \,|T_i] \cdot f^*(T_{i+1}^{j+}) + \mathrm{P}[I_j = -1 \,|T_i] \cdot f^*(T_{i+1}^{j-}). \tag{13.26}$$

As long as the stopping criteria is unmet (we have not yet identified the transition unit with required confidence and there are still units to inspect), we choose to inspect j^* - the unit which, after its inspection, yields the minimal expected number of remaining inspections. The complexity of the dynamic program is $O(N3^N)$ because there are 3^N possible T_i vectors and $f_i(T_i, j)$ must be computed for each j.

Since the dynamic program is not practical for large batches, several heuristic solutions are also introduced and compared, but we don't describe them in this survey.

13.6 Search for a state transition point with non-conforming objects in the normal state and conforming objects in the abnormal state

In another extension of [111], **Bendavid and Herer** [23] **(2009)** consider a process in which non-conforming units can also be produced in the IN state and conforming units in the abnormal state. The inspection is assumed to be error free.

Let α_I be the probability of producing a non-conforming unit when the process is in the normal state, and α_O the probability of producing a non-conforming unit when the process is in the abnormal state. It is assumed that $\alpha_O > \alpha_I$. C_I, C_P and C_S are defined as previously. The distribution of the transition point location is assumed to be arbitrary (not necessarily geometric)[20]. At each stage, we can inspect an item that hasn't been inspected yet, or to stop inspecting, accept all the non-inspected items and pay a penalty C_P for each for each Let $S = (s_1, \ldots, s_N)$ be a "vector of states" that describes the status of the units in the batch: $s_i = u$ means that unit i has not been inspected and its quality is unknown, $s_i = n$ means that unit i has been inspected and found to be non conforming, $s_i = c$ means that unit i has been inspected and found to be conforming. Let $P(S) = (p_1(S), \ldots, p_N(S))$ be the vector of probabilities that the process remained in the IN state given the vector S. Initially, $P(u, u, \ldots, u) = (p_1, \ldots, p_N)$.

$P_C^n(S) = (p_1^n(S), \ldots, p_N^n(S))$ and $P_C^c(S) = (p_1^c(S), \ldots, p_N^c(S))$ are the vectors of probabilities that the units are non conforming / conforming respectively given the vector S. The authors explain how to compute the vectors $P_C^n(S), P_C^c(S)$ using a $O(N)$ time complexity procedure. $f(S)$ is the cost of the optimal inspection/disposition policy for a given vector S. A notation $S\,|s_k \leftarrow c$ instead of S is used to represent the vector of states S with its kth element (which is presently u) replaced by c.

The cost of the optimal no-inspection policy, $W(S)$ is calculated using $O(N)$ operations as the expected penalty cost for all the units, i.e., in the optimal no-inspection policy we accept all units and pay penalties for every unit that

[20]Here also we see that the generalization of the problem to arbitrary distribution rather than geometric causes a rise in the complexity of the solution, which ceases to be polynomial

turned out to be non-conforming. The dynamic program is formulated as follows:

- "Initial condition: $f(S) = 0$ when $s_i \neq u$ for all $i = 1, \ldots, N$.
- Recursive function:

$$f(S) = \min \left[\min_{j \in \{1,\ldots,N | s_j = u\}} \left\{ C_I + p_j^c(S) f(S \,|s_j \leftarrow c) + p_j^n(S) f(S \,|s_j \leftarrow n) \right\}, W(S) \right]. \text{''} \tag{13.27}$$

- The goal: to find $f(S)$ when $S = (u, u, \ldots, u)$.

In a preprocessing step, the authors calculate the vectors of probabilities that the units are conforming or non conforming, for the 3^N possible vectors of states S, and keep them in a matrix of size $3^N \cdot N$. Thus, all the probabilities in the matrix can be calculated in $O(3^N N)$ time. For each of the 3^N vectors, the optimal cost of inspecting each of the N possible units must be calculated, thus the computational complexity of the dynamic program is $O(3^N N)$.

Since the computational complexity of the dynamic programming is very high, the authors develop several heuristics:

(1) The "greedy in uncertainty" selection rule instructs us to inspect unit i such that the probability that a transition occurs at or before it is closest to 0.5, i.e., $i = \text{argmin}_{i \in \{1,\ldots,N | s_i = u\}} \left| 0.5 - \sum_{j=1}^{i} p_j^T(S) \right|$. Recall that Herer and Raz (2000) [111] demonstrated that such a unit minimizes the expected remaining uncertainty regarding the location of the transition point in the batch.

(2) Under the "greedy in cost" selection rule, we inspect the unit that minimizes the expected no-inspection cost obtained after performing one inspection:

$$i = \text{argmin}_{i \in \{1,\ldots,N | s_i = u\}} \left\{ p_i^c(S) W(S, s_i \leftarrow c) + p_i^n(S) W(S, s_i \leftarrow n) \right\}.$$

(3) In the "myopic" stopping rule we inspect unit i if disposing of the batch without any additional inspection is more expensive than inspecting unit i and then disposing of the batch without any further inspections. We inspect i if $W(S) > h^1(S) \equiv C_I p_i^c(S) W(S, s_i \leftarrow c) + p_i^n(S) W(S, s_i \leftarrow n)$.

(4) For the "look ahead" stopping rule the authors define

$$h^j(S) = C_I + p_i^c(S) \min \left\{ W(S, s_i \leftarrow c) \cdot h^{j-1}(S, s_i \leftarrow c) \right\} +$$

$$p_i^n(S) \min \left\{ W(S, s_i \leftarrow n) \cdot h^{j-1}(S, s_i \leftarrow n) \right\}. \tag{13.28}$$

$h^i(S)$ can be interpreted as the expected cost obtained after performing up to j inspections and then proceeding according to the

optimal no-inspection policy. In the "look-ahead" stopping rule we perform the inspection iff $W(S) > h^{\lfloor \log N \rfloor + 1}(S)$.

The four combinations of a selection rule (1 or 2) with a stopping rule (3 or 4) create four heuristic policies, which are studied and compared in the article. The major conclusion is the dominance of the heuristic composed of the greedy in cost selection rule and the look-ahead stopping rule over all others.

C. Wang, Shih and Tsai [127] **(2011)** also develop an inspection policy for this problem, only their objective is different - not minimizing the overall cost subject to penalties for classification errors, but identifying the transition unit with a given confidence level γ. In this model it is allowed to inspect each unit more than once at a time, to increase the confidence level, but it is not allowed to return to inspect a unit after other units have been inspected. The suggested policy is not optimal in a sense of minimizing the number of inspections or the total cost as in [111], but each selection of item to inspect next is the one that minimizes the uncertainty of the transition unit, or equivalently minimizes the expected entropy.

The authors do not wish to find an optimal policy, but to build a heuristic according to the principle of minimizing the entropy with only one constraint - a given confidence level of identifying the transition point.

The heuristic suggested is not a recursive algorithm, and not costly in running time. Numerical examples indicate that for a batch of size < 35, the expected number of inspections to meet a confidence level of $\gamma = 0.95$ doesn't exceed 3.5. When the required confidence level is set to one, all units of the batch should be inspected. The heuristic is not compared to any optimal results or different heuristics.

14. Applications of dichotomous search games in economics

Some work is done in the field of applying dichotomous search models in economics. We discuss here two examples of such applications: Gathering information from inventories and making optimal wage request.

14.1 Gathering information from inventories

Firms may use their inventories to gain information about the product demand they face: by observing the inventories remaining from its sales, the firm can tell whether its quantity put up for sale is "high" or "low" relative to the demand. In [7], **Alpern and Snower (1987)** develop a model for a special case of this problem, where the price of the product is fixed through time, and propose an optimal "learning strategy" for the firm. Let p be the price of the firm's output, f the unit cost of production and h the cost of holding one unit of inventory for one period of time. The firm knows that its product demand D is constant through time and lies in a known interval $[\underline{D}, \bar{D}]$ over which it is uniformly distributed. At the beginning of time period k the firm produces an amount Q_k. The inventory stock carried from

the previous period is $(1-\delta)I_{k-1}$, where I_{k-1} is the inventory stock held at the end of that period and $0 \leq \delta < 1$ is the inventory depreciation rate. At period k the firm puts up for sale a quantity $G_k = (1-\delta)I_{k-1} + Q_k$, and the amount which it sells is $Z_k = \min(G_k, D)$. If $I_k = 0$, then the firm learns that the true demand must lie in a smaller uncertainty interval $[G_k, \bar{D}]$. If $I_k > 0$ then the true demand is revealed $D = Z_k = G_k - I_k$ and the firm future supplies will all be equal to D. If the demand is greater than the supply, the difference (the "lost sales") is not left for the next period.

Let α be the time discount factor. Let a "supply strategy" $S = (S_1, S_2, \dots)$ be a sequence of numbers which determines the supply decisions G_t given the information set $J_t = \{G_1, \dots, G_{k-1}, I_0, \dots, I_{k-1}\}$. For a strategy S and any D in $[0, 1]$ the firm's opportunity cost is defined by:

$$c(S, D) = \sum_{t=1}^{N-1} \alpha^{t-1}(D - S_t) + \alpha^{N-1} b(S_N - D), \tag{14.1}$$

where $N = N(S, D)$ is the least t with $S_t > D$ and

$$b = \frac{h + f(1 - \alpha(1-\delta))}{p - f}. \tag{14.2}$$

The firm's objective is to minimize the expectation of opportunity cost over all supply strategies. The main result of the paper states that the optimal quantity S_k^* for $k = 1, 2, \dots$ to be put up for sale in the k's period, provided that the previous supplies have resulted in stock-outs, is

$$S_1^* = \lambda = \frac{\alpha b - b - 1 + \sqrt{(b - \alpha b + 1)^2 + 4\alpha b}}{2\alpha b}, \tag{14.3}$$

$$S_k^* = 1 - (1 - \lambda)^k \quad \text{for} \quad k = 2, 3, \dots \tag{14.4}$$

The minimum expected cost is

$$V = \frac{\alpha b - b - 1 + \sqrt{(b + 1 - \alpha b)^2 + 4\alpha b}}{4b}. \tag{14.5}$$

In [8] **(1988) Alpern and Snower** address the same problem, but restricted to one or two periods.

Extensions of this simple model above are surveyed in [7]: "Through the application of the mathematical theory of dichotomous search, it has been extended to cover infinite time horizons (Aghion, Bolton and Jullien [3] (1987); Alpern and Snower, [7] (1987)), "conservative" (minimax) production strategies rather than Bayesian updating (Alpern and Shower [6] (1987)) and random variations on the actual product demand (Reyniers [113], 1987)."

Alpern and Snower [9] (1988) set out a methodology for a more general case of production decision, where the firm is allowed to change the price of the product between periods of time. They present the general model of the problem but do not propose a solution.

Subsequently, variations on the basic model have been studied. Among these, an information delay in which the outcomes of sales on period k are not known to the supplier until period $k+2$ (Reyniers [114],[116]) and an effect whereby stock outs decrease future demand (Reyniers [115]). Those studies fall beyond the scope of this paper.

14.2 Wage bargaining - optimal wage request

Alpern and Snower (1988) [8] apply dichotomous search ideas to labor economics. In their paper, the worker tries to find out his value to the firm by making wage demands. If the wage demand is lower than the worker's value to the firm, the worker is hired at that wage demand. This wage then becomes a new lower bound on the interval of uncertainty for the worker's value. It is assumed that the firm is myopic and hires the worker whenever his wage demand is below his value. This means that the firm is not strategic, i.e. does not try to manipulate the worker to lower his wage demand by not hiring him when his demand is above his value to the firm. It is also assumed that the worker approaches the same firm at every period.

For simplicity it is assumed that the worker only lives for two periods and that his value is uniformly distributed in $[0,1]$. Given that the worker is trying to learn about his value he will take into account both scenarios (hired/ not hired in the first period) when planning his strategy. The worker can solve a simple dynamic programming problem trying to maximize his total discounted expected payoff ($0 \leq \delta_w \leq 1$ is the worker's discount factor) over the two periods, in which the second period wage demand is determined first. It is shown that the optimal first period wage demand is $w_1 = \frac{1+\delta_w}{2+1.5\delta_w}$ and the optimal second period wage is $w_2 = \max(w_1, 0.5)$ if the worker is hired in the first period, or $w_2 = 0.5w_1$ otherwise. The worker's optimal expected payoff is:

$$P_w = \frac{(1+\delta_w)^2(1+0.75\delta_w)}{(2+1.5\delta_w)^2}. \tag{14.6}$$

The authors also compute the total expected payoff to the firm. Let $0 \leq \delta_f \leq 1$ be the firm's discount factor. First period unemployment probability is w_1 (since the worker's value is uniformly distributed in $[0,1]$) and second period unemployment probability is zero for those already accepted in the first period and 0.5 for those who were not accepted in the first period. The total expected discounted payoff to the firm in this model is:

$$\begin{aligned} P_f &= (1+\delta_f)\int_{w_1}^{1}(v-w_1)dv + \delta_f\int_{0.5w_1}^{w_1}(v-0.5w_1)dv = \\ &= \frac{\delta_f w_1^2}{8} + \frac{1+\delta_f}{2}(1-w_1)^2. \end{aligned} \tag{14.7}$$

Note that in this model the firm is not strategic, thus its expected payoff is determined by the worker's optimal policy.

Reyniers [117] **(1992)** examines a wage bargaining problem where the firm, which the worker approaches at every period, is allowed to behave strategically rather than just hire the worker whenever his wage demand is below his value. The firm has an incentive in preventing the worker from learning that he has a high value and will therefore try to influence the worker's second period wage demand through its hiring/firing decisions in the first period. The worker is assumed to be naive and believes that the firm is myopic, and thus makes wage demands as described above. In the second period, the firm does not have an incentive to be 'deceptive' and will hire the worker if his wage demand does not exceed his value. It is shown that the firm should hire the worker at his wage demand w_1 in period 1 if the worker's value v satisfies $v \geq w_1(1+0.5\delta_f)$. The firm's expected payoff is

$$P_f = \frac{\delta_f}{8}w_1^2(1+\delta_f)^2 + \frac{1+\delta_f}{2}(1-w_1(1+\frac{\delta_f}{2}))(1-w_1(1-\frac{\delta_f}{2}), \tag{14.8}$$

and the worker's expected payoff is

$$P_w = (1+\delta_w)w_1(1-w_1(1+\frac{\delta_f}{2})) + \frac{\delta_w}{4}w_1^2(1+\delta_f). \tag{14.9}$$

Thus, if the firm is extremely impatient ($\delta_f = 0$) the worker's expected payoff reduces to his expected payoff in the myopic form model. The more patient the firm is, the larger is the difference between worker's expected payoff in the two models.

Note that in [8] both the firm and the worker are assumed to be myopic, while in [117] the firm is assumed to be strategic and the worker is assumed to be myopic. We found no literature on the game where both the firm and the worker are myopic.

References

[1] N. Abigadol and A. Ben-Tal. A minmax search for the critical level of a system: The asymmetric case. *Naval Research Logistics Quarterly*, 32:137–154, 1985. (Cited on page 31.)

[2] J. Abrahams. Code and parse trees for lossless source encoding. *unpublished*, 1997. (Cited on page 67.)

[3] P. Aghion, P. Bolton, and B. Jullien. Learning through price experimentation by a monopolist facing unknown demand. *Working paper 8748, University of Californa-Berkley, 1987*, 1987. (Cited on page 90.)
[4] M. Ahmed, A. Belal, and K. Ahmed. Optimal insertion in two-dimensional arrays. *International Journal of Information Sciences*, 99:1–20, 1997. (Cited on page 65.)
[5] S. Alpern. Search for a point in interval, with high-low feedback. *Mathematical Proceedings of the Cambrige Philosophy Society*, 98:569–578, 1985. (Cited on page 47 and 74.)
[6] S. Alpern and D. Snower. Production desicions under demand uncertainty: the high-low search approach. *Discussion paper no. 223, Center for economic policy research (CERP)*, 1987. (Cited on page 90.)
[7] S. Alpern and D.J. Snower. Inventories as an information-gathering device. *World Bank Development Research, Report no. DRD267*, 1987. (Cited on page 1, 89, and 90.)
[8] S. Alpern and D.J. Snower. "high-low search" in product and labor markets. *AEA Papers and Proceedings*, 78:356–362, 1988. (Cited on page 1, 90, 91, and 92.)
[9] S. Alpern and D.J. Snower. A search model of optimal pricing and production. *Journal of Engineering Costs and Production Economics*, 15:279–284, 1988. (Cited on page 1 and 91.)
[10] A. Andersson. A note on searching in a binary search tree. *Software: Practice and Experience 21*, 10:1125–1128, 1991. (Cited on page 15.)
[11] S. Anily and R. Hassin. Ranking the best binary trees. *SIAM Journal on Computing*, 18:882–892, 1989. (Cited on page 14.)
[12] S.M. Arafat. An efficient simple algorithm for building optimal alphabetic trees in parallel. *Faculty of Computer and Information Sciences, Ain Shams University, Egypt*. (Cited on page 20.)
[13] M. Atallah, S. Rao Kosaraju, L. Larmore, G. Miller, and S. Teng. Constructing trees in parallel. *Proccedings of the 1st Symposium on Parallel Algorithms and Architectures*, pages 499–533, 1989. (Cited on page 19.)
[14] M. Avriel and D. Wilde. optimality proof for the symmetric fibonacci search technique. *The Fibonacci Quarterly*, 4:265–269, 1966. (Cited on page 25.)
[15] M.B. Baer. Alphabetic coding with exponential costs. *Information Processing Letters*, 110:139–142, 2010. (Cited on page 51.)
[16] A. Barkan and H. Kaplan. Partial alphabetic trees. *Journal of algorithms*, 58:81–103, 2006. (Cited on page 67.)
[17] V.J. Baston and F.A. Bostock. A high-low search game on the unit interval. *Mathematical Proceedings of the Cambrige Philosophy Society*, 97:345–348, 1985. (Cited on page 47 and 74.)
[18] P. Bayer. Improved bounds on the cost of optimal and balanced binary search trees. *M. Sc. Thesis, MIT, Cambrige*, 1975. (Cited on page 18.)
[19] A. Belal, M.A. Ahmed, and S.M. Arafat. Limiting the search for 2- dimentional alphabetic trees. *Fourth International Joint Conference on Information Sciences, North California, USA*, 1998. (Cited on page 64.)
[20] A. Belal, M. Selim, and S. Arafat. Merging optimal alphabetical trees in linear time. *Proceedings of the First International Conference on Intelligent Computing and Information Systems ICICIS, Cairo, Egypt*, 2002. (Cited on page 16, 17, 20, and 21.)
[21] A.A. Belal, M.S. Selim, and S.M. Arafat. Building optimal alphabetic trees recursively. *Proceedings of the Third WSEAS International Multiconference on Mathematics, Wolin Island, Poland*, 2002. (Cited on page 17 and 20.)
[22] A.A. Belal, M.S. Selim, and S.M. Arafat. Towards a dynamic optimal alphabetic tree. *International Journal of Cooperative Information Systems*, 4:46–50, 2004. (Cited on page 17.)
[23] I. Bendavid and Y.T. Herer. Economic optimization of off-line inspection in a process that also produces non-conforming units when in control and conforming units when out of control. *European Journal of Operational Research*, 195:139–155, 2009. (Cited on page 87.)

[24] D.A. Berry and R.F. Mensch. Discrete search with directional information. *Operationas Research*, 34:470–477, 1986. (Cited on page 47.)
[25] S.H. Cameron and S.G. Narayanamurthy. A search problem. *Operations Research*, 12:623–629, 1964. (Cited on page 1, 26, 27, 28, 29, 30, and 41.)
[26] Yung-Ching. Chu. An extended result of kleitman and saks concerning binary trees. *Discrete Aplied Mathematics*, 10:255–259, 1985. (Cited on page 19.)
[27] Y.H. Chun. Effects of inspection errors on dichotomous inspection procedures. *submitted to Journal of Operational Research Society*. (Cited on page 57, 85, and 86.)
[28] Y.H. Chun. Economic optimization of off-line inspection procedures with inspection errors (viewpoints). *Journal of Operational Research Society*, 59:863–870, 2008. (Cited on page 86.)
[29] Y.H. Chun. Bayesian inspection model for the production process subject to a random failure. *IIE Transactions*, 42:304–316, 2010. (Cited on page 80.)
[30] K.L. Chung, W.C. Chen, and F.C. Lin. On the complexity of search algorithms. *IEEE Transactions on Computers*, 41:1172–1176, 1992. (Cited on page 40.)
[31] F. Cicalese and U. Vaccaro. Binary search with delayed and missing answers. *Information processing letters*, 85:239–247, 2003. (Cited on page 56.)
[32] D. Coppersmith, M.M. Klawe, and N.J. Pippenger. Alphabetic minimax trees of degree at most t. *SIAM Journal on Computing*, 15:189–192, 1986. (Cited on page 50 and 51.)
[33] P. Damaschke. An optimal parallel algorithm for digital curve segmentation. *Theoretical computer science*, 178:225–236, 1997. (Cited on page 59.)
[34] R. De Prisco and A. De Santis. On optimal binary search trees. (Cited on page 18.)
[35] D. Dobkin and R.J. Lipton. Multidimentional searching problems. *SIAM Journal on Computing*, 5:181–186, 1976. (Cited on page 63.)
[36] P.S. Efraimidis. (a, b) fibonacci search. *Technical Report LPDP-2010-02, Department of Electrical and Computer Engineering, Democritus University of Thrace*. (Cited on page 34.)
[37] D.E. Ferguson. Fibonaccian searching. *Communications of the ACM*, 3:648, 1960. (Cited on page 24 and 40.)
[38] T.S. Ferguson. A problem of minimax estimation with directional information. *Statistics and Probability Letters*, 26:205–211, 1996. (Cited on page 73.)
[39] A. Finkelshtein, Y.T. Herer, T. Raz, and I. Ben-Gal. Economic optimization of off-line inspection in a process subject to failure and recovery. *IIE Transactions*, 37:995–1009, 2005. (Cited on page 81.)
[40] H. Fujiwara and T. Jacobs. On the huffman and alphabetic tree problem with general cost functions. *Algorithmica and Springer Science + Business Media New York*, 2013. (Cited on page 48.)
[41] T. Gagie. A new algorithm for building alphabetic minimax trees. *Fundamenta Informaticae*, 97:321–329, 2009. (Cited on page 50 and 51.)
[42] S. Gal. A discrete search game. *SIAM Journal of Applied Mathematics*, 27:641–648, 1974. (Cited on page 26 and 71.)
[43] S. Gal. A stochastic search game. *SIAM Journal of applied mathematics*, 34:205–210, 1978. (Cited on page 72 and 74.)
[44] M.R. Garey. Optimal binary search trees with restricted maximal depth. *SIAM Journal on Computing*, 3:101–110, 1974. (Cited on page 13, 52, and 53.)
[45] A.M. Garsia and M.L. Wachs. A new algorithm for minimum cost binary trees. *SIAM Journal on Computing*, 6:622–642, 1976. (Cited on page 10, 11, and 20.)
[46] P. Gawrychowski. Alphabetic minimax trees in linear time. *Institute of Computer Science, Wroclaw, Poland*, 2012. (Cited on page 51.)
[47] E.N. Gilbert. Games of identification or convergence. *SIAM Review*, 4:16–24, 1962. (Cited on page 70.)
[48] E.N. Gilbert and E.F. Moore. Variable length binary encodings. *The Bell System Technical Journal*, 38:933–968, 1958. (Cited on page 7 and 51.)
[49] M. Golin and J. Lin. More efficient algorithms and analyses for unequal letter cost prefix-free coding. *IEEE Transactions on Informatiom Theory*, 54:3412–3424, 2008. (Cited on page 63.)

[50] L. Gotlieb. Optimal multi-way search trees. *SIAM Journal on Computing*, 10:422–433, 1981. (Cited on page 60.)
[51] L. Gotlieb and D. Wood. The construction of optimal multiway search trees and the monotonicity pronciple. *International Journal of Computer Mathematics*, 9:17–24, 1981. (Cited on page 60.)
[52] M. Grotschel, L. Lovasz, and A. Schrijver. The elipsoid method and its consequences in combinatorial optimization. *Report No. 80151-OR, Institut fur Okonometry und Operations Research, Universitat Bonn W.Germany*, 1980. (Cited on page 69.)
[53] R. Hassin. On maximizing functions by fibonacci search. *The Fibonacci Quarterly*, 17:347–351, 1981. (Cited on page 25.)
[54] R. Hassin. A dichotomous search for a geometric random variable. *Operations Research*, 32, 1984. (Cited on page 13, 74, 75, 76, 77, and 78.)
[55] R. Hassin and M. Henig. Dichotomous search for random objects on an interval. *Mathematics of Operational Research*, 9:301–308, 1984. (Cited on page 58.)
[56] R. Hassin and M. Henig. Monotonicity and efficient computation of optimal dichotomous search. *Discrete Applied Mathematics*, 46:221–234, 1993. (Cited on page 21, 22, 23, and 53.)
[57] R. Hassin and R. Hotovely. Asymptotic analysis of dichotomous search with search and travel costs. *European Journal of Operational Research*, 58:78–89, 1992. (Cited on page 35, 42, and 43.)
[58] R. Hassin and N. Megiddo. An optimal algorithm for finding all the jumps of a monotone step-function. *Journal of Algorithms*, 6:265–274, 1985. (Cited on page 57, 58, and 59.)
[59] Q. He, Y. Gerchak, and A. Grosfeld-Nir. Optimal inspection order when process failure rate is constant. *International Journal of Reliability, Quality and Safety Engineering*, 3:25–41, 1996. (Cited on page 76 and 78.)
[60] Y.T. Herer and T. Raz. Optimal parallel inspection for finding the first nonconforming unit in a batch - an information theoretic approach. *Management science*, 46:845–857, 2000. (Cited on page 1, 61, 77, and 78.)
[61] K. Hinderer. On dichotomous search with direction-dependent costs for a uniformly hidden object. *Optimization*, 21:215–229, 1990. (Cited on page 26, 30, and 31.)
[62] K. Hinderer and M. Stieglits. Isotonicity of minimizers in polychotomous discreet interval search via lattice programming. *Mathematical methods of operational research*, 51:139–173, 2000. (Cited on page 23.)
[63] D.S. Hirschberg. On the complexity of searching a set of vectors. *SIAM Journal of Applied Mathematics*, 9:126–129, 1980. (Cited on page 64.)
[64] S.W. Hornick, S.R. Madilla, E.P. Mucke, H. Rosenberg, S. Sol Skiena, and I.G. Tollins. Searching on a tape. *IEEE Transactions on Computers*, 39:1265–1271, 1990. (Cited on page 40, 41, and 42.)
[65] A.J. Hu. Selection of the optimum uniform partition search. *Computing*, 37:261–264, 1986. (Cited on page 26, 36, and 41.)
[66] T.C. Hu. A new proof of the t-c algorithm. *SIAM Journal of Applied Mathematics*, 25:83–94, 1973. (Cited on page 10.)
[67] T.C. Hu, D.J. Kleitman, and J.K. Tamaki. Binary trees optimum under various criteria. *SIAM Journal of Applied Mathematics*, 37:246–256, 1979. (Cited on page 49 and 51.)
[68] T.C. Hu, L.L. Larmore, and J.D. Morgenthaler. Optimal integer alphabetic trees in linear time. *Algorithms - ESA 2005, Lecture notes in computer science*, 3669:226–237, 2005. (Cited on page 13.)
[69] T.C. Hu and J.D. Morgenthaler. Optimum alphabetic binary trees. *Lecture Notes in Computer Science, Springer-Verlag*, 1120:234–243, 1996. (Cited on page 11.)
[70] T.C. Hu and K.C. Tan. Path length of binary search trees. *SIAM Journal of Applied Mathematics*, 22:225–234, 1972. (Cited on page 10 and 52.)
[71] T.C. Hu and A.C Tucker. Optimal computer-search trees and variable-length alphabetical codes. *SIAM Journal of Applied Mathematics*, 21:514–532, 1971. (Cited on page 7, 8, 10, 11, 13, 16, 20, 49, and 75.)

[72] T.C. HU and P.A. Tucker. Optimal alphabetic trees for binary search. *Information Processing Letters*, 67:137–140, 1998. (Cited on page 15.)
[73] T.C. Hu and M.L. Wachs. Binary search on a tape. *SIAM Journal on Computing*, 16:573–590, 1987. (Cited on page 38.)
[74] D. Huffman. A method for the construction of minimum-redundancy codes. *Proccedings of the Institute of Radio Engineers*, 40:1098–1101, 1952. (Cited on page 5.)
[75] P. Humblet. Generalization of huffman coding to minimize the probability of buffer overflow. *IEEE Transactions Information Theory*, IT-27:230–232, 1981. (Cited on page 51.)
[76] H.K. Hwang and T.H. Tsai. An asymptotic theory for recurrence relations based on minimization and maximization. *Theoretical Computer Science*, 290:1475–1501, 2003. (Cited on page 21.)
[77] A. Itay. Optimal alphabetic trees. *SIAM Journal on Computing*, 5:9–18, 1976. (Cited on page 31, 53, 59, 60, 62, and 63.)
[78] S.M. Johnson. A search game. *The RAND corporation, memarandum RM-3717-PR*, 1964. (Cited on page 71.)
[79] R.M. Karp. A generalization of binary search. *Lecture Notes in Computer Science*, 709:27–34, 1993. (Cited on page 59.)
[80] M. Karpinski, L.L. Larmore, and W. Rytter. Correctness of constructing optimal alphabetic trees revisited. *DIMACS Technical Report 96-54*, 1996. (Cited on page 10.)
[81] J. Kiefer. sequential minimax search for a maximum. *Proceedings of the American Mathematics Society*, 4:502–506, 1953. (Cited on page 25.)
[82] J.H. Kingston. A new proof of the garsia-wachs alorithm. *Journal of Algorithms*, 9:129–136, 1988. (Cited on page 10.)
[83] D.G. Kirkpatrick and M.M. Klawe. Alphabetic minimax trees. *SIAM Journal on Computing*, 14:514–526, 1985. (Cited on page 50 and 51.)
[84] M. Klawe and B. Mumey. Upper and lower bounds on constructing alphabetic binary trees. *SIAM Journal of Discrete Mathematics*, 8:638–651, 1995. (Cited on page 10 and 11.)
[85] D.J. Kleitman and M.E. Saks. Set orderings requiring costliest alphabetic binary trees. *SIAM journal of algorithms in Discreet Mathematics*, 2:142–146, 1981. (Cited on page 18 and 19.)
[86] W.J. Knight. Search in an ordered array having variable probe cost. *SIAM Journal on Computing*, 17:1203–1214, 1988. (Cited on page 45.)
[87] D.E. Knuth. Optimum binary search trees. *Acta Informatica*, 1:14–25, 1971. (Cited on page 1, 3, 7, 22, and 48.)
[88] S. Kwek and K. Melhorn. Optimal search for rationals. *Information Processing Letters*, 86:23–26, 2003. (Cited on page 69.)
[89] L.L. Larmore. Heigth restricted optimal binary trees. *SIAM Journal on Computing*, 16:1115–1123, 1987. (Cited on page 52.)
[90] L.L. Larmore and D.S. Hirschberg. A fast algorithm for optimal length-limited huffman codes. *Journal of the ACM*, 37:464–473, 1990. (Cited on page 53.)
[91] L.L. Larmore and M. Przyticka. Length limited coding and optimal height-limited binary trees. *Tech report 88-01, ICS dept., University of California, Irvine, CA*, 1988. (Cited on page 53, 54, and 55.)
[92] L.L. Larmore and T.M. Przyticka. A fast algorithm for optimal height-limited alphabetic binary trees. *SIAM Journal on Computing*, 23:1283–1312, 1994. (Cited on page 53, 54, and 55.)
[93] L.L. Larmore and T.M. Przytycka. Parallel construction of trees with optimal weighted path length. *Proceedings of the 3rd ACM Symposium on Parallel Algorithms and Architectures*, pages 71–80, 1991. (Cited on page 20.)
[94] L.L. Larmore and T.M. Przytycka. A parallel algorithm for optimum height-limited alphabetic binary trees. *Journal of Parallel and Distributed Computing*, 35:49–56, 1996. (Cited on page 55.)
[95] L.L. Larmore and T.M. Przytycka. The optimal alphabetic tree problem revisited. *Journal of Algorithms*, 28:1–20, 1998. (Cited on page 12 and 13.)

[96] L.L. Larmore, T.M. Przytycka, and W. Rytter. Parallel construction of optimal alphabetic trees. *SPAA '93 Proceedings of the 5th annual ACM Symposium on Parallel Algorithms and Architectures*, pages 214–223, 1993. (Cited on page 20.)

[97] T. Lepala. On a generalization of binary search. *Information Processing Letters*, 8:230–233, 1979. (Cited on page 18 and 26.)

[98] N. Megiddo. Combinatorial optimization with rational objective functions. *Mathematical Operations Research*, 4:414–424, 1979. (Cited on page 68.)

[99] K. Melhorn. Nearly optimal binary search trees. *Acta informatica*, 5:278–295, 1975. (Cited on page 17, 18, and 26.)

[100] K. Melhorn. A best possible bound for the weighted path length of binary search trees. *SIAM Journal on Computing*, 6:235–239, 1977. (Cited on page 18.)

[101] R. Morris. Some theorems on sorting. *SIAM Journal of Applied Mathematics*, 17 No.1:423–439, 1969. (Cited on page 1, 26, and 30.)

[102] S. Murakami. A dichotomous search. *Journal of the Operations Research Society of Japan*, 14:127–142, 1971. (Cited on page 1, 26, 28, 30, and 31.)

[103] S. Murakami. A dichotomous search with travel cost. *Journal of the Operations Research Society of Japan*, 19:245–254, 1976. (Cited on page 35 and 36.)

[104] S. Nishihara and H. Nishino. Binary search revisited: Another advantage of fibonacci search. *IEEE Transactions on Computers*, C-36:1132–1135, 1987. (Cited on page 40 and 41.)

[105] K.J. Overholt. Efficiency of the fibonacci search method. *BIT Numerical Mathematics*, 13:92–96, 1973. (Cited on page 25.)

[106] C.H. Papadimitriou. Efficient search for rationals. *Information Processing Letters*, 8:1–4, 1979. (Cited on page 68.)

[107] D. Parker. Conditions for optimality of the huffman algorithm. *SIAM Journal on Computing*, 9:480–489, 1980. (Cited on page 51.)

[108] P. Ramanan. Testing the optimality of alphabetic trees. *Theoretical Computer Science*, 93:279–301, 1992. (Cited on page 16.)

[109] S. Rao Kosaraju. On a multidimensional search problem. *STOC '79 Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing*, 1979. (Cited on page 64.)

[110] S. Rao Kosaraju, T.M. Przytycka, and R. Borgstrom. On an optimal split tree problem. *Lecture notes in Computer Science, Algorithms and Data Structures, 6th international Workshop WADS99, Vancouver, Canada, August 11-14 1999, proceedings*, 1999. (Cited on page 66.)

[111] T. Raz, Y.T. Herer, and A. Grosfeld-Nir. Economic optimization of off-line inspection. *IIE Transactions*, 32:205–217, 2000. (Cited on page 1, 78, 80, 81, 82, 83, 84, 85, 87, 88, and 89.)

[112] S.P. Reiss. Rational search. *Information Processing Letters*, 8:89–90, 1979. (Cited on page 68.)

[113] D. Reyniers. Active learning about the demand distribution in the newsboy problem. 1987. (Cited on page 90.)

[114] D. Reyniers. A high-low search model of inventories with time delay. *Journal of Engineering Costs and Production Economics*, 15:417–422, 1988. (Cited on page 1 and 91.)

[115] D. Reyniers. Interactive high-low search: the case of lost sales. *Journal of the Operational Research Society*, 40:769–780, 1989. (Cited on page 91.)

[116] D. Reyniers. A high-low search algorithm for a newsboy problem with delayed information feedback. *Operational Research*, 38:838–846, 1990. (Cited on page 1 and 91.)

[117] D.J. Reyniers. Information and rationality asymmetries in a simple high-low search wage model. *Economics Letters*, 38:479–486, 1992. (Cited on page 1 and 92.)

[118] R.L. Rivest, A.R. Meyer, D.J. Kleitman, and J. Spencer. Binary search using unreliable comparisons. *Proceedings of the 15th Annual Allertion Conference on Communication, Control and Computing, Sept. 28-30, 1977*, 1977. (Cited on page 55 and 56.)

[119] R.L. Rivest, A.R. Meyer, D.J. Kleitman, and K. Winklmann. Coping with errors in binary search procedures. *Journal of Computer and System Sciences*, 20:396–404, 1980. (Cited on page 56.)

[120] F. Ruskey and T.C. Hu. Generating binary trees lexicographically. *SIAM Journal on Computing*, 1977. (Cited on page 14.)

[121] S.H. Sheu, Y.C. Chen, W.Y. Wang, and N. Shin. Economic optimization of off-line inspection with inspection errors. *Journal of Operational Research Society*, 54:888–895, 2003. (Cited on page 57, 84, 85, and 86.)

[122] D. Topkis. Minimizing a submodular function on a lattice. *Operational research*, 26:305–321, 1978. (Cited on page 24.)

[123] W.C. Tsai and C.H. Wang. Economic optimization for an off-line inspection, disposition and rework model. *Computers and Industrial Ingineering*, 61:891–896, 2011. (Cited on page 83.)

[124] A. Tzimerman and Y.T. Herer. Off-line inspections under inspection errors. *IIE Transactions*, 41:626–641, 2009. (Cited on page 57 and 86.)

[125] M.L. Wachs. On an efficient dynamic programming technique of f.f.yao. *Journal of Algorithms*, 10:518–530, 1989. (Cited on page 23 and 43.)

[126] C. Wang. Economic off-line quality control strategy with two types of inspection errors. *European Journal of Operational Research*, 179:132–147, 2007. (Cited on page 57, 84, 85, and 86.)

[127] C.H. Wang, N.H. Shih, and W.C. Tsai. Utilizing the information theory of entropy to solve an off-line inspection problem. *4OR-Q Journal of Operational Research*, 9:391–401, 2011. (Cited on page 89.)

[128] W. Wang, S. Sheu, Y. Chen, , and D. Horng. Economic optimization of off-line inspection with rework consideration. *European Journal of Operational Research*, 194:807–813, 2009. (Cited on page 82 and 83.)

[129] R.L. Wessner. Optimal alphabetic search trees with restricted maximal height. *Information Processing Letters*, 4:90–94, 1976. (Cited on page 53.)

[130] E. Wong. A linear search problem. *SIAM Review*, 6:168–174, 1964. (Cited on page 1 and 25.)

[131] F.F. Yao. Efficient dynamic programming using quadrangle inequalities. *Proceedings, 12th ACM Symposium on Theory of Computing*, pages 429–435, 1980. (Cited on page 44.)

[132] R.W. Yeung. Alphabetic codes revisited. *IEEE Transactions on Information Theory*, 37:564–472, 1991. (Cited on page 18.)

[133] E. Zemel. On search over rationals. *Operations Research Lettters*, 1:34–38, 1981. (Cited on page 68.)

E-mail address, Anna Sarid: annashva@yahoo.com