Yet Harder Knapsack Problems[☆]

Stasys Jukna[*,1]

Georg Schnitger

University of Frankfurt, Institut of Computer Science, D-60054 Frankfurt, Germany.

Abstract

Already 30 years ago, Chvátal has shown that some instances of the zero-one knapsack problem cannot be solved in polynomial time using a particular type of branch-and-bound algorithms based on relaxations of linear programs together with some rudimentary cutting-plane arguments as bounding rules. We extend this result by proving an exponential lower bound in a more general class of branch-and-bound and dynamic programming algorithms which are allowed to use memoization and arbitrarily powerful bound rules to detect and remove subproblems leading to no optimal solution.

Key words: Branch and bound, dynamic programming, memoization, branching program, knapsack, perfect matching

1. Introduction

An n-dimensional zero-one maximization problem $\mathcal{P}_n$ is specified by a set A of possible data items, a target function $\Phi : A^n \times \{0,1\}^n \to \mathbb{R}$, and a constraint predicate $P : A^n \times \{0,1\}^n \to \{0,1\}$. Problem instances are strings $a \in A^n$ of data items. Some of them are declared as *valid* instances. A solution for an instance $a \in A^n$ is a zero-one vector $x \in \{0,1\}^n$ such that $P(a,x) = 1$. Given a valid instance a, the goal is to maximize $\Phi(a,x)$ over all solutions x for a. A solution x achieving this maximum is *optimal*.

In this paper we concentrate on the classical zero-one knapsack problem: given a set of n items, each item i having an integer profit a_i and an integer weight

[☆]Research of both authors supported by a DFG grant SCHN 503/5-1.

[*]Corresponding author

Email addresses: jukna@thi.informatik.uni-frankfurt.de (Stasys Jukna), georg@thi.informatik.uni-frankfurt.de (Georg Schnitger)

[1]Affiliated with Vilnius University, Institute of Mathematics, Vilnius, Lithuania.

c_i, the problem is to choose a subset of the items such that their overall profit is maximized, while the overall weight does not exceed a given capacity b:

$$\begin{aligned}&\text{maximize } \Phi(a,x) = \textstyle\sum_{i=1}^{n} a_i x_i \\ &\text{subject to } \textstyle\sum_{i=1}^{n} c_i x_i \le b, \quad x_i \in \{0,1\},\, i = 1,\dots,n\end{aligned} \tag{1}$$

where the binary decision variables x_i are used to indicate whether item i is included in the knapsack or not. We will consider the space complexity of algorithms for the knapsack problem using the combined powers of branch-and-bound, dynamic programming, and backtracking arguments; however, the full generality of polynomial-time algorithms is outside their scope.

Every zero-one optimization problem gives a class of boolean functions, one for each problem instance $a \in A^n$. Namely, say that a boolean function $f_a : \{0,1\}^n \to \{0,1\}$ is the *optimum-function* for a given instance a, if for every vector $x \in \{0,1\}^n$,

$$f_a(x) = 1 \text{ iff } x \text{ is an optimal solution for } a. \tag{2}$$

A general algorithmic paradigm, known as *branch-and-bound* algorithm, consists of a systematic enumeration of all candidate solutions, where large subsets of fruitless candidates are discarded, by using an appropriate *pruning* or *bounding* rules. The work of such an algorithm on a given instance $a = (a_1, \dots, a_n)$ of a zero-one optimization problem starts at the root node and iteratively constructs a branching tree (a BT) on variables $x_1, \dots, x_n$ corresponding to the decisions about the n items in a. At each node some item a_i is tested and the two outgoing edges are labeled by the two possible decisions $x_i = 0$ and $x_i = 1$. There is no restrictions on which item is tested at what node. Along each path p from the root some subsequence $a_p = (a_i \colon i \in I)$ of items in a is considered and a sequence $x_p = (x_i \colon i \in I)$ of decisions about them is made. The subtree rooted in the last node of p defines a subproblem of the original optimization problem consisting of all 0-1 extensions of x_p. The algorithm then tries to cut off or prune this subtree using some pruning heuristic. Pruning a path means to declare its last node a 0-leaf (no optimal solution possible). End-nodes of non-pruned paths are called 1-leafs; these paths correspond to optimal solutions. The complexity measure of the algorithm is the total number of nodes in the tree, that is, the number of produced subproblems.

Thus, the complexity of a branch-and-bound algorithm is just the minimum number of nodes in branching tree for the optimum-function $f_a(x)$. The more powerful path-pruning rules are allowed, the fewer nodes are necessary. Let us stress that we allow to use different decision trees for different problem instances $a \in A^n$.

1.1. Previous results

Most of path-pruning rules for the knapsack problem usually use some kind of linear programming (LP) relaxation: prune a path if no its *fractional* extension is better than the best 0-1 solution found so far. The simplest of these rules is, given a partial 0-1 solution $(x_i\colon i \in I)$ (a path), to solve the following LP:

$$\begin{aligned}&\text{maximize } \textstyle\sum_{i\notin I} a_i x_i \text{ subject to}\\ &\textstyle\sum_{i\notin I} c_i x_i \le b - \sum_{i\in I} c_i x_i,\ 0 \le x_i \le 1\ (i \notin I)\end{aligned} \tag{3}$$

The path $(x_i\colon i \in I)$ can then be pruned if this LP does not have a fractional solution at least as good as a 0-1 solution obtained so far.

That this pruning rule, called also *fathoming*, may be very inefficient was observed by Jaroslow [9]. To see this, consider the knapsack problem (1) with $b = n$ and $a_i = c_i = 2$ for all i. That is, the set of optimal zero-one solutions for this instance consists of all 0-1 vectors with exactly $\lfloor n/2 \rfloor$ ones. But if $|I| < n/2$, then (3) *has* a fractional solution, which is at least as good as an optimal solution, and hence, the path $(x_i\colon i \in I)$ cannot be pruned. Thus, under the path-pruning rule (3), any BT for this problem must have at least $2^{n/2}$ nodes.

Krishnamoorthy [13] used counting arguments to show that, for every instance $(a_1, \ldots, a_n, b)$ of (1) with $c_i = a_i$ and b being a number larger than all a_i and relatively prime to all a_i, every BT using the fathoming rule (3) to prune paths must have at least about $(b/a)^n/n!$ nodes, where $a = \max_i a_i$.

Chvátal in [5] considered BTs for (1) with more powerful path-pruning rules based on rudimentary cutting-plane argument. The first rule, called *domination*, allows to prune a path $p = (x_i\colon i \in I)$ if there is another path $q = (y_i\colon i \in I)$ along which the same variables (with indexes in I) are tested, and such that $\sum_{i\in I} c_i x_i \ge \sum_{i\in I} c_i y_i$ but $\sum_{i\in I} a_i x_i \le \sum_{i\in I} a_i y_i$. That is, a path p can be pruned if there is another path that has—considering only the fixed variables—at least as much slack in the weight constraint and at least as good an objective value.

Note that this rule alone reduces the BT size in Jaroslow's example from $2^{n/2}$ to $O(n^2)$. Indeed, if we test the variables in the same order $x_1, \ldots, x_n$ along all paths then, at the $n/2$-th level all $\binom{n/2}{k}$ but one path with exactly k $(k = 0, 1, \ldots, n/2)$ tests $x_i = 1$ can be pruned by the domination rule. That is, the resulting BT will contain only $n/2 + 1$ paths.

The second rule allowed by Chvátal is a strengthening of the fathoming rule with (3) replaced by

$$\sum_{i\notin I} (a_i/d) x_i \le \lfloor (b - \sum_{i\in I} a_i x_i)/d \rfloor \quad 0 \le x_i \le 1\ \ (i \notin I)\,,$$

where d is the greatest common divisor of the integers a_i with $i \notin I$, and $\lfloor \alpha \rfloor = \max\{m \in \mathbb{Z}\colon m \le \alpha\}$. The path $(x_i\colon i \in I)$ is then truncated if this LP does not have a fractional solution better than an optimal 0-1 solution (or a temporal solution obtained so far). Since the coefficients a_i are integers, this rule is also legal, that is, does not remove any 0-1 solutions.

Chvátal considers instances $a = (a_1, \ldots, a_n)$ of (1) with $b = \lfloor \sum_i a_i/2 \rfloor$, $c_i = a_i$ for all i, and the a_i being numbers between 1 and $10^{n/2}$. Using probabilistic arguments, he shows that, under his pruning rules, almost all such instances require BTs of size at least $2^{n/10}$. In the same paper, he shows that this lower bound also holds for the following *explicit* instance constructed by Michael Todd (as cited in [5]) with profits/weights a_i of items defined by: $a_i := 2^{k+n+1} + 2^{k+i} + 1$ where $k = \lfloor \log 2n \rfloor$.

Gu et al. [6] and Husaker and Tovey in [8] extended Chvátal's result to a more general class of branch-and-bound algorithms, where the bound rule is allowed to use (at no cost) all so-called "lifted cover inequalities" to detect whether a given path can have an optimal extension.

Chung et al. [4] considered general knapsack problem (1) where the x_i may be arbitrary non-negative integers, that is, where one is allowed to take multiple copies of the same item. They proved that in this case even instances with moderately large profits a_i and weights c_i are hard for a particular type of branch-and-bound algorithms.

In this paper we give an instance of the 0-1 knapsack problem which is hard for a more general class of branch-and-bound algorithms than those considered in [9, 5, 6, 8].

1.2. Path pruning rule

First, together with the domination rule of Chvátal, we allow the most powerful fathoming rule:

$$\text{prune a path iff it cannot be extended to an optimal 0-1 solution.} \qquad (4)$$

The "only if" part—if a path is pruned then it cannot be extended to an optimal 0-1 solution—holds for most used heuristics, including those based on linear relaxations.

Remark 1. This is quite reasonable requirement because a BT must not only contain a path to a 1-leaf giving an optimal solution – it must also provide a *proof* or *witness* that this solution is optimal. That is, it must make sure that paths which do not lead to 1-leaves cannot correspond to optimal solutions. Without this restriction the model would be too powerful: *every* instance of every zero-one optimization problem could then be solved by a BT of linear size! For this, it would be enough just to guess an optimal solution, take one path to a 1-leaf corresponding to any one optimal solution, and let all remaining edges go directly to 0-leaves.

What makes our heuristic powerful is the "if" part: a path can be pruned as soon as it lacks an optimal extension. That is, we assume that fathoming is made by a "superior being" able to detect the absence of optimal extensions at no cost.

A direct consequence of our path-pruning rule (4) is that if an instance of an n-dimensional zero-one optimization problem has K optimal solutions, then it has a BT of size $O(nK)$. To see this, just take a full binary tree of depth n and cut-off all paths that are not consistent with any optimal solution; this way only K paths will survive. This observation implies that a lot of instances $(a_1, \ldots, a_n)$ of the knapsack problem have very small BTs. In particular, this holds for all instances which are *sum-free* in that $\sum_{i \in I} a_i \neq \sum_{j \in J} a_j$ holds for all disjoint nonempty subsets I, J of $[n] = \{1, \ldots, n\}$. Each such instance can have at most one optimal zero-one solution, and hence, the instance can be solved by a BT of size $O(n)$. Since instances used by Chvátal in [5] are also sum-free, all these instances have small BTs under our pruning rule (4), as well.

Actually, as observed by Chvátal in [5], *almost all* instances are sum-free (and hence, have BTs of linear size) as long as the range for the coefficients a_i is large enough. To see this, consider the set of all instances in $[M]^n$. Then at least a $1 - 3^n/M$ fraction of all M^n such instances are sum-free. To show this, fix a pair I, J of disjoint nonempty subsets of $[n]$, and fix an element $k \in I$. The number of strings such that $\sum_{i \in I} a_i = \sum_{j \in J} a_j$ is at most M^{n-1}, because in such strings the element a_k is determined by the remaining elements: $a_k = \sum_{j \in J} a_j - \sum_{i \in I \setminus \{k\}} a_i$. The number of pairs I, J does not exceed 3^n: for each of the n elements, we decide whether it belongs to I, to J or to neither. Thus, the number of not sum-free instances does not exceed $3^n M^{n-1}$. In particular, if $M = 3^{n+1}$ then at least 2/3 fraction of all instances in $[M]^n$ have BTs of linear size.

1.3. Free branching programs

Our next generalization of the model is that, besides branch-and-bound, we allow another algorithmic paradigm known as *memoization*, one of the main aspects of dynamic programming. It allows to remember the solution to common subproblems for the later use. That is, it allows to merge isomorphic subtrees of a BT. Both these paradigms – branch-and-bound and memoization – are captured by the following model of "free branching programs."

As mentioned above, the work of a branch-and-bound algorithm on a given problem instance $a \in A^n$ can be presented as branching tree at each node of which a decision $x_i = 0$ or $x_i = 1$ about some data item a_i is made. To incorporate *memoization* we allow some subtrees to be merged. We also allow to re-consider previously made decisions (test the same variable many times), as well as to behave in a *nondeterministic* manner. All this is captured by a classical model of branching programs (see, e.g., [16] for a comprehensive survey on this model).

A *nondeterministic branching program* (NBP) on boolean variables $x_1, \ldots, x_n$ is a directed acyclic graph with one source node, at some of whose edges some tests $x_i = 0$ or $x_i = 1$ are made. Each leaf (a node of zero fanout) is labeled either by "1" (optimal solution) or by "0" (no optimal solution possible). Such a program solves a zero-one optimization problem for a given problem instance $a \in A^n$ if a 0-1 vector x is an optimal solution for a if and only if x is consistent with all tests made along at least one path from the source node to a 1-leaf. (Edges at which no test is made are consistent with all vectors x.) The complexity measure is again the number of nodes in the underlying graph.

So as it is, the model of NBP is too powerful, much more powerful than, say, branching trees considered in [9, 5, 6, 8], and even more powerful than *any* nondeterministic Turing machine working with logarithmic memory. More adequate in the context of the branch-and-bound paradigm is the model of *free* NBP, where every path to a 1-leaf is required to be consistent, that is, do not contain two contradictory tests $x_i = 0$ and $x_i = 1$ on the same variable. This implies that every path from the source node, along which a contradictory test is made for the first time, must immediately go to the 0-leaf.

This "null-path freeness" is a severe restriction, but it can be justified as follows. After the algorithm has constructed a branching program (a "table" of partial solutions), it must be able to quickly reconstruct an optimal solution without probing all (exponentially many) possible paths leading to a 1-leaf. Instead, it should be possible to start at any 1-leaf, traverse backwards any one path until the source node is reached; the sequence of decisions along this path should give an optimal solution. A similar justification also applies to the model of so-called "priority BP" introduced in [3] and capturing the power of backtracking and simple dynamic programming algorithms.

As such, even the model of free NBP is much more powerful than decision trees constructed by branch-and-bound algorithms. So, we "granulate" the model by introducing additional restrictions:

1. An NBP is *read-once* (shortly, 1NBP) if along each path to a 1-leaf every variable is tested at most once.
2. An *deterministic branching program* (shortly, BP) is an NBP with a restriction that every inner node must have fanout exactly 2, and the two outgoing edges must be labeled by the tests $x_i = 0$ and $x_i = 1$ on the *same* variable x_i. Such a program is deterministic because for every vector $x \in \{0, 1\}^n$ there is only one path to a leaf.

Hence, branching trees (BT) are read-once BP (1BP) with an additional restriction that the underlying graph must be a tree. On the other hand, 1BP can be looked at as a "BT with memoization" where isomorphic subtrees (those corresponding to the same subproblem) can be merged. We have the following relations between these models (where $A \subset B$ means that, for some instances, model A is exponentially weaker than model B):

$$\text{BT} \subset \text{1BP} \subset \text{free BP} \subseteq \text{free NBP} \qquad \text{and} \qquad \text{1BP} \subset \text{1NBP}\,.$$

Separations 1BP $\subset$ free BP and 1BP $\subset$ 1NBP were shown in [12] using so-called pointer functions.

Just like standard branching programs for boolean functions (or languages) capture the space complexity of Turing machines for that function, the size of a BP for a particular instance $a \in A^n$ captures the number of partial solutions that a branch-and-bound algorithm must maintain during the execution on a. In this sense, the logarithm of the BP size is a lower bound on the amount of memory required by any branch-and-bound algorithm for that instance.

We stress that we are looking for a smallest branching tree or branching program for *one given instance* $a = (a_1, \ldots, a_n)$ of a zero-one optimization problem: given an instance a, we are looking for the size of a smallest branching tree or free branching program computing its optimum-function (2). That is, we consider the classical model of branching programs but restrict ourselves to special boolean functions corresponding to zero-one optimization problems.

We have already mentioned that if an instance of a zero-one optimization problem has only few optimal solutions, then it has a small BT. But a large number of optimal solutions alone does not imply that the NBP for that instance must be large. For example, Jaroslow's instance a of the knapsack problem mentioned in Section 1.1 has $2^{\Omega(n)}$ optimal solutions, but the optimum function $f_a(x)$ for this instance is very simple: $f_a(x) = 1$ if and only if $\sum_{i=1}^n x_i = \lfloor n/2 \rfloor$. Hence, this instance has a 1BP with $O(n^2)$ nodes.

1.4. Our result: a hard knapsack problem

To define an explicit instance requiring free NBPs of exponential size even under the heuristic (4), we use a q-ary encoding of integers. A *q-ary code* ($q \geq 2$) of a non-negative integer r is a string $(c_1, \ldots, c_m)$ of integers $c_i \in \{0, 1, \ldots, q-1\}$ such that $r = c_1 q^0 + c_2 q^1 + \cdots + c_m q^{m-1}$. We will use the trivial fact that every integer $r \leq (q^m - 1)/(q-1)$ has a *unique* q-ary code. We will consider the case when $m = 2n$ and

$$q := n + 1\,.$$

To describe a hard instance for the zero-one knapsack problem (1), we index the items by edges $(i, j) \in U \times V$ of the complete bipartite graph $K_{n,n} = U \times V$ with $U = \{1, \ldots, n\}$ and $V = \{n+1, \ldots, 2n\}$. As in Chvátal's paper [5] we consider a restricted version of the knapsack problem where profit of each item is equal to its weight. The weight of an item (edge) (i, j) is defined by:

$$a_{ij} := q^{i-1} + q^{j-1}\,. \tag{5}$$

Every zero-one vector $x = (x_{ij} \colon i \in U, j \in V)$ defines a bipartite subgraph $E_x := \{(i,j) \colon x_{ij} = 1\}$ of $K_{n,n}$ corresponding to the 1-positions of x, and the weight of this subgraph is

$$w(x) := \sum_{i \in U} \sum_{j \in V} a_{ij} x_{ij} \,.$$

Note that the weight a_{ij} of every edge (i,j) is an integer whose q-ary code is a binary vector of length $2n$ with exactly two ones in the i-th and $(n+j)$-th positions. As the capacity of the knapsack we will take the number b whose q-ary code is the vector $(1,1,\ldots,1)$ of length $2n$, that is, we define

$$b := q^0 + q^1 + \cdots + q^{2n-1} = \frac{q^{2n}-1}{q-1} \,. \tag{6}$$

Given a capacity t, consider the knapsack problem KNAP(a,t): maximize $w(x)$ subject to $w(x) \leq t$, x binary. Our first result is that, for $t = b$, no small free NBP can solve the problem KNAP(a,t).

Theorem 2. *Every free NBP for the zero-one knapsack problem* KNAP(a,b) *requires at least* $\binom{n}{n/2}$ *nodes.*

Our second result is that, for $t = rb$, no small read-once NBP can even approximate the problem KNAP(a,t) within a factor r.

Theorem 3. *For every* $1 \leq r \leq n$*, every 1NBP approximating the zero-one knapsack problem* KNAP(a,br) *with a factor of* r *requires at least* $\binom{n}{n/2}$ *nodes.*

In the proof of these theorems we will use one special property of the knapsack problem defined by the weights (5) which may be of some independent interest (see Lemma 5 below): a 0-1 vector x is an optimal solution for KNAP(a,b) if and only if the graph E_x forms a perfect matching, that is, consists of n vertex disjoint edges. The "if" direction is here trivial—more interesting is the "only if" direction.

Weights (5) to the basis $q = 2$ were already considered in [7] to show that the threshold function $T_n(x) = 1$ iff $\sum_{i,j} a_{ij} x_{ij} \geq b$ requires oblivious 1BP of exponential size. This was extended to arbitrary (non-oblivious) 1BPs in [1]. To extend this further to the more powerful models of free BP and free NBP we need an *exact* correspondence between optimal solutions and perfect matchings. We achieve this by taking larger basis $q = n + 1$.

Note that Theorem 3 does not state that the knapsack problem *is* hard to approximate: there is a simple branch-and-bound algorithm approximating this problem with the factor 2. The algorithm either accepts or rejects the highest profit item, and then greedily chooses items when ordered by their decreasing profit to weight ratio. But this algorithm accepts just one of two possible approximative solutions, whereas we require that none of approximative solutions can be rejected. That is, we (as well as authors of the papers cited in Section 1.1) require that the algorithm must also provide a *proof* or *witness* that a solution it finds is approximative (see Remark 1 above).

The proofs of both theorems are elementary: they just use standard arguments of circuit complexity. Our contribution is an application of these arguments to show the limitations of particular type of algorithmic paradigms.

2. Knapsack and perfect matchings

Our first goal is to relate optimal solutions for the instance of the knapsack problem $\mathrm{KNAP}(a,b)$ defined above with perfect matchings in a bipartite graph.

Lemma 4. *The q-ary code of the weight $w(x)$ is $(d_1, d_2, \ldots, d_{2n})$, where d_i is the degree of the i-th vertex in E_x.*

Proof. Since $0 \le d_i \le n < q$ for every vertex $i \in U \cup V$, the lemma follows by direct computation:

$$w(x) = \sum_{i\in U} d_i q^{i-1} + \sum_{j\in V} d_j q^{j-1} = \sum_{i=1}^{2n} d_i q^{i-1}\,. \quad \square$$

Thus, solutions of $\mathrm{KNAP}(a,b)$ correspond to subgraphs of $K_{n,n}$ of weight at most b. As a direct consequence of Lemma 4 we obtain the following graph-theoretic characterization of the optimal solutions for $\mathrm{KNAP}(a,b)$. A subgraph $E \subseteq K_{n,n}$ is a *perfect matching* if it consist of n vertex-disjoint edges.

Lemma 5. *A 0-1 vector x is an optimal solution for* $\mathrm{KNAP}(a,b)$ *if and only if E_x is a perfect matching.*

Proof. A vector x is an optimal solution for $\mathrm{KNAP}(a,b)$ if and only if $w(x) = b$. By Lemma 4, we know that the q-ary code of $w(x)$ is the sequence $(d_1, \ldots, d_{2n})$ of degrees of vertices in the graph E_x defined by x. On the other hand, since $b = q^0 + q^1 + \cdots + q^{2n-1}$, the q-ary code of b is the all-1 vector $(1, \ldots, 1)$. Hence, x is an optimal solution if and only if $d_i = 1$ for all i, that is, iff E_x is a perfect matching. $\square$

3. Proof of Theorem 2

The following theorem gives a general lower bound on the size of free NBPs. The theorem itself is an extension of a similar lower bound for read-once NBP proved in [11].

Let $S \subseteq \{0,1\}^n$ be a set of vectors. Let also m be the minimum number of 1s in a vector of S. For a subset of positions $I \subseteq [n] = \{1, \ldots, n\}$, let $d_I(S)$ denote the number of vectors in S having ones in all these positions:

$$d_I(S) = \left|\{x \in S \colon x_i = 1 \text{ for all } i \in I\}\right| .$$

If $I = \emptyset$ then we set $d_I(S) = |S|$. Define $d_k(S)$ as the maximum of $d_I(S) \cdot d_J(S)$ over all subsets I of size $|I| = k$ and all subsets $J \subseteq [n] \setminus I$ of size $|J| = m - k$. Hence, $d_k(S)$ is the maximum size $|T|$ of a subset of vector $T \subseteq S$ for which there exits a pair I, J of disjoint subsets of position such that $|I| = k$, $|J| = m - k$ and every vector $x \in T$ has 1s either in all positions of I or in all positions of J (or of both). Finally, let $d(S)$ be the minimum of $d_k(S)$ over all $1 \leq k \leq m$.

Let M be the maximum number of 1s in a vector of S. Say that a boolean function $f : \{0,1\}^n \to \{0,1\}$ *isolates* S if, for every vector x with at most $2M$ ones, we have that $f(x) = 1$ iff $x \in S$.

Theorem 6 (Criterion for free NBP)**.** *If a free NBP isolates a set $S \subseteq \{0,1\}^n$ then it must have at least $|S|/d(S)$ nodes.*

Proof. Take a free NBP G isolating S, and let $1 \leq k \leq m$ an integer for which $d(S) = d_k(S)$. Since each vector $x \in S$ must be accepted by G, there must be a path accepting this vector x, that is a path consistent with x and ending in a 1-leaf.

For each vector $x \in S$, fix a path accepting x, and split this path into two segments (p_x, q_x), where p_x is an initial segment along which exactly k 1-bits of x are tested. Let I_x denote the set of these 1-bits of x, and let J_x denote the set of 1-bits of x corresponding to the tests $x_i = 1$ made along q_x. Note that the sets I_x and J_x need not to be disjoint: we only know that $|I_x| = k$, $m \leq |I_x \cup J_x| \leq M$ and $x_i = 1$ iff $i \in I_x \cup J_x$.

For a node v of our program G, let S_v denote the set of all vectors $x \in S$ such that v is the end-node of the path p_x. It is enough to prove that $|S_v| \leq d_k(S)$. Let $\mathcal{I} = \{I_x : x \in S_v\}$ and $\mathcal{J} = \{J_x : x \in S_v\}$. For each pair $I \in \mathcal{I}$ and $J \in \mathcal{J}$ consider the combined vector $z_{I,J}$ defined by $z_{I,J}(i) = 1$ iff $i \in I \cup J$.

Claim 7. *For every $I \in \mathcal{I}$ and $J \in \mathcal{J}$ the combined vector $z_{I,J}$ belongs to S.*

Proof. Choose some $x, y \in S_v$ such that $I = I_x$ and $J = J_y$. The combined path (p_x, q_y) goes from the source node to the node v and then follows q_y until a 1-leaf. Since our program is null-path free, the path (p_x, q_y) must be consistent. Moreover, this path is consistent with the combined vector $z = z_{I,J}$. To show this, take an arbitrary bit i. If $z(i) = 1$ then the test $x_i = 1$ is made along at least one of the two paths p_x and q_y, and hence, the test $x_i = 0$ cannot occur in the other one, since the entire path (p_x, q_y) must be consistent. If $z(i) = 0$ then $i \notin I_x \cup J_y$, implying that the test $x_i = 1$ cannot be made along any of the two paths p_x and q_y, by the definition of the sets I_x and J_y. Hence, the vector z is consistent with all tests along the path (p_x, q_y). Since this path ends in a 1-leaf, the vector z is accepted by G. But since this vector has only $|I_x \cup J_y| \leq k + M \leq 2M$ ones and since our program isolates the set S, the program can accept z only if $z \in S$, as desired. □

To finish the proof of the theorem, fix an arbitrary $I_0 \in \mathcal{I}$. Then all vectors $z_{I_0,J}$ with $J \in \mathcal{J}$ have 1s on I_0 and, by Claim 7, all of them belong to S. This implies that $|\mathcal{J}| \leq d_{I_0}(S)$. Similarly, fix an arbitrary $J \in \mathcal{I}$ and an arbitrary its subset $J_0 \subseteq J \setminus I_0$ of size $|J_0| = m - k$ (we can do this since $|I_0| = k$ and $|I_0 \cup J| \geq m$). Then all vectors $z_{I,J}$ with $I \in \mathcal{I}$ have 1s on J, and hence, also on J_0. By Claim 7, all of them belong to S. This implies that $|\mathcal{I}| \leq d_J(S) \leq d_{J_0}(S)$. Finally, every $x \in S_v$ is uniquely determined by the pair (I_x, J_x), therefore $|S_v| \leq |\mathcal{I}| \cdot |\mathcal{J}|$, as claimed. □

Proof of Theorem 2. Let $S \subseteq \{0,1\}^{n^2}$ be the set of all optimal 0-1 solutions of KNAP(a,b). By Lemma 5, we know that $x \in S$ iff the graph E_x is a perfect matching. Thus, a free NBP for the knapsack problem KNAP(a,b) accepts a vector x if and only if $x \in S$. In particular, any such program must isolate the set of optimal solutions S. Since only $(n-k)!$ perfect matching can contain a fixed set of k edges, we have that $d_k(S) \leq k!(n-k)!$. Thus, Theorem 6 implies that every free NBP solving the problem KNAP(a,b) must have at least $|S|/d(S) \geq n!/k!(n-k)! = \binom{n}{k}$ nodes. Taking $k = n/2$ gives the desired lower bound. □

4. Proof of Theorem 3

Consider the knapsack problem KNAP(a, br). Optimal solutions for this problem are all vectors $x \in \{0,1\}^{n^2}$ of weight $w(x) = br$. Let G be a 1NBP approximating KNAP(a, br) within the factor r. Then G accepts a vector $x \in \{0,1\}^{n^2}$ if and only if $b \leq w(x) \leq br$. In particular, we have that

$$G(x) = 1 \text{ if } w(x) = b, \text{ and } G(x) = 0 \text{ if } w(x) < b. \tag{7}$$

Fix an integer k, $1 \leq k \leq n$, and set $S = \{x \colon w(x) = b\}$. For each vector $x \in S$, fix a path accepting x, and split this path into two segments (p_x, q_x), where p_x is an initial segment along which exactly k 1-bits of x are tested. Let I_x denote the set of all bits (not just of 1-bits) of x tested along p_x, and J_x the set of all bits of x tested along q_x. Bits in this case correspond to edges of $K_{n,n}$. By Lemma 5, we know that $x \in S$ iff the graph $E_x = \{e \colon x_e = 1\}$ is a perfect matching. This way the n edges of the perfect matching E_x are divided into two parts: the matching

$M_x := E_x \cap I_x$ with k edges, and the matching $E_x \cap J_x$ with the remaining $n-k$ edges. Finally, if we define the weight of an edge $e = (i,j)$ by $w(e) = q^{i-1} + q^{j-1}$, then the weight $w(x)$ of every vector $x \in \{0,1\}^{n^2}$ is exactly the total weight of the edges in the corresponding graph E_x:

$$w(E_x) = \sum_{e \in E_x} w(e)\,.$$

For a graph $E \subseteq K_{n,n}$, let $V(E)$ denote the set of vertices touched by at least one edge of E. Hence, $|V(I_x)| = 2k$ for every $x \in S$. For a node v of our program G, let S_v denote the set of all vectors $x \in S$ such that v is the end-node of the path p_x.

Claim 8. *For any two vectors $x, y \in S_v$ we have $V(I_x) = V(I_y)$.*

Proof. Assume that $V(I_y) \neq V(I_x)$ for some two vectors $x, y \in S_v$. Since $I_x \cap J_x = \emptyset$, and since the weight of a graph is defined as the sum of weight of its edges, for every vector $x \in S_v$ we have that $w(x) = w(I_x) + w(J_x)$. Moreover, since both $I_x \cap E_x$ and $I_y \cap E_y$ are matchings, Lemma 4 implies that the q-ary codes of $w(I_x)$ and $w(I_y)$ have only coefficients 0 and 1, and $V(I_x)$ is exactly the set of all 1-positions in the q-ary code of $w(I_x)$. Thus, $V(I_x) \neq V(I_y)$ implies that $w(I_x) \neq w(I_y)$. Assume w.l.o.g. that $w(I_y) < w(I_x)$, and consider the combined vector z such that $z_e = 1$ iff e belongs to at least one of the matchings $I_y \cap E_y$ and $J_x \cap E_x$. Since the program is read-once, these matchings must be disjoint. Hence,

$$w(z) = w(I_y) + w(J_x) < w(I_x) + w(J_x) = w(x) = b\,.$$

But the vector z is consistent with the combined path (p_y, q_x), and hence, is accepted by our program, contradicting (7). $\square$

Since only $k!(n-k)!$ perfect matchings can match a given set of $2k$ vertices, Claim 8 implies that $|S_v| \leq k!(n-k)!$. Hence, our program must have at least $|S|/k!(n-k)! = n!/k!(n-k)! = \binom{n}{k}$ nodes. Taking $k = n/2$ yields the desired lower bound. $\square$

5. Concluding remarks

In this paper we consider the space complexity of branch-and-bound and dynamic programming algorithms with memoization. We look at an instance $a \in A^n$ of a zero-one optimization problem as a boolean function f_a which accepts a 0-1 vector x iff x is an optimal solution for a. The space complexity of a is then measured as the minimum size of a branching program computing f_a. Such a branching program is a compact encoding of the set of partial solutions produced by an algorithm when working on instance a. Since, as in dynamic programming, the algorithm must be able to reconstruct an optimal solution going backwards from any "optimum" leaf, the branching program cannot have inconsistent paths to such leafs. This leads us to a subclass of all branching programs—the class of free NBP.

We exhibited an instance a of the n^2-dimensional zero-one knapsack problem such that its boolean function f_a requires free NBPs of size $2^{\Omega(n)}$. The main structural property of this instance a is that optimal zero-one solutions for it are exactly the characteristic vectors of perfect matchings. We have then shown that any 1NBP (read-once NBP) approximating the value of optimal solutions for this instance within a factor of n must also be of exponential size. It would be interesting to extend these bound to the model of free NBP.

An interesting problem remains whether exponential lower bounds for the knapsack problem can be proved in more general models of branching programs than those considered above. The read-once condition in a 1NBP is a "syntactic" one: every path to a 1-leaf must be a read-once path, that is, every variable along such path can be tested at most once. One could ask what happens if we relax this condition to a *semantic* one: every path to a 1-leaf is either inconsistent or is a read-once path. Let us call such programs *semantic* 1NBP. The following proposition shows that such a seemingly "innocent" relaxation may exponentially increase the power of BPs.

Proposition 9. *The zero-one knapsack problem* KNAP(a, b) *has a semantic 1NBP of size $O(n^3)$.*

Proof. The proof is a combination of Lemma 5 with an observation made in [10] that the so-called "exact perfect matching function" has small semantic 1NBP. By Lemma 5, a 0-1 vector $x = (x_{ij})$ in $\{0,1\}^{n^2}$ is an optimal solution for KNAP(a, b) iff its 1-positions form a perfect matching. Thus, if looked at as an $n \times n$ matrix, the vector x is an optimal solution iff x is a permutation matrix (has exactly one 1 in each row and each column). To test that a given square $(0,1)$ matrix is a permutation matrix, it is enough to test whether every row has at least one 1, and every column has at least $n-1$ zeroes. These two tests can be made by two NBPs G_1 and G_2 designed using the formulas

$$G_1(X) = \bigwedge_{i=1}^{n} \bigvee_{j=1}^{n} x_{ij} \quad \text{and} \quad G_2(X) = \bigwedge_{j=1}^{n} \bigvee_{k=1}^{n} \bigwedge_{\substack{i=1 \\ i \neq k}}^{n} \neg x_{ij}\,.$$

Let $G = G_1 \wedge G_2$ be the AND of these two programs, that is, the 1-leaf of G_1 is the source-node of G_2. The entire program has size $O(n^3)$. It is also semantically read-once because in G_1 only tests $x_{ij} = 1$ and in G_2 only tests $x_{ij} = 0$ are made; so every path in the whole program P is either inconsistent or is read-once. $\square$

So far, no explicit boolean function requiring semantical 1NBPs of exponential size is known. In particular, it is not known whether some instances of the zero-one knapsack problem require semantical 1NBPs of exponential size.

It would be also interesting to prove that other natural zero-one optimization problems require large free NBPs. For example, the maximum clique problem for a given graph $G = (V, E)$ can be formulated as the following integer linear program: maximize $\sum_{v \in V} x_v$ subject to $x_u + x_v \leq 1$ for all $\{u, v\} \notin E$, and $x_v \in \{0, 1\}$ for all $v \in V$. Hence, a zero-one vector x is an optimal solution for this problem iff $V_x = \{v \in V \colon x_v = 1\}$ is a clique of size $|V_x| = \omega(G)$, where $\omega(G)$ is the maximum number of vertices in a clique of G. The optimum-function $f_G(x)$ of a given graph in this case is

$$f_G(x) = 1 \text{ iff } V_x \text{ is a clique in } G \text{ and } \textstyle\sum_{v \in V} x_v = \omega(G).$$

It is known [14] that the number of maximum cliques in an n-vertex graph does not exceed $3^{n/3}$. This bound is achieved by so-called Moon–Moser graphs: these are complements of graphs consisting of $n/3$ vertex disjoint triangles. It is however easy to see that the maximum clique problem for these "rich" graphs has a 1BP of size $O(n)$: connect sequentially $n/3$ programs, each computing 1 iff exactly one of the three variables in the corresponding triangle are set to 1.

Note a big difference between f_G and a classical clique function $\mathrm{CLIQUE}_{n,k}$. This function has $\binom{n}{2}$ boolean variables, each variable x_e corresponding to a potential edge e. This way every zero-one vector $x \in \{0, 1\}^{\binom{n}{2}}$ defines a graph G_x on n vertices. Then $\mathrm{CLIQUE}_{n,k}(x) = 1$ iff G_x has a clique of size k. Thus, $\mathrm{CLIQUE}_{n,k}$ describes a property of *all* graphs, whereas the optimum-function f_G describes a property of *one* single graph. In this sense, the first function seems to be "harder" than the second one. And indeed, it is known [2] that, for $k = n/3$, the function $\mathrm{CLIQUE}_{n,k}$ requires 1NBP of exponential size. It would be interesting to find graphs G whose optimum-function f_G requires free NBPs (or at least 1NBP) of exponential size.

References

[1] B. Bollig, On the size of (generalized) OBDDs for threshold functions, Inf. Process. Lett. 109:10 (2009) 499–503.

[2] A. Borodin, A. Razborov and R. Smolensky, On lower bounds for read-k times branching programs, Comput. Complexity 3 (1993) 1–18.

[3] J. Buresh-Oppenheim, S. Davis, R. Impagliazzo, A stronger model of dynamic programming algorithms, Algorithmica 60(4) (2011), 938–968.

[4] C. S. Chung, M. S. Hung, W. O. Rom, A hard knapsack problem, Naval Res. Logistics, 35 (1988), 85–98.

[5] V. Chvátal,, Hard knapsack problems, Operation Research 28:6 (1980) 1402–1411.

[6] Z. Gu, G. L. Nemhauser, M. W. P. Savelsbergh, Lifted cover inequalities for 0-1 linear programs: Complexity, INFORMS J. on Computing 11 (1999) 117–123.

[7] K. Hosaka, Y. Takenaga, T. Kaneda, S. Yajima, Size of ordered binary decision diagrams representing threshold functions, Theor. Comput. Sci. 180(1-2) (1997) 47–60.

[8] B. Hunsaker, C. A. Tovey, Simple lifted cover inequalities and hard knapsack problems, Discrete Optimization 2(3) (2005) 219–228.

[9] R. G. Jaroslow, Trivial integer programs unsolvable by branch-and-bound, Math. Programming 6 (1974) 105–109.

[10] S. Jukna, A note on read-k times branching programs, RAIRO Theoret. Informatics and Appl. 29(1) (1995) 75–83.

[11] S. Jukna, A. Razborov, Neither reading few bits twice nor reading illegally helps much, Discrete Appl. Math. 85(3) (1998) 223–238.

[12] S. Jukna, A. Razborov, P. Savický, I. Wegener, On P versus $NP \cap \text{co-}NP$ for decision trees and read-once branching programs, Computational Complexity 8(4) (1999) 357–370.

[13] B. Krishnamoorthy, Bounds on the size of branch-and-bound proofs for integer knapsacks, Operation Res. Lett. 36(1) (2008) 19–25.

[14] J. W. Moon and L. Moser, On cliques in graphs, Israel J. of Math. 3 (1965) 23–28.

[15] A. A. Razborov, Lower bounds on monotone network complexity of the logical permanent, Math. Notes Acad. of Sci. USSR, 37(6) (1985), 485–493.

[16] I. Wegener, Branching programs and binary decision diagrams, SIAM Monographs on Discrete Mathematics and Applications 4, 2000.