Electronic Colloquium on Computational Complexity, Revision 1 of Report No. 41 (2012)

Limitations of Incremental Dynamic Programming

Stasys Jukna

Abstract We consider so-called "incremental" dynamic programming algorithms, and are interested in the number of subproblems produced by them. The classical dynamic programming algorithm for the n-dimensional Knapsack problem is incremental, produces nK subproblems and nK^2 relations (wires) between the subproblems, where K is the capacity of the knapsack. We show that any incremental algorithm for this problem must produce about nK subproblems, and that about $nK \log K$ wires (relations between subproblems) are necessary. We also give upper and lower bounds on the number of subproblems needed to approximate the knapsack problem.

Keywords Dynamic programming, Knapsack, branching programs, lower bounds

1 Introduction

Capturing the power and weakness of algorithmic paradigms is an important task pursuit over several last decades. The problem is a mix of two somewhat contradicting goals. The first of them is to find an appropriate mathematical model formalizing vague terms, as greedy algorithms, dynamic programming, backtracking, branch-and-bound algorithms, etc. The models must be expressive enough by being able to simulate at least known algorithms. But they should also be "reasonable" enough to avoid the power of arbitrary algorithms, to avoid problems as **P** versus **NP**, as well as the power of general boolean circuits.

Having found a formal model for an algorithmic paradigm, the ultimate goal is to prove *lower bounds* in them. If one succeeds in doing this, we have a provable limitation of a particular algorithmic paradigm. If one fails to prove a strong lower bound, matching an upper bound given by known algorithms, this is a strong motivation to search for more

Research supported by the DFG grant SCHN 503/5-1.

Universität Frankfurt, Institut für Informatik, Robert-Mayer Str. 11-15, Frankfurt am Main, Germany · Vilnius University, Institute of Mathematics, Akademijos 4, Vilnius, Lithuania
E-mail: jukna@thi.informatik.uni-frankfurt.de

ISSN 1433-8092

efficient algorithms. Note that we are seeking for *unconditional* lower bounds that are independent of any unproven assumptions, as the assumption that $\mathbf{P} \neq \mathbf{NP}$.

The problem of proving absolute lower bounds for particular algorithmic paradigms is important by at least two reasons: they show the limits of these paradigms, and the proofs of these lower bounds localize the weak points of the paradigms, which can lead to better heuristics.

In this paper we focus on the dynamic programming paradigm. There were many attempts to formalize this paradigm, and various refinements were obtained [6,20,25,15, 14,27], just to mention some of them. In all these attempts, the goal is to include more and more algorithmic features to cover more and more existing DP algorithms. But, as a rule, the resulting models are too powerful to prove strong lower bounds in them.

More tractable models of so-called "prioritized branching trees" (pBT) and "prioritized branching programs" (pBP) were recently introduced, respectively, by Alekhnovich et al. [3] and Buresh-Oppenheim et al. [11]. These models are based on the framework of "priority algorithms" introduced by Borodin et al. [9], and subsequently studied and generalized in [4,3,10,12,24]; this framework aims to capture the power of *greedy* algorithms. The models of pBT and pBP extend the power of greedy algorithms by adding a some aspects of *backtracking* and *dynamic programming*. The model of pBP adds to that of pBT an additional feature of *memoization*. In [3] it is shown that the Knapsack problem with capacity K about $n3^n$ requires pBTs of size $\binom{n/2}{n/4}$, whereas in [11] it is shown that detecting the presence of a perfect matching in bipartite graphs requires (restricted) pBPs of size $\exp(\Omega(n^{1/8}))$.

In this paper we pursue essentially the same goal as in [3] and [11], but in a different, more "conservative" direction. There is an old field—that of boolean circuit complexity—where the same goal (what small circuits cannot do) was pursuit for now more than 70 years. Along the way, many subtle lower-bounds arguments were invented there. So, it makes sense to look what could be useful from all this classical "tool-box" when analyzing the limitations of DP algorithms. A possible approach here is to modify boolean circuits so that they are able to simulate DP algorithms.

Our model

Just as in the case of prioritized BPs, our starting point is the fact that every DP algorithm implicitly constructs a *subproblem graph* (or a "table of partial solutions", as we know from algorithms courses). This graph has a directed wire from the node for subproblem u to the node for subproblem v if determining an optimal solution for subproblem v involves directly considering an optimal solution for subproblem u. Our main observation is that for some DP algorithms, these subproblem graphs "work" in a similar manner as classical branching programs for decision problems do—we only need to replace the underlying boolean semiring by other semirings. For more information about branching programs, the reader may consult, for example, the books [26,19].

This leads us to the model of "dynamic branching programs" (dynamic BP) that are able to simulate so-called "incremental" DP algorithms. Our hope is that such a more direct relation to the classical model of branching programs could help us to use lower-bound methods developed for this latter model. Proofs given in this paper support this hope: we apply lower bound arguments already invented for the boolean model.

Meaning of "incremental"

Let us explain what we mean under an "incremental" DP algorithm. In DP algorithms for optimization problems one usually uses max and plus (or min and plus) operations to produce the value $f(v)$ of a subproblem v from the values computed at the subproblems $u_1, \ldots, u_k$ having direct wires to v in the subproblem graph. We call a DP algorithm *incremental* if its subproblem graph has the following two properties. First, each wire $u_j \to v$ is responsible for at most one data item x_i in the given problem instance $x = (x_1, \ldots, x_n)$. Second, the transition function has the form

$$f(v) = \max\{f(u_1) + w_1, \ldots, f(u_k) + w_k\}$$

where w_i is the weight of the data item which the wire (u_i, v) is responsible for, if this item was accepted at that wire; if the item was rejected at the wire, then $w_i = 0$. We thus use the term "incremental" to stress that the value of a partial solution is incrementally (not globally) modified.

In a non-incremental algorithm, the transition function may have a more general form

$$f(v) = \max\Big\{\sum_{i\in I_1} f(u_i), \ldots, \sum_{i\in I_m} f(u_i)\Big\}$$

for some subsets $I_1, \ldots, I_m \subseteq \{1, \ldots, k\}$. Thus, incremental DP algorithms constitute a subclass of all DP algorithms where the usage of $+$-operation is restricted: one of the two inputs must be the weight of a data item, not the value of an another subproblem.

By the *size* of a DP algorithm we will mean the number of subproblems it produces, that is, the number of nodes in its subproblem graph. Given an optimization problem, a natural question is: what is the smallest possible size of a DP algorithm solving this problem? In this paper we prove almost matching lower bounds on the size of incremental DP algorithms solving the Knapsack problem.

The Knapsack problem

In this paper we concentrate on the Knapsack problem. In the n-dimensional Knapsack problem with an integer knapsack capacity K, a problem instance is a sequence of n pairs $a_i = (p_i, s_i)$ of natural numbers; p_i is the "profit", and s_i the "size" of the i-th item. We will assume that $s_i \leq K$ for all i. The goal is to pack items into the knapsack so that the total size does not exceed the capacity of the knapsack, and the total profit is as large as possible: maximize $\sum_{j\in S} p_j$ subject to $\sum_{j\in S} s_j \leq K$.

A standard DP algorithm for this problem is to define the subproblems by: $S(i,j) =$ the maximal total profit for filling a capacity j knapsack with some subset of items $1, \ldots, i$. The DP algorithm is then described by the recursion:

$$S(i,j) = \text{maximum of } S(i-1,j) \text{ and } S(i-1,j-s_i) + p_i \text{ for } s_i \leq j.$$

The value of the optimal solution is $\mathrm{opt}(a) = S(n, K)$. Note that this algorithm is incremental: when going from subproblem $S(i-1, j-s_i)$ to $S(i, j)$, the value is increased by just adding the profit p_i. The number of subproblems produced by this algorithm, and hence, the size of this algorithm is nK. The algorithm is "read-once" (along every branch of the recursion, every item is queried only once), and is "oblivious" (along all branches the items are queried in the same order).

In the *minimization* Knapsack problem we want to minimize the total size by keeping the total profit over some threshold K: minimize $\sum_{j \in S} s_j$ subject to $\sum_{j \in S} p_j \geq K$. In this case we can consider subproblems $T(i, j)$ = the minimum size using only items $1, \ldots, i$ and a profit of at least j. The DP algorithm is then described by the recursion:

$$T(i, j) = \text{minimum of } T(i-1, j) \text{ and } T(i-1, j-p_i) + s_i \text{ for } p_i \leq j.$$

The value of the optimal solution is $\mathrm{opt}(a) = T(n, K)$. The size of (the number of subproblems produced by) this algorithm is nK, and the algorithm is incremental, as well.

Our results

We have just seen that the n-dimensional Knapsack problem with knapsack capacity K can be solved by an incremental DP algorithm using nK subproblems and at most nK^2 relations (wires) between them. A natural question is: can any incremental DP algorithm solve the Knapsack problem using substantially smaller size? Our first result is a *negative* answer: any incremental DP algorithm for the Knapsack problem must have size $\Omega(nK)$, as long as $K \geq 3n$ (Theorem 1). Our next result is that, even if "redundant" paths in the subproblem-graph are allowed, at least [1] $\Omega(nK \log K)$ wires (relations between the subproblems) are necessary to solve the minimization Knapsack problem (Theorem 2). It remains open whether $\Omega(nK)$ *nodes* (subproblems) are necessary in this more general model. Finally, he also shows that the number of nodes in a dynamic BP approximating the Knapsack problem within a factor $1-\epsilon$ lies between n/ϵ and n^3/ϵ.

The model of "dynamic branching programs" introduced in this paper can be generalized in several ways, as sketched in the last section. Strong lower bounds for such generalized models would extend our knowledge about the limitations of DP algorithms, even if they are equipped with features that are not used in existing DP algorithms.

Of particular interest is to understand the role of the feature of allowing "redundant" paths in the subproblem graph, that is, paths that contribute "nothing" to the computed value, but whose presence may substantially reduce the total number of generated subproblems. In classical branching programs such "redundant" paths play a crucial role: their presence *provably* leads to exponential savings in program size. Interestingly, none of existing DP algorithms (we are aware of) use this "strange" feature, so it would be interesting to understand its *algorithmic* meaning. We will shortly discuss this issue in Section 6.

[1] All logarithms in this paper are to the basis 2.

2 Dynamic Branching Programs

We consider 0-1 optimization problems. In each such problem we have some finite set D of *data items*. Each data item $d \in D$ has its *weight* $w(d)$ (a real number). An input (or problem instance) is a sequence $x = (x_1, \ldots, x_n) \in D^n$ of data items. A *solution* for x is a subset $I \subseteq \{1, \ldots, n\}$ of (indexes of) data items in x. There is also some constraint predicate $F : D^n \times 2^{[n]} \to \{0, 1\}$. A solution I is a *feasible* solution for an instance x if and only if $F(x, I) = 1$. Given an input $x \in D^n$, the goal is to maximize (or minimize) the total weight $\sum_{i \in I} w(x_i)$ over all feasible solutions I for x. The value of an optimal solution for a given instance x is denoted by $\mathrm{opt}(x)$. For definiteness, we set $\mathrm{opt}(x) = 0$ if x has no feasible solutions.

For example, in the n-dimensional Knapsack problem with knapsack of capacity K, items are pairs of natural numbers (profit, size). The weight of such an item is its profit. The constraint has the form "the size of accepted items does not exceed K", and $\mathrm{opt}(x)$ is the biggest total profit of an optimal solution for x.

The model of "dynamic branching programs", we are now going to describe, is trying to reduce the original optimization problem to the shortest (or longest) path problem on subgraphs of one particular acyclic graph.

A *dynamic branching program* (dynamic BP) is a directed acyclic graph $P(x_1, \ldots, x_n)$ with two special nodes, the source node s and the target node t. Multiple wires[2], joining the same pair of nodes are allowed. The *size* of a program is the number of its nodes. There are two types of wires: unlabeled wires (*rectifiers*) and labeled wires (*contacts*). Each contact e is labeled by one of the variables x_i, meaning that e is *responsible* for the i-th item in the input sequence. The contact e may also have a *decision predicate* $\delta_e : D \to \{0, 1\}$ about the item it is responsible for, as well as its *survival test* $t_e : D \to \{0, 1\}$:

$$\circ \xrightarrow[\text{decision } \delta_e(x_i)]{\text{survival test } t_e(x_i)} \circ$$

Both these predicates depend only on the item x_i which e is responsible for. The meaning of the decision predicate δ_e is that the item x_i is accepted at e if and only if $\delta_e(x_i) = 1$. The meaning of the survival test t_e is the following: when input $x \in D^n$ comes, the contact e responsible for the i-th variable is removed from the program if $t_e(x_i) = 0$ (the contact has not passed the survival test on input x, or "dies" on item x_i), and e remains intact if $t_e(x_i) = 1$ (or remains "alive" on item x_i). Thus, being "dynamic" means that the structure of the program may depend on the actual problem instance. If the program has no survival tests, then we call it *static*.

A path p is *consistent* with a given input string $x \in D^n$, if this input passes all survival tests along p. A path is consistent, if it is consistent with at least one input string. If p is an s-t path, and if p is consistent with an input $x \in D^n$, then the *solution* produced by p on x is the a multi-set $I_p(x) = \{i \colon \delta_e(x_i) = 1 \text{ and } e \in p\}$ of items accepted along p. A program P *solves* a given optimization problem *on a subset* $A \subseteq D^n$ of problem instances, if the following holds:

[2] We prefer to use the word "node" instead of "vertex" as well as "wire" instead of "edge" while talking about branching programs, because inputs to programs may also be edges and vertices of a customary graph.

1. For every input $x \in A$, there is an s-t path p such that p is consistent with x and $I_p(x)$ is an optimal solution for x.
2. For every input $x \in D^n$ and for every s-t path p, if p is consistent with x, then $I_p(x)$ is a feasible solution for x.

That is, on inputs in A the program must produce optimal solutions, but it cannot produce any infeasible solution on the remaining inputs. A program solves the problem if it solves it on the set $A = D^n$ of all problem instances.

Remark 1 Since we consider not arbitrary, but only 0-1 optimization problems, every dynamic BP for such a problem must (implicitly) fulfill the following "accept at most once" condition. Let p be an s-t path consistent with some input $x \in D^n$. Then one item x_i of x can be rejected and then accepted later. But it cannot be accepted and then accepted again, for otherwise the weight of x_i would contribute more than once to the weight of the solution $I_p(x)$ produced by the path p on x.

We require that a BP (be it static or dynamic) satisfies the following "consistency conditions" for survival tests and decisions: if a contact e_1 precedes a contact e_2 on the same path, and if *both wires are responsible for the same variable* x_i, then we require that

(i) $t_{e_1}(x_i) = t_{e_2}(x_i)$;
(ii) $\delta_{e_1}(x_i) \leq \delta_{e_2}(x_i)$.

The first condition (i) requires that one and the same item cannot live and die along the same path. The second condition (ii) requires that if an item is accepted, then it cannot be rejected later at some other wire responsible for the same item. (This is also required in the model of prioritized BP [11] mentioned in Introduction.) Note however that once rejected, the item *can* be still accepted later.

A dynamic BP is *oblivious* if along every path the items are queried in the same order, and is *read-once* if along every path at most one contact is responsible for one and the same item.

Remark 2 Note that read-once BP need not be oblivious, and an oblivious BP need not be read-once. Every read-once BP automatically satisfies both conditions (i) and (ii). Every static BP automatically satisfies the condition (i) just because there are no survival tests at all.

How does a dynamic BP compute?

When an input $x \in D^n$ comes, each contact e responsible for the i-th item x_i receives its *weight* $w_e(x)$ which is 0, if this item is rejected, and is the weight $w(x_i)$ of this item, if the item is accepted at e. Unlabeled wires (rectifiers) e are responsible for no variable, have no survival tests and make no decisions; their weight is always zero.

The *weight* of a path on input $x \in D^n$ is the sum of the weights of its contacts. The value $P(x)$ of the program on the input x is the maximum (or a minimum, if we have a minimization problem) weight of all s-t paths consistent with x. That is, when an input $x \in D^n$ comes, we first remove all contacts e for which $t_e(x_i) = 0$ (the survival test is not passed on them), and then compute the value $P(x)$ as the maximum weight of an s-t path in the remaining sub-program of P.

To see that dynamic BP simulates incremental DP algorithms, let us observe that the program $P(x)$ computes its value by inductively assigning values to the nodes as follows. The start node s always has a zero value, $\mathrm{val}(s, x) = 0$ for all $x \in D^n$. Let now $e_1, \ldots, e_k$ be all the wires from nodes $v_1, \ldots, v_k$ to a node v that are consistent with the input x; that is, each wire $e_j = (v_j, v)$ is either a rectifier (an unlabeled wire), or is a contact which passed its survival test on input x. Then

$$\mathrm{val}(v, x) = \max\{\mathrm{val}(v_1, x) + w_{e_1}(x), \ldots, \mathrm{val}(v_k, x) + w_{e_k}(x)\}\,,$$

and $P(x) = \mathrm{val}(t, x)$, where t is the target node of P. In other words, at each wire the value is increased by its weight, and at every node the maximum of values coming from its (survived) predecessors is taken. Recall that rectifiers do not change the accumulated value: they are only used to "transport" the value. The presence of such "useless" wires may still substantially decrease the total number of wires (see Remark 7 below).

Many DP algorithms can be almost directly translated to dynamic (and even static) branching programs. We restrict ourselves to several illustrative examples.

Example 1 **Single Source Shortest Paths, Bellman–Ford.** A problem instance is an assignment of real weights w_{ij} to the edges of a directed complete graph K_n on vertices $[n] = \{1, \ldots, n\}$; infinitely large weight $w_{ij} = \infty$ is allowed. The assumption for the weighting is that there cannot be any cycles of negative weight. Note that the problem is only well-defined in the absence of such cycles. The goal is to find a lightest path from vertex $s = 1$ to all remaining vertices, that is, to find paths with the smallest total weight of their edges. Thus, feasible solutions are subgraphs of K_n containing a simple path from 1 to n; since this is a *minimization* problem, optimal solutions must necessarily be just simple paths. Data items are edges (i, j) of K_n, and their costs are the weights w_{ij}.

The Bellman–Ford algorithm for this problem can be easily implemented as a static BP. As subproblems we take $S_k(j)$ = length of a shortest path from 1 to j using at most k edges. The terminal values are $S_1(j) = w_{1j}$, $j = 2, \ldots, n$. The DP recursion is:

$$S_k(j) = \text{minimum of } S_{k-1}(j) \text{ and } S_{k-1}(i) + w_{ij} \text{ for all } i.$$

The optimal value for a vertex j is $S_{n-1}(j)$. The algorithm can be easily implemented as a static BP. A fragment of this BP is shown here:

$$\begin{tikzcd}
S_{k-1}(1) \arrow[drrrr, "\text{accept } x_{1j}"'] & \cdots & S_{k-1}(i) \arrow[drr, "\text{accept } x_{ij}"] & \cdots & S_{k-1}(j) \arrow[d] & \cdots \\
 & & & \cdots & S_k(j) & \cdots
\end{tikzcd}$$

The node (subproblem) $S_k(j)$ is entered by a rectifier (unlabeled wire) $S_{k-1}(j) \to S_k(j)$ as well as by contacts $S_{k-1}(i) \to S_k(j)$ with $i \neq j$ responsible for variables x_{ij}; the weight of each such contact is the weight w_{ij} of the edge (i, j). We also have a source node s with contacts going to all nodes $S_1(j)$ for $j = 2, \dots, n$. Each contact $s \to S_1(j)$ is responsible for the variable (edge) x_{1j}. The decisions of all contacts e are trivial: $\delta_e \equiv 1$ (always accept). The number of nodes in the constructed static BP is $O(n^2)$, and the number of wires is $O(n^3)$.

Example 2 **Maximum Value Contiguous Subsequence.** We are given a sequence $x_1, \dots, x_n$ of integer weights, and the goal is to find a contiguous subsequence with maximal weight. Thus, data items in this case are the weights x_i, each with $w(x_i) = x_i$. Valid solutions are contiguous intervals $I = \{i, i+1, \dots, j\}$, and their values are the sums $x_i + x_{i+1} + \cdots + x_j$. As subproblems we take $S(j)$ = maximum weight of an interval ending in j. The terminal value is $S(0) = 0$. The DP recursion is

$$S(j) = \max\{S(j-1) + x_j,\ x_j\}\,.$$

This algorithm can be implemented as a read-once static BP with $n+1$ nodes:

$$\begin{tikzcd}
S(0) \arrow[r, "x_1"'] \arrow[rr, bend left, "x_2"] \arrow[rrr, bend left, "x_3"] \arrow[rrrrr, bend left, "x_n"] & S(1) \arrow[r, "x_2"'] & S(2) \arrow[r, "x_3"'] & S(3) \arrow[r, "x_4"'] & \cdots \arrow[r, "x_n"'] & S(n)
\end{tikzcd}$$

All decisions are "accept". This program is not oblivious, but is read-once and is static.

Example 3 **Longest Common Subsequence.** Given two sequences $a = (a_1, \dots, a_n)$ and $b = (b_1, \dots, b_m)$ of characters, the goal is to find a longest common subsequence of them. A common subsequence of a and b is a sequence $c = (c_1, \dots, c_k)$ for which there exist two sequences of positions $1 < i_1 < \cdots < i_k < n$ and $1 < j_1 < \cdots < j_k < n$ such that $c_r = a_{i_r} = b_{j_r}$ for all $r = 1, \dots, k$. That is, a subsequence need not be *consecutive*, but must be *in order*. In this case, data items are pairs $x_{ij} = (a_i, b_j)$, each of weight 1. As subproblems we take $S(i,j)$ = length of the longest subsequence of $(a_1, \dots, a_i)$ and $(b_1, \dots, b_j)$. The terminal values are $S(0,j) = S(i,0) = 0$ for all i, j. The DP recursion is

$$S(i,j) = \max\{S(i-1,j-1) + 1, S(i,j-1), S(i-1,j)\}\,,$$

and the answer is $S(n,m)$. This algorithm can be easily implemented as a static read-once BP with $O(nm)$ nodes $S(i,j)$:

$$\begin{tikzcd}
S(i-1,1) & \cdots S(i-1,j-1) \arrow[dr, "\text{accept } x_{ij} \text{ iff } {a_i=b_j}"'] & S(i-1,j) \arrow[d, "\text{reject } x_{ij}"] & \cdots \\
& \cdots \quad S(j,j-1) \arrow[r, "\text{reject } x_{ij}"'] & S(i,j) & \cdots
\end{tikzcd}$$

Example 4 **Knapsack problem.** Recall that the standard DP algorithm for the maximization Knapsack problem is given by the recursion

$$S(i,j) = \max\{S(i-1,j), S(i-1,j-s_i) + p_i\}\,,$$

where $S(i,j) =$ the maximal total profit for filling a capacity j knapsack with some subset of items $1,\ldots,i$. The output is $S(n,K)$. This algorithm can be easily turned into an oblivious read-once dynamic BP with nK nodes as follows:

$$\begin{tikzcd}
S(i-1,1) & \cdots & S(i-1,k) \arrow[drr, "\text{accept } x_i", "\text{is } s_i = j-k?"'] & \cdots & S(i-1,j) \arrow[d, "\text{reject } x_i"] & \cdots \\
 & & & \cdots & S(i,j) & \cdots
\end{tikzcd}$$

As nodes we take the subproblems $S(i,j)$. For every $k = 1,\ldots,j$, there is a contact $S(i-1,k) \to S(i,j)$ responsible for the i-th item $x_i = (p_i, s_i)$. The contact $S(i-1,j) \to S(i,j)$ has no survival test, and makes the decision $\delta(x_i) \equiv 0$ (always reject). Each of the remaining contacts $S(i-1,k) \to S(i,j)$, for $k < j$, has the survival test $t_e(x_i) = 1$ if and only if $s_i = j - k$. The decision of each such contact e is $\delta_e(x_i) \equiv 1$ (always accept), and hence, the weight of each such contact is $w(x_i) = p_i$. Thus, when an input x with $x_i = (p_i, s_i)$ comes, only two (out of j) contacts $S(i-1,j) \to S(i,j)$ and $S(i-1,j-s_i) \to S(i,j)$ entering the node $S(i,j)$ will survive

There is also the start node $S(0,0)$ from which there is a contact responsible for the first item x_1 to each of the nodes $S(1,1),\ldots,S(1,K)$. Each contact $S(0,0) \to S(1,j)$ makes a trivial decision $\delta(x_1) \equiv 1$ (accept the first item), and has a survival test $t_e(a_1) = 1$ iff $s_1 \leq j$. The target node is $S(n,K)$. It is easy to see that the resulting dynamic BP is read-once and oblivious. The program has nK nodes and $O(nK^2)$ wires. The dynamic BP for the *minimization* Knapsack problem is similar.

Remark 3 Note that we have *one* program for *all* problem instances. But we do not require that *every* optimal solution for a given input must be produced by some s-t path: it is enough that one path produces an optimal solution, and none of the remaining paths produces an infeasible solution.

Remark 4 The model of static BP tries to solve the original problem by reducing it to the "heaviest" (or "lightest") s-t path problem on one particular *acyclic* graph. Namely, every instance $a \in D^n$ defines some weighting of the wires of the underlying graph G of the static BP P, and the output value $P(a)$ is the weight of the heaviest s-t path in G. If the program is dynamic (has survival tests), then $P(a)$ is the the weight of the heaviest s-t path in a *subgraph* of G defined by the instance a.

Remark 5 A similar in its "sole" model of so-called *combinatorial dynamic programs* was considered by Bompadre in [7]. This model is essentially our model of *static* BP with all decision predicates being "accept". Thus, a solution produced by a path is just the set of items tested along this path. Only the weights of contacts depend on the actual input. Besides being static, combinatorial DPs have one restriction, not present in our model: *each* feasible solution must be produced by at least one s-t path (see Definition 3, item 3 in [7]). In our model it is enough that: (1) at least one optimal solution is produced, and (2) every produced solution must be feasible. Using a reduction to monotone arithmetic circuits, exponential lower bounds on the number of wires in combinatorial DP are proved in [7] for some "permanent-like" optimization problems: Traveling Salesman Problem, the Bipartite Matching Problem, and the Min and Max s-t Cut Problems. First exponential lower bounds for monotone arithmetic circuit were proved by Jerrum and Snir [16], and these were also for "permanent-like" problems. The Knapsack problem is not "permanent-like", and for it, no lower bound was known in this model.

Remark 6 The way how a dynamic BP computes its value depends on what semiring we are working over. So as defined above, dynamic BPs work over the semiring $(\max, +)$ or $(\min, +)$. The weight of a path in this case is the *sum* of weights of the contacts accepting the corresponding items, and the value of the program is the maximum (or minimum) of the weights of all consistent paths. In the *boolean* semiring $(\{0,1\}, \vee, \wedge)$, the weight of a path is the AND of the weights of its contacts. That is, in this case we have AND instead of Plus, and OR instead of Max. Thus, if we let all decisions be "accept" ($\delta_e \equiv 1$), and set $w(0) = w(1) = 1$, then any dynamic BP P over the boolean semiring turns to a classical nondeterministic branching program computing some boolean function $f : \{0,1\}^n \to \{0,1\}$ by: $f(a) = 1$ if and only if there is an s-t path in P consistent with a.

Are our restrictions on dynamic BP reasonable?

Our restriction on survival tests $t_e : D \to \{0,1\}$ is twofold: first we require these tests be "local" (they can only depend on a single item in the input sequence), and we have the consistency condition (i) on them. Both these restrictions on survival tests are (more or less explicitly) present in other formalizations of DP algorithms, including the model of prioritized branching programs invented in [11].

It is easy to see that we cannot allow arbitrary survival tests $t_e : D^n \to \{0,1\}$, because the resulting model would be too powerful: any n-dimensional 0-1 optimization problem could then be solved by an oblivious read-once dynamic BP of size n. Indeed, we could then take a sequence $v_1, \ldots, v_n$ of nodes, and draw two parallel wires $e_{i,0}$ and $e_{i,1}$ from v_i to v_{i+1}, both responsible for the i-th item x_i. Let the decision made at the wire $e_{i,\alpha}$ be $\delta(x_i) \equiv \alpha$ (always accept or always reject). Finally, define the survival tests $t_{i,\alpha}(x)$ of these wires as follows. For each feasible input $x \in D^n$, fix an optimal solution $I_x \subseteq [n]$ for x. Then define $t_{i,1}(x_i) = 1$ if and only if $i \in I_x$, and $t_{i,0}(x_i) = 1$ if and only if $i \notin I_x$. Now, when an input $x \in D^n$ comes, only one s-t path will survive, and I_x is the solution produced by this path.

Our consistency condition on survival tests (present also in the model of prioritized BP [11]) is that one and the same item cannot survive and die along the same path. If we remove this condition, then the resulting model will be also very powerful (even if each test depends on a single data item): it will have the power of unrestricted branching programs. This happens even if all tests have the form $x_i = d$, and all decisions are constant.

Recall that a *nondeterministic branching program* (NBP) for a boolean function $f : \{0,1\}^n \to \{0,1\}$ is a directed acyclic graph where at some wires tests of the form "is $x_i = 0$?" or a test "is $x_i = 1$?" are made; if there are no rectifiers (wires at which no test is made), then such a program is usually called *contact scheme*. We also have a source node s and a target node t. Such a program accepts an input $x \in \{0,1\}^n$ if and only if this input passes all tests of at least one s-t path. The strongest lower bound for NBPs remains the lower bound $\Omega(n^{3/2}/\log n)$ proved by Nechiporuk [22]. Moreover, this bound is on the number of *wires*; concerning the number of *nodes* (the measure we are interested in), even super-linear lower bounds are not known.

Proposition 1 *Without the consistency condition on survival tests, dynamic BPs are at least as powerful as boolean nondeterministic branching programs.*

Proof With every boolean function $f : \{0,1\}^n \to \{0,1\}$ we can associate the following artificial maximization problem with a linear target function. In this problem, data items are boolean bits $a_i \in \{0,1\}$, each with $w(a_i) = 1$. Solutions are vectors $\delta \in \{0,1\}^n$. Such a solution δ is feasible for input instance $a \in \{0,1\}^n$ if $f(a) = 1$ and δ has exactly one 1. The goal is to compute $\mathrm{opt}(a) = \max \sum_{i=1}^n \delta_i \cdot w(a_i)$ over all feasible solutions δ for a. Note that every dynamic BP solving this problem must compute the function f.

Suppose now we have a nondeterministic branching program P computing f. We can transform P into a dynamic BP $P'(x)$ solving the optimization problem for f by just re-labeling the wires. Let e be a wire in P at which a test $t_e(x_i)$ of the form "is $x_i = 1$?" is made. We leave this test as a survival test of e, and define the decision predicate δ_e at e by: $\delta_e(x_i) \equiv 1$, if e is a wire leaving the start node s of P, and $\delta_e(x_i) \equiv 0$ otherwise. Thus, along each consistent s-t path exactly one item is accepted. Since the NBP accepts an input $a \in \{0,1\}^n$ if and only if there exists an s-t path consistent with a, the resulting dynamic BP solves the maximization problem for f. □

3 Lower Bound for Dynamic Programs

We have just shown that the n-dimensional Knapsack problem can be solved by a dynamic BP using nK nodes, where K is the capacity of the knapsack. Moreover, the resulting BP is read-once and oblivious. We will now show that this trivial upper bound is almost tight: $\Omega(nK)$ nodes are also necessary, even in the class of non-oblivious and not read-once programs. Moreover, this number of nodes is necessary already to solve the *subset-sum* problem, a special case of the Knapsack problem, where the profit of each item is equal to its size. To prove this, we first establish some properties of integer partitions, which may be of independent interest.

3.1 Integer partitions

Let $k \leq n$ be two fixed natural numbers. A *partition* of n into k blocks is a vector $x = (x_1, \ldots, x_k)$ of non-negative integers such that $x_1 + \cdots + x_k = n$. By a *test* we mean a pair (S,b), where $S \subseteq [k]$, and $0 \le b \le n$ is an integer. Such a test is *legal* if $0 \neq |S| \le k-1$. Say that a test (S,b) *covers* a partition x if $\sum_{i\in S} x_i = b$. Let us call S the *support*, and b the *threshold* of the test (S,b). Note that the (illegal) test $([k],n)$ alone covers all partitions. We are interested in how many *legal* tests we need to cover all partitions. So, let $\tau(n)$ denote the minimum number of legal tests that cover all partitions of n into k blocks. We have the following surprisingly tight result.

Lemma 1 $\tau(n) = n + 1$.

Proof The upper bound $\tau(n) \le n+1$ is easy: already tests $(\{1\}, b)$ with $b = 0, 1, \ldots n$ will do the job. To prove the lower bound $\tau(n) \ge n+1$, we argue by induction on n and on the number m of supports in the collection.

If $m = 1$ then for every n, all the tests have the same support S, say, $S = \{1, \ldots, r\}$. If some threshold b is missing, then the vector $x = (b, 0, \ldots, 0, n-b)$ is a partition of n, but it is covered by none of the tests, because the legality of the tests implies $r < k$. Thus, in this case $n+1$ tests are necessary.

For general m, fix one support S containing no other support (from our collection of tests) as a proper subset. Take the *smallest* number c which does not appear as a threshold b in any of our tests of the form (S,b). Thus, we must already have at least c tests $(S,0), (S,1), \ldots, (S,c-1)$ with support S in our collection.

The remaining tests (T,b) with $T \neq S$ in our collection can be modified in such a way that they cover all partitions of $n-c$ into $k - |S| = k - r$ blocks. Namely, fix a string of numbers $(a_i : i \in S)$ summing up to c, and concentrate on partitions of n containing this string. By ignoring the positions $i \in S$, these partitions give us all partitions of $n - c$ into $k - |S|$ blocks. Now we modify the remaining tests (T,b) so that they cover all these (shorter) partitions. If $T \cap S \neq \emptyset$, then replace (T,b) by the test $(T \setminus S, b')$ where $b' = b - \sum_{i\in S\cap T} a_i$. If $T \cap S = \emptyset$, then leave the test (T,b) as it is. By induction hypothesis, there must be at least $n-c+1$ such (modified) tests, giving a lower bound $n+1$ on the total number of tests. □

Let $\tau^+(n)$ denote the version of $\tau(n)$ in the case when only *positive* integers are allowed to participate in a partition; we call such partitions *positive* partitions.

Lemma 2 $\tau^+(n) \ge \tau(n-k)$.

Proof There is a 1-1 correspondence between positive partitions x of n and partitions x' of $n-k$ given by $x' = (x_1 - 1, \ldots, x_k - 1)$. Now suppose we have a collection of tests covering all positive partitions of n. Replace each test (S,b) by the test $(S, b - |S|)$. Note that $b \ge |S|$ if the test covers at least one positive partition x, because then $\sum_{i\in S} x_i = b$ and all $x_i \ge 1$. Since $\sum_{i\in S} x_i = b$ implies that $\sum_{i\in S}(x_i - 1) = b - |S|$, a partition x' of $n-k$ passes the test $(S, b-|S|)$ if the positive partition x of n passes the test (S,b). Thus, the new collection of test covers all partitions of $n-k$. □

3.2 A lower bound for subset-sum problem

We consider a special case of the Knapsack problem, where the profit of each item is equal to its size. In this problem, which we call the (n,K)-*knapsack problem* (to indicate the number of items, and the capacity of the knapsack), input instances are sequences $a = (a_1, \dots, a_n)$ of integers in $[K] = \{1, \dots, K\}$. The goal is to compute the maximum $\mathrm{opt}(a) = \max \sum_{i \in I} a_i$ over all subsets $I \subseteq [n]$ such that $\sum_{i \in I} a_i \le K$. In the *minimization* problem, the goal is to compute the minimum $\mathrm{opt}(a) = \min \sum_{i \in I} a_i$ over all subsets $I \subseteq [n]$ such that $\sum_{i \in I} a_i \ge K$.

Theorem 1 *If $K \ge 3n$ then every dynamic branching program solving the maximization or minimization (n, K)-knapsack problem must have at least $\frac{1}{2}nK$ nodes.*

Proof Take a dynamic branching program $P = (V, E)$ solving the (n, K)-knapsack problem. In particularly, this program must solve the problem on the set $A \subseteq [K]^n$ of all positive partitions of K, that is, on the set of all input strings $a = (a_1, \dots, a_n)$ such that $a_1 + \cdots + a_n = K$, and all a_i belong to $[K] = \{1, \dots, K\}$.

For every $a \in A$, there must be an *s*-*t* path which is consistent with a and has weight K on input a. Fix one such path, and call it the *optimal path* for a. Since none of the inputs in A has a zero component (partitions are *positive*), along each optimal path exactly n items must be accepted.

Fix now an integer $r \in \{1, \dots, n-1\}$, and stop the optimal path for a after exactly r items of a were accepted. Let p_a denote the first segment (until the "stop-node") and q_a the second segment of the optimal path for a. Let also $V_r \subseteq V$ denote the set of nodes v in our program such that the optimal path of at least one input instance $a \in A$ was stopped at v.

Recall that our set A of inputs is the set of positive partitions $a_1 + \cdots + a_n = K$ of K into n blocks.

Claim 1 *For every $r = 1, \dots, n-1$, the set of all positive partitions of K into n blocks can be covered by $|V_r|$ legal tests with supports of size r.*

Together with Lemma 2, this implies that $|V_r| \ge \tau^+(K) \ge K - n + 1$. This will prove the theorem, because then $|V| \ge (n-1)(K-n+1) = Kn - K - (n-1)^2$, which is $\ge \frac{1}{2}nK$ for $K \ge 3n$, as desired.

So, the rest is devoted to the proof of Claim 1.

Fix an integer $r \in \{1, \dots, n-1\}$ and a node $v \in V_r$. Let $A_v \subseteq A$ be the set of inputs $a \in A$ whose optimal paths were stopped at v, that is, $a \in A_v$ if and only if v is the last node of p_a. Recall that the optimal path for each input $a \in A_v$ consists of its initial segment p_a and its final segment q_a. Let

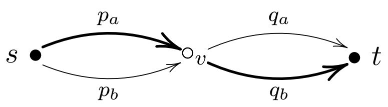
$$I_a = \{i \in [n] \colon x_i \text{ is tested along } p_a\} \text{ and } J_a = \{i \in [n] \colon x_i \text{ is tested along } q_a\}.$$

Let also

$$P_a = \{i \in I_a \colon a_i \text{ is accepted along } p_a\} \text{ and } Q_a = \{i \in J_a \colon a_i \text{ is accepted along } q_a\}\ .$$

That is, P_a is the partial solution produced by the path p_a on input a, and Q_a is the partial solution produced by the path q_a on this input. Hence, $|P_a| = r$ and $|Q_a| = n - r$ for every $a \in A$. Moreover, $P_a \cup Q_a$ is the solution for a produced by the entire *s*-*t* path (p_a, q_a), and $P_a \cap Q_a = \emptyset$ because we consider an 0-1 optimization problem (see Remark 1).

Take two arbitrary inputs $a \neq b \in A_v$. The initial and final segments of the optimal paths for a and b look like:



By a *combination* of a pair (a, b) we will mean any input $c \in [K]^n$ such that $c_i = a_i$ for all $i \in I_a \setminus J_b$, $c_i = b_i$ for all $i \in J_b \setminus I_a$, and $c_i \in \{a_i, b_i\}$ otherwise.

Claim 2 *Let c be a combination of (a, b). Then c is consistent with the combined path (p_a, q_b), and all items c_i with $i \in I_a \cap J_b$ are rejected along p_a.*

In particular, by taking a combination c with $c_i = a_i$ for all $i \in I_a \cap J_b$, this implies that all items a_i with $i \in I_a \cap J_b$ must be rejected along p_a as well.

Proof In the proof of this claim we will essentially use the consistency conditions (i) and (ii) on dynamic BP. The condition (i) is on survival tests: along every path, the survival tests on the same variable x_i must be the same. The second condition (ii) is on decision predicates: if an item is accepted on a path, then it cannot be rejected latter on that path.

We first show that input c is consistent with (p_a, q_b). We know that input a is consistent with the first segment p_a, and input b is consistent with the second segment q_b. Thus, input c passes all survival tests $t_e(x_i)$ made on variables x_i such that $i \notin I_a \cap J_b$. Take now an $i \in I_a \cap J_b$. Then the variable x_i is tested at some contact e_1 of p_a, and then is re-tested at some contact e_2 of q_b. We know that $t_{e_1}(a_i) = 1$ and $t_{e_2}(b_i) = 1$. So, regardless of what the actual value of c_i is ($c_i = a_i$ or $c_i = b_i$), the consistency condition $t_{e_1}(x_i) = t_{e_2}(x_i)$ implies that $t_{e_1}(c_i) = t_{e_2}(c_i) = 1$. Hence, input c passes all tests on variables x_i with $i \in I_a \cap J_b$, as well.

To show that all items c_i with $i \in I_a \cap J_b$ are rejected along p_a, suppose that there is contact e_1 of p_a testing the i-th variable x_i such that $\delta_{e_1}(c_i) = 1$ (item c_i is accepted at e_1). Since $i \in J_b$, the variable x_i is re-tested at some contact e_2 along the path q_b. The consistency condition (ii) implies $1 = \delta_{e_1}(c_i) \leq \delta_{e_2}(c_i)$. Thus, along the combined s-t path, the i-th item c_i of c is accepted at least two times (at contacts e_1 and e_2), implying that the solution produced by the that path on input c is *not* a feasible solution for c, a contradiction. □

We now show that, on all inputs $a \in A_v$, the initial segments p_a produce the same partial solution, and that the weight of this solutions is the same for all $a \in A_v$. For a path p and an input a consistent with it, let $w_p(a) = \sum_{i \in P_a} a_i$ denote the weight of this path on input a, that is, the sum of weights of items of a accepted along this path.

Claim 3 *For all $a, b \in A_v$, we have $P_a = P_b$ and $w_{p_a}(a) = w_{p_b}(b)$.*

Proof By Claim 2, we have that $P_a \cap Q_b = \emptyset$. Thus, the sets $P_a \cup P_b$ and $Q_a \cup Q_b$ are disjoint. Together with $|P_a| = |P_b| = r$ and $|Q_a| = |Q_b| = n - r$, this implies $P_a = P_b$.

To show that $w_{p_a}(a) = w_{p_b}(b)$, assume w.l.o.g. that $w_{p_a}(a) \geq w_{p_b}(b)$. Consider a combination c of (a, b) such that $c_i = b_i$ for all $i \in J_b$, that is, we set $c_i = b_i$ for all $i \in I_a \cap J_b$.

Then clearly, $w_{q_b}(c) = w_{q_b}(b)$. By Claim 2, for all $i \in I_a \cap J_b$, the items $c_i = b_i$ and a_i are rejected along p_a. But on the remaining variables x_i with $i \in I_a \setminus J_b$, the input c coincides with a. Thus, for this particular combination c, we have that $w_{p_a}(c) = w_{p_b}(a)$ and $w_{q_b}(c) = w_{q_b}(b)$.

Had we now a strict inequality $w_{p_a}(a) > w_{p_b}(b)$, then the combined path (p_a, q_b) would produce an solution for c of weight $w_{p_a}(c) + w_{q_b}(c) \geq w_{p_a}(a) + w_{q_b}(b) > w_{p_b}(b) + w_{q_b}(b) = K$. If our program solves the *maximization* (n, K)-knapsack problem, this would be an infeasible solution for c. Thus, the equality $w_{p_a}(a) = w_{p_b}(b)$ holds in this case. If the program solves the *minimization* (n, K)-knapsack problem then the same equality follows by interchanging the roles of inputs a and b. □

We can now finish the proof of Claim 1, and thus, the proof of Theorem 1 as follows. Take a node $v \in V_r$. By Claim 3, we know that every path p_a for $a \in A_v$ accepts the same set S of $|S| = r$ items of a, and all the weights $w_{p_a}(a) = \sum_{i \in S} a_i$ of these paths are the same. Thus, every node $v \in V_r$ gives a legal test (S, b) covering all partitions in A_v. Since the set A of all positive partitions of K into n blocks is the union of the sets A_v with $v \in V_r$, we can conclude that the set A can be covered by $|V_r|$ legal tests, as desired. □

4 Lower Bound for General Dynamic Programs

We now consider dynamic branching programs where only survival tests of the form ”is $x_i = d$?” are allowed, but there are no other restrictions, in particular, there are no consistency conditions. That is, along one path, two contradictory tests ”is $x_i = d_1$?” and ”is $x_i = d_2$?” for $d_1 \neq d_2$ may be made. We, however, assume that there are no rectifiers, that is, every wire *has* a survival test. Let us call such programs *general dynamic BP*. We are going to prove a non-trivial lower bound on the number of *wires* in such a program.

For this purpose, it will be convenient to consider the *minimization* Knapsack problem. Just as in the case of maximization Knapsack problem, the DP algorithm for the minimization gives rise to a (read-once and oblivious) dynamic BP with at most nK^2 wires. We will now show that about $nK \log K$ wires are also necessary even in the class of general dynamic BP.

We again consider the simplified version of the minimization Knapsack problem, where the profit of each item is equal to its weight. That is, data items are integers in $D = \{0, 1, \ldots, K\}$. As before, a solution for a problem instance $a \in D^n$ is a subset $I \subseteq [n]$. Such a solution I is *feasible*, if $\sum_{i \in I} a_i > K$. The goal is to minimize the sum $\sum_{i \in I} a_i$ over all feasible solutions I (if there are any). We already know (see Example 4) that nK^2 wires are enough, even in the class of oblivious read-once dynamic programs.

Theorem 2 *Every general dynamic BP solving the minimization (n, K)-Knapsack problem must have $\Omega(nK \log K)$ contacts.*

Our proof will be based on a classical result of Hansel [13] stating that any monotone contact scheme computing the threshold-2 function $Th_2^m(x_1, \ldots, x_m)$ must have at least $\Omega(m \log m)$ contacts. Recall that Th_2^m accepts a boolean vector if and only if it contains at least two 1s.

Proof Let $P(x_1,\dots,x_n)$ be a general dynamic BP solving the minimization Knapsack problem. We assume that the number n of items is even, and the capacity K is an odd number. To prove that P must have at least $\Omega(nK\log K)$ contacts, it is enough to show that, for every $i=1,\dots,n/2$, the program P must contain at least $\Omega(K\log K)$ contacts responsible for variables x_{2i-1} and x_{2i}. By symmetry, it is enough to show this only for $i=1$. That is, it is enough to show that the number of contacts responsible for x_1 and x_2 must be at least $\Omega(K\log K)$.

As before, a path is consistent with an input string if this string passes all survival tests along that path. With some abuse of notation, we will say that a dynamic program "accepts" an input string $a\in D^n$ if at least one s-t path is consistent with a, and "rejects" a if no s-t path is consistent with a. Thus, our program P accepts an input a if an only if $\sum_{i=1}^n a_i > K$.

Recall that our domain is $D=\{0,1,\dots,K\}$. Call a pair $(u,v)\in D^2$ an *even-odd pair*, if $u+v>K$, u is even and v is odd. Let $S\subseteq D$ be the second half of our domain, that is, $S=\{\lceil K/2\rceil,\lceil K/2\rceil+1,\dots,K\}$. Our proof consists of the following two steps:

(i) Modify $P(x_1,\dots,x_n)$ to obtain a program $P'(x_1,x_2)$ which accepts exactly even-odd pairs.
(ii) Modify $P'(x_1,x_2)$ to obtain a monotone contact scheme $Q(y_u\colon u\in S)$ of $m=|S|$ new boolean variables which computes the threshold-d function $Th_2^m(y_u\colon u\in S)$.

The modifications will not increase the total number of contacts: we only contract/remove some of contacts and/or replace their labels. By Hansel's result the scheme Q must have $\Omega(m\log m)$ contacts. Thus, at least so many contacts should have been responsible for variables x_1 and x_2 in program P, as desired.

We obtain the program $P'(x_1,x_2)$ of step (i) from the program P as follows:

- Remove all contacts making a test $x_i=d$ for $d\neq 0$ and $i\geq 3$.
- Remove all contacts making a test $x_1=d$ for d odd, and all contacts making a test $x_2=d$ for d even.
- Contract all contacts making tests $x_i=0$ for $i\geq 3$.

In this way, the program $P'(x_1,x_2)$ accepts a pair $(u,v)\in D^2$ if and only if the following holds: the program P accepted the input $(u,v,0,\dots)$, and $u+v>K$, and u is even, and v is odd. That is, P' accepts exactly even-odd pairs. Note that in P', only tests $x_1=d$ for $d\in D$ even, and tests $x_2=d$ for $d\in D$ odd are made.

The monotone boolean contact scheme $Q(y_u\colon u\in S)$ of step (ii) is obtained from $P'(x_1,x_2)$ as follows. First, remove from P' all decision predicates. Then replace the tests $x_1=d$ and $x_2=K-d$ for $d\in D$ even, by the (boolean) test

$$y_{\max\{d,K-d\}}=1\,.$$

Note that, for all $d\in D$, $u=\max\{d,K-d\}$ belongs to S, implying that the obtained tests are on the variables in our set $\{y_u\colon u\in S\}$. Moreover, for all $u\in S$, $y_{\max\{d,K-d\}}=y_u$. Thus, since K is odd, a test $y_u=1$ with $u\in S$ can only be obtained from

$$\begin{array}{llll} x_1=u & \text{and} & x_2=K-u & \text{if } u \text{ is even,}\\ x_1=K-u & \text{and} & x_2=u & \text{if } u \text{ is odd.}\end{array} \tag{1}$$

The monotone boolean contact scheme $Q(y_u\colon u \in S)$ computes the threshold-2 function $Th_2^m(y_u\colon u \in S)$. Since the scheme Q is monotone, it is enough to show that it accepts all vectors with exactly two 1s, and rejects all vectors with exactly one 1. To show this, take an arbitrary vector $b \in \{0,1\}^S$ with one or two 1s.

Assume first that b contains two 1s in positions $u \neq v$ in S. We have to show that $Q(b) = 1$, that is, there exist an s-t path in Q along which only tests $x_1 = u$ and $x_2 = v$ are made. For this, we use the fact that, if $P'(x_1, x_2)$ accepts the pair (u, v) then along at least one s-t path in P' only tests $x_1 = u$ and $x_2 = v$ are made; moreover, each of these tests must be made at least once, because otherwise P' would wrongly accept $(u, 0)$ or $(0, v)$.

Case 1a: u is even and v is odd. Since $u + v > K$, the pair (u, v) is an even-odd pairs, and program P' accepts it. That is, along at least one s-t path in P' only tests $x_1 = u$ and $x_2 = v$ are made. By (1), along the corresponding path in Q only tests $y_u = 1$ and $y_v = 1$ are made, implying that $Q(b) = 1$.

Case 1b: both u and v are even. Suppose $u > v$. Then $u + (K - v) > K$. Since $K - v$ is odd, the pair $(u, K - v)$ is an even-odd pair, and program P' accepts it. That is, along at least one s-t path in P' only tests $x_1 = u$ and $x_2 = K - v$ are made. By (1), along the corresponding path in Q only tests $y_u = 1$ and $y_v = 1$ are made, implying that $Q(b) = 1$.

Case 1c: both u and v are odd. Suppose $u > v$. Since $K - v$ is even, the pair $(K - v, u)$ is an even-odd pair, and program P' accepts it. That is, along at least one s-t path in P' only tests $x_1 = K - v$ and $x_2 = u$ are made. By (1), along the corresponding path in Q only tests $y_v = 1$ and $y_u = 1$ are made, implying that $Q(b) = 1$.

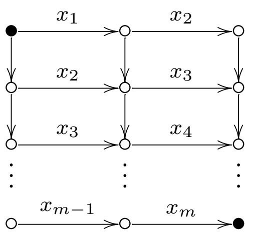
Assume now that b contains exactly one 1 in some position $u \in S$. We have to show that $Q(b) = 0$.

Case 2a: u is even. Suppose that $Q(b) = 1$. Then there is an s-t path in Q where only tests $y_u = 1$ are made. Since u is even, (1) implies that each such test could be obtained only from the test $x_1 = u$ or from the test $x_2 = K - u$. Thus, along the corresponding path in $P'(x_1, x_2)$, only these tests are made, implying that P' wrongly accepts the input $(u, K - u)$, a contradiction.

Case 2b: u is odd. Suppose that $Q(b) = 1$. Then there is an s-t path in Q where only tests $y_u = 1$ were made. Since u is odd, (1) implies that each such test could be obtained only from the test $x_1 = K - u$ or from the test $x_2 = u$. Thus, along the corresponding path in $P'(x_1, x_2)$, only these tests are made, implying that P' wrongly accepts the input $(K - u, u)$, a contradiction.

This completes the proof of Claim 4, and thus, the proof of Theorem 2. $\square$

Remark 7 In our proof it was essential that no rectifiers (unlabeled wires) were allowed. The reason is that using rectifiers, the threshold-2 function Th_2^m *can* be computed using only $2m - 2$ contacts:



It would be interesting to prove a non-trivial lower bound for the Knapsack problem in the general model where rectifiers are allowed. It would be also interesting to prove such a bound on the number of nodes, not only wires. The proof above cannot give larger than $\Omega(n \log K)$ lower bound on the number of nodes, because the complete m-vertex graph can be covered by $O(\log m)$ complete bipartite graphs, and hence, Th_2^m *can* be computed by a monotone contact scheme with $O(\log m)$ nodes.

5 Bounds for approximation

Approximation algorithms for the (general) knapsack problem with capacity K usually use an additional "argmax" feature. Given an input $(p_1, s_1), \ldots, (p_n, s_n)$, one first constructs a table of values

$$S(i,p) = \min\Big\{ \sum_{i\in I} s_i \colon I \subseteq \{1,\ldots,i\} \ \text{ and } \ \sum_{i\in I} p_i = p \Big\}\,. \tag{2}$$

That is, $S(i,p)$ is the smallest total size of a subset of the first i items whose total profit is exactly p. Then one scans the last "row" $S(n,1), S(n,2), \ldots, S(n,p), \ldots$ and outputs the maximal p for which $S(n,p) \leq K$. For every input, the running time of this algorithm is $O(n\sum_i p_i) = O(n^2 \max_i p_i)$. Thus, for inputs with small profits, that is, with $p_1 + \cdots + p_n \ll K$, this algorithm is more efficient than the $O(nK)$ algorithm we considered above.

In order to implement such algorithms, we can also add the "argmax" feature to dynamic branching programs. Namely, we now allow the program to have more than one target node $t_1, \ldots, t_N$. When input string x comes, the value $P(x)$ is now computed as follows. As before, the value $\mathrm{val}(t_p, x)$ computed at the node t_p on input x is the minimum (or maximum) of weights of s-t_p paths consistent with x. After that, the program outputs the maximal p for which $\mathrm{val}(t_p, x)$ is at most some given in advance threshold K. That is,

$$P(x) = \arg\max_p \left\{\mathrm{val}(t_p, x) \colon \mathrm{val}(t_p, x) \leq K\right\}.$$

We also add yet another feature: we now allow that every survival text $t_e(x_i)$ can also depend on the total sum of weights of all n items $x_1, \ldots, x_n$. Let us call this extended model an *argmax dynamic BP*.

Remark 8 It is not difficult to see that the lower bound, given in Theorem 1, remains true also for argmax dynamic BP. The reason is that we have proved this lower bound on a very special set A of inputs a such that $a_1 + \cdots + a_n = K$: we associated with every such input an optimal path, and argued that not too many of these paths can meet in a node.

Now, on every input $a \in A$, the program will output the answer $K = \mathrm{val}(t_K, a)$ computed at *one and the same* target node t_K. The dependence of survival tests on the the total sum $a_1 + \cdots + a_n$ of weights is irrelevant, because this sum is the same for all $a \in A$. Thus, when restricted to inputs in A, the argmax dynamic BP works just as a usual dynamic BP with one source node s and one target node t_K.

Proposition 2 *If the sum of profits in every input for an n-dimensional knapsack problem does not exceed t, then the problem can be solved by an argmax dynamic BP with $O(nt)$ nodes.*

Proof The values $S(i, p)$ defined in (2) can be computed recursively as follows. Each base value $S(1, p)$ equals s_1 if $p_1 = p$, and is infinite otherwise. The recursion is then

$$S(i, p) = \text{minimum of } S(i-1, p) \text{ and } S(i-1, p-p_i) + s_i \text{ for } p_i \leq p.$$

Here i runs from 1 to n, and p runs from 0 to t. By using construction, similar to that in Example 4, one can easily translate this algorithm into an argmax dynamic BP with $O(nt)$ nodes. A fragment of this BP is shown here:

$$\begin{tikzcd}
S(i-1,1) & \cdots & S(i-1,k) \arrow[drr, "\text{accept } x_i", "\text{is } p_i = p-k?"'] & \cdots & S(i-1,p) \arrow[d, "\text{reject } x_i"] & \cdots \\
 & & & \cdots & S(i,p) & \cdots
\end{tikzcd}$$

Note that the value of a BP is here computed as the *minimum* (not the maximum) weight of a consistent s-t path. □

This dynamic BP can be used to approximate the (n, K)-knapsack problem by relatively small dynamic BPs with the argmax feature.

One says that an algorithm $P(x)$ approximates a given maximization problem on a given set $A \subseteq D^n$ of inputs with the factor α, if $P(a) \geq \alpha \cdot \mathrm{opt}(a)$ holds for all $a \in A$. Clearly, the closer is α to 1, the better approximation we have.

Theorem 3 *For every $\varepsilon > 0$, the (n, K)-knapsack problem can be approximated with the factor $1 - \varepsilon$ by an argmax dynamic BP with $O(n^3/\varepsilon)$ nodes. Moreover, $\Omega(n/\varepsilon)$ nodes necessary for every $3n \leq K < 1/\varepsilon$.*

Proof We first prove the second claim (the lower bound $\Omega(n/\varepsilon)$). For optimization problems whose values are integers and maximal optimal value is an integer K, we have the following simple fact: if an algorithm approximates the problem with a factor $> 1 - 1/K$, then it solves the problem *exactly.*

To see this, suppose that the algorithm does not solve the problem exactly. Then there must be an input instance a on which the algorithm produces a value strictly smaller than

$\mathrm{opt}(a) \leq K$. The next best integer solution is $\mathrm{opt}(a) - 1$. So the best possible error ratio the program can get is $(\mathrm{opt}(a) - 1)/\mathrm{opt}(a) = 1 - 1/\mathrm{opt}(a) \leq 1 - 1/K$.

Suppose now that an argmax dynamic BP approximates the (n, K)-knapsack problem with the factor $1 - \varepsilon$. If $K < 1/\varepsilon$, then $1 - \varepsilon > 1 - 1/K$, implying that the program must solve the problem exactly. Since $K \geq 3n$, Theorem 1 and Remark 8 imply that the program must have at least $\frac{1}{2}nK = \Omega(n/\varepsilon)$ nodes.

We now turn to the proof of the upper bound $O(n^3/\varepsilon)$.

Recall that in the (n, K)-knapsack problem we are given a sequence $a = (a_1, \dots, a_n)$ of natural numbers $a_i \leq K$, and the goal is to compute the optimal value $\mathrm{opt}(a) = \sum_{i \in I} a_i$ over all solutions I such that $\sum_{i \in I} a_i \leq K$. With every input a we associate its scaling factor $r = \varepsilon(a_1 + \cdots + a_n)/n^2$. The scaled version of a is the vector $a' = (a'_1, \dots, a'_n)$, where $a'_i = \lfloor a_i/r \rfloor$. By the scaled (n, K)-Knapsack problem we will mean the Knapsack problem with profits $(a'_1, \dots, a'_n)$ and sizes $(a_1, \dots, a_n)$, and the same capacity K.

Note that the total weight $a'_1 + \cdots + a'_n$ of every scaled input a' does not exceed $t = (a_1 + \cdots + a_n)/r = n^2/\varepsilon$. By Proposition 2, there is an argmax dynamic BP $P'(x)$ of size at most some constant times $nt = n^3/\varepsilon$ which solves the scaled (n, K)-Knapsack problem (exactly). Important here is that the sets of feasible solutions in both problems (original and scaled) are the same—only their values may differ.

The survival tests in $P'(x)$ are of the form $x_i = d$ for a natural number d. We modify the program $P'(x)$ to a program $P(x)$ by replacing all survival tests $x_i = d$ by tests $\lfloor x_i/r \rfloor = d$, where the factor $r = r(x) := \varepsilon(x_1 + \cdots + x_n)/n^2$ depends on input x. These tests are legal since in an argmax BP we allow them to depend also on the total weight $x_1 + \cdots + x_n$ of the entire input. Note that an item a_i passes the test $\lfloor x_i/r \rfloor = d$ in program P if and only if the scaled item $a'_i = \lfloor a_i/r \rfloor$ passes the test $x_i = d$ in program P'. Thus, all feasible solutions produced by the program P' on scaled inputs a' are feasible solutions for non-scaled input a produced by program P. Of course, optimal solutions for a' may not be optimal for a. Let us show that still $P(a) \geq (1 - \varepsilon)\mathrm{opt}(a)$ holds.

Let I' be an optimal solution produced by $P'(x)$ on the scaled input a', and let I be an optimal solution for input a. The solution I' is also a feasible solution for a, and is produced by $P(x)$ on input a. Since I' is optimal and I is feasible for a', we have that $\sum_{i \in I'} a'_i \geq \sum_{i \in I} a'_i$. Thus,

$$P(a) \geq \sum_{i \in I'} a_i \geq r \sum_{i \in I'} a'_i \geq r \sum_{i \in I} a'_i \geq \sum_{i \in I} a_i - r|I| \geq \mathrm{opt}(a) - rn\,.$$

Together with $rn = \varepsilon(a_1 + \cdots + a_n)/n \leq \varepsilon \cdot \max_i a_i \leq \varepsilon \cdot \mathrm{opt}(a)$, the desired lower bound $P(a) \geq (1 - \varepsilon)\mathrm{opt}(a)$ follows. □

6 Conclusion and Open Problems

In this paper we introduced a model of dynamic branching programs (dynamic BP) which captures the power of so-called "incremental" dynamic programming algorithms, and proved a matching lower bound for the Knapsack problem in this model. Still, many questions remain open.

Late rejections The first natural question is to relax the consistency conditions (i) and (ii) for dynamic BPs. Can the lower bounds in Theorem 1 be proved without the consistency condition (ii) on the decision predicates? This condition allows that rejected items can be accepted later, but it does not allow already accepted items be later rejected. That is, dynamic BPs do not have this "late rejections" feature. It turns out that this feature may substantially increase the power of dynamic BPs. In programs with this feature, a natural way to define the solution produced by an s-t path as the set of items on which the last decision was "accept".

Consider the Maximum Weight Bipartite Matching problem. Inputs are n^2 non-negative weights of the edges of $K_{n,n}$, and the goal is to compute the maximum weight of a matching. If we allow late rejections, then this problem can be solved by a dynamic BP with $O(n^3)$ wires. Moreover, the resulting BP is static, i.e., has no survival tests. This is surprising because it is proved in [11] that this problem requires priority BPs of exponential size, even if the weights are restricted to 0 and 1.

The BP itself consists of n "acceptors" followed by n "rejectors". Each acceptor corresponds to one vertex u of $K_{n,n}$ on the left side, and consists of n parallel wires between two nodes that are responsible for all n edges incident to vertex u. All decisions here are "accept". Each rejector corresponds to one vertex v of $K_{n,n}$ on the right side, and consists of n node-disjoint paths of length $n-1$. Wires of each of the paths are responsible for all but one edge incident to v. All decisions here are "reject".

Take now any s-t path is this BP. The acceptor part of this path accepts one incident edge for each node on the left part. The rejector part rejects all but one incident edge for each vertex on the right part of $K_{n,n}$. Thus, the solution produced by the entire path is a matching. On the other hand, each perfect matching will be produced by at least one s-t path, namely – by the path whose rejector part does not reject any of the edges in this perfect matching. Thus, the BP solves the Maximum Weight Perfect Matching problem. This example shows that the feature of late rejections may substantially increase the power of dynamic BP. Can non-trivial lower bounds be proved for dynamic BPs with the late rejections feature?

Null-chains Consistency condition (i) seems to be a more severe one. If we completely remove this condition then, by Proposition 1, we will land into the realm of general model of nondeterministic branching programs (NBP), where even larger than n lower bounds on the number of nodes are not know so far. The consistency condition (i) results in branching programs, known also as "null-chain-free" programs, and for them exponential lower bounds are known [19].

Although allowing such "redundant" paths, followed by none of the inputs, seems to be an overkill (why should we do this?), it is known that their presence can substantially reduce the number of nodes (number of subproblems used). For example, as shown in [17], there are boolean functions that require NBPs of exponential size, if null-chains are forbidden, but can be computed by small NBPs when null-chains are allowed. Such is, for example, the Exact Perfect Matching function, that is, a boolean function which accepts a given 0-1 $n \times n$ matrix if and only if every row and every column has exactly one 1. Without null chains, $\binom{n}{n/2} = \Omega(2^n/\sqrt{n})$ nodes are necessary, whereas already $O(n^3)$ wires are enough if null-chains are allowed: just test whether every row has at least $n-1$ 0s, and whether each column has at least one 1. Every s-t path in this program is either read once, or is inconsistent, that is, makes two contradictory survival tests $x_{ij}=0$ and $x_{ij}=1$ on the same entry (i,j) of the input matrix.

It is therefore an interesting problem to understand the role of null-chains in dynamic programming algorithms. In particular, can the presence of null-chains reduce the size of DP algorithms solving "natural" optimization problems?

Read-k dynamic BPs? Another possible relaxation of the consistency condition could be to allow inconsistent paths in a dynamic BP, but to require that along every path (be it consistent of not) each item x_i is queried at most some given number k of times. In the case of boolean functions, such programs are known as (syntactic) *read-k* branching programs, and several exponential lower bounds for them are known [23,8,17]. For branching programs computing functions $f: D^n \to \{0,1\}$ for larger domains than $D=\{0,1\}$, exponential lower bounds are known even for "semantic" read-k programs, where it is only required that along every *consistent* path one item is queried at most k times [2,5,18]. Again, it would be interesting to prove strong lower bounds for read-k dynamic branching programs solving some natural *optimization* problems.

More general survival tests In our proofs of lower bounds it was essential that the survival tests $t_e(x_i)$ made on contacts e can only depend on one single data item x_i. In some cases, it is desirable to have more general survival tests, depending on more than one data item. For example, in the interval scheduling problem a natural test is "Is the finish-time of the i-th job smaller than the start-time of the j-th job?" Can some non-trivial lower bounds be proved for dynamic BPs with more general survival tests?

Min-Plus and Max-Plus circuits The next interesting problem is to eliminate the "incremental" restriction of BPs. Such a BP is just a $(\min,+)$ or $(\max,+)$ circuit in which the use of $+$-gates is restricted: one of the two inputs must be a weight of a single item.

A prominent example of a "non-incremental" DP algorithm, which does not directly translate to a dynamic BP, is the Floyd–Warshall algorithm for the all pairs shortest path problem. As subproblems it takes $S_k(i,j)=$ the length of a shortest paths from i to j that only uses vertices $1,\ldots,k$ as inner nodes. We set $S_0(i,j)=$ the weight of the edge (i,j). The DP recursion is then $S_k(i,j)=\min\{S_{k-1}(i,j), S_{k-1}(i,k)+S_{k-1}(k,j)\}$.

This algorithm gives us a $(\min,+)$-circuit of size $O(n^3)$. On the other hand, it is known (see [1, pp. 204–206]) that the complexity (number of arithmetic operations) of this problem is of the same order of magnitude as the complexity of computing the product of two matrices over the semiring $(\min,+)$. In this latter problem, we have two $n\times n$ matrices $A=(a_{ij})$ and $X=(x_{ij})$. The goal is to compute their "product" $M=AX$ where $M=(m_{ij})$ is an $n\times n$ matrix with $m_{ij}=\min\{a_{i1}+x_{1j}, a_{i2}+x_{2j},\ldots,a_{in}+x_{nj}\}$. It is clear that n^3 additions are always enough to compute M. On the other hand, Kerr [21] showed that n^3 additions are also necessary. This implies that, in the case when only Min and Plus operations are allowed, the Floyd–Warshall all pairs shortest paths DP algorithm is optimal!

Much fewer is known about the $(\min,+)$-circuit complexity of the s-t shortest path problem. It can be shown (see Example 1 below) that the Bellman–Ford algorithm for this problem translates to a static BP with $O(n^3)$ wires. Hence, this problem can be solved by a $(\min,+)$-circuit with $O(n^3)$ gates. Does the shortest s-t path problem for n-vertex graphs require $\Omega(n^3)$ gates in $(\min,+)$-circuits? The s-t shortest path problems is only a special case of the more general all-pairs shortest paths problem, but no faster algorithms for the former problem are known.

Relation to prioritized branching programs In our model of dynamic BP, each wire is responsible for one data item x_i of the input instance $x = (x_1, \ldots, x_n)$. This responsibility is input independent, that is, does not depend on actual costs of the items. One can, however, try to relax this condition and allow each wire e to have its *responsibility function* $\mathrm{ind}_e : D^n \to [n]$. When an input $x \in D^n$ comes, the wire e is responsible for the i-th item of x if $\mathrm{ind}_e(x) = i$.

Of course, we cannot allow arbitrary responsibility functions, for otherwise any 0-1 optimization problem would be solvable by very small static BP consisting of just one path: for each input x, fix one of its optimal solutions I_x, and let the i-th wire of the path be responsible for the i-th item in I_x.

In the model of prioritized BP, introduced in [11], non-constant responsibility functions are allowed. Unlike dynamic BP, the description of this model is more "algorithmic". The model is specified by giving some set V of nodes (states), one of which is a source-node, and some of which are sink-nodes. Each sink node v is assigned a real number $\mathrm{val}(v)$. Each non-sink node has its associated total ordering $\preceq_v$ of the set D of all data items, and a transition function $g_v : D \times \{0,1\} \to V \times M$, where M is the set of all monotone real function on one argument. All these objects (values of sinks, orderings, and transition functions) are input-independent.

The actual "branching program" $P(x)$ is constructed only when an input $x \in D^n$ comes. We start at the source node, and do the following. Each non-sink node v is responsible for that item x_i in input instance x, which comes as the first in the ordering $\preceq_v$. Then we follow the two outgoing wires: 0 (reject x_i) and 1 (accept x_i). These wires go to vertices determined by transition function g_v. The transition function g_v also associates monotone functions f_0 and f_1 with the outgoing wires. We stop the construction of $P(x)$ when no new non-sink node can be reached. The *size* of a pBP algorithm is the maximum, over all inputs $x \in D^n$, of the number of nodes in $P(x)$.

Note that (unlike dynamic BP) this model is "adversarial": the program cannot see the entire input string x but, after a node v gives an ordering $\preceq_v$, the adversary must reveal the first item of x in this order. This is why lower bounds for pBP are obtained in [11] using games played by a Solver (a pBP) and an Adversary.

The value $P(x)$ on input x is computed backwards, by inductively assigning values to nodes. The value of each sink node v is the value $\mathrm{val}(v)$ assigned by the algorithm (it does not depends on x). Suppose now that the children v_0 and v_1 already have assigned values $\mathrm{val}(v_0)$ and $\mathrm{val}(v_1)$. Let f_0 and f_1 be the monotone real functions assigned (by the transition function g_v) to the wires going to v_0 and v_1. Then the value of v is defined as the maximum (or minimum, if we have a minimization problem) of $f_0(\mathrm{val}(v_0))$ and $f_1(\mathrm{val}(v_1))$. The value output by the algorithm is then the value of the source node.

A pBP is *boolean* if sinks can only have values 0 or 1. In this case, the value of each node is just an OR of values of its two children. That is, in this case all functions f_i are trivial identity functions. Using interesting probabilistic arguments, it is proved in [11] that any boolean pBP computing the perfect matching function for bipartite graphs must have exponential size (=number of nodes). More precisely, they show that, for every pBP algorithm P, there exists an input instance (a bipartite $n \times n$ graph) for which the program $P(x)$ has at least $2^{\Omega(n^{1/8})}$ nodes.

The model of pBT (prioritized branching *trees*), introduced in [3], has a restriction that the underlying graph of $P(x)$ must be a tree. But it has an additional feature (not present in pBP) that the transition function g_v as well as the ordering $\preceq_v$ may depend on the items and decision about them made along the (unique) path to the node v. In [3] it is shown that the (n, K)-knapsack problem with capacity K about $n3^n$ requires pBTs of size $\binom{n/2}{n/4}$. It is also shown that the Knapsack problem can be $(1-\varepsilon)$-approximated by a pBT of size $(1/\varepsilon)^2$, and that any such pBT must have size $(1/\varepsilon)^{1/3.17}$, if $\varepsilon \geq 2^{-cn}$ for some constant c.

It is not clear whether one of the models—dynamic BP and pBP—includes the other. Intuitively, the later model “should” subsume the power of the former, due to more general responsibility functions. It would be interesting to formally prove or disprove this. It would also be interesting to prove lower bounds for dynamic BPs themselves equipped with some non-constant (say, priority-type) responsibility functions.

Acknowledgments

I am thankful to Joshua Buresh-Oppenheim, Russell Impagliazzo, Gerhard Paseman, and Georg Schnitger for fruitful discussions.

References

1. A. Aho, J. Hopcroft, and J. Ullman: *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974. 22
2. M. Ajtai: Determinism versus non-determinism for linear time RAMs with memory restrictions. J. Comput. Syst. Sci. 65(1), 2–37 (2002). 22
3. M. Alekhnovich, A. Borodin, J. Buresh-Oppenheim, R. Impagliazzo, A. Magen: Toward a model for backtracking and dynamic programming. Comput. Complexity 20(4), 679–740 (2011). Preliminary version in: Proc. of 20-th IEEE Conference on Computational Complexity, pp. 308—322 (2005). 3, 23
4. S. Angelopoulos and A. Borodin: The power of prioritized algorithms for facility location and set cover. Algorithmica 40(4), 271–291 (2004). 3
5. P. Beame, M. Saks, X. Sun, and E. Vee: Time-space trade-off lower bounds for randomized computation of decision problems. Journal of ACM 50(2), 154–195 (2003). 22

6. R. Bellman: Combinatorial processes and dynamic programming. In: Proc. of the 10-th Symp. in Applied Math. of the AMS, pp. 24—26 (1958). 3
7. A. Bompadre: Exponential lower bounds on the complexity of a class of dynamic programs for combinatorial optimization problems. Algorithmica, 62(3-4), 659–700 (2012). 10
8. A. Borodin, A. Razborov, and R. Smolensky: On lower bounds for read-k times branching programs. Computational Complexity 3, 1–18 (1993). 22
9. A. Borodin, M. N. Nielsen, and C. Rackoff: (Incremental) prioritized algorithms. Algorithmica 37(4), 295–326 (2003). 3
10. A. Borodin, J. Boyar, and K. S. Larsen: Priority algorithms for graph optimization problems. In: Proc. of 2nd Int. Workshop on Approximation and Online Algorithms. LNCS, vol. 3351, pp. 126—139. Springer, Berlin (2005). 3
11. J. Buresh-Oppenheim, S. Davis, R. Impagliazzo: A stronger model of dynamic programming algorithms. Algorithmica 60(4), 938–968 (2011). 3, 7, 11, 21, 23
12. S. Davis, and R. Impagliazzo: Models of greedy algorithms for graph problems. Algorithmica 54(3), 269–317 (2009). 3
13. G. Hansel: Nombre minimal de contacts de fermeture necessaires pour realiser une function booleenne symetrique de n variables, C. R. Acad. Sci. 258(25) (1964), 6037–6040 (in French). 16
14. P. Helman: A common schema for dynamic programming and branch and bound algorithms. J. ACM 36(1), 97–128 (1989). 3
15. P. Helman and A. Rosenthal: A comprehensive model of dynamic programming. SIAM J. Algebr. Discrete Methods 6, 319–334 (1985). 3
16. M. Jerrum and M. Snir: Some exact complexity results for straight-line computations over semirings. J. ACM 29(3), 874–897 (1982). 10
17. S. Jukna: A note on read-k times branching programs, RAIRO Theoret. Informatics and Appl. 29(1), 75–83 (1995). 21, 22
18. S. Jukna, A nondeterministic space-time tradeoff for linear codes. Inf. Process. Lett. 109(5), 286–289 (2009). 22
19. S. Jukna: *Boolean Function Complexity: Advances and Frontiers*. Springer, 2012. 3, 21
20. R. Karp and M. Held: Finite state processes and dynamic programming. SIAM J. Appl. Math. 15, 693–718 (1967). 3
21. L. R. Kerr: The effect of algebraic structure on the computation complexity of matrix multiplications. PhD Thesis, Cornell Univ., Ithaca, N.Y. (1970). 22
22. E. I. Nechiporuk: On a Boolean function, Soviet Math. Dokl. 7(4), 999–1000 (1966). 11
23. E. A. Okolnishnikova: Lower bounds on the complexity of realization of characteristic functions of binary codes by branching programs. In: Diskretnii Analiz, 51, pp. 61–83 (Novosibirsk, 1991), in Russian. 22
24. O. Regev: Priority algorithms for makespan minimization in the subset model, Inf. Process. Lett. 84(3), 153–157 (2002). 3
25. A. Rosenthal: Dynamic programming is optimal for nonserial optimization problems. SIAM J. Comput. 11(1), 47–59 (1982). 3
26. I. Wegener: *Branching Programs and Binary Decision Diagrams*. SIAM, 2000. 3
27. G. J. Woeginger: When does a dynamic programming formulation guarantee the existence of a fully polynomial time approximation scheme (fptas)? INFORMS J. Comput. 12(1), 57–74 (2000). 3

ECCC ISSN 1433-8092

http://eccc.hpi-web.de