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

Limitations of Incremental Dynamic Programs

Stasys Jukna*†‡§

April 17, 2012

Abstract

We consider so-called "incremental" dynamic programming algorithms, and are interested in the number of subproblems produced by them. The standard 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.

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

Much work has been conducted along these lines. Just to name a few, Hausmann and Korte [18] showed that there is no polynomial query approximative algorithm for the optimization problem over a general independence system which has a better worst-case behavior than the greedy algorithm. A query algorithm can ask in each step whether some solution is a feasible solution. Chvátal [15] defined a general scheme for branch-and-bound algorithms and proved exponential lower bounds to 0-1 Knapsack in this model. This last result was extended by Chung et al. [14] to integer Knapsack, and to a more general model capturing some aspects of

*Universität Frankfurt, Institut für Informatik, Robert-Mayer Str. 11-15, Frankfurt am Main, Germany.
†Afiliated with Vilnius University, Institute of Mathematics, Akademijos 4, Vilnius, Lithuania.
‡Email: jukna@thi.informatik.uni-frankfurt.de
§Research supported by the DFG grant SCHN 503/5-1.

ISSN 1433-8092

dynamic programming in [26]. Exponential lower bounds for 0-1 Knapsack problem in a class of query algorithms were proved by Hausmann et al. [19]. More generally, Khana et al. [29] formalized various local search paradigms, and Arora et al. [5] looked at general methods for generating linear relaxations for boolean optimization problems.

In this paper we focus on the dynamic programming paradigm. There were many attempts to formalize this paradigm, and various refinements were obtained [8, 27, 36, 21, 20, 37, 3, 13], just to mention some of them. In particular, Woeginger [37] introduces a model of so-called "DP-simple" algorithms in which only problems susceptible to a dynamic programming fully polynomial time approximation schemes can be expressed. More general, but still tractable models of so-called "prioritized branching trees" (pBT) and "prioritized branching programs" (pBP) were introduced, respectively, by Alekhnovich et al. [3] and Buresh-Oppenheim et al. [13]. These models are based on the framework of "priority algorithms" introduced by Borodin et al. [11], and subsequently studied and generalized by [4, 3, 12, 16, 35]; 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 power of *backtracking* and *dynamic programming*. In particular, already pBTs subsume the power of DP-simple algorithms. In [3] it is shown that the Knapsack problem requires pBTs of exponential size, whereas in [13] it is shown that detecting the presence of a perfect matching in bipartite graphs requires even pBPs of exponential size.

Our model Just like 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 change the underlying boolean semiring by other semirings.

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 do 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 $S(v)$ 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

$$S(v) = \max\{S(u_1) + c_1, \ldots, S(u_k) + c_k\}$$

where c_i is the cost 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 $c_i = 0$. That is, the cost of the problem solution is the subproblem cost plus a cost that is directly attributable to the

decision about the item itself. 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 can have a more general form

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

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 cost 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 the n-dimensional Knapsack problem with an integer knapsack capacity K, a problem instance is a sequence of n pairs $a_i = (p_i, w_i)$ of natural numbers; p_i is the "profit", and w_i the "weight" of the i-th item. We will assume that $w_i \le K$ 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:

$$\begin{array}{ll} \text{maximize} & p(S) := \sum_{j\in S} p_j \\ \text{subject to} & w(S) := \sum_{j\in S} w_j \le K \text{ and } S \subseteq [n] = \{1, \ldots, n\} \end{array} \tag{1}$$

A standard DP algorithm for the maximization 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:

$$S(i, j) = \max\{S(i-1, j), S(i-1, j-w_i) + p_i\}\,. \tag{2}$$

The value of the optimal solution is $\mathrm{Opt}(a) = S(n, K)$. Note that this algorithm is incremental: when going from subproblem $S(i-1, j-w_i)$ 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 weight by keeping the total profit over some threshold K:

$$\begin{array}{ll} \text{minimize} & w(S) \\ \text{subject to} & p(S) \ge K \text{ and } S \subseteq [n] \end{array} \tag{3}$$

In this case we can consider subproblems $T(i, j) =$ the minimum weight using only items $1, \ldots, i$ and a profit of at least j. The DP algorithm is then described by the recursion:

$$T(i, j) = \min\{T(i-1, j), T(i-1, j-p_i) + w_i\}\,. \tag{4}$$

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 3.1). Our next result is that, even if "redundant" paths in the subproblem-graph are allowed, at least $\Omega(nK \log K)$ wires (relations between the subproblems) are necessary to solve the minimization Knapsack problem (Theorem 4.1). It remains open whether $\Omega(nK)$ *nodes* are necessary in this more general model.

The proofs of our lower bounds are not involved: they use standard cut-and-paste arguments for branching programs together with some combinatorics. So, our contribution is rather conceptional: we isolate a natural subclass of DP algorithms where some non-trivial lower bounds can be proved. 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 when 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 5.

2 Dynamic Branching Programs

We consider 0-1 optimization problems. In each such problem we have some finite set D of *data items* together with their *cost function* $\text{cost} : D \to \mathbb{R}$. A *problem instance* is a sequence $a = (a_1, \ldots, a_n) \in D^n$ of data items. A *solution* for instance a is a binary vector $\delta = (\delta_1, \ldots, \delta_n) \in \{0,1\}^n$, where $\delta_i = 1$ means that the i-th item a_i is accepted; we will often view a solution δ as the set $I = \{i \in [n] \colon \delta_i = 1\}$ of accepted items. There is also some *constraint predicate* $C : D^n \times \{0,1\}^n \to \{0,1\}$. A solution δ is a *feasible* solution for an instance a if and only if $C(a, \delta) = 1$. The goal is to maximize (or minimize) the total cost $\sum_{i=1}^{n} \delta_i \cdot \text{cost}(a_i)$ over all feasible solutions δ for a. The value of an optimal solution for a given instance a is denoted by $\text{Opt}(a)$. For definiteness, we set $\text{Opt}(a) = 0$ if a has no feasible solutions.

For example, in the n-dimensional Knapsack problem with knapsack of capacity K, items are pairs of natural numbers (profit, weight), whose costs are equal to their profits. The constraint predicate has the form "the weight of accepted items does not exceed K", and $\text{Opt}(a)$ is the biggest total profit of accepted items.

Definition of dynamic branching programs A *static branching program* (static 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, joining the same pair of nodes are allowed. If there are no multiple wires and if (after removing the target node t) the underlying graph of a program

is a tree, then we call it a *static branching tree* (static BT). By the *size* of a program we will mean the number of inner nodes (that is, we do not count s and t as 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) and has a *decision predicate* $\delta_e : D \to \{0,1\}$ about the item it is responsible for: the item x_i is accepted at e if and only if $\delta_e(x_i) = 1$.

In a *dynamic branching program* (dynamic BP), each contact is also allowed to have its *survival test* $t_e : D \to \{0,1\}$. The meaning of this test is the following: when input $a \in D^n$ comes, the contact e responsible for the i-th variable is removed from the program if $t_e(a_i) = 0$ (the contact has not passed the survival test on input a), and e remains intact if $t_e(a_i) = 1$. Thus, being "dynamic" means that the structure of the program may depend on the actual problem instance.

A path p is *consistent* with a given input string $a \in D^n$, if this input passes all survival tests along p; in this case we also say that p is a *computation* on input a. We require that a BP (be it static or dynamic) satisfies the following two "consistency conditions".

Consistency conditions If a wire e_1 precedes a wire e_2 on some path, and if *both wires are responsible for the same variable*, then we require that

(i) $t_{e_1} \le t_{e_2}$ (once survived, always survived);

(ii) $\delta_{e_1} \le \delta_{e_2}$ (once accepted, always accepted).

The first condition (i) requires that if an item passed the first survival test along a path, then it cannot "die" later on a wire responsible for this item. The second condition (ii) requires that if an item is accepted, then it cannot be rejected later by a wire responsible for this item. 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 1. 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 $a \in D^n$ comes, each contact e responsible for the i-th variable x_i receives its weight $w_e(a)$ which is defined as

$$w_e(a) := \delta_e(a_i) \cdot \mathrm{cost}(a_i)\,.$$

That is, if the wire e is responsible for the i-th item, then $w_e(a) = 0$ if this item is rejected, and $w_e(a) = \mathrm{cost}(a_i)$ if this 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.

Let $\mathrm{Paths}(a)$ denote the set of all s-t paths in the program that are consistent with a given input $a \in D^n$. The *weight*, $w_p(a)$, of a path $p \in \mathrm{Paths}(a)$ on an input $a \in D^n$ is defined as the sum $w_p(a) := \sum_{e \in p} w_e(a)$ of the weights of its wires. The value $P(a)$ of the program on

the input a is the maximum (or a minimum, if we have a minimization problem) weight of all paths consistent with a:

$$P(a) = \max\Big\{ \sum_{e\in p} w_e(a) \colon p \in \mathrm{Paths}(a)\Big\}.$$

That is, when an input $a \in D^n$ comes, we first remove all wires e for which $t_e(a) = 0$, and then compute the value $P(a)$ as the maximum weight of an s-t path in the remaining sub-program of P. In other words, $P(a)$ is the value, $\mathrm{Val}(t)$, of the target node t defined inductively as follows. The start node s always has a zero value, $\mathrm{Val}(s) = 0$. 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 a; that is, each wire $e_j = (v_j, v)$ is either a rectifier (an unlabeled wire), or is a contact such that $t_{e_j}(a) = 1$. Then

$$\mathrm{Val}(v) = \max\{\mathrm{Val}(v_1) + w_{e_1}(a), \ldots, \mathrm{Val}(v_k) + w_{e_k}(a)\}. \tag{5}$$

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 my still substantially decrease the total number of wires (see Remark 7).

Remark 2. If a contact $e = (v_j, v)$ is responsible for the item x_i, then this item is accepted at e only if both $t_e(x_i) = 1$ and $\delta_e(x_i) = 1$ hold. So, one could wonder why not to assign to e just *one* predicate $\Delta_e(x_i) = t_e(x_i) \wedge \delta_e(x_i)$ and define the weight of the contact e as $w'_e(a) := \Delta_e(a_i) \cdot \mathrm{cost}(a_i)$? The reason for not doing this is explained by the way (5) the values at nodes are computed: out of all wires entering the node v, only survived wires can contribute to $\mathrm{Val}(v)$. That is, if the wire e survived the test ($t_e(a) = 1$), then it contributes either $\mathrm{Val}(v_j) + w_e(a)$ or $\mathrm{Val}(v_j)$ to the maximum (5). But if e does not survive the test ($t_e(a) = 0$), then it contributes nothing to $\mathrm{Val}(v)$.

The *solution* produced by a path $p \in \mathrm{Paths}(a)$ on input a is the set

$$I_p(a) = \{i \colon \delta_e(a_i) = 1 \text{ and } e \in p\}$$

of items accepted along p. A program P *solves* a given optimization problem *on a subset* $A \subseteq D^n$ of problem instances, if for every $a \in A$ the following holds:

1. For every input $a \in A$ there is a path in $\mathrm{Paths}(a)$ whose produced solution is optimal for a.

2. For every path $p \in \mathrm{Paths}(a)$, the solution $I_p(a)$ produced by p on input a is a feasible solution for a.

A program solves the problem if it solves it on the set $A = D^n$ of all problem instances.

Example 1 (Knapsack problem). The DP algorithm (2) for the maximization Knapsack problem (1) can be turned into an oblivious read-once dynamic BP with nK nodes as follows.

As nodes we take the subproblems $S(i, j)$ as defined in (2). For every $k = 1, \ldots, j$, there is a contact $S(i-1, k) \to S(i, j)$ responsible for the i-th item $x_i = (p_i, w_i)$. 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 $w_i = j - k$. The decision of each such contact e is $\delta_e(x_i) \equiv 1$ (always accept),

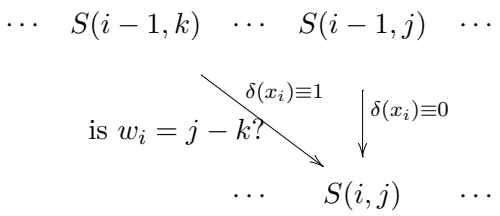


Figure 1: A fragment of a dynamic branching program for the Knapsack problem.

and hence, the weight of each such contact is $\text{cost}(x_i) = p_i$. Thus, when an input a with $a_i = (p_i, w_i)$ comes, only two contacts $S(i-1,j) \to S(i,j)$ and $S(i-1,j-w_i) \to S(i,j)$ entering the node $S(i,j)$ will survive (see Fig. 1). 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 $w_1 \le j$. 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 $\mathcal{O}(nK^2)$ wires. The dynamic BP for the *minimization* Knapsack problem (3) is similar.

More examples of dynamic BPs for other optimization problems are given in Appendix A. At this point, we only stress that in all these examples the DP algorithm *itself* gives us a dynamic BP without *any* changes in the algorithms; only the form of input instances has to be slightly modified, in some cases. This indicates that the model of dynamic BPs is apparently the "right" way to represent incremental DP algorithms.

Remark 3. Note that we have *one* program for *all* problem instances $a \in D^n$. But we do not require that *every* optimal solution for a 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 introduced in [9]. Here one associates with each problem instance a digraph whose wires keep some weights in such a way that there is a 1-to-1 between s-t paths and *all* feasible solutions for that instance. The minimal weight of a path must then be the value of an optimal solution for the given instance. Thus, this is not a "computational model" in a usual sense since there are no " partial computations" and no "partial solutions" in this model—only s-t paths have a "meaning". This is more a *class* of problems whose all feasible solutions may be encoded as s-t paths.

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 $\mathrm{cost}(0) = \mathrm{cost}(1) = 1$, 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 many other formalizations of DP algorithms, including the model of prioritized branching programs invented in [13]. We now argue that this is not a coincidence: without any restrictions on the survival tests the resulting model would be too powerful.

Proposition 1. *If survival tests can be arbitrary functions $t_e : D^n \to \{0,1\}$, then any n-dimensional* 0-1 *optimization problem can be solved by an oblivious read-once dynamic BP of size n.*

Proof. To construct a desired dynamic BP, 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}. 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 $t_{i,1}(x) = 1$ if and only if $i \in I_x$, and $t_{i,0}(x) = 1$ 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. □

We now argue that the consistency condition (i) is essential: even if we would require all survival tests be local (of the form "is $x_i = d$?" for a data item $d \in D$), and even if we would allow *constant* decisions ($\delta_e \equiv 0$ or $\delta_e \equiv 1$), the resulting model is still too powerful—at least as powerful as unrestricted branching programs! 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 remain the lower bound $\Omega(n^{3/2}/\log n)$ proved by Nechiporuk [32]. 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 2. *Without the restrictions (i) on survival tests, the model of dynamic BPs is at least as powerful as the model of 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 $\mathrm{cost}(a_i) = 1$. 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 $\mathrm{Opt}(a) = \max \sum_{i=1}^{n} \delta_i \cdot \mathrm{cost}(a_i)$ 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 relabeling 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 *simple Knapsack problem* where the profit of each item is equal to its weight. In this problem, which we call the (n, K)-*knapsack problem*, input instances are sequences $a = (a_1, \ldots, a_n)$ of integers in $\{0, 1, \ldots, K\}$, and 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 $\sum_{i \in I} a_i \leq K$; subsets I satisfying this last inequality are feasible solutions for instance a.

Let $\mathrm{Size}(n, K)$ denote the smallest size of a dynamic branching program solving the (n, K)-knapsack problem on the set of all inputs a with $\mathrm{Opt}(a) = K$. By Example 1, $\mathrm{Size}(n, K) \leq nK$ holds even in the restricted class of oblivious read-once dynamic branching programs. We now show that one cannot expect to do much better, even if neither the order nor the number of tests is restricted.

Theorem 3.1. *If* $K \geq 3n$ *then* $\mathrm{Size}(n, K) \geq \frac{1}{2}nK$.

To prove the theorem, we first establish some properties of 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 $x_1 + \cdots + x_k = n$. It is well known that there are $\binom{n+k-1}{n}$ such partitions. By a *test* we mean a pair (S, b), where $S \subseteq [k]$, and $0 \leq b \leq n$ is an integer. Such a test is *legal* if $0 \neq |S| \leq k - 1$. 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 do 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 $\tau(n) \leq n + 1$ is easy: already tests $(\{1\}, b)$ with $b = 0, 1, \ldots n$ will do the job. To prove the lower bound $\tau(n) \geq n + 1$, 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 with support S. The remaining tests (T, b) with $T \not\subseteq S$ can be modified in such a way that they cover all partitions of $n-c$ into $k-|S|$ 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. If some test (T, b) participates in covering any of such partitions, and if $T \cap S \neq \emptyset$, then replace (T, b) by the test $(T \setminus S, b')$ where $b' = b - \sum_{i \in S \cap T} a_i$. By induction hypothesis, there must be at least $n - c + 1$ such 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. $\tau^+(n) \geq \tau(n-k)$.

Proof. There is a 1-1 correspondence between positive partition 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 $b \geq |S|$ if the test covers at least one positive partition x, because then $\sum_{i \in S} x_i = b$ and all $x_i \geq 1$. 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 $n-k$. □

of Theorem 3.1. Take a dynamic branching program $P = (V, E)$ solving the (n, K)-knapsack problem on the set $A \subseteq [K]^n$ of all positive partitions of K. Thus, for every instance $a \in A$, the set $I = [n]$ is the only optimal solution, and its value is K. Important, however, is that on none of the remaining instances, the program can produce an infeasible solution.

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, along each optimal path exactly n items must be accepted.

Fix now an integer $r \in \{1, \ldots, n-1\}$, 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 $I_a = I_{p_a}(a) \subseteq [n]$ be the set of items accepted along the first segment p_a, and $J_a = I_{q_a}(a) \subseteq [n]$ the set of items accepted along the last segment of this path; hence, $I_a \cap J_a = \emptyset$, $|I_a| = r$ and $|J_a| = n - r$.

Take now an instance $b \in A$ whose optimal path was stopped at the same node as that of instance a.

Claim 1. $I_a \cap J_b = \emptyset$ and $w_{p_a}(a) = w_{p_b}(b)$.

Proof. Let $p = p_a$ and $q = q_b$, and define the combined input $c \in D^n$ by: $c_i = a_i$ if the i-th variable is queried along p, and $c_i = b_i$ otherwise. We know that the input a passes all survival tests along the path p, and input b passes all survival tests along the path q. If none of the variables queried along p is queried again along q, then the combined path (p, q) is clearly consistent with the combined input c. If some variable x_j is queried at some wire $e_1 \in p$ and at some wire $e_2 \in q$, then the consistency condition (i) implies that $1 = t_{e_1}(a_j) = t_{e_1}(c_j) \leq t_{e_2}(c_j)$. Thus, the path (p, q) is consistent with c also in this case.

Now assume that $j \in I_p(a) \cap I_q(b)$. Then there must be a wire $e_1 \in p$ and a wire $e_2 \in q$ such that $\delta_{e_1}(a_j) = \delta_{e_2}(b_j) = 1$. Since $c_j = a_j$ for all $j \in I_p(a)$, the consistency condition (ii) implies $1 = \delta_{e_1}(a_j) = \delta_{e_1}(c_j) \leq \delta_{e_2}(c_j)$. Thus, along the combined path (p, q), the j-th item c_j of c is accepted at least two times, implying that the solution produced by the path (p, q) on input c is *not* a feasible solution for c, a contradiction. Thus, $I_a \cap J_b = \emptyset$.

To show $w_{p_a}(a) = w_{p_b}(b)$, assume that $w_{p_a}(a) > w_{p_b}(b)$, and consider the same combined input $c \in D^n$ as above. Since $w_{p_a}(a) = \sum_{i \in I_a} a_i$ and $c_i = a_i$ for all $i \in I_a$, we have that $w_{p_a}(c) \geq w_{p_a}(a)$. Since $I_a \cap J_b = \emptyset$, we also have that $w_{q_b}(c) \geq w_{q_b}(b)$. The combined path (p_a, q_b) is consistent with the combined input c, but its weight is

$$w_r(c) = w_{p_a}(c) + w_{q_b}(c) \geq w_{p_a}(a) + w_{q_b}(b) > w_{p_b}(b) + w_{q_b}(b) = K\,.$$

Thus, the combined path produces an infeasible solution for the instance c, a contradiction. □

Let $V_r \subseteq V$ 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. Let also $A_v \subseteq A$ 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.

Claim 2. For every node $v \in V_r$ there exist a subset $S_v \subseteq [n]$ of size $|S_v| = r$ and an integer $b_v \in [K]$ such that $\sum_{i \in S_v} a_i = b_v$ holds for all $a \in A_v$.

Proof. By Claim 1, we know that $I_a \cap J_b = \emptyset$ for all $a, b \in A_v$. Since $|I_a| = r$ and $|J_a| = n - r$ for every $a \in A_v$, this implies that $I_a = I_b = S_v$ for all $a, b \in A_v$. Thus along the first segments p_a of optimal paths of all inputs $a \in A_v$ the same set $S_v \subseteq [n]$ of $|S_v| = r$ items is accepted. Furthermore, Claim 1 implies that all the weights $w_{p_a}(a) = \sum_{i \in S_v} a_i$ of these paths are the same. □

We can now finish the proof of Theorem 3.1 as follows. Our set A of inputs is the set of positive partitions of K into n blocks. By Claim 2, for every $r = 1, \ldots, n-1$, all these partitions can be covered by $|V_r|$ tests (S_v, b_v) for $v \in V_r$. Together with Lemma 2, this implies that $|V_r| \geq \tau^+(K) \geq K - n + 1$. Since this holds for every $r = 1, \ldots, n-1$, the total number $|V|$ of nodes in our program must be $|V| \geq (n-1)(K-n+1) = Kn - K - (n-1)^2$, which is $\geq \frac{1}{2}nK$ for $K \geq 3n$, as desired. □

4 Lower Bound for General Dynamic Programs

We now consider dynamic branching programs where only equality tests "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 $x_i = d$?" and "is $x_i = d'$?" for $d \neq d'$ may be made. We, however, assume that there are no rectifiers, that is, every contact *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 like in the case of maximization Knapsack problem, the standard DP algorithm 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 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). Let $\mathrm{Wires}(n, K)$ denote the smallest number of wires in a general dynamic BP solving this problem. We already know (see Example 1) that $\mathrm{Wires}(n, K) \leq nK^2$ holds even in the class of oblivious read-once dynamic programs.

Theorem 4.1. $\mathrm{Wires}(n, K) = \Omega(nK \log K)$.

Proof. Let $P(x_1, \ldots, x_n)$ be a general dynamic BP solving the minimization Knapsack problem on the set A. We assume that the number n of items as well as the capacity K are even numbers. To prove the lower bound $\mathrm{Wires}(n, K) = \Omega(nK \log K)$ on the number of contacts in P, it is enough to show that, for every $i = 1, \ldots, n/2$, the program P must contain at least $\Omega(K \log_2 K)$ 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 $\Omega(K \log_2 K)$.

Our proof will be based on a classical result of Hansel [17] that any monotone contact scheme computing the threshold-2 function $\mathrm{Th}_2^m(x_1, \ldots, x_m)$ 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.

With some abuse of notation, 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 a if an only if $\sum_{i=1}^n a_i > K$.

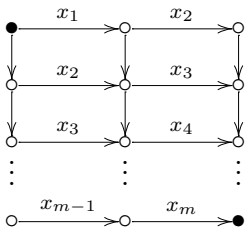
From P we obtain a dynamic program P' depending only on x_1 and x_2 by removing from P all wires making tests $x_i = d$ for $d \neq 0$ and $i \geq 3$, and by contracting all wires that make tests $x_i = 0$ for $i \geq 3$. Thus, the resulting program $P'(x_1, x_2)$ accepts a vector $(u, v) \in D^2$ if and only if the original program P accepts the vector $(u, v, 0, \ldots, 0)$, which happens if and only if $u + v > K$.

Now we turn the dynamic program $P'(x_1, x_2)$ into a monotone contact scheme P'' as follows. Let $S = \{K/2 + 1, \ldots, K\}$, and take $m = |S|$ boolean variables y_u, one for each element $u \in S$. Remove from P' all decision predicates, as well as all tests $x_1 = d$ and $x_2 = d$ for $d \notin S$. Further, replace all tests $x_1 = d$ and $x_2 = K - d$ for $d \in D$ by the test $y_d = 1$. By Hansel's theorem, it remains to show that the obtained monotone contact scheme P'' computes the threshold-2 function $\mathrm{Th}_2^m(y_u \colon u \in S)$. Since the scheme P'' 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 $b \in \{0, 1\}^m$ with one or two 1s.

Case 1: Vector b has exactly two ones in some positions $u \neq v \in S$. Consider the input $a = (u, v, 0, \ldots, 0)$ to P. Since $u + v > K$ and $u, v \leq K$, $I = \{1, 2\}$ is a solution (in fact, an optimal solution) for a. Hence, there must be an s-t path p in P along which only the tests $x_1 = u$ and $x_2 = v$ and perhaps some of the tests $x_i = 0$ for $i \geq 3$ are made. Thus, in program P'', the path p turns to the s-t path p'' along which only tests $y_u = 1$ and $y_v = 1$ are made. Hence, $P''(b) = 1$, as desired.

Case 2: Vector b has only one 1 in some position $u \in S$. Suppose that $P''(b) = 1$. Then there is an s-t path p'' in P'' along which only tests $y_u = 1$ are made. In P' this path has only tests $x_1 = u$ and/or $x_2 = v$ where $v = K - u$. But since $u + v = K$ is not larger than K, the input $a = (u, v, 0, \ldots, 0)$ has no feasible solution, implying that none of the s-t paths in P can be consistent with a, and hence, none of the s-t paths in $P'(x_1, x_2)$ can be consistent with the input (u, v). Hence, our assumption that P'' accepts vector b was wrong. □

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 $\mathcal{O}(\log m)$ complete bipartite graphs, and hence, Th_2^m *can* be computed by a monotone contact scheme with $\mathcal{O}(\log m)$ nodes.

5 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.

The first natural question is to relax the consistency conditions (i) and (ii) for dynamic BPs. Although all incremental DP algorithms we know can be simulated without any loss in efficiency by BPs satisfying these conditions, it would be interesting to relax any one of them. If we fail in proving strong lower bounds, this would be a serious indication that the current dynamic programming paradigm can be extended to a more powerful one.

Problem 1. Can the lower bounds in Theorem 3.1 be proved without the consistency condition (ii) on the decision predicates, that is, when already accepted items are allowed to be rejected later.

Null-chains Consistency condition (i) seems to be a more severe one. If we completely remove this condition then, by Proposition 2, 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 long known (see, e.g. [34, 30, 22, 25]). The absence of null-chains (paths that are consistent with none of the input strings) allows one to use "cut-and-paste" arguments to show that small programs must make errors. Strong lower bounds for NBP are also know when null-chains *are* allowed, but there is some restrictions on their structure [22].

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 [23], 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 $\mathcal{O}(n^3)$ 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 are made.

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 [33, 10, 22]. 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, 7, 24]. It would be interesting to prove strong lower bounds for read-k dynamic branching programs solving some natural *optimization* problems.

Dynamic circuits The next interesting problem is to eliminate the "incremental" restriction of BPs. This leads to the model of *dynamic circuits* over semirings like $(+, \min)$ (or $(+, \max)$). Such a circuit consists of fanin-2 Plus and Min gates (or Plus and Max gates). The size of a circuit is the total number of gates.

In a *dynamic circuit* we again allow survival tests on wires. Thus, when an input $x \in D^n$ comes, some wires in a dynamic circuit will disappear, and then the circuit computes its value in a standard manner. Of course, we cannot allow the survival tests be arbitrary (see Proposition 1). So, as in the case of dynamic BPs, let us assume that each of these tests can only depend on one input variable x_i.

It is clear that every dynamic BP with W wires can be transformed into a dynamic circuit $(+, \max)$-circuit with $\mathcal{O}(W)$ fanin-2 gates. Thus, Example 1 implies that $\mathcal{O}(nK^2)$ gates are enough to solve the Knapsack problem by a dynamic $(+, \max)$-circuit.

Problem 2. How many gates are necessary to solve the Knapsack problem by a dynamic $(+, \max)$-circuit?

Shortest path problems In the *all pairs shortest path* problem we are given a weighting of the edges of a complete directed graph on n vertices, and want to compute the weights of a shortest paths between all pairs of vertices. In the *s-t shortest path* problem we only want to compute the weight of a shortest path from s to t.

A prominent example of a DP algorithm that is *not* incremental 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 solution 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 static $(+,\min)$-circuit of size $\mathcal{O}(n^3)$. 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 [28] showed that n^3 additions are also necessary. This implies that, in the class of static $(+,\min)$-circuits, 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. Lyons [31] proved some tight bounds on the size of $(+,\min)$-circuits solving the shortest s-t path in two-layered acyclic digraphs. Namely, given a real $n \times n$ matrix A and two real vectors x and y of length n, we want to compute the value of $x^\top Ay$ over the semiring $(+,\min,0,\infty)$. This can be viewed as the shortest s-t path problem in a "complete two-layered digraph": we have the start vertex s, the target vertex t, and two n-element sets of intermediate vertices L and R with edges going from each $u \in L$ to each $v \in R$; we also have edges from s to all L, and edges from all R to t. The matrix A gives a weighting of the middle edges $L \times R$, whereas x gives the weights of edges leaving s, and y the weights of edges entering t. What Lyons [31] proves is that the smallest number of $+$-gates in any $(+,\min)$-circuit computing $x^\top Ay$ is equal to $n^2 + n$, and the smallest number of min-gates in such a circuit is n^2.

Problem 3. Does the shortest s-t path problem for general n-vertex graphs requires $\Omega(n^3)$ gates in $(+,\min)$-circuits?

It can be shown (see Example 5 below) that the Bellman–Ford algorithm for this problem can be solved by a static BP with $\mathcal{O}(n^2)$ nodes and $\mathcal{O}(n^3)$ wires. So, the following special case of Problem 3 naturally arises.

Problem 4. Does the shortest s-t path problem for n-vertex graphs requires $\Omega(n^3)$ wires in static branching programs?

Acknowledgments

I am thankful to Gerhard Paseman for a hint leading to a short proof of Lemma 1, and to 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.

[2] M. Ajtai, Determinism versus non-determinism for linear time RAMs with memory restrictions, J. Comput. Syst. Sci. 65(1), 2–37 (2002).

[3] M. Alekhnovich, A. Borodin, J. Buresh-Oppenheim, R. Impagliazzo, A. Magen, Toward a model for backtracking and dynamic programming, in: Proc. of 20-th IEEE Conference on Computational Complexity, pp. 308—322 (2005).

[4] S. Angelopoulos and A. Borodin, The power of prioritized algorithms for facility location and set cover, Algorithmica 40(4), 271–291 (2004).

[5] S. Arora, B. Bollobás, L. Lovász, and I. Tourlakis, Proving integrality gaps without knowing the linear program, Theory of Comput. 2(1), 19–51 (2006).

[6] E. Balas, New classes of efficiently solvable generalized Traveling Salesman Problems, Annals of Oper. Res. 86, 529–558 (1999).

[7] 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).

[8] R. Bellman, Combinatorial processes and dynamic programming, in: Proc. of the 10-th Symp. in Applied Math. of the AMS, pp. 24—26 (1958).

[9] A. Bompadre, Exponential lower bounds on the complexity of a class of dynamic programs for combinatorial optimization problems, Algorithmica, (2010), DOI: 10.1007/s00453-010-9475-0.

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

[11] A. Borodin, M. N. Nielsen, and C. Rackoff, (Incremental) prioritized algorithms, Algorithmica 37(4), 295–326 (2003).

[12] 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 (WAOA, Bergen, Norway, September 14-16, 2004), Springer Lect. Notes in Comput. Sci., Vol. 3351, 126—139 (2005).

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

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

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

[16] S. Davis, and R. Impagliazzo, Models of greedy algorithms for graph problems, Algorithmica 54(3), 269–317 (2009).

[17] 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).

[18] D. Hausmann and B. Korte, Lower bounds on the worst-case complexity of some orackle algorithms, *Discrtete Math.* 24, 261—276 (1978).

[19] D. Hausmann, R. Kannan, and B. Korte, Exponential lower bounds on a class of Knapsack algorithms, Math. of Operations Research, 6(2) (1981), 225–232.

[20] P. Helman, A common schema for dynamic programming and branch and bound algorithms, J. ACM 36(1), 97–128 (1989).

[21] P. Helman and A. Rosenthal, A comprehensive model of dynamic programming, SIAM J. Algebr. Discrete Methods 6, 319–334 (1985).

[22] S. Jukna, The effect of null-chains on the complexity of contact schemes, in: Proc. of FCT'89, Springer Lect. Notes in Comput. Sci. vol. 380, pp. 346–356 (1989).

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

[24] S. Jukna, A nondeterministic space-time tradeoff for linear codes, Inf. Process. Lett. 109(5), 286–289 (2009).

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

[26] S. Jukna and G. Schnitger, Yet harder knapsack problems, Theoret. Comput. Sci. 412, 6351–6358 (2011).

[27] R. Karp and M. Held, Finite state processes and dynamic programming, SIAM J. Appl. Math. 15, 693–718 (1967).

[28] L. R. Kerr, The effect of algebraic structure on the computation complexity of matrix multiplications, PhD Thesis, Cornell Univ., Ithaca, N.Y., 1970.

[29] S. Khanna, R. Motwani, M. Sudan, and U. V. Vazirani, On syntactic versus computational views of approximability, SIAM J. Comput. 28(1), 164–191 (1998).

[30] S. E. Kuznetsov, The influence of null-chains on the complexity of contact circuits, in: Probabilistic Methods in Cybernetics, vol. 20 (Kazan, 1984) (in Russian.)

[31] A. Lyons, Exact complexity results and polynomial time algorithms for derivative accumulation, Preprint ANL/MCS-P1700-1209, Math. and CS Division, Argonne Nat. Lab., December 2009.

[32] E. I. Nechiporuk, On a Boolean function, Soviet Math. Dokl. 7(4), 999–1000 (1966).

[33] 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.

[34] A. K. Pulatov, Lower bounds on the complexity of realization of characteristic functions of group codes by π-schemes, in: Combinatorial-Algebraic Methods in Applied Mathematics, pp. 81–95 (Gorkii, 1979), in Russian.

[35] O. Regev, Priority algorithms for makespan minimization in the subset model, Inf. Process. Lett. 84(3), 153–157 (2002).

[36] A. Rosenthal, Dynamic programming is optimal for nonserial optimization problems, SIAM J. Comput. 11(1), 47–59 (1982).

[37] 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).

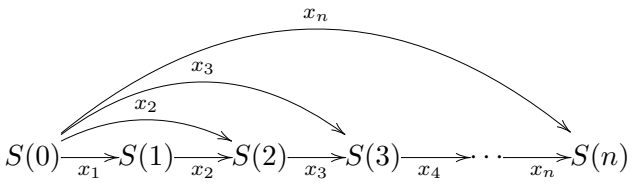


Figure 2: A static branching program for the Maximum Value Contiguous Subsequence problem. All decisions are "accept". Both wires entering the i-th node $S(i)$ are responsible for the i-th item x_i.

Appendix: More examples of dynamic branching programs

Let us first show that every 0-1 optimization problems *can* be solved by a dynamic BP.

Proposition 3. *Every n-dimensional* 0-1 *optimization problem over a set D of data items can be solved by an oblivious dynamic branching tree of size at most* $(2|D|)^{n+1}$.

Proof. Fix an n-dimensional 0-1 optimization problem whose data items come from some finite set D of size $|D| = m$. Take a $2m$-ary tree T of depth n. We let all $2m$ wires leaving a vertex v of depth $i-1$ be responsible for the i-th item x_i. For each $d \in D$ there are two contacts at which the same survival test "is $x_i = d$?" and two decisions $\delta_e \equiv 0$ and $\delta_e \equiv 1$ are made. Thus, for every sequence $a \in D^n$ of data items, and for every sequence $\delta \in \{0,1\}^n$ of decisions about them there is a *unique* path to a leaf of T which is consistent with a and all decisions are consistent with δ. We can now fix an optimal decision sequence for each $a \in D^n$, and put rectifiers (unlabeled wires from the corresponding m^n leaves to the target node. The total number of nodes in T is at most $(2m)^{n+1}$. □

The trivial upper bound in Proposition 3 is very "pessimistic": many optimization problems can be solved by much smaller dynamic BPs. We illustrate this by several examples.

Example 2 (Maximum Value Contiguous Subsequence). We are given a sequence $x_1, \ldots, x_n$ 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 $\text{cost}(x_i) = x_i$. Feasible solutions are contiguous intervals $I = \{i, i+1, \ldots, j\}$, 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 solution 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 as follows (see Fig. 2). The nodes are $S(1), \ldots, S(n)$ and one special node $S(0)$. The nodes form a path $S(0) \to S(1) \to S(2) \to \ldots \to S(n)$ consisting of contacts, each making a trivial decision $\delta_e \equiv 1$ (always accept). The contact $S(j-1) \to S(j)$ is responsible for item x_j, and hence, has weight x_j. There are also contacts from $S(0)$ to every node $S(j)$, $j = 2, \ldots, n$, each of weight x_j. This program is not oblivious, but is read-once and is static. It has $n+1$ nodes.

Example 3 (Longest Increasing Subsequence). A problem instance is a sequence $x = (x_1, \ldots, x_n)$ of integers, and the goal is to find a largest subset $I = \{i_1 < i_2 < \ldots < i_k\}$ of indices such that $x_{i_1} \le x_{i_2} \le \ldots \le x_{i_k}$. Each instance can be viewed as a weighting of the edges of the complete

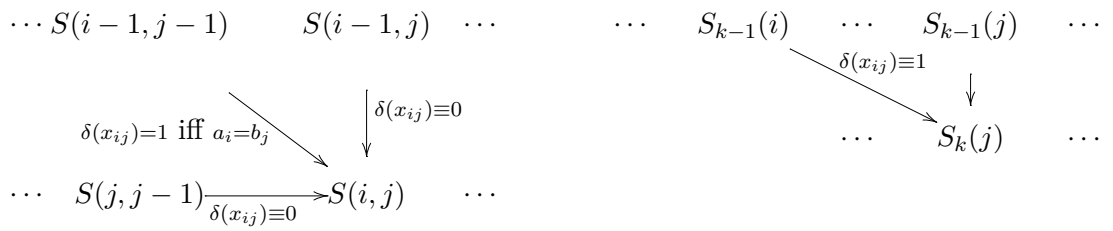


Figure 3: Fragments of a static branching programs for the Longest Common Subsequence problem (left), and for the Shortest Paths problem (right)

directed acyclic graph G_n on vertices $1, \ldots, n$, where two vertices i and j are adjacent if and only if $i < j$. Then every instance $x \in \mathbb{Z}^n$ turns to the 0-1 weighting $w : G_n \to \{0, 1\}$ of the edges of G_n given by $w_{ij} = 1$ if and only if $x_i \leq x_j$. Such a weighting leads to a subgraph of G_n consisting of edges (i, j) with $w_{ij} = 1$, and the goal is to find a longest path in this subgraph. If $S(j)$ denotes the length of the longest path ending in vertex j, then the DP solution is

$$S(j) = \max\{S(i) + 1 \colon w_{ij} = 1\}\,.$$

This algorithm can be turned into a read-once dynamic BP of size n by just taking D_n as the underlying graph, and allowing each wire $e = (i, j)$ to make the survival test "is $x_i \leq x_j$?", and a trivial decision $\delta_e \equiv 1$ (always accept). It then remains to add a rectifier (an unlabeled wire) from each of the n vertices to a new target node.

Example 4 (Longest Common Subsequence). Given two sequences of characters $a = (a_1, \ldots, a_n)$ and $b = (b_1, \ldots, b_m)$, the goal is to find a longest common subsequence of them. A common subsequence of a and b is a sequence $c = (c_1, \ldots, c_k)$ 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 $r = 1, \ldots, k$. 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 cost 1. A problem instance is then the $n \times m$ 0-1 matrix $M = (m_{ij})$ such that $m_{ij} = 1$ if and only if $a_i = b_j$. The goal is to find a largest number of 1-entries, no two of which lie the same row or in the same column. This number is also known as the *term-rank* of M. As subproblems we take $S(i, j)$ = length of the longest subsequence of $(a_1, \ldots, a_i)$ and $(b_1, \ldots, b_j)$. The terminal values are $S(0, j) = S(i, 0) = 0$ for all i, j. The DP solution is

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

This algorithm can be easily implemented as a static read-once BP with $\mathcal{O}(nm)$ nodes $S(i, j)$ for $i = 0, 1, \ldots, n$ and $j = 0, 1, \ldots, m$ (see Fig. 3). Each node $S(i, j)$ is entered by three contacts from $S(i, j-1)$, $S(i-1, j-1)$ and $S(i-1, j)$, each responsible for the data item (edge) (i, j). The decisions made on contacts $S(i, j-1) \to S(i, j)$ and $S(i-1, j) \to S(i, j)$ are $\delta(x_{ij}) \equiv 0$ (always reject), and that on the contact $S(i-1, j-1) \to S(i, j)$ is: $\delta(x_{ij}) = 1$ if and only if $a_i = b_j$. The target node is $S(n, m)$.

Example 5 (Single Source Shortest Paths, Bellman–Ford algorithm). 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. 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}. 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 solution is:

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

The optimal value for a vertex j is $S_{n-1}(j)$. The algorithm can be easily implemented as a static BP (see Fig. 3).

The node $S_k(j)$ 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 $j = 2, \ldots, n$. 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 $\mathcal{O}(n^2)$, and the number of wires is $\mathcal{O}(n^3)$.

Interestingly, it is shown in [13] that this problem requires prioritized branching *trees* of exponential size; thus, static BP $\not\subseteq$ pBT.

Example 6 (The k-TSP problem). Consider the n-city traveling salesman problem (TSP) defined on a complete directed (or undirected) graph with edges assigned their non-negative costs, and fix city 1 as the home city, where all tours start and end. Suppose now that we are given an integer k, $1 \leq k < n$, and an ordering $(1, \ldots, n)$ of the set of n cities, and we want to find a minimum cost permutation π of $\{1, \ldots, n\}$ (an associated tour) subject to the condition: $j \geq i + k$ implies $\pi(i) < \pi(j)$. That is, city i must be traversed before city j if $i + k \leq j$. It is shown in [6] that this problem can be solved in time $T = \mathcal{O}(k^2 2^k n)$. This is obtained by reducing the problem to the shortest s-t path problem. The reduction itself immediately gives us a static BP (with trivial decisions "always accept") for this problem with T nodes.

Table 1: This table summarizes the form of obtained dynamic BPs.

Problem	Static BP	Constant decisions	BP is read-once	Size
Max. Contiguous Subseq.	Yes	Yes	Yes	n
Longest Increasing Subseq.	**No**	Yes	Yes	n
Longest Common Subseq.	Yes	**No**	Yes	n^2
Shortest Path	Yes	Yes	**No**	n^2
k-TSP	Yes	Yes	**No**	$k^2 2^k n$
Knapsack	**No**	Yes	Yes	nK

ECCC ISSN 1433-8092

http://eccc.hpi-web.de