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 Knapsack problem is incremental, produces nK subproblems and nK^2 relations (wires) between the subproblems, where n is the number of items, and K is the knapsack capacity. 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. This holds even for the Subset-Sum problem. We also give upper and lower bounds on the number of subproblems needed to approximate the Knapsack problem. Finally, we show that the Maximum Bipartite Matching problem and the Traveling Salesman problem require exponential number of subproblems. The goal of this paper is to leverage ideas and results of boolean circuit complexity for proving lower bounds on dynamic programming.

Keywords Dynamic programming · Knapsack · Matching · 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 also should be not omnipotent, should avoid the power of arbitrary algorithms, problems like **P** versus **NP**, as well as the power of general boolean circuits.

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

Goethe-University Frankfurt, Faculty of Computer Science and Mathematics, Robert-Mayer Str. 11-15, Frankfurt am Main, Germany · Vilnius University, Institute of Mathematics and Informatics, Akademijos 4, Vilnius, Lithuania.
E-mail: jukna@thi.informatik.uni-frankfurt.de

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. The lower-bound proofs themselves localize the weak points of the paradigms, which can lead to better heuristics. On the other hand, 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. Let us stress that we are seeking for *absolute* lower bounds that are independent of any unproven assumptions, like the assumption that $\mathbf{P} \neq \mathbf{NP}$.

In this paper we focus on the dynamic programming paradigm. There were many attempts to formalize this paradigm, and various refinements were obtained [8,27,20,19,37], just to mention some earlier important contributions in this direction. In these attempts, the goal was to capture 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. [13]. These models are based on the framework of "priority algorithms" introduced by Borodin et al. [11], and subsequently studied and generalized by many authors [4,3,12,14,34,33], just to mention some of them. This framework aims to capture the power of greedy algorithms. The model of pBT extends the power of greedy algorithms by adding some aspects of backtracking and dynamic programming. The model of pBP adds to that of pBT an additional feature of "memoization" (or "cashing" or "merging" subtrees), an important aspect of dynamic programming. 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 (restricted) pBPs of exponential size.

In this paper we pursue essentially the same goal as in [3] and [13], but with exclusive focus on the dynamic programming paradigm. 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 at 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.

For example, the fact that every DP algorithm implicitly constructs a "subproblem graph" describing the dependencies between subproblems, leads to the model of so-called "dynamic branching programs." The fact that most DP algorithms are given by their recurrence relations using Plus and Min/Max operations suggests that the so-called "tropical" circuits constitute a natural model for DP algorithms. These two models are already capable to simulate most of fundamental DP algorithms for discrete optimization problems.

We will mainly consider 0/1 optimization problems. A problem instance (or an input) here is a sequence of data items together with a vector $x = (x_1, \ldots, x_n)$ of their real-valued weights. Solutions are subsets $S \subseteq [n] := \{1, \ldots, n\}$ of (indexes of) data items. The value of a solution S is the sum $\sum_{i \in S} x_i$ of weights of its items. Some solutions are declared as *feasible solutions*. The goal is then to find the maximal (or minimal) weight

of a feasible solution and/or an optimal solution itself. Such problems are called "zero-one" problems because one can identify solutions S with their characteristic 0/1 vectors $c_S \in \{0,1\}^n$, where $c_S(i) = 1$ means that the i-th item is "accepted" (belongs to S), and $c_S(i) = 0$ means that the i-th item is "rejected" (does not belong to S). That is, in 0/1 optimization problems, solutions are 0/1 vectors $c \in \{0,1\}^n$ maximizing (or minimizing) the sum $\sum_{i=1}^n c_i x_i$. Note that being a 0/1 optimization problem only means that decisions are 0 and 1, the weights may be arbitrary numbers.

For example, in the n-dimensional Knapsack problem with knapsack capacity K, data items are pairs of non-negative integers (size, profit). The weight of such an item is its profit. Feasible solutions are subsets of data items whose total size does not exceed the capacity K. The goal is to find a feasible solution whose total profit is maximized.

Remark 1 Note that in the Knapsack problem, the family $\mathcal{F}_x \subseteq 2^{[n]}$ of feasible solutions depends on the input x: $S \in \mathcal{F}_x$ if and only if the sum of sizes of items x_i with $i \in S$ does not exceed K. In many other 0/1 optimization problems, the family $\mathcal{F}$ of feasible solutions is input-independent. For example, in the Maximum Weight Bipartite Matching problem, we have a fixed complete bipartite $n \times n$ graph $K_{n,n}$, inputs are assignments $x : K_{n,n} \to \mathbb{R}$ of weights to its edges, and $\mathcal{F}$ is the family of all matchings in $K_{n,n}$. In the s-t shortest path problem, $\mathcal{F}$ consists of all subsets of edges of a complete n-vertex graph K_n containing an s-t path, in the TSP, $\mathcal{F}$ consists of all Hamiltonian cycles in K_n, etc. In such problems, the goal is to compute the function $f(x) = \max_{S \in \mathcal{F}} \sum_{i \in S} x_i$ or $f(x) = \min_{S \in \mathcal{F}} \sum_{i \in S} x_i$, where $\mathcal{F} \subseteq 2^{[n]}$ is a family of feasible solutions, one for all weights-vectors x.

Roughly speaking, dynamic programming is a method of solving optimization problems in which one first identifies a collection of subproblems and tackles them one by one, "easiest" first, using the answers to easy problems to help figure out more difficult ones, until the master problem is solved. This paradigm is based on the induction principle: describe a way to solve the problem, *assuming* solutions for all of its subproblems are known. The recurrence relation usually uses Min and Plus (or Max and Plus) operations.

Actually, the use of Plus operations in most DP algorithms is restricted: one of the inputs is an input variable (the weight or other numerical parameter of a treated data item) and not the value of another subproblem. We call such algorithms *incremental*. Such is, for example, the standard DP algorithm for the Knapsack problem (more examples are given in Sect. 2.4). As subproblems this algorithm takes $f(i,k)$ = the maximal total profit for filling a capacity k knapsack with some subset of items $1, \ldots, i$. The DP algorithm is then described by the recursion:

$$f(i,k) = \text{maximum of } f(i-1,k) \text{ and } f(i-1,k-s_i) + p_i, \tag{1}$$

where s_i is the size, and p_i the profit of the i-th item.

1.1 Our results

We first consider 0/1 optimization problems where feasibility of solutions does not depend on actual weights (see Remark 1). In Theorem 1, we give a general lower bound on the

number of subproblems produced by any incremental DP algorithm solving such a problem. As easy corollaries, we then show that any incremental DP algorithm for the Maximum Bipartite Matching problem as well as that for the Traveling Salesman problem must produce an exponential number of subproblems (Theorems 2 and 3).

In the rest of the paper, we concentrate on the Knapsack problem. If we have n items, and if K is the capacity of the knapsack, then the DP algorithm (1) produces nK subproblems. A natural question is: can any incremental DP algorithm solve the Knapsack problem using substantially smaller number of subproblems? Our general lower bound cannot be applied for this problem, because the family of feasible solutions is now input-dependent. Still, by establishing some properties of integer partitions, we are able to prove that any incremental DP algorithm for Knapsack must produce $\Omega(nK)$ subproblems (Theorem 4).

Our next result deals with a more general model where "redundant" paths in the subproblem graph are allowed; a path is "redundant" or "inconsistent" if no input instance uses it. It is known that in the case of *boolean* branching programs presence of such paths may exponentially reduce program size [22]. Still, using a classical lower bound of Hansel [17] for the threshold function, we can prove that at least[1] $\Omega(nK \log K)$ wires (relations between the subproblems) are necessary to solve the Knapsack problem (Theorem 5), even if "redundant" paths are allowed. It remains open whether $\Omega(nK)$ subproblems are necessary in this more general model. Finally, we show that the number of subproblems produced by incremental DP algorithms approximating the Knapsack problem within a factor $1+\epsilon$ lies between n/ϵ and n^3/ϵ (Theorems 6 and 7).

We simulate incremental DP algorithms by so-called "dynamic branching programs". The main idea of this model is to reduce the original optimization problem to the shortest (or longest) s-t path problem in a particular labeled acyclic digraph. The reduction itself is based on an observation that for many DP algorithms, their 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, one of the books [36,24].

2 Dynamic Branching Programs

Every dynamic programming algorithm implicitly constructs an acyclic "subproblem graph" (a "table of partial solutions"). This graph has a directed edge $A \to B$ from the node for subproblem A to the node for subproblem B if determining an optimal solution for subproblem B involves considering an optimal solution for subproblem A. We are mainly interested in the number of nodes (subproblems) in this graph.

Important aspect of *incremental* DP algorithms is that they actually reduce the original problem to the shortest or longest s-t path problem on its subproblem graph: if an item x_i is treated when going from a subproblem A to a subproblem B, then let the "length" of the edge $A \to B$ be the weight of x_i, if the item is accepted, and let the edge have zero length, otherwise. An optimal solution for a given input x is then a shortest (or longest) path in the subproblem graph. This observation is not new: in many books, discrete dynamic

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

programming is developed from the prototypical viewpoint of finding the shortest/longest paths in the subproblem graph or in some related directed acyclic graph.

There is, however, one small "subtlety": the actual graph (in which the shortest or longest path is searched) may *depend* on the actual input sequence of data items. This happens, for example, in the DP algorithm (1) for Knapsack. Here the node $f(i,k)$ is entered by two wires from $f(i-1,k)$ and from $f(i-1,k-s_i)$, and this latter node depends on the actual size s_i of the i-th item. In order to *locally* simulate this dependence, we just allow each wire to have its "survival test". This leads us to the model of "dynamic branching programs".

2.1 The Model

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. Values of variables are data items. 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 this variable. 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* $t_e : D \to \{0,1\}$, where D is the set of all data items:

$$\underset{\text{decision predicate } \delta_e(x_i)}{\overset{\text{survival test } t_e(x_i)}{\circ\!\!\longrightarrow\!\!\circ}}$$

Both $\delta_e(x_i)$ and $t_e(x_i)$ are *arbitrary* functions of one variable; they may arbitrarily depend on the data item x_i itself, not only on its weight. Contact e "accepts" the item x_i if $\delta_e(x_i)=1$, and "rejects" it if $\delta_e(x_i)=0$. The meaning of the survival test is that the contact e "dies" (disappears from the program) on input x if $t_e(x_i)=0$, and "survives" (remains intact) if $t_e(x_i)=1$. If the program has no survival tests, then we call it *static*.

Along one path, the same variable x_i may be treated many times. We however, require that a dynamic BP 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 contacts are responsible for the same variable x_i, then we require that

$$t_{e_1}(x_i) = t_{e_2}(x_i)\,, \tag{2}$$

$$\delta_{e_1}(x_i) \le \delta_{e_2}(x_i) \tag{3}$$

The first condition requires that along every path, survival tests on the *same* data item must have the same outcome. The second condition requires that, once accepted, the same item cannot be rejected later. Note, however, that once rejected, the same item *can* be later accepted. We discuss the issue of "late rejections"—when accepted items can be later rejected—in Sect. 8. The role of the first condition (2) is discussed in Sect. 2.3.

[2] We prefer to use the term "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.

A dynamic BP is *oblivious* if along every path the variables are queried in the same order, and is *read-once* if along every path, no variable is queried more than once. Note that a read-once BP needs not be oblivious, and an oblivious BP needs not be read-once. Every read-once BP automatically satisfies both conditions (2) and (3). Every static BP automatically satisfies the condition (2) just because there are no survival tests at all; condition (3) is also trivially satisfied, if all decision predicates are constant.

2.2 How Does a Dynamic BP Compute?

We first associate with wires their "lengths" as follows. Suppose that a contact e is responsible for the i-th variable. Then the *length* of a contact e *on input* $x \in D^n$ is the weight of the i-th item x_i, if this item is accepted at e, and is 0 if the item is rejected at e; rectifies always have zero length.

Now, when an input $x \in D^n$ arrives, we remove from P all contacts whose survival tests output 0 on x (these contacts "die" on this input). Then the value $P(x)$ computed by a program P on input x is just the length of a longest (or shortest, if we have a minimization problem) s-t path in the resulting subgraph P_x. We set $P(x) = 0$, if there are no s-t paths in P_x.

Being "dynamic" means that the subgraph P_x as well as the lengths of its edges (contacts) may depend on the actual problem instance x. If the program is static, then we have $P_x = P$ for all inputs x, and only lengths of edges may vary.

Remark 2 The shortest and longest s-t path problems on *acyclic* graphs with n vertices and m edges can be easily solved in time $O(n + m)$. Thus, the number of nodes in a dynamic BP simulating a given incremental DP algorithm corresponds to the total number of subproblems produced by the algorithm, whereas the number of wires corresponds to the actual "computational effort" of the algorithm, that is, the number of Max and Plus (or Min and Plus) operations performed. If the program has survival tests, then the actual "computational effort" may be smaller: after input x arrives, some wires disappear, and the subprogram P_x may have much fewer wires.

How does a program *produce* optimal solutions? A path p is *consistent* with a given input string $x \in D^n$, if this input passes all survival tests along p, that is, if all survival tests output 1 on x. The *solution produced by* p on x is the set of items accepted (by decision predicates) along p; the solution may be a multiset, if some items were accepted several times. Thus, the value $P(x)$ computed by a dynamic BP on an input $x \in D^n$ is the maximal (or minimal) value of a solution produced by an s-t path consistent with x. A program P *solves* a given optimization problem if for every input $x \in D^n$ the following holds:

1. At least one s-t path is consistent with x and produces an optimal solution for x.
2. None of the consistent with x s-t paths produces an infeasible solution for x.

Note that we do not require that *all* feasible solutions for x must be produced: we only require that no infeasible solution is produced.

Remark 3 A similar in its "sole" model of so-called *combinatorial dynamic programs* was recently introduced by Bompadre in [9]. This model is similar to our model of static BPs with all decision predicates being "accept". In combinatorial DPs, the correspondence between s-t paths and feasible solutions is less restrictive than in static BPs, but there is 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 [9]). Still, it is shown in [9] that many DP algorithms can be simulated in this model. Using a reduction to monotone arithmetic circuits, exponential lower bounds on the number of wires in combinatorial DPs are proved in [9] for some "permanent-like" optimization problems, that is, problems whose feasible solutions are (not necessarily all) permutations of some set. Examples of such problems are Traveling Salesman Problem, and the Bipartite Matching Problem. First exponential lower bounds for monotone arithmetic circuits were proved by Jerrum and Snir [21], 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.

2.3 Relation to Boolean Branching Programs

Recall that an *nondeterministic branching program* (NBP) is a directed acyclic graph where at some wires tests of the form "is $x_i = 0$?" or "is $x_i = 1$?" are made; unlabeled wires (rectifiers) are allowed. 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 along at least one s-t path. A program is nondeterministic, because one input may follow many s-t paths.

Dynamic BPs constitute a similar model of computation, where instead of the boolean semiring $(\{0,1\}, \vee, \wedge)$ we are working over semiring $(\max, +)$ or $(\min, +)$. The length of a path in this case is the sum of weights of accepted items, and the value of the program is the maximum or minimum of the lengths of all consistent s-t paths. That is, here we have Plus instead of AND, and Max/Min instead of OR.

It is not difficult to see that without the consistency condition (2) on survival tests, dynamic BPs are at least as powerful as NBPs. To show this, suppose we have an NBP P computing some non-constant boolean function $f(x)$. At its contacts, survival tests "is $x_i = 0$?" and 'is $x_i = 1$?" are made. Since f is not the constant-1 function, along each consistent path at least one variable must be tested. We let each bit x_i to have weight 1. We can transform P into a dynamic BP $P'(x)$ solving a 0/1 maximization problem by just adding to each contact the decision "always accept". Now if $P(x) = 1$, then at least one s-t path in P is consistent with x, implying that $P'(x) \geq 1$. If $P(x) = 0$, then no s-t path in P is consistent with x, implying that $P'(x) = 0$. Thus, the dynamic BP $P'(x)$ computes the same boolean function $f(x)$, and has the same size as the NBP $P(x)$.

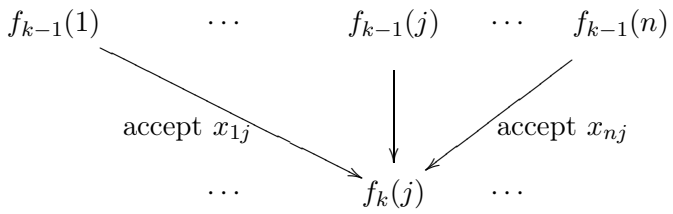
This fact is a strong word of caution: the best known lower bound for NBPs remains the lower bound $\Omega(n^{3/2}/\log n)$ proved by Nechiporuk [31] more than 45 years ago (see, e.g., [24, Sect. 15.1]). Moreover, this bound is on the number of *wires*; concerning the number of *nodes*, even super-linear lower bounds are not known; recall that neither fanin nor fanout of nodes in NBPs as well as in dynamic BPs is bounded.

Note, however, that it is the consistency condition (2) which makes the model of dynamic BPs more tractable than that of general NBPs. In the case of NBPs, this condition implies that there cannot be any "null-paths", that is, s-t paths along which two contradictory tests $x_i = 0$ and $x_i = 1$ on the same variable x_i are made. For such NBPs we can already prove even exponential lower bounds (see, e.g., [24, Sect. 16.3]).

2.4 Examples

Many DP algorithms can be directly translated to dynamic (and even static) branching programs just by using their recursion relations. We restrict ourselves to several illustrative examples.

Example 1 (Single Source Shortest Paths, Bellman–Ford–Moore) A problem instance is an assignment of real lengths x_{ij} to the edges of a directed complete graph K_n on vertices $[n] = \{1, \ldots, n\}$. The goal is to find shortest paths from vertex 1 to all remaining vertices. The Bellman–Ford–Moore algorithm for this problem takes as subproblems $f_k(j)$ = length of a shortest path from 1 to j using at most k edges. The terminal values are $f_1(j) = x_{1j}$, $j = 2, \ldots, n$. The DP recursion is: $f_k(j)$ = minimum of $f_{k-1}(j)$ and $f_{k-1}(i) + x_{ij}$ for all i. The optimal value for a vertex j is $f_{n-1}(j)$. The algorithm can be easily implemented as a static BP. A fragment of this BP is shown here:

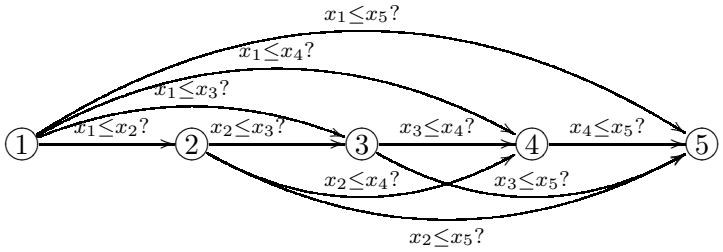


The node (subproblem) $f_k(j)$ is entered by a rectifier (unlabeled wire) $f_{k-1}(j) \to f_k(j)$ as well as by contacts $f_{k-1}(i) \to f_k(j)$ with $i \neq j$ responsible for edges (i, j); the length of each such contact is just the length x_{ij} of the edge (i, j). We also have a source node s with contacts going to all nodes $f_1(j)$ for $j = 2, \ldots, n$. Each contact $s \to f_1(j)$ is responsible for the edge $(1, j)$, and the decision is here also "accept". The number of nodes in the constructed static BP is $O(n^2)$, and the number of wires is $O(n^3)$. The BP itself reduces the shortest paths problem in K_n to the same problem in an *acyclic* graph on $O(n^2)$ vertices.

Example 2 (Maximum Value Contiguous Subsequence) A problem instance is a sequence $x_1, \ldots, x_n$ of real weights (positive and negative), and the goal is to find a contiguous subsequence with maximal weight. Thus, feasible solutions are contiguous intervals $i, i+1, \ldots, j$, and their values are the sums $x_i + x_{i+1} + \cdots + x_j$. As subproblems we take $f(j)$ = maximum weight of an interval ending in j. The terminal value is $f(0) = 0$. The DP recursion is $f(j) = \max\{f(j-1) + x_j,\ x_j\}$. This algorithm can be implemented as a read-once static BP with $n + 2$ nodes $s, 1, 2, \ldots, n, t$ and $O(n)$ wires. Each node $i \in [n]$

is entered by two contacts: $s \to i$ and $i-1 \to i$, both responsible for x_i. All decisions are "accept". This program is not oblivious, but is read-once and is static. There are also rectifiers (unlabeled wires) from all nodes $i \in [n]$ to the target node t.

Example 3 (Longest Increasing Subsequence) A problem instance is a sequence $x_1, \ldots, x_n$ of numbers, and the goal is to find a largest subsequence $i_1 < i_2 < \ldots < i_k$ of indexes such that $x_{i_1} \le x_{i_2} \le \ldots \le x_{i_k}$. If $f(j)$ denotes the length of the longest increasing subsequence ending in x_j, then the DP solution is $f(j) = 1 + \max\{f(i)\colon i < j \text{ and } x_i \le x_j\}$. This algorithm can be turned into a read-once dynamic BP. As items we take pairs $x_{ij} = (x_i, x_j)$ with $i < j$, each of weight 1. We have nodes $1, \ldots, n$, and contacts (i, j) for all $i < j$. Each such contact is responsible for the item $x_{ij} = (x_i, x_j)$, and makes a survival test "is $x_i \le x_j$?". All decisions are "accept". It then remains to add rectifiers (unlabeled wires) from a source node s to all nodes $1, \ldots, n$, and rectifiers from these nodes to a target node t. Here is a dynamic BP for $n = 5$ numbers (without nodes s and t shown):

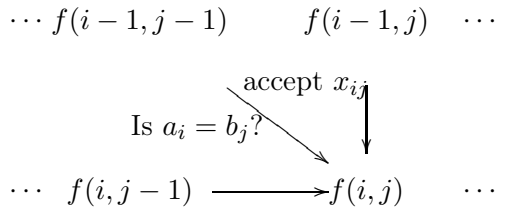


The program has $n+2$ nodes and $O(n^2)$ wires. After an input $x_1, \ldots, x_n$ arrives, all contacts (i, j) with $x_i > x_j$ will disappear, and the longest increasing subsequence is the sequence of inner nodes of a longest s-t path in the resulting subgraph.

Example 4 (Weighted Interval Scheduling) We have n intervals $x_1, \ldots, x_n$ on the real line, each with its value, and the goal is to find a subset of *pairwise disjoint* intervals of maximal total value. Each interval x_i is specified by its start-point s_i and end-point e_i. We assume that intervals are ordered so that $e_1 \le e_2 \le \ldots \le e_n$. As items we again take pairs $x_{ij} = (x_i, x_j)$ of intervals with $i < j$; the weight of such an item is the value $v(x_j)$ of the second interval x_j. As subproblems we take $f(j) =$ the value of an optimal solution for the first j intervals. Then $f(j)$ is the maximum of $f(j-1)$ and $f(i) + v(x_j)$ over all $i < j$ such that $e_i < s_j$. This DP algorithm can be turned into a dynamic read-once BP with n nodes in an obvious manner: it is enough to take a BP from the previous example, add rectifiers $i \to i+1$ for all $i = 1, \ldots, n-1$, and replace every survival test "is $x_i \le x_j$?" by the test "is $e_i < s_j$?", that is, by the test "is the end-point of the i-th interval smaller than the start-point of the j-th interval?".

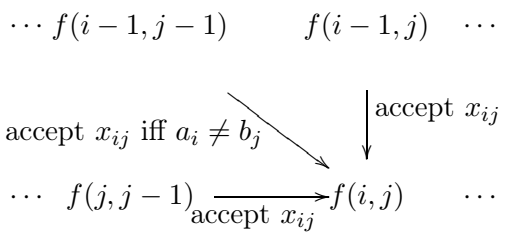
Example 5 (Longest Common Subsequence) Given two strings $a = (a_1, \ldots, a_n)$ and $b = (b_1, \ldots, b_m)$ of elements, the goal is to find a longest common subsequence of them. A common subsequence of a and b is a string $c = (c_1, \ldots, c_k)$ for which there exist two sequences of positions $1 < i_1 < \cdots < i_k < n$ 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 weight 1. As subproblems we take $f(i,j)$ = length of the longest common subsequence of $(a_1, \ldots, a_i)$ and $(b_1, \ldots, b_j)$. The terminal values are $f(0,j) = f(i,0) = 0$ for all i, j. The value $f(i,j)$ is computed as the maximum of $f(i,j-1), f(i-1,j)$ and of $f(i-1,j-1)+1$ if $a_i = b_j$. The answer is $f(n,m)$. This algorithm can be easily implemented as a dynamic read-once BP with $O(nm)$ wires:



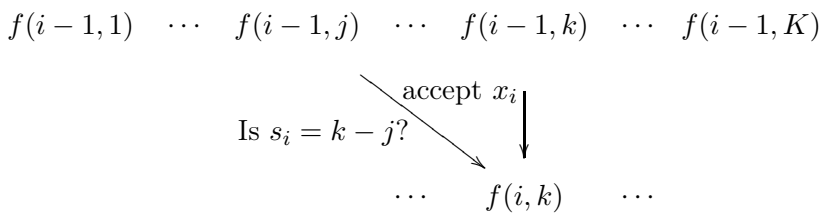
Wires $f(i,j-1) \to f(i,j)$ and $f(i-1,j) \to f(i,j)$ are rectifiers, and hence, have length 0. The survival test "is $a_i = b_j$?" ensures that the third term $f(i-1,j-1)+1$ occurs in the recursion relation only if $a_i = b_j$.

Example 6 (Pairwise Alignment) Here we are also given two strings a and b (say, two DNA sequences), and the goal is to determine the minimum number of "point mutations" required to change a into b, where a point mutation is one of: change a letter, insert a letter or delete a letter. Items in this case are also pairs $x_{ij} = (a_i, b_j)$, each of weight 1. As subproblems we take $f(i,j)$ = minimal number of point mutations required to change $(a_1, \ldots, a_i)$ to $(b_1, \ldots, b_j)$. The terminal values are $f(0,j) = j$ and $f(i,0) = i$ for all i, j; these values correspond to the cases when one of the strings is empty. The value $f(i,j)$ is computed as the minimum of $f(i,j-1)+1, f(i-1,j)+1$ and of $f(i-1,j-1)+w_{ij}$, where w_{ij} is 1 if $a_i \neq b_i$, and is 0 otherwise. The answer is $f(n,m)$. This algorithm can be easily turned into a static read-once BP with $O(nm)$ wires:



Here "accept" means "a point mutation happened".

Example 7 (Knapsack) Recall that in the Knapsack problem items are pairs $x_i = (p_i, s_i)$, where p_i is the profit and s_i the size of the i-th item. The standard DP algorithm (1) takes as subproblems $f(i,k) =$ the maximal total profit for filling a capacity k knapsack with some subset of items $x_1, \ldots, x_i$. The recursion relation is then $f(i,k) =$ the maximum of $f(i-1,k)$ and $f(i-1,k-s_i)+p_i$. The output is $f(n,K)$. This algorithm can be also easily turned into an oblivious read-once dynamic BP with nK nodes as follows:



As nodes we take the nK subproblems $f(i,k)$. All contacts entering such a node are responsible for the i-th item $x_i = (p_i, s_i)$. The rectifier $f(i-1,k) \to f(i,k)$ has length 0. Every contact $f(i-1,j) \to f(i,k)$ with $j < k$ makes a decision "always accept". At each such contact, a survival test "is $s_i = k-j$?" is made. Thus, only one of these contacts, namely $f(i-1,k-s_i) \to f(i,k)$, will survive, and its length is the profit p_i of the i-th item.

There is also the start node s from which there is a contact responsible for the first item x_1 to each of the nodes $f(1,1), \ldots, f(1,K)$. Each contact $s \to f(1,j)$ makes a trivial decision "always accept", and has a survival test "is $s_1 \le j$?". The target node is $f(n,K)$. It is easy to see that the resulting dynamic BP is read-once and oblivious. The program has nK nodes (by ignoring the source node) and $O(nK^2)$ wires. Note, however, that on each input, only $O(nK)$ of the wires will survive.

Example 8 (TSP) We have to visit cities 1 to n. We start in city 1, run through the remaining $n-1$ cities in some order, and finally return to city 1. Inputs are non-negative integer distances x_{ij} between cities i and j. In the Maximum Traveling Salesman problem, know also as the Taxicab Ripoff problem, the goal is to maximize the total travel length. The latter problem is usually motivated by the fact that good approximation algorithms for it yield good approximation algorithms for the Shortest Common Superstring problem (see, e.g.. [6]). A trivial algorithm for Maximum TSP checks all $(n-1)! = \Omega((n/e)^n)$ permutations. The DP algorithm suggested by Held and Karp [18] solves the problem this problem in exponential, but much shorter time $O(n^2 2^n)$. It takes as subproblems $f(S,i) =$ length of a longest path that starts in city 1, then visits all cities in $S \setminus \{i\}$ in an arbitrary order, and finally stops in city i. Here $S \subseteq \{2, \ldots, n\}$ and $i \in S$. Clearly, $f(\{i\}, i) = x_{1i}$. The DP recursion is:

$$f(S,i) = \max\left\{f(S \setminus \{i\}, j) + x_{ji} \colon j \in S \setminus \{i\}\right\}.$$

The optimal travel length is given as the maximal value of $f(\{2, \ldots, n\}, j) + x_{j1}$ over all $2 \le j \le n$. By replacing max by min, we obtain a DP algorithm for the (usual)

minimization TSP. Both algorithms are incremental, and can be directly simulated by read-once static BPs with $O(n2^n)$ nodes $f(S,i)$, and $O(n^2 2^n)$ wires. Each such node is entered by $|S|-1$ contacts $f(S\setminus\{i\},j) \to f(S,i)$, responsible for the variables $x_{j\,i}$. All decisions are "accept". Hence, the length of every contact is the distance of the edge it is responsible for.

Table 1 This table summarizes the form of obtained dynamic BPs; we ignore multiplicative constants.

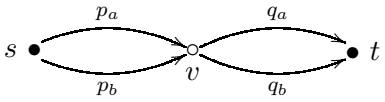
Problem	Static BP	Constant decisions	Read-once	Nodes	Wires
Single Source Shortest Paths	Yes	Yes	**No**	n^2	n^3
Max Contiguous Subsequence	Yes	Yes	Yes	n	n
Longest Increasing Subsequence	**No**	Yes	Yes	n	n^2
Interval Scheduling	**No**	Yes	Yes	n	n^2
Longest Common Subsequence	**No**	Yes	Yes	nm	nm
Pairwise Alignment	Yes	**No**	Yes	nm	nm
Knapsack with capacity K	**No**	Yes	Yes	nK	nK^2
TSP	Yes	Yes	Yes	$n2^n$	$n^2 2^n$

Let us stress that no effort was needed to turn the mentioned DP algorithms into dynamic BPs: the resulting BPs are just "graphic" representations of their recurrence relations.

3 A Structural Lemma

In this section, we establish some structural properties of dynamic BPs which will latter allow us to combine computation paths. These properties are consequences of consistency conditions (2) and (3).

Consider a dynamic BP P solving some 0/1 maximization problem with non-negative weights. Take any two inputs $a, b \in D^n$, and fix any two s-t paths in P producing optimal solutions for these inputs. Suppose that these paths meet in some inner node v. Let p_a denote the first segment of the path for a until the node v, and q_a the rest of this path; similarly for input b:



We say that a variable is *tested* along a path, if some contact of the path is responsible for this variable. Let $I_a = \{i \in [n] \colon x_i \text{ is tested along } p_a\}$ and $J_a = \{i \in [n] \colon x_i \text{ is tested along } q_a\}$. Let also $P_a = \{i \in I_a \colon a_i \text{ is accepted along } p_a\}$ and $Q_a = \{i \in J_a \colon a_i \text{ is accepted along } q_a\}$. That is, $P_a \subseteq I_a$ is the partial solution produced by the path p_a on input a, and $Q_a \subseteq J_a$ is the partial solution produced by the path q_a on this input. Hence, $P_a \cup Q_a$ is the solution for a produced by the entire s-t path (p_a, q_a). Note that $P_a \cap Q_a = \emptyset$ because we consider a 0/1 optimization problem, and $P_a \cup Q_a$ is an optimal solution for a (not just feasible).

By a *combination* of a pair (a, b) we will mean any input $c \in D^n$ such that $c_i = a_i$ for all $i \in I_a \setminus J_b$, $c_i \in \{a_i, b_i\}$ for all $i \in I_a \cap J_b$, and $c_i = b_i$ for all remaining i. That is, c coincides with a on all variables tested only along p_a, coincides with b on all variables tested only along q_b, and coincides with either a or b on every re-tested variable x_i.

Recall that a path is consistent with a given input, if this input passes all survival tests made along the path. If a path p is consistent with an input $a \in D^n$, then its *weight*, $w_p(a)$ on this input is the sum of weights of items of a accepted along p.

Lemma 1 (Cut-and-Paste Lemma)
1. *Every combination c of (a, b) is consistent with the combined path (p_a, q_b).*
2. *For every combination c of (a, b), all items c_i with $i \in I_a \cap J_b$ are rejected along p_a.*
3. *There is a combination c of (a, b) such that $w_{p_a}(c) = w_{p_a}(a)$ and $w_{q_b}(c) = w_{q_b}(b)$.*
4. *$P_a \cap Q_b = \emptyset$.*

Proof In the proof, we will essentially use the consistency conditions we posed on dynamic BPs. Recall that the first condition is on survival tests: along every path, the survival tests on the same variable x_i must be identical. The second condition is on decision predicates: if an item is accepted on a path, then it cannot be rejected latter on that path.

We first show (1), i.e., that input c is consistent with the combined (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 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 $t_{e_1}(x_i) = t_{e_2}(x_i)$ on survival tests 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 (2), assume the opposite, i.e., that some item c_i with $i \in I_a \cap J_b$ is accepted along p_a. Then there is a contact e_1 on p_a testing the i-th variable x_i such that $\delta_{e_1}(c_i) = 1$. 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 on decision predicates 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). But since the weights are non-negative, and since we consider a 0/1 maximization problem, the solution produced by the path (p_a, q_b) on input c is *not* a feasible solution for c, a contradiction. (This is why we stated the lemma only for BPs solving *maximization* problems.)

To show (3), consider a combination c of (a, b) with $c_i = b_i$ for all $i \in I_a \cap J_b$. Then c coincides with b on all variables tested along q_b, implying that $w_{q_b}(c) = w_{q_b}(b)$. By claim (2), for all $i \in I_a \cap J_b$, both items c_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, implying that $w_{p_a}(c) = w_{p_a}(a)$.

Claim (4) follows from (2) by taking a combination c with $c_i = a_i$ for all $i \in I_a \cap J_b$. □

4 A General Lower Bound

In this section we consider 0/1 maximization problems $f(x) = \max_{S \in \mathcal{F}} \sum_{i \in S} x_i$, where $\mathcal{F} \subseteq 2^{[n]}$ is a family of feasible solutions, and x_i is a weight of the i-th item. Note that here

being feasible is input-independent: the family $\mathcal{F}$ of feasible solutions is one and the same for all inputs x (see Remark 1). We say that the problem has the *unique optimum* property, if for every feasible solution $S \in \mathcal{F}$, there is an input x such that S is a unique optimal solution for x. Note that, if weights 0 and 1 are allowed, then every maximization problem has this property: just set $x_i = 1$ for $i \in S$, and $x_i = 0$ for $i \notin S$. The unique optimum property was also used in [3] and [13] to prove lower bounds for prioritized branching trees and branching programs.

Under the r-th degree, $d_r(\mathcal{F})$, of a family of sets $\mathcal{F} \subseteq 2^{[n]}$ we will mean the maximum number of sets in $\mathcal{F}$, all sharing some r elements in common. Thus, $d_0(\mathcal{F}) = |\mathcal{F}|$, and $s \leq r$ implies $d_s(\mathcal{F}) \geq d_r(\mathcal{F})$.

Theorem 1 *Let $1 < m \leq n$ be an integer, and consider a 0/1 maximization problem with the unique optimum property. Let $\mathcal{F}$ be the family of all feasible m-element solutions. Then the number of nodes in every dynamic BP solving this problem must be at least*

$$2 + \sum_{r=1}^{m-1} \frac{|\mathcal{F}|}{d_r(\mathcal{F}) \cdot d_{m-r}(\mathcal{F})}\,.$$

Proof The proof idea is borrowed from [25]. Let P be a dynamic BP solving our maximization problem. Associate with every solution $S \in \mathcal{F}$ an input a_S and an s-t path π_S such that S is a solution for a_S produced by π_S. Such an input a_S and a path π_S must exist, because we have the unique optimum property, and because the program must produce at least one optimal solution for each input. We concentrate on the set of inputs $A = \{a\colon a = a_S \text{ for some } S \in \mathcal{F}\}$. Fix an integer $1 \leq r < m$, and stop the path π_S after exactly r items of a_S are accepted. Let p_a denote the initial part, and q_a the second part of this path; hence, $\pi_S = (p_a, q_a)$.

Let V_r be the set of nodes at which at least one of the s-t paths π_S for $S \in \mathcal{F}$ was stopped. For a node $v \in V_r$, let $A_v \subseteq A$ be the set of all items $a \in A$, whose initial paths p_a end in v. Finally, let $\mathcal{F}_v \subseteq \mathcal{F}$ be the set of all feasible solutions of size m which are produced on some inputs by s-t paths going through the node v.

By an *r-rectangle* we will mean a family $\mathcal{R} \subseteq 2^{[n]}$ of subsets for which there exists two sequences $S_1, \ldots, S_p$ and $T_1, \ldots, T_q$ of subsets of $[n]$ such that $|S_i| = r$ for all i, $S_i \cap T_j = \emptyset$ for all $i \neq j$, and $\mathcal{R} = \{S_i \cup T_j\colon 1 \leq i \leq p, 1 \leq j \leq q\}$; we call such two sequences of sets a *generator* of $\mathcal{R}$.

Claim 1 For every node $v \in V_r$, the family $\mathcal{F}_v$ is an r-rectangle.

Proof Fix a node $v \in V_r$. Let $S_1, \ldots, S_p$ be the partial solutions produced on inputs $a \in A_v$ by initial segments of paths until these paths were stopped at node v; hence, $|S_i| = r$ for all i. Let also $T_1, \ldots, T_q$ be the partial solutions produced by second segments of these paths after the node $v$hence, $|T_j| = m - r$ for all j. By Lemma 1(4), we have that $S_i \cap T_j = \emptyset$ for all $i \neq j$. So, it remains to show that every set $S_i \cup T_j$ belongs to $\mathcal{F}_v$, i.e., that $S_i \cup T_j$ is a solution of size m produced on some input by an s-t path going through the node v.

Using the notation of the previous section, we have that $S_i = P_a$ and $T_j = Q_b$ for some inputs $a, b \in A_v$. Take the combined input c such that $c_i = a_i$ for all $i \in I_a \setminus J_b$, and $c_i = b_i$

for all remaining i. By Lemma 1(1), this input is consistent with the combined path (p_a, q_b). By Lemma 1(2), all items a_i and c_i with $i \in I_a \cap J_b$ are rejected along p_a. In all remaining positions $i \in I_a \setminus I_b$, input c coincides with a. Thus, $P_c = P_a$. Since c coincides with b on all variables tested along q_b, we also have that $Q_c = Q_b$. Thus, $S_i \cup T_j = P_c \cup Q_c$ is a feasible solution for input c produced by the combined path (p_a, q_b). Since $|P_c| = |P_a| = r$ and $|Q_c| = |Q_b| = m - r$, this solution has size m, and we are done. □

Claim 2 If $\mathcal{R}$ is an r-rectangle, and if each set in $\mathcal{R}$ has at least $m \geq r$ elements, then $|\mathcal{R}| \leq d_r(\mathcal{R}) \cdot d_{m-r}(\mathcal{R})$.

Proof Let $S_1, \ldots, S_p$ and $T_1, \ldots, T_q$ form a generator of $\mathcal{R}$. Hence, $|S_i| = r$, $|T_j| \geq m - r$ and $|\mathcal{R}| \leq p \cdot q$. On the other hand, for every fixed $1 \leq i_0 \leq s$, the number q of sets $S_{i_0} \cup T_1, \ldots, S_{i_0} \cup T_q$ cannot exceed $d_r(\mathcal{R})$, because all these sets contain a fixed set S_{i_0} of size $|S_{i_0}| = r$. Since $r \geq s$ implies $d_r(\mathcal{R}) \leq d_s(\mathcal{R})$, the same argument yields $p \leq d_{m-r}(\mathcal{R})$. □

Since every s-t path π_S must go through some node in V_r, we have that $\mathcal{F} = \bigcup_{v \in V_r} \mathcal{F}_v$. Claims 1 and 2 imply that $|\mathcal{F}_v| \leq d_r(\mathcal{F}) \cdot d_{m-r}(\mathcal{F})$ holds for every node $v \in V_r$. Thus,

$$|V_r| \geq \frac{|\mathcal{F}|}{d_r(\mathcal{F}) \cdot d_{m-r}(\mathcal{F})} .$$

Since this holds for every $r = 1, \ldots, m-1$, and the sets V_r are disjoint, and since two nodes (source and target) do not belong to any of the V_r, we are done. □

Remark 4 Theorem 1 also holds with $d_r(\mathcal{F}) \cdot d_{m-r}(\mathcal{F})$ replaced by the maximum number of sets in an r-rectangle contained in $\mathcal{F}$.

To give some applications, let us consider the Maximum Bipartite Matching problem. In this problem, items are edges (i, j) of the bipartite complete $n \times n$ graph $K_{n,n}$ together with their weights $x_{ij} \in \{0, 1\}$. Feasible solutions are sets of vertex-disjoint edges (matchings). The goal is to compute the maximum weight of a matching. This problem can be solved in polynomial time by a simple application of linear programming. But what about the DP complexity of this problem? It is shown in [13] that this problem requires prioritized branching programs of exponential size. Results of [9] also imply that this problem requires combinatorial dynamic programs of exponential size. We now show that dynamic BPs for this problem must be exponentially large as well.

Theorem 2 *Every dynamic BP solving the Maximum Bipartite Matching problem on $K_{n,n}$ must have at least 2^n nodes.*

Proof It is clear that this problem has the unique optimum property: for every matching, assign its edges weight 1, and assign 0 to remaining edges. The family $\mathcal{F}$ of feasible solutions of size n consists of all $n!$ perfect matchings. The r-th degree $d_r(\mathcal{F})$ of $\mathcal{F}$ is the number $(n-r)!$ of all perfect matchings sharing a fixed matching with r edges. By Theorem 1,

the total number of nodes in every dynamic BP solving the Maximum Bipartite Matching problem must be at least

$$2 + \sum_{r=1}^{n-1} \frac{|\mathcal{F}|}{d_r(\mathcal{F}) \cdot d_{n-r}(\mathcal{F})} = 2 + \sum_{r=1}^{n-1} \frac{n!}{(n-r)!r!} = \sum_{r=0}^{n} \binom{n}{r} = 2^n \,.$$

□

As a next example, consider the Maximum Traveling Salesman problem (Max TSP). Recall that in this problem we are given n^2 non-negative integers x_{ij}, and the goal is to find a permutation $(i_2, i_3, \ldots, i_n)$ of $\{2, 3, \ldots, n\}$ such that the sum $x_{1i_2} + x_{i_2 i_3} + \cdots + x_{i_n 1}$ is maximized. Each number x_{ij} corresponds to the distance between cities i and j. A TSP is *metric* if the distances x_{ij} are symmetric ($x_{ij} = x_{ji}$) and satisfy the triangle inequality $x_{ij} \leq x_{ik} + x_{kj}$. This special case of TSP is interesting because it allows much better approximations. We already know (see Example 8) that the general Max TSP can be solved by a static read-once BP of size $O(n2^n)$.

Theorem 3 *Every dynamic BP solving the metric Max TSP on n vertices must have 2^{n-1} nodes, even if weights are 1 and 2.*

Proof The triangle inequality holds because the sum of any two distances is at least 2, the maximal possible distance of a single edge. The problem itself has the unique optimum property: given a Hamiltonian cycle, assign distance 2 to its edges, and distance 1 to other edges. The family $\mathcal{F}$ of feasible solutions consists of all $(n-1)!$ permutations $(i_2, i_3, \ldots, i_n)$ of $\{2, 3, \ldots, n\}$: once i_2 and i_n are known, the distances x_{1i_2} and $x_{i_n 1}$ are predetermined. The r-th degree $d_r(\mathcal{F})$ of $\mathcal{F}$ is the number $(n-1-r)!$ of all permutations with some r values fixed. The same computations as in the proof above (with $n-1$ instead of n) yield the desired lower bound 2^{n-1} on the number of nodes in any dynamic BP solving the problem. □

5 Lower Bound for Knapsack

In Example 7 we have shown that the Knapsack problem can be solved by a dynamic BP using nK nodes, where n is the number of items, and 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.

In the (n, K)-*knapsack problem*, input instances are sequences $a = (a_1, \ldots, a_n)$ of integers in $[K] = \{1, \ldots, K\}$, and the goal is to maximize $\sum_{i \in S} a_i$ over all subsets $S \subseteq [n]$ such that $\sum_{i \in S} a_i \leq K$.

Theorem 4 *If $K \geq 3n$ then every dynamic branching program solving the (n, K)-knapsack problem must have at least $\frac{1}{2}nK$ nodes.*

We cannot apply the general lower bound given by Theorem 1 because now the families $\mathcal{F}_a \subseteq 2^{[n]}$ of feasible solutions *depend* on input instances a: $\mathcal{F}_a$ consists of subsets $S \subseteq [n]$ such that $\sum_{i \in S} a_i \leq K$. So, we need another arguments for Knapsack. To prove Theorem 4, we first establish some properties of integer partitions, which may be of independent interest. We then combine these properties with the structural lemma (Lemma 1) to derive the desired lower bound for the (n, K)-knapsack problem.

5.1 Integer Partitions

Let $k \geq 2$ be a fixed natural number. A *partition* of a natural number n into k blocks is a vector $x = (x_1, \ldots, x_k)$ of natural numbers such that $x_1 + \cdots + x_k = n$. 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 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.

Lemma 2 $\tau(n) = n + 1$.

Proof The upper bound $\tau(n) \leq n + 1$ is easy: already tests $(\{1\}, b)$ with $b = 0, 1, \ldots n$ 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 $b \in \{0, 1, \ldots, n\}$ 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, let $\mathcal{F}$ be the family of supports in our collection of tests. Take a support $S' \in \mathcal{F}$ which is not a proper subset of another support in $\mathcal{F}$. Let S be the complement of S', and replace every test (S', b) in our collection by the complementary test $(S, n - b)$; we can do this because these two tests cover the same set of partitions. What we achieved during this transformation is that now $\overline{S} \not\subseteq T$ holds for all supports T in our collection.

Take now the *smallest* number c which does not appear as a threshold b in any of our tests (S, b) with support S. Thus, we must already have at least c tests $(S, 0), (S, 1), \ldots, (S, c-1)$ 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 $k - |S|$ 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, as well as on the tests covering these partitions. By ignoring the positions $i \in S$, these partitions give us all partitions of $n - c$ into $k - |S|$ blocks. Now replace each test (T, b) with $T \not\subseteq S$ by $(T \setminus S, b')$ where $b' = b - \sum_{i \in S \cap T} a_i$. The new tests cover all these (shorter) partitions. Moreover, $\overline{S} \not\subseteq T$ implies that $|T \setminus S| < k - |S|$, that is, the new tests are legal. By induction hypothesis, there must be at least $n - c + 1$ such (modified) tests, giving a lower bound $m \geq c + (n - c + 1) = n + 1$ on the total number of original tests. □

Let $\tau_k^+(n)$ 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 3 $\tau_k^+(n) \geq n - k + 1$.

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 $b \geq |S|$ 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 $\sum_{i \in S}(x_i - 1) = b - |S|$, 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 $n - k$, and we obtain $\tau_k^+(n) \geq \tau(n - k) = n - k + 1$. □

5.2 Proof of Theorem 4

Take a dynamic branching program $P = (V, E)$ solving the (n, K)-knapsack problem. In particular, 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 $a = (a_1, \ldots, a_n)$ such that $a_1 + \cdots + a_n = K$, and all a_i belong to $[K] = \{1, \ldots, K\}$.

For every $a \in A$, there must be an s-t path which is consistent with a and has length 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. Of course, $S = [n]$ is the unique optimal solution for every input $a \in A$. The point, however, is that the program cannot produce any infeasible solution for the remaining inputs. We will use this fact to show that not too many of optimal paths for inputs in A can meet in an inner node.

Fix 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 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. For a node $v \in V_r$, let $A_v \subseteq A$ be the set of all items $a \in A$, whose initial paths p_a end in v.

Claim 3 On all inputs $a \in A_v$, the initial segments p_a produce the same partial solution $S \subseteq [n]$, and the value $b = \sum_{i \in S} a_i$ of these solutions is the same for all $a \in A_v$.

Proof Take arbitrary two inputs $a, b \in A_v$. Let, as before, $P_a \subseteq [n]$ be the partial solution produced on input a by the first segment p_a of the optimal path for a, and Q_a the partial solution produced on input a by the second segment q_a of this path. By Lemma 1(4), 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 that $P_a = P_b$ and $Q_a = Q_b$. In particular, this means that the initial parts p_a and p_b produce the same partial solution S on the corresponding inputs.

Let $w_p(a)$ denote the weight of a solution produced by a path p on input a. Thus $w_{p_a}(a) = \sum_{i \in S} a_i$ and $w_{p_b}(b) = \sum_{i \in S} b_i$. Assume that $w_{p_a}(a) \neq w_{p_b}(b)$, say, $w_{p_a}(a) > w_{p_b}(b)$. Let c be a combination of (a, b) guaranteed by Lemma 1(3); hence, $w_{p_a}(c) = w_{p_a}(a)$

and $w_{q_b}(c) = w_{q_b}(b)$. Then the combined path (p_a, q_b) produces an infeasible solution for c of weight $w_{p_a}(c) + w_{q_b}(c) = w_{p_a}(a) + w_{q_b}(b) > w_{p_b}(b) + w_{q_b}(b) = K$, a contradiction. □

By Claim 3, 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. Together with Lemma 3, this implies that $|V_r| \geq \tau_n^+(K) \geq K - n + 1$. Thus, the total number of nodes 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. □

6 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.

We again consider the simplified version of the Knapsack problem, where the profit of each item is equal to its weight. Namely, in the *minimization* (n, K)*-Knapsack problem*, input is a sequence $a = (a_1, \ldots, a_n)$ of natural numbers $a_i \leq K$, and the goal is to minimize $\sum_{i \in S} a_i$ over all subsets $S \subseteq [n]$ such that $\sum_{i \in S} a_i > K$. 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 (see Example 7).

Theorem 5 *Every general dynamic BP solving the minimization* (n, K)*-Knapsack problem must have* $\Omega(nK \log K)$ *contacts.*

The proof will use entirely different arguments than those above. Namely, we will use a classical result of Hansel [17] 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. A contact scheme is a nondeterministic branching program (see Sect. 2.3) without rectifiers. Such a scheme is monotone, if it does not have survival tests $x_i = 0$. The idea to use Hansel's result is borrowed from Rychkov [35].

Proof Let $P(x_1, \ldots, x_n)$ 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 $i = 1, \ldots, n/2$, 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

Table 2 How tests $x_1 = d$ and $x_2 = d$ get transformed, when $K = 5$.

d	$x_1 = d$	$x_2 = d$
0	$y_5 = 1$	remove
1	remove	$y_4 = 1$
2	$y_3 = 1$	remove
3	remove	$y_3 = 1$
4	$y_4 = 1$	remove
5	remove	$y_5 = 1$

"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 $D = \{0, 1, \ldots, K\}$. Call a pair $(u, v) \in D^2$ an *even-odd pair*, if

$$u + v > K,\ u \text{ is even and } v \text{ is odd.}$$

Let $S \subseteq D$ be the second half of our domain, that is, $S = \{\lceil K/2 \rceil, \lceil K/2 \rceil + 1, \ldots, K\}$. The proof consists of the following two steps:

(i) Modify $P(x_1, \ldots, x_n)$ 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 $m = |S| = \Omega(K)$ new boolean variables which computes the threshold-2 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. First, contract all contacts making tests $x_i = 0$ for $i \geq 3$. Then remove all contacts making a test:

$x_i = d$ for $d \neq 0$ and $i \geq 3$,
$x_1 = d$ for d odd,
$x_2 = d$ for d even.

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 accepts the input $(u, v, 0, \ldots, 0)$ (hence, $u + v > K$), u is even, and v is odd. That is, P' accepts exactly even-odd pairs. Recall 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

$$x_1 = d \text{ and } x_2 = K - d \text{ for } d \in D \text{ even}$$

by the (boolean) tests

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

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

$$\begin{array}{llll} x_1 = u & \text{or} & x_2 = K-u & \text{if } u \text{ is even,} \\ x_1 = K-u & \text{or} & x_2 = u & \text{if } u \text{ is odd.} \end{array} \tag{4}$$

Claim 4 The monotone boolean contact scheme $Q(y_u \colon u \in S)$ computes the threshold-2 function $Th_2^m(y_u \colon u \in S)$.

Proof 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 $y_u = 1$ and $y_v = 1$ are made. For this, we use the fact that $P'(x_1, x_2)$ accepts the pair (u, v) if and only if 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 and v have different parities, say, u is even and v is odd. Since $u + v > K$, the pair (u, 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 $x_2 = v$ are made. By (4), 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 (4), 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 (4), along the corresponding path in Q only tests $y_v = 1$ and $y_u = 1$ are made, implying that $Q(b) = 1$.

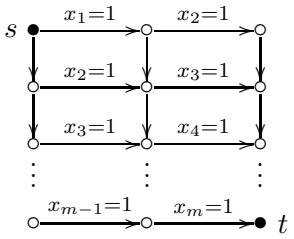
Assume now that b contains exactly one 1 in some position $u \in S$. We have to show that then $Q(b) = 0$.

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, (4) implies that each such test could only be obtained 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, (4) implies that each such test could only be obtained 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 5. □

Remark 5 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 edges of 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.

7 Bounds for Approximation

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

Approximation algorithms for the Knapsack problem use an additional "argmin" or "argmax" feature. In order to implement such algorithms, we can also add these features 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 arrives, the value $P(x)$ is now computed as follows. As before, the value $\mathrm{val}(t_k, x)$ computed at the node t_k on input x is the maximum length of an s-t_k path consistent with x. After that, the program outputs the minimal k for which $\mathrm{val}(t_k, x)$ is at least some given in advance threshold K. That is,

$$P(x) = \arg\min_k \{\mathrm{val}(t_k, x) \colon \mathrm{val}(t_k, x) \geq K\}\,.$$

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 *argmin dynamic BP*. Note that an argmin BP first simultaneously solves N *maximization* problems, and then applies the Argmin operation to their results. The *argmax* dynamic BP works dually: it first solves N *minimization* problems, and then applies the Argmax operation to their results.

Let us consider the *minimization* Knapsack problem with profit-threshold K. That is, given a sequence $p_1, \ldots, p_n$ of profits, and a sequence $s_1, \ldots, s_n$ of sizes of n items, the goal is to minimize $\sum_{i \in S} s_i$ over all $S \subseteq [n]$ such that $\sum_{i \in S} p_i \geq K$.

Proposition 1 *If we have n items, and if the sum of sizes in every input does not exceed t, then the minimization Knapsack problem an be solved by an argmin dynamic BP with at most nt nodes.*

By using argmax dynamic BPs, one can solve also the maximization Knapsack problem by using at most nt nodes, where t is the maximum possible total profit $p_1 + \cdots + p_n$.

Proof As subproblems we take $f(i,k)$ = the maximal total profit for filling *exactly* a capacity k knapsack with some subset of items $1,\ldots,i$. Each base value $f(1,k)$ equals p_1 if $s_1 = k$, and is zero otherwise. The remaining values $f(i,k)$ can be computed using the recursion (1). Since i runs from 1 to n, and k runs from 0 to at most t, we have at most nt subproblems. The algorithm can be converted into a dynamic BP with nt nodes, as shown in Example 7. The BP computes values $f(n,0), f(n,1), \ldots, f(n,k), \ldots, f(n,t)$. Now, apply the argmin operation to get the smallest k for which $f(n,k) \geq K$. □

Theorem 6 *For every $\epsilon > 0$, the minimization (n,K)-Knapsack problem can be approximated with the factor $1+\epsilon$ by an argmin dynamic BP with $O(n^3/\epsilon)$ nodes.*

The same holds for argmax dynamic BPs approximating the maximization (n,K)-Knapsack problem with the factor $1-\epsilon$: just round-down instead of rounding-up.

Proof Associate with every sequence $a = (a_1,\ldots,a_n)$ of natural numbers the scaling factor

$$r = r(a) := \max\left\{1, \frac{\epsilon(a_1 + \cdots + a_n)}{n^2}\right\}.$$

The scaled version of a is the sequence $a' = (a'_1,\ldots,a'_n)$, where $a'_i := \lceil a_i/r \rceil$, and $r = r(a)$ is the scaling factor of a. By the *scaled* Knapsack problem we will mean the minimization Knapsack problem with scaled sizes: compute the minimum of $\sum_{i\in S} a'_i$ over all $S \subseteq [n]$ such that $\sum_{i\in S} a_i \geq K$. Note that the total size $a'_1 + \cdots + a'_n$ of every scaled input does not exceed $t = (a_1 + \cdots + a_n)/r = \lceil n^2/\epsilon \rceil$. By Proposition 1, there is an argmin dynamic BP $P'(x)$ of size at most $nt = O(n^3/\epsilon)$ which solves the scaled 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 each survival test $x_i = d$ by the test $\lceil x_i/r \rceil = d$, where $r = r(x)$. These tests are legal since in an argmin 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 $\lceil x_i/r \rceil = d$ in program P if and only if the scaled item $a'_i = \lceil a_i/r \rceil$ passes the test $x_i = d$ in program P'. Thus, all feasible solutions produced by the program P' on scaled inputs are feasible solutions for non-scaled inputs, and are produced by program P. If $r(a) = 1$, then $a' = a$, and an optimal solution produced by P' on a' is also optimal for a, that is, $P(a) = \mathrm{opt}(a)$. This may be no more the case, if $r(a) > 1$. Let us show that also then $P(a) \leq (1+\epsilon)\mathrm{opt}(a)$ holds.

Let S' be an optimal solution produced by $P'(x)$ on the scaled input a', and let S be an optimal solution for input a; hence, $\mathrm{opt}(a) = \sum_{i\in S} a_i$. The solution S' is also a feasible

solution for a, and is produced by $P(x)$ on input a. Since S' is optimal and S is feasible for a', we have that $\sum_{i\in S'} a'_i \le \sum_{i\in S} a'_i$. Thus,

$$P(a) \le \sum_{i\in S'} a_i \le r \sum_{i\in S'} a'_i \le r \sum_{i\in S} a'_i \le \sum_{i\in S} a_i + r|S| \le \mathrm{opt}(a) + rn\,.$$

Together with $rn = \epsilon(a_1 + \cdots + a_n)/n \le \epsilon \cdot \max_i a_i \le \epsilon \cdot K \le \epsilon \cdot \mathrm{opt}(a)$, the desired upper bound $P(a) \le (1+\epsilon)\mathrm{opt}(a)$ follows. □

For optimization problems whose weights are *integers*, the following simple fact is often utilized to prove (absolute) inaproximabilty results (see, e.g., [16,3]).

Proposition 2 *Suppose we have a minimization problem with integer weights, and let $K > 0$ be the minimal optimal value. If an algorithm approximates the optimum within a factor $< 1 + 1/K$, then it solves the problem exactly.*

In the case of maximization problems, K is the maximal optimal value, and the factor is $> 1 - 1/K$.

Proof 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 larger than $\mathrm{opt}(a) \ge K$. The next best integer solution is $\mathrm{opt}(a) + 1$. So the best possible approximation factor the program can get is $(\mathrm{opt}(a) + 1)/\mathrm{opt}(a) = 1 + 1/\mathrm{opt}(a) \ge 1 + 1/K$. □

Theorem 7 *Let $K \ge 3n$ and $0 < \epsilon < 1/K$. Then every argmin dynamic BP approximating the minimization (n, K)-Knapsack problem with a factor of $1 + \epsilon$ must have at least $n/2\epsilon$ nodes.*

Proof In our case, K is the minimal optimal value, and the factor is $1 + \epsilon < 1 + 1/K$. Thus, by Proposition 2, it is enough to show that every argmin dynamic BP $P(x)$ solving the problem *exactly* must have at least $n/2\epsilon$ nodes. The lower bound of $nK/2$ given in Theorem 4 is proved for 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 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 argmin dynamic BP works just as a usual (maximization) dynamic BP with one source node s and one target node t_K, and the lower bound $nK/2 \ge n/2\epsilon$ of Theorem 4 holds. □

Remark 6 Here is yet another application of Proposition 2. It is known that the Maximum TSP with the triangle inequality can be approximated in polynomial time within the factor of $1 - \epsilon$ for $\epsilon \ge 9/44$ [29]. It is also known that there is a constant $\epsilon \ge 0$ such that factor $1-\epsilon$ cannot be achieved in polynomial time, unless $\mathbf{P} = \mathbf{NP}$ [5]. Proposition 2 can be used to show a much weaker (but absolute) lower bound: no dynamic BP of polynomial size can approximate this problem with a factor of $> 1 - 1/2n$, even if distances are 1 and 2. Indeed, we already know (Theorem 3) that dynamic BPs of exponential size are necessary to solve this problem exactly. Since the maximal optimal value in this case is $K = 2n$, the result follows.

8 Conclusion and Open Problems

In this paper we simulated incremental dynamic programming algorithms by dynamic branching programs (dynamic BP), and proved a matching lower bound on the size (number of subproblems) for the Knapsack problem in the latter model. We also proved lower and upper bounds on the size of dynamic BP approximating this problem, as well as a nontrivial lower bound on the number of wires when there are no consistency conditions on the paths in a BP. In the case when the family of feasible solutions does not depend on the actual weights of data items in the input (see Remark 1), the situation is somewhat easier: here we were able even to give a general lower-bounds criterion (Theorem 1). This immediately implied that Maximum Bipartite Matching problem as well as the Traveling Salesman problem require dynamic BPs of exponential size.

Still, many questions remain open. Say, an interesting numerical problem is to close the gap between n/ϵ and n^3/ϵ for the size of argmin dynamic BPs approximating the Knapsack problem. We conjecture that the upper bound is nearer to the truth. Most of the questions, however, concern proving lower bounds for dynamic BPs extended by various *additional features*. 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.

Late rejections Our consistency condition for decision predicates allows that rejected items can be accepted later, that is, "reject" decisions are revocable. But we do not allowed already accepted items be later rejected. That is, dynamic BPs do not have the "late rejections" feature. In programs with this feature, it is natural to define the solution produced by an s-t path as the set of items on which the last decision was "accept".

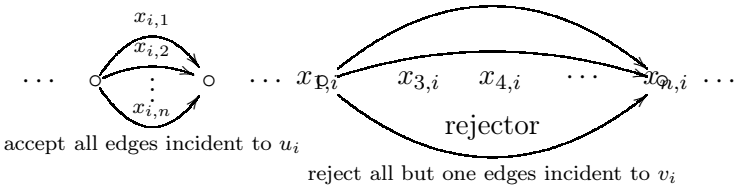
Interestingly, this additional feature (of late rejections) may exponentially decrease the size of dynamic BPs! Let us demonstrate this on the Assignment problem, known also as Maximum Weight Bipartite Matching problem. An input in this problem is a sequence of n^2 non-negative weights of the edges of $K_{n,n}$, and the goal is to compute the maximum weight of a matching. Since the weights are non-negative, optimal solutions are perfect matchings.

We already know (Theorem 2) that this problem requires dynamic BPs of exponential size, even if weights are 0 and 1. We also mentioned that this problem requires prioritized BPs ([13]) and combinatorial dynamic programs ([9]) of exponential size. On the other hand, we have the following.

Proposition 3 *If late rejections are allowed, the Assignment problem on $K_{n,n}$ can be solved by a static BP with $O(n^3)$ nodes.*

Proof The desired static BP consists of a sequence of n "acceptors" followed by n "rejectors". Each acceptor corresponds to one vertex 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 this vertex. All decisions here are "accept". Each rejector corresponds to one vertex 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 edges incident to v. Thus, the j-th of the paths for the i-th

vertex v_i is responsible for variables $x_{1,i}, \ldots, x_{j-1,i}, x_{j+1,i}, \ldots, x_{n,i}$. 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 edges for each vertex on the right part of $K_{n,n}$. Thus, the solution produced by the entire path is a (possibly empty) 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. □

Can non-trivial lower bounds be proved for dynamic BPs with the late rejections feature?

Although the late rejections feature allows for the possibility of exponentially reducing the number of nodes (subproblems), a BP P in this case does not give us an efficient algorithm to actually *compute* the values $P(x)$. Without this "late rejections" feature, one can quickly compute the value $P(x)$ by solving the longest (or shortest) s-t path problem in the underlying acyclic graph. The situation, however, changes drastically if late rejections are allowed: one may then be forced to consider all s-t paths in order to determine an optimal value. Thus, the model of BPs with the late rejection feature is only interesting as a compact *encoding* of optimal solutions.

Null-paths Our consistency condition for survival tests is that along every path, tests on the same data item must have the same outcome. But what can be said about dynamic BPs when "null-paths", i.e., paths with contradicting survival tests on the same data item, are allowed? Such paths contribute nothing to the solution, but it is known that in the case of *boolean* branching programs, the presence of null-paths may exponentially reduce the number of nodes, even if all consistent paths are required to be read-once [24, Sect. 13]. To understand the role of null-paths—even under the read-once restriction on consistent paths—is a challenge in boolean circuit complexity: no strong lower bound is known in this model, one of the weakest nondeterministic models of computation.

Still, in dynamic BPs the domain is much larger than $D = \{0, 1\}$, and there is a hope that strong enough lower bounds can be proved; Theorem 5 partly confirms this. An interesting problem also is to understand whether null-paths can substantially reduce the size of dynamic BPs 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 [32,10,22]. For branching programs computing functions $f : D^n \to \{0,1\}$ on 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, and there are no restrictions on inconsistent paths [2,7,23]. 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 and, apparently, on the total sum of weights of all n items. 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?" Thus, in this case, it would be desirable to have tests of the form $t_e(x_i, x_j)$. In Example 4 we have shown that the Interval Scheduling problem can be solved by a small dynamic BP, by viewing data items as *pairs* of jobs. With more general survival tests, we could take items be just jobs. Can some non-trivial lower bounds be proved for dynamic BPs with more general survival tests?

Relaxed responsibility In the model of dynamic BPs, 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. One can, however, try to relax this, and allow the variable x_i, which a contact must be responsible for, to depend on the actual input x.

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

We already mentioned models where the responsibility depends on the actual input: prioritized BTs [3] and prioritized BPs [13]. Even with this relaxed responsibility, strong lower bounds were proved in these models. So, one could allow such a relaxed responsibility also in dynamic BPs. Namely, we could allow that each contact e has a total ordering $\prec_e$ of the set D of all data items. When an input $(x_1, \ldots, x_n) \in D^n$ arrives, the contact e is responsible for that item x_i which comes as the first in the ordering $\prec_e$ of D. Thus, on another input $(y_1, \ldots, y_n)$, the same contact may be responsible for another data item y_j, $j \neq i$. Can non-trivial lower bounds be proved for dynamic BPs with such a relaxed responsibility?

Relation to prioritized branching programs Unlike dynamic BPs, the model of prioritized BPs (pBP) 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 $\prec_v$ 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 in 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 arrives. 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 $\prec_v$. 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.

The value $P(x)$ on input x is computed backwards, by inductively assigning values to nodes of P_x. The value of each sink node v is the value $\mathrm{val}(v)$ assigned by the algorithm (it does not depend 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 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. Using interesting probabilistic arguments, it is proved in [13] that any boolean pBP deciding whether a given bipartite $n \times n$ graph contains a perfect matching must have size $2^{\Omega(n^{1/8})}$.

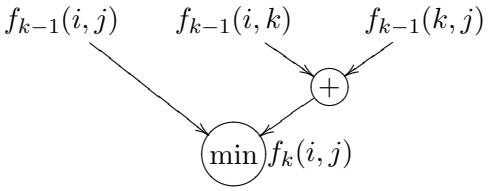
The model of pBT (prioritized branching *trees*), introduced earlier in [3], has a restriction that the underlying graph of $P(x)$ must be a tree. But it has an additional feature that the transition function g_v as well as the ordering $\prec_v$ may depend on the items and decisions about them made along the (unique) path to the node v. Thus, even though every pBT can be viewed as a pBP, as soon as one exploits the merging aspect of pBPs, one loses some of the power in the pBT model. In [3] it is, in particular, 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 $(1-\epsilon)$-approximated by a pBT of size $(1/\epsilon)^2$, and that any such pBT must have size $(1/\epsilon)^{1/3.17}$.

It would be interesting to understand whether one of the models—prioritized and dynamic BP—subsumes the power of the other. As mentioned above, boolean pBPs compute boolean functions. It would be therefore also interesting to understand what class of boolean branching programs such pBPs correspond to.

Non-incremental DP algorithms and tropical 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.

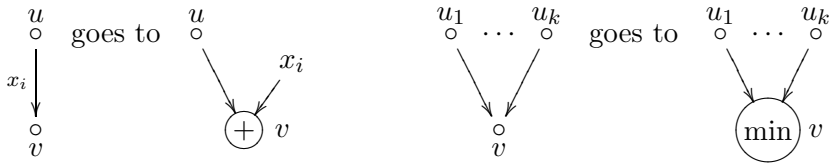
A prominent example of a "non-incremental" DP algorithm, which does not directly translate to a dynamic BP, is the Floyd–Warshall algorithm for the all-pairs shortest path problem. As subproblems it takes $f_k(i, j) =$ the length of a shortest paths from i to j that only uses vertices $1, \ldots, k$ as inner nodes. We set $f_0(i, j) =$ the length of the edge (i, j). The DP recursion is then $f_k(i, j) = \min\{f_{k-1}(i, j), f_{k-1}(i, k) + f_{k-1}(k, j)\}$.

This algorithm—as well as many other DP algorithms whose recurrence relation uses Plus and Min/Max operations only—can, however, be simulated by *tropical circuits*[3], i.e., by conventional circuits with fanin-2 Min and Plus (or Max and Plus) gates. Here is a fragment of such a circuit when implementing the Floyd–Warshall algorithm:



Inputs are the n^2 variables x_{ij}, one for each edge (i,j); their values are the real-valued lengths of edges. The entire circuit for the Floyd–Warshall algorithm has depth $2n$ and size (number of gates) $O(n^3)$. On the other hand, results of [28,15,30] imply that every tropical circuit for the all pairs shortest path problem must have $\Omega(n^3)$ Plus-gates (see also [1, pp. 204–206]). Thus, when restricted to Min and Plus operations, the Floyd–Warshall algorithm is optimal!

Much fewer is known about the tropical circuit complexity of the s-t shortest path problem. In particular, it is not known whether this problem also requires tropical circuits of size $\Omega(n^3)$. Static BPs constitute a special case of tropical circuits, where one of the two inputs to a Plus-gate must be a variable:



We already know (see Example 1) that the Bellman–Ford–Moore algorithm for s-t shortest path problem translates to a static BP with $O(n^2)$ nodes and $O(n^3)$ wires. Does this number of nodes/wires is also necessary? In other words, is the Bellman–Ford–Moore algorithm optimal at least in the class of incremental DP algorithms?

Lower bounds for tropical circuits can be obtained by proving corresponding lower bounds for monotone *boolean* circuits. To see this, suppose we have a tropical circuit solving a 0/1 optimization problem $f(x) = \min_{S\in\mathcal{F}} \sum_{i\in S} x_i$. In particular, the circuit must solve the problem on all inputs $x \in \{0,\infty\}^n$. The mapping $h : \{0,\infty\} \to \{0,1\}$ given by $h(0) = 1$ and $h(\infty) = 0$ is a homomorphism from the semiring $(\{0,\infty\}, \min, +, \infty, 0)$ to the boolean semiring $(\{0,1\}, \vee, \wedge, 0, 1)$: $h(\min\{x,y\}) = h(x) \vee h(y)$ and $h(x+y) = h(x) \wedge h(y)$. So, if we replace each Min-gate by a logical OR, and each Plus-gate by a logical AND, then the resulting monotone boolean circuit computes the boolean function $f_B(x) = \bigvee_{S\in\mathcal{F}} \bigwedge_{i\in S} x_i$.

Consider, for example, the Lightest Triangle problem: given non-negative weighting of the edges of K_n, find the weight of a lightest triangle. This problem can be solved by

[3] The Min-Plus semiring $(R+, \oplus, \otimes, \infty, 0)$ with operations $x \oplus y = \min(x,y)$ and $x \otimes y = x+y$, is called "tropical" in honor of Imre Simon who lived in Sao Paulo (south tropic). Tropical algebra and tropical geometry are intensively studied topics in mathematics.

a tropical circuit of size $O(n^3)$, by just trying all triangles. On the other hand, a lower bound on the monotone circuit complexity of detecting triangle-freeness ([24, Sect. 9.5.1]) immediately implies that every tropical circuit for this problem must have $\Omega(n^3/\log^4 n)$ gates.

The boolean version of the s-t shortest path problem is the boolean function STCONN which accepts a graph if and only if there is a path from s to t. It is known that this function requires monotone circuits of depth $\Omega(\log^2 n)$ [26], but unfortunately, no non-trivial lower bounds on the *size* are known.

It is not difficult to show that, when restricted to the boolean domain $D = \{0, 1\}$, tropical circuits are almost as powerful as monotone boolean circuits: just replace each AND gate $u \wedge v$ by a Min gate $\min(u, v)$, and each OR gate $u \vee v$ by $\min(1, u + v)$. The point, however, is that tropical circuits must work correctly on much larger domains $D \supset \{0, 1\}$. This is why lower bounds for tropical circuits do not translate to lower bounds for monotone boolean circuits. And indeed, when proving that any tropical circuit computing the "min-plus" product of two $n \times n$ matrices requires $\Omega(n^3)$ gates, Kerr [28] essentially uses the fact that $D \supseteq \{0, 1, 2\}$. In the context of understanding the limitations of DP algorithms, this fact is a "good news": it may be easier to fool small circuits when the domain is large. Moreover, the Plus-operation is not idempotent, and the Min-operation does not distribute over Plus.

Acknowledgments

I am thankful to Allan Borodin, Joshua Buresh-Oppenheim, Russell Impagliazzo, Gerhard Paseman, and Georg Schnitger for enlightening discussions. My thanks to anonymous referees for stimulating comments; I am especially obligated to one of them for suggesting to use even-odd pairs in the proof of Theorem 5.

References

1. A. Aho, J. Hopcroft, and J. Ullman: *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974 23
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, and A. Magen: Toward a model for backtracking and dynamic programming. Computational Complexity 20(4), 679–740 (2011) 2, 11, 19, 22, 23
4. S. Angelopoulos and A. Borodin: The power of prioritized algorithms for facility location and set cover. Algorithmica 40(4), 271–291 (2004) 2
5. S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy: Proof verification and the hardness of approximation problems. J. of ACM 45(3), 501–555 (1998) 20
6. A. Barvinok, E. Kh. Gimadi, and A. I. Serdyukov: The maximum traveling salesman problem. In: The Traveling Salesman problem and its variations, G. Gutin and A. Punnen, eds., 585–607, Kluwer (2002) 9

7. P. Beame, M. Saks, X. Sun, and E. Vee: Time-space trade-off lower bounds for randomized computation of decision problems. J. of ACM 50(2), 154–195 (2003) 22
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) 2
9. 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) 5, 13, 21
10. A. Borodin, A. Razborov, and R. Smolensky: On lower bounds for read-k times branching programs. Computational Complexity 3, 1–18 (1993) 22
11. A. Borodin, M. N. Nielsen, and C. Rackoff: (Incremental) prioritized algorithms. Algorithmica 37(4), 295–326 (2003) 2
12. A. Borodin, J. Boyar, K. S. Larsen, and N. Mirmohammadi: Priority algorithms for graph optimization problems. Theor. Comput. Sci. 411(1), 239–258 (2010) 2
13. J. Buresh-Oppenheim, S. Davis, R. Impagliazzo: A stronger model of dynamic programming algorithms. Algorithmica 60(4), 938–968 (2011) 2, 11, 13, 21, 22, 23
14. S. Davis and R. Impagliazzo: Models of greedy algorithms for graph problems. Algorithmica 54(3), 269–317 (2009) 2
15. M. E. Furman: Application of a method of fast multiplication of matrices in the problem of finding the transitive closure of a graph. Soviet Math. Doklady 11(5), 1252 (1970) 23
16. M. R. Garey and D. S. Johnson: Strong NP-completeness results: motivation, examples, and implications. J. of ACM 25, 499–508 (1978) 19
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), 6037–6040 (1964), in French 3, 16
18. M. Held and R. M. Karp: A dynamic programming approach to sequencing problems, SIAM J. on Appl. Math. 10, 196–210 (1962) 9
19. P. Helman: A common schema for dynamic programming and branch and bound algorithms. J. of ACM 36(1), 97–128 (1989) 2
20. P. Helman and A. Rosenthal: A comprehensive model of dynamic programming. SIAM J. Algebr. Discrete Methods 6, 319–334 (1985) 2
21. M. Jerrum and M. Snir: Some exact complexity results for straight-line computations over semirings. J. of ACM 29(3), 874–897 (1982) 6
22. S. Jukna: A note on read-k times branching programs. RAIRO Theoret. Informatics and Appl. 29(1), 75–83 (1995) 3, 22
23. S. Jukna, A nondeterministic space-time tradeoff for linear codes. Inf. Process. Lett. 109(5), 286–289 (2009) 22
24. S. Jukna: *Boolean Function Complexity: Advances and Frontiers*. Springer, 2012 3, 6, 21, 24
25. S. Jukna and A. A. Razborov: Neither reading few bits twice nor reading illegally helps much. Discrete Appl. Math. 85(3), 223–238 (1998) 12
26. M. Karchmer and A. Wigderson: Monotone circuits for connectivity require super-logarithmic depth. SIAM J. Discrete Math. 3(2), 255–265 (1990) 24
27. R. Karp and M. Held: Finite state processes and dynamic programming. SIAM J. Appl. Math. 15, 693–718 (1967) 2
28. L. R. Kerr: The effect of algebraic structure on the computation complexity of matrix multiplications. PhD Thesis, Cornell Univ., Ithaca, N.Y. (1970) 23, 24
29. L. Kowalik and M. Mucha: 35/44-approximation for asymmetric Maximum TSP with triangle inequality. Algorithmica 59(2), 240–255 (2011) 20

30. I. Munro: Efficient determination of the transitive closure of a directed graph. Inf. Process. Lett. 1(2),56–58 (1971) 23
31. E. I. Nechiporuk: On a boolean function. Soviet Math. Dokl. 7(4), 999–1000 (1966) 6
32. 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
33. M. Poloczek: Bounds on greedy algorithms for MAX SAT. In: Proc. of 19th Ann. European Symp. on Algorithms, C. Demetrescu and M. M. Halldórsson (Eds.). LNCS, vol. 6942, pp. 37–48. Springer, Berlin (2011) 2
34. O. Regev: Priority algorithms for makespan minimization in the subset model. Inf. Process. Lett. 84(3), 153–157 (2002) 2
35. K. L. Rychkov: On the complexity of generalized contact circuits. Diskretn. Anal. Issled. Oper. 16(5), 78–87 (2009), in Russian 16
36. I. Wegener: *Branching Programs and Binary Decision Diagrams*. SIAM, 2000 3
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) 2