Algorithmica (2011) 60: 938–968
DOI 10.1007/s00453-009-9385-1

A Stronger Model of Dynamic Programming Algorithms

Joshua Buresh-Oppenheim · Sashka Davis · Russell Impagliazzo

Received: 2 October 2009 / Accepted: 22 December 2009 / Published online: 6 January 2010

Abstract We define a formal model of dynamic programming algorithms which we call Prioritized Branching Programs (pBP). Our model is a generalization of the BT model of Alekhnovich et al. (IEEE Conference on Computational Complexity, pp. 308–322, 2005), which is in turn a generalization of the priority algorithms model of Borodin, Nielson and Rackoff. One of the distinguishing features of these models is that they not only capture large classes of algorithms generally considered to be greedy, backtracking or dynamic programming algorithms, but they also allow characterizations of their limitations. Hence they give meaning to the statement that a given problem can or cannot be solved by dynamic programming. After defining the model, we prove three main results: (i) that certain types of natural restrictions of our seemingly more powerful model can be simulated by the BT model; (ii) that in general our model is stronger than the BT model—a fact which is witnessed by the classical shortest paths problem; (iii) that our model has very real limitations, namely that bipartite matching cannot be efficiently computed in it, hence suggesting that there are problems that can be solved efficiently by network flow algorithms and by simple linear programming that cannot be solved by natural dynamic programming approaches.

S. Davis research partially supported by NSF Awards CCR-0313241 and CCR-0515332. Views expressed are not endorsed by the NSF.
R. Impagliazzo research partially supported by NSF Awards CCR-0313241 and CCR-0515332. Views expressed are not endorsed by the NSF.

J. Buresh-Oppenheim
Akamai Technologies, Cambridge, USA
e-mail: bureshop@gmail.com

S. Davis (✉) · R. Impagliazzo
University of California, San Diego, USA
e-mail: sdavis@cs.ucsd.edu

R. Impagliazzo
e-mail: russell@cs.ucsd.edu

Keywords Dynamic programming · Algorithmic paradigms · Priority algorithms

1 Introduction

Formalizing algorithmic paradigms can improve our methodology as computer scientists in many ways. If we were able to say precisely what we mean by terms like greedy algorithms, dynamic programming, network flow algorithms, local-search, etc., then, when confronted with a new problem, we could hope to either find an algorithm for it by placing it in one of these models, or show that it doesn't fit the model and move on to the next model. If a problem doesn't fit any of our common paradigms, then it is in some sense a problem we don't already know how to solve. From this point of view, knowing how these models relate to each other and whether they always give rise to tractible algorithms is important. Of course, this line of research can also help with our theoretical agendas. For example, it seems much easier to show that *NP* problems cannot be efficiently solved by common types of algorithms than it is to separate it from all of P.

Much work has been conducted along these lines. Just to name a few, [16] formalized various local search paradigms, [4, 5] looked at general methods for generating linear relaxations for boolean optimization problems (specifically vertex cover), and [10] defined a general scheme for branch-and-bound algorithms and applied it to Knapsack.

When it comes to dynamic programming, the body of work is particularly dense. Some of the original formalizations of dynamic programming are from Bellman [6] and Karp and Held [15]. Helman and Rosenthal, among others, refine and examine these models [13, 14, 19]. These models are extremely expressive (most problems can be formulated to fit the context), but also extremely powerful. That is, it is difficult to identify problems that can be formulated appropriately but are hard to compute. Nevertheless, Helman and Rosenthal succeed in getting weak lower bounds for problems such as shortest directed acyclic path in staged graphs, iterated matrix multiplication and optimal binary search tree. Woeginger [20], on the other hand, defines a model called DP-simple in which only problems susceptible to a dynamic programming FPTAS can be expressed. An interesting goal is to try to maximize expressibility while maintaining a notion of computational limitation.

We build on the framework established by Borodin, Nielsen and Rackoff [7], called priority algorithms, and subsequently studied and generalized by [1, 2, 8, 9, 12, 17], among others. Various versions of these models succeed in capturing large classes of greedy, dynamic programming, backtracking, and primal-dual algorithms, but they also allow either impossibility results or strong complexity lower bounds to be proven for natural problems that should not intuitively fit the models. They even have the potential to make fine distinctions between related techniques, and to get tight bounds on the trade-off between resources and solution quality in their models.

One modest, but lucid example comes from [2], which first made the connection between priority algorithms and dynamic programming. The best known algorithm for interval scheduling on m machines is due to [3] and uses network flow to achieve a running time of $O(n^2 \log n)$. Given the elegant and fast dynamic programming algorithm for interval scheduling on one machine, a natural question is whether dynamic

programming can offer a solution for m-machines that is simpler than [3], but doesn't sacrifice running time (or maybe even improves it). [2] prove that the complexity of any algorithm for this problem in their basic model, which captures, among many other things, the 1-machine algorithm, grows as $\Omega(n^m)$ (as we shall see, the complexity measure of this model is fundamentally related to the number of partial solutions that need to be maintained during the course of the algorithm). This suggests that the answer to the question is no.

In this paper, we introduce a new model that simulates the previous formal models of priority algorithms [7] and the BT model for backtracking and dynamic programming algorithms introduced by [2] (we rename the BT model *pBT* for prioritized Branching Tree, which is more suggestive of its syntactic properties and makes the connection to priority algorithms. BT originally stood for backtracking). Although a large class of dynamic programming algorithms were known to fit the pBT model, others, like the classic Bellman-Ford algorithm for the single source shortest path problem in graphs with negative weights edges, could not be seen to fit the model (and in fact, we prove here that it doesn't, at least without some extensions). Our model, pBP, for prioritized Branching Program, is meant to capture the defining characteristics of a larger range of classical dynamic programming algorithms.

Priority algorithms were defined as a formal framework for greedy algorithms. In this framework an instance is represented as a set of items. The algorithm orders the data items from the instance according to a priority function that is independent of the instance and, once it observes a data item, it has to commit to a decision which is irrevocable. For example consider the vertex cover problem. The graph can be represented as a set of data items, each one containing the name and adjacency list of one vertex. The algorithm begins with no knowledge of the contents of these items, but it can provide a general rule for ordering the items. In each step, the top item in the ordering is revealed and the algorithm makes a decision about it (e.g. including or excluding the current vertex in the vertex cover). This decision-making process restricts the algorithm to maintain a *single partial solution* during its computation. The pBT model generalizes the priority framework by allowing the algorithm to maintain a *tree of partial solutions*. A natural measure of the complexity of a pBT algorithm is the width of its tree. Various levels of adaptivity in how the algorithm chooses its ordering rule define a hierarchy of models: fixed, adaptive and fully adaptive. [2] were able to show a variety of non-trivial lower bounds. They showed an exponential separation between the power of fixed and adaptive pBT algorithms, they also proved exponential lower bounds for 3-SAT for the fully-adaptive model. Clearly, priority algorithms can be simulated without overhead by restricted pBT algorithms whose computation tree is a line and [2] showed a separation between width-1 and width-2 algorithms.

Our model, like the pBT model, maintains multiple partial solutions, but also allows memoization, which seems essential to the concept of dynamic programming. Consider the following intuitive definition [11]: "Dynamic-programming algorithms typically take advantage of overlapping subproblems by solving each subproblem once and then storing the solution in a table where it can be looked up when needed.... There is a variation of dynamic programming that offers the efficiency of the usual dynamic-programming approach while maintaining a top-down strategy.

The idea is to *memoize*." The Prioritized Branching Programs (pBP) algorithms combine the power of *branching* with the power of *memoization*. Branching allows multiple partial solutions to be maintained, while *merging* allows different branches of the computation to memoize the solution to common subproblems for later reuse (so in a sense $pBP = BRANCH + MERGE$). Any discrete optimization problem can be solved by a pBP algorithm, although possibly not an efficient one. The computation of a pBP algorithm consists of three phases. On a given input instance the algorithm generates a directed acyclic graph top-down successively as it sees more and more of the input. It then traverses the DAG bottom-up to obtain the value of the best solution it computed, and then finds the actual solution with one more top-down traversal. The number of states in the computational DAG (size) is closely related to the quality of the solution an algorithm finds. In particular, if we allow an exponential size we can solve any optimization problem.

Branching was shown to give pBT algorithms extra power which separates them from priority algorithms. This raises the analogous question for pBT versus pBP. Does merging make pBP more powerful than pBT? What kind of problems can a pBP algorithm solve efficiently? What are the limitations of the pBP model? These are the kinds of questions we address in our paper.

Our Results

Simulations Although merging is expected to make the model stronger we show that a natural subclass of pBP algorithms can be simulated by pBT algorithms (see Sect. 3). Essentially we show that full adaptivity is required to see the benefit of merging.

Shortest Paths We show that the promise problem of finding shortest paths in graphs with negative weights but no negative cycles can be solved by a pBP algorithm using $O(n^3)$ states: on promise instances the algorithm outputs the shortest path while on graphs containing negative weight cycles the algorithm would output some arbitrary set of edges, still using at most $O(n^3)$ states. On the other hand, we define a new and simple technique for proving negative results for pBT algorithms and use it to show that every pBT algorithm for the promise problem requires width $\Omega(2^{n^{1/9}})$ on some instance, although it could be one containing a negative weight cycle. Although the two results do not exactly separate the pBP and pBT models they give strong evidence that the pBP model properly contains the pBT model. (See Sects. 2.3, where we fully define the problem and the give the upper bound, and 4, where we give the lower bound.)

pBP Lower Bounds We develop a general technique for proving lower bounds on the number of states required by a natural restriction of pBP algorithms that compute using limited numeric precision, and instantiate it to prove an exponential lower bound on the size of any such pBP algorithm for maximum bipartite matching. Informally, it seems unusual for a dynamic programming algorithm to make use of numeric precision well beyond that of the optimization function defining the problem. In light of this, our result suggests that, not only can we not use dynamic programming

to improve the involved (though still polytime) network flow or linear programming algorithms for this problem, but we can't even approach the same running time (see Sect. 5).

2 The pBP Model: Definitions and Examples

We begin with formal definitions and then illustrate them with examples. Our definitions of problems and instances follow those of [7]. We will define infinite problems $\mathcal{P}$ as a collection of finite problems $\{\mathcal{P}_n | n \in \mathbb{N}\}$, where n is a size parameter. When n is understood, we will drop the subscripts. Algorithms will also be non-uniform, in that the size parameter is known to the algorithm, and we will not enforce any connection between the algorithms for different size parameters n (although we note that all upper bounds we present are highly uniform).

For each n, the problem $\mathcal{P}_n$ is specified by the following ingredients:

1. A universe of possible *data items*, $D = D_n$. There is a special element $end \notin D$ which is a marker to halt the computation before all data items are seen. So in a sense *end* signals the absence of unseen items.
2. A collection of *valid instances*, where each valid instance I is viewed as a set of data items, $I \subseteq D$. Each valid instance contains *end*.
3. A set $\Sigma = \Sigma_n$ of *decision options* for each data item.
4. An objective function $F = F_n$ (technically, a family of objective functions, one for each possible instance size, k, of $\mathcal{P}_n$) which takes an input of the form $(d_1, \sigma_1), \ldots, (d_k, \sigma_k)$ with $d_i \in D$ and $\sigma_i \in \Sigma$ and returns a real number, infinity, or negative infinity. (Without loss of generality, we can assume that the objective is to maximize F; we can model minimization by maximizing $-F$. We also usually assume $\{d_1, \ldots, d_k\}$ is a valid instance; however, we can give F an arbitrary value such as 0 if not. We can model search and decision problems by picking F to be a Boolean function which has value 1 on feasible outputs.)

Given an instance of the search/optimization problem $I \subset D_n$ the problem is to find a solution, which is an assignment of an option $\sigma_i \in \Sigma$ to each data item $d_i \in I - \{end\}$. So a *solution* for $I = \{d_1, \ldots, d_k, end\}$ is a set of the form $\{(d_i, \sigma_i) | i = 1, \ldots, k\}$, where each $\sigma_i \in \Sigma\}$, and we wish to, given I, find a solution such that $F((d_1, \sigma_1), (d_2, \sigma_2), \ldots, (d_k, \sigma_k))$ is maximized.

For example, in the maximum bipartite matching problem, the universe D_n could be all possible edges from a bipartite graph with left nodes $u_1, \ldots, u_n$ and right nodes $v_1, \ldots, v_n$, so $D_n = \{(u_i, v_j) | 1 \leq i, j \leq n\}$. Any subset of these edges would be a valid instance. Σ would be $\{accept, reject\}$, representing the choice to either include an edge in the matching or not. The additional item *end* has no choices associated with it, and represents terminating the algorithm when all edges have been labeled. F_n would return the number of accepted edges if they form a matching, and 0 otherwise.

2.1 Definition of a General pBP Algorithm

In the following, since we allow algorithms to be non-uniformly tuned to the size parameter, we omit this parameter as a subscript.

For any $\mathcal{P}$, a pBP algorithm specifies $\mathcal{S}$, a set of *computation states* (possibly infinite) and $\mathcal{T} \subset \mathcal{S}$, a set of *terminal states*. There is a special empty state $S_0 \in \mathcal{S}$, which is the initial (start) state of any pBP algorithm. (Note that $\mathcal{S}$ and $\mathcal{T}$ are problem-specific, but not instance-specific. Intuitively, one can think of each state $s \in \mathcal{S}$ as representing a "sub-problem" to be solved recursively, with the terminal states as the "base cases" of the recursion. The sub-problem might not be determined by s alone, but also by the unseen or unremembered parts of the input; e.g., S_0 always represents "The complete instance, whatever it is". Alternatively, like in a memoized dynamic programming algorithm, one can think of states as encoding the class of partial solutions to the problem that cause the state to be reached.) Let $\mathcal{M}(\mathbb{R})$ be the set of all monotone functions from $\mathbb{R} \cup \{\infty, -\infty\}$ to $\mathbb{R} \cup \{\infty, -\infty\}$. Let $\mathcal{O}(D)$ be the set of all total orderings of D.

A pBP algorithm $\mathcal{A}$ for a given optimization problem $\mathcal{P} = (D, \Sigma, F)$ is defined by specifying the set of states and three components for each state. In addition, it can also specify $\sigma_{def} \in \Sigma$, a default option. Let $s \in \mathcal{S}$ be any state, then the algorithm defines:

1. A priority ordering,

$$\pi_s \in \mathcal{O}(D \cup \{end\}).$$

   This is used to determine which data item is branched on in this state.
2. A state transition function

$$g_s : (D \cup \{end\}) \times \Sigma \mapsto \mathcal{S} \times \mathcal{M}(\mathbb{R}) \cup \{\perp\}.$$

   If d, σ maps to s', f, we think of this as defining a directed edge (s, s') labeled by (d, σ, f). We insist that the graph induced on $\mathcal{S}$ by $\{g_s\}$, which we call $\mathcal{DAG}_{\mathcal{A}}$, be a (multi)dag. As we shall see later, this requirement is usually ensured by imposing a natural layered structure on the graph. We can interpret such an edge intuitively as: if, recursively, we find a value v solution for the sub-problem at s', we can obtain a solution to the sub-problem at s of value $f(v)$ by appending (d, σ). Also, any transition where $d = end$ must go to some $s' \in \mathcal{T}$.
3. For $s \in \mathcal{T}$, the algorithm defines a value

$$v_s \in \mathbb{R} \cup \{\infty, -\infty\}.$$

   Intuitively, this represents "The value returned in the base case s".

The computation of a pBP algorithm on a given instance traces out a subgraph of the above graph. For a pBP algorithm $\mathcal{A}$, we define the computation of $\mathcal{A}$ on instance $I \subset D$ as follows. A pBP computation has three stages: build the (multi)dag, output the value of the best solution, and output the solution itself.

Stage I Build the pBP DAG.

The algorithm $\mathcal{A}$ builds $\mathcal{DAG}_{\mathcal{A}}(I)$ top down. Let $\mathcal{S}_1 = \{S_0\}$ and assume that at step $i \in 1, 2, 3 \ldots$ we have a set of frontier states $\mathcal{S}_i$. For each node $s \in \mathcal{S}_i - \mathcal{T}$, do the following: let $d \in I$ be the first item in I according to the ordering π_s. For each $\sigma \in \Sigma$, check if $g_s(d, \sigma) = \perp$; if not, say it equals

(s', f). Then, put s' in $\mathcal{S}_{i+1}$ and include both s' and the edge (s, s') labeled by (d, σ, f) in $\mathcal{DAG}_{\mathcal{A}}(I)$. It is the algorithm's responsibility to ensure that for some finite i, $\mathcal{S}_i$ is empty and that all sinks in the graph are in $\mathcal{T}$. Note that the only source in the graph is S_0.

Stage II Compute the value of the *best* solution.

The second stage begins after the $\mathcal{DAG}_{\mathcal{A}}(I)$ is fully defined. It determines the value of the optimal solution. This stage traverses the $\mathcal{DAG}_{\mathcal{A}}(I)$ bottom up and successively computes values for each state. The value of the start state S_0 will be the value of the algorithm's solution. First, consider any sink s in $\mathcal{DAG}_{\mathcal{A}}(I)$. Since $s \in \mathcal{T}$, the algorithm defines a value v_s. We assign this value to s. Now consider any s in $\mathcal{DAG}_{\mathcal{A}}(I)$ that does not yet have a value but such that all of its children do. Let the outdegree of s be k and let $s_1, \ldots, s_k$ be its (not necessarily distinct) children. They have assigned values $val(s_1), \ldots, val(s_k)$. Finally, assume that the edge (s, s_i), $i \in [k]$, is labeled by the monotone function f_i. Now we compute $val(s)$ as

$$\max\{f_1(val(s_1)), \ldots, f_k(val(s_k))\}.$$

When computing the value of each s, we also remember which of its outgoing edges contributed the maximum value by marking that edge. If there is a tie, it is broken arbitrarily; that is, the algorithm has no control over which of the maximum-value edges is chosen. Hence there will be one marked edge out of every non-sink in $\mathcal{DAG}_{\mathcal{A}}(I)$.

Stage III Recover the *best* solution.

The third stage recovers the actual solution by traversing $\mathcal{DAG}_{\mathcal{A}}$ top down by following the marked directed path from S_0 until a leaf state is reached. This path is well-defined and gives, for each of its edges, an assignment of a decision to an item. This partial assignment is then extended to a complete assignment by assigning any unlabeled item the default label, σ_{def}. The algorithm must ensure that this assignment is consistent (in that the same data item is not assigned different options), otherwise it outputs FAIL, and that when the problem's optimization function $F_{\mathcal{P}_n}$ is applied to this assignment, it yields the same value that the algorithm reported in Stage II.

The following definition recasts the pBT model in the above framework.

Definition 1 ([2]) A prioritized branching tree (pBT) algorithm (called BT in [2]) is a pBP algorithm $\mathcal{A}$ such that $\mathcal{DAG}_{\mathcal{A}}$ is a tree and all functions labeling its edges are the identity. In this case, we often denote $\mathcal{DAG}_{\mathcal{A}}(I)$ by $\mathcal{T}_{\mathcal{A}}(I)$.

It is clear that the pBP model subsumes the pBT model.

How does this model capture some notion of dynamic programming? Intuitively, think of each node in the pBP DAG as representing a subproblem of the original problem, whose optimal solution will tell us something about the optimal solution to the whole problem. The descendents of that node comprise the computation that will solve the subproblem and the edges flowing into that node define how the solution to

that subproblem will be used to solve the entire problem. In the typical implementation of a dynamic programming algorithm, solutions to subproblems are stored in a table. Think of phase 1 of the pBP computation as defining the relationship among the cells of this table and phase 2 as actually computing the cells of the table in the order in which those computations will be needed. Phase 3 simply traces back to recover the description of the optimal solution as most dynamic programming algorithms do after they are finished computing the entire table. We will see how this intuition[1] allows us to formalize many known dynamic programming algorithms in this model.

Just as in the pBT case, a pBP algorithm can solve any optimization problem by considering the items in some fixed order and exploring each possible assignment of decisions to those items on a separate path. So, similarly to the pBT case, the complexity measure of interest will be the number of states used in any execution of the algorithm. More formally, define

$$size_{\mathcal{A}_n} = \max\{|\mathcal{DAG}_{\mathcal{A}}(I)| : I \subset D_n, I \text{ is a valid instance of } \mathcal{P}_n\},$$

where $|G|$ denotes the number of states in G. Call a family of pBP algorithms $\{\mathcal{A}_n\}$ *polynomial* if there exists a polynomial p such that $size_{\mathcal{A}_n} \leq p(n)$ for all n.

We stress that size is the only complexity restriction on these algorithms. In particular, the functions g_s associated with each state in $\mathcal{S}$ and the monotone functions labeling the edges in the dag are totally arbitrary and need not even be computable (although, for all the standard dynamic programming examples where we formalize a dynamic programming algorithm in the pBP model, they will be of low complexity). The only things stopping the algorithm, then, from being omnipotent are the information theoretic constraints: decisions are based only on the part of the input seen so far; and if multiple paths arrive at the same state, the algorithm cannot "know" which of these paths it arrived by. It is very important that we force a node to commit to how its eventual value will contribute to the ultimate solution before we allow that node to see any new items.

2.2 Submodels of the pBP Model

The pBP model defined above is quite general. Here we will consider several natural ways to refine the model. These refinements will help us to classify more precisely various dynamic programming algorithms.

The first four restrictions deal with how many times an algorithm may view the same data item on a single path and what it may do with that data item. Any valid algorithm in the model must return a consistent path, but we may want a stronger consistency requirement.

Definition 2 (Read-Once) Consider any path in the computation *DAG* from the root state to a terminal state. Suppose each data item appears at most once on this path, then this path satisfies the Read-Once (RO) property.

[1] Perhaps it is more accurate to say that our model resembles the recursive-memoization implementation of dynamic programming in that we compute solutions only to those subproblems which we definitely need and the set of states reachable in the DP dag represent exactly those subproblems that are needed during the computation.

Definition 3 (Path Consistency) Call a path from the root state to a terminal state of the DAG *consistent* if for each data item $d \in \mathcal{D}$ there are not two edges along the path labeled $(d, \sigma, \ldots)$ and $(d, \sigma', \ldots)$, respectively, where $\sigma \neq \sigma'$.

Definition 4 (Syntactic Consistency) We call a pBP algorithm $\mathcal{A}$ syntactically consistent if, for every instance I, every path from the root node S_0 to a terminal state in $\mathcal{DAG}_{\mathcal{A}}(I)$ is consistent.

Definition 5 (Semantic Consistency) We call a pBP algorithm $\mathcal{A}$ semantically consistent if, for every valid instance I, the optimal path from the root node S_0 to a terminal state in $\mathcal{DAG}_{\mathcal{A}}(I)$ is consistent.

Definition 6 (Syntactic RO Property) We call a pBP algorithm $\mathcal{A}$ syntactically RO if, for every instance I, every path from the root node S_0 to a terminal state in $\mathcal{DAG}_{\mathcal{A}}(I)$ is RO.

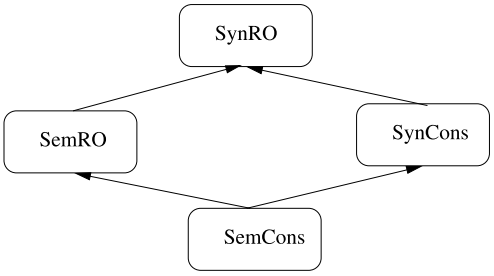
Definition 7 (Semantic RO Property) We call a pBP algorithm $\mathcal{A}$ semantically RO if, for every valid instance I, the optimal path from the root node S_0 to a terminal state in $\mathcal{DAG}_{\mathcal{A}}(I)$ is RO.

Definition 8 (Honest pBP Algorithm) For the path found in phase 3, our pBP definition guarantees that the data items and decisions on the edge labels along the path code a solution which has a value equal to $f_1(f_2(\ldots(f_k(v))\ldots))$, where f_i is the function labeling the i'th edge in the path, and v is the value assigned to the terminal state the path ends in. Intuitively, this should also be true for non-optimal solutions. Call an algorithm *honest* if the same is true for any path from the start state to a terminal state.

Figure 1 shows natural containments of the subclasses defined above. Let SynCons, SemCons, SynRO and SemRO denote the class of problems that can be solved by polynomial pBP algorithms that have the syntactic consistency, semantic consistency, syntactic read-once and semantic read-once properties, respectively. SemCons is the most powerful because it can trivially simulate the rest. The weakest class of algorithms is the class of SynRO.

We do not know whether the containments are proper.

Fig. 1 Submodel lattice



We can also restrict the variety of orderings that the states use. The following three variants also appeared in the case of the pBT model. Before we describe them, however, one structural point is in order: given a pBP algorithm $\mathcal{A}_n$ for a problem $\mathcal{P}_n$, we can create an algorithm $\mathcal{A}'_n$ such that $\mathcal{DAG}_{\mathcal{A}'_n}$ is leveled where $size_{\mathcal{A}'_n} \leq n \cdot size_{\mathcal{A}_n}$ if A_n is syntactic read-once or $size_{\mathcal{A}'_n} \leq (\max_I depth(\mathcal{DAG}_{\mathcal{A}_n}(I))) \cdot size_{\mathcal{A}_n} \leq (size_{\mathcal{A}_n})^2$ in general. To do this, let d be the depth of $\mathcal{DAG}_{A_n}$. The new state space will be $\mathcal{S} \times [d]$, where the levels are $\mathcal{S} \times \{i\}$ for each i, and we will make each edge increase exactly one level. Hence, we will often assume that $\mathcal{DAG}_{\mathcal{A}}$ is levelled.

1. Fully Adaptive (order) pBP algorithms.
   Each state can have an entirely arbitrary ordering.
2. Adaptive (order) pBP algorithms.
   Consider the DAG $\mathcal{DAG}_{\mathcal{A}}(I)$ defined by the algorithm $\mathcal{A}$. Here we require that for each instance I, all states at the same level use the same ordering. Hence, in any computation, all paths of the same length from the root will include the same data items. Note that such algorithms are either syntactic read-once or are not read-once at all.
3. Fixed (order) pBP algorithms.
   All states at the same level have the same ordering and that ordering is the same as the previous level's ordering except that the item viewed at the previous level is moved to the end of the ordering, i.e., the data items after "end" in the ordering are precisely the previously viewed data items, and the other data items are in the same order as in the start state. In essence, all states use the same ordering, but technically that ordering must be updated in each level so that the same input item isn't viewed repeatedly forever. Such algorithms are syntactic read-once.

Again, define Fixed pBP, Adaptive pBP and Fully Adaptive pBP, respectively, to be the classes of problems that can be solved by polynomial pBP algorithms with the corresponding property. It is clear that Fixed pBP $\subseteq$ Adaptive pBP $\subseteq$ Fully Adaptive pBP. While we will see strong evidence that this hierarchy is strict, it remains an open question.

2.3 Examples

Longest Increasing Subsequence (LIS) Consider the following pBP formulation for the longest increasing subsequence. The instance is an array of integers A of length n, coded as follows. A data item is a pair (a, i), where $a \in \mathbb{Z}$ and $i \in \{1, \ldots, n\}$ is the position in the array A, where a appears. A valid instance is a set of n data items in which each i from 1 to n occurs exactly once. For short we will use array index notation and will refer to the data item as $a[i]$. The set of decisions is $\Sigma = \{0, 1\}$, where 0 means that the current number is not chosen to be part of the LIS, and 1 means it is chosen. The objective function is the number of items assigned value 1.

The computation states are all pairs (B, i), where B is a sequence of integers of length at most n and $0 \leq i \leq |B|$ is a natural number. The intended meaning is that B is the prefix of the input A viewed so far and i is the index of the rightmost element of that prefix chosen to be in the LIS. A terminal state is a state which has observed the full input, therefore is identified with (B, i), where $|B| = n$ and $0 \leq i \leq n$.

- The algorithm will use a fixed ordering on the items. That (initial) ordering puts all items with index 1 first, followed by all items with index 2, etc. Within each index partition, the items are ordered arbitrarily.
- Consider a state $s = (B, i)$, and a new item (a, j). If $j \neq |B| + 1$, then the instance is invalid and we may as well terminate. Otherwise, $g_s((a, j), 0) = ((B \circ a, i), f(x) = x)$. If $i = 0$ or $a \geq B[i]$, then $g_s((a, j), 1) = ((B \circ a, j), f(x) = x + 1)$; otherwise $g_s((a, j), 1) = \perp$.
- All terminal nodes will have value 0.

Below we give an example of the DAG built by the algorithm for the input 5, 1, 2, 3: The forward edges are labeled with the data item and the decision made by the algorithm on each data item. The edges are labeled with the function *inc* ($f(x) = x + 1$) if the decision was to accept the data item and *id* ($f(x) = x$) otherwise. The best solution has length 3 and the bold backward arrows indicate the path of the winning solution. Following the bold path from the root to the leaf we can recover the solution as 1; 2; 3. Each level will have at most n states in any computation, so the size of the above algorithm is at most n^2.

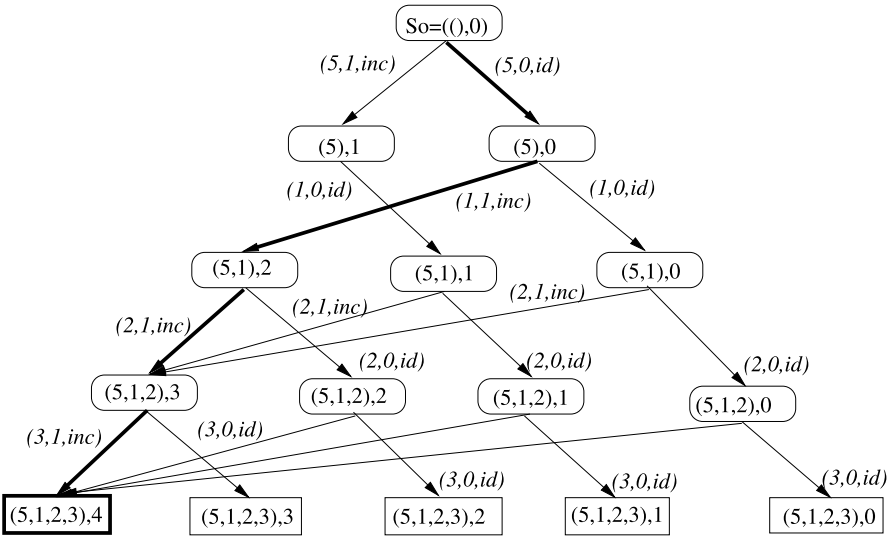


Fig. 2 pBP DAG for LIS on instance 5, 1, 2, 3

This algorithm can also be implemented in fixed order pBT with the same size. The reason is that both problems view data items in fixed predetermined order. If multiple paths would lead to the same state of the pBP algorithm, all but the best (or a best) of these paths can be pruned in the pBT algorithm. Later we will show a few general conditions under which a pBP algorithm can be transformed into a pBT algorithm using this technique. This is a somewhat counter-intuitive feature of the pBT model, and is due to the fact that we allow unrestricted computational power in the model, and only restrict access to the instance.

Single Source Shortest Path (SSSP) in Graphs with Negative Weights, Edge Model We consider the shortest paths problem where we are interested in finding a shortest weighted path from a given source s to any destination. Because the solution to the problem is a subset of edges which form a path, it is natural to use the *edge model*. Here each data item is an edge, represented as the names of the two end points and the weight (u, v, ℓ), where the names of nodes are the numbers in $[n]$. The set of valid instances is all directed graphs on $[n]$, but we attach a *promise* that the graph will have no negative weight cycles: if it violates this promise, the algorithm is allowed to output anything, including FAIL. Because the set of valid instances is all graphs, however, the size of the algorithm is the maximum size achieved over all graphs, not just the promise instances. The set of decision options is $\Sigma = \{1, 0\}$, meaning accepted and rejected, respectively. The number of vertices, n, and the name of the source, s are part of the problem definition and hence are known to the algorithm. We will implement a version of the well-known Bellman-Ford algorithm in the pBP model. Note that in the absence of negative weights, a priority algorithm can solve the problem by implementing Dijkstra's algorithm [12].

- The default label is 0; all edges not explicitly accepted along the final output path will be considered rejected.
- A computation state is encoded as the name of the currently reached vertex, u, the last vertex we rejected going to from u, v (with $v = 0$ meaning no edges have yet been rejected), and an upper bound, k, on the length (number of nodes) on the path from s to the current vertex, which is always less then or equal to $n - 1$.

  The terminal states are a special "no more neighbors" state and states where the value of k is $n - 1$.

  The start state is $(s, 0, 0)$.

  From the definition it is clear that the cardinality of the set of states is $O(n^3)$, because there are n nodes and the number of possible lengths is $n - 1$.
- Every terminal node $(u, v, k = n - 1)$ and "no more neighbors" has value 0.
- Consider a computation state $a = (u, v, k)$, where $k < n - 1$. The order π_a will first put all items of the form (u, v', ℓ) where $n \geq v' > v$ in order of increasing v' and put all other items after "end".
- Again, consider state a and assume (u, v', ℓ), $v' > v$ is the data item viewed.

  Set $g_a((u, v', \ell), 0) = ((u, v', k), f(x) = x)$, and set $g_a((u, v', \ell), 1) = ((v', 0, k + 1), f(x) = x + \ell)$. Otherwise, the data item viewed is "end", and we go to the "no more neighbors" state and terminate, with $f(x) = 0$ for the transition.

Unlike the LIS example, here it is not obvious how to efficiently implement this algorithm in the pBT model. Also note that, on promise instances, the computation will be semantically read-once, but not necessarily syntactically read-once. It seems difficult to explore paths in a graph using a small number of computation states without allowing for viewing the same edge more than once on a given computation path because the states cannot encode which nodes have already been visited. It also seems necessary to use full adaptivity.

It's clear that there are several variations of the problem that are solved by this algorithm. We chose the one we did because of some subtleties of the model and because it mostly closely matched our lower bound (see Sect. 4). The Bellman-Ford

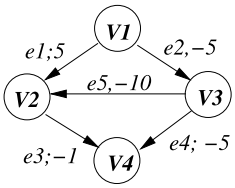


Fig. 3 Example SSSP instance graph G

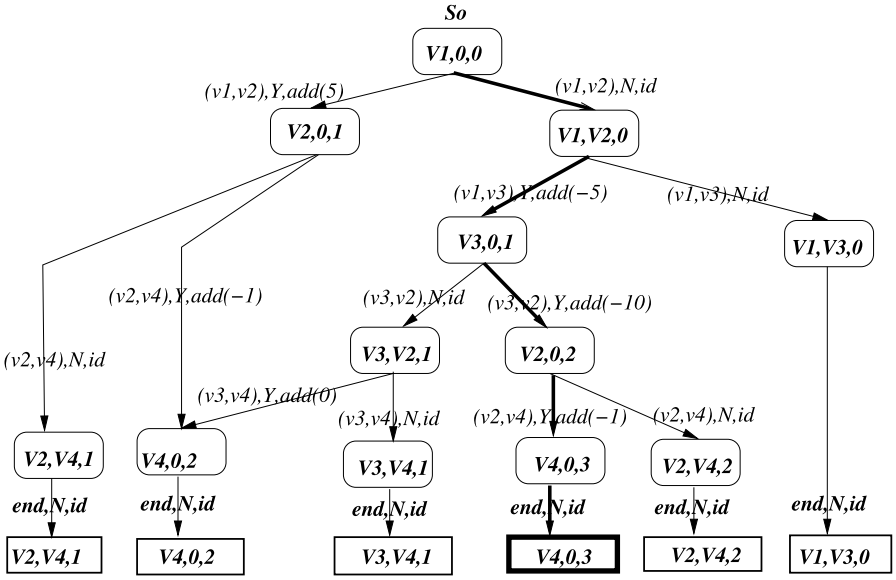


Fig. 4 pBP DAG for instance shown in Fig. 3

algorithm succeeds in finding a shortest path on all graphs when we restrict ourselves to paths with at most $n-1$ edges (without negative cycles, all shortest paths will have at most $n-1$ edges). In general, however, the solution to this problem might be a non-simple path, i.e. where some node x has out-degree greater than 1. It doesn't seem possible to make this algorithm work in the edge model in pBP because the algorithm would need the ability to first accept a given edge (one of the edges coming out of x) and then reject it (the next time the algorithm visits x. Another possibility for removing the promise is to require the algorithm to find either a shortest simple path or a negative-weight cycle. Here we do not seem to be able to remember enough state during the computation to recognize a cycle.

Next we illustrate the computation of the pBP algorithm for SSSP problem with the following example. Consider the instance graph $G = (V, E)$, shown on Fig. 3, where $V = \{V1, V2, V3, V4\}$ and $E = \{e1, e2, e3, e4, e5\}$. We want to find the shortest path from $s = V1$ to any other vertex in G. Each edge has a label and a weight. The shortest path in G originating at $V1$ is e_2, e_5, e_3 and has a total weight of -16. Figure 4 shows the computational of a pBP algorithm on the instance $G = (V, E)$.

Single Source Shortest Path (SSSP) in a DAG with Negative Weights, Node Model It seems reasonable that the complexity of a problem in these models should be sensitive to input representation—after all, the only limitations of these algorithms are information theoretic and the input representation varies the amount of information revealed in each step. While it's not clear whether pBT can compute the shortest path in a DAG efficiently in the edge input model, here we show it can in the node model.

Consider a graph represented in the *node model*. Here each data item is the name of a node (in $[n]$) together with its adjacency list and the weights of its adjacent edges. A decision will be, given a node, choose which of its out-neighbors will be its successor in the shortest path starting from s (or null if it is not in the path or is the end of the path). Note that, unlike the edge model, the items in this input model are dependent: for example, if v is mentioned as an out-neighbor in item u, then in any well-formed instance, u will be mentioned as an in-neighbor in item v. We assume s has no in-neighbors. We show that this problem can be solved by an adaptive order pBT of size $O(n^2)$.

The states will contain both the partial instance seen so far, and the partial solution, so will be of the form: a set of node items A, and a path P_v within A from s to some node $v \notin A$, from which we can compute the weight of this path w. We will maintain the following invariant: if any node in A is reachable from u, then $u \in A$.

The start state has an empty set A and a path consisting of the node s itself. At state (A, P_v) we first check to see whether P_v is the lexicographically first shortest path in A from s to v. If not, the branch aborts.

We order nodes so that nodes whose in-neighbors are all in A come first (breaking ties in some consistent manner). Given state (A, P_v) and item u, if $u \neq v$, then we branch on the null decision (i.e. u is not in the path) and go to state $(A \cup \{u\}, P_v)$. If $u = v$, we branch on the null decision (i.e. the path ends at v) and go to state $(A, P_v \circ \})$, which is a terminal state with value equal to the weight of path P_v. If $u = v$, we also branch on each out-neighbor of v, v', (i.e. v' follows v in the path) and go to state $(A \cup \{v\}, P_v \circ v')$. Since every node at a given level of the algorithm's computation tree has exactly the same A and sees the same next item, two paths in the computation tree cannot merge, so the algorithm really is in the pBT model. Also, in each of the potentially n levels of the tree, for each node v, there will be at most one active state whose path ends in v, so the size is at most $O(n^2)$.

3 Simulations of pBPs by pBTs and Nontrivial Upper Bounds

We show that, under certain natural conditions, fixed/adaptive pBT algorithms can simulate fixed/adaptive pBP algorithms without an increase in size.

First, a subtle point: our definition of pBT algorithms requires that, for a pBT algorithm $\mathcal{A}$, $\mathcal{DAG}_{\mathcal{A}}$ must be a tree. It is not hard to see that, given the weaker condition that $\mathcal{DAG}_{\mathcal{A}}(I)$ is a tree for every instance I, we can create an algorithm $\mathcal{A}'$ that satisfies the stronger condition without increasing the complexity. More generally, if two paths merge in $\mathcal{DAG}_{\mathcal{A}}$ but there is no instance I such that both of those paths appear in $\mathcal{DAG}_{\mathcal{A}}(I)$, then we may as well make two copies of the state where the merge occurred. The point is that increasing the size of the state space $\mathcal{S}$ does not

hurt the complexity of the algorithm if there is no instance which uses both copies of the original state.

Lemma 1 *Let $\mathcal{A}$ be a fixed* (*respectively*, *adaptive*) *order pBP algorithm for optimization problem $\mathcal{P}$. If the monotone functions labeling the edges of $\mathcal{DAG}_{\mathcal{A}}$ are all linear functions of the form $x + c_e$, where c_e is a constant that depends on the edge e, then there is a fixed* (*respectively*, *adaptive*) *order pBT algorithm $\mathcal{B}$ for $\mathcal{P}$ such that, $size_{\mathcal{B}} \leq size_{\mathcal{A}}$.*

Proof $\mathcal{B}$ simulates $\mathcal{DAG}_A$, but remembers the following information: the partial instance *PI*, the partial solution *PS*, the state in $\mathcal{DAG}_A$ reached, and the sum of the weights c_e along the transitions on its current path. At depth t, it views the same input as $\mathcal{DAG}_A$ at this state, but before simulating a transition, it uses *PI*, which is the same along every other path, to simulate all other $(t+1)$-step paths of $\mathcal{DAG}_A$; if any of them reach the same state with a larger sum of transition constants, or if a lexicographically prior partial solution gives the same sum and the same state, then the current branch is pruned. If it reaches a terminal state, then it assigns all unassigned data items the default value, and terminates with value equal to the objective function at the (now total) solution.

Since $\mathcal{B}$'s states remember *PI* and *PS*, they form a tree. If $\mathcal{B}$ on I simulated two paths that arrived at the same state, either one has a smaller sum of transition constants, or both have the same sum, and one has a lexicographically prior partial solution. Therefore, one of the two will be pruned. Thus, there is at most one unpruned path in $\mathcal{B}$ per reachable computation state in $\mathcal{DAG}_A(I)$, so the width is at most the complexity of $\mathcal{DAG}_A$. Finally, consider an algorithm that, when two transitions give a state s the same value, breaks ties according to the lexicographical ordering on the label. Then in the third phase of the algorithm, if from s_0 we reach node v, the chosen path will always have the highest sum of transition constants among possible paths from s_0 to v (otherwise, whatever value v is assigned, a higher value is assigned to s_0 along the other path) and among such paths, it will be the lexicographically first. Thus, the optimal solution's path, which is found in phase 3 of $\mathcal{DAG}_A$, is not pruned in $\mathcal{B}$, so in returning the best of the solutions found at its branches, $\mathcal{B}$ will return an optimal solution. □

As a corollary of this theorem, we see that the LIS and the LCS problems can be solved by polynomial-width fixed order pBT algorithms.

The following lemmas show another condition under which pBT algorithms can simulate FIXED or ADAPTIVE pBP algorithms for search problems.

Given a maximization $\mathcal{P}$, let $\mathcal{P}^v$ denote the same problem except that $\mathcal{F}_{\mathcal{P}^v}$ returns 1 on those solutions where $\mathcal{F}_P$ is at least v, and 0 otherwise.

Lemma 2 *Let $\mathcal{A}$ be an honest fixed* (*respectively*, *adaptive*) *order pBP algorithm for optimization problem $\mathcal{P}$. For any $v = v(n)$, consider the corresponding search problem $\mathcal{P}^v$. Then there is a fixed* (*respectively*, *adaptive*) *order pBT algorithm $\mathcal{B}$ for this search problem such that, $size_{\mathcal{B}} \leq size_{\mathcal{A}}$.*

Proof $\mathcal{B}$ simulates $\mathcal{A}$, remembering the partial instance PI, and the partial solution PS, as well as the current state in $\mathcal{DAG}_A$, and a value v_t given by: $v_0 = v$ and $v_t = f_t^{-1}(v_{t-1})$, where f_t is the monotone function labeling the transition at time t. As before, when $\mathcal{B}$ simulates the transition, if there is a path reaching the same state of $\mathcal{DAG}_A$ and yielding a smaller value of v_t or if there is a lexicographically prior such a path giving an equal value for v_t, the path is pruned. When we get to a terminal node, we give it value 1 if $\mathcal{A}$ gives it value at least v_t and 0 otherwise. Hence, at most one path that reaches any given state of $\mathcal{DAG}_A$ is not pruned, so the total size for $\mathcal{B}$ is at most that of $\mathcal{A}$.

Assume s_0 is given value 1 in $\mathcal{B}$. Then phase 3 returns a path in $\mathcal{B}$ $s_0, s_1, \ldots s_t$ with s_t a terminal, where all nodes along the path are given value 1. Each transition in $\mathcal{B}$ corresponds to a transition in $\mathcal{A}$, so we can look at the series of functions $f_1, \ldots f_t$ labeling the edges in this path in $\mathcal{A}$, and the series of values $v_0, \ldots v_t$ defined $v_0 = v, v_i = f_i^{-1}(v_{i-1})$. Then since s_t is a terminal given value 1, in $\mathcal{A}$, s_t is assigned a value $\geq v_t$. Assume that s_{i+1} is assigned a value in $\mathcal{A}$ which is $\geq v_{i+1}$. Since the value of s_i is the maximum of a number of terms, one of which is $f_i(v(s_{i+1}))$ and f_i is monotone, s_i is assigned a value in $\mathcal{A}$ at least $f_i(v_{i+1}) = f_i(f_i^{-1}(v_i)) \geq v_i$ by definition of inverse. Thus, $f_1(f_2(\ldots(v(s_t))\ldots)) \geq v$, which, by the honesty condition implies that the edges on the path code a solution of value at least v. So if $\mathcal{B}$ claims there is a solution, it successfully solves the search problem.

Then assume there is a solution of value $\geq v$. Let $s_0, \ldots s_t$ be the path returned by $\mathcal{A}$ in phase 3, coding an optimal solution (and hence one of value $\geq v$). Define the series of values $v_0, \ldots v_t$ by $v_0 = v, v_i = f_i^{-1}(v_{i-1})$, where f_i labels the transition. Since $\mathcal{B}$ only aborts a path to s_i when it finds another path where the composition of the inverse functions on v, w_i, along that path is as small, $\mathcal{B}$ contains paths to each state of $\mathcal{A}$ with search value w_i equal to the smallest such composition of any path to s_i. Thus, it will have a path to s_t of label at most v_t, and will thus give s_t value 1. Since the value of the root s_0 in $\mathcal{B}$ is the or of the values of the terminal states, this means that B returns a 1 in phase 2. By the previous paragraph, this means it finds a solution of value at least v in phase 3. □

Lemma 3 *Let $\mathcal{A}$ be a pBP algorithm for a problem $\mathcal{P}$ and let $V \subset \mathbb{R}$ be a finite set such that, for every instance I and every state s in $\mathcal{DAG}_{\mathcal{A}}(I)$, the value of s computed in phase II is contained in V. Let w be the maximum value in V. There exists a pBP algorithm $\mathcal{B}$ for $\mathcal{P}^w$ where $size_{\mathcal{B}} \leq |V| size_{\mathcal{A}}$ which uses the identity function as its only monotone function. Furthermore, if $\mathcal{A}$ is fixed (respectively, adaptive) order, then $\mathcal{B}$ will also be fixed (respectively, adaptive) order.*

Proof Let $\mathcal{S}$ be the state space of $\mathcal{A}$. The state space of $\mathcal{B}$ will be $((\mathcal{S} \setminus \{S_0\}) \times V) \cup \{S_0\}$. If t is a terminal state for $\mathcal{A}$ that returns value $val(t)$, then for all $v \in V$, (t, v) will be a terminal state for $\mathcal{B}$ that returns 1 if $v \leq val(t)$ and 0 otherwise. In general, if there is an edge (s, s') in $\mathcal{DAG}_{\mathcal{A}}$ labeled by (d, σ, f), then, for $v, v' \in V$, there will be an edge $((s, v), (s', v'))$ labeled by (d, σ, id) if $v \leq f(v')$. The root state of $\mathcal{B}$ will be S_0. If there is an edge (S_0, s') in $\mathcal{DAG}_{\mathcal{A}}$ labeled by (d, σ, f), then, for $v' \in V$, there will be an edge $(S_0, (s', v'))$ labeled by (d, σ, id) if $w \leq f(v')$. If a state (s, v) in $\mathcal{B}$ has no children and s is not a terminal state in $\mathcal{A}$, then make (s, v) a terminal state in $\mathcal{B}$ with value 0.

Given any instance I, it is clear by induction on the height of a state s in $\mathcal{DAG}_{\mathcal{A}}(I)$ that the state (s, v) will achieve value 1 on instance I in $\mathcal{B}$ if and only if state s achieved value at least v on instance I in $\mathcal{A}$. $\mathcal{B}$ will choose as its solution an arbitrary path from S_0 to a sink in $\mathcal{DAG}_{\mathcal{B}}(I)$ labelled $(d_1, \sigma_1), \ldots, (d_k, \sigma_k)$ such that every state along this path achieves value 1 (if there is such a path). This path corresponds to a path in $\mathcal{A}$ that contributed value w to the state S_0. Since every such path in $\mathcal{A}$ must constitute a valid solution of value w (note this is not necessarily true in a non-honest algorithm if w is not the max value), so must this path. □

Note that, in the previous lemma, if $\mathcal{A}$ was fixed or adaptive order, then $\mathcal{B}$ satisfies the assumptions of Lemma 1. Hence there is a pBT algorithm with the same size computing $\mathcal{P}^w$ in this case.

We now describe a non-trivial upper bound that seems to rely on the ability of pBPs to merge paths. Using the above lemmas, however, we will show that it also applies to pBTs. The phenomenon that yields this upper bound seems to occur for many problems—intuitively those that have a limited number of subproblems—but we will illustrate it for maximum independent set, especially as it was inspired by Robson's algorithm for that problem [18]. The upper bound would certainly work for any accept/reject problem on the nodes of a graph.

Consider the maximum independent set problem on a graph with n nodes. We will represent it in the node model: each item is the name of a node and the names of that node's neighbors. The decisions are to accept a node as part of the MIS or to reject it. The trivial (pBT) upper bound is simply to consider the nodes in some fixed order and to explore both decisions for each node, yielding an upper bound on size of 2^n. Instead, begin by branching on all decisions for nodes 1 through k. This yields 2^k partial solutions. Any one of them (assuming it is valid) defines a subproblem: find the MIS on the subgraph induced by those nodes in $[n] \setminus [k]$ that are not neighbors of any node accepted in the partial solution. There are at most $\ell = \sum_{i=1}^{n-k} \binom{n-k}{i} = 2^{n-k}$ such subproblems. Let $T_1, \ldots, T_\ell$ be pBP dags, each of size at most 2^{n-k}, solving each of these subproblems. Each of the at most 2^k leaves corresponding to valid partial solutions from the initial branching on items 1 through k should be identified with the root of one of the ℓ pBP dags. Hence, the entire dag has size $2^k + 2^{n-k} \cdot 2^{n-k}$. Setting $k = 2n/3$ gives an upper bound of $O(2^{2n/3})$.

It is clear that many paths merge in the above pBP. The initial branching has (potentially) $2^{2n/3}$ leaves, whereas there are only $2^{n/3}$ pBPs on the bottom. Note, however, that the algorithm can be done in fixed order, and that every edge is labelled with the function $f(x) = x$ (if we reject the current node) or $f(x) = x + 1$ (if we accept it). Hence, we can apply Lemma 1 to get a pBT algorithm.

We conclude this section by observing that most natural implementations of dynamic programming algorithms as pBPs seem to satisfy the conditions of Lemmas 1 and 2 (certainly those in Sect. 2 do). This is strong evidence that the separation between fixed order and adaptive order pBTs given in [2] also holds for pBPs.

4 Fully Adaptive pBT Lower Bound for the Shortest Path Problem

We present a general technique for proving lower bounds for pBT algorithms and will obtain an exponential lower bound for fully adaptive pBT algorithms solving the single source shortest path problem in the edge model, as defined in Sect. 2.3 above (i.e. where we allow arbitrary edge weights). The lower bound does have one qualification: above we considered a promise version of the problem where the promise was that the input graph contains no negative cycles. Here we show that any pBT algorithm that correctly computes a shortest path on the promise instances achieves exponential width on some instance, although it might be an instance with negative cycles. Actually, we focus on a simpler subproblem where every edge has weight -1, which amounts to finding the longest simple path from a source s.

The lower bound technique we define next can be used for problems where the set of decisions is $\Sigma = \{accept, reject\}$. Let $\mathcal{A}$ be a pBT algorithm for some problem and consider the tree $T = \mathcal{T}_A(I)$ for some fixed instance of this problem. Each state s in T is defined by the partial solution PS_s and the partial information $PI_s = \{PI_s^{in}, PI_s^{out}\}$. PI_s^{in} are data items which belong to the instance and are assigned decisions along the path from the root to s. A data item e belongs to PI_s^{out}, if there is a state s' along the path from the root to s, and the ordering function at the state s' is $\pi(s') = \{\ldots, e, \ldots, e' \ldots\}$ such that e' is the first data item in the total order $\pi(s')$ which belongs to the instance I. If $e \in PI^{out}$ then it is the case that $e \notin PI^{in}$. PS_s is that subset of PI_s^{in} that has been accepted.

Definition 9 (Consistency) We call an instance I consistent with a partial information $PI_s = \{PI_s^{in}, PI_s^{out}\}$ if $(PI_s^{in} \subseteq I)$ and $(I \cap PI_s^{out} = \emptyset)$.

Definition 10 (Unique Extension) Let PS_s be the set of data items accepted along a branch. Then PS_s is *uniquely extendible* with respect to PI_s if there is an instance I_s consistent with PI_s such that, the only solutions to the problem on I_s are equal to PS_s when restricted to PI_s.

Definition 11 (Agreeable Data Item) A new item e *agrees* with the partial information PI_s and the partial solution PS_s if there is a valid instance I_s consistent with PI_s which has some optimal solution consistent with PS_s that contains e.

Definition 12 (Competing Data Item) If a data item is not agreeable then it is competing.

The lower bound is defined as a game played by a Solver and an Adversary, which we call the Q-improbability game. The game proceeds in rounds.

1. The Adversary privately selects an instance I, which might not be a valid instance. Then sets $Q_0 \leftarrow 1$; $i \leftarrow 1$.
   The Solver initializes empty revealed information and partial solution: $PI_0^{in}, PI_0^{out}, PS \leftarrow \emptyset$.
2. Round i begins
   (a) The Solver picks a data item $d_i \notin PI_{i-1}$.

(b) The Adversary reveals whether $d_i \in I$ or not.

- If $d_i \notin I$ then the Solver updates $PI_i^{out} \leftarrow PI_{i-1}^{out} \cup \{d_i\}$ and next round begins.
- If $d_i \in I$, then the Solver updates $PI_i^{in} \leftarrow PI_{i-1}^{in} \cup \{d_i\}$. The Adversary picks probability q_i.
  With probability q_i, $PS \leftarrow PS \cup \{d_i\}$; $Q_i \leftarrow Q_{i-1} \times q_i$;
  With probability $1 - q_i$, PS remains unchanged; $Q_i \leftarrow Q_{i-1} \times (1 - q_i)$;

round i ends; $i \leftarrow i + 1$.

3. The game ends when, at the end of some round t, $Q_t \leq Q$. Let PI and PS denote the partial information and solution at the end of the game.
4. The Adversary wins if there exists a *valid* instance I' consistent with PI so that every optimal solution in I' agrees with PS; otherwise the Solver wins.

Lemma 4 *Let Π_n be a problem with a finite set of data items D_n. If there is a width w pBT algorithm for Π_n, then the Adversary wins with probability at most Qw.*

Proof Let $\mathcal{A}$ be a width w pBT algorithm for Π_n. Solver simulates $\mathcal{A}$ to pick the next data item as follows: Initially, Solver runs $\mathcal{A}$ and obtains the ordering $\pi_{S_0}(D)$. Let d be the first data item in the ordering, then Solver sets $d_1 = d$. At round i, the Solver maintains PI_i^{in}, PI_i^{out}, and PS_i. PS_i and PI_i^{in} uniquely define a branch of the pBT tree and a state in it; if not, then the Solver has clearly won since, if there were an instance I' witnessing the Adversary's victory, then $\mathcal{A}$ would fail on that instance. Therefore, let the state defined by PS_i and PI_i be s. The Solver then uses π_s and picks d_i to be the first item in the ordering. The game ends when the probability of the current path (determined uniquely by $PI = PI^{in}, PI^{out}, PS$) falls below Q.

The Adversary loses if the branch corresponding to PS and PI at the end of the game is not part of the pBT tree $\mathcal{T}_{\mathcal{A}}(I)$, Since $\mathcal{A}$ is width w algorithm and each branch has probability at most Q at this point, the probability the branch defined by PS and PI is in the tree is at most Qw. □

We now begin instantiating this general technique for the shortest path problem in the edge model. According to the definitions above, an edge (u, v) is competing with respect to a given partial solution PS if there exists edge $(u, x) \in PS$ or there exists edge $(w, v) \in PS$. Otherwise it is agreeable. A competing edge cannot be part of the solution because, adding such an edge to PS will destroy its validity (PS will no longer be a simple path).

Now we describe the strategy for the Adversary for the shortest path in graphs with negative weights but no negative cycles. Let t and p parameters which depend on n such that $t = n^{1/9}$ and $p = n^{-3/4}$. The Adversary picks an instance I at random from $G(n, p)$. Each edge if present has a weight of -1. Let $PI_i = PI_i^{in}, PI_i^{out}$ and PS_i be the revealed information and the partial solution after round i has finished. At round $i + 1$ the Solver selects an edge e: If e is competing with PI_i^{in} then $q_{i+1} = 0$. Otherwise $q_{i+1} = 1/2$. Then at the end of the round we have $Q_{i+1} = \frac{1}{2^{t_{i+1}}}$, where t_{i+1} is the number of agreeable edges seen so far. Without loss of generality, we assume that the Solver views all competing edges after each edge e is added to PS. The Q-improbability game is played for $Q = \frac{1}{2^t}$. In what follows we show that, with high

probability over the random instance I and the random choices of the Adversary, PS_i remains uniquely extendible with respect to PI_i until t agreeable edges have been viewed, establishing an exponential lower bound on the width of any pBT algorithm for the shortest path problem.

Lemma 5 *Let $PI = (PI^{in}, PI^{out})$ be the partial information and PS be the partial solution. If they satisfy the following three invariants, then PS is uniquely extendible with respect to PI*:

[P.1] *PI^{in} contains no cycles even in the undirected sense.*
[P.2] *$|PI^{in}|$ is $o(\sqrt{n})$.*
[P.3] *Any node with in(out)-degree 0 in PS has in(out)-degree $o(n)$ in $PI^{in} \cup PI^{out}$.*

Proof We will exhibit a set of edges M such that, M is disjoint from PI and $PS \cup M$ is the unique longest path starting at s in $PI^{in} \cup M$.

Since PI^{in} contains no cycles then PI^{in} is a directed acyclic graph. View PS as a set of disjoint simple directed paths, let those be $P_1, \ldots, P_k$ such that, P_i goes from, say, u_i to v_i. If PS doesn't touch s, then we assume P_1 begins at u_1 and ends at v_1, where $u_1 = v_1 = s$; otherwise, $u_1 = s$ and let $v_1 \neq s$ be the end of the path in PS leaving s. Since PI^{in} is a directed acyclic graph it defines a partial order on the paths P_i. Let $\prec$ be a total order on the paths $P_1, \ldots, P_k$ consistent with the partial order defined by PI^{in} such that, $P_i \prec P_j \Rightarrow i < j$. Let V' be the nodes not touched by PI^{in}, and let E' be the set of all possible edges minus those in $PI^{in} \cup PI^{out}$. We construct k disjoint simple directed paths $Q_1, \ldots, Q_k$ such that, Q_i goes from v_i to u_{i+1} for $i < k$ and Q_k goes from v_k to some node in V'. The set of simple paths $\{Q_1, \ldots Q_k\}$ satisfies the following conditions:

- $|Q_i| > |PI^{in}|, \forall i = 1, \ldots, k$,
- the intermediate nodes of each Q_i are from V',
- the edges of each Q_i are all from E'.

M will be the set of all edges in $\{Q_i\}$. Note that $PI^{in} \cup M$ is cycle free because the paths Q_i respect the order $\prec$.

Claim 6 *$P = PS \cup M$ is the unique longest path in $PI^{in} \cup M$ from s.*

Proof Let P' be any other path $P' \neq P$. They both must originate at s and since they are distinct they must diverge at some node $d \in V(P_1 \cup \cdots \cup P_k)$. It could not be the case that $d \in \{Q_1 \cup \cdots \cup Q_k\}$ because each Q_i is a simple path. Suppose $d \in P_i$. Let (d, u) be the edge in P' then $(d, u) \in PI^{in}$. If the path P' never rejoins with P, then it must skip at least one Q_i, but $|Q_i| > |PI^{in}|$, hence P' is shorter than P. The path P' cannot rejoin P within the segment P_i because PI^{in} has no cycles. Also note that P' cannot rejoin P at some segment P_m, for $m < i$ because this would imply a directed cycle in $PI^{in} \cup M$. On the other hand if the section of P' after node d rejoins with P then it must be at a segment P_j with $j > i$, thus missing all edges in Q_i, and because $|Q_i| > |PI^{in}|$ then P' is shorter. □

To construct the paths $\{Q_i\}$ from E' we need the following claim:

Claim 7 *Let $H = (S, E(S))$ be a graph of order m such that*[2] $|\Gamma(u)_{out}| > \frac{m}{2}$ *and* $|\Gamma(v)_{in}| > \frac{m}{2}$, *then there exists at least one node w such that $(u, w) \in E$ and $(w, v) \in E$.*

Proof By the pigeon hole principle. □

Initially $H = (V' \cup \{u_i, v_i\}_{i=1}^{k}, E')$. By invariant [P.2], $|V'| = (1 - o(1))n$. By invariant [P.3], let $|PI^{in}| = L \in o(\sqrt{n})$ and assume that L is even (if not, increment it by one). We need to construct k paths of length $L+1$, where $k \leq t+1$ and $kL \in o(n)$.

We construct paths $Q_1, \ldots, Q_k$, such that $Q_i = v_i, w_1^i, \ldots, w_L^i, w_{L+1}^i, u_{i+1}$. We begin with Q_1. Choose w_1^j for j even arbitrarily. Starting with $j = 2$, apply Claim 7 to obtain w_1^1 such that $(v_1, w_1^1), (w_1^1, w_2^1) \in E'$. Remove v_1, w_1^1 from H and proceed to apply the Claim to obtain $w_3^1, w_5^1, \ldots$. When finished with Q_1, build Q_2 in the same way, etc. Note that the Claim will always apply because at any point we will have removed at most $o(n)$ edges from H.

Our goal is to show that at the end of the Q-improbability game PI and PS satisfy invariants [P.1], [P.2], [P.3] with high probability. Edges which were rejected but were agreeable with the current partial solution PS are denoted by R. Whenever $\mathcal{A}$ accepts an edge (u, v) and adds it to PS, then all edges of G out of u or into v are revealed. Those edges are called competing with PS and are denoted by C. At the end of the Q-improbability game the edges present in the graph are $PI^{in} = R \cup C \cup PS$ and the revealed information $PI = PI^{in}, PI^{out}$. Define $t = |R| + |PS|$ (t is the number of agreeable items seen at the end of the Q-improbability game). We show tight bounds on the sizes of the sets of edges PS, R, C, and PI^{out}. Recall that $t = n^{1/9}$ and $p = n^{-3/4}$. The Adversary stops the game when t reaches $n^{1/9}$.

Consider the partial solution PS. Let X_i for $i = 1, \ldots, t$ be a family of indicator random variables such that $X_i = 0$ if the i-th edge was in PS and 0 otherwise. By the Adversary's strategy $\Pr[X_i = 0] = \Pr[X_i = 1] = 1/2$. Then the expected number of accepted edges is:

$$\mathrm{Exp}[|PS|] = \mathrm{Exp}\left[\sum_{i=1}^{t} X_i\right] = \sum_{i=1}^{t} \Pr[X_i = 1] = \frac{t}{2}.$$

Because $X_1, \ldots, X_t$ are independent, we can apply Chernoff's bound: for any sufficiently large constant δ, $\Pr[|PS| > (1+\delta)\frac{t}{2}] < e^{-\Omega(t)}$. The same argument establishes the same bound on the size of R.

Now we consider the set C of edges competing with PS (revealed after an agreeable edge is accepted by $\mathcal{A}$). Let D_i for $i = 1, \ldots, t$ be the number of edges incident to the nodes of the i-th edge. Then, for all $i = 1, \ldots, t$ we have $\mathrm{Exp}[D_i] = 2np$ and, for any sufficiently large constant δ $\Pr[D_i > (1+\delta)2np)] < e^{-\Omega(np)}$. Therefore,

$$\Pr\left[\sum_{i=1}^{t} D_i > (1+\delta)2tnp\right] < e^{-\Omega(tnp)}.$$

[2] We use the standard mathematical notation: $\Gamma(x)_{in}$ is the set of neighbors of x such that $(u, x) \in E$ and $\Gamma(x)_{out}$ is the set of neighbors of x such that $(x, u) \in E$.

The number of competing edges is $|C| \leq \sum_{i=1}^{t} D_i$.

Finally, we look at the number of edges in PI^{out} that agree with PS; call this set A. Each such edge is revealed when the Solver proposes an edge and the Adversary reveals that it is not present in the graph. Each time this happens, the edge has probability p of being in the graph. The probability that the game has $(1+\delta)t/p$ rounds, then, before t agreeable edges are added to PI^{in} is $e^{-\Omega(t/p)}$ (again for sufficiently large constant δ) again by Chernoff. Therefore, with at most this probability, A contains more than $(1+\delta)t/p$ edges.

We prove the validity of the invariants out of order.

Lemma 8 *With probability* $1-o(1)$ *invariant* [P.2] *holds and* $|PI^{in}| \in o(\sqrt{n})$.

Proof $PI^{in} = R \cup PS \cup C$. The above argument establishes that the sum of the sizes of these sets are at most $(1+\delta)(t+t+2tnp) \in o(\sqrt{n})$ by our choices of t and p. □

Lemma 9 ([P.1]) *With probability* $1-o(1)$, PI^{in} *is cycle free* (*in the undirected sense*).

Proof Consider growing PI^{in} as the game proceeds. We bound the probability that the final PI^{in} contains a cycle by taking the union bound over the probabilities that, each time we add new vertices to PI^{in}, there is an edge in I connecting any of these new endpoints with each other or with previous endpoints (excluding, of course, those pairs of endpoints that were added because they are connected by an edge). Invariant [P.2] implies that, with high probability, the eventual size (number of edges) of PI^{in} is at most $(1+\delta)(2t+2tpn)$, for sufficiently large constant δ, which is at most $3tpn$ since, by our choice of p and t, $t \in o(pn)$. Hence there are at most $6tpn$ endpoints. By the end of the game, the union bound will amount to at most $\binom{6tpn}{2} p < 36t^2p^2n^2p = 36(tn)^2p^3 = 36n^{-1/36} \in o(n)$. □

Lemma 10 *With probability* $1-o(1)$ *invariant* [P.3] *holds true.*

Proof Consider a node x that has out-degree 0 in PS (the in-degree 0 case is similar). Edges coming out of x in $PI^{in} \cup PI^{out}$ consist of those edges in PI^{in}, those edges in PI^{out} that are agreeable (which we called A above) and those edges in PI^{out} that are competing with PS. We've established that, with high probability, $|PI^{in}| \in o(\sqrt{n})$ and $|A| \in O(t/p) = o(n)$, so the number of edges in these categories coming out of x is certainly $o(n)$. Those edges in PI^{out} that are competing with PS must have their other endpoint in PS, so there are at most t of them coming out of x. In total, then, x has out-degree at most $o(n)$ in $PI^{in} \cup PI^{out}$. □

Lemma 11 *Let* $p = n^{-3/4}$ *and* $t = n^{1/9}$. *Consider the* Q*-improbability game, for* $Q = \frac{1}{2^t}$, *played by the Solver and the Adversary for the shortest path problem on a random graph* $G \sim \mathcal{G}_{dir}(n,p)$. *The probability, over random* G *and the random coin tosses of the Adversary, that PS is not uniquely extendible with respect to PI is* $o(1)$.

Proof By Lemma 5 the partial information and solution at the end of the Q-improbability game is uniquely extendible if all three invariants [P.1], [P.2], [P.3]

are satisfied. By Lemmas 9, 8, and 10 the probability that the partial information PI and the partial solution PS will violate any of the invariants is: $e^{-\Omega(t)} + e^{-\Omega(t)} + e^{-\Omega(tpn)} = o(1)$. Hence with probability $1 - o(1)$ at the end of the Q-improbability game the partial instance is uniquely extendible. □

Theorem 12 *Any fully adaptive pBT algorithm for shortest path with negative weights and no negative cycles on graphs of size n requires width* $\Omega(2^{n^{1/9}})$.

Proof Lemma 11 guarantees a single graph G for which the result of the $2^{n^{1/9}}$-improbability game is uniquely extendible with probability $1 - o(1)$. The theorem now follows directly from Lemma 4. □

5 Lower Bounds for Perfect Matching Problem in Bipartite Graphs

In this section we define a general technique for proving lower bounds for fully adaptive pBP algorithms which use values $\{0, 1\}$ on the states computed in stage II, and will use it to prove an exponential lower bound on the number of states of such pBP algorithms for the decision problem of perfect matching in bipartite graphs. By Lemma 3, this implies an exponential lower bound on the size of pBP algorithms for the optimization problem maximum bipartite matching, provided that the algorithm uses a limited (subexponential) number of different values in stage II. Since any pBT algorithm for maximum bipartite matching on a graph of size n could never use more than n different values, this immediately implies a lower bound on all fully adaptive pBT algorithms for maximum bipartite matching. And, in fact, we will see that the lower bound we present for pBP is a natural extension of the technique used for pBT.

Consider the DAG built by any pBP algorithm. Unlike the pBT model a pBP algorithm can merge paths, so intuitively in proving lower bounds we want to show that a family of instances exists such that, for any algorithm, there exists an instance on which merging paths is rare. The framework we use is as follows. Let $\mathcal{A}$ be a pBP algorithm using values $\{0, 1\}$ on states defined in stage II, for some problem Π as defined in Sect. 2.1. Let H be a valid instance of Π. For a given path in $\mathcal{DAG}_{\mathcal{A}}(H)$, as before, let PI^{in} and PI^{out}, be the set of data items observed, and the set of data items known to not be present in the instance, respectively, and denote $PI = PI^{in} \cup PI^{out}$ be the revealed information and $PS \subseteq PI^{in}$ be the partial solution, or data items that have been accepted. The ingredients for the lower bound are:

1. A *distribution on instances* $\mathcal{H}$.
2. A *path picking strategy* $\mathcal{S}$ defines rules for walking in $\mathcal{DAG}_{\mathcal{A}}(H)$, where H is a valid instance.

   Let $\gamma \in \mathcal{D}$ be a data item, and $\mathcal{D}_{\Sigma}$ be the space of probability distributions on Σ then

$$\mathcal{S} : PI \times PS \times \mathcal{D} \to \mathcal{D}_{\Sigma}$$

   Note that 2 imposes a *distribution on paths in* $\mathcal{DAG}_{\mathcal{A}}(H)$:

   Run $\mathcal{A}$ on H but always use $\mathcal{S}$ to pick the next state (The path is considered to go to a null state if there is no consistent transition in $\mathcal{DAG}_{\mathcal{A}}(H)$.

3. A property *alive* of a path. No path that is alive should be at a null state (for a correct algorithm).

To establish the lower bound on the number of states we consider the width w of the DAG built by the algorithm (recall that we can assume the DAG is levelled) and want to show that it is highly unlikely for two alive paths of the same length to reach the same state in $\mathcal{DAG}_{\mathcal{A}}(H)$. Say that two such paths "merge". Each alive path of a given length is at one of w states in $\mathcal{DAG}_{\mathcal{A}}(H)$, hence, if we pick p_1, p_2 independently according to the distribution above,

$$\Pr(p_1 \textit{ and } p_2 \textit{ merge} \mid p_1 \textit{ and } p_2 \textit{ are alive}) = \frac{\Pr((p_1 \textit{ and } p_2 \textit{ merge}) \wedge (p_1 \textit{ and } p_2 \textit{ are alive}))}{\Pr(p_1 \textit{ and } p_2 \textit{ are alive})} \geq \frac{1}{w}.$$

Therefore,

$$w \geq \frac{\Pr(p_1 \textit{ and } p_2 \textit{ are alive})}{\Pr(p_1 \textit{ and } p_2 \textit{ merge} \wedge p_1 \textit{ and } p_2 \textit{ are alive})}.$$

Thus to prove a lower bound on w, we need to show that (a) almost all paths are "alive" and (b) the probability that two "alive" paths merge is small.

Lemma 13 *Let p_1, p_2 be paths that are alive in $\mathcal{DAG}_{\mathcal{A}}(H)$, where $\mathcal{A}$ is a correct algorithm for the problem, with revealed information PI_1 and PI_2, respectively, and partial solutions PS_1 and PS_2. Assume that there exists an instance H' with H' consistent (see Definition 9) with $PI_1 \cup PI_2$, so that H' has a unique solution, which is consistent with PS_1 but not with PS_2. Then p_1 does not merge with p_2.*

Proof Consider $\mathcal{DAG}_{\mathcal{A}}(H)$. By the definition of merge, p_1 and p_2 lead to the same state v. Let PS be the unique solution of H'. Since PS_1 is consistent with PS then $PS_1 \subseteq PS$ and let $PS' = PS \setminus PS_1$. v must be on the path from the root to the unique solution in H', and the value returned along this path must be 1. But then the path that leads to v via PS_2 and then proceeds to a leaf via PS' will also be labelled with 1, and hence could be output by the algorithm as a solution for H. But that cannot be correct since the unique solution of H' is not consistent with PS_2. □

Note that this technique is a generalization and extension of the lower bound technique used for pBT algorithms. We use the move of the Adversary to define the distribution on instances and essentially the path picking strategy determines the moves of the Solver. However, being alive can place restrictions on both paths and instances, unlike the condition of "unique extensibility" which is mainly a property of the path. Similarly as the pBT lower bounds this technique is mainly suited for problems where data items are independent from each other (such as edges in a graph).

We now apply the technique to the matching problem.

The following claim will be used repeatedly later.

Claim 14 *Let G be a bipartite graph on nodes L and R where both halves are of order n, in which every vertex has degree at least $\frac{n}{2}$. Then G has a perfect matching.*

Proof We claim that the necessary and sufficient condition of Hall's theorem holds true, namely for each set of nodes X, $|\Gamma(X)| \geq |X|$. If not, there exists a set $X \subseteq L$ such that $|\Gamma(X)| < |X|$. Because each node in L is incident to at least half of the nodes on the opposite side, it has to be the case that $|X| > \frac{n}{2}$. Now consider $R - \Gamma(X)$. Each node in $R - \Gamma(X)$ has neighbors only in $L - X$, so has degree at most $n - |X| < \frac{n}{2}$, a contradiction. □

Now we will apply the framework above to obtain an exponential lower bound on pBP algorithms on the perfect matching problems in bipartite graphs.

1. The distribution on instances $\mathcal{H}$ we use is the space of all random bipartite graphs with independent edge probability $q = n^{-5/6}$, $G(n \times n, q)$.
2. The path picking strategy is defined as follows. Let $PI = PI^{in} \cup PI^{out}$ be the revealed information learned by $\mathcal{A}$ along the path p and $PS \subseteq PI^{in}$ be partial solution. An edge which can legally extend the current partial solution PS is added with probability $1/2$ and is rejected with the same probability. Let γ be the next data item observed, then we define the path picking strategy as follows:

$$\mathcal{S}(PI, PS, \gamma) = \begin{cases} \Pr(\gamma \textit{ is accepted}) = \frac{1}{2}, & \text{if } PS \cup \{\gamma\} \text{is a valid matching;} \\ \Pr(\gamma \textit{ is accepted}) = 0, & \text{if } PS \cup \{\gamma\} \text{ is NOT a valid matching;} \end{cases}$$

3. A path $p = (PS, PI)$ is *alive* if the following two conditions are satisfied: (1) There exists a graph H', consistent with PI ($PI^{in} \subseteq H$ and $PI^{out} \cap H = \emptyset$), so that H' has a unique perfect matching and this matching is consistent with PS. (2) PI^{in} is cycle free.

Given any pBP algorithm $\mathcal{A}$ the Adversary builds the $\mathcal{DAG}_{\mathcal{A}}$ on an instance $H \in_{\mathcal{H}} G(n \times n, q)$. Let p be the path in $\mathcal{DAG}_{\mathcal{A}}(H)$ picked by the strategy $\mathcal{S}$. We call an edge that could be added to PS and have the result be a partial matching at the step it was examined an *agreeable* edge. Edges that were rejected but were agreeable with the current partial solution PS are denoted by R. We can assume without loss of generality that whenever $\mathcal{A}$ accepts an edge (u, v) and adds it to PS, it examines and rejects all edges of H incident to u and v. Those edges are called *competing* with PS and cannot become a part of the solution on this branch. We denote this set of edges as C. Then the path is defined by the edges present in the graph $PI^{in} = R \cup C \cup PS$ and the edges known not to be present PI^{out}. We terminate the game when $|R| + |PS| = t = n^{1/8}$. Edges not in PI^{in} nor in PI^{out} are called *unexamined*.

We want to estimate the probability that a random path p is alive. To do that we define a set of invariants that imply that the path p is alive. Let S be the set of nodes matched in PS, T the set of nodes not in S but incident to some edge in PI^{in}, and U be the set of nodes with no incident edges in PI^{in}.

1. (P.1) PI^{in} is cycle free.
2. (P.2) Every node in S has at most $O(qn)$ neighbors in H.
3. (P.3) Every node in $T \cup U$ has $n - o(n)$ unexamined incident edges.

Lemma 15 *If invariants* (P.1), (P.2), *and* (P.3) *hold true then path* p *is alive.*

Proof Assume the invariants above hold true for graph H. We produce a new graph H' such that there is a unique solution consistent with the path p as follows. The edges of H' consist of PI_t^{in} and a matching for $T \cup U$ which we construct next (and no edges from PI^{out} are in H').

Note that $|S| \leq 2t$ by construction, and by (P.2), $|T| \leq O(t + qn|S|) = O(qnt) = o(n)$ (since every node in T is either incident to one of the at most t agreeable edges examined, or is a neighbor of a node in S).

Since PS is a partial matching in a bipartite graph, it has equal numbers of matched nodes in the two sides. Therefore, $T \cup U$ also has equal numbers of nodes on each side. Consider the graph on $T \cup U$, whose edges are the unexamined edges excluding edges between two nodes in T. By (P.3) and the above bound on the size of T, every node in this graph has at least $n - o(n) \geq \frac{1}{4}|T \cup U|$ edges incident to it. By Claim 14, we can find a matching, M in this graph. Note that M has no edges from $PI^{in} \cup PI^{out}$. Let the edge set of H' be $M \cup PI^{in}$.

Now we argue that H' has a unique perfect matching. $PS \cup M$ is a matching in H', since PS is a perfect matching on S and M is a perfect matching on $T \cup U$. Note that any node in U has exactly one incident edge in H', to the node it is matched to in M. Therefore, all edges in M incident to nodes in U must be included in any matching. But since all nodes in T are matched to nodes in U, this is all of M. Thus, the remaining part must be a matching on S. But all edges incident to nodes in S in H' are in PI^{in}, so if there were two such matchings, PI^{in} would contain a cycle. □

We need to prove that invariants (P.1), (P.2), and (P.3) hold true for most paths. We'll start with (P.2). We'll show something stronger: with high probability, every node has degree $O(qn)$ in H. Since H is a random bipartite graph with edge probability q, the degree of a node is the sum of n i.i.d. Boolean random variables with expectation q. Therefore, by Chernoff bounds, the probability that a node has degree $2qn$ is at most $\exp(-qn) = \exp(-n^{1/6})$ and the probability that there is such a node is at most $n \exp(-n^{1/6}) = o(1)$.

We'll use this to show that (P.1) holds with high probability (although not exponentially high). Along any path where PI^{in} contains a cycle, consider the last edge $e = \{u, v\}$ of the first cycle to be created. At the time that e was examined, the other edges in the cycle incident to u, v were already in PI^{in}, so $u, v \in S \cup T$ at that time. So we can bound the probability that there is a cycle in PI^{in} by the probability that an unexamined edge between nodes already in $S \cup T$ is in H. If (P.1) happens, $|S| + |T| = O(qnt)$ (and $S \cup T$ only grows along the path, so the final sizes are all nodes ever in $S \cup T$.) So the number of times an edge between nodes in $S \cup T$ is examined is at most $O(q^2n^2t^2)$. For each, since H is a random graph, the probability that the edge is in H is q. Thus, the probability that there is ever such an edge is $O(q^3n^2t^2 = n^{-5/2}n^2n^{1/4}) = O(n^{-1/4}) = o(1)$.

Finally, we prove the third invariant holds with high probability:

Lemma 16 (P.3) *With probability at least* $1 - o(1)$ *the number of examined edges incident to any node in* $T \cup U$ *is* $o(n)$.

Proof Fix a node u. We'll bound the probability that at least $2t + 2n^{1/8}/q = o(n)$ edges incident to u have been examined and $u \notin S$. Note that, since there are only $2t$

nodes ever in S, at most $2t$ of the edges incident to u can be competing. The rest were agreeable at the time they were examined. For each agreeable edge, independently, the edge is in H with probability q, and, if so, added to PS with probability $1/2$ by the strategy. Thus, if the above number of edges were examined, there would be at most a probability $(1-q/2)^{2n^{1/8}/q} \leq e^{-n^{1/8}}$ probability that none of them are in PS, and hence that $u \notin S$. The lemma then follows by a union bound over all n nodes u, for a total probability of $ne^{-n^{1/8}} = o(1)$. □

Now we can conclude that with probability $1-o(1)$ over the random choice of H and the randomness of the path picking strategy $\mathcal{S}$, there exists a graph H' which has a unique solution, consistent with the partial solution PS and PI^{in} is cycle free. Therefore the probability that a random path is alive is $1-o(1)$.

To prove Lemma 13 we need to bound the probability that two random paths p_1 and p_2 are both alive and can merge. Remember that by definition of being alive, such paths have no cycles in PI^{in}. Let p_1 and p_2 be two alive paths picked by the path picking strategy $\mathcal{S}$ and defined by the sets of edges $PS_1, R_1, C_1, PI_1^{out}$, and $PS_2, R_2, C_2, PI_2^{out}$, respectively.[3] Because p_1 and p_2 are alive then PI_1^{in} is cycle free and PI_2^{in} is cycle free.

We will show that, with very high probability, there exists a graph H' that is consistent with PI_1 and PI_2 and H' has a unique perfect matching consistent with P_1, but not with P_2, or vice versa.

The framework is as before: (1) We define a set of invariants that a random $H \in_{\mathcal{H}} G(n \times n, p)$ satisfies. (2) We show how this set of invariants implies we can construct from H another graph H' which has a unique solution consistent with one of the paths, but not the other. (3) Then we show that the set of invariants hold true with extremely high probability.

Consider first simulating the algorithm along randomly chosen path p_1 and then backing up and picking path p_2 (on the same underlying graph H). Edges that were examined in p_1 must be given the same values along p_2, but unexamined edges are still independent. Let S_1 be the set of matched vertices in PS_1, T_1 be the set of unmatched vertices that are adjacent to some edge in PI_1^{in}, and U_1 be the remaining set of vertices that are not adjacent to any edge in PI_1^{in}. Let S_2 be the set of matched vertices in PI_2^{in}. Let $s \in S_2 - S_1$ and $u \in U_1$. We say that u is a *unique neighbor* of s if the edge $\{u, s\}$ is the only edge incident to u in PI_2^{in} (and by definition of U_1, no such edges will exist in PI_1^{in}.) Let U be the set of nodes that have no incident edges in either PI_1^{in} nor PI_2^{in}, i.e., $U = U_1 \cap U_2$. We call an edge *unexamined* at a given stage in p_2 if it is unexamined in p_1 (ever), and unexamined up to that point in p_2. An edge is unexamined in p_1, p_2 if it is unexamined in both p_1 and in p_2.

We'll show that with all but exponentially small probability, p_1 and p_2 satisfy:

1. $PS_1 \neq PS_2$.
2. Every node not in $S_1 \cup S_2$ has $n - o(n)$ unexamined edges to nodes in U.

[3] As before, $PI_1^{in} = PS_1 \cup R_1 \cup C_1$, where $|R_1| + |PS_1| = t$ and PI_1^{out} define p_1. Likewise for p_2. $PI_1 = PI_1^{in} \cup PI_1^{out}$ and $PI_2 = PI_2^{in} \cup PI_2^{out}$.

3. Every node in $S_2 - S_1$ has $t+1$ unique neighbors in U_1, and every node in $S_1 - S_2$ has $t+1$ unique neighbors in U_2.

The following lemma is an instantiation of Lemma 13.

Lemma 17 *Suppose the above invariants hold, and p_1 and p_2 are both alive. Then there exists a graph H' such that p_1 and p_2 are consistent with H', H' has a unique perfect matching, and exactly one of PS_1 and PS_2 are consistent with this matching.*

Proof Since p_1 and p_2 are both paths for the same graph H, and $PS_1 \neq PS_2$, there had to be a first place where the paths diverged, which had to be an edge e where one accepted e, and the other rejected e. Without loss of generality, assume $e \in PS_2$ but $e \notin PS_1$.

Construct H' by first including PI_1^{in} and PI_2^{in}. Then we add the following additional edges: For each $s \in S_2 - S_1$, we choose a unique neighbor u of s in U_1, and add the edge $\{s, u\}$. Since there are $t+1$ such unique neighbors for each node, and at most t edges in R_1, we can do so in a way that does not include any edge rejected along p_1 (and, in particular, e.) Call these edges M. Let U' be the set of nodes still without incident edges. $|U'| \geq |U| - 2t$, so since every node not in $S_1 \cup S_2$ has $n - o(n)$ edges to nodes in U, the same is true for U'. Also, note that PS_1 and the edges we have just added form a perfect matching on $S_1 \cup S_2 \cup (U - U')$, so there are equal numbers of nodes on both sides within that set, and hence within the remaining nodes. All nodes in $U' \cup (V - U - S_1 - S_2)$ have $n - o(n)$ unexamined edges to U', so by Lemma 14, there is a perfect matching N of such nodes using unexamined edges where at least one endpoint is in U'. Let $H' = PI_1^{in} \cup PI_2^{in} \cup M \cup N$.

Note that $PS_1 \cup M \cup N$ is a perfect matching in H'. On the other hand, every node in U' has a unique incident edge in H', the edge in N. Thus, in any matching all edges in N must be used. Also, for each edge $\{s, u\}$ in M, u was a unique neighbor of s, and so had no other edges in PI^{in}. Thus, all edges in M must be used. This covers all nodes except for S_1. So any matching must be $M \cup N$ and a perfect matching on S_1. Since the algorithm examines all competing edges once an edge is added to PS_1 , every edge incident to a node in S_1 is in PI_1^{in}. Therefore, if there were two matchings on S_1 in PI^{in}, there would be two such matchings in PI_1^{in}, and hence a cycle in PI^{in}, contradicting the assumption that p_1 is alive.

Therefore, H' has a perfect matching. Note that this matching is consistent with PS_1, since we include all edges in PS_1, and have not included any edge in R_1. (M avoids such edges, and N only contains unexamined edges.) It is not consistent with PS_2, since it does not contain e. □

Next we estimate the probabilities that the invariants hold true. Since we are looking at paths for the same graph, the probability that $PS_1 = PS_2$ is equal to the probability that the branching strategy makes the same sequence of decisions. Since we make t random decisions along the two paths, this is exactly 2^{-t}.

The second invariant is similar to the previous lemma. Fix a node u. At most $O(t)$ edges incident to u are competing (along both paths). Every agreeable edge is in $\mathcal{H}$ with probability q, and then put in PS_b (if we are in path p_b when the edge is examined) with probability $1/2$. Thus, the probability that more than

$O(t + n^{1/8}/q) = o(n)$ edges are examined but $u \notin S_1 \cup S_2$ is exponentially small in $n^{1/8}$. Since $|U_1| = n - o(n)$, as is $|U_2|$, most of the $n - o(n)$ unexamined edges must be to nodes in U.

Finally, we prove the third invariant. Consider the first time a node s in $S_2 - S_1$ is added to S_2. At this time, there are (for the reason given before) with very high probability, $n - o(n)$ unexamined edges from s to $U = U_1 \cap U_2$. With all but exponentially low probability in $\Omega(qn)$ there are $\Omega(qn)$ such nodes adjacent to s. Call this set of nodes D. These are currently unique neighbors of s in U_1, and only fail to be unique neighbors at the end if some incident edge is later added to PI_2^{in}. Thus, the number of edges later added to PI^{in} incident to nodes in D bounds the number of nodes in D that are no longer unique neighbors of s. Such edges are either agreeable or competing edges. There are at most t agreeable edges considered, and at most $2t$ competing edges incident to each node in D will be examined (one per node added to S_2.) Each such edge is in PI_2^{in} independently with probability q. Thus, the expected number of competing edges incident to D is $q|D|$ and the probability that it is greater than $|D|/2$ is exponentially small in the size of $|D| = \Omega(qn)$. Thus, with probability $1 - O(\exp(-\Omega(qn) = 1 - \exp(-\Omega(n^{1/6}))$, u will have $\Omega(qn) - t = \Omega(qn) > t + 1$ unique neighbors in U_1. We then take a union bound over all nodes $s \in S_2 - S_1$, and by symmetry, the same will hold with the same probability for all $s \in S_1 - S_2$.

Thus, all invariants hold except with probability

$$O\left(\max\{2^{-t}, 2n\exp(-\Omega(n^{1/6})), \exp(-\Omega(n^{1/8}))\}\right) = O(\exp -\Omega(n^{1/8})).$$

Next we estimate the bound on the width of pBP with values on states $\{0, 1\}$. We showed that the probability a random path is alive is $1 - o(1)$ together with the result of Lemma 17 we conclude the following theorem.

Theorem 18 *Any fully adaptive pBP algorithm $\mathcal{A}$ with values on states computed in stage II in $\{0, 1\}$ will require $size_{\mathcal{A}_n} \geq e^{\Omega(n^{1/8})}$ for the perfect matching problem in bipartite graphs of order n.*

Applying Lemma 3, we get

Corollary 19 *Any fully adaptive pBP algorithm for maximum bipartite matching that uses a subexponential number of values in stage II will require exponential size.*

Corollary 20 *Any fully adaptive pBT algorithm $\mathcal{A}$ requires $size_{(A_n)} \geq \Omega(e^{n^{1/8}})$ for the maximum matching problem in bipartite graphs of order n.*

6 Future Work

There are many directions in which we would like to extend the current work. Here we briefly mention a few:

1. While most dynamic programming algorithms we have encountered fit into our model, some don't. Perhaps most notable is the optimal binary search tree algorithm and the matrix chain multiplication algorithm. It would seem that extending

our model to allow nondeterministic branching and partitioning of the set of items would allow us to compute it.
2. Our bipartite matching lower bound suggests that there are problems that can be computed by a simple application of linear programming but not by dynamic programming. Can we formulate an interesting result (perhaps along the lines of [5]) in the other direction? Perhaps knapsack is a good candidate as there is a dynamic programming FPTAS, but the integrality gap of the obvious linear relaxation can be arbitrarily bad.
3. Does it help a pBP algorithm to use high precision in the state-values computed in stage II of the computation? Of course this question is related to whether our lower bound holds for the general case and whether our results separate the fixed, adaptive and fully adaptive versions. Does rounding these values lead to some sort of approximation algorithm?
4. In general, can we find any algorithm that fits this model which we did not heretofore consider dynamic programming?
5. Can we separate fully adaptive pBT and pBP without the "promise" issue described above.

Acknowledgement The authors are thankful to Allan Borodin, Jeff Edmonds, Avner Magen and Toni Pitassi.

References

1. Angelopoulos, S., Borodin, A.: The power of priority algorithms for facility location and set cover. Algorithmica **40**(4), 271–291 (2004)
2. Alekhnovich, M., Borodin, A., Buresh-Oppenheim, J., Impagliazzo, R., Magen, A., Pitassi, T.: Toward a model for backtracking and dynamic programming. In: IEEE Conference on Computational Complexity, pp. 308–322 (2005)
3. Arkin, E.M., Silverberg, E.L.: Scheduling jobs with fixed start and end times. Discrete Appl. Math. **18**, 1–8 (1987)
4. Arora, S., Bollobás, B., Lovász, L.: Proving integrality gaps without knowing the linear program. In: FOCS, pp. 313–322 (2002)
5. Arora, S., Bollobás, B., Lovász, L., Tourlakis, I.: Proving integrality gaps without knowing the linear program. Theory Comput. **2**(1), 19–51 (2006)
6. Bellman, R.: Combinatorial processes and dynamic programming. In: Proceedings of the Tenth Symposium in Applied Mathematics of The American Mathematical Society, pp. 24–26 (1958)
7. Borodin, A., Nielsen, M.N., Rackoff, C.: (Incremental) priority algorithms. Algorithmica **37**(4), 295–326 (2003)
8. Borodin, A., Boyar, J., Larsen, K.S.: Priority algorithms for graph optimization problems. In: WAOA, pp. 126–139 (2004)
9. Borodin, A., Cashman, D., Magen, A.: How well can primal-dual and local-ratio algorithms perform? In: ICALP, pp. 943–955 (2005)
10. Chvatal, V.: Hard knapsack problems. Oper. Res. **28** (1980)
11. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms. MIT Press, Cambridge (2001)
12. Davis, S., Impagliazzo, R.: Models of greedy algorithms for graph problems. In: SODA, pp. 381–390 (2004)

13. Helman, P.: A common schema for dynamic programming and branch and bound algorithms. J. ACM **36**(1), 97–128 (1989)
14. Helman, P., Rosenthal, A.: A comprehensive model of dynamic programming. SIAM J. Algebr. Discrete Methods **6**, 319–334 (1985)
15. Karp, R., Held, M.: Finite state processes and dynamic programming. SIAM J. Appl. Math. **15**, 693–718 (1967)
16. Khanna, Motwani, Sudan, Vazirani: On syntactic versus computational views of approximability. SICOMP: SIAM J. Comput. **28** (1998)
17. Regev, O.: Priority algorithms for makespan minimization in the subset model. Inf. Process. Lett. **84**(3), 153–157 (2002)
18. Robson, J.M.: Algorithms for maximum independent sets. J. Algorithms **7**(3), 425–440 (1986)
19. Rosenthal, A.: Dynamic programming is optimal for nonserial optimization problems. SIAM J. Comput. **11**(1), 47–59 (1982)
20. Woeginger, G.J.: 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)