On the use of primal and dual knowledge in randomized constructive optimization algorithms

[1]Monaldo Mastrolilli [2]Christian Blum

[1]IDSIA, Galleria 2, 6928 Manno, Switzerland
monaldo@idsia.ch

[2]ALBCOM, LSI, Universitat Politècnica de Catalunya, Barcelona, Spain
cblum@lsi.upc.edu

Abstract

Many approximate heuristics for optimization are either based on neighborhood search or on the construction of solutions. Examples for the latter ones include ant colony optimization and greedy randomized adaptive search procedures. These techniques generally construct solutions probabilistically by sampling a probability distribution over the search space. Solution constructions are generally independent from each other. Recent algorithmic variants include two important features that are inspired by deterministic branch & bound derivatives such as beam search: the use of bounds for evaluating partial solutions, and the parallel and non-independent construction of solutions. In this paper we first give a theoretical reason of why these variants have the potential to improve over standard algorithms. Second, we confirm our theoretical findings by means of practical examples. Our results for the open shop scheduling problem clearly demonstrate the potential of using parallel and non-independent solution constructions.

Keywords: combinatorial optimization; probabilistic tree search; solution construction.

1 Introduction

Many heuristic search methods for combinatorial optimization are based on one of the following two principles: neighborhood search or solution construction. In neighborhood search is given a neighborhood function that assigns to each candidate solution a so-called neighborhood, which is a subset of the search space. Heuristic methods based on neighborhood search follow a trajectory in the directed graph G whose node set is the search space. A node v is connected to a node w by an arc if and only if w is in the neighborhood of v. The trajectories in G may be deterministic as in the case of a simple tabu search method [6], or they might result from a random process as in the case of simulated annealing [8].

On the other side, optimization techniques based on solution construction explore the search space in form of a search tree which is defined by the solution construction mechanism. Following a path from the root node to a leaf corresponds to the process of constructing a candidate solution. Inner nodes of the tree can be seen as partial solutions. The process of

moving from an inner node to one of its child nodes is called a construction step. A prominent example of deterministic constructive algorithms are greedy heuristics. They make use of a weighting function that gives weights to the child nodes of each inner node of the search tree. At each construction step the child node with the highest weight is chosen. Metaheuristics such as ant colony optimization (ACO) [4] or greedy randomized adaptive search procedures (GRASP) [5] employ repeated probabilistic (or randomized) solution constructions. For each inner node of the tree and each child node is given the probability of performing the corresponding construction step. These probabilities, which may depend on weighting functions and/or the search history of the algorithm, define a probability distribution over the search space. In this work we refer to the resulting probability of generating a satisfying—for example, optimal—solution as the **primal problem knowledge**.

Recent variants of metaheuristics such as ACO and GRASP include two important features that are inspired by deterministic branch & bound derivatives such as beam search [14]:

1. Lower or upper bounds are used for evaluating partial solutions; sometimes also for choosing among different partial solutions, or discarding partial solutions. Henceforth we will refer to this type of knowledge as the **dual problem knowledge**.

2. The extension of partial solutions may be done in more than one way. The number of nodes which can be selected at each tree level is usually limited from above by a parametric constraint, resulting in parallel and non-independent solution constructions.

In the literature exist only a few examples of these type of algorithms, including approximate and non-deterministic tree search (ANTS) procedures [11, 12, 13], and probabilistic beam search derivatives such as (Beam-ACO) [1, 2] and probabilistic beam search (PBS) [3]. These works give empiricial evidence for the usefulness of optimization algorithms that use randomized solution constructions incorporating the two features mentioned above.

The motivation of this paper is to study the reasons of why optimization algorithms using (randomized) parallel and non-independent solution constructions may have an advantage over standard versions that use independent solution constructions. The organization of the paper is as follows. In section 2 we introduce a tree search model (as a model for understanding the construction of solutions). In section 3 we introduce the notions of primal and dual problem knowledge more in detail, showing by means of examples why the exploitation of the dual problem knowledge may be beneficial. In section 4 we theoretically analyse a simple algorithm for exploiting the dual problem knowledge, while in section 5 we confirm our theoretical findings by means of practical examples.

2 A tree search model

Many exact or approximate algorithms that are commonly used in practice for the solution of NP-hard combinatorial optimization problems build solutions step by step. We refer to these algorithms as *constructive optimization algorithms* and describe a model that captures the essential elements common to all constructive procedures.

In general, we are given an optimization problem $\mathcal{P}$ and an instance x of $\mathcal{P}$. Typically, the set S_x of possible solutions is exponentially large in the size of the input x. The goal is

to find a solution to x belonging to some set $\mathrm{Sat}_x \subseteq S_x$ of satisfactory solutions.[1] Assume that each element y of Sat_x can be viewed as a composition of $l_{y,x} \in \mathbb{N}$ elements from a set Σ. From this point of view, Sat_x can be seen as a set of strings over an alphabet Σ. Any element y of Sat_x can be constructed by concatenating $l_{y,x}$ elements of Σ.

The following method for constructing elements of Sat_x is instructive: A solution construction starts with the empty string $Y = \epsilon$. The construction process consists of a sequence of construction steps. At each construction step, we select an element of Σ and append it to Y. The solution construction ends when either a leaf node of the tree is reached, or when it becomes clear that Y cannot be extended into any element of Sat_x (i.e., that no element of Sat_x has prefix Y). An algorithm of this kind can be described equivalently as a walk from the root v_0 of a tree to a node at level $l_{y,x}$ for any $y \in \mathrm{Sat}_x$. The tree has nodes for all $y \in \mathrm{Sat}_x$ and for all prefixes of elements of Sat_x. The root of the tree is the empty string. There is a directed arc from node v to node w if w can be obtained by appending an element of Σ to v (i.e. if $\exists a \in \Sigma : w = va$). The set of nodes that can be reached from a node v via directed arcs are called the children of v. Note, that the nodes at level i correspond to strings of length i. If v is a node corresponding to a string of length $l > 0$ then the length $l - 1$ prefix w of v is also a node. Thus, for every $y \in \mathrm{Sat}_x$, there is a path of length $l_{y,x}$ from the root to y. In addition to Sat_x and all its prefixes, the tree may contain other nodes (which correspond to unsatisfactory solutions). It is not necessary to store the entire tree. An efficiently computable predicate P on the set of strings, such that $P(v)$ is true if and only if string v is a node of the tree, is sufficient. P can be viewed as a pruning procedure. The children of any given node w can be computed by evaluating $P(wa)$ for all $a \in \Sigma$. This is sufficient for our purposes. We use the described tree search model as basis of our discussion.

Assumptions. The analysis (but not the algorithms) provided in this paper assumes that there is a unique satisfactory leaf node v_d, i.e. $\mathrm{Sat}_x = \{v_d\}$. Extensions to more general cases are left for future research. Without loss of generality, the target node v_d is at the maximum level $d \geq 1$ of the search tree. Moreover, let c denote the maximum number of children of any given node. We assume that the values of c and d are bounded by a polynomial in the input size. A probabilistic constructive optimization algorithm is said to be *successful* if it can find the target node v_d with high probability.

3 Primal and dual problem knowledge

Metaheuristics such as ACO and GRASP utilize repeated probabilistic solution constructions where each construction step is performed probabilistically. Being at node v, the probability to move to a child w of v is denoted by $Pr[w|v]$. At each node may also exist a certain probability $Pr[stop|v]$ for stopping (aborting) the solution construction. Note that these probabilities (sometimes called transition probabilities) define a probability distribution over the search space. Summarizing, a probabilistic solution construction works as follows: the process starts at the root v_0 of the search tree, and repeatedly chooses a child of the current node at random (i.e. with respect to the given probability distribution) until a leaf node is reached or the decision to stop has been made.

[1] The definition of what exactly a satisfactory solution is depends on the optimization goal. For example, we might want to find an optimal, or just a good-enough solution, or we might even be satisfied with any feasible solution.

In the following let us examine the success probability of repeated applications (or runs) of such a probabilistic solution construction. Given any node v_i at level i of the search tree, let $Pr[v_i]$ be the probability that node v_i is visited during the solution construction. Note that there is a single path from v_0, the root node, to v_i: we denote the corresponding sequence of nodes by $(v_0, v_1, v_2, ..., v_i)$. Clearly, $Pr[v_0] = 1$ and $Pr[v_i] = \prod_{j=0}^{i-1} Pr[v_{j+1}|v_j]$. Let $Success(\rho)$ denote the event of finding the target node v_d within ρ independent runs (that is, repetitions).[2] Note that the probability of $Success(\rho)$ is equal to $1 - (1 - Pr[v_d])^\rho$, and it is easy to check that the following inequalities hold:

$$1 - e^{-\rho Pr[v_d]} \leq 1 - (1 - Pr[v_d])^\rho \leq \rho Pr[v_d] \tag{1}$$

By (1), it immediately follows that the chance of finding node v_d is *large* if and only if $\rho Pr[v_d]$ is *large*, namely as soon as

$$\rho = O\left(1/Pr[v_d]\right) \quad . \tag{2}$$

In the following, we will not assume anything about the exact form of the given probability distribution. However, let us assume that the transition probabilities are heuristically related to the *attractiveness* of child nodes. In other words, we assume that in a case in which a node v has two children, say w and q, and w is known (or believed) to be *more promising*, then $Pr[w|v] > Pr[q|v]$. This can be achieved, for example, by defining the transition probabilities proportional to the weights assigned by greedy functions. Moreover, at any node v we can also have a non-zero probability $Pr[stop|v]$ that a solution construction is aborted at this point. This probability $Pr[stop|v]$ is equal to 1, if it is clear that the partial solution represented by node v can not be completed such that the result is a satisfyable solution; otherwise this probability may be related to the expectation that v can lead to a satisfyable solution, with the larger the expectation the smaller the probability to stop. This stopping probability is in most cases obtained by bounding information, and strongly influences the performance of many exact or approximate algorithms (see, for example, the pruning procedures in branch and bound algorithms). In the following we model the non-zero stopping probabilities by means of an additional virtual child node, that is, each node v with $Pr[stop|v] > 0$ has an additional virtual child node called *stop*. We call these nodes *virtual nodes*, because they are not really part of the search tree.

Clearly, the probability distribution reflects the available knowledge on the problem, and it is composed of two types of knowledge. If the probability $Pr[v_d]$ of reaching the target node v_d is "high", then we have a "good" problem knowledge. Let us call the knowledge that is responsible for the value of $Pr[v_d]$ the **primal problem knowledge** (or just primal knowledge). From the dual point of view, we still have a "good" knowledge of the problem if for "most" of the wrong nodes (i.e. those that are not on the path from v_0 to v_d) the probability that they are reached is "low". We call this knowledge the **dual problem knowledge** (or just dual knowledge). Observe that these two types of knowledge are complementary, but not the same. Let us make an example to clarify these two concepts. Consider the search tree of Figure 1, where the target node is v_5. Let us analyze two different probability distributions:

Case (a) For each v and a child w of v let $P[w|v] = 0.5$. Moreover, the stopping probabilities at all nodes are zero. This means that when probabilistically constructing a solution the probability of each child is the same at each construction step.

[2] Remember that the path from the root node to v_d corresponds to the unique satisfying solutions.

Case (b) The transition probabilities are defined as in case (a), but the stopping probabilities are set to 1 in the black nodes, i.e. $Pr[stop|black\ node] = 1$. This means that each of the black nodes has a virtual child that has probability 1, and the white children of the black nodes have probability 0.

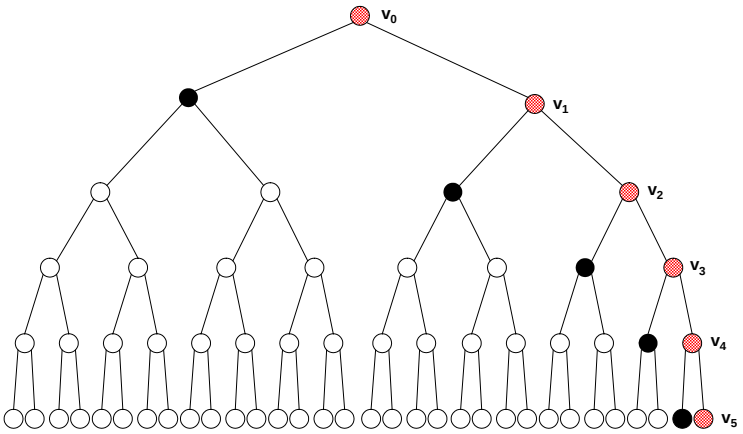


Figure 1: Example of primal and dual knowledge.

Note that in both cases the primal knowledge is "scarce", since the probability that the target node v_d is reached decreases exponentially with d, i.e. $Pr[v_d] = 2^{-d}$. However, in **case (b)** the dual knowledge is "excellent", since for most of the wrong nodes (i.e. the white nodes), the probability that any of them is reached is zero. Viceversa, in **case (a)** the dual knowledge is still "scarce", i.e. there is a "high" probability that a white node is reached.

By using the intuition given by the provided example, let us try to better quantify the quality of the available problem knowledge. Let V_i be the set of nodes at level i, and let

$$\ell(i) = \sum_{v \in V_i} Pr[v], \tag{3}$$

for $i = 1, \ldots, d$, which is equal to the probability that the solution construction process reaches level i of the search tree. We observe that the presence of the non-zero stopping probabilities (namely, $Pr[stop|black\ node] = 1$ in the example), can make the probabilities $\ell(i)$ smaller than one. **Case (b)** was obtained from **case (a)** by decreasing $\ell(i)$ (for $i = 1, \ldots, d$) down to 2^{i-1} (and without changing the probability $Pr[v_i]$ of reaching the ancestor v_i of the target node at level i), whereas in **case (a)** it holds that $\ell(i) = 1$ (for $i = 1, \ldots, d$). In general, good dual knowledge is supposed to decrease $\ell(i)$ without decreasing the probability of reaching the ancestor v_i of the target node v_d. This discussion may suggest that a characterization of the available problem knowledge can be given by the following *knowledge ratio*

$$K_{v_d} = \min_{1 \leq i \leq d} \frac{Pr[v_i]}{\ell(i)}, \tag{4}$$

with the larger the ratio the better the knowledge we have on the target node v_d. In **case (a)** it is $K_{v_d} = 1/2^d$, whereas the knowledge ratio of **case (b)** is $K_{v_d} = 1/2$, which is exponentially larger.

Finally, it is important to observe that the way of (repeatedly) constructing solutions in a probabilistic way does not exploit the dual problem knowledge. For example in **case (b)**, although the available knowledge is "excellent", the target node v_d is found after an expected number of runs that is proportional to $1/Pr[v_d] = 2^d$ (see Equation (2)), which is the same as in **case (a)**. In other words, the number of necessary runs only depends on the primal knowledge.

4 Expoiting the dual knowledge

Examples of constructive optimization algorithms that use and exploit the dual knowledge are incomplete derivatives of branch & bound procedures [9] such as, for example, beam search [14]. Their central ideas are as follows. First, they allow the extension of partial solutions in more than one way. Second, bounding information—in the case of minimization we talk about lower bounds—are used to evaluate partial solutions. The bounding information is used for the definition of the dual knowledge. For example, in case the lower bound of a partial solution is greater than the value of any satisfyable solutions, the stopping probability at this partial solution is set to one. These algorithms generally bound the number of partial solutions per tree level from above. In fact, they might be characterized as constructive optimization algorithms that employ **parallel and non-independent solution constructions**. In recent years, versions of these probabilistic algorithms were proposed in which the parallel and non-independent solution constructions are performed probabilistically in an iterative (or repeated) way. Examples are approximate and non-deterministic tree search (ANTS) procedures [11, 12, 13] and probabilistic beam search derivatives such as (Beam-ACO) [1, 2] or probabilistic beam search (PBS) [3]. The usefulness of these algorithms has been shown empirically by the application to various combinatorial optimization problems such as set covering, assembly line balancing, or shop scheduling.

With the aim of deriving theoretical evidence of why these algorithms have the potential to improve over the standard versions, we now define a simple algorithm that is, in a sense, representative for the above mentioned techniques. This algorithm—henceforth denoted by PTS(α) (for probabilistic tree search)—is pseudo-coded in Algorithm 1. Hereby, α denotes the maxiumum number of allowed extensions of partial solutions at each construction step; in other words, α is the maximum number of solutions to be constructed in parallel. We use the following additional notation: For any given set S of tree nodes, let $\mathcal{C}(S)$ be the set of children of the nodes in S, including any virtual stopping nodes.

The algorithm works as follows. At each construction step $i = 1, \ldots, d$ we have given a set of nodes R_{i-1} that have been reached in construction step $i-1$. As mentioned above, $\mathcal{C}(R_{i-1})$ denotes the set of nodes that can be reached from the nodes in R_{i-1}. Given $\mathcal{C}(R_{i-1})$, α probabilistic choices of nodes from $\mathcal{C}(R_{i-1})$ are performed, resulting in the set R_i of nodes.

Algorithm 1 PTS(α)

Require: $\alpha \in \mathbb{Z}^+$
Initialization: $R_0 \Leftarrow \{v_0\}$, $i \Leftarrow 1$ and $R_j \Leftarrow \emptyset$ for $j = 1, 2, \ldots, d$;
  while $\mathcal{C}(R_{i-1})$ is not empty and a target node is not reached **do**
    for $k = 1, \ldots, \alpha$ **do**
      Choose $w \in \mathcal{C}(R_{i-1})$ w.r.t. the probabilities $Pr[v|\mathcal{C}(R_{i-1})]$, $\forall\, v \in \mathcal{C}(R_{i-1})$;
      if $w \notin R_i$ **then**
        $R_i \Leftarrow R_i \cup \{w\}$;
      end if
    end for
    $i \Leftarrow i + 1$;
  end while.

The probability $Pr[w|\mathcal{C}(R_{i-1})]$ of a node $w \in \mathcal{C}(R_{i-i})$ to be chosen is defined as follows:

$$Pr[w|\mathcal{C}(R_{i-1})] = \frac{Pr[w]}{\sum\limits_{v \in \mathcal{C}(R_{i-1})} Pr[v]} \tag{5}$$

Observe that for any $w \in \mathcal{C}(R_{i-1})$, $Pr[w|\mathcal{C}(R_{i-1})]$ is equal to the probability that w is reached with a single probabilistic solution construction starting from the root, when this construction is limited to the subtree defined by the nodes that were reached at levels $j = 1, \ldots, i-1$.

4.1 Theoretical analysis of PTS(α)

In this subsection we study the probability of PTS(α) for reaching the target node v_d. We prove (see Theorem 1) that PTS(α) is successful with "high" probability as soon as α is of the order of $1/K_{v_d}$.[3] This means that the greater the knowledge ratio the smaller the required number of parallel solution constructions.

On the probability of reaching one single node. First, remember that there is a single path from v_0, the root, to v_d: as before we denote the corresponding sequence of nodes by $(v_0, v_1, v_2, ..., v_d)$. We associate to any node v the variable indicator I_v defined as follows

$$I_v = \begin{cases} 1 & \text{if } v \text{ is reached,} \\ 0 & \text{otherwise.} \end{cases}$$

For simplicity of notation we use $\Pr[I_v]$ (and $\Pr[\bar{I}_v]$) to denote $\Pr[I_v = 1]$ (and $\Pr[I_v = 0]$).

Theorem 1 *If $\alpha \geq \frac{1}{K_{v_d}} \ln(d+1)$ then the probability that PTS(α) reaches v_d is larger than $1/e$, namely $\Pr[I_{v_d}] > 1/e$.*

Proof. The probability of reaching v_d can be recursively computed as follows:

$$\Pr[I_{v_d}] = \Pr[I_{v_d}|I_{v_{d-1}}] \cdot \Pr[I_{v_{d-1}}]. \tag{6}$$

[3] As a matter of fact, the sufficient size of α is a bit larger than $1/K_{v_d}$ by a factor of $\log(d+1)$, which is indeed a "small" factor. The reader is referred to Theorem 1 for a more precise statement.

Probability $\Pr[I_{v_i}|I_{v_{i-1}}]$ can be put in the following form:

$$\Pr[I_{v_i}|I_{v_{i-1}}] = \sum_{R_{i-1}:v_{i-1}\in R_{i-1}} \Pr[I_{v_i} \wedge R_{i-1}|I_{v_{i-1}}],$$

where $Pr[I_{v_i} \wedge R_{i-1}|I_{v_{i-1}}]$ is the probability of reaching v_i and all nodes in R_{i-1}, given that v_{i-1} is reached and R_{i-1} is any subset of nodes at level $i-1$ such that $v_{i-1} \in R_{i-1}$. By using standard theorems from probability theory, we obtain the following derivations:

$$\begin{aligned}
\Pr[I_{v_i}|I_{v_{i-1}}] &= \sum_{R_{i-1}:v_{i-1}\in R_{i-1}} \Pr[I_{v_i}|I_{v_{i-1}} \wedge R_{i-1}] \cdot \Pr[R_{i-1}|I_{v_{i-1}}] \\
&= \sum_{R_{i-1}:v_{i-1}\in R_{i-1}} (1 - \Pr[\bar{I}_{v_i}|I_{v_{i-1}} \wedge R_{i-1}]) \cdot \Pr[R_{i-1}|I_{v_{i-1}}] \\
&= \sum_{R_{i-1}:v_{i-1}\in R_{i-1}} (1 - (1 - Pr^*[v_i|I_{v_{i-1}} \wedge R_{i-1}])^\alpha) \cdot \Pr[R_{i-1}|I_{v_{i-1}}],
\end{aligned}$$

where $Pr^*[v_i|I_{v_{i-1}} \wedge R_{i-1}] = \frac{Pr[v_i]}{\sum_{s\in\mathcal{C}(R_{i-1})} Pr[s]}$, that is the probability that the algorithm reaches v_i given that v_{i-1} and the nodes from R_{i-1} are reached (see Equation 5). Since $\ell(i) = \sum_{w\in V_i} Pr[w] \geq \sum_{s\in R_{i-1}} Pr[s]$, we get the following bound

$$\begin{aligned}
\Pr[I_{v_i}|I_{v_{i-1}}] &\geq \sum_{R_{i-1}:v_{i-1}\in R_{i-1}} \left(1 - \left(1 - \frac{Pr[v_i]}{\ell(i)}\right)^\alpha\right) \cdot \Pr[R_{i-1}|I_{v_{i-1}}] \\
&= 1 - \left(1 - \frac{Pr[v_i]}{\ell(i)}\right)^\alpha.
\end{aligned}$$

So by induction on Equation (6) (we use $e^{-\frac{t}{1-t}} < 1 - t < e^{-t}$, for $0 < t < 1$), we get

$$\begin{aligned}
\Pr[I_{v_d}] &\geq \prod_{i=1}^{d} \left(1 - \left(1 - \frac{Pr[v_i]}{\ell(i)}\right)^\alpha\right) \geq (1 - (1 - K_{v_d})^\alpha)^d \\
&> \left(1 - e^{-\alpha K_{v_d}}\right)^d > \exp\left(-\frac{e^{-\alpha K_{v_d}}}{1 - e^{-\alpha K_{v_d}}} d\right).
\end{aligned}$$

The claim follows by observing that $\exp\left(-\frac{e^{-\alpha K_{v_d}}}{1-e^{-\alpha K_{v_d}}} d\right) \geq e^{-1}$ as soon as $\alpha \geq \frac{1}{K_{v_d}} \ln(d+1)$. ■

Discussion. By Theorem 1, choosing $\alpha \geq \frac{1}{K_{v_d}} \ln(d+1)$ is a sufficient condition to ensure a "high" success probability of reaching v_d. Observe that the "important" value is given by $\frac{1}{K_{v_d}}$, whereas $\ln(d+1)$ is just a "small" factor (recall that we assume d to be polynomial in the input size), and therefore we omit it in the following discussion.

Recall that c denotes the maximum number of children of any node. The proposed algorithm PTS(α), when $\alpha = O\left(\frac{1}{K_{v_d}}\right)$, performs an expected number of extensions of partial solutions that can be easily bounded from above by $O(d \cdot c/K_{v_d})$. Viceversa, PTS($\alpha = 1$) needs an expected number of extensions of partial solutions to reach v_d that can be bounded by $O(d \cdot c/Pr[v_d])$. Again, the "important" factors are, respectively, $1/K_{v_d}$ for PTS($\alpha = O(1/K_{v_d})$, and $1/Pr[v_d]$ for PTS($\alpha = 1$). Observe that $K_{v_d} \geq Pr[v_d]$, and actually K_{v_d} can

be also exponentially larger than $Pr[v_d]$ in many settings (see, for example, Figure 1). The reason of this potential advantage is due to the fact that when $\alpha = 1$ the algorithm only uses the primal knowledge, whereas when $\alpha > 1$ the primal and the dual knowledge are exploited by parallel non-independent solution constructions. This gives evidence that a setting of $\alpha > 1$ may help to achieve substantial speedups when compared to PTS($\alpha = 1$), thanks to a better exploitation of the available problem knowledge.

At this point, the reader may object that the value of $\frac{1}{K_{v_d}} \ln(d+1)$ is not known a priori, even though the knowledge of that value is essential for "a correct" setting of α and therefore also for the success of PTS(α) (see Theorem 1). In the following Section we present a kind of self-tuning strategy for setting α in order to deal with this problem.

4.2 Self-tuning: A strategy for setting α

In the following we assume that the expected running time t_α of PTS(α) (for any α) can be approximated by an appropriate polynomial function of α, i.e. $t_\alpha = O(\alpha^\tau)$, for a suitably chosen constant $\tau \geq 0$. Actually, the value of t_α is at least $\Omega(\alpha)$ and at most $O(c \cdot d \cdot \alpha)$.

The general idea is to run PTS(α) repeatedly: run PTS(α) with $\alpha = \alpha_1 \geq 1$; if $PTS(\alpha_1)$ finds the desired target node we are done, otherwise restart PTS(α) from the beginning and run it with $\alpha = \alpha_2 \geq 1$, and so on. Such an experiment can be described by a *strategy* $S = (\alpha_1, \alpha_2, \alpha_3, \ldots)$, which is an infinite sequence of values from the set $\mathbb{Z}^+ \cup \{\infty\}$. We henceforth denote this algorithm by MS-PTS(S), which stands for multi-start PTS using strategy S. It is easy to see that any strategy S with $\alpha_i \geq 1$ causes the algorithm, at some point, to find the target node. The running time, however, is a random variable, i.e. the resulting algorithm is a Las Vegas algorithm.[4] Let $\delta = \left\lceil \frac{1}{K_{v_d}} \ln(d+1) \right\rceil$. By Theorem 1, we know that by choosing $\alpha \geq \delta$ there is a "high" probability that algorithm PTS(α) reaches the target node v_d. If we knew δ, then strategy $S^* = (\delta, \delta, \delta, \ldots)$ would be a (near-) optimal strategy that would be successful after only a few applications of PTS(δ).

Unfortunately, we do not know δ a priori. The ideas of an alternative strategy presented in the following are taken from Luby et al. [10]. Let T_{S^*} be the expected running time of MS-PTS(S^*) with S^* as described above. By Theorem 1 we have that $T_{S^*} = O(t_\delta)$. Next, we explain that, with no knowledge about δ, the expected time to reach the target node can be always bounded by $O(t_\delta \log t_\delta)$. This performance is achieved by a pure strategy $S^u = (\alpha_1, \alpha_2, \alpha_3, \ldots)$—henceforth called the *universal strategy*—of a very simple form that is easy to implement in practice. Formally, this strategy can be described as follows:

$$\alpha_i = \begin{cases} 2^{k-1} & \text{if } i = 2^k - 1, \\ \alpha_{i-2^{k-1}+1} & \text{if } 2^{k-1} \leq i \leq 2^k - 1. \end{cases}$$

This strategy starts as follows:

$$S^u = (1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, 1, \ldots)$$

Note that all values of α are powers of two, and each time a pair of runs of a given α has been completed, a run of twice that α is immediately executed.

[4] A Las Vegas algorithm A is a randomized algorithm that always produces the correct answer when it stops, but whose running time is a random variable.

By using the ideas in [10], it can be promptly proven that the expected running time of MS-PTS(S^u) is at most $O(t_\delta \log t_\delta)$, which is only a logarithmic factor slower than the expected time of MS-PTS(S^*). The intuition for the bound (see [10] for more details) is that S^u uses the following geometrically increasing values for α: $1, 2, 2^2, ..., 2^j, ...$, and stops the first time PTS(α) succeeds. Clearly, there exists an integer $n > 0$ such that $2^{n-1} \leq \delta \leq 2^n$. Once the strategy has performed a few runs with $\alpha = \delta$ (i.e. 2^n), it will succeed with a fairly high probability. The total time spent on runs with $\alpha = \delta$ will be about $O(t_\delta)$. But the strategy is "balanced", in the sense that the total time spent on runs with different settings of α is roughly equal. Since the number of different settings of α used up to this time is about $\log \delta$, the total running time can be bounded from above by $O(t_\delta \log t_\delta)$ (by our assumption on the expected running time t_α of PTS(α)).

5 Experimental evaluation

In the following we present empirical results that confirm our theoretical findings from the previous section. First, we present an experimental evaluation of MS-PTS(S^u) on artificial search trees. Then, we present the application of MS-PTS($\cdot$) using different strategies (including strategy S^u) to a real combinatorial optimization problem, the open shop scheduling (OSS) problem.

5.1 Experimental results with artificial search trees

We implemented a graphical simulation tool (AntSim) in Java. This tools provides the possibility to load existing search trees and/or draw new search trees. Furthermore, it allows to follow the simulation of the proposed algorithm step by step.[5] The goal of the following experimental evaluation is to validate the claimed properties of MS-PTS($\cdot$). With this aim, we needed to have full control on the structure of the input trees. Therefore, we generated a set of artificial trees with a "low" primal knowledge and a "high" knowledge ratio. The outline of the input tree generation procedure is as follows.

The procedure for producing a tree depends on 3 input parameters, namely c, d and K_{min}, where K_{min} is the minimally required knowledge ratio for each level. The chosen value combinations for these 3 parameters are displayed in the second column of Table 1 (see Appendix A). The target node v_d is at the maximum level d. The tree generation starts by producing the path from the root node to the target node, consisting of $d+1$ nodes. Then, the remaining tree is built level by level, node by node, starting from the root; we considered the nodes of each level in a random order, and for each node j we computed the maximum number $c(j)$ of children that j can have without violating the requirement that the knowledge ratio must be larger or equal to K_{min}; a random number of children between zero and $\min\{c, c(j)\}$ is added to the tree, where c is the overall maximum number of children of any node (a fixed parameter). The process is repeated for d levels. For each combination of parameters we generated a set of 10 instances, resulting in a total of 40 input trees. The probabilities for each solution extension are fixed such that each child node of a node has the same probability as the others. This means we have a low primal knowledge. The stopping probabilities are equal to one at any leaf node, and zero otherwise. Note that we simulate the "good" dual knowledge by allowing leaf nodes at each level of the search tree (not only at the maximum

[5] The interested reader can download AntSim at http://www.idsia.ch/∼monaldo/sw/antsim.zip.

level). Each leaf node of a level $l < d$ actually corresponds to a "black node" in the example shown in Figure 1, having stopping probability 1.

For each input tree we computed the probability of reaching the target node, namely $Pr[v_d]$; the inverse of this value is reported in the first column of Table 1 (see Appendix A). Observe that $1/Pr[v_d]$ is equal to the expected number of iterations of MS-PTS(S^1), where $S^1 = (1, 1, 1, \ldots)$), and therefore it represents a lower bound on the expected number of extensions of partial solutions of MS-PTS(S^1). We also computed, for each input tree, the knowledge ratio K_{v_d}. Let us consider for a moment the case in which we set $\delta = \lceil \frac{1}{K_{v_d}} \ln(d+1) \rceil$. Note that the expected number of extensions of partial solutions performed by PTS($\alpha = \delta$) before the target node is reached is at least the number of extensions of partial solutions performed to reach level $d-1$, plus one further partial solution extension, namely at least

$$LB_\delta = \left\lceil \frac{1}{K_{v_d}} \ln(d+1) \right\rceil \cdot (d-1) + 1 \ .$$

In Table 1 (column headed by MS-PTS(S^u)) we report the average number of partial solution extensions, out of 5 runs, required by MS-PTS(S^u) to reach the target node. In our computational experiments, the average number of partial solution extensions required by MS-PTS(S^1) was always larger than $1/Pr[v_d]$, and we decided therefore to omit the exact values. The advantage of MS-PTS(S^u) over MS-PTS(S^1) is evident already by using the values of $1/Pr[v_d]$ for comparison. Moreover, the reader can easily note that the number of partial solution extensions required by MS-PTS(S^u) is at most only few times LB_δ. Therefore, without the knowledge of the exact value of K_{v_d} (as required by Theorem 1), and by using the proposed universal strategy of Section 4.2, it is still possible to obtain the performances claimed by Theorem 1. We can conclude that the presented experimental results for artificial trees are in accordance with our theoretical analysis.

5.2 Application to open shop scheduling

Apart from the application to artificial trees, we also studied the behaviour of MS-PTS($\cdot$) when applied to a combinatorial optimization problem, namely the open shop scheduling problem (OSS).[6] The OSS problem can be formalized as follows. Given is a finite set of operations $O = \{o_1, ..., o_n\}$ which is partitioned into disjoint subsets $M = \{M_1, \ldots, M_{|M|}\}$. The operations in $M_i \in M$ have to be processed on the same machine. For the sake of simplicity we identify each set $M_i \in M$ of operations with the machine they have to be processed on, and call M_i a machine. Set O is additionally partitioned into disjoint subsets $J = \{J_1, \ldots, J_{|J|}\}$, where the set of operations $J_j \in J$ is called a job. Moreover, each operation $o \in O$ has a fixed processing time $p(o)$. We consider the case in which each machine can process at most one operation at a time. Operations must be processed without preemption (that is, once the processing of an operation has started it must be completed without interruption). Operations belonging to the same job must be processed sequentially.

A solution is given by permutations π^{M_i} of the operations in M_i, $\forall i \in \{1, ..., |M|\}$, and permutations π^{J_j} of the operations in J_j, $\forall j \in \{1, ..., |J|\}$. These permutations define processing orders on all the subsets M_i and J_j. Note that a permutation π of all the operations

[6] Our choice of the OSS problem was motivated by the fact that one of the existing algorithms that are similar in spirit to MS-PTS($\cdot$) has obtained good results for this problem; see [1].

represents a solution to an OSS instance. This is because a permutation of all operations naturally defines permutations of the operations of each job and of each machine. There are several possibilities to measure the cost of a solution. Here we deal with makespan minimization. Each operation $o \in O$ has a well-defined *earliest starting time* $t_{es}(o)$ with respect to a (partial) solution. Note that all the operations that do not have any predecessor have an earliest starting time of 0. Accordingly, the *earliest completion time* of an operation $o \in O$ with respect to a (partial) solution is denoted by $t_{ec}(o, s)$ and defined as $t_{es}(o, s)+p(o)$ (where $p(o)$ is the processing time of o). The objective function value $f(\pi)$ of a solution π (also called the makespan) is given by the maximum of the earliest completion times of all the operations:

$$f(\pi) \leftarrow \max\{t_{ec}(o) \mid o \in O\} \tag{7}$$

We aim at minimizing f.

In order to define algorithm PTS(α) for the OSS problem we first have to outline the solution construction mechanism that defines the search tree. Solutions are constructed by filling the n positions of a permutation π one after the other (from left to right). Given a (partial) permutation π_{i-1}—with the first $i-1$ positions already filled—we henceforth denote by O^+ the set of operations that are not yet part of π_{i-1}. The solution construction starts with an empty permutation. At each construction step i ($i = 1, \ldots, n$) is added one of the operations from O^+ at position i of the partial permutation π_{i-1}. Generally, good solutions are obtained by choosing at each construction step an operation $o \in O^+$ that has a low earliest starting time $t_{es}(o)$. This is the motivation for restricting set O^+ at each construction step as follows. Let $t_{\min} = \min\{t_{es}(o) \mid o \in O^+\}$ and $t_{\max} = \max\{t_{es}(o) \mid o \in O^+\}$. Then

$$\overline{O^+} = \{o \in O^+ \mid t_{es}(o) \leq t_{\min} + \beta(t_{\max} - t_{\min})\} \quad \text{with} \quad \beta \in [0, 1] \tag{8}$$

Note that the setting of β determines the size of the resulting search tree. Generally, if β is too small, the optimal solutions might be excluded from the resulting search tree. The setting of β—depending on the size of a problem instance, as explained later—is such that this does not happen.

Next we define the transition probabilities, that is, for each node v and each child w of v we define the probability $Pr[w|v]$. In the case of the OSS problem, a node v is a partial permutation π_{i-1}, and a child of π_{i-1} is obtained by putting one of the operations $o \in \overline{O^+}$ at position i of π_{i-1}. Therefore, we henceforth denote the probabilities $Pr[w|v]$ by $Pr[o|\pi_{i-1}]$. Moreover, a child that is obtained by placing operation $o \in \overline{O^+}$ at position i of π_{i-1} is characterized by the earliest starting time $t_{es}(o)$ of o. Given all children of a partial permutation π_{i-1}, we order them with respect to these earliest starting times (in increasing order). Henceforth we call the position of a child in this order the *rank of the child*, denoted by $r(o \mid \pi_{i-1})$. For example, if $o \in \overline{O^+}$ has the smallest earliest starting time among all operations in $\overline{O^+}$, then $r(o \mid \pi_{i-1}) = 1$. With these rank values we finally define the conditional probabilities for extending the partial permutation π_{i-1} by placing $o \in O^+$ at position i:

$$Pr[o \mid \pi_{i-1}] = \frac{\left(\sum_{j=1}^{i-1} (r(\pi(j) \mid \pi_{j-1}))^{-1}\right) + (r(o \mid \pi_{i-1}))^{-1}}{\sum_{l \in O^+} \left(\left(\sum_{j=1}^{i-1} (r(\pi(j) \mid \pi_{j-1}))^{-1}\right) + (r(l \mid \pi_{i-1}))^{-1}\right)} \tag{9}$$

The motivation for this definition is as follows. A popular greedy heuristic for the OSS problem consists in choosing at each construction step the child that has rank 1, that is, the child obtained by choosing the operation $o \in \overline{O^+}$ with the smallest earliest starting time among all the operations in $\overline{O^+}$. The setting of the transition probabilities as outlined above is such that the solution obtained by the greedy heuristic has the highest probability to be constructed. In general, the closer a solution is to the solution obtained by the greedy heuristic, the higher is its probability to be constructed.[7] The only exception to this setting of the transition probabilities occurs when the lower bound value $\mathrm{LB}(\pi_{i-1})$ of a given partial solution π_{i-1} is greater than the maximum value of any satisfyable solution. In our experiments we were only interested in finding optimal solutions. Therefore, the set of satisfyable solutions is the set of optimal solutions. In this case the additional virtual child of π_{i-1} (the virtual stopping node) has probability 1, and all the regular children have probability 0.

In the following we introduce the lower bound $\mathrm{LB}(\cdot)$ that we used. We denote the operation of a job $J_j \in J$ that was taken last into the partial solution π_{i-1} by o^{J_j}. Similarly, we denote the operation of a machine $M_i \in M$ that was taken last into the partial solution π_{i-1} by o^{M_i}. Observe that the partition of the set of operations O with respect to a partial solution π_{i-1} into O^- (the operations that are already in π_{i-1}) and O^+ (the operations that still have to be dealt with) induces a partition of the operations of every job $J_j \in J$ into ${J_j}^-$ and ${J_j}^+$ and of every machine $M_i \in M$ into ${M_i}^-$ and ${M_i}^+$. Given these notations and a partial permutation π_{i-1}, the lower bound $\mathrm{LB}(\pi_{i-1})$ is computed as follows:

$$\mathrm{LB}(\pi_{i-1}) \leftarrow \max \left\{ \max_{J_j \in J} \left\{ t_{ec}(o^{J_j}) + \sum_{o \in {J_j}^+} p(o) \right\}, \max_{M_i \in M} \left\{ t_{ec}(o^{M_i}) + \sum_{o \in {M_i}^+} p(o) \right\} \right\} \quad , \quad (10)$$

In other words, lower bound $\mathrm{LB}(\cdot)$ is computed by summing for every job and machine the processing times of the unscheduled operations, adding the earliest completion time of the operation of the respective job or machine that was scheduled last, and taking the maximum of all these numbers. As all the necessary numbers can be obtained and updated during the solution construction process, this lower bound can be very efficiently computed. With this information all elements of $\mathrm{PTS}(\cdot)$ are well defined.

5.2.1 PTS$^+(\alpha)$: A variation of PTS(α)

At each solution construction step i of algorithm $\mathrm{PTS}(\alpha)$, α elements are chosen probabilisitically from $\mathcal{C}(R_{i-1})$. This choice is realized as a *choice with replacement*. This means that any chosen element may be selected more than once, which is inefficient from an algorthmic point of view. Consider the tree in Figure 2: each time a node w has two children, assume that the transition probabilities are 0.5 respectively. The stopping probabilities are equal to one at any leaf node, and zero otherwise. Let the successful node v_d be the unique node at the maximum level d. It can be easily checked that $\mathrm{PTS}(\alpha)$ requires a setting of $\alpha = O(2^d)$ to have a high success probability, although the number of visited nodes is clearly bounded by a polynomial in d.

In fact, it is enough to reach any partial solution once. Therefore, we propose a version of $\mathrm{PTS}(\alpha)$—denoted by $\mathrm{PTS}^+(\alpha)$—which implements each construction step as a *choice without replacement*. This simple and natural *diversification* mechanism can increase the effectiveness

[7] Note that this idea is similar to the one that led to the development of limited discrepancy search (LDS) [7].

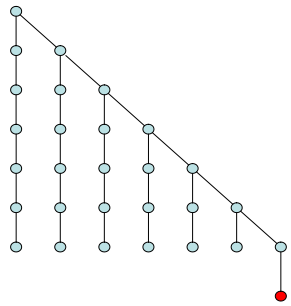


Figure 2: An example tree for which PTS(α) does not work very well.

and avoid some of the possible shortages of PTS(α), as for example the one represented by Figure 2. Moreover, the reader can easily check that it inherits the "good" properties of PTS(α).[8] The analysis of PTS$^+$(α) is suggested for future research. However, we are going to apply its multi-start version MS-PTS$^+$($\cdot$) to the OSS problem.

5.2.2 Computational results

We chose the 60 instances provided by Taillard in [15] as test problems for the experimental evaluation. This set comprises 6 subsets of instances, each one consisting of 10 instances of the same size: $|O| \in \{16, 25, 49, 100, 225, 400\}$. Moreover, we chose the following strategies for our algorithms:

S^i: With this strategy PTS(α) is always executed with the setting $\alpha = i$, that is, $S^i = (i, i, i, \ldots)$. We used 6 different settings of i: $i \in \{1, 2, 5, 10, 50, 100\}$, resulting in 6 different strategies.

S^u: The universal strategy as outlined in Section 4.2. For practical reasons we bounded the setting of α to $2^8 = 256$ from above. In other words, whenever $\alpha > 256$ with respect to the universal strategy, we set $\alpha = 256$.

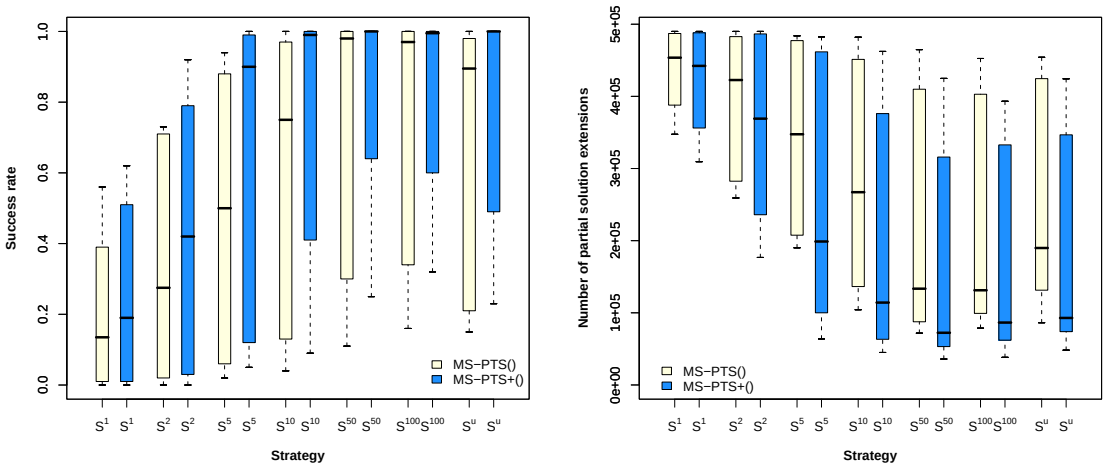
We applied MS-PTS($\cdot$) as well as MS-PTS$^+$($\cdot$) to each of the 60 problem instances 100 times. For each application we allowed a maximum of $|O| \cdot 10000$ extensions of partial solutions, which results in a maximum of 10000 solution constructions. We were only interested in finding optimal solutions.[9] The performance of the algorithms is measured by two different values:

1. The success rate. For example, an algorithm that succeeds in each of its 100 applications to a problem instance A, has a success rate of 1.0 for problem instance A.

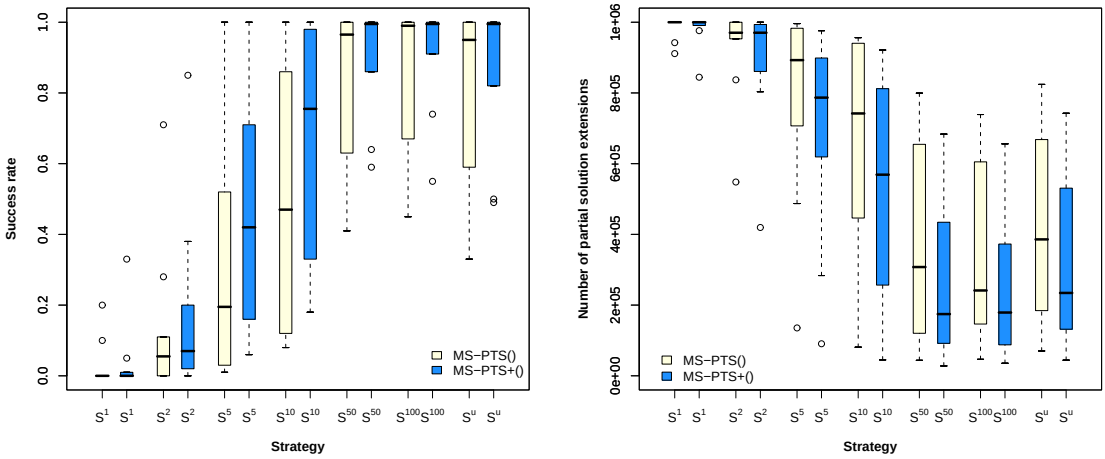
2. The average number of extensions of partial solutions needed to achieve success (or to terminate without success). In the following we will refer to this measure as the running time of an algorithm.

[8] Note that sampling without replacement can be (inefficiently) simulated by sampling with replacement (by disregarding already selected nodes).

[9] Note that the optimal solution value is known for each of the 60 problem instances (see, for example, [1]).



(a) Results for the 10 instances with $|O| = 49$.

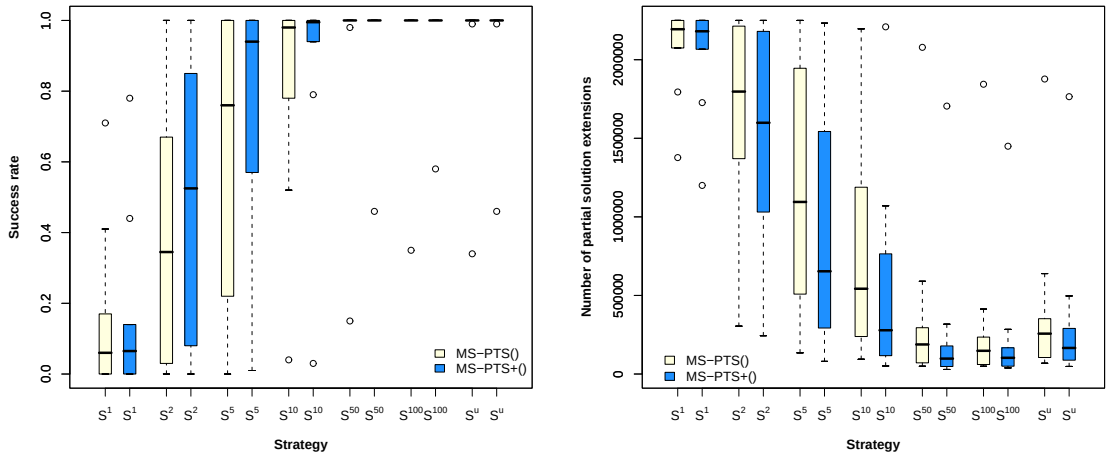


(b) Results for the 10 instances with $|O| = 100$.

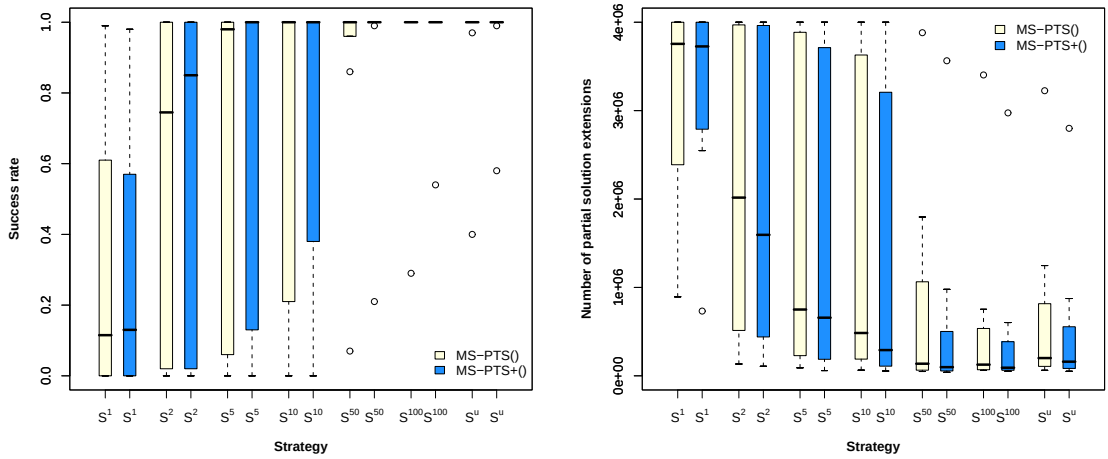
Figure 3: Results of MS-PTS$(\cdot)$ and MS-PTS$^+(\cdot)$. The graphics on the left hand side show results concerning the success rate of the algorithms, and the graphics on the right hand side show results concerning the running time.

We present only the results concerning the 4 bigger subsets of problem instances. They are sown in Figures 3 and 4.[10] For each subset of instances we have 2 graphics. The first one shows results concerning the success rate of the algorithms, and the second one concering the running times. All the results are shown by means of boxplots that visualize the distribution of the results over the 10 instances of each instance subset. The boxes are drawn between the first and the third quartile of the distribution, and the median is shown as a horizontal line in each box. The whiskers extend to data points that are no more than 1.5 times the interquar-

[10]The smaller instances can be solved very easily. Therefore, the differences between the tested strategies were not very clear. Concerning the setting of parameter β (see Equation 8): we chose $\beta = 0.2$ for the instances with $|O| \in \{49, 100\}$, respectively $\beta = 0.15$ for the instances with $|O| \in \{225, 400\}$.



(a) Results for the 10 instances with $|O| = 225$.



(b) Results for the 10 instances with $|O| = 400$.

Figure 4: Results of MS-PTS$(\cdot)$ and MS-PTS$^+(\cdot)$. The graphics on the left hand side show results concerning the success rate of the algorithms, and the graphics on the right hand side show results concerning the running time.

tile range away from the box. Outliers are shown as points. For example, the application of algorithm MS-PTS(S^1) 100 times to each problem instance of size $|O| = 49$ resulted in an average success rate and an average running time for each problem instance. As there are 10 instances of size $|O| = 49$ we have 10 average results concerning the success rate as well as the running times. The left most box of the left-hand-side graphic of Figure 3(a) shows the distribution of the 10 average success rates obtained by algorithm MS-PTS(S^1), whereas the left most box of the right-hand-side graphic of Figure 3(a) shows the distribution of the 10 average running times obtained by algorithm MS-PTS(S^1).

The results allow us to make the following observations:

1. All the strategies S^i with $i > 1$ result in a higher success rate achieved in less running time as compared to strategy S^1. Remember that S^1 is the only strategy that does not exploit the dual problem knowledge. Observe that even strategy S^2 allows in general to double the success rate while significantly reducing the running time as compared to strategy S^1.

2. The best strategies are S^{50} and S^{100}. However, observe that the universal strategy S^u is in general not much worse. This confirms the theoretical results regarding the universal strategy.

3. When comparing the results of algorithm MS-PTS$(\cdot)$ with the results of the extended version MS-PTS$^+(\cdot)$, we can note that MS-PTS$^+(\cdot)$ always results in a higher success rate and at the same time in a lower running time when compared to MS-PTS$(\cdot)$. This confirms that "choosing without replacement" is a better method for choosing among the possible extensions of partial solutions at each construction step.

Let us remark at this point that MS-PTS$(\cdot)$, respectively MS-PTS$^+(\cdot)$, is only the third algorithm in the history of the OSS problem that can solve all 60 problem instances to optimality. The first algorithm was the Beam-ACO approach proposed in [1], and the second one was a complete technique based on constraint programming proposed in [16]. Note that we do not compare to these algorithms in terms of computation time, because the aim of our experimental evaluation was to show the usefulness of employing parallel and non-independent solution constructions.

6 Conclusions and future work

In this paper we have dealt with probabilistic constructive optimization algorithms. In particular, we have studied—theoretically and experimentally—the potential of algorithms that exploit dual problem knowledge by means of parallel and non-independent probabilistic solution constructions. The results have shown that exponential speed-ups may be achieved by algorithms such as MS-PTS$(\cdot)$, which exploit the dual problem knowledge.

However, the success of MS-PTS$(\cdot)$ depends very much on the quality of the dual problem knowledge. The algorithm does not work, for example, when the lower bound used for defining the dual problem knowledge is not tight enough. We repeated the experiments for the OSS problem with lower bound values multiplied by a constant $q < 1$, which results in lower bound values that are less tight. The results showed that with decreasing q the differences between the various strategies disapeared, and the performance of the algorithms using strategies other than S^1 was becoming closer and closer to the performance of the algorithms using strategy S^1.

In the future, we plan to extend the theoretical analysis as well as the experimental work to more efficient versions of MS-PTS$(\cdot)$. For example, the dependence on a tight lower bound may be reduced by allowing more than α extensions of partial solution at each step, with the subsequent restriction of the chosen child nodes to a subset containing the α best ones with respect to the lower bound values.

Acknowledgments. We would like to thank Felipe Figueiredo and Michele Pedrazzi for the implementation of the Java tool. Monaldo Mastrolilli acknowledges support from the Swiss

National Science Foundation project 200021-104017/1, "Power Aware Computing", and by the Swiss National Science Foundation project 200021-100539/1, "Approximation Algorithms for Machine scheduling Through Theory and Experiments". Christian Blum acknowledges support from the Spanish CICYT project OPLINK (grant TIN-2005-08818-C04-01), and from the *Ramón y Cajal* program of the Spanish Ministry of Science and Technology of which he is a research fellow.

References

[1] C. Blum. Beam-ACO–hybridizing ant colony optimization with beam search: an application to open shop scheduling. *Computers and Operations Research*, 32:1565–1591, 2005.

[2] C. Blum, J. Bautista, and J. Pereira. Beam-ACO applied to assembly line balancing. In M. Dorigo, L. M. Gambardella, A. Martinoli, R. Poli, and T. Stützle, editors, *Proceedings of ANTS 2006 – Fifth International Workshop on Swarm Intelligence and Ant Algorithms*, volume 2463 of *Lecture Notes in Computer Science*, pages 14–27. Springer Verlag, Berlin, Germany, 2006.

[3] C. Blum, C. Cotta, A. Fernández, and F. Gallardo. A probabilistic beam search approach to the shortest common supersequence problem. Technical Report LSI-06-36-R, Dept. Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, 2006. Submitted to EvoCOP 2007.

[4] M. Dorigo and T. Stuetzle. *Ant Colony Optimization*. MIT Press, 2004.

[5] T. A. Feo and M. G. C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[6] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

[7] W. D. Harvey and M. L. Ginsberg. Limited discrepancy search. In C. S. Mellish, editor, *Proceedings of the 14th International Joint Conference on Artificial Intelligence, IJCAI 1995*, volume 1, pages 607–615, Montréal, Québec, Canada, 1995. Morgan Kaufmann, 1995.

[8] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[9] E. L. Lawler and D. E. Wood. Branch-and-bound methods: A survey. *Operations Research*, 14(4):699–719, 1966.

[10] M. Luby, A. Sinclair, and Z. Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47(4):173–180, Sept. 1993.

[11] V. Maniezzo. Exact and Approximate Nondeterministic Tree-Search Procedures for the Quadratic Assignment Problem. *INFORMS Journal on Computing*, 11(4):358–369, 1999.

[12] V. Maniezzo and A. Carbonaro. An ANTS heuristic for the frequency assignment problem. *Future Generation Computer Systems*, 16:927–935, 2000.

[13] V. Maniezzo and M. Milandri. An ant-based framework for very strongly constrained problems. In *Proceedings of ANTS 2002: 3rd International Workshop on Ant Algorithms*, volume LNCS 2463, pages 222–227, 2002.

[14] P. S. Ow and T. E. Morton. Filtered beam search in scheduling. *International Journal of Production Research*, 26:297–307, 1988.

[15] É. D. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operations Research*, 64:278–285, 1993.

[16] N. Tamura, A. Taga, S. Kitagawa, and M. Banbara. Compiling finite linear CSP into SAT. In F. Benhamou, editor, *Proceedings of CP 2006 – 12th International Conference on Principles and Practice of Constraint Programming*, volume 4204 of *Lecture Notes in Computer Science*, pages 590–603. Springer Verlag, Berlin, Germany, 2006.

Appendix A

Table 1: Average number of construction steps out of 5 runs required by MS-PTS(S^u) in order to reach the target node.

$Tree$	(c, d, K_{min})	$1/Pr[v_d]$	LB_δ	MS-PTS(S^u)
1	$(4, 20, 0.1)$	5971968.01	590	996.8
2	$(4, 20, 0.1)$	442368.00	590	1214.2
3	$(4, 20, 0.1)$	1327103.99	590	1396.8
4	$(4, 20, 0.1)$	2985984.01	590	1483
5	$(4, 20, 0.1)$	5308415.96	590	1647.6
6	$(4, 20, 0.1)$	995328.03	590	1089.2
7	$(4, 20, 0.1)$	746496.00	590	1395.8
8	$(4, 20, 0.1)$	60466175.88	590	1656.4
9	$(4, 20, 0.1)$	7962624.24	590	1530.4
10	$(4, 20, 0.1)$	21233663.84	590	1659.8
11	$(4, 20, 0.2)$	331776.00	305	794.6
12	$(4, 20, 0.2)$	1990656.06	305	738.6
13	$(4, 20, 0.2)$	7077887.95	305	833.6
14	$(4, 20, 0.2)$	7962624.24	305	851.2
15	$(4, 20, 0.2)$	186624.00	305	552.6
16	$(4, 20, 0.2)$	3981312.12	305	623.4
17	$(4, 20, 0.2)$	1492992.00	305	603
18	$(4, 20, 0.2)$	15925248.48	305	564.4
19	$(4, 20, 0.2)$	7077887.95	305	627.8
20	$(4, 20, 0.2)$	1679616.06	305	832
21	$(3, 30, 0.1)$	11943936.02	1016	1858.6
22	$(3, 30, 0.1)$	26873856.91	1016	2375.6
23	$(3, 30, 0.1)$	725594091.07	1016	1535.2
24	$(3, 30, 0.1)$	5038848.17	1016	2167
25	$(3, 30, 0.1)$	241864703.52	1016	2145.2
26	$(3, 30, 0.1)$	120932351.76	1016	1993.8
27	$(3, 30, 0.1)$	181398522.77	1016	1298.2
28	$(3, 30, 0.1)$	272097789.63	1016	4281
29	$(3, 30, 0.1)$	2754990367.07	1016	2386.8
30	$(3, 30, 0.1)$	181398522.77	1016	2789
31	$(2, 40, 0.01)$	4194304.00	5227	6160.8
32	$(2, 40, 0.01)$	131072.00	1483	3029.4
33	$(2, 40, 0.01)$	524288.00	1756	3582.8
34	$(2, 40, 0.01)$	1048576.00	4213	13629.8
35	$(2, 40, 0.01)$	1048576.00	2614	6579.8
36	$(2, 40, 0.01)$	524288.00	3511	5711
37	$(2, 40, 0.01)$	33554432.00	13963	47488.2
38	$(2, 40, 0.01)$	262144.00	2692	4075.4
39	$(2, 40, 0.01)$	2097152.00	1756	3163
40	$(2, 40, 0.01)$	16384.00	1171	868