# Embedding Balanced Binary Trees in the Hypercube


Markus Aderhold
University of British Columbia
Department of Computer Science
aderhold@cs.ubc.ca

James Slack
University of British Columbia
Department of Computer Science
jslack@cs.ubc.ca



**Abstract**

In the context of parallel computing, the problem of embedding binary trees that represent communication structures arises. Whereas much research has focused on arbitrary trees, we concentrate on the subclass of balanced binary trees. We investigate various stochastic local search approaches, aiming to find perfect embeddings quickly. The performance of the resulting algorithms is empirically assessed on trees with different structural properties. In particular, we show that all strongly balanced binary trees with 32 nodes can be embedded without dilation into the corresponding hypercube.


## 1 Introduction

Hypercube multiprocessor systems are a very prominent type of parallel machines, because their recursive structure and the fact that they contain structures like rings, 2-D-meshes, and higher dimensional meshes make them suitable for a large variety of problems. Many parallel algorithms use communication structures which can be represented by binary trees. In order to run these algorithms on a hypercube multiprocessor system, their communication graphs need to be embedded in the corresponding hypercube.

A particular embedding of a tree with $n$ nodes in a hypercube can be represented by a permutation of the $n$ nodes (we will give a formal definition of an embedding in Section 2). Even for moderate parallel systems (e.g., hypercubes with up to $n = 64$ nodes) the search space of candidate embeddings is quite big; for $n$ nodes, there are $(n-1)!$ candidate permutations for embeddings (we already ignore embeddings that differ only in the embedding of the root node but are otherwise "isomorphic"). Therefore, a reasonably fast embedding algorithm is needed.

Wagner [7] presented a heuristic algorithm to realize an embedding of a binary tree in two steps: First, the binary tree is mapped into a strongly balanced binary tree, where some edges may be dilated. Then, this strongly balanced binary tree is embedded into the optimal-sized hypercube by a "folding" algorithm. Wagner's algorithm relies on a conjecture about the "foldability" of strongly balanced binary trees into (smaller) strongly balanced binary trees; a tree with just one node can be trivially embedded into the zero-dimensional hypercube and the embedding of the

original tree can be obtained by unfolding this trivial tree again. If this conjecture about strongly balanced binary trees is true, the proposed folding yields a special way of embedding a strongly balanced binary tree in a hypercube without dilating any edges in this step. Wagner's conjecture can be seen as a special case of a conjecture by Havel and Morávek [2]:

**Conjecture (Havel and Morávek).** *Every balanced binary tree with $2^d$ vertices is a spanning tree of $H_d$, the d-dimensional hypercube.*

Wagner's heuristic algorithm for binary trees has a polynomial time complexity. In general, embedding trees in the hypercube is NP-complete [8], but the complexity of embedding *binary* trees is unknown; the proof of the NP-completeness given in [8] uses "bushy" trees and therefore cannot be applied to the embedding of binary trees.

Smedley [6] analyzed and compared various embedding methods, including stochastic local search algorithms such as Simulated Annealing and a Greedy Approach. Simulated Annealing was found to have the worst performance among the algorithms under consideration, whereas the constructive greedy approach, combined with a subsequent local search phase, showed the best performance with respect to solution quality; the quality of an embedding is measured by its total (or average) dilation. Whereas Smedley investigated how to embed *arbitrary* binary trees in hypercubes, in this paper we focus on embedding *balanced* binary trees and in particular look at the associated decision problem whether there exists an embedding of (maximal) dilation 1. The Simulated Annealing algorithm accepts a swap of the embedding of two nodes based on the Metropolis acceptance criterion, which takes into account the current temperature and the increase in total dilation caused by the swap. The Greedy Approach by Smedley begins with an empty embedding and successively embeds single nodes that at each step have a maximum "gain". The gain is influenced both by the dilation of the edge under consideration and by the number of embedded neighbours of a given node.

Another approach by Helmer and Eisenberg [3] is based on the idea of a generic complete search algorithm that involves backtracking but also uses some randomization. They found that an algorithm that uses a depth first ordering of the tree nodes usually got trapped and had to backtrack very frequently, so its overall performance was quite poor. A breadth first embedding strategy, combined with a hypercube-dimension based heuristic and what they called "intelligent backtracking" proved to be much more suitable. They also identified classes of "easy" and "difficult" trees for the embedding problem, which we used for the evaluation of our embedding algorithms.

In this paper, we focus on devising algorithms for embedding balanced binary trees into the optimal-sized hypercube (which according to Havel's conjecture should always succeed) as fast as possible. We adopt the two-exchange neighbourhood for our Iterative Best Improvement algorithm that Smedley used for the Simulated Annealing algorithm. We improve the performance by proposing a new evaluation function. Our greedy construction process is completely different and uses a breadth first ordering of the tree nodes. Furthermore, we explore the use of a tabu mechanism and a population-based approach. In passing, we prove a special case of Havel's conjecture, namely that all strongly balanced binary trees with 32 nodes can be embedded into the corresponding hypercube.

The paper is organized as follows: In Section 2, we introduce the relevant terminology for the embedding problem. The Randomized Breadth First approach by Helmer and Eisenberg [3] is given

in Section 3. Section 4 explains how to transform the embedding problem into a propositional satisfiability (SAT) or constraint satisfaction problem (CSP). An algorithm to solve the resulting CSP encodings is given. We develop and discuss two stochastic local search approaches in Section 5. Finally in Section 6, we evaluate and compare our algorithms on many different balanced binary trees with up to 64 nodes, where we especially focus on the embedding of "hard" trees.

## 2 Definitions

An *embedding* of a binary tree $T = (V, E)$, $|V| = 2^d$ for some $d \in \mathbb{N}$, in the corresponding hypercube[1] $H_d$ is an injective function $\varphi : V \rightarrowtail \{0, \ldots, 2^d - 1\}$ such that $hd(\varphi(u), \varphi(v)) = 1$ for all $(u, v) \in E$, where $hd : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ denotes the Hamming distance between two numbers in binary representation. (Note that this implies that $\varphi$ is also surjective.)

A *perfect matching* of $T$ is a subset $E_m \subseteq E$ of its edges such that each node $v \in V$ is the endpoint of exactly one edge in the matching. $T$ is called *strongly balanced* if it has a perfect matching. We say that a binary tree $T$ is *(colour-)balanced* if there exists a bipartition of the vertices into two equal-sized sets such that no edge lies entirely within one of the parts. This means that there exists a two-colouring which uses the two colours equally.

Not all binary trees can be embedded in the corresponding hypercube. But if such a tree represents the communication structure of a parallel program, this does not mean that this program cannot be executed on a hypercube multiprocessor system. The program can still run on the system if some edges (i.e., communication paths) are dilated. Especially for local search algorithms, we will consider such candidate embeddings as well. These embeddings differ from the (perfect) embeddings introduced above in that they violate the Hamming distance constraints. Formally, an edge $(u, v) \in E$ is *dilated* by a candidate embedding $\varphi$ if $hd(\varphi(u), \varphi(v)) > 1$. The *dilation* of a candidate embedding $\varphi$ is the maximum dilation over all edges:

$$dilation(\varphi) := \max_{(u,v) \in E} hd(\varphi(u), \varphi(v))$$

Consequently, a candidate embedding $\varphi$ is an embedding if $dilation(\varphi) = 1$. Finally, note that as soon as there exists *one* embedding of a tree in the corresponding hypercube, there are always *many* other (perfect) embeddings in addition: We can map the root of the tree to any node of the hypercube and adjust the remainder of the embedding accordingly. Furthermore, we can use the hypercube's symmetry and rotate the embedding arbitrarily.

## 3 Randomized Breadth First Embedding

Following from Helmer and Eisenberg's results [3], we implemented a breadth first embedding algorithm as a baseline for the empirical evaluation of our new stochastic methods. The breadth first algorithm is simply a randomized greedy construction algorithm that embeds the root node

[1] Throughout this paper we will use the terminology "corresponding hypercube" to denote a hypercube with exactly the same number of nodes as the tree under consideration.

**input:** binary tree $T = (V, E)$, $V = \{v_0, \ldots, v_{n-1}\}$
**output:** embedding $\varphi$ of $T$ in the corresponding hypercube

$d \leftarrow log_2(n)$
$\varphi \leftarrow \emptyset$
$worklist \leftarrow breadthFirstTraverse(root)$
$currentNode \leftarrow worklist[0]$
$currentNode.hypercubeNumber \leftarrow 0$
$currentNode \leftarrow currentNode.nextWork$
while $currentNode \neq \emptyset$ do
  if $\exists$ a random $dim \in [0, d-1]$ such that
      $(\varphi(currentNode.Parent)$ xor $2^{dim})$ has not been assigned to some $v_i$ then
    // Assign tree node to hypercube node
    $\varphi(currentNode) \leftarrow \varphi(currentNode.Parent)$ xor $2^{dim}$
    $currentNode \leftarrow currentNode.nextWork$
  else
    // Backtrack
    $\varphi(currentNode) \leftarrow \emptyset$
    $currentNode \leftarrow currentNode.prevWork$

Figure 1: Randomized Breadth First Algorithm

and traverses the binary tree by following the breadth of the tree. Nodes are embedded in a first-fit heuristic with backtracking and no global evaluation function.

Helmer and Eisenberg argued that depth first heuristics for selecting the next candidate node were inferior since the heuristic delayed the complete embedding of degree 3 nodes and their children, which they claim constrained the embedding. Their explanation seemed to be purely from empirical evidence of the trees that they could embed with their algorithm. Their algorithm failed to embed trees with mostly nodes of degree 3 but was very successful in trees that were long and stringy. To compare their algorithm with our stochastic algorithms, we implemented their pseudo-code as a single program as accurately as possible. Figure 1 gives the pseudo-code for this algorithm.

Simple evaluation of run times with our implementation of this algorithm reinforces the claim by Helmer and Eisenberg that backtracking methods for this problem perform poorly. Due to our poor run time results during a simple empirical evaluation, please refer to [3] for a complete evaluation of the families of trees that were successfully embedded. It is worth noting from this paper that this algorithm performs poorly even under attempts to use multiple independent processors working with balanced subsets of the problem domain. Furthermore, Helmer and Eisenberg limited the number of backtracks and used a small number of complete restarts after reaching their backtrack limit to obtain most of the successful embeddings. We do not provide run time distributions since the algorithm was unable to complete embeddings for colour balanced trees with many nodes of degree 3.

# 4 Encoding as SAT and CSP Problems

The embedding problem can be expressed in various encodings. Several different well studied and optimized methods can be used when problems are encoded in a specific logic format. One of our original goals was to encode our problem as a satisfiability (SAT) problem and use an existing SAT solver. Other encodings were also explored when it was determined that our SAT encoding of the embedding problem has an exponential number of clauses. The most promising encoding, a variant of the finite discrete constraint satisfaction problem (CSP), was then analyzed empirically.

## 4.1 Encoding as a SAT Problem

An alternative approach to embedding a hypercube is encoding the embedding problem as a satisfiability problem (SAT) and using a generic optimized SAT solver to determine an embedding. The embedding problem can be written in a very simple set of statements:

1. All tree nodes must be mapped to unique hypercube nodes.
2. If two tree nodes are connected by an edge, then the tree nodes must have a hypercube embedding Hamming distance of exactly 1.

Our first attempt to formalize the embedding as SAT produced the set of equations in Equations 1 to 4.

$$F = \left( \bigwedge_{\substack{0 \leq k < m < n \\ k \neq \pi(m) \,\wedge\, m \neq \pi(k)}} \mathit{different}(\vec{e}_k, \vec{e}_m) \right) \wedge \left( \bigwedge_{\substack{0 \leq k, m < n \\ m = \pi(k)}} \bigvee_{i=1}^{d} \mathit{differonly}(\vec{e}_k, \vec{e}_m, i) \right) \tag{1}$$

where

$$\begin{aligned} \vec{e}_k &= e_{k,1} e_{k,2} \ldots e_{k,d} \quad (e_{k,i} \in \{0,1\}) \\ &= \text{embedding of node } k \text{ in } H_d \ (0 \leq k < n), \\ m = \pi(k) &:\Leftrightarrow \text{node } m \text{ is the parent of node } k \end{aligned}$$

$$\mathit{differonly}(\vec{e}_k, \vec{e}_m, i) := (e_{k,i} \text{ xor } e_{m,i}) \wedge \bigwedge_{\substack{1 \leq j \leq d \\ j \neq i}} \overline{(e_{k,j} \text{ xor } e_{m,j})} \tag{2}$$

$$\mathit{different}(\vec{e}_k, \vec{e}_m) := \bigvee_{i=1}^{d} (e_{k,i} \text{ xor } e_{m,i}) \tag{3}$$

$$e_{k,i} \text{ xor } e_{m,i} \; := \; (\overline{e_{k,i}} \vee \overline{e_{m,i}}) \wedge (e_{k,i} \vee e_{m,i}) \tag{4}$$

However, it was not clear if the number of conjunctive normal form (CNF) clauses in the equations, which is required as input to generic SAT solvers, was exponential in the number of nodes, $n$. Therefore, rather than attempt to expand the complex logical operations into CNF, we derived an alternative embedding for all nodes being different. This produced the set of equations in Equations 5 to 7.

$$F \;=\; \left( \bigwedge_{1 \leq l \leq d} \; \bigwedge_{s_i(l) \in S(l)} \left( \bigvee_{s_{i,j}(l) \in s_i(l)} s_{i,j}(l) \wedge \bigvee_{s_{i,j}(l) \in s_i(l)} \overline{s_{i,j}(l)} \right) \right) \wedge \tag{5}$$

$$\left( \bigwedge_{\substack{1 \leq m,k \leq n \\ m = \pi(k)}} \; \bigvee_{p_i \in P} (\vec{e}_m(p_i) \vee \vec{e}_k(p_i)) \right) \wedge \tag{6}$$

$$\bigwedge_{\substack{1 \leq m,k \leq n \\ m = \pi(k)}} \; \bigwedge_{1 \leq i < j \leq d} \begin{matrix} (e_{m,i} \vee e_{m,j} \vee \overline{e_{k,i}} \vee \overline{e_{k,j}}) \wedge \\ (e_{m,i} \vee \overline{e_{m,j}} \vee \overline{e_{k,i}} \vee e_{k,j}) \wedge \\ (\overline{e_{m,i}} \vee e_{m,j} \vee e_{k,i} \vee \overline{e_{k,j}}) \wedge \\ (\overline{e_{m,i}} \vee \overline{e_{m,j}} \vee e_{k,i} \vee e_{k,j}) \end{matrix} \tag{7}$$

where

$$\begin{array}{rcl}
\vec{e}_k & = & e_{k,1} e_{k,2} \ldots e_{k,d} \quad (e_{k,i} \in \{0,1\}) \\
 & = & \text{embedding of node } k \text{ in } H_d \; (0 \leq k < n), \\
m = \pi(k) & :\Leftrightarrow & \text{node } m \text{ is the parent of node } k \\
S & = & \text{all } \binom{n}{n/2+1} \text{ permutations of } n/2+1 \text{ tree nodes} \\
 & & \quad \text{from a tree with } n \text{ nodes} \\
S(l) & = & \text{a column } 1 \leq l \leq d \text{ of S} \\
s_i(l) & = & \text{a single permutation enumerated as } i \text{ from } S \text{ at} \\
 & & \quad \text{dimension } l \\
s_{i,j}(l) & = & \text{a single bit value of node } j \text{ in } s_i(l) \text{ corresponding} \\
 & & \quad \text{to some } e_{m,l} \\
 & \in & \{0,1\} \\
P & = & \text{set of all truth value assignments from 0 to } 2^d - 1 \\
 & & \quad d \text{ bits wide} \\
\vec{e}_m(p_i) & = & \text{application of permutation of truth values } p_i \text{ to } \vec{e}_m \\
 & & \quad \text{with disjunctions} \\
 & e.g.: & \vec{e}_m(1001) :\Leftrightarrow e_{m,1} \vee \overline{e_{m,2}} \vee \overline{e_{m,3}} \vee e_{m,4}
\end{array}$$

These equations work in a similar manner as the previous equations. Equation 5 will only allow $n$ bit positions in all embedded nodes to be set to the value 1 as well as only allowing $n$ bit positions

in all embedded nodes to be set to the value 0. This restriction is quite strong and is very difficult to encode in CNF; it does not take much effort to arrive at a $n^2$ clause disjunctive normal form (DNF) encoding to restrict embeddings in a similar way. The CNF equation does this by requiring all sets of $n/2 + 1$ nodes that there must be at least 1 node with a different bit value than the other n/2 nodes; since the bits are restricted to two values, either half of the equation must hold.

Equations 6 and 7 form the bounds on all adjacent binary tree nodes. First, Equation 6 is used to force adjacent nodes to be different in at least one bit position. It does this by requiring that when an embedding value of $\vec{e}_m$ (between 0 and $n-1$) fails to match in any bit with a certain permutation, it must be true that $\vec{e}_k$, which is identical to $\vec{e}_m$ in the equation must match at least one bit. Then, Equation 7 is used to force adjacent nodes to be different in at most one bit position. This is achieved by looking at any two bits of $\vec{e}_m$ and $\vec{e}_k$ in the same bit position. If any of the parts of any of the four sub-equations in Equation 7 fail, that means that there are at least two bits different in $\vec{e}_m$ and $\vec{e}_n$. Therefore, the two equations work together to force the number of bit differences to be strictly greater than zero and strictly less than two; the adjacent nodes thusly must differ in exactly one bit.

Careful analysis of these equations reveals the complexity of encoding our problem as a SAT instance as being exponential in the number of nodes in our binary tree. This can be observed from Equation 5. The first line of this equation, which effectively constrains the problem such that all tree nodes are embedded in unique locations in the hypercube, requires:

$$\begin{aligned}\text{Number of clauses 5} &= (d \text{ dimensions})(\tbinom{n}{n/2+1} \text{ permutations})(2 \text{ clauses}) \text{ disjunctions} \\ &= \tfrac{dn}{n/2+1}\tbinom{n}{n/2} \text{ disjunctions of length } n/2+1\end{aligned}$$

Since $\binom{n}{n/2}$ grows exponentially with $n$, the number of clauses in our SAT encoding also grows exponentially with $n$. The other two Equations 6, 7 yield a substantially smaller number of clauses:

$$\begin{aligned}\text{Number of clauses 6, 7} &= (n-1 \text{ sets of 2 connected nodes}) \\ &\quad (2^d \text{ permutations of dimensions } + \\ &\quad\ (4 \text{ clauses})(\tbinom{d}{2} \text{ ways to select 2 dimensions})) \text{ disjunctions} \\ &= (n-1)(n+4\tbinom{d}{2}) \text{ disjunctions of length } 2d \text{ and } 4\end{aligned}$$

The number of clauses in these parts of our encoding grows with $n^2$. This information does not help reduce the number of clauses required for the complete equation, and therefore, our encoding has an exponential number of terms. Of course this means that encoding our problem as a SAT problem will take exponential time with respect to generating the clauses; the complexity of the SAT encoding encouraged us to find alternative methods of encoding the problem. The next section describes our constraint satisfaction problem (CSP) encoding.

## 4.2 Encoding as a CSP Problem

Following the disappointing number of clauses in general SAT encodings, we discovered that the problem is fundamentally encodable as a constraint satisfaction problem (CSP). Our SAT encoding

prompted attempts to assign hypercube dimensions to tree edges instead of assigning hypercube nodes to tree nodes. This can be observed from the exponential length of our SAT encoding, which is due to the complexity of ensuring no hypercube node has two binary tree nodes assigned. Our CSP encoding attempts led to an encoding with a single constraint type that we could then employ a CSP based algorithm to solve. Simply put, our CSP algorithm attempts to assign dimensions to edges and succeeds when the assignment does not violate the constraints. This means that for each edge of the tree there is a variable with domain $\{0, \ldots, d-1\}$. However, no analysis has been done to turn a CSP encoded instance into a SAT encoded instance based on the complexity of the CSP constraint expressed as a SAT encoding.

The single constraint to this solution is that the paths described by adjacent edges cannot form cycles. A set of edges in a path forming a cycle is equivalent to embedding two binary tree nodes in the same hypercube node. Our CSP solution effectively works since we can enumerate all paths in a given tree and only have to look at a subset of paths that might contain cycles. The number of paths in a tree with $n$ nodes is $\binom{n}{2}$, but some of those paths are not able to contain a unique cycle.

We know that paths with an even number of edges may form a cycle if each edge is used an even number of times. This is trivial with paths of 2 edges; traveling along dimension $d$ twice will form a cycle for any node and any dimension. By enumerating all binary tree paths with even length, we can search for cycles by maintaining counts of the dimensions being used; paths of all even lengths from 2 to the maximum even length path are necessary for this algorithm to work. Hence, for each such path there is a constraint on the variables denoting edge dimensions so that not all dimensions along this path are used an even number of times.

Empirical evidence suggests that there are exactly $\frac{n}{2}(\frac{n}{2}-1)$ even length paths in colour balanced binary trees with $n$ nodes. This is not true for all trees with $n$ nodes; as a simple counterexample, consider a single node with $n-1$ children. However, this does not affect the algorithm, so it will not be discussed further.

A straightforward way to obtain all even length paths is to use an algorithm for determining All-Pairs-Shortest-Paths (APSP) using a simple $n^3$ algorithm like Floyd's algorithm so the paths can be recovered. In order to use such an algorithm, the instance is converted into an $n$-by-$n$ distance matrix; if two tree nodes $i$ and $j$ are connected in a binary tree, then the matrix entries $(i, j)$ and $(j, i)$ are set to 1, otherwise the entries are set to 0. Since the size of the matrix, $n^2$, is sufficiently small, this step does not add significant overhead to the running time; this step is only intended to be performed once for initializing. The edge indices are stored in lists of paths and each edge has a list of paths of which it belongs. This aids in the update of the evaluation functions as the neighbourhood is explored.

After finding all paths and effectively encoding a CSP instance, an algorithm based on the Min-Conflict Heuristic (MCH) [5] was used to solve the instance and find a solution. The neighbourhood of interest for a current edge embedding can be thought of as a one-exchange neighborhood; the neighbours of an edge assignment are all edge assignments that differ in exactly one dimension-to-edge assignment. Since restarting the algorithm would be costly due to expensive restarts, an algorithm variant that probabilistically uses either the MCH approach or a random walk seemed appropriate.

Starting with the generic MCH algorithm, we developed a variant called WeightedMCH for our

**input:** binary tree $T = (V, E)$, $V = \{v_0, \ldots, v_{n-1}\}$, $0 \leq walkProbability \leq MAXWALK$
**output:** embedding $\varphi$ of $T$ in the corresponding hypercube

```
(distanceMatrix, pathMatrix) ← APSP(T)
φ ← ∅
evenPaths ← GetEvenPaths(distanceMatrix, pathMatrix)
edges ← RandomAssign(n)
unsat ← isSatisfied?(EvaluateEdges(edges))
while unsat do
    with probability walkProbability/MAXWALK
        // Random walk step
        unsat ← isSatisfied?(RandomWalk(edges, evenPaths))
    else
        // Weighted probabilistic step
        unsat ← isSatisfied?(WeightedMCH(edges, evenPaths))
φ ← edgeAssign(edges)
```

Figure 2: CSP based algorithm for embedding problem

CSP based approach. WeightedMCH selects an edge to flip randomly by looking at all edges that can be flipped using dynamic information to bias the decision. Bias is used in the edge decision such that the edge to be flipped is weighted based on the number of unsatisfied paths/constraints the edge is currently in; more unsatisfied constraints results in a higher probability that the edge will be selected next; edges that are only in satisfied constraints will never be selected by WeightedMCH.

WeightedMCH randomly, uniformly chooses a new dimension for a chosen edge. Any new dimension choice for an edge in an unsatisfied path is sufficient; limited empirical evidence suggests this approach is reasonable due to the amount of computational effort required to find the best dimension. If the new edge/dimension in the embedding assignment has at least as few unsatisfied constraints as the previous edge/dimension assignment, then the algorithm will continue with the edge assigned to the new choice in dimension. Otherwise, a backtrack is made to the previous edge/dimension. Backtracks are inexpensive with the data structure decisions in our implementation. Figure 2 is a high-level pseudo-code representation of our CSP based algorithm that uses the modified MCH algorithm.

Deciding on the optimal walk probability is essential to optimizing the performance of our algorithm. The probability is determined by first running the algorithm with a small number of runs with varying walk probabilities. Figure 3 shows a fine grained run-time distribution (RTD) evaluation that ran with 1000 runs on an informed range of walk probabilities. An optimal walk probability was determined for each tree that we used to evaluate our functions with, but empirically, a walk probability of $15/1000$ proved optimal for a small set of trees from several classes that we tested. Figure 4 is a plot of RTDs for these classes and shows the relation between the classes of trees tested with our determined walk probability. See Section 6 for more analysis of our algorithm, including fitting an exponential function and comparisons of the trees we tested versus other trees in their class.

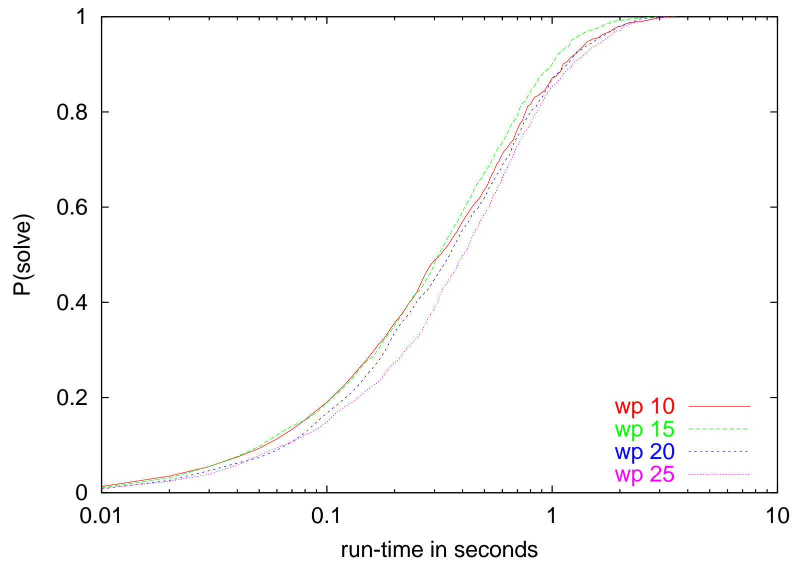


Figure 3: Determining the optimal walk probability for CSP based algorithm, 1000 runs

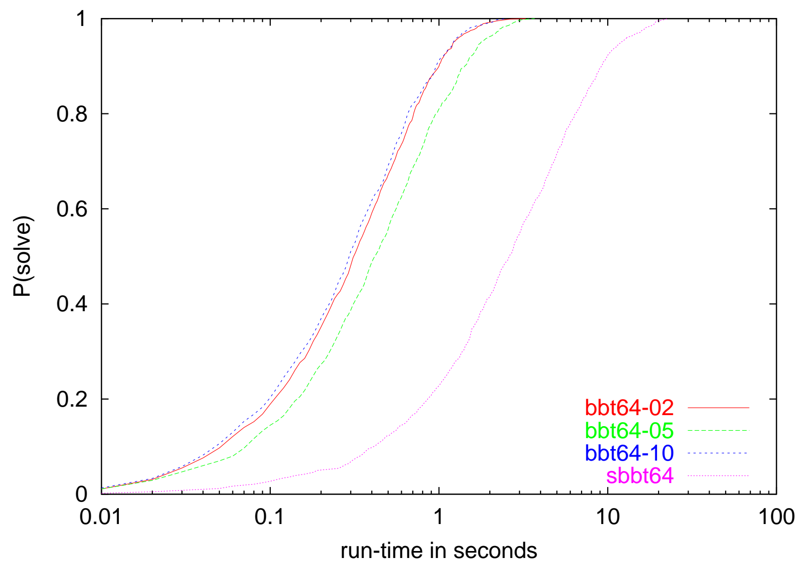


Figure 4: Comparing different classes of trees with CSP based algorithm, walk probability 15/1000 with 1000 runs

# 5 Stochastic Local Search Algorithms

The general idea of local search is to start with a candidate embedding and to improve it "locally" by applying small changes until a valid solution is found. Our local search approach works as follows: The input of the algorithm is a binary tree $T = (V, E)$, where $|V| = 2^d$ for some $d \in \mathbb{N}$. The set $\Phi_i$ of *candidate embeddings* of $T$ in $H_d$ is the set of all injective mappings from $V$ to $\{0, \dots, 2^d - 1\}$, i.e.,

$$\Phi_i := \{\varphi \mid \varphi : V \rightarrowtail \{0, \dots, 2^d - 1\}\}.$$

In order to guide the local search, we need a way to assess the quality of candidate embeddings. The notion of the (maximum) dilation of an embedding from Section 2 is not very differentiating, because it does not take the number of dilated edges into consideration. Therefore, we use the *total dilation* $\Delta$ of a candidate embedding,

$$\Delta \; : \; \Phi_i \to \mathbb{N} \; : \; \varphi \mapsto \sum_{(u,v) \in E} hd(\varphi(u), \varphi(v)).$$

With $\Delta$ as an *objective function*, we now have an optimization (minimization) problem. A very straightforward *evaluation function* $f_e$ turns the total dilation value into values that are easier to interpret, because they indicate the excess dilation of edges:

$$f_e \; : \; \Phi_i \to \mathbb{N} \; : \; \varphi \mapsto \Delta(\varphi) - |E| = \sum_{(u,v) \in E} (hd(\varphi(u), \varphi(v)) - 1)$$

Since each $\varphi \in \Phi_i$ is injective, $hd(\varphi(u), \varphi(v)) \geq 1$ for each edge $(u, v) \in E$, so a candidate embedding is a perfect embedding if and only if $\Delta(\varphi) = |E|$, which is equivalent to $f_e(\varphi) = 0$. Unfortunately, this evaluation function still does not provide much guidance; it does not take into account *which* edges are dilated. If we aim at finding perfect embeddings, however, it is a big difference whether an edge in the middle of the tree is dilated or an edge close to a leaf node. In the first case, after repairing the dilation, the embedding of all subtrees may have to be adjusted, so there is no point of fixing the embedding of leaf nodes beforehand. Therefore, we will also consider (and prefer) an evaluation function that weights the dilation by the size of the corresponding subtree:

$$f_s \; : \; \Phi_i \to \mathbb{N} \; : \; \varphi \mapsto \sum_{(u,v) \in E} (hd(\varphi(u), \varphi(v)) - 1) \cdot size(T_v),$$

where $size(T_v)$ denotes the size of the subtree of $T$ that is rooted at node $v$ (we assume that all edges $(u, v) \in E$ are given such that $v$ is a child node of $u$). This evaluation function preserves the property that all perfect embeddings are characterized by $f_s(\varphi) = 0$, but it especially penalizes edge dilations at higher levels. Besides, it does not create extensive plateaus in the search space, as we shall see later. If we do not explicitly mention that we used another evaluation function, we used $f = f_s$.

In the following subsections, we discuss various stochastic local search techniques that are based on these ideas. The run-length and run-time distributions we show in this section in order to support our claims are all based on the first instance of the test set `BBT64-02`, the hardest test set among those that we found (see Section 6.1 for a description).

## 5.1 Initialization Techniques

Probably the simplest initialization method is *Random Picking*, which just randomly chooses a $\varphi \in \Phi_i$. Another approach is *Greedy Randomized Construction Search*. It starts with a partial embedding $\varphi : V \rightharpoonup \{0, \ldots, 2^d-1\}$ that only maps the root node to 0. Then, as long as there exists a node $v \in V \setminus \text{dom } \varphi$ such that $v$'s parent node $u$ has already been assigned a hypercube node number (i.e., $u \in \text{dom } \varphi$), $\varphi$ is extended by mapping $v$ to a value $m \in \{0, \ldots, 2^d - 1\} \setminus \varphi(V)$ that is randomly picked among those values which minimize $hd(\varphi(u), m)$. Unless otherwise stated, we use the greedy initialization for our local search approaches.

The latter construction mechanism already gives rise to a simple stochastic local search algorithm: Repeatedly construct a candidate embedding $\varphi$ until an embedding with $f(\varphi) = 0$ is found. Obviously, the ordering of the nodes has an influence on the construction search. Similar to the algorithm by Helmer and Eisenberg [3], we use a breadth first ordering. However, our algorithm never backtracks, which is a very important difference (see also Section 6.3).

## 5.2 Iterative Best Improvement

An *Iterative Best Improvement* local search algorithm can use one of the initialization functions from the previous subsection and then improve this candidate embedding by searching in a certain neighbourhood. Two candidate embeddings $\varphi_1, \varphi_2 \in \Phi_i$ are *neighbours* if and only if there exist two nodes $u, v \in V$, $u \neq v$, such that

- $\varphi_1\big|_{V'} = \varphi_2\big|_{V'}$, where $V' := V \setminus \{u, v\}$, and
- $\varphi_2(u) = \varphi_1(v)$ and $\varphi_2(v) = \varphi_1(u)$.

Intuitively, $\varphi_2$ can be obtained from $\varphi_1$ by swapping the embedding of nodes $u$ and $v$. In each step, one of the neighbouring candidates with minimal evaluation function value is randomly selected according to a uniform distribution. Figure 5 gives the pseudo-code for this algorithm for an evaluation function $f$ (e.g., $f = f_s$). Since we are only interested in perfect embeddings, there is no need to keep track of an incumbent candidate embedding (the best embedding found so far), because we can terminate the search as soon as we hit a perfect embedding. The algorithm also includes a limited tabu mechanism: In order to avoid immediately undoing a worsening step, the two nodes involved in the swap are declared tabu (alternatives are discussed below). At any time only two nodes are tabu so that the search is not overly restricted. The search is terminated and immediately restarted if no improving step has been found after a specified number of steps (see below).

Of course we would like to know if the tabu mechanism is a good choice. Therefore, we measured run-length distributions which count the number of search steps for various tabu mechanisms and dynamic cutoff values. The tabu mechanisms we considered are:

**tabu0** After a worsening swap, no nodes are declared tabu.

**input:** binary tree $T = (V, E)$, $V = \{v_1, \ldots, v_n\}$
**output:** embedding $\varphi$ of $T$ in the corresponding hypercube or $\emptyset$

```
φ ← init(T)
tabu ← ∅
while not terminate and (f(φ) > 0) do {
    N ← ∅
    for i = 1 to n − 1 do
        for j = i + 1 to n do
            if (tabu ∩ {i, j} = ∅) then {
                φ' ← ⟨φ with φ(v_i) and φ(v_j) swapped⟩
                N ← N ∪ {φ'}
            }
    m ← min{f(φ') | φ' ∈ N}
    φ' ← ⟨randomly chosen from {φ' ∈ N | f(φ') = m}⟩
    if (f(φ') > f(φ)) then tabu ← {i ∈ {1, ..., n} | φ'(v_i) ≠ φ(v_i)}
    φ ← φ'
}
if (f(φ) = 0) then return φ else return ∅
```

Figure 5: Iterative Best Improvement Algorithm

**tabu1rel** One of the two nodes involved in the worsening swap (randomly chosen) is declared tabu. After an improving step, the tabu status is released.

**tabu1** One of the two nodes involved in the worsening swap (randomly chosen) is declared tabu, but the tabu status is not released after an improving step. (The tabu status of this node is automatically released after another non-improving step.)

**tabu2rel** Both nodes whose embeddings were swapped are declared tabu as a result of a worsening step. After an improving step, the tabu status is released for both nodes.

**tabu2** Both nodes whose embeddings were swapped are declared tabu, but the tabu status is not released after an improving step. (Again, the tabu status is automatically released after another non-improving step.)

Since the optimal dynamic restart strategy may vary between those alternatives, we determined for each tabu mechanism the optimal number of non-improving steps after which the search should be restarted among 20, 50, 100, 150, 200, 300, and 400. As Figure 6 (which is based on the individually approximately optimal restart strategies) shows, the differences between the tabu mechanisms are very marginal, which gives some interesting insight into the topology of the search space: For each neighbour of a local minimum, there are typically several other neighbours that are at least as good as the local minimum, so the search rarely falls back into a previously visited local minimum. As the `tabu2` variant performed slightly better than `tabu0` and `tabu1` in the initial phase (up to 5000 search steps), we chose this tabu mechanism for our future investigations. The "release"

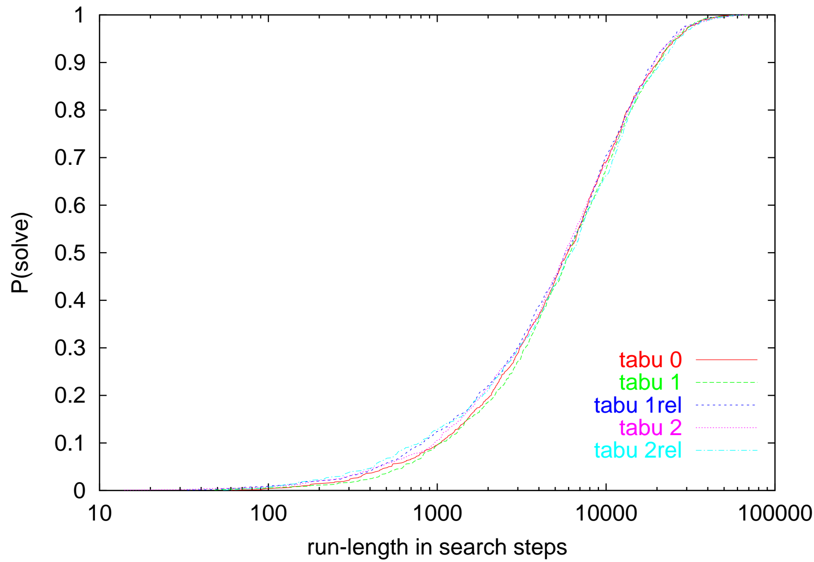


Figure 6: Comparison of five tabu mechanisms for Iterative Best Improvement

variants showed better performance in the initial search phase, but later on were not quite as good as `tabu2`.

As mentioned above, we optimized the restart strategy for each tabu mechanism. Figure 7 illustrates our experiments for the "winner" of the competition, `tabu2`. The three worst performing strategies restart the search after 20, 50, or 100 non-improving search steps, respectively. This indicates that a good amount of "patience" is necessary to escape from locally minimal evaluation function values and to finally reach a global minimum. At the same time, we see that "impatience" is punished: The search needs a certain time to reach promising regions of the search space, so restarting too eagerly discards this initial effort. On the other hand, the performance decreases if we spend too much time in a non-promising region before giving up and restarting. We observed this performance drop beginning at cutoff values of 300, so our value of choice is 200. (Of course, the optimal value depends on the instance. Since we are mainly interested in embedding the "hardest" trees, we chose 200 for all instances; the "easier" the trees are, the less important becomes the cutoff value, because embeddings are typically found quite early.)

Let us now have a quick look at two other implementation choices. Firstly, we could use another evaluation function, in particular $f_e$. Secondly, the initialization could be done by Random Picking. In Figure 8, we see that Random Picking causes the algorithm to perform considerably more search steps. Even worse is the situation for the evaluation function $f_e$. The main reason for this is that with $f_e$, the search landscape has undesirable plateaus: The (discrete) evaluation function values often are in the range between 1 and 7, and many steps arbitrarily trade off the dilation of one edge against the dilation of another edge. The use of $f_s$, however, directs the search, so the embedding of the roots of big subtrees is fixed before the "details" towards the embedding of leaf nodes are settled. Note that both evaluation functions lend themselves to an efficient cache-and-update implementation. Each pair of connected nodes contributes "locally" to the evaluation

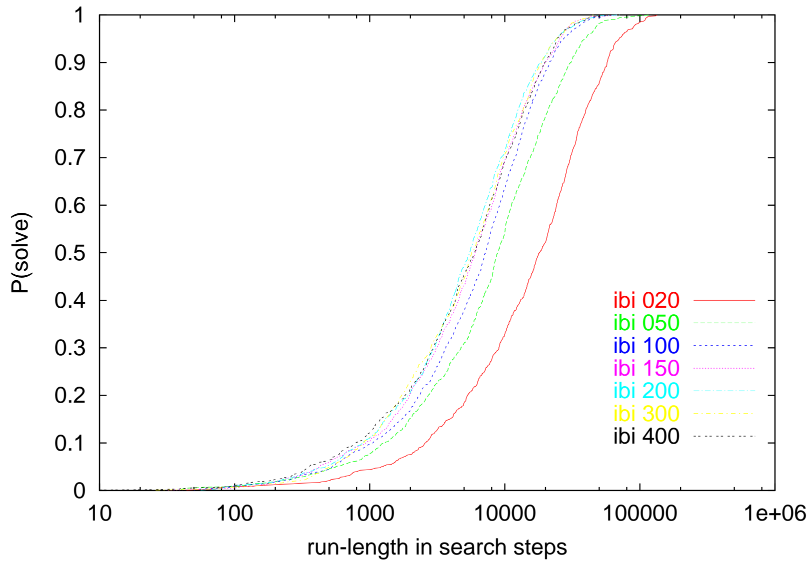


Figure 7: Determining the optimal dynamic restart strategy for Iterative Best Improvement

function value, so after every swap only few calculations are necessary to assess the quality of the modified candidate embedding.

One might argue that a Random Picking initialization is much easier (and thus faster) to compute than the more complicated Greedy Construction. This is true to some extent, but at a closer look the Greedy Construction does not take too much time: As long as there are "sufficiently many" unused hypercube nodes, it is possible to embed a node at Hamming distance 1 to its parent node, which limits the choice considerably; all nodes at a greater distance do not need to be considered. This intuitive explanation is backed up by Figure 9, which is based on actual running times and not on the number of search steps (we will describe the running environment in Section 6).

We conclude the description of the Iterative Best Improvement algorithm with the following summarizing observation: The algorithm is fairly robust with respect to parameter settings (tabu mechanism and restart strategy). Although the presented analyses of run-time behaviour used an instance from the test set `BBT64-02`, we obtained very similar results for other balanced binary trees. Since most other instances are "easier", it is more important to optimize the algorithm for the difficult cases. For example, a generous setting of the cutoff value has a rather limited effect, because the search is terminated anyway if a solution is found earlier.

## 5.3 A Population-based Algorithm

As the necessity of dynamic restarts indicated, some diversification is needed to find perfect embeddings. On the other hand, it is not always advisable to just restart the whole search process. Often it is a good idea to merely swap the embeddings of two subtrees, because the current candidate embedding might use some regions of the hypercube inefficiently. This forms the idea of our

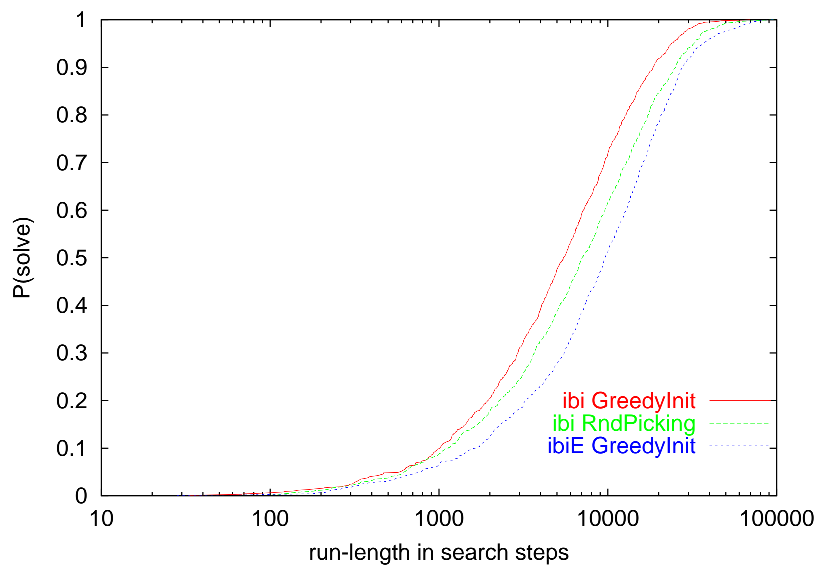


Figure 8: Using Random Picking or $f_e$ in the Iterative Best Improvement approach

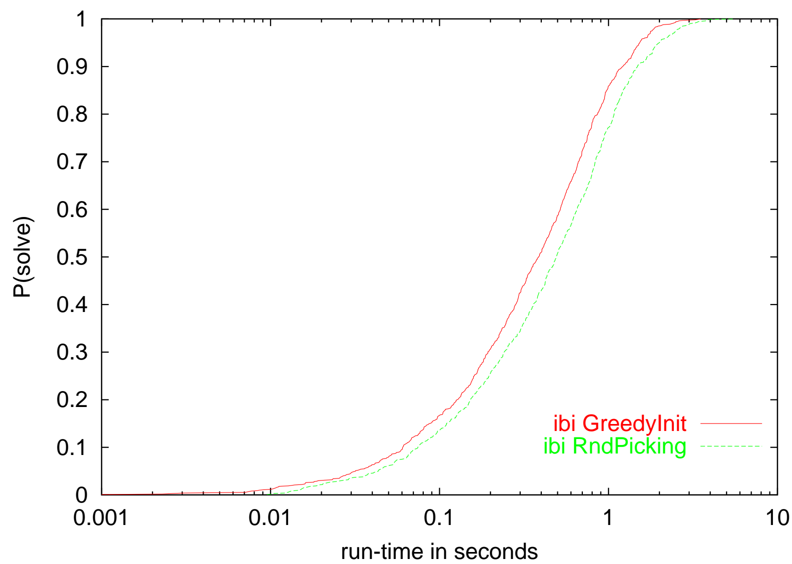


Figure 9: Run-time distributions for different initialization techniques

**input:** binary tree $T = (V, E)$, $V = \{v_1, \ldots, v_n\}$
**output:** embedding $\varphi$ of $T$ in the corresponding hypercube or $\emptyset$

for $i = 1$ to $k$ do {
  $\varphi_i \leftarrow \mathrm{IterativeBestImprovement}(\mathrm{init}(T))$
  if ($f(\varphi_i) = 0$) then return $\varphi_i$
}
while not $terminate$ do {
  // Mutation
  for $i = 1$ to $k$ do {
    $\langle$randomly choose a node $u$ of $T$ with 2 children, $v_1$ and $v_2\rangle$
    $\varphi_{k+i} \leftarrow \langle\varphi_i$ with $\varphi_i(v_1)$ and $\varphi_i(v_2)$ swapped$\rangle$
  }
  // Local Search
  for $i = 1$ to $k$ do {
    $\langle$Mark one of the nodes in which $\varphi_{k+i}$ and $\varphi_i$ differ as tabu for the local search phase$\rangle$
    $\varphi_{k+i} \leftarrow \mathrm{IterativeBestImprovement}(\varphi_{k+i})$
    if ($f(\varphi_{k+i}) = 0$) then return $\varphi_{k+i}$
  }
  // Selection
  $(\varphi_i)_{1 \le i \le k} \leftarrow \langle$the $k$ best candidate embeddings from $(\varphi_i)_{1 \le i \le 2k}\rangle$
}
return $\emptyset$

Figure 10: A population-based local search algorithm

population-based local search algorithm.

The population-based approach maintains a set of $k$ candidate embeddings, which are independently generated by an initialization function (Greedy Randomized Construction). Each candidate embedding is locally optimized by the Iterative Best Improvement technique from above. Due to the hypercube's symmetry, there are many ways of embedding the same tree in the hypercube (if there is one at all), so there is little hope that a recombination of two candidate embeddings yields a useful candidate embedding. This is why our algorithm is only loosely related to the idea of genetic algorithms. One can also see it as a population-based Iterated Local Search algorithm.

To each of the $k$ candidate embeddings we apply a *mutation* step (in the Iterated Local Search context, this is called *perturbation*). It randomly selects a node with two child nodes and swaps the embedding of those two children (*not* of the whole subtree). Iterative Best Improvement then leads these new candidate embeddings to local optima (for example, by adjusting the embeddings of the subtrees). In order to avoid that the mutation is immediately undone, one of the child nodes is declared tabu for the local search phase. The $selection$ process then selects the $k$ best candidate embeddings among the $2k$ candidates for the next generation, where ties are broken randomly. Figure 10 shows pseudo-code of this algorithm. It terminates when a fixed number of generations has been generated or, of course, when an embedding has been found.

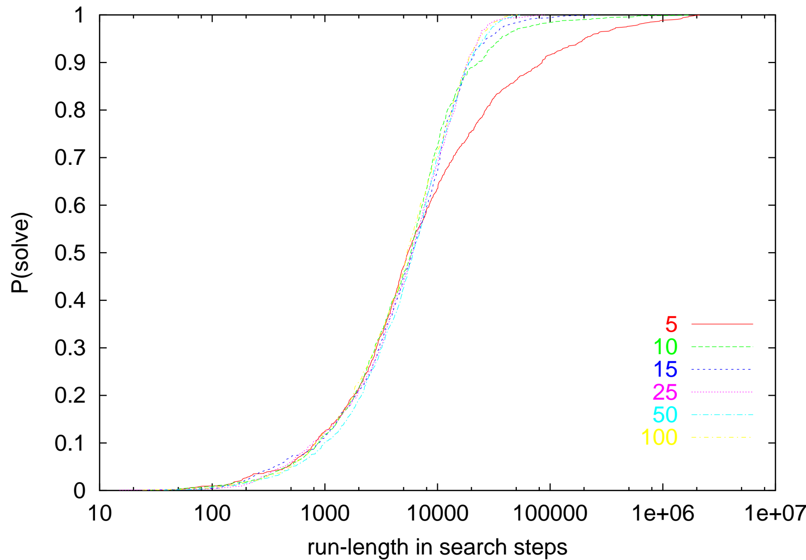


Figure 11: Effect of the population size $k$ on the population-based algorithm

Obviously, we need to find a suitable population size $k$. Figure 11 shows the run-length distributions for population sizes between 5 and 100. For low run-lengths ($< 10000$), there is no significant difference between the varying population sizes. This is not very surprising, because in this range we basically have the Iterative Best Improvement algorithm from above with a limited number of restarts (given by the population size). If those attempts do not succeed, however, the very limited amount of diversification in small populations results in a decreasing success probability for longer runs. But it is interesting to note that for very long runs we still reach a success rate of 100%. This indicates that the mutation (or perturbation) mechanism is strong enough to prevent complete stagnation; it simply takes many generations until sufficiently diverse candidate embeddings have been explored. A population of size 25 seems to be a good compromise between a sufficient amount of diversification and the cost of maintenance.

We also studied the impact of "migration": Here, we only keep $k - k'$ of the best candidate embeddings of a generation and introduce $k'$ new, re-initialized candidate embeddings, hoping to increase the diversification even more. Interestingly, this had virtually no influence on the behaviour of the algorithm; the run-length distributions are almost perfectly congruent with the original ones (and we therefore do not present them graphically). This underlines once more that the mutation mechanism works successfully and at the same time gives us the benefit of being computationally cheaper than a re-initialization.

We already pointed out that the number of embeddings of a tree is typically quite large. This makes an analysis of the search landscape relatively difficult, because properties such as Fitness-Distance Correlation [4] are hardly identifiable if the set of all perfect embeddings is unknown. However, we already found some indirect clues about the search space topology when we analyzed the Iterative Best Improvement algorithm. Furthermore, the population-based algorithm in Figure 10 is actually based on this insight: The Iterative Best Improvement algorithm does not keep track

of the best candidate embedding that has been found during the search phase. We argued that this was not necessary, because we were not interested in a candidate embedding unless it was a perfect one. But in the population-based approach we *are* interested in the result of the subsidiary local search phase even if it did not succeed. Hence, one might wonder if the additional effort to store an incumbent candidate embedding could pay off. The answer is: No. Recall that we saw that the different tabu mechanisms had virtually no effect on the performance of the Iterative Best Improvement algorithm. We concluded that after each worsening step there always seems to be an alternative step that brings us to a different, but at least equally good candidate embedding as the one we left before. Consequently, we immediately regain the solution quality we just lost. In fact, experiments showed that the quality of the candidate embedding returned by the Iterative Best Improvement algorithm is not worse than the quality of the best candidate embedding that has been encountered during the search.

In addition, we tried some alternate selection criteria. Limited experiments did not show any increase in performance (often rather a decrease), so we just briefly mention two major alternatives:

- *Random Selection*: From the $2k$ candidate embeddings, $k$ candidate embeddings are randomly selected regardless of their evaluation function values.
- *Pairwise Selection*: Since $\varphi_{k+i}$ is obtained from $\varphi_i$, we select $k$ candidate embeddings from the set of $2k$ candidate embeddings by pairwise comparing the evaluation function values of $\varphi_i$ and $\varphi_{k+i}$, always selecting the better one. (Notice that this variant is very closely related to Iterated Local Search, performed on all members of the population.)

# 6 Performance Evaluation

Having presented several embedding algorithms, we now analyze their relative performance on various balanced binary trees. In Section 6.1, we describe the test sets. Afterwards, we briefly report our results on the extensive embedding experiment of all strongly balanced binary trees with 32 nodes. Finally, we evaluate the performance of the embedding algorithms on trees of varying difficulty. All of our experiments were run on Intel Xeon 2 GHz dual processor machines running under Linux. Our run-time distributions are based on CPU seconds.

Previously, we already used run-length distributions to assess the quality of single embedding algorithms. Figure 12 shows that there exists a linear correlation between the number of local search steps and the run-time of the Iterative Best Improvement algorithm. As neither mutation nor selection cause significant overhead, this result also holds for the population-based algorithm. While this justifies the use of run-length distributions for tuning a single algorithm, we present run-*time* distributions for our comparative evaluation of different algorithms, because the complexity of local search steps varies considerably between them.

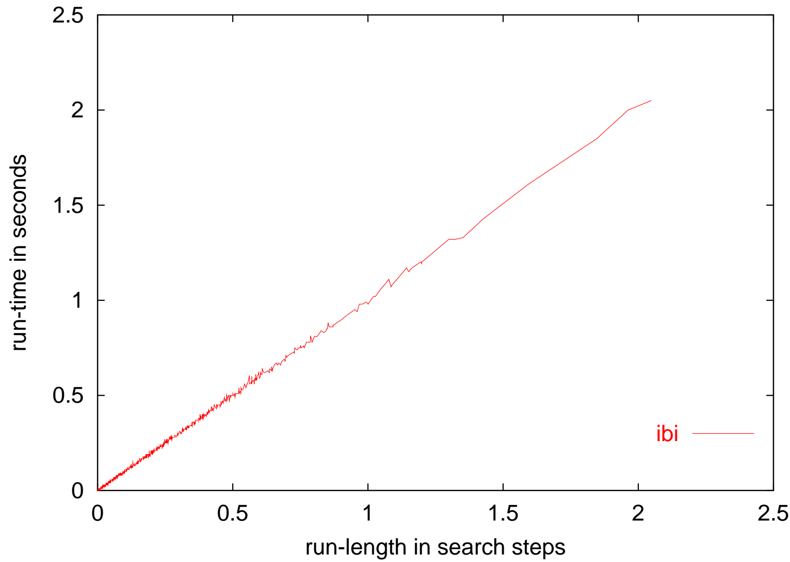


Figure 12: Correlation of run-length and run-time (Iterative Best Improvement on `bbt64-02`)

### 6.1 Tree Generation

The "easiest" binary trees are the strongly balanced ones. In order to prove that all strongly balanced binary trees with 32 nodes can be embedded into the corresponding hypercube (see Section 6.2), we generated this set exhaustively according to the algorithm described in [1]. Using the same algorithm, we generated some strongly balanced binary trees with 64 nodes as well, which we refer to as `SBBT64`.

Since we are interested in seeing how our algorithms perform on "harder" trees, we followed the approach by Helmer and Eisenberg [3] and generated trees that they reported to be the most difficult ones. The idea is to generate a tree with few nodes of degree 2 and many nodes with degree 3, because the degree 3 nodes pose tighter constraints on the embeddings. The generation works as follows, where $m$ is a parameter that controls the number of degree 2 nodes (the description is taken from [3]):

1. Generate a full tree of $n - m$ nodes.
2. Randomly choose a node to "stretch" by adding a new node between it and its parent.
3. Repeat step 2 until the tree contains $n - 1$ nodes.
4. Choose the last node to stretch such that stretching this node will result in a colour balanced tree. It is possible that there is no node with this property. When this occurs, restart with step 1.

In this paper, the sets of trees generated according to this algorithm are referred to as `BBT64-MM`,

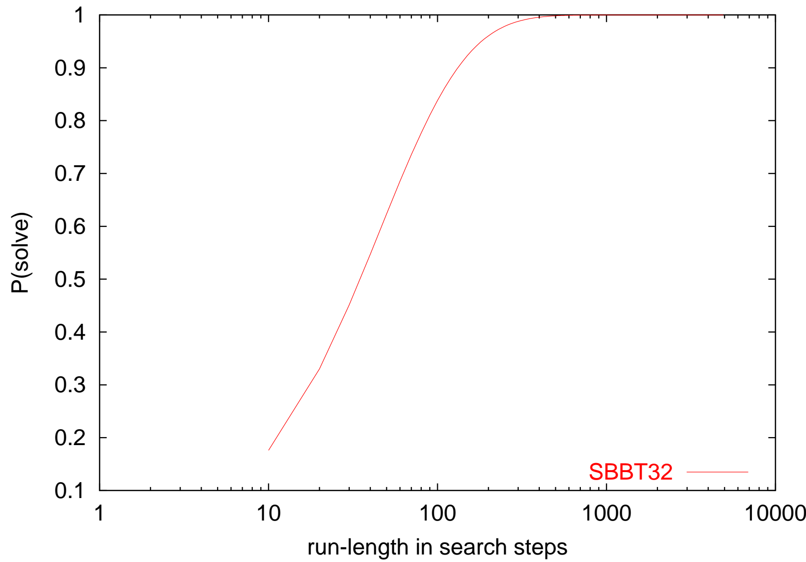


Figure 13: Statistics of the embedding of all strongly balanced binary trees with 32 nodes

where `MM` indicates the value of $m$ (all trees have 64 nodes in total). We write `bbt64-MM` to denote the first instance of the corresponding test set.

### 6.2 Embedding All Strongly Balanced Binary Trees with 32 Nodes

The simple Repeated Greedy Randomized Construction Search algorithm from Section 5.1 was already good enough to embed all 368,422,352 strongly balanced binary trees with 32 nodes into the corresponding hypercube. Due to the tremendous amount of data, we measured the run-lengths in multiples of 10, so the averaging run-length distribution (number of initializations) over all those trees in Figure 13 begins at 10.[2] The simplicity of embedding this kind of trees is reflected in the fact that 64,891,995 trees (17.5%) were successfully embedded after less than 10 tries. Only 12,219 trees (less than 0.00332%) needed more than 1,000 guesses until an embedding was found, which explains the long right tail in the plot. Besides, every tree was embedded exactly once, so the variability in search cost also stems from the underlying "random guesses". In total, this experiment took about 17 CPU hours.

### 6.3 Performance Comparison

Although presenting only sparse performance results, Helmer and Eisenberg found the trees of `BBT64-02` to be very difficult to embed (in fact, they found them so difficult that their embedding

[2]Each run of the embedding algorithm generates an integer value, the run-length, which amounts to about 1.4 GB of data. Storing the run-lengths in bins of size 10 reduces the amount of data significantly. However, we can no longer differentiate between runs of lengths between 1 and 10, for example.

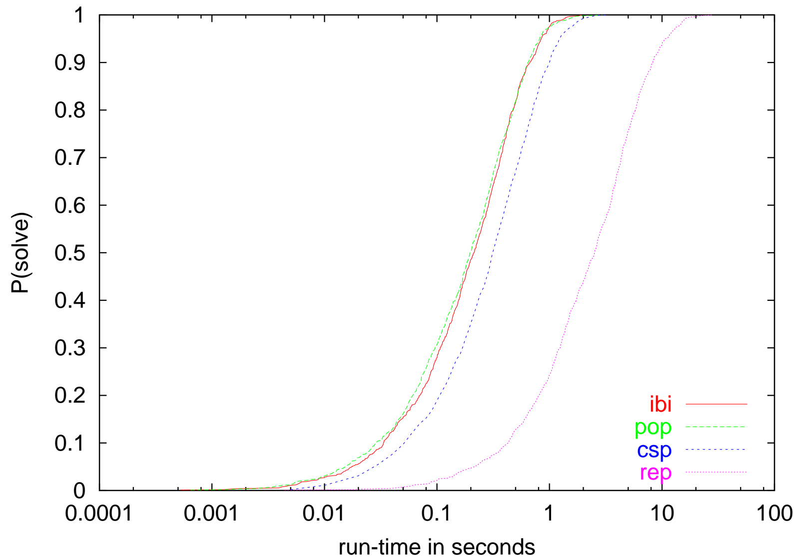


Figure 14: Run-time distributions for `bbt64-02`

algorithm had a "success rate" of 0%). This is most probably due to their use of backtracking. Our reimplementation of this randomized breadth first search confirmed the bad performance, so there was no way of measuring run-time or run-length distributions within a reasonable amount of time. The run-time distributions in this section therefore focus on the other algorithms, all of which reached a success rate of 100% within a few seconds.

Figure 14 shows the run-time distributions of our Iterative Best Improvement algorithm (ibi), the related population-based algorithm (pop), the local search algorithm for a CSP representation of the corresponding embedding problem from Section 4.2 (csp), and the simple repeated initialization approach (rep) for the first instance of `BBT64-02`. The population-based algorithm probabilistically dominates the Iterative Best Improvement algorithm only very marginally. The CSP variant performs nearly equally as good, whereas repeated initialization search takes approximately ten times as long to find embeddings. For `bbt64-05`, shown in Figure 15, the situation is quite similar, although IBI and POP are almost indistinguishable in their overall performance. The repeated initialization approach works slightly faster than for `bbt64-02`, indicating that trees in `bbt64-05` are "easier" in the sense that an embedding is easier to guess.

CSP cannot exploit the decreasing difficulty of the embedding problem for trees with an increasing number of nodes of degree 2. Whereas the curves for IBI, POP and REP move constantly to the left, the run-time distribution for CSP remains almost invariant when we move on to test instance `bbt64-10`, shown in Figure 16. The repeated initialization approach is even able to "overtake" CSP. Most amazingly, CSP has the most difficulties with strongly balanced binary trees that are fairly easy for the other algorithms according to Figure 17. This may be due to the long paths in strongly balanced trees. However, the analysis of this phenomenon would require examining the number of unsatisfied constraints over time for classes of paths grouped by length. If long paths cause problems and it is often harmful to have unsatisfied long paths, the WeightedMCH algorithm

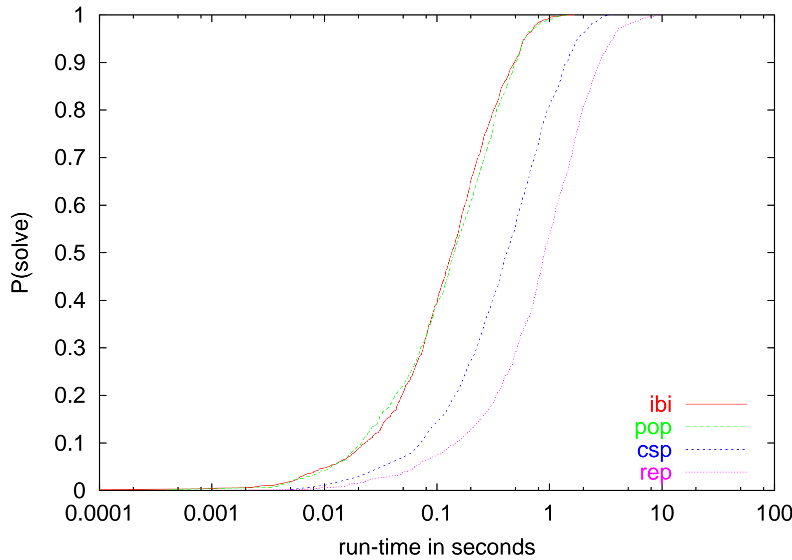


Figure 15: Run-time distributions for bbt64-05

may need some modifications so that longer paths are more likely to be satisfied. Perhaps adding a tabu mechanism to the edges in the longest paths once the constraints have become satisfied may improve the competitiveness of this algorithm with respect to our other approaches, but this area of interest has not been explored.

Altogether, we conclude that there is virtually no difference in performance between the Iterative Best Improvement and the population-based algorithm. Both run-time distributions fit almost perfectly an exponential distribution such as $1 - \exp(-1.22 \cdot 10^{-4} \frac{1}{\mathrm{s}} \cdot t)$ for bbt64-02. Maybe a more intelligent perturbation mechanism could push the search even more quickly towards a solution, but it would have to be a computationally cheap mechanism; otherwise we might merely reduce the run-length and not the total run-time.

## 6.4 Comparing Representative Trees

Thus far for each class of trees we have only used one representative tree for the analysis of our algorithms. For a more complete analysis of the classes of trees we compared our representative tree (the first one of each class) against more trees that are in the same class of difficulty. Figure 18 shows run-time distributions of five randomly sampled trees from classes BBT64-02 and BBT64-10 using the Iterative Best Improvement algorithm.[3] Obviously, the classes appear clustered in the run-time distribution. While two adjacent classes like BBT64-09 and BBT64-10, for example, may well overlap (meaning that an instance of the "more difficult" class probabilistically dominates an instance of the "easier" class), there is an overall trend that trees of the same class are about of the same difficulty.

[3] We only found five trees in BBT64-02, so we actually sampled *all* trees. The other classes that we considered contain considerably more trees, however.

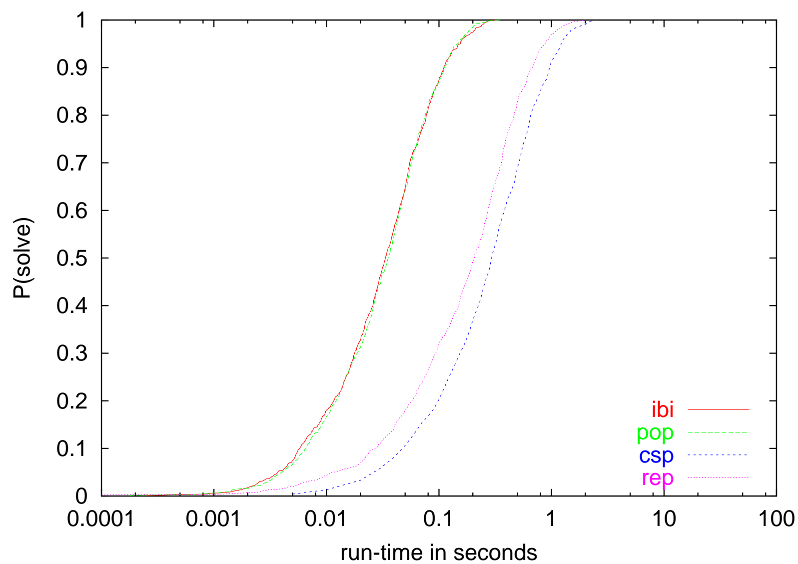


Figure 16: Run-time distributions for `bbt64-10`

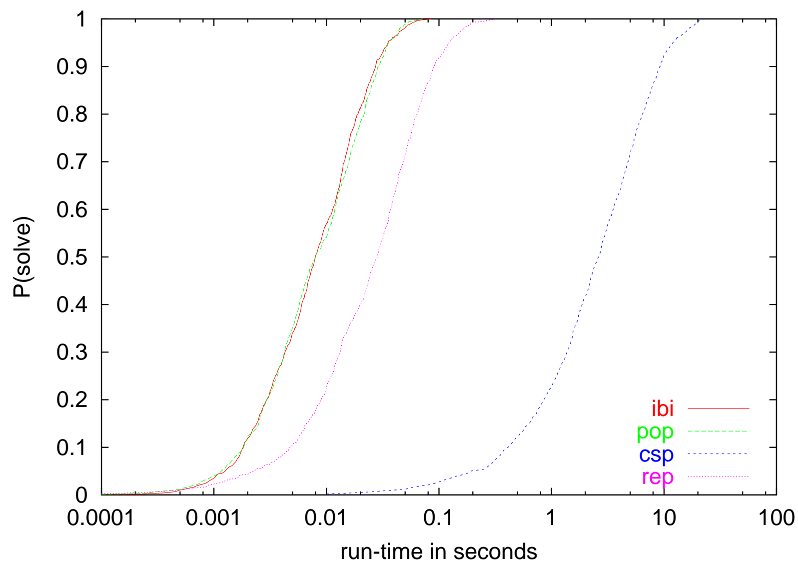


Figure 17: Run-time distributions for `sbbt64`

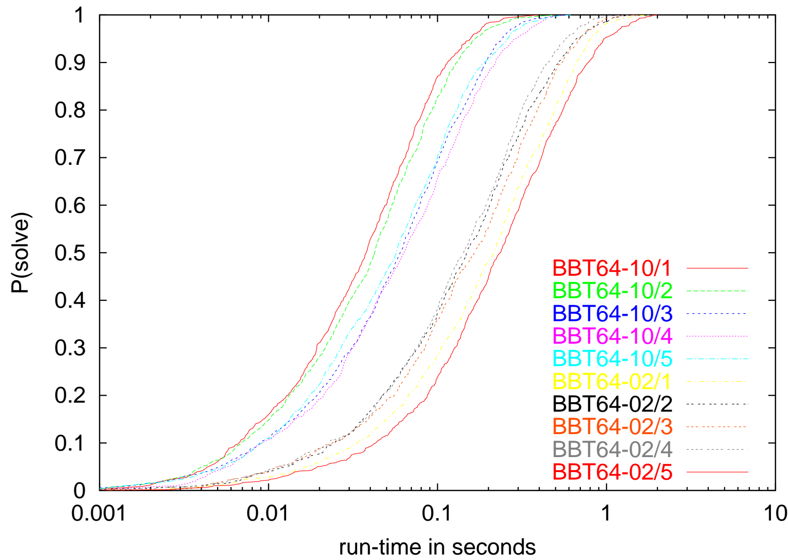


Figure 18: Run-time distributions for samples of `BBT64-02` and `BBT64-10`

Because CSP showed unexpected results on strongly balanced binary trees, for further analysis we compared the run-time distribution of the representative tree with an averaging run-time distribution of up to 10,000 trees from the same class. This produced Figures 19 to 22. As shown in the figures, `bbt64-02`, `bbt64-05`, and `bbt64-10` show similar run-time distributions as the "average" trees of the respective classes. Although CSP performed relatively poorly on `sbbt64`, we can see that according to the average run-time distribution of trees from `SBBT64` this tree is not the most difficult one for CSP.

# 7 Conclusions

In this paper, we explored various approaches to embedding balanced binary trees in hypercubes. We described how to encode the problem as SAT and CSP instances. Furthermore, we presented two stochastic local search algorithms with a specialized evaluation function that is specifically tailored to trees that are very likely to be embeddable. In particular, we justified implementation choices and parameter settings by empirical evaluations.

For the comparative evaluation of our algorithms, we generated and successfully embedded balanced binary trees with different characteristics. It turned out that an Iterative Best Improvement approach consistently showed best performance. While our algorithms are naturally unable to prove or disprove Havel's conjecture, they may yet be useful in attempts of disproving it: When looking for a counterexample (i.e., a tree that cannot be embedded in the hypercube), one can quickly run one of the algorithms to find out if there is no simple way of embedding the tree—that is, one can quickly reject false counterexamples.

It might be interesting to investigate the performance of the presented algorithms on other em-

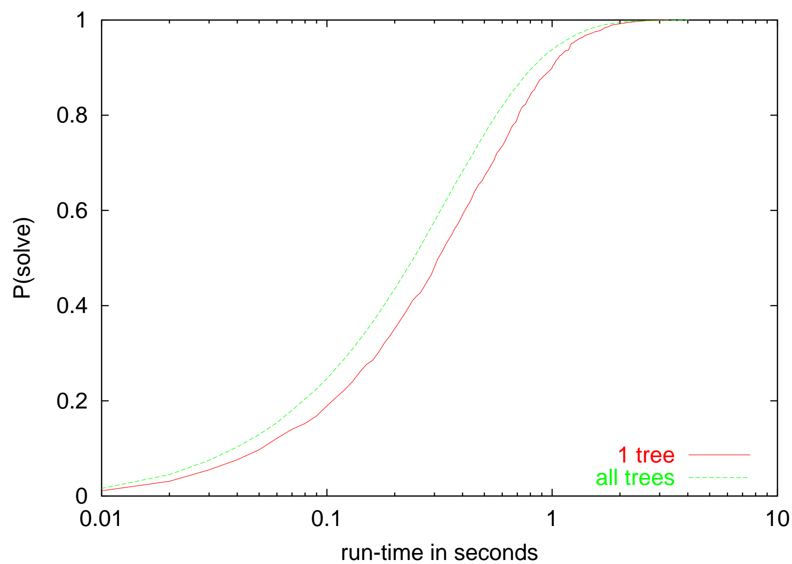


Figure 19: Comparing the first tree with all trees of `BBT64-02` using CSP based algorithm

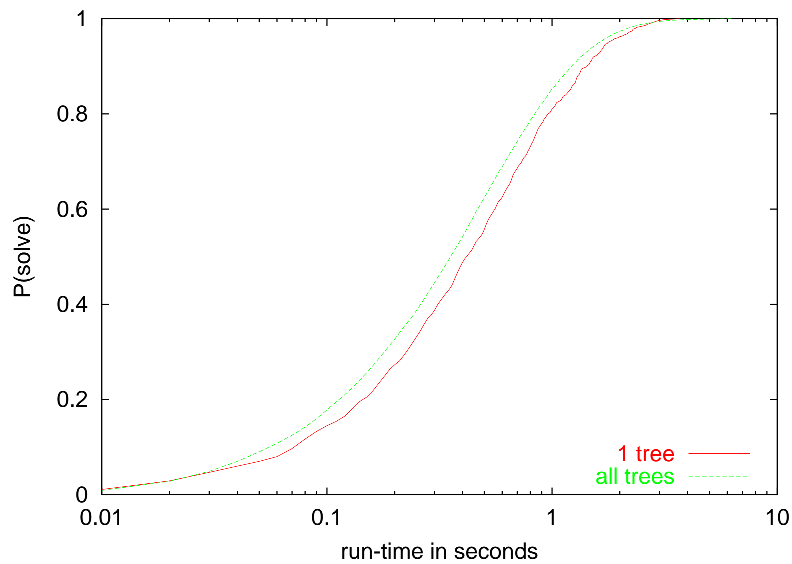


Figure 20: Comparing the first tree with all trees of `BBT64-05` using CSP based algorithm

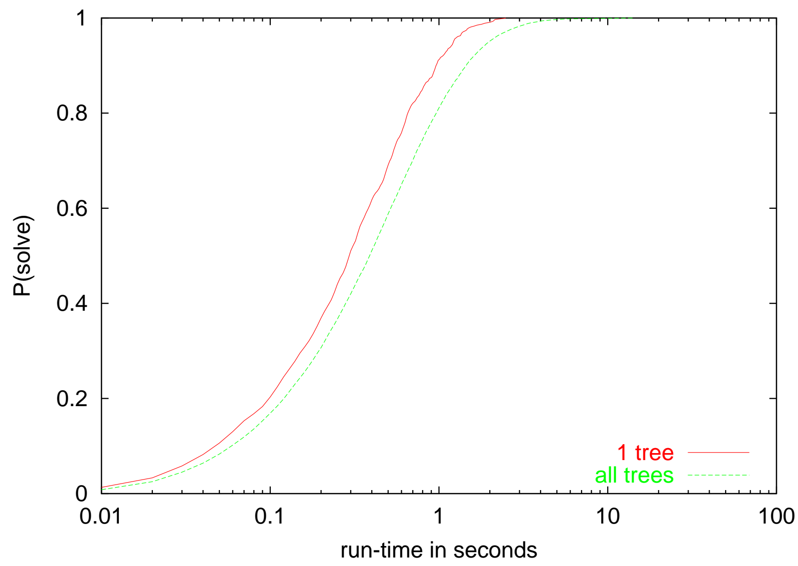


Figure 21: Comparing the first tree with all trees of `BBT64-10` using CSP based algorithm

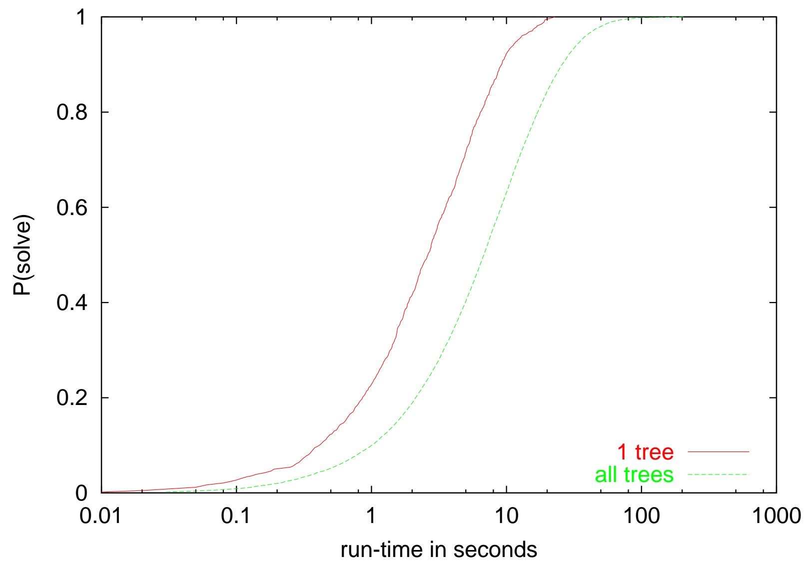


Figure 22: Comparing the first tree with all trees of `SBBT64` using CSP based algorithm

beddable, but not necessarily balanced binary trees. Our algorithms do not explicitly exploit the balance of trees, so they can be applied to more general classes of trees. Future research could compare the difficulty of embedding those trees versus balanced binary trees. Concerning the algorithms, a clever recombination operator, which might combine good embeddings of subtrees, could lead to a genetic algorithm and thus improve on the performance of our population-based algorithm. Another interesting approach may use the concept of Ant Colony Optimization, where the pheromone trails might mark promising embeddings of particular nodes.

## References

[1] M. Aderhold. Embedding strongly balanced binary trees in the hypercube. University of British Columbia, December 2002.

[2] I. Havel and J. Morávek. B-valuations of graphs. *Czechoslvak Math. J.*, 22:338–351, 1972.

[3] S. Helmer and A. Eisenberg. Exploring issues of embedding color balanced binary trees into 64 node hypercubes. University of British Columbia, December 2002.

[4] H. H. Hoos and T. Stützle. *Stochastic Local Search—Foundations and Applications*, chapter 5. Morgan Kaufmann Publishers, USA, to appear.

[5] H. H. Hoos and T. Stützle. *Stochastic Local Search—Foundations and Applications*, chapter 6. Morgan Kaufmann Publishers, USA, to appear.

[6] G. Smedley. Algorithms for embedding binary trees into hypercubes. Master's thesis, University of British Columbia, 1989.

[7] A. S. Wagner. Embedding all binary trees in the hypercube. *Journal of Parallel and Distributed Computing*, 18:33–43, 1993.

[8] A. S. Wagner and D. Corneil. Embedding trees in the hypercube is NP-complete. *SIAM Journal on Computing*, 19(4):570–590, June 1990.