Searching Stochastically-Generated Multi-Abstraction-Level Design Spaces

Louis Steinberg
Department of Computer Science, Rutgers University
New Brunswick, NJ 08903
lou@cs.rutgers.edu

Abstract

We present a new algorithm called Highest Utility First Search (HUFS) for searching trees characterized by a large branching factor, the absence of a heuristic to compare nodes at different levels of the tree, and a child generator that is both expensive to run and stochastic in nature. Such trees arise naturally, for instance, in problems such as microprocessor design which involve candidate designs at several levels of abstraction and which use stochastic optimizers such as genetic algorithms or simulated annealing to generate a candidate at one level from a parent at the previous level. This report explains the HUFS algorithm and presents experimental results demonstrating the advantages of HUFS over alternative methods.

INTRODUCTION

Motivation

In many kinds of engineering design tasks, the design process involves working with candidate designs at several different levels of abstraction. E.g., in designing a microprocessor, we might start with an instruction set, implement the instructions as a series of pipeline stages, implement the set of stages as a “netlist” defining how specific circuit modules are to be wired together, etc. There are typically a combinatorially large number of ways a design at one level can be implemented at the next level down, but only a small, fixed set of levels, perhaps a dozen or so. Thus the search space is a tree with a small, fixed, uniform depth but a huge branching factor. These characteristics are important when considering how to do design in such domains.

Furthermore, the partial solutions at different levels are entirely different types of things, and it is hard to come up with heuristic evaluation functions that allow us to compare, say, a set of pipeline stages with a netlist. In fact, even if $S_1$ and $S_2$ are two alternative sets of pipeline stages with $S_1$ better than $S_2$ by the standard metrics, it is possible that there are netlists $N_1$ and $N_2$ descended, respectively, from $S_1$ and $S_2$ such that $N_1$ is clearly worse than $N_2$. It even possible, though unlikely, that the average descendant of $S_1$ is worse than the average descendant of $S_2$.

The huge branching factor and the limitations in comparing designs within and across levels make it hard to apply standard tree-search algorithms to this search space. Note that A*, for instance, is irrelevant both because we do not care about the path length to our solution (all paths from the root to a leaf are of equal length) and because we cannot afford to generate all the children of even one node.

Recently, a number of techniques for stochastic optimization have been shown to be useful for such problems. These techniques include simulated annealing (Ingber 1996; William P. Swartz 1993), genetic algorithms (Michalewicz 1996; Goldberg 1989; Rasheed 1998; Blaize, Knight, & Rasheed 1998), and random-restart hill climbing (Zha et al. 1996). A design at one level is translated into a correct but poor design at the next level, and a stochastic optimizer is used to improve this design. An inherent feature of a stochastic method is that it can be run again and again on the same inputs, each time potentially producing a different answer. These alternatives can each be used as inputs to a similar process at the next lower level. Thus, these optimizers can be seen as generating a tree of design alternatives. These trees are much smaller than the original trees, and consist only of relatively high-quality alternatives. However, there can still be significant variations in quality among alternatives and these trees can still have a large branching factor (in the thousands for examples we have looked at). Also, it can take from minutes to days of computer time to run an optimizer and generate a single descendant. So, there is still the problem of controlling the search within the smaller tree, that is, for deciding which alternative to generate a child from next. We will refer to such trees as Stochastic/Abstraction, or S/A, trees.

We have developed an algorithm for searching S/A trees called Highest Utility First Search (HUFS) (Steinberg, Hall, & Davison 1998). HUFS assumes that there is no global heuristic scoring function, but that there is a scoring function for each level. HUFS assumes that the score of an object, while not an infallible predictor, is correlated to some degree with the quality of the object’s children. In our testbed domain (described below), a score represents a cost to be minimized, so we...
will assume lower scores are better.

Finding the globally optimal design in these kinds of problems is computationally intractable, so our goal is not a search process which finds the best design but one which has the best tradeoff between the quality of the result we get and the cost in search time it takes to find that result. In decision theoretic terms (Luce & Raiffa 1957; Tribus 1969) we are looking for the search process with the highest expected utility. In our initial work we followed Russell and Wefald (Russell & Wefald 1991) in defining utility to be the difference

\[ \text{value of result} - \text{cost of time} \]

We assume that we have a score function \( S_0(d) \) which tells us the quality of a ground-level design alternative \( d \), and a value function \( V_0(s) \) which tells us the value of having a design with score \( s \). We also assume that for each level \( i \) the optimizer for level \( i \) costs some fixed amount \( c_i \) to run. So, if a search process takes \( n_i \) runs of the optimizer for level \( i \) and results in a design with score \( s \), then the utility of the search is

\[ V_0(s) - \sum_i n_i * c_i \]

In more recent work we have explored other definitions of utility; this will be discussed later.

HUFS is based on reasoning about the utility of the design process. To determine this utility, we use information about the Child Score Distribution of the design alternatives in the tree. The Child Score Distribution if of an alternative \( d \) is a probability distribution \( G(s|d) \):

\[ G(s|d) = P( S_0(d') = s | \text{child}(d', d)) \]

that is, \( G(s|d) \) is the probability that a randomly chosen child of \( d \) will have score \( s \).

The Example Problem: Module Placement

The initial example problem that we have been using to drive our work is the problem of positioning rectangular circuit modules on the surface of a VLSI chip: a given set of rectangles must be placed in a plane in a way that minimizes the area of the bounding box circumscribed around the rectangles plus a factor that accounts for the area taken by the wires needed to connect the modules in a specified way.

The input to the placement problem is a "netlist". A netlist specifies a set of modules, where each module is a rectangle of fixed size along with a set of "ports". A port is simply a location within the rectangle where a wire may be connected. In addition to giving the modules, a netlist specifies which ports of which modules must be connected by wires.

The output from a placement problem is a location and orientation for each module. Modules may be rotated by any multiple of 90 degrees and/or reflected in X, Y, or both. The modules’ rectangles may not overlap.

We break the placement process into two stages. First we choose a structure called a "slicing tree". A slicing tree is a binary tree. Each leaf is a module to be placed. Each non-leaf node represents the commitment to place a particular group of modules next to another group, in a given relative position. The slicing tree does not determine the reflections, however. The module or group of modules that correspond to any node in the tree can still be replaced by their reflections in X and/or Y. Reflecting a module cannot change the circuit’s bounding box area but it can change wire lengths and thus wire area.

A placement is generated from a slicing tree by a hill climbing process that starts with a random choice of reflections and considers each of a set of local modification operators until it finds one that improves the initial tree (according to a heuristic score function). It then applies that operator to get a new tree and begins again considering the operators. If no operator improves the current tree, we are at a local optimum and we return the current tree as output result. A similar process is used to generate a layout from a slicing tree.

HUFS

This section will give an overview of HUFS and the results published in our AAAI98 paper (Steinberg, Hall, & Davison 1998). For most of the details of the math, the reader is referred to that paper.

One way to understand HUFS is to view it as the result of applying a series of modifications to the algorithm commonly used today for S/A trees. We will call this current algorithm Waterfall. We will describe Waterfall and the sequence of improvements which leads from Waterfall to HUFS, and then describe the empirical tests we have done using HUFS.

Waterfall

In practice, human engineers faced with a design task like those discussed above often take the following simple approach: they work from top down, using CAD tools such as optimizers to generate a small, fixed number of alternatives at a given level. They choose the best of the designs generated at this level, using some heuristic that compares alternatives within the level, and then use this best design as the (only) parent from which to generate designs at the next level. This process is carried out level by level until the resulting designs are at the lowest level. The best of these ground-level designs is then output as the result of the search.

This method has the advantages of simplicity, low storage cost, and predictable execution time, but if we know \( G(s|d) \) for each alternative in the tree, we can devise methods which have higher expected utility. (We’ll discuss later how \( G(s|d) \) can be estimated.)

Optimal Stopping - Single Level

Let us start with a simplified situation, and assume that we have only one level of optimization (and thus two levels of representation: the problem specification is the root of our search tree and its children are solutions, i.e. leaves).
Since the result of one run of the optimizer does not affect the following runs, we can view the scores of successive children as being independently drawn from some fixed distribution. This allows us to apply a result from the area of statistics known as Optimal Stopping (Shiryaev 1978; Chow, Robbins, & Siegmund 1971).

Suppose we generate \( n \) children from the root, and that they have scores \( s_1 \ldots s_n \). The utility of this process is then the value of the best child we find minus the cost of doing \( n \) runs: \( V(\min_{1 \leq i \leq n} s_i) - c_0 \cdot n \). In the single-level case our only decision is whether to stop and return the best child so far as our result, or to generate another child. When should we stop?

Note that generating another child only improves our utility if that child is better than the best so far. Let \( s_b \) be the best score we’ve seen so far. The lower (better) \( s_b \) is, the less likely a new child is to be an improvement, and the smaller each improvement will be. Thus, the average benefit from generating a new child decreases as \( s_b \) gets better. But the cost of generating a child is constant, so as \( s_b \) decreases, the net incremental utility of generating one more child also decreases. It can be shown that we get the highest overall utility for the design process if we keep generating children until we find a score so good that the incremental utility of generating another child is negative.

If we know the Child Score Distribution \( G \), the value function \( V \), and the cost \( c \) we can calculate the incremental utility for a given \( s_b \). So, the first improvement for a single-level search is to not generate a fixed number of children at each level, but rather to generate children until we stop and return the best child so far as our result, or to generate another child. When should we stop?

Note that generating another child only improves our utility if that child is better than the best so far. Let \( s_b \) be the best score we’ve seen so far. The lower (better) \( s_b \) is, the less likely a new child is to be an improvement, and the smaller each improvement will be. Thus, the average benefit from generating a new child decreases as \( s_b \) gets better. But the cost of generating a child is constant, so as \( s_b \) decreases, the net incremental utility of generating one more child also decreases. It can be shown that we get the highest overall utility for the design process if we keep generating children until we find a score so good that the incremental utility of generating another child is negative.

If we know the Child Score Distribution \( G \), the value function \( V \), and the cost \( c \) we can calculate the incremental utility for a given \( s_b \). So, the first improvement for a single-level search is to not generate a fixed number of children at each level, but rather to generate children until we stop and return the best child so far as our result, or to generate another child. When should we stop?

Waterfall with Optimal Stopping

Now let us consider extending these results to multiple levels. If we had \( G \), \( V \), and \( c \) at each level we could treat each level as a separate single level problem and apply the single level method above: we would start at the root, generating children from it as if this were a single level problem, until the single level method said to stop. Then we would take the best child we had generated and use it as the parent in similar process at the next level down, and so on.

In fact, we assume that we do have \( G \) and \( c \). The value of \( c \) can be estimated empirically if need be, and we will discuss how we get \( G \) in the next subsection. Also, we assume that \( V_0 \), the value of a final result, is provided by the user. However, this still leaves us in need of \( V \) for the other levels. What is the value of a design alternative such as a slicing tree, which is not at the lowest level?

An alternative at any level above the lowest has no value in and of itself. Its only value comes from the fact that it can be used to generate children, grandchildren, etc. In other words, the value of a non-ground design alternative is the utility of searching under it.

Let us suppose we have a function \( G(s_c|s_p) \) that estimates the probability that the score of a child is \( s_c \) given that the score of its parent is \( s_p \). (We will see in the next subsection how we can estimate \( G(s_c|s_p) \).) Then we can calculate the value of an alternative at level \( i \) with score \( s_p \) as \( V_i(s_p) = U_i(G(s_c|s_p), V_{i-1}, c_{i-1}) \). So if \( V_0 \) is given by the user we can calculate \( V_1, V_2 \), and so on. If we do this, it turns out that, for instance, the utility we calculate for a netlist is not just a utility for generating slicing trees from that netlist, it is also the utility of the whole search under the netlist all the way down to the placement level, assuming we use Waterfall with Optimal Stopping as our search algorithm.

Estimating and Updating \( G \)

To explain the next improvement in the search method, we need to describe how we estimate the Child Score Distributions, \( G \), for the design alternatives. We estimate \( G(s|d) \) in a two stage process. First we produce a “prior” estimate based on the score of \( d \) itself. Then, as we generate children from \( d \) and see their scores we revise the prior estimate, thus producing a sequence of posterior estimates.

For all our estimates, we model \( G \) as coming from some parameterized family of distributions. In most of our work we have used a very simple family which we call the Triangle Distributions. A triangle distribution is defined by 3 positive real parameters, \( l \), \( m \), and \( r \). The distribution has its highest probability at a score \( m \) and falls off linearly on either side, reaching 0 at scores of \( m-l \) and \( m+r \). (See Figure 1.) However, since we never know with certainty exactly what values \( l \), \( m \), and \( r \) have for any given design, we represent our uncertainty by giving a probability distribution \( H(l, m, r) \) over possible sets of parameter values. \( G \) is thus represented as a mixture of triangle distributions, weighted by \( H \).

Our apriori estimates of \( H \) assign independent normal distributions to each parameter, with the mean and standard deviation given by linear functions of the parent’s score. The coefficients of these linear functions were determined by fitting to data we derived from taking 20 netlists, generating 20 slicing trees from each, and 20 placements from each slicing tree. Note that this gives a substantial startup cost for applying these methods in a new domain, but that this cost can be amortized over all subsequent use of the method in that domain. Also note that this cost grows linearly with the number of levels. If there had been a level below placements, we would have needed 20 designs at this level from each of 20 placements per netlist, not from each of 400 placements per netlist.
Our estimates of $H$ are revised based on the actual child scores as we produce them, using standard Bayesian methods. It turns out that if two alternatives are siblings, the revisions we make to their $H$s as we generated their children are highly correlated. So when we revise the posterior $H$ of an alternative, we also revise the apriori $H$s of its siblings.

In practice, then, we never use Waterfall with Optimal Stopping, but rather Waterfall with Updating and Pseudo-Optimal Stopping, which is the following algorithm:

- First, calculate $V_i(s_p)$ as in the previous section, for $i = 1, 2, \ldots$: $G(s_i|s_p)$ at each level is determined by imagining a design alternative with score $s_p$, and calculating the apriori $H$ we would assign based on this score. This $H$ is used to determine the $G$, and $G$ is used together with $c_{i-1}$ and $V_{i-1}$ to calculate the utility that searching under such a design alternative would have. $V_i(s_p)$ is just this utility.
- Then, start with the root (the problem specifications) as the current parent to generate children from, and work downwards level by level. At each level, do the following:
  - Initialize $H$, and thus $G$, from $S(p)$, where $p$ is the current parent.
  - Repeat the following until the optimal stopping criteria apply, that is, until the incremental utility of generating one more child at this level is negative.
    - Generate a child from the current parent
    - Update $H$ and thus $G$ for the parent based on the new child’s score.
  - Choose the best child at the current level as the parent for the next level.

We use the term "pseudo-optimal" here because once we introduce updating, the optimal stopping rule is no longer optimal. However, as we will see, our empirical experience is that it is a useful heuristic.

Waterfall with Updating, Pseudo-Optimal Stopping, and Parent Changing

Now, as we generate children from a parent and update its utility, it often happens that its utility is reduced, reduced so much that this parent no longer has the highest utility among its siblings. In this case it makes sense to switch parents. That is, when working at a given level it makes sense to let the parent at each step be the alternative at that level that currently looks best, i.e., has the highest estimated utility. Doing so gives us Waterfall with Updating, Pseudo-Optimal Stopping, and Parent Changing.

Highest Utility First Search (HUFS)

There is one final step to take to convert Waterfall with Updating, Pseudo-Optimal Stopping, and Parent Changing into Highest Utility First Search. Notice that, even though the score functions $S_I$ and $S_j$ at levels $i$ and $j \neq i$ cannot be compared, the utility estimate $U^S(d)$ is comparable across all levels of the tree: the utility of a netlist is the expected utility of searching under it for a placement, as is the utility of a slicing tree. The utility of a placement is just its value, since no search cost needs to be paid to turn a placement into a placement.

Since all utilities are comparable, there is no reason to confine ourselves to the current level when we are looking for the best parent to generate the next child from. We should look at all design alternatives at all levels, and choose the one with the highest utility. If the alternative with the highest utility is at the lowest level, this means that the action of returning it as the answer right now, with no further computing, has higher utility than searching under anything at any higher level, so that is what we should do, stop and return it.

This amounts to Best First Search where best is determined by the utility estimates, hence our naming it Highest Utility First Search. Note that the utility estimate is still based on Waterfall with Pseudo-Optimal Stopping (but without Parent Changing); it does not reflect the actual search algorithm. We are looking at this issue as part of our current work.

Summary

We started with a single level search and precise knowledge of $G$, and used the stopping rule from the statistics literature which is optimal for this case. We extended it to multiple levels (which the literature does not do) to get Waterfall with Optimal Stopping. We then moved to the actual situation where we have only imprecise knowledge of $G$, and represented that knowledge by a probability distribution on $G$’s parameters. This led to the need to update $G$ after we generate each child and see its score. This updating may give the parent a worse score than one of its siblings, so the next modification was that when we generated a child we used the parent with the highest utility among the current parent and its siblings. Finally, we saw that the utility estimate was a heuristic estimation of how good each alternative
in the tree was and could be compared across levels, giving us best first search where best means highest utility, i.e. HUFS.

**Empirical Test**

To test HUFS we ran it and waterfall on a set of 19 random netlists, which did not include any of the netlists we used for calibration. To save time, the tests were run on pre-generated data. For each netlist we generated 50 slicing trees, and for each of these 50 trees we generated 100 placements. When we ran HUFS or waterfall with this data, instead of calling the optimizer to generate a slicing tree we chose randomly (with replacement) one of the trees we had pre-generated for this netlist, and similarly for generating a placement from a tree.

Using this test data, we tested HUFS for each of 4 different settings of $c$, the cost per run of the optimizer: 1600, 3200, 6400 and 12800.

Our measure of merit was how many optimizer runs one had to do to have a 95% probability of achieving a given total score, summed over the 19 netlists. Figure 2 shows how this number of runs grows as the given score decreases for HUFS and for waterfall.

Table 1 presents the same data. The column labeled “HUFS/WF” is the ratio of the number of optimizer runs taken by HUFS to those taken by Waterfall for the same score.

As can be seen from the data, HUFS produces an equivalent quality design using 30% to 50% of the optimizer runs compared to waterfall. These results demonstrate that, at least for this particular problem, HUFS is a significant improvement over waterfall. Furthermore, HUFS did this well even though we modeled the score distributions as “triangle” distributions, which did not correspond very closely to the actual distributions, and we used few enough optimizer runs in the calibration phase that calibration was very feasible.

**Ongoing Work**

Up to this point we have summarized the results in (Steinberg, Hall, & Davison 1998). This section will discuss our ongoing work. Results since that paper results fall into two main areas: we have generalized our model of utility and we have applied HUFS to improve a genetic algorithm based optimizer. We will discuss these areas in the following two subsections.

**Generalizing Utility**

As we mentioned earlier, it is common in the literature to model the utility of some process as an intrinsic utility of the result minus a time cost that depends on when the result is available. Specifically, cost is often assumed to be linear with time. While this model is commonly used, it has two drawbacks for our use in this work. The first is that it requires expressing time cost and value of a resulting design in the same units. It is not at all clear that a designer can readily say, for instance, how many seconds of CPU time it is worth taking to reduce the area of a chip by one square micron.

The second drawback is that this model of utility makes no provision for a deadline by which we must have the answer. If it is worth a minute of CPU time to reduce circuit area from 1,000,000 square microns to 999,999 it is worth 511,200 CPU minutes (one CPU year) to reduce the area to 488,800 square microns. There is no way to say, for instance, that the chip has to be ready for production in 3 months or, no matter how good it is, our competitors will capture most of the market.

Because of these problems, we have extended our methods to handle a more general form of utility. We allow the user to provide a function $V_0(s, t)$ which gives the value of having a ground-level design as a function of both the designs score, $s$, and the time, $t$, at which it becomes available, subject to the following constraints:

- $V_0$ must be monotonic non-increasing in $s$ and $t$. That is, it is never better to get a larger score or to take more time.

- There must be some deadline $t_{max}$ such that $V_0(s, t) = 0$ for all $t > t_{max}$.

- For a given score, the loss in value per unit time must not decrease. That is, $V_0(s, t + 1) - V_0(s, t)$ must not get smaller as $t$ grows.

Under these conditions the basic optimal stopping result, that stopping when incremental utility is $< 0$
is optimal, still holds, and we can calculate the overall utility for a search using Waterfall with Optimal Stopping, assuming a precisely known $G$, as we did with the previous formulation of utility as intrinsic value minus linear time cost.

Our initial experiments with this formulation for utility have focused on the situation where we have some specified score $s_g$ that we wish to achieve, and the goal of the design process is to maximize the probability of achieving this score or better by time $t_{max}$. This is equivalent to specifying

$$v_0(s, t) = \begin{cases} 1 & \text{for } s \leq s_g \text{ and } t \leq t_{max}, \\ 0 & \text{else} \end{cases}$$

**HUFS and Genetic Algorithms**

In a Genetic Algorithm (GA) we start with a population of candidate solutions to some problem, and have operators that modify the population by adding new candidates to the population and removing old ones. There is a random element to these operators; if we run a GA several times on the same problem, it is typical to get a different result each time, and as with other stochastic optimizers it is common when using a GA on a real problem to run a few times and use the best of the results obtained.

Furthermore, we don't have to restart the GA from the beginning in order to get a different result. If we back up to any intermediate population and restart, we will still in all likelihood produce a different sequence of populations and a different final result. Thus, there is an implicit tree of populations that we can search. The children of a population are all the populations that might be produced by applying one step of the GA to the parent population. We have applied HUFS to the search of this implicit tree, and have achieved similar quality designs in a half to a third as many GA steps, compared to restarting the GA from the beginning or even to doing Waterfall on the implicit tree. Preliminary results indicate that, as in the rectangle placement problem, using HUFS lets us achieve the same quality results with half to a third the number of optimizer runs.

**Summary and Conclusions**

We have presented our work on HUFS and related topics. The main novel contributions of this work are

- We have demonstrated a new method for searching trees that are characterized by a large branching factor, the absence of a heuristic to compare nodes at different levels of the tree, and a child generator that is both expensive to run and stochastic in nature.

- The method reasons about the entire probability distribution of heuristic scores of the children of a node in a search tree.

**Acknowledgements**

The work presented here is part of the “Hypercomputing & Design” (HPCD) project; and it is supported (partly) by ARPA under contract DABT-63-93-C-0064 and by NSF under Grant Number DMI-9813194. The content of the information herein does not necessarily reflect the position of the Government and official endorsement should not be inferred.

**References**


