Highest Utility First Search Across Multiple Levels of Stochastic Design

Louis Steinberg

J. Storrs Hall*

Brian D. Davison

Department of Computer Science, Rutgers University
New Brunswick, NJ 08903
{lou,davison}@cs.rutgers.edu

Abstract

Many design problems are solved using multiple levels of abstraction, where a design at one level has combinatorially many children at the next level. A stochastic optimization method, such as simulated annealing, genetic algorithms or multi-start hill climbing, is often used in such cases to generate the children of a design. This gives rise to a search tree for the overall problem characterized by a large branching factor, objects at different levels that are hard to compare, and a child-generator that is too expensive to run more than a few times at each level. We present the Highest Utility First Search (HUFS) control algorithm for searching such trees. HUFS is based on an estimate we derive for the expected utility of starting the design process from any given design alternative, where utility reflects both the intrinsic value of the final result and the cost in computing resources it will take to get that result. We also present an empirical study applying HUFS to the problem of VLSI module placement, in which HUFS demonstrates significantly better performance than the common "waterfall" control method.

INTRODUCTION

Some parts of some problems are naturally decomposed into successive 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, maybe a dozen or two at the extreme. Thus the search space is a tree with depth of a dozen or two but with huge, branching factors. 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 and a netlist. The huge branching factor and the disparate types at different levels make it hard to apply standard tree-search algorithms such as A* or Branch-and-Bound to this search space.

Recently, a number of techniques for stochastic optimization have been shown to be useful for specific levels of such problems. These techniques include simulated annealing (Ingber 1996), genetic algorithms(Michalewicz 1996; Goldberg 1989), 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. Thus, these optimizers generate a much smaller tree of greatly enhanced quality. However, even this tree has a large branching factor (in the thousands for examples we have looked at), and the cost of generating a single descendant is so high we cannot possibly generate more than a few at each level, so there is still the problem of controlling the search within the smaller tree.

In practice, human engineers faced with a design task that has this structure often take a very simple "waterfall" approach: they work from top down, using CAD tools such as optimizers to generate only a few (often one) alternatives at a given level. They choose the best design at this level by some heuristic that compares alternatives *within* a level, and use this best design as the (only) parent from which to generate designs at the next level.

This paper presents an alternative to the waterfall control method, called "Highest Utility First Search" (HUFS). HUFS applies ideas from the decision theory(Tribus 1969) to explore the tree of alternatives in a much more flexible manner than waterfall. We will describe HUFS and present empirical data from one example design task showing that HUFS can be a significant improvement over waterfall search.

HUFS is based on the idea that a good control method is not one that finds the best design, but one that gives the best tradeoff between computation cost

[0]Current address: Institute for Molecular Manufacturing, 123 Fremont Ave, Los Altos, CA, josh@imm.org

and design quality, and that therefore control decisions should be based on an analysis of the *utility*, of each possible computation, i.e. the value of the result minus the cost of doing the computation.

In our context, we can use the notion of the utility of a computation to define the utility of a design alternative. The utility of a design alternative d, which we will refer to as $Udesign(d)$, is the expected utility of a design process that starts with d as input and returns a ground-level design from among the descendants of d. I.e., the utility of d is the expected difference between the value of the final design we will get if we use d as our starting point and the cost of the computation it will take to get this design. (This formulation of utility was taken from (Russell & Wefald 1991).)

The basic idea of HUFS is very simple:

> Find the design alternative d_{opt} with the highest *Udesign*, among all the design alternatives you currently have on all the levels, and generate one child from d_{opt}, that is, run the appropriate level's optimizer with d_{opt} as input. Repeat this process until d_{opt} is a ground-level design, then stop and return d_{opt} as the result

To do his, however, we need some way of estimating $Udesign(d)$. Below we will explain how HUFS does this and will present HUFS in more detail. We will then describe our empirical test of HUFS, and then discuss related work. In order to provide a concrete example to use in these sections, we will first describe the design problem we have used as the main testbed for our research on HUFS.

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.

The optimizer for slicing trees starts by generating a random binary tree. We define a set of neighbors as those trees that can be reached from the current tree by choosing two nodes and interchanging the subtrees rooted at those nodes. At each step of the optimizer, we generate the neighbors of the current tree in a random order until we find a neighbor that is better than the current tree. When we find a better one, we make that the current tree and repeat. If no neighbor is better, the optimizer halts.

The second stage of the placement process converts a slicing tree into a full, specific placement by choosing a set of reflections for each node in the slicing tree and then optimizing the reflections, Reflections are optimized in the same way that slicing trees are, with one set of reflections defined to be a neighbor of another if they can be reached from each other by reflecting one node of the slicing tree in one dimension.

Expected Utility and Highest Utility First Search

This section will first explain the process HUFS uses to estimate $Udesign(d)$, the expected utility of design using alternative d as the starting point. We will see that some of the information HUFS uses in this process is not directly provided by HUFS' input, so we will then explain how HUFS gets this additional information. Finally, we will give the HUFS algorithm.

Calculating $Udesign(d)$

Let us start by considering a simplified case. We will focus on a single-level problem such as that of generating full placements from a slicing tree. Also we will assume that HUFS has all of the following information:

- The *cost*, c, of the CPU time to run the optimizer once. We assume that each run has the same fixed cost.
- A heuristic *Score function* $S(f)$, where f is a full placement. S gives an estimate of how good a design f is. We assume a lower score is a better design.
- A *Utility function*, $U(s)$, that represents utility we place on having a child with score s. This utility needs to be in the same units as c. We will assume that utilities are non-negative. Since a larger utility is better, $U(s)$ is monotonic decreasing.
- The *Child-Score Distribution*, $G(s)$. This is a probability distribution. It gives the probability that if we generate a full placement its score will be s. G will

depend on which slicing tree we are using as input to the optimizer, but for the moment let us assume that we are dealing with one specific slicing tree and that we know its G.

Since for now we are considering a single level problem, there is no question of which optimizer to run or which input to give it — there is only one optimizer, and only one possible slicing tree to use as input. All that needs to be chosen is whether to do another run or to stop and declare the best design we have so far to be our final answer.

To make this choice, we consider the utility of doing one more run. If the utility of doing that one run is positive, i.e., if the utility of doing the run exceeds the cost, it clearly makes sense to do that run, and if not it makes sense to stop. We assume we know c, the cost of a run, so to determine the utility we just need to determine the utility of the run. We call this utility the *expected incremental utility*, EIU, of doing a run. It is the average expected increase in utility (based on the information we have now) from the best design we have now to the best design we will have after the run.

The better the current best score is, the less likely it is that another run will do better, so the EIU depends on the best score we have so far. If s_b is the current best score and N is a random variable representing the score we will get on the next run,

$$\begin{aligned} EIU(s_b) &= E(\max(U(s_b), U(N)) - U(s_b)) \\ &= \sum_s G(s)(\max(0, U(s) - U(s_b)) \end{aligned}$$

Where E is Expected Value and s ranges over all possible child scores. (Note that the probability that $N = s$ is just $G(s)$.) Since U is monotonic decreasing,

$$\begin{aligned} EIU(s_b) &= \sum_s G(s)\max(0, U(s) - U(s_b)) \\ &= (\sum_{s<s_b} G(s)U(s)) - U(s_b)\sum_{s<s_b} G(s) \end{aligned}$$

Note that s_b cannot increase as we do more runs, and that as s_b decreases, $EIU(s_b)$ cannot increase. Therefore, once the EIU is less than c it will stay less than c for all further runs, so the expected utility of doing any number of further runs is negative, and the rational control decision at this point is to stop.

In other words, if we define the *threshold score* s_t to be such that $EIU(s_t) = c$, then the optimal strategy for a single-level problem is to continue generating children until we get one whose score is less than s_t.

Now, given the stopping criterion, we can calculate expected values for U^f, the utility of the resulting full placement, and for C, the total cost of the optimizer runs. From these we will calculate an expected value for the utility of the overall process. Note the distinction between U^f, the utility of *having* the final design, and $U(d)$, the utility of doing the work to *compute* the design, which is $U^f - C$.

The final full placement is the first one we find whose score is less than s_t, so the Expected Value of its utility is the average over these scores of $U(s)$, weighted by the *relative* probability of each score, i.e., the probability of getting that score given that we got some score less than s_t:

$$E(U^f) = \sum_{s<s_t} U(s)G(s) / \sum_{s<s_t} G(s)$$

The chance of finding a score under s_t in one run is $\sum_{s<s_t} G(s)$ so the average number of runs to find such a score is $1/\sum_{s<s_t} G(s)$ and the Expected Value of C is

$$E(C) = c / \sum_{s<s_t} G(s)$$

The Expected Utility of the overall design process is $E(U^f - C)$ but because the scores of successive children are independent, and the cost of a single optimizer run is a constant and therefore independent of the score of the child it produces, U^f and C are independent, and $E(U^f - C) = E(U^f) - E(C)$. so

$$Udesign(d) = \frac{\sum_{s<s_t} U(s)G(s)}{\sum_{s<s_t} G(s)} - \frac{c}{\sum_{s<s_t} G(s)}$$

A curious property to note is that $EIU(s_t) = c$ implies (by algebra on the formula above for EIU) that

$$U(s_t) = \frac{\sum_{s<s_t} U(s)G(s) - c}{\sum_{s<s_t} G(s)}$$

which is just $Udesign(d)$. That is, the utility of the design process is just the utility of the threshold score s_t. Since we stop for any score better than s_t the expected utility of the score we stop at will be better than $U(s_t)$. But the expected utility of the design process is the expected utility of the design we stop with minus the expected cost of the runs, and subtracting the cost of the runs brings us exactly back to $U(s_t)$.

In general, then, the expected utility of generating a final ground-level design from a design alternative d at the next higher abstraction level is

$$Udesign(d) = U(s_t(G, U, c))$$

where $s_t(G, U, c)$ is the score such that $(\sum_{s<s_t} U(s)G(s)) - U(s_t)\sum_{s<s_t} G(s) = c$

That is, the utility of a given non-ground object is determined by its G.

In this subsection we have shown how to compute $Udesign(d)$ from G, U, and c in a single-level case. However, in general we do not know a priori what G is. Different slicing trees have different Gs, and it is not reasonable to ask that their Gs be provided as inputs to HUFS. Therefore, HUFS has to estimate the Gs for itself. In following subsections we will discuss how it does so, and then show how the approach we have discussed for a single-level problem can be extended to multiple levels. Before we cover these topics, however, we will explain the notational conventions we use in the rest of this paper.

We will number the abstraction levels from the lowest level to the highest, with ground-level designs (e.g. full placements) being at level 0. So a slicing tree is at level 1 and a netlist at level 2. We will use superscripts to denote levels. Thus for our example problem either d^{tree} or d^1 would refer to a slicing tree design alternative, while d^0 would be a full placement.

The distribution $G(s)$ depends not only on the level but on which specific parent we are generating from. We will let $G(s|d)$ be the child-score distribution for d, that is,

$$G(s|d) = P(S(d') = s | d' \text{ is a child of d})$$

We will use similar notation with other probability distributions we mention.

Estimating G

In general, we will not know the exact G for any design alternative. Instead, we make a heuristic estimate of the G. If from this estimate it appears worthwhile to generate children we do so, and as we see the scores of these children we update our estimate using a Bayesian method. This section will describe how we form our initial of the G and how we update it.

We model the Gs for a given level as all coming from some parameterized family of distributions, e.g. the family of normal distributions with parameters specifying the mean and standard deviation. We assume the family is specified as part of the input to HUFS. Given the family we can specify a particular G by a tuple r containing a value for each of the family's parameters. Thus, $r =< 10, 2 >$ might represent a normal distribution with mean 10 and standard deviation 2. Also, we define the function $R(d)$ to mean the R tuple that corresponds to $G(d|d)$, the G of design alternative d.

The user specifies the family of distributions by giving the function $G(s|r)$, where,

$$G(s|r) = P(S(child) = s | R(parent(child)) = r)$$

For example, if the family is the normal distributions, $G(s| < \mu, \sigma >) = e^{-(s-\mu)^2/(2\sigma^2)}/\sqrt{2\Pi}\sigma$.

The problem of determining $G(s|d)$ then becomes the problem of determining $R(d)$. We cannot determine exactly what $R(d)$ is, but we can estimate the probability that $R(d) = r$ for any specific r. This is equivalent to estimating the probability that $G(s|d)$ is any given member of the family of distributions. We define the probability distribution $H(r|d)$ to be the probability that $R(d) = r$. We make an initial estimate of $H(r|d)$ and use it to make our initial estimate of $G(s|d)$, then as we see scores of d's children we update our estimate of $H(r|d)$ and use it to get our updated estimate of $G(s|d)$.

Our initial estimate is based on the parent's score. Just as we assumed we have a heuristic score function, S^0, on the children, we assume we have a similar function, S^1, on the parents. In our example, S^1, i.e. $S^{tree}(t)$, is the sum of the module area from the tree and an estimate of wire area based on the number of tree edges between modules that must be connected.

So, in addition to the $G(s|r)$, the user needs to provide a function

$$H(r|s) = P(R(d) = r | S(d) = s)$$

Continuing our example, if $H(< 10, 2 > |100) = 0.1$, then a parent whose score is 100 has probability 0.1 of having a child distribution with mean 10 and standard deviation 2.

When we generate children from d, we update our estimate of $H(r|d)$ by using $H(r|S(d))$ (that is, $H(r|s)$ where $s = S(d)$) as our prior estimate and $H(r|d)$ as our posteriori estimate, and the standard Bayesian formula (Tribus 1969),

$$sP(R(d) = r | \text{child scores} = \mathrm{s_1 \ldots s_n}) = \frac{P(R(d) = r) P(\text{child scores} = \mathrm{s_1 \ldots s_n} | \mathrm{R}(d) = r)}{P(\text{child scores} = \mathrm{s_1 \ldots s_n})}$$

i.e.,

$$H(r|d) = \frac{H(r|s) \prod_{i=1}^{n} G(s_i|r)}{\int H(r'|s) \prod_{i=1}^{n} G(s_i|r') dr'}$$

Now, given our estimate of $H(r|d)$, we can estimate $G(s|d)$ as

$$G(s|d) = \int H(r|d) G(s|r) dr$$

Since $G(s|d)$ is only an estimate of the true G, and especially since it is an estimate that is changed as a result of generating children, the control strategy of stopping when $EIU(s_b) < c$ is not necessarily the optimal rational strategy. In the previous subsection, where we assumed a known, fixed G, once the EIU was less than c, further optimizer runs could never increase the EIU, so there could be no point in continuing. Here, however, even if the current G is less than c another optimizer run could change the CSD in a way that increases the EIU, and thus making it rational to continue.

One way to look at this situation is to say that doing an optimizer run gets you an improved estimate of the parent's H, that doing so has utility, and this utility may justify doing an optimizer run even when the EIU by itself does not. We would like to find a method to include this utility in our accounting, but for now it is ignored.

In summary, we can estimate a design alternative's $G(s|d)$ from its $H(r|s)$, its score, and the scores of any children we have generated. From the alternative's $G(S|d)$ we can estimate its $Udesign$. So far, however, we have assumed we had a single-level problem.

Multiple Levels: Estimating $Udesign$

Now we turn to the task of calculating $Udesign$ for a multi-level problem. If we knew U^{i-1}, S^{i-1}, c^{i-1}, and $G(s^{i-1}|d^i)$ we could apply our single level method to compute $Udesign(d^i)$:

$$Udesign(d^i) = U^{i-1}(s_t(G(s^{i-1}|d^i), U^{i-1}, c^{i-1}))$$

where $s_t(G(s|d), U, c)$ is the score such that

$$(\sum_{s<s_t} U(s)G(s|d)) - U(s_t) \sum_{s<s_t} G(s|d) = c$$

In fact, we assume we are given S^{i-1} and c^{i-1} as part of the input to HUFS. We can use the method discussed in the previous subsection to estimate $G(s^{i-1}|d^i)$ from $G(s^{i-1}|r^i)$, $H(r^i|s^i)$, and S^i, all of which we assume are provided to HUFS, and from the scores of any children of d^i we have generated.

The only difficulty is in determining U^{i-1}. Since a ground-level design is the output the user is asking the design system for, we assume the user can tell us what a such a design is worth, so we assume U^0 is supplied to HUFS. But even if we know what, e.g., a full placement is worth, how can we determine the utility of a design alternative at a higher level, e.g. a slicing tree?

In fact, in and of itself a slicing tree has no utility — it only has utility as a starting point for generating placements. Thus it makes sense to define its utility in terms of the utility of the placement we would ultimately end up with if we started with this slicing tree. Of course, we have to subtract from this utility the cost of getting from the slicing tree to that placement. But the utility of the resulting placement minus the cost of finding it is just the utility of the slicing tree. So, the utility of a slicing tree is just its *Udesign*.

Now, when calculating $U^{tree}(s)$ we may not have a specific tree whose utility is s. For instance, in calculating s_t we integrate $U(s)$ over a range of values of s. Thus, what we need is $U^{tree}(s)$, which takes a tree score, not a tree, as its argument. In other words, we need to be able to calculate the a priori utility a hypothetical tree *would* have if its score were some given s. Since we know nothing about this tree but its score, we estimate its $H(r|d)$ by just applying this level's $H(r|s)$ to the score. In general for a design alternative d^i with no children,

$$G(s^{i-1}|s^i) = \int H(r|s^i)G(s^{i-1}|r)dr$$

where s^{i-1} is the child score whose probability we are calculating, and s^i is a parent score.

From $G(s^{i-1}|s^i)$ we can calculate the utility of a d^i whose score is s:

$$U(d^i|s) = U(d^{i-1}|s_t(G(s^i - 1|s^i), U^{i-1}, c^{i-1}))$$

So we can compute U^{tree} from $H(r^{tree}|s^{tree}$, $U(d^{placement}|s)$, and $c^{placement}$. Given a netlist L, we can compute an a priori $H(r^{netlist}|L)$ for L from L's score and $H(r^{netlist}|s^{netlist}$, and update $H(r^{netlist}|L)$ from the scores of the trees we generate from L. From $H(r^{netlist}|L)$ we can compute $G(s^{tree}|L)$, and from that, from the function U^{tree}, and from the cost c^{tree} of generating a slicing tree we can compute a threshold score $s_t(L)$ for generating slicing trees from netlist L. This allows us to compute $Udesign(L) = U^{tree}(s_t(L))$.

Furthermore, our definition of U^{tree} allows us to combine the two single-level analyses (one for generating trees from netlists and one for generating placements from trees) to show that $Udesign(L)$ is not only the utility of generating trees from L, it is also the utility of the whole multi-level process of generating placements from L. (See (Steinberg, Hall, & Davison 1998) for this proof.)

Thus, in general, for level $i > 0$,

$$U^i(s) = U^{i-1}(s_t(G(s^{i-1}|s))), U^{i-1}, c^{i-1}))$$

The HUFS Algorithm

The HUFS algorithm is a best first search where "best" means "largest *Udesign*". We start with a single, top-level design alternative representing the initial problem specifications. At each step, we find the design alternative with the largest *Udesign*, generate one child from it, and compute the child's *Udesign*.

Now that the parent design alternative has a new child, we recompute the Bayesian update of the parent's $H(r|d)$ using all the child scores including this new one, and compute a revised utility, and hence a new *Udesign*, for the parent from the new $H(r|d)$. This new utility for the parent is used in turn to revise the $H(r|d)$ of *its* parent, and so on — we propagate the change in utility through all the ancestors of the new child.

Note that our formula for the utility of an alternative implicitly assumes that it and the alternatives below it will be designed with a modified waterfall search, going down level by level but using utilities to decide when to go to the next level. search. If we are using full HUFS, we should be able to get a higher quality design and/or take less time to do the design, and thus the utility of an alternative will be higher than the value of this formula. Ideally, HUFS should be adjusted to take this into account.

Empirical Evaluation

We now turn to the empirical studies we did to evaluate HUFS.

We will first discuss our implementation of HUFS on the example problem described above, then we will discuss the waterfall control method we used as our standard of comparison, and finally the tests we ran and their results.

HUFS for the Placement Problem

To implement HUFS for the placement problem, we needed the costs of the optimizers, the value function for placements, the score functions at all levels, and the $G(s|r)$ and $H(r|s)$ for each optimizer. As will be seen, we actually tested HUFS for a range of costs, although we kept the costs of the two levels equal. We rather arbitrarily set $U^{placement}(s) = 10^6 - s$. The score functions were all simple heuristics based on the information available in a design at each level. See (Steinberg, Hall, & Davison 1998) for more details.

In order to provide netlists both for calibrating the Gs and Hs and as test data for our experiments, we wrote a program to generate netlists with the modules'

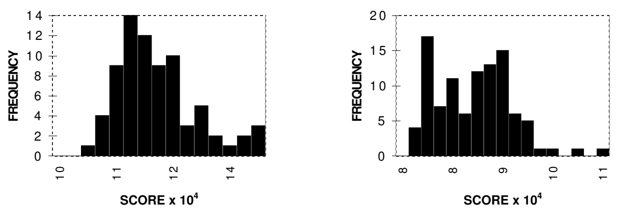

Figure 1: Child Score Distributions (G) for Two Netlists

heights and widths, the number and location of ports, and the specific interconnections all chosen randomly. All netlists in these tests had 20 modules.

To get data to determine the $G(s|r)$s and $H(r|s)$s we generated 8 netlists and ran the respective optimizers to generate 30 slicing trees each from these netlists, and 30 placements each from 8 of these slicing trees.

Figure 1 shows the actual distribution of child scores for two netlists. That is, these are the scores of the slicing trees that are the netlists' children. The distributions of child scores for slicing trees were similar. We modeled the distributions with a very simple family we call the "triangle distributions". These are piecewise linear functions, with three parameters: l, m, and r. The function is a line sloping up from 0 probability at score $m - l$ to a peak probability at score m, and then a line sloping down from there to 0 probability at score $m + r$.

From the calibration data we saw no reason not to use normal distributions for the $H(r|s)$'s, so we set the a priori probability of a parameter vector $< l, m, r >$ to

$$H(< l, m, r > |s) = Z(s, lm(s), ld) * Z(s, mm(s), md) * Z(s, rm(s), rd)$$

where $Z(s, m, d)$ is the normal distribution function with mean m and standard deviation d applied to score s. The functions $lm(s)$, etc, are linear functions of s (except for mm of the distribution of placements, which needed a quadratic function to fit the data well.) The parameters ld, etc., are constants. The values for these constants and for the coefficients of the mean functions were determined by fitting to the data we had collected. We then ran HUFS 5 times on each of the 8 netlists, and adjusted the parameters slightly (e.g., we had underestimated md). At that point we froze the parameters and proceeded to test HUFS.

HUFS is implemented in Common Lisp and takes about 15 seconds on a Sun Ultrasparc I to update the Hs after an optimizer has been run and then to choose the next design alternative to generate children from.

As a standard for comparison we used a waterfall search. This process took a netlist, generated some prespecified number of slicing trees from it, and chose the one that had the lowest score. It then generated the same number of placements from the chosen slicing tree, and chose the placement with the lowest score as its final result. We had the waterfall search generate equal numbers of children at each level because some preliminary experiments indicated that, for a given total number of children generated, the quality of the resulting designs was optimal when the ratio of children at the two levels was roughly one to one, and that the quality was quite insensitive to the precise ratio.

The 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. The setting of 1600, for instance, means that the cost of doing one additional optimizer run would be justified by an increase in the value of our final placement of 1600. Given our $V^{placement}$, this means a decrease in placement score of 1600.

For each setting of c, we ran HUFS 100 times on each netlist, and took both the average score of the 100 resulting placements and also the "95th percentile" scores — the score that was achieved or surpassed by 95 percent of the runs. We believe the 95th percentile score is a more realistic measure than the average score. An engineer normally only designs a given circuit once, so the primary measure of merit should be the quality a tool can be *counted on* to produce each time it is used. We then averaged the 95th percentile scores of the separate netlists to obtain a combined 95th percentile score for the test set, and similarly we averaged the average scores. We did not take the 95th percentile of the separate netlist scores because some netlists are inherently harder than others to place, and it was not clear how adjust for this in determining what the 95th percentile netlist was.

To compare HUFS with waterfall we started with 2 optimizer runs per waterfall trial (i.e., one per level) and did 1000 trials on each of the 19 test netlists. As with HUFS we took the average (over the 19 netlists) of the 95th percentile (over the 1000 runs for a netlist) score. We repeated this with 4 optimizer runs per waterfall (2 per level), then 6, etc., until there were enough runs that waterfall achieved the same overall score that HUFS had gotten. Finally, we re-did the tests using averages in place of 95th percentile scores.

Figure 2 plots the 95th percentile results. Successive points from left to right represent results for the successive values of c, with 1600 on the left. Note that the designs are better, and hence optimizer runs higher, to the left.

Table 1 presents the same data. The column labeled

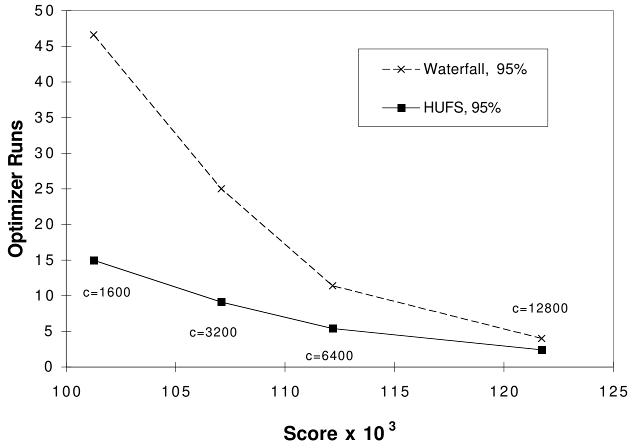

Figure 2: Optimizer runs vs. 95th percentile score for HUFS and waterfall

c	HUFS Score	HUFS Runs	Waterfall Runs	HUFS/WF Runs
1600	101252.	15.6	50.0	0.31
3200	107086.	8.8	26.0	0.34
6400	112174.	5.6	12.0	0.47
12800	121729.	2.3	6.0	0.38

Table 1: Optimizer runs vs. 95th percentile score for HUFS and waterfall

"HUFS/WF" is the ratio of the number of optimizer runs taken by HUFS to those taken by waterfall for the same score. Table 2 gives the results using averages instead of 95th percentiles.

As can be seen from the data, HUFS produces an equivalent quality design using 30% to 70% 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.

In order to bound how well HUFS could do if it had even better estimates for the score distributions, we also tried it with perfect knowledge of G.. Since the empirical tests "generated" children by selecting them

c	HUFS Score	HUFS Runs	Waterfall Runs	HUFS/WF Runs
1600	92523.	15.6	26.0	0.60
3200	96272.	8.8	12.0	0.73
6400	99978.	5.6	8.0	0.70
12800	107958.	2.3	4.0	0.57

Table 2: Optimizer runs vs. average score for HUFS and waterfall

c	HUFS Score	HUFS Runs	Waterfall Runs	HUFS/WF Runs
1600	79049	71	–	–
3200	85984	32	–	–
6400	102323	6.2	42	0.15
12800	118661.	2.4	6.0	0.4

Table 3: Optimizer runs vs. 95th percentile score for exact HUFS and waterfall

c	HUFS Score	HUFS Runs	Waterfall Runs	HUFS/WF Runs
1600	78132.	71	–	–
3200	82896.	32	–	–
6400	94230	6.2	18	0.34
12800	104510.	2.4	6	0..4

Table 4: Optimizer runs vs. average score for exact HUFS and waterfall

at random from a pre-generated list, each had an equal probability if being chosen. From this we calculated the utility of each design alternative at each level, and used these exact utilities in place of the estimates actually used by HUFS. We call this "exact HUFS".

Table 3 and 4 give the results. Waterfall reaches an asymptote at a score of about 88000 – no matter how many optimizer runs it uses, it will not match the scores achieved by exact HUFS for costs of 3200 and 1600. Of course on a real problem we would not normally be able to calculate the exact distributions, but the result of this test shows there is a potential for improving the performance of HUFS by giving it better estimates of the distributions.

It also raises new issues in the construction and selection of abstraction levels: the predictability of the child score distributions may be as important as the average score. Is there a way to find abstraction levels for which the distributions are more predictable? Or can deeper reasoning about an abstraction give us a better prediction of the distributions? Research on these questions would be useful.

Related Work

The two bodies of literature that are most relevant to our work on HUFS are the work on utility-based metareasoning by Russell and Wefald reported in (Russell & Wefald 1991) and the work on monitoring anytime algorithms by Zilberstein and colleagues. Another relevant paper is (Etzioni 1991).

The key points that Russell and Wefald make are that it is often impossible due to time constraints for a problem solver to do all computations that are relevant to the problem it is solving, and therefore it can be useful to reason explicitly about the utility of alternate computations, and to use this reasoning to guide the choice of which computations to actually do. They also note that this utility can often be expressed as the difference between an *intrinsic utility* of the solution itself and a *time cost* that accounts for the decreased in utility as delay increases. Both our focus on utility and our formulation of utility as value minus cost of computation time were inspired by this work.

Russell and Wefald also present applications of their ideas to game-tree search and to problem solving (state-space) search. However, these problems do not have the large branching factors and different types of objects at each level of the search tree, and the specific methods they use do not apply to our problem here.

Hansen and Zilberstein (Hansen & Zilberstein 1996a), (Hansen & Zilberstein 1996b) are concerned with *anytime algorithms* (Boddy & Dean 1994). An anytime algorithm is one that can be stopped after working for a variable amount of time. If it is stopped after working for a short time, it will give lower quality results than if it is stopped after working for a longer time. The single-level problem discussed above, i.e. repeated execution of one stochastic optimizer, is thus an anytime algorithm — if there is more time, more runs can be done and the average quality of the result will be better, and if there is less time fewer runs can be done and the quality will be worse.

(Hansen & Zilberstein 1996a) defines the "myopic expected value of computation" (myopic EVC) which is equivalent in our terms to $EIV - c$, and their rule for stopping, stop when myopic EVC is negative, is equivalent to our rule, stop when $EIV < c$. However, Hansen and Zilberstein are concerned with the general case of anytime algorithms (and also with the cost of the monitoring, which we do not consider), and thus do not derive any more specific formula for myopic EVC. They also do not consider multi-level systems.

(Zilberstein 1993) and (Zilberstein & Russell 1996) also define a stopping rule similar to ours and prove that, under conditions similar to those that hold in our single-level case, it is optimal.

It is worth noting that while repeated stochastic optimization can be seen as an anytime algorithm, HUFS as a whole is not an anytime algorithm. If it is stopped before any ground-level design is produced, then it gives no answer at all. It would be interesting to see if HUFS could be turned into an anytime algorithm.

Etzioni (Etzioni 1991) describes an approach to a planning problem that is quite different from our problem here, but he uses a notion called "marginal utility". Marginal utility is the incremental value divided by the incremental cost, and is analogous to our $EIV - c$ but is based on a model of utility as "return on investment" rather than our model of utility as "profit". He also includes an interesting learning component to estimate means of distributions for cost and value.

Summary

In summary, we have presented a method for searching a tree where

- the branching factor is very large,
- the nodes at different level are different types of entities, hard to compare with each other, and
- we have a method of generating random children of a node, but it is too expensive to run more than a few times per level of the tree.

as is the case with design systems that work by translating a design down through a hierarchy of abstraction levels, using stochastic optimization to generate children of a node. This search is based on optimizing the utility of the result, i.e. the value of the final design produced minus the cost of the computation time it took to produce it.

Our search control method, Highest Utility First Search (HUFS), is based on a method for estimating, for any design alternative in the tree, what the average utility of our final design will be if we start with this alternative and produce the final design from it. At each point where we must choose a design alternative to translate and optimize, we simply choose the alternative with the highest estimated utility. When this alternative is at the lowest level abstraction level, i.e. is a leaf of the tree, we stop and return this alternative as our result.

We presented HUFS and its implementation for a two-level system that solves the problem of placing circuit modules on a VLSI chip, and showed that HUFS performed significantly better than the waterfall approach of working in a strict top-down, level by level manner. See (Steinberg, Hall, & Davison 1998) for further details and discussions of such issues as the scalability of HUFS and problems with the models of value and time cost used in HUFS.

Finally, we believe the general approach of combining a utility-based analysis with statistical measures such as G shows great promise for many kinds of search problems, and we plan to explore the broader application of this approach.

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. The content of the information herein does not necessarily reflect the position of the Government and official endorsement should not be inferred.

Thanks are due to Saul Amarel, Robert Berk, Haym Hirsh, Sholmo Zilberstein, and the anonymous reviewers for for comments on earlier drafts.

References

Boddy, M., and Dean, T. 1994. Deliberation scheduling for problem solving in time-constrained environments. *Artificial Intelligence* 67:245–285.

Etzioni, O. 1991. Embedding decision-analytic control in a learning architecture. *Artificial Intelligence* 49:129–159.

Goldberg, D. E. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Reading, Mass.: Addison-Wesley.

Hansen, E., and Zilberstein, S. 1996a. Monitoring anytime algorithms. *SIGART Bulletin Special Issue on Anytime Algorithms and Deliberation Scheduling* 7(2):28–33.

Hansen, E., and Zilberstein, S. 1996b. Monitoring the progress of anytime problem-solving. In *Proceedings of the 13th National Conference on Artificial Intelligence*, 1229–1234.

Ingber, L. 1996. Adaptive simulated annealing (ASA): Lessons learned. *Control and Cybernetics* 25(1):33–54.

Michalewicz, Z. 1996. *Genetic Algorithms + Data Structures = Evolution Programs.* New York: Springer-Verlag.

Russell, S., and Wefald, E. 1991. *Do the Right Thing.* MIT Press.

Steinberg, L.; Hall, J. S.; and Davison, B. 1998. Highest utility first search: a control method for multilevel stochastic design. Technical Report HPCD-TR-59, High Perfomance Computing and Design Project, Department of Computer Science, Rutgers University, New Brunswick, NJ.

Tribus, M. 1969. *Rational Descriptions, Decisions and Designs.* New York: Pergamon Press.

Zha, G.-C.; Smith, D.; Schwabacher, M.; Rasheed, K.; Gelsey, A.; and Knight, D. 1996. High performance supersonic missile inlet design using automated optimization. In *AIAA Symposium on Multidisciplinary Analysis and Optimization '96.*

Zilberstein, S., and Russell, S. 1996. Optimal composition of real-time systems. *Artificial Intelligence* 82(1-2):181–213.

Zilberstein, S. 1993. *Operational Rationality Through Compilation of Anytime Algorithms.* Ph.D. Dissertation, University of California at Berkeley.