SteppingStone: An Empirical and Analytical Evaluation*

David Ruby and **Dennis Kibler**
Department of Information & Computer Science
University of California, Irvine
Irvine, CA 92717 U.S.A.
druby@ics.uci.edu

Abstract

Decomposing a difficult problem into simpler subproblems is a classic problem solving technique. Unfortunately, the most difficult subproblems can be as difficult, if not more difficult, than the original problem. This is not an obstacle to problem solving if the difficult subproblems recur in other problems. If the difficult subproblems recur often, then its solution need only be learned once and reused. SteppingStone is a learning problem solver that decomposes a problem into simple and difficult-but-recurring subproblems. It solves the simple subproblems with an inexpensive constrained problem solver. To solve the difficult subproblems, SteppingStone uses an unconstrained problem solver. Once it solves a difficult subproblem, it uses the solution to generate a sequence of subgoals, or steppingstones, that can be used by the constrained problem solver to solve this difficult subproblem when it occurs again. In this paper we provide analytical evidence for SteppingStone's capabilities as well as empirical results from our work with the domain of logic synthesis.

Introduction

In this paper we describe recent work with SteppingStone. In previous work with SteppingStone [Ruby and Kibler, 1989] we demonstrated its capabilities in the classic tile-sliding domain. In this paper we introduce an analytical model for its behavior. We also demonstrate its ability to operate on optimization problems with empirical results from the logic synthesis domain.

SteppingStone

SteppingStone is broken into five components: (1) goal ordering, (2) constrained search, (3) memory, (4) unconstrained search, and (5) learning. Each of these components operate largely independently. Figure 1 outlines how these components interface with each other.

*This work was partially supported by a grant from the National Science Foundation.

SteppingStone operates on problems defined with a state space representation consisting of a set of goals, a set of operators, and an initial state. The goal orderer takes as input a set of goals. It orders these goals so that the constrained search method will likely solve them. It does this by ordering them so as to reduce the likelihood of subgoal interactions using a domain independent heuristic we call *openness* [Ruby and Kibler, 1989]. It produces an ordered set of subgoals as output.

The constrained search component takes as input an ordered set of subgoals and produces a solution for the subgoals. It attempts to solve the subgoals in the prescribed order and is constrained to protect each solved subgoal. An impasse occurs when constrained search is unable to solve a subgoal. When reaching an impasse, memory is called.

Memory takes as input a context. A context consists of the subgoal currently being solved, the currently protected (or solved) subgoals, and the current state. Memory consists of a set of steppingstone records. Each of these records consist of a context and a new block of ordered subgoals. Note that the current state slot is empty for the context in a steppingstone record. The protected subgoals of the context in a steppingstone record represent those solved subgoals undone while resolving the impasse when the steppingstone record was learned. If the input context matches the context of a steppingstone record, then the block of ordered subgoals (steppingstones) contained in the record are returned to the constrained search component. For the input context to match the context of a steppingstone record, the subgoal solved by the steppingstone record must bind to the subgoal currently being solved and the protected subgoals of the steppingstone record must also be protected in the input context. When a new sequence of subgoals are returned, constrained search follows these subgoals in order to solve the current subgoal as well as all of the protected ones. If constrained search is unable to follow all of the subgoals or the final state arrived at after following all of the subgoals does not improve upon the impasse, the system reverts to

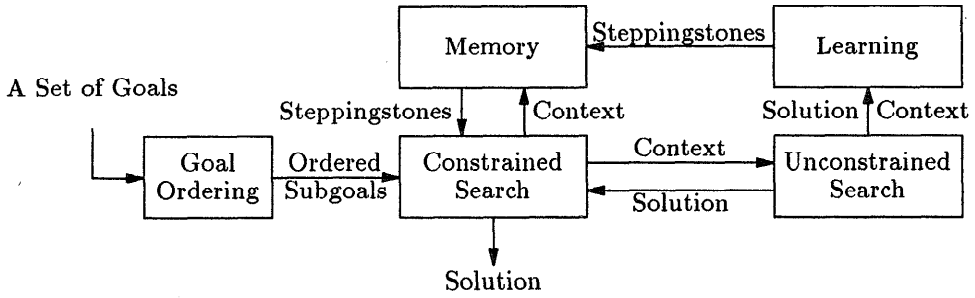

Figure 1: Overview of SteppingStone

the original impasse state.

When memory fails to return any useful steppingstones the constrained search component calls the unconstrained search component. The unconstrained search component takes as input a context, just as the memory component did. Unconstrained search relaxes the protection on the solved subgoals in its search for a solution. If it resolves the impasse, it returns the sequence of moves found to the constrained search component. The unconstrained search component also sends its impasse solution, along with the context, to the learner.

The learner takes as input a context and a solution. It uses the context to aid in generalizing over the sequence of moves making up the solution to generate a new block of ordered subgoals. These ordered subgoals are then passed to memory for reuse on other problems. We'll now examine these components in more detail.

Steppingstones as Plans

SteppingStone learns plans for solving recurring subproblems. SteppingStone represents its plans as sequences of subgoals (steppingstones) for resolving an impasse. A sequence of subgoals consists of an ordered set of partial state descriptions (subgoals). The constrained problem solver uses these subgoals as steppingstones to lead it through the impasse. These steppingstones are indexed by the subgoal they reduce and the previously solved subgoals that are undone and resolved. After following a sequence of subgoals, any previously solved subgoals remain solved and the subgoal difference generating the impasse is reduced.

Figure 2 gives an example of a sequence of subgoals for solving an impasse from the 8-puzzle domain. For one of these subgoals to be true in a state the tiles listed in the subgoal must be in the same position as they are in the state. The blank squares in these subgoals are allowed to match any tile. The subgoal sequence provides a method for correctly placing the 3-tile when the 1-tile and 2-tile have already been correctly placed. These subgoals can be followed by the constrained problem solver to lead to a state where the 1-tile, 2-tile and 3-tile are all correctly placed. Note the blank is not

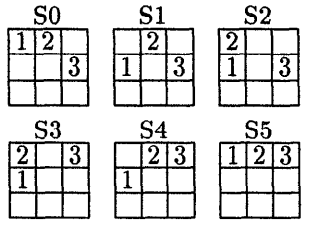

Figure 2: Steppingstones from Memory

included in any of the subgoals because it was not protected when the subgoal sequence was learned.

Steppingstones as a representation of a plan differ in several ways from macro-operators [Laird *et al.*, 1987]. With macro-operators it can be difficult to represent some types of generalizations. For example, the sequence of operations needed to move from the first subgoal in Figure 2 to the last one will be dependent upon where the blank is initially. Any reuse of a plan for this solution based on operations will also be dependent upon the position of the blank. The sequence of subgoals provided is independent of the blank.

Steppingstones allow for the use of heuristic generalization while still guaranteeing that any state generated is a legal state. This is not possible with macro-operators. If new macro-operators are ever learned that produce results not originally possible with the initial domain operators, an illegal state might be produced. This limits the types of generalizations that can be allowed with macros. Steppingstones do not have this limitation. Since they can only be used if they can be followed using the initial set of domain operators, they can be overgeneral and still guarantee that any state generated is legal.

Steppingstones pay for their increased expressiveness with a higher application cost. Steppingstones must be instantiated with search. Steppingstones demonstrate how search can be used to compensate for representation limitations of a given formulation of a domain.

Learning New Steppingstones

If the system has no knowledge concerning how to resolve an impasse, SteppingStone resorts to its unconstrained problem solver, localized brute-force search [Ruby and Kibler, 1989]. Once a sequence of moves for resolving an impasse is found, the learner generalizes it to derive a new subgoal sequence. The first step in this process is to translate the sequence of moves to the sequence of states generated by the moves. This sequence of states can be regarded as a sequence of very specific subgoals. Since these subgoals solve an impasse, we assume that only those portions of the state involved in the impasse context are relevant. Here, the impasse context includes the protected subgoals that were undone and the subgoal that was being solved. This allows the subgoals to be generalized by including in each subgoal only those portions of the state involved with the subgoal being solved and the protected subgoals that needed to be undone to solve the impasse. The removed portion of the state is variablized. For the example subgoals given in Figure 2, tile-1 and tile-2 are the portions of the state involved with previously solved subgoals. Tile-3 is the part of the state involved with the subgoal being solved. The rest of the state is variablized and allowed to match anything.

Only the moves used to resolve an impasse and the impasse context are used to generate the new steppingstones. This allows any method for finding the moves to resolve an impasse to be used to generate new steppingstones. SteppingStone can use an impasse solution provided by an expert as easily as the system uses the results of its brute-force search procedure.

Analytic Models

In this section we present an analysis of the effectiveness of the SteppingStone approach. In order to perform this analysis, we will make strong assumptions about the domain and the problem solving process. In particular we will model the problem solvers in SteppingStone as bounded breadth-first search or hill-climbing search. The goal of this analysis is to analytically evaluate the value of memory in SteppingStone. In our analysis we will compare two situations: SteppingStone with no memory and SteppingStone with a complete memory.

Breadth-First Model

In both of our models, we assume that the branching factor is the constant b. In our first model, we assume that the constrained problem solver does a breadth first search to depth d and the unconstrained problem solver does a search to depth $k * d$, where $k > 1$. We also assume that SteppingStone can solve all problems without the use of memory.

Before any learning takes place, SteppingStone will oscillate between two search processes defined by the constrained and unconstrained problem solving process. Suppose the constrained problem solver is called c times while the unconstrained one is called u times. Then the total computation cost for a memoryless SteppingStone is bounded above by:

$$(c + u) * b^d + u * b^{k*d}. \quad (1)$$

Now suppose that sufficient learning takes place such that the unconstrained problem solver need never be called. We define b_m as the number of steppingstone sequences that will match an impasse and assume that it is the same for all impasses. Since all but the last of the steppingstones that match might fail to resolve the impasse, to resolve an impasse might require trying every steppingstone that matches. The resulting search cost is bounded above by:

$$(c + u) * b^d + u * b_m * k * b^d. \quad (2)$$

Roughly speaking, the effect of memory is to replace the term b^{k*d} by the term $b_m * k * b^d$. As long as b_m is not large, this demonstrates how an appropriate memory of past difficulties can yield an exponential decrease in computation cost.

In one regard, this analysis has been too pessimistic. The factor b_m can be large without negatively affecting the search as long as the likelihood of the usefulness of a returned steppingstone sequence is high. In particular, if s is the likelihood a matched subgoal sequence from memory will succeed, then b_m should be replaced by $1/s$.

Hill-Climbing Model

A better model of the constrained problem solver used by SteppingStone is an incremental hill-climbing algorithm. This algorithm operates under the constraint that each solved subgoal must remain solved. It then attempts to solve the current subgoal by hill climbing towards it using some measure of that subgoal's completion. By incremental we mean the hill climbing occurs between subgoals and not from a state to the final goal. Note that unlike previous search-based approaches like that of Maclearn [Iba, 1989], the heuristic measure used is only over the current subgoal, not all subgoals. Both previously solved subgoals and future subgoals are ignored by the heuristic. The heuristic used is only a measure of the degree to which the single subgoal being solved is completed.

As above, let us first assume that a memoryless steppingstone is sufficient to solve a problem. Let m be the length of a solution to a subgoal from hill climbing. Also let l be the length of a solution to a subgoal by the unconstrained problem solver. In this case equation 1 becomes:

$$(c + u) * m * b + u * b^l. \quad (3)$$

Now, after learning is complete, a bound on the computational cost is:

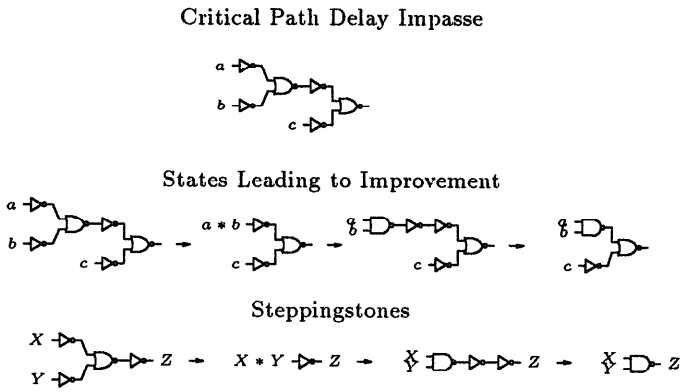

Figure 3: Steppingstones for Optimizing Critical Path

$$(c + u) * m * b + u * b_m * l * b. \quad (4)$$

As before, the factor b_m can be replaced by $1/s$ where s is the likelihood of success. In any case, the major effect is to replace the exponential factor b^l by the factor $1/s * l * b$.

Steppingstones for Optimization

Problems where the goal is to find a structure that attempts to optimize one or more parameters form an important class of difficult real-world problems. To demonstrate SteppingStone can operate effectively on problems of this type we conducted a series of experiments with the logic synthesis task of VLSI design

Logic Synthesis

One important domain that requires optimizing real-valued constraints as well as meeting a set of Boolean constraints is the synthesis of digital logic. In logic synthesis, a functional specification of a circuit is mapped into combinational logic using a library of available components. These components are taken from a technology-specific library. These libraries vary depending upon the technology and particular manufacturer chosen. The synthesized circuit is optimized for a set of constraints.

Operating SteppingStone on the logic synthesis task requires a state space representation of the problem. Logic synthesis can be represented with a start state defined by a functional description of a circuit, along with a set of constraints. Boolean algebra provides a good language for the functional description of a circuit. The goal is a realizable circuit using components from an available library that satisfies a set of hard constraints and optimizes a set of soft constraints.

Operators for this domain map parts of the functional description to components from the technology-specific library. These mappings are well-defined and ensure the correctness of the resulting design. Mapping a functional description to a realizable design is a simple task. Finding a realizable design that satisfies a set of hard and soft constraints is much more difficult. To ensure global optimality requires an exhaustive enumeration of the design space.

Figure 3 gives an example of how steppingstones for optimizing critical path delay time can be learned. The initial state to this problem is the Boolean equation $a \wedge b \wedge c$. The goal is a realizable circuit that is optimized for its critical path delay time. A realizable circuit is found by mapping Boolean subexpressions of the circuit into actual components. Assume the following components are available: inverters, 2-input nand-gates, and 2-input nor-gates. One mapping for the Boolean expression $X \wedge Y$ is a nor-gate, with $\neg X$ and $\neg Y$ for inputs. An alternative mapping is to a nand-gate with X and Y as inputs and an inverter on its output. Using mappings like these in a depth-first fashion, SteppingStone generates a circuit that is realizable, but that is unlikely to be optimal for critical path delay time.

The impasse state presents a circuit for $a \wedge b \wedge c$ that is realizable. Initially, the system has no knowledge of how to optimize a circuit, so an impasse occurs. Unconstrained search is used to find a circuit with an improved critical path delay time. The states shown are those generated by the sequence of moves leading from the impasse state to the improved state. The steppingstones are generated by removing from these states all but those portions involved in the previously solved subgoal (realizable) that were modified while generating the improved state. These final steppingstones appear at the bottom of Figure 3.

Note that the steppingstones presented in Figure 3 are goals that can match many states. The only requirement is that the variables X, Y, and Z are bound consistently in each of the subgoals in the sequence. Since steppingstones are used heuristically and only if

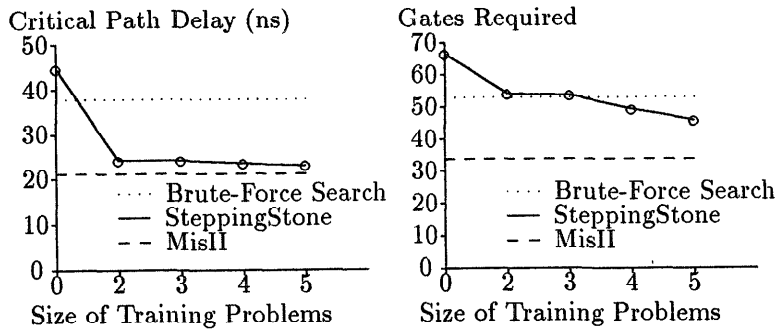

Figure 4: Average Performance on Problems of Size 30

grounded operations can achieve them, this type of generalization is sound.

Steppingstones for Logic Synthesis

To demonstrate SteppingStone's ability to learn optimization knowledge we conducted a series of experiments. A component library was created with components available from the LSI Logic Corporation. The components chosen and the critical path delay time/gates required were: 3-input *nand*=4.2ns/2 gates, 2-input *nand*=2.9ns/1 gate, 3-input *nor*=2.4ns/2 gates, 2-input *nor*=2.2ns/1 gate, *inverter*=2.9ns/1 gate.

SteppingStone was initially trained on random Boolean equations small enough for unconstrained search to produce close to optimal designs. These equations used the connectives *and*, *or*, and *not*. There are 2^{2^n} different equations of this type with n inputs. With this library of components, there are approximately three ways of implementing an *and* or *or*. Thus, for a problem of size n there are at least of order 3^{n-1} different possible solutions. This large search space makes this problem difficult for brute-force methods. The subgoals described earlier are both highly interacting and different in character from those traditionally used, making the problem difficult for goal-based approaches.

SteppingStone was trained on four successive sets of problems. Each set of problems differed in the number of inputs. The first training set had 2-input problems. The number of inputs increased until the last training set had 5-input problems. Training in a set continued until ten successive problems were solved without learning any additional knowledge. Testing was done after finishing each set of training problems. The system was tested on three sets of twenty-five random problems. These sets were drawn from problems with 10, 20, and 30 inputs respectively. Learning and unconstrained search were turned off during testing.

Figure 4 shows how the average critical path delay time of the circuits synthesized decreased as learning increased for the random problems with thirty inputs. Figure 4 also shows how the space required for the circuits decreased as well with learning. Similar results were found for the other test problems. In order to judge the difficulty of these problems and the quality of the solutions generated by SteppingStone, we used the existing logic synthesis application system MisII [Brayton *et al.*, 1987] to optimize the problems for their critical path delay time. Although, MisII had capabilities not available to SteppingStone, it served to provide a good lower bound on the solution quality. The averaged results of MisII on the test problems are also plotted in Figure 4. Note that SteppingStone almost matched the critical path delay time performance of MisII. SteppingStone did not perform as well at optimizing for the space required because it lacked opportunities for learning space optimization knowledge. Opportunities for learning this knowledge occurred only in those parts of the circuit off of the critical path. In the small training problems, few opportunities for space optimization occurred off of the critical path.

To further estimate the difficulty of these problems a simple brute-force approach was also tried. The best solution found using the brute-force approach with a cutoff of 500,000 search tree nodes was recorded for each of the test problems. The averaged results are also plotted in Figure 4.

After training on all four training sets, 34 subgoal sequences were learned. Given that the number of random Boolean functions with n inputs is 2^{2^n}, or 2^{32} for problems of size five, the amount of learning is extremely small. As with the tile-sliding domain [Ruby and Kibler, 1989], this small amount of learning is due to SteppingStone's decision to learn only when its constrained problem solver is unsuccessful and the recurrence of these subproblems. After learning these 34 steppingstones, the amount of search required to find the solutions to the problems with thirty inputs averaged 2,841 nodes expanded.

Comparison of Analytic Models and Empirical Results

To validate our analytic models of SteppingStone we used them to analyze the empirical results in the logic synthesis domain. Some adjustments of the general model will be made to better fit some specific characteristics of the logic synthesis domain. In addition, because the hill-climbing model best matches the approach used in our current implementation, we use it for the analyses.

For the random problems with thirty inputs the average length of a solution was 94 moves. The average number of subgoals solved by the constrained problem solver without generating an impasse, defined as c in our models, was 1. The average length of the solution to these subgoals, m in our hill-climbing model, was 41.9 moves.

The branching factor for the random problems with thirty inputs was computed indirectly from the total amount of search required by the unconstrained problem solver to find an impasse solution. The branching factor, b, when searching for critical path optimizations on the thirty input problems averaged approximately 7.

The average number of times memory was used when solving a problem was 21.7. This corresponds to the average number of subgoals per problem that unconstrained search would have to solve, u in our models. The average length of a solution to a subgoal found using memory was 2.4 moves. This corresponds to the average length of a solution that before learning must be found by unconstrained search. Unfortunately, the length of these solutions varied from 1 move to 9 moves. Because the amount of search before learning is exponential in the length of the subgoal solution, performance before learning is dominated by the cost of finding the longest solution. With the longest solution being 9 moves and a branching factor of 7, the amount of search required would be 7^9, or in excess of 40,000,000 nodes. With a search cutoff below this our model predicts the solutions found will be of lower quality. We conducted an experiment with an empty memory and a search cutoff for unconstrained search of 30,000 nodes and, as predicted by our model, the quality of the solutions found was not as high as that produced after learning.

After learning, the model assumes that the constrained problem solver must search about as far when failing on a subgoal as when it succeeds. For the logic synthesis domain this was not the case, as failure occurred with no search since constrained search could not hill climb from an impasse state. Thus, we can replace $(c + u)$ in equation 4 by c. We also replace the branching factor of memory, b_m, by the better estimate of $1/s$ where s is the probability that a subgoal sequence returned from memory will succeed. Thus, the amount of work after learning previously modeled by equation 4 is better modeled by:

$$c * m * b + u * 1/s * l * b \qquad (5)$$

For the steppingstones learned, the average success rate, s, on the random problems with thirty inputs was 0.0594, so $1/s$ is 16.8. For l we use the average length of a solution to a subgoal found using memory, 2.4 moves. Thus, the amount of work predicted by the model after learning is 6,418. The actual average amount of search after learning was 2,841. Given the assumptions of the model, the accuracy of its predictions greatly increase our confidence in it.

Summary

SteppingStone gains its power through the integration of several techniques. It decomposes a problem into subproblems and solves the simple subproblems with an inexpensive constrained problem solver. The more difficult subproblems are initially solved with an expensive unconstrained problem solver. The solutions to these more difficult problems are used to learn further decompositions. These new learned decompositions break the difficult subproblems into simpler subproblems that can be solved by the constrained problem solver. We provided analytical results indicating that a significant decrease in the amount of search required can be expected from this type of learning. We provided empirical evidence from a difficult real-world domain that the search required for problem solving was significantly reduced after learning. In addition, we demonstrated that the analytical model successfully predicted the general results. We intend to continue to explore SteppingStone's capabilities through a combination of empirical and analytical methods.

References

[Brayton *et al.*, 1987] B. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. Mis: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design*, 6:1062–1081, 1987.

[Iba, 1989] G. A. Iba. A heuristic approach to the discovery of macro-operators. *Machine Learning*, 3:285–317, 1989.

[Laird *et al.*, 1987] J. Laird, A. Newell, and P. S. Rosenbloom. SOAR: An architecture for general intelligence. *Artificial Intelligence*, 33:1–64, 1987.

[Ruby and Kibler, 1989] D. Ruby and D. Kibler. Learning subgoal sequences for planning. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 609–614, Detroit, MI, 1989.