

Iterative Strengthening: An Algorithm for Generating Anytime Optimal Plans*

Randall J. Calistri-Yeh
ORA
301 Dates Dr.
Ithaca, NY 14850-1326
calistri@oracorp.com

Abstract

In order to perform adequately in real-world situations, a planning system must be able to find the “best” solution while still supporting anytime behavior. We have developed a method for incrementally optimizing plans called *iterative strengthening* that can be used in many situations where other optimization methods are not appropriate. In particular, iterative strengthening supports optimized planning within an “anytime” environment using multiple simultaneous optimizing parameters, and it can be adapted to support inadmissible heuristics and undecidable domains.

1 Introduction

In order to perform adequately in real-world situations, a planning system must do more than simply generate a plan that satisfies the user’s goals. In many domains, a given problem statement may have multiple solutions, and the user typically will want the *best* solution (although the criteria for “best” may change from one user to another or one problem to another). Additionally, many domains are time-critical and require support for “anytime” behavior. In this context, an anytime algorithm is one in which a solution is incrementally refined over time; if the algorithm is run to completion it will find an optimal solution, but the user can interrupt it at any point and demand a useful (but not necessarily optimal) solution.

We have developed an algorithm called *iterative strengthening*, a flexible method of producing opti-

mized plans where the user’s criteria for optimization may change during the planning session. Iterative strengthening has the following properties: (1) The underlying knowledge base is independent of any specific optimizing parameters. (2) The method supports multiple simultaneous optimizing parameters. (3) Users can easily switch between sets of optimizing criteria. (4) The method supports optimized planning within an “anytime” environment. (5) The method is consistent with Prolog-style inference engines.

We have implemented this method in the ALPS planning system¹ and have tested it in the domain of crisis-action transportation planning with optimality criteria such as total transport time, number of aircraft, and probability of success.

2 Iterative Strengthening

Iterative strengthening is an algorithm that can be used to search for an optimized solution in situations where there may be no control over the order of node expansion and in situations where the user may demand an answer before the optimal solution has been found. Iterative strengthening is related to the concept of *iterative deepening*, in which the system searches to a given depth in the search tree for a solution, and if none is found, the system restarts the search from the beginning with a larger depth cutoff.

The iterative strengthening algorithm first performs an unconstrained search for any satisficing solution to the planning problem. When it finds that solution, it restarts the search, but now constrains the solution to be “better” than the first solution by some “increment”, where “better” is measured by an op-

*Support for the research described herein has been provided by the Advanced Research Projects Agency through Rome Laboratory Contract Number F30602-93-C-0018. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

¹See Calistri-Yeh and Segre, “The Design of ALPS: An Adaptive Learning and Planning System”, in *Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems (AIPS-94)*, pages 207–212, June 1994.

```

begin procedure iterative-strengthening(goal, increments)
  constraints ←  $\phi$ ;
  answer ← plan(goal, constraints,  $\phi$ );
  if (answer =  $\phi$ )
    then return("No solution");
  else begin loop
    if (user-interrupt)
      then return("Best solution so far is", answer);
    constraints ← strengthen(constraints, increments,
                             answer);
    new-plan ← plan(goal, constraints, answer);
    if (new-plan =  $\phi$ )
      then return("Optimal solution is", answer);
      else answer ← new-plan;
    end loop;
end procedure.

```

Figure 1: The iterative strengthening algorithm.

timization function specified by the user and “increment” is a function applied to the optimization parameters of the current plan. For example, if the goal is to find the plan that takes the minimum time to execute, and if the system has already found a plan that takes n minutes, it will restart the search constraining the new plan to $n - \delta$, where δ is a user-defined constant. The system continues strengthening the optimization parameters until no more solutions can be found; the last solution is the optimal answer.²

Figure 1 shows a pseudo-code description of the iterative strengthening procedure. It relies on an underlying planner (the **plan** function) whose behavior is minimally specified: the planner must accept parameters of the goal to solve, the current optimality constraints, and the most recent solution to the goal, and must return a solution to the goal that does not violate the constraints if such a solution exists.

Although iterative strengthening may take longer to find the final optimized plan than an algorithm such as A*, iterative strengthening has the advantage that it can be interrupted at any time after the initial plan is found and will always have a valid plan available for the user. Since this initial plan is found using satisficing criteria instead of optimizing criteria, it is likely that iterative strengthening will generate a valid plan significantly faster than algorithms such as A*. In other words, iterative strengthening supports incremental improvements to existing valid plans; it can deliver an initial plan promptly and then spend any remaining time improving it until an optimal plan is discovered or until the available planning time is exhausted.

²Technically, the last solution is optimal *modulo* δ . All plans with values in $[n - \delta, n)$ are considered equivalent, and the first such plan located is returned.

3 Flexibility of Optimality Criteria

One of our goals was to make iterative strengthening as flexible as possible regarding optimization criteria; in particular, we did *not* want to require domain engineers to write entirely separate sets of planning rules for each type of optimization. We have accomplished this flexibility by using two runtime-configurable hooks:

- **opt-eval** specifies a function that evaluates the optimization function for a partial plan. It takes as parameters the current values of the optimization parameters and the current partial plan.
- **strengthen** specifies a function that calculates the new optimization parameters during the next iteration of the iterative strengthening function. It takes as parameters the current values of the optimization parameters, the increments to apply to those parameters, and the last successful plan.

Using these hooks, it is possible to write planning rules for generic optimality functions. Typically, the underlying planner will call the **opt-eval** function every time the current plan has been extended; if the extended plan exceeds the optimization parameters, the planner can backtrack immediately. Similarly, each time a complete plan has been found, the iterative strengthening module itself invokes the **strengthen** function to further constrain the search parameters for the next iteration.

To optimize on a different set of optimality criteria, it is necessary to change only these two hooks to point to different functions. The underlying knowledge base and set of planning rules need not change at all.

4 The ALPS Planning System

The goal of the ALPS project is to design and prototype a next-generation, distributed, real-time, AI planning architecture. The current application focus of the ALPS project is crisis-action transportation planning and scheduling. Within this domain, ALPS uses the iterative strengthening algorithm to generate optimal plans using a variety of criteria such as total transport time, number of aircraft, and probability of success. The particular implementation of iterative strengthening used by ALPS can be described by specifying the definitions of the two hooks from Section 3. The **opt-eval** function recalculates the parameters from the last successful plan (ignoring the current optimization parameters) and applies the increments to

those updated parameters. The `plan` function invokes the ALPS inference engine (essentially a resolution theorem prover enhanced with various speedup techniques) on the top-level goal.³

5 Node Expansion Requirements

Planners that are designed to produce optimal solutions typically implement some form of *best-first search*, often based on an algorithm such as A*. This method has the property that if suitable heuristic functions are used, the first complete plan found is guaranteed to be the best. In contrast, planners whose underlying inference engines are resolution theorem provers almost always focus on *satisficing* solutions rather than *optimizing* solutions. Rule selection, unification, and backtracking all occur in a fixed order, and only the first solution generated is of interest.

One desirable property of iterative strengthening is that it does not depend on the order of node expansion. The underlying planner is free to expand the search space any way it wants to. This property makes optimal planning available to a wide variety of planning architectures, including our ALPS system, that otherwise would not be able to address optimization issues.

6 Admissibility Requirements

As with the A* search algorithm, the basic version of iterative strengthening requires an *admissible* search heuristic: the heuristic must be *monotonic* (a maximizing heuristic must never increase) and *optimistic* (a maximizing heuristic must never underestimate the final value). The effect of these two requirements is that as soon as a partial plan violates the current optimization cutoff, the entire subtree rooted at that partial plan can be pruned because all possible extensions of the partial plan are guaranteed to violate the cutoff.

In many circumstances, iterative strengthening's optimization criterion can be used directly as an admissible search heuristic. For example, if we want to minimize the number of aircraft used in a transportation plan, we can simply use the number of currently allocated aircraft as an admissible heuristic. Unfortunately, not all optimizing functions can be translated

³ALPS does not directly use the previous answer to guide the search for the next answer; however, since the inference engine is able to preserve the state of the search space from one invocation to the next, ALPS will restrict its efforts to that part of the search space not yet explored.

directly into an admissible search heuristic. For example, if we wanted to *maximize* the number of aircraft instead of minimizing them, we cannot use the number of aircraft currently allocated for the search heuristic because it is neither monotonic⁴ nor optimistic. These two examples illustrate a fundamental difference in complexity between searching for minimizing and maximizing solutions: an admissible search function allows us to prune the search space as soon as the current optimal plan is exceeded, while with inadmissible functions we must continue to search until the search space is exhausted.

Even though it may be much more expensive to search for an optimal plan using inadmissible optimization criteria, there may be situations where it is necessary. The iterative strengthening algorithm can be easily extended to support search for inadmissible heuristics, but at a substantial runtime penalty. The basic concept of iterative strengthening remains the same: the system finds the first solution with an unconstrained optimization parameter. It then strengthens the constraints on the optimization parameter based on the `strengthen` function. But when the underlying planner searches for subsequent solutions, it will test the constraint values only after it has found a candidate plan, rather than use the constraints as a threshold cutoff to force backtracking as soon as a search path exceeds the optimization parameter.

End users should choose their heuristics carefully and must be prepared for significant performance penalties if they select inadmissible heuristics. On the more positive side, most other optimizing algorithms cannot use inadmissible heuristics *at all*, and those that can will necessarily be forced to pay the same performance penalty since it is inherent in the problem. And even more encouragingly, the admissibility of the optimality heuristic has absolutely no impact on finding the first satisficing solution, so iterative strengthening can still be used effectively as an anytime algorithm even with inadmissible heuristics.

7 Decidability Requirements

In order to guarantee an optimal solution, iterative strengthening requires that the domain theory and the underlying planner are *decidable*: given any query in the domain language, the planner must either return a valid plan or report failure. Decidability is required because the final step of the iterative strengthening algorithm involves a search to determine that there

⁴It is monotonically *increasing* for a *maximizing* function, which is not allowed.

are no superior plans. Unfortunately, many domains are undecidable with resolution theorem provers.

There are two ways to get around the decidability requirement. One is to sacrifice completeness by enforcing a limit on each iteration of plan improvement based on how long it took to find the current plan. If the search exceeds the limit, the planner will report failure and return the last successful plan as the optimal answer (even though there may have been a better plan that was not found).

The second method is even simpler: since iterative strengthening is designed to be an anytime algorithm, termination conditions may not be terribly important. The user can interrupt the system at any time and demand the best answer so far; eventually, the user will get tired of waiting and decide that the current answer is good enough.

Neither of these methods is particularly satisfying, since the user will never know whether the answer is *really* the best. However, as with admissibility, the inherent difficulty of the problem means that no other algorithm can expect to do any better, and these methods allow iterative strengthening to perform in situations that most other optimizing algorithms cannot handle at all.

8 Discussion

The appropriateness of iterative strengthening depends on properties of the domain, the application, and the implementation.

Although iterative strengthening can be used in any domain (possibly using the extensions above to overcome admissibility and decidability requirements), it is more efficient in some domains than in others. Specifically, iterative strengthening will perform best in domains with the following properties (listed in decreasing order of importance):

1. The solution space is sparse with respect to unique optimizing function values, relative to the granularity of the `strengthen` increment size. A consequence of this property is that the iterative strengthening algorithm will need to loop only a small number of times to progress from the first satisficing solution to the final optimal solution.
2. The optimality function is admissible. A consequence of this property is that the theorem prover can backtrack and the search space can be pruned as soon as the current optimality parameters have been exceeded.
3. The domain theory is decidable. A consequence of this property is that the iterative strengthening algorithm will terminate with an optimal answer without sacrificing completeness or correctness.
4. Changes to the optimality evaluation become incrementally smaller as a plan is constructed. This means that for each iteration backtracking and pruning can occur early in the search (and hence a larger subtree can be pruned), which helps only if the optimality function is admissible.
5. The solution space is dense with respect to unique solutions. In this case, an initial satisficing plan can be found rapidly. However, note that a dense solution space will impede optimal search unless those solutions are clustered around sparse optimizing function values.

If all five of these properties hold, iterative strengthening will perform almost as well as satisficing search. At the other extreme, in the worst scenario (an inadmissible optimality function and a large number of solutions all with unique optimizing values), iterative strengthening may perform even worse than exhaustive search because it will not prune and will search several areas of the search space multiple times.

Iterative strengthening is particularly appropriate in applications where anytime behavior is desired. As discussed above, if the implementation of the underlying inference engine is such that search control is fixed and cannot be altered, then iterative strengthening may be the only feasible method. Also, if the optimality constraints are inadmissible or the domain theory is undecidable, iterative strengthening may again be the only choice.

9 Summary

The basic ideas behind iterative strengthening are not new; they are closely related to the general technique of *branch and bound*, which has been used for many years. Our contributions are to offer a particular formalization of this technique, to analyze the properties of this formalization under various situations, and to demonstrate the usefulness of this method in a specific implementation within the ALPS system. We have shown how iterative strengthening can be modified to deal with inadmissible optimality criteria and undecidable domain theories that are typically excluded by other methods, and we have discussed the tradeoffs involved in these modifications.