Synthesizing Algorithms with Performance Constraints [§]

Robert D. McCartney
Department of Computer Science
Brown University
Providence, Rhode Island 02912

Abstract

This paper describes MEDUSA, an experimental algorithm synthesizer. MEDUSA is characterized by its top-down approach, its use of cost-constraints, and its restricted number of synthesis methods. Given this model, we discuss heuristics used to keep this process from being unbounded search through the solution space. The results indicate that the performance criteria can be used effectively to help avoid combinatorial explosion. The system has synthesized a number of algorithms in its test domain (geometric intersection problems) without operator intervention.

I. Introduction

Algorithm synthesis can be defined as the translation from a functional specification of a task to an operational one, i.e. given what to do, determine how to do it. This paper describes MEDUSA, an experimental algorithm synthesizer that was developed to explore synthesis techniques. MEDUSA was designed with the following goals in mind:

- Synthesis should be done without user intervention
- Algorithms will be produced to meet some given performance constraints
- The synthesizer should be reasonably efficient, *i.e.*, it should be considerably better than exhaustive search.

Algorithm synthesis in general is very difficult; it requires large amounts of domain and design knowledge, and much of design appears to be complex manipulations and intuitive leaps. We have attempted to circumvent these problems by working with a restricted set of synthesis methods in a restricted domain. The underlying hypothesis is that a fairly restricted set of methods can be used to produce algorithms with clean design and adequate (if not optimal) performance.

The domain used to develop and test MEDUSA is planar intersection problems from computational geometry. This domain has a number of characteristics that make it a good test area:

- Nearly all objects of interest are sets, so most algorithmic tasks can be defined in terms of set primitives.
- There exist a number of tasks that are not very hard (*i.e.* linear to quadratic complexity); algorithms in this range are practical for reasonably large problems.
- Although all of the objects are ultimately point sets, most can be described by other composite structures (*e.g.* lines, planar regions), so object representation is naturally hierarchical.
- Problems in this domain are solvable by a variety of techniques, some general and some domain-specific. Choosing the proper technique from a number of possibilities is often necessary to obtain the desired performance.

The test problems (with associated performance constraints) used in developing this system are given in table 1. These problems have many similarities (minimizing the amount of domain knowledge needed), but differ enough to demand reasonable extensibility of techniques. MEDUSA is implemented in LISP. It uses and modifies a first-order predicate calculus database using the deductive database system DUCK. [7]. The database contains knowledge about specific algorithms, general design techniques, and domain knowledge, and is used as a *scratchpad* during synthesis. Useful DUCK features include data dependencies, datapools, and a convenient bi-directional LISP interface.

II. The Synthesis Process

The synthesis process is characterized by three features: it proceeds top-down, it is cost-constrained, and subtasks can only be generated in a small number of ways.

A. Top Down:

Synthesis proceeds top-down, starting from a functional description of a task and either finding a known algorithm that performs its function within the cost constraint, or generating a sequence of subtasks that is functionally equivalent; this continues until all subtasks are associated with a sequence of known algorithms (primitives). This leads quite naturally to a hierarchical structure in which the algorithm can be viewed at a number of levels of abstraction. Furthermore, it allows the synthesis process to be viewed as generation with a grammar (with the known algorithms as terminals).

[§]This work owes a lot to the continuing support, encouragement and advice of Eugene Charniak, and has been supported in part by the Office of Naval Research under grant N00014-79-C-0529

Table 1: Test problems for algorithm synthesizer.

Task	Cost Constraint
Detect intersection between 2 convex polygons	N
Report intersection between 2 convex polygons	N
Detect intersection between 2 simple polygons	$N \log N$
Report intersection between 2 simple polygons	$(N + S) \log N$
Report connected components in a set of line segments	$(N + S) \log N$
Report connected components in a set of isothetic line segments	$N \log N + S$
Report intersection among N k-sided convex polygons	$Nk \log N$
Report intersection of N half-planes	$N \log N$
Report intersection of N half-planes	N^2
Report intersection of N isothetic rectangles	N
Report intersection of N arbitrary rectangles	$N \log N$
Report the area of union of set of isothetic rectangles	$N \log N$
Report the perimeter of union of set of isothetic rectangles	$N \log N$
Report the connected components of set of isothetic rectangles	$N \log N + S$
Detect any three collinear points in a point set	N^2
Detect any three collinear points in a point set	N^3
Detect any three lines in a line set that share an intersection	N^2
Report all lines intersecting a set of vertical line segments	$N \log N$
Report all lines intersecting a set of x-sorted vertical line segments	N

B. Cost-constrained:

Synthesis is cost-constrained; included in the task specification is a performance constraint (maximum cost) that the synthesized algorithm must satisfy. We take the view that an algorithm is not known until its complexity is known with some (situation dependent) precision. We chose asymptotic time complexity on a RAM *(big-Oh)* as the cost function for ease of calculation, but some other cost measure would not change the synthesis process in a major way.

Two reasonable alternatives to using a cost-constraint that are precluded by practical considerations are having the synthesizer produce 1) optimal (or near-optimal) algorithms, or 2) the cheapest algorithm possible given its knowledge base. To produce an optimal algorithm, the system be able to deal with lower bounds, which is very difficult [1], so not amenable to automation. Producing the cheapest possible algorithm is probably equivalent to producing every possible algorithm for a task. This is at best likely to be exponential in the total number of subtasks, making it impractical for all but the shortest derivations.

C. Subtask generation:

A key function in MEDUSA is subtask generation; given a description of a task, return a sequence of subtasks that is functionally equivalent. One of the ways we simplify synthesis in this system is by using only four methods to generate subtasks.

The first method is to use an equivalent skeletal algorithm. A skeletal algorithm is an algorithm with known function, but with some parts incompletely specified (its subtasks); *e.g.* an algorithm to report all intersecting pairs in a set of objects may have as its subtask a two object intersection test. The cost of a skeletal algorithm is specified as a function of its subtask costs. These algorithms range from quite specific algorithms (*e.g.* a sort algorithm with a generic comparison function) to quite general algorithmic paradigms (*e.g.* binary divide-and-conquer). These are a convenient way to express general paradigms, and allow generalizations of known algorithms whose subtasks can be designed to exploit specific task characteristics.

The second subtask generation method is to transform the task into an equivalent task one using explicit domain information. This allows the use of logical equivalence in decomposing tasks *e.g.* the fact *A contains B if and only if B is a subset of A and their boundaries do not intersect* allows the decomposition of a containment test of two polygons into a conjunction of the tests for subset and boundary intersection.

The third subtask generation method uses case decomposition. Suppose that there is some set of disjoint cases, at least one of which is true (a disjunction). An equivalent algorithm is determine which case holds, then solve the task given that case is true. The necessary subtasks for each case is an algorithm to test whether the case holds, and an algorithm to do the original task

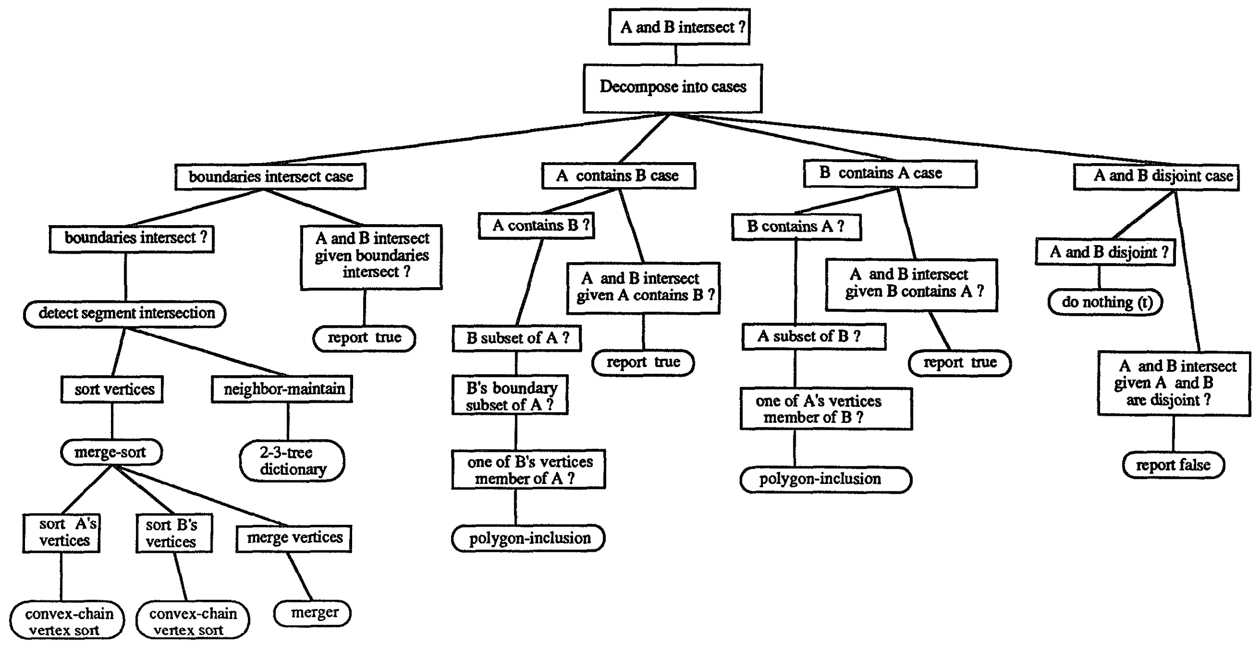

Figure 1: Synthesis of polygon intersection algorithm. Tasks are represented by rectangles, known and skeletal algorithms by ovals.

given the case holds. We restrict this by considering only disjunctions where exactly one disjunct is true (termed *oneof* disjunction by de Kleer [6]). Care must be taken to ensure that the case decomposition chosen is relevant to the task at hand.

The fourth way to generate a subtask is to use some dual transform; specifically, we transform a task and its parameters into some dual space and solve the equivalent task there. This can be a "powerful" technique [3], allowing the use of algorithms and techniques from related problems and domains. For example, suppose we want to detect whether any three points in a finite point set are collinear. Given that we have a transform that maps lines to points and *vice-versa*, and if two objects intersect in the primal if and only if they intersect in the dual, then we can recast this problem as 1) map the points in the input set to a set of lines, then 2) detect whether any three lines in this line set share an intersection.

III. Example: detect intersection of two convex polygons

The synthesis process can be illustrated with an example (shown graphically in Figure 1.): determine whether two convex polygons (A and B) intersect, time linear in the total number of vertices.

First, the task is decomposed into four cases: the boundaries intersect, A contains B, B contains A, or the polygons do not intersect. Since the cost of the task is the sum of its subtasks, each subtask has the linear time constraint. This simple propagation of the parent's constraint will hold for the rest of the subtasks in this example as well.

Working first on the *boundaries intersect* case, we synthesize an algorithm to see if the boundaries intersect. We use a skeletal algorithm, a sweep-line algorithm to detect line-segment intersection [8], which applies since a polygon boundary is a set of line-segments. It has two components (subtasks): one to sort the vertices of the segments in X-order, one to perform a dynamic neighbor-maintain on the segments in Y-order. To sort the vertices, we use a skeletal mergesort algorithm: its subtasks are 1) sort A's vertices, 2) sort B's vertices, and 3) merge the two sorted vertex sets. The two sorts are each equivalent to a known algorithm that sorts the vertices of a convex chain in linear time, the third is equivalent to the standard linear merge algorithm (with constant-time comparisons).

The dynamic neighbor-maintain is a dictionary algorithm. Set items are line-segments; they are to be put into some structure on the basis of their relative Y positions. The input to this algorithm is a linear number of queries; the queries used are insert, delete, and report-neighbors (*i.e.*, for a given segment, return the segments directly above and below it). This is equivalent to a known algorithm, a dictionary implemented with a 2-3 tree. The cost of this algorithm is the sum of its query costs, each of which is equal to the log of the working set (the excess of inserts over deletes). The working set here is bounded by the number of line segments in the two sets that intersect some vertical line; since the polygons are convex, the number of segments intersecting any line is bounded by a constant. Since the number of queries in the neighbor maintain is linear, and the working set is always constant bounded, this algorithm satisfies the linear constraint.

Detecting whether the polygons intersect given that the the boundaries intersect is equivalent to the known algorithm "report true", since boundary intersection implies intersection.

Next, we work on the *A contains B* case, first trying to get an algorithm to see if the case holds, with the additional precondition that the boundaries do not intersect (since we already tested that case, and would only reach here if it were false). By definition, A contains B if and only if B is a subset of A and their boundaries do not intersect. Since the boundary intersection is null, an equivalent task is to determine whether B is a subset of A, which is equivalent to determining whether B's boundary is a subset of A, since A and B are bounded regions. Since the boundaries are closed chains, and they do not intersect, either B's boundary is a subset of A, or B's boundary and A are disjoint. Therefore, it suffices to check whether any non-null subset of B is a subset of A, so for simplicity we use a singleton subset of B (any member) and test for inclusion in B. This is equivalent to a known polygon-point-inclusion algorithm, which is linear in the number of vertices, finishing the algorithm to detect whether A contains B. The second part of this case, determining whether the polygons intersect given that A contains B, is equivalent to "report true".

Next, we work on the *B contains A* case, with the added preconditions that the boundaries do not intersect and A does not contain B. It differs from the previous slightly, since the task *A subset of B?* is equivalent to one point of A being in B because of the added precondition that A does not contain B, but otherwise it is just the previous case with the parameters reversed.

Finally, we work on the *A and B disjoint* case, with the added preconditions that the boundaries do not intersect and that neither contains the other. These added preconditions imply that A and B are disjoint, so determining whether the case holds is equivalent to "do nothing (always true)", and determining whether the polygons intersect given that they are disjoint is equivalent to "report false".

Therefore this question can be resolved using the following sequence of operations (with cost proportional to the sum of the number of sides of the two polygons).

```
Run detect-segment-intersections algorithm using the following components:
    • Sort the polygon vertices using mergesort with components
        – sort each polygon's vertices using convex-chain-vertex sort
        – merge the two polygon's vertices.
    • Use 2-3-tree dictionary during scan.
If detect-segment-intersections returns true, report true
else Do a polygon-point-inclusion for polygon A, any vertex of B.
    if that shows intersection, report true
    else Do a polygon-point-inclusion for polygon B, any vertex of A.
        if that shows intersection, report true
        else report false.
```

IV. Synthesis mechanics

Synthesis can be represented by a single routine that takes a task and 1) generates an equivalent decomposition (subtask sequence), 2) calls itself for each subtask in its decomposition that is not completely specified, and 3) computes the cost of the task as a function of the costs in the decomposition. The cost constraints propagate forward from task to subtask, the costs percolate back from subtask to task. The important control mechanisms are those that pick from a group of possible decompositions, choose which active task to work on. It is also necessary to be able to find equivalent decompositions, propagate constraints, and combine costs.

A. Choosing among alternative decompositions

A problem inherent in synthesis is the possibility of combinatorial explosion due to multiple decomposition choices, since it is impossible in general to know *a priori* that a decomposition will lead to a solution within the constraint. If a "dead-end" is reached (no decomposition possible, or time constraint violated), some form of backtracking must be done. Unless severely limited, backtracking will lead to exponential time for synthesis. To reduce backtracking, we use a sequence of heuristics to choose the candidate most likely to succeed.

The first heuristic is to favor known equivalent algorithms to everything and skeletal algorithms to unknown algorithms (by unknown algorithms we mean any that are neither known nor skeletal)—we partition the possible decompositions into those three classes and pick from the most favored non-empty class. If a solution is known, there is no reason to look any further; similarly if a skeletal algorithm exists, it is probably worth examining since it gives a decomposition of the problem that often leads to a solution. Although any known algorithm within the constraint is adequate, we favor one of zero cost (a *no-op*) over one of constant cost over any others, since the test is cheap and it is aesthetically pleasing to avoid unnecessary work. If there is more than one skeletal or unknown algorithm left after this filtering, the choice is dictated by the second or third heuristic.

The second heuristic, which chooses among alternative skeletal algorithms, uses the time constraint as a guide to choose the algorithm most likely to succeed. For example, suppose the constraint is N log N; likely candidates are divide-and-conquer and sweep-line (or some other algorithm involving a sort preprocess). To implement this we have associated a "typical cost" with each of the skeletal algorithms, that is, the cost that the skeletal algorithm usually (or often) takes. The mechanism used is to choose the alternative whose typical cost most closely approximates the constraint. We do not choose the alternative that is typically cheapest; in fact we want the most expensive possibility within the constraint, based on the hypothesis that less efficient algorithms are typically simpler. More concretely, suppose we have as a task reporting the intersection of N half planes with alternative constraints $N \log N$ and N^2. In the first case, the decision to use divide-and-conquer based on the time constraint leads to an $N \log N$ solution; in the second, the looser constraint would lead to a different choice, to process the set sequentially, building the intersection one half-plane at a time (linear cost for each half-plane addition). The step in the sequential reduction, computing the intersection of a half-plane with the intersection of a set of half lanes in linear time is simpler than the merge step in the divide-and-conquer, which is to intersect two intersections of half-plane sets in linear time. If more than one skeletal algorithm has the same typical cost, and none has a cost closer to the constraint, the third heuristic is used to choose the best one.

The third heuristic, which is used if the others lead to no choice, is to compare all of the alternatives, choosing the one with the most specific precondition. The intuition is that a less-generally applicable algorithm is likely to be more efficient than a more generally applicable one. The *specificity* of a precondition is the size of the set of facts that are implied by the precondition; this definition gives equal weight to each fact, but is reasonably simple conceptually. We approximate this measure by only considering certain geometric predicates, which is more tractable computationally.

B. Ordering subtasks in synthesis

If all subtasks in a decomposition were independent, the order in which the subtasks were performed in the algorithm would be unimportant. This is not always the case; consider the case determination tasks in the example. The fact that the boundaries did not intersect was important to the solution of the containment determinations, and the fact that the boundaries did not intersect and neither polygon contained the other made the test for A and B disjoint trivial. In general, the testing of any conjunction, disjunction, or oneof disjunction of predicates is highly order-dependent, since each predicate test is dependent on which predicates were already tested. It may be that not all orderings lead to a solution within the time constraint, so part of the synthesis task is to determine this order of execution.

The method we use to get the subtask algorithms in the conjunctive case is the following (with analogous methods for *oneof* disjunction and disjunction):

Given a set of conjunctive tasks $c_1, c_2, \ldots, c_k$:

1. *Find and report an algorithm to test one of these (say* c_i*).*
2. *Use this method to solve conjunctive tasks* $c_1 \mid c_i,\ c_2 \mid c_i, \ldots, c_k \mid c_i$.

These algorithms will be combined in the order they were synthesized into a nesting of if-then-else's in the obvious way. (The combination of the cases in the example shows this combining for the disjunctive case.) This method is guaranteed to find an order of the subtasks if one exists without interleaving (that is, if there is a sequence such that the synthesizer could find an algorithm for each conjunct given the previous conjuncts in the sequence were true); adding a precondition to a task can only increase the number of equivalent decompositions.

The most efficient use of this method is to work depth first on the the tasks in the proper order. If the order is incorrect, a fair amount of effort may be expended on tasks that fail; in the worst case, the number of failed conjuncts is quadratic in the number of conjuncts. Working breadth first can lower the number of failures, those with longer paths than successful siblings, but since preconditions are added to active tasks whenever a sibling finishes, the partial syntheses done on these active tasks may also be wasted work. In MEDUSA, work is done primarily depth-first, but if a path too far ahead of its siblings, another path is worked on basically depth-first with some catch up to avoid long failing paths.

For this method to be reasonably efficient, the tasks must be tried in something close to the proper order. We currently use a three-level rating scheme for subtasks, top preference to simple set predicates (like null tests), lowest preference to predicates involving a desired result, and middle preference for the rest. This is the rating that led to the order of the cases in the example: the boundaries-intersect case was done first since it is a simple set predicate *(is the intersection of A and B null?)*, the A and B disjoint case was done last since it is the desired result, and the two containment tests were done second and third (with equal preference), since they are not in either of the other categories.

C. Finding equivalent decompositions

A basic function in the synthesizer is to find an algorithm equivalent to a task using one of the methods given in II.C. This is done by queries to the database, unifying variables and checking for equivalence. There is a certain amount of deductive effort involved in getting all of the equivalent decompositions, much of it on decompositions that will be filtered out by the heuristics. Our system tries to reduce this wasted effort via a "lazy fetching" approach; rather than fetching all of the equivalent decompositions, it fetches all equivalent known algorithms and sets up closures to fetch the others (skeletals and unknowns). This fits well with our known/skeletal/unknown filter heuristic explained in the previous section: if a known algorithm exists, the actual fetching of the others is never done, similarly with skeletals vs. unknowns. Since we get closures for all of the equivalent decompositions, we can always fetch them if they are needed during backtracking.

D. Propagating constraints and combining costs

Since much of the control of the system is based on costs, it is necessary to manipulate and compare cost expressions. Costs are symbolic expressions that evaluate to integers; they are arithmetic functions of algorithm costs, set cardinalities, and constants. We have an expression manipulator that can simplify expressions, propagate constraints, and compare expressions. The use of asymptotic costs simplifies the process considerably.

V. Results and planned extensions

Currently, MEDUSA will synthesize all of the problems in table one. In doing so, it uses a variety of "standard" algorithmic paradigms (generate-and-test, divide-and-conquer, sweep-line), and uses such non-trivial algorithm/data structure combinations as priority queues and segment trees. In general, the choice heuristics work effectively to pick among possible decompositions; the most common reason for failure is that the "typical cost" given for skeletal algorithms is different from the attainable cost due to specific conditions. The rating scheme for ordering dependent subtasks works adequately since usually the number of subtasks is small, but as it fails to preferentially differentiate most predicates the order is often partially incorrect. More specific comparisons are being examined.

As expected, controlling the use of duality has been difficult. The problem is that transforming the task is rather expensive (in terms of synthesis), and the possibility of one or more dual transforms exists for nearly any task. Our current solution is to only allow duality to be used as a "last resort"; subtask generation using duality is only enabled after a synthesis without duality has failed at the top level. Although this works, it has the undesirable features that

1. synthesis of an algorithm involving duality can take a long time, as it first has to fail completely, and
2. an algorithm not involving duality will always be preferred to one using duality, even if the latter is much simpler and more intuitive.

We are examining less severe control strategies to better integrate duality as a generation method.

VI. Related work

A number of researchers are examining the algorithm synthesis problem; some of the recent work (notably CYPRESS [9], DESIGNER [4], and DESIGNER-SOAR [10]) has similar goals and uses computational geometry as a test domain. MEDUSA differs most in the central role that efficiency has in its operation, and the relatively higher-level tasks that it is being tested on. It uses a more limited set of methods than DESIGNER and DESIGNER-SOAR, which both consider things like weak methods, domain examples, and efficiency analysis through symbolic execution. The use of design strategies in CYPRESS is similar to our use of skeletal algorithms, but are more general (and formal), leading to a greater deductive overhead; we chose to have a larger number of more specific strategies. In some respects our goals have been more modest, but MEDUSA was designed to automatically design algorithms in terms of its known primitives, while the others are semi-automatic and/or do partial syntheses.

This work is influenced by LIBRA[5] and PECOS[2], which interacted in the synthesis phase of the PSI automatic programming system. The primary influences were the attempt to substitute knowledge for deductions and the use of efficiency to guide the synthesis process.

References

1. Aho, A., J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.
2. Barstow, David R. "An experiment in knowledge-based automatic programming," *Artificial Intelligence* 12, pp.73-119 (1979).
3. Chazelle, Bernard, L.J. Guibas, and D.T. Lee. "The power of geometric duality," *Proc. 24th IEEE Annual Symp. on FOCS,* 217-225, (November 1983).
4. Kant, Elaine. "Understanding and automating algorithm design," *IEEE Transactions on Software Engineering,* Vol. SE-11, No. 11, 1361-1374. (November 1985).
5. Kant, Elaine. "A knowledge-based approach to using efficiency estimation in program synthesis," *Proceedings IJCAI-79,* Tokyo, Japan, 457-462 (August 1979).
6. de Kleer, Johan. "An assumption-based TMS," *Artificial Intelligence* 28, pp.127-162 (1986).
7. McDermott, Drew. *The* DUCK *manual,* Tech. Rept. 399, Department of Computer Science, Yale University, June 1985.
8. Preparata, Franco P., and Michael Ian Shamos. *Computational Geometry: An Introduction* , Springer-Verlag, 1985.
9. Smith, Douglas R. "Top-down synthesis of divide-and-conquer algorithms," *Artificial Intelligence* 27, pp. 215-218, (1985).
10. Steier, David. "Integrating multiple sources of knowledge into an automatic algorithm designer," Unpublished thesis proposal, Carnegie-Mellon University, September 1986.