Prof. Kurt VanLehn
Learning Research and Development Center
University of Pittsburgh
Pittsburgh, PA 15260

March 4, 1993

Dr. Susan Chipman
Scientific Officer Code 1142CS
Office of Naval Research
800 N. Quincy St.
Arlington, VA 22217-5000

Dear Susan,

This letter is the quarterly progress report for grant N00014-91-J-1529, entitled *Analysis of Symbolic Parameter Models*. This report covers only the work done at the University of Pittsburgh. The report period is October 1, 1992 to December 30, 1992.

Early in this report period we finally got enough bugs into Deb to that we could see the expected combinatorial explosion. We began exploring some standard techniques for reducing its search.

Because Deb is implemented in Prolog and uses its chronological backtracking technique, our first attempt was to try dependency-directed backtracking, which is usually called intelligent backtracking in the Prolog community. Although there are several version of intelligent backtracking, we discovered that none of them would reduce Deb's search. Intelligent backtracking is primarily oriented towards finding a single solution quickly by avoiding as many failure paths as possible. Although Deb goes down many "failure" paths, we need it to do that, so they are not really "failures" that should be avoided. After Deb has made many choices (each consisting of selecting a bug from a set of alternatives) and gone down a particular branch of the search tree, at the end of the branch, we collect its answer (i.e., the subtraction answer generated by that combination of bugs), then explicitly invoke backtracking by executing the Prolog "fail" command. Technically, this is a failure path, but we need Deb to go down it anyway. Deb has very few failure paths that are actually superfluous, so there is little that intelligent backtracking can do to improve the search.

We realized that we had conceptuallized the problem incorrectly. The problem was not to reduce search by pruning dead branches from the search tree, but rather to reduce search by merging nodes of the search tree that were equivalent. This is another fairly standard technique in the search literature, but it is seldom used because it is so space-intensive. Thus, we began by installing metering tools in Deb that would count the number of mergable nodes, and thus allow us to estimate how much savings could be achieved. We varied the number of bugs in Deb and plotted curves for the number of equal nodes and estimated savings in search time. All the curves were exponential, as we expected. Thus, the more bugs we have, the more valuable node merger should be.

Thus encouraged, we began implementation of a node-merging version of Deb. This turned out to be surprisingly tricky, because there is no simple definition of when two nodes in the search are equal. A node in the search consists of a parameter vector, where each parameter's value is listed (or "nil" if it has not yet been assigned a value), and a state, which represents the current state of the subtraction problem and the working memory of the subtraction problem solver. The search is such that two nodes never have the same parameter vector. The straightforward definition of node equivalence, wherein two nodes are equal if they have the same parameter vector and the same state, would mean that no nodes can be merged, so there would be no savings in the search. On the other hand, if nodes are defined as equivalent when only their states are equal, the solver can merge states that really shouldn't be merged.

Towards the end of the report period, we found a definition of node equivalence that seems to work, but we have not yet tested it thoroughly. The definition is this. Suppose we have two nodes, an old one that is the root of a fully generated search tree, and a new one. These two nodes are equal if

- their states are equal, and
- for each parameter in their parameter vectors, either
  - the values arethe same in both nodes, or

– the parameter is never accessed again in the search tree rooted at the old node.

If a new node is equivalent to an old one under this definition, then the search tree rooted at the old node is identical to the one that would be generated by expanding the new node. Thus, we can merge the two nodes, and avoid generating the search tree rooted at the new node.

We are in the process of implementing this search reduction technique, which by the way is completely general and has nothing to do with subtraction, and hope to report soon on its efficacy.

Best regards,

[illegible]

Kurt VanLehn
Assoc. Prof. of Computer Science
LRDC Senior Scientist

cc: ONR Resident Representative N66005, OSU Research Center
Director, Naval Research Laboratory, Code 2627
Defense Technical Information Center
University of Pittsburgh Contracts Office