An approach to the problem of detranslation of computer programs

R. N. Horspool and N. Marovac*

School of Computer Science, McGill University, 805 Sherbrooke Street West, Montreal, Canada, H3A 2K6

A crucial problem in the decompilation or disassembly of computer programs is the identification of executable code, i.e. the separation of instructions from data. This problem, for most computer architectures, is equivalent to the Halting Problem and is therefore unsolvable in general. The problem of identifying instructions is examined in this paper and an algorithm that will 'usually' perform a correct analysis is described. In addition, other aspects of disassembly are discussed and algorithms outlined.

(Received October 1978; revised May 1979)

1. Introduction

The general task of reconstructing a source program from its binary code (absolute binary or relocatable binary) is known as decompilation or disassembly. The term decompilation is usually used when the regenerated source code is in a high level language, whereas disassembly describes the regeneration of an assembly language source program.

Disassemblers and decompilers are useful tools for any computer installation to possess. They are invaluable when the original source code is not obtainable, either because it was not originally available or because it has been lost, and modifications to the program are required. In these circumstances, it is far easier to detranslate and modify a source language version of the program than it is to analyse and patch its binary code. We observe that detranslation can also be used to facilitate software piracy. Many software vendors distribute only compiled versions of their important programs, so that the programs are less easy to modify and therefore more difficult to transport to other computer installations. Clearly, the availability of a good detranslator reduces the efficacy of such precautions (MMS, 1978).

Detranslators can also be used as debugging tools. When a program or operating system 'crashes', very readable core memory dumps can be obtained with a disassembler. By comparing the reconstructed source code with the source code of the programs before they were executed, the programmer can spot many kinds of errors at a glance. (Ideally, the comparison of the memory contents with the original programs should be automated—leading to the so-called 'comparative dump'.)

The process of detranslating a program does not produce a unique result and, in general, is an impossible task. There could be infinitely many different source programs that compile or assemble to yield the same binary code. The differences between such 'equivalent' programs can go beyond the use of different labels and variable names. With programs, there may be several different constructs (e.g. an instruction, an integer and a floating point number) that would all translate to yield the same bit pattern in memory.

The impossibility of being able to perform a complete detranslation of a program in general is due to two causes. The first cause is inherent in the original von Neumann computer concept, upon which most modern computers are based, viz. program instructions and data are stored together in the computer's memory and have indistinguishable representations. This makes detranslation of programs, as well as reading and interpretation of program dumps, a difficult task due to the difficulty of separating instructions from data. It is, of course, unlikely that a programmer would write a floating point constant intending it to be executed as an instruction. A reasonable objective for a disassembler, therefore, is to disassemble storage items according to how those items are actually used in the course of execution. That is, items that are to be executed should be disassembled to instructions, items that are to be used as the sources of integer constants should be disassembled to integer constants, etc. However, even ignoring such programming practices as self-modifying code (where a storage location appears to be used both as a data item and as an instruction), the determination of how an item is used is difficult. If both data addresses and branch destination addresses can be computed at execution time (and instruction and data do not reside in separate address spaces), then even the separation of instructions from data alone is impossible in general. To perform this separation, we would have to be able to determine exactly which addresses can be computed and used as data operands and which addresses can be used as branch instructions. However, if we could perform this determination, we would also be able to solve the 'halting problem' (Davies, 1958). Hence the problem is obviously unsolvable in general.

A second detriment to complete and correct de-translation of object programs is a problem of semantic interpretation. Even if we know how to separate instructions from data, we are still faced with the problem of attaching the correct semantics to certain classes of instructions or data. Using PDP-11 assembler (DEC, 1979) as a source, we will illustrate the problem with an example of a common ambiguity:

First, if we have

```
012700
000200
```

appearing in the instruction sequence, it can be disassembled to either

```
    MOV    #200,R0     move constant 200 into register
                       R0
or to
    MOV    #X, R0      move address of label X (which is
                       200) into register R0.
```

This latter interpretation is possible if the location 200 happens to lie within the program bounds (it is impossible if we are disassembling relocatable binary code, because relocatable programs are not guaranteed to occupy particular memory locations). Only an analysis of the program to determine how the contents of register R0 are eventually used could ever decide which interpretation is the more plausible one.

2. The structure of a disassembler

The problem of distinguishing instructions from data is probably the most important and most difficult problem

*Present address: Department of Mathematics and Computer Science, Massachusetts State College at Framingham, Framingham, MA 01701, USA encountered in detranslating computer programs. It is crucial for both disassembly and for decompilation. With disassembly, the identification of instructions is the major problem. In the case of decompilation, however, there may be other hurdles to overcome. Various instruction sequences have to be matched against all the possible high level language constructs. This is essentially a pattern matching problem, but the answer can usually be found by an enumeration of the possibilities. The pattern matching approach has been studied by Hollander (1973). We insist, however, that the pattern matching algorithm should not be applied until there has been some separation of instructions and data. An incorrect separation could lead to nonsensical results later.

If we are decompiling, the object code has usually (but not always) been generated by a compiler for the target high level language. This compiler may enforce rigid rules as to the layout of instructions and data. When the rules are known, they should certainly be used to assist the decompilation process. However, the more sophisticated the compiler and the more optimisations that it performs, the less rigid the rules will be. Consequently, an accurate method of separating the instructions and the data is usually required. Throughout this paper, we devote most of our attention to this one problem. We will discuss it exclusively in the context of disassembling. However, we stress that we consider this problem to be directly relevant also to the decompilation process.

We will now show how a disassembler can be organised and constructed. The disassembly process can be structured into:

(*a*) separation of instructions from data

(*b*) determination of data formats and the placement of labels

(*c*) generation of the source code.

Since the last phase is a straightforward text processing task (once the analysis of the code has been completed) we will not discuss it further. In the following we describe phases (*a*) and (*b*) respectively.

2.1 *A basic algorithm for identifying instructions*

Suppose that the program to be disassembled is known to meet the following restrictions:

1. It contains no self-modifying code that changes operation codes or operands of branch instructions.
2. Branch addresses are not dynamically computed, but appear explicitly in the operand fields of branch instructions.
3. Conditional branch instructions are not used where the flow of control is, in fact, unconditional.
4. Subroutines return control to their calling points, not to their calling points plus some number of bytes (in order to skip over parameter lists).

Assuming these stringent conditions to be satisfied, we can construct a simple algorithm that traces the flow of control throughout the program. The algorithm is broadly similar to methods that have been described elsewhere (Friedman, 1974; Housel, 1973; Housel and Halstead, 1973; Sassaman, 1966).

The output of this algorithm is a map of the program to be disassembled. The map should indicate the regions of the program that are exclusively instructions. It can be implemented either as a bit map (for speed) or as a list of addresses showing where sequences begin and end (for storage economy). As a by-product of the tracing process, the algorithm can also output all the addresses to which control can be transferred (these locations should receive labels) and the locations of the branch instructions which can transfer control to those addresses. The algorithm proceeds as follows:

1. Initialise the stack S to hold the entry point address of the program; if there are several entry points, they should all be stacked. Initialise the map to show all locations holding data items.
2. If the stack S is empty then stop; the algorithm is finished.
3. Pop an address A from S.
4. If location A is already recorded in the map as holding an instruction then go back to step 2.
5. Record location A as holding an instruction.
6. Decode the instruction at location A.
7. If the instruction is a branch (conditional or unconditional) or a subroutine link, then push the destination address on to S.
8. If the instruction is a stop instruction, a subroutine return or an unconditional branch then go to step 2.
9. Increment A by the length of the instruction just analysed and go back to step 4.

If we are building the map as a list of addresses, then it should be noted that each address removed from the stack is a possible beginning of a block of instructions. Each address A that is current when step 8 transfers back to step 2 is the end of a block of code. Furthermore, the addresses that are placed on the stack are exactly those locations that should be labelled.

2.2 *Placing labels and determining data formats*

The previous phase mainly determines the instruction areas in the program to be disassembled. The major task which remains is to analyse the data areas, to attach labels to the data items in these areas and to select the most appropriate formats for disassembling data items. There are two sources of information indicating how data items should be disassembled. Firstly, the binary code of the data item may be valid only for particular formats and may be extremely unlikely for other formats. Secondly, the semantics of the instructions that access data items also determine which data formats are plausible and which are not.

The obvious first step is to make a pass through all the instructions. Each instruction should be decoded to determine the locations of its operands and which data format(s) is (are) the most plausible for those operands. Each such location should be recorded if it lies within the program, so that we can attach a label to the location later.

A final pass through the program is now sufficient to generate the source code. There are some minor difficulties that have to be resolved on this pass. First, the data operand of some instruction may fall within a sequence of instructions and thus signify self-modifying code. If the operand address does not coincide with the start of an instruction, the disassembler must use a label with an additive offset for the operand address, for example,

```
LABEL:  MOVE   #100,B       move constant 100 into B
        .
        .
        .
        INC    LABEL+2      increment the constant
```

A second problem arises if we have conflicting information as to what format a data item has. This situation can occur if a temporary variable is sometimes used to hold integer values and sometimes floating point values, say. In the absence of a sophisticated flow analysis to determine how the variable's value is first used, there is no easy solution. One compromise is to decode it as an octal constant and indicate the integer and floating point equivalents in a comment.

A third, but similar, problem occurs if we have to decode a data item about which we have no information. This can happen whenever the program computes data addresses dynamically, since the disassembler cannot, in general, deduce the possible range of data addresses. One solution is to use the same data format as for the previous data item previously decoded. This is on the supposition that we are processing the elements of an array. However, it is probably safer to assume nothing and decode the item as an octal constant again.

3. Generalising the identification of instructions

Our basic flow tracing algorithm, given in Section 2.1, required some strong restrictions to hold. We now discuss how these restrictions could be relaxed and describe a more general powerful algorithm for distinguishing instructions from data.

One severe restriction was in requiring the destination of a branch instruction to be explicitly given as the operand of that instruction. This prohibits assembler coding that might be used to obtain the effect of a 'case statement' (implemented as a jump table) or the effect of label variables. It also makes the algorithm inapplicable to programs for such machines as the IBM 360 or 370 (IBM, 1970) where addresses are given relative to a base register. Without sophisticated analysis, the disassembler cannot be sure of the contents of the base registers. Even if we just consider the PDP-11, we cannot easily handle the usual parameter passing method. The arguments for a subroutine are placed immediately after the subroutine link instruction. This implies that the subroutine must perform some kind of arithmetic on the return address. A typical coding sequence is:

```
        JSR     R5,SUBR        link to subroutine
ARG1:   .WORD   0              first parameter
ARG2:   .WORD   0              second parameter
RETN:           .              return address
                .
                .
SUBR:   MOV     (R5)+,LOC1     copy 1st parm to local
                               store
        MOV     (R5)+,LOC2     copy 2nd parm to local
                               store
                .
                .
        RTS     R5             return control
```

Of course, this particular coding style could be checked for in the dissassembler, but it could not handle all possible variations (e.g. to allow a variable number of arguments).

The obvious conclusion to be drawn is that simple scanning is not sufficient. An alternative approach is for the disassembler to perform interpretive execution of the program. In this way, absolutely reliable information about the program can be obtained. It can not only identify instructions but it can also determine the formats of data items. The drawback is that the information is usually incomplete. There is, in general, no means of forcing the interpreted program to take all possible control flow paths or to access all data items. As a simple example of why we cannot trace all control flow paths, consider the following program outline (in a mixture of high level and low level notation):

```
Read($X$);
$Y := F(X)$;
Jump to location $Y$;
```

Even if the permissible domain of input values of X were known, there is no way that the range of values for Y can be determined. It is infeasible to execute interpretively the F function for all possible values of X and it is also impossible, in general, to determine the range by analysis of F. (This is another unsolvable problem.) Without knowing the possible values of Y, we cannot locate all the places in the program that can be jumped to from the third statement.

Because of the difficulties described above, we have constructed a new algorithmic method for distinguishing instructions from data. This new method is a generalisation of flow tracing and can also use information obtained from other sources. The additional information may be obtainable from interpretive execution, program documentation or from the program's relocation dictionary (if we are disassembling relocatable binary code).

3.1 *The generalised flow tracing algorithm*

As we have already pointed out, the basic flow tracing method is inadequate. It places too restrictive conditions on the form of the assembly program. Conversely, interpretive executive cannot be relied upon to provide complete information about the program. In our view, the ideal algorithm should be related to the flow tracing algorithm, because of the simplicity of the approach. However, if additional information about the program is available (perhaps from interpretive execution), then this information should be usable. The algorithm described here meets these objectives.

It is impossible, in general, to identify all instructions and data items with perfect accuracy. Instead, our algorithm has a different objective and this objective is attainable. The algorithm searches for a self-consistent interpretation of the program that maximises the number of locations indicated as holding instructions. This alternative goal is intuitively reasonable. It forces the algorithm to locate and identify plausible instruction sequences in the middle of a data area—even when no branches to these sequences have been recognised. We do not claim that our revised goal is equivalent to the original objective of identifying instructions and data items accurately. There are some programs where our substitute goal will lead to incorrect results (we hope such programs are rare). Nor do we claim that our cost measure—the number of instructions identified—is the only plausible measure. It is simply a measure that has the correct properties. Other cost measures, such as the number of bytes of instruction area, or the number of basic blocks of code, are equally usable.

Our algorithm is designed to be as general as possible. Depending on the architecture of the computer in question, it may be possible to make simplifications. The algorithm makes use of whatever redundancies are present in the bit pattern encodings of instructions. If there were a computer that had instruction encodings with zero redundancy, then any sequence of bits could be disassembled into a valid sequence of instructions. Hence it would become almost impossible to distinguish instructions from data. Fortunately all computers known to the authors have redundant instruction encodings. Part of this redundancy is due to the existence of bit patterns that are immediately recognisable as being invalid as instructions. For example, the operation code may be invalid (unassigned or 'reserved for future use' in the manufacturer's parlance). Or, if the operation code is valid then the particular combination of operands present may be invalid (e.g. odd-numbered registers are illegal as the first operands of multiply and divide instructions on the IBM 360/370 computers).

Once we know that a particular memory location X does not hold a valid instruction, we can make inferences about the contents of other locations in the program to be disassembled. The location(s) immediately preceding X cannot hold an instruction unless that instruction would cause an unconditional control transfer. The general principle is that control should never reach an invalid instruction and we will use this principle in determining which locations hold instructions and which hold data.

Branch instruction operands must satisfy strong conditions also. If we can determine the destination(s) of a branch instruction then the destination(s) must correspond to valid instructions in the program. As a second general principle, we will assume that no program ever transfers control to a location in the middle of another instruction—even if a valid instruction happens to begin at that location. This ability to jump into the middle of an instruction is present for all computers with more than one different instruction size (e.g. 2, 4 and 6 bytes for the PDP-11 and IBM 360/370 computers). Another way of stating our principle is that no two instructions in the disassembled program may overlap in storage.

The algorithm presented below is an automatic method of scanning the code and finding instruction sequences that satisfy our two principles. It is presented in a high level form, without any regard to the amount of storage required. We discuss methods to reduce storage requirements later.

The algorithm operates with two basic data structures. These are the CONFLICTS and REQUIRES relations, both of which initially have domains that represent all locations occupied by the program under analysis. The domain of locations is modified and contracted as the algorithm proceeds. We will use the term 'node' to describe an element of this domain after the direct relationship between locations and elements of the domain has been lost. If *X* REQUIRES *Y* is true, then we are to understand that an identification of location *X* holding (the start of) an instruction would have to be accompanied by an identification of *Y* as an instruction. If *X* CONFLICTS *Y* is true, then we are to understand that *X* and *Y* cannot both be identified as holding instructions.

The algorithm

1. The two relations are initialised so that all entries are false. Then various true entries are filled by a single pass through the program as follows:
   (*a*) if a location *X* does not hold a valid instruction (e.g. an illegal operation code) then *X* CONFLICTS *X* is set true
   (*b*) if a valid instruction starting at location *X* overlaps with a valid instruction starting at *Y*, then *X* CONFLICTS *Y* is set true (The relation is clearly transitive, so that *Y* CONFLICTS *X* must also be made true)
   (*c*) if a valid instruction begins at location *X* and this instruction is not an unconditional branch instruction, then *X* REQUIRES *Y* is set true—where *Y* is the location following this instruction
   (*d*) if a valid branch instruction (either conditional or unconditional) begins at location *X* and location *Y* is known to be a destination of this instruction, then *X* REQUIRES *Y* is set true.

   Some information obtained from external sources should be incorporated into the two relations at this point. If location *X* is known to hold a data item, then *X* CONFLICTS *X* is set true. If location *X* is known to hold an instruction then EP REQUIRES *X* is set true—where EP represents the entry point of the program. If it is known that a branch instruction at location *X* jumps (even sometimes) to location *Y*, then *X* REQUIRES *Y* is set true.
2. At this point, the domain of the two relations contains all the locations in the program. The domain is now contracted in two ways: by deleting locations that cannot hold instructions and by compressing instruction loops into single nodes.
   (*a*) All cycles in the REQUIRES relation are found. That is, we find all circular chains of nodes such that each node in the chain requires the following node in the chain
   (*b*) if any two nodes in a cycle are in conflict, then all the nodes in the cycle are deleted from the problem domain. Also, if there is a node *X* that requires a node that is being deleted, then this node *X* must be deleted too
   (*c*) all cycles that are free from internal conflicts are compressed into single nodes. For example, if *A*, *B* and *C* form a cycle then *B* and *C* can be deleted. However *A* has to inherit the properties of *B* and *C*. For example, if a node *X* REQUIRES (CONFLICTS) *B* then *X* REQUIRES (CONFLICTS) *A* is set true. Similarly, if *B* REQUIRES (CONFLICTS) *X* then *A* REQUIRES (CONFLICTS) *X* is set true.
3. A transitive closure on the REQUIRES relation is performed. For all triples of nodes, *X*, *Y* and *Z*: if *X* REQUIRES *Y* and *Y* REQUIRES *Z* then *X* REQUIRES *Z* is set true.
4. For all pairs of nodes *X* and *Y*: if *X* REQUIRES *Y* and *X* CONFLICTS *Y* then *X* is deleted from the domain. (Any node that requires a deleted node can automatically be deleted—though this step would also delete such nodes if the nodes were accessed in a suitable order.)

   At this point, the REQUIRES relation can be represented as a DAG (Directed Acyclic Graph). An arbitrary node and its descendants (the nodes it requires and the nodes they require, etc.) form a tree. Each such tree represents a self-consistent subsolution to the problem. This is because all the locations represented by the tree can be identified as instructions without there being any (discernible) way for control flow to reach a data item. We now have to find a set of these trees that satisfies our algorithm's goal.
5. For all pairs of nodes *X* and *Y*: if *X* REQUIRES *Y* then *X* CONFLICTS *Y* is set true. (We are now indicating that we do not want both of the trees rooted at *X* and *Y* respectively to be in the solution set. This is simply because *Y* is a subtree of *X* and would therefore be implicitly present in any solution containing *X* anyway.)
6. Weights are computed for each node in the domain. Using our suggested cost measure, each weight is equal to the number of locations represented by the node.
7. A set of nodes is found such that the sum of weights is maximal and no two nodes are in conflict. The set is also constrained to include a node that has the program's entry point as a constituent location. In practice, very few sets of nodes that are conflict-free should be possible. However, in the worst case (when the machine's instruction set contains little redundancy), the search for a suitable set is a combinatorial problem. A branch-and-bound method to solve the problem is described in Appendix 1.

The set of nodes found in the final step represents the desired solution to the problem. Each node in the set, along with all their descendant nodes in the REQUIRES DAG, specifies one or more locations. All these locations are to be identified as holding the first bytes of instructions. All other locations that do not form the second and third (etc.) bytes of instructions are identified as being data.

The following observations can be made about the algorithm. First, if locations are initially known not to hold instructions then these locations need never form part of the problem domain. Secondly, if all branch instructions have determinable destinations then the solution found will include the solution found by the simple flow tracing method. If no branch destinations can be determined then the algorithm is equivalent to scanning the code searching for plausible instruction sequences. (A plausible sequence would have to end with an unconditional branch instruction.)

Thirdly, if a high proportion of branch destinations are determinable, it would probably be more efficient to use the basic flow tracing method first. Then the general algorithm given here could be used to augment the solution, but at less cost because much of the program would already have been identified.

Finally, we should point out that there are two programming 'tricks' that can defeat our general algorithm. The first is to place spurious conditional branch instructions in the program. Here are two examples:

```
            ADD     A,B    result known to be non-zero
            BNE     LAB    always branches to LAB
            .WORD   XX     an illegal instruction
or
            ADD     A,B    result known to be non-zero
            BEQ     LAB    never branches to LAB
                .
                .
                .
      LAB:  .WORD   XX     an illegal instruction
```

If such 'unconditional conditional branches' are interspersed throughout the program, our algorithm would be unable to recognise instruction sequences—because it would appear that control can reach data items.

The second trick is a more likely occurrence (but not considered to be 'good style'). This is the notorious self-modifying program, where operation codes are constructed before they are to be executed. A simple example is:

```
          MOV     OP,NEXT   plant op-code at NEXT
NEXT:     .WORD   XX        an illegal instruction
                  .
                  .
OP:       INC     X
```

Once again, our algorithm would fail to recognise a sequence of instructions because of control apparently flowing into a data item. Only the interpretive execution method would be immune to both of these problems.

Conclusion

The main subject of this paper is the problem of separation of instructions from data in detranslation of computer programs. We have illustrated that this problem is in general unsolvable, and consequently modified the objective in our attempt to solve it. The alternative objective is to construct a maximal set of instruction locations. There are various dependency relationships between locations assumed to hold instructions. These relationships are used to categorise locations according to their memberships in one or more 'trees'. Each node in a tree represents a group of locations. Choosing these locations to hold instructions would imply that all the locations corresponding to descendant nodes in the tree must also hold instructions.

The problem of separation of instructions from data is thus reduced to a combinatorial problem of searching for a maximal set of trees out of all the candidate trees, for which we apply a branch-and-bound method. We have also shown that the alternative problem in general belongs to the NP-complete class.

Acknowledgements

Research reported in this paper was supported by the National Science and Engineering Research Council of Canada.

Appendix 1 A branch-and-bound technique to find the maximal self-consistent set of trees in the generalised flow tracing algorithm

The final step of the algorithm presented in Section 3.1 requires that we find a set of trees that has maximal weight but where no two trees conflict (according to the CONFLICTS relation). The solution to this problem defines which locations in the program being disassembled should be identified as instructions. The problem is inherently combinatorial and any combinatorial algorithm that enumerates sets of trees that are conflict-free and computes their weights could be applied. However branch-and-bound techniques are easily adapted to this problem and should greatly reduce the size of the enumeration.

The initial problem can be decomposed into smaller subproblems. Out of the solutions to these subproblems, the one with the greatest weight is to be selected as the solution to the initial problem. The subproblems may, in turn, be decomposed into still smaller subproblems until trivial problems (that may be solved directly) are ultimately reached. The branch-and-bound technique allows many of the generated subproblems to be deleted before they are solved—because the solutions to these subproblems can be determined to be inferior to the solutions of other subproblems. The essential element of the process is the existence of a 'bounding function' which may be applied to any subproblem and which computes an upper bound to that subproblem's solution. Once one subproblem has been solved, we may delete any unsolved subproblems that cannot have weights greater than the weight of the existing solution. A specific algorithm for finding the maximal set of trees is described below.

The basic problem of finding a set of trees with maximal weight but no internal conflicts is broken down into smaller subproblems. Each subproblem is characterised by a pair of disjoint sets $\langle S, R\rangle$. S is a set of trees, no two of which are in conflict with each other. R is a set of trees, none of which is in conflict with any member of S, but which may be in conflict with each other. The solution to the subproblem $\langle S, R\rangle$ is a set P with maximal weight which contains all elements of S and zero or more elements of R. The solution must be self-consistent, i.e. no two elements of P can be in conflict.

There are three subroutines that can be applied to a subproblem $\langle S, R\rangle$:

1. When $|R| = 1$, the *solution* routine constructs $S \cup R$ as the solution of $\langle S, R\rangle$. The value of this solution is also computed as the sum of the weights of the elements of $S \cup R$.
2. When $|R| > 1$, the *partitioning* routine subdivides the problem $\langle S, R\rangle$ into two smaller problems. The suggested method is:
   (*a*) Find x, the element of R that has the largest weight.
   (*b*) Form as the first subproblem $\langle S\cup\{x\}, R\text{-}\{x\}\text{-CONF}(R, x)\rangle$, where CONF($R$, x) is the set of all elements of R that are in conflict with x.
   (*c*) Form as the second subproblem $\langle S, R\text{-}\{x\}\rangle$. This alternative eliminates x from further consideration as it has been already explored under point (b).
3. The *elimination* routine decides whether a problem $\langle S, R\rangle$ can be ignored (i.e. deleted from the problems under consideration). The test is simple, $\langle S, R\rangle$ can be deleted if the sum of the weights of the elements of $S \cup R$ is less than Z, the value of the best known solution. (This sum of weights is the bounding function referred to above.)

Using these three subroutines, the branch-and-bound algorithm has a very simple structure, as follows:

1. Determine which trees contain the entry point location of the program under consideration. For each such tree, t, we construct a problem $\langle\{t\}, U\text{-CONF}(U,t)\rangle$. where U represents the set of all trees in the REQUIRES relation. All these problems are held on a problem list. Z the value of the best known solution is initialised to zero.
2. The following three steps are repeated until the problem list is empty:
   2.1 Remove a problem $\langle S, R\rangle$ from the problem list for consideration. (We recommend that the problem with the smallest R set in the list be chosen.)
   2.2 If $|R| = 1$, then invoke the solution routine. If the value of the solution should be greater than Z then we set Z

to this new value and remember the solution that gave rise to it.

2.3 If $|R| > 1$, then invoke the elimination routine. If the problem is not to be deleted, then the partitioning routine is invoked. The two new subproblems are appended to the problem list.

3. When the process halts, Z is the value of the maximal solution and we have recorded what this solution is.

Appendix 2 A proof that the problem of finding the maximal self-consistent set of instruction locations is NP-complete

The proof given here was constructed by Nathan Friedman (1978). It shows that our formulation of the instruction identification problem is NP-complete (Karp, 1972). From this, we might suspect that no efficient algorithm for its solution will be found. (Our branch-and-bound technique for the last step of the algorithm may, in the worst case, simply try all combinations of trees to see which combination gives the maximal solution.)

We do not prove that the general disassembly problem is NP-complete. This problem is, of course, unsolvable and it is meaningless to discuss the efficiency of algorithms for its solution. We have substituted a similar problem—the problem of finding a maximal self-consistent set of instruction locations—and it is this problem that we show to be NP-complete. We assume that the problem is defined by a REQUIRES relation, a CONFLICTS relation and a weighting function (as constructed by the algorithm of Section 3.1). This assumption is not obviously reasonable. Given some computer, we cannot construct a program that corresponds to some arbitrary pair of REQUIRES and CONFLICTS relations. However, it is possible to construct a collection of subroutines such that the subroutines require each other and conflict with each other according to any desired pair of relations. Hence our formulation of the problem can be justified for any sufficiently rich computer architecture.

To prove that the problem is NP-complete we must demonstrate two things. We must show the existence of a non-deterministic algorithm that solves our problem or equivalently an algorithm that verifies a solution. Secondly, we must show that another problem in the class of NP-complete problems can be reduced to an instance of our disassembly problem.

Part 1

We will first construct a non-deterministic algorithm to solve a simpler problem than our formulation of the disassembly problem. Given some value, K, it will determine whether any set of nodes exists such that no two nodes conflict, no nodes outside the set are required and the sum of the weights for these nodes is greater than K.

This algorithm is trivial to specify. If there are n nodes in the domain (of CONFLICTS or REQUIRES) it would first (non-deterministically) select m different nodes from the domain, where m is a number (non-deterministically) chosen in the range 1 to n. Having chosen the set of nodes, it verifies that the set satisfies all the conditions specified. If the set is satisfactory, the algorithm halts with SUCCESS, otherwise it halts with FAILURE. The running time for this algorithm is easily bounded. It is certainly less than

$$C * (n^2 + n * \log w)$$

where C is some constant and w is the largest weight for any of the nodes. It takes time proportional to n^2 to check the nodes for meeting the CONFLICTS and REQUIRES restrictions (or less time, depending on the data structures used to represent the relations). It takes time proportional to $n * \log w$ to add the weights of the nodes in the set.

Now we can show that this algorithm is polynomial-reducible to an algorithm for solving the disassembly problem. We compute the sum of weights for all the nodes in the domain and call this KMAX. We can now run the algorithm above with various values of K in the range 0 to KMAX. The largest value of K for which the algorithm halts with SUCCESS is the value of the desired solution to the disassembly problem. The set of nodes that is selected for this K is the solution wanted. Assuming that all the weights are integers (as they are for all the suggested cost measures), we can perform a binary search to find this particular value of K. The number of times the algorithm is invoked would be log(KMAX).

Hence an upper bound on the running time to solve the disassembly problem is

$$C * \log(\text{KMAX}) * (n^2 + n * \log w)$$

which is less than

$$C * (\log n + \log w) * (n^2 + n * \log w)$$

Therefore, if the 'size' of the input to the disassembly algorithm is measured by n or $\log w$ or any polynomial function of these two, then the algorithm has a running time bounded by some polynomial function of the 'size' of the input. Hence, the algorithm presented above belongs in the class of NP-algorithms.

Part 2

The second step is to show that some problem, known to be NP-complete, is reducible to the disassembly problem. Let us write a proposition PROP in Conjunctive Normal Form with k clauses and 3 literals per clause in the form:

$$\text{PROP} = \bigvee_{i=1}^{k} \bigwedge_{j=1}^{3} P_{ij}^{e_{ij}}$$

where each P_{ij} represents one of the n literals $L_1, L_2, \ldots L_n$; each $e_{ij} = \pm 1$ and $L_i^{-1} = \neg L_i$. Now, the satisfiability of PROP (determining if there exists truth assignments to the literals such that PROP is true) is an NP-complete problem (Cook, 1971).

The satisfiability of PROP can be modelled as follows: Given that there are n distinct literals appearing in PROP then we construct a domain with $2n$ nodes, which we name $X_1, X_2, \ldots X_n$ and $Y_1, Y_2, \ldots Y_n$. For each literal L_i in PROP, X_i represents L_i and Y_i represents $\neg L_i$.

We set up the CONFLICTS relation so that X_i CONFLICTS Y_i and Y_i CONFLICTS X_i. No other pairs of nodes are in conflict. We set up the REQUIRES relation so that for each conjunction of three literals appearing in PROP, the three nodes corresponding to these three literals (or negations of literals, as appropriate) all require each other (6 REQUIRES combinations are all true). All other REQUIRES entries are false.

All the nodes are given a weight of one (any positive value would be suitable), and now an algorithm to solve the disassembly problem is applied. The maximal set of non-conflicting nodes found by the disassembly algorithm cannot contain more than n elements. This follows from the observation that only one of the two nodes corresponding to a literal can appear in the solution. If the set has exactly n elements we can conclude that PROP is satisfiable and the nodes in the set specify which literals or negations of literals may be made true to cause PROP to be true. Conversely, if the set has less than n elements then PROP cannot be satisfiable.

Hence, if arbitrary REQUIRES and CONFLICTS relations are possible, an algorithm to solve our formulation of the disassembly problem could also solve the satisfiability problem.

Combining the two steps, we infer that the disassembly problem, as formulated in this paper, is an NP-complete problem.

References

Cook, S. A. (1971). The Complexity of Theorem Proving Procedures, *Proc. of Third Annual ACM Symposium on Theory of Computing*, pp. 151-158.

Davies, M. (1958). *Computability and Unsolvability*, McGraw-Hill.

Digital Equipment Corporation. (1979). PDP-11 Processor Handbook.

Friedman, F. L. (1974). Decompilation and the Transfer of Minicomputer Operating Systems, Ph.D. Thesis, Dept. of Computer Sciences, Purdue University.

Friedman, N. (1978). Personal Communication.

Hollander, C. R. (1973). Decompilation of Object Programs, Digital Systems Laboratory, Tech. Report 54, Stanford University.

Housel, B. C. (1973). A Study of Decompiling Machine Languages into High-Level Machine-Independent Languages, Ph.D. Thesis, Dept. of Computer Sciences, Purdue University.

Housel, B. C. and Halstead, M. H. (1973). A Methodology for Machine Language Decompilation, IBM Research Report RJ1316, IBM San Jose Research Laboratory.

IBM Corporation (1970). IBM System/370 Principles of Operation, Form No. GA22-7000.

Karp, R. M. (1972). Reducibility among Combinatorial Problems, In *Complexity of Computer Computations*, edited by R. E. Miller and J. W. Thatcher, Plenum Press, New York.

MMS Staff Report (1978). Protecting Proprietary Software, *Mini-Micro Systems*, August 1978, pp. 78-79.

Sassaman, W. A. (1966). A Computer Program to Translate Machine Language to FORTRAN, *Proceedings of Spring Joint Computer Conference*, 1966, pp. 235-241.

Book review

Machine Intelligence, edited by J. E. Hayes, D. Michie and L. I. Mikulich, 1979; 492 pages. (*Ellis Horwood*, £27·00)

Artificial Intelligence (AI) is a wide but coherent subject. It has spawned important studies which helped the advancement of many areas, especially computer science. It is essentially an applied subject but it has stimulated a great deal of theory. This volume is the ninth in a distinguished series which made major contributions to the subject; it is based on the ninth international machine intelligence workshop held at Repino, near Leningrad, in April 1977.

The first section on abstract models for computation has three short papers. The first by Popplestone on Relational Programming uses relations of the type used by Codd in data base systems to study computations which can be represented by Horn clauses in predicate logic. Searching a large data base can involve this type of computations. The paper shows that relations can provide an effective theoretical model in this area. Stefanuk's paper uses graph-theoretic concepts to optimise the order of evaluation in LISP, taking into account side effects. Yonezawa and Hewitt's paper on modelling distributed systems deals with a topic which is important in computer systems as well as AI. The authors review the notion of an actor, a concept which includes the properties of a process as well as a data structure. Actors communicate through messages. A method of defining distributed computations using actors is illustrated by applying it to an airline reservation system. Other work and possible applications are reported.

The next two sections on representations for abstract and real world reasoning contain four long papers. Blesdoe's paper on automatic theorem proving is comprehensive with several appendices—the methods attempt to construct maximal sets with a given property—and is applicable to theorems in analysis, topology and program verification. Nilson's paper uses a production system for automatic deduction. Two kinds of trees are used: goal trees and fact trees with dual properties. Production rules manipulate these trees until they connect making it possible to use forward as well as backward reasoning. McCarthy's paper on first order theories of concepts and propositions introduces a formal language to deal with philosophical notions such as knowledge, belief, wants, etc, e.g. if Pat knows Mike's telephone number which is the same as Mary's it does not follow that Pat knows Mary's telephone number. The work is important and necessary if AI is to succeed in making computer programs with general intelligence. The final paper in this section is on a theory of approximate reasoning by Zadeh. It is concerned with linguistic truth values of the form very true, more or less true, not very false, etc; semantic equivalence and entailment, e.g. X is very small implies X is small, quantifiers of the form most, many, few, some, etc. It uses the methods of fuzzy logic and possibility distributions instead of probability distributions; various operations such as projection and particularisation are defined. The paper includes a comprehensive bibliography.

Section 4, on search and problem solving, has two papers, the first by Samyolenko describes how to develop adaptive heuristic search methods by varying weighting coefficients or by using fuzzy logic. The methods are applied to various problems: the travelling salesman, the design of communication networks and adaptive routing. The other paper in this section is by Tyugu on a computational problem solver. Problems in Physics or Geometry expressed in natural language are translated by a linguistic processor with the aid of a semantic memory or data base giving definitions and relations between basic concepts, e.g. circle, radius, area, perimeter, etc. Relations are applied to derive the unknown variables from the known ones by constructing a semantic network.

A section on inductive processes has two papers. The first on inference of functions with an interactive system by Jouannaud and Guiho is along the lines of Summers' work. The second paper in this section reports on a truly ambitious project, Lenat's automated scientific theory formation. The program uses a script consisting of some 100 basic concepts in set theory, a large body of some 250 heuristic or production rules, and an agenda mechanism to control the activation of these rules. The interactive program proceeds to discover new concepts and theorems, e.g. disjoint sets, sets of the same length, numbers, prime numbers, etc.

The first paper in the section on perception and world models is by Arbib on local organising processes and motion schemas in visual perception including a comprehensive list of references. The mathematics of control theory is used to describe low level visual circuitry in animals and how these lead to high level schemas in perception including moving objects. The other paper in this section is on the formation of the world model in AI systems by Gladun and Rabinovich. A semantic network is described with a hierarchical structure and associative elements to represent concepts and relations and speed up search, e.g. in planning the actions of robots. A section on robot and control systems has two papers. The first by Okhotsimski, Gurfinkel, Devyanin and Platonov is on an integrated walking robot. Simulation was used to develop strategy and algorithms before the construction of a 6-legged robot. The organisation of the algorithms allows for high level long range and low level short range planning of movements to overcome obstacles and pits. The paper by Pospelov and Pospelov on the influence of AI methods on the solution of traditional control problems refers to the inadequacy of classical control theory in social and economic systems, studies hierarchical heuristic methods for scheduling and allocation problems and finally describes DILOS, an interactive natural language system for carrying out logical and computational problems including an intelligent data bank.

Chess has stimulated much important research in AI and the section on chess includes a bibliography of computer chess by Marsland. The paper by Moussouris, Holloway and Greenblatt on a chess

Continued on page 236