MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Working Paper No. 310 July 1988

Parallel Flow Graph Matching for Automated Program Recognition

by

Patrick M. Ritto

Abstract

A flow graph matching algorithm has been implemented on the Connection Machine which employs parallel techniques to allow efficient subgraph matching. By constructing many different matchings in parallel, the algorithm is able to perform subgraph matching in polynomial time in the size of the graphs. The automated program recognition system can use this algorithm to help make a more efficient flow graph parser. The process of automated program recognition involves recognizing familiar data structures and algorithmic fragments (called *clichés*) in a program so that a hierarchical description of the program can be constructed. The recognition is done by representing the program as a flow graph and parsing it with a graph grammar which encodes the clichés. In order to find clichés in the midst of unfamiliar code, it is necessary to run the parser on all possible subgraphs of the graph, thus starting the parser an exponential number of times. This is too inefficient for practical use on large programs, so this algorithm has been implemented to allow the matchings to be performed in polynomial time.

Copyright © Massachusetts Institute of Technology, 1988

A.I. Laboratory Working Papers are produced for internal circulation, and may contain information that is, for example, too preliminary or too detailed for formal publication. It is not intended that they should be considered papers to which reference can be made in the literature.

Contents

1 Introduction **3**

2 Undirected Graphs **4**

2.1 Data Descriptions 4

2.2 Matchings 4

2.3 Storage of G and H 4

2.4 The Algorithm 5

2.5 Complexity Analysis 6

2.6 Memory Management 9

2.6.1 Local Memory Organization 9

2.6.2 Free Processor Availability 10

2.6.3 Limitations 11

3 Flow Graphs **11**

3.1 Labels 11

3.2 Directed Edges 12

3.3 Ports 12

3.4 Free Processor Availability 13

3.5 Multiple Graph Matching 14

3.6 Program Recognition 14

4 Conclusion **15**

1 Introduction

The process of automated program recognition involves determining the design and function of a program from its source code. This is done by recognizing familiar data structures and algorithmic fragments in the source code so that a hierarchical description of the program can be constructed. These fragments are called *clichés*. A prototype recognition system [7] has been developed for use in the Programmer's Apprentice project [6]. In this system, source code is converted into a graph and then parsed with a graph grammar representing the clichés and the implementation relationships between them. A language-independent graph representation for programs based on the Plan Calculus ([4, 5]) is being used. This graph is then converted into a flow graph for use in the parser. The resulting parse tree gives a description of the program's function and design based on the clichés that were found. In order to recognize clichés embedded in unrecognizable code, the graph parser must be started an exponential number of times (see [7]). Thus, a new approach involving parallel techniques was attempted. This paper describes an algorithm implemented on the Connection Machine [2], which performs parallel flow graph matching in polynomial time for later use in a flow graph parser for program recognition.

The Connection Machine is a SIMD array of up to 64K processors each interconnected by a high bandwidth communication network. It is particularly efficient for pattern recognition because each processor can construct a local matching in parallel with the others. The graph matching algorithm (described in more detail in Section 2.4) stores a partial graph matching in each active processor. Then, at each step of the algorithm, a new set of partial matchings is generated with one more vertex matched. These new matchings occupy a new set of processors and the old processors join the free pool. The algorithm terminates when all the matchings are complete.

The undirected graph algorithm is described in Section 2 and then generalized in Section 3 to allow flow graph matching and the matching of many different graphs in parallel. Section 4 contains conclusions and future work.

2 Undirected Graphs

The undirected graph form of this algorithm is based on Little's proposed subgraph matching algorithm [3] for use in matching large highly interconnected graphs. This algorithm involves finding all occurrences of a graph H as a subgraph in G.

2.1 Data Descriptions

G and H are connected graphs where H is typically small, on the order of 30 vertices, while G can be very large, around 100 vertices. The maximum vertex degrees of G and H, G_{deg} and H_{deg}, are typically no larger than 10 (see Section 2.6).

2.2 Matchings

Each active processor (i.e., each processor not in the free pool) has one partial matching stored in its local memory. The partial matching is a mapping from vertices in H to vertices in G. It is stored as a table indexed by the vertex number, $0, \ldots, |H|-1$, containing vertices of G as elements. If a particular vertex is unmatched, a special value is stored to note this.

2.3 Storage of G and H

The description of G is stored in a distributed way (due to its size, it will not fit in local memory) by storing each element of its adjacency list in a different processor. Thus, if a particular processor needs to lookup the information about G, it must use the communications network to obtain that data. In the existing implementation, the ith vertex of G has its neighbors stored in the processor with cube address i. Then, when an active processor needs to lookup the neighbors of vertex i in G, it performs a **cm:get** instruction to obtain those neighbors from processor i.

The description of H is stored directly in each active processor's local memory. It is stored in the form of an adjacency list in the local memory. Thus, H must be small

enough so that it can fit in the 4K bits of memory along with the matching (see Section 2.6).

The vertices of H must be ordered such that every vertex in the ordering (except the first vertex) is adjacent to at least one vertex earlier in the ordering. The first vertex may be any arbitrary vertex in the graph. Thus, there can be many different valid orderings of a given H. Each vertex in H is matched in this order so that as the matching proceeds, the next vertex to be matched will always have at least one neighbor which is already matched. The motivation for this special ordering is discussed in more detail in Section 2.4. It requires that the graph H be connected.

2.4 The Algorithm

The algorithm works by generating an initial set of partial matchings where vertex 0 of H is matched to each vertex in G, creating an initial set size of $|G|$ active processors. Then, successive generations of the algorithm are performed where each processor first computes all of the possible extensions to its matching in which one more vertex of H is matched to a vertex in G; then, new processors are allocated with the new extended partial matchings. Thus, sets of partial matchings are continuously generated until all the vertices of H are matched. Here is the algorithm:

1. Initialization: $|G|$ partial matchings are generated by taking each vertex in G in turn and matching it with vertex 0 of H. Thus, each possible match for that H vertex is attempted.

2. Successor Generation: For $k = 1$ to $|H|$ generate a new set of partial matchings with vertices 0 through k matched as follows:

   (a) Lookup the neighbors of vertex k in the description of H. Call this set N_h. $N_h \leq H_{deg}$.

   (b) For each element n in N_h, check the partial matching to see if n has already been matched. For those n that have been matched, collect their matches in

the set M. (Due to the ordering of H discussed above, at least one element of N_h must have a match). $|M| \leq |N_h| \leq H_{deg}$.

(c) For each element m in M, lookup the neighbors of m in G. Call this set N_m. This gives $|M|$ such sets of neighbors, each $|N_m| \leq G_{deg}$.

(d) Take the intersection of the sets N_m to obtain one set N_T. Remove vertices from N_T which are already matched in the local matching. $|N_T| \leq G_{deg}$.

(e) For each element v in N_T, allocate a new processor from the free pool and copy the old partial matching into its local memory adding the match of vertex k to v to that matching. If N_T is empty, then no new partial matchings are allocated and the matching has failed.

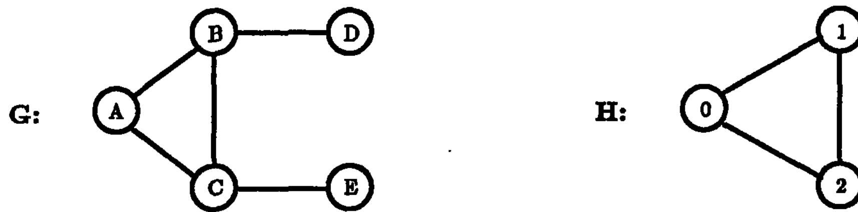
(f) Join the free pool.

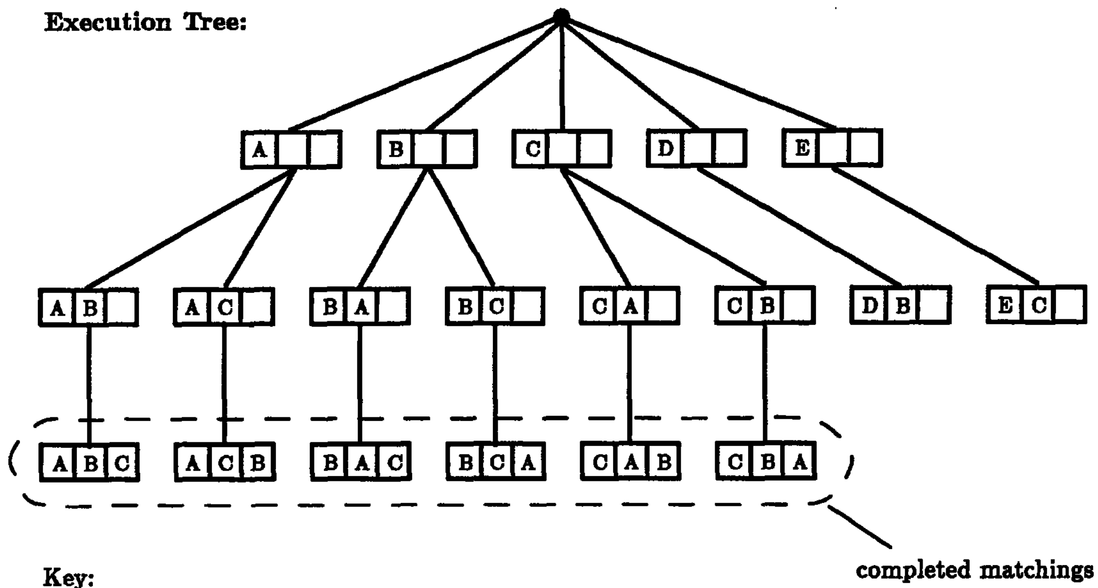
When the algorithm terminates, the remaining allocated processors contain all of the possible completed matchings. The algorithm guarantees the generation of **all** possible matchings (i.e., occurrences of H in G) because it attempts all feasible matching combinations (assuming an infinite free pool of processors – see Section 2.6). The ordering of the vertices of H is necessary so that in step 2b above, the set M will have at least one element. If M were empty, then, to insure all possible matchings, every vertex in G must be matched to k. This causes the generation of $O(|G|)$ new partial matchings which is undesirable due to the size of G and the exponential growth of the search tree.

A sample execution of the algorithm can be found in Figure 1. The graph G contains 5 vertices and H contains 3 vertices. The levels of the execution tree show all the partial matchings existing after each successor generation. The leaves of the tree at level $|H|$ represent all the possible completed matchings.

2.5 Complexity Analysis

The algorithm takes up time for initialization (i.e., storing graph descriptions into the processors) and performs $|H|$ iterations of successor generation. Initialization time





x y z = 0 matched to x,
1 matched to y,
2 matched to z.

Figure 1: A Sample Algorithm Execution

is linear in the size of H and the size of G; typically, initialization takes no more than $50ms$ of CM time.

The time for one successor generation can be divided up into the following steps:

- Finding the set N_h : constant time, $t_{lookuph}$.
- Calculating the set M : $H_{deg} * t_{lookupm}$.
- Calculating the sets N_m : $H_{deg} * t_{lookupg}$.
- Intersecting the sets N_m : $O(H_{deg} * G_{deg})$.
- Allocating new matchings; this must be done a maximum of G_{deg} times:
  - Allocating a new processor : constant time, t_{cons}.
  - Copying the description of H : $|H| * t_{send}$.
  - Copying the partial matching : $k * t_{send}$.

The time constants given above are typically very small. On the average, they fall roughly into the following ranges (measured in CM time):

- $t_{lookuph}$, $t_{lookupm}$: $350\mu s$ - $700\mu s$
- $t_{lookupg}$: $50\mu s$ - $1.5ms$
- t_{cons} : $5ms$ - $6ms$
- t_{send} : $100\mu s$ - $2ms$

The ranges for $t_{lookupg}$ and t_{send} are wide because they depend on the traffic along the network at any given time (they are **cm:send** instructions). The present implementation of the algorithm makes no attempt to optimize such traffic. The $t_{lookuph}$ and $t_{lookupm}$ times are from the **cm:aref** instruction; the t_{cons} time is derived from

the **cm:processor-cons** instruction. In tests performed with $|G| = 50$, $|H| = 10$, $H_{deg} = G_{deg} = 5$, the total CM time for algorithm execution is on the order of $3s$. This gives about $300ms$ per generation as compared with Little's estimate of $50ms$ per generation. The large difference between these times is primarily due to the amount of time required to copy matching information into the successor processors; optimization of the copying process (i.e., using larger message packets and optimizing successor allocation for less traffic) would increase the algorithm's efficiency significantly.

The dominating factor in the above analysis is the time it takes to copy the description and matching into the new processor (step 2e of the algorithm). This takes $O(|H| * G_{deg})$ time per generation, and therefore, since there are $|H|$ total generations, $O(|H|^2 * G_{deg})$ time overall. Thus the execution time is polynomial in the maximum degree of G and the size of H.

2.6 Memory Management

The memory management consists of two separate issues: how to store the information in 4K bits of local memory and what to do when there are no more processors left to allocate.

2.6.1 Local Memory Organization

The information stored in each processor's local memory is shown in Fig. 2. In the low memory addresses, below the stack, are stored various flags for indicating processor state (i.e., whether or not the processor is in the free pool), and the value of k for that matching. If the processor's cube address is in the range 0 to $|G| - 1$, then it will also contain the neighbors of one of G's vertices as well. If not in this range, then that area is left blank. In the high memory addresses, above the stack, are stored the description of H, including its size and adjacency list, and the partial matching. The partial matching is stored as a table indexed by vertices of H. An array of $|G|$ bits is also reserved so that checking whether a G vertex has been matched is efficient (as in step 2d of the algorithm). The rest of the high memory is used as temporary storage

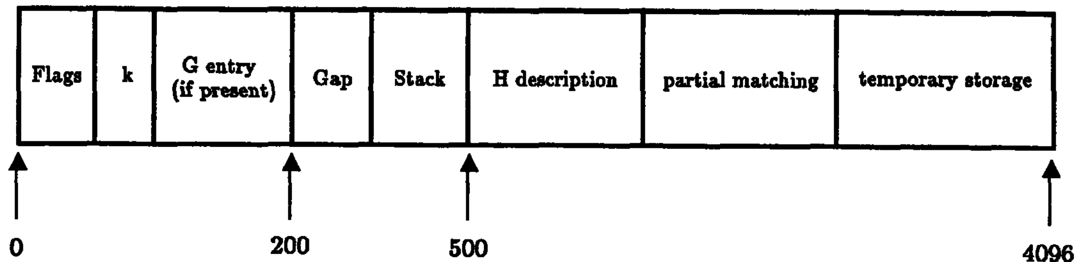


Figure 2: Local Memory Map

space during the calculation of the successor matchings. The optimal size of the stack depends on the size of the problem. Several of the CM instructions used require stack space in proportion to the size of their arguments. Since the problems found in program recognition give rise to relatively small arguments requiring little stack space, a size of around 300 bits is employed.

2.6.2 Free Processor Availability

The algorithm causes an exponential growth in the number of selected processors during the first few generations and later on is pruned by the vertex neighboring constraints. For a large problem, it is not clear how quickly this constraint will take effect. Thus, an exponential growth in the number of selected processors must be anticipated. If the number of processors in the free pool is not enough to generate the new matchings, the present algorithm fails. The problem of running out of processors to allocate has not yet been solved and is still being researched.[1] It seems possible to solve it by checking the free pool at each step and postponing some matchings until others have completed. However, the amount of bookkeeping and memory management this would involve becomes unwieldy. Furthermore, with the added constraints introduced by flow graphs (see Section 3.4), the branching factor of the tree is greatly reduced on the average.

[1] Jim Little, Todd Cass [personal communication].

2.6.3 Limitations

The problem of memory management becomes important as the sizes of G and H become large. There are only 4K bits of memory per processor, and the entire description of H plus the partial matching must be stored there. Since the description of G is stored in a distributed manner, the size of G does not play as crucial a role as the size of H. The two most important sizes in the local memory are the description of H and the matching.

It takes $log_2(|H|)$ bits to encode an H vertex. There are $|H|$ adjacency list entries stored in the description, each of size $H_{deg} * log_2(|H|)$ bits. Thus, the description of H takes up $|H| * H_{deg} * log_2(|H|)$ bits. For the matching, $|H| * log_2(|G|)$ bits are required since there are up to $|H|$ vertices matched, each one to a G vertex. Therefore, an approximate formula specifying these constraints is: $[|H| * H_{deg} * log_2(|H|)] + [|H| * log_2(|G|)] \leq$ 4K bits.

3 Flow Graphs

In order to apply this graph matching algorithm to program recognition, it was necessary to adapt it to deal with flow graphs. A flow graph is a directed acyclic graph where the vertices have distinct labels and input and output ports. For any given vertex, each incoming edge arrives at an input port and all the outgoing edges exit from an output port. An example of a simple flow graph is shown in Figure 3. By definition, flow graphs are acyclic. However, the parallel matching algorithm works on cyclic directed graphs as well.

In order for two flow graphs to match, their vertices must have the same labels and port structure as well as the same edges and directivity. The undirected graph version of the algorithm was easily adapted to work on flow graphs as described in this section.

3.1 Labels

The label for each vertex is stored along with the graph description. So, the labels of the vertices of H are stored in each processor, and the labels of the vertices of G

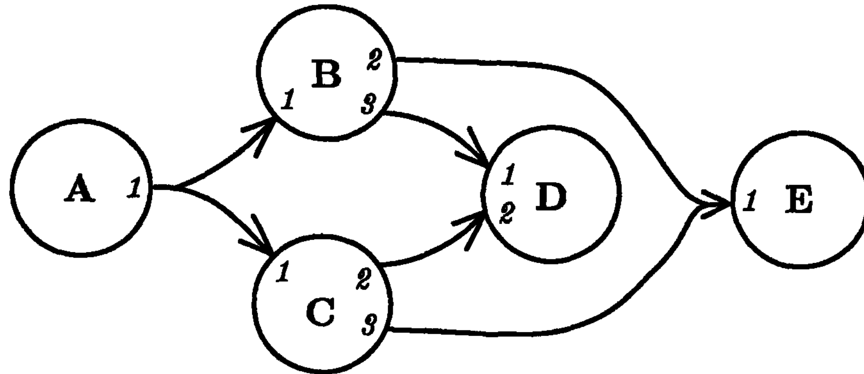


Figure 3: A Simple Flow Graph

are stored in a distributed manner where the label for vertex i is stored in processor i. Since H is small, this fits within the local memory of the processors and does not limit the size of H significantly (it presents no problem for G since it's representation is distributed). Whenever a new set of matchings is to be allocated, the labels of the newly matched vertices are checked and if they do not match, then that matching is discarded. The time required for this extra checking is not significant and the search tree is pruned a large amount due to this constraint.

3.2 Directed Edges

Having directed edges makes it necessary to store an extra bit of information for each vertex in the graph adjacency lists to indicate the direction of the neighbor connection. This direction is checked similarly to the labels above when a new partial matching is allocated. If the direction is not correct, the new matching is discarded. This also helps prune the search tree and requires only constant time per generation.

3.3 Ports

In order to check ports correctly, the entire flow graph is represented internally as

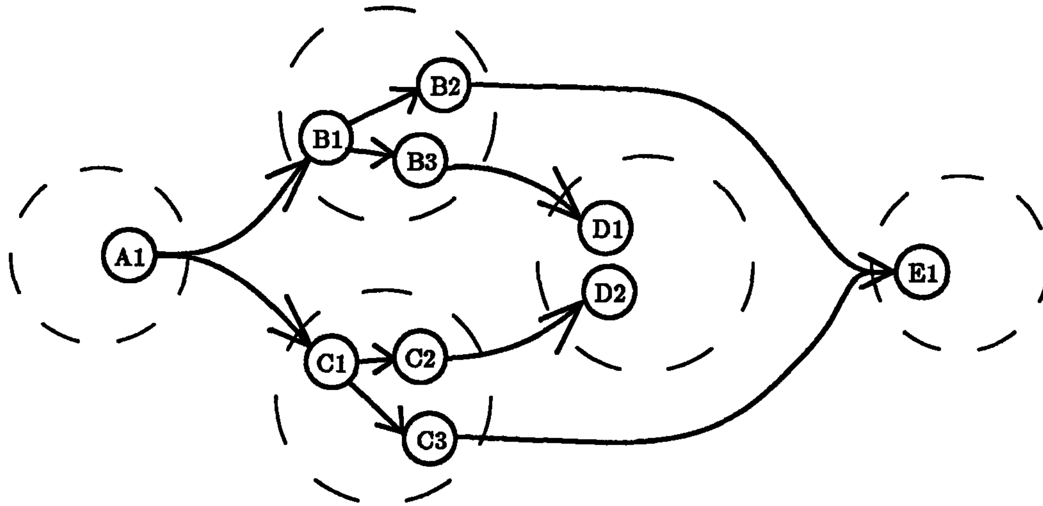


Figure 4: A Flow Graph With Port Vertices

a directed graph where each port becomes a uniquely labeled vertex. All the input port vertices have edges which are directed into the output port vertex. An example of this is shown in Figure 4 where the graph in Figure 3 has been converted into this representation. In order for a successful match to take place, each of the new port vertices must match and thus the original node in the flow graph containing those ports will match.

3.4 Free Processor Availability

With these additions to the algorithm, a great deal of new constraint is introduced: labels must match, edges must have the same direction, and ports must match. Since these are checked dynamically (i.e., at every generation), the average branching factor drops dramatically. As yet, no tests with flow graphs derived from typical programs have given rise to exponential growth of the processor tree. They have all followed a polynomial growth. However, the problem is not eliminated by these new constraints, merely made less threatening. More testing must be performed to determine whether

or not the added constraint enforces a better than exponential growth.

3.5 Multiple Graph Matching

More than one H can be matched to the same G at the same time by storing different H's in each of the processor's local memories. This presents no major problem as each processor must merely keep track of where it is in the matching and when it has completed. Then, completed matchings Are returned as the algorithm executes (not just at the end) as the smaller H's finish before the larger H's do. When new matchings are allocated, both the partial matching **and** the description of H must be copied to the new processor(s) since the same H isn't stored in every processor any longer. Thus, the time is still polynomial in the size of the largest H, but all of the smaller matchings are also obtained. If the size of the largest H is H_{max}, the algorithm will run in time $O(H_{max}^2 * G_{deg})$. This works quite well and adds very little extra time to the algorithm. The space required in local memory will be proportional to H_{max} (as described by the constraint in Section 2.6.3). With this new feature, the algorithm becomes more useful for program recognition because several clichés can be matched at once.

3.6 Program Recognition

Flow graphs are useful in automated program recognition because they can canonically and abstractly represent the data flow constraints found in programs. In the prototype recognition system, programs are converted into flow graphs whose vertices represent operations and whose edges structurally represent the data flow between the operations. Control flow constraints are represented by annotations, called attributes, on the flow graph's vertices and edges.

Clichés are found by searching the program's flow graph for subgraphs structurally matching the clichés' flow graphs and then checking that the attributes of these subgraphs satisfy the clichés' control flow constraints. The parallel algorithm presented in this paper can be used to perform the search for subgraphs which satisfy data flow constraints of clichés. The algorithm is able to match multiple clichéd subgraphs all

over the program's flow graph simultaneously. Not only does this allow clichés to be found in polynomial time, but it also facilitates partial recognition, which is the ability to recognize clichés in the midst of unfamiliar code.

4 Conclusion

An algorithm has been presented which performs multiple flow graph matching in parallel on the Connection Machine in polynomial time in the sizes of the two graphs. In the future, an automated program recognition system can employ this algorithm in order to perform flow graph parsing by hierarchically matching the clichés to the program. More research is required to enable it to perform this hierarchical subgraph matching. A possible approach is to perform a bottom-up parse by matching, substituting with "non-terminals" and then matching again repeatedly.

The problem of free processor availability must be solved, though in the context of automated program recognition it may not be necessary. One possible approach is to suspend some processors when there are not enough to allocate the next generation. The suspended processors would then complete their matchings only after the other processors had finished. This algorithm operates under the assumption that there are always enough processors in the free pool to continue execution. The CM-2 [1], which has the ability to simulate a large space of "virtual" processors existing on some smaller set of physical processors, makes this assumption reasonable. Given enough memory to store the graph information, it can provide a very large free pool which is sufficient for most real problems.

References

[1] Connection Machine Model CM-2 Technical Summary. Technical Report, Thinking Machines Corporation, April 1987.

[2] William D. Hillis. *The Connection Machine.* MIT Press, Cambridge, MA., 1985.

[3] J. J. Little. Parallel Algorithms for Computer Vision on the Connection Machine. Memo 928, MIT Artificial Intelligence Lab., November 1986.

[4] C. Rich. A Formal Representation for Plans in the Programmer's Apprentice. In *Proc. 7th Int. Joint Conf. Artificial Intelligence*, pages 1044–1052, August 1981.

[5] C. Rich. Inspection Methods in Programming: Clichés and Plans. 1988. Submitted. Also published as MIT-AIM-1005.

[6] C. Rich and R. C. Waters. The Programmer's Apprentice Project: A Research Overview. Memo 1004, MIT Artificial Intelligence Lab., November 1987. Submitted to IEEE Software/IEEE Expert Special Issue on the Interactions between Expert Systems and Software Engineering.

[7] L. M. Wills. Automated Program Recognition. Technical Report 904, MIT Artificial Intelligence Lab., January 1987. Master's Thesis.