Comparing Causal-Link and Propositional Planners: Tradeoffs between Plan Length and Domain Size

Atif M. Memon[*]Martha E. Pollack[†]and Mary Lou Soffa[‡]
Department of Computer Science,
University of Pittsburgh,
Pittsburgh, PA 15260
e-mail addresses: {atif,pollack,soffa}@cs.pitt.edu

Abstract

Recent studies have shown that propositional planners, which derive from Graphplan and SATPLAN, can generate significantly longer plans than causal-link planners. We present experimental evidence demonstrating that while this may be true, propositional planners also have important limitations relative to the causal-link planners: specifically, they can generate plans only for smaller domains, where the size of a domain is defined by the number of distinguishable objects it contains. Our experiments were conducted in the domain of code optimization, in which the states of the world represent states of the computer program code and the planning operators are the optimization operators. This domain is well-suited to studying the trade-offs between plan length and domain size, because it is straightforward to

[*]Partially supported by the Andrew Mellon Pre-doctoral Fellowship, awarded by the Andrew Mellon Foundation.

[†]Partially supported by the Air Force Office of Scientific Research (F49620-98-1-0436) and by the National Science Foundation (IRI-9619579).

[‡]Partially supported by the National Science Foundation (CCR 9808590, and EIA 9806525).

manipulate both these factors. On the basis of our experiments, we conclude that causal-link and propositional planners have complementary strengths.

1 Introduction

The field of AI planning has changed markedly in the past five years with the development of a class of algorithms that employs propositional representations for the planning problem. The first such algorithm was Graphplan [3], a two-phased graph expansion/solution extraction algorithm, which forms the basis of recent systems such as IPP [7]. A second class of propositional planners is derived from SATPLAN [5], including, for example, BLACKBOX [6]; these algorithms model planning as satisfiability, so that extremely fast SAT algorithms can be used to solve planning problems.

The recent literature includes a number of studies demonstrating the power of these new approaches. In particular, it has been shown that these approaches are capable of generating significantly longer plans than those that can be generated by the older, causal-link-based planners exemplified by the UCPOP system [10]. In this paper, however, we demonstrate that the propositional planning approaches also have certain important limitations when compared to the causal-link planners. We present experimental evidence showing that while propositional planners can find longer plans than causal-link planners, they have more difficulty finding plans in large domains, where the size of the domain is defined by the number of distinguishable objects it contains.

Our experiments were conducted in the domain of code optimization. Modern compilers perform many code optimizations to improve the memory and run time performance of computer programs. The process of optimizing a program can be modeled naturally as a planning problem in which the states of the world correspond to states of the program code (initially and then after each optimization is applied), and the planning operators are the optimization operators. As in all interesting planning problems, the operators have rich sets of preconditions and effects, and they interact with one another, i.e., the application of one operator can influence the applicability of subsequent operators. This domain is well-suited to an investigation of the trade-offs between plan length and domain size because we can easily manipulate both these factors..

In the next section, we give a very brief introduction to code optimization and then in Section 3, describe its representation as a planning problem. Next, in Section 4, we present the results of experiments we conducted using this domain to compare the performance of a propositional planner and a causal-link planner. On the basis of these experiments, in Section 5 we conclude that these classes of planners have complementary strengths and suggest that future research should aim at combining these strengths.

2 The Code Optimization Problem

Code optimization is a challenging problem and has received much attention in the compiler design community. Compilers apply a large number of *optimizing transformations* to programs, with the goals of (1) reducing the number of statements executed at run time, (2) code size, and (3) resource demands, such as register and memory usage, thereby improving the program's performance [1]. Optimizing transformations, which we will also refer to as optimizations, can be viewed as actions that modify the state of the program being compiled. Like the actions seen in more traditional AI planning domains, optimizations have preconditions and effects, and they interact with one another [13, 14]. The preconditions and effects of an optimization typically involve both the text of the program—the operators and operands in each instruction—and the data flow and control flow dependencies that exist between instructions.

Examples of simple classical optimizations include the following:

Constant propagation can be applied when a program contains an assignment of the form $x \leftarrow c$ where x is a variable and c is a constant, and also includes a subsequent use of the variable x in an instruction that is not *reachable*[1] by any other assignment to x. Constant propagation replaces the use of x by c in the latter instruction.

Copy propagation can be applied when the program includes a copy statement of the form $x \leftarrow y$, and all future uses of x are only reachable from this statement. Copy propagation replaces all uses of x by y.

[1]An instruction j is reachable from instruction i if control can flow from instruction i to instruction j.

```
N = 10;
M = 0;
For I = 1 to N do {
  Read(x);
  y = x;
  if (y > M) then M = y}
Write(M);
```

(a) High Level Program to find the Maximum of N Numbers

```
1.  Mov, N, 10
2.  Mov, M, 0
3.  Mov, I, 0
4.  Add, I, I, 1
5.  Input, X
6.  Mov, Y, X
7.  Jle, (9), Y, M
8.  Mov, M, Y
9.  Jeq, (4), I, N
10.Output, M
```

(b) 3 Address Code

Figure 1: (a) High-level program and (b) its 3-address representation

Dead code elimination removes all "useless" instructions from the program. An example is an instruction of the form $x \leftarrow y$ where the variable x is never subsequently used in the program. Typically, such a condition would result from the application of some other optimization.

Classical code optimizations are usually not applied at the source level. Instead, the high-level source program is first translated to a standard machine-independent intermediate representation commonly known as 3-address code. Figure 1 shows a small high-level program for calculating the maximum of **N** numbers along with its translation to 3-address code. Several optimizations may apply to this program. For example, we can propagate the constant N=10 to Instruction 9 or we can propagate the variable Y=X to Instructions 7 and 8. Figure 2 shows the resulting code, labeled (2) and (3) respectively, that is obtained as a result of applying these optimizations to the original code which is labeled (1). See [1] for more detailed explanations of these optimizations.

Figure 2 also illustrates that optimizations can interact with one another. As a side-effect of applying constant propagation to the current program, we enable the application of another optimization, namely dead code elimination, which was initially not applicable. If we apply copy propagation to the

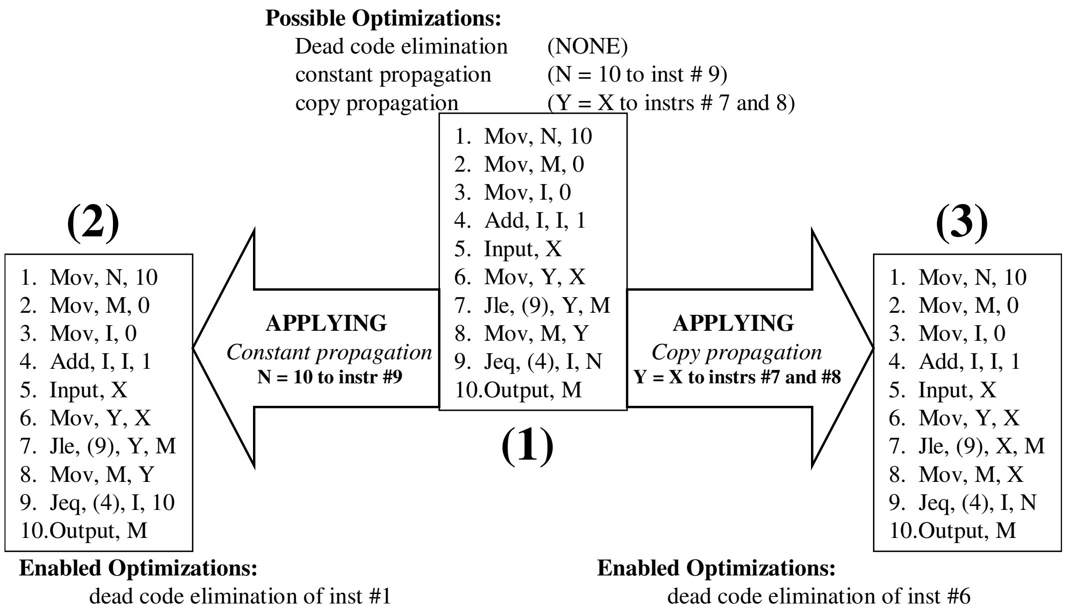


Figure 2: Code Optimizations Interact

initial program then we enable another application of dead code elimination. In this simple example, the application of one optimization enables another. Optimizations can also disable other optimizations, and the order of applying optimizations can effect the final code produced.

For a given program, the performance improvement that results from the overall optimization process may depend upon choosing the best sequence of optimizations to apply. Unfortunately, as the set of optimizations grows and the individual optimizations become more complex, the interactions among them become more intricate. Modern compilers rely on default ordering schemes using distinct phases for applying optimizations [14]; although these have been designed to work well overall, there is no guarantee that the schemes will produce high-quality results for every program.

3 Code Optimization as Planning

We now describe how the code optimization process can be encoded as an AI planning problem. We let the states of the world correspond to states of the

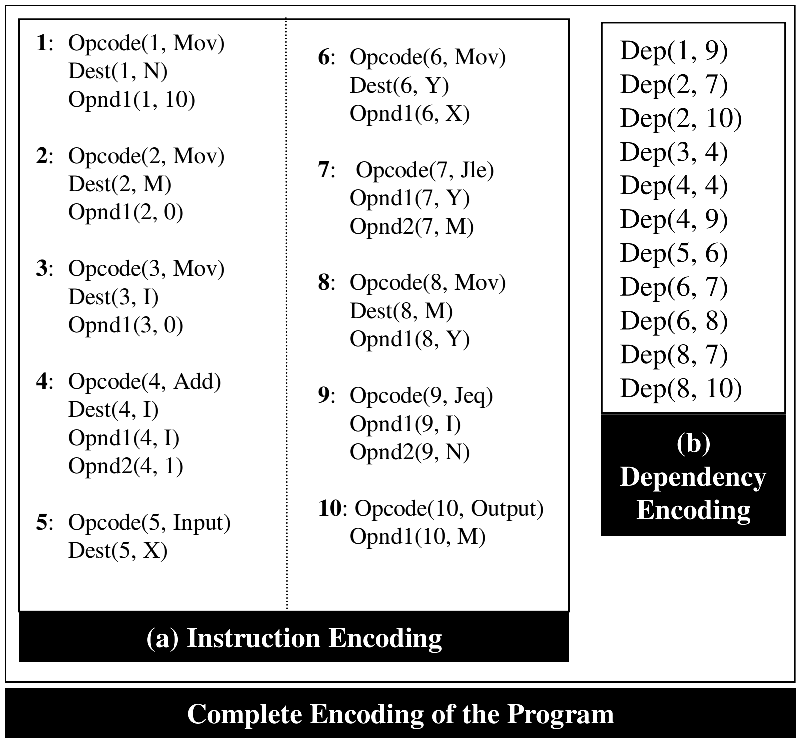


Figure 3: Modeling a Program

program code. We need to represent not only the program text, but also the data flow and control flow of the program. Program text is encoded using four predicates: `opcode`, `dest`, `opnd1`, and `opnd2`. These represent the operator and arguments (destination, first operand, and second operand) for each instruction in a 3-address code program. Data- and control-dependencies are encoded using the `dep` operator, which represents a dependency between two instructions. We say that instruction j is dependent on instruction i if i produces a result that is used by j. Figure 3 shows our encoding of the 3-address code for the program in Figure 1.

The initial state of a code optimization planning problem is a complete description of the input program using the representation just described. The goal state is more interesting. It can be defined as one in which no further optimizations apply. Since optimizations interact, not all goal states will be equally good. One thus needs to use a strong measure of program quality to guide the heuristic search, so that the first goal state found will represent a high-quality program.

In the experiments described below, we use a slightly different notion of goal state. This modified formulation is designed to facilitate our analysis of the effect of plan length on planner performance. Note that any legal sequence of optimizations applied to a program produces an optimized program, albeit not necessarily the best quality program. Thus, in contrast to most AI planning problems, in this domain any intermediate state can be viewed as a goal state. We exploit this fact in our experiments by directly manipulating the length of the plans sought. Thus, we describe a goal state as one in which exactly n optimizations have been applied to the input program. Note that we still require a heuristic function to determine the quality of the program in each node so as to guide the search towards good solutions.

To date, we have not developed strong heuristics for this purpose. Without such heuristics, one might view state-space planning as a better approach to the compiler optimization problem. However, as noted earlier, the optimizations strongly interact with one another, and thus we expect that, as in other domains with rich plan interactions, plan-space planning will be more efficient than state-space planning [2, 8] (but *cf.* [12]). Also, recall that our primary aim in this paper is not to solve the compiler optimization problem, but to explore the relative performance of causal-link and propositional planners, and this domain is especially amenable to the manipulations needed for this comparison.

Returning to the encoding of the compiler optimization problem, the remaining step is to represent each optimization with a planning operator. As mentioned above, optimizations are typically described in terms of preconditions and effects, even in the compiler literature [14]. The preconditions and effects can be encoded using the same set of predicates we employ in representing the state of the program. Figure 4 provides an example for one of the simplest code optimizations: constant propagation, which was described earlier. Figure 4(a) and (b) informally illustrate the preconditions and effects, respectively, while Figure 4(c) shows an encoding of this optimization using the formalization from UCPOP planning system [10]. Similar representations were used for operator encodings throughout our experiments.

Note that even to encode constant propagation, we need to make use of quantified conditions. In principle, one can achieve this with a STRIPS formulation, by generating a separate operator for each conditional effect of an action [7]. However, such encodings lead to an exponential number of operators making even small planning problems intractable. Other code optimizations are more complex, involving operations such as updating dependencies, deleting or moving instructions, creating code, or creating temporary variables, constants, labels. Encoding these optimizations requires the use of conditionals as well as quantifiers. Therefore, for efficient representation, we need a planning system that employs at least a subset of the ADL representation [9] with quantification and conditionals.

4 Experimentation

Our main hypothesis was that propositional planners and causal-link planners have complementary strengths: while propositional planners can generate longer plans than can causal-link planners, causal-link planners can generate plans for larger domains. To test this hypothesis and quantify the impact of plan length and domain size on planner performance, we conducted a series of experiments using the compiler optimization domain. Note that actually optimizing a real program would involve constructing a large plan for a large domain. A medium-sized 3-address code program has on the order of 500 instructions, involving 30 opcodes, 50 variables, and 20 constants; approximately 100-250 optimizations would be applied to it.

In our experiments, we used the UCPOP [10] system as a representative of causal-link planners, and the IPP [7], and BLACKBOX [6] systems as representatives of propositional planners. In Experiment 1, we compare UCPOP and IPP directly on plans with varying plan length and domain size. We could not include BLACKBOX in the experiment, because BLACKBOX is restricted to the STRIPS representation and does not support universal quantification or conditional effects. However, in Experiment 2, we compared UCPOP and BLACKBOX on some select problems that do not pose a representational challenge, to get some indication of whether the trends we see with IPP also occur in BLACKBOX. Finally, in Experiment 3, we return to a comparison of UCPOP and IPP, analyzing the influence of branching factor on their performance.

Table 1: The Core Programs

Name	**Vars**	**Consts**	**Opcodes**	**Lines**
Average	8	3	4	10
OddEven	6	4	3	6
Volume	5	3	2	5
Maximum	7	3	4	9
Sum	3	2	2	3
Squares	5	3	4	6

4.1 Influence of Plan Length and Domain Size

The design for Experiment 1 is as follows. We constructed six realistic core programs, listed in Table 1. They are small 3-address programs that perform simple computations, e.g., computing the average of a set of numbers, finding the maximum, or summing two numbers. The table lists the number of variables (**Vars**), constants (**Consts**), operators (**OpCodes**), and lines of code (**Lines**) in each program. To construct the input for the planning systems, we concatenated multiple copies of the core programs, carefully renaming the variables and constants. This repetition is similar to what a compiler would produce by loop unrolling, in which copies of the body of a loop are generated. We varied the number of concatenated copies of each core program from 1 to 90; thus, for example, we constructed programs based on **Average** that had anywhere from 10 to 900 lines of codes. The number of concatenated copies of a core program serves as our measure of domain size.

The operator library included the three optimizations described earlier: dead-code elimination (DCE), constant propagation (CST), and copy propagation (CPY). In addition to varying the size of the input program (and thus, the domain size), we also varied the length of the plans generated. As described in the previous section, the code optimization problem has special features that allowed us to control the length of the plan produced in any run by specifying that a goal state is one in which a fixed number of optimizations have been applied. We considered three ways of implementing this specification. One possibility was to modify each optimization operator so that it increments a system variable with each plan step. This approach increases the arity of each operator, and is thus potentially detrimental to IPP performance. Another approach was to keep a separate operator for incrementing the system variable and alternate between this operator and the original plan operators. This produces longer plans: a plan that would include n steps in the first approach will require $2n$ steps here. We opted for this second approach to avoid introducing a bias against IPP.[2] A third approach involves controlling the plan length by modifying the algorithms of the two planners to terminate plan generation when a specific length plan was obtained. We opted not to choose this approach, as it would require making changes to the planners' implementation and possibly introduce inefficiencies, leading to unfair comparison.

Plan length was varied from 1 to 10 actual optimizations, i.e., from 2 to 20 total plan steps: our results are reported in terms of the number of actual optimizations. We aborted any run in which memory use exceeded 200 MB (and there was a threat of the system crashing on the machine we were using) or in which none of the available optimizations were applicable. For each run, we measured both the time taken to generate a plan and memory used. We used the RIFO simplification strategy for IPP, and the ZLIFO strategy for UCPOP. RIFO removes irrelevant facts and operators, resulting in smaller propositional formulae. ZLIFO is a generally good strategy for flaw selection in UCPOP, but it is not necessarily the best; consistent with the results of Pollack et al. [11], our preliminary experiments suggested that LCFR-DSep probably would have improved the efficiency of UCPOP. Again, however, we wanted to ensure that we were not introducing any bias in favor of UCPOP. Thus, we also use CPU time as a basis of comparison, despite the fact that that UCPOP is implemented in LISP, whereas IPP is a C implementation. We rely on the CPU-time reports generated by each of the planners. The independent parameters are summarized below:

Domain Size Number of concatenated copies of a core program used as input; Values: $\{1 \ldots 90\}$.

Core Program One of the size core programs from Table 1; Values: {Average, OddEven, Volume, Maximum, Sum, Squares}.

[2]We conducted experiments that confirmed our analysis: the first approach led to worse performance by IPP in all cases; the second led to worse performance by UCPOP in all cases.

Plan Length Number of code optimizations in the final plan; Values: $\{1 \ldots 10\}$.

Operators Number of code optimizations in the operator library; Values: $\{1, 2, 3\}$.

Planning System Values: {IPP, UCPOP}.

We fixed the number of Operators to three (i.e., we used all three code optimizations), and otherwise employed a straightforward factorial design. Thus we ran a total of $90 \times 6 \times 10 \times 1 \times 2$ conditions. For analysis, we separated the results into six sets, one for each of the core programs. The planners displayed very similar patterns of behavior on each of the six core programs, and thus in this paper we describe these patterns for a typical core program ("Sum"). The complete data set is available upon request.

Figure 5 shows the plan times for IPP and UCPOP for fixed domain sizes (S) and increasing plan length. The x-axis shows plan length varying from 1 to 10. The y-axis shows the planning time in seconds. The curves for UCPOP (dotted lines) are for five different domain sizes (S=1..5). We see that UCPOP takes longer as plan length increases and fails to generate plans for length>5. It was, however, able to generate plans for domain size = 90. In contrast, IPP (solid lines) was successful in generating plans for only three domain sizes (S=1..3). We see that for small domain sizes (S<3) IPP's time usage grows very slowly with plan length, but even for $S = 3$, IPP shows rapid growth.

Figure 6 shows memory usage for the same experiments. Here, we see that although UCPOP takes more time to search for a plan, it does no worse than IPP in terms of memory usage for plans of the same length.

Figure 7 shows the same results from a different perspective. Here we show plan times for IPP and UCPOP for fixed plan lengths (PL) and increasing domain size. The x-axis shows the domain size varying from 1 to 10. When PL is small, the results for UCPOP (dotted lines) are all close to the x-axis, indicating very slow growth as domain size increases. On the other hand, IPP (solid lines) shows a rapid growth as domain size increases even slightly.

The related data on memory usage is shown in Figure 8. Here we see that IPP makes heavy demands on the memory as domain size increase, whereas UCPOP uses much less memory.

Table 2: Plan Time of IPP and UCPOP at a Glance

	IPP'S Time	**UCPOP'S Time**
As domain size grows	grows rapidly	grows slowly
As plan length grows	grows slowly	grows rapidly

The complementary behavior of both planning systems is summarized in Table 2. UCPOP takes more time to generate long plans whereas IPP takes more time on larger domain problem instances.

4.2 Comparison with BLACKBOX

Next we considered the question of whether IPP's performance is solely a result of its particular solution extraction algorithms. To this end, we conducted a set of experiments using BLACKBOX, which is similar to IPP in its use of a propositional representation, but differs significantly in its approach to finding a solution: it employs SAT solving techniques.[3]

Because of BLACKBOX's expressive limits, we could not include Dead Code Elimination (DCE) in the operator library. For Experiment 2, then, we used the same design as Experiment 1, except for fixing the number of operators to two (CST and CPY) and using UCPOP and BLACKBOX as the planning systems. We used BLACKBOX's default simplifiers.

Figure 9 and 10 show the planning time and memory usage respectively for BLACKBOX and UCPOP for problems with increasing domain sizes (cf. Figure 7 and 8). Note that the x-axes in these graphs is not uniform: the increments from 1 to 10 are stretched, relative to those between 10 and 90, to allow for more direct comparison with the previous figures. Although BLACKBOX's performance appear to be better than IPP's on these problems, once the domain size is above 10, BLACKBOX exhibits the same pattern of performance relative to UCPOP as did IPP: it is simply unable to cope with the large domains.

Kambhampati et al. [4] observed that Graphplan planners, such as IPP,

[3]Note that BLACKBOX can be made to use several different solvers, including GraphPlan. For our experiments, we hard-coded BLACKBOX to use only one SAT solver, namely `satz`.

might perform worse than state-space search in cases where the constructed graph has layers that contain preconditions of many domain actions, when in fact only a few subsets of these preconditions are reachable at that layer. This observation may be true, but it appears to not be the only factor (and perhaps not even the dominant factor) influencing IPP's performance in our experiments. After all, BLACKBOX exhibits a very similar pattern of behavior to IPP, despite the fact that it uses a very different solution finding algorithm. Additionally, we observed informally that once IPP completes its graph construction, it nearly always solves the problem; similarly once BLACKBOX formulates the SAT problem, it also nearly always finds a solution. Most of the memory demands occur during graph or SAT problem formulation. Kambhampati's observation does not address the behavior of the planner during this phase of plan construction; rather, it addresses the solution finding phase. We, however, observe the inefficiencies of propositional planners during problem representation. An illustration of the memory demands of the planners is seen in Figure 11, which provides a detailed memory trace for IPP on a single problem; note that virtually all its memory demands occur during graph construction. We collected memory traces for all problems with both IPP and BLACKBOX and consistently saw the same behavior. On the basis of the evidence, and our analyses of domain size-memory usage interactions shown in Figures 6, 8, and 10, we conclude that what makes it difficult for such planners to perform efficiently is the increased memory demands posed by domains with a large number of distinguishable objects.

4.3 Influence of Branching Factor

In our final experiment, we returned to a comparison of UCPOP and IPP, so that we could again avoid limits on expressive power, and studied the the influence of the branching factor on these systems' performance. We measure the branching factor by the number of operators available to the planner. The experimental design was identical to that of Experiment 1, except that instead of using all three optimizations, here we also varied the set of optimizations in the operator library, allowing it to contain one, two, or three optimizations. Again for all 6 programs, we saw similar patterns of behavior, and provide data for a typical program.

Figures 12 and 13 show IPP's plan time and memory usage, respectively, for experiments in which the plan length was set to one and the domain size and set of optimizations was varied. With either constant propagation (CST) or copy propagation (CPY), IPP was unable to produce a result for domain size>5. In addition, when both of these optimizations were given simultaneously (CST+CPY), IPP performed worse, and was unable to produce a plan even for domain size=5. We do not show IPP's performance with dead code elimination (DCE), because when this optimization was included in the operator library, IPP could not produce plans beyond domain size 1. In general, whenever IPP was given a set S of operators, it would perform much worse than with a proper subset $S' \subset S$.

An interesting contrast is seen in Figures 14 and 15, which show the influence of changing the size of the operator library, and thus the branching factor. Note that the x-axis varies from domain sizes of 1 to 90 on UCPOP (as opposed to 1 to 5 for IPP). Also, for UCPOP, the x-axis is stretched between 1 and 10. As Figure 14 shows, UCPOP was extremely fast for small domain sizes, and increasing the number of optimizations available had little effect; in fact, until domain size equals approximately ten, the effect is negligible. Dead code elimination (DCE) is challenging for UCPOP, as it was for IPP; making available a larger set of operators (CST+CPY+DCE) actually improves performance relative to requiring only the use of DCE. This behavior of UCPOP is due to the availability of a greater number of operators. Figure 15 shows that UCPOP makes minimal memory demands except in the case of DCE where it requires up to 50MB for domain size 90. This should be compared with IPP's use of 150MB even for CPY with a domain size of four.

In summary, larger operator sets appear to slow IPP's performance and result in increased memory usage. This effect can be attributed to the propositionalization process: as the number of operators increases, a new proposition must be added for each possible instantiation of that operator with respect to all the constants and variables in the input program. In contrast, UCPOP relies on a more dynamic algorithm, searching for solutions and binding dynamically.

5 Conclusions

In this paper we have compared the performance of two classes of planning systems: propositional planners, which are generating significant interest in the planning community, and causal-link planners. We used IPP and BLACKBOX as the main representatives of the first class, and UPCOP as the representative of the second class of planners. We studied each system's performance on code optimization, an interesting and realistic problem which we showed could be encoded as a planning problem. Our primary experiments were aimed at investigating the influence of plan length and domain size on the performance of these planners; we also conducted an experiment aimed at evaluating the influence of branching factor.

Our main conclusion is that the propositional planners and the causal-link planners have complementary strengths and weaknesses. As demonstrated in the recent literature, propositional planners are capable of generating longer plans than causal-link planners. Our experiments are consistent with this finding, but they also show that propositional planners are incapable of finding plans in some larger domains. Some of the planners make use of preprocessors and simplifiers to reduce the domain size. However, they fail in large domains, specifically because the simplifiers are invoked after the propositional formulation is complete, and in large domains the combinatorial explosion does not allow the propositional formula to be represented. Moreover, increased branching factor may have more of a negative effect on propositional planners, especially for medium and large domains. In our view, these results are perhaps more negative than positive: after all, many—perhaps most—real-world problems will require planners capable of generating long plans for large domains.

Our analysis of time and memory usage, however, suggests a possible approach to designing planners to meet this challenge. IPP's main weakness is its inability to handle large domains; this results from the combinatorial explosion during propositionalization. In our experiments, we observed that the large majority of IPP's memory usage occurs during its first phase, graph expansion. One possibility, then, is to avoid complete propositionalization, and instead develop techniques for demand-driven propositionalization and graph construction. That is, the process of creating the plan graph might be interleaved with the search process; where the Graphplan algorithm creates one complete graph level at a time, we are proposing the demand-driven extension of the graph. By this we mean that the plan graph be extended only when needed by the solution finding algorithm and pruned when memory demands increase. Exploration of this idea is deferred to future work.

References

[1] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Surveys*, vol. 26, no. 4, pp. 345–420, Dec. 1994.

[2] A. Barrett and D. Weld, "Partial-order planning: Evaluating possible efficiency gains," *Artificial Intelligence*, vol. 67, no. 1, pp. 71–112, 1994.

[3] A. L. Blum and M. L. Furst, "Fast planning through planning graph analysis," *Artificial Intelligence*, vol. 90, no. 1–2, pp. 279–298, 1997.

[4] S. Kambhampati, E. Parker, and E. Lambrecht, "Understanding and extending Graphplan," in *Proceedings of the 4th European Conference on Planning (ECP-97): Recent Advances in AI Planning*, S. Steel and R. Alami, Eds., Berlin, Sept. 24– 26 1997, vol. 1348 of *LNAI*, pp. 260–272, Springer.

[5] H. Kautz and B. Selman, "Planning as satisfiability," in *Proceedings of the 10th European Conference on Artificial Intelligence*, B. Neumann, Ed., Vienna, Austria, Aug. 1992, pp. 359–363, John Wiley & Sons.

[6] H. Kautz and B. Selman, "Blackbox: A new approach to the application of theorem proving to problem solving," in *AIPS-98 Workshop on Planning as Combinatorial Search*, Pittsburgh, PA, USA, June 1998, AAAI Press / The MIT Press.

[7] J. Koehler, B. Nebel, J. Hoffman, and Y. Dimopoulos, "Extending planning graphs to an ADL subset," in *Proceedings of the 4th European Conference on Planning (ECP-97): Recent Advances in AI Planning*, S. Steel and R. Alami, Eds., Berlin, Sept.24 –26 1997, vol. 1348 of *LNAI*, pp. 273–285, Springer.

[8] S. Minton, M. Drummond, J. L. Bresina, and A. B. Philips, "Total order vs. partial order planning: Factors influencing performance," in *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, 1992, pp. 83–92.

[9] E. P. D. Pednault, "ADL: Exploring the middle ground between STRIPS and the situation calculus," in *Proceedings of KR'89*, Toronto, Canada, pp 324-331, May 1989.

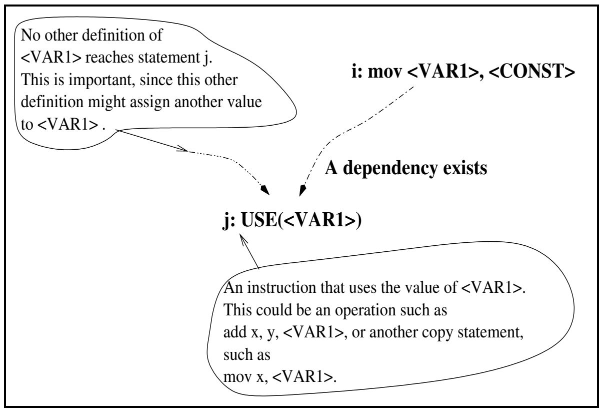
[10] J. S. Penberthy and D. S. Weld, "UCPOP: A sound, complete, partial order planner for ADL," in *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning*, W. Nebel, Bernhard; Rich, Charles; Swartout, Ed., Cambridge, MA, Oct. 1992, pp. 103–114, Morgan Kaufmann.

[11] M. E. Pollack, D. Joslin, and M. Paolucci, "Flaw selection strategies for partial-order planning," *Journal of Artificial Intelligence Research*, vol. 6, no. 6, pp. 223–262, 1997.

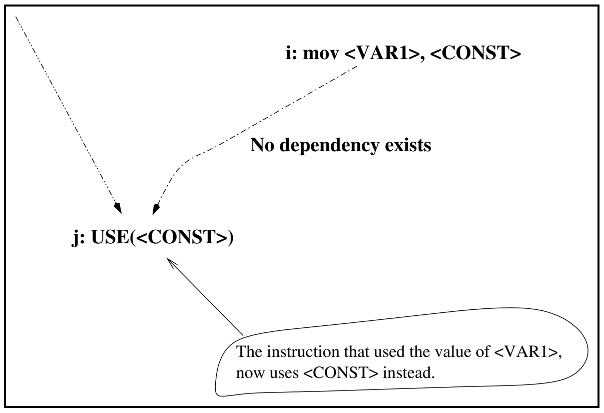
[12] M. Veloso and P. Stone, "FLECS: Planning with a flexible commitment strategy," *Journal of Artificial Intelligence Research*, vol. 3, pp. 25–52, June 1995.

[13] D. Whitfield and M. L. Soffa, "Investigating properties of code transformations," in *Proceedings of the 1993 International Conference on Parallel Processing. Volume 2: Software*, P. B. Choudhary, Alok N.; Berra, Ed., Syracuse, NY, Aug. 1993, pp. 156–160, CRC Press.

[14] D. L. Whitfield and M. L. Soffa, "An approach for exploring code improving transformations," *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 6, pp. 1053–1084, Nov. 1997.


No other definition of <VAR1> reaches statement j. This is important, since this other definition might assign another value to <VAR1> .
i: mov <VAR1>, <CONST>
A dependency exists
j: USE(<VAR1>)
An instruction that uses the value of <VAR1>. This could be an operation such as add x, y, <VAR1>, or another copy statement, such as mov x, <VAR1>.

(a) Preconditions for Constant Propagation


i: mov <VAR1>, <CONST>
No dependency exists
j: USE(<CONST>)
The instruction that used the value of <VAR1>, now uses <CONST> instead.

(b) Effects of Constant Propagation

```
(:operator constprop    :parameters (?i ?j ?c ?x)
  :precondition (and
      (opcode ?i mov)
      (dest ?i ?x)
      (opnd1 ?i ?c)
      (opnd1 ?j ?x)
      (const ?c)
      (dependency ?i ?j)
      (forall (instructions ?k)
          (or (eq ?i ?k) (not (dependency ?k ?j)))))
  :effect (and
          (not (dependency ?i ?j))
          (not (opnd1 ?j ?x))
          (opnd1 ?j ?c)))
```

(c) Encoding Constant Propagation in UCPOP

Figure 4: Modeling an Optimization

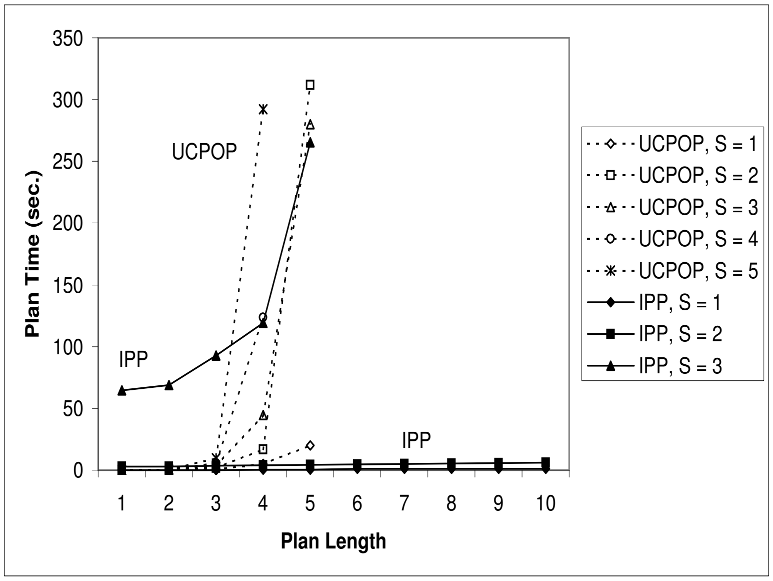


Figure 5: Results of Experiments 1 (IPP/UCPOP); Effect of Plan Length on Plan Time

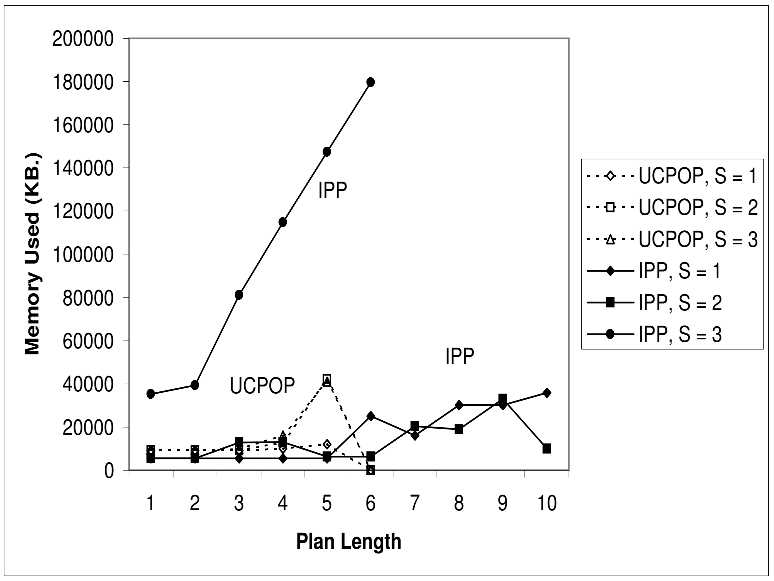


Figure 6: Results of Experiments 1 (IPP/UCPOP); Effect of Plan Length on Memory Usage

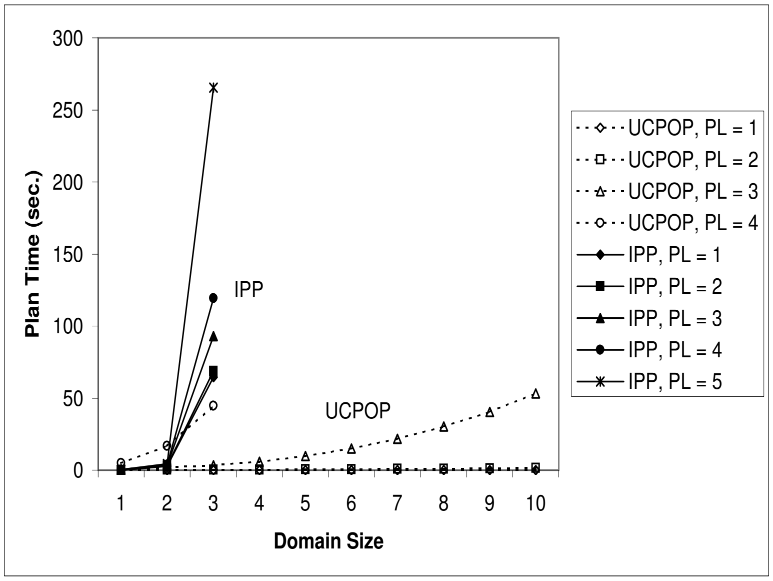


Figure 7: Results of Experiments 1 (IPP/UCPOP); Effect of Domain Size on Plan Time

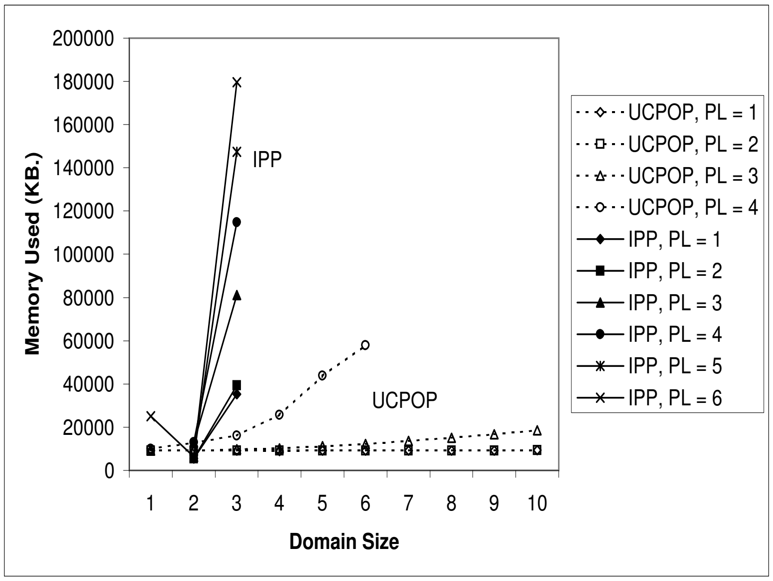


Figure 8: Results of Experiments 1 (IPP/UCPOP); Effect of Domain Size on Memory Usage

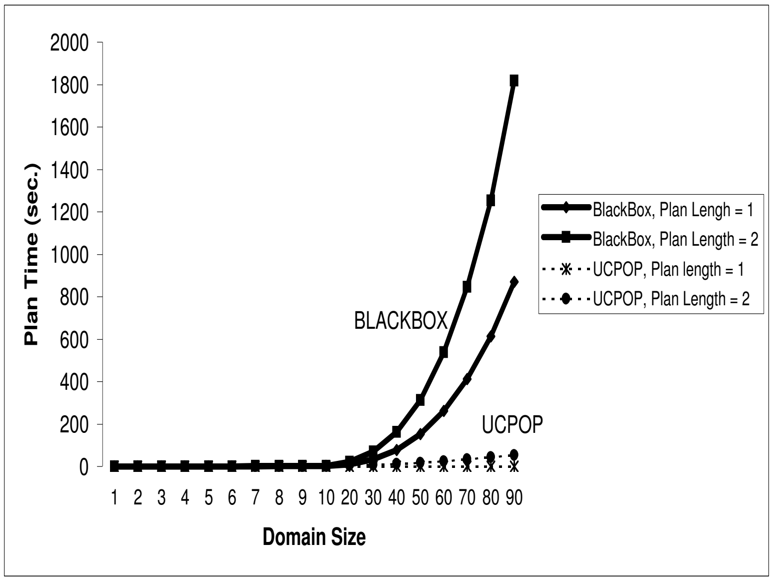


Figure 9: Results of Experiment 2 (BLACKBOX/UCPOP); Effect of Domain Size on Plan Time

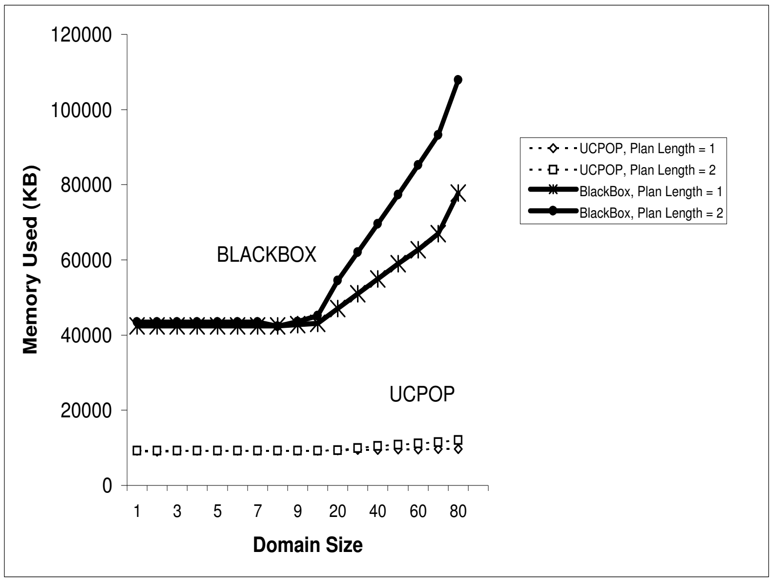


Figure 10: Results of Experiment 2 (BLACKBOX/UCPOP); Effect of Domain Size on Memory Usage

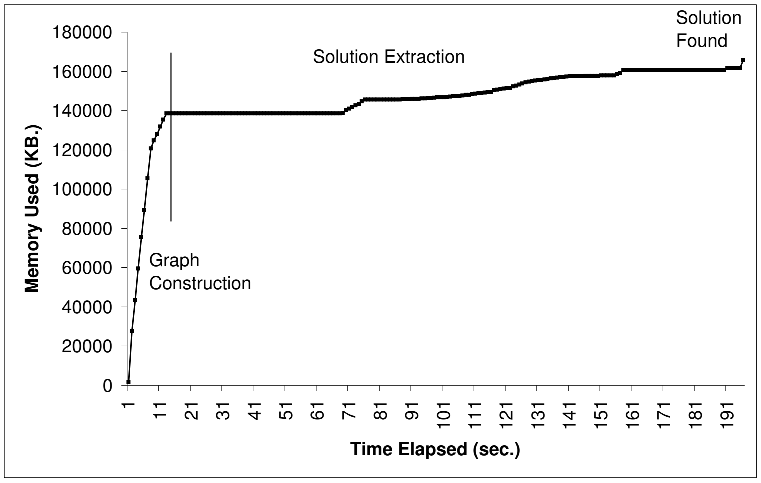


Figure 11: Memory Trace During Execution (IPP)

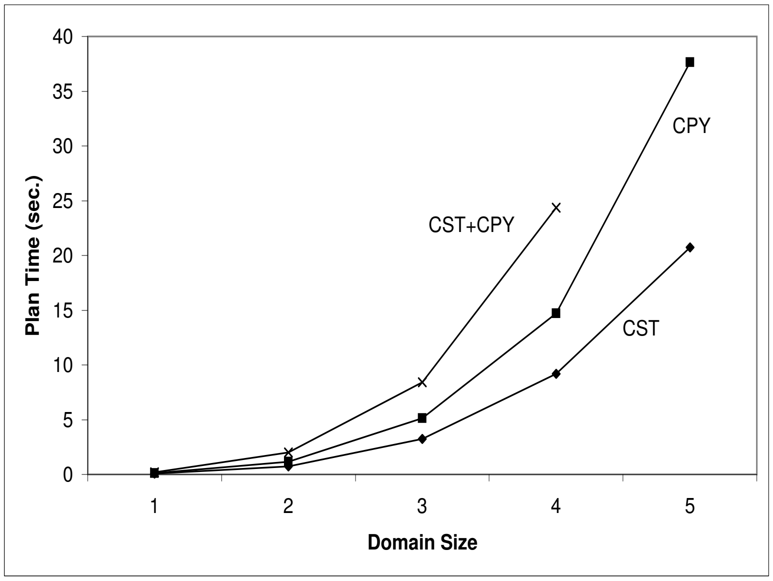


Figure 12: Results of Experiments 2; Influence of Branch Factor on IPP's Plan Time

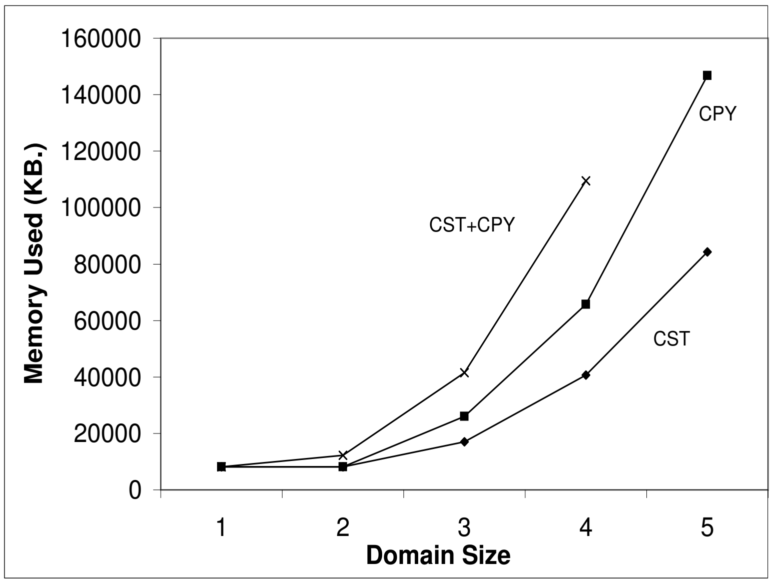


Figure 13: Results of Experiments 2; Influence of Branch Factor on IPP's Memory Usage

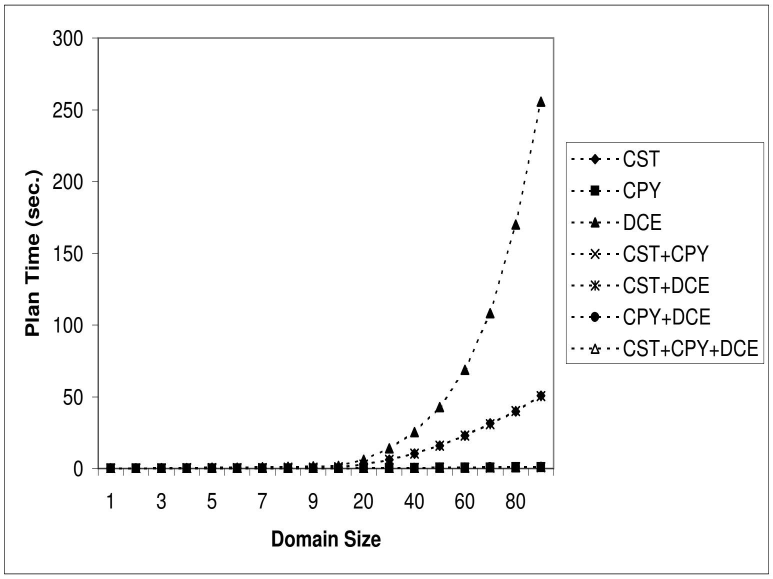


Figure 14: Results of Experiments 2; Influence of Branch Factor on UCPOP's Plan Time

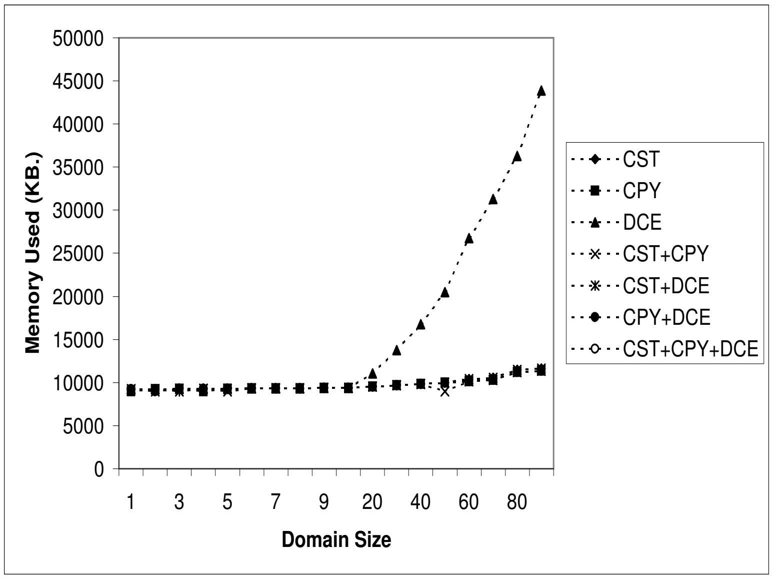


Figure 15: Results of Experiments 2; Influence of Branch Factor on UCPOP's Memory Usage