Implementing Dynamic Programming Recurrences in Constraint Handling Rules with Rule Priorities

Ahmed Magdy[1], Frank Raiser[2], and Thom Frühwirth[2]

[1] Computer Science and Engineering, German University in Cairo
ahmed.mabrouk@student.guc.edu.eg
[2] Institute of Software Engineering and Compiler Construction, University of Ulm
{frank.raiser,thom.fruehwirth}@uni-ulm.de

Abstract. Dynamic Programming (DP) is an important technique used in solving optimization problems. A close correspondence between DP recurrences and Constraint Handling Rules with rule priorities (CHR$^{\mathrm{rp}}$) yields natural implementations of DP problems in CHR$^{\mathrm{rp}}$. In this work, we evaluate different implementation techniques with respect to their runtime. From our results we derive a set of guidelines for implementing arbitrary DP problems in CHR$^{\mathrm{rp}}$.

1 Introduction

Constraint Handling Rules (CHR) ([1]) is a simple high-level programming language that is embedded in a host language. It combines the elements of Constraint (Logic) Programming and rule-based languages ([2]). It was found for the purpose of embedding user-defined constraint solvers in the host language. It evolved through the years to become a more powerful general-purpose programming language ([1]).

An extension of CHR was introduced called Constraint Handling Rules with Rule Priorities (CHR$^{\mathrm{rp}}$) ([3]). It gives CHR rules priorities to control the execution order. It simplifies implementing rule-based algorithms because these algorithms often require some rules to be tried before others for efficiency or correctness reasons ([4]).

An important conclusion, reached by [5], implies that "every algorithm can be implemented in CHR with the best-known time and space complexity". Another conclusion, reached by [6], states that the efficiency of the CHR$^{\mathrm{rp}}$ implementation comes close to the state-of-the-art K.U.Leuven CHR system (and sometimes even surpasses it).

Dynamic Programming (DP) is an important technique used in solving optimization problems. DP follows the optimal substructure property where a solution of a large problem contains within it optimal solutions to smaller subproblems. There is correspondence between Dynamic Programming recurrences and CHR$^{\mathrm{rp}}$ rules that we will exploit in this paper.

In this paper, an implementation to Matrix Chain Multiplication DP problem is introduced and evaluated. Different modifications to the previous problem are proposed and evaluated in order to improve the runtime. Other modifications are tested on four DP problems which are Knapsack ([7]), Edit Distance ([8]), Viterbi ([9]), CYK ([10]). Then guidelines are proposed that help in implementing DP problems in CHR$^{\mathrm{rp}}$. These guidelines are formulated based on the evaluations done on the five DP problems.

2 Preliminaries

This section reviews CHR$^{\mathrm{rp}}$ syntax and semantics. For a more thorough introduction, see [3] or [6]. This section also reviews dynamic programming. For a more thorough introduction, see [11].

2.1 Constraint Handling Rules with Rule Priorities

Syntax. A constraint is of the form $c(t_1, ..., t_n)$ with arity n. Each t_k is a value defined by the host language. There are two types of constraints: built-in constraints and CHR constraints. The host language provides data types and pre-defined constraints which are called the *built-in* constraints. CHR constraints are also called user-defined constraints and are solved by the CHR$^{\mathrm{rp}}$ program.

CHR$^{\mathrm{rp}}$ programs consist of three types of rules shown below.

Simplification	r @ H^r <=> $g \mid B$ pragma priority(p).
Propagation	r @ H^k ==> $g \mid B$ pragma priority(p).
Simpagation	r @ $H^k \setminus H^r$ <=> $g \mid B$ pragma priority(p).

where r represents an optional rule name. H^r and H^k are called *heads* of the rule and represent one or more CHR constraints. g is called the *guard* of the rule and it is zero or more built-in constraints that must be satisfied to apply the rule. B is the *body* of the rule. It's a sequence of CHR constraints and built-in constraints. p can be a static number or an arithmetic expression. The variables in the arithmetic expression can only be from the variables in the head constraints of the corresponding rule.

Semantics. A *constraint store* is a multiset of constraints. Initially, the store starts with the constraints of the initial query. Then, the rules of the CHR$^{\mathrm{rp}}$ program are applied, which manipulates the contents of the constraint store. A rule is applied if its head constraints are in the constraint store and the rule's guard holds. The body of the applied rule is added to the constraint store.

The difference between the three rules is in the way the program deals with the head constraints if the rule fires. In simplification rules, head constraints are removed from the constraint store. In propagation rules, head constraints are kept in the constraint store. Simpagation rules combine the actions of propagation and simplification rules: head constraints before the backslash are kept in the constraint store, and head constraints after the backslash are removed from the constraint store ([1]).

Join Ordering and Indexing. Finding partner constraints to match a rule is crucial in CHR. Join ordering is the order in which these partner constraints are looked up. The time complexity for that operation should be minimized to obtain the optimal complexity. The time complexity of a program depends on the join ordering ([1]). Therefore, the lookup of partner constraints should be efficient. Indexing is an important approach that decrease the lookup time. There are several approaches for indexing. The traditional one uses attributed variables for constant lookup time ([1]).

2.2 Dynamic Programming

Dynamic programming (DP) is an important technique used in solving many optimization problems. Optimization problems are the kind of problems that have several solutions but one of them is better than the others. The idea of dynamic programming is that some intermediate computation may be repeated due to overlapping sub-problems. Therefore, intermediate results are stored and reused when necessary. DP exhibits the optimal substructure property. A problem exhibits this property when the optimal solution of the large problem contains within it optimal solutions of its smaller sub-problems ([11]).

Matrix Chain Multiplication. It is the problem of multiplying n matrices. The order by which matrices are multiplied affects the number of scalar multiplications. To multiply three matrices, there are two ways: $(A_1A_2)A_3$ and $A_1(A_2A_3)$.

Assume A_1 has dimensions 10×100, A_2 has dimensions 100×5, and A_3 has dimensions 5×50. The first option will result in $\underbrace{10 \times 100 \times 5}_{A_1\times A_2} + \underbrace{10 \times 5 \times 50}_{(A_1A_2)\times A_3} = 7500$ multiplications. The second option will result in $\underbrace{100 \times 5 \times 50}_{A_2\times A_3} + \underbrace{10 \times 100 \times 50}_{A_1\times(A_2A_3)} =$ 75000.

DP can be used to find the optimal number of scalar multiplications to multiply a chain of matrices. Then the optimal order can be calculated. The number of scalar multiplications can be computed using (1).

$$m(i,j) = \begin{cases} 0 & i = j \\ \min_{i<k\leq j}\{m(i,k) + m(k+1,j) + p_{i-1}p_kp_j\} & i < j \end{cases} \tag{1}$$

where $m(i,j)$ represents the cost of multiplying matrices from matrix i to matrix j, p_i the second dimension of matrix i, and p_0 the first dimension of the first matrix[3].

The optimal number of scalar multiplications can be computed by calculating $m(1,n)$, where n is the number of matrices to multiply. The DP algorithm runs in $O(n^3)$ time complexity and $O(n^2)$ space complexity ([11]).

[3] The first dimension of matrix i must be equal to the second dimension of matrix $i-1$

3 Implementation

In this section, an implementation of the Matrix Chain Multiplication is introduced.

Constraint p/2 represents the matrices dimensions. The first attribute is the index, and the second one is the dimension. It resembles the p_i in (1). The number of matrices is stored in numOfMatrices/1 constraint as its only attribute.

The constraint that holds intermediate calculated values is called cell/3. The first and second attributes are the same as the first and second arguments of (1). The third one represents the value calculated.

The recursive part in the recurrence is represented as a propagation rule at priority two.

```
cell(I,K,C1) , cell(K1,J,C2) , p(I0,P1) , p(K,P2) , p(J,P3) ,
currentSize(D) ==> K1=:=K+1 , D=:=J-I , I0=:=I-1 |
   R is C1+C2+P1*P2*P3 , cell(I,J,R) pragma priority(2).
```

The head constraints of the rule (without currentsize(D) constraint) resembles the right hand side of (1). The body of the rule resembles the left hand side of (1).

Constraints matched with the propagation rules that represent the recurrence should contain optimal values. Using a non-optimal value will add to the runtime. Therefore, a new constraint (currentSize/1) is defined to ensure that the constraints matched are optimal. The attribute of the constraint represents the size of sub-problems that can be solved by the propagation rule. The attribute starts with the smallest possible sub-problem and is incremented until the whole problem is solved.

$m(i, j)$ calculates the number of scalar multiplications to multiply matrix i to matrix j which are $j-i$ matrices. This difference is an indication to the size of the problem. The problem $m(i, j)$ depends on sub-problems $m(i, k)$, $m(k+1, j)$, where k varies from $i+1$ to j. This implies that both sub-problems are smaller in size than the $m(i, j)$ problem since $k - i \leq j - i$ and $j - k + 1 \leq j - i$.

The attribute of the currentSize constraint is incremented at priority three after all cell constraints from the propagation rules are generated.

Initially, currentSize's attribute is set to zero. This satisfies the base case of the recurrence when $j-i = 0$. At the base case, $m(i, j)$ is set to zero. Therefore, cell(I,I,0) constraints are added to the store.

Selecting the optimal choice (minimum number of scalar multiplications) is done by a simpagation rule at the highest priority. After the program finishes computations, the result will be C in the constraint cell(1,X,C), where X is the number of matrices.

4 Evaluation

In this section, the initial implementation of the Matrix Chain Multiplication is evaluated. Furthermore, two modifications to the initial implementation are proposed to improve the runtime. These modifications are then evaluated to determine their effect. Two other modifications are evaluated on Knapsack's problem implementation. The evaluations are performed on a Dual-Core processor 2.0 GHz machine using SWI-Prolog 5.6.55.

4.1 Initial Implementation

Hypothesis. The Matrix Chain Multiplication implementation runs in $O(n^3)$

Test. The implementation is tested on 20 sizes of the problem. For each size, ten trials are randomly generated. Average time between the ten trials is calculated and plotted. Smallest test case is two matrices and the increment between each set and the next is one matrix. The generated graph is shown in Fig.1.

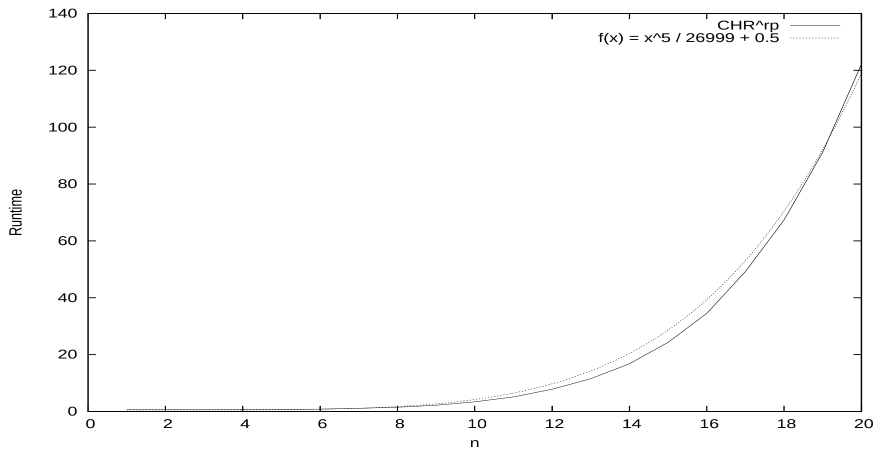

Fig. 1. Matrix Chain Multiplication in CHR$^{\mathrm{rp}}$.

Result. The generated graph can be approximated by a polynomial of 5^{th} order. Therefore, a factor of n^2 is added to the runtime.

4.2 Replacing Guards with Constraints

Guards are not indexed. Replacing the guards of the form `A=:="arithmetic expression"` with constraints can lead to indexing of these constraints and improving the runtime in finding the matching partner constraints.

Hypothesis. Replacing guards from the propagation rules that corresponds to the recurrence with constraints improves the runtime.

Test. Test cases from the previous benchmark are used in this test case to compare the results. The generated graph is plotted against the previous graph in Fig.2

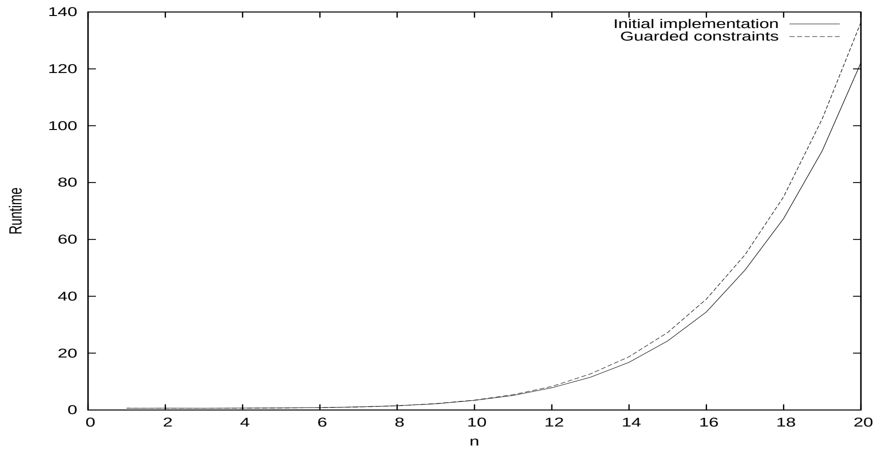

Fig. 2. Guards vs. Guard constraints.

Result. Runtime increased by a factor ranging from 1.1 to 1.2. Therefore, the hypothesis is not true.

4.3 Editing Order of Constraints

The order of occurrence of constraints in a rule can influence the order of matching partner constraints. If the compiler matches partner constraints in the order as they appear in the rule, then there will be more than a constant factor in the complexity to find the matching constraints. Consider the following example:

```
x(I) , y(J) , z(I,J) <=> ...
```

where the number of `y` constraints is big relative to `x` and `z` constraints.

If `x(I)` is the constraint that tries to find partner constraints, then it will try to match `y(J)`. Therefore, it will try several `y(J)` till it reaches the one that matches with `z(I,J)`. Therefore, in the worst case, the number of `y` constraints will be added to runtime.

The factor that increased in the runtime in the previous implementation can be because of the join ordering where the number of head constraints increased and they are not ordered. Therefore, the order of constraints in the recurrence rules is edited so that the rule starts with the `currentSize` constraint, followed by a constraint that has the same attribute of the `currentSize` constraint, followed by a constraint that have common attributes with the previous constraints and so on.

Hypothesis. Ordering constraints in the recurrence rules improves the runtime.

Test. This modification is tested on the implementation introduced in subsection 4.2 because this implementation contains constraints that share many attributes where indexing will be applied. Therefore, by applying this modification to that implementation, the difference in runtime will be observable. The generated graph is plotted in Fig.3

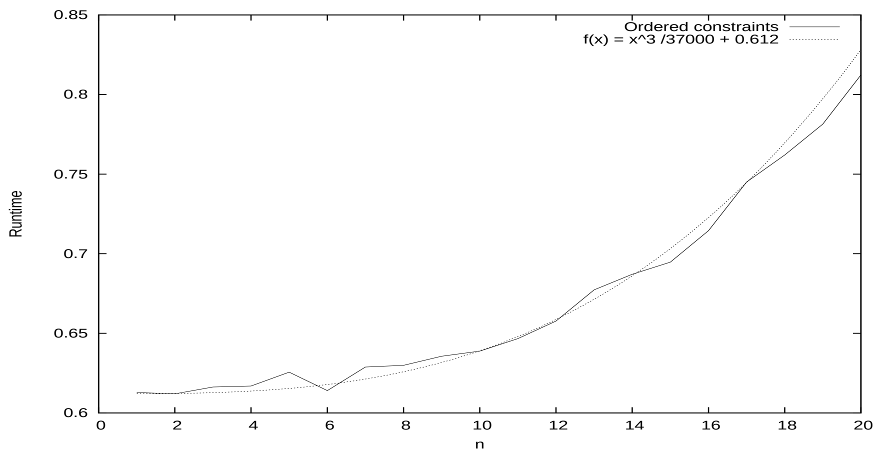

Fig. 3. Ordered constraints in Matrix Chain Multiplication.

Results. The runtime is improved by a factor of n^2. Therefore, the hypothesis is true.

4.4 Other Evaluations

Two modifications are evaluated on the Knapsack problem. The first one is typing constraints' attributes to ground. This modification improved the runtime by a factor nearly equal to 1.2. The second modification is typing constraints' attribute to integer. This modification resulted in the same runtime as typing constraints to ground.[4]

5 Implementing DP problems in CHR^{rp}

In this section, guidelines are introduced that help in implementing DP recurrences in CHR^{rp}.

[4] Generated graphs are shown in Appendix A.

Guideline 1. DP depends on reusing previously calculated values. Therefore, constraint `cell(+,+,+)` is defined to store these values. The first two attributes represent the arguments of the DP recurrence. The last attribute represents the value calculated. Typing constraint's attribute to ground improved runtime as observed in subsection 4.4

Guideline 2. Optimization Rule is defined to make the optimal choice and remove the non-optimal constraint from the constraint store.

It is defined at priority one with a simpagation rule to drop the non-optimal constraint. Some DP problems do not have an optimization rule. The CYK ([10]) problem is such an example. The optimization rule looks like :

```
optimizationRule @ cell(X,Y,Z1) \ cell(X,Y,Z2) <=>
                    "optimization choice" | true pragma priority(1).
```

Guideline 3. Recurrence Rules are defined to perform the actual computation of the DP recurrence and generate `cell` constraints.

The recurrence rules are propagation rules. This ensures that previously calculated results remain in the store to be further used. They are defined at priority two. Their head constraints represent the right hand side of the recurrence, while their body represent the left hand side of the recurrence.

Due to the uncertainty of the order of application of the recurrence rules (since all recurrence rules have the same priority), a non-optimal constraint of a smaller sub-problem could be used to solve a bigger one before optimizing the smaller sub-problem. That would cause to use non-optimal values which would lead to unnecessary computations and adds to the runtime.

Hence, explicit control of the order in which rules are fired is used to ensure that the values used are optimal.

Guideline 4. A `currentSize(+)` constraint is defined to enforce explicit control of the order in which rules are fired.

Its attribute represents the size of the sub-problems that can be solved by the recurrence rules and hence, is problem specific. Not all problems require that kind of control. For a problem like the CYK ([10]) problem, there is no optimal choice made. Therefore, `currentSize` constraint is not used for the CYK implementation.

The attribute of the `currentSize` constraint is incremented when all the sub-problems of that size are solved. It is incremented at priority 3 with a simpagation rule of the form :

```
expand @ maximum(X) \ currentSize(Y) <=> Y<=X |
       NextY is Y+1 , currentSize(NextY) pragma priority(3).
```

where `maximum`'s attribute represents the maximum size of the problem.

Guideline 5. If the recurrence rule requires a guard of the form `A=:="arithmetic expression"`, then replace it with a constraint that satisfies it.

```
cell(A,B,X) , cell(C,D,Y) ==> A=:=B+C | ...
```

is transformed to :

```
cell(A,B,X) , cell(C,D,Y) , guardSum(A,B,C) ==> ...
```

This is done, because when the compiler indexes the constraints it does not include the guards in the indexing. By replacing it with a constraint, it is indexed.

The `guardSum` constraints are added to the store when constraints they will depend on are added to the store. The overhead of generating the `guardSum` constraints is linear in the number of constraints they depend on because for each constraint generated there is at least one constraint that matches it. Hence, generating the constraints introduces a constant factor only to the overall complexity.

It was observed in subsection 4.2 that this modification increased the runtime. However, ordering the constraints after applying this guideline improved the runtime by more than a constant factor.

Guideline 6. The order of head constraints within a rule is significant. The compiler tries to match constraints in the order they appear in the rule as observed in subsection 4.3. Hence, to make use of indexing of constraints, the recurrence rule starts with the `currentSize` constraint, followed by a constraint that has the same attribute of the `currentSize` constraint, followed by a constraint that has common attributes with the previous constraint, and so on.

Guideline 7. Result Rules are defined to get the result and print it.

They are defined at priorities four and five. Two priorities are used because the result of some DP problems is the maximum or minimum of several values. Therefore, the value is calculated at priority four and printed at priority five.

These guidelines are applied on the five DP problems and then evaluated. All generated graphs could be approximated to the complexities of the corresponding problems[5].

6 Conclusion

Five DP problems were implemented then evaluated. Modifications were proposed and evaluated in order to improve the runtime. Then, guidelines following the five implementations and modifications are introduced that helps in implementing DP recurrences in CHR^{rp}.

[5] Generated graphs are shown Appendix B

Future Work. Guidelines introduced are tested on DP problems that have a recurrence consisting of two attributes only. Tests should be done to see if these guidelines can be extended on DP problems that have recurrences consisting of three or more attributes.

`currentSize` constraint is problem specific. However, it is related to the recurrence formulation. The size of the implemented problem was defined by reasoning but no direct mapping from the recurrence was introduced. A possible research is to find the attributes that control the size of the problem directly from the recurrence.

Space complexity was not investigated in this paper. Modifications to the implementations can be researched to reach the optimal theoretical space complexity.

Theoretical complexity analysis can be done to prove that this formulation is in the optimal time complexity.

Related Work. A Viterbi implementation was proposed by [12] and benchmarked. This implementation was then extended and optimized to achieve the theoretical best time complexity. This paper was published so recently that comparisons between our results and their results could not be done.

References

1. Sneyers, J., Van Weert, P., Schrijvers, T., De Koninck, L.: As time goes by: Constraint Handling Rules – A survey of CHR research between 1998 and 2007. TPLP **10**(1) (2010) 1–47
2. Schrijvers, T.: Analyses, optimizations and extensions of Constraint Handling Rules. PhD thesis, K.U.Leuven, Belgium (2005)
3. De Koninck, L., Schrijvers, T., Demoen, B.: User-definable rule priorities for CHR. In Leuschel, M., Podelski, A., eds.: PPDP '07, ACM Press (2007) 25–36
4. Gabbrielli, M., Meo, M.C., Mauro, J.: On the expressive power of priorities in CHR. In López-Fraguas, F., ed.: PPDP '09, ACM Press (2009) 267–276
5. Sneyers, J., Schrijvers, T., Demoen, B.: The computational power and complexity of Constraint Handling Rules. ACM TOPLAS **31**(2) (2009)
6. De Koninck, L., Stuckey, P.J., Duck, G.J.: Optimized compilation of CHR$^{\mathrm{rp}}$. Technical Report CW 499, K.U.Leuven, Dept. Comp. Sc., Leuven, Belgium (2007)
7. Martello, S., Toth, P.: Knapsack Problems: Algorithms and Computer Implementation. John Wiley and Sons (1990)
8. Wagner, R.A., Fischer, M.J.: The string-to-string correction problem. J. ACM **21**(1) (1974) 168–173
9. Rabiner, L.R., Juang, B.H.: An introduction to hidden markov models. **3**(1) (1986) 4–16
10. Sipser, M.: Introduction to Theory of Computation. 2^{nd} edn. Thomson Course Technology (2006)
11. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. 2nd edn. McGraw-Hill Science / Engineering / Math (2003)
12. Christiansen, H., Have, C.T., Lassen, O.T., Petit, M.: The viterbi algorithm expressed in Constraint Handling Rules. In Van Weert, P., De Koninck, L., eds.: CHR '10, K.U.Leuven, Dept. Comp. Sc., Technical report (2010) To appear.

A Knapsack Modifications

A.1 Ground Typing versus Not Typing

Hypothesis Typing variables to ground is more efficient.

Test This optimization is tested on the Knapsack problem. In definition of constraints, `constraint/3` is transformed to `constraint(+,+,+)`. For example, `cell/3` is transformed to `cell(+,+,+)`.

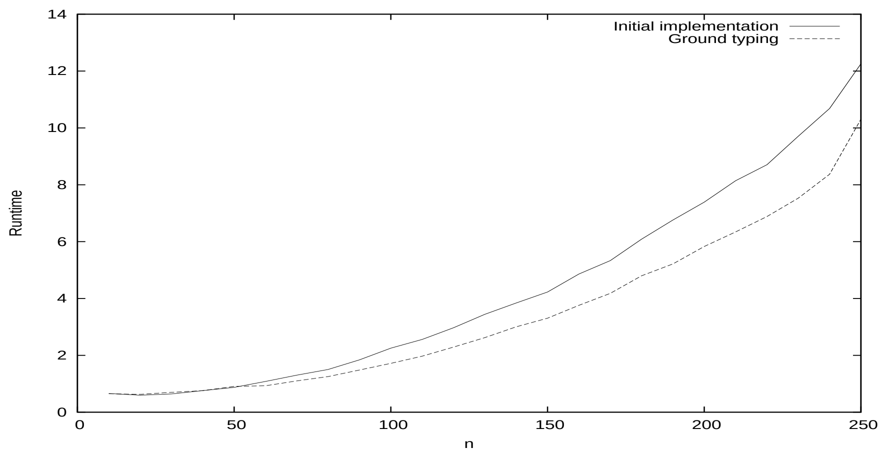

Fig. 4. Typed vs. non-typed.

Results Typing is indeed more efficient by a constant factor approximately equal 1.2.

A.2 Typing Ground versus Typing Integers

It's possible to specify the type of variable to be natural integer. This can give more information to the compiler to do more optimization.

Hypothesis Typing integers is more efficient.

Test This optimization is tested on the Knapsack problem. In the definition of constraints, `constraint(+,+,+)` is transformed to `constraint(+int,+int,+int)`. For example, `cell(+,+,+)` is transformed to `cell(+int,+int,+int)`. This transformation is done for all constraints that have integer attributes.

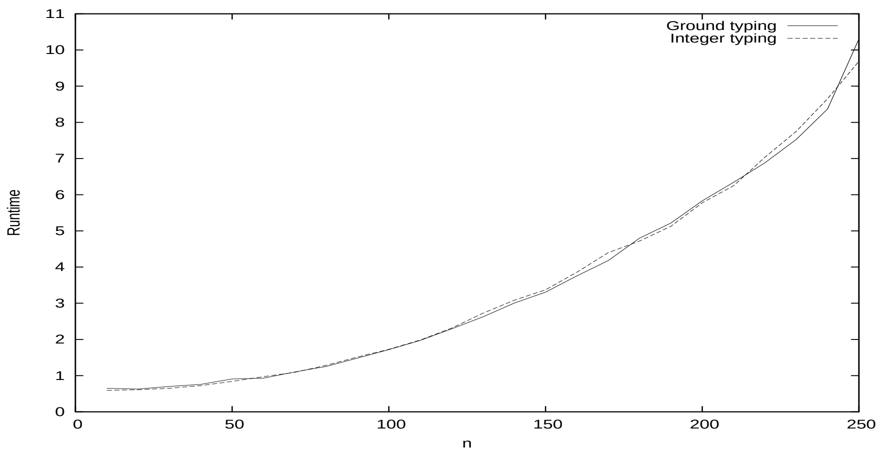

Fig. 5. Ground-typed vs. integer-typed.

Result They produced the same performance.

B Final Implementations

In this section, final implementations of the five DP problems are evaluated after applying the guidelines.

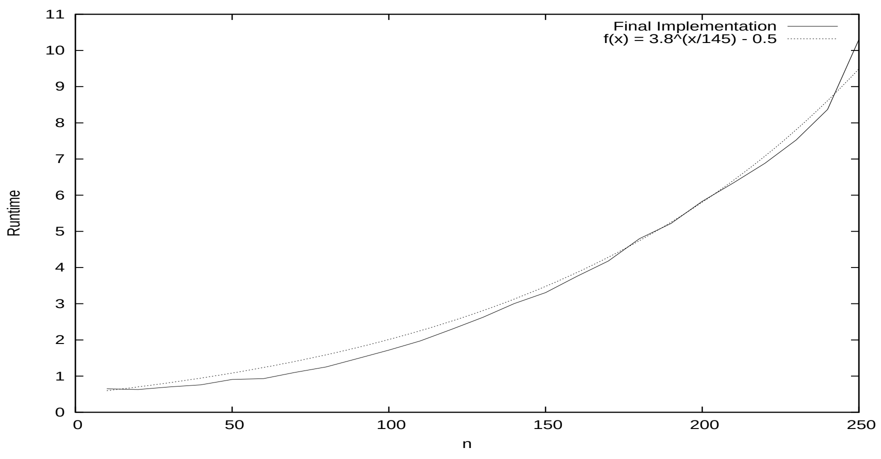

Fig. 6. Knapsack.

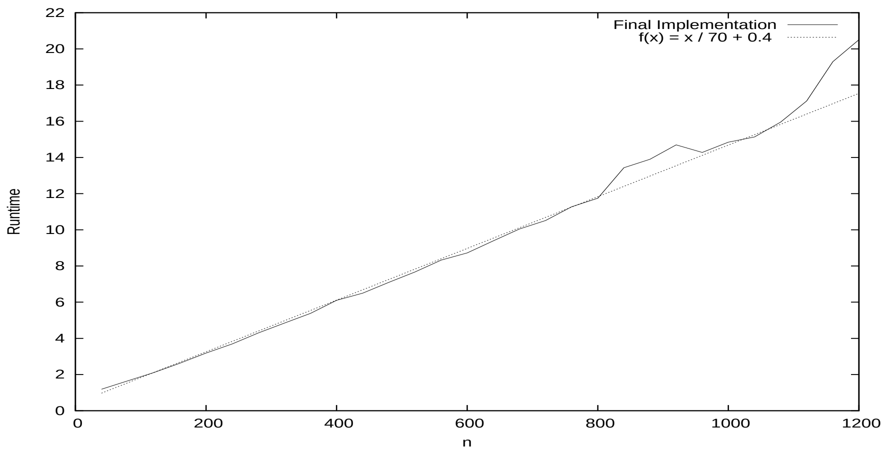

Fig. 7. Edit Distance with varying first string.

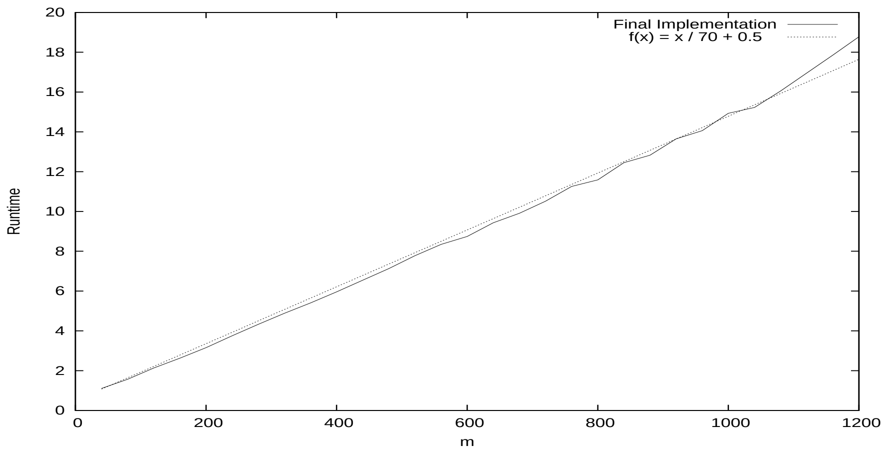

Fig. 8. Edit Distance with varying second string.

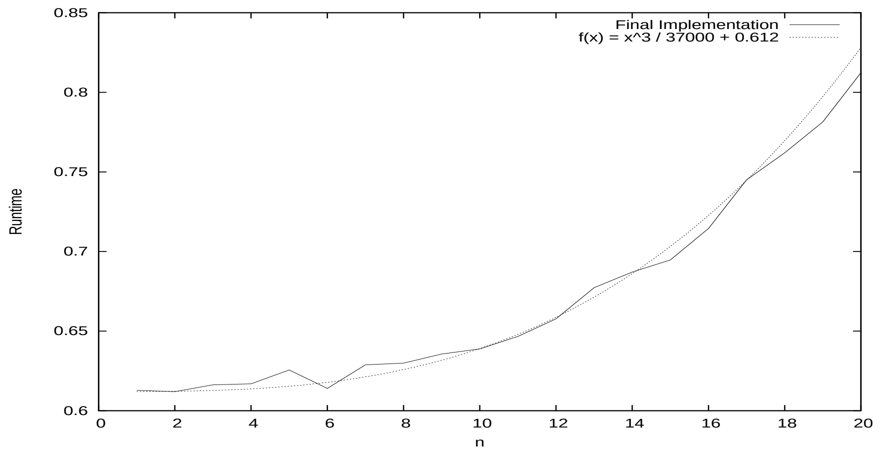

Fig. 9. Matrix Chain Multiplication.

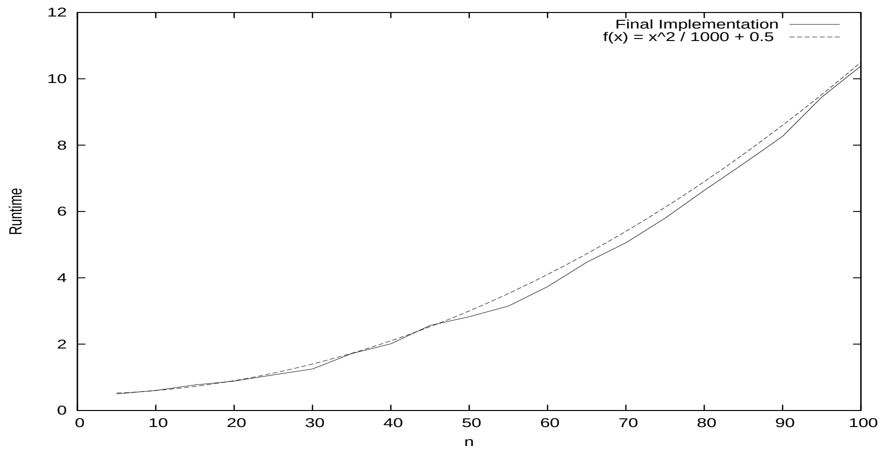

Fig. 10. Viterbi with varying number of states.

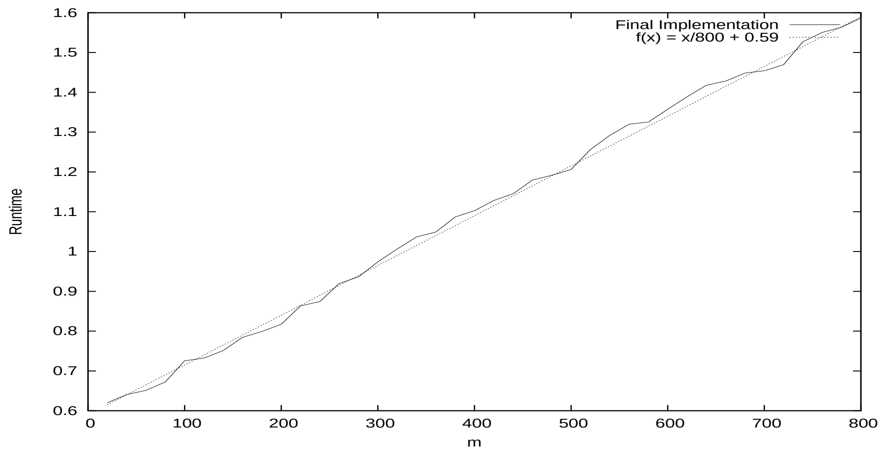

Fig. 11. Viterbi with varying sequence length in sparse transition matrix.

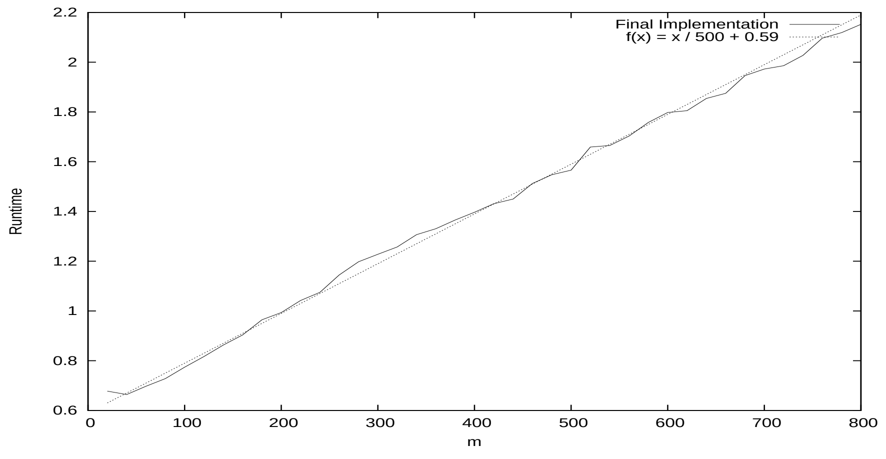

Fig. 12. Viterbi with varying sequence length in complete transition matrix.

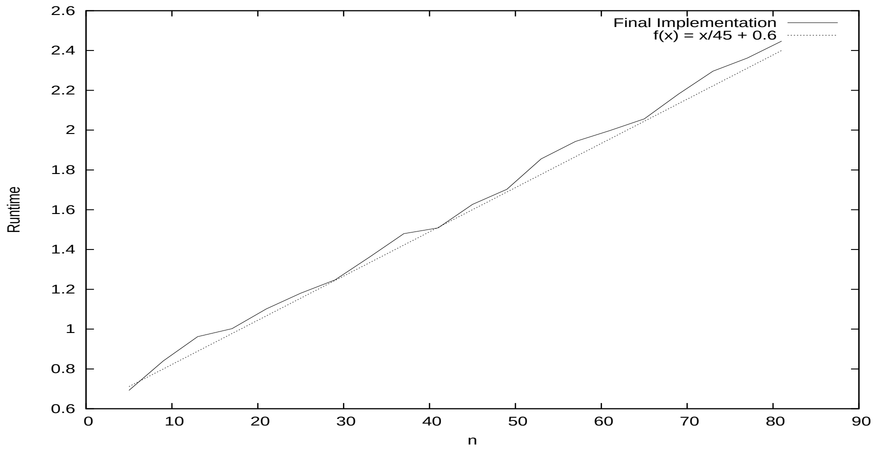

Fig. 13. Viterbi with varying the degree of sparsity of the transition matrix.

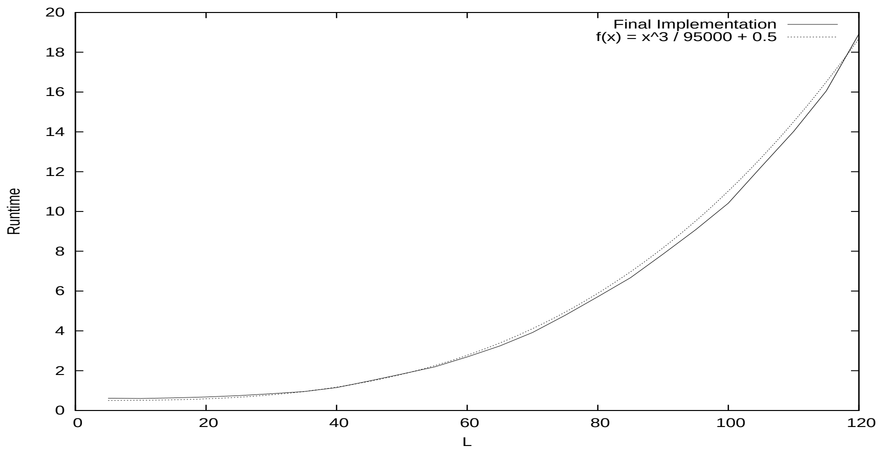

Fig. 14. CYK.