

Appendix A Proofs for optimal number of combinations in Table 3

We give proofs for the formulas in Table 3 for optimal number of combinations for the recursive rule for all 3 recursion variants—left recursion ($\text{path}(X, Y)$, $\text{edge}(Y, Z)$), right recursion ($\text{edge}(X, Y)$, $\text{path}(Y, Z)$), and double recursion ($\text{path}(X, Y)$, $\text{path}(Y, Z)$)—and all graph types. The proofs use the numbers of edges and paths, and of values of variables (x , y , and z) holding the vertices.

Cmpl_n , with edges $\{(i, j) : i \text{ in } 1..n, j \text{ in } 1..n\}$

Given edge holds for all pairs of vertices. By base case rule of path , path holds for all of them too. Therefore, for all 3 recursion variants, all combinations of n vertices for all three variables must be counted, yielding n^3 combinations, as shown in Table 3.

MaxAcyc_n , with edges $\{(i, j) : i \text{ in } 1..n-1, j \text{ in } i+1..n\}$

Given edge holds for all vertex pairs (i, j) where $i < j$. By base case rule of path , path holds for all of them too. Therefore, for all 3 recursion variants, for each vertex i from 1 to n for variable y , there is a path or edge from each vertex from 1 to $i - 1$ for x and a path or edge to each of vertex $i + 1$ to n for z . So the number of combinations is $\sum_{i=1..n} (i - 1)(n - i)$, yielding the closed form in Table 3.

Cyc_n , with edges $\{(i, i+1) : i \text{ in } 1..n-1\} + \{(n, 1)\}$

Given edge holds for all vertex pairs $(i, i + 1)$ for i from 1 to $n - 1$ and for pair $(n, 1)$. By transitivity, path holds for all pairs of vertices. For left (resp. right) recursion, for each path pair (i, j) , there is only one edge going from j (resp. to i), and thus the number of combinations is the size of path , which is n^2 , as shown in Table 3. For double recursion, the analysis is as for Cmpl_n , yielding n^3 in Table 3.

$\text{CycExtra}_{n,k}$, with edges $\{(i, (i-1 + t*n/(k+1)) \bmod n + 1) : i \text{ in } 1..n, t \text{ in } 1..k\} + \text{Cyc}_n$

This is as Cyc_n but adds edges from each vertex to k vertices, so there are $n + nk$ edges total. Still, path holds for all pairs of vertices. For left recursion, for each path pair (i, j) , there are k more edges going from j , and thus the number of combinations is $n^2 + n^2k$, as shown in Table 3. For right recursion, for each of $n + nk$ edges, there are paths to n vertices, and thus the number of combinations is again $n^2 + n^2k$, as shown in Table 3. For double recursion, the analysis is the same as for $\text{Cyc}_{n,k}$.

Path_n , with edges $\{(i, i+1) : i \text{ in } 1..n-1\}$

Given edge holds for all vertex pairs $(i, i + 1)$ for $i = 1..n$. By transitivity, path holds for all pairs (i, j) where $i < j$, so the number of path pairs is $\sum_{i=1..n} (n - i)$, which equals $\frac{1}{2}(n - 1)(n - 2)$. For left (resp. right) recursion, for each path pair (i, j) , there is one edge going from j (resp. to i), and thus the number of combinations is the number of path pairs, i.e., $\frac{1}{2}(n - 1)(n - 2)$, as in Table 3. For double recursion, the calculation is the same as for MaxAcyc_n , yielding the closed form in Table 3.

$\text{PathDisj}_{n,k}$, with edges $\{(i, i+k) : i \text{ in } 1..(n-1)*k\}$

This is the same as Path_n , but multiplied by k for each recursion variant.

Grid_n , with edges $\{(j, j+1) : i \text{ in } 1..n, j \text{ in } (i-1)*n+1..i*n-1\} + \{(j, j+n) : i \text{ in } 1..n-1, j \text{ in } (i-1)*n+1..i*n\}$

For left recursion, for each vertex in position (i, j) of the grid not in the last row or column, there are paths from $i \times j - 1$ vertices before and edges to 2 vertices after; for each vertex in the last column on row i but not last row, there are paths from $i \times n - 1$ vertices and an edge to 1 vertex; for each vertex in the last row but on column j not last column, there are paths from $n \times j - 1$ vertices and an edge to 1 vertex. Thus, the number of combinations is $\sum_{i=1..n-1, j=1..n-1} (i \times j - 1) \times 2 + \sum_{i=1..n-1} (i \times n - 1) + \sum_{j=1..n-1} (n \times j - 1)$, yielding the closed form in Table 3.

For right recursion, the counting is similar, except to consider each vertex not in the first row or column, then each vertex in first column but not first row, and each vertex in first row but not first column, edges from 2 or 1 vertex before, and paths to vertices after. Thus, the number of combinations is $\sum_{i=2..n, j=2..n} 2 \times ((n - i + 1) \times (n - j + 1) - 1) + \sum_{i=2..n} ((n - i + 1) \times n - 1) + \sum_{j=1..n} (n \times (n - j + 1) - 1)$, yielding the closed form in Table 3.

For double recursion, for each vertex in position $i = 1..n$ and $j = 1..n$ in the grid, there are paths from $i \times j - 1$ vertices before, and paths to $(n - i + 1) \times (n - j + 1) - 1$ vertices after. Thus the number of combinations is $\sum_{i=1..n, j=1..n} (i \times j - 1) \times ((n - i + 1) \times (n - j + 1) - 1)$, yielding the closed form in Table 3.

BinTree_h, with edges $\{(i, 2i) : i \text{ in } 1..2^{h-1}\} + \{(i, 2i+1) : i \text{ in } 1..2^{h-1}\}$ and

BinTreeRev_h, with edges $\{(2i, i) : i \text{ in } 1..2^{h-1}\} + \{(2i+1, i) : i \text{ in } 1..2^{h-1}\}$

Left and right recursions are as proved in Section 4.1. Double recursion for **BinTree_h** (resp. **BinTreeRev_h**) considers each level $i = 1..h$ from root to leaves, with 2^{i-1} vertices at level i , each having paths from (resp. to) $i - 1$ vertices, and to (resp. from) $2^{h-i+1} - 2$ vertices. Thus the number of combinations is $\sum_{i=1..h} 2^{i-1} \times (i - 1) \times (2^{h-i+1} - 2)$, yielding the closed form in Table 3.

x_{n,k}, with edges $\{(i, n+1) : i \text{ in } 1..n\} + \{(n+1, n+1+j) : j \text{ in } 1..k\}$

All 3 recursion variants just join the two sets of edges, and thus the number of combinations is nk , as shown in Table 3.

y_{n,k}, with edges $\{(i, n+1) : i \text{ in } 1..n\} + \{(i, i+1) : i \text{ in } n+1..n+k-1\}$

For left recursion, each vertex i from 1 to $k - 1$ in the stem of **Y** has paths from $n + i - 1$ vertices. Thus the number of combinations is $\sum_{i=1..k-1} (n + i - 1)$, yielding the closed form in Table 3.

For right recursion, vertex $n + 1$ has edges from n vertices and paths to $k - 1$ vertices; each vertex i from 2 to $k - 1$ in the stem of **Y** has edge from one vertex and paths to $k - i$ vertices. Thus the number of combinations is $n \times (k - 1) + \sum_{i=2..k} (k - i)$, yielding the closed form in Table 3.

For double recursion, each vertex i as in left recursion has also paths from $n + i - 1$ vertices, but has paths to $k - i$ vertices. Thus, the number of combinations is $\sum_{i=1..k-1} (n + i - 1) \times (k - i)$, yielding the closed form in Table 3.

w_{n,k}, with edges $\{(i, n+1 + (i+j-1) \bmod n) : i \text{ in } 1..n, j \text{ in } 1..k\}$

This graph has only paths of length 1. All 3 recursion variants have the number of combinations being 0, as shown in Table 3.

Appendix B Performance measurements and analysis

We designed and performed detailed experiments with all three rule systems XSB, clingo, and Soufflé, all three rule variants, and all 12 input graph types.¹ We describe these experiments, analyze the measured running times, and compare with the analyzed optimal complexities.

For XSB, with the general programming power of Prolog, we wrote the scripting and timing code directly in XSB. We obtain separate times for (1) `LoadRules`—load rules (including `table` directive), using `consult`, (2) `ReadData`—read data from a file of facts, using `load_dync`, (3) `Query`—query all path facts, using `fail` to repeat without writing, and (4) `WriteRes`—write results to file, using `write`.

For clingo, with its interface for using it from Python, we wrote the scripting and timing code in Python. We obtain separate times for (1) `LoadRules`—load rules (including `show` directive), using `load`, (2) `ReadData`—read data from a file of facts, using `load`, (3) `Ground`—ground rules, using `ground`, and (4) `Solve`—search for all path facts, using `solve`, and (5) `WriteRes`—write results to file, using `write`.

For Soufflé, with its interface for using it from C++, we wrote the scripting and timing code in C++. We obtain separate times for (1) `RulesToC++`—compile rules (with type and input-output declarations) to C++, using `souffle` command line, (2) `C++ToExec`—compile C++ to executable, using `g++` command line, (3) `LoadRules`—load compiled executable for rules, using `newInstance`, (4) `ReadData`—read data from a directory of files of facts, one for each given predicate, using `loadAll`, (5) `Query`—compute all path facts, using `run`, and (6) `WriteRes`—write results to file in CSV format, using `printAll`.

For each input graph type, we generate graphs, exactly as defined in Table 2, with increasing values for n and h , and with both constant values and increasing values for k .

For each experiment, we report average CPU time over 5 runs. The variations across runs are negligible. All measurements were taken on an Apple M3 Pro chip, 2024, with a 4.05 GHz 11-Core CPU and 18 GB unified memory, 512 GB SSD, running MacOS Sonoma 14.5, XSB 5.0.0, clingo 5.7.1, and Soufflé 2.4.1-50-ge6cc66820.

B.1 Running times, confirmations, and exceptions

We discuss running times first for graph types with a single parameter, n or h , and then for graph types with two parameters, n and k , but we compare across all graph types, and for all recursion variants and all systems.

Graph types with a single parameter. We first examine the running times for five drastically different graph types with parameter n for number of vertices: `Cmpln`, `Cyclen`, `MaxAcycn`, `Pathn`, and `Gridn`, including densest (`Cmpl` graphs) and sparsest connected (`Path` graphs). We explain how the calculated precise complexities in Table 3 and the different optimized inference efficiencies in Section 4.2 are confirmed by the sameness and differences as well as trends and constants in actual inference times—`Query` for XSB

¹ Our benchmarking experiments are available at <https://github.com/Sirnej/trans-bench>.

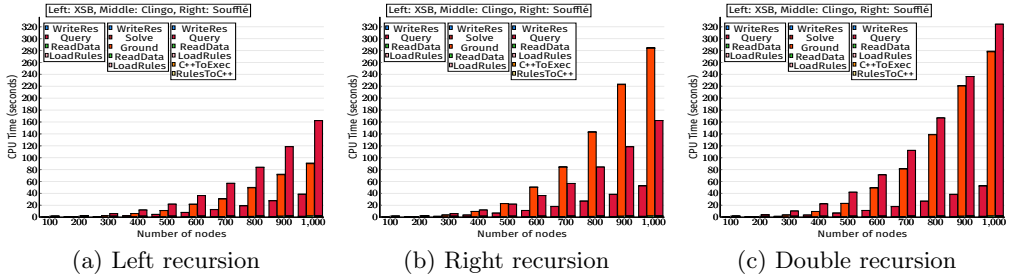


Fig. B1: Running times on `Cmpl` graphs, up to 1000 vertices, 1000000 edges.

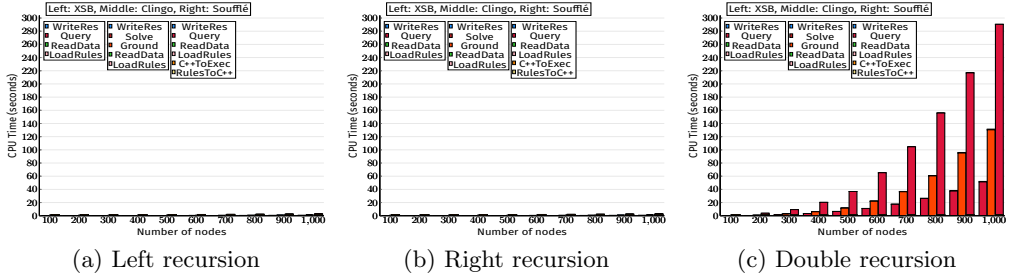


Fig. B2: Running times on `Cyc` graphs, up to 1000 vertices, 1000 edges.

and Soufflé and `Ground` and `Solve` for clingo. The input and output times are also explained by data and result sizes, but they are relatively small here, so we focus on inference times.

Figs. B1 to B5 show the running time details for `Cmpln`, `Cyclen`, `MaxAcycn`, `Pathn`, and `Gridn`, for the number of vertices up to 1000 and the number of edges varying the mostly widely possible, from 1000000 (for `Cmpl1000`) to 999 (for `Path1000`) for a graph to stay connected. First, we observe the following that hold across all Figures, not just the five figures here—all consistent with the analysis in Section 4.2—except for an XSB performance bug discovered on the simplest case (w graphs, left recursion), described later:

- Soufflé has an extra compilation time (about 1.5 seconds, with tiny `RulesToC++`, in yellow, and virtually all `C++ToExec`, in orange). Of course, this time needs to be incurred only once for each program, not each run of the program.
- clingo’s grounding time (`Ground`, in slightly lighter red) dominates its solving time (`Solve`) which is tiny, because optimized grounding already inferred all resulting facts.
- XSB is the fastest in all runs, confirming the overhead of tabling is minimal compared with the cost of grounding in clingo and semi-naïve evaluation in Soufflé.

We also confirmed the trends of being superlinear in n for these graph types, as well as, for all other graph types, the expected trends of being linear or superlinear in n and k and of an explosive increase with respect to n , as analyzed in Table 3, except for the case where the XSB performance bug was discovered.

For more interesting detailed comparison among different recursions with different inference methods, we first examine Fig. B1, for `Cmpln`. It is the most expensive case, with 1,000,000,000 combinations for 1000 vertices and 1,000,000 edges. Table 3 says that

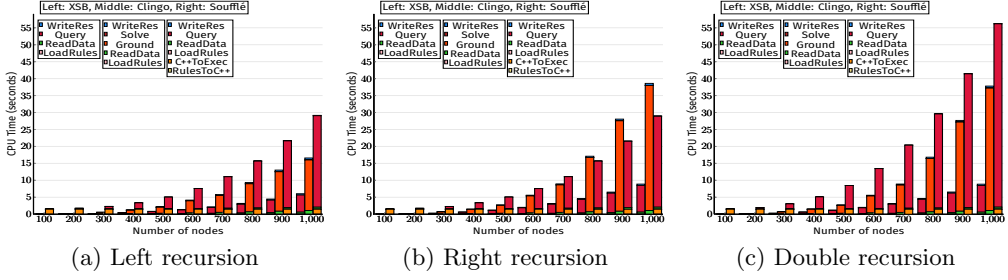


Fig. B3: Running times on MaxAcyc graphs, up to 1000 vertices, 499500 edges.

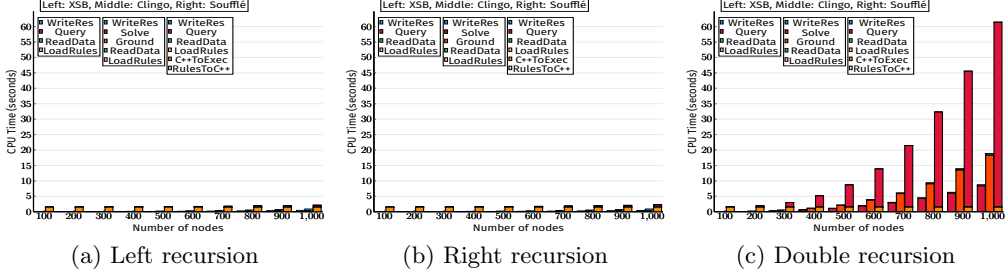


Fig. B4: Running times on Path graphs, up to 1000 vertices, 999 edges.

all three recursion types have the same complexity, but Fig. B1 shows it is only partially the case, and the difference confirms the analysis in Section 4.2:

- For both XSB and clingo, right and double recursions have about the same running times, but slower than left recursion (by about 40% for XSB and 220% for clingo). The exceptions are actually consistent with the analysis in Section 4.2: left recursion is faster because both consider hypotheses of rules from left to right, and clingo is slower because it uses ground rules and semi-naïve evaluation.
- For Soufflé, left and right recursions have about the same running times, consistent with the analysis in Section 4.2, but double recursion is twice as slow. Indeed, with semi-naïve, considering all new `path` facts in each iteration for double recursion has to essentially process each `path` fact twice, compared with only once for left or right recursion.

Examining Figs. B2 to B4 for other graph types shows and confirms additional details.

- For `cyclen`, Table 3 says that double recursion has the same complexity as `Cmp1n`, whereas left and right recursions are a linear factor faster. Fig. B2 indeed supports those for XSB and Soufflé, compared with Fig. B1 for `Cmp1n`. For clingo, it is faster in Fig. B2(c) than in Fig. B1(c), an effect of grounding over sparser input.
- For `MaxAcycn`, Table 3 says that all three recursion types have the same complexity, about 1/6 of that of `Cmp1n`. Fig. B3 confirms that for XSB and Soufflé, including the constant 1/6. For example, Soufflé on double recursion takes about 54 seconds (55.5-1.5) at 1000 vertices here, and about 322 seconds (323.5-1.5) in Fig. B1. For clingo, slightly sparser graphs here have slightly less running times than in Fig. B1.
- For `Pathn`, the relationship of Fig. B4 to Fig. B3 is exactly similar as the relationship of Fig. B2 to Fig. B1.

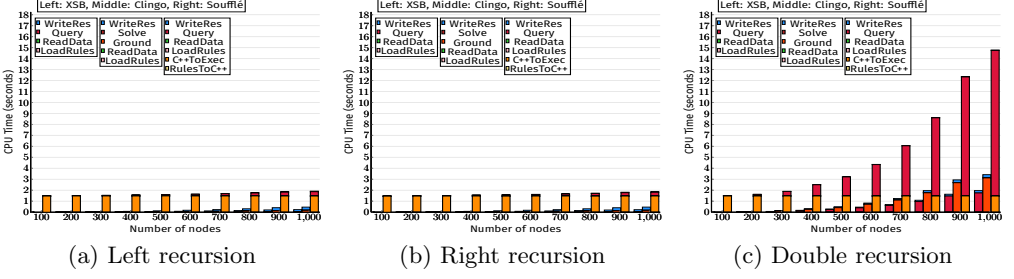


Fig. B5: Running times on `Grid` graphs, up to $\lfloor \sqrt{1000} \rfloor^2$ ($= 961$, closest from below to form a grid, and calculated this way for all x-axis labels) vertices, 1860 edges.

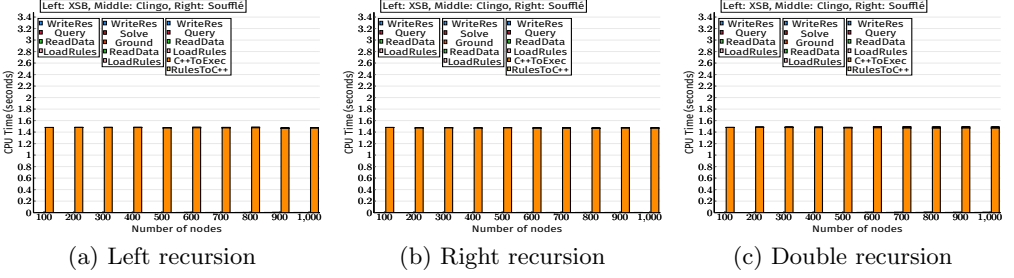


Fig. B6: Running times on `BinTree` graphs, up to $2^{\lfloor \log_2 1000 \rfloor} - 1$ ($= 511$, closest from below to 1000 to form a binary tree, and calculated this way for all x-axis labels) vertices, 510 edges.

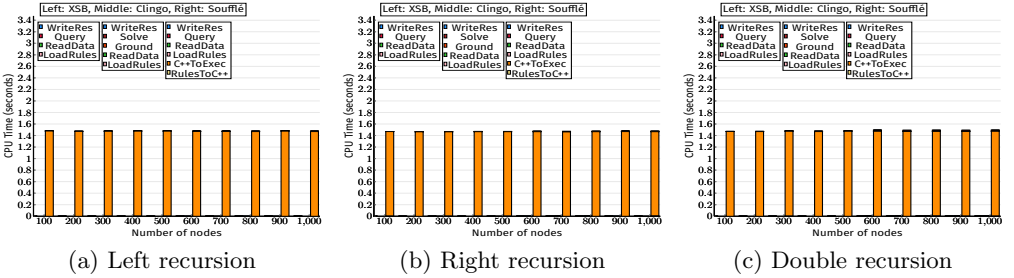
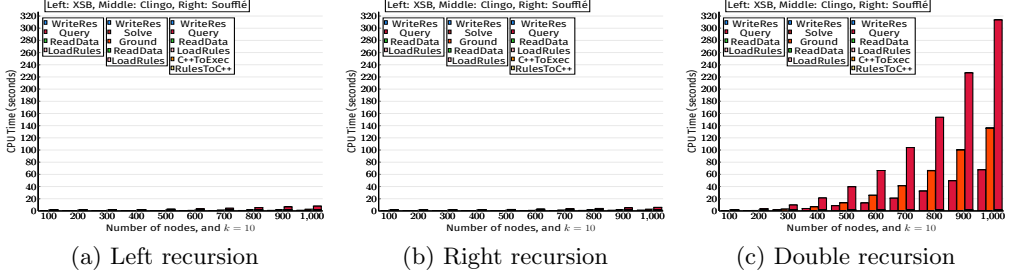
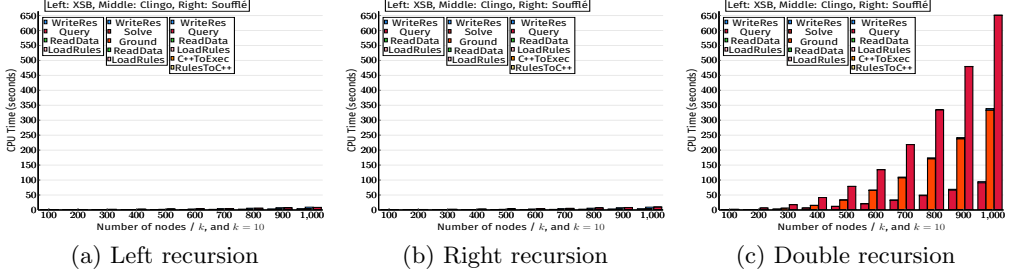


Fig. B7: Running times on `BinTreeRev` graphs, up to $2^{\lfloor \log_2 1000 \rfloor} - 1$ vertices, 510 edges.

Similar analysis as above applies to `Gridn`, where n is (the closest from below to) the square root of the number of vertices. The running times are shown in Fig. B5. The main difference from first four graph types is that the inference times are noticeably smaller, and that other times besides inference times are more visible. This is due to much shorter paths to follow (shorter than the square root of n).

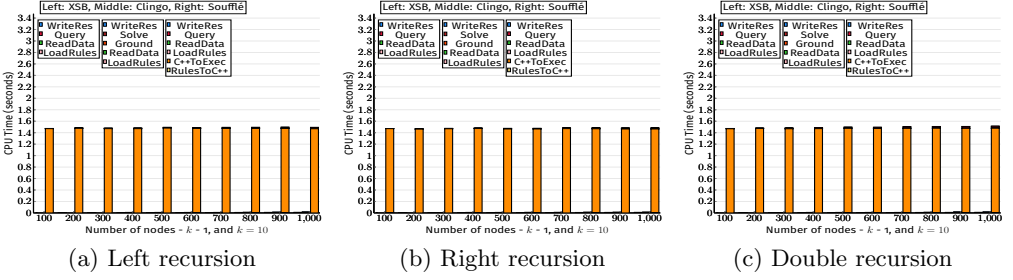
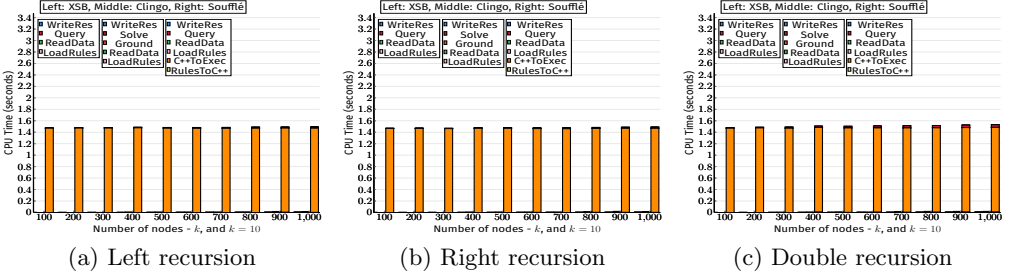
Similar analysis applies also to `BinTreen` and `BinTreeRevn` with parameter n for level of nodes, except that paths are even drastically shorter (logarithm of the number of nodes). The running times, shown in Figs. B6 and B7, are so small that only the compilation time shows distinctively. We also experimented with trees with drastically more nodes and observed that the running times increase explosively with respect to n , and that both tree types have about the same running times, all consistent with analyzed.

Fig. B8: Running times on *CycExtra* graphs, up to 1000 vertices, 11000 edges.Fig. B9: Running times on *PathDisj* graphs, up to 10000 vertices, 9990 edges.

Graph types with two parameters. The remaining five graph types—*CycExtra*_{*n*,*k*}, *PathDisj*_{*n*,*k*}, *X*_{*n*,*k*}, *Y*_{*n*,*k*}, and *W*_{*n*,*k*}—have two parameters *n* and *k*. We discuss results with two sets of experiments, both with increasing *n* up to 1000: (1) with constant *k* = 10 for all five graph types, and (2) with increasing *k* = *n* for the last three graph types. The latter helps greatly in showing the different impacts on performance by the last three graph types. It also led us discover an XSB performance bug in the simplest case of left recursion on *w* graphs.

Figs. B 8 to B 12 show the running times for all five kinds of graphs for constant *k* = 0. They confirm the analysis as before and additionally the constant factor differences except for what we believe is a good cache locality.

- Comparing Fig. B 8 for *CycExtra*_{*n*,10} with Fig. B 2 for *Cyc*_{*n*}, we can see they are similar for double recursion, consistent with the analysis in Section 4. Running times for left and right recursions are too small in these figures, but they generally are about 4-5 times for XSB and Soufflé and about 3 times for clingo (e.g., from 0.10 to 0.45 seconds for XSB, from 0.66 to 1.85 seconds for clingo, and from 1.28 to 6.17 seconds for Soufflé, for left recursion on 1000 vertices). These constant factors are smaller than the 10-times-larger analyzed in Section 4. We believe this is due to the good cache locality of 10 added edges starting from a same vertex in *CycExtra*_{*n*,10}, contrasting each edge starting from a different vertex in *Cyc*_{*n*}.
- Comparing Fig. B 9 for *PathDisj*_{*n*,10} with Fig. B 4 for *Path*_{*n*}, we can clearly see that the running times are about 10 times as large for double recursion, again with analysis in Section 4. For left and right recursions, this is true too even though the running times are again too small to see in the figures (e.g., from 0.05 to 0.59

Fig. B10: Running times on x graphs, up to 1011 vertices, 1010 edges.Fig. B11: Running times on y graphs, up to 1010 vertices, 1009 edges.Fig. B12: Running times on w graphs, up to 2000 vertices, 10000 edges.

seconds for XSB, from 0.33 to 3.78 seconds for clingo, and from 0.60 to 6.31 seconds for Soufflé). Indeed, each $\text{PathDisj}_{n,10}$ graph is exactly the same as 10 Path_n graphs.

- Figures for the last three graph types show that the running times for $k = 10$ are all so small that the compilation time stands out again. But those small running times do show different coloring, prompting the next set of experiments.

Figs. B13 to B15 show the running times for the last three graph types for $k = n$, i.e., k increases as n increases. Fig. B14 for $y_{n,n}$ can be analyzed similarly as for Path_n , but the most interesting cases are Figs. B13 and B15 for $x_{n,k}$ and $w_{n,k}$, respectively.

- Fig. B13 for $x_{n,n}$ shows two new aspects clearly: (1) `WriteRes` times (in blue) are totally obvious, and they are much larger for XSB and clingo than for Soufflé; and (2) `solve` times (in darker red) in clingo are obvious on top of `Ground` times (in lighter red).

Fig. B13 also shows an exception from before: inference time for left recursion got worse, relative to right recursion, for all of XSB, clingo, and Soufflé. XSB is about

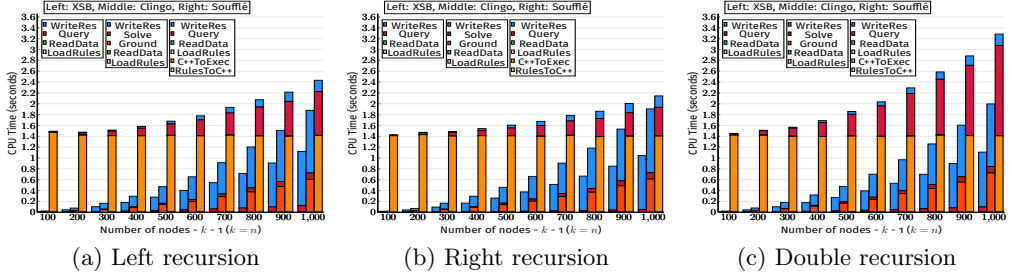


Fig. B 13: Running times on x graphs, up to 2001 vertices, 2000 edges.

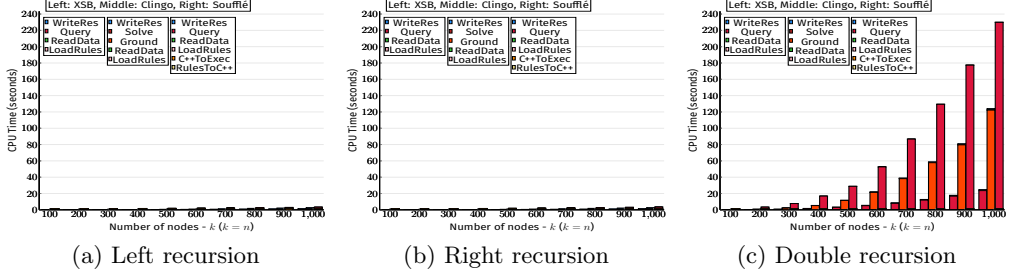


Fig. B 14: Running times on y graphs, up to 2000 vertices, 1999 edges.

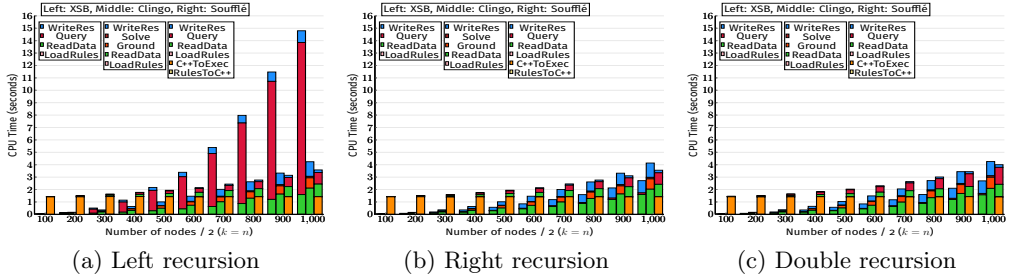


Fig. B 15: Running times on w graphs, up to 2000 vertices, 1000000 edges.

150% slower, instead of faster; clingo is about 1% faster, instead of much faster; Soufflé is about 50% slower, instead of the same. Note though that these variations are relatively very small compared with the times for writing results by XSB and clingo and compilation by Soufflé.

The exception is due to the specifics of x graphs having only paths of lengths 1 and 2, where faster table lookups for `path` pairs in left recursion is overwhelmed by the sheer number of them that fail anyway. In particular, $x_{n,n}$ has $\#edge = 2n$ and $\#path = 2n + n^2$. Since XSB and clingo consider hypotheses from left to right, left recursion looks up $2n + n^2$ `path` pairs first, and only n of them succeed, whereas right recursion looks up $2n$ `edge` pairs first, and n of them succeed. We believe this difference in the number of lookups is also the case for Soufflé. Optimal computation with minimum increment can achieve the same efficiency for all recursion types.

- Fig. B 15 for $w_{n,n}$ shows one more new aspect: `ReadData` times (in green) are totally obvious, and they are about twice as high for XSB and clingo than for Soufflé.

The biggest surprise is the discovery of an XSB performance bug: The running times of XSB on left recursion grow drastically higher than clingo and Soufflé and than analyzed. This is actually the simplest case among all graph types we consider, where `path` is exactly `edge` with no other paths at all.

This performance bug was confirmed by the XSB team. They then analyzed and determined that it is due to hash collisions in this case. They also showed that additional, more sophisticated indexing directives in XSB could be added to the program to bypass this bug.