Using the Monet database system as a platform for graph processing
Inf/Scr-10-15

Steven Woudenberg (0308943)

November 15, 2010

Contents

Abstract

Graph theory is a specific part of computer science and is about finding shortest paths, maximum flow, matchings, coloring, etcetera. For all of these problems dedicated algorithms are created to solve the problem as efficient and as fast as possible. These dedicated algorithms use a main memory data structure which stores the graph and are easy accessible. The algorithm together with the data structure can efficiently perform actions on the graph in order to find the answer to a graph problem, for example the shortest path.

The actions performed on the data structures are implemented, either by the programmer himself or by the creators of the programming language the algorithm is written in. It is possible to use a database system as data structure but because operations on a database are usually more slow than operations on a main memory data structure this will not yield good results. The Monet database system is a database system which achieves great performance improvements compared to other database systems like PostgreSQL and MySQL because of the use of BATs and because of the use of strategies to use the available hardware as much as possible.

Monet is especially good in performing operations on very large sets of data: the more nodes and edges a graph contains, the better Monet should perform in comparison with other database systems and in comparison with dedicated main memory algorithms. The shortest path algorithm will be, with help of Datalog and it's Magic sets, transformed into an algorithm which uses the Monet database system as data structure and SQL queries to execute the algorithm.

Unfortunately the dedicated main memory Dijkstra algorithm outperforms the Monet algorithm, which actually only finds graph connectivity, with a factor of twenty. Other shortest path problems will also not yield very promising results. In order for a Monet algorithm to approach the performance of a dedicated main memory algorithm the problem must be solvable by an algorithm based on the breadth first search strategy. The problem which will most probably yield the best results is the problem where the bipartiteness of a graph must be checked. When random graphs are generated good performances are achieved but the characteristics of the graph are very important with regards to the performance: the number of edges, the number of nodes and the way edges are created.

1 Introduction

Nowadays it is almost impossible to picture a world without navigation systems and planning software. These systems rely heavily on the performance of network algorithms. The most important algorithms used are the algorithms to find the shortest path from a starting point to a destination. In 1959 Dijkstra thought of the Dijkstra algorithm [11] which is used in a lot of planning programs even up till today. A lot of improvements and adjustments are implemented to create faster algorithms or to improve the performance of the algorithm for a specific use.

When edges in a network, or graph, have negative weights and a shortest path must be found, the Dijkstra algorithm will not yield a correct result but the Bellman-Ford algorithm does. There are many other algorithms to find a shortest path from a starting point to a destination or from one starting point to all possible destinations or from all possible starting points to all possible destinations. Network algorithms are still often used and the performance is very important: both the correctness of the shortest path and the time it takes to find the shortest path.

The Monet database [3] is a database system which claims to achieve significant speed improvements by comparison with other database systems. For example a normal database system will take quite a bit of time when performing a join, especially when very large tables have to be joined. The Monet database system is very different from the normal database systems: the join is one of the basic and most used operations and can be performed very fast by fully utilizing the capacities of the main memory and the processor [15].

The use of a graph is inevitable when performing a network algorithm. The most common way to use a graph is to first put all the information in a data structure, preferably in the main memory, after which the algorithm will be applied. The way the data structure is implemented, used and stored is very relevant with regards to the runtime of the network algorithm and therefore these choices have to be made with caution.

The general idea described in this paper is to store the entire graph in a Monet database and perform queries on this database which must yield the same results as the network algorithms. To achieve this the actions performed by the network algorithms must be translated into database queries and the runtime of the algorithms must depend on the database performance as much as possible. By doing so, all the benefits and performance improvements of the Monet database system are used to increase the performance of the network algorithms as much as possible.

Joins and unions are database operations which can be performed very efficient by the Monet database system. DataLog [8], [16] is a rule based database language which uses the join and union operations a lot in order to calculate the result of a query. This combines very well with the Monet database system. The general idea described in this paper was not yet complete: the queries which will be used will be based on the way DataLog creates and executes queries.

DataLog itself can be optimized by using Magic Sets [8], [17]. These Magic Sets can also be used for the DataLog which is used to apply the network algorithms on the Monet database system. This might yield much better results compared to applying DataLog without Magic Sets. With the use of Magic Sets the performance of the combination of the Monet database system and DataLog would yield good performances and might compete with other implementations of network algorithms.

1.1 Definitions

This section will define some of the terminology which will be used throughout this paper.

Monet This is the short name of the Monet database system. When 'Monet achieves very good results' it means that the Monet database system yields good results.

Performance The performance of a database consists out of two parts: whether or not the result returned by a database system is correct and the time it took for the database system to calculate this result. In this paper it is taken for granted that the found results are correct, so the performance solely depends on the time it will take a database system to find the correct result. The faster the final result is found, the better the performance.

The Benelux map In this paper the roadmap of the Netherlands, Belgium and Luxembourg (hence BeNeLux) is used to test the algorithms. This roadmap only contains edges with positive costs. In total 1.598.250 nodes are present and they are connected with 3.756.335 directed edges.

2 Monet

This section is about the Monet database system. First the difference between Monet and other open source database system will be discussed. Then it will be explained why the use of Monet might yield good results when it is used as the foundation of network algorithms.

2.1 The advantages of the Monet database system

Databases became more and more important in the computational world. Nowadays databases are used in a wide range of different applications. Fairly simple applications are employee databases or the digital yellow pages. These databases are used to store a massive amount of data which has to be accessed in an structured way. A typical operation on such databases is the selection of a small amount of records, for example requesting the telephone number and address of a specific individual or requesting information about all employees who started their job in the year 2007.

The first relational database management systems were developed in the 1970's and since then a lot of progress have been made in boosting the performance of these databases. Especially when data mining became more and more important. Data mining is the principle of the extraction of patterns out of big amounts of data. Two good examples of data mining are the Albert Heijn bonus card and a large part of the field of molecular biology. Albert Heijn uses the bonus card to keep track of everything their costumers buy and all this information is stored in a very big database. Albert Heijn applies data mining on this database to determine which products are popular at the moment or to determine which products are one way or another related to each other. With the results of the data mining Albert Heijn can adjust their commercials and marketing campaigns so they will have more effect and Albert Heijn can make more profit.

In biology there are up until today a lot of uncertainties with regards to the DNA and the effects changes of the DNA has on the chances to develop diseases like cancer. Data mining performed on huge databases with DNA data can help the biologists to determine the exact connection between DNA and for example cancer.

With the upcoming data mining needs, better database support was needed. The queries executed by the data mining programs were much more demanding and it took a lot of time to perform these queries. Monet is one of the database systems which is especially developed to perform large queries on huge amounts of data as fast as possible. Both algorithmic and hardware optimizations are used by the developers of Monet to increase the performance.

The bottleneck of performing queries for most database systems are the main memory and the cpu. These are the main points of improvement and the developers of Monet have found interesting ways of increasing the performance by improving the use of the main memory and the cpu. The optimization which is most characteristic for Monet is the use of BATs: Binary Association Tables. A classical database table consists of multiple columns and a lot of rows where a row is the same as a record. In Monet these tables can also be inserted but under the hood these tables are split up into a number of BATs: every column of the original inserted table will be stored in its own table (BAT). This way of storing a table is called vertical decomposing and was originally described in [10].

The main idea behind splitting the tables is that for most queries only a small number of columns of a table are needed to perform the query (like a join or a selection of all persons born on a certain date) and therefore it is not at all required to put an entire table into main memory in order to perform such a query. When a lot of columns of a table have to be returned a lot of joins must be performed on the BATs, but the Monet developers have also been optimizing this operation. Nowadays the amount of main memory has become the bottleneck in the performance

of database systems but by using BATs this bottleneck will become less of a problem [14], [7].

On the website of Monet [3] a performance study is shown: a number of test queries are fired at a Monet database, a PostgreSQL database and a MySQL database which were installed 'out of the box' (so no tweaking configuration files is done). In almost all test cases Monet performs best. Not only does Monet performs best, the difference between the open source databases is quite large: in a lot of test queries the runtime of Monet is less than one tenth of the runtime of PostgreSQL or MySQL. For the performance study the TPC-H benchmark [4] is used, which is a widely accepted benchmark to define the performance of a database system (mostly commercial database systems) against large scale decision support applications. The developers of Monet describe the improvements they have made as *'The operations are cache- and memory-aware with supreme performance'*.

2.2 Why use Monet?

A network, or a graph, is essential for network algorithms and when a network algorithm is executed the graph will be loaded into a specific kind of data structure. Some algorithms use very specific data structures so the algorithm yield a better performance. As an example let's take a look at a small graph with only five nodes and some directed edges (figure 1).

The most straightforward way to store the information of this graph into a data structure is to store each edge together with the distance of that edge. The table where this is done is shown in figure 2.

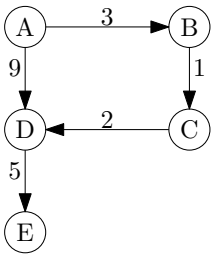

Figure 1: A small graph

source	sink	distance
A	B	3
A	D	9
B	C	1
C	D	2
D	E	5

Figure 2: A simple way of storing a graph

This looks like a simple table which can easily be stored into a database. Usually a database is not used as a data structure because generally it will take more time to retrieve data from the database than it will take to retrieve data from a data structure which is stored in main memory. When using Monet this theory still holds: there is always a (small) overhead in retrieving data from a database. Why should Monet be used then? The idea of only using the database as the data structure is not enough: all the actions performed on the data structure by the network algorithms should also be taken care of by the database system.

So two changes have to be made: store the entire graph in a Monet database and rewrite the network algorithm into SQL. To perform the network algorithm on the graph only the SQL statements have to be fired at the database and the result of the algorithm will be the outcome of the queries. Because all of the progress made during the development of Monet the presumption is that it can yield a pretty good performance when Monet is the database system used in the idea described above.

3 DataLog

This section is about DataLog, a rule base database language. The basics will be explained briefly and after this the Magic Sets will be explained which is the foundation of a faster and smarter way to perform network algorithms on the way suggested in the introduction. Ullman wrote a lot about DataLog: the basics are explained in [16] and the more complex parts, including the Magic Sets, are explained in [17]. A paper which is very usefull to understand the basics and more about DataLog is the paper with the great title 'What You Always Wanted to Know About Datalog (And Never Dared to Ask)' [8].

3.1 Introduction to DataLog

The name DataLog finds it origin in the use of a rule based language such as Prolog on a database system: **Data**base Pro**log**. Prolog is a logic programming language which was born during the creation of a program which should process natural languages [9]. The queries used in Prolog are generally not suggesting how to retrieve the answer of the query; the rules of the program do. The best way to explain how DataLog works is by the use of the example in figure 3.

(1) *sibling(X, Y) :- parent(X,Z) & parent(Y,Z) & $X \neq Y$.*

(2) *cousin(X, Y) :- parent(X,Xp) & parent(Y, Yp) & sibling(Xp, Yp).*

(3) *cousin(X, Y) :- parent(X,Xp) & parent(Y, Yp) & cousin(Xp, Yp).*

(4) *related(X, Y) :- sibling(X, Y).*

(5) *related(X, Y) :- related(X,Z) & parent(Y,Z).*

(6) *related(X, Y) :- related(Z, Y) & parent(X,Z).*

Figure 3: An example of a DataLog program: this specific program consists of six rules.

In the example six rules are specified. A rule can consist of predicates (all the rules) and mathematical expressions (only rule (1) in this example). A predicate is something which can be calculated by using the given rules and by using the information in the database. *parent(X, Y)* is a relation stored in the database and *parent(X, Y)* means that Y is a parent of X; *parent(Luke, Anakin)* means that Anakin is Luke's parent.

Because *parent* is a predicate which is stored in the database it is quite straightforward to calculate results for rule (1). Rule (1) states that when persons X and Y have a parent in common they are siblings of each other. The last part of rule (1), $X \neq Y$, makes sure someone cannot be a sibling of himself. For example the query *sibling(Luke, Leia)* will have 'true' as a result because both Luke and Leia have Anakin as a parent. It is also possible to request all siblings of someone: *sibling(Luke, Y)* will have *Leia* as the result.

In the same way it is possible to ask queries like *cousin(Luke, Y)* which will return all cousins of Luke. The definition of a cousin is stated in rules (2) and (3): two persons are cousins when their parents are siblings or when their parents are cousins themselves. The rules of finding the cousins of someone are very much alike the one where the siblings of someone are found except for the recursive step in rule (3).

Rule (2) can return a result by finding two persons who's parents are siblings of each other, which can be determined by using rule (1). Rule (3), however, needs results found by rules (2) and (3) to find it's results. Because of this all the results for rule (3) cannot be found at once and it will take several iterations to find all results. To show this by an example the family

graph in figure 4 is used. This graph shows that a is a parent of both c and d, c is a parent of f and g, etcetera. The graph is not really representative for the real world because some persons in the graph only have one parent and a lot of siblings have children together...

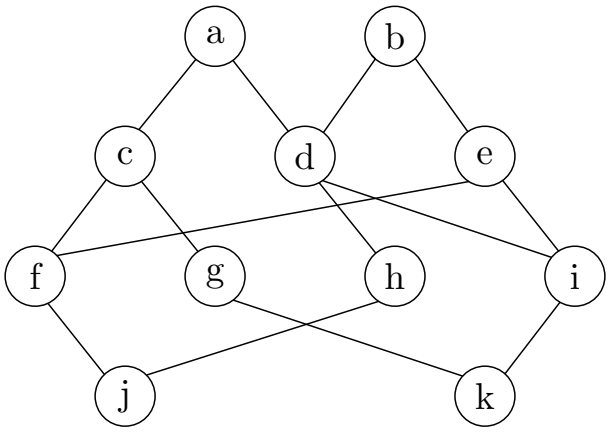

Figure 4: An example family graph with some strange relations

To find all cousins the query *cousin(X, Y)* is requested. Initially the set of cousins is empty so applying rule (3) will not yield any results and rule (2) has to be applied first. The results after applying rule (2) for the first time will be *fh, fi, ii, gh, gi, hi* and *jk*. Applying rule (2) another time will yield exactly the same results because it only depends on the database relation *parent* and the intentional relation *sibling*, which itself only depends on database relation *parent*. After this first iteration some results have been added to the *cousin* relation and therefore rule (3) can be applied. Applying rule (3) yield the results *jj* and *kk*. Again new results for the *cousin* relation have been found so rule (3) should be applied again. No new results will be found in the next iteration so after only two iterations all results for *cousin(X, Y)* have been found.

The same strategy holds for the *related* rule: initially only rule (4) will yield results when applied because the *related* relation is initially empty. After the first iteration it is no use to use rule (4) again because the same results will be found and now only rules (5) and (6) will be used to find new results. The new found results for each iteration are shown in table 1.

Table 1: The new found results of each iteration for calculating *related(X, Y)*

Iteration round	Results found
1	*cd, de, fg, hi, fi*
2	*df, dg, ch, di, ci, eh, ei, gj, fk, hk, ij*
3	*fh, dj, gh, jk, gi, dk, cj, ii, ck, ej, ek*
4	*fj, hj, gk, ik*
5	*jj, kk*

3.2 DataLog essentials

As mentioned in the introduction of this section a detailed description of DataLog can be found in the books written by Ullman [16], [17]. A more basic explanation can be found in [8] but some theory have been very usefull for optimizing the use of DataLog with Monet and with graph

algorithms and will be briefly described in this subsection. A much more specific explanation of all terminology can be found in the books written by Ullman.

3.2.1 Naive and semi naive evaluation strategies

When calculating results for a query the evaluation strategy used is very important with regards to the performance of the evaluation. This is especially true when recursion is involved which was shown for the *cousin* and the *related* relations. As an example the calculation of the query *related(X,Y)* is used. In the first iteration five results have been found: *cd, de, fg, hi* and *fi* and these results will be used by rules (5) and (6) to find other results. The results of the *related* relation increases after the second iteration: *df, dg, ch, di, ci, eh, ei, gj, fk, hk* and *ij* will be added to the results found in the first iteration.

The third iteration will use all found answers for the *related* relation up until this moment to try and find new results. Eleven new results will be found (as can be seen in table 1) but in total 22 results are found. In a more formal way the set of all found results up until a given iteration i is R_i. The set of results found in the first iteration is S_1, the set of results found in the second iteration is S_2 and the set of results found for the i^{th} iteration is S_i. For iteration j it hold that

$$R_j = R_{j-1} \bigcup S_j = S_1 \bigcup S_2 \bigcup \ldots \bigcup S_j \tag{1}$$

When the naive evaluation strategy is used, the first iteration will have S_1 (and therefore $R_1 = S_1$) as a result which only contains results found by rule (4). The next iteration S_2 will be calculated without non-recursive rule (4) because this will only yield the same results, but rules (5) and (6) can be applied using R_1 as the current known results for the *related* relation. $R_2 = R_1 \bigcup S_2$ and for the next iteration R_2 will be used as the current set of results of *related* relation.

The third iteration will yield result set S_3 and $R_3 = R_2 \bigcup S_3$. However, in this iteration a lot of results have been calculated for the second time: all results which are in S_2 will also be in S_3 because of the presence of the results of S_1 in both R_1 and R_2. Actually, because of the definition $R_j = R_{j-1} + S_j$ it holds that: $S_{a-1} \subseteq S_a$ for $a > 2$. The more iteration are calculated the more results will be calculated multiple times. Even worse: the results calculated in the second iteration will be calculated $j - 1$ times when there are j iterations.

To prevent the recalculation of results and thereby decreasing the amount of time and effort put into finding new results, the semi naive evaluation strategy can be used. As opposed to the naive evaluation strategy the semi naive evaluation strategy tries to minimize the number of times an earlier found result will be used in finding new results. The most basic way to do so is to only use the results found in the last iteration to find new results in the next iteration.

Equation (1) will remain the same but to calculate iteration i, R_{i-1} will not be used, but S_{i-1} will be used: only the results found in the previous iteration. All earlier results are sure to find results which have already been found in a previous iteration.

The name of this method is not without reason 'semi naive': it is far less naive than the naive evaluation strategy (hence the 'semi') but it is still quite naive in the same way the naive evaluation strategy is naive: some results will be calculated more than once while this is not at all required. It is easily imaginable that there are several ways a result can be calculated in a large family graph and that the different ways reach the result in a different iteration: this result will be used multiple times to find new results. It is possible to prevent this and create a less naive evaluation algorithm but this implies extra calculations to decide whether or not a result has already been used in an earlier iteration or not. This problem will be discussed further in section 4.2.

3.2.2 Evaluation approaches

There are basically two general ideas how an evaluation can be executed: top down and bottom up, also named forward chaining and backward chaining respectively. Both algorithms have their advantages and their disadvantages which will be briefly discussed. The basic idea of the bottom up approach is to start the evaluation as discussed in 3.1: simply start calculating results until the requested result is found. When the query *related(f,k)* is requested the top down approach will start by using rule (4) to find some results and then iteration after iteration rules (5) and (6) will be applied until the result *fk* is present in the result set.

The basic idea of the top down approach is not to blindly start by applying rules to find some results but the query is used to define what to do next. When the query *related(f,k)* is requested the query will be matched to the heads of all present rules to define which queries are relevant. In this example rules (4), (5) and (6) are relevant because they all have the head *related(X,Y)*. Now first rule (4) will be applied with the given constants and rule (4c) is created:

(4) *related(X,Y) :- sibling(X,Y).*

(4c) *related(f,k) :- sibling(f,k).*

sibling(f,k) in its turn will be matched with rule (1) and the result of the query will be empty because f and k are not siblings of each other. Because of this rule (4c) will also yield no results so rule (4) cannot lead to results to the query *related(f,k)*. Fortunately rules (5) and (6) can also be used and first rule (5) will be applied with the given constants:

(5) *related(X,Y) :- related(X,Z) & parent(Y,Z).*

(5c) *related(f,k) :- related(f,Z) & parent(k,Z).*

In order to find out if this rule can be made true the queries *related(f,Z)* and *parent(k,Z)* have to be calculated. Because *related(f,Z)* is the first subgoal of the rule it will be evaluated first and therefore the 'new' query is *related(f,Z)* and the rules for which the head matches are rules (4), (5) and (6). When rule (4) is applied some results are found: all siblings of f so the results are *fh* and *fi*. In rule (5c) these results can be used to find if *related(f,k)* can be made true. In order to be true the value for Z must be h of i and therefore *parent(k,h)* or *parent(k,i)* must be true. Fortunately *parent(k,i)* is true so it is found that *related(f,k)* is true. When both *parent(k,h)* and *parent(k,i)* would have returned false, rule (5) would be applied on *related(f,Z)* to find results and these results would have been used to try and make rule (5c) true, etcetera.

Both approaches have some advantages and some disadvantages. The major disadvantage of the bottom up approach is the possible evaluation of a lot of useless results. This is, however, the big advantage of the top down approach: only results which are relevant will be evaluated and used to try and find the results of a query. Because the speed of which the answer to a query is found is the most important property, next to the correctness of the answer, these properties would plead to use the top down approach. The top down approach also has one major disadvantage however: it is not certain that the correct result will be found, even if the result exists... Because the order of the rules and the order of the subgoals define the evaluation order it can occur that some rules will never be used to find new results while these rules are the only rules which can find the result. In the example rules (5) and (6) both are recursive and rule (6) will never be reached. In some situations it will happen that a rule will never, ever be reached because of the recursion of the rules. On the contrary the bottom up approach will always find all results.

It is very hard to decide which approach to use because they both have a major advantage and a major disadvantage on the two most important performance measures: the correctness of the result and the amount of time it will take to find this correct result.

3.3 Magic Sets

Chapters twelve and thirteen of Ullman [17] discuss the top down and bottom up approaches in detail. Both the exact algorithms and all needed sub algorithms are discussed. The question is raised how to create a good approach so the advantages of both the top down and the bottom up approach are used: the fastest way to find the correct result of a query. The conclusion is found in chapter thirteen and the name of this conclusion is Magic Sets.

3.3.1 Rule/goal tree approach

Some new concepts have to be introduced as well as an 'in between solution': rule/goal trees. The rule/goal tree is an approach where the root of the tree is the requested query and its children are the rules which are relevant to the query. The children of these rule nodes are the subgoals of the rules, whose children will be the rules which are relevant to the goal, etcetera: hence the name rule/goal tree.

To give an intuition how the rule/goal tree works the rules in figure 5 will be used. The same database is used as in the earlier examples, which contains the *parent(X, Y)* relation.

(7) *anc(X, Y) :- parent(X, Y).*

(8) *anc(X, Y) :- parent(X,Z) & anc(Z, Y).*

Figure 5: DataLog rules defining ancestors (anc in the rules).

When the requested query is *anc(k, Y)* this will be the root node of the rule/goal tree. Because both rules (7) and (8) are relevant the root node will have two children which are the rules which are already preprocessed just like it was done in the top down approach:

(7c) *anc(k, Y) :- parent(k, Y).*

(8c) *anc(k, Y) :- parent(k,Z) & anc(Z, Y).*

First the results for rule (7c) will be calculated which is done with help of the database relation *parent*. After these calculations the left branch of the tree will end. The right branch consists of rule (8c) which has two subgoals. As was mentioned before, the children of this node will be the subgoals of the rule so two children are created: *parent(k,Z)* and *anc(Z, Y)*. After this, first the results for the first child will be calculated and these results will be used to calculate the results for the second child. Because both subgoals contain the variable Z and the left child has already found all possible values for Z which are allowed by it, these are the only values which should be used in trying to find results for the right child. Mind that Z is a set of possible values and not exactly one constant. All possible values Z can have after evaluating the first child will be passed on to the right child.

So the root node is *anc(k, Y)*. The left child is rule (7c) which will find all parents of k which are both g and i as can be seen in figure 4. These results will be passed on to the root node and will be part of the final result. The right child is rule (8c) with its two subgoals which will be its children. The left child is *parent(k,Z)* and the right child is *anc(Z, Y)*. The left child will find the same results as rule (7c): g and i. This information is passed on to the other subgoal: it is only usefull to find results for *anc(Z, Y)* where $Z = \{g, i\}$. The first levels of the rule/goal tree of query *anc(Z_0, W)* is shown in figure 6 where Z_0 is a set which only contains k in the example and W is a set of variables, just like in the example.

The evaluation of queries with the rule/goal tree approach will be done one level at a time. Because of this strategy the possibility of getting lost in recursion and never finding the correct

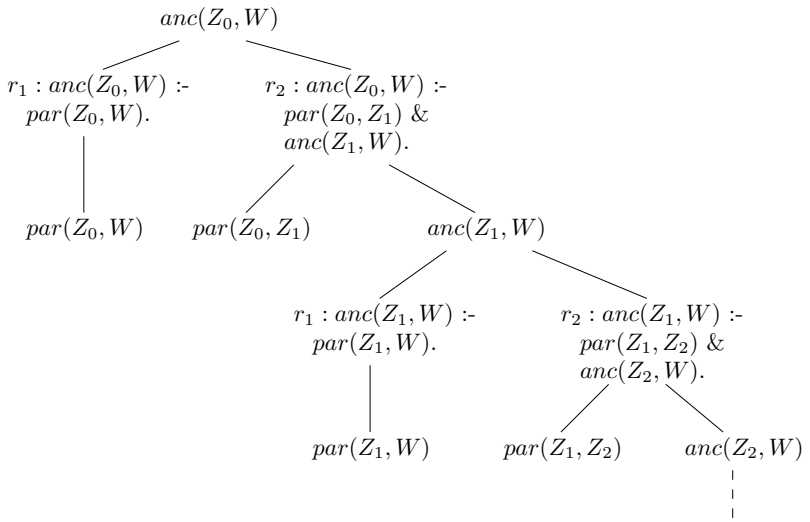

Figure 6: The first levels of the rule goal tree for query *anc(Z_0, W)*.

result does not longer exist. The rule/goal approach also uses the constants in the query and earlier found results to try and push the search in the right direction. The technique of passing found information from one branch to another is called 'sideways information passing'. This is a very important technique to make sure as less irrelevant evaluations are done as possible. In the example the results of the left subquery resulted in results g and i which are passed on to the right subquery. This shows the sideways information passing perfectly.

An important aspect when using the rule/goal trees is the order of the subgoals of a rule. In the example the information found in the first subgoal of rule (8c), *parent(k,Z)*, finds some results and passes these results on to the second subgoal. What would happen if rule (8c) would swap the subgoals and rule (8d) is used in the rule/goal tree approach?

(8d) *anc(k,Y) :- anc(Z,Y) & parent(k,Y).*

Now first the subgoal *anc(Z,Y)* will be calculated which does not contain any constants so it will try to find all ancestors of everybody. With this result *parent(k,Y)* will be evaluated. It is not hard to see that the use of rule (8d) instead of rule (8c) will lead to a far worse performance because the search space is not yet limited. The order of the subgoals is very important when the time for evaluating a query should be minimal.

3.3.2 From rule/goal trees to Magic Sets

An improvement on the rule/goal tree approach is the QRGT: the Queue-based Rule/Goal Tree expansion. When new node is expanded in the rule/goal tree, its children will be added to the queue and after the expansion of a node the first element of the queue will be used to continue the algorithm. Because of the use of the queue the algorithm becomes more a breadth first search algorithm than a depth first search algorithm in order to try and combine the advantages of both the bottom up approach and the top down approach.

Unfortunately the rule/goal tree approach and the QRGT approach have some major disadvantages: it is hard to define the termination criteria and a lot of queries are evaluated more than once. Actually it is very difficult to define the termination criteria: when can the algorithm

be certain all possible results have been found? This is easy for a query which only contains constants but when variables are involved it is very hard.

The Magic Set approach is a very good approach: it will only perform evaluations when they are relevant to the query, all results will be found and the definition of the termination criterion is very simple. The ideas used by the rule/goal tree and the QRGT will be used in the Magic Set approach: especially the sideways information passing is essential.

To incorporate all these functionality into the Magic Set strategy a new set of rules will be created from the original set of rules and the requested query. When the original query is fired at the new set of rules the normal way DataLog calculates results is used to evaluate the query: the semi naive evaluation strategy. So instead of creating a complicated approach the rules themselves will be changed. Ullman [17] describes the steps how to generate the new set of rules from the original set of rules and the query. The new set of rules of the ancestor example (figure 5) and the query *anc(k,Y)* is shown in figure 7.

(m1) $m_{anc}(X)$ *:-* $sup_{2.1}(X, Z)$*.*

(m2) $sup_{1.0}(X)$ *:-* $m_{anc}(X)$*.*

(m3) $sup_{2.0}(X)$ *:-* $m_{anc}(X)$*.*

(m4) $sup_{2.1}(X, Z)$ *:-* $sup_{2.0}(X)$ *& parent(X,Z).*

(m5) *anc(X,Y) :-* $sup_{1.0}(X)$ *& parent(X,Y).*

(m6) *anc(X,Y) :-* $sup_{2.1}(X, Z)$ *& anc(Z,Y).*

(m7) $m_{anc}(k)$*.*

Figure 7: The new set of DataLog rules according to the transformation for the use of Magic Sets.

The sup-rules ((m2) through (m4)) are used for the sideways information passing, rules (m5) and (m6) are the original rules including sup-rules so sideways information passing is actually performed. Rules (m1) and (m7) are the Magic rules: they define the Magic Set. Because of the use of this Magic Set only results which are relevant will be used to calculate new results. The sup-rules are created as supplementary relations. As mentioned before the sideways information passing will pass the results found by one subgoal to the next. The $sup_{1.0}$ and $sup_{2.0}$ rules are created to incorporate the Magic sets in the sideways information passing. The $sup_{2.1}$ rule is created because rule (8c) has two subgoals and the results found by the first subgoal must be passed on to the second subgoal: the results found by $sup_{2.0}$ will be passed on to the subgoal *anc(Z,Y)*. The termination criterion with the use of this new set of rules is very simple: when in one iteration no new results are found, all possible answers have been found and the result for the initial query has been found.

There are some constraints on the use of Magic Sets: it is very important to use rectified rules as original rules. Rectified rules make sure that when multiple rules have the same predicate name in the head, the arguments of the heads corresponds to each other. Every head must have the same number of arguments and all of the arguments should be unique. When this is not true it can easily happen that the QRGT will outperform the Magic Set approach. Actually Magic Sets will perform **at least as good** as QRGT on the original set of rules when the rules are rectified and the order of the subgoals is thought through.

When taking a look at the rules in figure 7 it is clear that a few optimizations are possible. For instance, when rule (m5) is evaluated it uses rule (m2) which will only be 'forwarding' to rule (m1). Rule (m2) is not used anywhere else so the rule is actually not needed at all. The same holds for rule (m3). Some other optimizations can be done to create a new set of rules

which is as compact and fast as possible. When removing rules (m2) and (m3) the references to $sup_{1.0}(X)$ and $sup_{2.0}(X)$ must be replaced by $m_{anc}(X)$.

The name of the Magic Set approach is because of the use of the Magic Set which in the example is m_{anc}. This is actually just a single column table which contains the relevent values found up until a certain moment. When the ancestors of k are requested the Magic Set initially contains only k. Next the direct ancestors of k are found: g and i and they will be added to the Magic Set. Next the direct ancestors of g and i (the grandparents of k) are found: c, d and e. They will also be added to the Magic Set, etcetera. Every 'iteration' only the values in the Magic Set will be used to find new ancestors so only relevant results will be found (which is the big advantage of the top down approach). The ancestor example is maybe not the best example because here **only** relevant results are found and used to find new results. In section 4 the Magic Set technique will be applied to the shortest path problem and there not **only** relevant results will be found.

3.4 Working with Datalog, Magic sets and SQL

Initially the project started by translating Datalog queries into SQL queries which would be executed on a Monet database. In order to understand what Datalog is about and how it performs queries in detail the books of Ullman [16], [17] were very helpfull. This section explains Datalog very briefly in comparison with how much time it took in order to fully understand all techniques of evaluating Datalog queries. In the first stage of the project a translator was build to translate Datalog queries into SQL queries. This is much harder than it sounds: there are a lot of special cases and exceptions which must be taken into account.

3.4.1 Translating Datalog to SQL

The family rules from figure 3 were used as main test cases. Rule (1) is not hard to implement, although this was the first step to define a standard translating schedule: how should the basic parts of an SQL query look like? The query for rule (1) will be:

SELECT p1.child, p2.child FROM parents p1, parents p2 WHERE p1.parent = p2.parent AND p1.child <> p2.child

The 'FROM' part has always been pretty trivial: all subgoals from the body of a Datalog rule must be presented by a table in the 'FROM' part. The 'SELECT' part also seems quite trivial: simply select all variables which are present in the head of the Datalog query. All equations and inter-subgoal conditions must be taken care of in the 'WHERE' part of the SQL query.

These standards hold for most of the time for most queries. It will become more difficult when the query *sibling(k,Y)* is requested: should k be in the result of should only the siblings of k be in the result? It is not difficult to define the selection on the query but it is of no use to keep irrelevent columns in the result. Whether or not a column is relevant actually depends on the queries which will be applied after the current one. When only the siblings of k are requested it is clear only the siblings have to be shown. When the query is used as a subquery to calculate a cousin relation, it is not always clear which columns have to be shown.

Initially no materialization was realized: when a query like *cousin(f,Y)* was requested an SQL query would be created where the result of rules (2) and (3) would be combined by the 'UNION' operation. Even worse: the *sibling* subgoal would be calculated in the 'FROM' part of the *cousin* query. A lot of recursion will happen because of the properties of Datalog and the family rules, so the queries became larger than the Monet database system could handle. As an example rule (2) would become:

SELECT p1.child, p2.child FROM parents p1, parents p2, (SELECT p3.child AS X, p4.child AS Y FROM parents p3, parents p4 WHERE p3.parent = p4.parent AND p3.child <> p4.child) AS sibling WHERE p1.parent = sibling.X AND p1.parent = sibling.X AND p2.parent = sibling.Y

Here the *sibling* relation is evaluated as a table in the FROM part of the *cousin* query. When rule (3) is created the entire query of rule (2) would become a part of the FROM part which will lead to a large query. Because rules (2) and (3) combined are the final result of the *cousin(X,Y)* query the two SQL queries must also be combine by a UNION operation. When more recursion is used the size of the queries will increase very fast.

When materialization is used it is still not trivial to define the elements which should be the result of a query or of a subquery. It will most probably yield a bad performance when a lot of columns are materialized which will not be used again. Not only the rules from figure 3 are used to test the translator but the *same generation* rules too, which are shown in figure 8.

(sg1) *sg(A,A) :- person(A).*

(sg2) *sg(A,B) :- parent(A,C) & sg(C,D) & parent(B,D).*

Figure 8: Datalog rules to define whether or not two persons are of the same generation.

Rule (sg1) became a problem: how should this be translated? Should rule (sg1) only be applied when both arguments are the same or can it also be applied when one argument is a constant and the other is a variable? This was just one of the more complicated aspects which had to be taken care of. Sometimes rules were used which did not have a body: they simply were the head of a rule like rule. Such rules are also found when the Magic set approach is used: the Magic set itself is such an example, rule (m7). These rules are actually not implicit relations but should be stored in a table in the database, just like the *parent* relation.

The real translation from Datalog subgoals to SQL is also not as easy as it seams. All arguments of Datalog queries and of subqueries must be translated into SQL tables and especially into the specific columns of these tables. For every Datalog query a data structure is created to be sure this translation works fine and the arguments of Datalog queries correspond to the correct tables and columns of the Monet database. The best way to solve the table reference problem was to give every table in the 'FROM' part of the query a new name and use this name in the rest of the SQL query.

The most complicated part was the recursion. Especially when materialization was not yet used it was very complicated to put all theory into a program which yields correct results. The debugging of this program was very hard because the recursion was spread throughout three different methods and the debugging also seemed kind of a recursive activity... Applying all exceptions and special cases in these recursive steps took much time and effort.

3.4.2 Implementing the Magic sets

The general idea of the use of a Magic set was not very hard to grasp but how it worked exactly was very complicated. After reading the books of Ullman it became more clear but it took a couple of weeks until the Magic set translation program was finished. The program is capable of translating a wide range of different Datalog rules into a set of rules which can be used by the Magic set approach. After rereading the second book it became even more clear.

In the second book, [17], Ullman describes how the rules for the Magic set program can be created out of the normal Datalog rules. The rules which are created will be divided into five

different groups: the Magic rules, the basic supplementary rules ($sup_{1.0}$, $sup_{2.0}$, etcetera), all other supplementary rules, the transformed original rules and the initial values of the Magic set. For each of these groups a mathematical description is given how the rules should be created but not all of the parts were very clear. After a while the solution was not to create the rules in order of their groups but first create the rules in group two, then all rules of group three, then group one rules, group four rules and finally all rules of group five. The heads of group two rules are needed in order to create group three rules and the heads of group three rules are needed in order to create group one rules.

After this problem was solved some other problems arose. When a query is requested without any contants as arguments, what would the Magic program look like? Actually this problem was very easy: without any known constant it is of no use to create a Magic program. When one of the arguments of a query is a constant the Magic program can be created but when more arguments of a query are constants it is much more difficult to create a Magic program. This problem is also briefly mentioned in section 4.3. Sometimes it might yield good results when one Magic set is created with multiple constants but with the problems described in this paper it will not.

Because materialization had to be used in order to keep the queries processable and a Magic program contains quite a number of Datalog rules it became harder and harder to create a translator which could take care off an entire Magic program. For every unique head in the set of Magic rules a materialized table had to be created and the problem of calls to the rules with a different boundedness became relevant: head $sup_{1.0}(X, Y)$ could be called on with different variations of variables and constants as arguments like $sup_{1.0}(k, Y)$ and $sup_{1.0}(X, z)$. Only relevant information from the tables should be used, so the queries to where materialized table is used should be written with extreme caution.

3.4.3 The end of the translator

Because of problems described at the end of the previous section, together with the very small amount of information of creating a Magic program with a query with multiple contants, it became clear this approach would not be very helpful in the quest of creating Monet based algorithms. The ideas used by the Magic programs, like the Magic sets and the use of the semi naive evaluation strategy, will be very useful and they will be used.

4 Shortest paths

The previous sections discussed the Monet database system and the basics of DataLog and its Magic Sets. In this section the shortest path problem will be described and the Dijkstra algorithm will be explained. Next the shortest path problem will be transformed into a form where the Monet database system and the Magic Sets can be used. The last part of this section will show the results of combining the shortest path problem with Monet and with the Magic Sets.

4.1 Introduction to shortest paths

Especially with the current lifestyle where time is precious and everything must go as fast as possible it is crucial no time will be wasted by asking for directions or by not taking the fastest route from home to work. Thanks to navigation systems it is easy to find the fastest way from a starting point to a destination: simply type the name of the destination and the navigation system will do the rest. Under the hood the navigation systems will use algorithms to find the shortest path. These algorithms are not only used by navigation systems; planning software also uses them to decide the exact route of a mail truck or a stocking truck.

The most basic shortest path problem is to find the shortest path from a source to a sink in a graph which consists of nodes and edges. This is the problem discussed in this section. The next section (section 5) will take a look at some other graph problems like the shortest path between all node pairs in the graph. The formal definition of the shortest path problem is: given a graph *G(V,E)* with weighted, directed edges, find the path from the *source* node to the *goal* node where the sum of the weights of the edges is minimal compared to all other paths from the *source* node to the *goal* node.

As an example take a look at the small graph shown in figure 9: there are eight nodes in this graph and they are connected to each other with weighted, directed edges. What the exact meaning of the weights are is irrelevant: it can be the time it will take to go from node A to node B, it can be the distance, it can be the costs of a toll road, etcetera. In the example graph it is not hard to see which paths are the shortest because it is just a small graph. When graphs become larger it is much harder to find the shortest path between two random nodes. For example the roadmap of the Netherlands, Belgium and Luxembourg contains 1.598.250 nodes and 3.756.335 (directed) edges. Because of this finding the shortest path from Groningen to Luxembourg will not be as trivial as finding the shortest path between any two nodes in figure 9.

The best known algorithm to find shortest path between two nodes is the Dijkstra algorithm [11]:

1. Given a graph *G(V,E)* with weighted, directed edges, a source V_0 and a goal V_g the algorithm starts by giving all nodes the distance from V_0 to that node. Initially V_0 will have distance zero and all other nodes will have distance infinity. Set node V_0 as the current node.
2. Find all unvisited neighbours of the current node. For every neighbour node V_i calculate the path length from the source node to V_i by adding the costs of the arc used to reach V_i to the costs of the current node. The new distance of V_0 to V_i is the minimum of the current distance of V_i and the distance which just has been calculated.
3. When all neighbours are taken care off the current node is marked as 'visited' and the shortest path from V_0 to this node is final. This node will not be used in any other calculations.

4. Set the node with the shortest distance which was not visited yet as the current node and continue from step 2. When the goal node, V_g is marked as 'visited' the algorithm can terminate because the shortest path from V_0 to V_g is found.

As an example the shortest path in figure 9 between A and C is requested. First all nodes will get the initial distance infinity and the distance of node A will be set to zero. A is selected as the current node because it has the smallest distance and all neighbours will be visited. Node F will get the new distance value $MIN(\infty, (0 + 6)) = 6$. In the same way the new distance of B will become 5 and the distance of D will become 3. Now all neighbours of A are taken care off and A will be marked as 'visited'. The next node which will be set as the current node is the node with the shortest distance, which is D. This iteration and all other iterations are illustrated in figures 10 through 16.

Now D will visit all its neighbours and the new value of E will become 6, D will be marked as 'visited' and the new node which will be marked as current node is B.

B will visit all its neighbours and the new value of C will become 9, B will be marked as 'visited' and the new node which will be marked as current node can both be E and F. This is no problem: simply choose one of the two nodes and continue with the algorithm. When F is chosen no neighbours will be found, F will be marked as 'visited' and E will be chosen as the current node.

E will visit all its neighbours and node C is found with a distance of 9. However, the distance of E added to the costs of the arc from E to C is $6 + 2 = 8$ and the new value of C will become 8. It is possible for a node to get different distance values throughout the execution of the algorithm. This is why the algorithm can only terminate when the goal node is marked as 'visited': then it is certain no shorter path can be found.

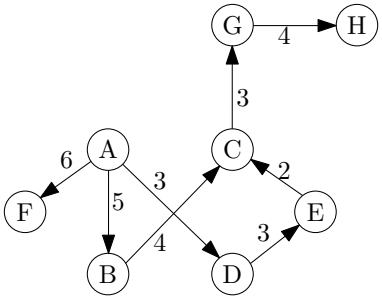

Figure 9: The example graph for Dijkstra

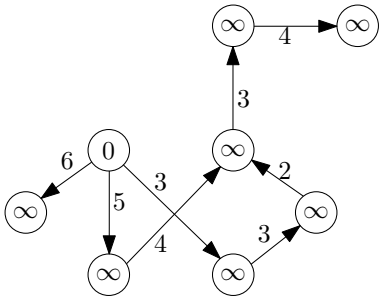

Figure 10: The graph after the initialization

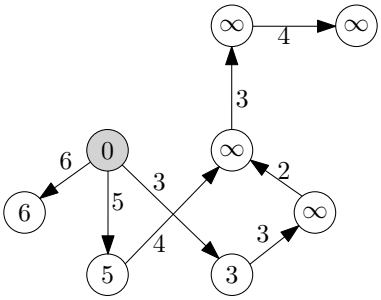

Figure 11: The graph after the first iteration

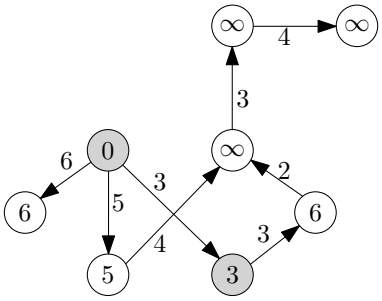

Figure 12: The graph after the second iteration

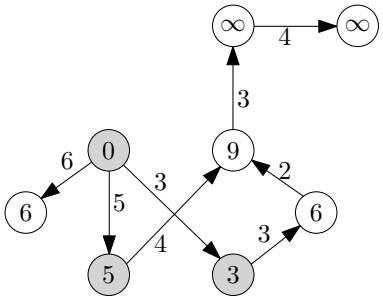

Figure 13: The graph after the third iteration

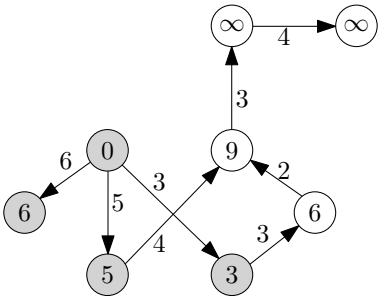

Figure 14: The graph after the fourth iteration

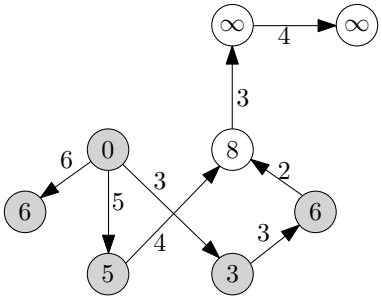

Figure 15: The graph after the fifth iteration

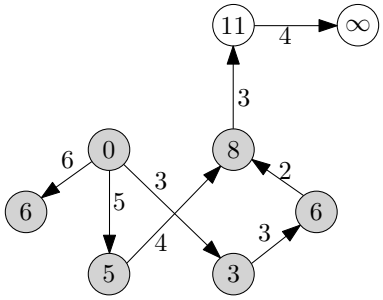

Figure 16: The graph after the sixth iteration

The Dijkstra algorithm is up until today used throughout the world to find shortest paths between two nodes. Sometimes the algorithm itself is used and some other times adjustments have been made to improve the performance of the algorithm for a specific problem. The algorithm itself is quite easy and it is clear when the algorithm can terminate. Usually the graph is read into main memory and the algorithm is applied on that data structure.

4.2 Combining Monet, DataLog and the shortest path problem

In this subsection the shortest path problem will be combined with the Monet database system and with DataLog.

4.2.1 Monet as the data structure

The first step of combining Monet, DataLog and the shortest path problem is to review the data which can be used to find a shortest path. One of the easiest ways to store a road network with weighted, directed edges is to store the source node of an edge, the sink node of the same edge and the costs of this edge. This should be done for every edge and every edge will be stored in one tuple in the database. The dataset used is the roadmap of the Netherlands, Belgium and Luxembourg. Here the nodes have IDs which are integers and the costs of an edge is also an integer, which represents the costs between the two nodes. The table which needs to be created to store all edges is of the form: source (integer), sink (integer), distance (integer). Such a table looks similar to the table in figure 2, described in section 2.2.

The entire graph of the roadmap of the 'Benelux' can be stored in one single table which will be named 'edges'. The names of the columns of the edges-table will be 'source', 'sink' and 'distance' as is suggested by the previous paragraph. The Benelux roadmap contains a total of 3.756.335 edges and therefore the edges-table in the database will consist out of 3.756.335 rows. This table is stored in a Monet database system and will be the data structure to find shortest paths.

When the Dijkstra algorithm is translated into SQL queries and is applied on the Monet data structure it will take approximately 40 hours to determine the length of the shortest path from Groningen to Maastricht. This is because a lot of iterations are needed and approximately eleven iterations can be done in one second. 1.579.570 nodes will be marked as 'visited' when the shortest path is found which implies it will take 1.579.570 iterations to find the shortest path between Groningen and Maastricht: it will take about $1579570/11 = 143597$ second which is a bit less than 40 hours. It is clear the overhead of using a database as a data structure with the exact Dijkstra algorithm is far too big.

It is not hard to explain why this approach will not yield good enough results: Monet is capable of performing database operations, like joins and unions, in a very efficient way on very large data sets. The roadmap of the Benelux is a very large data set but only a few nodes are used in one iteration of the Dijkstra algorithm so the advantages of Monet will not at all be exploited the way they should be. The advantages of the Dijkstra algorithm, however, are that it tries to only use relevant results to find other results and the termination criterion is very clear. These properties might become usefull in the creation of a shortest path algorithm which works with Monet.

4.2.2 Shortest paths with DataLog

It is possible to express the shortest path problem in DataLog. The head of the DataLog rules to find the shortest path from source X to sink Y will look something like:

path(X,Y,C,P).

Where the variable C stand for the costs and the variable P stands for the path found. Different kinds of queries can be requested:

- *path(123,Y,C,P)* will find all paths with source *123* to any sink. The costs of the paths and the exact paths taken are stored in C and P respectively.
- *path(123,Y,35,P)* will find all paths from source *123* to any sink, with distance *35*. The paths themselves are stored in P.
- *path(123,642,C,P)* will find all paths from source *123* to sink *642*. The costs of the paths and the exact paths taken are stored in C and P respectively.

The queries above do not only find the shortest path but find all possible paths and store the length of these paths. The shortest paths can easily be found by taking the minimum of all C's. Every iteration only the shortest paths between two nodes have to be stored. When first the values *path(123,642,8,{123,234})* are found and one iteration later the values *path(123,642,4,{123,245})* are found, the new found path must be stored and the longer path can be forgotten. When this is applied with every iteration only the shortest paths will be stored. Because paths which are no longer relevant are not stored, the performance can increase because these paths will also not be used to find new paths.

The last query, *path(123,642,C,P)*, will be the kind of query which will be used most often. The DataLog rules to define this program are:

(S1) *path(X,Y,C,P) :- edge(X,Y,C) & $P = \{X\}$.*

(S2) *path(X,Y,C,P) :- edge(X,Z,C_1) & path(Z,Y,C_2,P_2) & $C = C_1 + C_2$ & $P = concat(X, P_2)$.*

Rule (S1) states that if there is an edge between nodes X and Y with distance C, there is a path between nodes X and Y with distance C (see figure 17). In the P variable the source nodes will be stored, so in rule (S1) node X will be stored in the path variable.

Rule (S2) states that if there is an edge between nodes X and Z and there is a path between nodes Z and Y, there exists a path between X and Y (see figure 18). The distance of the path between X and Y is the sum of the distance of the edge between X and Z and the distance of the path between Z and Y: $C = C_1 + C_2$. The path variable P works the same as in rule (S1): the source of the edge will be added to the rest of the path found, which will be created by the subgoal *path(Z,Y,C_2, P_2)*.

When Magic Sets are used they will have the most effect on the predicates used in a set of rules. The expressions (the calculations of C and P) will not be taken into account when only

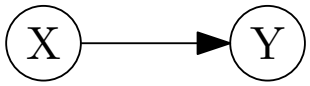

Figure 17: An edge between X and Y is a path between X and Y

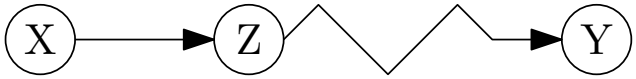

Figure 18: An edge between X and Z and a path between Z and Y means there is a path between X and Y

relevant results have to be found according to the Magic Set. This sounds quite counter-intuitive because when a new path form X to Y is longer compared to the path from X to Y which was already known, the new found path becomes irrelevant. This is true but still the rest of the path from Y to the sink must be calculated and the Magic Sets claim to take care of the optimization not to recalculate known paths.

The Magic Set approach will actually work the same with rules (S1) and (S2) as it will with rules (S3) and (S4):

(S3) *path(X, Y) :- edge(X, Y).*

(S4) *path(X, Y) :- path(X,Z) & edge(Z, Y).*

4.3 Graph connectivity and connecting nodes

When rules (S3) and (S4) will be evaluated, actually the **graph connectivity** of a graph will be calculated. When the query $path(Groningen, Y)$ is requested all nodes which are present in the same connected graph as *Groningen* will be returned in the result. Graph connectivity has to be used in order to find shortest paths and the first part of this subsection is about the graph connectivity rules and the Magic Set used to calculated the graph connectivity efficiently. In the second part of this subsection the graph connectivity algorithm is modified so not the entire graph connectivity is calculated but the algorithm will terminate when the *sink* node is reached.

4.3.1 Graph connectivity

Three important characteristics of the Magic Set approach are:

1. Only relevant results will be found
2. Results will not be calculated more than once
3. The termination criterion is clear

In figure 7 in section 3.3.2 a set of rules defined by the Magic Sets approach is shown. Let's first take a look at finding paths from a single source node to any sink nodes: the graph connectivity. In the examples *Groningen* is the source node: all nodes which are connected to *Groningen* will be found. As mentioned before one row of the edge-table in the Monet database consists of three Integers: the cities are also defined as Integers. For better understanding the names of the cities will be used in the examples because there is actually no difference between the use of Integers or Strings. Just for reference: the city of *Groningen* has number 1260411 in the edge-table.

The idea of the Magic Set approach is to store a list m_{path} with relevant results which will be used to find more relevant results. When *Groningen* is the source it is no use to randomly check all paths from all nodes to all other nodes: the Magic Set will make sure only paths with starting point *Groningen* will be found. The rules found by the Magic Set approach on rules (S3) and (S4) with the query *path(Groningen, Y)* are shown in figure 19.

(MS1) $m_{path}(X)$:- $sup_{2.0}(X)$.

(MS2) $sup_{1.0}(X)$:- $m_{path}(X)$.

(MS3) $sup_{2.0}(X)$:- $m_{path}(X)$.

(MS4) $sup_{2.1}(X, Z)$:- $sup_{2.0}(X)$ & *path(X,Z).*

(MS5) *path(X,Y)* :- $sup_{1.0}(X)$ & *edge(X,Y).*

(MS6) *path(X,Y)* :- $sup_{2.1}(X, Z)$ & *edge(Z,Y).*

(MS7) $m_{path}(Groningen)$.

Figure 19: The new set of DataLog rules according to the transformation for the use of Magic Sets on the shortest path rules (S3) and (S4) and with query *path(Groningen, Y).*

In the first iteration all neighbours of *Groningen* will be found and they will be added to the Magic Set m_{path}. In the next iteration all neighbours of all nodes in m_{path} will be found and they will be added to the Magic Set as well. It is not hard to define these actions as SQL statement so this idea can be applied on the Monet database. As mentioned before there is a table 'edges' which contains all edges of the roadmap of the Benelux: source (Integer), sink (Integer), distance (Integer). An extra table must be created: the Magic Set list must also be stored. The Magic Set table will only contain one column: node (Integer).

One iteration of discovering paths with *Groningen* as the source can be written like:

SELECT edges.sink FROM edges, mpath WHERE edges.source = mpath.node

With this query all neighbours of *Groningen* are found. With a simple 'INSERT INTO mpath' in front of the query all result will be inserted into m_{path} and the Magic Set will grow. The next iteration will find all neighbours of all nodes in m_{path}, these neighbours will be stored in m_{path} and the next iteration will start. When the query mentioned above is applied, the general idea of using only relevant results is translated from the Magic Sets to SQL.

Next to only finding relevant results, another important advantage of using Magic Sets is that results will not be calculated more than once. In the proposed method above a lot of results will be calculated very often. Initially m_{path} only contains the integer 1260411. In the first iteration three neighbours are found which will be added to m_{path}, so the updated m_{path} contains: 1260411, 1185118, 1260412 and 1260413. In the next iteration all neighbours of *Groningen* will be found again and, even worse, all neighbours of *Groningen* will find *Groningen* again. Because of this m_{path} will contain *Groningen* four times after the second iteration. Actually this problem can be split into two problems:

1. m_{path} must only contain unique values
2. Nodes for which the neighbours have been found should not find their neighbours again

The first problem exists because it is possible to reach a node through two different paths. In the current situation it is never necessary to find all neighbours of a node more than once. In order to keep all values in the Magic Set m_{path} unique the DISTINCT operator in SQL can be used:

SELECT DISTINCT edges.sink FROM edges, mpath WHERE edges.source = mpath.node

The second problem is a bit more complicated: it is not enough to only keep track of all relevant results. Actually the Magic Set m_{path} should only contain unique values which have not yet been used before to find their neighbours. In order to create such a table some extra tables have to be created. From now on the four tables showed in figure 20 will be present in the database.

Table name	Columns		
edges	source int	sink int	distance int
mpath	node int		
mpathtotal	node int		
mtemp	node int		

Figure 20: The four tables used to simulate the Magic Set approach

Initially *Groningen* is stored in both the *mpath* and the *mpathtotal* tables. In one iteration all neighbours of the nodes in *mpath* will be found and stored in the *mtemp* table in such a way only unique nodes are stored. Next all values from the *mpath* table will be deleted. All nodes which are present in the *mtemp* table and are not present in the *mpathtotal* table will be stored in the *mpath* table: only nodes which have not yet found their neighbours will be stored in the *mpath* table and in the next iteration their neighbours will be found.

Now all nodes from the *mtemp* table can be removed and all new found values which are currently in the *mpath* table must be copied into the *mpathtotal* so they will not be chosen again to find their neighbours in one of the next iterations. The idea which is applied here is actually the same idea which is used in the semi naive evaluation strategy (section 3.2): only new found results should be used to find other results. The only difference is that in the current database implementation it will **never** happen that a node will find its neighbours more than once, while this is possible with the semi naive evaluation strategy when one node can be reached through several paths.

The algorithm in pseudo code is shown in program 1. All queries used to perform the algorithm are shown in program 2. In these programs two different termination criteria can be found. Only the first termination criterion is relevant for finding the graph connectivity. The second termination criterion will be explained later.

The termination criterion used with this graph connectivity algorithm is easy: when no elements are present in the *mpath* table no new neighbours have been found and the entire connectivity is known. The queries which are used for the termination criteria are performed by Monet but the results are returned to Java. In Java the final check will be done to decide whether or not the algorithm can terminate. The while loop is also programmed in Java.

An interesting observation is done while performing the queries from program 2: the LEFT JOIN used in the third query yield much better results than the query which uses NOT IN and yields the same results. This query is:

*INSERT INTO mpath(SELECT mtemp.node FROM mtemp WHERE mtemp.node NOT IN (SELECT * FROM mpathtotal))*

When the algorithm is used with *Groningen* as the source the original algorithm will take approximately 79 seconds while the algorithm with the 'NOT IN' query takes approximately 207 seconds to find the same results. It is not clear why this difference exists.

Program 1 Pseudo code for the Magic set approach in Monet

//initialization phase
insert *source* into *mpath*
insert *source* into *mpathtotal*

//the loop
while *sink* not found and *mpath* not empty {
    //insert all neighbours of the current front into the *mtemp* table
    INSERT INTO *mtemp* ($\pi_{sink}(edges \bowtie_{source=node} mpath)$)

    empty *mpath*

    //insert all node values of the *mtemp* table which are not present in the *mpathtotal*
    //table into the *mpath* table
    INSERT INTO *mpath* ($\pi_{mtemp.node}(\sigma_{mpathtotal.node=NULL}(mtemp \bowtie mpathtotal))$)

    copy all values from *mpath* to *mpathtotal*
    empty *mtemp*

    //The termination criteria
    if *mpath* is empty: terminate //entire graph connectivity is found
    if *sink* in *mpath*: terminate //sink is reached
}

Program 2 The SQL queries which are used to implement the algorithm shown in program 1

INSERT INTO mtemp (SELECT DISTINCT edges.sink FROM edges, mpath WHERE edges.source = mpath.node)

DELETE FROM mpath

INSERT INTO mpath (SELECT mtemp.node FROM mtemp LEFT JOIN mpathtotal ON mtemp.node = mpathtotal.node WHERE mpathtotal.node IS NULL)

*INSERT INTO mpathtotal (SELECT * FROM mpath)*

DELETE FROM mtemp

SELECT COUNT(*) FROM mpath WHERE mpath.node = 1260411

SELECT COUNT(*) FROM mpath

4.3.2 Connecting nodes

When an algorithm has to decide whether or not a path from one node to another exists, it is not at all necessary to calculate the entire graph connectivity: when the *sink* node has been reached the algorithm can terminate. The requested query which will be used in the shortest path problem will give both the *source* and the *sink* node, so the query will not be *path(Groningen, Y)* but *path(Groningen, Maastricht)*. The graph connectivity algorithm can be used to determine whether or not a path exists between *Groningen* and *Maastricht* when an extra termination criterion is added. This is the second termination criterion of which was written about earlier. When the *sink* node is detected in the *mpath* table, the *source* and the *sink* node are connected and the algorithm can terminate. When the algorithm terminates because no elements are left in the *mpath* table it is clear no path between the source and the sink can be found. The last two SQL queries in figure 4.3.1 are the SQL implementations used for termination criteria: the second last query should return 1 or higher when the *sink* node has been reached and the last query will return 0 when the entire graph connectivity has been calculated. With this extra termination criterion it is shown how the graph connectivity algorithm is a basis for the shortest path algorithm.

When the edges table is the only information source it is not possible to define a more efficient way of calculating whether or not a path between two nodes exists than the proposed idea: use the graph connectivity algorithm with the extra termination criterion. At least, not when only the source node is known. There are more efficient ways if both the source and the sink node are known, so in the example of determining if a path between *Groningen* and *Maastricht* exists a better method can be used.

When Magic Sets are used a problem occurs: now the requested query does not contain only one constant, like *anc(k, Y)*, but two contants: *path(Groningen, Maastricht)*. When the approach described by Ullman [17] is used a set of rules will be created which will not yield correct results. This is because only one Magic Set will be created which contains values for both the source and the sink values: $m_{path}(Groningen, Maastricht)$ is initially the only tuple in the Magic Set. In the next iterations no extra tuples will be added to the Magic Set and only if an arc between *Groningen* and *Maastricht* exists, the new set of rules will yield a positive result.

So instead of using only one Magic Set for both the source and the sink it is much better to use two different Magic Sets: m_{pathX} is the Magic Set for the source nodes and m_{pathY} is the Magic Set for the sink nodes. This idea can also be used to improve the performance of the suggested algorithm: not only start at the *source* node to find a path to the *sink* node but also start at the *sink* node to find a path to the *source* node. Actually this can also be done with the Dijkstra algorithm and the name of this improved Dijkstra algorithm is bidirectional Dijkstra.

Up until this moment the search for a path was much alike shown in figure 21: from one source all possible paths will be tried. It is much more efficient to start the search from both sides: both from the source and from the sink. When the two search areas touch each other it is certain a path from the source to the sink is present. This way a lot less nodes will be visited and therefore a lot less calculations are needed and the performace will most probably increase. Especially when there are a lot of nodes and edges it will be far more efficient to use the bidirectional method instead of the onedirectional method. A graphical representation of the two way search is shown in figure 22.

In order to implement this the original algorithm must be applied on both the source and the sink node. Note the problem which occurs here: because all edges are one way it is not guaranteed that if a path exists from X to Y, a path from Y to X also exists. Or maybe this path exists but the costs can be different. So when applying the algorithm with *Maastricht* as the source node the edges must be treated like they are reversed: like driving backwards through

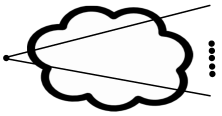

Figure 21: Trying to find a path from the source to the sink with a single way algorithm

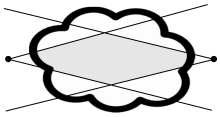

Figure 22: Trying to find a path from the source to the sink with a two way algorithm: less nodes will be visited

a one-way street against the traffic. To apply the algorithm some extra tables have to be created and all tables which will be used are shown in figure 23.

Table name	Columns		
edges	source int	sink int	distance int
mpath_source	node int		
mpathtotal_source	node int		
mtemp_source	node int		
mpath_sink	node int		
mpathtotal_sink	node int		
mtemp_sink	node int		

Figure 23: The seven tables used to simulate the bidirectional Magic Set approach in SQL

The queries which were discussed in the one way implementation will be used on both the tables for the sink and the source node. One iteration of the new algorithm will consist of applying one iteration on the source node and one iteration on the sink node. The termination criterion is very clear: when both searches reach each other there is a path present between the source and the sink. Actually the graph connectivity will be calculated for both the source and the sink node in the graph and as soon as the sets of nodes connected to the source and sink node will overlap it is certain a path between the sink and source node is present.

There are several ways to implement this in an SQL query but the most efficient query must be chosen because the performance must be as good as possible. The query which can be calculated fastest would probably be the comparison between *mpath_sink* and *mpath_source*: if a node is present in both tables a path has been found. It is possible, however, this method will not yield correct results: if in an iteration *mpath_source* contains node X and *mpath_sink* contains node Y and X and Y are neighbours, it is not certain the algorithm will terminate. After the next iteration *mpath_source* will contain Y and *mpath_sink* will contain X and a path has been found while *mpath_source* and *mpath_sink* never contained the same node in the same iteration.

The best way to solve this problem is to use one of the *mpath* table and compare this with the other *mpathtotal* table: if a node is present in both tables a path is found and the algorithm can terminate. The bidirectional search will improve the performance of the oneway search which is showed in section 4.4.

4.4 Preliminary results and conclusions

When the Dijkstra algorithm is directly applied on the Monet data structure without the queries discussed in the previous section the performance is very, very bad. It will take approximately 40 hours to calculate the shortest path from *Groningen* to *Maastricht*. This is mostly because a lot of small operations must be applied every iteration and a database is not at all the designated

way to do this. It is far more efficient to use a main memory data structure to perform such an algorithm. A nice side note is that Monet is much, much faster compared to MySQL in performing the basic Dijkstra algorithm so the claim of the Monet development team also holds in this specific situation: Monet outperforms MySQL. Even when all tables in MySQL are indexed.

The three different algorithms which are discussed will be compared to each other. The three algorithms are:

new Only the results found in the last iteration will be used to find new results. All elements in the result table will be unique. This algorithm is much like the initial algorithm explained in the beginning of the section.

unique Only the result found in the last iteration which have not been found before will be used to find new results. All elements in the table will be unique.

twoway Works the same as the *unique* algorithm but now from both the source and the sink. All elements in the tables will be unique.

Each of these algorithms are executed five times in a row with *Groningen* as the source and *Maastricht* as the sink. The only thing the algorithms do is trying to decide whether or not a path exists between *Groningen* and *Maastricht*. The results of the experiments are shown in figure 24.

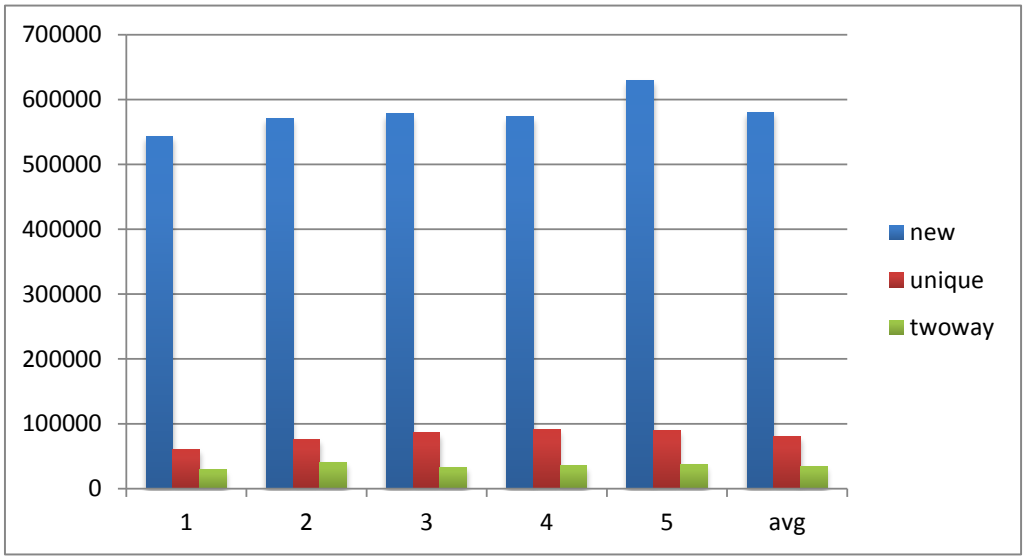

Figure 24: The performance in milliseconds of the three algorithms for five runs and the average performance

As expected the **new** algorithm performs worst, the **unique** algorithm performs better and the **twoway** algorithm performs best. The difference between the **new** and the **unique** algorithms is best explained by taking a look at the size of the m_{path} table during the execution of the algorithms. Initially both m_{path} tables contain only one single element: 1260411, the ID of *Groningen*. Only after the first couple of iterations it is already shown that the **new** algorithm will find a lot of nodes which have also been found in an earlier iteration. The number of unique values in the m_{path} tables of both algorithms are shown in figure 25. The left Y-axis is for the m_{path} size of the **unique** algorithm and the left Y-axis is for tor the m_{path} size of the **new** algorithm.

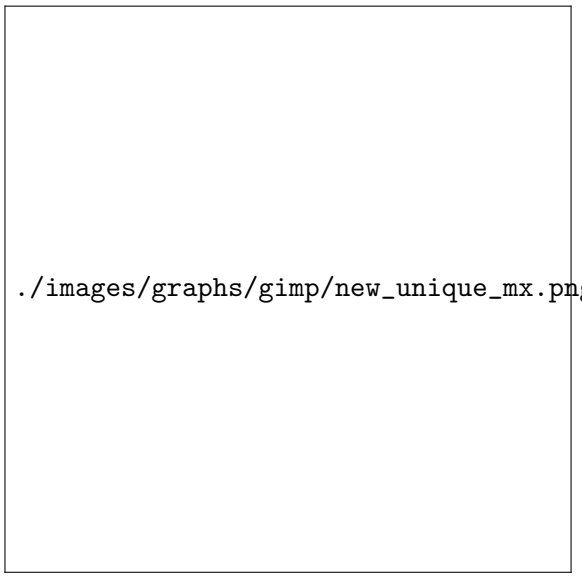

Figure 25: The size of the *mx* table during the execution of the **new** and **unique** algorithms

After the last iteration, where *Maastricht* is reached, a total of 1560200 unique nodes are stored in the m_{path} table of the **new** algorithm, while the maximum of unique values stored in the m_{path} table of the **unique** algorithm is 3105. In the last iteration the neighbours of 1559382 nodes must be found in the **new** algorithm which takes about 1098 milliseconds, against the neighbours of 808 nodes in the **new** algorithm which takes about 84 milliseconds. Because the size of the m_{path} table is much, much smaller during the execution of the algorithm the performance of the **unique** algorithm is much better compare to the performance of the **new** algorithm. The times mentioned in this paragraph and in the rest of this chapter are the average runtimes of the algorithms: every algorithm is executed five times and the average of the runtimes is calculated for each algorithm.

Another interesting observation in figure 25 is the parabolic behavior of the size of the m_{path} table when the **unique** algorithm is used. This behavior is explained by the fact that the algorithm will reach the borders of the graph: when no neighbours can be found which have not yet been found before, no new results will be added to the m_{path} table. Two of these borders are the border between the Netherlands and Germany and between the Netherlands and the North sea. When no limit is set to the number of iterations of the **unique** algorithm it will finally find all nodes which are connected to each other and from that point on the m_{path} table is empty.

The time needed to execute the query in order to decide whether or not the sink node is reached depends somewhat on the size of the m_{path} table but it only increases from initially 1 millisecond to 8 milliseconds during the execution of the algorithm. This is generally less than 5 percent of the total runtime of one iteration and it is really minimal compared to the time loss of the **new** algorithm where all found results of an iteration are used to find new results.

The **twoway** algorithm is the best algorithm described here. As is explained near figures 21 and 22 the number of nodes for which the neighbours will be calculated is less compared to the **unique** algorithm. When the algorithms terminate a total of 1.560.199 nodes have been visited by the **unique** algorithm and 1.158.900 nodes have been visited by the **twoway** algorithm: less than 75%.

The size of the m_{path} table seems to be the most important factor with regards to the performance of an algorithm. Intuitively this is correct because the most difficult database

operations of the given queries will be applied on the m_{path} table. All other tables are not at all as important for the runtime. As mentioned before the query to calculate whether or not the algorithm can terminate only takes about 5 milliseconds. When the **twoway** algorithm is used the size of the m_{path_source} and m_{path_sink} tables are kept as small as possible and this yields the best performance.

In figures 26 and 27 the number of unique nodes in the m_{path} table and the calculation time per iteration are shown while deciding whether or not a path exists between *Groningen* and *Maastricht*. When the size of the m_{path} table increases the time one iteration takes also increases. When the size of the m_{path} table decreases, like in figure 27, it will take a number of iterations but the time an iteration takes will also decrease. The delay might be something internal in Monet like caching.

In all experiments done with the **unique** and the **twoway** algorithms the runtime for one iteration first increases and after the 'peak' in the number of unique nodes in the m_{path} table the runtime will decrease pretty fast. Figures 28 and 29 show the runtime of the algorithm together with the size of the m_{path} table for the calculation of the entire graph connectivity. Figure 28 has *Groningen* as source node and figure 29 has *Maastricht* as source node.

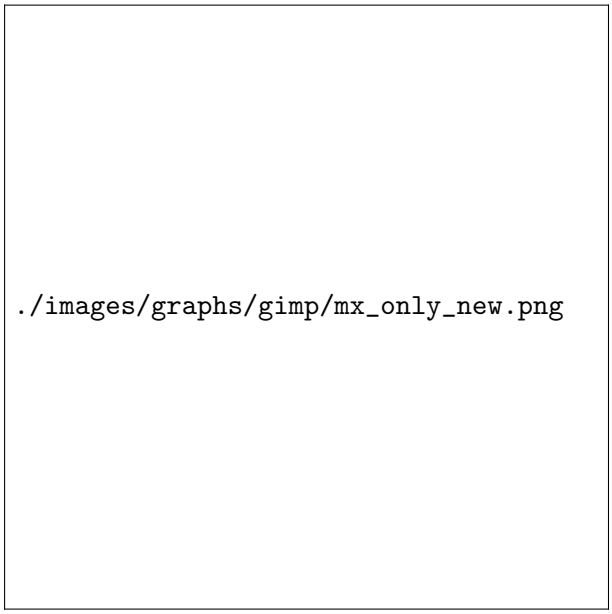

Figure 26: The runtime and size of the m_{path} table with the **new** algorithm

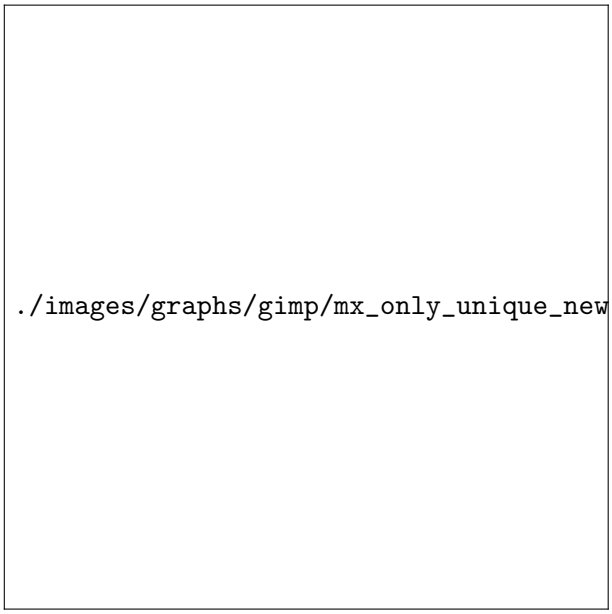

Figure 27: The runtime and size of the m_{path} table with the **unique** algorithm

In these graphs it is clear the runtime depends heavily on the size of the m_{path} table. It is not possible to decrease the size of the m_{path} table more than is done in the **unique** algorithm and is used in the **twoway** algorithm. Because of this it will be very hard to decrease the runtime of the algorithms and therefore increase the performance of the algorithms.

4.4.1 Speed differences

Another observation is done while studying the results: the first run with the Monet database system as data structure is faster than all other runs. Even worse: the more runs are done, the more time a run takes. This can be seen in figure 24 and table 2 and in almost all other experiments it is also true. When it is the other way around, like the Dijkstra algorithm is in table 2 it is more logical. In the first run a lot of data must be stored in main memory and in

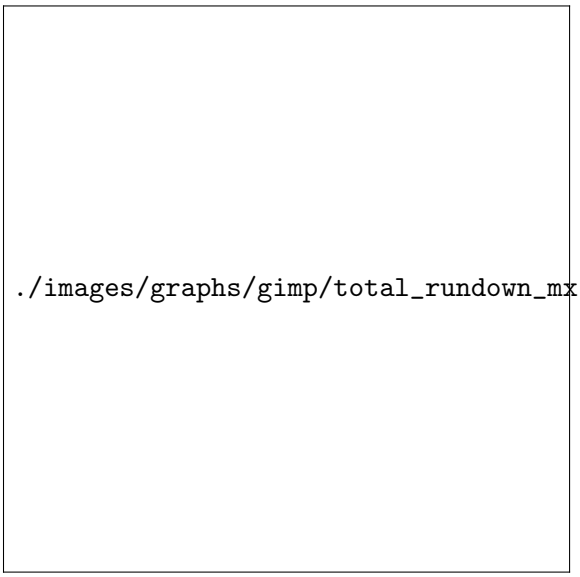

Figure 28: The runtime and size of the m_{path} table with *Groningen* as source until all nodes have been reached.

Figure 29: The runtime and size of the m_{path} table with *Maastricht* as source until all nodes have been reached.

the second run a lot of this data is still present in main memory so executing the algorithm will yield a better performance. But what about Monet? The first run of the **twoway** algorithm is more than 10 seconds faster compared to the second run. It is not clear why Monet behaves the way it does.

4.5 Problems to find shortest paths

This section is about the differences between the graph connectivity algorithm described in the previous sections and the shortest path algorithm with which it all started. The graph connectivity algorithm is limited because of the second termination criterion: when the sink node is reached the algorithm will terminate. First the performance of the **twoway** algorithm will be compared with the performance of a main memory Dijkstra algorithm. After this comparison the necessary additions to the **twoway** algorithm in order to calculate the shortest paths will be discussed.

4.5.1 Monet based graph connectivity versus main memory Dijkstra

When the results of the **twoway** algorithm of section 4.4 are compared to the results of the Dijkstra algorithm performed with a main memory data structure the difference is very big, as is shown in table 2.

On average the Dijkstra algorithm with main memory data structure performs more than twenty times as fast as the **twoway** algorithm. When the bidirectional Dijkstra algorithm will be used this will probably yield even better results than the original Dijkstra algorithm. So no improvements have been found, especially when the results of the algorithms is taken into account: the **twoway** algorithm will only give an answer to the question: 'Can the *sink* be reached from the *source*?' while the Dijkstra algorithm also finds the shortest path. The extra aspect which are necessary to transform the **twoway** algorithm into a shortest path algorithm are discussed in the rest of this section. All of these additions will only have a negative effect on

Run	Two way	Dijkstra	Factor
1	28929	2667	10.85
2	39946	1611	24.80
3	33409	1603	20.84
4	35239	1613	21.85
5	37108	1624	22.85
avg	34926	1823	19.15

Table 2: Five executions of the **twoway** algorithm compared with five executions of the Dijkstra algorithm

the performance of the basic **twoway** algorithm. The performance of a shortest path algorithm with Monet as the data structure will not yield a performance which even comes near the performance of a main memory Dijkstra algorithm.

The reason why the performance of the **twoway** algorithm is very bad compared to the main memory Dijkstra algorithm is probably because a lot of small operations are performed. In the conclusions the statement has been made that the smaller the m_{path} table, the better the performance. Discrete this statement holds but the power of Monet lies in its ability to perform database operation on huge sets of data in a relative small amount of time. The Dijkstra algorithm itself performs a couple of small operations every iteration and the profit made by trying to perform larger operations every iteration to find the path faster (as is done with the **twoway** algorithm) is actually negative.

When an algorithm which needs a lot of operations on large sets of data is combined with Monet it will probably yield much better results compared with the main memory algorithm. It is not expected the algorithm with Monet works better than the main memory algorithm but it will probably approach the runtime better than the shortest path algorithm did. Section 5 describes other (network) algorithms in order to try and find an algorithm which will yield good results with Monet as the data structure.

In order to transform the **twoway** algorithm into a shortest path algorithm some modifications must be applied which are not trivial. Because the steps which are done in the **twoway** algorithm are also necessary in order to find the shortest path, the modifications will only decrease the performance of the **twoway** algorithm. Because of this it is of no use to implement these modifications. To be complete the problems are discussed in this section.

4.5.2 Correct usage of the semi naive evaluation idea

The use of the semi naive evaluation strategy with the shortest path algorithm will be different from the use of the semi naive evaluation strategy with the **twoway** algorithm. In the **twoway** algorithm the *front* is defined as the nodes which are found in the last iteration except for the nodes which have been found in an earlier iteration. When the algorithm must decide whether or not a path exists between two points this use of a *front* works fine. When a shortest path must be found the definition of the *front* must be changed. When for example the *twoway* algorithm is applied on the graph shown in figure 30 the path from A to E will first be found through B and because B will not be in the *front* again the shortest path will be *A,B,E* with a distance of 9, while the shortest path is actually *A,C,B,E* with a distance of 8.

So not only new found nodes which have not yet been found in earlier iterations should be added but also the nodes with a new found shortest path must be added to the *front*. When initially all nodes, except for the source and the sink, have a value of ∞ the *front* will be all nodes for which a new shortest path has been found in the last iteration. The effect of the new definition of a *front* can be quite big: when in the 30^{th} iteration a shorter path to the second

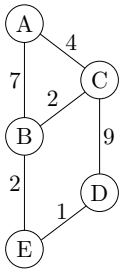

Figure 30: A bad definition of the *front* will return a wrong shortest path

node of a path is found it will take 29 extra iterations until all nodes up until iteration 30 are updated. A lot of extra calculations are necessary in order to be sure the shortest paths are found.

4.5.3 Termination criteria

With the Dijkstra algorithm it is clear when the shortest path is been found and therefore when the algorithm can terminate. It is much harder to define the termination criteria for the **twoway** algorithm. First let's take a look at the **unique** algorithm which is also used in the **twoway** algorithm. In the experiments the algorithm was terminated when the sink node, *Maastricht*, has been reached. This is not the correct termination criterion if the shortest path is to be found. As can be seen in figure 31 the shortest path is not always the path between two nodes with the least number of edges.

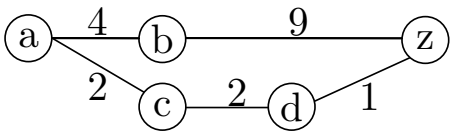

Figure 31: The shortest path from a to z is not through b, but through c and d

Worst case there is a path with lots and lots of edges which runs outside of the rest the graph and this path creates the shortest path because the edges have small costs. The algorithm must not terminate before this entire path is explored and it is certain no shorter path can be found. The sooner it is certain the shortest path is found, the sooner the algorithm can terminate and the better the performance of the algorithm will be.

When the sink node has been reached and a path is found with distance D a termination criterion can be created: when a new found path has a distance which is larger than D it is of no use to continue the search with that specific path. When no new nodes are found with a distance smaller than the current shortest path to the sink node the algorithm can terminate and the shortest path has been discovered. More formal this termination criterion can be written like:

When no new nodes are found which have a smaller distance to the source node than the shortest known distance of the source node to the sink node, the shortest path has been found and the algorithm can terminate.

A better termination criterion is hard to find. This termination criterion can also be used with the **twoway** algorithm, although it is more difficult.

Let's define the distance from the source to a node i as $D_{source,i}$ and the distance from the sink to a node j as $D_{sink,j}$. When a node k already has a $D_{source,k}$ value and a $D_{sink,k}$ is found a new shortest path from the source to the sink might have been found. When $D_{source,k} + D_{sink,k}$ is smaller than the currently known shortest path, a new shortest path is found. With this algorithm the newly defined *front* is used.

The easiest termination criterion is to stop the algorithm when both the smallest $D_{source,i}$ ($MIN(D_{source,i})$) and $MIN(D_{sink,j})$ are larger than the shortest path found from the source to the sink. It is also correct if the algorithm will terminate when $MIN(D_{source,i})+MIN(D_{sink,j}) \geq$ the shortest path: it will not be possible to create a shorter path. A better termination criterion is very hard to find. When the suggested termination criterion is used a lot of extra operations will be necessary after the first path is found with the **twoway** algorithm. The number of iterations which are calculated will also increase. In very bad cases the path with the largest number of edges is the shortest path from the source to the sink or a lot of recalculation of a path must be done, like it is explained the paragraph about the definition of the *front*.

Implementing a good termination criterion is not very difficult but it can have a disastrous effect on the performance of the algorithm.

4.5.4 Store the shortest path and the costs of this shortest path

In the **twoway** algorithm the simple DataLog rules for finding a path are used: (S3) and (S4). These rules do not take care of the costs of a shortest path and of storing the path itself. When rules (S1) and (S2) are used the length of the paths, together with the exact paths, are stored. It is pretty straightforward to implement rules (S1) and (S2) in SQL statements: two new columns must be added to the tables *mpath*, *mpathtotal* and *mtemp*. One column for the costs of the shortest path found up until the specific moment and one column to store the shortest path itself. It will not take a lot of time to store this 'extra' information but nevertheless the performance of the shortest path algorithm will decrease.

As mentioned in the beginning of this section only the shortest path between two points have to be stored in order to eventually find answer to the requested shortest path query. For every iteration the found paths have to be compared to the shortest paths found up until that moment and only the shortest paths must be selected to be stored and used in the next iteration. These comparions will decrease the performance of the **twoway** algorithm as well.

4.6 Density of the graphs

The graph used for all experiments is the roadmap of the Benelux with 1.598.250 nodes and 3.756.335 directed edges. The edges / nodes ratio is approximately 2.35 which makes it quite a sparse graph. How would the **twoway** algorithm perform on a more dense graph? And how would the main memory Dijkstra algorithm perform on such a graph? When a graph with the same number of nodes and about three times the number of edges is created Dijkstra will take more time before the shortest path is calculated. It is not certain whether or not the performance of the **twoway** algorithm will decrease. The performance of the **twoway** algorithms also depends on the number of edges but it also depends on the minimal number of iterations necessary to go from the source to the sink. When more edges will be created in the graph this will probably mean less iterations of the **twoway** algorithm will be needed in order to decide whether or not a path exists between the source and the sink, so the performance will increase.

It is not very easy to create random graphs with a higher density because in a roadmap the edges are not placed randomly at all: the planar representation of a roadmap will not contain long edges and the edges will (almost) never cross. It is hard to create a random graph for which these conditions hold. When a purely random graph is created with 1.598.250 nodes and 3.756.335 edges, just like the Benelux roadmap, the **twoway** algorithm will only take approximately 30 iteration to find the entire graph connectivity. In the original Benelux roadmap more than 1000 iterations are needed to define the entire graph connectivity with *Groningen* as a source.

It might be useful to do some more research in this direction: what are the conditions under which Monet based algorithms will perform best? When all of these conditions are known final conclusions can be drawn with help of experiments run with these conditions.

4.7 Final conclusion

The use of the **twoway** algorithm takes about 25 times as much time as the Dijkstra algorithm performed on a main memory data structure. When the **twoway** algorithm must be altered in such a way so it will also be able to find the shortest path the performance will only decrease. Compared to other open source database systems like MySQL and PostgreSQL the Monet database system performs the **twoway** algorithm very fast but it can never compete with dedicated main memory algorithms like Dijkstra.

When more dense graphs are used the Monet based algorithm might have a slower descending performance compared to main memory algorithms but it is hard to create graphs which can be used to run such experiments.

The idea of using Monet as the data structure and SQL queries as the algorithm where the performance depends on Monet as much as possible does not yield very good results in this specific case. This is probably because the dedicated algorithm performs a lot of small operations on small parts of the dataset and the Monet database systems is very good in performing database operations on large sets of data. Section 5 is about other algorithms which are potentially more likely to use the qualities of Monet.

5 Other graph problems

The results of combining Monet, Datalog and the shortest path problem are not very impressive when compared to the performance of the main memory Dijkstra algorithm. This is probably because the Dijkstra algorithm can do a lot of small operations on a part of the graph to find the shortest path while Monet performs much better when operations are performed on large datasets. In this section some other algorithms will be discussed which might yield better results when applied with Monet as the data structure.

The developers of Monet claim the database system can perform database operations on large datasets in an efficient way. The difference between the performance of the Monet database system and other database systems is significant, especially the performance of the join operation is very good. The join operation is used a lot in the **twoway** algorithm in section 4 but a lot of unnecessary calculations are done compared to the Dijkstra algorithm so in the end the performance of the **twoway** algorithm cannot even come close to the performance of the Dijkstra algorithm.

5.1 Other path problems

5.1.1 Bellman-Ford

The Dijkstra algorithm will calculate the shortest path from one source to one sink in a graph where all edges have positive costs. When there are edges with negative costs the Dijkstra algorithm is not guaranteed to return a correct answer. The Bellman-Ford algorithm [6] does. Unfortunately this algorithm also performs a lot of small operations on a small part of the graph. Especially because it is still an algorithm to find the shortest path from one source to one sink it cannot make use of the qualities of Monet the way it was initially intended.

5.1.2 Floyd-Warshall

The Floyd-Warshall algorithm [12] will return the shortest paths between *all* pairs of vertices, even when edges have negative costs. The idea of finding the shortest paths between all pairs of vertices in a graph seems to be more fit for an algorithm which uses Monet as a data structure.

Unfortunately some of the same problems will arise as with the single source and single sink shortest path problem: the most efficient way to find the shortest path between all pairs of nodes is to do a couple of small operations every iteration. This was one of the problems why the shortest path algorithm did not yield impressive results. When all nodes are used as source nodes and one iteration will find all neighbours of the front, calculates new possible shortest paths for all these nodes and puts all nodes with a lower found shortest path in the front it can take a lot of iterations before the algorithm will terminate. Especially when a new shortest path is found for a node which is in the beginning of a path: this new distance must propagate through all the paths of which that node is the source node. To give an intuition about this problem lets take a look at the example graph in figure 32.

For this moment let's assume only the shortest path from A to E is requested. In the first iteration the shortest path from A to G will be 13. In the same iteration the shortest paths to B and D are found. In the second iteration a shortest path from A to F is found of length 17 and the path runs through E. In the same iteration a new shortest path from A to E has been found of length 11 which runs through D. After the second iteration three nodes will be in the front: C, F and E.

In the third iteration the G will be discovered by finding all neighbours of F and the shortest path from A to F at that moment is still 17, so the shortest path from A to G will have length 25. In the same iteration E will find node F again and the new found shortest path from A to

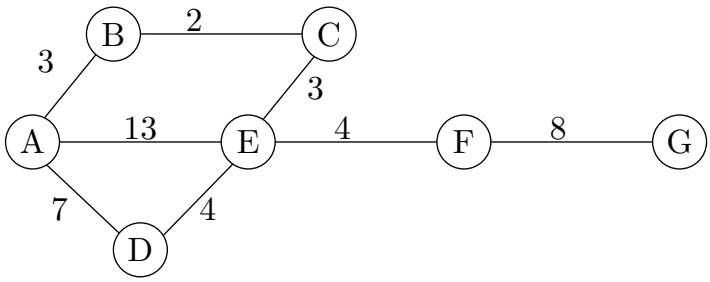

Figure 32: In the first three iterations a new shortest path from A to E is found and the rest of the path to G must be updated.

F will be 15. A new shortest path from A to E is also found: a path with length 8 which runs through B and C. So three nodes will be in the new front: E, F and G.

In the fourth iteration both F and G will get a new length for their shortest paths and in the fifth iteration G will get a new length for the third time. The problem which arises here is the propagation of new found shortest paths through the rest of the graph. With this small graph it already takes quite a lot of extra iterations, let alone how many extra iterations are necessary when graphs are used with millions of nodes. When the shortest paths for all node pairs are requested this problem will also occur a lot which will result in a pretty bad performance.

The Floyd-Warshall algorithm uses dynamic programming in order to find the shortest paths for all node pairs which will take $O(|V|^3)$ time. The updating part is a very inefficient step in the algorithm. In order to make use of the strength of Monet a lot of calculations must be done at once and it will happen quite often, just like in the Floyd-Warshall algorithm, a shorter shortest path to a node is found. The shortest path value of the node must be updated which in itself is not a very inefficient operation but when applying the algorithm on a random graph with a couple of million nodes this update operation has to be done quite a lot. This will decrease the performance because it is much faster to change such a value in a main memory data structure compared to changing it in a database system.

The last option which was worth trying is to use the exact Floyd-Warshall algorithm with Monet as data structure. The expectation is that the algorithm will not yield a very impressive performance because the data which can be stored in main memory now has be accessed through a database layer. Initially the table of the shortest paths from all nodes to each other must be set to ∞ except for the distance from a node to itself: this distance is 0. Monet could not create this table for the Benelux dataset which would have to contain about 1.6 million fields for every 1.6 million rows. Another option is not to give all these paths an initial length but only when a path is found the length is stored: this is a much more sparse representation. In order to determine whether or not a path already exists and whether or not the value of that path is longer than the length of the new found path a number of SQL queries are needed.

These queries will, again, perform small actions on just a small part of the dataset and they are applied a lot. These queries are applied, next to the queries which are used to perform the algorithm itself which also perform small actions on a small part of the dataset. Because of this and because the database layer have to be used instead of directly accessing the data in main memory, Floyd-Warshall with Monet as a data structure will not yield impressive results.

5.1.3 Conclusions on shortest path problems

Whether a shortest path is to be found in a graph with only positive edges, a shortest path is to be found in a graph with positive and negative edges or the shortest paths for all node pairs are to be found: the same problems arise. The dedicated main memory algorithms will perform small operations on small parts of the graph in order to find the correct result. It is possible to create algorithms with Monet as a data structure and with SQL statement to perform an algorithm which yield the same results but the performance of such algorithms is always worse than the main memory algorithms like Bellman-Ford an Floyd-Warshall.

5.2 Breadth first search problems

It is obvious another kind of algorithm should be chosen in order to benefit more from the properties of Monet. Algorithms which are based on the breadth first search algorithm can make more use of the qualities of Monet because the join operation is used a lot. Especially when every node has a lot of successors, large datasets will be created and must be joined with other large datasets. Unfortunately there are not a lot of algorithms which use the breadth first search in its purest form because it will use a lot of memory during the execution of the algorithm.

5.2.1 Subsetsum problem

A problem which can be solved by using a breadth first search is the Subsetsum problem [5]. The general explanation of the Subsetsum problem is: given a set N of numbers, will the sum of a non-empty subset of N be exactly a given number S? As an example: does a subset of set $\{2, 3, 4, 6, 7, 14, 16\}$ exists with the sum of 28? The answer is 'yes' because $6 + 7 + 14 = 28$.

When initially the table with all the elements of set N is used the first iteration will be to join this table with itself and the table *solutions* will be created. This table will contain two columns: the subset of elements used and the sum of this subset. From this point on an iteration will be the join of the *solutions* table with the table of all elements of *N*. A solution here is one row from the *solutions* table and is not the same as the final result: it is just one point in a series of iterations.

In the algorithm it is not allowed to use the same element twice which causes a problem while joining. In the example above the *solutions* table after one iteration must not contain the solution $\{2, 2\}$. Another problem is to keep all solutions in the *solutions* table unique: after the first iteration both $\{3, 6\}$ and $\{6, 3\}$ will be present.

The first problem must be solved in order for the results to be correct. To solve this problem some extra tests must be included in the queries. These tests will take increasingly more time when the number of executed iterations is growing: the addition of an element can only take place when this element is not yet used in that specific solution up until that moment. When the i^{th} iteration is executed the element must be compared to i elements before it can be added to the solution. Of course very efficient algorithms can be used when the elements are sorted so the time it will take to decide whether or not the current element is present or not will be minimal but it still takes more and more time.

The second problem will cause the performance to drop when the size of the tables increase. The solution is to only keep unique solutions in the *solutions* table. This will also have a negative effect on the performance: as the size of the tables increase it will be harder to decide whether or not one solution is unique because the increasing size of the *solutions* table and the increasing number of used elements in the solutions. The second option yields better results compared to the original problem, especially when the size of the *solutions* table becomes very large.

Another difficulty with the Subsetsum problem is it NP-completeness [5]: the time it will take to solve the problem increases very fast as the size of the problem grows. When the initial number of elements is quite small it is possible to find the final result very fast but as the initial number of elements increases it will take exponentially more time to find the final result. Either when an algorithm specific data structure in main memory is used or when Monet is used as a data structure it will take a huge amount of time to solve the Subsetsum problem when a lot of elements are used.

The main advantage of Monet is the good performance when operations on large sets of data must be applied. When with the Subsetsum problem large sets of data are used it means a lot of elements are used and the performance of the algorithm will decrease because the problem is NP-complete. The performance advantage of Monet will be far less significant compared to the disadvantage which arises because of the NP-completeness of the problem so implementing the Subsetsum problem with Monet as data structure is of no use.

5.2.2 NP problems

Problems for which it was potentially promising to implement an algorithm with Monet as a data structure but are also in NP are: the Knapsack problem (Subsetsum is actually a special case of the Knapsack problem), the Bin-packing problem and the Clique problem. For all these problems the same conclusion can be drawn as for the Subsetsum problem: the advantage which can be gained by using Monet as a data structure cannot compete against the NP-completeness of the algorithms.

5.3 Bipartiteness of a graph

The problems discussed in this paper are not yet suitable to solve with Monet as the data structure. A problem which can be solved with a breadth first search like algorithm which is not in NP is a problem which is most suitable to solve with Monet. The number of suited algorithms is unfortunately quite minimal. A problem which might yield good results is to define whether or not a given graph is bipartite.

5.3.1 Bipartite graph basics

A bipartite graph is a graph with nodes and edges ($G = (N, E)$) and the nodes, N can be split into two sets: V and U. All edges in the graph connect two nodes with each other and every edge has exactly one node from the V set and one node from the U set. So actually all neighbours of a node which is a member of V are members of set U and the other way around. If somewhere in the graph two nodes are direct neighbours and they are members of the same set, the graph is not bipartite. An example of a bipartite graph is shown in figure 33.

Bipartite graphs are often used for matching problems like assigning students to hospitals during their doctors training or to assign staff members to tasks. When a number of persons have to perform a number of tasks and it has to be a one-to-one mapping (one person can do at most one job at a time and on job needs exactly one person), an edge is drawn between a person and a task if the person is capable of performing the task. All persons are in set V and all tasks are in set U. After creating such a bipartite graph the Marriage theorem [13] can be applied which yields a perfect matching: all tasks are matched with a person who is able to perform that task. It can occur that one person is the only one who can perform two specific tasks and therefore a perfect matching is impossible because only one task can be assigned to one person. When a matching is possible the Marriage theorem, or Hall's theorem, will find the perfect matching.

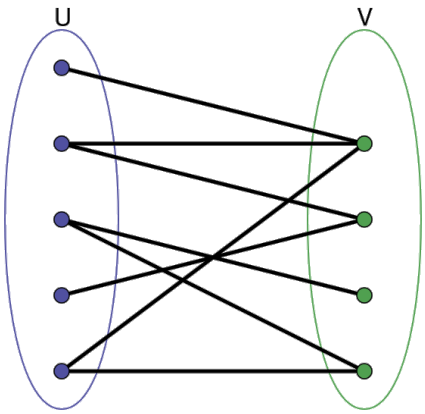

Figure 33: An example graph which is bipartite: all nodes are a member of V or U and only edges between the two sets are present.

The normal bipartite problem is to divide all nodes of a given graph into two sets. When the graph contains an odd-size cycle the graph cannot be bipartite: this requirement is often used to decide whether or not a graph is bipartite. Initially none of the nodes in the graph are assigned to a set and during the execution of the algorithm the sets will be filled and when an odd-sized cycle is found the algorithm will terminate because the graph cannot be bipartite.

The bipartite problem described in this section is a bit different from the usual bipartite graph problem: the nodes in the graph are already divided over the two sets. Changing this algorithm into the usual bipartite graph problem is not difficult and is described at the end of section 5.3.2.

5.3.2 Is a graph bipartite?

The marriage theorem can only be applied on bipartite graphs. Sometimes it is not certain a graph is actually bipartite but it is essential to be sure of the bipartiteness of the graph. The idea of determining the bipartiteness of a graph is essentially not difficult. Initially a random node is chosen which is a member of, let's say, set V. All neighbours of the start node must be members of set U, all the neighbours of these neighbours must be members of set V, etcetera: in essence a breadth first search is applied.

Every iteration all neighbours of the last found nodes will be found and it must be checked that all of these nodes are in the other group as their neighbours which were found in the previous iteration. The termination criteria are very clear:

1. When a neighbour of node X is found which is a member of the same set as node X, the graph is not bipartite and the algorithm can terminate.
2. When all nodes have been visited by the algorithm and the algorithm did not yet terminate, the graph is bipartite and the algorithm can terminate.

The graph can be represented as a table *edges* with three columns: source, sink and set. The two different sets will be set 1 and set 2 and the nodes are, just like in the Shortest path problem, represented by their IDs. The value of the *set* column determines the set of the sink node. Besides the *edges* table some tables must be created in order to apply the techniques of the Semi naive evaluation strategy and the Magic set. All tables which will be used are shown in figure 34.

Table name	Columns		
edges	source int	sink int	set int
mx	node int		
temp_curr	node int	set int	
curr	node int	set int	

Figure 34: The four tables which are used to decide whether or not a graph is bipartite

The *edges* table contains all information about the graph itself. The *mx* table contains all nodes which have been visited so they will not be used to find their neighbours more than once. The *curr* table contains all nodes, together with the set of which they are a member, for which the neighbours must be found in the next iteration. The *temp_curr* table is created so the Semi naive evaluation strategy and the Magic set technique can be used.

Initially one random edge is chosen from the *edges* table and the source node, together with the set it is a member of, is stored in the *curr* table. The node is also stored in the *mx* table. All neighbours of all nodes present in the *curr* table will be requested and they will be stored in the *temp_curr* table. The *curr* table will be emptied and all unique nodes for which no neighbours have been found yet and which are stored in the *temp_curr* table will be stored in the *curr* table. To define which nodes already have found their neighbours the *mx* table is used. Finally all nodes from *curr* will be stored in the *mx* table and the *temp_mx* table is emptied.

This algorithm will eventually visit all nodes of the graph which are connected to the start node. Now only the termination criteria must be implemented. Every iteration all nodes in the *temp_mx* table must be a member of the same group. This group must not be the same group as in the previous iteration. This check must be done just before the *curr* table is emptied because that is the only place where the group information is stored for all nodes of the last iteration.

The pseudo code for the algorithm is shown in program 3. The queries which are used to implement the algorithm are shown in program 4.

Program 3 Pseudo code for the algorithm to decide whether or not a graph is bipartite

```
//initialization phase
insert random node and its set into curr
insert the same node and its set into mx

//the loop
while all neighbours are of another group as the current nodes and curr is not empty {
        //insert all neighbours of the current front into the temp_curr table
        INSERT INTO temp_curr (π_{edges.sink,edges.set}(curr ⋈_{curr.node=edges.source} edges))

        //termination criterion when the graph is not bipartite
        if at least one element in temp_curr is of the same group as elements in curr: terminate

        empty curr

        //insert all node and set values of the temp_curr table, for which the node values are
        not present in the mx table, into the curr table
        INSERT INTO curr (π_{temp_curr.node,temp_curr.set}(σ_{mx.node=NULL}(temp_curr⟕mx)))

        copy all values from curr.node into mx
        empty temp_curr

        //termination criterion when the graph is bipartite: no more neighbours are found
        if curr is empty: terminate
}
```

Program 4 The SQL queries which are used to implement the algorithm shown in program 3

INSERT INTO temp_curr (SELECT edges.sink, edges.set FROM curr, edges WHERE curr.node = edges.source)

SELECT COUNT(*) FROM temp_curr, (SELECT DISTINCT(set) FROM curr) AS curr WHERE temp_curr.set = curr.set

DELETE FROM curr

*INSERT INTO curr (SELECT temp_curr.node, temp_curr.set FROM (SELECT DISTINCT * FROM temp_curr) AS temp_curr LEFT JOIN mx ON temp_curr.node = mx.node WHERE mx.node IS NULL)*

INSERT INTO mx (SELECT node FROM curr)

DELETE FROM temp_curr

SELECT COUNT(*) FROM curr

The second and the last query are the implementation of the termination criteria. The second query returns the number of elements of the current iteration which are not in the correct group in order for the graph to be bipartite. As long as this query returns zero the graph is bipartite but as soon as the result is larger than zero the algorithm can terminate and the graph is not bipartite. Just like in program 1 the termination criteria are performed by a query but the actual check is done in the Java code.

The last query returns the number of elements of which the neighbours has to be found in the next iteration. When no elements are present in the *curr* table no new neighbours will be found and the algorithm can be terminated. When the algorithm is terminated by this rule the graph is bipartite.

The described algorithm works on a graph where the nodes are already divided into two sets. It is not hard to change this algorithm into the usual bipartite problem where nodes have to be divided into two sets. When a node is selected which is already member of one of the sets, it must be the other set than the selected node of the previous iteration. If a node is not yet a member of one of the sets it will become a member of the set of which the selected node of the previous iteration is not a member.

5.3.3 Experiments with bipartiteness

In order to verify the intuition about the performance of the described bipartiteness problem some experiments are done. These experiments involved random graphs, a basic main memory implementation of a bipartite algorithm and the described Monet based algorithm. The main memory algorithm used is not about detecting odd-sized cycles in the graph. The graph is represented by a HashMap with the ID of a node as the key and the Node itself as a value. The

Node contains the ID of the Node, the set it belongs to, all IDs of its neighbours and whether or not it has been visited before.

The algorithm is very straight forward: start with a random node and store the ID of this node in a HashSet so only unique IDs will be present. As long as the HashSet is not empty get the first node, test whether or not all neighbours are not in the same set as the random node, add all neighbours which have not yet been visited to the HashSet and mark the current node as 'visited'. As soon as one of the neighbours is a member of the same set the algorithm can terminate: the graph is not bipartite. When the HashSet is empty at any point during the execution of the algorithm the graph is bipartite.

The graphs used to for a basic comparison between the two algorithms are randomly created: two sets of nodes are created and between these sets the edges are placed randomly. When the algorithms are applied on such graphs the Monet based algorithm performs very good: it takes the algorithm generally less than 3 seconds to decide whether or not a graph with 200.000 nodes and 4.000.000 edges is bipartite. The main memory based algorithm takes about 36 seconds to draw the same conclusion. The creation of the data structures will take quite a lot of time too: it takes about 75 seconds to store the graph in a database (which only have to be done once, of course) and it takes about 10 seconds to read the graph into main memory.

The Monet based algorithm seems to perform very good but unfortunately this is not really fair: all nodes of the graphs which have been created are visited by the Monet algorithm in about 7 iterations. This is the main problem with the randomly generated graphs: the randomly created edges can be placed between any two nodes. When more iterations have to be used to visit all nodes of the graph the performance of the Monet algorithm will decrease.

A random graph with approximately the same number of nodes and edges as the Benelux roadmap has been created and the Monet algorithm only took about 6.4 seconds to decide whether or not the graph was bipartite. With such graphs the algorithm had to run about 30 iterations. The main memory algorithm, however, took about 11 minutes to find the same result as the Monet algorithm, reading the graph into main memory included. With the original Benelux roadmap and *Groningen* as the source it took more than 1000 iterations to define the entire graph connectivity while with a randomly created graph with the same amount of nodes and edges it only takes 30 iterations.

Mainly because of this difference and a bit because the used main memory implementation is not the best implementation it is not at all a fair comparison and not much can be said about the performance of the Monet based bipartiteness algorithm. In order to get some more information the randomness of the graphs must be limited. With the earlier created random graphs there is no limit from where to where an edge can go. In the newly created random graphs there are two sets, set A and set B, which contain the same number of nodes but no node is a member of both sets. When a graph is created a node, a, from set A is randomly chosen and from a small subset of B a neighbour, b, of a will be chosen. The subset of nodes in set B for every node in set A is known.

With this restriction graphs will be created which will take the Monet algorithm much more iterations before all nodes are visited. The size of the subset of a node a is very important in the runtime of the algorithms and this size will be referred to as the *neighbour_size*. The originally created random graphs have a very large *neighbour_size*: the number of nodes in one of the sets. When the *neighbour_size* decreases, the Monet algorithm will need more iterations before it is certain a graph is bipartite and the runtime will increase. Because of the results of the graph connectivity algorithm it is expected that the runtime of the Monet algorithm will increase pretty fast when the *neighbour_size* will decrease.

In the experiments a graph is created which is half the size of the Benelux roadmap and the node/edge proportion is kept the same. The runtimes of both the Monet algorithm and the

main memory algorithm on graphs with different *neighbours_sizes* are shown in figure 35.

neighbour_size	50	700	1400	2100
Monet iterations	14905	986	564	340
Runtime Monet (ms)	1014355	115361	82281	42626
Runtime main memory (ms)	23251	41527	56414	59390

Figure 35: Results of both algorithms on graphs with a fixed number of nodes and edges but with a different *neighbour_size*

This figure shows that when the *neighbour_size* increases, the number of iterations needed by the Monet algorithm decreases. The expectation that the performance of the Monet algorithm would decrease when the *neighbour_size* gets smaller is correct. The performance of the main memory algorithm actually increases with a decreasing *neighbour_size*. There is a point where the performance of the two algorithms cross: with a *neighbour_size* between 1400 and 2100 the Monet algorithm starts outperforming the main memory algorithm. When higher values for the *neighbour_size* are used the Monet algorithm will yield a better performance compared to the main memory algorithm.

It is not very clear why the main memory algorithm performs better when a smaller *neighbour_size* is used. The most obvious reason is because the cache is used more optimal when a small *neighbour_size* is used. The neighbour, node b, is more likely to have some same neighbours as node a when a small *neighbour_size* is used so the chances increase that the neighbours of node b are still in the cache when node b has to be evaluated. When a large *neighbour_size* is used the neighbours of two nodes who are neighbours of each other become much more random.

With a randomly created graph with approximately 800.000 nodes and 1.880.000 edges and with a relatively high *neighbour_size*, the Monet algorithm outperforms the main memory algorithm. It is not tested what happens when smaller or larger graphs are used but it is expected that the relation between the size of the graph and the size of the *neighbour_size* are somehow related. When the bipartiteness of a huge graph must be calculated and a very small *neighbour_size* is used, a main memory algorithm will outperform the Monet algorithm. When the *neighbour_size* is large enough the Monet algorithm will yield better results. It is interesting to take a closer look at what happens when the bipartiteness of smaller and larger graphs have to be found with different values for the *neighbour_size*.

5.3.4 Conclusions on bipartiteness

The algorithm to decide whether or not a graph is a bipartite graph is the algorithm most likely to take advantage of the qualities of Monet. The techniques described earlier in this paper are all applicable to the bipartiteness problem: the use of a Magic set, the Semi naive evaluation strategy, the use of Monet as a data structure and a breadth first search like algorithm can be used with it.

The power of Monet also lies in the size of the datasets: the larger the datasets the better Monet will work compared to other database systems and hopefully compared to dedicated main memory algorithms. If the performance of the Monet implementation of the bipartiteness algorithm will not come close to the performance of a dedicated main memory implementation of the bipartiteness algorithm it is pretty safe to say the main idea of this paper has failed.

With the performed experiments it is clear the Monet algorithm can outperform a main memory algorithm but only under certain conditions. When large graphs are used and the *neighbour_size* is large enough the Monet algorithm will have a better performance than the main memory algorithm. When the *neighbour_size* decreases the main memory algorithm will perform

better and better and the Monet algorithm will perform worse and worse. This is because of the same reasons as are discussed in section 4.6: when a low number of iterations with the Monet based algorithm are required in order to visit all nodes connected to the randomly chosen first node, the Monet algorithm performs very good. When the number of iterations increase, the performance of the Monet algorithm will decrease.

The graph size and the size of the *neighbour_size* are most probably somehow related to each other and the performance of the Monet based algorithm depends on these variables. It is interesting to continue experimenting with these variables in order to define more concrete conclusions.

5.4 Results and conclusions

Just like with the single source to single sink shortest path problem, the other path problems will not use the advantages of Monet enough. Dedicated main memory algorithms perform small operations on small parts of the graph in order to finish one iteration. The results of one iteration are necessary in order to calculate the results for the next iteration. It is possible to use a Monet based algorithm to solve these problems but when the idea of the dedicated main memory is implemented in SQL there will be a lot of overhead and when another implementation is used a lot of unnecessary calculations will be done. Also a lot of extra iterations are necessary in order to be sure the algorithm will yield correct results.

Problems for which the solutions can be based on the breadth first search algorithm and for which the data sets can be very big are more suitable to be solved with Monet as the data structure. Some problems like the Subsetsum problem are such problems but unfortunately a lot of such problems are NP-complete. Because of the NP-completeness it is of no use to implement these algorithms with Monet as the data structure: the relative performance of Monet will increase when the size of the datasets increase but with NP-complete problems it will still take a very long time to solve the problem.

A problem which is not NP-complete, for which the breadth first search algorithms works pretty good and where very large datasets can occur is checking the bipartiteness of a graph. Even the termination criteria are very clear and not hard to calculate. The experiments show when graphs with certain characteristics are used the Monet based algorithm can outperform the main memory algorithm. To get more concrete results more experiments have to be done where the size of the graph and the size of the *neighbour_size* must vary. Based on these experiments it can be decided with which exact conditions a Monet based algorithm to decide the bipartiteness of a graph can compete with a main memory algorithm.

6 Software

During the entire project a lot of programming is done. The program language is Java 6 and the environment used is Eclipse. In order to understand Datalog and create a link between Datalog and SQL two Datalog engines are used: DES [1] and IRIS [2]. Initially the idea was to create a translator from Datalog queries to SQL queries and use the translated queries to apply Datalog rules on a Monet database system. This idea is not used later on but during the process of creating the translator it was very informative and educational to use DES and IRIS in order to create a good understanding of Datalog and Magic sets. This section describes the not commonly known software used: DES, IRIS and Monet.

6.1 DES

DES stands for Datalog Educational System and is an open source implementation of a basic deductive database system. It can load Datalog rules and the statements which are supposed to be present in the database. When the load is successful Datalog queries can be fired at the system and the results are returned just like a Datalog database system would do. It is also possible to link DES to an existing database system and use the information in the database to answer queries. DES is an easy system to use and it is a nice way of learning more about Datalog while playing around with rules, statements and queries.

6.2 IRIS

DES is not a system which is optimized to handle queries as fast as possible or to take care of huge amounts of data. It is also not possible to call on DES from Java. In order to create a test to check if the translation of a Datalog query to an SQL query is successful IRIS can be used. IRIS stands for Integrated Rule Interface System and it is an reasoning engine for rule-based languages like Datalog. The JARs and the documentation are downloadable from the website.

IRIS is not as easy to use as DES but when the documentation is read it is possible to create a simple evaluation machine in which rules can be inserted and queries can be asked. The results returned are correct and can be transformed into a format for which it is easy to compare it with the results of a traditional database system, like PostgreSQL, MySQL or Monet. It took quite a lot of time to get IRIS to work as is expected.

One very annoying aspect is the 'column dancing' of IRIS: it is not possible to determine in which order the columns of the results table will appear. Because of this it is difficult to compare the results of a Datalog query with an SQL query and a special 're-ordering method' had to be created so the results of both the Datalog query and the SQL query are ordered the same way.

After building a shell around all IRIS specific code it was easy to automate the process of comparing the results of an SQL query with the results of an Datalog query and a quite complicated translator was written.

6.3 MonetDB

The Monet database system is explained in section 2. It is not very hard to install a Monet database system on a Linux machine and within a couple of minutes it is clear how databases can be created and entered. The most important difference between database systems as PostgreSQL and MySQL is the presence of the main process Merovingian which is necessary to start Monet databases. Once logged in to a database Monet works pretty much the same as PostgreSQL and MySQL.

A JDBC is downloadable so Monet can be used from Java programs. Because of the presence of the JDBC it is easy to connect to the Monet database and perform queries on it. Unfortunately the Monet database does not support all functionality offered by the JDBC and a lot of exceptions were thrown during the testing of queries. Initially very large queries were created and the error

java.sql.SQLException: Read from localhost:50000: End of stream reached (mserver still alive?)

was shown a lot. When the in between result were materialized this problem did not occur anymore. Some other problems did occur, though. When the **twoway** algorithm was applied about six or seven times in a row the Monet server would crash and the error usually was:

java.sql.SQLException: merovingian: an internal error has occurred, refer to the logs for details, please try again later

After this error it takes a while before the Monet database is rebooted and is reachable from the Java environment again. Because of this it was not easy to do all experiments and all testing. Especially because the first experiment run on a newly started Monet database yields much better results than all experiments run after the first experiment.

Because all the shortcomings of Monet were clear before the final experiments were run it was not very problematic to keep a look out for unexpected results and crashing databases. So after rerunning some of the experiments the results as are presented in this paper were found.

7 Conclusions

In this paper the use of the Monet database system as a data structure for graph problems is suggested. Usually dedicated algorithms are created which use a specific main memory data structure to solve the graph problems as fast as possible. When a Monet database is used as a data structure and the operations to solve a problem are SQL queries, the performance depends on the Monet database system. With main memory algorithms this depends on the implementation of the programmer or on the implementation of the programmers of the programming language used. Also Monet is known to yield good performances when large datasets are used.

The Datalog database language is a good basis for the transformation of a graph problem algorithm into SQL queries. The shortest path problem is the first algorithm considered. The more basic algorithm used to discover if a path exists between two nodes in a graph, which is the core algorithm which must also be used in order to find a shortest path between the same two nodes, does not yield a very impressive performance. This algorithm is actually the graph connectivity algorithm with the small addition that it can also terminate when a certain node is discovered. The Dijkstra algorithm with main memory data structure, which really finds the shortest path, is approximately at least twenty times as fast as the Monet based graph connectivity algorithm with the use of a sink node.

One iteration of the dedicated algorithm with a main memory data structure performed with the shortest path problem will only perform operations on a small part of the graph. The results of this iteration are needed in order to yield correct results in the next iteration. Problems which are best to solve with such kind of algorithms will not yield an impressive performance when they are solved with a Monet based algorithm.

Other shortest path problems, like from one to all other nodes or from all nodes to all nodes, have the same problem as the shortest path algorithm from one node to one other node. When a graph problem can be solved with a breadth first search approach based algorithm it will probably yield much better results than the shortest path algorithms. Some NP-complete problems can be solved by using a breadth first search approach but because they are NP-complete it will take exponentially more time when the dataset increases. So solving NP-complete problems with Monet based algorithms is also not a good idea.

A problem which seems to fit very well to the newly created description for a Monet based algorithm is to decide whether or not a graph is bipartite. The basic Monet algorithm is a lot like a breadth first search and the Magic set technique together with the naive evaluation strategy will increase the performance of the algorithm. The experiments done show that with certain conditions the Monet based algorithm can yield a better performance than the main memory algorithm. More experiments have to be done before a more concrete conclusion can be drawn on when the Monet based algorithm outperforms the main memory algorithm.

Only breadth first search based algorithms without a lot of smart tricks and adjustments on small parts of a graph can yield good results when the Monet database system as a data structure and SQL queries are used to perform the algorithm.

8 Future work

The conclusion of this paper is that some algorithms might yield a pretty good performance as long as they are based on a breadth first search approach and not a lot of small operations have to be done on a small part of the graph. Such an algorithm is the algorithm to determine wether or not a graph is bipartite.

As is mentioned in the conclusions some more experiments have to be done in order to come to more concrete conclusions with regards to the bipartite graph algorithm. The main memory algorithm which is used is very basic and some optimizations can be done in order to achieve a better performance. Other experiments should show with which conditions the Monet based algorithm outperforms the main memory algorithm: the size of the graph, the node / edge ratio and the size of the *neighbour_size* are important in these experiments.

Monet is very good in handling large datasets and when more dense graphs are used, the datasets will increase. Main memory algorithms might not be as good in handling large datasets as the Monet based algorithm and this might yield results in the advantage of Monet based algorithms. So when graphs are created the ratio edges / node is relevant and it is interesting to research the performance of both the Monet based algorithm and the main memory algorithm when the ratio increases.

Whether or not the proposed research yields good results another interesting research is to directly write the algorithms in MAL instead of using SQL to execute the algorithms on a Monet database. MAL stands for MonetDB Assembly Language. In this paper algorithms are created based on Datalog rules with Magic sets and are written down in SQL statements. It is also possible to directly write the created algorithms in MAL which might yield a better performance. It is not at all an easy assignment to create a MAL program which outperforms the SQL optimizers of Monet but the advantages of Monet could be used more efficient.

References

[1] Datalog educational system. http://www.fdi.ucm.es/profesor/fernan/des/.

[2] Iris reasoner. http://iris-reasoner.org/.

[3] Monet db website: Query processing at light speed. http://monetdb.cwi.nl.

[4] Tpc benchmark h. http://www.tpc.org/tpch/spec/tpch2.12.0.pdf.

[5] *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, 2001.

[6] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1).

[7] P. A. Boncz, S. Manegold, and M. L. Kersten. Database Architecture Optimized For The New Bottleneck: Memory Access. Very Large Data Base Endowment., 1999. bibliographical data to be processed – In Proceedings of the International Conference on Very Large Data Bases (VLDB), pp 54-65, Edinburgh, United Kingdom, September 1999 – Very Large Data Base Endowment. – 12.

[8] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. on Knowl. and Data Eng.*, 1(1):146–166, 1989.

[9] Alain Colmerauer and Philippe Roussel. The birth of prolog. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 37–52, New York, NY, USA, 1993. ACM.

[10] George P. Copeland and Setrag N. Khoshafian. A decomposition storage model. In *SIGMOD '85: Proceedings of the 1985 ACM SIGMOD international conference on Management of data*, pages 268–279, New York, NY, USA, 1985. ACM.

[11] Edsger. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[12] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.

[13] Philip Hall, B. Manasterf, and Joseph G. Rosensteinj. Effective matchmaking (recursion theoretic aspects of a theorem of, 1971.

[14] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing Database Architecture For The New Bottleneck: Memory Access. 2000. bibliographical data to be processed – The VLDB Journal, 9(3):231-246, December 2000 Springer-Verlag – Springer-Verlag – 16.

[15] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing main-memory join on modern hardware. Technical report, Amsterdam, The Netherlands, The Netherlands, 1999.

[16] Jeffrey D. Ullman. *Principles of database and knowledge-base systems, Vol. I.* Computer Science Press, Inc., New York, NY, USA, 1988.

[17] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. W. H. Freeman & Co., New York, NY, USA, 1990.