Efficient Data Structures For Decision Diagrams

Supervisors:

Thomas Léauté

Radoslaw Szymanek

Professor:

Boi Faltings

Semester Project

Automn 2008 term

Stéphane Rabie

Summary

Introduction

Dynamic Programming Optimization (DPOP) is an algorithm proposed to solve distributed constraint optimization problems. In order to represent the multi-values functions manipulated in this algorithm, a data structure called **Hypercube** was implemented. A more efficient data structure, the **Utility Diagram**, was then proposed as an alternative to the Hypercube. DPOP also required the implementation of several operations (such as **join**, **project**, **slice**, **split** and **reorder**) on these two data structures.

This project is a follow-up of Nacereddine Ouaret's master thesis, which consisted in implementing all of these data structures and theirs associated operations. As DPOP may have to work on very large decision diagrams, and perform a lot of successive operations on them, having implementations which are efficient in term of speed and memory is critical. The aim of this project was therefore to seek for new ways to improve the already implemented functions for hypercubes and utility diagrams, both in term of execution time and memory consumption.

This report will thus first present a quick overview of hypercubes, utility diagrams, and their associated operations (a more complete description of these objects, as well as the details about their original implementation, can be found in Nacereddine Ouaret's report on Efficient Data Structures for Decision Diagrams). The second part will then cover various improvements made to their implementation during the course of this project. Finally, a variant for the methods used by hypercube, more economical in term of memory as it reuses existing hypercubes rather than creating new ones, will be presented.

1) Hypercubes, Utility Diagrams, and theirs operations

1.1 Hypercubes

A hypercube is represented by three parameters: its variables V, its domains D and its values U. $V = \{V_1, ..., V_n\}$ is the array of the variables contained in the hypercube, $D = \{D_1, ..., D_n\}$ is an array of domains ordered such that D_i is the domain corresponding to the variable V_i. Each D_i is an array where each entry is a possible value for V_i. Finally, $U = \{U_1, ..., U_m\}$ is an array containing the utility values of the hypercube. There is a utility value for every combination of possible variables values (which means that m is equal to the product $|D_1|...|D_n|$), and these values are ordered so that given a variables assignment, its corresponding index in the utilities array U can be found by a relatively simple formula.

1.2 Utility Diagrams

Hypercubes assume that for every combination of variables values, there is a utility value, but this may not be always the case. As a result, hypercubes are an inefficient method to represent sparse multi-values functions, and that is why utility diagrams were introduced to overcome this problem. With utility diagrams, a multi-valued function is represented by a graph, which provided a much more compact, and therefore efficient, way to encode the possible combinations of variables values.

A diagram is thus composed of nodes and edges. Nodes are labeled by the names of their corresponding variables, except the terminal node which is just here to indicate that there are no more variables. The edges are labeled by a set of values that the variable (corresponding to the node from which the edge comes) can take. The diagram is constructed such that every possible path from the root to the terminal node in the graph corresponds to a possible variables assignment.

The array of utility values then contains the values of the function and contains an entry for every path in the diagram which starts at the root and ends at the terminal node. Path numbers are generated according to a depth-first traversal of the diagram, assuming an order on the edges coming out of the nodes.

Besides, in this implementation of utility diagrams, each edge is also characterized by a step parameter. This step value is an integer which can be recursively computed as the sum of these two values:
- the step value of the previous edge in the list of edges of the parent node
- the number of paths from the destination of the previous edge to the terminal node

If this edge is the first of the list, its step value is then equal to 0.
The important property of step values is that by computing the sum of steps on all edges of a given path, we directly obtain the index of the corresponding value in the utilities array.

1.3 Join

Joining two multi-values functions consists in creating a new function which is equal to their sum, its variables list being the union of the variables of the two input functions.

For hypercubes, we can then simply create the domains of the new hypercube: if the variable belongs to the two input hypercubes, its domain is equal to the intersection of the two original domains (if this intersection is empty, a null hypercube is returned); if the variable belongs to only one hypercube, this domain is simply kept in the new hypercube. The entries of the new utilities array are then obtained by computing the sum of the utility values corresponding to the same variables values in the original hypercubes.

The principle of the operation is the same for utility diagrams, but as we cannot compute a global domain for each variable, the join is performed by recursively computing the intersection between the edges domains for each pair of nodes to join.

1.4 Project

Projecting some variables out of a decision diagram (hypercube or utility diagram) consists in reducing its dimension by removing these variables from its variables list. This also implies the reduction of the utilities array as for each of utilities values set obtained by fixing the variables to keep, and varying the variables to project out over their respective domains, only the maximum (or minimum, depending on a boolean parameter which is an additional argument of the function) is kept.

1.5 Slice

Slicing a decision diagram simply consists in reducing the domains of some of its variables, which also implies reducing the utilities array as only the values corresponding to combinations of variables values included in the provided sub-domains are kept.

1.6 Split

Splitting a utility diagram consists in removing the utility values that are greater (or lower, depending on a boolean parameter which is an additional argument of the function) than a provided threshold. For hypercubes, some variables domains have also to be reduced, and in most cases, some utility values smaller than the threshold have to be kept because of the assumption that there is a utility value for each possible combination of variables values. However, as there is not such assumption for utility diagrams, only the right values will be kept when splitting a utility diagram. The split operation can thus be used to obtain a utility diagram from a hypercube by having all values below (or above) a given threshold removed from the hypercube, which in most cases will produce a sparse multi-valued function.

2) Improvements of the code

The first step of this project was a preliminary warm-up phase consisting in small tasks, such as adding more commentaries when necessary, correcting some bugs and writing new boolean functions to be used by assertions (for example, checking that the variables order in two or more decision diagrams is consistent before joining them).

Besides performing these few improvements, the main goal of this phase was primarily to get a good overview of the source code before really starting modifying it. Some parts of this phase, which are described in section 2.1 and 2.2, required however more time and thought.

1.1 Test for augment method

For utility diagrams, the augment method takes for arguments a list of variables values and a utility value, adds a new path corresponding to the input variables values to the utility diagram, and adds the input utility value at the place corresponding to this new path in the utilities array.

This method is quite important as it is used in several other methods, and is not trivial to implement: when adding a new path, we have to check which part of this path already exists in the utility diagram, and create the part of this path which is not already present, then update the step values of all edges which have been modified by the introduction of this new path.

For these reasons, the augment method had to be tested, in the same way methods such as join or project were, in order to make sure the augmentation of a utility diagram with a new path does not create some errors. A first test function for the augment method was already implemented and its principle was very simple: create a random utility diagram, add a random new path to this diagram with the augment method, then check that its associated utility value has effectively been added to this diagram.

This test had the major drawback that it only tested whether the path was really added in the utility diagram, but not if this path was added at the right place or if the steps values were rightly updated. That is why a more complex test was implemented during the warm-up phase. This new test function still creates a random utility diagram, then constructs step by step another utility diagram by augmenting it successively with every path in the original hypercube, and finally checks the two utility diagrams are identical.

As it is more complete, this test first outputted many failures (whereas the first test was always successful), which helped spotting and correcting several errors in the code of the augment method that the first test was unable to detect.

1.2 Projection assignments for Utility Diagrams

The projection function is special as it actually returns a ProjOutput object which consists in two fields: a *space* object which is the actual hypercube or utility diagram resulting from the projection, and an *assignments* data structure giving the optimal values of the projected out variables for each utility values of the new decision diagram.

In order to represent the *assignments* object, a Hypercube cannot be used as it requires its utilities values to be comparable or addable, but the assignments are represented as an array list of variables values, which is not comparable. That is why a BasicHypercube class was created, which basically is a Hypercube class in which the utilities are neither addable neither comparable, and the computation of the assignments BasicHypercube was implemented for hypercubes.

However, this was not implemented for utility diagrams, so another important part of the warm-up phase was to modify the project method so that we also keep track of the optimal variables assignments corresponding to utility values that are finally kept in the utility diagram. The representation of these assignments in the projection output represented a supplementary difficulty.

Indeed, the first idea was to create a new class BasicUtilityDiagram, analogous to BasicHypercube, but for utility diagrams. However, doing so required to deeply modify the Node and Edge classes in order to make them compatible with this new class, which would have taken too much time within the scope of this project, so this solution was abandoned.

Instead, the assignments are still represented with a BasicHypercube object, but as there may not exist a utility value for every variables values, a null value is used in the utility array when there is no path corresponding to these variables values in the utility diagram (see Figure 1 for an example of how the optimal assignments are represented for the result of a utility diagram projection). As a result, the utility array may be very sparse and a BasicHypercube is not the ideal way to represent the assignments for a utility diagram projection output either, so properly implementing the BasicUtilityDiagram class might be the next step in the future.

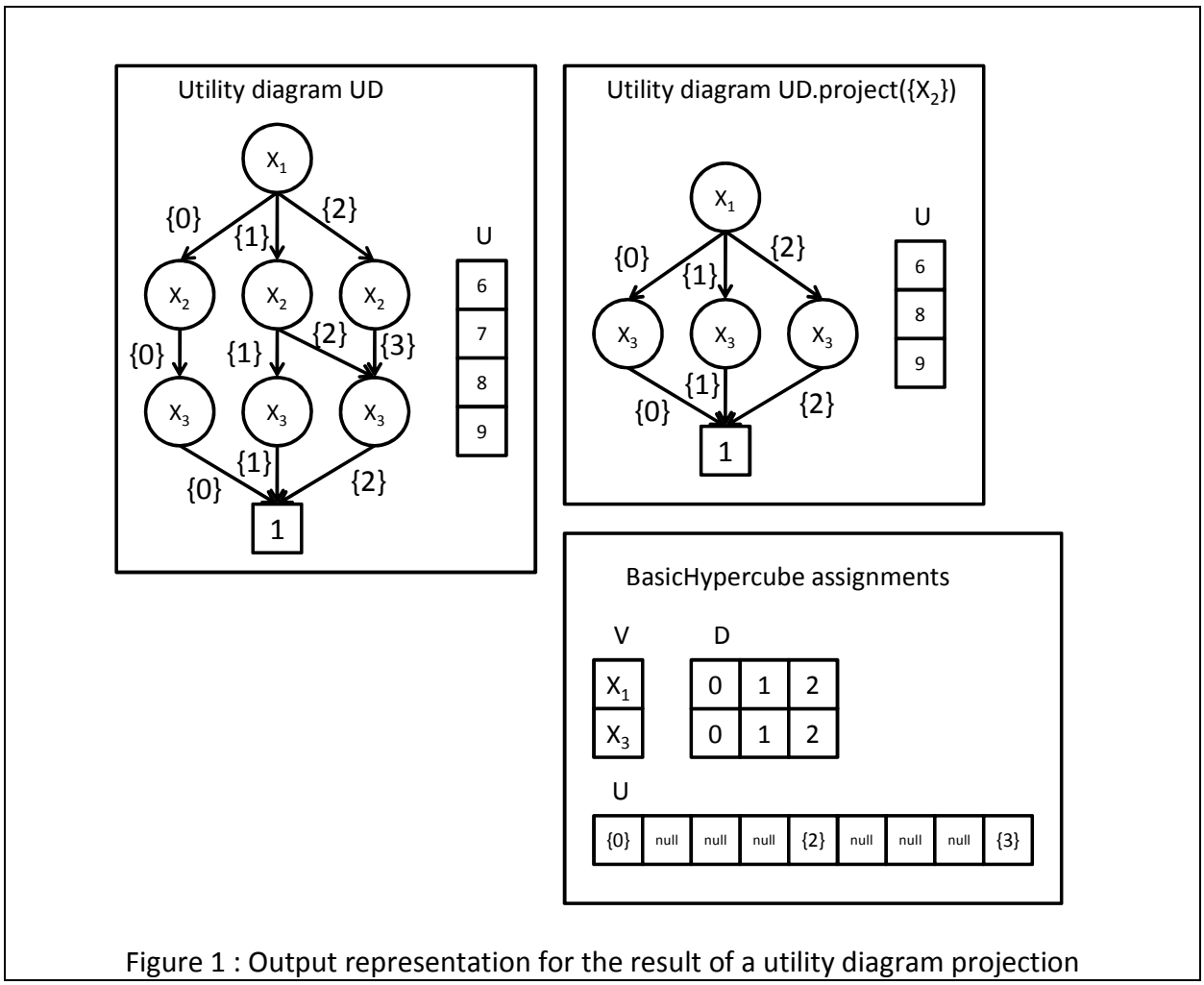


Figure 1 : Output representation for the result of a utility diagram projection

1.3 Join functions without assumptions about the variables order

In the original implementations of the join function both for hypercubes and utility diagrams, it was always assumed that the variables order between the two decision diagrams is consistent (i.e. for instance, if variables X1 and X2 are present in both diagrams, and X1 is before X2 in the first diagram, X1 is necessarily also before X2 in the second diagram), which limits the usability of the join function.

It is not such a problem for hypercubes, as the method changeVariablesOrder can always be used to reorder one of the hypercubes so that the variables order is finally consistent between the two hypercubes. However, this operation can be expensive; therefore a new implementation of the join operation that relaxes the assumption about the variables order and does not call the method changeVariablesOrder was implemented.

The first part of this implementation was first to write a joinAnyOrder method which can join two hypercubes no matter the variables order, without having to pre-reorder one of the hypercubes. Theoretically, using joinAnyOrder between two hypercubes is faster than first using changeVariablesOrder on one of the hypercubes then joining them, however experimental results show that, practically, the difference is negligible as the changeVariablesOrder method is relatively cheap for hypercubes.

The problem was however more serious for utility diagrams as no changeVariablesOrder method was implemented, and as a result, it was impossible to join two utility diagrams with inconsistent variables orders. The first step to correct this was therefore to implement a changeVariablesOrder for utility diagrams. This reordering algorithm is very simple:

```
UtilityDiagram changeVariablesOrder(String[] new_order)
01: r ← new empty utility diagram
02: for every path p in the original utility diagram
03:     q ← p reordered according to new_order
04:     augment r with q
05: return r
```

Though making the joining of two diagrams possible no matter what their variables order are, this algorithm is quite naive and when the size of the utility diagram to be reordered grows, the execution time of this method quickly becomes very high. In order to improve that, a joinNoReorder method for utility diagrams, which performs the reordering in the same time as the join, and thus do not call the changeVariablesOrder method, was implemented, but the time taken by this method on big utility diagrams (i.e. with several thousands of utilities values) is also too high, so a more efficient implementation has to be found before its use can become really interesting.

1.4 Optimization of methods for Utility Diagrams

For most methods for utility diagrams (with the notable exception of the projection function, in which a temporary array list is used to store the utility values while exploring the diagram to project), two passes through the utility diagram are needed: one to construct the resulting utility diagram, then one to construct the utilities array corresponding to the new utility diagram. Indeed, as a raw array is used to store the utilities values, it cannot be created before its size, equal to the number of solutions in the resulting diagram, is known, and contrary to the case of hypercubes, this size cannot be easily pre-computed.

In order to optimize the concerned functions (join, slice and split), an idea was then to write a variant in which the utility diagram is walked through only once, and where the diagram and its utilities array are constructed in this same pass. That is why a “OnePass” variant was written for all these functions: an array list (which does not require knowing the size of the new utility diagram beforehand) is used to store the utility values that will be kept in the resulting diagram during its construction, and that list is finally converted into the resulting utilities array at the end of the algorithm.

By using the accumulated steps value, it is easy to get the value corresponding to some path when the terminal node is reached. However, we have to make sure that these paths are explored in the right order so that the corresponding reached utilities values are inserted in the correct place in the array list, which is not always the case, especially with the join function (see Figure 2 for such a counter-example).

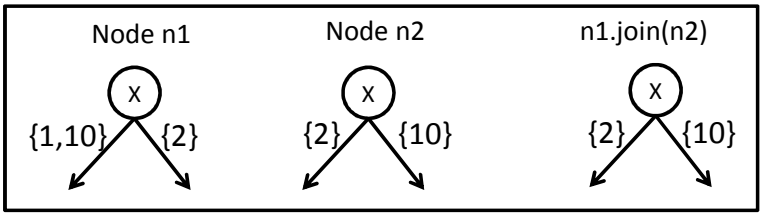


Figure 2 - With the classical implementation, edge {10} is created *before* edge {2} in the resulting node, as edge {1, 10} is explored before edge {2} in node n1, whereas the utilities values reached through edge {10} are situated *after* utilities values reached by edge {2} in the resulting utilities array, as edges are ordered by increasing lowest value.

Thus, in some cases, when exploring the edges leaving from a node, we have to first pre-compute the list of all potential resulting edges and sort them before recursively exploring them. For the join method for example, the following transformation had to be done to this part of the original algorithm:

```
Node join(Node node1, Node node2)
...
01: if(node1 and node2 correspond to the same variable)
02:     r ← new node having the same name as node1, with no edges
03:     for every edge e1 of the outgoing edges of node1 and  e2 of the outgoing edges of node2 do
04:         d ← intersection of the domains of e1 and e2
05:         if (d is not null)
06:             dst ← join(destination of e1, destination of e2)
07:             if (dst is not null)
08:                 add an edge to r having d as domain and dst as destination
09:     if (r does not contain any edges)
10:         r ←  null
11:     return  r
...
```

which thus becomes :

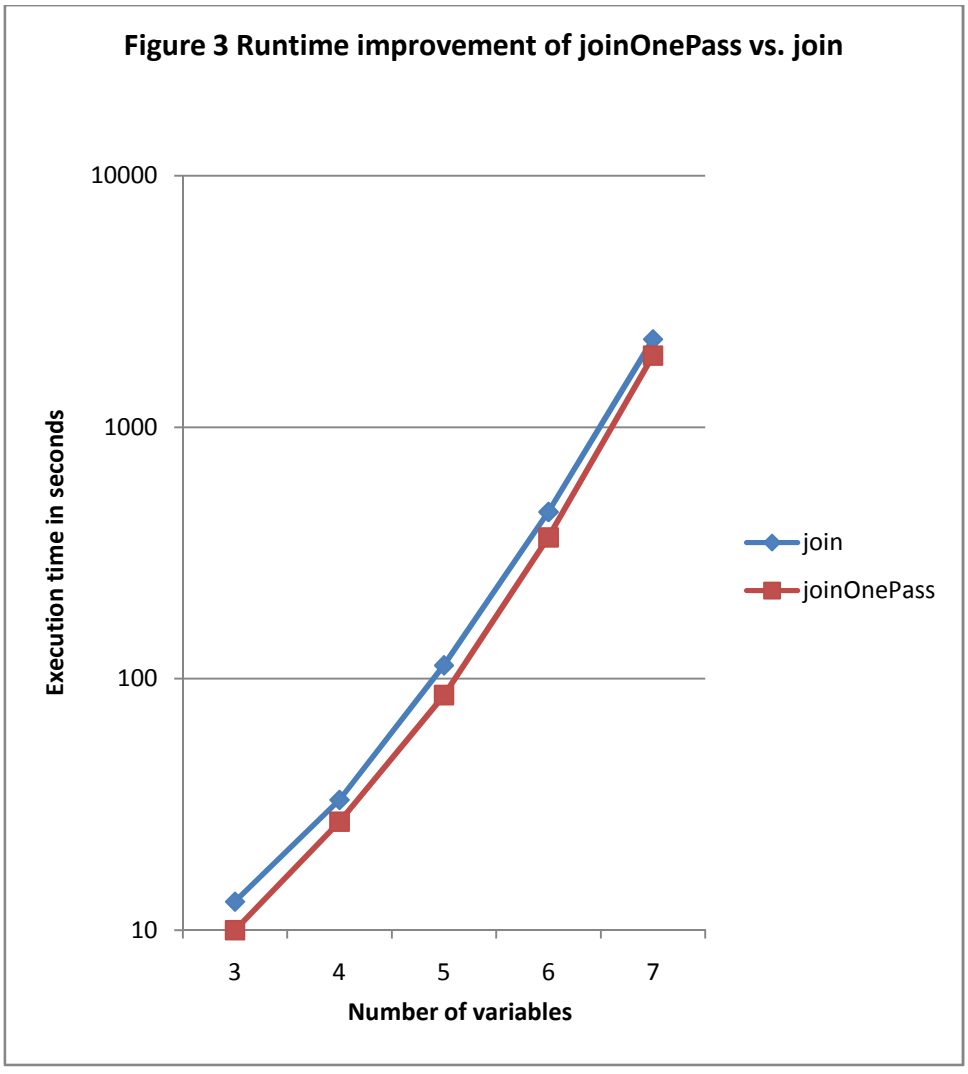
```
Node joinOnePass(Node node1, Node node2)
...
01: if(node1 and node2 correspond to the same variable)
02:     r ← new node having the same name as node1, with no edges
03:     l ← new empty sorted domains list
04:     for every edge e1 of the outgoing edges of node1 and  e2 of the outgoing edges of node2 do
05:         d ← intersection of the domains of e1 and e2
06:         if (d is not null)
07:             insert d at the right place in l
08:      for every domain d in l do
09:             e1 ← outgoing edge of node1 corresponding to domain d
10:             e2 ← outgoing edge of node2 corresponding to domain d
11:             dst ← join(destination of e1, destination of e2)
12:             if (dst is not null)
13:                 add an edge to r having d as domain and dst as destination
14:     if (r does not contain any edges)
15:         r ←  null
16:     return  r
...
```

It is important to note that this problem only arises when edges are labeled by domains containing more than one variable values. If the edges were always labeled by a single variable value, the edges of the new node would systematically be created in the right order, so a more specialized version of joinOnePass, in which the pre-computation of the domains list is not needed, could be written: this function would only work on utility diagrams in which all edges are labeled by single-valued domains, but would be more efficient for this special case of diagrams.

In order to compare the performance between different variants for a same operation, 10.000 random utility diagrams were generated, and the time taken to perform a given operation on all of them was measured for each implementation of this operation (the input parameters of the method being also randomly generated).

For slice and split, the gain is very small (about 5 %), which is not unexpected as for the split method, the creation of the utilities array is very cheap (it basically consists in just taking the original array and dropping all values above or below the input threshold). For join however, the execution time difference between the two implementations, presented in Figure 3, is much higher (about 25 %).

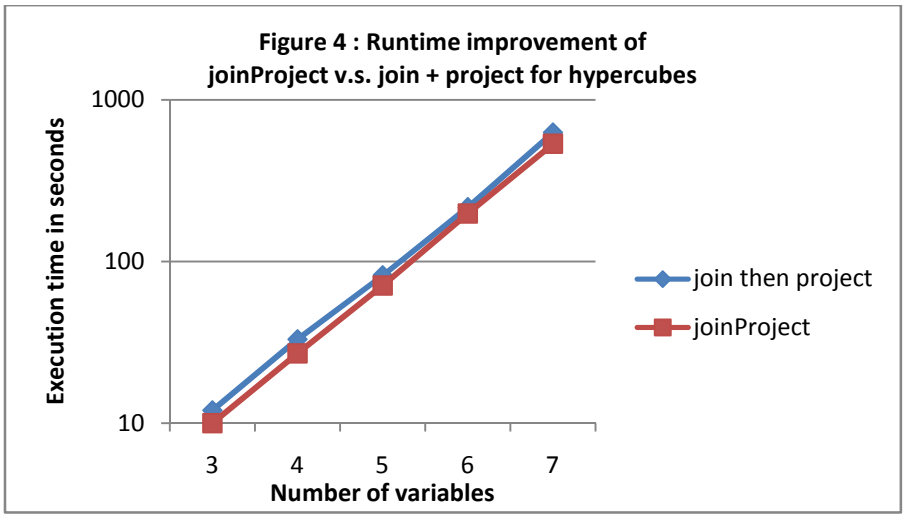
Figure 3 Runtime improvement of joinOnePass vs. join

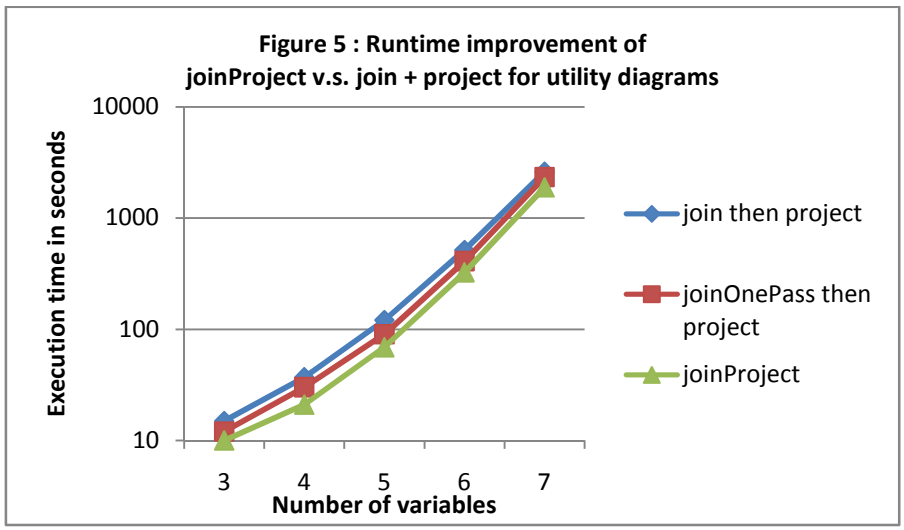


1.5 Implementation of joinProject

In DPOP, sequences of operations often consist in successively joining a diagram with another one, then projecting out some of its variables. A method performing both join and project at once rather than in sequence would therefore definitely prove itself useful, and would potentially require less time and memory as some layers introduced by the join would not even have to be created if the projection then removes these layers.

For these reasons, a new method joinProject was implemented both for hypercubes and utility diagrams: this method takes as arguments a diagram and a list of variables, and then returns the result that would be obtained if the diagram was first joined with the input diagram, and the given variables were then projected out. Figure 4 shows the execution time taken by joinProject on 10.000 random hypercubes, compared with the execution time of a join followed by a projection on these same random hypercubes. The experiments results for utility diagrams are presented in Figure 5; as the implementation of joinProject required only one pass through the diagram, it is actually fairer to compare it with joinOnePass, so the result of the application of joinOnePass followed by a projection has also been measured. These results showed a performance gain varying between 20% and 30% for joinProject.





3) Methods variants for Hypercubes

Most of the methods used in the hypercube class, such as join, project, slice or changeVariablesOrder create and return a brand new hypercube, which may be quite inefficient if the original hypercube is no longer needed. As the utilities arrays can be very large, with thousands of elements, this might be quite memory consuming in a program like DPOP where a lot of successive operations have to be made on hypercubes. When calling a method which creates a new hypercube, rather than creating a new one and discarding the original one, it could be more useful to modify the existing one. Hence, implementing a variant for all these methods, which when possible does not require the creation of a new -possibly big- utilities array, but only modifies the existing one, might then solve this memory issue.

In fact, with the exception of the join method, all methods always return a hypercube with fewer utilities, so the current utilities array can simply be reused: we just have to make sure that the attribute corresponding to the number of utilities is correctly updated, as because of these new methods, the actual number of utilities of the hypercube may be smaller than the real size of its utilities array. For instance, if we start from a hypercube with 100 elements, then project out some of its variables so that there are only 50 elements left, the number of utility values of this hypercube will be updated to 50, but its utilities array will still have a size of 100, even if only its 50 first elements are actually relevant to the modified hypercube. A crucial difficulty in all these implementations is then that we need to make sure that we do not erase utilities values that will be needed later to complete the operation.

3.1 Slice

Slicing a hypercube basically consists in selecting some parts of the utilities array that will be kept, and discarding the other blocks from the array. As a result, the modification of the utilities array into the result of the slice can be done by just copying specific blocks of the array in order to move the kept blocks over the dropped blocks. As these blocks are moved in the order (if a block is situated before another one in the array, it will be moved before) and always towards the left, this algorithm ensures that the erased parts are no longer necessary for the rest of the operation.

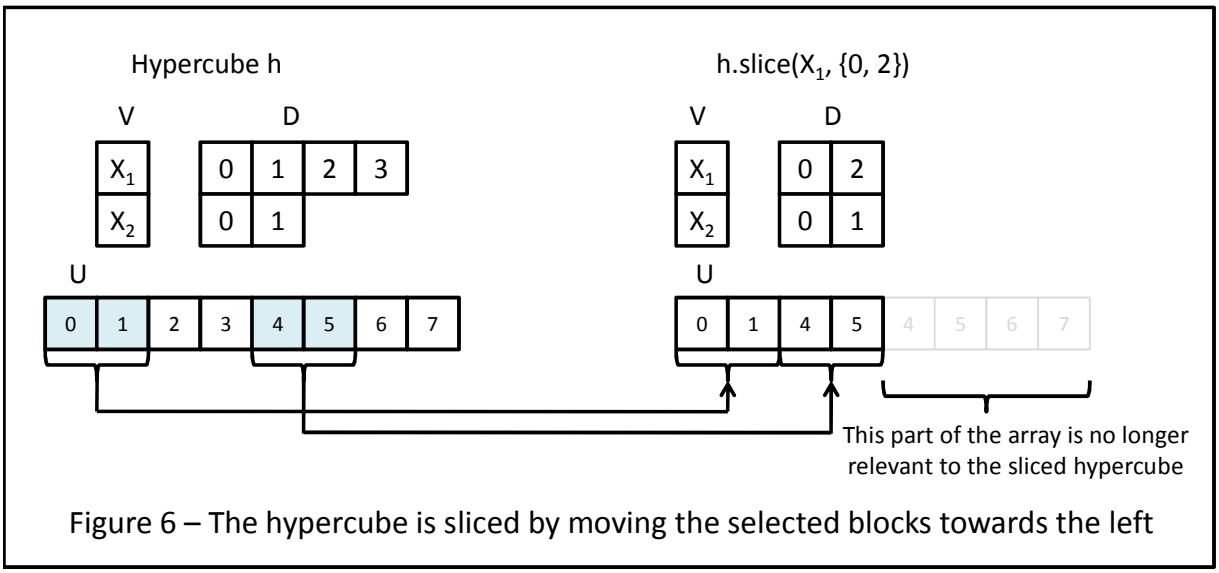


Figure 6 – The hypercube is sliced by moving the selected blocks towards the left

The main difficulty is to calculate the size and the number of the blocks to be copied. For this implementation, we look at the variables list from the end and search for the first variable whose domain is actually sliced, and the size of the blocks is then equal to the product of domains sizes of all variables covered before finding the first sliced variable (see Figure 6 for an example). But this computation is not always optimal, as in some cases (as displayed in Figure 7), two different blocks may very well be actually adjacent in the utilities array. In order to really find the optimal blocks, a more complex pre-computation would have to be made in order to represent the blocks sequence to be copied (see Figure 8 for an example of an irregular blocks sequence).

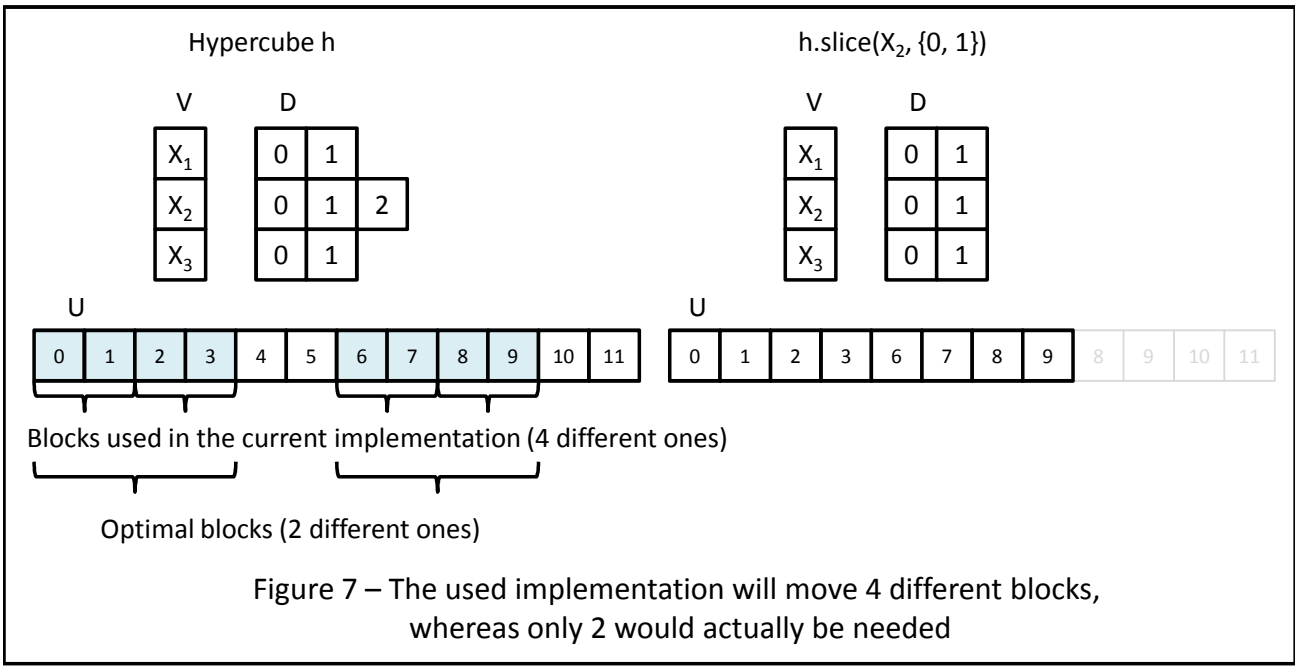


Figure 7 – The used implementation will move 4 different blocks, whereas only 2 would actually be needed

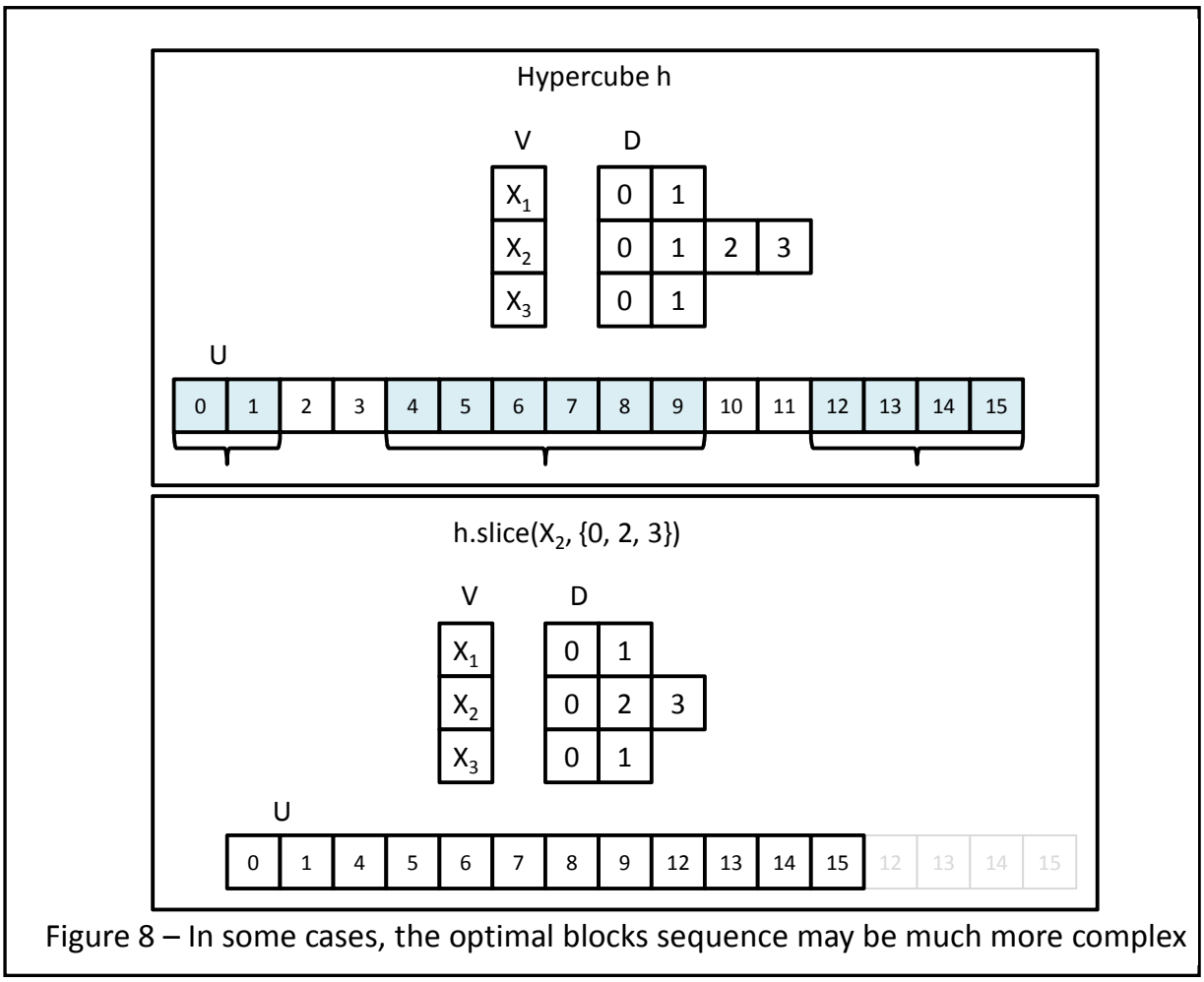


Figure 8 – In some cases, the optimal blocks sequence may be much more complex

3.2 Project

For the projection function, we simply cover linearly each possible original variables assignment. For each variables combination, the corresponding utility value in the original hypercube is taken, compared with the utility value corresponding to the same variables combination but without the projected out variables in the modified hypercube (which is always situated before the first value, as the utilities array is linearly modified), and possibly replace the latter with this new value (an example is presented in Figure 9).

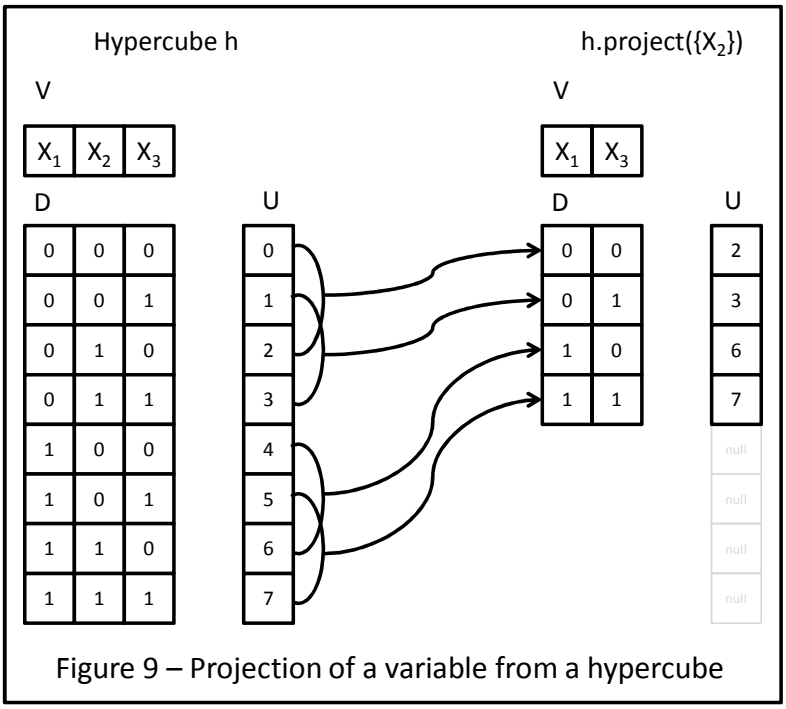


Figure 9 – Projection of a variable from a hypercube

We can thus use the following algorithm:

void ***applyProject***(String[] *projected_variables*)

01: **For** *i* from 0 to (**number of utilities** – 1)
02: *v* ← next variables assignments (corresponding to index *i* in the utilities array)
03: *v'* ← remove values corresponding to projected out variables from *v*
04: *i'* ← index corresponding to *v'* in the modified array
05: *u* ← **values**[*i*]
06: *u'* ← **values**[*i'*]
07: if (*u* > *u'* or values[*i'*] == **null**)
08: **values**[*i'*] ← *u*
09: **if** (*i* > *i'*)
10: **values**[*i*] ← **null**

3.3 ChangeVariablesOrder

Reordering a hypercube by only modifying the current utilities array is much more difficult, because the size of the array is not reduced, and therefore we have to be careful about not overwriting utility values when reordering them.

For a given variables assignment, a temporary variable is used to store its associated utility value. Then the variables values are reordered according to the new order, and we look in the array for the emplacement corresponding to this new assignment in the array. The temporary value is then put in this place of the array, while the previous value situated at this position is stored in the temporary variable. Then the current variables assignment is reordered again, and this process is repeated until we encounter an already reordered utility value, which means a cycle has been covered and that we can start reordering the next utility value in the array (see Figure 10 for an example).

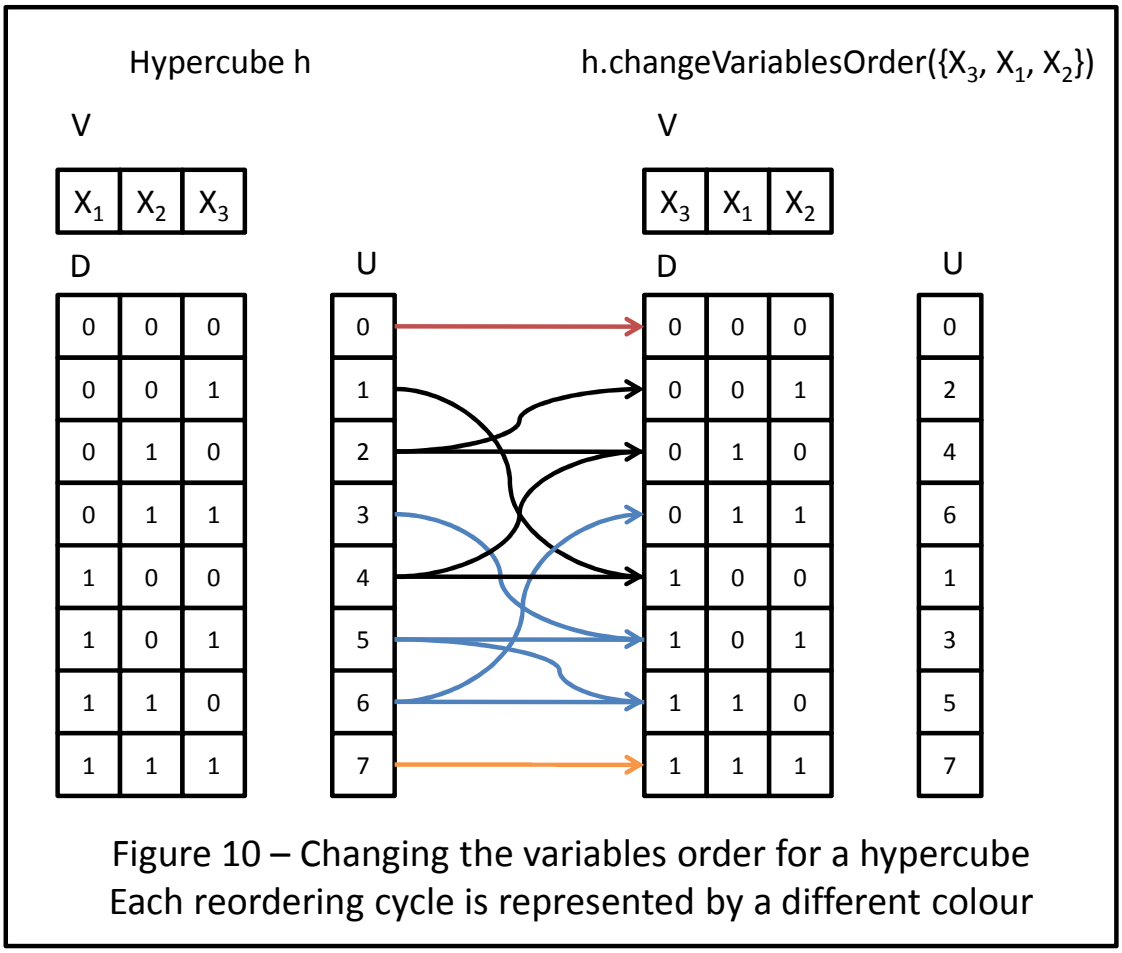


Figure 10 – Changing the variables order for a hypercube
Each reordering cycle is represented by a different colour

The difficulty is then to find the value which has yet to be reordered in the array, which means we have to use a data structure in order to keep track of the reordered utility values. To do so, several solutions are possible, such as a boolean array with as many elements as the utilities array (however, the size of the boolean array in the memory is much smaller than the original array), but as this array may potentially be very large, a version of this algorithm using instead a integer HashSet was also implemented: each time a value is reordered, its index is added to the HashSet, and to know whether a value has already been reordered, we just have to check whether its index is present in this HashSet.

As adding and searching for elements in the HashSet takes some time, it is important to keep its size as small as possible. An optimization of the code was then to remove indexes from the list when it is sure they will not be covered again in the utilities array (so their presence in the list is no longer necessary), which slightly improved the performances of this function, and gives the following algorithm:

```
void applyChangeVariablesOrder(String[] new_order)
01: already_reordered ← new empty HashSet
02: For i from 0 to (number of utilities – 1)
03:     if (i does not belong to already_reordered)
05:         u ← values[i]
06:         v ← variables assignment corresponding to index i in the utility array
07:         v' ← v reordered according to new_order
08:         i' ← index corresponding to v' in the modified array
09:         while (i != i')
10:             add i' to already_reordered
11:             swap values[i'] with u
12:             v ← variables assignment corresponding to index i' in the original utility array
13:             v' ← v reordered according to new_order
14:             i' ← index corresponding to v' in the modified array
15:         values[i] ← u
16:     else
17:         remove i from already_reordered
```

However, experimental results (see Figure 20 in section 3.6) showed that the boolean array implementation actually gives better result in term of both execution time and memory consumption, so it is this version of the algorithm that was finally kept.

3.4 Augment

The augment function is a new method which for the moment is only used as an auxiliary function for the join method. The augment method takes as argument a list of new variables and theirs domains, and adds these variables to the hypercube, whose utilities array is extended by simply repeating its current utilities values.

The major difference from the previous methods is that the number of utilities values is not reduced, but increased, so we first have to pre-compute the number of utilities values of the augmented hypercube (which is equal to the number of utilities of the original hypercube, multiplied by the product of the domains sizes of the new variables) then check that the current utilities array is large enough to contain all these values. If this not the case, it is not possible to augment the hypercube by just modifying its utilities array, so it is necessary to create a new hypercube.

Otherwise, the augmentation is quite simple and just consists in copying the utilities values block in the array as much as necessary (in this case, the new variables are added at the beginning of the hypercube ; if they were added at the end, we would have to repeat each utility value individually as it is shown on Figure 11).

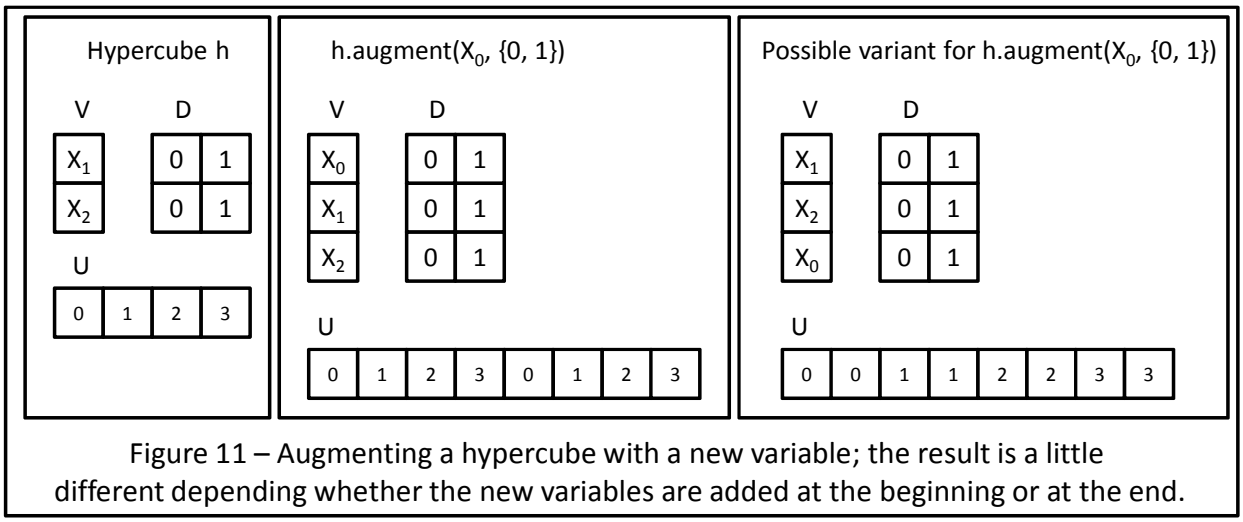


Figure 11 – Augmenting a hypercube with a new variable; the result is a little different depending whether the new variables are added at the beginning or at the end.

3.5 Join

With the previous functions, joining a hypercube with another one basically consists in a slice followed by an augmentation. The join can then be divided into at most three steps (which are illustrated on Figure 12):

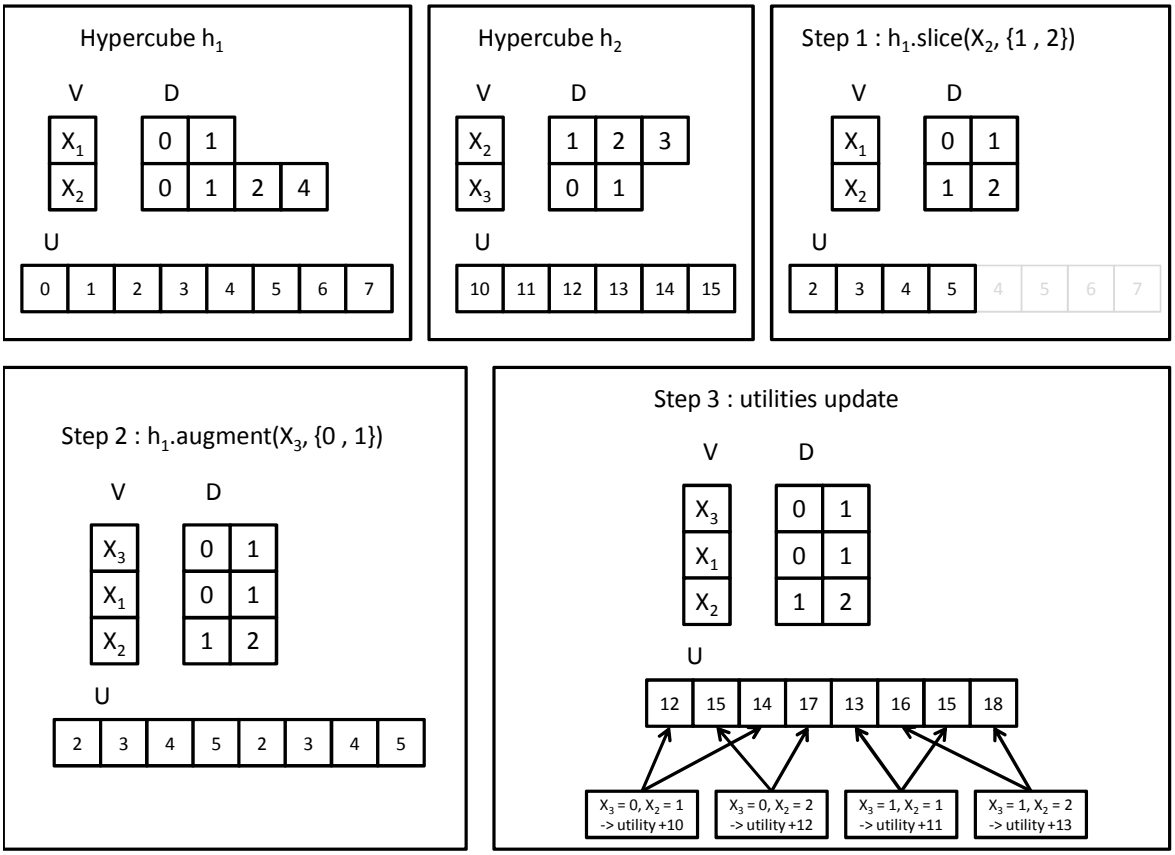


Figure 12 – Three steps are needed to join two hypercubes

- For each variable present in the first hypercube (and possibly in the second one), we compute the intersection between its domains in each hypercubes, and we slice the first hypercube according to these variables and theirs intersected domains. This step is needed only when the two hypercubes do not agree on the domain of at least one of their common variable: the hypercube must then be sliced so that only the values that are in the intersection are kept in the result.
- The first hypercube is augmented with the variables present only in the second hypercube. It is at this point that a new hypercube may be created only if the utilities array size of the first hypercube is not large enough to contain all the utilities values of the join result.

  This step is needed only when the two hypercubes do not agree on the list of variables: the variables which belong only to the input hypercube must then be added to the current hypercube.

- The utilities values have finally to be updated by adding the corresponding values of the second hypercube. For each variables assignments of the modified hypercube, the corresponding utility value in the second hypercube is retrieved and is added to the correct utility value in the resulting utilities array.

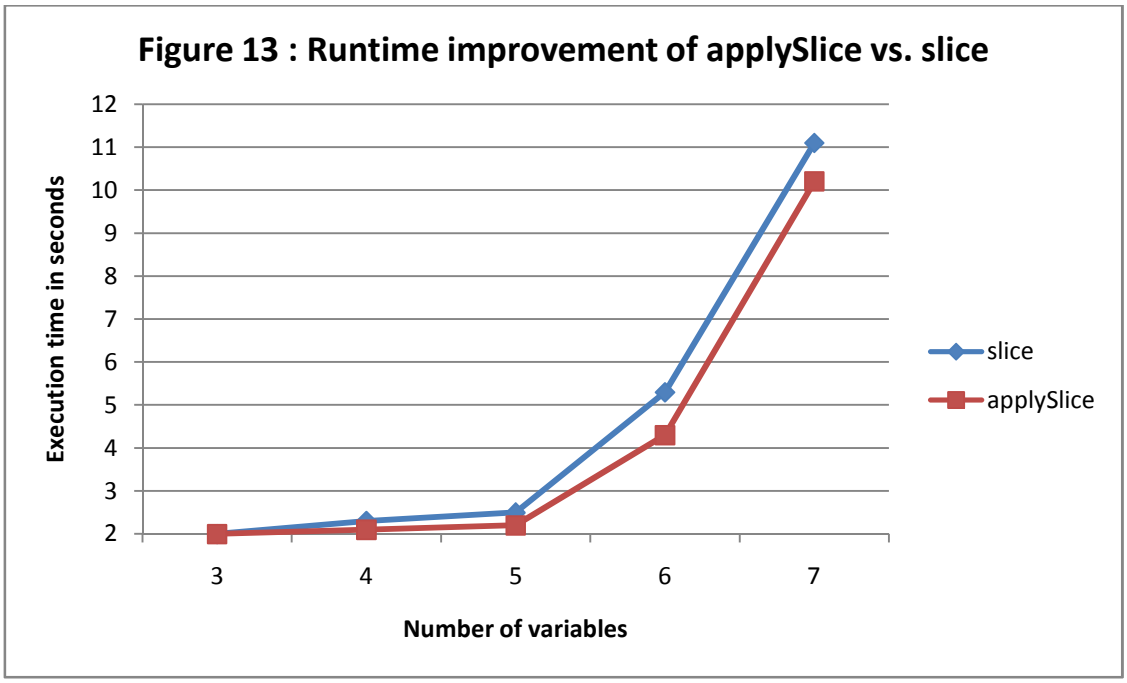
This function has been implemented in a way that works no matter what the variables orders of the two hypercubes are, so with this version of join, even if two hypercubes disagree on the variables order, they can nonetheless be joined together without needing a preliminary reordering. However, due to the use of the augment function, the final variables order of the resulting hypercube is fixed: the first variables will always be the variables present only in the second hypercube, followed by the variables present in the first hypercube. If we want the output to have a different variables order, a reordering with the changeVariablesOrder method is needed.

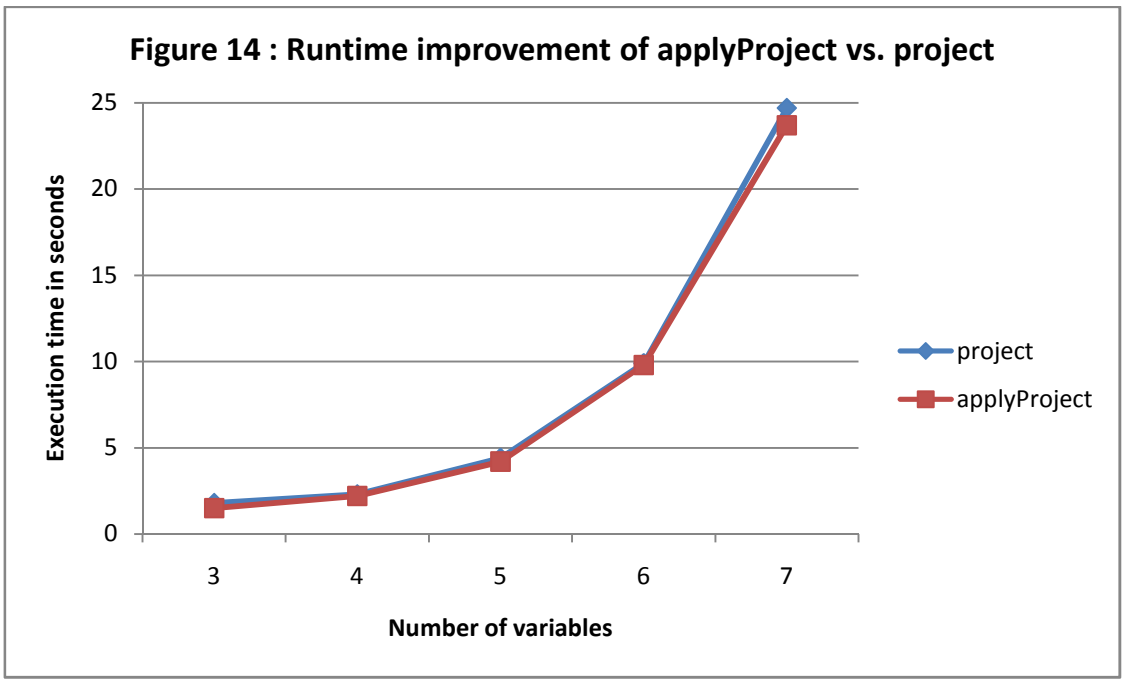
The choice of the hypercube that will actually be modified is an interesting question. In order to increase the chance that the augment function will not require the creation of a brand new array, it is indeed smarter to first check which one of the two hypercubes has the largest utilities array, and then perform the necessary modifications on this hypercube.

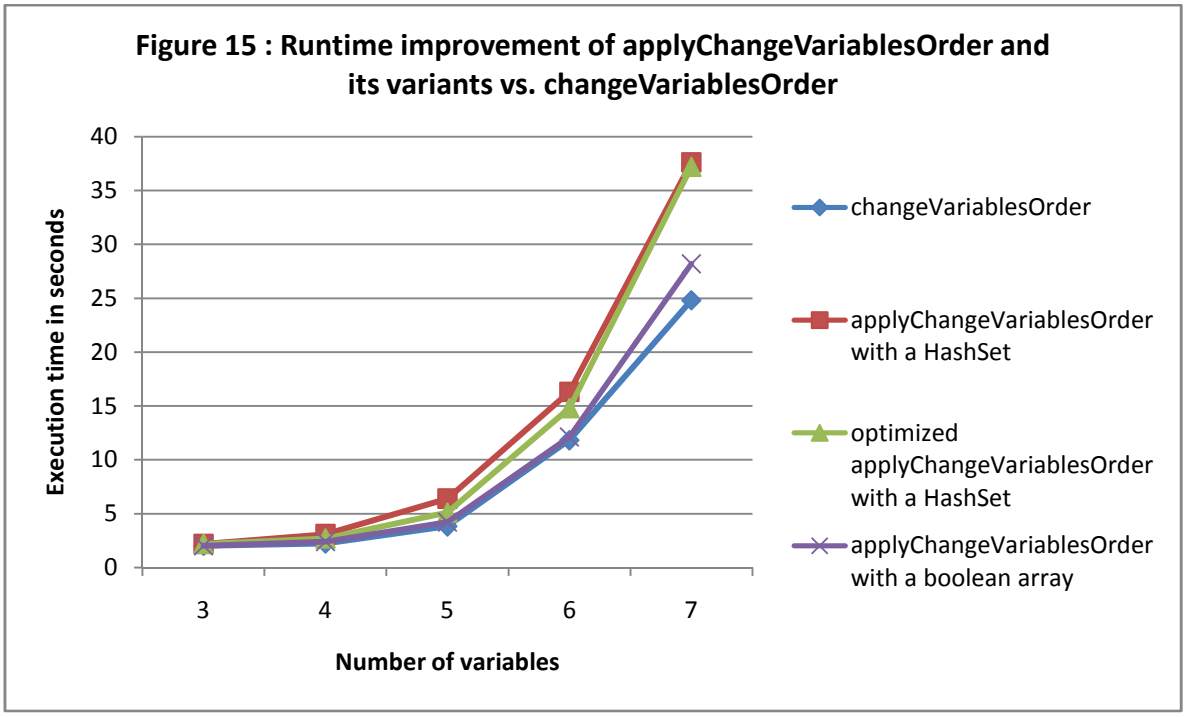
Thus, when joining two random hypercubes, experimental results show that if the hypercube which is modified is arbitrary (i.e. we simply take the hypercube on which the applyJoin method is called), the creation of a new utilities array is necessary in 75% of cases, whereas if the modified hypercube is the one with the largest utilities array, the proportion of cases where a new utilities array has to be created drops to 58%.

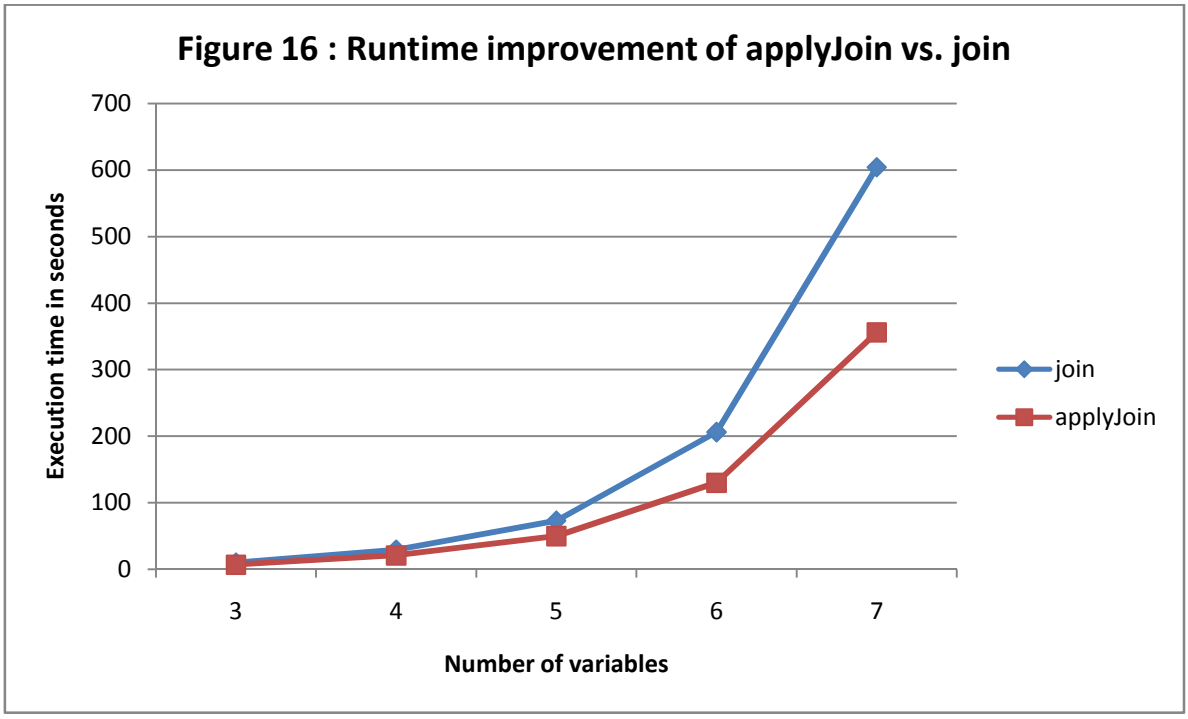
3.6 Experimental results

In order to test the performances of these new operations implementations, 10.000 randomly generated hypercubes have again be used in order to compare their execution time. The results for the slice, project, changeVariablesOrder and join are respectively shown on Figures 13, 14, 15, 16.









As expected, for changeVariablesOrder, the new implementation takes more time than the classical one, as the algorithm is much more complicated to set up. The performances of the different variants of applyChangeVariablesOrder are displayed: when a boolean array is used to keep track of the already reordered utilities, the execution time is much smaller than when a HashSet is used for this purpose, even with the optimization consisting in removing the no longer used indexes from the HashSet.

For slice and project methods, the time gain is very small (about 5-10 %), but for join, the gain is quite high, which is quite surprising insofar as because of the usual trade-off between memory and time, one would expect that improving memory consumption would result in larger runtimes.

A possible explanation is that the most expensive part of the former implementations is to cover individually each element of the utilities arrays of the input hypercubes by calling the getUtilityValue method which needs to access the hypercube's steps HashMap and then perform some computations on its values. In applySlice and applyProject however, the getUtilityValue is not needed as in the case of applySlice, the utilities are directly copied and moved by blocks, and in applyProject, the original array is just linearly covered. As for applyJoin, only the utilities array of the second hypercube has to be covered, as the utilities of the first hypercube are simply moved by blocks when this hypercube is sliced during the first part of the algorithm, contrary to the join method where both input hypercubes have to be covered with getUtilityValue in order to compute the utilities of the new hypercube.

In order to test the performance in term of memory consumption, for each method the following algorithm has been used:

```
01: For i from 1 to 10.000
02:     input[i] ← generate a random hypercube with n variables
03:     output[i] ← apply the method to test on input[i]
```

When this scenario is run for a given number of variables n (n varying from 3 to 10), either the 10.000 instances are computed successfully, or the program runs out of memory. In the latter case, the current value of i is recorded so that we know how many instances were ran before the memory shortage occurred. For each method, a "score" value is computed this way:

score = 3*(number of instances for 3 variables that were run before the program ran out of memory)
+ 4*(number of instances for 4 variables that were run before the program ran out of memory)
+ ... + 10*(number of instances for 10 variables that were run before the program ran out of memory)

Thus, for any given method, we obtain a value ranging from 0 to 52, which accounts how well this method performed in term of memory consumption. The score values for slice, project and join are respectively displayed in Figures 17, 18 and 19 for different amounts of memory allocated to the JVM, and clearly show an improvement of the "apply" versions over the previous methods, especially for applyProject.

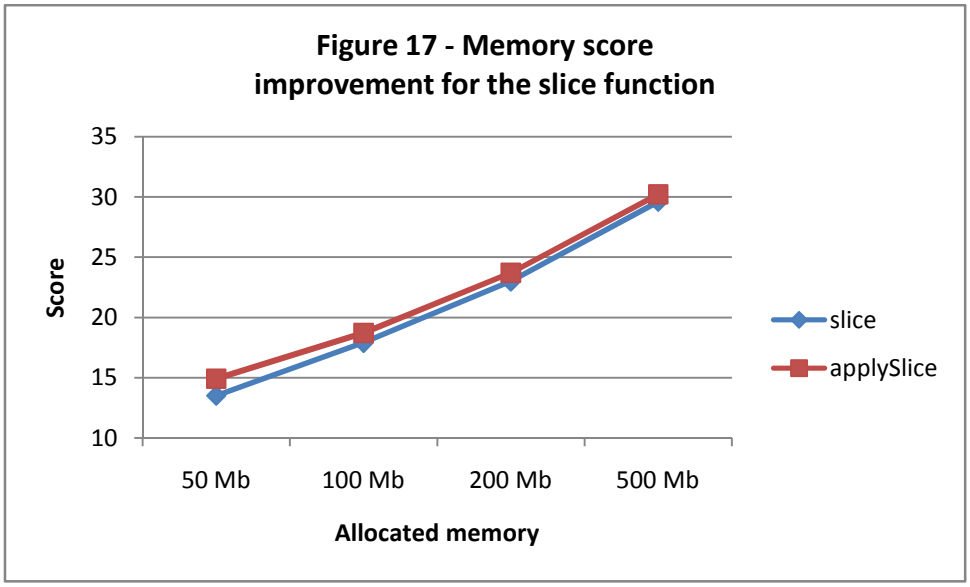


Figure 17 - Memory score improvement for the slice function
35
30
25
20
15
10
Score
50 Mb
100 Mb
200 Mb
500 Mb
Allocated memory
slice
applySlice

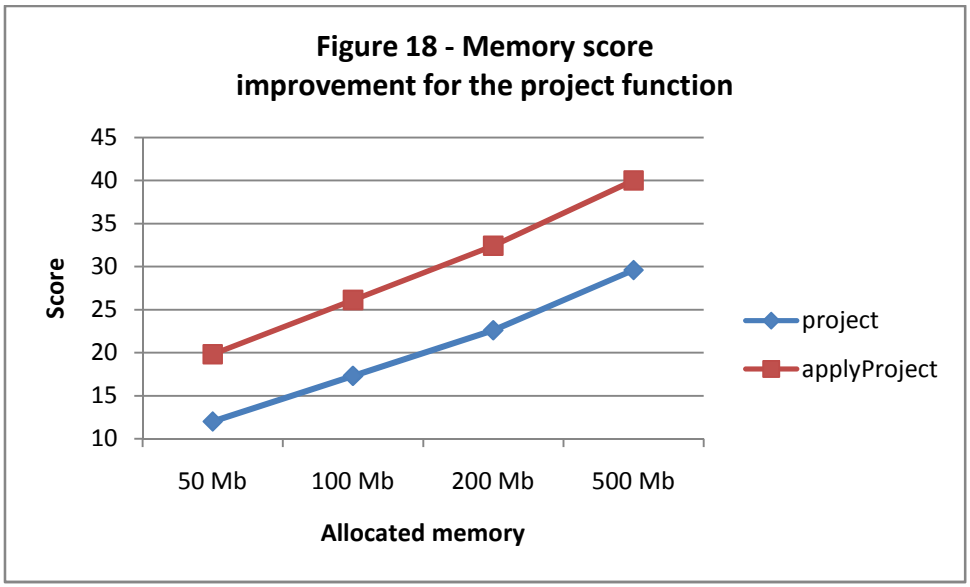


Figure 18 - Memory score improvement for the project function
45
40
35
30
25
20
15
10
Score
50 Mb
100 Mb
200 Mb
500 Mb
Allocated memory
project
applyProject

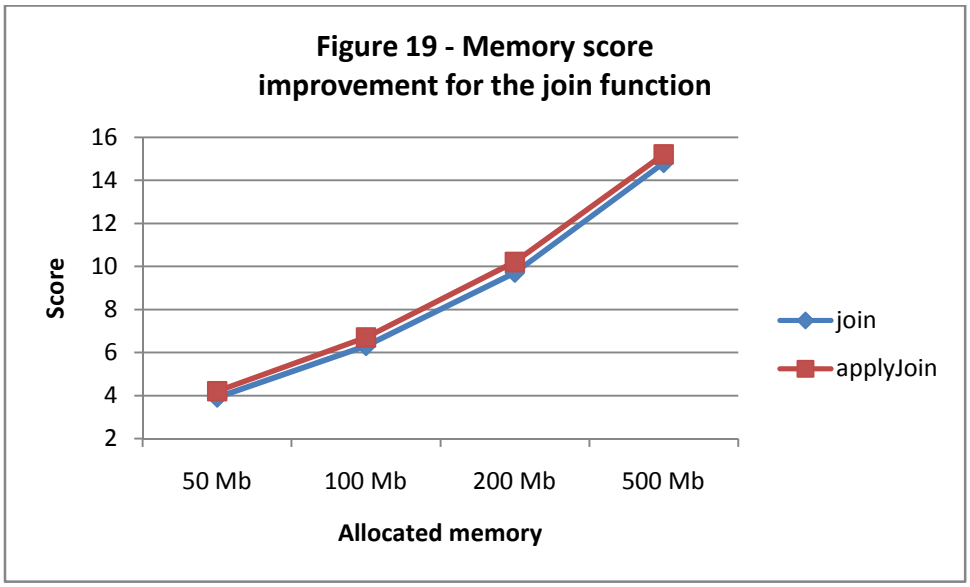
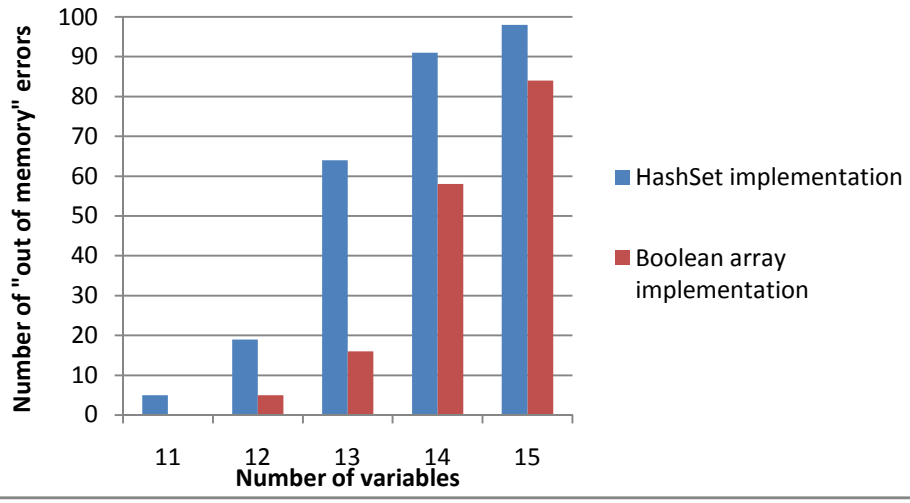


Figure 19 - Memory score improvement for the join function
16
14
12
10
8
6
4
2
Score
50 Mb
100 Mb
200 Mb
500 Mb
Allocated memory
join
applyJoin

For the join function, the difference is quite small, which makes sense as it has already been previously established that in more than 50% of cases, the hypercube resulting from a join operation has a higher number of utilities than the original hypercubes and hence applyJoin nonetheless requires the creation of a new utilities array. However, if a join is performed on a hypercube whose number of utilities has already decreased (for example due to a project or slice operation), the probability that the join will not require the creation of a new array increases, so in the context of DPOP, where join are often applied after a projection, the performance gain of applyJoin might be a lot higher.

In order to check which implementation of applyChangeVariablesOrder is better in term of memory consumption, the following test was used: for a given number of variables (varying between 11 and 15), 100 random hypercubes are generated and one of the versions of applyChangeVariablesOrder is applied on them; the number of reordering which failed due to a lack of memory is then counted. The results are shown on Figure 20 and clearly show that unlike what was originally expected, the implementation of applyChangeVariablesOrder which uses a boolean array tends to consume less memory than the other one.

Figure 20 - Performances of the two versions of applychangeVariablesOrder in term of memory consumption



Conclusion

The experimental results have shown that the methods variants implemented during the course of this project definitely increase the performances of the various functions, both for hypercubes and utility diagrams.

These improvements could be pushed further by writing more specialized versions of some methods (for example, a better joinOnePass for utility diagrams whose edges are only labeled by single values) or by combining several different improvements into a better function (for instance, implement a version of joinProject which does not require the creation of a new utilities array, as it has been done for the other operations in part 3).

As the modifications presented in part 3 of this report have proven their efficiency for hypercubes, future work could be aiming at doing the same thing for utility diagrams. A first step could thus be to only reuse the utilities array (and still create a new diagram as it is done with the current implementation). The next step would then be to also reuse the diagram and modify it into the resulting diagram. Even if doing that might be too complex for operations such as join, it may for example be interesting for the slice function, which would basically just consist in removing some edges (the ones whose labels do not belong to the sliced domains) from the diagram.