Asynchronous Forward Bounding (AFB): Implementation and Performance Experiments

Álexandra Olteanu
`alexandramihaela.olteanu@epfl.ch`

Supervisors:
Thomas Léauté (`thomas.leaute@epfl.ch`)
Prof. Boi Faltings (`boi.faltings@epfl.ch`)

EPFL Artificial Intelligence Laboratory (LIA)
`http://liawww.epfl.ch/`

August 18, 2011

Contents

1 Introduction

1.1 DCOP

In Artificial Intelligence, the field of Distributed Constraint Optimization (DCOP) addresses problems in which a group of agents must, in a distributed manner, choose values for a set of variables such that the cost of a set of constraints over the variables is either minimized or maximized. In this framework, the constraints are known and enforced by distinct participants (agents) and the variables can be assigned values from predefined domains. So the variables and constraints are distributed among multiple agents, which communicate with each other.

Advantages of using distributed versus centralized solvers, as described in [1], include the cost of formalizing the problem — in the centralized environment, each participant has to formulate its constraints on all imaginable options beforehand —, privacy — in the centralized scenario, the solver will see all the constraints —, load balancing and redundancy leading to more reliable systems, while asynchronous parallel execution can make the system more efficient. Common real-life applications include meeting scheduling in large organizations, where privacy needs make centralized constraint optimization difficult; allocating sensor nodes to targets in sensor networks, where the limited communication and computation power of individual sensor nodes makes centralized constraint optimization difficult; coordinating teams of unmanned air vehicles, where the need for rapid local responses makes centralized constraint optimization difficult.

A simple branch and bound algorithm for solving DCOPs is SyncBB [6]. This is a distributed complete algorithm, but which does not make use of parallel computation. Asynchronous Forward Bounding (AFB) as presented in [3] can be seen as an improvement on SyncBB, which tries to make better use of parallel computation.

1.2 Project Overview

The goal of this project was to implement, test, and evaluate the AFB algorithm. The algorithm was implemented in Java, as part of the open-source FRODO platform for DCOP [7]. Because there was no previous implementation of AFB in FRODO, a significant part of the project consisted in designing and implementing AFB.

We have evaluated our implementation of AFB against other DCOP algorithms already implemented in FRODO on two classes of benchmark problems:

- the distributed kidney exchange problem [5]. This is a problem class that has been studied in the centralized setting, but which is new to the DCOP community.

- Randomly generated Max-DisCSPs, in an attempt to reproduce the results in [4].

2 AFB Implementation

2.1 DCOP Definition

Formally, a DCOP is represented by a tuple $< A,X,D,R >$, where:

- A is a finite set of agents $\{A_1, A_2, \dots ,A_n\}$
- X is a finite set of variables $\{X_1, X_2, \dots ,X_m\}$. An agent can be responsible for one or more variables.
- D is a set of domains $\{D_1, D_2, \dots ,D_m\}$, where D_i contains a finite set of values that can be assigned to variable X_i
- R is a set of relations (constraints). Each constraint, C, defines a non-negative cost for every possible assignment of values to a subset of variables. If a constraint involves only two variables, it is called a *binary constraint.*

An assignment is a pair including a variable and a value from the variable domain. A partial assignment, PA, is a set of assignments in which each variable appears at most once. The *cost* of a partial assignment is the sum of all applicable constraints among the variables in the PA, with respect to their assigned values. A full assignment is a PA that contains all variables in X. The goal in DCOP is to find a *full assignment with minimal cost.*

2.2 AFB Description

The AFB paper we used as main reference [3] presents a one-variable per agent version of the AFB algorithm, assuming also that all constraints are at most binary.

A total order among agents is assumed (A_0 is assumed to be the first agent in the order, and A_{n-1} is the last). This order induces agent priorities, with A_0 having the highest priority and A_{n-1} the lowest.

The main concept in the AFB algorithm is a partial assignment (PA). This holds assignments to variables up to a current index. One single most recent (up-to-date) version of this partial assignment can exist at each point in time, which we call the current partial assignment (CPA). The CPA is passed via messages between agents, and represents the partial assignment that agents attempt to extend into a complete and optimal solution by adding assignments of their own variables to it. The CPA also holds the accumulated cost of constraints between all assignments it contains, as well as a unique timestamp, which will help to identify which is the most up-to-date PA. Due to the asynchronous nature of the algorithm, multiple CPAs may be present at any moment in time, but the most up-to-date CPA will be the one with the highest timestamp.

Only one agent performs an assignment on the CPA at a time. Copies of the CPA are sent forward and are asynchronously processed by multiple agents. A new assignment is sent forward in messages called *FB_CPA*, to all agents whose assignments are not yet on the CPA. These are lower priority agents and their variables have unassigned values. This kind of message requests a lower priority agent to compute an upper bound on the cost increment caused by adding an assignment to its variable. The estimated cost is sent back to the agent who sent the *FB_CPA* in a *FB_ESTIMATE* message. This agent uses all the received bounds to prune sub-spaces of the search-space which do not contain a full assignment with a cost lower than the best full assignment found so far.

The computation of the bound is described in [3] as follows. Denote by *cost((i,v), (j,u))* the cost of assigning value v to agent A_i and value u to agent A_j, i.e. the summed cost of all applicable constraints. For each agent A_i and each value $v \in D_i$, the minimal cost of the assignment *(i,v)* incurred by an agent A_j, is denoted by: $h_j(v) = min_{u \in D_j} cost((i,v),(j,u))$. The total minimal cost of assigning value v to variable i is then:

$$h(v) = \sum_{j>i} h_j(v) \ .$$

A bound estimate, denoted *f(v)*, for a given PA and $v \in D_i$, can be computed as:

$$f(v) = h(v) + \sum cost\ v\ has\ with\ assignments\ in\ the\ PA.$$

The bound A_i returns in an *FB_ESTIMATE* message is the lowest possible value for f(v), across all $v \in D_i$:

$$bound\ estimate\ for\ agent\ A_i = min_{v \in D_i} f(v)$$

This forward-bounding mechanism is the key to AFB's asynchronous nature. While the CPA is owned by one agent — the one currently making an assignment — many copies are sent forward and a collection of agents compute concurrently the lower bounds for that same CPA. Thus, the unassigned agents are constantly working, either when they receive the CPA, or when they need to compute bounds for some other partial assignment.

The estimations from all lower priority agents can be accumulated and summed up by the agent that initiated the forward bounding process to compute a lower bound on the cost of a complete assignment extended from the CPA. If the received estimations indicate that the CPA cost exceeds the current known upper bound, the agent will generate a new CPA (i.e. assign a new value to its variable) with a higher timestamp and continue the search with this new CPA. If, in turn, it cannot generate anymore assignments or itself, it will initiate a backtrack, causing the previous agent in the ordering to assign a new value to its variable.

A more detailed definition of a CPA cost and the bounds, as described in [3], is as follows:

- **Past Cost**: the cost of all constraints for higher priority variables already assigned in the CPA
- **Local Cost** for a certain value assignment to the current variable: the cost that assigning the particular value to the variable would add to the CPA (this comes from constraints with lower priority variables)
- **Future Cost** for a certain value assignment to the current variable: the lowest possible cost variables with lower priority would add, given that the particular value is assigned to the variable.

The lower bound on the cost of a partial assignment is, in fact, then the sum of these three costs.

Notice that it is possible that the assigning agent already sent its CPA forward by the time the estimations for that assignment are received. Thus, whenever receiving any type of message, one should check that this is still up-to-date. This is done via the timestamp mechanism. In short, each agent keeps a local assignment counter which it increases whenever it performs an assignment. Whenever it sends a message containing a PA, the agent copies its current counter onto the message. Each message, therefore each PA, has associated a vector containing the counters of all the agents it passed through. The *i-th* element of the vector corresponds to the assignment counter of agent *i*. To determine the most up-to-date timestamp, a lexicographical comparison of the vectors is performed.

2.3 AFB implementation

2.3.1 Overview

Our implementation makes use of the FRODO existing modules *VariableElection* and *LinearOrdering* to construct the variable ordering. This gives a variable ordering or priority. The first variable in the ordering has the highest priority, while the last variable in the ordering has the lowest priority.

The design is inspired by the SyncBB design, but several different approaches of data encapsulation are made to better model the interactions in AFB.

The code can be found in a new package, *ch.epfl.lia.frodo.algorithms.afb*.

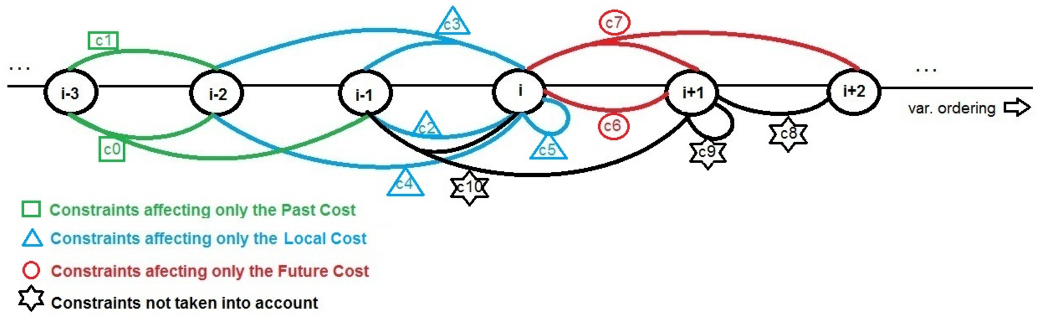
...
c1
c0
i-3
i-2
i-1
i
i+1
i+2
c3
c2
c4
c5
c10
c7
c6
c9
c8
...
var. ordering
Constraints affecting only the Past Cost
Constraints affecting only the Local Cost
Constraints afecting only the Future Cost
Constraints not taken into account

Figure 1: Constraints selection for cost computing at variable i.

Three major extensions to the algorithm originally described in [3] are included in the implementation:

- an agent can hold multiple variables, which is a common assumption in practical implementations of DCOP algorithms;
- the constraint graph can have multiple components;
- n-ary constraints are supported.

The first two modifications are easily incorporated by the use of *VarInfo* and *CompInfo* classes, which encapsulate component and variables specific information. When an agent receives a message, it needs to first identify the correct *CompInfo*, i.e. component it is regarding and also *VarInfo*, i.e. variable, that will process it. This is similar to the SyncBB implementation, though modification had to be made to allow message sending between all possible agents. Unlike SyncBB, an AFB agent (which we will now refer to as a variable) needs to send messages to all lower priority variables, requesting upper bound estimates. These agents need to reply with these estimates to higher priority variables. So, in general, each variable needs to be able to communicate with all other variables.

The literature only assumes binary constraints, so we had to define a way to extend these to n-ary. There are two issues to look at:

- which constraints to take into account when computing the **Local Cost** for a variable assignment, with respect to a given PA;
- which constraints to take into account for the computation of the h estimate, given a certain variable assignment. This does not depend on the current PA (no other variable assignments are taken into account) and is an estimation of the best **Future Cost** that can be expected.

Figure 1 tries to clarify our approach. The **Past Cost** is not affected by generalizing the constraints type to n-ary. All constraints regarding any number of variables of higher priority than the current variable, i, account to the **Past Cost** (ternary constraint c0 and binary constraint c1 in the graph). All constraints in which the current variable, i, is the last variable in the constraint (with the respect to the variable ordering) account for the **Local Cost** (in the graph: unary constraint c5, binary constraints c2 and c4, as well as ternary constraint c3). All constraints that involve the current variable, i, and variables with lower priority than i account for the **Future Cost** (in the graph: binary constraint c6 and ternary constraint c7). Notice that each such constraint must include i, so constraints like c8 and c9 are not included.

Constraints like c10, which include variable i, but not as first or last variable in the constraint are excluded also, because counting them would cause a double effect. For example, c10 should already be counted for the **Future Cost** of variable i-1. If we count it in the **Future Cost** of variable i also, variable i-1 would receive a future bound estimate which counted c10 twice and this would not be correct.

The implemented algorithm was tested for correctness with JUnit tests, similar to the SyncBB algorithm. These can be found in the *ch.epfl.lia.frodo.algorithms.afb.test* package. We used randomly generated CSPs and several categories of tests: with or without using the XCSP format, with or without restricting the sign of the utilities, with integer or real values for the variables and utilities, with or without counting NCCCs and with different kinds of pipes for communication (TCP or QueueIO). We used at most 1000 tests per category, totaling 12000 tests with 100% success rate.[1]

2.3.2 AFB Pseudocode

A major difficulty in the implementation was coming up with a complete, working version of the pseudocode, which the AFB paper [3] was lacking. Several details were missing and we will point them out in what follows.

This section presents the full AFB pseudocode. For simplicity, we will follow the general assumptions in the literature: the constraint graph is considered to be connected (only has one component), each agent is responsible for only one variable and all constraints are binary. We have already explained in the previous paragraph all the details of how we extended these assumptions in our FRODO implementation.

Recall that D_i denotes the domain for variable i. The following are member attributes of each agent:

- *CPA* — the current partial assignment;
- n — the number of agents;

[1]All JUnit tests pass successfully on Windows and Mac OS X, but for some reason some of them timeout on Linux; this has to be further investigated.

- *timestamp* — timestamp array on each agent (holds last known assignment counters for all variables);
- *assignmentCounter* — the current assignment counter for the agent's variable;
- *B* — the best cost known for a solution so far;
- *bestCPA* — the assignments associated with the best known solution so far;
- *estimates* — all received estimates from lower priority agents;
- *h* — h computed for all values in the variable's domain;
- *domainIndex* — the last used index in the domain of the variable;

The algorithm is run on each of the agents. Firstly, the *init* procedure is called, after which messages are responded to until a *TERMINATE* message is received.

Procedure init

```
   // executing on agent i
1  $B \leftarrow \infty$;
2  for $j \leftarrow 0$ to $n-1$ do
3      $assignmentCounter[j] \leftarrow 0$;
4  for $j \leftarrow 0$ to $n-i-2$ do
5      $estimates[j] \leftarrow NULL$;
6  for $v \in Di$ do
7      $compute\_h$(i,$v$);
8  if $i = 0$ then
9      $CPA \leftarrow$ generate empty CPA;
10     $assign\_CPA()$ ;
11 End Procedure.
```

Procedure compute-h(i,v)

```
   // executing on agent i
1  for $v \in Di$ do
2      $h[v] \leftarrow \sum_{j=i+1}^{n-1}$ cost of constraints($v, u$);
3  End Procedure.
```

Procedure assign-CPA

```
// executing on agent i
for $j \leftarrow 0$ to $n - i - 2$ do
    $estimates[j] \leftarrow NULL$;
if $CPA.assignments[i] = NULL$ then
    // assignment for variable i has not yet started
    $assignmentCounter[i] \leftarrow 0$ ;
else
    // if CPA has an assignment for variable i, remove it
    $CPA.assignemnts[i] \leftarrow NULL$;
    update $CPA.cost$ // removing cost added by constraints
        triggered by the removed assignment
$found \leftarrow false$ ;
while $!found$ $AND$ $domainIndex$ <$Di.length - 1$ do
    $domainIndex \leftarrow domainIndex + 1$;
    $v \leftarrow Di[domainIndex]$;
    $CPA.assignemnts[i] \leftarrow v$;
    if $CPA.cost + f(CPA, i, v)$<$B$ then
        $found \leftarrow true$ ;
if $!found$ then
    $backtrack()$ ;
else
    $assignmentCounter[i] \leftarrow assignmentCounter[i] + 1$ ;
    $timestamp[i] \leftarrow assignmentCounter[i]$ ;
    if $i = n - 1$ then // the last variable in the ordering

        $B \leftarrow CPA.cost$ ;
        $bestCPA \leftarrow CPA.assignments$;
        $broadcast(UB\_MESSAGE, bestCPA, B)$ ;
        if $B = 0$ then // already reached optimal cost

            $terminate()$ ;
        $assign\_CPA()$ ;
    else
        $send(CPA\_MSG$, copy of $CPA, i)$ to variable $i + 1$ ;
        for $j \leftarrow i + 1$ to $n - 1$ do
            $send(FB\_CPA$, copy of $CPA, i)$ to variable $j$ ;
End Procedure.
```

Function f(PA,i,v)

```
  // executing on agent i
1 return $h[v]+$ Local Cost of adding assignment $i \leftarrow v$ to $PA$;
2 End Function.
```

In the *init* procedure, each agent initializes B to Infinity (no known upper bound yet), initializes its local members and computes the h values for its variable. Only the first agent generates an empty CPA and then begins the search process by calling *assign_CPA* (lines 8 to 10). This is a method that is called whenever an assignment needs to be found for a variable. An assignment is chosen as the first unused value in the variable's domain such that the current CPA cost summed with the added cost of this assignment is less than B (lines 8 to 14). If no such value is found, a *backtrack* call is initiated (line 16). Otherwise, the *assignmentCounter* of the variable and the agent *timestamp* are updated (lines 18 to 19). Also, if this is not the last variable in the ordering, a *CPA_MSG* is sent to the next agent and *FB_CPA* messages are sent to all lower priority agents (lines 30 to 32). For the last variable in the ordering, B and *bestCPA* are updated, a new *UB_MESSAGE* is broadcasted to all agents and procedure *assign_CPA* is called recursively to try to possibly find a lower cost assignment for this last variable (lines 22 to 28).

The *backtrack* procedure removes the assignment and estimates and resets the *timestamp* for the current variable in the PA. It then passes this information on, in a *CPA_MSG*, to the previous agent, for a variable reassignment (line 7). If there is no previous agent, i.e. the current agent is the first agent in the ordering, it means that the search space has been exhausted and thus the *terminate* procedure is called (line 9). This simply checks if a solution message has already been sent and broadcasts one if not.

Upon receiving a *SOLUTION_MESSAGE*, an agent checks to see if such a message has already been recorded (which can happen more than once in the asynchronous scenario) (line 1), records one if needed and terminates all agents (line 3). On the other hand, receiving a *UB_MESSAGE* only triggers an update to the local fields B and *bestCPA*, assuming, of course, that the message if not out of date.

An important event in the algorithm is receiving a *CPA_MSG*. When this happens, an agent first checks if it is relevant, i.e. not out of date, using the timestamp mechanism. If it is out of date, then the message is discarded (lines 1 to 2). Otherwise, the agent saves the received *PA* in its local *CPA* variable, as this represents the most up-to-date CPA. It also resets its *domainIndex*, if the message comes from a higher priority agent (lines 4 to 6). Then, it checks whether the received *PA* (without an assignment to its own variable) does not exceed the allowed cost B. If it does not exceed

Procedure backtrack(PA,i)

```
   // executing on agent i
1  for $j \leftarrow 0$ to $n - i - 2$ do
2      $estimates[j] \leftarrow NULL$;
3  $PA.assignments[i] \leftarrow NULL$;
4  update $PA.cost$;
5  $timestamp[i] \leftarrow 0$ ;
6  if $i > 0$ then
7      $send(CPA\_MSG, PA, i)$ to variable $i - 1$ ;
8  else
       // the highest priority agent
9      $terminate()$ ;
10 End Procedure.
```

Procedure terminate

```
   // executing on agent i
1  if solution message was not already sent then
2      $broadcast(SOLUTION\_MESSAGE, bestCPA, B)$;
3  End Procedure.
```

Procedure receive-SOLUTION-MESSAGE(msg, assignments, cost)

```
   // executing on agent i
1  if A solution has not already been recorded then
2      Record assignments and optimal cost cost;
3      $broadcast(TERMINATE)$ ;
4  End Procedure.
```

Procedure receive-UB-MESSAGE(msg, newCPA, newB)

```
   // executing on agent i
1  if $newB < B$ then // eliminate out-of-date UB messages
2
3      $bestCPA \leftarrow newCPA$;
4      $B \leftarrow newB$;
5  End Procedure.
```

this bound, it tries to assign a value to its variable (or replace its existing assignment in case it has one already) by calling *assign_CPA*. However, if the bound is exceeded, a *backtrack* is initiated since the cost is already too high (lines 7 to 11).

Procedure receive-CPA-MSG(msg, PA, j)

```
   // executing on agent i
1  if $compare(msg.timestamp, timestamp, i)$ = 'bigger' then
2      $timestamp \leftarrow msg.timestamp$ ;
3      $CPA \leftarrow PA$;
4      if $j < i$ then // sender was a higher priority variable
5
           // reset domain index
6          $domainIndex[i] \leftarrow -1$ ;
7      if cost up to variable i-1 $\geq B$ then // no improvement can be
          found
8
9          $backtrack(CPA, i)$;
10     else
11         $assign\_CPA()$ ;
12 End Procedure.
```

Function compare(timestamp_1, timestamp_2, upToIndex)

```
   // executing on agent i
1  for $i \leftarrow 0$ to $upToIndex$ do
2      if $timestamp_1[i] < timestamp_2[i]$ then
3          return 'smaller';
4      if $timestamp_1[i] > timestamp_2[i]$ then
5          return 'bigger';
6  return 'equal';
7  End Function.
```

An agent receiving a forward bounding request (*FB_CPA* message) from an agent *j* also uses the timestamp mechanism to discard irrelevant messages. If the message is relevant, the agent computes its estimate (lower bound) of the cost incurred by the lowest cost assignment to its variable (line 3). This computation was described in detail in the previous section and is the sum of the **Local Cost** and **Future Cost** for that variable. This estimation is sent back to the originating agent j in an *FB_ESTIMATE* message.

```
Procedure receive-FB-CPA(msg, PA, j)
  // executing on agent i
1 if compare(msg.timestamp, timestamp, i - 1) = 'bigger' then
2     $timestamp \leftarrow msg.timestamp$;
3     $estimate \leftarrow min_{v \in Di} f(PA, i, v)$ ;
4     $send(FB\_ESTIMATE, PA, i, estimate)$ to variable $j$;
5 End Procedure.
```

When receiving an *FB_ESTIMATE* from a lower priority agent, an agent ignores it if it is an estimate to an already abandoned partial assignment (also identified by using the timestamp mechanism). Otherwise, it saves this estimate (line 2) and checks if it causes the current partial assignment to exceed the bound *B*. If this is the case, the agent calls *assign_CPA* in order to try to re-assign its variable (lines 3 to 8).

```
Procedure receive-FB-ESTIMATE(msg, PA, j, estimate)
  // executing on agent i
1 if compare(msg.timestamp, timestamp, i) = 'bigger' then
2     $estimates[j - i - 1] \leftarrow estimate$ ;
3     $cost \leftarrow PA.cost$ ;
4     for $k \leftarrow 0$ to $n - i - 2$ do
5         $cost \leftarrow cost + estimates[k]$;
6         if $cost \geq B$ then
7             $assign\_CPA()$;
8             return ;
9 End Procedure.
```

Notice that, whenever passing a *PA* from one agent to another, a clone (copy) of the object is made to avoid overrides.

An important detail to notice was that the *domainIndex* for each variable i, holding the index of the last assigned value from D_i, needs to be updated also. More specifically, in the receive *CPA_MSG* procedure, we need to look at the sender variable for the message. If the sender was a variable with higher priority, then it is trying to assign a new variable in the *CPA*, so the *domainIndex* should be reset (lines 4 to 6). Otherwise, the sender is either a lower priority agent, in which case this is a backtrack step, so the *domainIndex* should remain the same so that the search for a new value for the current variable can continue; or the sender is the agent itself, which can happen for instance in the *FB_ESTIMATE* message processing, if the current estimated cost exceeds the known upper bound.

Another difficult observation to make, and not obvious from the AFB paper [3], was how the timestamp mechanism should really work. Section 3.3. in the paper states that the two timestamp arrays should be compared lexicographically, up to the *i-1* position. In fact, most of the times, they need to be compared up to and including the i-th position. Let us look at the case when a *CPA_MSG* is received, for instance. The point of comparing the timestamps of the sent message with that last known by the current agent is to see if the received message is out of date. If the i-th position of the timestamps shows different values, this indicates that a new assignment was made to variable i and the operation is no longer valid. The only exception is when receiving *FB_CPA* messages. In this case we only look up to position *i-1* and do not take into account the counter for position i, since variable i is only estimating a **Future Cost** and in doing so, it uses assignments of higher priority variables, not its own assignment.

2.3.3 Optimizations

We propose an optimization for storing the estimates. Receiving estimates is the result of an upper bound request, which is only sent to lower priority agents. So, in general, agent i only needs to store *n-i-1* estimates, where n is the total number of variables.

One can also notice that the h bound can be computed once per variable, since it is independent of the assignments of higher priority variables and, in fact, depends only on the assignment of that variable. Also, to avoid recomputation, we store the values for all possible assignments of each variable in a global attribute and only compute this once, before the algorithm starts. Notice that privacy is still respected, since each variable will only know the computed value of h for its own values. This approach ensures that when we need to compute an estimate, via the function f, we only need to compute the **Local Cost** and add it to the precomputed value h.

Regarding the timestamp mechanism, [3] suggests that the *assignmentCounter* of each variable can be reset to zero, to reduce the message size. We have done this in the *backtrack* method: before proceeding to call *assign_CPA* on the previous (higher priority) agent, the current agent resets it counter to zero (line 5). This makes sense because only the positions up the current variable (at most) are relevant in timestamp comparison. Moreover, in the *assign_CPA* method, we have identified another possibly to reset the *assignmentCounter* of a variable, for the case when its assignment has been reset (line 4).

Another small optimization lies in the way out-of-date messages are tested for the *UB_MESSAGE*s. The paper does not cover this aspect, which the implementation of the algorithm showed to be necessary. However, instead of doing the regular lexicographic comparison of the timestamp arrays, we found that it is better to just test if the upper bound the message carries is better (smaller) than the known upper bound, B, that the agent holds (line 1). If this is not the case we ignore the message. This is only a simple value comparison and is, in fact, representative for the meaning of an out-of-date *UB_MESSAGE* message: if the message is out of date, it means that another *UB_MESSAGE* updated B with a better value.

The *assign_CPA* method also does two small tricks in the case when the current variable is the last one in the ordering. Firstly, it updates the agent best know upper bound (B) and corresponding assignments (*bestCPA*) and then broadcasts this information to all agents (lines 22 to 24). This ensures that B is updated immediately, thus faster than it would be if it were to wait for the broadcast message to arrive. This allows for faster pruning of the search space. Secondly, it checks to see if the current B has reached optimal cost, zero, in which case it prematurely terminates all agents (lines 25 to 27).

The final code was further optimized using the TPTP (Eclipse Test & Performance Tool) Plugin for Eclipse (`http://www.eclipse.org/tptp/`), as well as the default profiler that comes with Netbeans IDE (`http://netbeans.org/`).

3 AFB Experiments

3.1 Real life problem experiments

We considered small, random instances of the distributed kidney exchange problem to test the performance of the AFB algorithm because they have the same characteristics as a real-life problem. The reason we used small instances is, partly, that we wanted to run the experiments in a reasonable amount of time and, partly, hardware limitations (the amount of memory that can be made available to the JVM for DPOP-based algorithms).

3.1.1 Kidney Exchange Problem Overview

Kidney transplants are one of the most frequent organ transplants. In the USA, for instance, tens of thousands of patients are waiting for a kidney transplant, and thousands die every year before they get one. It is a known fact that a person can live a normal life with only one kidney, so close friends or relatives might be willing to donate a kidney. Unfortunately, often enough, these people are biologically incompatible with the patient. To solve the problem, hospitals have started looking into the possibility of swapping donors: if patient A's friend can give a kidney to patient B instead, and B's friend can give a kidney to A in return, then every patient gets a kidney. It is even possible to consider three-way exchanges: A's friend gives to B, whose friend gives to C, whose friend gives back to A. Because the

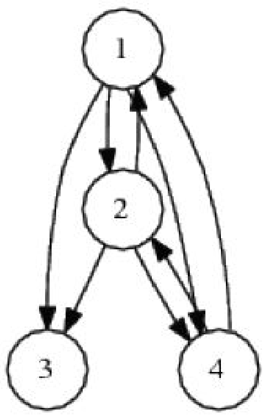

Figure 2: A compatibility graph and the corresponding DFS for a kidney exchange scenario of 6 pairs. The arrows indicate the direction in which the organs can be exchanged.

transplants have to take place simultaneously, only two-way or three-way exchanges are feasible in practice.

Figure 2, from [5], shows an example compatibility graph with 4 patient-donor pairs. In this scenario, pair3 can receive from both 2 and 1, but can give to no other pair. There are three possible two-way exchanges (1,2),(1,4),(2,4) and one possible three-way exchange in two possible directions (1,2,4) or (2,1,4).

It is possible to formulate this problem as a DCOP and solve it with algorithms such as AFB. We have used the Kidney Exchange Problem Generator in FRODO. We use the following formulation, which is different from the one initially proposed in [5]. In the DCOP, each agent corresponds to a patient-donor pair, and owns two variables: one modeling the decision of whom to give a kidney to, and the other of whom to receive a kidney from. The constraints are the following.

- Each agent has an internal constraint over its two variables, which declares a utility equal to:

  10 if the agent receives a kidney from the same agent it gives its kidney to (2-way cycle);

  $-\infty$ if it gives without receiving or receives without giving;

  0 otherwise.

- Binary inter-agent constraints enforce that agents agree on who gives to whom;

- Ternary constraints involving pairs of neighboring agents declare a utility of 29 if they are involved in a 3-way cycle, and 0 otherwise.

The utility values have been chosen in such a way that each transplantation gives a utility of 10, but 3-way exchanges are more complicated to set up than 2-way exchanges and are therefore penalized with a cost of 1.

3.1.2 Experimental Setup

Because of the high connectivity of the problem and limited memory (2GB) available on the testing machine, we were only able to test these cases up to a size of at most 10 pairs.

We compared the following algorithms in FRODO:

- our implementation of AFB
- SynchBB (ch.epfl.lia.frodo.algorithms.synchbb.SynchBBsolver)
- ADOPT (ch.epfl.lia.frodo.algorithms.adopt.ADOPTsolver)
- DPOP (ch.epfl.lia.frodo.algorithms.dpop.DPOPsolver)
- ASODPOP (ch.epfl.lia.frodo.algorithms.asodpop.ASODPOPsolver)
- ODPOP (ch.epfl.lia.frodo.algorithms.odpop.ODPOPsolver)

For this comparison, we measured the **total number of messages exchanged**, the **total message size**, the **total algorithm runtime** and the **Non-Concurrent Constraint Checks (NCCCs)**. Typically, the performance of distributed algorithms is measured with two independent metrics [2]:

- communication load - in our case the number of messages and total message size.
- run time - in our case total algorithm runtime and NCCC.

NCCCs are an indication of logical runtime in terms of number of *atomic* steps of computation needed. The atomic step here is a constraint check.

Code for these experiments can be found in the benchmark package *ch.epfl.lia.frodo.benchmarks.kidneys*. We have added a class *KidneyExperiment.java*, that creates a new instance the Java Virtual Machine to solve a given problem with a given algorithm (this ensures that each algorithm gets an equal treatment). The timeout for solving a problem was set to 2 minutes. The problem sizes (numbers of patient-donor pairs) ranged from 5 pairs to 9 pairs. For each problem size, we generated 101 problems (choosing an odd number makes the median better defined) and computed the median for each of the metrics we measured. The results for all algorithms were finally printed in text files.

We have written a script *doExperiments.ps1* to generate and process all the data. It is written in Windows Powershell and can be run on other

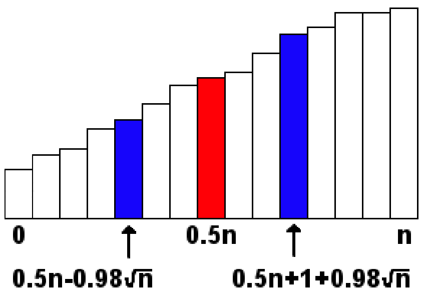

Figure 3: Represented median(red) and confidence interval (blue)

platforms using the Mono package (`www.go-mono.com/mono-downloads/download.html`) and Pash (`http://sourceforge.net/projects/pash/`). We intend to rewrite this script in Python in the near future. Finally, this script dynamically generates Matlab scripts to plot the results for each metric. A logarithmic scale is used for the **total algorithm runtime**.

The graphics represent the median with 95% confidence interval over the 101 samples. To compute the confidence intervals for the median we sort all the results for each metric. We then choose the median value and the confidence intervals indexes, as shown in Figure 3 from [5]. The number of considered problems n must be at least 71 for this formula to hold. In experimental results, if a confidence interval is not present, this means that the interval is of length 0.

All experiments have been run on a Toshiba Protege R705-P41, 2.53 GHz Intel Core i5-460M dual-core processor, on Windows 7 with 2GB of JVM memory.

3.1.3 Results

Figure 4 shows the simulated time for all algorithms. Notice that for problem size=10, Adopt grows at a high exponential rate and times out after 2 minutes. The ASO-DPOP and O-DPOP graphics overlap for all of the four metrics. The number of messages and total information exchanged are fairly high for AFB (Figures 5 and 6), they are only surpassed by Adopt for small problem sizes. The NCCC counts are also very high for AFB (one order of magnitude higher than SyncBB on average for all sizes; Figure 7), but this was due to a bug that was since fixed, and we did not have the time to re-run the experiments. However, the total runtime of AFB is not that much higher than that of the other algorithms (Figure 4) and its rate of increase in runtime with respect to the increase in problem size is comparable to the best one, of SyncBB. However, we have recently found room for improvement to reduce the message sizes (for example, currently our backtrack messages also include the CPA. This could be removed, since the agent receiving the

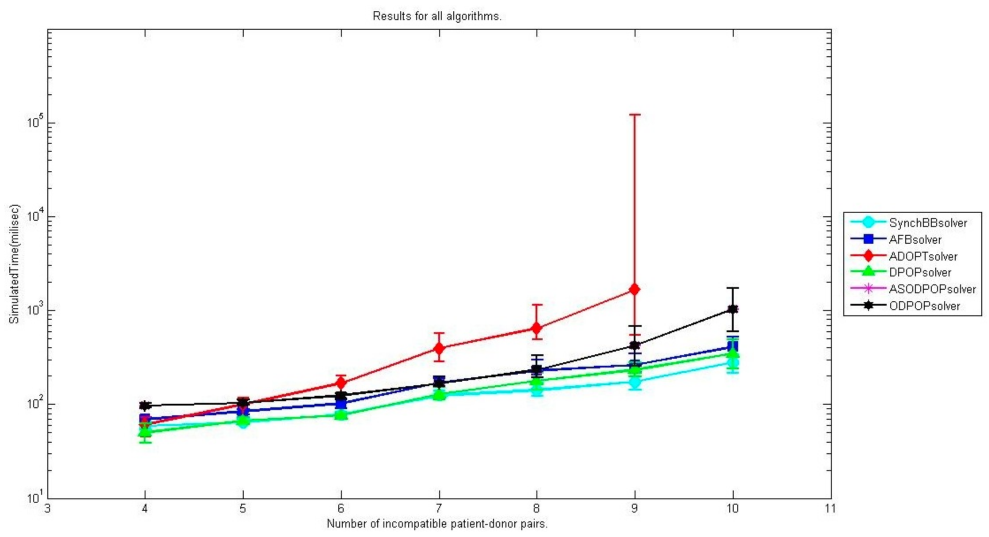

Figure 4: Median with 95% confidence intervals for algorithm runtime in milliseconds (sample size = 101)

backtrack checks if the backtrack message is up-to-date, and if this is the case, it means that the PA the agent holds is the most up-to-date one, thus making the CPA included in the backtrack messages unnecessary). Our results suggest that SyncBB offers the best tradeoff between communication load and runtime on this problem class.

In order to get a better idea of how much worse the other algorithms perform, we also looked at the same data, using SyncBB as a baseline. More specifically, for each problem instance and for each metric, we recorded the relative result of each algorithm as the actual result from which we subtracted the SyncBB result. We then computed the medians and confidence intervals. We thought this process would also help reduce the variability of the results.

Figures 8 to 9 show the relative results on a logarithmic scale. An effect of using the logarithmic scale is that it can only show positive values, while the negative values are excluded. This means that only the sections of relative increase with respect to SyncBB are showing in each graph.

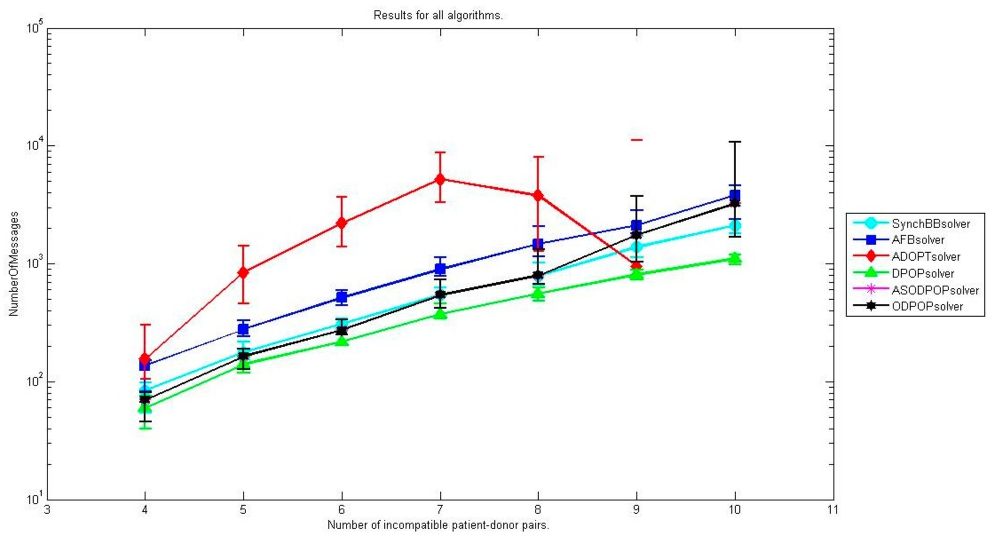

Figure 5: Median with 95% confidence intervals for total number of messages (sample size = 101)

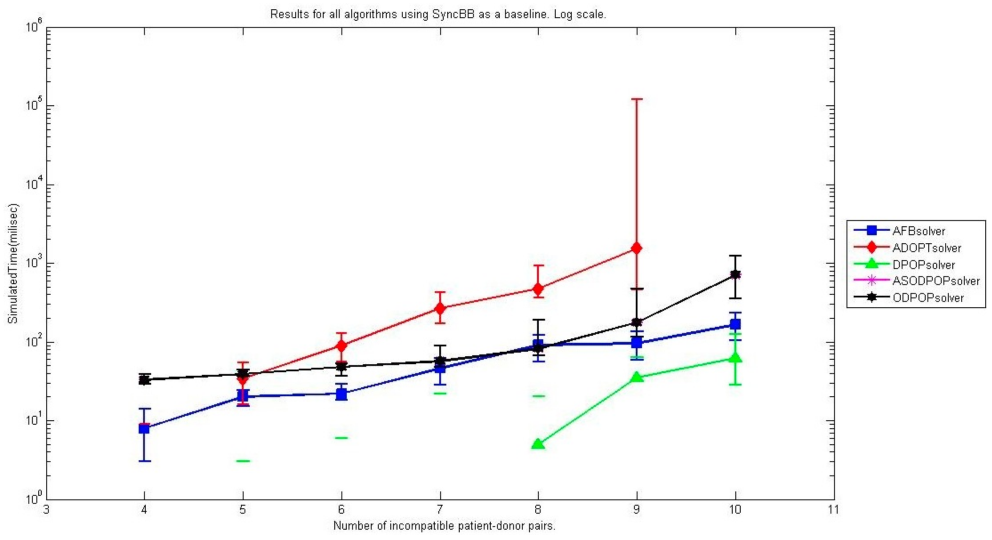

Figure 8: Median with 95% confidence intervals for total runtime, using SyncBB as a baseline (sample size = 101)

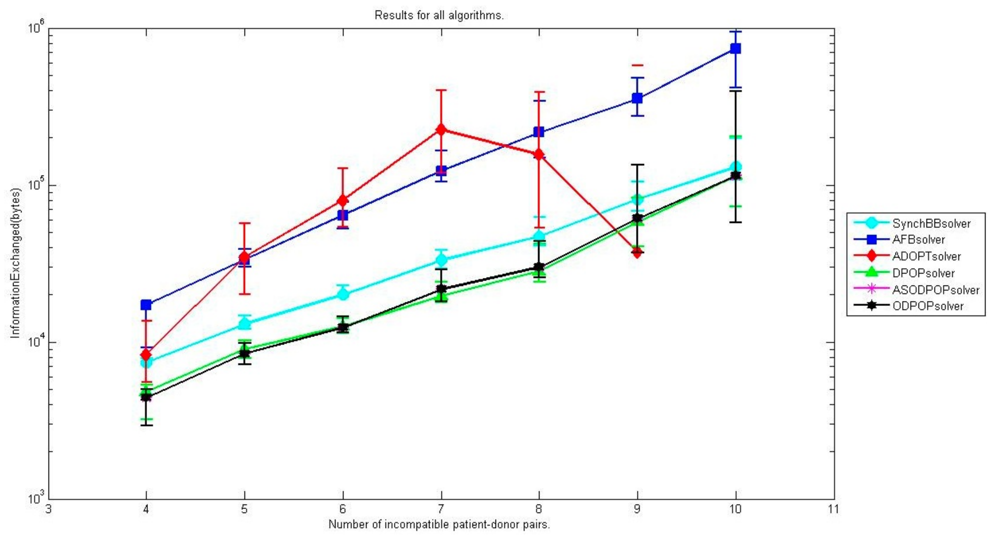

Figure 6: Median with 95% confidence intervals for total information exchanged (sample size = 101)

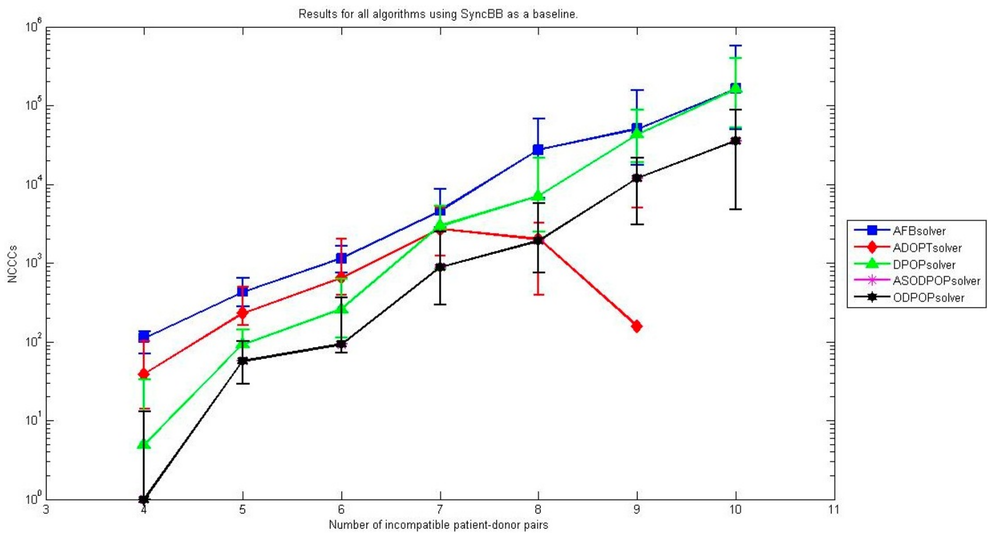

Figure 9: Median with 95% confidence intervals for NCCC, using SyncBB as a baseline (sample size = 101)

Notice that DPOP actually takes less time to run than SyncBB for problem sizes up to 8 (Figure 8 indicates a negative logarithm, so there is a relative decrease in runtime for DPOP, with respect to SyncBB). For sizes greater or equal to 8, DPOP shows a relative degradation of performance

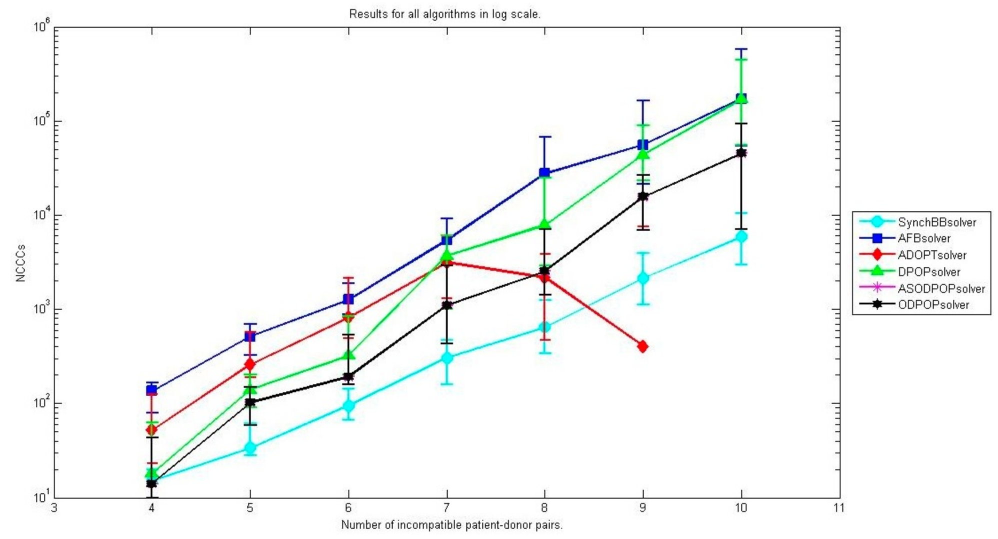

Figure 7: Median with 95% confidence intervals for NCCC (sample size = 101)

(equivalent to an increase in runtime) of about one order of magnitude. AFB's relative degradation of performance with respect to SyncBB runtime is almost constant along all problem sizes, while ODPOP and ASODPOP start off with a smaller degradation (up to size 8) but seem to decrease much faster beyond that point. Adopt is faster than SyncBB only for very small problems (size 4), but for sizes bigger than 4 it increases in run time much faster than any of the other algorithms. DPOP constantly decreases in number of messages and information exchange with respect to SyncBB (Figure 10, ODPOP and ASODPOP show a slight increase in number of messages, but fairly constant in information exchanged. Adopt shows an initial rapid increase for sizes up to 7, and then decreases. AFB's shows an increase in both number of messages (relatively small and linear), and information exchanged (higher).

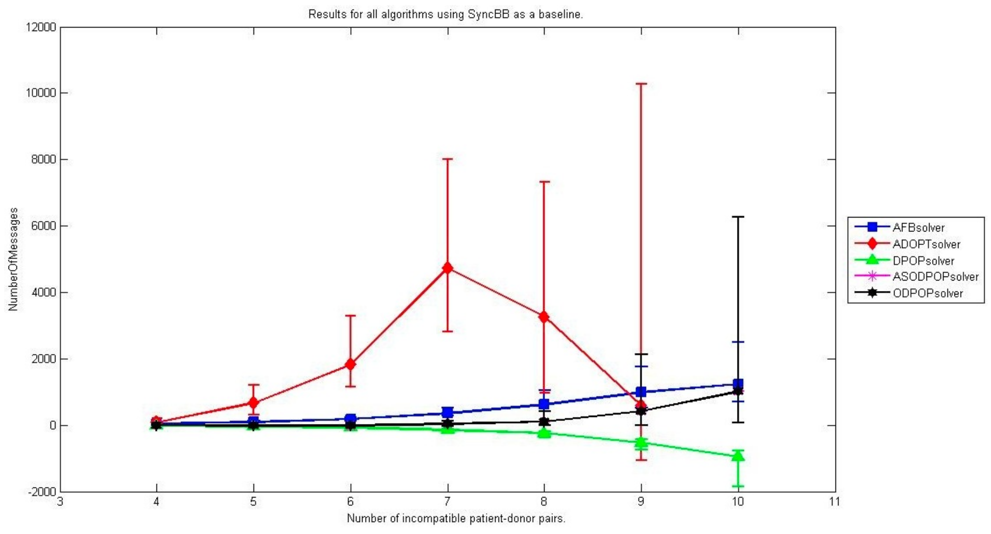

Figure 10: Median with 95% confidence intervals for total number of messages, using SyncBB as a baseline (sample size = 101)

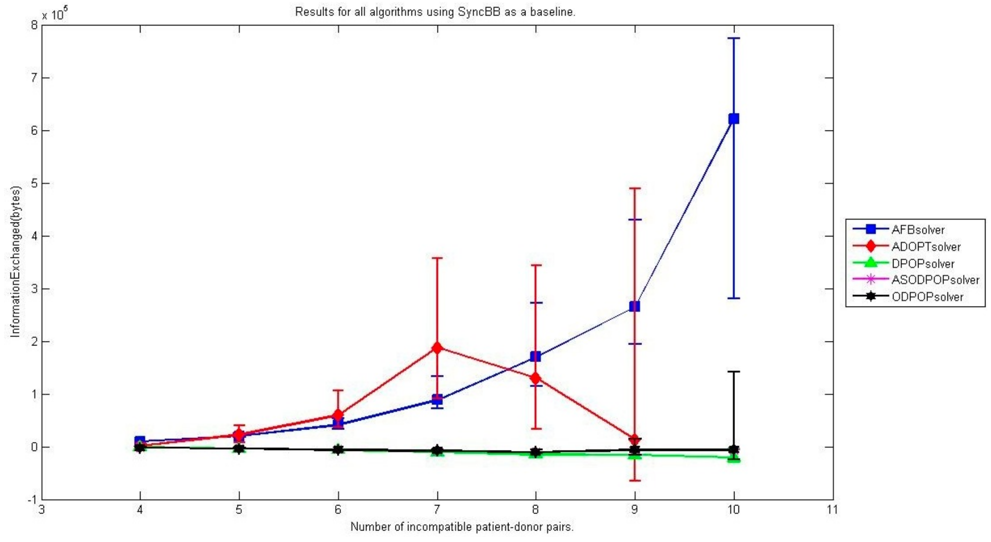

Figure 11: Median with 95% confidence intervals for information exchanged, using SyncBB as a baseline (sample size = 101)

These results seem to indicate that AFB is not the best suited algorithm for the kidney exchange problem. This could be due to the problem's variety and complexity. We should also keep in mind that our results only apply to small instances of the problem, and it could be the case for larger instances or for different problem classes, that AFB might perform better.

3.2 Max-DisCSP experiments

To get a better ideea of AFB's performance, we decided to try to reproduce the results in [4]. These use a different problem class, called Max-DisCSPs.

3.2.1 Max-DisCSP overview

Max-DisCSP is a subclass of DCOP problems, where all constraints are binary and costs are equal to 0 or 1. They are minimization problems, so reducing the costs amounts to minimizing the number of constraints that are violated. Variables are assigned values from a uniform domain with values $1 \dots k$. The constraint graph is generated randomly, based on:

- *density*, p_1: for each possible pair of variables, the probability that a constraint exists between them.
- *tightness*, p_2: for a certain constraint and each possible value assignment to its variables, the probability that the constraint is violated (i.e. has cost=1).

Therefore, a Max-DisCSP is characterized by the number of variables (n), the uniform domain size (k), the constraint graph density (p_1) and tightness of constraints (p_2).

3.2.2 Experiment setup

We mostly used the same approach to setup this experiment as we did for the kidney problem. We have implement a new random problem generator with adjustable density and tightness for Max-DisCSP problems. Given n, k, p1 and p2, the generator outputs a new problem in an .xml file. The code can be found in the *ch.epfl.lia.frodo.benchmarks.maxdiscsp* package.

We used the same settings as the paper: problem size 10, domain size 10 and we measured the performance, in terms of NCCC, for the following algorithms:

- AFB
- SyncBB
- Adopt
- DPOP

One thing to mention is that the non-concurrent constraint checks count is automated in FRODO, and is done in the same way for each algorithm. We were not convinced by the argument in the paper [4] that different atomic operations can be counted for each of the DisCOP algorithms and compared uniformly.

We varied constraint tightness from 0.1 to 0.99, and used values p1=0.4 and p1=0.7 for density, in two different setups. The timeout for each problem run was 2 minutes.

3.2.3 Results

Figure 12 and Figure 13 show the results for low density problems (p1=0.4), as presented in [4] and as resulting from our experiments, respectively. The two AFB plots look similar enough and are within the same order of magnitude. This result indicates that our implementation of AFB is correct with respect to constraint checks. There is one exception, for tightness p2=0.99, when we did not find an increase in NCCC. This can be explained by a phase-transition in AFB's performance, as the tightness of the problems increases beyond some point [3]. Other DisCOP algorithms were not yet found to display such a behavior.

A very important observation is that our results for SyncBB are found to be approximately two orders of magnitude ***lower*** than the ones in [4] (starting just below 10^3 as opposed to well over 10^4). This was a surprising result, as we found SyncBB outperforms AFB on tightness values up to and including p2=0.9. However, SyncBB timed out for the most difficult problems (tightness p2=0.99) so we cannot say for sure if AFB's phase transition makes it more effective for these kinds of problem instances. We could infer that NCCC and actual algorithm runtime, both measures of performance, are somewhat correlated and a timeout in SyncBB, and not AFB, would also indicate a possible higher NCCC than AFB for very high values of tightness.

Our plot for the Adopt algorithm is smoother (possibly because we may have used more samples) and it does not show any timeout. In fact, we found Adopt also outperforms AFB for most values of tightness (except for the very high tightness values, p2 >0.90), whereas [4] suggests this only happens for p2 <0.7.

DPOP results are consistent with the paper, showing very little change regardless of the problem's tightness. This is to be expected, given the fact that DPOP does not perform any search or punning. This stability comes with a performance loss: DPOP performs worse than all the other algorithms for most problems (tightness up to 0.8).

Looking at the results for high density (p1=0.7) problems (Figure 14 and Figure 15), we notice a general increase in NCCC for all algorithms and a more rapid increase for AFB, SyncBB and Adopt compared to the lower density problems. This increase yields more timeouts, specifically:

- Adopt is the first one to timeout, starting with tightness value p2=0.7.

- AFB times out for tightness value p2=0.9.
- SyncBB is the last one to timeout, beyond tightness values p2 > 0.9.

Notice that SyncBB still outperforms AFB on almost all tightness values, except on the limit value p2=0.99. We can now conclude that AFB's phase-transition property makes it more effective for very tight Max-DisCSP problems. We could probably reach the same conclusion for lower density problems, if we had a higher timeout limit.

In matching with the results from the paper, Adopt is now shown to be outperformed by AFB, but we have found the degradation of performance to be more rapid for Adopt than for AFB. In other words, as the problems become more difficult, Adopt shows an exponential loss in performance.

DPOP's results are consistent with [4]: NCCC is higher than that of all the other algorithms, but it always terminates. It is again shown to be resistant to tightness changes, as opposed to the rapid increase of the other algorithms.

In conclusion, our experiment showed values and behavior similar to [4] for all algorithms, except SyncBB. We could not validate the claim that AFB is the clear "winner" for the whole range of problem difficulty [4]. In fact, we found that most of the times, SyncBB is the winner. AFB outperforms SyncBB only on random problems that are highly dense and very difficult (tightness 0.99).

Our intuition was that this discrepancy in the results must be due to a difference in implementation. After contacting the authors of [3], we found that they used a fixed variable ordering heuristic (probably the lexicographic order). FRODO, on the other hand, uses more effective heuristics: the *min-width* heuristic for SynchBB and AFB, and the *most-connected* heuristic for DPOP and ADOPT.

This makes for a very interesting observation: with a poor variable ordering heuristic, AFB outperforms SynchBB. The forward bounding mechanism makes it possible for high-priority agents to discover early in time that lower-priority agents are stuck exploring a fruitless part of the search space. They can then abandon this part and restart in another, possibly better subspace, yielding a performance gain for AFB. On the other hand, with a smart variable ordering heuristic, SynchBB does not get stuck deep in the search tree and AFB's pruning power no longer compensates for the computational overhead of performing forward bounding. As such, AFB is outperformed by SynchBB most of the time (and sometimes even by ADOPT).

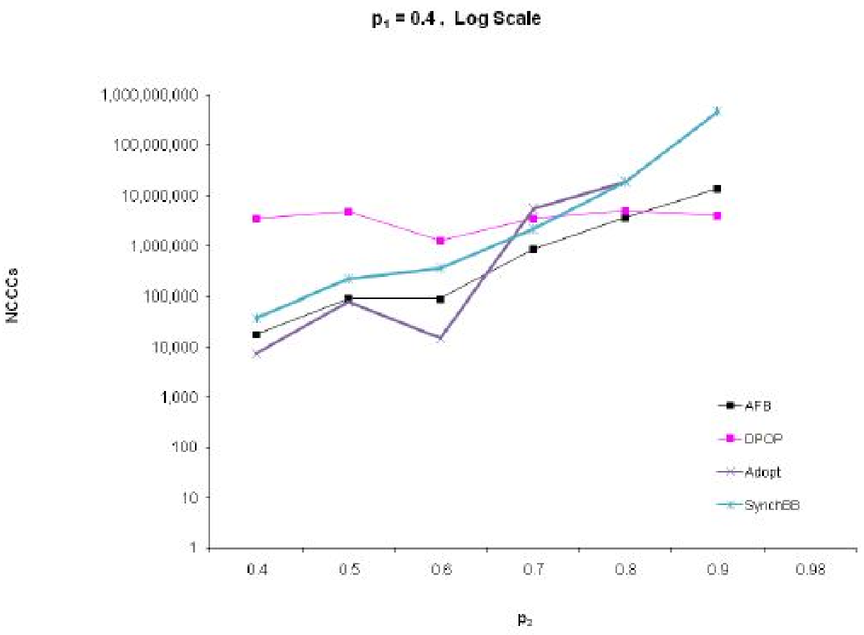

Figure 12: Empirical evaluation on random Max-DisCSP of low constraints density, as published in [4]

4 Conclusion

The main goal for this project was to understand and implement the AFB algorithm, as introduced in [3] and to compare it with different algorithms implemented in FRODO. Our initial experiments were done on small size, randomly generated distributed kidney exchange problems. We have seen that AFB does not offer the best tradeoff between communication load and runtime for this class of problems. In this case, SyncBB performs faster and uses less network resources, so it may be the case that AFB is not very well suited for the distributed kidney exchange problem class. Finally, we wanted to possibly find a class of problems for which AFB performs better. The paper [3] was interesting because it presented excellent results for AFB applied to Max-DisCSP problems, therefore we decided to try and reproduce these results. However, our experiments did not lead to the same conclusions. Most of the hypotheses were validated by our results, but, interestingly enough, we found that SyncBB again performs better than AFB fort most types of Max-DisCSP problems. In these last experiments, we have seen AFB's phase transition effect, which does make it be the best choice for problems with very high constraint tightness. We found that using a smart order heuristic for the variables can dramatically influence algorithms' performance. During our experiments, we have identified several

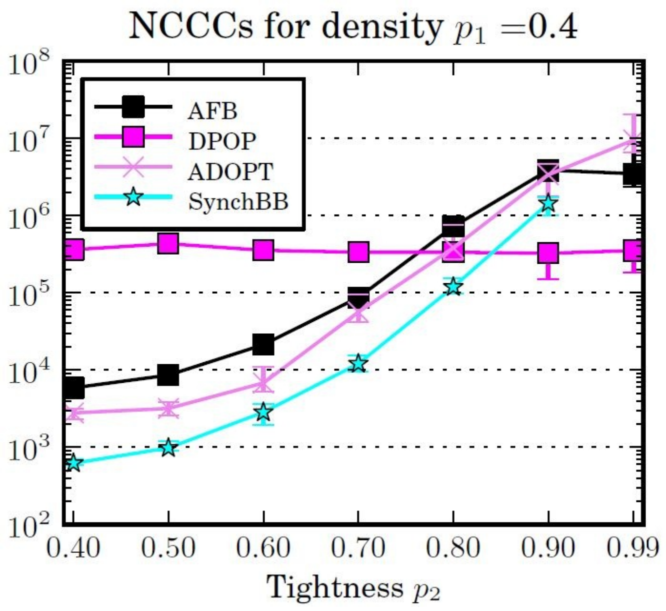

Figure 13: Random Max-DisCSP-NCCC comparison for low constraints density problems (90 samples for each tightness value).

future work items that could possibly clarify and improve these results.

5 Future work

We have identified room for improvement in the message sizes of AFB. Implementing this changes would help to see how much the sizes of messages can be reduced, compared to the other algorithms.

We could also implement a different baseline comparison method for the kidney experiment, using division instead of subtraction. This would allow us to make claims such as "Algorithm X is y times faster than SynchBB".

The Max-DisCSP problem experiments could be extended to:

- increase the timeout limit to complete the plots (for AFB in particular, it would be interesting to see how big is the increase in performance or very tight problems).
- run more problem instances to validate our claims.
- include ODPOP and ASODPOP algorithms.
- plot other metrics, such as total message size, information exchange and actual run time.

Several versions and adjustments to AFB are described in the literature. These would also be interesting to implement and compare.

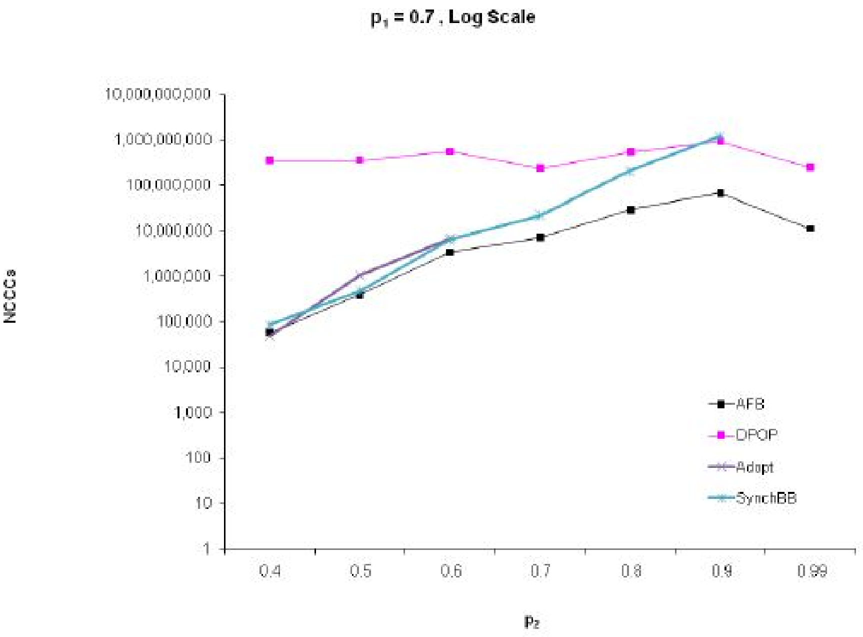

Figure 14: Empirical evaluation on random Max-DisCSP of high constraints density ([4])

One of them is the AFB with back-jumping [3]. This is a mechanism that allows an agent to "back-jump" to some higher priority agent, not necessarily the previous one. For example, if agent A_i computes an upper bound for assignments up to agent A_{i-2} and this is higher than the current known upper bound, that it can ignore all possible assignments for A_{i-1}. Intuitively, this should decrease the algorithm runtime.

The heuristic to compute h and a lower bound estimate, based on the current assignment $A_i = v$, can be replaced with other heuristics to compute a lower bound, according to [2].

Another extension could be inducing a value ordering heuristic, such as the minimum-cost heuristic in the AFB-minC [3]. In this case, values with lower costs due to assignments of higher priority agents are selected first. Experiments suggest that this can substantially improve performance.

Finding a problem class on which AFB constantly outperforms the other algorithms could also be interesting. In our experiments, we have seen how the choice of the problem can yield different performance results.

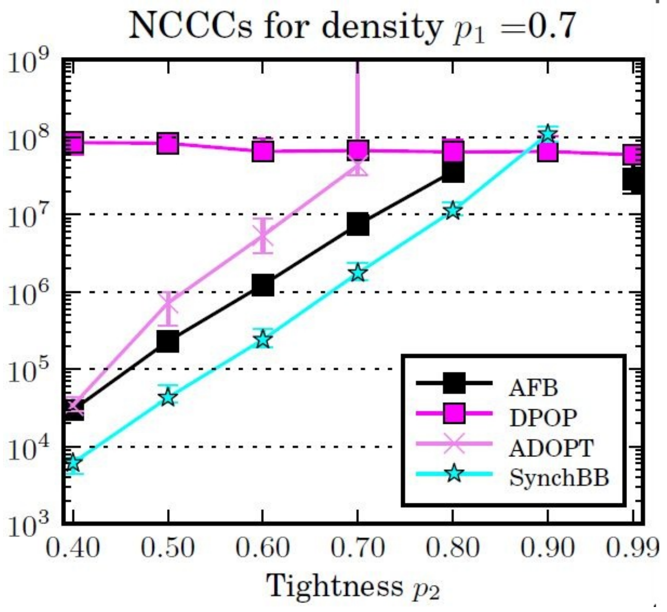

Figure 15: Random Max-DisCSP-NCCC comparison for high constraints density problems (63 samples for each tightness value).

References

[1] Boi Faltings. *Distributed Constraint Programming*, chapter 20, pages 699–729. Foundations of Artificial Intelligence. Elsevier, August 2006.

[2] Amir Gershman, Amnon Meisels, and Roie Zivan. Asynchronous forward-bounding for distributed constraints optimization. In Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, editors, *Proceedings of the Seventeenth European Conference on Artificial Intelligence (ECAI'06)*, pages 103–107, Riva del Garda, Italy, August 29–September 1 2006. IOS Press.

[3] Amir Gershman, Amnon Meisels, and Roie Zivan. Asynchronous forward bounding for distributed COPs. *Journal of Artificial Intelligence Research*, 34:61–88, February 2009.

[4] Amir Gershman, Roie Zivan, Tal Grinshpoun, Alon Grubshtein, and Amnon Meisels. Measuring distributed constraint optimization algorithms. In *Proceedings of the AAMAS'08 Distributed Constraint Reasoning Workshop (DCR'08)*, pages 17–24, Estoril, Portugal, May 13 2008.

[5] Jonas Helfer, Thomas Léauté, and Boi Faltings. Heuristics for distributed pseudo-tree regeneration. Semester project report, EPFL Artificial Intelligence Lab (LIA), January 9 2010.

[6] Katsutoshi Hirayama and Makoto Yokoo. Distributed partial constraint satisfaction problem. In Gert Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming (CP'97)*, volume 1330, pages 222–236, Linz, Austria, October 29–November 1 1997. Springer.

[7] Thomas Léauté, Brammert Ottens, and Radoslaw Szymanek. FRODO 2.0: An open-source framework for distributed constraint optimization. In Katsutoshi Hirayama, William Yeoh, and Roie Zivan, editors, *Proceedings of the IJCAI'09 Distributed Constraint Reasoning Workshop (DCR'09)*, pages 160–164, Pasadena, California, USA, July 13 2009.