Chapter 11

Asynchronous Reordering Heuristics

... not for the swift is the race,
nor does the mighty win the battle,
nor even the wise gains bread,
nor the intelligent wealth,
nor the learned grace,
for all of them depend on time and circumstances.
Solomon, Ecclesiastes 9:11, Cornilescu

In the previous chapter we have introduced a framework for dynamic agent reordering in asynchronous search with polynomial space complexity. Reordering of agents is interesting, especially for improving fairness and privacy, as foreseen in (Silaghi *et al.* 2000a). A well known solution for dealing with failures is to treat recovered agents as new agents (Debes 2000). It is easy to add new agents on the lowest priority position. If the initial position of the agent is preferred, the reordering is an elegant solution for achieving the required configuration.

Previous studies (Armstrong & Durfee 1997) show that known reordering techniques did not have a strong impact on efficiency in distributed search (up to 3 times). The dynamic techniques accredited by (Armstrong & Durfee 1997) to bring some limited improvements over random static order are the random dynamic reordering and versions of the AWC's heuristic. It is important to note that static order has performed as well as the best dynamic reordering!

The question that arises from here, is: *What kind of impact can dynamic reordering have on polynomial space asynchronous search.* This chapter exemplify how heuristics can be implemented in ABTR. While aggregation had had a higher impact on efficiency in polynomial space search than in search with unbounded space, the impact of reordering remained low in our tests.

As several other researchers agree, one of the reasons of the modest results of the reordering may lay in the small size of the problems and in the good static order.

11.1 Cost of the heuristics

In asynchronous search protocols, agents are allowed to freely select decisions from sets of allowed decisions. For example, in AAS agents can select among sets of possible aggregates. With PAS, agents can select among different ways of splitting the problem. Any constraint satisfaction researcher also notices that the most common decision is encountered in ABTR: With ABTR, agents can almost at any time choose a new order, and this choice has a great impact on the quality of the process.

In this chapter we only discuss reordering decisions. Theoretically, the impact of value reordering can be stronger than for variable reordering. However, it seems that in centralized settings, good variable orders can be guessed with probabilities leading to better results (Geelen 1992).

In general, the chosen orders are computed to minimize some function that estimates the cost (typically cost of the computation). Such functions are referred to as heuristics.

Algorithm designers analyze the trade-offs between the accuracy of a heuristic and its cost. Most successful heuristics are very cheap (Geelen 1992). In well-known centralized algorithms, the computation of heuristics for variable reordering reduces to simple measurements such as comparing the size of domains or the number of unsatisfied constraints. These are often so cheap that some evaluation techniques based on constraint checks even don't take them into account.

Unlike centralized search, in asynchronous search each heuristic measuring data from other agents requires message round-trips. Also, each reordering has to be announced by broadcasting messages. Moreover, with polynomial space asynchronous search, many nogoods, and therefore much work, can be lost after each reordering. As a consequence we infer that an efficient heuristic should:

- not depend much on external measures.
- not propose reordering too often.

The first criteria is satisfied by the random reordering, as well as by AWC. None of the heuristics described in (Armstrong & Durfee 1997) is concerned with the second criteria since the testing algorithm they use stores all the nogoods. In the following we first describe how trivial heuristics can be imported from centralized backtracking to ABTR. In Section 11.3 we describe a promising application of ABTR to fairness in cooperation search. In the last part of this chapter we detail the development of a complex heuristic, ABTR-wc. Since ABTR-wc brings small improvements for our tests, its impact remains small in comparison with the effect of aggregation or consistency maintenance.

11.2 Min-domain and Min-volume

Among heuristics for variable reordering, the min-domain first and max-connected first have proved to be the most successful. Min-domain first advises to instantiate next a variable that has a small number of values. This heuristic is intuitive as it follows the binary search paradigm: hierarchical splitting of search space with minimal breadth. The advantage of minimal breadth is that probabilistically, conflicts have to be detected only once for eliminating a larger part of the search space. Nevertheless, artificial examples where min-domain is the worst order can be easily built by defining a CSP with more than two variables and one binary constraint that makes impossible any combination of values for the two variables with highest number of values.

Max-connected first is a variable ordering heuristic advising to instantiate next the uninstantiated variable connected to the highest number of unchecked constraints. Two intuitive explanations are given now. The instantiation of the most connected variables lead to the instantiation of hard subproblems first. If we think at the dual representation of the CSP, hard subproblems can be seen as variables with small domains, and we can apply again the arguments for the min-domain heuristic. On the other side, (Sabin & Freuder 1997) explains how the removal of highly connected nodes can lead to tractable problems when the graph of the remaining CSP can be described with a tree.

In DisCSPs, these heuristics are related to the AWC heuristic that promotes constrained agents. A natural equivalent of min-domain is a heuristic that we have called min-volume (Silaghi *et al.* 2001g). Min-volume defines an order on agents based on the size of their current search space:

$$Volume_j(A_i) = \Pi_{x^k \in CSP(A_i), k \leq j} |allowed(D_{x^k})|.$$

where $allowed(D_{x^k})$ is $set(a)$ where a is the strongest assignment known for x^k. $set(a)$ was introduced in Definition 9.8. To enable the transfer of this technique to extensions of AAS, we define

$$view_j(A_i) = \{a \mid a \in view(A_i), ord(signature(a)) \leq j\}.$$

We recall that ord of a signature, h, is the position of the generating agent when h was built. $view(A_i)$ was introduced in Definition 9.9.

To be noted that min-volume is an approximation of the *Number of Local Solutions (Num Sln)* heuristic that performed best in (Armstrong & Durfee 1997).

On position $j+1$, one places the agent with minimal $Volume_j$. We have tested several incrementally improved implementations of the min-volume heuristic for ABTR with $A_j \equiv R^j$:

- After each modification of its view, A_i sends a **heuristic**($Volume_j(A_i)$) message to each higher priority agent A^j. The **heuristic** messages are tagged with a counter to ensure FIFO order.
- To repair the fact that the information transmitted via **heuristic** is often old by the time of the reception, one can attach to it $view_j(A_i)$. On reception, the relevant context will also be delivered with **ok?** messages.
- To reduce network traffic, agents only send **heuristic** messages when these can be attached to **nogood** messages. An agent A^j estimates the volume of a lower priority agent A_i by using $view(A^j)$, public knowledge concerning the var(A_i), and the last received $view_j(A_i)$.
- To reduce the number of proposed reorderings, $A^j(o)$ only proposes a new order o' when it estimates that $Volume(A^{j+1}(o')) > k * Volume(A^{j+1}(o)), k \geq 1$. k is a threshold of the ratio of volumes that promises a better order.

These heuristics integrated in DMAC have underwent preliminary tests for 100 randomly generated problems with 20 agents, 25 variables of 8 values and 21 binary constraints with a tightness of 55% (approximatively. the peak of complexity for DMAC). The tests were lead for two initial orderings:

- two constraints in A^{20}, and one constraint in each other agent.
- two constraints in A^1, and one constraint in each other agent.

For the first ordering all the mentioned heuristics required app. 1.5 more sequential messages than DMAC, and they approached to DMAC for $k = 5/3$. For the second ordering, only the heuristic with $k = 5/3$ was tested and it was app. 1.5 times more efficient than DMAC. While in (Armstrong & Durfee 1997), dynamic and static Num Sln perform alike, with polynomial space search, dynamic min-domain performed slightly less well than static min-domain.

11.3 The fair heuristic

When the distributed search is used for negotiations, the position of an agent A in the ordering on agents corresponds to certain advantages and drawbacks for A. With some given problems, agents prefer to be the first positioned ones such that they could propose their preferred alternatives first. For other problems, agents want to be positioned later such that they need to reveal less about their constraints/domains. In ABTR one can reorder randomly or 'round robin' the agents in order to enhance fairness. Then, one still has to check or trust that the agents respect the chosen reordering strategy.

However, when the agents form equal sized coalitions, fairness can be ensured in ABTR by majority voting. Let us assume that the agents want to solve a problem where privacy is essential. Agents want to place other agents from their coalition on last positions. A fair alternation of agents from different coalitions, when we have equal sized coalitions, is obtained when the function of $R^k, k>0$ is undertaken by *an entity* defined by the set of agents $A^1, ..., A^k$. The agents in R^k choose the agent A^{k+1} by majority voting. Fairness results from the fact that for these problems, low position agents will propose on next lower positions agents from other coalitions.

The majority voting can be implemented by letting any A^i in R^k broadcast to any other agent in R^k the set of agents that A^i would like to see on position A^{k+1}. The set is tagged with the history of $O^{crt}(R^k)$. E.g., the agent chosen for the position A^{k+1} can be the currently lowest position agent among those with the highest number of occurrences in the proposals. The time t_r

```
when received (nogood,A_j^v,¬N,⟨o,h⟩) do
    validOrder← getOrder(⟨o,h⟩);
    if (¬validOrder ∧ (A_i ≠CEA(¬N))) then return;
    discard→ false;
    if ((((⟨x_i,d,c⟩∈N ∧ (A_i knows (M→(x_i≠d))) ∧ ¬(better ¬N than ¬M))
            ∨ invalid(¬N))) then
        if (I do want to discard ¬N) then
            discard ← true;
        else
            store ¬N as redundant constraint;
    else
        put ¬N in nogood-list for x_i=d;
    if ¬discard then
        when ⟨x_k,d_k,c_k⟩, where x_k is not connected, is contained in ¬N do
            send add-link to A_k;
            add ⟨x_k,d_k,c_k⟩ to agent view;
        add other new assignments in N to agent view;
    when (validOrder and valid ¬N and d=current_value)
29.1    C_{v-1}^order++;
        new_order ← A^1,...,A^{v-1},A_j,...;
        getOrder(⟨new_order,new-signature⟩);
29.2    make sure to send O^crt or (reorder,O^crt) to all A^j, j≥v;
    check_agent_view;
```

Algorithm 29: *Procedure of A_i^v for receiving* **nogood** *messages in ABTR-wc.*

required by R^k for deciding a new order is then equal to the maximum message delivery delay τ. The correctness, completeness and termination properties of ABTR are therefore maintained.

Similarly, when the quality of the found solution is essential and privacy is less important, a fair reordering is obtained when the function of R^k is undertaken by the set of agents $A^{k+1}, ..., A^n$. With equal sized coalitions, a fair alternation is obtained when the agent A^{k+1} is decided with the majority of the votes of the agents forming the entity R^k.

11.4 AWC-like reordering heuristic (ABTR-wc)

In the previous chapter we have described a reordering technique that allows for using **heuristic** messages for guiding reordering but we did not propose any specific heuristic. ABT performs Forward Checking since the domains of the future variables are pruned after each assignment. However, using **heuristic** messages can be expensive since 3/2 message round-trips $A^i \rightarrow A^j \rightarrow R^i \rightarrow A^{i+1}$ are required for getting the result of pruning.

11.4.1 Adapting a heuristic

In this section we show how an idea related to the one behind the heuristic used in AWC can be reused with ABTR1. In AWC each agent A_i that discovers a new nogood $\neg N$ increases its own priority. Let us assume that the lowest priority assignment in N concerns x^v. In ABTR the agent A_i has to send $\neg N$ to A^v. We can spare messages by letting A^v decide the new order that gives more priority to A_i. To make sure that it increases the priority of A_i in the general case, A^v can offer to A_i its position v or a lower one. To do this, A^v has to act for some $R^{v-t}, t > 0$. Let each A^k act for R^{k-1} and let it cede its position to the sender of each valid nogood received (Algorithm 29). This protocol is called ABTR-wc.

ABTR-wc is a protocol allowing for nogood removal. Whenever some nogoods are removed, heuristics can be used to decide the nogoods that should be stored.

Definition 11.1 *$\neg N_1$ is a* better nogood *then $\neg N_2$ if N_1 contains only higher priority variables than the lowest priority variable in N_2. The priority of a variable is the priority of the agent owning it.*

Theorem 11.1 *Better or valid nogoods can be received by agent A^i in ABTR-wc only in finite time after the agents $A^j, j < i$ have reached quiescence.*

Proof. Each valid nogood received by A^i eliminates a value. Each value eliminated by such a nogood will never be available since the nogoods eliminating them (or better nogoods received later) use fixed assignments of quiescent agents and cannot be invalidated. There are maximum dn such values and better nogoods can be received for a value only n times. Therefore, they are received in a finite time. □

It is required that R^k does not send reorders beyond delay $t_r + t_h + 2\tau$ after an instantiation is done by some $A^j, j \leq k$.

Corollary 11.1.1 *$\exists t_h$ such that, when proposing reordering, the agent A^k acts legally for R^{k-1} and ABTR-wc is an instance of ABTR1.*

There exists a finite t_h such that any valid or better nogood is received by R^{k-1}/A^k in a time bounded by t_h after the quiescence of the agents $A^l, l < k$.

11.4.2 Saving effort across reordering for ABTR-wc

In the Algorithm 29, the line 29.1 actually allows for several versions since agent A_i is free to put whatever order among the agents following A_j. By ABTR-wc we denote a version where the agents following A_j are ordered lexicographically. This convention requires no additional information. However, due to reordering involved on future agents many of the nogoods own by successors are invalidated.

In a version of ABTR-wc, that we denote ABTR-wc1 (Algorithms 30 and 31), each agent maintains one more ordering called *last_known_successor_order* and denoted L^o. The line 11.2 in Algorithm 11 is modified in ABTR-wc1 such that the sent nogood is also accompanied with the ordering O^{crt}. L^o holds the most recent ordering among O^{crt} and the orderings received with **nogood** messages. In ABTR-wc1 the *new_order* at line 29.1 becomes $A^1, ..., A^{v-1}, A_j, A_i, ...$, where the agents following A_i are ordered according to L^o.

A further modification we make to ABTR-wc1 is that each new proposed ordering is sent at line 29.2 to all agents so that all of them can update their *last_known_successor_order*. This last version is denoted ABTR-wc2.

11.5 Experiments

We have run tests on random problems with 20 agents. The agents were placed on distinct computers of our lab. Since here the reordering was introduced in ABT which is a slow algorithm, the size of the problems had to be small. We have generated problems of variable tightness for a density of 27% where each variable has 3 values. Bigger problems can be addressed when this technique is included in algorithms such as AAS or DMAC. Each point in Figure 11.1 is averaged over 100 random problems and shows the average number of sequential messages (half network round-trips) required to solve the problem. The number of round-trips is the only important cost when agents are placed remotely on Internet and the local problems are not hard. The experiments show that ABTR-wc2 performed somewhat better in average than ABT and performed better than other versions of ABTR-wc. For under-constrained problems (percentage of feasible tuples over 85%) where solutions are found without resorting to many **nogood** messages, few reorderings are proposed by ABTR-wc and therefore the new algorithms perform quite similarly with ABT.

It is possible that better results could be achieved if larger problems would be treated. However, so far the size of the problems that I could address in reasonable time is rather small.

```
when received (ok?,⟨x_j, d_j, c_{x_j}⟩,⟨o, h⟩) do
    getOrder(⟨o, h⟩);
    if(old c_{x_j}) return;
    add(x_j,d_j,c_{x_j}) to agent_view;
    reconsider stored and invalidated nogoods;
    check_agent_view;
when received (nogood,A_j,¬N,⟨o, h⟩[,L]) do
    getOrder(⟨o, h⟩);
    if I am not CEA(¬N) then return;
    [if L newer than L^o then L^o←L];
    discard ← false;
    if ((((⟨x_i, d, c⟩∈N ∧ (A_i knows (M→(x_i≠d))) ∧ ¬(better ¬N than ¬M))
            ∨ invalid(¬N))) then
        if (I do want to discard ¬N) then
            discard ← true;
        else
            store ¬N as redundant constraint;
    else
        put ¬N in nogood-list for x_i=d;
    if ¬ discard then
        for every ⟨x_k, d_k, t_k⟩ ∈ N, where x_k is not connected do
            send add-link to A_k;
            add ⟨x_k, d_k, t_k⟩ to agent_view;
        add the other new assignments in ¬N to agent_view;
30.1    reconsider stored and invalidated nogoods;
    old_value ← current_value; check_agent_view;
    when old_value = current_value and if A_j has lower priority than A_i
30.2    send (ok?,⟨x_i, current_value, C^i_{x_i}⟩,O^{crt}) to A_j;
```

Algorithm 30: *Procedures of A_i for receiving messages in ABTR and ABTR-wc1. Code between '[' ']' is only for ABTR-wc1*

11.6 Summary

We have described the first non-random heuristics for concurrently and asynchronously reordering variables during systematic complete asynchronous search with polynomial space requirements. Asynchronous reordering is required for security reasons in managing coalitions in automated negotiation (see (Silaghi *et al.* 2001c)), and it is useful for management of multi-agent systems. Moreover, it increases freedom of agent decisions. We exemplify with details the development of ABTR-wc, a reordering heuristic for ABTR that requires no heuristic message. As with most distributed asynchronous algorithms, certain resemblance (Havens 1997; Silaghi *et al.* 2000a) can be found between the behavior of ABTR-wc and Dynamic Backtracking (Ginsberg 1993b). Here the resemblance extends also to the reordering heuristic. Developing ABTR, we have realized that the solution detection algorithm in (Silaghi *et al.* 2000g) cannot be used with reordering and public constraints. A new solution detection algorithm for ABTR was then designed and its correctness is proved. That algorithm is presented in Chapter 20.

```
procedure check_agent_view do
    when agent_view and current_value are not consistent
        if no value in D_i is consistent with agent_view then
            backtrack;
        else
            select d ∈ D_i where agent_view and d are consistent;
            current_value ← d; C^i_{x_i}++ ;
            send (ok?,⟨x_i, d, C^i_{x_i}⟩,O^crt) to lower priority agents in outgoing links;
procedure backtrack do
    nogoods ← {V | V = inconsistent subset of agent view};
    when an empty set is an element of nogoods
        broadcast to other agents that there is no solution;
        terminate this algorithm;
    for every V ∈ nogoods do
        select (x_j,d_j,t_{x_j}) where x_j has the lowest priority in V;
31.1    send (nogood,A_i,V,O^crt_j[,O^crt]) to A_j;
        remove (x_j,d_j,t_{x_j}) from agent_view;
        reconsider stored and invalidated explicit nogoods;
    check_agent_view;
function getOrder(⟨o,h⟩) → bool //ABTR
    when h is invalidated by the signature O^crt then return false;
    [when newer h than L^o or same h as for L^o but longer o
        L^o ← ⟨o,h⟩];
    when not newer h than O^crt
        return true;
    I ← reorder position for h and the signature of O^crt;
    invalidate assignments for x^j, j ≥ I;
    ⟨o,h⟩ → O^crt;
31.2 make sure that send (ok?,⟨x_i, some value, c_{x_i}⟩,O^crt) will be performed
         to all lower priority agents in outgoing links;
    return true;
```

Algorithm 31: *Procedures of A_i in ABTR and ABTR-wc1. Code between '[' ']' is only for ABTR-wc1*

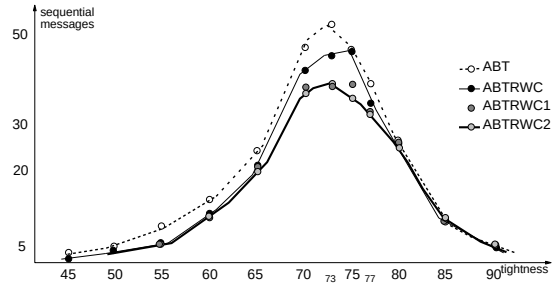

Figure 11.1: Experiments