DynABT: Dynamic Asynchronous Backtracking for Dynamic DisCSPs

Bayo Omomowo, Inés Arana and Hatem Ahriz

School of Computing,
The Robert Gordon University, Aberdeen
{bo,ia,ha}@comp.rgu.ac.uk

Abstract. Constraint Satisfaction has been widely used to model static combinatorial problems. However, many AI problems are dynamic and take place in a distributed environment, i.e. the problems are distributed over a number of agents and change over time. Dynamic Distributed Constraint Satisfaction Problems (DDisCSP) [1] are an emerging field for the resolution problems that are dynamic and distributed in nature. In this paper, we propose DynABT, a new Asynchronous algorithm for DDisCSPs which combines solution and reasoning reuse i.e. it handles problem changes by modifying the existing solution while re-using knowledge gained from solving the original(unchanged) problem. The benefits obtained from this approach are two-fold: (i) new solutions are obtained at a lesser cost and; (ii) resulting solutions are stable i.e. close to previous solutions. DynABT has been empirically evaluated on problems of varying difficulty and several degrees of changes has been found to be competitive for the problem classes tested.

Key words: constraint satisfaction, distributed AI, dynamic problems

1 Introduction

A Constraint Satisfaction Problem (CSP) can be defined as a triple $Z = (X, D, C)$ containing a set of variables $X = \{x_1....x_n\}$, for each variable x_i, a finite set $D_i \in D$ of possible values (its domain), and a set of constraints C restricting the values that the variables can take simultaneously. A solution to a CSP is an assignment to all the variables such that all the constraints are satisfied.

Dynamic Constraint Satisfaction problems (DCSPs) were introduced in [2] to handle problems that change over time. Loosely defined, a DCSP is a sequence of CSPs, where each one differs from the previous one due to a change in the problem definition. These changes could be due to addition/deletion of variables, values or constraints.Since all these changes can be represented as a series of constraint modifications [3], in the remainder of this paper we will only consider constraint addition and retraction. Several algorithms have been proposed for solving DCSPs e.g Dynamic Backtracking for Dynamic Constraint Satisfaction Problems [4] and Solution Reuse in Dynamic Constraint Satisfaction Problems [5].

A Distributed Constraint Satisfaction Problem (DisCSP) is a CSP in which variables, domains and constraints are distributed among autonomous agents [6]. Formally, a DisCSP can be described as a four tuple Z = (X, D, C, A) where

– X, D and C remain as described in CSPs and
– A is a set of agents with the mapping assigning variables to agents

Agents are only aware of their local constraints and the inter-agent constraints they are involved in and do not have a global view of the problem due to privacy, security issues and communication costs [7]. Solving a DisCSP consist of finding an assignment of values to variables by the collective and coordinated action of these autonomous agents which communicate through message passing. A solution to a DisCSP is a compound assignment of values to all variables such that all constraints are satisfied.

Various algorithms have been proposed for solving DisCSPs e.g Asynchronous Backtracking algorithm (ABT) [8], Asynchronous weak-Commitment search Algorithm (AWCS) [9] and Distributed Breakout algorithm (DBA) [10]. In DisCSPs, the following assumptions are usually made: (i) There is one variable per agent (ii) Agents are aware of their neighbours and constraints they share with them, (iii) Message delays are finite though random and messages arrive in the order they are sent between two related agents [8] and we shall also be making these assumptions in this paper.

Many hard practical problems can be seen as DisCSPs. Most DisCSP approaches however assume that problems are static. This has a limitation for dynamic problems that evolve over time e.g timetabling shifts in a large hospital where availability of staff changes over time. In order to handle this type of problems, traditional DisCSP algorithms naively solve from scratch every time the problem changes which may be very expensive or inadequate, i.e. there may be a requirement for the solution to the new (changed) problem to remain close as possible to the original solution.

Distributed and Dynamic Constraint Satisfaction Problems (DDisCSPs) can be described as a five tuple (X,D,C,A,δ) where

– X, D, C and A remain as described in DisCSPs and
– δ is the change function which introduces changes at different time intervals

This definition is different from that of DisCSPs only in the introduction of the change function δ, which is a representation of changes in the problem over time [1]. DDisCSPs can be used to model problems which are distributed in nature and change over time.

Problem changes which have been widely modelled as a series of constraint additions and removals can be episodic(where changes occur after each problem has been solved) or occur while a problem is being solved. In this paper, we shall assume that changes shall be episodic.

Amongst the DDisCSP algorithms is the Dynamic Distributed Breakout Algorithm (DynDBA) [1] which is the dynamic version of DBA - a distributed local search algorithm inspired by the breakout algorithm of [11]. In DBA, agents assign values to their variables and communicate these values to neighbouring agents by means of messages. Messages passed between agents are in the form of *OK* and *Improve* messages. When agents discover inconsistencies they compute the best possible improvement to their violations and exchange it with neighbouring agents. Only the agent with the best possible improvement among neighbours is allowed to implement it. When an inconsistent state cannot be improved, i.e. a quasi local minimum is reached, the weights on violated constraints are increased [10], thus prioritising the satisfaction of these constraints.

In DynDBA, agents solve problems just like in the DBA algorithm but have the ability to react to changes continuously in each cycle with the aid of *pending lists* for holding new neighbours and messages.

In this paper we introduce our Dynamic Asynchronous Backtracking Algorithm (DynABT) which is based on the Asynchronous Backtracking Algorithm (ABT) [6] to handle DDisCSPs.

The remainder of this paper is structured as follows: section 2 describes ABT; next, section 3 introduces DynABT; this algorithm is evaluated in section 4 and; finally conclusions are presented in section 5.

2 Asynchronous Backtracking Algorithm (ABT)

Asynchronous Backtracking (ABT) is an asynchronous algorithm for DisCSPs in which agents act autonomously based on their view of the problem. ABT places a static ordering amongst agents and each agent maintains a list of higher priority agents and their values in a data structure known as the *agentview*. Constraints are directed between two agents: the *value-sending* agent(usually higher priority agent) and the *constraint-evaluating* agent(lower priority agent). The value-sending agents make their assignments and send them to their lower priority (constraint-evaluating) neighbours who try to make consistent value assignments. If a constraint-evaluating agent is unable to make a consistent assignment, it initiates backtracking by sending a *nogood* message to a higher priority agent, thus indicating that it should change its current value assignment. Agents keep a *nogood list* of backtrack messages and use this to guide the search. A solution is found if there is quiescence in the network while unsolvability is determined when an empty nogood is discovered. The correctness and completeness of ABT has been proven in [8].

ABT sends a lot of obsolete messages and uses a lot of space for storing nogoods. Therefore, various improvements to ABT have been proposed [12–15] which either reduce the number of obsolete messages or the space required for storing nogoods. In addition there is a version of ABT which uses just one nogood per domain value [15] which is of interest to us. This version uses the nogood recording scheme of Dynamic

Backtracking [16] when recording and resolving nogoods but maintains the static agent ordering of ABT. Thus, a nogood for an agent x_k with value a is represented in the form $x_i = b \cap x_j = c \Rightarrow x_k \neq a$, where x_i and x_j are neighbouring agents with values b and c. In the remainder of this paper, we will use ABT to refer to the version which keeps just one nogood per eliminated value.

3 DynABT

DynABT is an asynchronous, systematic algorithm for dynamic DisCSPs. Based on ABT, it repairs the existing solution when the problem changes. DynABT combines solution reuse, reasoning reuse and justifications where a justification for the removal of a value states the actual constraint causing the removal in the explanation set recorded for the removed value.

Like in ABT, DynABT agents maintain a list of higher priority agents and their values in their *agentview* and a list of values inconsistent with their *agentview* in the nogood store. Higher priority agents send their value assignments to lower priority agents in the form of *info messages*. When an *info message* is received, the agent updates its *agentview* and checks for consistency. When its value is inconsistent, the agent composes a nogood but, unlike ABT nogoods, these are coupled with a set of justifications (actual constraints causing the violations). A nogood in DynABT is now of the form $x_i = b \cap x_j = c\{C_1, ..C_n\} \Rightarrow x_k \neq a$, where x_k currently has value a. Thus, the justification included in the nogoods acts as a pointer to which nogoods should become obsolete when constraints are retracted. We shall call the ABT with this new form of nogood recording ABT^+.

In DynABT (see Algorithms 1 to 5), each agent initialises its variables, starts the search and solves the problem like in ABT. However agents monitor the system to see if there are any changes and if so, react appropriately. Problem changes are handled in a two phase manner namely the *Propagation phase* (see Algorithm 2) and the *Solving phase* (ABT^+). In the propagation phase, agents are informed of constraint addition/retraction and they promptly react to the situation by updating their constraint lists, neighbour lists, agentview and nogoods where necessary. After all changes have been propagated, the new problem is at a consistent starting point, the *canProceed* flag is set to true and the agents can move on to the *Solving phase* and solve the new problem in a way similar to the ABT algorithm.

Three new message types (*addConstraint*, *removeConstraint* and *adjustNogood*) are used in order to handle agent behaviour during the propagation phase. When an agent receives an *addConstraint* message, the agent updates its constraint and neighbour lists where necessary (see Algorithm 3). When a *removeConstraint* message is received the agent modifies its neighbour list by excluding neighbours that only share the excluded constraint from its neighbour list and removing them from its agentview. The constraint is then removed and the nogood store is updated by removing nogoods whose justification contains the retracted constraint (see algorithm 4).

When a constraint is removed, an *adjustNogood* message is broadcasted to agents that are not directly involved in this constraint. The agents receiving this message update their nogoods store by removing the nogoods containing the retracted constraint as part of its justification and returning the values to their domains (see Algorithm 5). This step ensures that values that have been invalidated by retracted constraints are returned and made available since the source of inconsistency is no longer present in the network. Performing these processes during the propagation stage ensures that the new problem starts at a consistent point before the search begins.

Algorithm 1 DynABT

$changes \leftarrow 0$; $changeBox \leftarrow empty$; $canProceed \leftarrow true$
ABT^+ (ABT with nogoods containing justifications)
repeat
  $changes \leftarrow monitorChanges$
  if (changes) **then**
    $canProceed \leftarrow false$
    PropagateChange(changeBox)
    current value $\leftarrow$ value from the last solution
    ABT^+()
  end if
until termination condition met

Algorithm 2 PropagateChanges

PropagateChange(changebox)
while $changeBox \neq empty \cap canProceed \leftarrow false$ **do**
  $con \leftarrow getChange$; $changeBox \leftarrow changeBox - con$
  Switch (con.msgType)
  con.removeConstraint : removeConstraint(con);
  con.addConstraint : includeConstraint(con);
  con.adjustNogood : incoherentConstraint(con);
end while

Algorithm 3 IncludeConstraint

IncludeConstraint(con)
$newCons \leftarrow$ con.getConstraint()
add new neighbours in newCons to neighbour list
$constraintList \leftarrow constraintList \cup newCons$

Algorithm 4 ExcludeConstraint

ExcludeConstraint(con)
incoherentConstraint(con)
$constraint \leftarrow$ con.getConstraint()
Remove unique neighbours in constraint from neighbour list
Delete unique neighbours from agentView
Remove constraint from constraintlist

Algorithm 5 AdjustNogoods

IncoherentConstraint(con)
$constraint \leftarrow$ con.getConstraint()
for each nogood in nogoodstore **do**
  if $contains(nogood, constraint)$ **then**
    return eliminated value in nogood to domain
    remove nogood from nogoodStore
  end if
end for

3.1 Sample Execution

Figure 1a represents a DisCSP involving four agents(*a*, *b*, *c*, *d*) each with its own variable and domain values enclosed in brackets and having 3 *Not Equal* constraints ($C_1, C2, C3$) between them. Let us assume that the initial DisCSP was solved with the solution (*a = 1*, *b =0*, *c = 0*, *d = 0*) and the following nogoods were generated:

- Agent a : $(() \{C_1\} \Rightarrow a \neq 0)$
- Agent c : $((a = 1) \{C_2\} \Rightarrow c \neq 1)$
- Agent d : $((a = 1) \{C_3\} \Rightarrow d \neq 1)$

In Figure 1b, we assume that the solved problem has now changed and the constraint between a and d(C_3) has been retracted and a new constraint between c and d (C_4)has been added. At this stage, DynABT goes into the *propageChanges* mode in which agents c and d are informed of a new constraint between them and also agent a and d are made aware of the loss of the constraint between them. In addition to this set of messages, agents b and c are also sent adjustNogood messages, informing them of the loss of constraint C_3 and the need for them to adjust their nogoods if it is part of their justification sets. When these messages have been fully propagated (agent d will adjust its nogood and regain the value 0 back in its domain), the nogood store of the agents will now be in the form below:

- Agent a : $(() \{C_1\} \Rightarrow a \neq 0)$
- Agent c : $((a = 1) \{C_2\} \Rightarrow c \neq 1)$

The agents can now switch back to the solving mode because the problem is at a consistent starting point and the algorithm can now begin solving again. A new solution to the problem will be (*a = 1*, *b =0*, *c = 0*, *d = 1*) with d having to change its value to 1 in order for the new problem to be consistent.

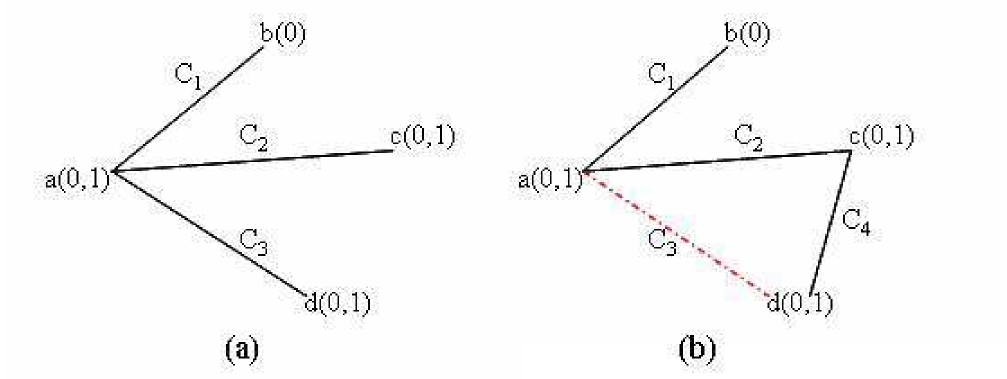

Fig. 1.

In our implementation, we have used a system agent for detecting quiescence just as been done in [15], in addition to this, we have also used it to communicate changes in the problem to the agents and also set the *canProceed* flag of agents to true when it determines that all propagation has been done. Completion of the propagation stage is determined in the following way: every time an agent receives any of the three messages (*addConstraint*, *removeConstraint* and *adjustNogood*) and performs the appropriate computation, the agent sends a dummy message back to the system agent indicating that it has received and treated a propagation message. The system agent can determine the total number of such messages to receive when all agents have received messages and acted on them in the *propagateChanges* and can therefore set the *canProceed* flag of all agents to true. This total number of messages can be calculated in the following way: Let x represents the number of constraints of a certain arity r added to the new problem and let N be the total number of agents in the network and y be the total number of constraint removed from the problem. The total messages to receive can be computed as $tot = (\sum(x_i * r_i)) + N * y$. In our implementation, we have reported these messages as part of the cost incured by DynABT.

3.2 Theoretical Properties

DynABT is sound, since whenever a solution is claimed, there is quiescence in the network. If there is quiescence in the network, it means that all agents have satisfied their constraints. If not all constraints have been satisfied, then there will be at least an agent unsatisfied with its current state and at least one violated constraint in the network. In this case, the agent involved would have sent at least a message to the culprit agent closest to it. This message is not obsolete and the culprit agent involved on receiving the message, will act on it and send out messages thus breaking our quiescence claim. It therefore follows that whenever there is quiescence in the network, agents are satisfied with their current state and whatever solution inferred is sound.

In the DynABT, agents update their nogood list when they receive *info messages* and evaluate constraints, during domain wipe out and also when changes are introduced. Nogoods are always generated in two ways: (1) when a constraint is violated because of an *info message* (this knowledge is explicitly enclosed in the constraint) and (2) When a domain wipe-out occurs and all nogoods are resolved into one. In essence, all the nogoods that can be generated are logical extension of the constraint network, therefore the empty nogood cannot be inferred if the network is satisfiable.

Also because every nogood discovered by an agent will always involve higher priority agents, which are eventually linked to the agent through the *addLink Message*, it follows that agents will not keep obsolete nogoods forever, since they will be informed of value changes by higher priority agents and thus update their nogood store, ensuring that the algorithm will terminate. We now need to show that when changes occur, these properties are still preserved.

When constraints are added to the problem, previous nogoods invalidating domain values remain consistent and since the nogood stores remain unchanged during constraint addition, these nogoods are preserved. Therefore when constraints are added to the problem, the soundness property of the algorithm is preserved.

When a constraint is retracted, nogoods are updated to exclude the retracted constraint and the associated values are returned to the agent's domain. If these values are still useless, this inconsistency will be rediscovered during search since they will violate constraints with some other agents and, therefore, solutions are not missed.
The *adjustNogoods* method ensures that all agents (whether participating in a retracted constraint or not) update their nogoods store and all nogoods containing retracted constraints as part of their justification are removed and the associated values returned to their domain.

Because retraction triggers the updating of the nogood store in a cautious manner in which nogoods are quickly forgotten but can be rediscovered if necessary during search, DynABT is complete and does terminate

4 Experimental Evaluation

In order to evaluate DynABT, ABT, DynABT and DynDBA have been implemented in a simulated environment. The implementations of DynABT and ABT use the Max Degree heuristic.

Two sets of experiments were conducted using both randomly generated problems and graph colouring problems: (i) Comparing DynABT with ABT; (ii) Comparing DynABT and DynDBA. In all our comparisons with DynDBA, we have modified the DynDBA algorithm to make it react to changes episodically and also improved it by increasing the weight of a newly added constraint within a neighbourhood to the maximum constraint weight within that neighbourhood. This encourages DynDBA to satisfy the newly added constraints quicker.

In all our experiments, we have introduced a rate of change δ as a percentage of the total constraints/edges in the problem ($\delta \in \{2, 6, 32\}$). These changes [1] were made to be uniform between restriction and retraction. For example, if 4 changes are introduced, 2 are constraint additions and 2 are constraint retractions, thus ensuring that the overall constraint density remains unchanged.

In our experiments with randomly generated problems, we used with parameters (n, d, p_1, p_2) where n = number of variables = 30, d = domain size = 10, p_1 = density = 0.2, p_2 = tightness with values 0.1 - 0.9 step of 0.1. The range of tighness 0.1 - 0.4 contains solvable problems, 0.5 contains a mixture of both solvable and unsolvable(52% - 48%) and tightness 0.6 - 0.9 problems are unsolvable. For the unsolvable region, stability cannot be measured, as there is no solution to the problem. Each problem was solved and the solution obtained was kept for future reuse. Constraint changes were introduced and the new problem was solved. In all, 100 trials were made for each tightness value and a total of 1800 problems (900 original problems + 900 changed problems) were solved for each rate of change.

For our evaluation with graph colouring problem, we generated graph colouring problems with nodes = 100, d = 3 and degree k (4.1 - 4.9 step 0.1). These problems ranges from solvable through phase transition to unsolvable problems. In all, 100 trials were made per degree and a total of 1800 problems (900 original problems + 900 changed problems) were solved for each rate of change.

We measured the number of messages sent, Concurrent Constraint Checks (CCC) as defined[2] in [17] and the solution stability. For solution stability, we measure the total distance between successive solutions when both exists (the number of variables which get different values in both solutions). All the results reported are the mean and median of the observed parameters and we have only presented results of observed parameters when resolving. We also measured CPU time (not reported here) and it correlated to the trends observed with messages and concurrent constraint checks.

4.1 Comparison with ABT

For random problems, results obtained in table 1 show a reduction in the cost incured when a new problem is solved using previous solution, i.e. DynABT significantly outperforms ABT on small and intermediate changes while on large problems, ABT peforms better than DynABT: this is due to the fact that the new problem is substantially different from the previous one because of the quantity of changes involved and also because DynABT incurs more cost as changes increase during the propagation phase.

For Graph Colouring problems, the results obtained in table 2 are mixed between DynABT and ABT. With small and intermediate changes DynABT performs better than ABT on messages and CCC in the solvable region betwee 4.1 - 4.4 and the Phase

[1] all constraints/edges have equal probability of being selected for retraction
[2] Cost of transmitting a message is zero in our implementation

Table 1. DynABT vs ABT.

Random Problems												
t	Avg Messages		Avg CCC		Avg Stability		Median Msgs		Median CCC		Median Stability	
	DynABT	ABT	DynABT	ABT	DynABT	ABT	DynABT	ABT	DynABT	ABT	DynABT	ABT
	Density 0.2, changes 2(%)											
0.1	152	**106**	40	**34**	**0.09**	0.99	151	**104**	38	**32**	0	0
0.2	152	**133**	**43**	58	**0.27**	3.59	151	**131**	**42**	54	**0**	3
0.3	**161**	221	**53**	163	**0.96**	7.53	**156**	198	**49**	117	**0**	6
0.4	**283**	1262	**140**	1019	**2.41**	10.99	**189**	650	**61**	437	**0**	10
0.5	**45868**	83071	**25129**	45882	**5.96**	11.90	**8566**	63325	**3998**	35946	**1**	7
0.6	**6879**	27778	**2442**	10069	-	-	**1194**	23200	**360**	8439	-	-
0.7	**1482**	12204	**373**	3413	-	-	**65**	10134	**14**	2916	-	-
0.8	**495**	5301	**103**	1193	-	-	**65**	4769	**13**	992	-	-
0.9	**92**	1964	**15**	413	-	-	**65**	1776	**12**	386	-	-
	Constraint Changes 6(%)											
0.1	280	**106**	44	**33**	**0.32**	2.35	279	**105**	42	**31**	**0**	2
0.2	281	**132**	**53**	59	**0.76**	6.17	279	**130**	**52**	56	**1**	5
0.3	293	**208**	**80**	153	**1.65**	11.33	285	**192**	**65**	109	**1**	11
0.4	**547**	1084	**270**	946	**5.28**	15.69	**327**	732	**81**	453	**2**	16
0.5	**83746**	86068	**44929**	47981	**12.39**	16.58	**56532**	60330	**30847**	35965	**14**	19
0.6	**17797**	27951	**6075**	10168	-	-	**12986**	24057	**4462**	8552	-	-
0.7	**3952**	12487	**971**	3619	-	-	**1771**	10337	**356**	3179	-	-
0.8	**1320**	5052	**251**	1116	-	-	**193**	4389	**34**	997	-	-
0.9	**351**	1944	**58**	427	-	-	**193**	1570	**33**	375	-	-
	Constraint Changes 32(%)											
0.1	986	**105**	77	**34**	**1.5**	5.73	984	**104**	74	**33**	**1**	6
0.2	991	**134**	120	**62**	**3.92**	12.95	988	**131**	119	**57**	**4**	13
0.3	1023	**201**	198	**119**	**7.15**	18.72	1005	**193**	170	**105**	**7**	19
0.4	1670	**987**	**718**	1020	**13.87**	21.89	1214	**644**	**331**	403	**14**	22
0.5	138979	**120871**	84532	**80589**	**23.13**	24.03	91770	**87106**	58606	**55678**	**24**	25
0.6	39080	**38525**	**15397**	17323	-	-	**31594**	32527	**11341**	14132	-	-
0.7	**10788**	13216	**2903**	4286	-	-	**8312**	10413	**2124**	3598	-	-
0.8	**3733**	5255	**741**	1329	-	-	**3298**	4508	**582**	1103	-	-
0.9	1929	**1781**	**329**	406	-	-	**1436**	1479	**207**	372	-	-

transition region of 4.5, while in the unsolvable region from 4.6 - 4.9, ABT performs better. This behaviour is due to the fact that more cost is incured during the propagation stage of DynABT, when agents are modifying their nogoods before the new search starts. With large changes, ABT performs better than DynABT. However, with both problems, DynABT outperforms ABT on solution stability for all degrees of changes, which suggests that reusing solution, improves stability.

4.2 Comparison with DynDBA

In order to compare DynABT with DynDBA the latter algorithm was allowed a cut-off of at least 50% more cycles than DynABT when solving a problem because DynDBA is a two-phased algorithm(it takes an agent two cycles to make a value change compared to DynABT in which values can be changed in one cycle).

For our comparison with DynDBA, we have only presented results for solvable problems for both algorithms because DynDBA being an incomplete algorithm, cannot determine a problem is unsovable. For our Comparison with DynDBA on random problems, results from table 3 shows that DynABT outperforms DynDBA in terms of messages sent and concurrent constraint checks.

Table 2. DynABT vs ABT.

Graph Colouring problems												
deg	Avg Messages		Avg CCC		Avg Stability		Median Msgs		Median CCC		Median Stability	
	DynABT	ABT	DynABT	ABT	DynABT	ABT	DynABT	ABT	DynABT	ABT	DynABT	ABT
	Density 0.2, changes 2(%)											
4.1	**1107**	1556	**141**	691	**8.31**	26.41	**937**	1358	**72**	564	**3**	25
4.2	**3908**	4987	**668**	1544	**20.37**	33.5	**1133**	3737	**87**	1310	**4**	39
4.3	**6598**	9089	**1422**	2411	**20.67**	31.13	**1278**	5199	**91**	1777	**5**	34
4.4	**15173**	22450	**2682**	5246	**36.29**	52.42	**7921**	18607	**742**	4187	**54**	62
4.5	**73051**	84269	**14949**	17969	**16.33**	39.94	**1377**	25921	**95**	6199	**2**	44
4.6	177267	**170370**	39241	**35248**	-	-	205565	**188612**	42964	**35633**	-	-
4.7	**96149**	129664	**20166**	26006	-	-	**108513**	123541	**20957**	24526	-	-
4.8	**112436**	129634	**24159**	26271	-	-	**123294**	125345	**24648**	24999	-	-
4.9	**85917**	99803	**17027**	19782	-	-	98236	**92975**	**18319**	19395	-	-
	Constraint Changes 6(%)											
4.1	2234	**1657**	225	648	**17.1**	36.58	1953	**1452**	**100**	398	**14**	37
4.2	6779	**6775**	1230	1919	**31.07**	46.3	**3685**	4380	**381**	1343	**36**	51
4.3	**9601**	10861	1868	2706	**33.07**	46.53	**5753**	7574	**947**	1841	**39**	48
4.4	**26788**	29292	4915	6708	**52.37**	61.86	**16683**	20158	**3205**	4904	**61**	66
4.5	**75872**	77264	14510	16812	**34.54**	49.08	**9213**	22545	**1433**	5540	**36**	58
4.6	159183	**139501**	32148	28569	-	-	107662	**94313**	20425	**18939**	-	-
4.7	171735	**163474**	34317	32337	-	-	152500	**150572**	30855	**30847**	-	-
4.8	163775	**140212**	32315	28694	-	-	154104	**127484**	28749	**25688**	-	-
4.9	131197	**117767**	23826	22452	-	-	118803	**106355**	21776	**20325**	-	-
	Constraint Changes 32(%)											
4.1	12408	**6993**	1305	2050	**42.18**	56.23	8295	**2689**	**339**	1007	**43**	57
4.2	19706	**13531**	2826	3613	**50.90**	60.03	12011	**5855**	**895**	2175	**52**	61
4.3	53886	**45458**	9984	10446	**51.11**	60.43	15515	**10114**	**1890**	3139	**52**	62
4.4	77395	**68769**	14230	16216	**57.16**	65.76	23579	**20745**	**3707**	4843	**59**	67
4.5	183139	**158873**	35975	34616	**55.48**	62.03	**53710**	66718	**11238**	15459	**56**	62
4.6	267015	**215258**	51557	44995	-	-	139890	**109348**	25002	**23862**	-	-
4.7	345887	**302694**	62451	61463	-	-	272262	**211478**	51923	**43559**	-	-
4.8	375866	**344063**	71460	68691	-	-	215529	**177088**	40890	**35357**	-	-
4.9	285956	**236785**	49152	45935	-	-	242464	**193879**	40676	**37038**	-	-

DynDBA outperforms DynABT in terms of solution stability. Our take on this is the fact that DynDBA with its min-conflict heuristic helps the algorithm find a stable solution. However it was shown that both algorithms depreciate in terms of solution stability as changes increase. This could be due to the fact that as more changes are introduced, the difference between the inital problem and the new problem is more pronounced, therefore new solutions are needed.

Table 4 presents results of our experiment on Graph Colouring problems. DynABT also outperforms DynDBA on messages and CCCs while DynDBA performs better on solution stability.

5 Summary and Conclusions

We have presented DynABT, an asynchronous, systematic search algorithm for DDisCSPs. An empirical comparison between DynABT and ABT on dynamic random problems shows a significant reduction in computational effort and a substantial gain in solution stability. Comparison with ABT however produces mixed results with DynABT outperforming ABT on messages and concurrent constraint checks on problems with small changes. With Intermediate changes, ABT performs better than DynABT

Table 3. DynABT vs DynDBA

Random Problems, density 0.2												
t	Avg Msgs		Avg CCC		Avg Stability		Median Msgs		Median CCC		Median Stability	
	DynABT	DynDBA	DynABT	DynDBA	DynABT	DynDBA	DynABT	DynDBA	DynABT	DynDBA	DynABT	DynDBA
	Density 0.2, changes 2(%)											
0.1	**152**	403	**40**	121	**0.09**	0.1	**151**	401	**38**	120	0	0
0.2	**152**	402	**43**	125	0.27	**0.21**	**151**	402	**42**	120	0	0
0.3	**161**	485	**53**	151	0.96	**0.45**	**156**	404	**49**	120	0	0
0.4	**283**	2291	**140**	735	2.41	**1.99**	**189**	408	**61**	130	0	0
0.5	**17941**	42675	**10442**	12515	5.96	**1.79**	**225**	408	**72**	120	1	**0**
	Constraint Changes 6(%)											
0.1	**280**	415	**44**	121	0.32	**0.28**	**279**	410	**42**	120	0	0
0.2	**281**	430	**53**	131	0.76	**0.63**	**279**	411	**52**	120	1	1
0.3	**293**	582	**80**	187	1.65	**0.97**	**285**	416	**65**	125	1	1
0.4	**547**	4497	**270**	1511	5.28	**4.31**	**327**	520	**81**	180	2	2
0.5	**51970**	209324	**28851**	64411	12.39	**3.7**	**15807**	20235	8783	**6380**	14	**2**
	Constraint Changes 32(%)											
0.1	986	**523**	**77**	139	1.5	**1.25**	984	**457**	**74**	120	1	1
0.2	991	**673**	**120**	193	3.92	**2.51**	988	**476**	**119**	150	4	**2**
0.3	**1023**	1387	**198**	426	7.15	**4.32**	1005	**811**	**170**	270	7	**4**
0.4	**1670**	8097	**718**	2691	13.87	**10.34**	**1214**	4624	**331**	1540	14	**10**
0.5	**122384**	608790	**75763**	193657	23.13	**18.33**	**63667**	586656	**38303**	172440	24	**15**

in the unsolvable region while DynABT outperforms ABT in terms of stability for all categories of problem changes. Experimental results also show that DynABT requires less messages and constraint checks than DynDBA. However, the latter produces more stable solutions.

We have also shown that both DynABT and DynDBA cope well with problems where the rate of change is small but, as the number of changes increases, performance decreases. This is unsurprising since, with a high rate of change, the new problem is substantially different from the previous one. Future work will investigate other ways of improving the performance of DynABT in terms of solution stability.

References

1. Mailler, R.: Comparing two approaches to dynamic, distributed constraint satisfaction. In: AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems, New York, NY, USA, ACM Press (2005) 1049–1056
2. Dechter, A., Dechter, R.: Belief maintenance in dynamic constraint networks. In: Seventh National Conference on Artificial Intelligence (AAAI), St. Paul, MN, USA (1988) 37–42
3. Verfaillie, G., Jussien, N.: Constraint solving in uncertain and dynamic environments: A survey. Constraints **10**(3) (2005) 253–281
4. Verfaillie, G., Schiex, T.: Dynamic backtracking for dynamic constraint satisfaction problems. In: Proceedings of the ECAI'94 Workshop on Constraint Satisfaction Issues Raised by Practical Applications, Amsterdam, The Netherlands. (1994) 1–8
5. Verfaillie, G., Schiex, T.: Solution reuse in dynamic constraint satisfaction problems. In: National Conference on Artificial Intelligence. (1994) 307–312
6. : Distributed constraint satisfaction: foundations of cooperation in multi-agent systems. Springer-Verlag (2001)

Table 4. DynABT vs DynDBA

Graph Colouring Problems												
t	Avg Msgs		Avg CCC		Avg Stability		Median Msgs		Median CCC		Median Stability	
	DynABT	DynDBA	DynABT	DynDBA	DynABT	DynDBA	DynABT	DynDBA	DynABT	DynDBA	DynABT	DynDBA
	Density 0.2, changes 2(%)											
4.1	**1107**	42481	**141**	1553	**8.31**	14.82	**937**	3650	**72**	120	3	3
4.2	**3908**	116632	**668**	3989	20.37	**14.41**	**1133**	1449	87	**60**	4	**2**
4.3	**6598**	171520	**1422**	5923	20.67	**16.17**	**1278**	2439	91	**90**	5	**3**
4.4	**15201**	208067	**2697**	6967	36.29293	**21.82**	**7907**	10592	717	**360**	54	**5**
4.5	**14217**	449156	**2742**	14685	16.32558	**10.03**	**1355**	4414	**81**	135	2	2
	Constraint Changes 6(%)											
4.1	**2234**	70844	**225**	2539	**17.1**	24.43	**1953**	30074	**100**	1080	**14**	17
4.2	**6779**	213290	**1230**	7511	**31.07**	31.37	**3685**	75800	**381**	2700	36	**26**
4.3	**9497**	305758	**1856**	10498	**33.07**	33.21	**5640**	109373	**914**	3810	39	**30**
4.4	**23708**	462501	**4305**	15322	52.37	**35.71**	**15843**	235621	**3114**	8025	61	**35**
4.5	**19548**	528244	**3661**	17237	34.54	**24.46**	**7334**	429983	**983**	14340	36	**19**
	Constraint Changes 32(%)											
4.1	**9617**	196977	**804**	6761	**42.18**	46.9	**8255**	98926	**331**	3552	**43**	50
4.2	**15210**	434704	**1725**	15188	50.9	**48.96**	**11885**	234240	**850**	8145	52	**51**
4.3	**22483**	445826	**3677**	14872	**51.11**	54.05	**14419**	273725	**1601**	9540	**52**	54
4.4	**33362**	581442	**5372**	18767	57.16	**54.81**	**21107**	474626	**3121**	14634	59	**56**
4.5	**54785**	1090806	**10535**	35821	**55.48**	56.43	**30829**	980126	**6647**	32670	**56**	58

7. Meisels, A.: Distributed constraints satisfaction algorithms, performance, communication. In: Tenth International Conference on Principles and Practice of Constraint Programming, Canada (2004) 161–166
8. Yokoo, M., Durfee, E.H., Ishida, T., Kuwabara, K.: The distributed constraint satisfaction for formalizing distributed problem solving. In Proc. of the 12th.DCS (1992) 614–621
9. Yokoo, M.: Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In Montanari, U., Rossi, F., eds.: Proceedings of the First International Conference on Principles and Practice of Constraint Programming (CP-95). Volume 976 of Lecture Notes in Computer Science., Springer (1995) 88–102
10. Yokoo, M., Hirayama, K.: Distributed breakout algorithm for solving distributed constraint satisfaction problems. In: Second International Conference on Multiagent Systems. (1996) 401408
11. Morris, P.: The breakout method for escaping from local minima. In: AAAI. (1993) 40–45
12. Zivan, R., Meisels, A.: Dynamic ordering for asynchronous backtracking on discsps. Constraints **11**(2-3) (2006) 179–197
13. M.C.Silaghi, D.Sam-Haroud, B.: Generalized dynamic ordering for asynchronous backtracking on discsps. In: Second Asia-Pacific Conference on Intelligent Agent Technology (IAT),, ,Maebashi, Japan (2001)
14. M.C.Silaghi, D.Sam-Haroud, B.: Hybridizing abt and awc into a polynomial space, complete protocol with reordering. TR4 EPFL-TR-01/36, Swiss Federal Institute of Technology, Lussane, Switzerland (May 2001)
15. Bessière, C., Maestre, A., Brito, I., Meseguer, P.: Asynchronous backtracking without adding links: a new member in the abt family. Artif. Intell. **161**(1-2) (2005) 7–24
16. Ginsberg, M.L.: Dynamic backtracking. Journal of Artificial Intelligence Research **1** (1993) 25–46
17. Meisels, A., Kaplansky, E., Razgon, I., Zivan, R.: Comparing performance of distributed constraints processing algorithms. In: Proceedings of the AAMAS-2002 Workshop on Distributed Constraint Reasoning, Bologna (July 2002) 86–93