Concurrent Forward Bounding for DCOPs

Arnon Netzer, Amnon Meisels, and Alon Grubshtein

Department of Computer Science
Ben-Gurion University of the Negev
Beer-Sheva, 84-105, Israel
{netzerar,am,alongrub}@cs.bgu.ac.il

Abstract. A new search algorithm for solving distributed constraint optimization (DCOP) problems is presented. The proposed algorithm performs concurrent search on non intersecting parts of the global search space, using multiple search processes. Each Search Process uses synchronous forward bounding in its sub-space, to prune the remaining search space. Concurrent Forward Bounding (ConcFB) dynamically spawns new Search Processes in parts of the search space which have higher chances to produce tighter bounds on the cost of the global solution.

An extensive experimental evaluation is presented comparing ConcFB to the leading DCOP algorithms. ConcFB significantly outperforms other DCOP algorithms in both run time and communication load.

A new measure of concurrency of computation for DCOP algorithms is introduced the Concurrency Factor (CF). Each feature of ConcFB is shown to increase its concurrency. Unlike other DCOP algorithms, the CF of ConcFB increases with problem density and difficulty.

1 Introduction

Distributed Constraint Optimization Problems (DCOPs) form a powerful framework for distributed problem solving that has a wide range of application in Multi Agent Systems, [1] [2]. In a typical DCOP agents need to assign values in a way that would minimize (or maximize) the sum of the resulting constraint costs. A DCOP algorithm searches the combinatorial assignments space, in order to find the combination of assignments that will produce this minimum.

Several DCOP algorithms where introduced over the last year. These include SyncBB [3], ADOPT [2], ODPOP [4], BnB-ADOPT [5] and Asynchronous Forward-Bounding [6].

The present paper proposes a new DCOP algorithm: Concurrent Forward Bounding (ConcFB). The proposed algorithm is based on two main ideas: it performs multiple search processes on disjoint parts of the search space [7], each search process performs Forward Bounding in order to achieve early pruning of the search space[6]. The algorithm incorporates dynamic splitting that focuses the search on parts of the search space that are more likely to produce lower bounds.

ConcFB benefits from the pruning power of forward bounding, and complements it with concurrency of several search processes. When the performance of the proposed...
ConcFB algorithm is compared to the leading DCOP algorithms, it turns out to be superior in terms of both measures - run time and network load; The ConcFB algorithm is described in section 3, and its correctness proof is in section 4. Section 5 presents extensive experimental evaluation of ConcFB and its comparison evaluation with the leading DCOP algorithms. The paper conclude in 6 in which Concurrency Factor (CF), which is an objective quantitative measurement of the degree of concurrency of a DCOP algorithm, is introduced.

2 Distributed Constraints Optimization

A DCOP is a tuple \( \langle A, \mathcal{X}, D, R \rangle \). \( A \) is a finite set of agents \( A_1, A_2, ..., A_n. \) \( \mathcal{X} \) is a finite set of variables \( X_1, X_2, ..., X_m. \) Each variable is held by a single agent (an agent may hold more than one variable). \( D \) is a set of domains \( D_1, D_2, ..., D_m. \) Each domain \( D_i \) contains a finite set of values which can be assigned to variable \( X_i. \) \( R \) is a set of relations (constraints). Each constraint \( C \in R \) defines a non-negative cost for every possible value combination of a set of variables, and is of the form \( C : D_{i_1} \times D_{i_2} \times \ldots \times D_{i_k} \rightarrow R^+. \) A binary constraint refers to exactly two variables and is of the form \( C_{ij} : D_i \times D_j \rightarrow R^+. \) A binary DCOP is a DCOP in which all constraints are binary. An assignment is a pair including a variable, and a value from that variable’s domain. A partial assignment (PA) is a set of assignments, in which each variable appears at most once. \( \text{vars}(PA) \) is the set of all variables that appear in PA, \( \text{vars}(PA) = \{ X_i \mid \exists a \in D_i \land (X_i, a) \in PA \}. \) A constraint \( C \in R \) of the form \( C : D_{i_1} \times D_{i_2} \times \ldots \times D_{i_k} \rightarrow R^+ \) is applicable to PA if \( X_{i_1}, X_{i_2}, \ldots, X_{i_k} \in \text{vars}(PA). \) The cost of a partial assignment PA is the sum of all applicable constraints to PA over the assignments in PA. A full assignment is a partial assignment that includes all the variables (\( \text{vars(PA)} = \mathcal{X} \)). A solution is a full assignment of minimal cost.

In the duration of this paper we assume all constraints to depend on exactly two variables (binary constraints), and one variable per agent. The terms variable and agent are used interchangeably.

3 Concurrent Forward Bounding

ConcFB performs multiple concurrent search processes on disjoint parts of the search space. A simple example of producing such disjoint subspaces would be to randomly choose a variable \( X_i, \) take each of its possible assignments \( d \in D_i \) to be the only allowed assignment for a given search process, and set each search process to scan the entire sub-space that has a specific value \( d \) assigned to \( X_i. \) This will result in a number of search processes which equals the domain size of \( X_i. \) We will later show how search processes can be dynamically spawned in “interesting” parts of the global search space.

Each Search Process (SP) maintains a global order of the agents. Each SP also maintains a Current Partial Assignment (CPA), holding all the assignments of variables that are assigned values in the SP. Each agent holds a designated data structure for every SP that passes through it, maintaining information about the specific SP. In particular
each variable holds a current partial domain of values that are not assigned yet, to the specific SP.

When an agent assigns a value to its variable on a CPA that represent a given SP, it sends forward a Comp-LB message to every neighbor (e.g. constrained agent) that is after it in the global order of the specific SP. The Comp-LB message requests the receiving agent to compute a Lower Bound (LB) on the partial assignment on the CPA. When an agent receives all of the Lower Bound estimations for a given assignment in a given SP, it adds them to the cost of the CPA, and compares the resulting cost to the Upper Bound. If the result is larger or equal to the Upper Bound, the agent tries to assign a different value. If no such assignment exists, it backtracks. If the result is smaller than the Upper Bound, the CPA, with the var assignment and the LBs collected so far, is sent to the next variable in the global order of the specific SP.

When a variable receives a Comp-LB request for a given CPA, it calculates the cost of each of its assignments for the given CPA. The Lower Bound for the Variable is taken to be the minimum over the costs:

$$LB^X_j = \min_{d \in \text{Dom}(X_j)} \sum_{X_i \in \text{CPA}} \text{Cost}(X^a_i, d)$$

Where $X^a_i$ is the assignment of variable $X_i$, and the lower bound is computed by $X^j_i$.

Search processes can split, dynamically generating new SPs. A split is generated in response to SPLIT message. An agent that receives a SPLIT request checks the amount of assignments left in the current domain of the specific SP. If more than one assignment is left, then the current domain is split in half, one half remain in the original SP, and the other is used as the current domain of a newly created SP. If the amount of assignments left in the current domain is not enough for split, a SPLIT request is sent to the next agent in the global order of the SP, requesting it to split. Agents must keep track of all split SPs and their originating SP. A backtrack can be performed only after all split SPs and originating SP have exhausted their search of the sub space.

A global variables order used by all SPs will result in an uneven work distribution between agents, The present realization uses random order of variables for each SP.

Each time a new global upper bound is found, it is broadcast to all agents. When an agent receives an Upper Bound message that is lower than it’s current value, it disregards the incoming message. Every message carries the best Upper Bound known to its sender.

### 3.1 Algorithm Description

Figures 1 and 2 presents the pseudocode of every agent running ConcFB. There are six types of messages used in ConcFB:

- CPA - Contains the SP_ID and the Current Partial Assignment for that given SP. The CPA contains an $LB\_List$ of lower bounds collected so far. An agent receiving a CPA message tries to extend it with its own assignment.
- BackTrack_CPA - Informs of inability to extend the CPA.
- Comp_LB - A copy of a CPA for a given SP. Requesting to compute an estimate of the lower bound for the given CPA.
- Split - An agent receiving a Split message will attempt to split the SP.
- UpperBound - An agent will replace its upper bound with the new one if the new one is smaller.

The algorithm starts with the First Agent creating the initial Search Processes (routine Initialize_SPs() in Figure 1). Each SP has a domain of 1 value out of the First Agent Domain (line 3). The variable order is randomized for each SP (line 4). Each SP calls assign_CPA() to start the Search Process.

Upon receiving a CPA message, an agent creates a local structure for the SP on the received CPA. This data structure includes the current domain for the SP, the CPA received and the LB_List received. If the LB_List contains an LB previously obtained by the agent, this LB is removed from the LB_List (line 4 in Received_CPA()). An origin_SP.splitList is created, in which the newly received SP serves as both the origin_SP and an item in the splitList (line 5). If the number of assignments in the CPA is large enough, a Split message is sent to the SP originator (line 7). The agent then tries to assign its own value on the CPA by calling Assign_CPA() (line 8).

Upon receiving a Backtrack_CPA message the agent locate its local structure SP by using the SP_ID. It then removes its assignment for the relevant SP (line 2 of Received_BacktrackCPA()), and removes the local_cost from the SP.CPA.cost. If the current domain of the relevant SP is empty, or if the agent has no constraints with any of the unassigned agents of the relevant SP.CPA, the agent tries to backtrack (line 5). Otherwise, it tries to assign a new value by calling Assign_CPA() (line 7).

Any LB received via a Reported_LB message, from an agent that already appears in SP.LB_List replaces the existing one. Otherwise the new LB is added to SP.LB_List (Received_ReprotedLB() line 2). The agent waits for all Reported_LB messages to arrive. If all needed LB arrived (line 3), then, if the cost of the CPA plus the sum of all contributions in SP.LB_List is lower than the UpperBound (line 4), the CPA is sent to the agent who is next in the global order of the specific SP. Otherwise the assignment is no good and the agent calls the Received_BacktrackCPA() (line 6).

When an agent receives a Split message and the SP cannot be split, the message is sent to the next agent in the global order (line 8 in Received_Split()).

When an agent tries to assign a value, it first checks whether the domain of the relevant SP was not exhausted (line 1 in Assign_CPA()). If it was, the agent Backtracks. Otherwise, the assignment with the lowest cost with respect to the relevant SP.CPA is selected (line 2). If the cost of SP.CPA with the added assignment and the lower bounds collected for this CPA is not smaller than the known Upper Bound, the agent backtracks (line 5). If the cost of SP.CPA is smaller than the known Upper Bound and SP.CPA has a full assignment, a new Upper Bound has been found. The new Upper Bound is sent to all agents and a Backtrack is performed on the relevant SP (lines 8 and 9). If it is not a full assignment, a Collect_LB message is sent to all unassigned neighbors.

When an agent attempts to Backtrack, it first removes the relevant SP from its origin_SP.splitSet (line 2 of Backtrack()) and removes all data structures relevant to this SP. If the origin_SP.splitSet of the relevant SP is not empty then a BackTrack cannot be performed yet, and the routine ends. If origin_SP.splitSet is empty, then it is time to Backtrack. The origin_SP.splitSet is removed, and a Backtrack_CPA message
ConcFB_Main:
1. \textit{done} \leftarrow \text{false}
2. if (IA)
3. \textbf{initialize\_SPs()}
4. while (not \textit{done})
5. pop \textit{msg}
6. if (\textit{msg.ub} < \textit{UB})
7. \textit{UB} = \textit{msg.ub}
8. \textbf{switch} \textit{msg.type}
9. CPA \leftarrow \textit{Received\_CPA}(\textit{msg.CPA, msg.LB\_List})
10. Backtrack\_CPA \leftarrow \textit{Received\_Backtrack\_CPA}(\textit{msg.SP\_ID})
11. Comp\_LB \leftarrow \textit{Received\_Comp\_LB}(\textit{msg.CPA})
12. Reported\_LB \leftarrow \textit{Received\_Reported\_LB}(\textit{msg.SP\_ID, msg.LB, msg.sender})
13. Split \leftarrow \textit{Received\_Split}(\textit{msg.SP\_ID})
14. Terminate \leftarrow \textit{done} = \text{true}

\textbf{Initialize\_SPs():}
1. for \textit{i} \leftarrow 1 \text{ to } \textit{domain.size}
2. create \textit{SP}_i
3. \textit{SP}_i.\text{domain} \leftarrow \textit{domain} [\textit{i}]
4. \textit{SP}_i.\text{Randomize\_Agents\_Order()}
5. Add \textit{SP}_i to rootSp.splitSet
6. Assign\_CPA(\textit{SP}_i)

\textbf{Received\_CPA}(\textit{CPA, LB\_List}):  
1. Create local \textit{SP} data structure
2. \textit{SP.CPA} \leftarrow \textit{CPA}
3. \textit{SP.LB\_List} \leftarrow \textit{LB\_List}
4. \textit{SP.LB\_List.remove}(current, \textit{ar})
5. Add \textit{SP} to \textit{SP.splitSet}
6. if (\textit{SP.CPA.nof\_Assigned\_Vars} > \text{Split Depth})
7. send \text{Split to } \textit{SP.initiator}
8. Assign\_CPA(\textit{SP})

\textbf{Received\_Backtrack\_CPA}(\textit{SP\_ID}):  
1. retrieve local \textit{SP} with \textit{SP\_ID}
2. \textit{SP.removeLast\_Assignment}
3. \textit{SP.CPA.cost} \leftarrow \text{LocalCost}
4. if (\textit{SP.current\_Domain.isEmpty} or no lower priority neighbors)
5. Backtrack()
6. else
7. Assign\_CPA(\textit{SP})

\textbf{Received\_Comp\_LB}(\textit{CPA}):  
1. \textit{LB} = \text{min}_{\text{d} \in \text{Domain}} \sum_{X \in \textit{SP.CPA}} \text{Cost}(X^*_a, \textit{d})
2. send \text{Reported\_LB} to \text{Calc\_LB originating variable}

Fig. 1. Pseudo code of ConcFB.
Received_ReportedLB($SP_{ID}, LB, neighbor$):
1. retrieve local $SP$ with $SP_{ID}$
2. $SP.LB_List[neighbor] = LB$
3. if (got ReportedLB from all unassigned neighbors)
4.   if ($SP.CPA.cost + SP.LB_List.sum < UpperBound$)
5.     send CPA($SP.CPA, SP.LB_List$) to next agent
6. else
7.   Received_BackTrackCPA($SP_{ID}$)

Received_Split($SP_{ID}$):
1. retrieve local $SP$ with $SP_{ID}$
2. if ($SP.currentDomain.size > 1$)
3.   create $newSP$
4.   $newSP.domain \leftarrow SP.splitDomain$
5.   Add $newSP$ to $SP.splitSet$
6.   Assign_CPA($newSP$)
7. else
8.   send Split to next agent

Assign_CPA($SP$):
1. if ($SP.Domain$ not empty)
2.   $SP.CPA \leftarrow$ best local assignment
3.   $SP.CPA.cost+ = LocalCost$
4.   if ($SP.CPA.cost + SP.LB_List.sum \geq UB$)
5.     Backtrack($SP$)
6. else
7.   if (is full($SP.CPA$))
8.     Found_new UB($SP.CPA.cost$)
9.     Backtrack($SP$)
10. else
11.   send Collect LB($SP.CPA$) to all unassigned neighbors
12. else
13.   Backtrack($SP$)

Backtrack($SP$):
1. $Origin.SP \leftarrow$ retrieve Origin $SP$ of $SP_{ID}$
1. remove $SP$ from $origin.SP.splitSet$
3. if ($origin.SP.splitSet.isEmpty$)
4.   remove $origin.SP.splitSet$
5. if ($ID == FirstAgent$)
6.   if ($SPList.isEmpty$)
7.     send Terminate to all Agents
8.   done $\leftarrow$ true
9.   solution $\leftarrow UB$
10. send Backtrack_CPA to last assignee

Fig. 2. Pseudo code continue.
is sent to the last assignee of the relevant SP (line 10). If \( \text{origin}_SP\_splitSet \) is empty, and this is the first global agent, then the algorithm stops. A Terminate message is sent to all agents, and the Upper Bound registered in the first global agent is the algorithm’s result.

### 3.2 Dynamic Splitting

Figure 3 presents an example of the CPA behavior for each of 4 Search Processes, a histogram of the number of assignments of the CPA in each variable, and the total number of CPA assignments, per SP. There are 14 variables with an average of 4 neighbors per variable, and a domain size of 4. Variable order was the same for all search processes, and no dynamic splitting was used.

It is clear from Figure 3 that most of the CPA assignments are in the middle part of the histogram. In other words, in the middle depth of the search space. It is also clear that the distribution of CPA assignments is not uniform among the search processes. In this example SP2 has about half the CPA assignments than any of the other SPs. This will result in SP2 exhausting its subspace much earlier than the other SPs, and lowering the concurrency of the algorithm. Spawning more SPs of the harder working SPs will even the work load between them, and insure that enough SPs are alive throughout the search.

![Fig. 3. CPA assignments per SP](image)

Figure 4 presents the histogram of the number of assignments of the CPA in the last 3 variables, and the number of Upper Bounds found per SP. As can be seen, the number of Upper Bounds found can vary significantly between SPs. In fact though SP1 and SP3 has the same total number of CPA assignments (Figure 3) SP1 has 8 times more Upper Bounds occurring. Spawning more SPs that are producing more Upper Bounds, that may enable a faster detection of Upper Bounds and a better pruning of the search space.
3.3 Random Variable Ordering

Figure 5 presents the distribution of constraints checks between the variables, when there is no random reordering of variables for each SP. The figure is the averaged result of 50 ConcFB runs over random problem with 14 variables, an average of 4 neighbors per variable, and domain size of 4. It is clear that the work load is not distributed evenly between variables. In fact most of the work is done by the variables with lower priority. This is due to the fact that in forward bounding algorithms most of the work is done by the computation of lower bounds in look ahead, and not in the CPA assignments.

Figure 6 shows the result of the same run, with random variable ordering for each Search Process. Figure 6 shows an even spread of work load between the variables (except for the first one which is first for all search processes). The highest number of CC per variable is about half in the random order scenario. Clearly a better distribution of work among the agents.
Fig. 6, total CC random variable ordering

4 Correctness of Conc-FB

To prove the correctness of a search algorithm for DCOP one needs to prove that it is sound, complete and that it terminates.

Theorem 1. ConcFB is sound

Proof. The only place in the algorithm that terminates the search and reports a solution is in Backtrack() line 8. In which the UB is reported as a solution. A new UB is created only in Assign_CPA() line 7, and only for CPAs that are fully assigned (line 6). Therefore, the solution necessarily represents the cost of a full assignment.  

Lemma 1. ConcFB traverses all sub-spaces

Proof. The search space is divided into subspaces in two functions - Initialize_SPs() and Receive_Split(). The domain of the first agent is split in Initialize_SPs() (line 3). Each of the resulting domains is assigned a Search Process that attempts to extend it to a full assignment.

In Receive_Split() The part of the domain of an agent within a given SP, is split. One resulting SP is the original SP. The new SP is searched by the resulting CPA (line 3). Hence all subspaces are being searched in Conc_FB.  

Theorem 2. ConcFB is Complete

Proof. To prove the completeness of ConcFB one must show that only CPAs that cannot be expanded to a full assignment with lower cost than the UB are pruned. In addition one must show that no SP on a subspace search is stopped before it is completed.

Pruning is done in 2 places in the algorithm. The first is in Assign_CPA() after a variable seeking its best assignment for extending a given CPA(line 2) discovers that adding the new assignment causes the CPA cost to be greater than the UB (line 3). this is clear case their where no solution can be found by extending the assignment of the CPA.

The second pruning place in the algorithm is in function Received_ReportedLB(). In this case after receiving lower bounds from all lower priority neighbors CPA.cost +
Accumulated \( \text{LB} < \text{UB} \). Clearly any full assignment extending the CPA will cost no less, and there is no need to try and extend this CPA.

Finally we insure that no CPA is backtracked before it is fully searched (or pruned) in line 2 of \( \text{Backtrack}() \), by making sure that all SPs that are trying to extend the CPA have completed (e.g. returned) before the backtrack can be performed.

Theorem 3. ConcFB Terminates

Proof. Since no assignments on a CPA is visited more than once during the search process, and since each CPA is either extended or backtracked at each step of the algorithm, and since the search space is finite, it is obvious that the algorithm will terminate.

5 Experimental Evaluation

Two measures of performances are routinely used to evaluate distributed DCOP algorithms. Runtime in the form of non-concurrent constraints checks (NCCCs), and network load in the form of the total number of messages sent.

The present evaluation compares ConcFB to two DCOP algorithms - BnB-ADOPT [5] and ODPOP [4].

For ODPOP the main computational operation is the comparison of combination of assignments, sent to each computing agent by its offspring on the pseudo tree [4]. This operation performed by each agent in order to find an assignment for itself that is compatible with all assignments of its ancestors on the pseudo-tree.

All experimental evaluation of run time, are given for the three algorithms in non concurrent logical operations. For ODPOP these are compatibility checks, and in BnB-ADOPT and ConcFB they are constrain checks.

BnB-Adopt was shown to be superior to ADOPT and to NCBB [5]. The present evaluation use BnB-Adopt with DP2 as a preprocessing phase for h vals [8], for reference the synchronous branch & bound algorithm (SBB) with best val assignment was used.

Experiments were performed on randomly generated DCOPs with 12 agents, domain size 3 and costs randomly assigned between 0 and 10000. The average number of neighbors varied between 2 and 5, all results where averaged over 50 randomly generated problems.

Figure 7 presents run time in terms of Non Concurrent Logic Operations (NCLO) of all 4 algorithms. For 5 neighbors ODPOP performed more then \( 10^7 \) NCLOs, and was left out of Figure 7 (3 orders of magnitude slower than all others).

The result of removing ODPOP from Figure 7 are in Figure 8. It is easy to see that the advantage of ConcFB over BnB-ADOPT grows significantly when the average number of neighbors grows.

The total number of messages for the same experiments follows a similar pattern and are shown in Figure 9. In this case BnB-ADOPT is much worse then all other algorithms. The high message count for BnB-ADOPT is to be expected, since BnB-ADOPT is using the same message passing and communication scheme of ADOPT [5], and ADOPT was shown to have a very high message count [9].
**Fig. 7.** NCLO - all 4 algorithms

**Fig. 8.** NCLO - ConcFB and BnB-ADOPT

**Fig. 9.** Number Of Messages
When BnB-ADOPT is left out, the picture becomes clearer. The total number of sent messages (Figure 10) of both ODPOP and ConcFB stays low as the density of the DCSP increases. ODPOP sends less messages than ConcFB. As the number of neighbors increases, the difference in number of messages between the two algorithms decreases. For 5 neighbors they send the same amount of messages.

Another experiment was done on MaxCSP problems. In this case, we randomly generated MaxCSP problems with 14 variables, domain size equals to 6 and an average of 4 neighbors (P1 = 2.85). Figure 11 shows the run time in NCCCs for each algorithm. ODPOP is not presented since it could not complete even P2=01 problems in reasonable time. The total number of messages for the same experiment are in Figure 12.

6 Discussion

A new algorithm for Distributed Constraints Optimization Problems (DCOPs) was presented. The algorithm performs concurrent search processes on disjoint parts of the DCOP. Each search process uses forward bounding on its search space to enhance its
The proposed algorithm - Concurrent Forward Bounding (ConcFB) was described and its correctness proven.

Two measures of performance were used in the extensive performance evaluation, and ConcFB was compared to the leading DCOP algorithms. ConcFB is four times faster than BnB-ADOPT for randomly generated DCOP in medium density and sends six times less messages. For the same problems it sends twice as many messages as ODPOP but its run time is ten times faster. As the difficulty of the random DCOP grows, the speed up of ConcFB compared to ODPOP grows exponentially. For DCOP with 5 neighbors, on the average, ODPOP does not complete its run in a few hours (Figure 7).

An interesting measure that can demonstrate the efficiency of computation during search of DCOP algorithms can be defined as follows. The Concurrency Factor (CF) is the ratio of the total amount of computation performed by all agents running an algorithm, to the amount of non-concurrent computation. The latter is the common measure of distributed asynchronous run-time [10].

In a completely synchronized algorithm (e.g. SBB) the total number of constrain checks is equal to the NCCC, hence $CF = 1$, on the other hand, the theoretical bound for the $CF$, is when all agents are completely unsynchronized, and there is no dependency between all computation. In this case $CF = N$ where $N$ is the number of agents. It is expected that in most DCOP problems, since agent computation is dependent upon other agents, the real values of $CF$ will be much lower than $N$.

In order to gain insight into the concurrent behavior of ConcFB one can compare the CF of different variants of the algorithm. 4 variant of the ConcFB algorithm were used, 1 - running only a single SP. 2 - running multiple SPs but no random variable ordering and no dynamic splitting. 3 - running multiple SPs with random variable ordering but no dynamic splitting. 4 - full featured ConcFB.

As is evident in Figure 13 the total computation (CC) is only lower for the full featured ConcFB. In contrast, the amount of non-concurrent computation (e.g. run time in NCCC) decreases as the CF increases.

Figure 14 shows the Concurrency Factor of all algorithms. The CF of the synchronous algorithm SBB is exactly 1. ODPOP uses a pseudo tree to achieve concurrency. This deteriorates as the number of neighbors grows. BnB-ADOPT is very concurrent by nature, in addition to the concurrency introduced by working on different
Fig. 13. behavior of ConcFB algorithm

branches of the pseudo tree. This explains the lowering of the $CF$ on BnB-ADOPT as the number of neighbors grows. In contrast to all other algorithms, the concurrency of ConcFB actually benefits from increase in the number of neighbors.

Fig. 14. Concurrency Factor - all algorithms

References