A Hybrid of Inference and Local Search for Distributed Combinatorial Optimization

Adrian Petcu and Boi Faltings
EPFL, Switzerland
{adrian.petcu, boi.faltings}@epfl.ch

Abstract

We present a new hybrid algorithm for local search in distributed combinatorial optimization. This method is a mix between classical local search methods in which nodes take decisions based only on local information, and full inference methods that guarantee completeness.

We propose LS-DPOP(k), a hybrid method that combines the advantages of both these approaches. LS-DPOP(k) is a utility propagation algorithm controlled by a parameter k which specifies the maximal allowable amount of inference. The maximal space requirements are exponential in this parameter. In the dense parts of the problem, where the required amount of inference exceeds this limit, the algorithm executes a local search procedure guided by as much inference as allowed by k. LS-DPOP(k) can be seen as a large neighborhood search, where exponential neighborhoods are rigorously determined according to problem structure, and polynomial efforts are spent for their complete exploration at each local search step.

We show the efficiency of this approach with experimental results from the distributed meeting scheduling domain.

1 Introduction

Constraint satisfaction and optimization are powerful paradigms that model a large range of tasks like scheduling, planning, optimal process control, etc. Traditionally, such problems were gathered into a single place, and a centralized algorithm was applied in order to find a solution. However, problems are sometimes naturally distributed, so Distributed Constraint Satisfaction (DisCSP) was formalized [15]. DisCSPs are divided between a set of agents, which have to communicate among themselves to solve it.

The DisCSP formalism has a number of advantages over its centralized counterpart. Centralized solving may be infeasible due to privacy and data integration problems. Dynamic systems are another reason: by the time we manage to centralize the problem, it has already changed.

To address distributed optimization, complete algorithms like ADOPT and DPOP have been recently introduced. ADOPT [12] is a backtracking based bound propagation mechanism. It operates completely decentralized, and asynchronously. Its downside is that it may require a very large number of messages, thus producing big communication overheads. *DPOP* [14] is a complete algorithm based on dynamic programming which generates a linear number of messages. However, *DPOP* is time and space exponential in the induced width of the problem. Therefore, in case the problems have high induced width, the messages generated in the high-width areas of the problem get large, and *DPOP* may be infeasible.

For such difficult problems, local search methods have been developed. These methods start with a random assignment, and then gradually improve it by applying incremental changes. Their advantage is that they require linear memory, and in many cases provide good solutions with a small amount of effort. However, the decisions taken are often myopic in the sense that they take into account only local information, thus getting stuck into local optima rather easily. Large neighborhood search [1] tries to overcome this problem by exploring a much larger set of neighboring states before moving to the next one. Dynamic programming has already been recognized as an efficient way to explore exponential size neighborhoods with a polynomial effort: see [6], or the hybrid technique of Kask and Dechter from [7] (see Section 6). In a parallel line of research, local search has also been combined with complete search algorithms, for example in [10] or [5].

For distributed environments, there are distributed local search methods like DSA ([8, 2]) / DBA([17]) for optimization, and DBA for satisfaction ([16]). To our knowledge, the concept of large neighborhoods has not been exploited in distributed environments.

We propose a distributed algorithm that combines the advantages of both these approaches. This method is a utility propagation algorithm controlled by a parameter k which specifies the maximal allowable amount of inference. The maximal space requirements are exponential in this param-

eter. In the dense parts of the problem, where the required amount of inference exceeds this limit, the algorithm executes a local search procedure guided by as much inference as allowed by k. If this parameter is equal to the induced width of the graph or larger, then the algorithm is full inference, therefore complete. Larger values of k are conjectured to produce better results.

We show the efficiency of this approach with experimental results from the distributed meeting scheduling domain.

The rest of this paper is structured as follows: Section 2 formally describes the optimization problem. Section 3 presents the *DPOP* algorithm from [14], upon which we will build our present work. Section 4 presents the hybrid optimization algorithm. Section 5 presents an experimental evaluation. Section 6 presents the relationship between this approach and existing work. Section 7 concludes.

2 Definitions and Notation

Definition 1 (DCOP) *A discrete* distributed constraint optimization problem *(DCOP) is a tuple* $< \mathcal{X}, \mathcal{D}, \mathcal{R} >$*:*

- $\mathcal{X} = \{X_1, ..., X_n\}$ *is a set of variables*
- $\mathcal{D} = \{d_1, ..., d_n\}$ *is a set of finite variable domains*
- $\mathcal{R} = \{r_1, ..., r_m\}$ *is a set of relations, where a relation* r_i *is any function with the scope* $(X_{i_1}, \cdots, X_{i_k})$*,* $r_i : d_{i_1} \times .. \times d_{i_k} \rightarrow \mathbb{R}$*, which denotes how much utility is assigned to each possible combination of values of the involved variables. Negative utilities mean cost.* [1]

DCOPs are multiagent instances of the *valued CSP* framework, where each variable and constraint is owned by an agent. The goal is to find a complete instantiation $\mathcal{X}^*$ for the variables X_i that *maximizes* the sum of utilities of individual relations.

A simplifying assumption [15] is that each agent controls a virtual agent for each one of the variables X_i that it owns. To simplify the notation, we use X_i to denote either the variable itself, or its (virtual) agent.We also assume here only unary and binary relations [2]

2.1 Depth-First Search Trees (DFS)

LS-DPOP uses a DFS traversal of the problem graph.

Definition 2 (DFS tree) *A DFS arrangement of a graph G is a rooted tree with the same nodes and edges as G and the property that adjacent nodes from the original graph fall in the same branch of the tree (e.g.* X_0 *and* X_{12} *in Figure 1).*

DFS trees have already been investigated as a means to boost the performance of optimization algorithms [12, 3, 14]. Due to the relative independence of nodes lying in different branches of the DFS tree, it is possible to perform search in parallel on independent branches, and then combine the results.

Figure 1 shows an example DFS tree that we shall refer to in the rest of this paper. We distinguish between *tree edges*, shown as solid lines (e.g. $X_1 - X_2$), and *back edges*, shown as dashed lines (e.g. $12 - 8$, $4 - 0$).

Definition 3 (DFS concepts) *Given a node* X_i*, we define:*

- **parent** P_i / **children** C_i*: these are the obvious definitions (e.g.* $P_4 = X_2$*,* $C_0 = \{X_1, X_8\}$*).*
- **pseudo-parents** PP_i *are* X_i*'s ancestors directly connected to* X_i *through back-edges (*$PP_5 = \{X_0\}$*).*
- **pseudo-children** PC_i *are* X_i*'s descendants directly connected to* X_i *through back-edges (*$PC_1 = \{X_4\}$*).*
- **separator** Sep_i *of* X_i*: ancestors of* X_i *which are directly connected with* X_i *or with descendants of* X_i *(e.g.* $Sep_{15} = \{X_9\}$ *and* $Sep_{11} = \{X_0, X_8, X_9, X_{10}\}$*). Removing the nodes in* Sep_i *completely disconnects the subtree rooted at* X_i *from the rest of the problem.*

Definition 4 (Induced width) *Given an ordering of the nodes in a graph, the width ([9, 3]) induced by the given ordering is obtained by processing the nodes in the reverse ordering, and connecting at each step all their neighboring predecessors in the ordering. The width induced by a DFS ordering equals the size of the largest separator of any node in the problem.*

3 DPOP: optimization based on inference

The basic utility propagation scheme *DPOP* has been introduced in [14]. *DPOP* is an instance of the general bucket elimination scheme from [3], which is adapted for the distributed case, and uses a DFS traversal of the problem graph as an ordering. *DPOP* has 3 phases:

Phase 1 - a **DFS traversal** of the graph is done using a distributed DFS algorithm, like in [14], which works for any graph requiring a linear number of messages. The outcome is that all nodes consistently label each other as parent/child or pseudoparent/pseudochild, and edges are identified as tree/back edges. The DFS tree serves as a communication structure for the other 2 phases of the algorithm:

[1]Hard constraints (that explicitly forbid/enforce certain value combinations) can be simulated with soft constraints by assigning $-\infty$ to disallowed tuples, and 0 to allowed tuples. Maximizing utility thus avoids assigning such value combinations to variables.

[2]However, all *-DPOP algorithms easily extend to non-binary constraints, with minor modifications to the DFS and UTIL phases.

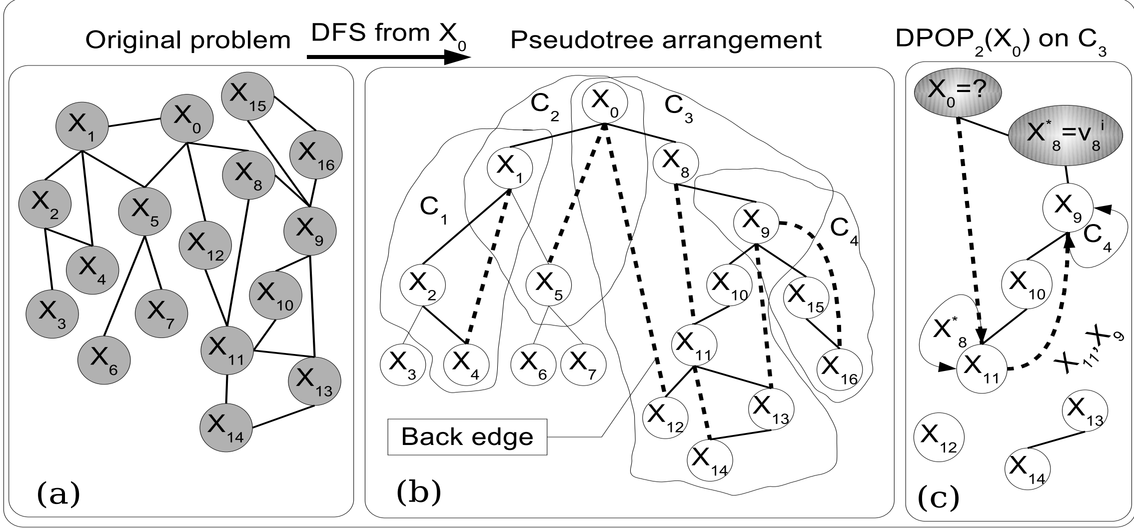

Figure 1. *A problem graph, one possible rooted DFS tree, and an execution detail of DPOP in C_3.*

UTIL messages (phase 2) travel bottom-up, and VALUE messages (phase 3) travel top down, only via tree-edges.

Phase 2 - **UTIL propagation**: the agents (starting from the leaves) send *UTIL* messages to their parents. The subtree of a node X_i can influence the rest of the problem only through X_i's separator, Sep_i. Therefore, a message contains the optimal utility obtained in the subtree for each instantiation of Sep_i. Thus, messages are size-exponential in the separator size (which is in turn bounded by the induced width).

Phase 3 - **VALUE propagation** top-down, initiated by the root, when phase 2 has finished. Each node determines its optimal value based on the computation from phase 2 and the *VALUE* message it has received from its parent. Then, it sends this value to its children through *VALUE* messages.

It has been proved in [14] that *DPOP* produces a linear number of messages. Its complexity lies in the size of the *UTIL* messages: the largest one is space-exponential in the width of the DFS ordering used.

4 *LS-DPOP* - local search/inference hybrid

We keep the basic utility propagation mechanism from DPOP, but we introduce a control parameter k which specifies the maximal amount of inference (maximal message dimensionality). In the dense parts of the problem, the exact propagation produces messages with more dimensions than this limit. In such cases, the algorithm executes a local search procedure guided by as much inference as allowed by k. The nodes whose processing by inference would exceed the k limit are the ones who execute the local search procedure. All other nodes execute the normal utility propagation protocol.

4.1 Detecting areas of high induced width

During the utility propagation procedure from *DPOP*, each node computes the *UTIL* message for its parent. In high width areas, some nodes have to send messages whose dimensionality exceeds k. In such cases, those nodes choose $dims - k$ dimensions of the message, mark them as *local search* dimensions, project them out of the outgoing message, and add these dimensions to the *context* of the message. Thus, the final dimensionality of the message is k (size limit observed). The dimensions to be marked as LS are chosen according to their level in the pseudotree. This is easy to determine for each node just by finding their position in the node's root path.

Example 1 (Detecting high width) *Consider C_3 in Figure 1(b). If we run LSDPOP with $k = 2$, then the messages $UTIL_{12}^{11}$ and $UTIL_{13}^{11}$ proceed normally as in DPOP, with $dims(UTIL_{12}^{11}) = \{11, 0\}$ and $dims(UTIL_{13}^{11}) = \{11, 9\}$. However, $dims(UTIL_{11}^{10}) = \{10, 0, 8, 9\}$, thus it exceeds $k = 2$. Therefore, X_{11} marks X_0 and X_8 (the 2 highest nodes in $dims(UTIL_{11}^{10})$) as LS nodes, projects them out of $UTIL_{11}^{10}$, and adds them to the context of $UTIL_{11}^{10}$. Thus, $dims(UTIL_{11}^{10}) = \{10, 9\}$ and $context(UTIL_{11}^{10}) = \{0^*, 8^*\}$.*

The propagation continues, and when the respective messages arrive at X_8 and X_0, they know that they must revert to local search. Note that in this example, X_0 is labeled as LS only in C_3, and not in C_2 (k not exceeded in C_2), so it will receive an exact message from C_2, and it will perform local search in C_3, together with X_8.

4.2 Local search in independent clusters

In the example of Figure 1, we notice that there are 4 independent parts which do not communicate between themselves except for some "frontier" nodes. These 4 cyclic subgraphs ($C_1 - C_4$), separated by the nodes X_0, X_1, X_9 can be explored separately for optimal solutions, and then the results assembled through the same *UTIL/VALUE* propagations. The advantage of this separation becomes apparent if we consider that many such separate problem components could be too complex to apply the exact *DPOP* propagation, and it may be needed to apply the local search mechanism. Then, it is obvious that by applying local search on each independent component C_t separately, we restrict the search space that needs to be explored from $d^{|LS|}$ to $d^{|LS(C_t)|}$, where $|LS|$ is the total number of *LS* nodes in the whole problem, and $|LS(C_t)|$ is the number of *LS* nodes in the component C_t. This, together with optimal combination of these local optima through *UTIL/VALUE* propagations, gives us a much better chance of finding a better overall local optima.

Frontier nodes are identified by considering the sizes of their separators: if the separator size is 1, the node is a frontier node. For example, X_1 is a frontier node for C_1 because $Sep_1 = \{X_0\}$.

If a frontier node is also designated a LS node in one of its subtrees, then that node will send its *UTIL* message to its parent only after having explored through local search the respective subtree. For example, assume C_4 hanging out from X_9 would be so complex as to require local search. Then X_9 would be marked as *LS*, and it would first participate in the local search in C_4, and only after a local optimum is reached there, would it start its propagation(s) in C_3. The utilities computed as the local optima for each of its values in C_4 are then added to the messages going through C_3. The process is logically equivalent to replacing C_4 with a unary constraint on X_9.

4.2.1 One local search step

In the subgraphs where local search is required, the *LS* nodes start by assigning themselves values. Then, we can run a DPOP-like propagation on the cyclic subgraph for each *LS* node X_n. For each propagation, we consider all LS nodes assigned with their current values, except for X_n. Such a propagation is just a simple variation of the $DPOP$ one, where instead of applying projections for all nodes, we execute slices for the nodes in the LS except X_n. Thus, X_n can determine how much utility each one of its values gives for the whole cyclic subgraph in which it is involved, *provided the other LS nodes maintain their current values*. It does so by joining all incoming $UTIL$ messages, and projecting out any other dimensions than itself. The result is a vector (one dimension) with the desired valuations. The value giving the maximal valuation can be proposed as the next value (in case it is different than the current value).

Figure 1.(c) shows an example execution of a local search step for X_0. All *LS* nodes send to their pseudochildren value messages, announcing their current values. The propagation starts normally from the leaves (X_{12} sends X_{11} a message with X_{11} and X_0 as dimensions). X_{11} performs normally the join between the messages it received from its children. Note that the message it received initially from X_{13} can be reused, since there is no link in that subtree with any LS node. Additionally, since X_8 is considered fixed at its present value, the relation $X_8 - X_{11}$ is logically replaced by a corresponding unary constraint on X_{11} (this is the slice of R^8_{11} along the current value of X_8, computed by X_{11}). The join is performed also with this induced unary constraint, and the relation R^{10}_{11}. $X11$ projects itself out of the join, and sends the message to X_{10}. The propagation continues until X_8, which performs the join $UTIL^8_9 \oplus R^0_8$. Instead of projecting itself out of the join to compute $UTIL^0_8$, X_8 performs a slice of this join along its current value (the one previously announced to X_{11}). It then sends $UTIL^0_8$ to X_0, who receives complete information about how much each of its values is worth for the whole C_3, provided X_8 keeps its current value.

X_0 can now compute $\Delta X_0 = UTIL^0_8 \perp_{X_0} -UTIL^0_8[X_0 = v_0]$, which is the maximal improvement that the whole C_3 can achieve if X_0 changes from its current value to the new optimal one, X_8 keeps its present value, and all the other nodes in C_3 change to their new optimal values.

X_0 also initiates a top-down propagation with itself as a *LS* node. It sends X_8 $UTIL^8_0$, with $dims(UTIL^8_0) = \{X_0, X_8\}$ (actually, this message is exactly R^8_0, since X_0 does not have anything else to join for sending to X_8. R^{12}_0 is taken into account by X_{12}, when sending out $UTIL^{11}_{12}$).

X_8 joins this message with $UTIL^8_9$, and performs a slice of this join, along its current value. The result is exactly the same vector as X_0 receives from X_8 as $UTIL^0_8$. What we achieved with the uniform propagation is thus the ability of X_8 to have the same information as X_0 about the possible improvements X_0 can make if X_8 keeps the current value.

After having run all propagations (with one of the *LS* nodes being allowed to change at the time), each *LS* node X_i can thus compute ΔX_j for each other *LS* node X_j in the same cyclic subgraph. In other words, each *LS* node X_i can thus compute the maximal improvements that each other *LS* node X_j can make, provided only X_j is allowed to change.

For the change itself, one can apply any policy known in current local search methods, and guide this policy by the Δs computed like this. The termination policy can be either a maximal number of cycles, or detection of local/global minima by detecting that all *LS* nodes have $\Delta = 0$.

Correctness In the current formulation, only the node with the highest improvement changes its value. Thus, the algorithm executes a hill climbing procedure for the nodes designated as LS, and exact inference for the rest, therefore it will reach a local maximum given by local maxima in each individual cyclic subgraph.

4.3 Large neighborhood exploration - analysis and complexity

Let us assume that in a cyclic subgraph C_t there are cc_t nodes designated as LS nodes, n_t total nodes, and m_t edges. The size of the neighborhood completely explored at each local search step is $cc_t \times d \times d^{n_t - cc_t}$ (for all values of each *LS* node, complete exploration of the *non-LS* nodes). The effort for each step consists of $2 \times (n_t - 1)$ *UTIL* messages sent for exploring C_t. The largest message is of size d^{k+1}. Thus, each step explores an exponential size neighborhood with a polynomial amount of effort.

Assume the termination policy for the local search process involves at most k local search steps. The whole process is then equivalent to exploring $k \times cc_t \times d \times d^{n_t - cc_t}$ neighboring states. An exhaustive search method would require at least as many messages (big communication overhead), while classical local search would not be guaranteed to completely explore this part of the search space.

5 Experimental evaluation

Our experiments were performed on distributed meeting scheduling problems. We modeled a realistic scenario, where a set of agents working for a large organization try to jointly find the best schedule for a set of meetings. The organization itself has a hierarchical structure: a tree with departments as nodes, and a set of agents working in each department. We generate meetings with high probability within departments, and with a lower probability between agents belonging to parent-child departments.

We model this problem as a DCOP following [11]. Specifically, each agent A_i has a set of variables X_i^j, one for each meeting it is involved in. Each such variable X_i^j is controlled only by the agent A_i, and represents the time when meeting j of agent A_i will start (X_i^j has time slots t_q as values). There is an equality constraint connecting the equivalent variables of all agents involved in a particular meeting (all agents must agree on a start time for their meeting). If a meeting has p participants, it is sufficient to create $p-1$ equality constraints that connect the corresponding variables in a chain (no need to fully connect them pairwise). Since an agent cannot participate in 2 meetings at the same time, there is an all-different constraint on all variables X_i^j belonging to the same agent.

Algorithm 1: *LSDPOP - local search/inference hybrid.*

LSDPOP($\mathcal{X}, \mathcal{D}, \mathcal{R}, k$): each agent X_i does:

***UTIL* propagation protocol**

1 wait for *UTIL* messages $(X_c, UTIL_c^i)$ from all children $X_c \in C_i$
2 **if** *any of* $UTIL_c^i$ *contains myself as LS node* **then** execute LS procedure
3 **else**
4     $JOIN_i^{P_i} = \left(\left(\bigoplus_{c \in C_i} UTIL_c^i \right) \oplus \left(\bigoplus_{c \in \{P_i \cup PP_i\}} R_i^c \right) \right)$
5   **if** X_i *is root* **then** start *VALUE* propagation
    else
6   **if** $|dims(JOIN_i^{P_i})| > k$ **then**
7   sort $dims(JOIN_i^{P_i})$ by root path (P_i is always last)
8   mark the first $|dims(JOIN_i^{P_i})| - k$ non-LS dimensions from the JOIN as LS, project them out and add them to the context of $JOIN_i^{P_i}$. P_i is always kept in.
9   compute $UTIL_i^{P_i} = JOIN_i^{P_i} \perp_{X_i}$ and send it to P_i
    end
end

Local search procedure

10 assign a value according to heuristic (can be random)
11 **while** *termination criteria for local search not met* **do**
12   send *VALUE*$(X_i \leftarrow current_value)$ messages to all PC_i
13   wait for all corresponding *UTIL* messages to arrive
14   join them, and slice through $(X_i \leftarrow current_value)$; store
end
get and store in $agent_view$ all *VALUE* messages $(X_c \leftarrow v_c^*)$
15 $v_i^* \leftarrow argmax_{X_i} \left(JOIN_i^{P_i}[v(P_i), v(PP_i)] \right)$
16 Send *VALUE*$(X_i \leftarrow v_i^*)$ to all C_i and PC_i

***VALUE* propagation**$(X_c \leftarrow v_c)$

17 **if** *sending node* X_c *is pseudoparent* **then**
18   perform slice $R_i^k[X_c = v_c]$ and join it with *UTIL* messages from children
19   project self out of this join, add $X_c \leftarrow v_c$ to the context of the message and send it to parent
end
20 get and store in $agent_view$ all *VALUE* messages $(X_c \leftarrow v_c^*)$
21 $v_i^* \leftarrow argmax_{X_i} \left(JOIN_i^{P_i}[v(P_i), v(PP_i)] \right)$
22 Send *VALUE*$(X_i \leftarrow v_i^*)$ to all C_i and PC_i

We model the utility that each agent A_i assigns to each meeting M_j at each particular time $t_q \in dom(X_i^j)$ by imposing unary constraints on the variables X_i^j; each such constraint is a vector private to A_i, and denotes how much utility A_i associates with starting meeting M_j at each time t_q. The objective is to find a schedule s.t. the overall utility is maximized.

We have run 2 series of experiments with random problems generated as specified before. In the first part, we generated "easy" problems, such that they can be solved by the complete algorithm as well, in order to see how far from the global optima the local search method is. The problems had induced width 8, and the domain size was 8, meaning the largest message in the complete algorithm has $8^8 \approx 16.5M$ values. These problems are quite close to the feasibility limit for a complete algorithm.

The results of these experiments are presented in Table 1. Each row represents an execution with an increasing bound k. The columns represent (in order): the k bound, *LS#* is the total number of nodes executing the local search procedure, *%Non-LS* is the percentage of nodes executing the normal propagation, *Cycles* is the number of independent subgraphs identified, *Avg LS/Non-LS nodes per cycle* is the average number of LS/non-LS nodes in a single component, *Sol %off* is the distance from the optimal solution in percent, and *Effort/step* is an upper bound on the total amount of data transmitted within an independent component, for one local search step.

We have run the algorithm with increasing k, and noticed relatively small increases in solution quality (percent off the true optimum decreases slowly) and exponential increases of the amount of effort spent for each local search step.

We notice that small values of k are already producing good solutions, with relatively low effort. We explain this by the fact that even small values of k allow for a large percentage of nodes to execute the exact propagation, and thus at each local search step, a large exponential neighborhood is explored. For example, imposing $k = 1$ (first row in Table 1) still leaves on the average almost 70% of the nodes to execute the exact propagation. On the average, in a subgraph, 13 *non-LS* nodes adjust optimally to the values of the 6 *LS* nodes, which is equivalent to exploring 8^{13} neighboring states at each *LS* step.

The second sets of experiments involved much larger and more difficult instances of the same meeting scheduling problems. In this case, the problems were generated with 200 agents, 498 variables and 1405 constraints. The induced width was 20, making for a 8^{20} maximal message size, which renders complete methods completely infeasible. We ran again *LSDPOP* with increasing k, and noticed a similar behavior: a large percentage of nodes execute exact propagation even for small k, and solution quality improves slowly with increasing k. We conjecture that these results are close to the true optimum.

6 Related Work

The nodes involved in the local search process can be thought of as *cycle cutset nodes* [4, 3]. From this perspective, there are a number of similar existing approaches.

Kask and Dechter present in [7] a method of combining a local search algorithm (GSAT) with inference. That method is formulated for constraint satisfaction problems, in a centralized setting. A subset of the problem nodes are given as cycle cutset nodes, and local search is performed on this subset. For each instantiation of the cutset nodes, a tree inference algorithm is applied to the rest of the problem. The differences between these methods are manyfold. First, our method is distributed, and is defined for optimization, not satisfaction. Second, the set of nodes that perform local search is identified at runtime (not given a priori). Third, we allow for inference with maximal width greater than 1, controlled by k. Finally, we separate the problem in distinct cyclic subgraphs which are explored separately, and the subsolutions are aggregated in a distributed fashion.

Petcu and Faltings present in [13] a distributed cycle cutset optimization method. The idea of isolating independent cyclic subgraphs appears there, too, but unfortunately there is no efficient method presented for identifying cycle cutsets nodes, nor for isolating independent cyclic subgraphs. Here, the DFS traversal of the graph is an excellent way to achieve both goals. There, exhaustive search is performed on the cycle cutset variables, as opposed to local search/propagation here. The synchronization problems between cycles from that method are solved here by simply making each node that borders 2 cyclic subgraphs wait for complete exploration of all its subtrees before sending its message to its parent.

7 Conclusions and future work

We have presented the first approach to large neighborhood search in distributed optimization. Exponential neighborhoods are rigorously determined according to problem structure, and polynomial efforts are spent for their complete exploration at each local search step.

The algorithm explores independent parts of the problem simultaneously and asynchronously, and then combines the results, all in a distributed fashion. The experimental results show that this approach gives good results for low width, practically sized dynamical optimization problems. For loose problems, most of the search space is optimally explored, and only small, tightly connected components are explored by local search. This increases the chance that the algorithm avoids some of the local optima, especially for loose problems.

k	LS#	%Non-LS	Cycles	Avg LS/cycle	Avg non-LS/cycle	Sol %off	Effort/step
1	68	68	11	6	$13 \rightarrow d^{13}$	10.86	640 $(O(d^2))$
2	39	81	9	4	$19 \rightarrow d^{19}$	10.62	3072$(O(d^3))$
3	25	88	8	3	$23 \rightarrow d^{23}$	9.71	20480$(O(d^4))$
4	15	93	6	2	$33 \rightarrow d^{33}$	9.3	131072$(O(d^5))$
5	5	97	2	2	$105 \rightarrow d^{105}$	8.25	786432$(O(d^6))$
6	2	99	1	2	$214 \rightarrow d^{214}$	7.26	4194304$(O(d^7))$
∞	0	100	0	0	$216 \rightarrow d^{216}$	0.0	$(O(d^8))$

Table 1. LSDPOP experiments: 100 agents, 59 meetings, 199 variables, 514 constraints, width 8

k	LS nodes	%Non-LS	Cycles	Avg LS/cycle	Avg non-LS/cycle	Solution	Effort/step
1	194	61	10	19	$30 \rightarrow d^{30}$	7910.0	4032$(O(d^2))$
2	131	73	10	13	$36 \rightarrow d^{36}$	7946.0	23040$(O(d^3))$
3	96	80	9	10	$44 \rightarrow d^{44}$	7964.0	139264$(O(d^4))$
4	73	85	9	8	$47 \rightarrow d^{47}$	7980.0	884736$(O(d^5))$
5	58	88	9	6	$48 \rightarrow d^{48}$	8021.0	6029312$(O(d^6))$

Table 2. LSDPOP experiments: 200 agents, 498 variables, 1405 constraints, width 20

For future work we plan to experiment with several different value switching policies (like simultaneous switches by several variables or allowing non-improving switches) and different termination policies.

References

[1] R. K. Ahuja, O. Ergun, J. B. Orlin, and A. P. Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Appl. Math.*, 123(1-3):75–102, 2002.

[2] M. Arshad and M. C. Silaghi. Distributed simulated annealing and comparison to DSA. In *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems, IOS Press, "Frontiers in Artificial Intelligence"*, 2004.

[3] R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[4] R. Dechter and Y. E. Fattah. Topological parameters for time-space tradeoff. *Artificial Intelligence*, 125(1-2):93–118, 2001.

[5] C. Eisenberg. *Distributed Constraint Satisfaction For Coordinating And Integrating A Large-Scale, Heterogeneous Enterprise*. Phd. thesis no. 2817, Swiss Federal Institute of Technology (EPFL), Lausanne (Switzerland), September 2003.

[6] O. Ergun and J. Orlin. Dynamic programming methodologies in very large scale neighborhood search applied to the traveling salesman problem. Technical Report 4463-03, MIT, Sloan School of Management, Dec. 2004.

[7] K. Kask and R. Dechter. A graph-based method for improving GSAT. In *AAAI/IAAI, Vol. 1*, pages 350–355, 1996.

[8] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.

[9] T. Kloks. *Treewidth, Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer, 1994.

[10] J. Lever. A local search/constraint propagation hybrid for a network routing problem. In *FLAIRS Conference*, 2004.

[11] R. T. Maheswaran, M. Tambe, E. Bowring, J. P. Pearce, and P. Varakantham. Taking DCOP to the real world: Efficient complete solutions for distributed multi-event scheduling. In *AAMAS-04*, 2004.

[12] P. J. Modi, W. M. Shen, and M. Tambe. An asynchronous complete method for distributed constraint optimization. In *Proc. AAMAS*, 2003.

[13] A. Petcu and B. Faltings. A distributed, complete method for multi-agent constraint optimization. In *CP 2004 - Fifth International Workshop on Distributed Constraint Reasoning (DCR2004)*, Toronto, Canada, Sep 2004.

[14] A. Petcu and B. Faltings. DPOP: A scalable method for multiagent constraint optimization. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence, IJCAI-05*, pages 266–271, Edinburgh, Scotland, Aug 2005.

[15] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *International Conference on Distributed Computing Systems*, pages 614–621, 1992.

[16] M. Yokoo and K. Hirayama. Distributed breakout algorithm for solving distributed constraint satisfaction problems. In V. Lesser, editor, *Proceedings of the First International Conference on Multi–Agent Systems*. MIT Press, 1995.

[17] W. Zhang and L. Wittenburg. Distributed breakout algorithm for distributed constraint optimization problems - DBArelax. In *Proceedings of the International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS-03)*, Melbourne, Australia, 2003.