Distributed Task Scheduling Subject To Arbitrary Constraints

Justin Jackson ∗ Anouck Girard ∗∗

∗ PhD Candidate, University of Michigan, Ann Arbor, MI 48109 USA
(e-mail: jjjack@umich.edu)

∗∗ Assistant Professor, University of Michigan, Ann Arbor, MI 48109 USA

Abstract: This paper presents a distributed scheduling problem and a distributed algorithm to solve it. The problem is motivated by military missions where tasks are assigned to various vehicles and tasks must be scheduled to satisfy constraints between them. The distributed scheduling algorithm is capable of finding a solution in the presence of arbitrary scheduling constraints, e.g., logical or temporal constraints. These types of missions present the challenge of planning using computer systems that are distributed in the sense of information and authority. This motivates the development of distributed planning algorithms that can overcome these challenges. The distributed scheduling algorithm used here is demonstrated on an example involving two unmanned air vehicles and two unmanned ground vehicles and is shown to be correct and complete.

Keywords: Distributed, scheduling algorithms, control, constraints, agents, communication

1. INTRODUCTION

Consider the following motivational example. Two unmanned air vehicles (UAVs) and two unmanned ground vehicles (UGVs) are tasked to prosecute either of two targets. Each vehicle has only local knowledge of the structure of the communication network. In this scenario, the communication topology is limited as depicted in Figure 1. Actual causes of such situations may include range limitations, terrain, and heterogeneous communication protocols. These limitations motivate the development of planning algorithms that can operate in the presence of an arbitrary communication networks.

![Fig. 1. UAV and UGV example.](image)

The example mission in Figure 1 requires a schedule where either one of the two targets are destroyed, but not both. To achieve this, a UAV must track the chosen target while a UGV attacks it. The tracking vehicle must be in communication with the attacking vehicle so that the attack can be coordinated. The assignment of tracking and attacking to appropriate vehicles has been made using the method presented in Jackson et al. [2011]. This is an example of a distributed system. While Lynch [1996], Lamport [1978] and Mullender [1993] define distributed systems by physical separation, the idea that we use is the following: 1) no agent has complete information describing the full system, or 2) no agent has complete authority over the actions of the other agents. The motivating example was chosen because it allows us to illustrate distributed scheduling in the presence of logical and temporal constraints on a problem of tractable size.

1.1 Original Contributions

The primary contributions of this work are as follows.

1) The presentation of the problem of distributed task scheduling subject to arbitrary constraints.

2) An adaptation of a distributed backtracking method to solve the scheduling problem.

Distributed constrained scheduling is important in situations where multiple agents must schedule the tasks that are related by constraints. Constraints may be logical or temporal in nature, e.g., perform one task before another, or perform one task after another. Distributed constrained scheduling presents the following challenges: 1) the problem size scales exponentially in the number of tasks being scheduled, and 2) the constraints considered are arbitrary predicates. The Distributed Non-Sequential Backtracking (DNSB) method is used to solve this problem. This method requires only local information about the communication network topology and the constrained tasks. The DNSB algorithm is correct and complete. That is, it eventually finds a solution or outputs that there is no solution.
1.2 Literature Review

Constraint satisfaction is well studied and distributed constraint satisfaction has been studied for many years in centralized settings by Russell and Norvig [2010] and Minton et al. [1990], and in distributed settings by Yokoo et al. [1992], Yokoo [1995].

Robustness in distributed systems has also been studied by Lynch [1996], Lamport [2001], Chandra et al. [2007]. Distributed databases have been analyzed and implemented by Burrows [2006]. These systems require scalable protocols that are robust to process failure as demonstrated in Burrows [2006].

Assignment and scheduling have been studied and applied in academic settings such as in Lin and Kernighan [1971], Laguna et al. [1995] and practical implementations such as Burrows [2006], CAEN HPC Group [2007]. Military mission planning problems are often formulated as scheduling and assignment problems as in Faied et al. [2009].

1.3 Overview

Section 2 introduces the notation that will be used throughout this paper; Section 3 describes the distributed scheduling problem of this paper; Section 4 develops the technical approach that is used in the solution method; Section 5 describes the solution method; Section 6 illustrates the solution method on the example used throughout this paper; Section 7 discusses conclusions and future work; correctness and completeness proofs are presented in the appendix of Section 8.

2. NOTATION AND PRELIMINARIES

The set of tasks is

\[ T = \{t_1, \ldots, t_{N_T}\}, \]

where \( N_T > 0 \) is the number of tasks. For the example of Section 1, the set of tasks is

\[ T = \{t_1, t_2, t_3, t_4\}, \]

and \( N_T = 4 \). The tasks physically correspond to \( t_1 \equiv \text{track target 1} \); \( t_2 \equiv \text{attack target 1} \); \( t_3 \equiv \text{track target 2} \); and \( t_4 \equiv \text{attack target 2} \).

The set of agents is

\[ A = \{a_1, \ldots, a_{N_A}\}, \]

where \( N_A > 0 \) is the number of agents. For the example of Section 1, the set of agents is

\[ A = \{a_1, a_2, a_3, a_4\}, \]

and \( N_A = 4 \). The agents physically correspond to \( a_1 \equiv \text{UGV1} \); \( a_2 \equiv \text{UAV1} \); \( a_3 \equiv \text{UAV2} \); and \( a_4 \equiv \text{UGV2} \).

A task assignment is defined as a mapping from the set of tasks to the set of agents and is

\[ TA : T \to A. \]

That is, every agent knows which tasks it is assigned to. For the example of Section 1, the task assignment is

\[ TA = \{(t_1, a_1), (t_2, a_2), (t_3, a_3), (t_4, a_3)\}. \]

The duration of a task is defined as a function that maps from \( TA \) to the integers,

\[ D : TA \to \mathbb{N}. \]

Tasks are assumed to have known duration. For the example of Section 1, \( D(t, a) = 1 \) for all \((t, a) \in TA \). That is, tasks have unit duration.

A task schedule is a mapping from the set of tasks \( T \) to a finite subset of the set of integers \( N_{max} \subseteq \mathbb{N} \). That is,

\[ TS : T \to N_{max}. \]

where \( N_{max} = \{0, \ldots, s_{max}\} \) and \( s_{max} \), the scheduling horizon, is an integer that is known a priori. We refer to the discrete integers as time slots. In practice, the scheduling horizon should be chosen to be an upper bound on the mission time. For the example of Section 1, \( s_{max} = 10 \). A schedule maps tasks to start times. That is, given the set of tasks and a schedule, operating on the set of tasks with the schedule gives the start time of each of the tasks. A sub-schedule is a restriction of a schedule to a subset of tasks.

The zero start time is reserved and indicates that a task is not performed. That is, \( TS(t) = 0 \) where \( t \in T \), indicates that \( t \) is not performed. Note that all tasks are assigned, but not necessarily performed. An example schedule for the example of Section 1 is

\[ TS_1 = \{(t_1, 6), (t_2, 6), (t_3, 5), (t_4, 5)\}. \]

Informally, scheduling constraints are used to characterize the allowable schedules. Scheduling constraints are formally defined as functions that map from the set of schedules to the set \{0, 1\}. The number of constraints is \( N_{cl} > 0 \). These constraints are defined as

\[ p_m : N_{max} \to \{0, 1\}, m = 1, \ldots, N_{cl}. \]

If \( p_m(TS) = 1 \) we say that the constraint is satisfied by the schedule, otherwise the constraint is violated. The set of tasks that are involved in a constraint is called a cluster. The number of task clusters is \( N_{cl} > 0 \). The task clusters are defined as

\[ T_{m} \equiv T, m = 1, \ldots, N_{cl}. \]

Task \( t_i \) is said to be consistent if for every \( m \) where \( t_i \in T_m \), \( p_m(TS) = 1 \). The constraints for the example of Section 1 are

\[ p_1(TS) = [TS(t_2) > 0] \land TS(t_1) = TS(t_2)], \]

\[ p_2(TS) = [TS(t_2) > 0] \lor (TS(t_3) > 0) \land [TS(t_3) = 0] \land [TS(t_3) = 0]], \]

\[ p_3(TS) = [TS(t_3) > 0] \land [TS(t_3) = TS(t_3)], \]

and the corresponding task clusters are \( T_1 = \{t_1, t_2\} \), \( T_2 = \{t_2, t_3\} \), and \( T_3 = \{t_3, t_4\} \). Constraint (12) means that if target 1 is attacked, then it should be attacked at or after time slot 5, and it should be tracked by a ground vehicle at the same time. Constraint (13) means that either target 1 should be attacked or target 2 should be attacked, but not both. Constraint (14) means that if target 2 is attacked, then it should be attacked after time equal 5, and it should be tracked by a ground vehicle at the same time.
A cluster union is defined for each task $t_i$ and is the set of tasks with which task $t_i$ shares a cluster. Formally a cluster union is,

$$C_i = \{t_k \mid 3m \leq N_{cl} : t_i \in \mathcal{T}_m \text{ and } t_k \in \mathcal{T}_m\}, i = 1, \ldots, N_t.$$  

(15)

The cluster unions for each of the tasks in the example of Section 1 are, $C_1 = \{t_2\}; C_2 = \{t_1, t_3\}; C_3 = \{t_2, t_4\};$ and $C_4 = \{t_3\}.$

Definition: Feasible schedule: A schedule $T S$ is called feasible if $\forall t_i \in \mathcal{T}, t_i$ is consistent.

The schedule,

$$T S_2 = \{(t_1, 0), (t_2, 0), (t_3, 5), (t_4, 5)\},$$  

(16)

is feasible. The schedule in (9) is not, this is due to the violation of constraint $p_2$ in (13). Note that requiring that all tasks be consistent is equivalent to requiring that a schedule satisfy all constraints.

We use the following standard notions from graph theory. An (undirected) graph is a pair $(\mathcal{V}, \mathcal{E})$ of vertices and edges such that each edge is a couple of vertices. A graph is called complete if and only if every couple of vertices is an edge. For the graph $(\mathcal{V}, \mathcal{E}),$ if $\mathcal{V}' \subseteq \mathcal{V},$ the subgraph induced by restriction to $\mathcal{V}'$, denoted $(\mathcal{V}, \mathcal{E})|_{\mathcal{V}'}$, is the graph $(\mathcal{V}', \mathcal{E}')$, where

$$\mathcal{E}' = \{v_1, v_2\} \in \mathcal{E} \mid v_1 \in \mathcal{V}' \text{ and } v_2 \in \mathcal{V}'\}. $$  

(17)

In other words, the induced subgraph is obtained by retaining only vertices in $\mathcal{V}'$ and the edges connecting them. The distance between two vertices $v, w \in \mathcal{V}$ is $d(v, w)$ and represents the number of edges that must be traversed to move from $v$ to $w$ across the graph. The diameter of a graph $G = (\mathcal{V}, \mathcal{E})$ is,

$$\text{diam}(G) = \max_{v, w \in \mathcal{V}} d(v, w).$$  

(18)

The neighborhood of a vertex $v \in \mathcal{V}$ is the set $\mathcal{N}_v = \{w \in \mathcal{V} \mid v \in \mathcal{E}\}.\text{The agents in (3) have communication capability described by an undirected, connected communication graph,}$

$$G_c = (\mathcal{A}, \mathcal{E}_c).$$  

(19)

There is an edge between two agents if and only if they are able to communicate directly with each other. The type of communication assumed here is acknowledgement-based, where each agent knows when communication is established with another agent. The edge set of the communication graph for the example of Section 1 is,

$$\mathcal{E}_c = \{(a_1, a_2), (a_2, a_3), (a_3, a_4)\}. $$  

(20)

For this work, a task assignment is assumed to satisfy the following:

$$\left(\mathcal{A}, \mathcal{E}_c\right)|_{T A(\mathcal{T}_m)} \text{ is complete, } m = 1, \ldots, N_{cl}. $$  

(21)

Physically, (21) means that if two agents are assigned to tasks that both belong to a cluster, those agents can communicate. This becomes important when attempting to find a feasible schedule, as communication is necessary to evaluate the constraints and solve the problem of finding a feasible schedule in a distributed manner.

3. PROBLEM DEFINITION

Each agent $a_j \in \mathcal{A}$ is assumed to know the following data:

(1) $t_i : T A(t_i) = a_j,$
(2) $\mathcal{T}_m$ and $p_m \text{ s.t. } t_i \in \mathcal{T}_m,$
(3) $N_{a_j},$
(4) $t_i : (t_i, a_k) \in TA \text{ and } a_k \in N_{a_j},$
(5) $D (\text{the duration function}),$
(6) $s_{\max}.$

where $i, l = 1, \ldots, N_t$ and $m = 1, \ldots, N_{cl}.$

The problem discussed here is for the agents in $\mathcal{A}$ to collectively find a feasible task schedule $T S,$ using only the available data together with direct communication between neighbors by (19). This is distributed task scheduling subject to arbitrary constraints. The number of possible schedules is $O((s_{\max} + 1)^N_t).$ For the example of Section 1 there are $11^4 = 14,641$ possible schedules.

The difficulty of the problem is due to: the number of possible schedules; the predicate constraints; and the fact that the input data and computational resources are distributed across the communication network. The size of the search space is polynomial in the length of the scheduling horizon and exponential in the number of tasks.

4. TECHNICAL APPROACH

This section describes the approach used to solve the scheduling problem presented in Section 3 and further develops the tools used in this approach.

4.1 Problem Abstraction using Processes

For every multi-index $(i, j)$ such that $T A(t_i) = a_j,$ define a quadruple,

$$[t_i, a_j] = (States_{t_i}, start_{t_i}, trans_{t_i}, msgs_{t_i}). $$  

(22)

This quadruple is called a process. Define a set of messages $M,$ possibly infinite and closed under union. The set $States_{t_i}$ is the state space of process $[t_i, a_j], i.e., a set of configuration quantities that may be boolean, integer, or real valued that describe the configuration of the process and represent its memory; $start_{t_i} \in States_{t_i}$ is the state at which process $[t_i, a_j]$ begins operation;

$$trans_{t_i} : M \times States_{t_i} \rightarrow States_{t_i}, $$  

(23)

$$msgs_{t_i} : States_{t_i} \rightarrow M. $$  

(24)

Processes advance this state appropriately through the function $trans_{t_i},$ which accepts incoming messages and produces a new state from the current state. The function $msgs_{t_i}$ is responsible for reading the new state, and based on this, sending appropriate messages. Let $Processes$ be the set of processes defined in (22).

Define the undirected process graph $G_p = (Processes, \mathcal{E}_p),$ where

$$\mathcal{E}_p = \{([t_i, a_j], [t_k, a_l]) \mid [a_j, a_l] \in \mathcal{E}_c\}. $$  

(25)

There is an edge between processes $[t_i, a_j]$ and process $[t_k, a_l]$ if and only if there is an edge between agent $a_j$ and agent $a_l$ in the communication graph. The process graph for the example of Section 1 is shown in Figure 2.

The vertex set of the process graph for the example of Section 1 is given by (6) and the edge set follows from (20) and (25).
4.2 Traditional Backtracking

In a centralized setting, that is when the problem data are not distributed among several agents, a traditional backtracking algorithm such as that presented in Russell and Norvig [2010] can be used to find a feasible schedule.

Given an ordered set \( N_{\text{max}} \), define a set of sequences \( N_{\text{max}}^N \) of length equal or less than \( N \). Let \( ts_i \in N_{\text{max}}^N \) be one such sequence of length \( i \), with elements \( ts_i(j) \in ts_i \), \( j \in \{1, \ldots, i\} \). Similar to (10), define constraints \( p_{\text{m}} : N_{\text{max}}^N \rightarrow \{0,1\} \), on sequences \( ts_i \in N_{\text{max}}^N \), \( m = 1, \ldots, N_d \). The problem here is to find a sequence \( ts_i \in N_{\text{max}}^N \) such that \( i = N \) and all of the constraints \( p_{\text{m}}(ts_i) = 1 \), \( m = 1, \ldots, N_d \). This can be done by the method of backtracking.

Define the set \( \text{untried}_i \subseteq N_{\text{max}}^N \). Define the function \( \text{expand} : N_{\text{max}}^N \times N_{\text{max}}^N \rightarrow N_{\text{max}}^N \) that accepts as input a sequence \( ts_i \), and \( \text{untried}_i \) and returns a new sequence \( ts_{i+1} \in N_{\text{max}}^N \).

Define the function \( \text{backtrack} : N_{\text{max}}^{N} \rightarrow N_{\text{max}}^{N} \) that accepts as input a sequence \( ts_{i+1} \) and returns the sequence \( ts_i \).

\[
\text{Data: } ts_0 = \emptyset \\
\text{untried}_0 = N_{\text{max}}^N \\
\text{while } i < N, \text{ or } \exists m : p_m(ts_i) = 0 \text{ do} \\
\text{if } \text{untried}_i = \emptyset \text{ then} \\
\text{no solution exists} \\
\text{return } ts_i = \emptyset \\
\text{else if } \exists m : p_m(ts_i) = 0 \text{ then} \\
\text{ts}_{i+1} := \text{backtrack}(ts_i) \\
\text{else} \\
\text{ts}_{i+1} := \text{expand}(ts_i, \text{untried}_i) \\
\text{untried}_{i+1} := \text{untried}_i \backslash ts_{i+1}(i + 1) \\
\text{return } ts_{i+1} = N_{\text{max}}^N \\
\text{end} \\
\text{Result: } ts_i \\
\text{Algorithm 1: Backtracking}
\]

The procedure in algorithm 1 performs backtracking. If all values in \( N_{\text{max}}^N \) are exhausted for \( ts_1 \), there is no sequence \( ts \in N_{\text{max}}^N \) that can satisfy the given constraints. The expansion of new, possibly feasible, schedules is performed at line 9 in algorithm 1. The removal, or pruning of subsequent portions of the set \( N_{\text{max}}^N \) that do not contain a feasible schedule is done during backtracking at line 7 of algorithm 1.

4.3 Distributed Backtracking

\textbf{Distributed backtracking} uses a set of processes to implement the traditional backtracking algorithm in a distributed way. Each element \( ts_N(i), i = 1, \ldots, N \) in the sequence \( ts_N \in N_{\text{max}}^N \) will have a process \( [t_i, a_i] \) associated with it. The process \( [t_i, a_i] \) is responsible for setting the value of \( ts_N(i) \) in \( N_{\text{max}}^N \) and for running the \textit{expand} and backtrack functions. The processes send appropriate messages to relay knowledge of the results of applying the \textit{expand} and backtrack functions.

The remainder of this paper details the Distributed Non-Sequential Backtracking (DNSB) algorithm that is used to solve the distributed scheduling problem of Section 3. The DNSB algorithm is based on the Asynchronous Weak Commitment (AWC) search of Yokoo [1995], the Distributed Backtracking algorithm of Yokoo et al. [1992], and it includes the improvements offered in Minton et al. [1990]. The structure of the DNSB algorithm is inspired by Lynch [1996]. The DNSB algorithm is shown here to inherit the completeness properties of the traditional backtracking algorithm.

5. SOLUTION PROCEDURE

The \textit{trans} and \textit{msg} algorithms are run by processes \( [t_i, a_i] \in \text{Processes} \) in synchronous rounds. These distributed algorithms are designed to perform the distributed backtracking search to find a feasible schedule. The schedule at round \( r \) is referred to as \( TS_r \). The state \( s \) is a process of \([t_i, a] \) as follows.

\[
\text{state}_{ij} = (j, i, \text{highPrChange}_{ij}, \text{untried}_{ij}, \text{current}_{ij}, \text{violated}_{ij}, \text{btRequest}_{ij}, \text{consistent}_{ij}, \text{foundSolution}_{ij}, \text{maxHighPr}_{ij}, \text{minHighPr}_{ij}, \text{sendBt}_{ij}) \\
\text{noSol}_{ij} = \{j, i, \text{highPrChange}_{ij}, \text{untried}_{ij}, \text{current}_{ij}, \text{violated}_{ij}, \text{btRequest}_{ij}, \text{consistent}_{ij}, \text{foundSolution}_{ij}, \text{maxHighPr}_{ij}, \text{minHighPr}_{ij}, \text{sendBt}_{ij}\} \\
\text{bt}_{ij} = \{(j, i, \text{bt}), TS(t_i), PR(t_i), \text{consistent}_{ij}, TS(\text{minHighPr}_{ij}), \text{foundSolution}_{ij}, \text{maxHighPr}_{ij}, \text{minHighPr}_{ij}, \text{sendBt}_{ij}\} \\
\text{bt}_{ij} = \{(j, i, \text{bt}), TS(t_i), PR(t_i), \text{consistent}_{ij}, TS(\text{minHighPr}_{ij}), \text{foundSolution}_{ij}, \text{maxHighPr}_{ij}, \text{minHighPr}_{ij}, \text{sendBt}_{ij}\} \\
\text{bt}_{ij} = \{(j, i, \text{bt}), TS(t_i), PR(t_i), \text{consistent}_{ij}, TS(\text{minHighPr}_{ij}), \text{foundSolution}_{ij}, \text{maxHighPr}_{ij}, \text{minHighPr}_{ij}, \text{sendBt}_{ij}\}
\]

The three types of messages are the following: the \( M_{\text{ack}} \in M_{\text{ack}} \) tell that \( t_i \) is currently consistent; \( M_{\text{ack}} \in M_{\text{req}} \) request that backtracking be initiated; and \( M_{\text{sol}} \in M_{\text{sol}} \) communicate that there does not exist a feasible schedule. The messages \( M \) are defined as follows.

\[
M = M_{\text{ack}} \cup M_{\text{req}} \cup M_{\text{sol}} \\
M_{\text{ack}} = \{(j, i, ok, TS(t_i), PR(t_i), \text{consistent}_{ij}, TS(\text{minHighPr}_{ij}), \text{foundSolution}_{ij}, \text{maxHighPr}_{ij}, \text{minHighPr}_{ij}, \text{sendBt}_{ij})\} \\
M_{\text{req}} = \{(j, i, bt, TS(t_i), PR(t_i), \text{consistent}_{ij}, TS(\text{minHighPr}_{ij}), \text{foundSolution}_{ij}, \text{maxHighPr}_{ij}, \text{minHighPr}_{ij}, \text{sendBt}_{ij})\} \\
M_{\text{sol}} = \{\}\text{where } i, j \text{ are such that } [t_i, a_i] \in \text{Processes}.
\]

For process \( [t_i, a_i, t_k] \), the boolean quantity \( \text{highPrChange}_{ij} = 1 \) if \( TS(t_k) \neq TS(t_i) \) for any \( t_k \in C \) where \( PR(t_k) > PR(t_i) \). The vector \( \text{untried}_{ij} \) is an ordered list of untied values in \( N_{\text{max}}^N \). The integer quantity \( \text{current}_{ij} \) is the index of the value in \( \text{untried}_{ij} \) corresponding to \( TS(t_i) \). The set \( \text{violated}_{ij} \) is the set of those \([t_k, a_i] \) for which there exists a \( p_m = 0 \) and \( t_i, t_k \in \text{untried} \). When a message \( M_{\text{ack}} \) is received from a process \( [t_k, a_i] \), \( \text{btRequest}_{ij} = 1 \). If \( \text{violated}_{ij} \subseteq \{t_k : PR(t_k) > PR(t_i)\} \), then \( \text{consistent}_{ij} = 1 \). If \( \text{consistent}_{ij} = 1 \) and \( \text{untried}_{ij} = 1 \) for all \( [t_k, a_i] \) where
\( t_k \in C_i, \) foundSolution\(_{ij} := 1 \). The tasks maxHighPr\(_{ij}\) and minHighPr\(_{ij}\) are defined as follows,

\[
\begin{align*}
\text{maxHighPr}_{ij} &= \arg \max_{t_k} \text{PR(violated}_{ij}), \\
\text{minHighPr}_{ij} &= \arg \min_{t_k} \{ t_k \in \text{violated}_{ij} \},
\end{align*}
\]

\[\text{s.t. } \text{PR}(t_k) > \text{PR}(t_i)\].

The termination condition for this distributed algorithm is:

**Termination condition:**

\( \forall [t_i, a_j] \in \text{Processes}, \) foundSolution\(_{ij} := 1 \) or \( \forall [t_i, a_j] \in \text{Processes}, \) foundSolution\(_{ij} := -1 \).

The function findConsistent() is used by trans\(_{ij}\) to set the value of TS(t\(_i\)) to one that does not violate any constraints. If there is no such value, it sets a value that minimizes the number of violated constraints as in Minton et al. [1990]. The function setBacktrack() saves the pair \( (t_k, T S(t_k)) \) so that a backtrack request may be sent by process \( [t_i, a_j] \) to process \( [t_k, a_i] \). The function sendMsg() sends an M\(_k\) message to all neighbors; a M\(_A\) message to process \( [\text{minHighPr}_{ij}, TA(\text{minHighPr}_{ij})] \); and a M\(_\text{noSol}\) message to all neighbors.

**Algorithm 3:** backTrack\(_{ij}(\cdot)\)

**Data:** state\(_{ij}\), M

1. if M = M\(_\text{noSol}\) then
2. \hspace{1em} foundSolution := -1
3. end
4. if highPrChange\(_{ij} = 1\) then
5. \hspace{1em} untried\(_{ij} := \text{N}_\text{max}\)
6. \hspace{1em} consistent\(_{ij} := \text{findConsistent}(\cdot)\)
7. end
8. if \( \min \{ \text{PR(violated}_{ij}) \} > \text{PR}(t_i) \) then
9. \hspace{1em} consistent\(_{ij} := \text{findConsistent}(\cdot)\)
10. end
11. if consistent\(_{ij} = 0\) then
12. \hspace{1em} setBacktrack(minHighPr\(_{ij}, TS(\text{minHighPr}_{ij}))\)
13. end
14. sendBt := 1
15. \hspace{1em} backTrack\(_{ij}(\cdot)\)
16. end
17. if btRequest\(_{ij} = 1\) then
18. \hspace{1em} btRequest\(_{ij} := 0\)
19. \hspace{1em} if highPrChange\(_{ij} = 0\) then
20. \hspace{1em} \hspace{1em} backTrack\(_{ij}(\cdot)\)
21. \hspace{1em} end
22. \hspace{1em} end
23. if \( \forall t_k \in C_i, \) consistent\(_{ij} = 1 \) and consistent\(_{ij} = 1\) then
24. \hspace{1em} foundSolution\(_{ij} := 1\)
25. end
26. Result: state\(_{ij}\)

**Algorithm 2:** trans\(_{ij}\)

For process \( [t_i, a_j] \), algorithm 2 is the trans\(_{ij}\) function. Algorithm 3 performs the backtracking operation and is called by trans\(_{ij}\), and algorithm 4 is the msg\(_{ij}\) function. Line 4 of algorithm 2 tests whether any values TS(t\(_k\)) where \( PR(t_k) > PR(t_i) \) have changed. If this is the case, untried\(_{ij}\) is reset to N\(_\text{max}\). Lines 5 and 6 of algorithm 2 perform the operation of line 11 of algorithm 1.

Lines 8-14 of algorithm 2 tests whether task \( PR(t_i) \) is less than all tasks in violated\(_{ij}\). If this is the case, process \( [t_i, a_j] \) will be responsible for initiating backtracking by sending backtrack requests associated with the violated constraints it is involved in. In this way, either line 9 of algorithm 2 will find a value of TS(t\(_i\)) for which \( t_i \) is consistent or lines 11 and 12 of algorithm 2 will initiate backtracking. This performs the operation of line 7 of algorithm 1.

Lines 15-20 of algorithm 2 responds to requests for backtracking. If all of the possible values for TS(t\(_i\)) have been exhausted, a backtracking request is sent to process \( [\text{minHighPr}_{ij}, TA(\text{minHighPr}_{ij})] \), lines 7 and 8 of algorithm 3. If all possible values have not been exhausted, the current value TS(t\(_i\)) is removed from untried\(_{ij}\) and a new value is found using findConsistent() by lines 11 and 12 of algorithm 3. This prunes portions of the search space that do not contain a feasible schedule, i.e., performs the operation of line 10 of algorithm 1.

Lines 21-23 of algorithm 2 test whether any t\(_k\) ∈ Ci, are not consistent. If all of these tasks have consistent values TS(t\(_k\)), process \( [t_i, a_j] \) declares that it has found a solution.

Lines 2-4 of algorithm 3 tests whether \( PR(t_k) > PR(t_i) \) for all t\(_k\) ∈ violated\(_{ij}\). If this is the case and process \([t_i, a_j]\) has exhausted all possible start times in N\(_\text{max}\), process \([t_i, a_j]\) declares that there is no solution.

**Lemma 5.1.** For all \([t_i, a_j] \in \text{Processes}, consistent\(_{ij} = 1\) if and only if TS feasible.

**Lemma 5.2.** A process \([t_i, a_j] \in \text{Processes sets}\)

\( \text{foundSolution}_{ij} := -1 \) if and only if there does not exist a feasible schedule.

**Lemma 5.3.** All processes \([t_i, a_j] \in \text{Processes eventually}\) either all set foundSolution\(_{ij} := 1\) or all set foundSolution\(_{ij} := -1\).
Theorem 5.4. The DNSB algorithm is correct and complete.

The proofs of Lemmas 5.1-5.3 and Theorem 5.4 can be found in the Appendix.

6. RESULTS

The DNSB algorithm in algorithms 2, 3, and 4 is implemented in C++. Each agent is implemented as a class that controls several processes. The communication graph is shown in Figure 1. The agents communicate by: encoding messages in character strings; sending the character strings over a reliable channel; and appropriately decoding messages received. The initial schedule for the example of Section 1 is

$$TS = \{(t_1, 0), (t_2, 0), (t_3, 0), (t_4, 0)\}$$

which is not feasible, due to violation of the constraints (12)-(14). The DNSB algorithm finds the feasible schedule of (16) after 12 rounds. When the constraint of (13) is replaced with the constraint,

$$p_2(TS) = |(TS(t_2) > 0) \land (TS(t_3) > 0)| \land$$
$$|(TS(t_2) = 0) \land (TS(t_3) = 0)|$$

which clearly results in a problem with no feasible schedule, all processes \([t_i, a_j] \in \text{Processes output} \text{foundSolution} = -1\) in 24 rounds.

7. CONCLUSIONS AND FUTURE WORK

This paper develops the Distributed Non-Sequential Backtracking algorithm. The DNSB algorithm is applied to arbitrarily constrained scheduling problems. The DNSB is distributed in the sense of Section 1 and is correct and complete. Future work includes the coupling of this work with distributed assignment methods to achieve distributed assignment and scheduling in one package. Ongoing work on planning in the presence of network failure will also be included to guarantee fault tolerance.

REFERENCES

M. Burrows. The chubby lock service for loosely-coupled distributed systems, 2006.


8. APPENDIX I

Proof of Lemma 5.1:

"\rightarrow" Assume that for all \([t_i, a_j] \in \text{Processes, consistent}_{ij} = 1\). This implies that \(p_m(TS) = 1, m = 1, \ldots, N_{\text{ct}}, which implies that TS is feasible.

"\leftarrow" Assume TS is feasible. A feasible TS implies that \(\forall[t_i, a_j] \in \text{Processes, violated}_{ij} = \emptyset\). This implies that \(\text{violated}_{ij} \subseteq \{t_k : PR(t_k) > PR(t_k)\} = \emptyset\), which implies that for all \([t_i, a_j] \in \text{Processes, consistent}_{ij} = 1\).

Proof of Lemma 5.2:

"\rightarrow" Assume some process \([t_i, a_j] sets \text{foundSolution}_{ij} = -1\). This implies that all values TS(t_i) \in N_{\text{max}}, by line 11 in algorithm 3 have been eliminated as possibly belonging to a feasible schedule.

"\leftarrow" Assume there does not exist a feasible schedule. That is, there exists at least one constraint \(p_i : \forall TS \in N_{\text{max}}, p_i(TS) = 0\). Let \(t_i = \max_{t \in T} PR(T_i)\), eventually \([t_i, a_j]\) will receive \(bt\) messages for all values in \(N_{\text{max}}\). This implies that eventually, \(\text{foundSolution}_{ij} = 1\).

Proof of Lemma 5.3:

Processes \([t_i, a_j] \in \text{Processes eventually either 1 all set \text{foundSolution}_{ij} = 1 \text{ or 2) all set \text{foundSolution}_{ij} = -1. 1) follows directly from Lemma 5.1, 2) follows from Lemma 5.2 and lines 1 and 2 of algorithms 2 and 4.

Proof of Theorem 5.4:

Correctness follows directly from Lemmas 5.1 and 5.2. Completeness follows from Lemmas 5.1-5.3. That is, termination is implied by Lemma 5.3 and correctness is guaranteed by Lemmas 5.1 and 5.2.