ROBUST DISTRIBUTED COORDINATION OF HETEROGENEOUS ROBOTS THROUGH TEMPORAL PLAN NETWORKS

Andreas F. Wehowsky, Stephen A. Block, and Brian C. Williams

Computer Science and Artificial Intelligence Laboratory,
Massachusetts Institute of Technology, Cambridge, MA, 02139, USA,
Email: andreas@wehowsky.dk, {sblock, williams}@mit.edu

ABSTRACT

Real-world applications of autonomous agents require coordinated groups to work in collaboration. Dependable systems must plan and carry out activities in a way that is robust to failure to and uncertainty. Previous work has produced algorithms that provide robustness at the planning phase, by choosing between functionally redundant methods, and at the execution phase, by dispatching temporally flexible plans. However, these algorithms use a centralized architecture in which all computation is performed by a single processor. As a result, these implementations suffer from communication bottlenecks at the master processor, require significant computational capabilities, and do not scale well.

This paper introduces the plan extraction component of a robust, distributed executive for contingent plans. Contingent plans are encoded as Temporal Plan Networks (TPNs), which compose temporally flexible plans hierarchically and provide a choose operator. First, the TPN is distributed over multiple agents, by creating a hierarchical ad-hoc network and mapping the TPN onto this hierarchy. Second, candidate plans are extracted from the TPN with a distributed, parallel algorithm that exploits the structure of the TPN. Third, temporal consistency of the candidate plans is tested using a distributed Bellman-Ford algorithm. This algorithm is empirically validated on randomized contingent plans.

Key words: distributed AI; planning; plan execution and monitoring.

1. INTRODUCTION

The ability to command coordinated groups of autonomous agents is key to many real-world tasks, such as the construction of a Lunar habitat. In order to achieve this goal, we must perform robust execution of contingent, temporally flexible plans in a distributed manner. Methods have been developed for the dynamic execution [1] of temporally flexible plans [2]. These methods adapt to failures that fall within the margins of the temporally flexible plans and hence add robustness to execution uncertainties.

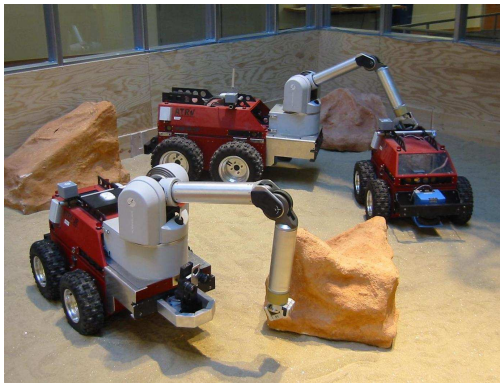

Figure 1. Multiple Rover testbed

To address plan failure, [3] introduced a system called *Kirk*, that performs dynamic execution of temporally flexible plans with contingencies. These contingent plans are encoded as alternative choices between functionally equivalent sub-plans. In Kirk, the contingent plans are represented by a Temporal Plan Network (TPN) [3], which extends temporally flexible plans with a nested choose operator. To dynamically execute a TPN, Kirk continuously extracts a plan from the TPN that is temporally feasible, given the execution history, and dispatches the plan, using the methods of [4]. Dynamic execution of contingent plans adds robustness to plan failure. However, as a centralized approach, Kirk is extremely brittle to poor communication at the master processor due to the communication bottleneck. In addition, Kirk does not scale well as the size of the plan is increased.

We address these two limitations through a distributed version of Kirk, which performs distributed dynamic execution of contingent temporally flexible plans. This pa-

per focuses on the algorithm for dynamically selecting a feasible plan from a TPN. Methods for performing distributed execution of the plan are presented in [5]. Our key innovation is a hierarchical algorithm for searching a TPN for a feasible plan in a distributed manner. In particular, our plan selection algorithm, called the Distributed Temporal Planner (DTP), is comprised of three stages.

1. Distribute the TPN across the processor network,
2. Generate candidate plans through distributed search on the TPN, and
3. Test the generated plans for temporal consistency.

This paper begins with an example TPN and an overview of the way in which DTP operates on it. We provide a formal definition of a TPN and then discuss the three stages of DTP. Finally, we discuss the complexity of the DTP algorithm and present experimental results demonstrating its performance.

2. EXAMPLE SCENARIO

In this section, we discuss at a high level the three step approach taken by DTP to solve an example problem. A TPN is to be executed by a group of seven processors, $p1, \ldots, p7$. The TPN is represented as a graph in Fig. 2, where nodes represent points in time and arcs represent activities. A node at which multiple choices exist for the following path through the TPN is a *choice node* and is shown as an inscribed circle.

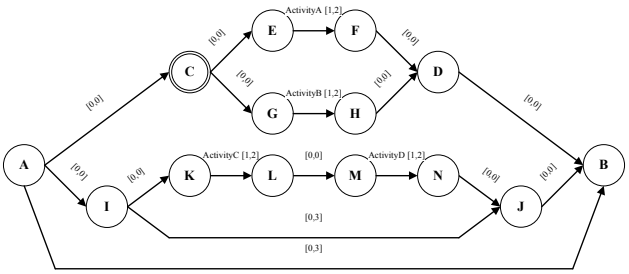

Figure 2. Example TPN

First, the TPN itself is distributed over the processors to allow the plan selection to take place in a distributed fashion. To facilitate this, a leader election algorithm is used to arrange the processors into a hierarchy (Fig. 3). The hierarchical structure of the TPN is then used to map subnetworks to processors. For example, the master processor $p1$ handles the merging of multiple branches of the plan at the start node (node A) and the end node (node B). It passes responsibility for each of the two main subnetworks to the two processors immediately beneath it in the hierarchy. Nodes *C,D,E,F,G,H* are passed to $p2$ and nodes *I,J,K,L,M,N* are passed to $p3$.

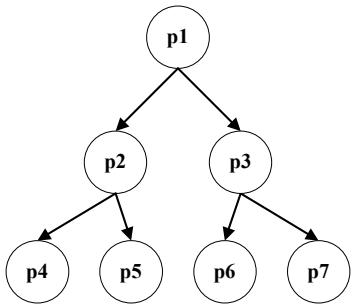

Figure 3. A three-level hierarchy formed by leader election

The processors then work together to extract a temporally consistent plan from the TPN. The first stage generates a candidate plan, which corresponds to selecting a single subnetwork from the plan at each of the choice nodes. This is done in a hierarchical fashion, where each processor sends messages to its neighbors, requesting that they make selections in the subnetworks for which they are responsible. These selections are made in parallel. In this example, only the subnetwork owned by $p2$ (nodes C,D,E,F,G,H) contains a choice of path, so $p2$ must decide between `ActivityA` and `ActivityB`, whereas $p3$ has no choice to make.

Having generated a candidate plan, the third and final step of DTP is to test it for consistency. Again, this is done in a hierarchical fashion, where consistency checks are first made at the lowest level and successful candidates are then checked at an increasingly high level. For example, $p2$ and $p3$ simultaneously check that their subnetworks are internally consistent. If so, $p1$ then checks that the two candidates are consistent when executed in parallel. In DTP, candidate generation and consistency checking are interleaved, such that some processors generate candidates while others simultaneously check consistency.

3. TEMPORAL PLAN NETWORKS

A TPN augments temporally flexible plans with a `choose` operator and is used by DTP to represent a contingent, temporally flexible plan. The `choose` operator allows us to specify nested choices in the plan, where each choice is an alternative sub-plan that performs the same function.

The primitive element of a TPN is an `activity`$[l, u]$, which is a hardware command with a simple temporal constraint. The simple temporal constraint $[l, u]$ places a bound $t^+ - t^- \in [l, u]$ on the start time t^- and end time t^+ of the network to which it is applied. A TPN is built from a group of activities and is defined recursively using the `choose`, `parallel` and `sequence` operators, which derive from the Reactive Model-based Programming Language (RMPL) [6].

- `choose`$(TPN_1, \ldots, TPN_N)$ introduces multiple subnetworks of which only one is to be chosen. A choice variable is used at the start node to encode the currently selected subnetwork. A choice variable is *active* if it falls within the currently selected portion of the TPN.
- `parallel`$(TPN_1, \ldots, TPN_N)\,[l, u]$ introduces multiple subnetworks to be executed concurrently. A simple temporal constraint is applied to the entire network. Each subnetwork is referred to as a *child* subnetwork.
- `sequence`$(TPN_1, \ldots, TPN_N)\,[l, u]$ introduces multiple subnetworks which are to be executed sequentially. A simple temporal constraint is applied to the entire network. For a given subnetwork, the subnetwork following it in a `sequence` network is referred to as its *successor*.

Graph representations of the `activity`, `choose`, `parallel` and `sequence` network types are shown in Fig. 4. Nodes represent time events and directed edges represent simple temporal constraints.

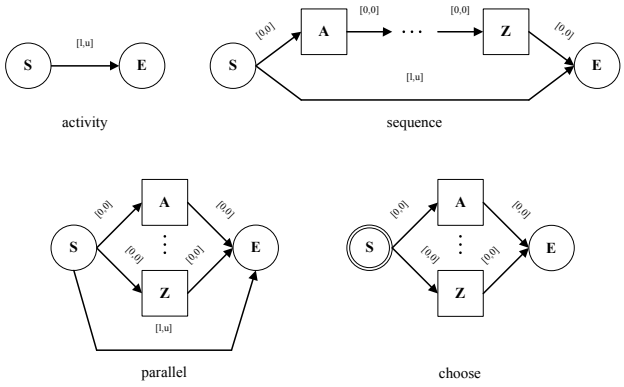

Figure 4. TPN Constructs

Definition 1 *A **feasible solution** of a TPN is an assignment to choice variables such that 1) all active choice variables are assigned, 2) all inactive choice variables are unassigned, and 3) the currently selected temporally flexible plan is temporally consistent. A temporally flexible plan is **temporally consistent** if there exist times that can be assigned to all events such that all temporal constraints are satisfied.*

4. TPN DISTRIBUTION

The DTP algorithm distributes the computation involved in finding a feasible solution to the TPN over all available processors. Consequently, the processors must be able to communicate with each other, in order to coordinate their actions. We therefore establish an ad-hoc communication network such that adjacent processors are able to communicate. In addition, an overall leader must be selected to communicate with the outside world and initiate planning.

4.1. Ad-Hoc Processor Network Formation

We use the leader election algorithm in [7] to arrange the processors into a hierarchical network, an example of which is shown in Fig. 3. For each node, the node immediately above it in the hierarchy is its *leader*, those at the same level within that branch of the hierarchy are its *neighbor leaders* and those directly below it in the hierarchy are its *followers*. The leader election algorithm forms the hierarchy using a message passing scheme and in doing so, ensures that every node can communicate with its leader, as well as all neighbor leaders and followers. In addition, the hierarchical nature of the network lends itself well to the distribution of the TPN, which is also hierarchical.

4.2. TPN Distribution over the Processor Network

We implement the distribution of the DTP computation by assigning to each processor responsibility for a number of nodes from the TPN graph representation. Each processor maintains all the data from the TPN relevant to the nodes for which it is responsible.

This distribution scheme requires that processors responsible for TPN nodes linked by temporal constraints are able to communicate. The algorithm in Fig. 5 distributes the TPN over the processor hierarchy such that this communication is available. It allows distribution down to the level at which a processor handles only a single node. This allows DTP to operate on heterogeneous systems that include computationally impoverished processors.

We now demonstrate the distribution algorithm using the TPN in Fig. 2 and the processor hierarchy in Fig. 3. The TPN is supplied from an external source, which establishes a connection with the top leader, *p1*. The TPN is a `parallel` network at the highest level, so processor *p1* assigns the start and end nodes (nodes *A,B*) to itself (line 7). There are two subnetworks, which *p1* assigns to its two followers, *p2* and *p3* (lines 15-18). *p1* passes the `choose` network (nodes *C,D,E,F,G,H*) to *p2* and the `sequence` network (nodes *I,J,K,L,M,N*) to *p3*. *p2* and *p3* then process their networks in parallel. *p2* assigns the start and end nodes (nodes *C,D*) to itself (line 7). The network has two subnetworks, which *p2* assigns to two of its followers, *p4* and *p5* (lines 15-18). *p2* passes `ActivityA` (nodes *E,F*) to *p4* and `ActivityB` (nodes *G,H*) to *p5*. Since activities can not be decomposed, *p4* and *p5* assign nodes *E,F* and *G,H*, respectively, to themselves (lines 3-4). Meanwhile, *p3* receives the `sequence` network and assigns the start and end nodes (nodes *I,J*) to itself (line 7). The network has two subnetworks, which *p3* assigns to two of its followers, *p6* and *p7* (lines 15-18). *p3* passes `ActivityC` (nodes *K,L*) to *p6* and `ActivityD` (nodes

1: wait for TPN
2: $n \leftarrow$ number of followers of p
3: **if** TPN is of type `activity` **then**
4: assign start and end nodes of TPN to p
5: **else**
6: $k \leftarrow$ number of subnetworks
7: assign start and end nodes to p
8: **if** $n = 0$ **then**
9: **if** p has a neighbor leader v **then**
10: send $\frac{k}{2}$ subnetworks of TPN to v
11: assign $\frac{k}{2}$ subnetworks of TPN to p
12: **else**
13: assign TPN to p
14: **end if**
15: **else if** $n \geq k$ **then**
16: **for** each of k subnetworks of TPN **do**
17: assign subnetwork of TPN to a follower of p
18: **end for**
19: **else if** $n < k$ **then**
20: **for** each of n subnetworks of TPN **do**
21: assign subnetwork to a follower of p
22: **end for**
23: assign remaining $(k - n)$ subnetworks of TPN to p
24: **end if**
25: **end if**

Figure 5. TPN Distribution Algorithm for node p

M,N) to *p7*. *p6* and *p7* then assign nodes *K,L* and nodes *M,N*, respectively, to themselves (lines 3-4).

5. CANDIDATE PLAN GENERATION

Having distributed the TPN across the available processors, DTP conducts search for candidate plans. These plans correspond to different assignments to the choice variable at each choice node [8]. DTP uses parallel, recursive, depth first search to make these assignments. This use of parallel processing is one of the key advantages of DTP over traditional centralized approaches. DTP is implemented using a distributed message-passing architecture and uses the following messages during candidate plan generation.

- `findfirst` instructs a network to make the initial search for a consistent set of choice variable assignments.
- `findnext` is used when a network is consistent internally, but is inconsistent with other networks. In this case, DTP uses `findnext` messages to conduct a systematic search for a new consistent assignment, in order to achieve global consistency. `findnext` systematically moves through the subnetworks and returns when the first new consistent assignment is found. Therefore, a successful `findnext` message will cause a change to the value assigned to a single choice variable, which may in turn cause other choice variables to become active or inactive.
- `fail` indicates that no consistent set of assignments was found and hence the current set of assignments within the network is inconsistent.
- `ack`, short for acknowledge, indicates that a consistent set of choice variable assignments has been found.

Whenever a node initiates search in its subnetworks, using `findfirst` or `findnext` messages, the relevant processors search the subnetworks simultaneously. This is the origin of the parallelism in the algorithm.

DTP operates on three network types formed from the four types fundamental to a TPN. These are `activity`, `parallel-sequence` and `choose-sequence`, as shown in Fig. 6, where the subnetworks $A_i, \ldots, Z_i$ are of any of these three types. We handle the simple temporal constraint present on a `sequence` network by considering a `sequence` network as a special case of a `parallel-sequence` network, in which only one subnetwork exists.

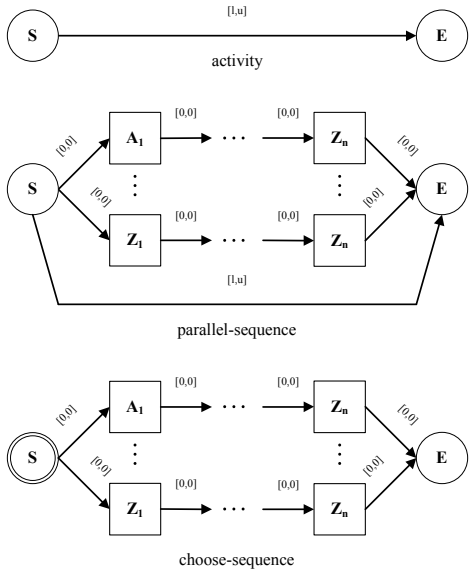

Figure 6. Constructs for DTP

This choice of network types requires that a network is able to communicate directly with its successor. This is made possible by the Sequential Network Identifier (SNI), which is a pointer to the start node of the successor network.

The following three sections describe the actions carried out by the start node of each network type on receipt of a `findfirst` or `findnext` message. Note that while a simple temporal constraint $[l, u]$ is locally inconsistent if $l > u$, we assume that the TPN is checked prior to running DTP, to ensure that all temporal constraints are locally consistent. This assumption means that only `parallel-sequence` networks can introduce temporal inconsistencies.

Activity During search, an `activity` node propagates request messages forward and response messages backward.

Parallel-Sequence Network On receipt of a `findfirst` message, the start node v of a `parallel-sequence` network S calls `parallel-findfirst(`v`)` (Fig. 7). The node initiates a search of S's subnetworks and of any successor network, in order to find a temporally consistent plan. First, the start node sends `findfirst` messages to the start node of each child subnetwork of the `parallel-sequence` structure (lines 2-4) and to the start node of the successor network, if present (lines 5-7). These searches are thus conducted in parallel. If any of the child subnetworks or the successor network returns a `fail` message (line 12), then no consistent assignment to the choice variables exists and the start node returns `fail` (line 13).

```
 1: parent ← sender of msg
 2: for each child do
 3:    send findfirst to w
 4: end for
 5: if successor B exists then
 6:    send findfirst to B
 7: end if
 8: wait for all responses from children
 9: if successor B exists then
10:    wait for response from B
11: end if
12: if any of the responses is fail then
13:    send fail to parent
14: else
15:    if check-consistency( v ) then
16:       send ack to parent
17:    else
18:       if search-permutations(v) then
19:          send ack to parent
20:       else
21:          send fail to parent
22:       end if
23:    end if
24: end if
```

Figure 7. `parallel-findfirst(node v)`

Conversely, suppose that all child subnetworks and the successor network return `ack` messages, indicating that variable assignments have been made such that each is internally temporally consistent. The start node must then check for consistency of the entire `parallel-sequence` network S (line 15). This is performed by a distributed Bellman Ford consistency checking algorithm, which is explained in the next section. If the consistency check is successful, the start node returns an `ack` message to its parent (line 16) and the search of the `parallel-sequence` network is complete.

If, however, the consistency check is not successful, the start node must continue searching through all permutations of assignments to the child subnetworks for a globally consistent solution. It calls `search-permutations(`v`)` (line 18) and sends an `ack` message to its parent if this is successful and a `fail` message otherwise.

In the `search-permutations(node` v`)` function (Fig. 8), the start node sends `findnext` messages to each subnetwork (lines 1-2). If a subnetwork returns `fail`, the start node sends a `findfirst` message to that subnetwork to reconfigure it to its original, consistent solution (lines 11-12) and we move on to the next subnetwork. If at any point, a subnetwork returns `ack`, the start node tests for global consistency and returns `true` if successful (lines 4-6). If the consistency check is unsuccessful, we try a different permutation of variable assignments (line 8) and continue searching. If all permutations are tested without success, the function returns `false` (line 15).

```
 1: for w = child-0 to child-n do
 2:    send findnext to w
 3:    wait for response
 4:    if response = ack then
 5:       if check-consistency( v ) then
 6:          return true
 7:       else
 8:          w ← child-0
 9:       end if
10:    else
11:       send findfirst to w
12:       wait for response
13:    end if
14: end for
15: return false
```

Figure 8. `search-permutations(node v)` *function*

When the start node v of a `parallel-sequence` network receives a `findnext` message, it executes `parallel-findnext(`v`)` (Fig. 9). First, the start node calls `search-permutations(`v`)` to systematically search all consistent assignments to its subnetworks, in order to find a new globally consistent assignment (line 1). If this is successful, the start node sends `ack` to its parent (line 2). If it fails, however, the start node attempts to find a new assignment to the successor network. If a successor network is present, the start node sends a `findnext` message and returns the response to its parent (lines 3-6). If no successor network is present, then no globally consistent assignment exists and the node returns `fail` (line 8).

Choose-Sequence Network When the start node of a `choose-sequence` network receives a `findfirst` message, it executes the `choose-findfirst()` function (Fig. 10). The node searches for a consistent plan by making an appropriate assignment to its choice variable. It also initiates a search in any successor network. To do so, it first sends a `findfirst` message to the successor network if present (lines 2-4). It then systematically assigns each possible value to the network's choice variable and, in each case, sends a `findfirst` message to the enabled subnetwork (lines 5-7). If a subnetwork returns

```
1: if search-permutations() then
2:    send ack to parent
3: else if successor B exists then
4:    send findnext to B
5:    wait for response
6:    send response to parent
7: else
8:    send fail to parent
9: end if
```

Figure 9. `parallel-findnext(node v)` *function*

`fail`, indicating that no consistent assignment exists, the current value of the choice variable is trimmed from its domain to avoid futile repeated searches (line 18), and the next value is assigned.

```
 1: parent ← sender of msg
 2: if successor B exists then
 3:    send findfirst to B
 4: end if
 5: for w = child-0 to child-n do
 6:    choicevariable ← w
 7:    send findfirst to w
 8:    wait for response from child w
 9:    if response = ack then
10:       if successor B exists then
11:          wait for response from successor B
12:          send response to parent
13:       else
14:          send ack to parent
15:       end if
16:       return
17:    else
18:       remove w from child list
19:    end if
20: end for
21: send fail to parent
```

Figure 10. `choose-findfirst()` *function*

As soon as a subnetwork returns `ack`, indicating that a consistent assignment to the subnetwork was found, the start node waits for a response from the successor network (if present) to determine whether or not a consistent assignment was found to it too (line 11). Once a response has been received from the successor network, the start node forwards this response to its parent and the search terminates (line 12). If no successor network is present, the network is consistent and the start node returns `ack` to its parent (line 14).

If all assignments to the network's choice variable are tried without receipt of an `ack` message from a child subnetwork, the start node returns `fail` to its parent, indicating that no consistent assignment exists (line 21).

When the start node of a `choose-sequence` network receives a `findnext` message, it executes the `choose-findnext()` function (Fig. 11). The start node first attempts to find a new consistent assignment for the network while maintaining the current value of the choice variable. It does so by sending `findnext` to the currently selected subnetwork (lines 1-2). If the response is `ack`, a new consistent assignment has been found, so the start node returns `ack` to its parent and the search is over (lines 4-6).

```
 1: w ← current assignment
 2: send findnext to w
 3: wait for response
 4: if response = ack then
 5:    send ack to parent
 6:    return
 7: end if
 8: while w < child-n do
 9:    w ← next child
10:    send findfirst to w
11:    wait for response
12:    if response = ack then
13:       send ack to parent
14:       return
15:    else
16:       remove w from child list
17:    end if
18: end while
19: if successor B exists then
20:    send findnext to B
21:    for w = child0 to child-n do
22:       choice variable ← w
23:       send findfirst to w
24:       wait for response from child w
25:       if response = ack then
26:          break
27:       end if
28:    end for
29:    wait for response from B
30:    send response to parent
31: else
32:    send fail to parent
33: end if
```

Figure 11. `choose-findnext()` *function*

If this fails, however, the start node searches through unexplored assignments to the network's choice variable, in much the same way as it does on receipt of a `findfirst` message (lines 8-18). Finally, if this strategy also fails, the start node attempts to find a new consistent assignment in any successor network, by sending a `findnext` message to the node referenced by its SNI parameter (lines 19-20). Note that the start node must reset the local network to the previous consistent configuration, because the unsuccessful search has left it in an inconsistent state. This is achieved by repeating the search process used on receipt of a `findfirst` message (lines 21-28). Once the successor network has replied, the start node forwards the response to its parent (lines 29-30).

6. TEMPORAL CONSISTENCY CHECKING

Each of the candidate assignments generated during search on the TPN must be tested for temporal consistency, which is implemented by the `check-consistency(node` v`)` function. Consistency checking is performed with the distributed Bellman-Ford Single Source Shortest Path algorithm [9], which is run on the distance graph corresponding to the currently active portion of the TPN. Temporal inconsistency is detected as a negative weight cycle [2]. The consistency checking process is interleaved with candidate generation, such that DTP simultaneously runs multiple instances of the distributed Bellman-Ford algorithm on isolated subsets of the TPN.

The distributed Bellman-Ford algorithm has two key advantages. First, it requires only local knowledge of the network at every processor. Second, when run synchronously, it runs in time linear in the number of processors in the network. DTP ensures synchronization by the fact that whenever a node initiates search in its subnetworks, it waits for responses from all processors in the form of `ack` or `fail` messages before proceeding.

7. PERFORMANCE ANALYSIS

The overall time complexity of the centralized planning algorithm is worst-case exponential. The backtrack search used to assign choice variables has worst-case time complexity N^e, where N is the number of nodes and e is the size of the domain of the choice variables. The Bellman-Ford algorithm used for consistency checking has complexity $N^2 log N + NM$, where M is the number of edges.

DTP also has exponential overall time complexity. The backtrack search remains N^e in the worst case, but we can expect significant computational savings from the fact that the distributed Bellman-Ford algorithm runs in time N.

A C++ implementation of DTP was used to test the run time performance of DTP by simulating an array of processors searching for a feasible solution to a TPN. Exactly one node was assigned to each processor and the number of nodes in the TPN was varied between 1 and 100. In each case, the number of TPN constructs (`parallel`, `sequence` or `choose`) was varied between 3 and 30 and the maximum recursive depth was varied between 4 and 10. Run time was measured by the number of listen-act-respond cycles completed by the processor network.

Fig. 12 shows a plot of the number of cycles against the number of nodes. The results show that the variation in the number of cycles is approximately linear with the number of nodes. This shows that in practice, the run time is dominated by the distributed Bellman Ford consistency checking algorithm, which is linear in the number of nodes, not by the backtrack search, which is exponential.

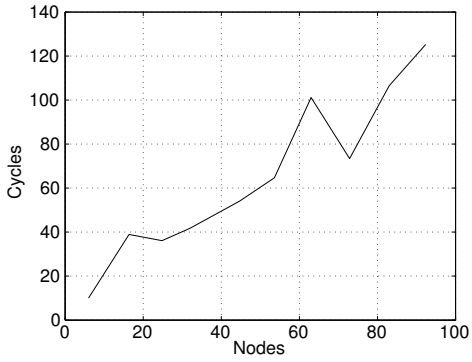

Figure 12. Number of cycles vs. number of nodes

This is because the worst-case exponential time complexity of DTP occurs only when the TPN is composed entirely of `choose` networks, in which case there is no opportunity for parallel execution. However, typical TPNs used in real applications consist largely of `parallel` and `sequence` networks. This allows processors to conduct parallel search and consistency checks, which greatly reduces the time complexity of DTP.

8. DISCUSSION

This paper introduced the Distributed Temporal Planner (DTP), which is the plan selection component of a distributed executive that operates on contingent, temporally flexible plans. DTP distributes both data and processing across all available agents. First, DTP forms a processor hierarchy and assigns subnetworks from the TPN to each processor. It then searches the TPN to generate candidate plans, which are finally checked for temporal consistency. DTP exploits the hierarchical nature of TPNs to allow parallel processing in all three phases of the algorithm.

ACKNOWLEDGMENTS

This work was made possible by the sponsorship of the DARPA NEST program under contract F33615-01-C-1896.

REFERENCES

1. Morris, P. and Muscettola, N. Execution of temporal plans with uncertainty. In *AAAI-00*, 1999.

2. Dechter, R., Meiri, I., and Pearl, J. Temporal constraint networks. *Artificial Intelligence, 49:61-95, 1991*, 1990.

3. Kim, P., Williams, B., and Abramson, M. Executing reactive, model-based programs through graph-based temporal planning. In *Proc. of IJCAI 2001, Seattle, WA*, 2001.

4. Tsamardinos, I., Muscettola, N., and Morris, P. Fast transformation of temporal plans for efficient execution. In *AAAI-98*, 1998.

5. Stedl, J. L. A formal model of tight and loose team coordination. Master's thesis, MIT, Cambridge, MA, September 2004.

6. Williams, B. C., Ingham, M., Chung, S., and Elliott, P. Model-based programming of intelligent embedded systems and robotic explorers. In *IEEE Proceedings, Special Issue on Embedded Software*, 2003.

7. Nagpal, R. and Coore, D. An algorithm for group formation in an amorphous computer. In *Proc. of PDCS 1998, Las Vegas, NV*, 1998.

8. Mittal, S. and Falkenhainer, B. Dynamic constraint satisfaction problems. In *AAAI-1990*, 1990.

9. Lynch, N. *Distributed Algorithms*. Morgan Kaufmann, 1997.