Efficient Extraction of Optimal, Temporally Flexible Plans

Aisha Walcott and Brian Williams
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology, Cambridge, MA 02139 USA
{aisha,williams}@mit.edu

***Abstract*—Robots that operate in hazardous and dynamic environments must be equipped with onboard reasoning capabilities that enable them to autonomously generate mission plans. To operate in such environments, the mission plans must be continuous and temporally flexible. For critical missions, such as search and rescues, the success of the mission relies not only on the feasibility of the plan, but also on the quality of the plan. Thus, to generate robust efficient, temporally flexible plans, a measure of utility is required.**

We introduce a novel, forward heuristic planner that quickly extracts the optimal, temporally flexible plan, given a hierarchical description of alternative contingency plans and a utility function. Contingency plans are encoded as Temporal Plan Networks (TPNs). Our work presents two key innovations. The first is a compact encoding of a TPN search state. The second is an admissible and informative heuristic, TPN-Max. We empirically validate our solution on TPNs with increasing level of difficulty. Our results show that the TPN-Max heuristic performs significantly better, for TPNs, than the Max heuristic used in the forward heuristic planners [1].

I. INTRODUCTION

Today's emergencies, such as search and rescues, natural disasters, and fires, continue to pose great challenges and threats to rescuers and emergency personnel (Figure 1). The deployment of robots at the World Trade Center site, for example, highlighted the potential for robots to aid in humans in such. This work focuses on the development of an onboard autonomous system that enables mobile robots to select optimal mission plans. More specifically, we present a planner that searches for the best plan given the temporal constraints.

There is an extensive body of work on temporal executives that achieve robustness, by operating on a least commitment plan that leaves temporal flexibility. Temporal flexibility is exploited by an executive to achieve robustness to temporal disturbances. This is accomplished by dynamically scheduling activities and projecting their consequences into the future, in order to ensure correctness. The robustness breaks down when the temporal disturbances perturb the plan and cause a plan failure. In order to resolve such failures a slow planning process must be invoked.

Research presented in [6] improves robustness to such plan failures by extending temporal planning and execution to the execution of contingent Temporally Flexible Plans (TFPs). These plans are encoded in a model called a Temporal Plan Network (TPN). TPNs are comprised of a nesting of alternative, temporally flexible sub-plans. Temporal flexibility is achieved by generating plans without a fixed time schedule. The TPN planner presented in [6] adopts techniques from temporally flexible planners such as HSTS [8] and IXTET [7]. These planners adjust to varying execution times of activities by enforcing temporal constrains on the minimal set of activities that ensure successful plan execution.

While TPNs have proven to be a robust approach to contingent plan execution, the TPN planner presented in [6] makes no guarantee on the quality of the executed plan in terms of maximizing utility. That is, contingency and temporal flexibility, while providing robustness, are insufficient for critical missions where the cost executing the activities in the plan affects the success of the plan.

This research extends TPNs to include a measure of utility, and provides a novel online method for continuously updating the selection of the optimal TFP. We develop a forward heuristic TPN planner, and present two key innovations. The first is a compact encoding of a TPN search state. The second is an effective, admissible heuristic called TPN-Max. Our results show significant a reduction in both space and time using the TPN-Max heuristic for TPNs, as compared to the Max heuristic introduced in [1], and a the uniform cost search.

II. EXAMPLE SCENARIO

Consider an urban search and rescue mission, called Search-and-Sense, in which an agile autonomous air vehicle (AAV) is deployed (Figure 1). The goals of the mission are to collect images while exploring the environment and search for victims. To explore the environment the robot must navigate a corridor and search one of three places for victims; the Office, Corridor-B, or the Lab.

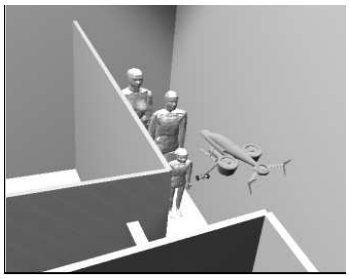

Fig. 1. Example mission in which an autonomous air vehicle searches a building for victims.

ISBN 970-769-070-4, ISRA 2006 Proceedings

In this mission, the cost of searching either the Corridor-B or the Lab is much greater than the cost of exploring the Office. Additionally, the Office is closest to the AAV, and the most likely place where victims may be trapped. A feasible planner might select a sub-optimal plan that includes Corridor-B or the Lab, neglecting the Office where the victims are located.

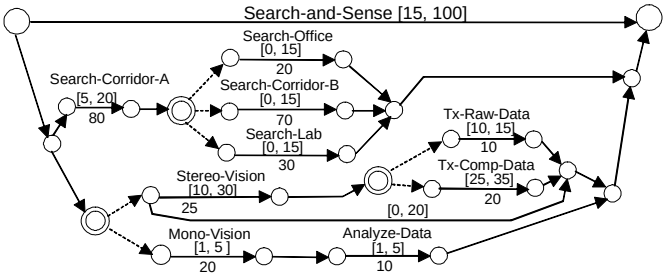

Fig. 2. Example TPN for a Search-and-Sense mission.

In this example, the feasible planner does not include the cost of executing activities when searching for a plan; thus, a highly sub-optimal plan might be selected. Our goal is to modify a temporally flexible planning model and incorporate the cost of executing activities. An example of this model, called a Temporal Plan Network (TPN) is given in Figure 2.

III. Temporal Plan Networks

Temporal Plan Networks were introduced in [6]. They draw from Temporal Constraint Networks [2] and Simple Temporal Networks [8]. TPNs support activities, simple temporal constraints, predecessor and successor relations, and contingencies. TPNs are similar in structure to Activity Networks [5]; however TPNs support flexible time bounds, concurrency, and mutex relations. Figure 3 and Figure 4 provide a subset of the TPN grammar and mapping to its graphical equivalent. For a complete definition of TPNs see [6].

TPN→ (`Sequence` $TPN_1,\ldots, TPN_N$) | (`Choose` $TPN_1,\ldots, TPN_N$)
(`Parallel` $TPN_1,\ldots, TPN_N$) | *Primitive*
Primitive→ `Activity`(cost) [lb,ub] | `With-Timing` <*TPN*> [lb,ub]

Fig. 3. Grammar used to encode a Temporal Plan Network.

A TPN recursively combines the primitives, `Activity` and `With-Timing`, with the operators `Sequence`, `Choose` and `Parallel` to represent a hierarchical description of alternative contingency plans. Given a utility function, each activity is parameterized with an associated cost.

Our algorithm operates on TPN graphs. The nodes in a TPN graph are called *events* and represent temporal events that relate the start and end times of activities and of TPN sub-graphs. The global start and end of a TPN are events labeled Start and End. Special events in the TPN created by the choose operator are referred to as *decision points* (shown in Figure 5 with double circles). In Figure 5, events d_1, d_2, and d_3 are decision points. The arcs in a TPN are labeled with flexible temporal bounds and impose an ordering between events. An arc may have at most one activity. For example, the arc b→c is labeled with the activity Search-Corridor-A with a temporal bound of [5,20], which implies that the activity must take a minimum of 5 units of time and a maximum of 20 units of time to execute. Events at the tail of an arc are called *targets*. For example, event 'a' has two targets: b and d_1. Finally, a sequence of continuous events and arcs a TPN is called a *thread*. An example of a thread in Figure 5 is a→b→c→d_2.

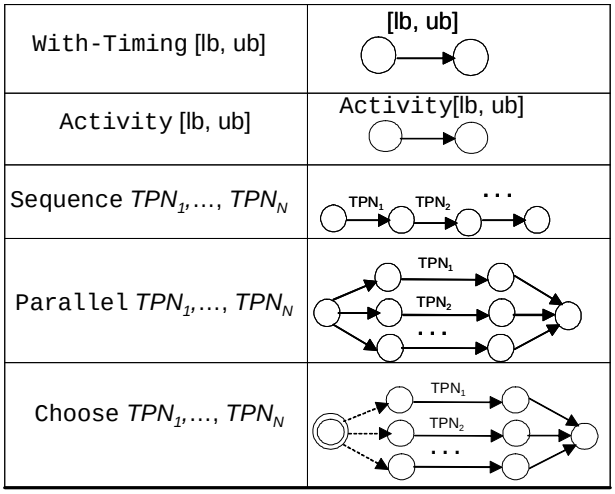

Fig. 4. Mapping from TPN grammar to TPN Graph.

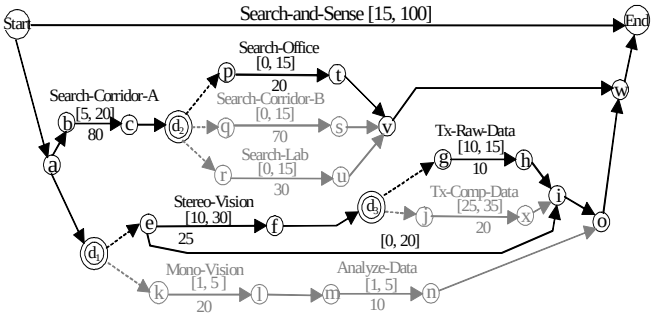

Fig. 5. Bold sub-graph represents the optimal feasible plan for the Search-and-Sense mission.

Kim et. al. developed a TPN planner called Kirk [6]. Kirk applies a modified network search algorithm in order to extract a *feasible* temporally flexible plan in a TPN [6]. A feasible plan is a *complete* and *consistent* set of contiguous threads in a TPN.

A *complete plan* is a TPN sub-graph with three properties:

1) The sub-graph originates at the start event and ends at the end event.
2) The sub-graph contains only one thread extending from each decision point in the sub-graph.
3) The sub-graph includes all threads extending from each non-decision event.

A *consistent plan* is a TPN sub-graph with one property:

1) The sub-graph does not violate any temporal constraints. That is, the temporal constraints of the sub-graph are satisfiable [6] [8]. This is detected by

applying a graph search algorithm to detect temporal inconsistencies [8].

While the Kirk planner searches a TPN for a feasible plan, it does not address planning problems for which the cost of executing activities is critical to the success of the mission. Thus, we develop an optimal forward heuristic planner that incorporates the costs of executing activities, and uses an admissible TPN heuristic, called TPN-Max, to efficiently and systematically explore the search space. Our planner extracts the optimal TFP if, and only if, one exists. The bold sub-graph in Figure 5 represents the optimal TFP for the Search-and-Sense mission. The optimal TFP is, in turn, used as input to an executive, which schedules and dispatches the activities in the plan to the specified robot(s) [7] [8].

IV. Optimal Plan Selection

We formulate the problem of selecting the optimal feasible TFP as a state space search problem. We define a partial plan, present a compact encoding of a TPN search state, and describe the search tree representing the TPN search space. Finally, the algorithms for the Expand function that maps TPN search states to successor states are given.

A. Partial Plan Encoding

The search space of an optimal planning problem consists of all possible *partial plans* in a TPN. A partial plan is an incomplete plan, comprised of threads in a TPN sub-graph. More specifically, a partial plan is a set of contiguous concurrent threads in a TPN that originate at the Start event, all threads have not been extended to the End event (Figure 6). A partial plan satisfies two properties of a complete plan, that is, properties 2) and 3).

We compactly encode a partial plan p as a pair $\langle PPFringe, PPChoices\rangle$ where:

- *PPFringe*(p) = $\{e_1,e_2,\ldots,e_n\}$, the set of un-extended terminal events, e_i. A terminal event is the event at the end of a thread.
- *PPChoices*(p) = $\{\langle d_1,e_i\rangle, \langle d_2,e_j\rangle, \ldots, \langle d_n,e_k\rangle\}$ the set of pairs of choices made at decision points in p's threads. Each pair contains a decision point, d_j, and one target event e_i, of the decision point.

Figure 6 depicts a partial plan denoted by the following pair of sets: $\langle\{i,t,End\}, \{\langle d_1,e\rangle, \langle d_2,p\rangle, \langle d_3,g\rangle\}\rangle$.

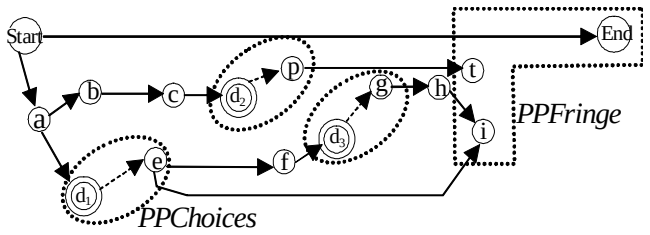

Fig. 6. Example of a partial plan from the Search-and-Sense TPN.

B. TPN Search State

The choices, *PPChoices*, in a partial plan are used to denote a TPN search state. All partial plans with equivalent sets of *PPChoices* map to the same search state. The partial plan in Figure 6, for example, maps to the search state $\{\langle d_1,e\rangle, \langle d_2,p\rangle, \langle d_3,g\rangle\}$.

C. Search Tree

Our forward heuristic planner constructs a search tree in order to systematically explore the search space of an optimal TFP problem. The search tree represents the set of all unique possible search states.

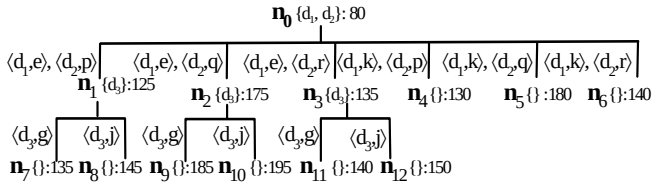

Fig. 7. Complete search tree representing the search space of the Search-and-Sense TPN. Leaf nodes are nodes with no children, eg. n_7 - n_{12}.

The tree is made up of *nodes* and *branches* that are uniquely labeled. A node n is labeled with a set a set of un-extended terminal events denoted *fringe*(n). A branch b in the search tree is labeled by a set of pairs of choices denoted *choices*(b). The set of choices contain pairs of decision points with targets. A *path* through the tree is a sequence of nodes and branches originating at the root. The complete search tree for the Search-and-Sense TPN is given in Figure 7.

```
procedure Node_To_Partial_Plan(Search-Tree tree, Node
n, TPN tpn) returns a partial plan.
1.   PPFringe ← Fringe(n)
2.   PPChoices ← {}
3.   t ← n, initialize search tree node
4.   while Parent(t) ≠ Root(tree) do
5.     PPChoices ←PPChoices ∪Branch(t,Parent(t))
6.     t ← Parent(t)
7.   endwhile
8.   Create TPN sub-graph partial plan
9.   Initialize visited of each TPN event to false
10.  e ← Start_Event(tpn)
11.  stack ← {}
12.  Push(stack,e)
13.  while not(Empty(stack)) do
14.    e ← Pop(stack)
15.    if not(Visited(e))then
16.      Visited(e) ← true
17.    if Decision_Point(e) == true then
18.      target ← Get_Target(e,PPFringe)
19.      arcs ← arcs ∪ {Get_TPN_Arc(tpn,e,target)}
20.      events ← events ∪ {target}
21.      Push(stack,target)
22.    else
23.      for t_i ∈ targets(e) and not(Visited(t_i)) do
24.        arcs ← arcs ∪{Get_TPN_Arc(tpn,e,t_i)}
25.        events ← events ∪{t_i}
26.        if t_i ∉ PPFringe then
27.          Push(stack,t_i)
28.      endfor
29.  endwhile
30.  return Partial_Plan(arcs, events)
```

Fig. 8. Procedure to map a search tree node to its equivalent partial plan.

The root of the tree represents the initial state of an optimal TFP problem. The root is a partial plan which is comprised of a set of threads that originate at the Start event and either at the

first decision point reached or the End event. The root is encoded as follows:

-*PPFringe*(root) = {e_1, e_2, …, e_n}, where each event e_i is either a decision point or the end event of the TPN.

-*PPChoices*(root) = {}.

Node n_0 in Figure 4 is the root node that corresponds to the partial plan comprised of three threads: 1) Start→a→b→d_1, 2) Start→a→b→c→d_2, and 3) Start→End; Node n_0 is compactly encoded as ⟨{d_1,d_2,End}, {}⟩.

The search tree is constructed by applying the Expand function to the least-cost leaf node in the tree. We use an evaluation function, as in A* search, to determine the estimated cost of a feasible plan through a node, and to determine the order in which leaf nodes are expanded [9]. The evaluation function is the sum of the path cost, g(*n*), and a heuristic, *H*(*n*). The path cost of each node is shown in Figure 7. We describe the equation for computing *H*(*n*) in Section 4.

A node in the search tree maps to a unique partial plan *p* in the TPN. The procedure is given in Figure 8, and consists of two main steps. The first step is to construct the encoding of *p* as ⟨*PPFringe*(*p*), *PPChoices*(*p*)⟩ (Lines 1-7). The second step is to map the encoding to its actual partial plan in the TPN (Lines 8-28).

D. Search Tree Expansion

To construct the search tree the Expand function is applied to the least-cost leaf node in the current tree. The Expand function is comprised of two procedures: 1) Extend threads (Figure 9) and 2) Generate child nodes (Figure 11).

1) Extend Threads: To extend threads of partial plan corresponding to a node *n*, threads from each event in *fringe*(*n*) are extended until either a decision point is reached or the end event is reached along each thread. This is accomplished by applying a modified version of a Depth-First Search (DFS) algorithm (Figure 9). As threads are extended during the modified DFS, activity costs are added to the current path cost, *g*(*n*), of node *n*. When an event t_i is reached during DFS, two cases are checked.

-*Case 1* (Lines 10-11): t_i is a decision point signaling a choice between possible threads. At this point t_i, is not extended further, and *fringe*(*n*) is updated to include the event t_i. For example, the fringe of search tree node n_1, *fringe*(n_1), is set to {e,p}. The threads of n_1, originating from events e and p are extended, as illustrated in Figure 10. First, e is extended along the thread e→f→d_3. The event d_3 is a decision point, and thus, is not extended further (Figure 8a). Then DFS is continued, extending the thread e→i→o→w→End. At this point, two threads in the partial plan that corresponds to n_1 have converged at the End event (Figure 10b). To detect when threads converge Case 2 is applied.

-*Case 2* (Lines 8-9): t_i is an event where two or more threads have re-joined and converged. This case serves two purposes: 1) to avoid redundant extension of threads, and 2) to test for temporal consistency, as done in [6]. To detect when threads converge, we maintain a table that maps each search tree node to its corresponding set of selected events in its partial plan, denoted *selected_events*. The selected events of a node *n* are the events reached during expansion of node *n*. Threads have converged at an event *e*, if *e* is in either in the set *selected_events*(*n*) or in the set of selected events of an ancestor of node of *n*. For example, in Figure 7, *selected_events*(n_0) = {Start, End, a, b, d_1, d_2} and *selected_events*(n_1) = {e, f, d_3, i, o, w, p, t, v}, where threads converge at events w and End (Figure 10c).

```
procedure Extend-Events(Node n,TPN tpn) returns
updated n, if temporally consistent; otherwise false.
1.  stack ← {}
2.  updated_fringe ← {}
3.  for each event f_i ∈ fringe(n) do
4.    Push(f_i,stack)
5.    Visited(f_i) ← true
6.    while not(Empty(stack)) do
7.      e ← Pop(stack)
8.      if Threads_Converge(e) and
         not(Temporally_Consistent(e)) then
9.        return false
10.     else if Decision-Point(e) = true then
11.       updated_fringe ← updated_fringe ∪ e
12.     else
13.       for each t_i ∈ targets(e,tpn) do
14.         if not(Visited(t_i)) then
15.           g(n) ← g(n) + cost(e,t_i)
16.           Visited(t_i) ← true
17.           selected_events ← selected_events ∪ {t_i}
18.           Push(stack,t_i)
19.       endfor
20.   endwhile
21. endfor
22. Fringe(n) ← updated_fringe
23. return n
```

Fig. 9. Procedure to extend the fringe events of a search tree node. This is the first step of the Expand function.

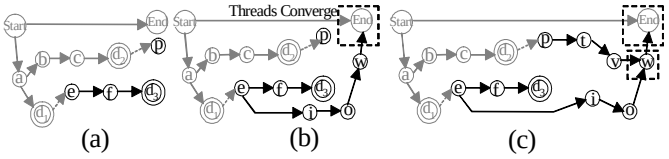

Fig. 10. Illustration of extending threads of node n_1 shown in bold. The gray threads were extended by node n_0.

2) Generate Child Nodes: Once a search tree node *n* has been extended, its fringe either is empty or contains decision points. If the fringe is empty then the node corresponds to a complete plan. Otherwise, the targets of the decision points in *fringe*(*n*) are used to generate new child nodes in the search tree. This is done by creating the set of all possible choices by computing the cross-product between the target events of each decision point in the *fringe*(*n*) (Figure 11, Lines 1-6). For example, the root node n_0 has two decision points in its fringe d_1 and d_2. The event d_1 has two target choices, either e or k; and d_2 has three target choices, p, q, or r. Thus, there are six sets of choices that result from performing the cross product between the targets of d_1 and d_2: {e,k}×{p,q,r} = {{e,p}, {e,q}, {e,r}, {k,p}, {k,q}, {k,r}}. Each set represents a new child node, which is initialized and inserted into the search tree (Figure 11, Lines 7-13) as new leaves.

```
procedure Generate-Child-Nodes(Search-Tree-Node n,
Search-Tree tree) inserts child nodes into tree.
1.  targets_sets ← {}
2.  for each decision-point d_i ∈ Fringe(n) do
3.    choices_i ← {t_j|{t_j ∈ Targets(d_i)}
4.    targets_sets ← targets_sets ∪ {choices_i}
5.  endfor
6.  combinations ← Cross_Product(targets_sets)
7.  for each set cset_i ∈  combinations do
8.    child_node
9.    Parent(child_node) ← n
10.   Fringe(child_node) ← cset_i
11.   g(child_node) ← g(Parent(child_node))
12.   Insert(tree,child_node)
13. endfor
```

Fig. 11. Procedure to generate and insert new nodes into the search tree. This is the second and final step of the Expand function.

V. TPN HEURISTIC

To efficiently focus the search towards the optimal TFP, we develop heuristic equations applied to TPN events, denoted $h(e_i)$, and the heuristic used for search tree nodes, denoted $H(n)$. The heuristic cost of each TPN event is computed prior to the search for the optimal TFP. We present the evaluation function $f(n)$ which represents the estimated cost of a search tree node n. The estimated cost of a search tree node n is an underestimate of the actual cost of a *solution* through n. A solution is feasible temporally flexible plan. Our forward heuristic TPN planner finds the optimal feasible solution by expanding the leaf nodes in order of their estimated cost.

A. Heuristic for TPN Events

To efficiently focus the search towards the optimal TFP, we adopt a strategy similar to that of forward heuristic planners, such as HSP [1] and FF [4]. These planners define a planning problem as a state space search problem, and extract heuristics from the encoding of the problem. The heuristic is extracted from the *relaxed* version of the problem. A relaxed planning problem is a simpler version of the problem. We define a relaxed TPN as one in which the temporal constraints are not considered; thus, backtracking to a consistent plan is not required. With a relaxed TPN, the optimal pre-planning problem is reduced to a shortest path problem in the TPN search space, where the shortest path is the least-cost complete plan.

We apply the three heuristics, Min, Additive and Max, presented in [1], to compute an exact estimate for events in a TPN. The Min heuristic is applied to a disjunction of sub-goal, and the Additive and Max heuristics are applied to a conjunction of sub-goals. A *disjunction* of sub-goals is a set of goals where only one is selected. A *conjunction* of sub-goals is a set of goals where all the goals are selected. A disjunction of sub-goals in a TPN is represented by decision points, where only one thread is selected. The Min heuristic for TPNs is given in Equation 1, and also applies to the TPN primitives `Activity` and `With-Timing` (Figure 12). The heuristic cost of d_1 (Figure 5), for example, is the thread with the minimum cost to the End event: $h(d_1) = \min(h(k) + c(d_1,k), h(e) + c(d_1,e)) = \min(30,35) = 30$.

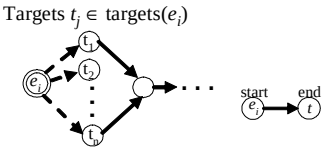

Fig. 12. Example of a TPN sub-graphs with disjunctive sub-goals.

$$h_{\min}(e_i) = \min_{t_j \in \text{targets}(e_i)} \left(cost(e_i, t_j) + h(t_j) \right) \qquad (1)$$

A conjunction of sub-goals in a TPN is represented by multiple threads extending from a non-decision event e_i. These are constructed by the `Parallel` operator. In this case, all threads from e_i must be selected. To compute the heuristic cost of a non-decision event with multiple target threads, we apply the HSP Additive heuristic [1], given in Equation 2.

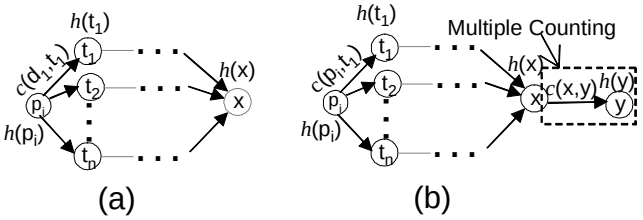

Fig. 13. Example of a TPN sub-graphs with conjunctive sub-goals. (a) Shows a TPN sub-graph ending at event x. (b) Shows a TPN sub-graph ending at event y. If the additive heuristic is applied directly then the cost of the thread from x to y is counted multiple times, resulting in an inadmissible heuristic.

$$h_{add}(e_i) = \sum_{t_j \in \text{targets}(e_i)} \left(cost(e_i, t_j) + h(t_j) \right) \qquad (2)$$

The additive heuristic is admissible if threads extending from a non-decision point converge at an event that has no targets, as shown in Figure 13a. However, if the threads converge at an event with at least one target, then the Additive heuristic is inadmissible, as shown in Figure 13b. This is the result of counting the cost of a thread multiple times. For example, in Figure 13 the heuristic costs $h(t_1)$, $h(t_2)$,…$h(t_n)$ each include the cost $h(x)$. Thus, if the Additive heuristic is used, then the cost $h(p_i)$ would be overestimated, and thus, is inadmissible heuristic.

To address the issue of multiple counting [1] suggests the Max heuristic (Equation 3). While the Max heuristic is an admissible heuristic for problems with dependent sub-goals, it often severely underestimates and not very informative. Recall, in a TPN dependent sub-goals are threads that rejoin and converge at the same event.

$$h_{\max}(e_i) = \max_{t_j \in \text{targets}(e_i)} \left(cost(e_i, t_j) + h(t_j) \right) \qquad (3)$$

We propose an exact heuristic for TPN sub-graphs with dependent sub-goals, called TPN-Additive given in Equation 4. This equation takes advantage of the construction of a TPN,

where there is a start and end event for each sub-graph created by the `Parallel` operator. The start event is referred to as the *parallel_start* and the end event is referred to as the *parallel_end*. For example, in Figure 13 the event p_i is the *parallel_start* and the event x is the *parallel_end.* With known parallel start and end events, we can lookup the cost of a *parallel_end* and subtract out the times that it is multiple counted (Equation 4).

$$h_{tpnadd}(e_i) = h_{add}(e_i) - \left[\left(|\text{targets}(e_i)| - 1\right) * h\left(parallel\_end(e_i)\right)\right] \quad (4)$$

Our forward heuristic TPN planner computes the heuristic costs of each TPN event by traversing backwards from the TPN End event to the Start event. At each event the dynamic programming principle is applied (see Equation 5) [10].

$$h(e_i) = \begin{cases} h_{\min}(e_i), & e_i \text{ is a decision point or has one target} \\ h_{tpnadd}(e_i), & e_i \text{ is a } parallel\_start \end{cases} \quad (5)$$

B. Search Tree Node Evaluation Function

During optimal TFP search, we use the heuristic costs of the events to compute the estimated cost of a solution through a search tree node *n*. The heuristic cost of *n*, *H*(*n*), is computed from the heuristic costs of the TPN events in the *fringe*(*n*). If *H*(*n*) is computed as the sum of the heuristic costs of each of event in its fringe, then the cost of a shared sub-goal might be counted multiple times. For example, in Figure 7, the search tree node n_0 contains decision points d_1 and d_2 and any thread extending from d_1 or d_2 will converge at the event w. If $h_{tpnadd}(d_1)$ and $h_{tpnadd}(d_2)$ are summed, then $h_{tpnadd}(w)$, their shared sub-goal, would be counted twice. To avoid this remaining element of multiple counting, we define *H*(*n*) to be the maximum over the events in *fringe*(*n*) (Equation 6). We call *H*(*n*) TPN-Max. The evaluation function for a search tree node is given in Equation 7. For example $f(n_0)$=80 + max($h(d_1)$, $h(d_2)$) = 80 + max(20,30)=110.

$$H(n) = \max_{e_i \in fringe(n)} \left(h(e_i)\right) \quad (6)$$

$$f(n) = g(n) + H(n) \quad (7)$$

In short, the heuristic cost of all events in a TPN is computed a priori. The TPN-Max heuristic is admissible, but approximate, because it computes the cost of a relaxed TPN ignoring the temporal constraints. Our essential contribution is that the TPN-Additive estimate is much more informative than that of the HSP Max heuristic.

VI. Experimental Results

We generate experiments with TPNs containing sub-graphs created by all three of the operators to highlight the performance of each heuristic (Figure 14). The activity costs and temporal constraints are randomly generated. We increase the level of difficulty by increasing the number of decision points from 2 to 12, and thus, increasing the number of states. Our results show a significant reduction in computation time and space complexity as the problem difficulty increases, for the TPN-Additive heuristic as compared to HSP Max.

On average, as the problems become more difficult, the

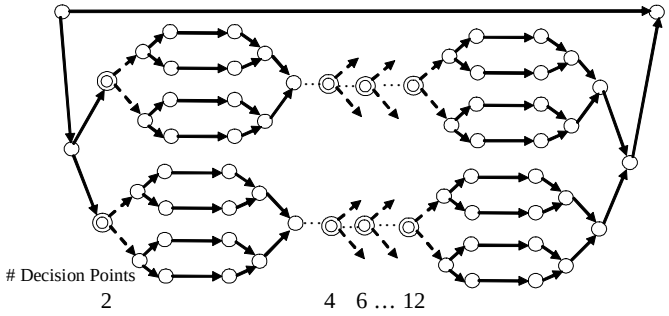

Fig. 14. Example TPN structure used in experiments. Heuristic functions were compared for TPNs of this form with increasing numbers of decision points.

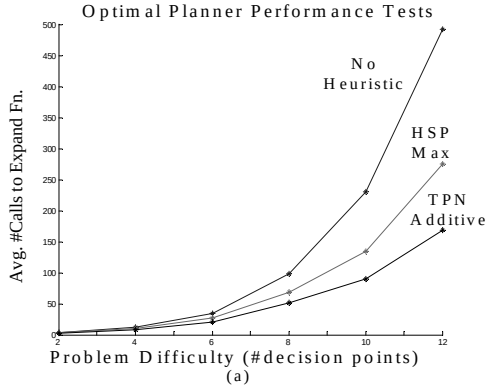

Fig. 15. Results comparing the average number of calls to the Expand function when using each heuristic.

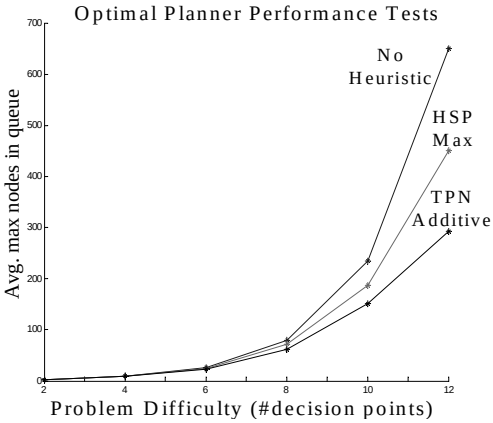

Fig. 16. Results comparing the average of the maximum nodes in the queue for expansion.

TPN-Additive explores makes less calls to the Expand function than HSP Max (Figure 15), a reduction in computation time. For example, with 12 decision points using no heuristic resulted in a mean of 492 calls, HSP Max resulted in a mean of 275 calls, and TPN-Additive resulted in a mean of 169 calls. TPN-Additive had 39% fewer calls compared to HSP Max. Similar results are shown when comparing the average of the maximum number of nodes in the priority queue, which corresponds to space complexity (Figure 16). In

this case, with 12 decision points TPN-Additive stored 35% fewer nodes than HSP Max.

VII. Discussion

Our TPN plan executive adopts techniques inherent to heuristic search planners that can solve problems with durative actions. We formulate our problem as a state space search, extract a heuristic from the problem encoding and then apply A* search or other optimal state space search algorithms. This is similar to other planners, such as FF [4], HSP [1] and SAPA [3].

To summarize, we propose a novel solution to optimal plan selection through temporal plan networks. We provided a compact encoding of a TPN search state along with an informative heuristic based on the encoding of a TPN. Given these two contributions, we develop a systematic, fast forward heuristic planner. The major computation occurs during the Expand function. The Extend Threads procedure runs in $c \times O(nm)$, where n is the number of events, m is the number of arcs in the TPN, and c is the number of times threads converge. In the worst case, all possible feasible plans are generated before the least-cost plan is selected. This is a result of backtracking to the next best node in the search tree if the current node being expanded corresponds to a temporally inconsistent partial plan. Empirical analyses show that our heuristic, TPN-Additive, considerably out-performs the HSP Max heuristic when applied to TPNs.

Acknowledgment

This research was supported by the Lucent CRFP Fellowship Program, and in part by MURI Airforce grant.

References

[1] Bonet, B. and Geffner, H. 2001. Planning as heuristic search. *AIJ: Sp. Is. Heuristic Search*, 129:5-33.

[2] Dechter, R., Meiri, I., Pearl, J., 1991. Temporal Constraint Networks. Artificial Intelligence, 49:61-95.

[3] Do, M. and Kambhampati, S. 2002. Planning Graph-based Heuristics for Cost-sensitive Temporal Planning. *AIPS.*

[4] Hoffman, J. and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253-302.

[5] Elmaghraby. S. E. 1964. An Algebra for the Analysis of Generalized Activity Networks. Management Science.

[6] Kim, P. Williams, B. and Abrahmson, 2001. Executing Reactive, Model-based Programs through Graph-based Temporal Planning. *IJCAI*.

[7] Lemai, S. and Ingrand, F. 2004**.** Interleaving Temporal Planning and Execution in Robotics Domains. *AAAI*. AAAI Press, Menlo Park, California.

[8] Muscettola, N., Morris, P. and Tsmardions, I. *KR*, 1998. Reformulating temporal plans for efficient execution.

[9] Nilsson, N. 1971. Problem-Solving Methods in Artificial Intelligence. McGraw-Hill, New York.

[10] Walcott, A. 2004. Unifying Model-Based Programming and Path Planning Through Optimal Search. S.M. Thesis, Massachusetts Institute of Technology.