Robert Effinger

Final Project Report

Cognitive Robotics

16.412J - Spring 2004

May 13, 2004

Table of Contents

2.) Overall Description

2.a) Brief Group Project Description

For the Final Group Project in 16.412J, myself, Dan Leaute, Seung Chung, and Dan Lovell developed an autonomous cooperative UAV demonstration using the Cloud-Cap Autopilot Airplane Simulator. Our group devised a scenario and map in-which a team of UAVs can cooperatively plan to extinguish forest fires. The goal of this project is to create a cooperative multi-UAV activity planner that only needs as input high-level mission goals and specifications(such as put-out-fire or image-fire-damage). The planner should then be able to decompose the high-level mission goals into lower-level goals, and motion commands, and then execute them autonomously. Allowing the operator to specify high-level mission goals instead of sending detailed sequential commands to the UAVs reduces planning time and operator error, and increases plan flexibility.

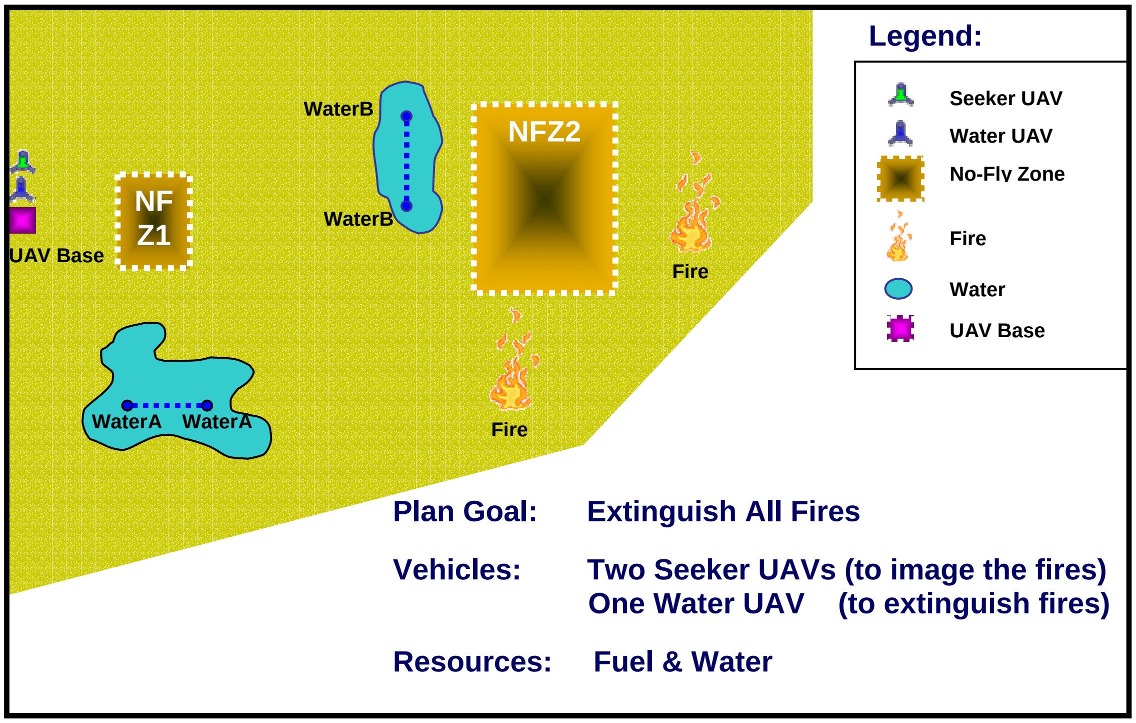


Figure 1: Autonomous Firefighting Cooperative UAV Scenario

To accomplish the goals set out above, the group has devised an integrated planner architecture that leverages the strengths of four separate planners. The four planners are Kirk, a strategic high-level mission planner, PDDL, a generative activity planner, dStarLite, a roadmap path-planner, and a MILP-based kinodynamic receding horizon path-planner. Kirk allows the operator to specify goals at an abstract level, in RMPL, The Reactive Model-Based Programming Language. The generative planner can then expand the high-level goals into lower-level activities. Then, the MILP-based receding horizon kinodynamic path planner generates an optimal motion plan, including obstacle avoidance. The dStarLite road-map path planner is called by all three afore-mentioned planners to get shortest-path distance to goal measurements and estimates.

2.b) Individual Accomplishments

My individual contributions are three-fold:

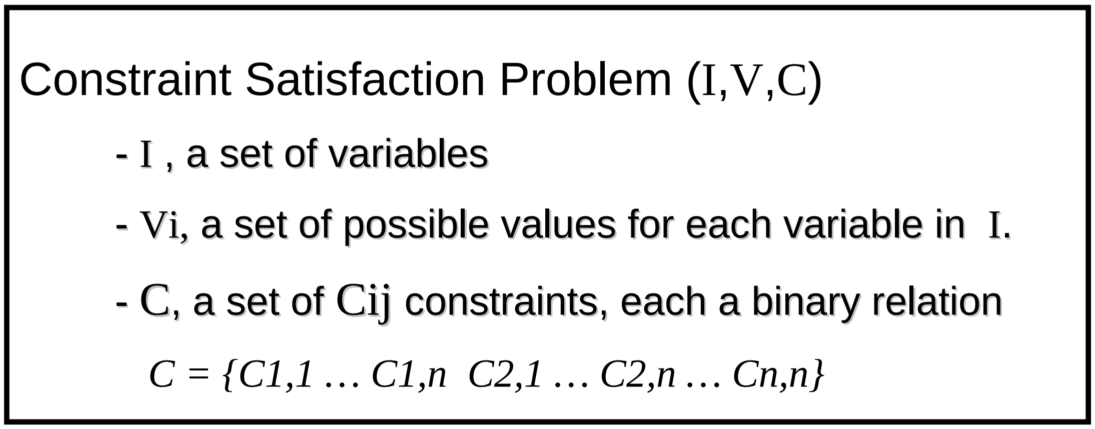
1.) Created a framework within Kirk to solve a TPN as a conditional CSP.

2.) Implemented Dynamic Backtracking within Kirk (to make the search more efficient)

3.) Interface with the Generative and Roadmap Path Planners

3.) Create Framework for TPN to be solved as a Conditional CSP

The definition of a classic CSP (not a conditional CSP) is as follows:



Constraint Satisfaction Problem (I,V,C)

- I , a set of variables
- Vi, a set of possible values for each variable in I.
- C, a set of Cij constraints, each a binary relation

$C = \{C1,1 \ldots C1,n \;\; C2,1 \ldots C2,n \ldots Cn,n\}$

Figure 2: Definition of a classic CSP

The solution to a classic CSP is found when each variable I is assigned a value from its domain Vi and the set of all Constraints {C} is satisfied.

In a conditional CSP, however, the set of variables {I} that need to be assigned values may grow or shrink in size depending on the values already assigned to variables in the partial solution, {P}. In the context of a TPN, this occurs when there are nested decision nodes. A decision node is considered to be nested if it's inarc belongs to a path that emanates from the outarc of another decision node. A nested decision node only needs to be assigned a value when the "parent" decision node, or "enabling" decision node has a particular outarc selected as it's choice. For all other choices of outarcs that the "parent" decision node can take, the "nested" decision node does not need to be considered as a part of the problem.

To frame the TPN search process as a conditional CSP, two functions were added to the TPN data structure, and a LIFO queue was added to keep track of which decision nodes need to be assigned values before the search is complete. The two functions are:

1.) getNextDecisionStartNodes()

2.) deactivateNestedDecisionStartNodes()

1.) getNextDecisionStartNodes()

The function getNextDecisionStartNodes() recursively "walks" the TPN and returns the first set (actually returns a map, not a set) of decision start node(s) that it encounters. (The function can return more than one decision start node if there are parallel branches in the TPN)

The search is initiated by calling getNextDecisionStartNodes(startNode) on the startNode of the TPN. Any decision start nodes that are found are pushed onto the LIFO queue.

The search process begins by trying to assign a consistent choice for the top decision start node on the LIFO queue. If a consistent value for the first decision start node is found, it is popped from the queue, and getNextDecisionStartNodes() is called with the first decision start node as the argument. Any decision node(s) that are returned from the function are pushed onto the LIFO queue, and the process is repeated until the LIFO queue is emptied. An empty LIFO queue means that the search is complete and a full and consistent assignment is made to the TPN. This process is illustrated in Figure 3, Part A. Also in Part A, it is shown that the first assignment to decision start node 2 is inconsistent. This brings us to the function #2, deactivateNestedDecisionStartNodes().

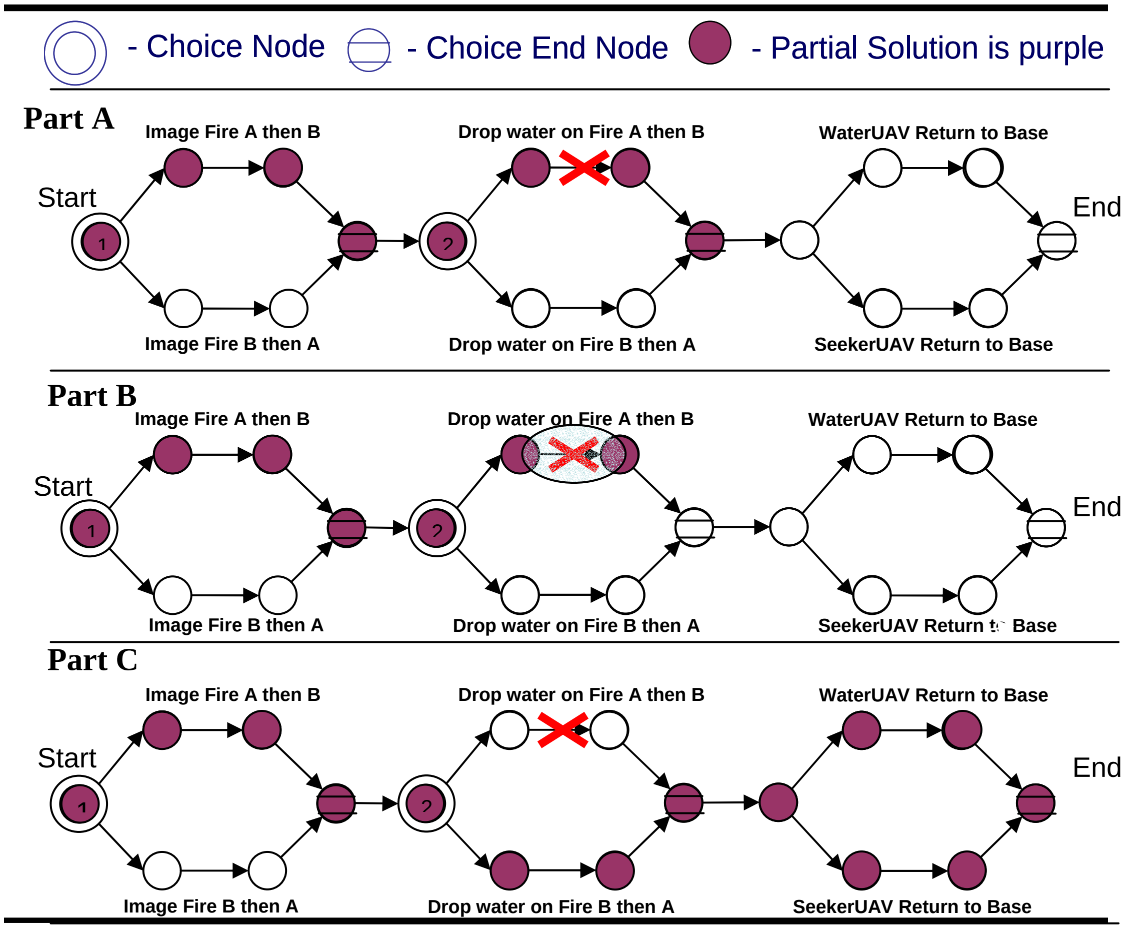


Figure 3: Searching a TPN as a Conditional CSP

2.) deactivateNestedDecisionStartNodes()

Suppose, for example, in Figure 4, Part A, that there were nested decision nodes within the “Drop water on Fire A then B” command. Also suppose that some of those decision nodes have already been assigned choices, and are a part of the partial solution {P} of the conditional CSP. If the assignment to decision start node #2 is changed, these nested decision nodes (represented by the shaded oval in Figure 4, Part B) need to be removed from the partial solution {P}.

This is accomplished by calling deactivateNestedDecisionStartNodes() whenever the assignment to a decision node is changed. Once the inconsistent choice for decision # 2 has been retracted (and all nested decision nodes removed from {P}) the next outarc for decision node #2 can be selected, as shown in Figure 4, Part C. The function getNextDecisionStartNodes() is called on Decision Node #2, and returns an empty set. (since there are no more decision nodes in the graph) Therefore, the LIFO queue becomes empty, and the TPN has a full and consistent assignment.

3.c) Pseudo-code and function explanations:

```
getNextDecisionStartNodes( currentNode )
{
   if ( currentNode = decisionStartNode)
         return currentNode
   else
      for all outarcs[i] of currentNode,
         map_o_startNodes.insert( getNextDecisionStartNodes(endnode->outarcs[i]))
      return map_o_startNodes
}
```

Figure 4: getNextDecisionStartNodes() Pseudo-code

The getNextDecisionStartNodes(currentNode) function in Figure 3 works recursively by immediately returning when a decisionStartNode is encountered, and recursively calling itself on all outarcs of any other type of node. All decision start nodes that are returned from the recursive calls to getNextDecisionStartNodes() are inserted into a map. Using a map ensures that even if a decision start node is encountered by the recursive function more than once while walking the TPN, only one instance of the decision start node is returned by the function.

```
deactivateNestedDecisionStartNodes( currentNode )
{
   for all decisionStartNodes[i] between currentNode->start  and currentNode->end
         if ( decisionStartNode[i] ∈ {P} )  // where {P} is the partial solution
            remove decisionStartNode[i] from partial solution {P}
}
```

Figure5: deactivateNestedDecisionStartNodes() Pseudo-code

The deactivateNestedDecisionStartNodes() function works by considering all decisionStartNodes inbetween the current decisions start and end nodes, and if necessary removes them from the partial solution {P}. The actual implementation of this function is a-bit more detailed, but the overall idea is captured above.

4.) Dynamic Backtracking Implementation within Kirk

a.) Overall Description of Dynamic Backtracking

Dynamic Backtracking is an efficient search algorithm that allows dynamic variable ordering, and also retains partial solutions to a CSP whenever they are not part of an identified conflict. To work effectively, the dynamic backtracking algorithm needs a conflict detection sub-routine (i.e. the incremental temporal consistency checker in Kirk). The dynamic backtracking algorithm is proven to terminate, and is also complete, and requires $O(i^2v)$ memory to remember conflict information, where (i) is the number of variables, and (v) is the largest domain size of the variables (i).

b.) Pseudo-code

The pseudo-code below is a description of the dynamic backtracking algorithm that was implemented within the Kirk architecture, and was obtained from:

Verfaillie, Gerard and Schiex, Thomas "Dynamic Backtracking for Dynamic Constraint Satisfaction Problems" ONERA-CERT, Toulouse Cedex, France.

A short description of each function is given, and the modification necessary to consider searching a conditional CSP using dynamic backtracking is described at the end.

```
dbt( csp )
{
   let V be the set of the variables
   return dbt-variables(∅ , V )
}
```

dbt(csp)
The overarching function call for dynamic backtracking, returns solution or failure

```
dbt-variables( $V_1$ , $V_2$ )
{
   ; $V_1$ is a set of assigned variables
   ; $V_2$ is a set of unassigned variables
      if $V_2 = \emptyset$
      then return success
      else let v be a variable chosen in $V_2$
           let d be its current domain
           if dbt-variable($V_1$,v,d) = failure
           then return dbt-bt-variable($V_1$,$V_2$,v)
           else return dbt-variables($V_1 \cup \{v\}$, $V_2 - \{v\}$)
}
```

dbt-variables(V1 , V2)
The main recursive function that assigns a new variable if assignment is consistent, and calls the backtrack function if a new variable assignment is not consistent.

```
dbt-variable( $V_1$ , v , d )
{
   if d = ∅
   then return failure
   else let val be a value chosen in d
        if dbt-value(V1,v,val) = success
        then return success
        else return dbt-variables(V1,v,d – {val})
```

dbt-variable(V1,v,d)
   tries to assign a variable one of the value v in it's domain, either returns success, or recursively tries again with the discounted domain, and returns failure if no domain left

```
dbt-value( $V_1$ , v , val)
{
   assign-variable(v,val)
   let be c = backward-checking(V1,v)
   if c = success
  then return success
  else let $V_3$ be the set of the variables of c
        unassign-variable(v)
        create-eliminating-explanation(v,val,$V_3$-{v})
        return failure
}
```

dbt-value(V1, v , val)
   called by dbt-variable() to test if the assignment of value (v) to variable (i) is a success

```
dbt-bt-variable( $V_1$ , $V_2$ , v)
{
   let V3 be the conflict set of v i.e., the union of the
   eliminating explanations of all its eleiminated values
   if V3 = ∅
   then return failure
   else let v' be the last variable of $V_3$ in $V_1$
        let val' be its current value
        let $V_4$ be the set of variables following v' in $V_1$
        unassign-variable(v')
        create-eliminating-explanations(v',val',$V_3$-{v'})
        remove-eliminating-explanations(v',$V_4 \cup V_2$)
        dbt-variables($V_1$-{v'},$V_2 \cup$ {v'})
}
```

dbt-bt-variable(V1 , V2 , v)
   Does the necessary clean-up in the elimination explanation database, and also the partial solution {P} when a variable's domain is wiped-out and the algorithm must backtrack.

Modification to search a conditional CSP using Dynamic Backtracking:

In the paper cited above, there is also an algorithm ddbt() (stands for dynamic dynamic backtracking) describing how to search a dynamic CSP using dynamic backtracking (conditional and dynamic CSPs are equivalent in this sense) The only difference between conditional CSPs and dynamic CSPs is that in a dynamic CSP, variables are added to the problem randomly(i.e. queueing network), and in a conditional CSP, one knows before hand which assignments to variables cause other variables to enter and leave the problem(i.e. a TPN).

```
ddbt( $V_1$ , $V_2$ , AC , RC)
{
  ; $V_1$ is the ordered set of assigned variables,
  ; result of the previous search
  ; $V_2$ is the set of unassigned variables,
  ; result of the same search
  ; AC is the set of added constraints
  ; RC the set of removed constraints
      let be $V = V_1 \cup V_2$
      let be $V_3$ = remove-assignments($V_1$, AC)
      remove-variable-eliminating-explanations(RC,V)
      return dbt-variables($V_1$-$V_3$,$V_2 \cup V_3$)
}
```

ddbt(V1, V2 , AC , RC)

The basic idea of this function is simple: if a variable is removed from the problem, remove any of the constraints that depend on that variable from the problem as well. (which just means erasing any elimination explanations depending on the variable, since the timing constraint arcs in Kirk (which represent constraints to be removed) are “disabled” automatically in Kirk and effectively disappear when the variable is not supposed to be in the problem) And if a variable is added to the problem, make sure any constraints involving that variable are re-added to the problem. This is also simple in the context of Kirk, because when a variable is added into the problem it’s constraints (the timing constraint arcs) are inherently added in as well.

3c.) Insights from Testing Dynamic Backtracking on Representative TPNs

I debugged my dynamic backtracking implementation in Kirk, by testing the algorithm against three fairly interesting and representative TPN structures. There are two main insights to be gained from these tests. (other than the fact that my implementation works ☺)

1.) Dynamic Backtracking could be very useful in terms of re-planning, because dynamic backtracking essentially holds onto consistent partial plans as long as possible. (i.e. less destructive to original plan)

2.) The nature of conflict extraction in a TPN relies on negative cycles in the graph. In many cases, these negative cycles traverse a large portion of the TPN, and the conflict returned is far from minimal. (i.e. many decisionNodes are returned in the conflict-set that aren't actually the source of the conflict) In these cases, Dynamic Backtracking can't perform much better than conventional backtracking.

The three test TPNs are described below:

(In each of the follwing testTPN's the .rmpl file was written such that the TPNs search from top to bottom according to the figures drawn.)

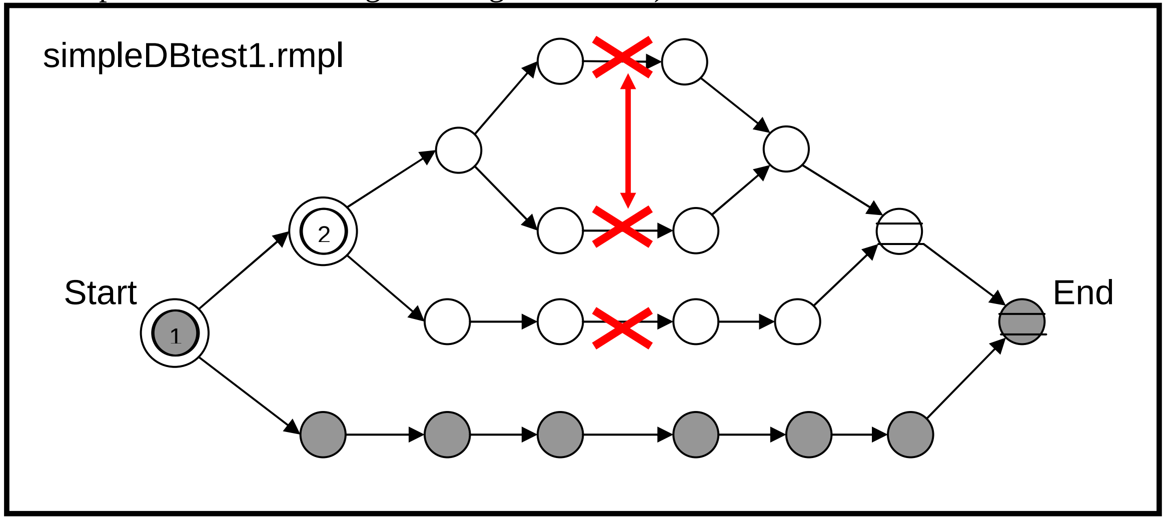


Figure 6: simpleDBtest1.rmpl

In this TPN, the top parallel branch was assigned conflicting parallel activities, and the middle branch (belonging to choice node #2) contains an inconsistent primitive. This TPN was used to test the getNextDecisionStartNode() and deactivateNestedDecisionNodes() functions since choice node #2 must be removed from the problem when choice node #1 switches its assignment to the lower path.

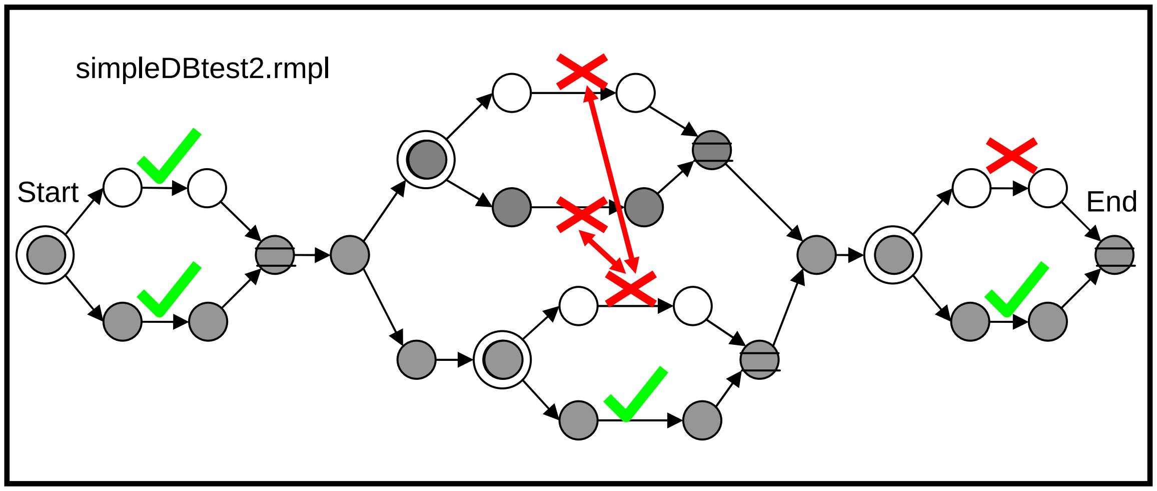


Figure 7: simpleDBtest2.rmpl

This TPN is not much different from the first, but requires the TPN to back up to a decision node that was found during the same call to getNextDecisionStartNodes( ) (and neither of them were nested decision nodes) which brought out a subtle bug in the algorithm that had to be fixed. Namely, if a decision start node is not a nested decision node, even when backtracking from it, it cannot be deleted from the LIFO queue. This is because any variable that is not nested is required to have an assignment before the TPN is assigned a complete solution.

Remembering that Kirk has to search from the top down, Kirk had to come across several inconsistencies in Figure 7 before finding the consistent solution.

(third TPN is on the next page)

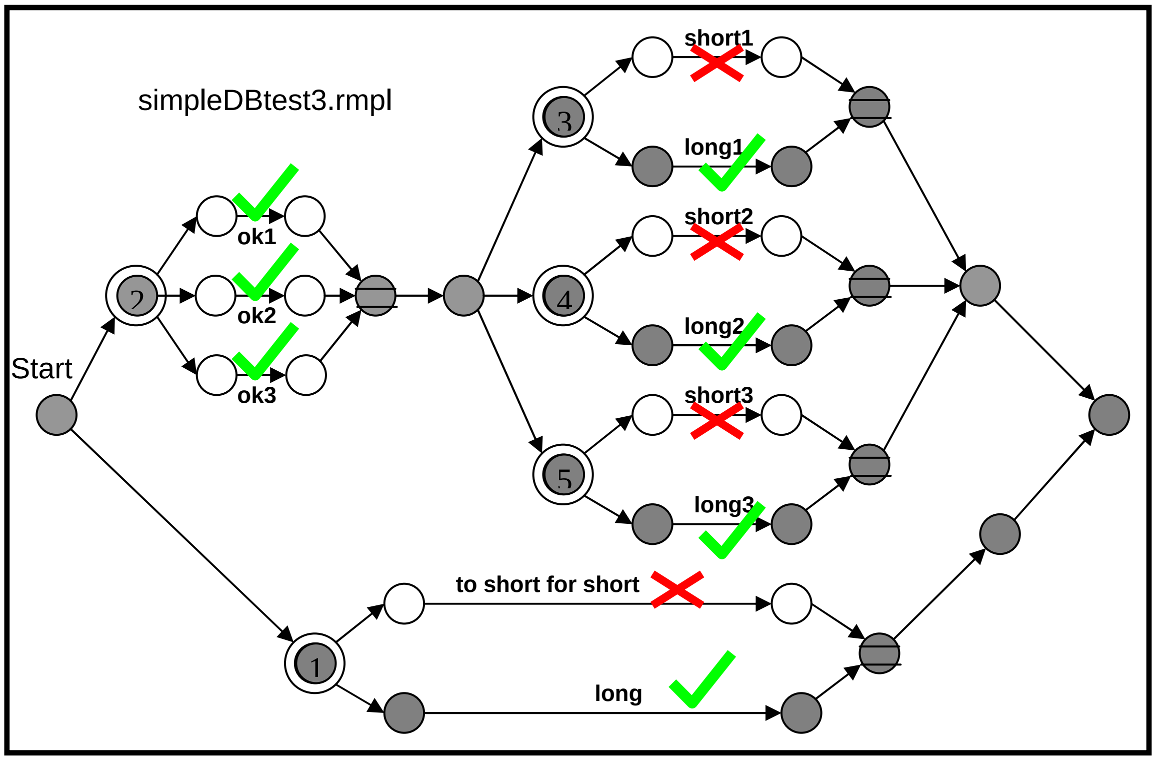


Figure 8: simpleDBtest3.rmpl

This TPN brings out an interesting issue with conflict-extraction in Kirk. I'm not certain if minimal conflict extraction can be done in Kirk, and if not, this could potentially degrade the performance of dynamic backtracking. Since in Kirk, the conflict is extracted as a negative cycle in the graph, in this example the conflicts involving the primitives "shortX" and "too short for short" can only currently extract the conflicts {1,2,3} , {1,2,4} , {1,2,5} where none of these are minimal conflicts, since the assignment to {2} is obviously consistent.

In this example, the decision nodes are instantiated in the order of their indices. This means that if Kirk were able to give the dynamic backtracking algorithm one of the minimal conflicts {1,3} , {1,4} , {1,5} then it would know to skip over reassigning node #2 since it is a partial assignment that was not a part of the conflict. However, at present, the non-minimal conflict causes dynamic backtracking to fare equally as poorly as chronological backtracking in this case.

I believe, that in general, the conflict-detection capability provided by dynamic backtracking will cause Kirk to perform better on large TPNs (even if the conflicts extracted are non-minimal). However, if there are methods to extract minimal conflicts in this context, it appears that they may be very useful.

5.) Interfaces with the Generative and Roadmap Path Planners

The interfaces with the generative and roadmap Path Planners are fairly straightforward.

5a.) Generative Path Planner

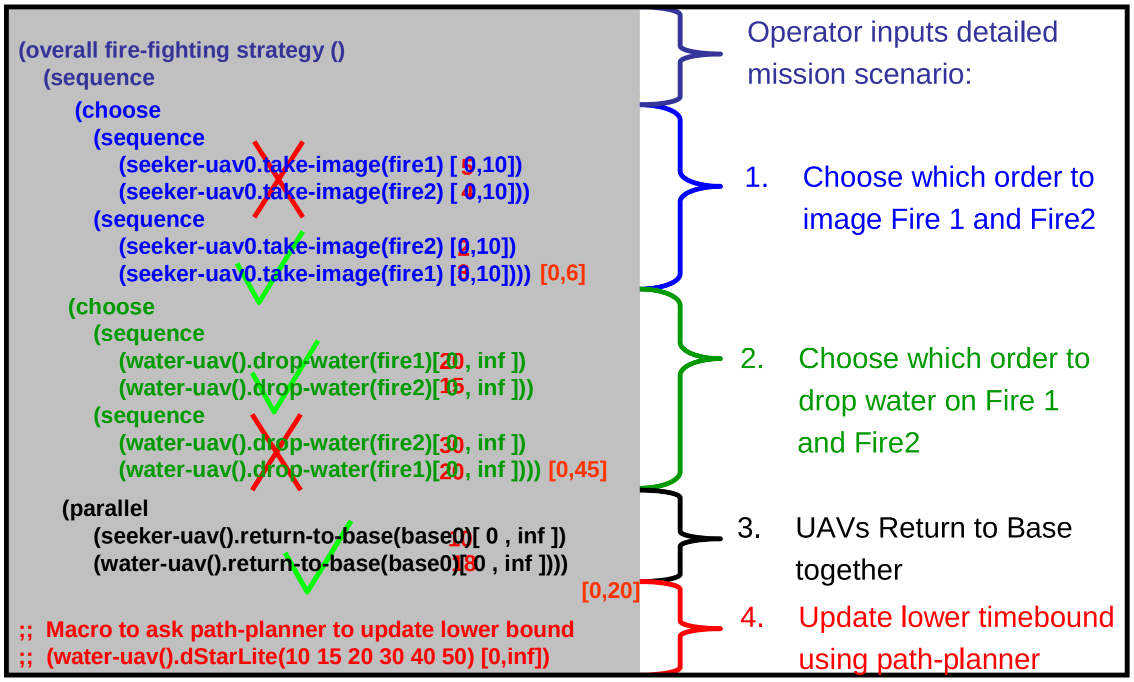
There are two options for the interface of Kirk with the Generative Path Planner.

1. The first is to directly pass the pointer to the TPN object after Kirk finds a consistent and full assignment.

2. Create a simple XML parser that can output from Kirk and be read in by the generative path planner.

5b.) Roadmap Path Planner

Kirk, the high-level planner, interfaces with the Roadmap Path Planner in order to get a lower bound on the time it might take to perform an activity. This allows the planner to preemptively prune out inconsistent portions of the search space.



```
(overall fire-fighting strategy ()
   (sequence
       (choose
         (sequence
            (seeker-uav0.take-image(fire1) [ 0,10])
            (seeker-uav0.take-image(fire2) [ 0,10]))
         (sequence
            (seeker-uav0.take-image(fire2) [0,10])
            (seeker-uav0.take-image(fire1) [0,10]))) [0,6]
      (choose
         (sequence
            (water-uav().drop-water(fire1)[20, inf ])
            (water-uav().drop-water(fire2)[15, inf ]))
         (sequence
            (water-uav().drop-water(fire2)[30, inf ])
            (water-uav().drop-water(fire1)[20, inf ]))) [0,45]
     (parallel
         (seeker-uav().return-to-base(base0)[ 0 , inf ])
         (water-uav().return-to-base(base0)[0 , inf ])))
                                                      [0,20]
;;  Macro to ask path-planner to update lower bound
;;  (water-uav().dStarLite(10 15 20 30 40 50) [0,inf])
```

Operator inputs detailed mission scenario:
1. Choose which order to image Fire 1 and Fire2
2. Choose which order to drop water on Fire 1 and Fire2
3. UAVs Return to Base together
4. Update lower timebound using path-planner

Figure 9: Calling the Roadmap Path Planner to Update Activitiy Lower Bounds

For example, in the RMPL file shown above, the red numbers represent updated lower bounds obtained for each activity from the roadmap path planner. In the first task, “choose which order to image Fire1 and Fire2”, the roadmap path planner tells Kirk that the quickest possible time to image Fire1 and then Fire2 is 5 + 4 = 9. As shown in orange, however, the maximum amount of time allotted for this activity is 6. Therefore, this sequence of actions is inconsistent and can be pruned from the search space.

6.) Branch and Bound with Dynamic Backtracking for Anytime and Optimal Search

Adding the ability to estimate the “cost” of a partial solution within Dynamic Backtracking and the Kirk architecture appears to be very straightforward, and could provide an anytime search algorithm that outputs an optimal planning solution.

Once a consistent assignment is found to the TPN, this solution (and it’s estimated cost) is kept as the incumbent solution. The algorithm continues to search partial solutions to the problem, backtracking whenever the cost estimate of the partial solution exceeds that of the incumbent solution. This provides an anytime algorithm that can update the quality of it’s solution, and output the optimal solution once the entire TPN has been searched in this “branch and bound” fashion.

7.) Summary and Conclusions

- Search as a Conditional CSP is implemented in Kirk using 2 functions and a Q:
    - getNextDecisionStartNodes( )
    - deactivateNestedDecisionNodes( )
    - LIFO Queue (a stack)

- Dynamic Backtracking is implemented in Kirk
    - implementation works (solves the three example TPNs in < 0.01 seconds)
    - completeness and termination guaranteed, $O(i^2v)$ memory
    - dynamic backtracking useful in terms of re-planning, because partial and consistent plans are kept around (i.e. less destructive to original plan)
    - The nature of conflict extraction in a TPN relies on negative cycles in the graph. Tese negative cycles are often from minimal. In these cases, Dynamic Backtracking can't perform much better than conventional backtracking.
    - Dynamic Backtracking probably performs better on very large problems, even if the conflict extraction mechanism is non-minimal, since on large problems any conflict-detection is better than none (because of the ugly nature of the exponential)

- Interface with Roadmap Path Planner and Generative Planner
    - Roadmap Path Planner interface is in place and working
    - Generative Planner interface is agreed upon, but not completed

- Implementing a Branch-and-Bound incumbent solution approach would allow an anytime algorithm that could systematically search for an optimal solution to the TPN.