A Focussed Dynamic Path Finding Algorithm with Constraints

Louise Leenen
Cyber Defence Research Group
Council for Industrial and Scientific Research
PO Box 395, Pretoria 001
Email: lleenen@csir.co.za

Alex Terlunen
Smart Systems Research Group
Council for Industrial and Scientific Research
PO Box 395, Pretoria 001
Email: aterlunen@csir.co.za

***Abstract*—The Military Unit Path Finding Problem (MUPFP) is the problem of finding a path from a starting point to a destination where a military unit has to move, or be moved, safely whilst avoiding threats and obstacles and minimising path cost in some digital representation of the actual terrain [1]. The MUPFP has to be solved in an environment where information can change whilst the optimal path is being calculated, i.e. obstacles and threats can move or appear and path costs can change.**

In previous work, the authors formulated the MUPFP as a constraint satisfaction problem (CSP) where path costs are minimised whilst threat and obstacle avoidance constraints are satisfied in a dynamic environment [2]. In this paper the previous algorithm is improved by adding a heuristic to focus the search for an optimal path. Existing approaches to solving path planning problems tend to combine path costs with various other criteria such as obstacle avoidance in the objective function which is being optimised. The authors' approach is to optimise only path costs while ensuring that other criteria such as safety requirements, are met through the satisfaction of added constraints. Both the authors' previous algorithm and the improved version presented in this paper are based on dynamic path planning algorithms presented by Stenz [3], [4]. Stenz's original D* algorithm solves dynamic path finding problems (by optimising path costs without satisfying additional constraints) and his Focussed D* algorithm employs a heuristic function to focus the search. Stenz's algorithms only optimises path costs; no additional factors such as threat and obstacle avoidance are addressed.

***Keywords* —optimisation, path finding, constraint programming, dynamic A* search**

I. Introduction

Path planning algorithms calculate an optimal path for an object from a start point to a destination point, whilst avoiding obstacles and minimising costs. It has many applications such as computer games, transportation, robotics, networks, and others. In a static algorithm, the environment is known and an optimal path is calculated before the object is moved. In a dynamic algorithm, the environment may not be completely known before the object starts moving, or it may change whilst the object is moving. In this case, the path plan has to be updated while it is being executed. An overview of methods to solve the path finding problem is given by Leenen et al. [2].

The modelling of problems in terms of constraints has the advantage of a natural, declarative formulation. A CSP formulation of a problem states what must be satisfied, without specifying how it should be satisfied. It consists of a set of variables, a set of domains for the variables, and a set of constraints. Each constraint is defined over a subset of the set of variables. A constraint is a logical relation involving one or more variables, where each variable has a domain of possible values. A constraint thus restricts the possible values that variables can have. Constraints can specify partial information about variables, are declarative and may be non-linear. A solution to a CSP specifies values for all the variables such that all the constraints are satisfied.

In the military unit path finding problem, a military unit aims to avoid obstacles and threats, or to pass threats with at least a minimum distance between them and the threat, while moving to their destination as quickly as possible. In the MUPFP a balance has to be maintained between the two main criteria, route speed and safety [1]. Although there are various methods to solve the MUPFP, the existing approaches combine various criteria in a weighted objective function which is optimised. Our CSP approach, on the other hand, is to optimise only path costs while ensuring that certain other criteria, such as safety requirements, are met. Our objective function is a pure cost function. We adopt a constraint-based approach with a clear distinction between the goal of obtaining an optimal path cost and satisfying safety measures. A constraint programming (CP) approach allows for flexibility in terms of modeling different constraints. If a new requirement has to be met, the addition of a suitable constraint or the modification of existing constraints will suffice to model the new problem.

In previous work [5] we formulated the MUPFP as a CSP and modified the D* algorithm (dynamic A* search) [3] by adding threat and obstacle avoidance constraints. The D* algorithm builds an optimal path by repairing potential optimal paths in every step as new information becomes available. The Focussed D* algorithm [4] focusses the search for an optimal path with an heuristic function to reduce the graph expansion and thus the total calculation time required. In this paper we extend our previous algrorithm by adding a heuristic similar to that used in the Focussed D* algorithm.

There are many general techniques that can be used to solve CSPs such as integer programming, local search, and neural networks, but tree search in conjunction with backtracking and consistency checking is widely used. CP refers to the computational systems used to solve CSPs. It emerged from a number of disciplines such as Artificial Intelligence, Computational Logic, Programming Languages, and Operations Research. CP has proven to be effective at solving combinatorial and overconstrained problems. Several texts provide an introduction to CP and CSPs [6], [7].

In Section II, the formulation of the MUPFP as a CSP is described and an overview of Stenz's Focussed D* algorithm is given. In Section III, our improved algorithm is presented and Section IV shows an example.

II. Background

The formulation of the MUPFP as a CSP was initially presented by Leenen et al. [2]. We also give an overview of Stenz's Focussed D* algorithm [4].

A. The Military Unit Path Finding Constraint Satisfaction Problem

A constraint-based approach to the military path finding problem offers flexibility in an environment that can effectively be represented by constraints, where new information can be added with ease. The terrain map is presented as a graph. A node represents a geographical location and an edge represents a route between two nodes, i.e. two geographical areas. The cost of an edge represents the difficulty of moving through a particular part of the actual route represented by that edge. We represent the avoidance of danger through constraints. An example of a threat is a sniper whose presence at a certain node is known. In this formulation we include constraints to disallow movement to a node where there is a known obstacle or a threat, and constraints that disallow movement within a certain distance of a node containing a known threat. Threats and obstacles are dynamic in the sense that they can move or disappear. Edge costs can also change; if a route becomes more difficult to traverse, for instance due to muddy conditions, then the representing edge costs will be increased.

Consider the graph with a set of nodes, $N = \{x_1, x_2, \ldots, x_k\}$ and a set of edges, $A = \{(x_i, x_j) | x_i, x_j \in N\}$. Every edge in A, (x_i, y_j), has an associated cost value, $c(x_i, y_j)$. The function $status$: $N \rightarrow \{O, T, U\}$ defines whether there is an obstacle (O) or a threat (T) present at a node, or if the node is unoccupied (U). In the latter case there is no known obstruction. Let the set of variables in the solution path be $V = \{V_1, V_2, \ldots, V_n\}$, where each variable represents one node in the solution path. A path is a sequence of nodes in a graph such that there is edge from each node in the sequence to the next node in the sequence. The domain of each variable is $Dom = \{x_1, x_2, \ldots, x_k\}$. The set of constraints, C, contains at least the ones listed below:

- C1: An all-different constraint, $V_1 \neq V_2 \neq \ldots \neq V_n$ ensures each node in the solution path is unique.
- C2: $V_1 = s_{start}$ & $V_n = s_{goal}$ with s_{start} the starting node in the path and s_{goal} the destination node.
- C3: For every $V_{i+1}, i = 1, \ldots n-1$, if $V_i = x_j$ then $V_{i+1} = x_m$ if and only if there exists an edge $(x_j, x_m) \in N$. This constraint ensures that the value assignment of the variables forms a path.
- C4: For every $V_i, i = 2, \ldots n-1, V_i \neq x_j$ if $status(x_j) = \{O, T\}$. This constraint disallows a node containing a threat or an obstacle to form part of the solution path.
- C5: For every $V_i, i = 2, \ldots n-1, V_i \neq x_j$ if there exists an edge (x_i, x_j) such that $status(x_j) = \{T\}$. This ensures that the solution path does not venture too close to a threat. We model a safe distance to be more than a single edge, i.e. a node that is a neighbour of a threat cannot be included in the optimal path.

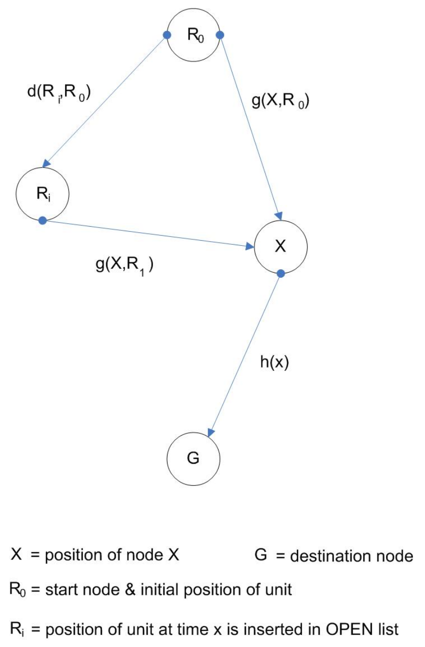
The solution is an assignment of values for the variables in the set V such that $min\ \Sigma_{s=1,\ldots n-1} c(x_i, y_j)$ where $V_s = x_i$ and $V_{s+1} = y_j$, and all constraints are satisfied.

A CSP has a finite, known number of variables. We thus have to decide on a value for n (number of decision variables in the CSP) before solving our CSP. The D* algorithm calculates an optimal solution path and then retraces this path while testing for changes in cost values. If the initial solution path contains m edges, then we assign values to $n = m + 1$ variables. The value of n may change if there had been cost changes and another execution of the algorithm is required.

B. Overview of Stenz's Focussed D Algorithm

Stenz introduced a D* algorithm for solving dynamic path finding problems in 1994 [3] and a Focussed D* algorithm in 1995 [4]. A brief overview of the latter algorithm follows. The algorithm initially calculates an optimal path from the destination node, G, back to the start node, S, using a focussing heuristic function. The optimal path can then be traversed from the start point until new information is received and adjustments are required. Replanning is then done and the new optimal path is calculated from the current point of traversal. The current position of the military unit is at node R, and initially R is equal to S.

Similar to the D* algorithm, Focussed D* maintains an OPEN list of states for expansion: RAISE states and LOWER states. A RAISE state means the path cost has increased and this information has to be propagated, while a LOWER state indicates that there may have been a path cost reduction. When a state is removed from the OPEN list, it is expanded to pass cost changes to its neighbours which are placed on the OPEN list in turn. For each node X, an estimate of the total sum of the edge costs from the node to the destination node, $h(X)$, is maintained as well as a focussing heuristic, $g(X, R)$, that estimates the path cost from node R (the current position of the military unit) to the node X. For each node X, an estimate of the total path cost from R to G via X: $f(X, R) = h(X) + g(X, R)$ is maintained. Ideally, this estimate is equal to the minimum cost from the start node via node X to the destination node. The algorithm also maintains the key value, $k(X)$, of each node. The key value of a node X is the minimum of its $h(X)$ value before possible modifications to map values, and all its $h(X)$ values since the node X was first placed on the OPEN list. $k(x)$ identifies X as either a RAISE state or a LOWER state. The algorithm keeps track of current best paths by setting back-pointers from each node *back* to its current predecessor in the solution path. The OPEN list is sorted according to a biased f value for a node x, $f_B(x) = f(x, R_i) + d(R_i, R_0)$. The d function is called the accrued bias function and $d(R_i, R_0)$ keeps track of the cost of movement of the military unit from its initial position at R_0 to R_i (the unit's position at the time a node x was inserted on the OPEN list). Ties in f_B values are resolved by taking the minimum f value, and ties in f values by k values.



R0
d(R i,R 0)
g(X,R 0)
Ri
X
g(X,R 1)
h(x)
G
X = position of node X
G = destination node
R0 = start node & initial position of unit
Ri = position of unit at time x is inserted in OPEN list

Fig. 1. An illustration for the focussing heuristic

To summarise, the military unit starts at some position R_0 and the estimated total path cost is given by $f(X, R_0)$. After an initial optimal path has been calculated the unit starts traversing this optimal path until it reaches some position R_i. Sensors may detect some changes to the graph values and recalculate the optimal path from the current position R_i to the destination node at G. For every node X that is added to the OPEN list from this stage, the estimated total cost from R_i to G is given by $f(X, R_i)$, but $f_B(X, R_i) = f(X, R_i) + d(R_i, R_0)$ represents the distance the unit has already travelled from R_0 to R_1 added to $f(X, R_i)$. This avoids having to recalculate these values for all the nodes that were inserted on OPEN before the unit moved to R_i. The true path costs are calculated at the time of expansion. Figure 1 illustrates these concepts.

The first phase of the algorithm halts when the current node at which the military unit is positioned, is removed from the OPEN list. The backpointers are then followed from the current position of the unit to the destination node. If any information has changed, the traversal is halted and the algorithm recalculates an optimal path from the node where the discrepancy has been identified to the destination node.

III. The Focussed Constraint D* Algorithm

In this section, we present an algorithm to solve the MUPFP which improves on our previous constraint-based algorithm by using a heuristic to focus the search for an optimal path. In our solution approach, we model constraints to avoid obstacles and to keep a specified distance from known threats. For simplicity, we model this distance to be "one node away", i.e. our path will not include any node that is a neighbour of a threat node. When we consider the list of neighbours of a particular node, x, for expansion, we check if a neighbour node, y, is an obstacle or a threat (i.e. we perform constraint checking).

Our algorithm calls a function, *ComputeOptimalPath*, to compute an optimal path for the current graph whilst satisfying safety constraints, and then it enters a loop which traverses the optimal path until changes in the environment are detected: the cost of an edge has been modified, or the status of a node has been modified. The *ComputeOptimalPath* function computes an optimal path in a similar way as the Focussed D* algorithm, but with the addition of constraint checking. The main algorithm, called *FConstraint_DStar*, allows the military unit to traverse this path as long as no cost changes or status changes are observed. As soon as one or more changes have been detected, traversal will be halted and corrections made. The *ComputeOptimalPath* function is then called to calculate a new optimal path from the unit's current position.

In the case of a cost change, we follow steps similar to such an occurrence in the Focussed D* algorithm by identifying nodes to be reinserted into the OPEN list. Various values of nodes which may be affected by the change in cost, are adjusted. In the case of a node whose status has changed, the algorithm does the necessary checks to identify nodes that have to be reevaluated by calling the function *StatusChange* in which one of the following cases are identified:

- The status of a node s changed from an obstacle or unoccupied to that of a threat. s cannot be included in an optimal path and *RevertPath* is called for each of its neighbours, x. This function traces expanded paths through the node x and makes adjustments to predecessor nodes. This is necessary because the node x is now the neighbour of a threat node and may not be included in a current best path to any node from the goal node. These paths need to be destroyed.
- If the status of a node s changed from unoccupied to an obstacle, then *RevertPath* is called to ensure that s is not included in a best path from the goal node to any other node.
- If the status of a node s changed from an obstacle to unoccupied, then the neighbour nodes of s are inserted into the OPEN list if they qualify for consideration.
- The status of a node s changed from a threat to either unoccupied or an obstacle. In this case the neighbours of s had not been considered for expansion prior to the status change, so the neighbour nodes are now considered for insertion into the OPEN list.

We now describe the algorithm in greater detail. The existence of the following information and data structures are assumed:

A CSP with n number of variables, a set of constraints C, and a set of domain values Dom. A graph with a set of nodes, N, a set of edges, A, a designated source node, s_{start}, and a designated destination node, s_{goal}. $OpenQ$ is a priority queue that represents the OPEN list of nodes to be expanded, sorted in non-decreasing order according to f_B values. R_{curr} is the current state (of the unit) on which the search is focussed and d_{curr} is the accrued bias from the unit's start state to its current state. Every node $x \in N$ has the following labels: $status(x)$, $h(x)$, $k(x)$, $f(x)$, $f_B(x)$, $r(x)$ and $b(x)$. $b(x)$ is a backpointer that can point to another node in N; $r(x)$ is the position of the military unit at the time x is inserted into $OpenQ$; $h(x)$ is the estimated path cost from x to s_{start}; $k(x)$ is the key value; $f(x)$ is the estimated path cost from R_{curr} via x to

s_{goal}; and $f_B(x)$ is the biased estimated total path from cost R_i to the destination node via x where R_i is the military unit's position at the time x was inserted onto $OpenQ$. Every edge $(x, y) \in A$ has an associated cost, $c(x, y)$ and $s(x, y)$ is the cost of the edge returned by a sensor. A function, $status : N \to \{U, O, T\}$ where U is an unoccupied node, T a threat and O an obstacle. A labelling function, $t : N \to \{NEW, OPEN, CLOSED\}$. NEW indicates that a node has not yet been expanded, $OPEN$ that the node is a member of the OPEN list, and $CLOSED$ that the node has been considered and removed from the OPEN list. $PutOn$ is a set of nodes that may have to be returned to $OpenQ$ after a status change in a node.

Neither the start node nor the destination node is an obstacle, a threat, or the neighbour of a threat. The following functions are called in the algorithm:

$ComputeOptimalPath(val)$ computes an optimal path from the source node to the destination node while satisfying the threat and obstacle constraints. The parameter val is a global variable. $ConstraintCheck(s)$ performs the constraint checking: it checks whether any neighbour of the node s is an obstacle, a threat, or the neighbour of a threat. $StatusChange(s, NewStatus)$ is called when a node s is identified to have received a new status value. It makes the necessary modifications to variables such that a new optimal path can be computed and calls the two repair functions $RevertPath$ and $DangleNode$ if necessary. Note that the second parameter of $RevertPath$ is optional. $DangleNode(s)$ and $RevertNode(s, i)$ are repair functions that are invoked when a node's status has changed and path repairs are required. $Initialise()$ initialises variables and inserts the goal node into $OpenQ$. $Neighbours(s)$ returns all s's neighbours. $InsertOnQpenQ(s, value)$ insert a node s into the queue $OpenQ$ after calculating its new k, f and f_B values. $MinState(val)$ returns the node on the $OpenQ$ with the minimum f value. If the military unit has moved since the insertion of the node on $OpenQ$, required adjustments are made. The ordered pair val contains the f and k values of the returned node. $ModifyCost(x, y, NewCost)$ is called when the cost of an edge (x, y) has been changed. It returns relevant nodes to $OpenQ$. $Cost(v)$ computes the estimated minimal path cost, $f(v, R_{curr})$, from the unit's current position, R_{curr}, via the node v to the destination node. It returns an ordered pair of values. $GVal(x, y)$ is a user provided function that computes the focussing heuristic $g(x, y)$ which is an estimate of the path cost from the node x to the node y. $LessEq(A, B)$ and $Less(A, B)$ compare ordered pairs of values. $Enqueue(Q, s, v)$ adds a node s with a value v to a priority queue Q and $Dequeue(Q)$ removes and returns the first element from a priority queue, Q.

The main function, *FConstraint_DStar*, is shown as Function 1. It calls *Initialise*, shown as Function 2. After computing an optimal path in line 2 of the main function, there is a *while* loop in lines 5-29 that runs until the optimal path has been traversed and the destination state has been reached. *ComputeOptimalPath* is similar to the first phase of Stenz's Focussed D* algorithm with the addition of constraint checking, and it is shown as Function 3. It calls *MinState*, *ConstraintCheck*, *LessEq*, *Less*, *Cost* and *InsertOnOpenQ* shown as Functions 4 to 9 respectively.

Function 1 FConstraint_DStar()

```
1:  Initialise();
2:  ComputeOptimalPath($val$);
3:  if $t(s_{start}) = NEW$ then
4:      return NoPath;
5:  while $R \neq s_{goal}$ do
6:      $DCurrUpdate = OtherUpdate = false$;
7:      if there is some edge $(x, y)$ for which $c(x, y) \neq s(x, y)$ then
8:          $OtherUpdate = true$;
9:          if $R_{curr} \neq R$ then
10:             $d_{curr} = d_{curr} +$ GVal$(R, R_{curr}) + \epsilon$;
11:             $R_{curr} = R$;
12:             $DCurrUpdate = true$;
13:         for all edges $(x, y)$ where $c(x, y) \neq s(x, y)$ do
14:             ModifyCost($x, y, s(x, y)$);
15:     if there is some node $s$ for which $status(s) \neq status\_sensor(x, y)$ then
16:         $OtherUpdate = true$;
17:         if $R_{curr} \neq R$ then
18:             if $DCurrUpdate = false$ then
19:                 $d_{curr} = d_{curr} +$ GVal$(R, R_{curr}) + \epsilon$;
20:                 $R_{curr} = R$;
21:         for all nodes $s$ such that $status(s) \neq status\_sensor(s)$ do
22:             StatusChange($s, status\_sensor$);
23:         for all $x \in PutOn$ do
24:             if $t(x) \neq NEW$ then
25:                 InsertOnOpenQ($x, h(x)$);
26:         $PutOn = \emptyset$;
27:     if $OtherUpdate = true$ then
28:         ComputeOptimalPath($val$);
29:     $R = b(R)$;
```

Function 2 Initialise()

```
$OpenQ = \emptyset$;
$PutOn = \emptyset$;
$R = R_{curr} = s_{start}$;
$d_{curr} = 0$;
for $s \in N$ do
    $h(s) = f(s) = f_B(s) = k(s) = \infty$;
    $t(s) = NEW$;
    $b(s) = NULL$;
InsertOnOpenQ($s_{goal}, 0$);
```

After an optimal path has been calculated, the *while* loop in line 5 (of Function 1) is executed. If a sensor detects that the cost of any edge (x, y) has been changed (line 7), *ModifyCost* is called in line 14 after adjustments have been made for possible partial traversal of a path by the military unit (lines 8-12). *ModifyCost* is shown in Function 10. The function checks if the nodes x and y are included in some best current path from the start to the goal. In this case, the ancestor node is reinserted into $OpenQ$ if it has a $CLOSED$ status. Otherwise, if either x or y is $CLOSED$ it also has to be re-inserted on $OpenQ$. Note that changes in edge costs or the status of any node are allowed at any time. These changes can be detected by sensors and compared to the stored values. $StatusChange$ (line 22 of the main function) is called when

Function 3 ComputeOptimalPath(val)

```
1:  while $t(s_{start}) \neq CLOSED$ AND $OpenQ \neq empty$ do
2:      s = MinState($val$);
3:      if $s \neq NULL$ then
4:          $k_{val} = k(s)$;
5:          Dequeue($OpenQ, s$);
6:          $t(s) = CLOSED$;
7:          $NSet$ = Neighbour($s$);
8:          for all $x \in NSet$ do {Remove any invalid nodes in loop}
9:              if ¬ ConstraintCheck($x$) then
10:                 $NSet = NSet - \{x\}$;
11:         if $k_{val} < h(s)$ then {a RAISE state}
12:             for all $x \in NSet$ do
13:                 if $t(x) \neq NEW$ AND LessEq($Cost(x), val$) AND
                    $h(s) > h(x) + c(x, s)$ then {path via x better than via s}
14:                     $b(s) = x$;
15:                     $h(s) = h(x) + c(x, s)$;
16:         if $k_{val} = h(s)$ then {path via s is optimal}
17:             for all $x \in NSet$ do {can path cost be lowered via x?}
18:                 if $(t(x) = NEW)$
                    OR $(b(x) = s$ AND $h(x) \neq h(s) + c(s, x))$
                    OR $(b(x) \neq s$ AND $h(x) > h(s) + c(s, x))$
                    then
19:                     $b(x) = s$;
20:                     InsertOnOpenQ($x, h(s) + c(s, x)$);
21:         else {s is a LOWER state}
22:             for all $x \in NSet$ do
23:                 if $(t(x) = NEW)$
                    OR $(b(x) = s$ AND $h(x) \neq h(s) + c(s, x))$
                    then
24:                     $b(x) = s$;
25:                     InsertOnOpenQ($x, h(s) + c(s, x$);
26:                 else if $b(x) \neq s$ AND $h(x) > h(s) + c(s, x)$
                    then
27:                     InsertOnOpenQ($s, h(s)$)
28:                 else if $b(x) \neq s$ AND $h(s) > h(x) + c(x, s)$
                    AND Less($val, Cost(x)$)
                    AND $t(x) = CLOSED$ then
29:                     InsertOnOpenQ($x, h(x)$)
30:     else
31:         $val = NoVal$;
```

Function 4 MinState(val)

```
1:  $s$ = Top element on $OpenQ$; {Assume OpenQ is not empty}
2:  if $r(s) \neq R_{curr}$ then
3:      $h_{new} = h(s)$;
4:      $h(s) = k(s)$;
5:      Dequeue($OpenQ, s$);
6:      $t(s) = CLOSED$;
7:      InsertOnOpenQ($s, h_{new}$);
8:      return NULL;
9:  else
10:     $val =< f(s), k(s) >$;
11:     return $s$;
```

Function 5 ConstraintCheck(s)

```
1:  if $status(s) = O$ then {is $s$ an Obstacle}
2:      return false;
3:  for $x \in$ Neighbour($s$) do
4:      if $status(x) = T$ then {is $s$ a 'Threat'}
5:          return false;
6:  return true; { All the other checks failed, so the node is safe}
```

any node s has had a change in its status, after adjustments have been made for possible partial traversal of a path by the unit (lines16-20). This function appears as Function 12. If the new status of a node s is that of a threat (lines 3-11 in *StatusChange*), then it cannot be considered for inclusion in any best path and neither can its neighbours. $RevertPath$ is called for every neighbour x of s to destroy partial best paths that include x. If the status of s changed from unoccupied to an obstacle (lines 12-14), $RevertPath$ is called to destroy partial best paths that include s. In the case of a status change from an obstacle to unoccupied (lines 15-18), all unoccupied neighbours that are not NEW are reinserted into $OpenQ$. Finally, in the case of a change from a threat to a non-threat or an obstacle, the complete neighbourhood of s had been invalid nodes prior to the status change. Thus we have to consider every node x which is a neighbour of s, and place all the neighbours of x into $OpenQ$ unless they are NEW or unoccupied. $RevertPath$ calls $DangleNode$ to identify invalid best path created by the status change. These functions are shown as Functions 11 and 13 respectively. Lines 23-26 of the main function consider nodes that were placed in the set $PutOn$ by calls to $DangleNode$. Nodes in this set have to be inserted into $OpenQ$ if they have any neighbour that is either $CLOSED$ or $OPEN$.

$ComputeOptimalPath$ removes a node s with a minimal f_B value from $OpenQ$ (line 5) and removes all its invalid neighbours from its neighbour set (lines 8-10). An invalid neighbour is a node that does not satisfy one of the safety constraints, i.e. it is an obstacle, a threat or the neighbour of a threat. In lines 11-15 we identify whether s is a RAISE state (its k value is smaller than its h value) i, or a LOWER

Function 6 LessEq(A, B)

$A =< a_1, a_2 >$ and $B =< b_1, b_2 >$

```
1:  if $a_1 < b_1$ OR ($a_1 = b_1$ AND $a_2 \leq b_2$) then
2:      return true;
3:  else
4:      return false;
```

Function 7 Less(A, B)

$A =< a_1, a_2 >$ and $B =< b_1, b_2 >$

```
1:  if $a_1 < b_1$ OR ($a_1 = b_1$ AND $a_2 < b_2$) then
2:      return true;
3:  else
4:      return false;
```

Function 8 Cost(v)

```
1: $f(v, R_{curr}) = h(v)$+GVal$(v, R_{curr})$;
2: return $< f(v, R_{curr}), h(v) >$;
```

state (its k value is larger than its h value) in lines 21-29. In these cases it propagates the changed costs. If s is optimal (its k value is equal to its h value) it expands the neighbours of s (lines 16-20). In line 3 of $StatusChange$, we identify that the node s has changes its status to that of a threat. This means that neither s nor any of its neighbours may be included in the best path from the goal node to any other valid node. If s is currently on $OpenQ$, it is removed (line 6). In lines 8-10, the function ensures that the invalidity of a neighbour node of s is recognised. If s has become an obstacle node, then lines 12-14 ensure that it (and paths through it) is made invalid. Lines 15-24 deal with a previously invalid node that has become valid and must now be considered for inclusion in the best paths from the goal to other nodes by inserting relevant nodes into $OpenQ$. The $RevertPath(s, i)$ function identifies and destroys partial best paths that include the node s which has become an invalid node. The parameter i is optional and is used to avoid the re-checking of a known invalid node, for example a node which is known to have become a threat. Lines 2-3 ensure that s is not on $OpenQ$. In lines 5-6 the ancestor neighbour of s is re-inserted on $OpenQ$ if necessary. In lines 8-10 we check for the existence of best paths that include the newly invalid node s. In this case, the function $DangleNode$ is called to destroy backpointers for nodes on such a path. In $DangleNode(s)$, node s forms part of a best path that has become invalid. If it is on $OpenQ$, it is removed (lines 2-3). The node s is initialised to NEW. The neighbours of s are checked for any paths through which it will also now be invalid (lines 7-10), or otherwise for a neighbour that may have to be reinserted into $OpenQ$ (lines 11-12).

Function 9 InsertOnOpenQ(s, h_{new})

```
1: if $t(s) = NEW$ then
2:     $k(s) = h_{new}$;
3: else if $t(s) = OPEN$ then
4:     $k(s)$ =min$(k(s), h_{new})$;
5:     Dequeue($s$);
6: else
7:     $k(s)$ =min$(h(s), h_{new})$;
8: $h(s) = h_{new}$;
9: $r(s) = R_{curr}$;
10: $f(s) = k(s)$+GVal$(s, R_{curr})$;
11: $f_B(s) = f(s) + d_{curr}$;
12: $t(s) = OPEN$;
13: Enqueue($OpenQ, s, f_B(s), f(s), k(s)$);
```

IV. Implementation and an Example

The algorithm has been implemented in $C++$ and QT (a graphical user interface). The user can generate a random graph by specifying the size and other details such as edge costs, obstacle and threat nodes and a start and destination node. The status of variables are shown and the user can opt to change any edge's cost or node's status at any time step. The example is based on the graph in Figure 2. The start node is node number

Function 10 ModifyCost(x, y, $NewCost$)

```
1: $c(x, y) = NewCost$;
2: if $b(y) = x$ then
3:     if $t(x) = CLOSED$ then
4:         InsertOnOpenQ($x, h(x)$);
5: else if $b(x) = y$ then
6:     if $t(y) = CLOSED$ then
7:         InsertOnOpenQ($y, h(y)$);
8: else
9:     if $t(y) = CLOSED$ then
10:        InsertOnOpenQ($y, h(y)$);
11:    if $t(x) = CLOSED$ then
12:        InsertOnOpenQ($x, h(x)$);
```

Function 11 RevertPath(s,i)

```
1: $h(s) = k(s) = f(s) = f_B(s) = \infty$;
2: if $t(s) = OPEN$ then
3:     Dequeue($OpenQ, s$);
4: $t(s) = NEW$;
5: if $b(s) \neq NULL$ AND $t(b(s)) \neq NEW$ then
6:     InsertOnOpenQ($b(s), h(b(s))$);
7: $b(s) = NULL$;
8: for $x \in$ Neighbour($s$) do
9:     if ConstraintCheck($x$) AND $i \neq x$ AND $b(x) = s$ then
10:        DangleNode($x$);
```

Function 12 StatusChange(s, $NewStatus$)

```
1: $OldStatus = status(s)$;
2: $status(s) = NewStatus$;
3: if $NewStatus = T$ then {new status is a 'Threat'}
4:     $h(s) = k(s) = f(s) = f_B(s) = \infty$;
5:     if $t(s) = OPEN$ then
6:         Dequeque($OpenQ, s$);
7:     $t(s) = NEW$;
8:     for $x \in$ Neighbour($s$) do
9:         if $t(x) \neq$ NEW then
10:            RevertPath($x, s$);
11:    $b(s) = NULL$;
12: else if $OldStatus = U$ then {changed from 'Unoccupied'$\Rightarrow$'Obstacle'}
13:     if $t(s) \neq$ NEW then
14:         RevertPath($s$);
15: else if $OldStatus = O$ then {changed from 'Obstacle'$\Rightarrow$'Unoccupied'}
16:     for all $x \in$ Neighbour($s$) do
17:         if $t(x) \neq NEW$ AND $status(x) = U$ then
18:             InsertOnOpenQ($x, h(x)$);
19: else if $OldStatus = T$ then {was status 'Threat' before}
20:     for all $x \in$ Neighbour($s$) do
21:         if $t(x) = U$ then
22:             for all $y \in$ Neighbour($x$) do
23:                 if $x \neq s$ AND $t(y) \neq NEW$ AND $status(y) = U$ then
24:                     InsertOnOpenQ($y, h(y)$);
```

Function 13 DangleNode(s)

```
1: $temp = b(s)$;
2: if $t(s) = OPEN$ then
3:     Dequeue($OpenQ, s$);
4: $t(s) = NEW$;
5: $h(s) = f(s) = f_B(s) = k(s) = \infty$;
6: $b(s) = NULL$;
7: for all $x \in$ Neighbour($s$) do
8:     if $x \neq temp$ AND ConstraintCheck($x$) then
9:         if $b(x) = s$ then
10:            DangleNode($x$);
11:        else if $t(x) = CLOSED$ then
12:            $PutOn = PutOn + \{x\}$;
               {Nodes in PutOn will later be considered for insertion on OpenQ}
```

2 (the node with a flag at the top of the graph between nodes 4 and 9) and the destination node is node number 16 (the node with a flag at the bottom middle of the graph between nodes 13 and 14). A direct distance (between nodes in the graph) heuristic function is used to calculate the $GVal$ function. Some of the distances between nodes are displayed in Table 1. For simplicity, all the edges have a cost of 1.

Phase 1: Calculate an initial optimal path: At the end of this phase we show node expansion for the previous version of this algorithm [2] which did not include a heuristic function. This illustrates how the addition of a heuristic function decreases node expansion.

The nodes numbered 1 (in the middle and to the left side of the graph) and 18 (right, close to the bottom right side of the graph) are obstacles (shaded darker), and all the other nodes are unoccupied. We show how the algorithm, function *FConstraint_DStar*, calculates the initial optimal path. The destination node is inserted into *OpenQ* by the *Initialise* function with $k(16) = h(16) = r(16) = 0$, $f(16) = fB(16) = 1.04$ and $R_{curr} = R = 2$. In the first execution of the *while* loop in *ComputeOptimalPath*, $s = 16$, and its neighbours, nodes 13 and 14, are placed into *OpenQ* with the vectors $\langle 1,99;1,99;1\rangle$ and $\langle 2,69;2,69;1\rangle$ respectively. The following steps are performed during each execution of the *while* loop : $s = 13$ and $OpenQ$: $5\langle 2,54;2,54;2\rangle$; $9\langle 2,59;2,59;2\rangle$; 14 $\langle 2,69;2,69;1\rangle$. $s = 5$ and *OpenQ*: $9\langle 2,59;2,59;2\rangle$; 14 $\langle 2,69; 2,69;1\rangle$; $3\langle 3,59;3,59;3\rangle$. $s = 9$ and *OpenQ*: $14\langle 2,69; 2,69;1\rangle$; $2\langle 3;3;3\rangle$; $12\langle 3,27;3,27;3\rangle$; $3\langle 3,59;3,59;3\rangle$. $s = 14$ and *OpenQ*: $2\langle 3;3;3\rangle$; $7\langle 3,12;3,12;2\rangle$; $6\langle 3,27;3,27; 2\rangle$; $12\langle 3,27;3,27;3\rangle$; $3\langle 3,59;3,59;3\rangle$. $s = 2$ and *OpenQ*: $7\langle 3,12;3,12;2\rangle$; $6\langle 3,27;3,27;2\rangle$; $12\langle 3,27;3,27;3\rangle$; $3\langle 3,59; 3,59;3\rangle$. The optimal path, 2-9-13-16, has a total path cost of 3.

Note: Without the focussing heuristic, i.e. with the authors' previous algorithm, the path expansion to obtain the same optimal path would have been much wider:
$s = 13$ and $OpenQ$: 14, 9, 5. $s = 14$ and *OpenQ*: 9, 6, 7, 5. $s = 9$ and *OpenQ*: 6, 7, 5, 12, 2. $s = 6$ and *OpenQ*: 7, 5, 12, 11, 2, 17. $s = 7$ and *OpenQ*: 5, 12, 17, 11, 2. $s = 5$ and *OpenQ*: 12, 17, 3, 11, 2. $s = 12$ and *OpenQ*: 17, 3, 11, 2, 15. $s = 17$ and *OpenQ*: 3, 11, 2, 15. $s = 3$ and *OpenQ*: 11, 2, 15, 0, 19. $s = 11$ and *OpenQ*: 2, 15, 0, 19, 8. $s = 2$ and *OpenQ*: 15, 0, 19, 8, 4.

Phase 2: The Status of a Node Changes from Unoccupied to Obstacle: The execution of *FConstraint_DStar* continues where we left off in Phase 1 above. Execute the *while* loop (lines 5-29) once such that $R = 9$. This means the unit started traversing the optimal path and is now at node 9. Suppose node 13's status has changed from Unoccupied to Obstacle, that is, $status\_sensor(13) = O$. When the *while* loop is executed again, the *if* statement in line 15 is satisfied as well as the *if* statements in lines 17-18. $d_{curr} = 0.6$ (assuming $\epsilon = 0.01$) and $R_{curr} = 9$. The function *StatusChange(13,O)* is called and it, in turn, calls *RevertPath(13)* in line 14. Node 16 is inserted into *OpenQ* (line 6 of *RevertPath)* with $f(16) = 1.30$, $f_B(16) = 1.30 + 0.6 = 1.9$ and $r(16) = 9$. The neighbours of node 13, nodes 5, 9, 16 and 18, are evaluated in lines 8-10 of *RevertPath*. Nothing is done for nodes 16 and 18 (an obstacle) and *DangleNode* is called for nodes 5 and 9 (line 11). The best paths through nodes 5 and 9 must now be invalidated. The call *DangleNode(5)* results in node 3 being removed from *OpenQ* and the call *DangleNode(9)* results in the calls *DangleNode(2)* and *DangleNode(12)* where node 12 is removed from *OpenQ*. Now *OpenQ* contains: $16\langle 1.9;1.3;0\rangle$; $7\langle 3,12;3,12;2\rangle$; $6\langle 3,27;3,27;2\rangle$. *ComputeOptimalPath* is called again with $s = 16$. The only viable neighbour of 16 is 14. A new optimal path is then expanded through from the current position of the military unit, 9 to 2-0-8-11-6-14-16.The miltary unit can then reach the destination through this path if no new changes are detected.

Phase 3: The Status of Two Nodes Change: In Phase 1 an optimal path was computed for the initial graph depicted in Figure 2. Nodes 1 and 18 are obstacles and all other nodes are unoccupied. The start node is 2 and the destination node is 16. The optimal path is nodes 2-9-13-16, and *OpenQ* contains: $7\langle 3,12;3,12;2\rangle$; $6\langle 3,27;3,27;2\rangle$; $12\langle 3,27;3,27;3\rangle$; $3\langle 3,59;3,59;3\rangle$. This phase continues from this point in the execution of the main algorithm. Before the *while* loop in lines 5-29 of *FConstraint_DStar* is executed, the following nodes change their status: node 12 becomes a threat and node 18 becomes unoccupied. ($R_{curr} = R = 2$.) The *if* statement in line 15 is true but the *if* statement in line 17 is false. The *for* loop in lines 21-26 is executed and the call *StatusChange(12,T)* is made. Node 12 is initialised to a NEW status and removed from *OpenQ*, and the *if* statement in lines 3-11 (of *StatusChange*) is executed. *RevertPath(9,12)* is called. Node 9 is initialised to NEW, node 13 is re-inserted into *OpenQ* and $b(9) = NULL$. The call *DangleNode(2)* results in the initialisation of node 2 to *NEW* and $b(2) = NULL$. To complete the call to *StatusChange(12,T), b(12)* becomes $NULL$. Now *OpenQ* contains: $13\langle 1,99;1.99;1\rangle$; $7\langle 3,12;3,12;2\rangle$; $6\langle 3,27;3,27;2\rangle$;$3\langle 3,59;3,59;3\rangle$. Back in *FConstraint_DStar*, the second iteration of the *for* loop in lines 21-26 has to be executed: A call to *StatusChange(18,U)* is made and node 13 is re-inserted on *OpenQ*. This re-insertion has to be made to check whether the unit has moved since the last insertion of node 13 on *OpenQ*. *ComputeOptimalPath* is called in line 28 of the main function. Node 13 is taken from *OpenQ* and a path to node 18 is expanded. The following nodes are expanded in turn: 10,19, 11, 17, 0, 8 and 2. The optimal path is 2-0-3-5-13-16.

Stenz has shown that his Focussed D* algorithm [4] cuts down the number of expanded considerably compared to his original algorithm [3]. The Focussed Constraint algorithm

TABLE I. Direct Distance between Some Nodes in Example Graph

Node	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0										.85										
1			.90																	
2	.3	.9		.59	.41	.54	1.27	1.12	.84	.59	.91	1.15	.27	.99	1.69		1.04	0.88	.85	.71
3			.59																	
4			.41																	
5			.54																	
6			1.27																	
7			1.12																	
8			.84																	
9	0.85		.59		1		1.85	1.53	1.32			1.75					1.3	1.45		
10			.91																	
11			1.15							1.75										
12			.27																	
13			.99																	
14			1.69																	
15																				
16			1.04							1.3										
17			.88							1.45										
18			.85																	
19			.71																	

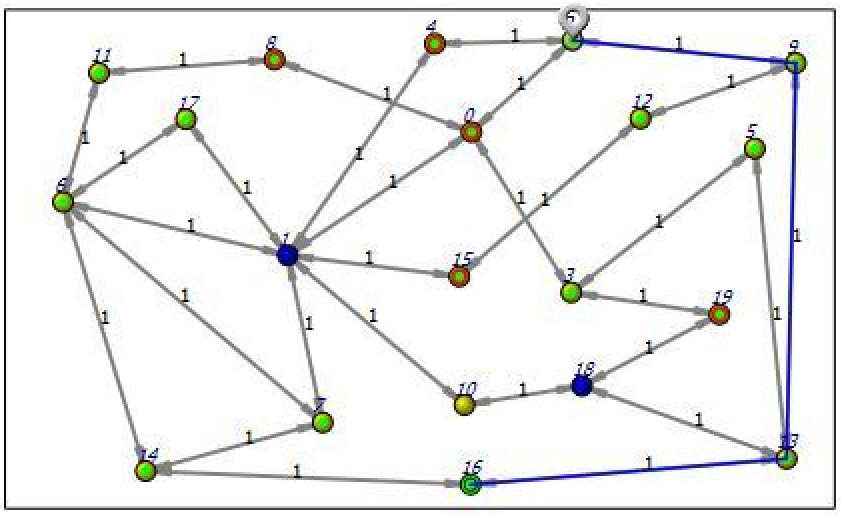


Fig. 2. Intial Example Graph

presented in this paper is based on the Focussed D* algorithm of Stenz and will thus improve our original algorithm (based on Stenz's original algorithm) in a similar way. This aim of this paper was to modify and extend our original contraint algorithm by focussing the search similar to that done by Stenz.

V. Conclusions

The authors introduce a constraint-based algorithm with a focussing heuristic to solve a CSP formulation of the Dynamic Military Unit Path Finding Problem. This algorithm improves the authors' previous constraint-based D* algorithm [2] and is based on Stenz's Focussed D* algorithm for dynamic path finding problems [4]. The advantage of a CSP approach to path finding problems is the flexibility it provides in modelling problems with particular characteristics such as the military unit path finding where path costs have to be minimised whilst safety aspects have to be taken into account. Experimentation with other heuristics to direct the search should be done. Future work should also consider generalisations of the current constraints for different applications of the algorithm.

References

[1] A. M. Mora, J. J. Merelo, J. L. J. Laredo, P. A. Castillo, C. Millan, and J. Torrecillas, "Balancing safety and speed in the military path finding problem: Analysis of different aco algorithms," in *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO'07)*, London, UK, July, 2007.

[2] L. Leenen, A. Terlunen, and W. le Roux, *A Constraint Programming Solution for the Military Unit Path Finding Problem*, ser. Mobile Intelligent Autonomous Systems: Recent Advances. Bota Raton, USA: Taylor & Francis Group, 2012, pp. 225–240.

[3] A. Stentz, "Optimal and efficient path planning for partially-known environments," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, vol. 4, 1994, pp. 3310–3317.

[4] A. Stenz, "The focussed D* algorithm for real-time replanning," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI'95)*, vol. 2, 1995, pp. 1652–1659.

[5] L.Leenen, J. Vorster, and W. le Roux, "A constraint-based solver for the military unit path finding problem," in *The 2010 Spring Simulation Multiconference*, Florida, USA, 2010.

[6] R. Dechter, *Constraint processing*, 1st ed. San Fancisco: Morgan Kaufman Publishers, 2003.

[7] F. Rossi, P. van Beek, and T. Walsh, *Handbook of Constraint programming (ed)*, 1st ed. Elsevier, 2006.