Methodology for Mobility Tactical Decision Aids Incorporated into the Joint Mapping Tool Kit

George B. McKinley, Benjamin T. Webb, David A. Horner

September 2002
The contents of this report are not to be used for advertising, publication, or promotional purposes. Citation of trade names does not constitute an official endorsement or approval of the use of such commercial products.

The findings of this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.
Methodology for Mobility Tactical Decision Aids Incorporated into the Joint Mapping Tool Kit

by George B. McKinley, Benjamin T. Webb, David A. Horner

Geotechnical and Structures Laboratory
U.S. Army Engineer Research and Development Center
3909 Halls Ferry Road
Vicksburg, MS  39180-6199

Final report
Approved for public release; distribution is unlimited

Prepared for U.S. Army Corps of Engineers
Washington, DC  20314-1000
Preface

The study reported herein was conducted by members of the staff of the U.S. Army Engineer Research and Development Center (ERDC), Geotechnical and Structures Laboratory (GSL), Engineering Systems and Materials Division (ESMD), Mobility Systems Branch (MSB), Vicksburg, MS. Sponsor for the project was Headquarters, U.S. Army Corps of Engineers. The work was conducted between October 2000 and September 2001.

The study was conducted under the general supervision of Dr. David W. Pittman, Acting Director, GSL; Dr. Albert J. Bush III, Chief, ESMD; and Dr. David A. Horner, Chief, MSB. The overall development--the logic and computer programming--was accomplished by Messrs. George B. McKinley, Benjamin T. Webb, and David A. Horner, MSB.

Messrs. McKinley, Webb, and Horner prepared the report.

At the time of publication of this report, Dr. James R. Houston was Director of ERDC, and COL John W. Morris III, EN, was Commander and Executive Director.

The contents of this report are not to be used for advertising, publication, or promotional purposes. Citation of trade names does not constitute an official endorsement or approval of the use of such commercial products.
1 Introduction

Background

The Joint Mapping Tool Kit (JMTK) represents the mapping, charting, geodesy, and imagery functionality for the Global Command and Control System (GCCS) under the Defense Information Infrastructure Common Operating Environment (DIICoE). The JMTK is one of the common support applications of the DIICoE. One goal for JMTK is to eventually replace the Army’s Terrain Evaluation Module (TEM), the Air Force’s Common Mapping Toolkit, the Navy’s Chart, and the National Security Agency’s Oilstock.

The Mobility Systems Branch (MSB), Geotechnical and Structures Laboratory (GSL), U.S. Army Engineer Research and Development Center (ERDC), Vicksburg, MS, has a long history of providing Tactical Decision Aids (TDA) for military planning systems. These TDA algorithms are based on the NATO Reference Mobility Model edition II (NRMMII). The NRMMII is an Army Model and Simulation Office (AMSO) standard for ground vehicle movement. Several of these TDAs have been incorporated into the TEM and are now being transferred to the JMTK.

Purpose and Scope

The purpose of this report is to present the methodology and show example results from the mobility TDAs that are to be incorporated into the JMTK. These TDAs are the optimum route selector, the Combat Maneuver Model (CMM), the mobility network generator, and the isochrone generator. The optimum route selector chooses a route of a given width between designated map positions and locates choke points along that route. The CMM computes the time for a unit of vehicles in one of four basic combat formations to traverse the selected route. The mobility network generator produces an edge/node network of corridors and categorizes the resultant edges based on the largest military unit that the edge can handle. The isochrone generator predicts the time required for a single vehicle to reach all areas within a speed matrix from an initial location. Application programmer interfaces (API) for all the mobility TDAs are presented in Appendix A.
Definitions

The following are definitions of terms used in the report:

a. Artificial Intelligence (AI). The part of computer science concerned with designing intelligent computer systems; that is, systems that exhibit the characteristics associated with intelligence in human behavior (Barr and Feigenbaum 1981).

b. Binary tree. A tree structure in which each node has one parent and two descendants.

c. Forward reasoning. The application of operators to bring the situation forward from its initial state to one satisfying a goal condition.

d. Heap. A data structure which represents a complete binary tree sequentially in an array and may be used to implement a priority queue.

e. Heuristic. Knowledge of the problem domain which helps improve problem-solving performance.

f. Isochrone. A line on a chart connecting points at which an event occurs simultaneously or which represents the same time or time difference.

g. Node. The states, as they are represented in the search tree, created by operators as the search proceeds.

h. Operators. A set of rules that transforms the problem from one state to another.

i. Ordered search. A search routine that always selects the most promising node as the next node to expand.

j. Priority queue. A data structure, which always contains the largest key at the top, but the rest of the keys are not necessarily in the order in which they are stored.

k. Search tree. The tree that is constructed and grows as the search proceeds.

l. State-space representation. A problem-solving system that uses forward reasoning and whose operators each work by producing a single new object in the database.

m. States. Data structures that show the condition of the problem at each stage of its solution.

n. Unified Soil Classification System (USCS). A system which classifies soils according to their texture and plastic qualities and on their grouping with respect to engineering behavior.
Overview

The NRMMII (Ahlvin and Haley 1992) is a computer code used to predict the steady-state operating capability of a given vehicle in a prescribed terrain. The NRMMII determines the maximum possible speed versus resisting force relation of the driving elements for a vehicle considering its power-train capability. The model then predicts various impediments to vehicle motion as a function of specific terrain factors.

The input data for the model are segregated into vehicle, terrain, and “scenario” data. The vehicle data describe various physical aspects in engineering terms, such as the power train, surface traction elements, sizes, weights, geometry, etc. of the vehicle. The terrain data describe in engineering terms various aspects of the terrain pertaining to the vehicle’s operational surface such as soil properties, slope, vegetation, macro- and micro-geometry, etc. “Scenario” data consist of generic vehicle and terrain data that are independent of a specific terrain or vehicle and remain constant for a model execution. Examples of “scenario” data include the driver's reaction time, weather conditions, climatic conditions, vegetation avoidance/override strategies, etc.

Implementation

The model implementation consists of four parts: input routines, data preprocessors, primary functional submodels, and output routines. The input routines read the specified vehicle, terrain, and scenario data into the model. The preprocessors compute factors that are independent of the vehicle/terrain combinations, which the various predicting algorithms require. The primary functional submodels consist of various predicting algorithms and the logic to combine the individual results into a final speed prediction. The output routine produces the various statistics and reports.

There are two primary functional modules, which are invoked as required by the input terrain description. They are a cross-country terrain performance prediction module and an on-road speed prediction module. The two modules share many of the submodels.
Cross-country Prediction Module

Figure 1 shows a flowchart of the cross-country prediction module. The cross-country module operation is as follows: The vehicle’s maximum “theoretical” motion capability is determined in the power-train submodel. The result is the maximum force versus speed relation at the wheel or track. This relation includes the internal vehicle parasitic and accessory power losses but excludes all external resistance losses.

The surface traction and resistance submodel uses the soil and surface properties to determine the maximum actual traction available, the motion resistance as a result of the surface, the relation between the actual vehicle speed, and the theoretical speed resulting from wheel or track slippage. The maximum traction is used to determine if vehicle motion can occur (the tractive force must exceed the total of all resistances). The resistance force submodel is used to determine the maximum possible theoretical vehicle speed from the tractive force versus speed relation, and the slip relation is used to determine the actual maximum vehicle speed. The soil/vehicle interface submodel, comprising the bulk of the predicting algorithms, consists of empirical relations that are derived.
from the results of field-testing of vehicles over the past 40 years. The Wheels versus Tracks test program provided new test results that allowed further refinement of the existing relations and provided information allowing for the delineation of the USCS soil types in the model. The need to incorporate this information into the NATO Reference Mobility Model was a primary factor in prompting the revision effort that yielded NRMMII.

The longitudinal slope submodel determines the resistance that results from operating on a slope (positive or negative) in the direction of the vehicle travel. The lateral slope (side slope) submodel may optionally be included to consider the vehicle’s stability when operating on side slopes.

The braking submodel determines the maximum permissible speed at which the vehicle can make a controlled stop within the available visibility. The situation in which a vehicle cannot stop while traveling down-slope is considered a “NOGO” condition.

The dynamic reaction of a vehicle to the micro-geometry of the terrain is determined in the NRMMII vehicle dynamics module. These results reflect the limiting speed of the vehicle while not exceeding a driver-preferred vibration level expressed in watts of absorbed energy called “the absorbed power criteria.” This information is passed to the main prediction model by way of a table of terrain surface roughness values corresponding to maximum vehicle speed values for specific absorbed power levels. These results are used to determine the limiting speed for a given terrain surface roughness condition.

The obstacle override submodel determines whether the vehicle can negotiate the obstacles in the terrain and, if so, provides the override resistance. The NRMMII obstacle-crossing module, OBS78B, determines the obstacle interface performance for a generic set of obstacles. The OBS78B module output consists of the minimum obstacle clearance, maximum traction required, and resistance to cross a set of obstacles with given heights, widths, and angles. This information is transferred to the cross-country module by way of tables of values, which are used to find the clearance, traction, and resistance for the specific obstacle.

The maximum obstacle-crossing speed as a function of obstacle height for a given maximum impact acceleration level is determined in VEHDYN II. These results are transferred to the cross-country prediction module as a table of values.

The obstacle override submodel determines whether the vehicle can negotiate the obstacles. If so, the obstacle override resistance is determined. An acceleration and braking submodel is included to provide the average speed resulting from accelerating toward the maximum speed between obstacles and then braking to the maximum obstacle-crossing speed based on the acceptable impact speed. The vegetation override submodel determines the override resistance for several vegetation and obstacle override strategies.

The maneuver submodel determines the area denied when maneuvering around obstacles and vegetation for several vegetation and obstacle avoidance strategies. Area denied is the portion of the terrain (expressed as a percent of the
total area) that a vehicle cannot traverse while avoiding particular terrain features. The area denied, calculated from the various vegetation and obstacle avoidance/override strategies, is used in the maneuver speed submodel to provide speeds for the various combinations.

The final prediction of “speed made good” is made as follows: The minimum of all speeds predicted from resistances and strategies is chosen. Checks are made to determine that there is sufficient traction available for movement. Resistances are summed and the result is used to obtain the force-controlled speed from the power-train relation (with suitable adjustments for wheel slip, etc.). The various obstacle and vegetation override/avoidance strategies are examined to determine the combination, which yields the fastest speed. The reduced speeds resulting from maneuvering are applied to the area of vegetation and obstacles being avoided; acceleration and braking average speeds between obstacles are computed when obstacles are overridden.

Figure 2 shows the overall scheme used for predictions. The tractive force versus speed relation is corrected for the soil. The individual resistances such as slope, soil, and override resistances (R1, R2, R3) are summed to produce the total resisting force. The least of the speed-limiting factors such as ride and visibility limits (S1, S2, S3) is selected as the maximum limited speed which defines an envelope of possible vehicle operation (shaded area). For most applications, the maximum speed within the envelope of operation is the predicted speed.

Figure 2. Example of speed-predicting scheme
On-road Prediction Module

The on-road prediction module operates in a manner similar to that of the cross-country module and shares many submodels with that module. As there are no obstacles or vegetation to consider for on-road predictions, the submodels involving obstacle override, obstacle avoidance, vegetation override, and vegetation avoidance are bypassed. An additional surface traction submodel is included to predict for operation on hard surfaces. A curvature submodel is included to predict for the effects of on-road horizontal curves.

The on-road curvature submodel uses the Association of State Highway and Transportation Officials-recommended curvature speed relations to determine the maximum road curvature speed. Factors affecting vehicle stability, i.e., sliding and slipping on curves, are also considered. Resistances derived from turning wheels or tracks are also computed.

The final on-road speed prediction is comprised of the minimum of the maximum resistance limited speed derived from the slip-corrected tractive force versus speed relation and the various limiting factors, namely surface roughness, curvature, and visibility. Figure 3 is a flowchart of the on-road prediction module.

Output

The model’s primary output is a mobility forecast based on speed predictions made for specific areal units of terrain and for specific linear portions of a road network which are both referred to as terrain units. Speed predictions are made for up-, down-, and level-slope conditions. The specific factor limiting the predicted speed is also generated for up-, down-, and level-slopes. Predictions may be made for more than one vehicle or for different configurations of a vehicle on a given terrain. Thus, the vehicle with the slowest predicted speed on a terrain unit is also available as output. Results produced by the model’s individual submodules, such as forces acting on the vehicle, power-train operating point, etc., are also available as output but are currently not implemented in the JMTK.

SWIMCRIT

The SWIMCRIT model is a derivative of the NRMMII configured to evaluate vehicle performance when crossing gaps at low, mean, and high hydrologic stages. SWIMCRIT predicts performance based on a stylized trapezoidal-shaped feature. If the base width is zero, the trapezoid is a “V” style obstacle. The types of potential crossing are:

a. *Spanned or bridged gaps.* The vehicle is not able to rest entirely on the gap bottom (i.e., part of the vehicle touches both gap banks).

b. *Forced gaps.* The vehicle can rest on the bottom entirely between banks, and water depth does not exceed the vehicle fording depth.
c. Swimmable gaps. The vehicle can rest on the bottom entirely between banks, and the water depth exceeds the vehicle fording depth, causing flotation.

Figure 4 shows a schematic definition of each crossing type. SWIMCRIT, as implemented in the JMTK, predicts the average speed, the total time, and a reason code for each crossing.
Figure 4. Schematic definition of SWIMCRIT crossing types
3 Optimum Route

Search Algorithm

The optimum route algorithm uses an AI algorithm referred to as \( A^* \). \( A^* \) is an ordered state-space search which uses a heuristic to decrease the number of nodes which must be expanded, while still finding an optimal solution (Nilsson 1971). The evaluation function in the \( A^* \) algorithm involves two parts. The first part computes the cost of traveling from the start cell to the current cell along an optimal route. A problem arises in computing this cost because the unit frontage may cover more than one cell. The program solves this problem by averaging the inverses of the speeds contained in the unit frontage and then inverting that average. This average is then scaled to a range of 1 (good) to 1,000 (bad). A minimum speed of 0.1kph is used for these computations. Once this scaled value has been computed for each terrain cell, the \( A^* \) search algorithm is used to determine the path of least cost.

The cost for each terrain cell, \( c_i \), is determined by placing the center of the combat unit at the center of the terrain cell, \( i \), and then summing the inverses of the speeds contained in all the terrain cells which the combat unit covers. For example, a company-sized unit may be represented as a circle with a diameter of 700 m on a terrain database comprised of square terrain cells, which are 100 by 100 m. Thus, a company-sized unit covers the 37 terrain cells whose centers fall within 700 m of the center cell. A circle is used to model the avenue frontage instead of a square, because the area encountered when moving in a diagonal direction between cells is more consistent with the area encountered when moving in a horizontal or vertical direction.

The optimum route routine requires an elevation matrix as well as matrices containing the up-, down- and across-slope speeds for each grid cell as produced by the NRMMII. A slope aspect matrix is computed from the elevation matrix using the mean slope method (Ritter 1987). Two different slopes are calculated: the vertical slope takes the average elevation of the top three pixels of a 3- by 3-pixel window, subtracts the average elevation of the bottom three pixels and then divides by the distance between the two rows; similarly, a horizontal slope is computed using the average elevation of the right three pixels minus the average elevation of the left three pixels. The percent slope of the center pixel of the 3 by 3 window is computed as follows:
\[ \text{slope} = 100 \times \sqrt{\text{vslope}^2 + \text{hslope}^2} \] 

where

\begin{align*}
\text{slope} &= \text{percent slope for the center cell of a 3 by 3 window} \\
\text{vslope} &= \text{vertical slope} \\
\text{hslope} &= \text{horizontal slope}
\end{align*}

The slope aspect is then computed by taking the arc tangent of the vertical slope over the horizontal slope. This value is then converted from radians to degrees. The slope aspect is subtracted from 270 when the horizontal slope is greater than 0. Otherwise, the slope aspect is subtracted from 90. This slope aspect is then classed into the eight possible directions of movement when traveling between cells on a map matrix. These classes are shown in Figure 5. When movement across a grid cell is in the same general direction as the slope aspect, travel is modeled as down-slope. For example: if the slope aspect is in direction 6, then directions 5, 6, and 7 would be down-slope; directions 1, 2, and 3 would be up-slope; and directions 4 and 8 would be across-slope.

![Figure 5. Slope aspect classes](image)

The sum of the costs for all cells encountered between the start cell and the current cell along the best path currently known is referred to as \( g^* \). The second
portion of the evaluation function is an estimate of the cost to reach the goal cell from the current cell and uses the following heuristic:

\[ h^* = \text{dist} \times \min C \]  

(2)

where

- \( h^* \) = estimated cost of reaching the goal cell from the current cell
- \( \text{dist} \) = number of cells that would be crossed by traveling in a straight line between the current cell and the goal cell
- \( \min C \) = minimum cost computed for any cell on the map

This heuristic satisfies the admissibility condition, which requires that \( h^*(n) \) is less than or equal to \( h(n) \), which is the actual cost of an optimum path from cell \( n \) to the goal cell. If \( h^* \) satisfies this admissibility condition and if all costs are positive and can be bounded from below by a positive number, then \( A^* \) is guaranteed to find a solution path of minimal cost if any solution path exists. The entire evaluation function \( f^* \) for a cell in the search is given by \( f^*(n) = g^*(n) + h^*(n) \).

The \( A^* \) search algorithm was implemented as follows:

Put the starting grid cell on a list called OPEN, of untraveled grid cells
Compute an \( f^* \) value for this grid cell
Set the flag SOLVED to false
While NOT SOLVED is true do
  If OPEN is empty then
    Exit with Failure {No solution exists}
  End If
  Select from OPEN a grid cell \( (GXY) \) with the lowest associated \( f^* \) value
  Remove GXY from OPEN
  Place GXY on a list called CLOSED of traveled grid cells
  If GXY is the goal cell then
    Set SOLVED to TRUE
  Else
    Expand grid cell GXY creating the eight adjacent successor grid cells
    For each successor cell SXY of GXY do
      Compute an \( f^* \) value for SXY
      If SXY is not on OPEN or CLOSED then
        Add SXY and its associated \( f^* \) value to OPEN
        Attach a pointer from SXY back to GXY {In order to trace back a solution path once the goal is reached}
      Else
        If SXY’s new \( f^* \) value is less than the previous value then
          Substitute the new \( f^* \) value for the old one
          Point SXY back to GXY instead of its previous predecessor
        If SXY was found on the CLOSED list then
          Move SXY back to OPEN
      End If
    End If
  End If
End If
End Do {For}
End If
End Do {While}
Create list of traveled cells by following pointers from goal cell to starting cell
Reverse list of traveled cells

The OPEN list is implemented as a heap (Sedgewick 1983) with \(-f^*\) used as the key, thus causing the cell with the lowest \(f^*\) value to be positioned at the top of the heap. A matrix is also used to allow quick decisions on whether or not a cell is on the OPEN list. The CLOSED list is implemented as only a matrix which contains the \(f^*\) value required to reach each cell from the start location and also a pointer back to each cell’s predecessor. This implementation allows for relatively quick searches.

**Choke Points**

Choke points are modeled in the optimum route selection by adjusting the average speed to represent the effect of narrowing the formation’s frontage. This choke-adjusted speed is based on applying a penalty to the average speed of a piece of terrain based on the percent choke encountered. The choke computation is performed in incremental steps, which begin at a center cell and work outward until all the cells required to approximate the full width of the unit are used in the computations. This progression for a 700-m-wide unit frontage being selected on a matrix comprised of 100- by 100-m cells is shown in Figure 6. At each step, the average speed is multiplied by the speed correction factor corresponding to that step. This speed correction factor is computed as follows:

\[
spdcor = \left(\frac{step}{tstep}\right)^2
\]

where

- \(spdcor\) = computed speed correction factor
- \(step\) = the current choke modeling step
- \(tstep\) = the total number of steps required for the modeled formation’s frontage

Although it may appear that the ratio of the current width for a step (i.e., 1/7, 3/7, 5/7, and 1 for the 700-m example) is more mathematically correct than the ratio of the current step to the total steps (i.e., 1/4, 2/4, 3/4, and 1 for the 700-m example), comparisons of results using both methods showed the ratio of steps to provide more acceptable results. The ratio of widths tends to severely limit the amount of choke that is accepted by the search. The maximum speed selected from the computed speed at each step (using the choke modified average speed) is retained as the speed for that cell. The choke step at which the maximum speed occurred is used as the basis for the largest unit that could proceed with the mass of that unit centered on that particular cell.
Figure 6. Steps used in choke point computations

The first example (Figure 7) shows an optimum route with the centerline and border displayed. The choke avoidance flag was activated, thus the route went around the entire NOGO area (red) rather than through the gap which was approximately one-half the desired width of the route. The second example (Figure 8) shows an optimum route selected on the same data set with the choke avoidance flag not activated. The $A^*$ search algorithm selected a route through the small gap in the NOGO area. The border of the optimum route displays the effects of this choke point. Figure 9 shows an optimum route selected on a database that also includes roads (depicted in blue) and drainage features (shown in gray). The input road and drainage vectors are placed in a matrix of grid cells using Bresenham’s algorithm (Newman and Sproull 1979). This algorithm performs the gridding operation with integer arithmetic and requires neither division nor multiplication. Bresenham’s algorithm for the case of $0 \leq \text{deltay} \leq \text{deltax}$ is as follows:

$$e := 2 \times \text{deltay} - \text{deltax};$$

for $i := 1$ to $\text{deltax}$ do begin

Set($x,y$); {Make the value in Grid cell $x,y$ the road speed or crossing time}
if $e>0$ then begin

$y := y + 1;$
$e := e + (2 \times \text{deltay} - 2 \times \text{deltax});$
end;
else $e := e + 2 \times \text{deltay};$
$x := x + 1;$
end;
Figure 7. Optimum route selected with choke avoidance activated

Figure 8. Optimum route selected with choke avoidance not activated
In the above algorithm, $e$ is an error term. The sign of $e$ is used to determine whether to increment the $y$-coordinate of the current point. A positive $e$ value indicates that the exact path of the line lies above the current point; therefore, the $y$-coordinate is incremented. If $e$ is negative, the $y$-coordinate value is left unchanged. The variables $\text{deltax}$ and $\text{deltay}$ are the absolute values of the differences in the $x$ and $y$ directions between the two points. The optimum route in Figure 9 follows the roads, which in this example were given a far greater speed than the cross-country terrain.

**Combat Maneuver Model**

The ERDC CMM (McKinley et al. 1993) predicts the time required for a group of vehicles to traverse a series of terrain units as produced by the optimum route routine. The vehicles must travel in one of four basic formations: column, bounding overwatch, combat lines, and parallel columns. The minimum and maximum following distances for vehicles within a column formation, in addition to a maximum allowed speed, are input to the routine, thus allowing the modeling of both open and closed column formations (Figure 10). In the bounding overwatch (Figure 11), the first vehicle travels out to a set distance and then waits for the second vehicle to reach it before proceeding. Once the second vehicle reaches the minimum following distance from the first vehicle, the first vehicle performs another bound. All other vehicles in the bounding overwatch are modeled as if they are in a column with minimum and maximum following distances. The combat line (Figure 12) consists of single vehicles traveling along parallel paths while staying within a set distance in relationship to each other. The parallel columns (Figure 13) are modeled as a combat line (for the lead
Following Distance:
20-30 m for Closed Column
75-125 m for Closed Column

Figure 10. Column formation

vehicle in each column) and as a column (for all but the lead vehicle in each column).

Acceleration and deceleration (AC/DC) of the individual vehicle are not modeled in this version of the CMM. This allows the CMM to work with much less input data from the speed prediction model NRMMII. The CMM selects the proper speed from among the up-, down- and across-slope speeds by using the same slope aspect methodology that is used in the optimum routes selection routine. The CMM models the speed across a choke point by averaging the speeds across the route’s width, then applying the choke point speed correction factor as described in the optimum route methodology to modify that average speed. Then this average speed is used as the speed across the entire formation’s width at the choke point. The CMM overlays the vector paths onto the speed matrices and produces a series of speeds for each separate path. For three parallel columns, there would be 3 paths, for a combat line of 20 vehicles there would be 20 paths, and for bounding overwatch and columns there would be 1 path per route. Although this version of the CMM does not model AC/DC, it maintains the modeling of the accordion effect (i.e., the bunching up of the vehicles in slow terrain and their spreading out in fast terrain). This is
accomplished by monitoring each vehicle’s progress at a 5-sec time interval. Each time interval is evaluated twice. First, each vehicle will traverse the terrain, traveling at the predicted speed, until the time interval is over. At the completion of each time interval, the position of each vehicle is checked to determine that the formation’s unity is maintained. If distances between vehicles are too large or too small, certain vehicles are required to proceed at a slower pace over the time frame to maintain proper vehicle spacing within the formation. Time intervals when the first and last vehicle reach waypoints are the main output from the routine.
Figure 12. Combat line formation
Figure 13. Parallel column formation
4 Mobility Network Generator

Thinning Algorithms

The mobility corridor (MC) network algorithm uses a modification of the classical thinning algorithm to produce a skeleton of the areas with a speed above a desired threshold. This thinning algorithm reduces the areas of the map matrix that have values greater than a designated threshold. The pixels comprising those areas with values greater than the threshold are set to 1 and all other pixels are set to 0. The classical thinning algorithm (Pavlidis 1982) was implemented as follows:

Set a pass counter N to 0
Set the flag REMAIN to \textit{true}
While REMAIN is \textit{true} do
    Increment N
    Set REMAIN to \textit{false} \{No change has been made\}
    For J=0, 2, 4, and 6 do
        For all pixels P in the map matrix do
            If P is 1 and if P’s J-neighbor (Figure 14) is 0 then
                Set flag SKEL to \textit{false}
                For all six patterns PA shown in Figure 15 do
                    If the neighborhood of P matches any of the patterns PA, set SKEL to \textit{true}
                    \{For a group of pixels to match a pattern, at least one of each group of pixels marked with A or B must be nonzero\}
                End Do \{For\}
            End If
        End Do \{For\}
    End If
End Do \{For\}
For all pixels P in the map matrix do
    If P is 3 then
        Set P to 0
    End If
Figure 14. Enumeration of neighbor pixels in the thinning algorithms

Figure 15. Patterns used in the classical thinning algorithm
Figure 16 shows example data resulting from the application of this algorithm. It is apparent from this test case that the classical thinning algorithm yields far too many unwanted disconnects when applied to this type of problem.

In an effort to develop an algorithm that would produce a network with few or no disconnects, the classical thinning algorithm was modified (McKinley, Falls, and Stuart 2000). The first modification consisted of allowing the last four patterns in Figure 15 to be matched by either a value of one or two when previously the match had to be made by a two. The second modification was to retain all pixels that had only one neighbor pixel set to one. Figure 17 shows the improved output resulting from the application of the modified algorithm.

**Network Creation**

The thinned areas are converted to an edge/node network format. The first step in this process is to find edges and nodes by summing the number of neighboring pixels that are nonzero for each pixel that is nonzero. When summing these nonzero neighboring pixels, if a diagonal neighbor pixel is nonzero and either the vertical or horizontal neighbor pixel adjacent to that diagonal neighbor is nonzero, then the diagonal neighbor pixel is not included in the sum. For example, if neighbor pixel 3 in Figure 14 is nonzero, and either neighbor 2 or neighbor 4 is nonzero, then neighbor pixel 3 will not be used in the sum. Another special case is depicted in Figure 18. In this case, the thinning process yielded four adjacent pixels. The solution was to not include a connection between the upper two adjacent pixels in their sums. Nodes are the pixels assigned a sum of one or a sum greater than two. Each node’s location is stored and the node is assigned a number. The node numbers are generated sequentially. Segments are formed connecting the adjacent cells using the same rules for their generation as was used in the pixel enumeration.
Figure 16. Result of classical thinning applied to test case
Figure 17. Result of modified classical thinning applied to test case
The next step toward network creation involves traversing (between nodes) the segments that form each edge. Each edge is added to the edge list of the two nodes that it connects. Once the pixels forming an edge are identified, the nodes that are located at each end of the edge are stored. Then the largest formation that the edge can support is computed based upon the 35th percentile width of those pixels comprising the edge. This percentile was based on the assumption that approximately one-third of the corridors length could be choked. The width associated with each pixel is based on the number of passes required to thin the area surrounding that pixel. It is assumed that the average number of thinning passes for the edge multiplied by the minimum of the X and Y cell resolutions is one-half the width of the MC represented by the edge. This assumption appears valid, since the lines are generally reduced from two sides on each thinning pass. The widths used to class the edges are shown in Table 1. Edges having average widths less than 500 m are currently assigned a company level by default. Figure 19 shows an example on a simple 40-km by 40-km data set, which yielded the expected eight edges. The four internal edges are about 10 km wide and are classified as division corridors. The four external edges are about 5 km wide and are classified as brigade corridors. Figure 20 shows another simple example, which yielded the expected six edges. The diagonal edge in this figure is greater than 12 km wide and is thus classified as a corp corridor.
### Table 1
**Widths Used to Class Edges**

<table>
<thead>
<tr>
<th>Unit</th>
<th>Width, m</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corp</td>
<td>12,000</td>
</tr>
<tr>
<td>Division</td>
<td>6,000</td>
</tr>
<tr>
<td>Brigade</td>
<td>3,000</td>
</tr>
<tr>
<td>Battalion</td>
<td>1,500</td>
</tr>
<tr>
<td>Company</td>
<td>500</td>
</tr>
</tbody>
</table>

**Figure 19.** Example mobility corridor network on simple data set

**Figure 20.** Example mobility corridor network on modified simple data set
In isochrone (time contours) computation, the cost of traveling from one grid cell to another is the time in seconds required to perform that move. This time is computed from the predicted speeds for both the exited and the entered cells and the distance traveled in each cell, which is based on the size of the grid cells. The cost function \( f^* \) for traveling from cell \( j \) to cell \( i \) would be the time required to travel along an optimal path from the start cell to cell \( j \) plus the time required to travel from cell \( j \) to cell \( i \). The basic ordered-search algorithm was implemented as in the optimum route selection, except that \( h^* \) would be 0. Also, the solution is found when the OPEN list becomes empty. The OPEN and CLOSED lists are implemented in the same manner that they are in the optimum route routine, except that there is no need to keep track of pointers to predecessors since no paths need to be retraced.

Figure 21 shows the isochrones generated when starting at the center of the speed map shown in Figure 19. This figure shows how the shape of the time band changes when the search works around the NOGO areas. NOGO areas are again modeled using a speed of 0.1 kph.

Figure 22 shows a further example of isochrones created when using the speed map shown in Figure 20. Of special note is the difference in the time bands in the upper-left corner where there is not a NOGO area.
Figure 22. Isochrones (time contours) on a modified simple data set

Figure 23 shows a further example of isochrone computations. In this map, a river (depicted in blue) with a long crossing time runs diagonally through the map and has a pronounced effect on the predicted times.

Figure 24 shows a further result of running the isochrone code. This example has both a river (depicted in blue) and roads (shown in gray). The faster on-road speeds allow the vehicle to reach far more of the map in much less time. Because of the slow speeds assigned to the cross-country data in this example, the major factor in determining the time to reach a location is the proximity of that point to a road.
Figure 24. Isochrones (time contours) computed for map with cross-country, roads, and drainage features
6 Conclusions and Recommendations

Conclusions

Based on the results of this investigation, the following conclusions can be drawn:

\( a. \) The \( A^* \) algorithm has been implemented to select mobility corridors and locate choke points along those corridors.

\( b. \) The CMM predicts the speed of a vehicle formation over selected mobility corridors.

\( c. \) The classical thinning algorithm has been modified to create continuous networks of mobility corridors that are classed by their average widths.

\( d. \) The \( A^* \) algorithm has successfully been implemented to create isochrones (time contours) using speeds produced by the NRMMII.

Recommendations

Based on the information presented in this study, it is recommended to:

\( a. \) Accept the models presented in this report as producers of Tactical Decision Aids for the JMTK.

\( b. \) Continue the refinement of these routines based on feedback from users of the JMTK.
References


Appendix A
Application Programmers’ Interface
1. API Reference ................................................................. A6

1.1. General .............................................................................. A6

1.1.1. Constants ......................................................................... A6
   1.1.1.1. MOBVECSHAPE_POINT ............................................................ A6
   1.1.1.2. MOBVECSHAPE_LINE ............................................................. A6
   1.1.1.3. MOBVECSHAPE_POLYGON .................................................... A6
   1.1.1.4. MOBCOVERAGE_SLOPE ........................................................ A6
   1.1.1.5. MOBCOVERAGE_VEGETATION ............................................. A6
   1.1.1.6. MOBCOVERAGE_SURFACE_MATERIAL ............................... A7
   1.1.1.7. MOBCOVERAGE_OBSTACLES .............................................. A7
   1.1.1.8. MOBCOVERAGE_TRANSPORTATION ................................. A7
   1.1.1.9. MOBCOVERAGE_DRAINAGE .............................................. A7

1.1.2. Data Structures ............................................................. A7
   1.1.2.1. MobLocation ............................................................................. A7
   1.1.2.2. MobVecShape .......................................................................... A7
   1.1.2.3. MobHistWeather ....................................................................... A8
   1.1.2.4. MobWeather ............................................................................. A8
   1.1.2.5. MobExternalType ...................................................................... A8
   1.1.2.6. MobAttrDMAFF ......................................................................... A9
   1.1.2.7. MobFeatDMAFF ........................................................................ A9
   1.1.2.8. MobExternalDMAFF ................................................................ A9
   1.1.2.9. MobAttrFACC .......................................................................... A9
   1.1.2.10. MobFeatFACC ....................................................................... A9
   1.1.2.11. MobExternalFACC .................................................................. A9

1.1.3. Functions ......................................................................... A10
   1.1.3.1. MobApiGetError ...................................................................... A10

1.2. Initialization ...................................................................... A11
   1.2.1. Functions ................................................................................. A11
      1.2.1.1. MobApiInitialize .................................................................. A11

1.3. Mobility Predictions .......................................................... A11
   1.3.1. Vehicles .................................................................................. A11
      1.3.1.1. Data Structures .................................................................... A11
1.3.2. Cross Country

1.3.2.1. Constants

1.3.2.1.1. MOBCC_STATIC_DATA_CONST

1.3.2.1.2. MOBCC_DYNAMIC_DATA_CONST

1.3.2.1.3. MOB_UP_SPEED

1.3.2.1.4. MOB_UP_REASON

1.3.2.1.5. MOB_DOWN_SPEED

1.3.2.1.6. MOB_DOWN_REASON

1.3.2.1.7. MOB_CROSS_SLOPE_SPEED

1.3.2.1.8. MOB_CROSS_SLOPE_REASON

1.3.2.1.9. MOB_AVERAGE_SPEED

1.3.2.1.10. MOB_AVERAGE_REASON

1.3.2.1.11. MOB_CONTROLLING_VEHICLE

1.3.2.1.12. MOB_END

1.3.2.1.13. MOB_VISIBILITY

1.3.2.1.14. MOB_SLIPPERY_SURFACE

1.3.2.2. Data Structures

1.3.2.2.1. MobExternalCC

1.3.2.2.2. MobCC

1.3.2.3. Functions

1.3.2.3.1. MobApiGetDataCC

1.3.2.3.2. MobApiCreateHistoricalCC

1.3.2.3.3. MobApiCreateCC

1.3.2.3.4. MobApiFreeCC

1.3.2.3.5. MobApiInitDynamicCC

1.3.2.3.6. MobApiUpdateDynamicCC

1.3.2.3.7. MobApiOverrideValueCC

1.3.2.3.8. MobApiPredictCC

1.3.3. On Road

1.3.3.1. Constants

1.3.3.1.1. MOBRD_STATIC_DATA_CONST

1.3.3.1.2. MOBRD_DYNAMIC_DATA_CONST

1.3.3.1.3. MOB_UP_SPEED

1.3.3.1.4. MOB_UP_REASON

1.3.3.1.5. MOB_DOWN_SPEED

1.3.3.1.6. MOB_DOWN_REASON

1.3.3.1.7. MOB_AVERAGE_SPEED

1.3.3.1.8. MOB_AVERAGE_REASON

1.3.3.1.9. MOB_CONTROLLING_VEHICLE

1.3.3.1.10. MOB_END

1.3.3.1.11. MOB_VISIBILITY

1.3.3.1.12. MOB_SLIPPERY_SURFACE

1.3.3.2. Data Structures

1.3.3.2.1. MobRDPrim

1.3.3.2.2. MobRDExternalDMAFF

Appendix A  Application Programmers' Interface
1.3.3.2.3. MobRDExternalFACC .......................................................... A19
1.3.3.2.4. MobExternalRD ................................................................. A19
1.3.3.2.5. MobRD ............................................................................... A19
1.3.3.3. Functions............................................................................. A19
1.3.3.3.1. MobApiGetDataRD ............................................................... A19
1.3.3.3.2. MobApiCreateHistoricalRD .............................................. A20
1.3.3.3.3. MobApiCreateRD ............................................................... A20
1.3.3.3.4. MobApiFreeRD ................................................................. A20
1.3.3.3.5. MobApiInitDynamicRD .................................................... A20
1.3.3.3.6. MobApiUpdateDynamicRD ............................................... A20
1.3.3.3.7. MobApiOverrideValueRD ................................................ A20
1.3.3.3.8. MobApiPredictRD ........................................................... A21
1.3.4. Gap Crossing ...................................................................... A22
1.3.4.1. Constants............................................................................ A22
1.3.4.1.1. MOBGAP_STATIC_DATA_CONST ........................................ A22
1.3.4.1.2. MOBGAP_DYNAMIC_DATA_CONST ..................................... A22
1.3.4.1.3. MOB_CONTROLLING_VEHICLE ......................................... A22
1.3.4.1.4. MOB_GAP_CROSSING_SPEED ......................................... A22
1.3.4.1.5. MOB_GAP_REASON ............................................................ A22
1.3.4.1.6. MOB_GAP_CROSSING_TIME ............................................ A22
1.3.4.1.7. MOB_END ........................................................................ A23
1.3.4.1.8. MOB_VISIBILITY ............................................................. A23
1.3.4.1.9. MOB_SLIPPERY_SURFACE .............................................. A23
1.3.4.1.10. MOB_WATER_STAGE ...................................................... A23
1.3.4.2. Data Structures ................................................................. A23
1.3.4.2.1. MobExternalGAP ............................................................. A23
1.3.4.2.2. MobGAP ........................................................................... A23
1.3.4.3. Functions............................................................................ A23
1.3.4.3.1. MobApiGetDataGAP ........................................................... A23
1.3.4.3.2. MobApiCreateHistoricalGAP ........................................... A24
1.3.4.3.3. MobApiCreateGAP ........................................................... A24
1.3.4.3.4. MobApiFreeGAP ............................................................... A24
1.3.4.3.5. MobApiInitDynamicGAP .................................................. A24
1.3.4.3.6. MobApiUpdateDynamicGAP ............................................ A24
1.3.4.3.7. MobApiOverrideValueGAP .............................................. A24
1.3.4.3.8. MobApiPredictGAP ........................................................ A25
1.4. Applied Mobility ..................................................................... A25
1.4.1. General ............................................................................... A25
1.4.1.1. Data Structures ................................................................. A25
1.4.1.1.1. MobMapDefinition .......................................................... A25
1.4.1.1.2. MobMap ........................................................... A26
1.4.1.1. Functions...................................................................... A26
1.4.1.1.1. MobApiCreateMap .......................................................... A26
1.4.1.1.2. MobApiFreeMap ................................................................. A26
1.4.2. Optimum Route ................................................................................. A27
1.4.2.1. Constants .................................................................................. A27
1.4.2.1.1. MOBOPTIMUMROUTE_FORMATION_COLUMN .................................. A27
1.4.2.1.2. MOBOPTIMUMROUTE_FORMATION_BOUNDING_OVERWATCH ................. A27
1.4.2.1.3. MOBOPTIMUMROUTE_FORMATION_COMBAT_LINES .................................. A27
1.4.2.1.4. MOBOPTIMUMROUTE_FORMATION_PARALLEL_COLUMNS .................. A27
1.4.2.1.5. MOBOPTIMUMROUTE_MAXCOLUMNS ............................................ A27
1.4.2.2. Data Structures ........................................................................ A27
1.4.2.2.1. MobOptimumRouteColumn .............................................................. A27
1.4.2.2.2. MobOptimumRouteBoundingOverwatch ........................................... A27
1.4.2.2.3. MobOptimumRouteCombatLines ..................................................... A27
1.4.2.2.4. MobOptimumRouteParallelColumns ............................................... A28
1.4.2.2.5. MobOptimumRouteFormation .......................................................... A28
1.4.2.2.6. MobOptimumRouteParameters ....................................................... A28
1.4.2.2.7. MobOptimumRoute ........................................................................... A28
1.4.2.3. Functions .................................................................................. A28
1.4.2.3.1. MobApiGetOptimumRoute ............................................................... A29
1.4.2.3.2. MobApiFreeOptimumRoute .............................................................. A29
1.4.3. Corridors ..................................................................................... A29
1.4.3.1. Constants .................................................................................. A29
1.4.3.1.1. MOBCORRIDOR_NOGO_URBAN ......................................................... A29
1.4.3.1.2. MOBCORRIDOR_COMPANY .............................................................. A29
1.4.3.1.3. MOBCORRIDOR_BATTALION ............................................................ A29
1.4.3.1.4. MOBCORRIDOR_BRIGADE ............................................................... A29
1.4.3.1.5. MOBCORRIDOR_DIVISION .............................................................. A29
1.4.3.2. Data Structures ........................................................................ A30
1.4.3.2.1. MobCorridor .................................................................................. A30
1.4.3.3. Functions .................................................................................. A30
1.4.3.3.1. MobApiGetCorridors ......................................................................... A30
1.4.3.3.2. MobApiFreeCorridor ......................................................................... A30
1.4.4. Isochrones (Time Contours) .......................................................... A30
1.4.4.1. Functions .................................................................................. A30
1.4.4.1.1. MobApiTimeContour ......................................................................... A30

Appendix A  Application Programmers' Interface  A5
1. API Reference

1.1. General

1.1.1. Constants

1.1.1.1. MOBVECSHAPE_POINT
   This constant is used in conjunction with the data structure MobVecShape (1.1.2.2) to specify the type variable. When the type variable is equal to MOBVECSHAPE_POINT, the MobVecShape data structure represents a point feature.
   
   #define MOBVECSHAPE_POINT     1

1.1.1.2. MOBVECSHAPE_LINE
   This constant is used in conjunction with the data structure MobVecShape (1.1.2.2) to specify the “type” variable. When the type variable is equal to MOBVECSHAPE_LINE, the MobVecShape data structure represents a linear feature.
   
   #define MOBVECSHAPE_LINE      2

1.1.1.3. MOBVECSHAPE_POLYGON
   This constant is used in conjunction with the data structure MobVecShape (1.1.2.2) to specify the type variable. When the type variable is equal to MOBVECSHAPE_POLYGON, the MobVecShape data structure represents a polygonal feature.
   
   #define MOBVECSHAPE_POLYGON   3

1.1.1.4. MOBCOVERAGE_SLOPE
   This constant is used in conjunction with the data structures MobFeatDMAFF (1.1.2.7) and MobFeatFACC (1.1.2.10) to specify the cov variable. When the cov variable is equal to MOBCOVERAGE_SLOPE, the data structures represent the attributes for the slope layer of Interim Terrain Data (ITD)/Vector Product Interim Terrain Data (VITD).
   
   #define MOBCOVERAGE_SLOPE   "SLP"

1.1.1.5. MOBCOVERAGE_VEGETATION
   This constant is used in conjunction with the data structures MobFeatDMAFF (1.1.2.7) and MobFeatFACC (1.1.2.10) to specify the cov variable. When the cov variable is equal to MOBCOVERAGE_VEGETATION, the data structures represent the attributes for the vegetation layer of ITD/VITD.
   
   #define MOBCOVERAGE_VEGETATION  "VEG"
1.1.1.6. MOBCOVERAGE_SURFACE_MATERIAL

This constant is used in conjunction with the data structures MobFeatDMAFF (1.1.2.7) and MobFeatFACC (1.1.2.10) to specify the cov variable. When the cov variable is equal to MOBCOVERAGE_SURFACE_MATERIAL, the data structures represent the attributes for the surface material layer of ITD/VITD.

#define MOBCOVERAGE_SURFACE_MATERIAL "SMC"

1.1.1.7. MOBCOVERAGE_OBSTACLES

This constant is used in conjunction with the data structures MobFeatDMAFF (1.1.2.7) and MobFeatFACC (1.1.2.10) to specify the cov variable. When the cov variable is equal to MOBCOVERAGE_OBSTACLES, the data structures represent the attributes for the obstacle layer of ITD/VITD.

#define MOBCOVERAGE_OBSTACLES "OBS"

1.1.1.8. MOBCOVERAGE_TRANSPORTATION

This constant is used in conjunction with the data structures MobFeatDMAFF (1.1.2.7) and MobFeatFACC (1.1.2.10) to specify the cov variable. When the cov variable is equal to MOBCOVERAGE_TRANSPORTATION, the data structures represent the attributes for the transportation layer of ITD/VITD.

#define MOBCOVERAGE_TRANSPORTATION "TRN"

1.1.1.9. MOBCOVERAGE_DRAINAGE

This constant is used in conjunction with the data structures MobFeatDMAFF (1.1.2.7) and MobFeatFACC (1.1.2.10) to specify the cov variable. When the cov variable is equal to MOBCOVERAGE_DRAINAGE, the data structures represent the attributes for the drainage layer of ITD/VITD.

#define MOBCOVERAGE_DRAINAGE "DRN"

1.1.2. Data Structures

1.1.2.1. MobLocation

This data structure is used to specify a location in decimal degrees of latitude and longitude relative to the World Geodetic System 1984 datum.

```c
typedef struct {
    double latitude;
    double longitude;
} MobLocation;
```

1.1.2.2. MobVecShape

This data structure is used to specify a vector shape. The shape may be a point, linear, or polygonal feature. The type variable will be one of the constants MOBVECSHAPE_POINT,
MOBVECSHAPE_LINE, or MOBVECSHAPE_POLYGON. The point_count variable is the number of MobLocation data structures stored in the points array. The elevation array is the elevation in meters above mean sea level of each MobLocation in the points array. The elevation variable may be NULL if no elevation data are available.

typedef struct {
    int                      type;
    int          point_count;
    long                 *elevation;
    MobLocation *points;
} MobVecShape;

1.1.2.3. MobHistWeather
This data structure is used to specify historical weather conditions. The month variable is used to specify the month of the year (i.e. 1=JAN, 2=FEB, 3=MAR, …). The condition variable is used to specify a dry(=1), average(=2), or wet(=3) month.

typedef struct {
    int month;
    int condition;
} MobHistWeather;

1.1.2.4. MobWeather
This data structure is used to specify actual weather conditions. The month variable is used to specify the month of the year (i.e. 1=JAN, 2=FEB, 3=MAR, …). The precipValid flag specifies that the value in precipInches variable is valid. The precipInches variable is the number of decimal inches of precipitation for the previous 24 hr. The tempValid flag specifies that the value in tempCelsius variable is valid. The tempCelsius variable is the average temperature for the previous 24 hr in decimal degrees Centigrade. The snowValid flag specifies that the values in snowDepthInches and snowDensity variables are valid. The snowDepthInches variable is the number of decimal inches of accumulated snow. The snowDensity variable is the specific gravity of the accumulated snow.

typedef struct {
    int month;
    int precipValid;
    double precipInches;
    int tempValid;
    double tempCelsius;
    int snowValid;
    double snowDepthInches;
    double snowDensity;
} MobWeather;

1.1.2.5. MobExternalType
This enumerated type is used in the data structures MobExternalDMAFF (1.1.2.8) and MobExternalFACC (1.1.2.11) to specify the coding scheme used. MOB_FACC specifies the Feature and Attribute Coding Catalogue (FACC) scheme. MOB_DMAFF specifies the DMA Feature File (DMAFF) scheme.
typedef enum {
    MOB_FACC = 1,
    MOB_DMAFF = 2,
} MobExternalType;

1.1.2.6. MobAttrDMAFF

This data structure is used to specify a DMAFF attribute/value pair. The id variable is the attribute name. The value variable is the value for the attribute.

typedef struct {
    char *id;
    long   value;
} MobAttrDMAFF;

1.1.2.7. MobFeatDMAFF

This data structure is used to specify the DMAFF coverage type, feature code, and attribute/value pairs. The cov variable specifies the coverage name and should be one of the constants MOBCOVERAGE_SLOPE (1.1.1.4), MOBCOVERAGE_VEGETATION (1.1.1.5), MOBCOVERAGE_SURFACE_MATERIAL (1.1.1.6), MOBCOVERAGE_OBSTACLES (1.1.1.7), MOBCOVERAGE_TRANSPORTATION (1.1.1.8), or MOBCOVERAGE_DRAINAGE (1.1.1.9). The f_code variable specifies the feature code. The dmaffAttrCount variable specifies the number of attribute/value pairs in the dmaffAttr array.

typedef struct dmaff_rec {
    char *cov;
    char *f_code;
    int dmaffAttrCount;
    MobAttrDMAFF *dmaffAttr;
} MobFeatDMAFF_rec, *MobFeatDMAFF;

1.1.2.8. MobExternalDMAFF

This data structure is used to specify external DMAFF records for the data structures MobExternalCC (1.3.2.2.1), MobExternalRD (1.3.3.2.4), and MobExternalGAP (1.3.4.2.1). The type variable must be MOB_DMAFF (1.1.2.5). The loc variable specifies a latitude-longitude for the external DMAFF records in a MobExternalCC only. The featCount variable specifies the number of valid MobFeatDMAFF records. The feat variable is an array of MobFeatDMAFF records.

typedef struct {
    MobExternalType type;
    MobLocation loc;
    int featCount;
    MobFeatDMAFF_rec *feat[4];
} MobExternalDMAFF_rec,*MobExternalDMAFF;

1.1.2.9. MobAttrFACC

This data structure is used to specify a FACC attribute/value pair. The id variable is the attribute name. The value variable is the value for the attribute.
typedef struct {
char *id;
long   value;
} MobAttrFACC;

1.1.2.10. MobFeatFACC

This data structure is used to specify the FACC coverage type, feature code, and attribute/value pairs. The cov variable specifies the coverage name and should be one of the constants MOBCOVERAGE_SLOPE (1.1.1.4), MOBCOVERAGE_VEGETATION (1.1.1.5), MOBCOVERAGE_SURFACE_MATERIAL (1.1.1.6), MOBCOVERAGE_OBSTACLES (1.1.1.7), MOBCOVERAGE_TRANSPORTATION (1.1.1.8), or MOBCOVERAGE_DRAINAGE (1.1.1.9). The f_code variable specifies the feature code. The faccAttrCount variable specifies the number of attribute/value pairs in the faccAttr array.

typedef struct {
    char   *cov;
    char   *f_code;
    int       faccAttrCount;
    MobAttrFACC *faccAttr;
} MobFeatFACC_rec,*MobFeatFACC;

1.1.2.11. MobExternalFACC

This data structure is used to specify external FACC records for the data structures MobExternalCC (1.3.2.2.1), MobExternalRD (1.3.3.2.4), and MobExternalGAP (1.3.4.2.1). The type variable must be MOB_FACC (1.1.2.5). The loc variable specifies a latitude-longitude for the external FACC records in a MobExternalCC only. The featCount variable specifies the number of valid MobFeatFACC records. The feat variable is an array of MobFeatFACC records.

typedef struct {
    MobExternalType type;
    MobLocation loc;
    int       featCount;
    MobFeatFACC_rec feat[4];
} MobExternalFACC_rec,*MobExternalFACC;

1.1.3. Functions

1.1.3.1. MobApiGetError

This function returns a character string describing the last error generated from an API call. The function returns NULL if there is no error.

const char *MobApiGetError();

<table>
<thead>
<tr>
<th>Return</th>
<th>const char *</th>
<th>Text description of last error, NULL = no error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Output</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>
1.2. Initialization

1.2.1. Functions

1.2.1.1. MobApiInitialize

This function initializes the API and must be called before any other API calls are made. This function is also used to zero the list of vehicles and other internal data.

```c
int MobApiInitialize(char *path, MobLocation sw, MobLocation ne);
```

<table>
<thead>
<tr>
<th>Return</th>
<th>status</th>
<th>0=Failed, 1=Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>path</td>
<td>Full path to mobility database directory</td>
</tr>
<tr>
<td></td>
<td>sw</td>
<td>South West corner of area to be analyzed</td>
</tr>
<tr>
<td></td>
<td>ne</td>
<td>North East corner of area to be analyzed</td>
</tr>
</tbody>
</table>

| Output | none |

1.3. Mobility Predictions

1.3.1. Vehicles

1.3.1.1. Data Structures

1.3.1.1.1. MobVehicleDataRec

This data structure is used to maintain the contents of a vehicle file including carriage returns and line feeds.

```c
typedef char *MobVehicleDataRec;
```

1.3.1.2. Functions

1.3.1.2.1. MobApiAddVehicleFromFile

This function loads the data for a vehicle specified by the parameter `filename`. A maximum of 100 vehicles may be loaded. The full path to the file will be generated from the `path` parameter of the last call to `MobApiInitialize` (1.2.1.1) in the following manner:

```c
path + "vehicles/" + filename
```

```c
int MobApiAddVehicleFromFile(char *filename);
```

<table>
<thead>
<tr>
<th>Return</th>
<th>Status</th>
<th>0=Failed, 1=Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>filename</td>
<td>Filename for vehicle data</td>
</tr>
<tr>
<td></td>
<td>none</td>
<td></td>
</tr>
</tbody>
</table>

1.3.1.2.2. MobApiAddVehicleFromData

This function loads the data for a vehicle specified by the parameter `vehicle`. A maximum of 100 vehicles may be loaded. This function provides a method to allow the contents of the vehicle files provided with the API to be stored in a database.
int MobApiAddVehicleFromData(MobVehicleDataRec vehicle);

<table>
<thead>
<tr>
<th>Return</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Failed</td>
</tr>
<tr>
<td>1</td>
<td>Success</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input</th>
<th>vehicle</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Vehicle data record</td>
</tr>
</tbody>
</table>

| Output | none |

1.3.2. Cross Country

1.3.2.1. Constants

1.3.2.1.1. MOBCC STATIC DATA CONST
This constant defines the size of the array for the staticData parameter to the functions MobApiGetDataCC (1.3.2.3.1) and MobApiCreateCC (1.3.2.3.3).

#define MOBCC_STATIC_DATA_COUNT 32

1.3.2.1.2. MOBCC DYNAMIC DATA CONST
This constant defines the size of the array for the dynamicData parameter to the functions MobApiGetDataCC (1.3.2.3.1) and MobApiCreateCC (1.3.2.3.3).

#define MOBCC_DYNAMIC_DATA_COUNT 20

1.3.2.1.3. MOB_UP_SPEED
This constant is used in the format parameter to the function MobApiPredictCC (1.3.2.3.8) to specify that the up-slope speed prediction should be returned in the results parameter.

#define MOB_UP_SPEED 1

1.3.2.1.4. MOB_UP_REASON
This constant is used in the format parameter to the function MobApiPredictCC (1.3.2.3.8) to specify that the up-slope reason code should be returned in the results parameter.

#define MOB_UP_REASON 2

1.3.2.1.5. MOB_DOWN_SPEED
This constant is used in the format parameter to the function MobApiPredictCC (1.3.2.3.8) to specify that the down-slope speed prediction should be returned in the results parameter.

#define MOB_DOWN_SPEED 3

1.3.2.1.6. MOB_DOWN_REASON
This constant is used in the format parameter to the function MobApiPredictCC (1.3.2.3.8) to specify that the down-slope reason code should be returned in the results parameter.

#define MOB_DOWN_REASON 4
1.3.2.1.7. MOB_CROSS_SLOPE_SPEED
   This constant is used in the format parameter to the function MobApiPredictCC (1.3.2.3.8) to specify that the cross-slope speed prediction should be returned in the results parameter.

#define MOB_CROSS_SLOPE_SPEED 5

1.3.2.1.8. MOB_CROSS_SLOPE_REASON
   This constant is used in the format parameter to the function MobApiPredictCC (1.3.2.3.8) to specify that the cross-slope reason code should be returned in the results parameter.

#define MOB_CROSS_SLOPE_REASON 6

1.3.2.1.9. MOB_AVERAGE_SPEED
   This constant is used in the format parameter to the function MobApiPredictCC (1.3.2.3.8) to specify that the average speed prediction, computed from the up-, down-, and cross-slope speed predictions, should be returned in the results parameter.

#define MOB_AVERAGE_SPEED 7

1.3.2.1.10. MOB_AVERAGE_REASON
#define MOB_AVERAGE_REASON 8

1.3.2.1.11. MOB_CONTROLLING_VEHICLE
   This constant is used in the format parameter to the function MobApiPredictCC (1.3.2.3.8) to specify that the zero based index of the slowest vehicle should be returned in the results parameter.

#define MOB_CONTROLLING_VEHICLE 9

1.3.2.1.12. MOB_END
   This constant is used in the format parameter to the function MobApiPredictCC (1.3.2.3.8) to specify the end of the list. No value will be returned in the results parameter.

#define MOB_END 1000

1.3.2.1.13. MOB_VISIBILITY
   This constant is used in the key parameter to the function MobApiOverrideValueCC (1.3.2.3.7) to specify that the value parameter is the driver visibility in meters. Driver visibility is the maximum distance the driver can see in front of the vehicle. The default value is 300 m. Fog, smoke, haze, precipitation, and nighttime can affect driver visibility.

#define MOB_VISIBILITY 1

1.3.2.1.14. MOB_SLIPPERY_SURFACE
   This constant is used in the key parameter to the function MobApiOverrideValueCC (1.3.2.3.7) to specify that the value parameter is the slippery surface flag. A value of 0 indicates that the surface is not slippery. A value of 1 indicates that the surface is slippery.
#define MOB_SLIPPERY_SURFACE   2

1.3.2.2. Data Structures

1.3.2.2.1. MobExternalCC

This data structure is used to specify external FACC or DMAFF records.

typedef union {
    MobExternalType   type;
    MobExternalDMAFF_rec  dmaff;
    MobExternalFACC_rec  facc;
} MobExternalCC_rec,*MobExternalCC;

1.3.2.2. MobCC

1.3.2.3. Functions

1.3.2.3.1. MobApiGetDataCC

This function is used to obtain a static terrain data record and a dynamic terrain data record from an internal terrain data record. The variable staticData must be dimensioned MOBCC_STATIC_DATA_COUNT (1.3.2.1.1). The variable dynamicData must be dimensioned MOBCC_DYNAMIC_DATA_COUNT (1.3.2.1.2).

int MobApiGetDataCC(MobCC cc, long *staticData, long *dynamicData);

<table>
<thead>
<tr>
<th>Return</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>cc</td>
</tr>
<tr>
<td></td>
<td>Internal terrain data record</td>
</tr>
<tr>
<td>Output</td>
<td>staticData</td>
</tr>
<tr>
<td></td>
<td>Static terrain data record</td>
</tr>
<tr>
<td></td>
<td>dynamicData</td>
</tr>
<tr>
<td></td>
<td>Dynamic terrain data record</td>
</tr>
</tbody>
</table>

1.3.2.3.2. MobApiCreateHistoricalCC

This function is used to create an internal terrain data record from a MobExternalCC (1.3.2.2.1) record and a MobHistWeather (1.1.2.3) record. The memory allocated for the MobCC (1.1.2.4) record must be released by calling the function MobApiFreeCC (1.3.2.3.4).

MobCC MobApiCreateHistoricalCC(MobExternalCC ext, MobHistWeather hw);

<table>
<thead>
<tr>
<th>Return</th>
<th>MobCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>ext</td>
</tr>
<tr>
<td></td>
<td>External terrain data record</td>
</tr>
<tr>
<td></td>
<td>hw</td>
</tr>
<tr>
<td></td>
<td>Historical weather record</td>
</tr>
<tr>
<td>Output</td>
<td>none</td>
</tr>
</tbody>
</table>

1.3.2.3.3. MobApiCreateCC

This function is used to create an internal terrain data record from a static terrain data record and a dynamic terrain data record. The variable staticData must be dimensioned MOBCC_STATIC_DATA_COUNT (1.3.2.1.1). The variable dynamicData must be dimensioned MOBCC_DYNAMIC_DATA_COUNT (1.3.2.1.2). The use of static and dynamic terrain data records can decrease the time it takes to make a prediction.
**MobCC** MobApiCreateCC(long *staticData, long *dynamicData);

<table>
<thead>
<tr>
<th>Return</th>
<th>MobCC</th>
<th>Internal terrain data record, NULL = Failed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>StaticData</td>
<td>Static terrain data record</td>
</tr>
<tr>
<td></td>
<td>DynamicData</td>
<td>Dynamic terrain data record</td>
</tr>
<tr>
<td>Output</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

### 1.3.2.3.4. MobApiFreeCC
This routine is used to free the memory allocated for a **MobCC** (1.3.2.2.2) record by the functions **MobApiCreateHistoricalCC** (1.3.2.3.2) and **MobApiCreateCC** (1.3.2.3.3).

**void** MobApiFreeCC(MobCC cc);

<table>
<thead>
<tr>
<th>Return</th>
<th>none</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>cc</td>
</tr>
<tr>
<td>Output</td>
<td>none</td>
</tr>
</tbody>
</table>

### 1.3.2.3.5. MobApiInitDynamicCC
This function is used to initialize the dynamic terrain data of an internal terrain data record from a **MobHistWeather** (1.1.2.3) record. This allows the internal terrain data record to be update with actual weather values using the function **MobApiUpdateDynamicCC** (1.3.2.3.6).

**int** MobApiInitDynamicCC(MobCC cc, MobHistWeather hw);

<table>
<thead>
<tr>
<th>Return</th>
<th>status</th>
<th>0=Failed, 1=Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>cc</td>
<td>Internal terrain data record</td>
</tr>
<tr>
<td></td>
<td>hw</td>
<td>Historical weather record</td>
</tr>
<tr>
<td>Output</td>
<td>none</td>
<td></td>
</tr>
</tbody>
</table>

### 1.3.2.3.6. MobApiUpdateDynamicCC
This function is used to update an internal terrain data record from a **MobWeather** (1.1.2.4) record.

**int** MobApiUpdateDynamicCC(MobCC cc, MobWeather weather);

<table>
<thead>
<tr>
<th>Return</th>
<th>status</th>
<th>0=Failed, 1=Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>weather</td>
<td>Daily weather record</td>
</tr>
<tr>
<td>Input/Output</td>
<td>cc</td>
<td>Internal terrain data record</td>
</tr>
</tbody>
</table>

### 1.3.2.3.7. MobApiOverrideValueCC
This function is used to override driver visibility and slippery surface for an internal terrain data record.

**int** MobApiOverrideValueCC(MobCC cc, int key, long value);

<table>
<thead>
<tr>
<th>Return</th>
<th>status</th>
<th>0=Failed, 1=Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>cc</td>
<td>Internal terrain data record</td>
</tr>
<tr>
<td></td>
<td>key</td>
<td>Override value key</td>
</tr>
<tr>
<td></td>
<td>value</td>
<td>Data value</td>
</tr>
<tr>
<td>Output</td>
<td>none</td>
<td></td>
</tr>
</tbody>
</table>
Override Value Key | Data Value Description
---|---
MOB_VISIBILITY | Driver visibility in meters.
MOB_SLIPPERY_SURFACE | 0=Not Slippery, 1=Slippery

### 1.3.2.3.8. MobApiPredictCC

This function is used to predict cross-country mobility for an internal terrain data record. The values returned in the *results* array are determined by the values in the *format* array.

```c
int MobApiPredictCC(MobCC cc, int *format, long *results);
```

<table>
<thead>
<tr>
<th>Return</th>
<th>status</th>
<th>0=Failed, 1=Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>cc</td>
<td>Internal terrain data record</td>
</tr>
<tr>
<td></td>
<td>format</td>
<td>List of format constants which describe the results</td>
</tr>
<tr>
<td>Output</td>
<td>results</td>
<td>Output values as specified in the list of format constants</td>
</tr>
</tbody>
</table>

### Format Constants

<table>
<thead>
<tr>
<th>Format Constants</th>
<th>Description of Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOB_UP_SPEED</td>
<td>Speed Of Vehicle in KPH.</td>
</tr>
<tr>
<td>MOB_UP_REASON</td>
<td>A reason code as described in the reason code table below.</td>
</tr>
<tr>
<td>MOB_DOWN_SPEED</td>
<td>Speed Of Vehicle in KPH.</td>
</tr>
<tr>
<td>MOB_DOWN_REASON</td>
<td>A reason code as described in the reason code table below.</td>
</tr>
<tr>
<td>MOB_CROSS_SLOPE_SPEED</td>
<td>Speed Of Vehicle in KPH.</td>
</tr>
<tr>
<td>MOB_CROSS_SLOPE_REASON</td>
<td>A reason code as described in the reason code table below.</td>
</tr>
<tr>
<td>MOB_AVERAGE_SPEED</td>
<td>Speed Of Vehicle in KPH.</td>
</tr>
<tr>
<td>MOB_AVERAGE_REASON</td>
<td>A reason code as described in the reason code table below.</td>
</tr>
<tr>
<td>MOB_CONTROLLING_VEHICLE</td>
<td>A zero based index of the vehicle which has the slowest predicted speed.</td>
</tr>
<tr>
<td>MOB_END</td>
<td>No results returned used to specify end of format list.</td>
</tr>
</tbody>
</table>

### Cross-Country Reason Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-99</td>
<td>Unknown &quot;NO-GO&quot;</td>
</tr>
<tr>
<td>-10</td>
<td>Water &quot;NO-GO&quot;</td>
</tr>
<tr>
<td>-9</td>
<td>Tipping on side slope “NO-GO”</td>
</tr>
<tr>
<td>-8</td>
<td>Sliding on side slope NO-GO</td>
</tr>
<tr>
<td>-7</td>
<td>Soil “NO-GO” on level (VCI)</td>
</tr>
<tr>
<td>-6</td>
<td>Obstacle override “NO-GO”</td>
</tr>
<tr>
<td>-5</td>
<td>Vegetation override “NO-GO”</td>
</tr>
<tr>
<td>-4</td>
<td>Obstacle belly interference &quot;NO-GO&quot;</td>
</tr>
<tr>
<td>-3</td>
<td>Obstacle clearance interference &quot;NO-GO&quot;</td>
</tr>
<tr>
<td>-2</td>
<td>Soil &amp; slope resistance “NO-GO”</td>
</tr>
</tbody>
</table>
Inability to brake (visibility) “NO-GO”

0  No prediction made

1  Ride dynamics (vride) limit

2  Tire speed limit

3  Soil, slope, & vegetation resistances

4  Visibility

5  Maneuver around obstacles & vegetation

6  Maneuver around vegetation (overriding obstacles)

7  Obstacle impact (VOOB) speed

8  Obstacle override force

9  Driver prudence overriding vegetation

10 External (scenario) limit

11 AASHTO curvature speed limit

12 Sliding on curves/side slopes

13 Tipping on curves/side slopes

20 Urban

99 Unknown cause

Note: Negative codes represent “NO-GO” reasons.
Positive codes represent controlling factor reasons.

1.3.3. On Road

1.3.3.1. Constants

1.3.3.1.1. MOBRD_STATIC_DATA_CONST
This constant defines the size of the array for the staticData parameter to the functions MobApiGetDataRD (1.3.3.3.1) and MobApiCreateRD (1.3.3.3.3).

#define MOBRD_STATIC_DATA_COUNT 32

1.3.3.1.2. MOBRD_DYNAMIC_DATA_CONST
This constant defines the size of the array for the dynamicData parameter to the functions MobApiGetDataRD (1.3.3.3.1) and MobApiCreateRD (1.3.3.3.3).

#define MOBRD_DYNAMIC_DATA_COUNT 20

1.3.3.1.3. MOB_UP_SPEED

#define MOB_UP_SPEED 1

1.3.3.1.4. MOB_UP_REASON

#define MOB_UP_REASON 2
1.3.3.1.5. MOB_DOWN_SPEED
#define MOB_DOWN_SPEED 3

1.3.3.1.6. MOB_DOWN_REASON
#define MOB_DOWN_REASON 4

1.3.3.1.7. MOB_AVERAGE_SPEED
This constant is used in the format parameter to the function MobApiPredictRD (1.3.3.3.8) to specify that the average speed prediction, computed from the up- and down-speed predictions, should be returned in the results parameter.
#define MOB_AVERAGE_SPEED 7

1.3.3.1.8. MOB_AVERAGE_REASON
#define MOB_AVERAGE_REASON 8

1.3.3.1.9. MOB_CONTROLLING_VEHICLE
This constant is used in the format parameter to the function MobApiPredictRD (1.3.3.3.8) to specify that the zero-based index of the slowest vehicle should be returned in the results parameter.
#define MOB_CONTROLLING_VEHICLE 9

1.3.3.1.10. MOB_END
This constant is used in the format parameter to the function MobApiPredictRD (1.3.3.3.8) to specify the end of the list. No value will be returned in the results parameter.
#define MOB_END 1000

1.3.3.1.11. MOB_VISIBILITY
This constant is used in the key parameter to the function MobApiOverrideValueRD (1.3.3.3.7) to specify that the value parameter is the driver visibility in meters. Driver visibility is the maximum distance the driver can see in front of the vehicle. The default value is 300 m. Fog, smoke, haze, precipitation, and nighttime can affect driver visibility.
#define MOB_VISIBILITY 1

1.3.3.1.12. MOB_SLIPPERY_SURFACE
This constant is used in the key parameter to the function MobApiOverrideValueRD (1.3.3.3.7) to specify that the value parameter is the slippery surface flag. A value of 0 indicates that the surface is not slippery. A value of 1 indicates that the surface is slippery.
#define MOB_SLIPPERY_SURFACE 2
1.3.3.2. Data Structures

1.3.3.2.1. MobRDPrim

typedef struct {
    int nLoc;
    double *lat;
    double *lon;
    double *elev;
} MobRDPrimRec, *MobRDPrim;

1.3.3.2.2. MobRDExternalDMAFF

typedef struct {
    MobExternalType type;
    MobExternalDMAFF_rec dmaff;
    int numPrims;
    MobRDPrim *prims;
} MobRDExternalDMAFF_rec, *MobRDExternalDMAFF;

1.3.3.2.3. MobRDExternalFACC

typedef struct {
    MobExternalType type;
    MobExternalFACC_rec facc;
    int numPrims;
    MobRDPrim *prims;
} MobRDExternalFACC_rec, *MobRDExternalFACC;

1.3.3.2.4. MobExternalRD

typedef union {
    MobExternalType type;
    MobRDExternalDMAFF_rec dmaff;
    MobRDExternalFACC_rec facc;
} MobExternalRD_rec, *MobExternalRD;

1.3.3.2.5. MobRD

1.3.3.3. Functions

1.3.3.3.1. MobApiGetDataRD

int MobApiGetDataRD(MobRD rd, long *staticData, long *dynamicData);

<table>
<thead>
<tr>
<th>Return</th>
<th>Status</th>
<th>0=Failed, 1=Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>rd</td>
<td>Internal terrain data record</td>
</tr>
<tr>
<td>Output</td>
<td>staticData</td>
<td>Static terrain data record</td>
</tr>
<tr>
<td></td>
<td>dynamicData</td>
<td>Dynamic terrain data record</td>
</tr>
</tbody>
</table>
### 1.3.3.3.2. MobApiCreateHistoricalRD

```c
MobRD MobApiCreateHistoricalRD(MobExternalRD ext, MobHistWeather hw);
```

<table>
<thead>
<tr>
<th>Return</th>
<th>MobRD</th>
<th>Internal terrain data record, NULL = Failed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>ext</td>
<td>External terrain data record</td>
</tr>
<tr>
<td></td>
<td>hw</td>
<td>Historical weather record</td>
</tr>
<tr>
<td>Output</td>
<td>none</td>
<td></td>
</tr>
</tbody>
</table>

### 1.3.3.3.3. MobApiCreateRD

```c
MobRD MobApiCreateRD(long *staticData, long *dynamicData);
```

<table>
<thead>
<tr>
<th>Return</th>
<th>MobRD</th>
<th>Internal terrain data record, NULL = failed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>staticData</td>
<td>Static terrain data record</td>
</tr>
<tr>
<td></td>
<td>dynamicData</td>
<td>Dynamic terrain data record</td>
</tr>
<tr>
<td>Output</td>
<td>none</td>
<td></td>
</tr>
</tbody>
</table>

### 1.3.3.3.4. MobApiFreeRD

```c
void MobApiFreeRD(MobRD rd);
```

<table>
<thead>
<tr>
<th>Return</th>
<th>none</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>rd</td>
</tr>
<tr>
<td>Output</td>
<td>none</td>
</tr>
</tbody>
</table>

### 1.3.3.3.5. MobApiInitDynamicRD

```c
int MobApiInitDynamicRD(MobRD rd, MobHistWeather hw);
```

<table>
<thead>
<tr>
<th>Return</th>
<th>status</th>
<th>0=Failed, 1=Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>rd</td>
<td>Internal terrain data record</td>
</tr>
<tr>
<td></td>
<td>hw</td>
<td>Historical weather record</td>
</tr>
<tr>
<td>Output</td>
<td>none</td>
<td></td>
</tr>
</tbody>
</table>

### 1.3.3.3.6. MobApiUpdateDynamicRD

```c
int MobApiUpdateDynamicRD(MobRD rd, MobWeather weather);
```

<table>
<thead>
<tr>
<th>Return</th>
<th>status</th>
<th>0=Failed, 1=Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>weather</td>
<td>Daily weather record</td>
</tr>
<tr>
<td>Input/Output</td>
<td>rd</td>
<td>Internal terrain data record</td>
</tr>
</tbody>
</table>

### 1.3.3.3.7. MobApiOverrideValueRD
int MobApiOverrideValueRD(MobRD rd, int key, long value);

Return  status  0=Failed, 1=Success
Input    rd  Internal terrain data record
         key  Override value key
         value  Data value
Output   none

1.3.3.3.8. MobApiPredictRD

int MobApiPredictRD(MobRD rd, int *format, long *results);

Return  status  0=Failed, 1=Success
Input    rd  Internal terrain data record
         format  List of format constants which describe the results
Output   results  Output values as specified in the format parameter

<table>
<thead>
<tr>
<th>Format Constants</th>
<th>Description of Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOB_UP_SPEED</td>
<td>Speed Of Vehicle in KPH.</td>
</tr>
<tr>
<td>MOB_UP_REASON</td>
<td>A reason code as described in the</td>
</tr>
<tr>
<td></td>
<td>reason code table below.</td>
</tr>
<tr>
<td>MOB_DOWN_SPEED</td>
<td>Speed Of Vehicle in KPH.</td>
</tr>
<tr>
<td>MOB_DOWN_REASON</td>
<td>A reason code as described in the</td>
</tr>
<tr>
<td></td>
<td>reason code table below.</td>
</tr>
<tr>
<td>MOB_AVERAGE_SPEED</td>
<td>Speed Of Vehicle in KPH.</td>
</tr>
<tr>
<td>MOB_AVERAGE_REASON</td>
<td>A reason code as described in the</td>
</tr>
<tr>
<td></td>
<td>reason code table below.</td>
</tr>
<tr>
<td>MOB_CONTROLLING_VEHICLE</td>
<td>A zero based index of the vehicle</td>
</tr>
<tr>
<td></td>
<td>which has the slowest predicted speed.</td>
</tr>
<tr>
<td>MOB_END</td>
<td>No results returned used to specify</td>
</tr>
<tr>
<td></td>
<td>end of format list.</td>
</tr>
</tbody>
</table>

On-Road Reason Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-99</td>
<td>Unknown &quot;NO-GO&quot;</td>
</tr>
<tr>
<td>-7</td>
<td>Soil “NO-GO” on level (VCI)</td>
</tr>
<tr>
<td>-6</td>
<td>Obstacle override “NO-GO”</td>
</tr>
<tr>
<td>-5</td>
<td>Vegetation override “NO-GO”</td>
</tr>
<tr>
<td>-4</td>
<td>Obstacle belly interference &quot;NO-GO&quot;</td>
</tr>
<tr>
<td>-3</td>
<td>Obstacle clearance interference &quot;NO-GO&quot;</td>
</tr>
<tr>
<td>-2</td>
<td>Soil &amp; slope resistance “NO-GO”</td>
</tr>
<tr>
<td>-1</td>
<td>Inability to brake (visibility) “NO-GO”</td>
</tr>
<tr>
<td>0</td>
<td>No prediction made</td>
</tr>
<tr>
<td>1</td>
<td>Ride dynamics (vride) limit</td>
</tr>
<tr>
<td>2</td>
<td>Tire speed limit</td>
</tr>
<tr>
<td>3</td>
<td>Soil, slope, &amp; vegetation resistances</td>
</tr>
<tr>
<td>4</td>
<td>Visibility</td>
</tr>
</tbody>
</table>
1.3.4. Gap Crossing

1.3.4.1. Constants

1.3.4.1.1. MOBGAP_STATIC_DATA_CONST
This constant defines the size of the array for the staticData parameter to the functions MobApiGetDataGAP (1.3.4.3.1) and MobApiCreateGAP (1.3.4.3.3).

#define MOBGAP_STATIC_DATA_COUNT 24

1.3.4.1.2. MOBGAP_DYNAMIC_DATA_CONST
This constant defines the size of the array for the dynamicData parameter to the functions MobApiGetDataGAP (1.3.4.3.1) and MobApiCreateGAP (1.3.4.3.3).

#define MOBGAP_DYNAMIC_DATA_COUNT 20

1.3.4.1.3. MOB_CONTROLLING_VEHICLE
This constant is used in the format parameter to the function MobApiPredictGAP (1.3.4.3.8) to specify that the zero-based index of the slowest vehicle should be returned in the results parameter.

#define MOB_CONTROLLING_VEHICLE 9

1.3.4.1.4. MOB_GAP_CROSSING_SPEED

#define MOB_GAP_CROSSING_SPEED 20

1.3.4.1.5. MOB_GAP_REASON

#define MOB_GAP_REASON 21

1.3.4.1.6. MOB_GAP_CROSSING TIME

#define MOB_GAP_CROSSING_TIME 22
1.3.4.1.7. MOB_END
This constant is used in the format parameter to the function MobApiPredictGAP (1.3.4.3.8) to specify the end of the list. No value will be returned in the results parameter.

#define MOB_END 1000

1.3.4.1.8. MOB_VISIBILITY
This constant is used in the key parameter to the function MobApiOverrideValueGAP (1.3.4.3.7) to specify that the value parameter is the driver visibility in meters. Driver visibility is the maximum distance the driver can see in front of the vehicle. The default value is 300 m. Fog, smoke, haze, precipitation, and nighttime may affect driver visibility.

#define MOB_VISIBILITY 1

1.3.4.1.9. MOB_SLIPPERY_SURFACE
This constant is used in the key parameter to the function MobApiOverrideValueGAP (1.3.4.3.7) to specify that the value parameter is the slippery surface flag. A value of 0 indicates that the surface is not slippery. A value of 1 indicates that the surface is slippery.

#define MOB_SLIPPERY_SURFACE 2

1.3.4.1.10. MOB_WATER_STAGE

#define MOB_WATER_STAGE 3

1.3.4.2. Data Structures
1.3.4.2.1. MobExternalGAP
typedef union {
   MobExternalType type;
   MobExternalDMAFF_rec dmaff;
   MobExternalFACC_rec facc;
}MobExternalGAP_rec,*MobExternalGAP;

1.3.4.3. Functions
1.3.4.3.1. MobApiGetDataGAP
int MobApiGetDataGAP(MobGAP gap, long *staticData, long *dynamicData);

<table>
<thead>
<tr>
<th>Return</th>
<th>status</th>
<th>0=Failed, 1=Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>gap</td>
<td>Internal terrain data record</td>
</tr>
<tr>
<td>Output</td>
<td>staticData</td>
<td>Static terrain data record</td>
</tr>
<tr>
<td></td>
<td>dynamicData</td>
<td>Dynamic terrain data record</td>
</tr>
</tbody>
</table>
1.3.4.3.2. MobApiCreateHistoricalGAP

\texttt{MobGAP MobApiCreateHistoricalGAP(MobExternalGAP extGAP,}
\texttt{MobHistWeather hw);};

| Return | gap | Internal terrain data record, NULL = Failed |
| Input  | ext | External terrain data record  |
| hw     |     | Historical weather record    |
| Output | none|

1.3.4.3.3. MobApiCreateGAP

\texttt{MobGAP MobApiCreateGAP(long *staticData, long *dynamicData);};

| Return | gap | Internal terrain data record, NULL = failed |
| Input  | staticData | Static terrain data record  |
|        | dynamicData | Dynamic terrain data record |
| Output | none |

1.3.4.3.4. MobApiFreeGAP

\texttt{void MobApiFreeGAP(MobGAP gap);};

| Return | none |
| Input  | gap | Gap record to free    |
| Output | none |

1.3.4.3.5. MobApiInitDynamicGAP

\texttt{int MobApiInitDynamicGAP(MobGAP gap, MobHistWeather hw);};

| Return | status | 0=Failed, 1=Success |
| Input  | gap    | Internal terrain data record  |
| hw     |        | Historical weather record |
| Output | none   |

1.3.4.3.6. MobApiUpdateDynamicGAP

\texttt{int MobApiUpdateDynamicGAP(MobGAP gap, MobWeather weather);};

| Return | status | 0=Failed, 1=Success |
| Input  | weather | Daily weather record    |
| Input/Output | gap | Internal terrain data record |

1.3.4.3.7. MobApiOverrideValueGAP

\texttt{int MobApiOverrideValueGAP(MobGAP gap, int key, long value);};

| Return | status | 0=Failed, 1=Success |
| Input  | gap    | Internal terrain data record  |
|        | key    | Override value key       |
|        | value  | Data value               |
| Output | none   |
1.3.4.3.8. MobApiPredictGAP

```c
int MobApiPredictGAP(MobGAP gap, int *format, long *results);
```

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gap</td>
<td>Internal terrain data record</td>
</tr>
<tr>
<td>format</td>
<td>List of format constants which describe the results</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>results</td>
<td>Output values as specified in output format</td>
</tr>
</tbody>
</table>

**Format Constants**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOB_CONTROLLING_VEHICLE</td>
<td>A zero-based index of the vehicle which has the slowest predicted speed.</td>
</tr>
<tr>
<td>MOB_GAP_CROSSING_SPEED</td>
<td>Gap crossing speed in KPH.</td>
</tr>
<tr>
<td>MOB_GAP_REASON</td>
<td>A reason code as described in the reason code table below.</td>
</tr>
<tr>
<td>MOB_GAP_CROSSING_TIME</td>
<td>Gap crossing time in seconds.</td>
</tr>
<tr>
<td>MOB_END</td>
<td>No results returned used to specify end of format list.</td>
</tr>
</tbody>
</table>

**Code**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-8</td>
<td>“NO-GO”, Insufficient traction available</td>
</tr>
<tr>
<td>-4</td>
<td>“NO-GO”, Geometric interference within gap</td>
</tr>
<tr>
<td>-2</td>
<td>“NO-GO”, Vegetation too dense</td>
</tr>
<tr>
<td>-1</td>
<td>“NO-GO”, Water crossing</td>
</tr>
<tr>
<td>0</td>
<td>No prediction made</td>
</tr>
<tr>
<td>1</td>
<td>Span</td>
</tr>
<tr>
<td>2</td>
<td>Ford</td>
</tr>
<tr>
<td>3</td>
<td>Swim</td>
</tr>
</tbody>
</table>

Note: Negative reason codes range from -15 to -1. Negative reason codes not listed above are the sum of multiple negative reason codes listed above.

### 1.4. Applied Mobility

#### 1.4.1. General

#### 1.4.1.1. Data Structures

1.4.1.1.1. MobMapDefinition

```c
typedef struct MobMapDefinition {
    double origin_latitude;
    double origin_longitude;
    double resolution_latitude;
    double resolution_longitude;
    int width_dimension;
    int height_dimension;
} MobMapDefinition;
```
1.4.1.1.2. MobMap

1.4.1.1. Functions

1.4.1.1.1. MobApiCreateMap

\[ \text{MobMap MobApiCreateMap(MobMapDefinition } \ast \text{md);} \]

<table>
<thead>
<tr>
<th>Return</th>
<th>MobMap</th>
<th>Mobility cell map, NULL = Failed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>md</td>
<td>Mobility Map definition record</td>
</tr>
<tr>
<td>Output</td>
<td>none</td>
<td></td>
</tr>
</tbody>
</table>

1.4.1.1.2. MobApiFreeMap

\[ \text{void MobApiFreeMap(MobMap map);} \]

<table>
<thead>
<tr>
<th>Return</th>
<th>none</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>map</td>
</tr>
<tr>
<td></td>
<td>Mobility cell map to free</td>
</tr>
<tr>
<td>Output</td>
<td>none</td>
</tr>
</tbody>
</table>

1.4.1.1.3. MobApiMapAddCC

\[ \text{int MobApiMapAddCC(MobMap map, long } \ast \text{elev, long } \ast \text{speed);} \]

<table>
<thead>
<tr>
<th>Return</th>
<th>status</th>
<th>0=Failure, 1=Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>elev</td>
<td>2-dim data array with same definition as Mobility cell map</td>
</tr>
<tr>
<td></td>
<td>speed</td>
<td>3 element array of pointers to 2-dimensional data arrays of speed values with same definition as mobility cell map</td>
</tr>
<tr>
<td>Input/Output</td>
<td>map</td>
<td>Mobility cell map</td>
</tr>
<tr>
<td>Output</td>
<td>none</td>
<td></td>
</tr>
</tbody>
</table>

1.4.1.1.4. MobApiMapAddRD

\[ \text{int MobApiMapAddRD(MobMap map, MobVecShape } \ast \text{shape, long } \ast \text{speed);} \]

<table>
<thead>
<tr>
<th>Return</th>
<th>status</th>
<th>0=Failure, 1=Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>shape</td>
<td>Geometric shape of road feature</td>
</tr>
<tr>
<td></td>
<td>speed</td>
<td>Speed data for shape</td>
</tr>
<tr>
<td>Input/Output</td>
<td>map</td>
<td>Mobility cell map</td>
</tr>
<tr>
<td>Output</td>
<td>none</td>
<td></td>
</tr>
</tbody>
</table>

1.4.1.1.5. MobApiMapAddGap

\[ \text{int MobApiMapAddGAP(MobMap map, MobVecShape } \ast \text{shape, long } \ast \text{pred);} \]

<table>
<thead>
<tr>
<th>Return</th>
<th>status</th>
<th>0=Failure, 1=Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>shape</td>
<td>Geometric shape of gap feature</td>
</tr>
<tr>
<td></td>
<td>pred</td>
<td>Mobility prediction</td>
</tr>
<tr>
<td>Input/Output</td>
<td>map</td>
<td>Mobility cell map</td>
</tr>
<tr>
<td>Output</td>
<td>none</td>
<td></td>
</tr>
</tbody>
</table>
1.4.2. Optimum Route

1.4.2.1. Constants

1.4.2.1.1. MOBOPTIMUMROUTE_FORMATION_COLUMN

#define MOBOPTIMUMROUTE_FORMATION_COLUMN 1

1.4.2.1.2. MOBOPTIMUMROUTE_FORMATION_BOUNDING_OVERWATCH

#define MOBOPTIMUMROUTE_FORMATION_BOUNDING_OVERWATCH 2

1.4.2.1.3. MOBOPTIMUMROUTE_FORMATION_COMBAT_LINES

#define MOBOPTIMUMROUTE_FORMATION_COMBAT_LINES 3

1.4.2.1.4. MOBOPTIMUMROUTE_FORMATION_PARALLEL_COLUMNS

#define MOBOPTIMUMROUTE_FORMATION_PARALLEL_COLUMNS 4

1.4.2.1.5. MOBOPTIMUMROUTE_MAXCOLUMNS

#define MOBOPTIMUMROUTE_MAXCOLUMNS 64

1.4.2.2. Data Structures

1.4.2.2.1. MobOptimumRouteColumn

typedef struct {
    long FormationType;
    long MinimumVehicleSpacing;
    long MaximumVehicleSpacing;
} MobOptimumRouteColumn;

1.4.2.2.2. MobOptimumRouteBoundingOverwatch

typedef struct {
    long FormationType;
    long BoundingDistance; /* meters */
} MobOptimumRouteBoundingOverwatch;

1.4.2.2.3. MobOptimumRouteCombatLines

typedef struct {
    long FormationType;
    long VehicleSpacingSideToSide; /* meters */
    long MaximumDepthOfVehicles; /* meters */
} MobOptimumRouteCombatLine;
1.4.2.2.4. MobOptimumRouteParallelColumns

typedef struct {
    long    FormationType;
    long    MinimumVehicleSpacing; /* meters */
    long    MaximumVehicleSpacing; /* meters */
    long    VehicleSpacingSideToSide; /* meters */
    long    MaximumDepthOfColumnLeadVehicles; /* meters */
    long    NumberOfColumns;
    long    NumberOfVehiclesPerColumn[MOBOPTIMUMROUTE_MAXCOLUMNS];
} MobOptimumRouteParallelColumns;

1.4.2.2.5. MobOptimumRouteFormation

typedef union {
    long                                FormationType;
    MobOptimumRouteColumn             Column;
    MobOptimumRouteBoundingOverwatch BoundingOverwatch;
    MobOptimumRouteCombatLine         CombatLine;
    MobOptimumRouteParallelColumns    ParallelColumns;
} MobOptimumRouteFormation;

1.4.2.2.6. MobOptimumRouteParameters

typedef struct {
    long                    Width;          /* Unit Width in meters */
    long                    ChokeFlag;      /* 1=Avoid choke points 0=not */
    long                    MaximumSpeed;   /* KPH */
    long                    NumberOfVehiclesInFormation;
    MobOptimumRouteFormation Formation;
} MobOptimumRouteParameters;

1.4.2.2.7. MobOptimumRoute

typedef struct {
    MobVecShape   route;
    MobVecShape   border;
    long         *times;
} MobOptimumRoute;

1.4.2.3. Functions
1.4.2.3.1. MobApiGetOptimumRoute

\textit{MobOptimumRoute *MobApiGetOptimumRoute(MobMap map, int loc_count,}
\textit{ MobLocation *loc, MobOptimumRouteParameters *params);}:

<table>
<thead>
<tr>
<th>Return</th>
<th>route</th>
<th>Mobility optimum route</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>map</td>
<td>Mobility cell map</td>
</tr>
<tr>
<td></td>
<td>loc_count</td>
<td>Count of way points including start and end location</td>
</tr>
<tr>
<td></td>
<td>loc</td>
<td>Way point locations including start and end location</td>
</tr>
<tr>
<td></td>
<td>params</td>
<td>Mobility optimum route parameters</td>
</tr>
<tr>
<td>Output</td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>

1.4.2.3.2. MobApiFreeOptimumRoute

\textit{void MobApiFreeOptimumRoute(MobOptimumRoute *route);}:

<table>
<thead>
<tr>
<th>Return</th>
<th>none</th>
<th>none</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>route</td>
<td>Mobility optimum route to free</td>
</tr>
<tr>
<td>Output</td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>

1.4.3. Corridors

1.4.3.1. Constants

1.4.3.1.1. MOBCORRIDOR_NOGO_URBAN

\#define MOBCORRIDOR_NOGO_URBAN 1

1.4.3.1.2. MOBCORRIDOR_COMPANY

\#define MOBCORRIDOR_COMPANY 2

1.4.3.1.3. MOBCORRIDOR_BATTALION

\#define MOBCORRIDOR_BATTALION 3

1.4.3.1.4. MOBCORRIDOR_BRIGADE

\#define MOBCORRIDOR_BRIGADE 4

1.4.3.1.5. MOBCORRIDOR_DIVISION

\#define MOBCORRIDOR_DIVISION 5

Appendix A   Application Programmers' Interface  A29
1.4.3.2. Data Structures

1.4.3.2.1. MobCorridor

typedef struct {
    int     shapes_count;
    MobVecShape *shapes;
    long   *types;
} MobCorridor;

1.4.3.3. Functions

1.4.3.3.1. MobApiGetCorridors

MobCorridor *MobApiGetCorridors(MobMap map);

<table>
<thead>
<tr>
<th>Return</th>
<th>corridors</th>
<th>Mobility corridor record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>map</td>
<td>Mobility cell map</td>
</tr>
<tr>
<td>Output</td>
<td>none</td>
<td></td>
</tr>
</tbody>
</table>

1.4.3.3.2. MobApiFreeCorridor

void MobApiFreeCorridor(MobCorridor *corridor);

<table>
<thead>
<tr>
<th>Return</th>
<th>none</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>corridor</td>
</tr>
<tr>
<td>Output</td>
<td>none</td>
</tr>
</tbody>
</table>

1.4.4. Isochrones (Time Contours)

1.4.4.1. Functions

1.4.4.1.1. MobApiTimeContour

long *MobApiTimeContour(MobMap map, MobLocation *start);

<table>
<thead>
<tr>
<th>Return</th>
<th>time</th>
<th>2-dim array of longs with same definition as Mobility cell map</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>map</td>
<td>mobility map</td>
</tr>
<tr>
<td></td>
<td>start</td>
<td>start point</td>
</tr>
<tr>
<td>Output</td>
<td>none</td>
<td></td>
</tr>
</tbody>
</table>
Methodology for Mobility Tactical Decision Aids Incorporated into the Joint Mapping Tool Kit

George B. McKinley, Benjamin T. Webb, David A. Horner

The purpose of this report is to present the methodology and show example results from the mobility Tactical Decision Aids (TDAs) that are incorporated into the Joint Mapping Tool Kit. These TDAs are the optimum route selector, the Combat Maneuver Model (CMM), the mobility network generator, and the isochrone generator. The optimum route selector chooses a route of a given width between designated map positions and locates choke points along that route. The CMM computes the time for a unit of vehicles in one of four basic combat formations to traverse the selected route. The mobility network generator produces an edge/node network of corridors and categorizes the resultant edges based on the largest military unit that the edge can handle. The isochrone generator predicts the time required for a single vehicle to reach all areas within a speed matrix from an initial location. Application programmer interfaces for all the mobility TDAs are presented in Appendix A.