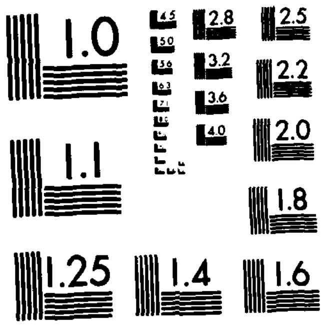
AD-A143 922   INTELLIGENT ADVISORS FOR CROSS-COUNTRY ROUTE PLANNING   1/1
(U) SMART SYSTEMS TECHNOLOGY INC MCLEAN VA MAY 84
ETL-0365 DAAK70-83-P-3175

UNCLASSIFIED   F/G 15/7   NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3

ETL-0365

2

AD-A143 922

INTELLIGENT ADVISORS FOR CROSS-COUNTRY ROUTE PLANNING

FINAL REPORT

May 1984

Prepared for:

U.S. Army Engineer Topographic Laboratory
Fort Belvoir, Virginia
Under MERADCOM Contract
DAAK70-83-P-3175

SMART SYSTEMS TECHNOLOGY

84 07 31 049

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ETL-0365	2. GOVT ACCESSION NO. AD-A143922	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) INTELLIGENT ADVISORS FOR CROSS-COUNTRY ROUTE PLANNING		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s)		8. CONTRACT OR GRANT NUMBER(s) DAAK70-83-P-3175
9. PERFORMING ORGANIZATION NAME AND ADDRESS Smart Systems Technology, Inc. Suite 300, 6870 Elm Street McLean, Virginia 22101		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS U.S. Army Engineer Topographic Laboratories Fort Belvoir, Virginia 22060		12. REPORT DATE May 1984
		13. NUMBER OF PAGES 55
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Artificial intelligence
Computer science
Expert systems
Military planning

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Several methods for computing good cross-country routes between map positions from information contained in map databases and other intelligence sources are evaluated. The problem of determining cross-country mobility from features gathered from map databases is addressed, with the goal of finding the optimal path between two positions on a map.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

INTELLIGENT ADVISORS FOR CROSS-COUNTRY ROUTE PLANNING

FINAL REPORT

Prepared for:

U.S. Army Engineer Topographic Laboratory
Fort Belvoir, Virginia
Under MERADCOM Contract
DAAK70-83-P-3175

Prepared by:

Smart Systems Technology, Inc.
Suite 300
6870 Elm Street
McLean, Virginia 22101

May 1984

A1

Preface

This is the final report on the non-training portion of MERADCOM Contract Number DAAK70-83-P-3175, entitled "Advanced Training in Logic Programming Systems, Instructional Materials, and a Computer-Based Intelligent Advisor."

Smart Systems Technology greatly appreciates the interest and support of Dr. Robert Leighty, Ms. Connie Wickham, and Mr. John Benton, at the Artificial Intelligence Center, U.S. Army Engineer Topographic Laboratory (USAETL), Fort Belvoir. Dr. Olin Mintzer, at USAETL, was very helpful during discussions about the Intelligence Preparation of the Battlefield (IPB) process.

This report concludes that there are viable methods to compute cross-country mobility and that the US Army would benefit greatly from an intelligent advisor program that assists intelligence officers during the IPB process.

Table of Contents

List of Figures

1.0 Introduction

The general problem is how to assist Army personnel, within the current military intelligence framework, to evaluate enemy objectives and plan effectively against them. An important aspect of that problem is the accurate determination of cross-country mobility for enemy and friendly vehicles from terrain and weather information. This mobility information is integral to the battle planning process that determines the potential moves and counter-moves of military units. In the present report, several methods for computing good cross-country routes between map positions from information contained in map databases and other intelligence sources will be evaluated. Different mission objectives constrain and even define the criteria for a good route, and alternative ways of integrating these route finding methods with the Army's Intelligence Preparation of the Battlefield (IPB) process will be discussed. The Appendices contain experimental code used during the project.

We have addressed the problem of determining cross-country mobility from features gathered from map databases. Our goal is to find the optimal path, with respect to some criteria, between two positions on a map. The route must be trafficable for the vehicles under consideration. The route should also satisfy certain desired criteria, for example:

1: take the least amount of time to negotiate,
2: use the least amount of resources,
3: be safe from attack from known enemy positions,
4: be safe from attack from unknown but suspected enemy positions,
5: be accessible to cover for safety from attack,
6: be immune from expected weather effects,
7: be out of sight of enemy positions,
8: be free of choke points and obvious targets, such as bridges or tunnels.

The best route may depend on the type of unit, or mix of units, traveling the route. See [1] and [2]. Any convoy of units may have significantly different characteristics than the individual units it is composed of. For example, non-machinized units may slow a convoy down; infantry that may be considered undefended on their own, may be considered safe in the company of tanks that can return enemy fire.

There is an interesting assortment of Defense Mapping Agency (DMA) map database products that can be used to generate map features. Some of these features (e.g. change in elevation, and vegetation) can be used to determine the trafficability of regions of the map. Other features can be used to construct constraint measures, such as using elevation in conjunction with known enemy locations to determine observability.

When searching for a route, one can seldom find exactly what is desired. It is an interesting problem to choose which criteria or constraint is to be relaxed when there is no perfect path. Another interesting problem, one that is very important from a human engineering standpoint, is how significantly different solution routes can be generated, and compared for tactical merit. It is necessary for a system to be able to explain how a route was selected, plus its benefits and weaknesses, so that the user may have confidence in the solutions offered by the system.

2.0 Approaches to Planning Optimal Routes

Depending on the assumptions about the terrain, there are linear and non-linear programming (dynamic programming) algorithms that mathematically guarantee [3] an optimal solution to finding the minimum cost path between two points. The terrain model is usually represented as a two-dimensional rectangular or hexagonal grid of sampled features. Each element, or point, of the grid contains the local terrain features that can be used to compute the "cost" of traversing that element of the grid. Note that the accuracy of the solution depends on the coarseness of the grid. Finer grids require more computations to arrive at their more accurate solutions. In practice, the fineness of the grid is limited by the resolution of the features in the underlying terrain databases.

To find the optimal route from grid element A to grid element B, one need only find those sequences (there could be more than one, though that seldom occurs) of connected grid elements beginning with A and ending with B that have the minimum value of total cost. The total cost is the sum of the traversal costs of the elements on the path. There can be many possible paths between A and B. Usually, the route finding algorithms are constructed so that only cycle-free routes can be generated, because if each grid element has a positive, non-zero cost, then a route with a cycle must cost more than the same route with the cycle removed, even though it goes between the same endpoints.

The simplest way to find the lowest cost path is to allow "rings" of equal total path cost spread out from point A until point B is reached. The basic algorithm is composed of the following steps (see Nilson [4]):

1) Assign point A a total cost value of zero.
2) Make point A be the only point on the "open" list.
3) Mark the "closed" list empty.
4) If the "open" list is empty, then terminate with failure.
5) Select those points from the "open" list that have the lowest total cost and place them on the "closed" list.
6) Collect all the points that are neighbors of the selected points, but do not appear on either the "open" or "closed" lists. This step is often called node (point) expansion.
7) If B is one of the collected points, then terminate with success.
8) Compute the total cost of each collected point by adding the point's traversal cost to the current lowest total cost value determined in step 5).
9) Place all the collected point on the "open" list and go back to step 4).

For example, if we are in uniform terrain where each grid element has unit traversal cost, then Figure 2-1 shows the total costs that would be computed. Note that path costs are based on 4- rather than 8-connectivity. In the example, the minimum cost path from A to B is the straight line between them. It can be found by choosing the lowest total cost point next to B, then selecting the lowest total cost point next to it, etc., until point A can be chosen.

```
     5
    545
   54345
  5432345
 543212345
54321A12345B
 543212345
  5432345
   54345
    545
     5
```

Figure 2-1: Rings of total path cost from A to B

More interesting terrain tends to distort the nice geometric patterns found in Figure 2-1 into patterns such as shown in Figure 2-2. In the right part of Figure 2-2, "+" characters indicate the points whose traversal cost are twice as high as the points denoted by " " characters, and "." characters denote points that cannot be traversed. In the left part of Figure 2-2, unprocessed points are denoted by " "s and processed points are indicated by numeric "total cost" values, except for infinite values, which are denoted by "*"s.

```
        989             +++++++++....
      98767*            +++++++++....
    9876545*            +++++++++....
  987654323*            +++++++++....
987654321A1*  9B               A ... B
 9876543212***89                 ...
  987654323456789
   9876543456789
    98765456789
     987656789
      9876789
       98789
        989
         9
```

Figure 2-2: Ring warping in non-uniform terrain

Note that the trafficability of real terrain is much more interesting than the simple example shown in Figure 2-2.

While the path finding algorithm presented above is easy to understand, most programs use a modified version that allows the "rings" of total cost to spread from both points A and B. It is necessary to keep two separate "open" and two separate "closed" lists for the two endpoints and to alternately expand points belonging to the two endpoints. The lowest cost path is found when a point collected from one endpoint is already on the other endpoint's "open" list. The modified algorithm processes approximately half as many points, but it still finds the same solution. Figure 2-3 shows the total costs that would be computed by the modified algorithm, given the identical configuration used in Figure 2-1. Visual comparison of Figures 2-1 and 2-3 confirm the approximate factor of 2 reduction in processed points. Note that two paths must be unwound (one back to A and the other back to B) and the one back to B must be reversed to produce the solution path from A to B.

```
   3
  323    2
 32123  212
321A12321B12
 32123  212
  323    2
   3
```

Figure 2-3: Rings of total path cost from A and B

One problem with the algorithms, as they have been described, is that the cost of traversing a grid element really depends on the direction of motion. For example, if you are on a hillside, then the costs of moving up, down, or across the slope are very different. It may not even be possible to move across a slope that can be climbed. Real algorithms must handle these contingencies.

2.1 Multiple constraints

So far, we have considered optimizing the path with respect to a single parameter, a traversal cost or trafficability measure for a particular type of vehicle. How can we compute optimized paths for many parameters? One must assume that the constraints, such as travel time, resource utilization, vehicle types in convoy, concealment from enemy observation, cover from enemy fire, access to cover from enemy fire, minimum channel widths, and weather effects (e.g. precipitation), can be represented by functions in a given range.

One approach would be to optimize with respect to a linear combination of the different constraints. First off, it allows the user, by choosing weights, to order the different constraints in importance. Secondly, the user can manipulate the constraint weighting factors to discover the "boundaries" (robustness) of his solution. This enables the user to determine those constraints that the selected route is very sensitive to. Lastly, it does allow the system to degrade reasonably in situations where no perfect path exists. For example, if concealment was an important factor, but no totally concealed path exists, then the algorithm will quite naturally minimize the visible parts of the path.

The problem with this approach is that when there are many constraints, it is hard to predict which constraints will be sacrificed for the overall good of the path. Also, it is unlikely that the same mix of constraint factors are appropriate for the entire cross-country maneuver. For example, concealment may be unimportant when far from the enemy, but crucial when near, or vice versa. Lastly, it is difficult to *explain how the path was* arrived at, except to say that the constraint factors were such and so, which is not very revealing. On the other hand, once the system comes up with a path, it is relatively easy to display those parts that are fast, or concealed, or not concealed, or expensive in resources, etc.

Another problem with the approach is that it is not designed to find significant alternative cross-country routes. The best route can be found. But running the system longer to find poorer solutions, or studying the route that was found, does not lead to significant alternative routes. Instead, many routes, nearly identical to the best route are returned. It's not that alternative routes are not eventually found. But many similar routes are found first and a great deal of computation can occur before the next interesting solution is reached. The problem is intrinsic with the method. Since the basic building block of routes are "steps," rather than "blocks" or "corridors," alternative solutions are distinguished by small rather than large changes. Worse yet, there seems to be no reasonable metric that determines when a path is different enough to be significant or interesting.

It should be noted that more sophisticated constraint analysis than that allowed by a linear weighting scheme is certainly needed. If you truly want a path that is both concealed and offers cover, then you cannot use a linear weighting scheme because it could allow one or both of those constraints to be violated in areas where the other factors have very low cost. Also, linear weighting has difficulty handling situations where any of the constraint functions have unusually high values. Some of the constraint functions are binary valued, while others are really cost functions. These different types of functions must be manipulated differently if routes acceptable to people are to result.

2.2 Heuristic Route Planning

The main purpose in introducing heuristics into the route finding process is to control the computational cost of the algorithms while still producing acceptable solutions. By far the simplest method (see Nilsson [4]) is to select nodes from the "open" list with the least "modified" total cost. A point's modified total cost is its total cost plus a heuristic component. Traditionally, the heuristic component is an estimate of the cost of getting from the current point to the goal point. If costs are always greater than 0 and the heuristic component never overestimates the actual costs of traversing paths, then we have an instance of an A* [4] algorithm and are guaranteed to find the optimal solution. It is more difficult to define such a heuristic function when the desired solution path is obtained by applying more than one constraint.

If fast computation time is more important than finding the best solution, as opposed to finding an adequate solution, then the influence of the heuristic component can be increased, and the algorithm will produce total cost "rings" that exhibit more directivity toward the other endpoint (the goal). This is particularly true when the heuristic function is just the distance from a point to the endpoint. See Appendix A for an example dynamic programming algorithm programmed in Lisp.

2.3 Hierarchical Route Planning

There are several reasons for considering hierarchical approaches to route planning. First of all, if it is possible to define reasonable corridors ahead of time, then it will be possible to precompute many of constraints along those corridors. This means that we can search for an optimal solution with much less computation because the search space is so much smaller. A cross-country route is composed of at most several dozen corridors, instead of hundreds or thousands of little steps. Since the corridors have s[illegible]e significance, it is possible to compute and compare significant alternative routes. It is easy to display the alternatives and allow the commander to make a choice. Ancillary information can be associated with the corridors to explain the benefits, dangers, and unique features of individual corridors. The relevant information from any corridor that is part of a cross-country route can be displayed. This goes a long way toward providing a reasonable explanation for the choice of a particular route. Best of all, as we shall find out in Chapter 5, Army intelligence already uses the notion of mobility corridors in battle planning. In some parts of the world, the corridors are already known.

3.0 Predicate Calculus Approaches to Route Planning

The main attractions of the predicate calculus based approach are, 1) the ability to explain generated paths, 2) the ability to find significant alternative paths, and 3) the ability to integrate the general route planning process with autonomous vehicle control systems. So far, predicate calculus based route planning systems seem to exist in the pure research environment of universities. A simple experimental route planning system appears as Appendix B.

By their nature, predicate calculus models tend to lend themselves to hierarchical implementations. The advantages of a hierarchical approach were discussed in Section 2.3. In particular, the ability to generate interesting alternative routes and structured explanation are possible. Since the predicates in a predicate calculus system are defined (designed) to construct acceptable routes, the proof that a route is acceptable actually describes the structure of the solution and the constraints imposed during the solution.

Sacerdoti [5] has written much about the structure of planning algorithms. SPAM [6,7] is an example of a predicate calculus-based route planning system based on DUCK [8]. Route planning involving robots are described in [9] and [10].

In SPAM, for example, the basic idea is that it takes both topological and metric information to effectively plan and execute (follow) routes across terrain. A natural implementation of a "cognitive map" could comprize an assertional database to represent topological information, and a "fuzzy map" to represent the metric information. The word fuzzy is supposed to convey the notion that much of the metric information is imprecise (e.g. to the right). The system expects to execute the route, after planning it, so that it can use its sensors to update and reduce the fuzziness of the metric information. The general route planning strategy is to determine the overall direction and topology of the route and then fill in the details by deciding how to avoid the barriers. If something unexpectedly appears (or fails to appear), when the system is executing a plan, then the system will construct and execute a new plan for the rest of the trip, after updating the database with the current situation.

The map is represented as a group of regions in a kind of fuzzy generalized cylinder [11] notation. The coarsest representation of the region is usually quite inaccurate, but is easy to work with and can give accurate enough answers in many cases. More detailed descriptions of the region are called refinements, are also generalized cylinders, and may also have further refinements. This representation scheme is sufficient to describe entrances, corridors, barriers, and enclosures; the basic vocabulary necessary to represent interesting route planning problems.

The strategy of the system, when trying to get from point A to point B, is to try and find any corridors that will help it along the way. It then plans how to get from point A to the nearest corridor by going through an entrance. The system prefers to stay in corridors for as long as practical before finding an exit and planning the trip to the next corridor. It must also plan the trip from an exit of the last corridor to point B. A corridor is a region where no obstacle completely blocks the path for objects smaller then "x." So the search for relevant corridors depend both on position, orientation, and the size of the object that is navigating the route. If the system determines that its goal is enclosed by some other region, then it plans how to find an entrance into the inner region. The system is free to consider a number of different entrances, usually choosing the shortest route. Hopefully, this conveys the flavor of the predicate calculus approach to route planning.

4.0 Graphic Overlay Approaches to Route Planning

Another approach to planning the best cross-country route involves making the various constraints manifest to a human operator. The basic assumption is that the operator's natural ability to see and reason can be exploited to plan superior routes. First, we need to represent each of the relevant constraints spatially as an image. For example, assuming that the system is told (or can lookup) the locations of enemy units and knows the elevations in the region from a map database, it could construct a binary image that indicates those positions on the map that the enemy can observe. By overlaying (intersecting) a constraint image with a trafficability image on a display, it will be possible to see the allowable paths in the region. Since most of the constraints are functions of parameters, giving the user the ability to manipulate the parameters can make the entire process be very interactive. For example, the user might vary the amount of precipitation and watch what happens to the trafficability image. Also, the effect of introducing a new constraint on the current constraint "image," could be quickly ascertained. This ability to manipulate one constraint while holding the others constant would help indicate how "brittle" or "robust" any solution route is. For example, it might be very important to know that the fastest proposed route, would become four times slower if it rains; or cannot even be considered if not being seen by the enemy is important. It is easy to display information like concealment, observability, and trafficability as images. It would require much computation to display a "total cost from point A" image, but it might be very interesting to really see the rings of uniform total cost. A significantly less expensive method would allow the user to indicate a route by a series of (as few as 2) dots. The system could compute the total cost of a feature along the route quickly. This could determine how fast a path could be negotiated, or how much resources would be consumed, etc. Note that the graphics overlay approach displays all alternative routes (if any exist) because it works in parallel and never picks a best route. It leaves that to the user.

5.0 Relation to Intelligence Preparation of the Battlefield

How can the route planning techniques discussed in Chapters 2, 3, and 4 be integrated with the US Army's Intelligence Preparation of the Battlefield (IPB) process? IPB systematically integrates weather, terrain, enemy doctrine, and the mission with the battlefield environment. The information is used to evaluate the capabilities and vulnerabilities of the enemy. IPB relies on graphic presentations to convey the intelligence information. This information leads to expectations about possible enemy activity that can be used to manage the intelligence collectors. The IPB process comprizes 5 parts:

1) Threat Assessment
2) Areas of Influence and Interest Evaluation
3) Terrain Analysis
4) Weather Analysis
5) Threat Integration

The 5 parts of IPB are performed and updated continuously so that the intelligence officer can always supply the commanding officer with the most up-to-date information available.

Threat assessment is achieved by studying the latest intelligence information, such as the composition, organization, tactical doctrine, weapons (and other equipment), and logistically supporting elements of enemy forces, to determine enemy capabilities. This indicates how the enemy would react offensively and defensively in a simplified world where terrain and weather effects are ignored. Much of this information is portrayed graphically through the use of "templates," symbols placed on maps to indicate the enemy units and the logical connections between the enemy units.

The evaluation of areas of influence and interest depend on the tactical situation and expectations about the enemy's objective. The areas are labelled and called Named Areas of Interest (NAIs).

Terrain analysis is concerned with how terrain affects the ability of friendly and enemy forces to move, shoot, and communicate. Terrain analysis considers the the five following factors:

1: Observation and Fields of Fire
2: Concealment and Cover
3: Obstacles
4: Key Terrain
5: Avenues of Approach and Mobility Corridors

Observation is concerned with the necessity of line-of-sight (LOS) for many battlefield weapons. *Fields of fire* relates to the effects of terrain on direct (LOS) and indirect fire weapons. Concealment is protection from enemy observation. Cover is protection from the firing of enemy weapons. Obstacles are natural or artificial terrain features that adversly impact the mobility of military forces. Key terrain is any tactically important area that must be seized, kept, or controlled for the success of a mission. Avenues of approach and mobility corridors are routes that friendly and/or enemy forces may use to reach key terrain or other mission objectives. Many graphic products are produced to display the trafficability and intervisibility dictated by the terrain and the positions of units.

Weather analysis uses historical weather data and weather predictions to estimate the expected effects on the ability of friendly and enemy forces to move, shoot, and communicate. Weather (in conjunction with terrain features) can radically affect cross-country mobility and tactical operations such as close air support. Again, graphic products are used to display the analysis.

Threat Integration constructs a picture of potential battles that can unfold by considering the enemy forces, their doctrine, and the expected modifications due to terrain and weather effects. This advance knowledge about what the enemy can do leads to the ability to efficiently monitor those NAIs on the battlefield that predict and confirm the intentions of the enemy.

Intelligence officers should have access to tools that compute optimum paths over terrain according to the constraints dictated by mission objectives and the tactical situation. The officers should be able to manipulate the constraints in order to understand how robust their routes really are. The system should give reasonable explanations of how a route was constructed. The system should indicate other possible routes with their advantages and dangers. The US Army already has the capability to compute cross-country mobility information, in the field, using a pocket calculator [12]. Extentions of this kind of automation should be made available to intelligence officers during the IPB process.

6.0 Conclusion

We have presented three different approaches, heuristic and dynamic programming algorithms, logic programming, and graphic display overlaying, that can be used to compute and plan cross-country routes, given cross-country mobility data. The process of computing cross-country mobility from terrain features is already well understood by USAETL (see [12]). USAETL is also familiar with all the Defence Mapping Agency (DMA) terrain and map products that can be used to automatically produce terrain features. These terrain features not only can be used to compute cross-country mobility, but can also be used to compute the constraints (such as concealment, cover, observability, etc.) needed by the route planning algorithms.

All three route planning schemes discussed in this report require further research. Heuristic and dynamic programming approaches are comparatively well understood, so that if accurate mobility and constraint features can be computed, then the construction of a route planning system is largely an engineering problem. However, the mathematics of multiple constraint satisfaction does not seem to correspond well with human problem solving intuition, and this may lead to ergonomical problems in a route planning product. The predicate calculus approach is still being actively researched in the universities and the search strategies and representations involved should be considered advanced research topics. Logic programming solutions will become feasible as newer, faster computers evolve. Graphic display overlaying is only a concept; no experiments have been undertaken to verify its applicability. We consider it a very promising technique because it marries the display capabilities of computer equipment, applied to digital map databases, with the visual image processing and cognitive abilities of intelligence officers. Also, USAETL has all the necessary facilities (personnel, hardware, and access to map data) to research and develop the idea futher.

It will be necessary to integrate the possible solutions to the cross-country route planning process with the IPB process as it is practiced. This will require an expert who is quite familiar with the goals and methods of IPB.

In a more general spirit, what kind of an intelligent tool could improve the efficiency and accuracy of the reports generated by intelligence officers during the IPB process? Several areas where automation would be beneficial are:

- allowing intelligence officers access to tools that compute optimum paths over terrain with respect to supplied constraints. See the end of Chapter 5.

- constructing a planning/bookkeeping system that assists the intelligence officer in keeping track of the potential plans and the NAIs that they depend on, so that unlikely plans *can be dropped, and confirmed* plans can attain prominence. It would be natural for the system to generate many of the IPB tables, etc.

- producing the hardcopy graphic products of the IPB process, maps, overlays, etc., by accessing map and other databases.

The first problem area has been the focus of this report. The second topic is a very hard and very interesting general problem in the creation and maintenance of plans. The third area is largely an engineering problem, but it does depend on what equipment the Army is willing to allocate to IPB stations. Substantial inputs from IPB experts, with detailed knowledge of IPB procedures, would be necessary to develop any of these systems. However, constructing intelligent advisor programs in these areas would save manpower, decrease threat assessment turnaround time, and most importantly, improve the quality and potentially the safety of military plans.

7.0 References

1. Lamas System Manual, CDRL Item A00B, TRW, Defense and Space Systems Group, 1 Space Park, Redondo Beach, California, 90278, February, 1978.

2. Location and Movement Analysis System (LAMAS), CDRL Item A008, TRW, Defense and Space Systems Group, 1 Space Park, Redondo Beach, California, 90278, June, 1978.

3. Luenberger, David, Introduction to Linear and Nonlinear Programming, Addison-Wesley, 1973.

4. Nilsson, Nils, Problem-solving Methods in Artificial Intelligence, McGraw-Hill, New York, 1971.

5. Sacerdoti, Earl, A Structure for Plans and Behavior, American Elsevier, New York, 1977.

6. McDermott, Drew, and Davis, Ernest, Planning Routes through Uncertain Territory, Department of Computer Science, Yale University, New Haven, Connecticut. To appear in AI Journal.

7. Davis, Ernest, Organizing Spatial Knowledge, Technical Report 193, Department of Computer Science, Yale University, New Haven, Connecticut, 1981.

8. McDermott, Drew, DUCK: A Lisp-Based Deductive System, Department of Computer Science, Yale University, May 1983, available from Smart Systems Technology, 6870 Elm Street, McLean, Virginia, 22101.

9. Thompson, Alan, The Navigation system of the JPL Robot, IJCAI5, pages 749-758, 1977.

10. Rosenberg, Richand, and Rowat, Peter, Spatial Problems for a Simulated Robot, IJCAI 7, 1981.

11. Binford, Thomas, Visual perception by computer, IEEE Conf. on Systems and Control, IEEE, 1971.

12. Pearson, Alexander, and Wright, Janet, Synthesis Guide for Cross-Country Movement, ETL-0220, U.S. Army Engineer Topographic Laboratories, Fort Belvoir, Virginia, 22060, February, 1980.

8.0 Appendix A: Experimental Dynamic Programming Code

Experimental dynamic programming functions were developed in Franz Lisp on a VAX 11/780 computer. The four files dp.l, ccm.l, dplot.l, dptst.l appear in this appendix.

In order to use this software, you should move to directory [sstsys.hayes.ipb] and type "lisp" (after the command prompt) and then type "(dptst)".

To the query for path coordinates, respond with the list "((10 10) (20 20))" or "((20 20) (30 30))".

To the query for a value for TST-F*, respond with the number "1.0" or a value from the set (.25 .33 .5 .66 .75). This factor controls the degree of optimal (1.0) verses direct (0.1) path desired.

To the query for a value for the maximum iteration count, respond with a number (50 is a good choice).

To the query for a value for CONT, respond with "t" if you want to ignore the maximum iteration count and continue, or "nil", otherwise. The usual response is "t".

To the query about GCs, respond with "t" if you want to be informed about garbage collections, or "nil", otherwise.

To the query about debug output, respond with "0", if you do not want any, "1", if you want to see the arguments passed to the dynamic programming algorithm, "2", if you want to see the coordinates of the points being processed, or "3", if you want to see the state of the "open" and "closed" queues during processing. If you select "2", then you automatically receive the "1" output too, etc.

```
;
;   dp.l    Dynamic Programming Code
;
(eval-when (compile)
           (load "sst$lib:util")
           (load "sst$lib:ws")
           )
;
(declare (macros t)
         (special poport dp-dbg* xdisp* ydisp*)
         (localf dp-insert-entry-1 dp-copyl)
         )
;
!; dpa dpa WORKSPACE for Dynamic Programming Code in file DP.L
;
;     MAIN Routines:
; DPL    - Finds a route through a sequence of points
; DP     - Finds a route between two points
; DPI    - Internal route finder
; DPC    - Continue a previously failed search
!;
;
(workspace-push 'dpa)
;
;  Abstract Data Type : XYCOORD
;
;(defdt XYCOORD
;       (names xcoord . ycoord)
;       (types NUMBER   NUMBER)
;       (ident coordp)
;       )
;
;
!; coordp (coordp exp)
;
;     COORDP returns T if EXP is of type XYCOORD (a coordinate),
; else NIL.
!;
(de coordp (c)
  (and (consp c)
       (numberp (car c))
       (numberp (cdr c))
       )
  )
;
!; xcoord (xcoord coord)
;
;     XCOORD returns the X coordinate associated with the
; coordinate COORD.
!;
(de xcoord macro (exp)
  (maclobber exp `(car ,(cadr exp)))
  )
;
```

```
;
!; ycoord (ycoord coord)
;
;     YCOORD returns the Y coordinate associated with the
; coordinate COORD.
!;
(de ycoord macro (exp)
  (maclobber exp `(cdr ,(cadr exp)))
  )
;
;  Abstract Data Type : DP_ENTRY_NODE
;
;(defdt DP_ENTRY_NODE
;       (names dp-cost dp-coord dp-node)
;       (types NUMBER  XYCOORD  DP_NODE)
;       )
;
(de dp-cost macro (exp)
  (maclobber exp `(car ,(cadr exp)))
  )
;
(de dp-coord macro (exp)
  (maclobber exp `(cadr ,(cadr exp)))
  )
;
(de dp-node macro (exp)
  (maclobber exp `(caddr ,(cadr exp)))
  )
;
!; dp-dbg* dp-dbg* (symbol) DP's debug print flag
;
;     If DP-DBG* is Ø, then DP prints no debug information. If
; DP-DBG* is 1, then the arguments to DP are printed out. If
; DP-DBG* is 2, then the coordinates of nodes that are closed
; are listed. If DP-DBG* is 3, then queue information is also
; listed.
!;
(defv dp-dbg* Ø)
;
!; dpl (dpl open-fct weight-fct back-fct prt-fct coord-list max cont)
;
;     DPL applies DP to find a route between each adjacent pair
; of coordinates in COORD-LIST. OPEN-FCT, WEIGHT-FCT, BACK-FCT,
; MAX, and PRT-FCT are described in more detail in DP. If CONT is
; NIL and DP fails to return a route, then DPL returns NIL. If
; CONT is T, then DPC is called until a path is found. If CONT is
; anything else, then the user is asked whether to continue or
; not. If the response is "no", then DPL returns NIL, otherwise,
; DPL tries again to find a sub-path.
!;
```

```
(de dpl (of wf bf pf cl max cont)
  (do [(al nil)
       (dpans nil)]
      [(null (cdr cl)) al]
      (:= dpans (dp of wf bf (car cl) (cadr cl) max pf))
      (cond [(car dpans) t]
            [(null cont) (return nil)]
            [(eq cont t)
               (:= dpans
                   (do [(dpa (dpc nil dpans) (dpc nil dpa))]
                       [(car dpa) dpa]
                       ))]
            [(:= dpans
                 (do [(dpa dpans)]
                     [(car dpa) dpa]
                     (ttymsg t "Failed to find route from "
                               (car cl) " to " (cadr cl) "."
                             t "Try again? ")
                     (cond [(is-yes (read))
                              (:= dpa (dpc nil dpa))]
                           [t (return nil)])
                     ))
               t]
            [t (return nil)]
            )
      (cond [al (:= al (nconc al (cdr (car dpans))))]
            [t (:= al (car dpans))])
      (:= cl (cdr cl))
      )
  )
;
!; dp (dp open-fct weight-fct back-fct
;          start-coord end-coord max prt-fct)
;
;     DP applies a dynamic programming algorithm to find a route
; (path) between the coordinates START-COORD and END-COORD that
; requires the consideration of less than 2*MAX intermediate
; points. OPEN-FCT is the name of a function that creates a new
; NODE. OPEN-FCT is called with the arguments COORD B-COORD
; F-COORD ENTRY. COORD is the coordinate of the point that is to
; be opened. B-COORD is the coordinate of the starting point of
; the path to COORD. F-COORD is the coordinate of the goal for
; the path. ENTRY is a DP queue-entry to the point on the path
; that precedes COORD. (DP-NODE ENTRY) will return the NODE
; associated with ENTRY. WEIGHT-FCT is the name of a function
; that returns the "weight" or "cost" associated with a node.
; WEIGHT-FCT is called with the argument NODE, a node created by
; OPEN-FCT. BACK-FCT is the name of a function that returns a
; pointer to the the entry that precedes a node. BACK-FCT is
;calledwiththeargument NODE. PRT-FCT is thenameofa
; function that prints out a node. PRT-FCT is called with the
; argument NODE. PRT-FCT is necessary since a node may contain
; circular objects or pointers to deeply nested structures. DP
; returns an object of type DP_ANS.
!;
```

```
(de dp (of wf bf cl c2 n pf)
  (cond [(plusp dp-dbg*)
             (ttymsg t "DP: of= " of " wf= " wf " pf= " pf
                       " cl= " cl " c2= " c2 " max= " n)])
  (dpi of                                                 ;open-fct
       wf                                                 ;weight-fct
       bf                                                 ;back-fct
       pf                                                 ;print-fct
       cl                                                 ;start-coord
       c2                                                 ;end-coord
       n                                                  ;max
       Ø                                                  ;count
       (list (dp-make-entry of wf cl cl c2 nil))           ;sc-olst
       (list (dp-make-entry of wf c2 c2 cl nil))           ;ec-olst
       nil                                                 ;sc-clst
       nil                                                 ;ec-clst
       )
  )
;
!; dpc (dpc max dp-answer)
;
;     DPC takes the DP_ANSWER DP-ANSWER, the result of a
; previously unsuccessful call to DP, and applies the DP
; algorithm again for another 2*MAX steps. If MAX is NIL or less
; than 1, then the original value of MAX (from DP-ANSWER) is
; used. If DP-ANSWER indicates that a route has already been
; found, then DPC just returns DP-ANSWER.
!;
(de dpc (max dpans)
  (cond [(car dpans) dpans]
        [(and (numberp max) (plusp max))
           (:= dpans (dp-copyl (cdr dpans)))
           (rplaca (cddddddr dpans) max)
           (apply (function dpi) dpans)]
        [t (apply (function dpi) (cdr dpans))])
  )
;
!; dpi (dpi of wf bf pf cl c2 n c sco eco scc ecc)
;
;     DPI is the internal workhorse routine for DP. It accepts
; the necessary functions, coordinates, numbers, and queues and
; presses forward with the DP algorithm.
!;
;
```

```
(de dpi (of wf bf pf cl c2 max cnt
         sc-olst ec-olst sc-clst ec-clst)
  (do [(n max (subl n))
       (exp-node nil)
       (op-node nil)
       (path nil)
       (stop nil)]
      [(minusp n) (list nil of wf bf pf cl c2 max cnt
                        sc-olst ec-olst sc-clst ec-clst)]
      (cond [(> dp-dbg* 2)
             (ttymsg t "DP: n = " n " cnt= " cnt " stop =" stop
                     t "s-o = " (e (dp-print pf sc-olst))
                     t "s-c = " (e (dp-print pf sc-clst))
                     t "e-o = " (e (dp-print pf ec-olst))
                     t "e-c = " (e (dp-print pf ec-clst)))])
      (:= cnt (addl cnt))
      (cond [sc-olst
             (:= exp-node (car sc-olst))
             (cond [(> dp-dbg* 1)
                    (ttymsg t "Closing: "
                              (dp-coord exp-node))])
             (:= sc-olst (cdr sc-olst))
             (cond [(:= op-node
                      (dp-adjacentl (dp-coord exp-node)
                                    ec-olst))
                    (:= op-node (dp-least op-node))
                    (:= path (nconc (nreverse
                                     (dp-path exp-node bf))
                                    (dp-path op-node bf)))
                    (return (list path of wf bf pf cl c2
                                  max cnt sc-olst ec-olst
                                  sc-clst ec-clst))])
             (for [c in (dp-step (dp-coord exp-node)
                                 sc-olst
                                 sc-clst)]
                  (do (:= sc-olst
                        (dp-insert-entry
                          (dp-make-entry of wf c cl c2
                                         exp-node)
                          sc-olst))
                    )
                  )
             (:= sc-clst (cons exp-node sc-clst))
             ]
            [t (:= stop t)])
      (cond [ec-olst
             (:= exp-node (car ec-olst))
             (cond [(> dp-dbg* 1)
                    (ttymsg t "Closing: "
                              (dp-coord exp-node))])
             (:= ec-olst (cdr ec-olst))
```

```
                (cond [(:= op-node
                           (dp-adjacentl (dp-coord exp-node)
                                         sc-olst))
                      (:= op-node (dp-least op-node))
                      (:= path (nconc (nreverse
                                        (dp-path op-node bf))
                                      (dp-path exp-node bf)))
                      (return (list path of wf bf pf cl c2
                                    max cnt sc-olst ec-olst
                                    sc-clst ec-clst))])
                (for [c in (dp-step (dp-coord exp-node)
                                    ec-olst
                                    ec-clst)]
                    (do (:= ec-olst
                              (dp-insert-entry
                                (dp-make-entry
                                   of wf c c2 cl exp-node)
                                ec-olst))
                      )
                    )
                (:= ec-clst (cons exp-node ec-clst))
                ]
              [stop (return (list path of wf bf pf cl c2
                                  max cnt sc-olst ec-olst
                                  sc-clst ec-clst))])
      )
   )
;
!; dp-make-entry (dp-make-entry open-f w-f coord
;                               start-c end-c pentry)
;
;     DP-MAKE-ENTRY constructs a DP entry with form (path-weight
; COORD node). The function OPEN-F is called to create NODE.
; PATH-WEIGHT is calculated by adding the weights at PENTRY and
; NODE (using function W-F).
!;
(de dp-make-entry (of wf c sc ec entry)
  (:= of (funcall of c sc ec entry))
  `(,(plus (funcall wf (dp-node entry)) (funcall wf of)) ,c ,of)
  )
;
!; dp-adjacentl (dp-adjacentl coord entry-list)
;
;     DP-ADJACENTL returns a list of entries from ENTRY-LIST that
; are adjacent to COORD. The function DP-ADJACENT is used to
; determine adjacency.
!;
(de dp-adjacentl (c el)
  (cond [(null el) nil]
        [(dp-adjacent c (dp-coord (car el)))
           (cons (car el) (dp-adjacentl c (cdr el)))]
        [t (dp-adjacentl c (cdr el))])
  )
```

```
;
!; dp-adjacent (dp-adjacent coord1 coord2)
;
;     DP-ADJACENT returns T if the coordinates COORD1 and COORD2
; are 4-connected, else NIL.
!;
(de dp-adjacent (c1 c2)
  (cond [(equal (xcoord c1) (xcoord c2))
            (equal 1 (abs (difference (ycoord c1) (ycoord c2))))]
        [(equal (ycoord c1) (ycoord c2))
            (equal 1 (abs (difference (xcoord c1) (xcoord c2))))])
  )
;
!; dp-least (dp-least entry-list)
;
;     DP-LEAST returns the entry on ENTRY-LIST that has the least
; path-cost.
!;
(de dp-least (el) (dp-least-1 (car el) (cdr el)))
;
(de dp-least-1 (e el)
  (cond [(null el) e]
        [(greaterp (dp-cost e) (dp-cost (car el)))
            (dp-least-1 (car el) (cdr el))]
        [t (dp-least-1 e (cdr el))])
  )
;
!; dp-coords (dp-coords entry back-function)
;
;     DP-COORDS returns a list of the coordinates from entry
; ENTRY back to its starting point. The function BACK-FUNCTION is
; used to trace back through the entries.
!;
(de dp-coords (e bf)
  (cond [(null e) nil]
        [t (cons (dp-coord e)
                 (dp-coords (funcall bf (dp-node e)) bf))])
  )
;
!; dp-path (dp-path entry back-function)
;
;     DP-PATH returns a list of the entries from entry ENTRY back
; to its starting point. The function BACK-FUNCTION is used to
; trace back through the entries.
!;
(de dp-path (e bf)
  (cond [(null e) nil]
        [t (cons e (dp-path (funcall bf (dp-node e)) bf))])
  )
```

```
;
!; dp-step (dp-step coord open-node-list closed-node-list)
;
;     DP-STEP returns a list of coordinates that are adjacent to
; the coordinate COORD but not already associated with a node on
; either of the lists OPEN-NODE-LIST or CLOSED-NODE-LIST.
!;
(de dp-step (c onl cnl)
  (for [d in '(0  1 2 3)]
       [when (not (or (dp-there (dp-chain c d) onl)
                      (dp-there (dp-chain c d) cnl)))]
       (save (dp-chain c d))
       )
  )
;
!; dp-chain (dp-chain coord dir)
;
;     DP-CHAIN returns the coordinates of the point that is in
; direction DIR from COORD. DIR must be one of the numbers
; {0 1 2 3} corresponding to {up right down left}.
!;
(de dp-chain (c d)
  `(,(plus (xcoord c) (cxr d xdisp*))
    .,(plus (ycoord c) (cxr d ydisp*)))
  )
;
!; xdisp* xdisp*(4-hunk)
;
;  xdisp* is a 4-hunk with X displacements for 4-connected chain
;         codes.
!;
(defv xdisp* (hunk 0 1 0 -1))
;
!; ydisp* ydisp*(4-hunk)
;
;  ydisp* is a 4-hunk with Y displacements for 4-connected chain
;         codes.
!;
(defv ydisp* (hunk 1 0 -1 0))
;
!; dp-there (dp-there coord entry-list)
;
;     DP-THERE returns the entry in ENTRY-LIST with coordinates
; COORD, or NIL if there is none.
!;
(de dp-there (c el)
  (cond [(null el) nil]
        [(equal c (dp-coord (car el))) (car el)]
        [t (dp-there c (cdr el))])
  )
```

```
;
!; dp-insert-entry (dp-insert-entry entry entry-list)
;
;     DP-INSERT-ENTRY inserts the dp-entry ENTRY into the list of
; entries ENTRY-LIST according to its path-weight.
!;
(de dp-insert-entry (e el)
  (cond [(null el) (list e)]
        [t (dp-insert-entry-1 e el)
           el])
  )
;
(de dp-insert-entry-1 (e el)
  (cond [(lessp (dp-cost e) (dp-cost (car el)))
           (rplacd el (cons (car el) (cdr el)))
           (rplaca el e)]
        [(null (cdr el)) (rplacd el (cons e nil))]
        [t (dp-insert-entry-1 e (cdr el))])
  )
;
!; dp-print (dp-print print-fct entry-list)
;
;     DP-PRINT uses DP-PRINT-ENTRY to print out each entry in
; ENTRY-LIST. DP-PRINT-ENTRY uses PRINT-FCT to printout the NODE
; in the entry.
!;
(de dp-print (pf el)
  (for [e in el]
       (do (DP-PRINT-ENTRY pf e))
       )
  )
;
!; dp-print-entry (dp-print-entry print-fct entry)
;
;     DP-PRINT-ENTRY prints out the "cost" and "coordinates" of
; the entry ENTRY and uses PRINT-FCT to print out the "node"
; associated with ENTRY.
!;
(de dp-print-entry (pf e)
  (ttymsg t (dp-cost e) " " (dp-coord e) " "
          (e (funcall pf (dp-node e))))
  )
;
;  (dp-copyl l)  returns a top-level copy of the list L.
;
(de dp-copyl (l)
  (cond [(null l) nil]
        [t (cons (car l) (dp-copyl (cdr l)))])
  )
;
(remprop 'dpa 'wsviolated)
(princ "DP Loaded")
(terpr)
```

```
;
;   ccm.l    Cross Country Mobility Code
;
(eval-when (compile)
           (load "sst$lib:util")
           (load "sst$lib:ws")
           (load "dp")
           )
(declare (macros t)
         (special ccm-wv* ccm-nodes*)
         (localf )
        )
(workspace-push 'ccm)
;
!; pathfinder (pathfinder)
;
;     PATHFINDER is the user-interface to the dynamic-programming
; algorithm in DP.L.
!;
(de pathfinder ()
  (prog [map cl max cont ccm-wv* path]
        (:= ccm-wv* (hunk .125 .125 .125 .125
                          .125 .125 .125 .125))
        (:= cont 'read)
        (ttymsg t "What Map are we using? ")
        (:= map (read))
        (ccm-init map)
   newc (ttymsg t "What are the coordinates of the points that"
                t "our path must include?")
        (ttymsg t " Starting  at: " (car cl)
                t " Passing thru: "
                  (nreverse (cdr (reverse (cdr cl))))
                t " Ending    at: " (car (last cl))
                t " is this ok? ")
        (cond [(not (is-yes (read))) (go newc)])
   neww (ttymsg t "The current weight vector is :"
                t ccm-wv*
                t "Is it ok? ")
        (cond [(not (is-yes (read)))
                 (ttymsg "type in an index (Ø to 7) and a weight"
                         t)
                 (rplacx (read) ccm-wv* (read))
                 (go neww)])
   nmax (ttymsg t "Please input an INTEGER value for MAX: ")
        (cond [(not (fixp (:= max (read)))) (go nmax)])
        (:= path (dpl 'ccm-open-f
                      'ccm-weight-f
                      'ccm-back-f
                      'ccm-prt-f
                      cl
                      max
                      cont))
        (return path)
        )
  )
```

```
;
!; ccm-init (ccm-init name)
;
;     CCM-INIT reinitializes the system if we switch maps or
; begin, but otherwise allows previous nodes to be remembered
; from search to search.
!;
(de ccm-init (name)
  (cond [(eq name (get 'ccm-init 'ccm-init)) t]
        [t (:= ccm-nodes* nil)
           (:= (get 'ccm-init 'ccm-init) name)])
  )
;
!; make-coord (make-coord potential-coord)
;
;     MAKE-COORD returns the coord associated with
; POTENTIAL-COORD. If POTENTIAL-COORD is (x . y), (x y), or
; (x y ...) then (x . y) is returned. Otherwise, NIL is returned.
; The "x" and "y" must be integers.
!;
(de make-coord (l)
  (cond [(and (consp l) (fixp (car l)) (fixp (cdr l))) l]
        [(and (consp l) (fixp (car l))
              (consp (cdr l)) (fixp (cadr l)) (null (cddr l)))
           (rplacd l (cadr l))])
  )
;
!; ccm-ev-f (ccm-ev-f node)
;
;     CCM-EV-F evaluates the cost of passing through the node
; NODE with respect to a global weight vector named CCM-WV*.
!;
(de ccm-ev-f (node)
  (do [(i 6 (subl i))
       (w (times (cxr 7 node) (cxr 7 ccm-wv*))
          (plus w (times (cxr i node) (cxr i ccm-wv*))))]
      [(equal 2 i) w]
      )
  )
;
!; ccm-open-f (ccm-open-f coord start-coord end-coord
;                          previous-entry)
;
;     CCM-OPEN-F opens a node at coordinate COORD. A NODE is an
; 8 position HUNK where index i is:
;
; 0  Overall weight (cost) associated with node
; 1  Pointer back to PREVIOUS-ENTRY
; 2  Coordinate of this Point
; 3  Distance from COORD to END-COORD
; 4
; 5
; 6
; 7
!;
```

```
(de ccm-open-f (c sc ec node)
  (let [(h (ccm-find-node c ccm-nodes*))]
       (cond [h (rplacx 1 h node)
                (rplacx 3 h (straight-dist c ec))
                (rplacx Ø h (ccm-ev-f h))]
             [t (:= h (makhunk 8))
                (rplacx 1 h node)
                (rplacx 2 h c)
                (rplacx 3 h (straight-dist c ec))
                (rplacx 4 h 1)
                (rplacx 5 h 1)
                (rplacx 6 h 1)
                (rplacx 7 h 1)
                (rplacx Ø h (ccm-ev-f h))
                (:= ccm-nodes* (cons h ccm-nodes*))])
       h
       )
  )
;
(defv ccm-nodes* nil)
;
(de ccm-find-node (c nl)
  (cond [(null nl) nil]
        [(equal c (cxr 2 (car nl))) (car nl)]
        [t (ccm-find-node c (cdr nl))])
  )
;
!; ccm-weight-f (ccm-weight-f node)
;
;     CCM-WEIGHT-F returns the cost or weight associated with
; node NODE.
!;
(de ccm-weight-f (node)
  (cond [(hunkp node) (cxr Ø node)]
        [t Ø])
  )
;
!; ccm-back-f (ccm-back-f node)
;
;     CCM-BACK-F
!;
(de ccm-back-f (node)
  (cond [(hunkp node) (cxr 1 node)])
  )
;
```

```
;
!; ccm-prt-f (ccm-prt-f node)
;
;     CCM-PRT-F prints out the node NODE.
!;
(de ccm-prt-f (node)
  (ttymsg t "{" (cxr 0 node) ",("
              (car (cxr 2 node)) ":" (cdr (cxr 2 node)) "),"
              (cxr 3 node) ","
              (cxr 4 node) ","
              (cxr 5 node) ","
              (cxr 6 node) ","
              (cxr 7 node) "}")
  )
;
!; nexto (nexto coord1 coord2)
;
;     NEXTO is a predicate that returns T if coordinates COORD1
; and COORD2 are next to each other (8 connected), else NIL.
!;
(de nexto (c1 c2)
  (and (lessp (abs (difference (xcoord c1) (xcoord c2))) 2)
       (lessp (abs (difference (ycoord c1) (ycoord c2))) 2))
  )
;
!; straight-dist (straight-dist coord1 coord2)
;
;     STRAIGHT-DIST returns the "straight" distance between the
; coordinates COORD1 and COORD2. A coordinate has the form (x . y)
; or (z x . y).
!;
(de straight-dist (c1 c2)
  (sqrt (plus (expt (difference (xcoord c1) (xcoord c2)) 2)
              (expt (difference (ycoord c1) (ycoord c2)) 2)
              (expt (difference (zcoord c1) (zcoord c2)) 2)
              ))
  )
;
(remprop 'ccm 'wsviolated)
(princ "CCM Loaded")
(terpr)
```

```
;
;   dplot.l  Plotting Routines
;
(eval-when (compile)
           (load "sst$lib:util")
           (load "sst$lib:ws")
           (load "dp")
           )
;
(declare (macros t)
         (special poport $gcprint dplot-p* dplot-pl* dplot-ip*)
         (localf plotstrip plotyhd plotln
                 dplot-f-y-pnts dplot-prt-prep
                 init-next-p next-p get-p
                 dplot-min-x dplot-max-x dplot-min-y dplot-max-y
                 dplot-minx dplot-maxx dplot-miny dplot-maxy)
         )
;
!; dplot dplot WORKSPACE for Plotting Routines in file DPLOT.L
;
!;
;
(workspace-push 'dplot)
;
!; prtpts (prtpts coord-list)
;
;     PRTPTS
!;
(de prtpts (cl)
  (let [($gcprint nil)]
       (ttymsg "# of points= " (length cl) t)
       (do [(c '||)
             (nc 80 (1- (- nc (flatc c))))]
           [(null cl) (terpr)]
           (:= c (list (caar cl) (cdar cl)))
           (cond [(< (flatc c) nc)]
                 [t (terpr)
                    (:= nc 80)]])
           (princ c)
           (princ " ")
           (:= cl (cdr cl))
           )
       )
  )
;
!; plotpts (plotpts coord-list)
;
;     PLOTPTS
!;
(de plotpts (cl) (plotpts2 cl nil) )
;
```

```
;
!; plotpts2 (plotpts2 coord-list1 coord-list2)
;
;     PLOTPTS2
!;
(de plotpts2 (cl ocl)
  (let [(y-max (dplot-max-y cl))
        (y-min (dplot-min-y cl))
        (x-min (dplot-min-x cl))
        (x-max (dplot-max-x cl))
        ($gcprint nil)]
       (ttymsg "Min-X= " x-min (t 11) " Max-Y= " y-max
               (t 25) "# of Coords= " (length cl) t
               "Min-Y= " y-min (t 11) " Max-X= " x-max
               (t 25) "# of Strips= "
                      (/ (+ (- x-max x-min) 79) 79)
               t)
       (do [(min-x x-min (+ min-x 79))
            (i 1 (1+ i))]
           [(> min-x x-max) nil]
           (plotstrip min-x (min (+ min-x 78) x-max)
                      y-min y-max i cl ocl)
           )
       )
  )
;
(de plotstrip (x-min x-max y-min y-max sn cl ocl)
  (ttymsg "Strip # " sn "  Min-X= " x-min "  Max-X= " x-max t)
  (plotyhd x-min x-max)
  (do [(i y-max (1- i))]
      [(< i y-min) nil]
      (cond [(zerop (mod i 10))
                (princ (mod (/ i 10) 10))]
            [(zerop (mod i 5))
                (princ "-")]
            [t (princ " ")])
      (plotln i x-min x-max cl ocl)
      )
  (plotyhd x-min x-max)
  )
;
(de plotyhd (min max)
  (princ "+")
  (do [(i min (1+ i))]
      [(> i max) (terpr)]
      (cond [(zerop (mod i 10))
                (princ (mod (/ i 10) 10))]
            [(zerop (mod i 5))
                (princ "!")]
            [t (princ " ")])
      )
  )
```

```
;
(de plotln (y x-min x-max cl ocl)
  (let [(pts (dplot-f-y-pnts y x-min x-max 0 cl))
        (opts (dplot-f-y-dot-pnts y x-min x-max ocl))]
       (:= pts (sortcar pts (function <)))
       (:= opts (sort opts (function <)))
       (:= pts (dplot-prt-prep pts))
       (:= opts (dplot-prt-dot-prep opts pts))
       (do [(i x-min (1+ i))]
           [(and (null pts) (null opts)) (terpr)]
           (cond [(equal i (caar pts))
                     (princ (cdar pts))
                     (:= pts (cdr pts))]
                 [(equal i (car opts))
                     (princ ".")
                     (:= opts (cdr opts))]
                 [t (princ " ")])
           )
       )
  )
;
(de dplot-f-y-pnts (y x xt n cl)
  (cond [(null cl) nil]
        [(and (equal (ycoord (car cl)) y)
              (not (> x (xcoord (car cl))))
              (not (< xt (xcoord (car cl))))
              )
           (cons (cons (xcoord (car cl)) n)
                 (dplot-f-y-pnts y x xt (1+ n) (cdr cl)))]
        [t (dplot-f-y-pnts y x xt (1+ n) (cdr cl))])
  )
;
(de dplot-f-y-dot-pnts (y x xt ocl)
  (cond [(null ocl) nil]
        [(and (equal (ycoord (car ocl)) y)
              (not (> x (xcoord (car ocl))))
              (not (< xt (xcoord (car ocl))))
              )
           (cons (xcoord (car cl))
                 (dplot-f-y-dot-pnts y x xt (cdr ocl)))]
        [t (dplot-f-y-dot-pnts y x xt (cdr ocl))])
  )
;
(de dplot-prt-prep (pts)
  (cond [(null pts) nil]
        [(and (cdr pts)
              (eq (caar pts) (caadr pts)))
           (dplot-prt-prep `((,(caar pts)) .,(cddr pts)))]
        [(cdar pts) (cons `(,(caar pts) .,(get-p (cdar pts)))
                          (dplot-prt-prep (cdr pts)))]
        [t (cons `(,(caar pts) . "*")
                 (dplot-prt-prep (cdr pts)))])
  )
;
```

```
;
(de dplot-prt-dot-prep (opnxs pts)
  (cond [(null opnxs) nil]
        [(and (cdr opnxs)
              (eq (car opnxs) (cadr opnxs)))
           (dplot-prt-dot-prep `(,(car opnxs) .,(cddr opnxs))
                               pts)]
        [(not (assoc (car opnxs) pts))
           (dplot-prt-dot-prep (cdr opnxs) pts)]
        [t (cons (car opnxs)
                 (dplot-prt-dot-prep (cdr opnxs) pts))])
  )
;
(defv dplot-p*
      '(Ø 1 2 3 4 5 6 7 8 9
        a b c d e f g h i j k l m n o p q r s t u v w x y z
        |A| |B| |C| |D| |E| |F| |G| |H| |I| |J| |K| |L| |M|
        |N| |O| |P| |Q| |R| |S| |T| |U| |V| |W| |X| |Y| |Z|)
      )
;
(defv dplot-pl* (length dplot-p*))
;
(defv dplot-ip* nil)
;
(de init-next-p ()
  (:= dplot-ip* nil)
  )
;
(de next-p ()
  (:= dplot-ip* (cdr dplot-ip*))
  (cond [(null dplot-ip*) (:= dplot-ip* dplot-p*)])
  (car dplot-ip*)
  )
;
(de get-p (n)
  (:= n (mod n dplot-pl*))
  (nth n dplot-p*)
  )
;
(de dplot-min-x (cl)
  (cond [(null cl) Ø]
        [t (dplot-minx (xcoord (car cl)) (cdr cl))])
  )
;
(de dplot-minx (v cl)
  (cond [(null cl) v]
        [(< (xcoord (car cl)) v)
           (dplot-minx (xcoord (car cl)) (cdr cl))]
        [t (dplot-minx v (cdr cl))])
  )
;
(de dplot-max-x (cl)
  (cond [(null cl) Ø]
        [t (dplot-maxx (xcoord (car cl)) (cdr cl))])
  )
```

```
;
(de dplot-maxx (v cl)
  (cond [(null cl) v]
        [(> (xcoord (car cl)) v)
           (dplot-maxx (xcoord (car cl)) (cdr cl))]
        [t (dplot-maxx v (cdr cl))])
  )
;
(de dplot-min-y (cl)
  (cond [(null cl) 0]
        [t (dplot-miny (ycoord (car cl)) (cdr cl))])
  )
;
(de dplot-miny (v cl)
  (cond [(null cl) v]
        [(< (ycoord (car cl)) v)
           (dplot-miny (ycoord (car cl)) (cdr cl))]
        [t (dplot-miny v (cdr cl))])
  )
;
(de dplot-max-y (cl)
  (cond [(null cl) 0]
        [t (dplot-maxy (ycoord (car cl)) (cdr cl))])
  )
;
(de dplot-maxy (v cl)
  (cond [(null cl) v]
        [(> (ycoord (car cl)) v)
           (dplot-maxy (ycoord (car cl)) (cdr cl))]
        [t (dplot-maxy v (cdr cl))])
  )
;
(remprop 'dplot 'wsviolated)
(princ "DPLOT Loaded")
(terpr)
```

```
;
;   dptst.l    DP test case Code
;
(eval-when (compile)
           (load "sst$lib:util")
           (load "sst$lib:ws")
           (load "dp")
           )
;
(declare (macros t)
         (special dp-dbg* tst-max* tst-f* poport $gcprint)
         (localf )
         )
;
(workspace-push 'dp-test)
;
!; dptst (dptst)
;
!;
(de dptst ()
  (prog [cl cont dpla path opns n opn-dmp ipt dp-dbg* $gcprint]
    top (dptst-init)
        (ttymsg t "What are the coordinates of the points that"
                t "our path should pass through? ")
        (cond [(:= ipt (read)) (:= cl ipt)])
        (:= cl (for [x in cl]
                    (save (make-coord x))
                    ))
        (ttymsg t "Value for TST-F* (" tst-f* ") factor?")
        (cond [(numberp (:= ipt (read))) (:= tst-f* ipt)])
        (ttymsg t "What is the maximum iteration count? ")
        (:= n (read))
        (ttymsg t "Value for CONT? ")
        (:= cont (read))
        (ttymsg t "Do you want to know about opened nodes? ")
        (:= opn-dmp (is-yes (read)))
        (ttymsg t "Do you want to know about GCs? ")
        (:= $gcprint (is-yes (read)))
        (ttymsg t "Debug output? (0=none, ... 3=full) ")
        (:= dp-dbg* (read))
        (ttymsg t " Starting  point  is: " (car cl)
                t " Intermediate points: "
                  (nreverse (cdr (reverse (cdr cl))))
                t " Ending   point   is: " (car (last cl))
                t " Maximum iterations      : " n
                t)
        (:= dpla (dpl 'tst-op-f 'tst-w-f 'tst-b-f 'tst-p-f
                      cl n cont opn-dmp))
        (:= path (for [x in (car dpla)]
                      (save (dp-coord x))
                      ))
        (:= opns (for [x in (cadr dpla)]
                      (save (dp-coord x))
                      ))
        (ttymsg t "Do you want to see the coordinates? ")
```

```
          (cond [(is-yes (read))
                    (or (memq 'dplot (workspaces))
                        (load 'dplot))
                    (prtpts path)])
          (ttymsg t "Do you want to see a plot? ")
          (cond [(is-yes (read))
                    (or (memq 'dplot (workspaces))
                        (load 'dplot))
                    (plotpts2 path opns)])
          (ttymsg t "DPTST again? ")
          (cond [(is-yes (read)) (go top)])
          )
    )
;
(de dptst-init ()
    (cond [(get 'dptst-init 'dptst-init)
               t]
          [t (allocate 'list 100)
             (allocate 'flonum 50)
             (allocate 'fixnum 20)
             (putprop 'dptst-init t 'dptst-init)])
    )
;
(de make-coord (l)
    (cond [(and (consp l) (fixp (car l)) (fixp (cdr l))) l]
          [(and (consp l) (fixp (car l))
                (consp (cdr l)) (fixp (cadr l)) (null (cddr l)))
             (rplacd l (cadr l))])
    )
;
!; tst-op-f (tst-op-f coord start-coord end-coord node)
;
;     TST-OP-F
!;
(de tst-op-f (c sc ec node)
    (cond [(or (< (xcoord c) 1)
               (< (ycoord c) 1)
               (> (xcoord c) 100)
               (> (ycoord c) 100)
               )
             (:= sc 1000000.0)]
          [t (:= sc
                 (times (diff tst-max*
                              (min (straight-dist c '(50 . 50))
                                   (straight-dist c '(25 . 25))
                                   (straight-dist c '(25 . 75))
                                   (straight-dist c '(75 . 25))
                                   (straight-dist c '(75 . 57))
                                   ))
                        tst-f*))])
    (list (plus sc (straight-dist c ec)) sc node)
    )
;
```

```
;
(defv tst-f* 1.0)
;
(de tst-w-f (x)
  (cond [x (car x)]
        [t 0])
  )
;
(de tst-b-f (e)  (caddr e)  )
;
(de tst-p-f (n)
  (ttymsg "{" (car n) "," (cadr n) "}")
  )
;
!; straight-dist (straight-dist coord1 coord2)
;
;     STRAIGHT-DIST returns the "straight" distance between the
; coordinates COORD1 and COORD2. A coordinate has the form
; (x . y).
!;
(de straight-dist (c1 c2)
  (sqrt (plus (expt (difference (xcoord c1) (xcoord c2)) 2)
              (expt (difference (ycoord c1) (ycoord c2)) 2)
              ))
  )
;
(defv tst-max* (straight-dist '(0 . 0) '(25 . 25)))
;
(remprop 'dp-test 'wsviolated)
(princ "DP-TST Loaded")
(terpr)
```

9.0 Appendix B: Experimental Predicate Calculus Model

This code is written in DUCK [8], a non-monotonic deductive retrieval system and language. This version is for a Symbolics 3600, but it will work in Franz Lisp on VAX 11/780 computers after cosmetic changes. An implementation detail, the system uses fairly elaborate forward chaining, so that route queries will be answered more quickly.

In order to use this software, you shoula move to directory [sstsys.hayes.ipb] and type "duck" (after the command prompt) and then type "(duck)". It will take at least 5 minutes for the "lisprc" file to load in the demo software befor you can type "(duck)". You will be in a read, deduce, and print loop. The prompt will be "g>". Interesting queries to type include:

```
(travel !<infantry> begin end)
(travel !<tank> begin end)
(travel !<truck> begin end)
(safe-travel !<infantry> begin end)
(safe-travel !<tank> begin end)
(safe-travel !<truck> begin end)
(travel !<infantry ?x> begin end)
(safe-travel !<infantry ?x> begin end)
```

Remember to respond "e", after each solution so that you can see the path taken and its cost. The "e" stands for "explain". After typing the "e", you will be in "walk mode", denoted by the prompt "w>". Type a "q" to exit walk mode back to the read-deduce-print loop. Now if you type an "a" (for "alternative"), the deductive retriever will respond with the next solution path. Typing a "q" will pop you out of the read-deduce-print loop back into Lisp.

```
; -*- Mode: Lisp; Package: NISP; Readtable: *nisp-readtable* -*-
;
;       Tank.1  -- To play Army in FOPC+ (Duck)
;                  4.1.84
;
;(workspace 'tank)
;(allocate 'list 200)
;
;     Define the data-type PLACE
;    Declare a number of PLACEs on the map
;
(terpri)(princ "Define PLACEs :")
(defducktype PLACE SYMBOL)
(duclare begin         PLACE)
;(duclare nob-hill       PLACE)
;(duclare heather-field PLACE)
;(duclare pea-pass       PLACE)
;(duclare ravens-croft  PLACE)
(duclare river-hole    PLACE)
(duclare west-bridge   PLACE)
(duclare west-river    PLACE)
(duclare west-ford     PLACE)
(duclare east-bridge   PLACE)
(duclare east-river    PLACE)
(duclare east-ford     PLACE)
(duclare johnson-hole  PLACE)
(duclare nathen-crag   PLACE)
(duclare end           PLACE)
;
;     Define the coordinates of places on the map
;    using the "position" predicate.
;
(terpri)(princ "Define POSITIONs :")
(duclare position (fun PROP (PLACE FIXNUM FIXNUM) ()))
(/:/:/: position template /:/:/:
     ((position ?p ?x ?y) (?p " is at (" ?x "," ?y ")")))
;
(rule begin-position          (position begin         0  0))
;(rule nob-hill-position       (position nob-hill       5  5))
;(rule heather-field-position (position heather-field 7  1))
;(rule pea-pass-position       (position pea-pass      10  7))
;(rule ravens-croft-position  (position ravens-croft 11  4))
(rule river-hole-position     (position river-hole   13  6))
(rule west-bridge-position    (position west-bridge  14 10))
(rule west-river-position     (position west-river   16  7))
(rule west-ford-position      (position west-ford    18  4))
(rule east-bridge-position    (position east-bridge  15 11))
(rule east-river-position     (position east-river   17  8))
(rule east-ford-position      (position east-ford    19  5))
(rule johnson-hole-position   (position johnson-hole 17 10))
(rule nather-crag-position    (position nathen-crag  19 11))
(rule end-position            (position end          17 12))
```

```
;
;     Define the data-type UNIT
;    Declare different UNITs
;
(terpri)(princ "Define UNITs :")
(defducktype UNIT SYMBOL)
(duclare infantry UNIT)
(duclare truck    UNIT)
(duclare tank     UNIT)
;
;     Define the data-type SURFACE
;    Declare different SURFACEs
;
(terpri)(princ "Define SURFACEs :")
(defducktype SURFACE SYMBOL)
(duclare highway  SURFACE)
(duclare road     SURFACE)
(duclare path     SURFACE)
(duclare bridge   SURFACE)
(duclare river    SURFACE)
(duclare ford     SURFACE)
;
;     Define the data-type GRADE
;    Declare different GRADEs
;
(terpri)(princ "Define GRADEs :")
(defducktype GRADE SYMBOL)
(duclare level GRADE)
(duclare grade GRADE)
(duclare steep GRADE)
;
;     Define the data-type DANGER
;    Declare different DANGERs
;
(terpri)(princ "Define DANGERs :")
(defducktype DANGER SYMBOL)
(duclare rifle-fire       DANGER)
(duclare machine-gun-fire DANGER)
(duclare mortar-fire      DANGER)
(duclare artillary-fire   DANGER)
(duclare bazzuka-fire     DANGER)
(duclare land-mines       DANGER)
;
;     Declare the Proposition "danger-to"
;
(terpri)(princ "Define Danger-to's :")
(duclare danger-to
        (fun PROP (UNIT DANGER) ()))
(/:/:/: danger-to template /:/:/:
     ((danger-to ?u ?d)
       (?d " is a danger to " ?u)))
;
```

```
;
(rule infantry-danger-1  (danger-to infantry rifle-fire))
(rule infantry-danger-2  (danger-to infantry machine-gun-fire))
(rule infantry-danger-3  (danger-to infantry mortar-fire))
(rule infantry-danger-4  (danger-to infantry artillary-fire))
(rule infantry-danger-5  (danger-to infantry bazzuka-fire))
(rule infantry-danger-6  (danger-to infantry land-mines))
;
(rule truck-danger-1     (danger-to truck    rifle-fire))
(rule truck-danger-2     (danger-to truck    machine-gun-fire))
(rule truck-danger-3     (danger-to truck    mortar-fire))
(rule truck-danger-4     (danger-to truck    artillary-fire))
(rule truck-danger-5     (danger-to truck    bazzuka-fire))
(rule truck-danger-6     (danger-to truck    land-mines))
;
(rule tank-danger-1 (not (danger-to tank     rifle-fire)))
(rule tank-danger-2 (not (danger-to tank     machine-gun-fire)))
(rule tank-danger-3 (not (danger-to tank     mortar-fire)))
(rule tank-danger-4      (danger-to tank     artillary-fire))
(rule tank-danger-5      (danger-to tank     bazzuka-fire))
(rule tank-danger-6      (danger-to tank     land-mines))
;
;    A map
;
;                                * end
;                            * e-b   * n-crag
;                          * w-b * j-hole
;
;                                * e-river
;                  * p-pass    * w-river
;                        * r-hole
;        * n-hill                    * e-ford
;                   * r-croft      * w-ford
;
;
;            * heather-field
;  * begin
;
;    Declare the Proposition "path"
;
(terpri)(princ "Define Path's :")
(duclare path
        (fun PROP (PLACE PLACE SURFACE GRADE (1st DANGER)) ()))
(/:/:/: path template /:/:/:
     ((path ?p1 ?p2 ?s ?g ?d)
       ("The " ?g ?s " from " ?p1 " to " ?p2 " has dangers " ?d))
     )
;
```

```
;
;     This is the description of the connections in the map
;
(rule path-begin-to-west-bridge
      (path begin         west-bridge   highway  level
            (tup rifle-fire)))
(rule path-begin-to-river-hole
      (path begin         river-hole    road     level (tup)))
;(rule path-begin-to-nob-hill
;     (path begin         nob-hill      highway  level (tup)))
;(rule path-begin-to-heather-field
;     (path begin         heather-field road     grade (tup)))
;(rule path-nob-hill-to-pea-pass
;     (path nob-hill      pea-pass      highway  level (tup)))
;(rule path-nob-hill-to-heather-field
;     (path nob-hill      heather-field road     grade (tup)))
;(rule path-heather-field-to-nob-hill
;     (path heather-field nob-hill      road     grade (tup)))
;(rule path-heather-field-to-ravens-croft
;     (path heather-field ravens-croft  road     level (tup)))
;(rule path-pea-pass-to-west-bridge
;     (path pea-pass      west-bridge   highway  grade
;           (tup mortar-fire bazzuka-fire)))
;(rule path-pea-pass-to-river-hole
;     (path pea-pass      river-hole    road     grade (tup)))
;(rule path-pea-pass-to-ravens-croft
;     (path pea-pass      ravens-croft  road     level (tup)))
;(rule path-ravens-croft-to-pea-pass
;     (path ravens-croft  pea-pass      road     level (tup)))
;(rule path-ravens-croft-to-west-ford
;     (path ravens-croft  west-ford     road     grade (tup)))
(rule path-river-hole-to-west-bridge
      (path river-hole    west-bridge   road     grade
            (tup mortar-fire bazzuka-fire)))
(rule path-west-bridge-to-river-hole
      (path west-bridge   river-hole    road     grade
            (tup mortar-fire bazzuka-fire)))
(rule path-river-hole-to-west-river
      (path river-hole    west-river    path     grade (tup)))
(rule path-river-hole-to-west-ford
      (path river-hole    west-ford     road     grade (tup)))
;(rule path-west-ford-to-river-hole
;     (path west-ford     river-hole    road     grade (tup)))
(rule path-west-bridge-to-east-bridge
      (path west-bridge   east-bridge   bridge   level
            (tup machine-gun-fire)))
(rule path-west-river-to-east-river
      (path west-river    east-river    river    level (tup)))
(rule path-west-ford-to-east-ford
      (path west-ford     east-ford     ford     level
            (tup rifle-fire machine-gun-fire)))
(rule path-east-bridge-to-end
      (path east-bridge   end           path     steep (tup)))
(rule path-east-bridge-to-johnson-hole
      (path east-bridge   johnson-hole  highway  grade (tup)))
```

```
(rule path-johnson-hole-to-east-bridge
      (path johnson-hole  east-bridge   highway  grade (tup)))
(rule path-east-river-to-johnson-hole
      (path east-river    johnson-hole  path     level (tup)))
(rule path-east-ford-to-nathen-crag
      (path east-ford     nathen-crag   road     grade (tup)))
(rule path-johnson-hole-to-nathen-crag
      (path johnson-hole  nathen-crag   highway  grade (tup)))
(rule path-nathen-crag-to-johnson-hole
      (path nathen-crag   johnson-hole  highway  grade (tup)))
(rule path-nathen-crag-to-end
      (path nathen-crag   end           highway  level (tup)))
;
;    Different units can move over different surfaces
;   with different grades at different rates.
;
;             Infantry  trucks    tanks
;
; Highway & Bridge
;  -level     6        50        30
;  -grade     4        30        10
;  -steep     3        10         4
; Road
;  -level     6        35        25
;  -grade     4        20        10
;  -steep     3         8         4
; Path
;  -level     5         *         3
;  -grade     3         *         1
;  -steep     2         *         *
; River
;  -level     1         *         *
;  -grade     1         *         *
;  -steep     *         *         *
; Ford
;  -level     2         2         1
;  -grade     2         2         1
;  -steep     *         *         1
;
;     Declare the Proposition "travel-cost"
;
(terpri)(princ "Define Travel-cost's :")
(duclare travel-cost
         (fun PROP (UNIT SURFACE GRADE FIXNUM) ()))
(/:/:/: travel-cost template /:/:/:
     ((travel-cost ?u ?s ?g ?c)
       (?u " moving on "?g ?s " costs " ?c)))
;
```

```
;
(rule highway-travel-cost-1
     (travel-cost infantry highway level 6))
(rule highway-travel-cost-2
     (travel-cost truck    highway level 50))
(rule highway-travel-cost-3
     (travel-cost tank     highway level 30))
(rule highway-travel-cost-4
     (travel-cost infantry highway grade 4))
(rule highway-travel-cost-5
     (travel-cost truck    highway grade 30))
(rule highway-travel-cost-6
     (travel-cost tank     highway grade 10))
(rule highway-travel-cost-7
     (travel-cost infantry highway steep 3))
(rule highway-travel-cost-8
     (travel-cost truck    highway steep 10))
(rule highway-travel-cost-9
     (travel-cost tank     highway steep 4))
;
(rule bridge-travel-cost
     (-> (travel-cost ?unit highway ?grade ?cost)
         (travel-cost ?unit bridge  ?grade ?cost)))
;
(rule road-travel-cost-1
     (travel-cost infantry road    level 6))
(rule road-travel-cost-2
     (travel-cost truck    road    level 25))
(rule road-travel-cost-3
     (travel-cost tank     road    level 35))
(rule road-travel-cost-4
     (travel-cost infantry road    grade 4))
(rule road-travel-cost-5
     (travel-cost truck    road    grade 20))
(rule road-travel-cost-6
     (travel-cost tank     road    grade 10))
(rule road-travel-cost-7
     (travel-cost infantry road    steep 3))
(rule road-travel-cost-8
     (travel-cost truck    road    steep 8))
(rule road-travel-cost-9
     (travel-cost tank     road    steep 4))
;
(rule path-travel-cost-1
     (travel-cost infantry path    level 5))
(rule path-travel-cost-2
     (travel-cost tank     path    level 3))
(rule path-travel-cost-3
     (travel-cost infantry path    grade 3))
(rule path-travel-cost-4
     (travel-cost tank     path    grade 1))
(rule path-travel-cost-5
     (travel-cost infantry path    steep 2))
;
```

```
;
(rule river-travel-cost-1
      (travel-cost infantry river   level 1))
(rule river-travel-cost-2
      (travel-cost infantry river   grade 1))
;
(rule ford-travel-cost-1
      (travel-cost infantry ford    level 2))
(rule ford-travel-cost-2
      (travel-cost truck    ford    level 2))
(rule ford-travel-cost-3
      (travel-cost tank     ford    level 1))
(rule ford-travel-cost-4
      (travel-cost infantry ford    grade 2))
(rule ford-travel-cost-5
      (travel-cost truck    ford    grade 2))
(rule ford-travel-cost-6
      (travel-cost tank     ford    grade 1))
(rule ford-travel-cost-7
      (travel-cost tank     ford    steep 1))
;
;     Define function to compute distance between coordinates
;
(terpri)(princ "Define Crow-fly :")
(func crow-fly FLONUM (x1 FIXNUM y1 FIXNUM x2 FIXNUM y2 FIXNUM)
      (/:= x1 (difference x2 x1))
      (/:= y1 (difference y2 y1))
      (sqrt (plus (times x1 x1) (times y1 y1)))
      )
;
;     Declare the Proposition "unit-path-time"
;    which defines the time it takes for a unit to move
;    between different places on the map.
;
(terpri)(princ "Define Primary unit-path-time's :")
(duclare unit-path-time
         (fun PROP
              (UNIT PLACE PLACE FLONUM (lst DANGER) (lst PLACE))
              ()))
(/:/:/: unit-path-time template /:/:/:
     ((unit-path-time ?u ?p1 ?p2 ?t ?d ?i)
      (?u " takes " ?t " hours"
       " from " ?p1 " thru " ?i " to " ?p2
       " under " ?d)))
;
```

```
;
(lisprule primary-path-cost-def ->
          (path ?pos1 ?pos2 ?surface ?grade ?dangers)
  (for-first-ans (fetch '(position ?pos1 ?x1 ?y1))
   (for-first-ans (fetch '(position ?pos2 ?x2 ?y2))
    (for-each-ans
         (fetch '(travel-cost ?unit ?surface ?grade ?speed))
     (for-first-ans (match ?time
                                 (quotient (crow-fly ?x1 ?y1 ?x2 ?y2)
                                           ?speed))
      (add '(unit-path-time
                  ?unit ?pos1 ?pos2 ?time ?dangers (tup)))
      )))) )
;
;
(terpri)(princ "Define Disjoint-sets function :")
(func disjoint-sets BOOLEAN (l1 EXP l2 EXP)
      (cond [(null l1) t]
            [(member (car l1) l2) nil]
            [t (disjoint-sets (cdr l1) l2)])
      )
;
;
(terpri)(princ "Define Secondary unit-path-time's :")
(lisprule secondary-path-cost-def ->
     (unit-path-time ?unit ?pos1 ?pos2 ?time1 ?dangers1 (tup))
  (for-each-ans
       (fetch '(unit-path-time
                     ?unit ?pos2 ?pos3 ?time2 ?dangers2 (tup)))
   (for-first-ans (match ?time (plus ?time1 ?time2))
    (for-first-ans (match ?dangers
                                (append '(tup) ?dangers1 ?dangers2))
     (for-first-ans (match ?thru (list 'tup ?pos2))
      (cond [(eq ?pos1 ?pos3) nil]
            [t (add '(unit-path-time ?unit ?pos1 ?pos3
                                     ?time ?dangers ?thru))
               ])
      )))) )
;
(terpri)(princ "Define Tertiary unit-path-time's :")
(lisprule tertiary-path-cost-def ->
     (unit-path-time ?unit ?pos1 ?pos2 ?time1 ?dangers1 (tup))
  (for-each-ans
       (fetch '(unit-path-time
                   ?unit ?pos2 ?pos3 ?time2 ?dangers2 (tup ?t1)))
   (for-first-ans (match ?time (plus ?time1 ?time2))
    (for-first-ans (match ?dangers
                                (append '(tup) ?dangers1 ?dangers2))
     (for-first-ans (match ?thru (list 'tup ?pos2 ?t1))
      (cond [(eq ?pos1 ?pos3) nil]
            [(eq ?pos1 ?t1) nil]
            [t (add '(unit-path-time ?unit ?pos1 ?pos3
                                     ?time ?dangers ?thru))])
      )))) )
;
```

```
;
(terpri)(princ "Define Quadiary unit-path-time's :")
(lisprule quadiary-path-cost-def ->
    (unit-path-time ?unit ?pos1 ?pos2 ?time1 ?dangers1 (tup ?t1))
  (for-each-ans
       (fetch
        '(unit-path-time
               ?unit ?pos2 ?pos3 ?time ?dangers2 (tup ?t2)))
   (for-first-ans (match ?time (plus ?time1 ?time2))
    (for-first-ans (match ?dangers
                           (append '(tup) ?dangers1 ?dangers2))
     (for-first-ans (match ?thru (list 'tup ?t1 ?pos2 ?t2))
      (cond [(eq ?pos1 ?pos3) nil]
            [(eq ?pos1 ?t2) nil]
            [(eq ?t1 ?t2) nil]
            [(eq ?t1 ?pos3) nil]
            [t (add '(unit-path-time ?unit ?pos1 ?pos3
                                      ?time ?dangers ?thru))])
      )))) )
;
(terpri)(princ "Define Quiniary unit-path-time's :")
(lisprule quiniary-path-cost-def ->
    (unit-path-time ?unit ?pos1 ?pos2 ?time1 ?dangers1 (tup ?t1))
  (for-each-ans
       (fetch
        '(unit-path-time
               ?unit ?pos2 ?pos3 ?time2 ?dangers2 (tup ?t2 ?t3)))
   (for-first-ans (match ?time (plus ?time1 ?time2))
    (for-first-ans (match ?dangers
                           (append '(tup) ?dangers1 ?dangers2))
     (for-first-ans (match ?thru (list 'tup ?t1 ?pos2 ?t2 ?t3))
      (cond [(eq ?pos1 ?pos3) nil]
            [(eq ?pos1 ?t2) nil]
            [(eq ?pos1 ?t3) nil]
            [(eq ?t1 ?t2) nil]
            [(eq ?t1 ?t3) nil]
            [(eq ?t1 ?pos3) nil]
            [t (add '(unit-path-time ?unit ?pos1 ?pos3
                                      ?time ?dangers ?thru))])
      )))) )
;
```

```
(terpri)(princ "Define Sexary unit-path-time's :")
(lisprule sexary-path-cost-def ->
     (unit-path-time
           ?unit ?posl ?pos2 ?timel ?dangersl (tup ?tl ?t2))
  (for-each-ans
       (fetch
        '(unit-path-time
               ?unit ?pos2 ?pos3 ?time2 ?dangers2 (tup ?t3 ?t4)))
   (for-first-ans (match ?time (plus ?timel ?time2))
    (for-first-ans (match ?dangers
                          (append '(tup) ?dangersl ?dangers2))
     (for-first-ans (match ?thru
                           (list 'tup ?tl ?t2 ?pos2 ?t3 ?t4))
      (cond [(eq ?posl ?pos3) nil]
            [(eq ?posl ?t3) nil]
            [(eq ?posl ?t4) nil]
            [(eq ?tl ?t3) nil]
            [(eq ?tl ?t4) nil]
            [(eq ?tl ?pos3) nil]
            [(eq ?t2 ?t3) nil]
            [(eq ?t2 ?t4) nil]
            [(eq ?t2 ?pos3) nil]
            [t (add '(unit-path-time ?unit ?posl ?pos3
                                    ?time ?dangers ?thru))])
      )))) )
;
(terpri)(princ "Define Septary unit-path-time's :")
(lisprule septary-path-cost-def ->
     (unit-path-time
           ?unit ?posl ?pos2 ?timel ?dangersl (tup ?tl ?t2))
  (for-each-ans
       (fetch
        '(unit-path-time
           ?unit ?pos2 ?pos3 ?time2 ?dangers2 (tup ?t3 ?t4 ?t5)))
   (for-first-ans (match ?time (plus ?timel ?time2))
    (for-first-ans (match ?dangers
                          (append '(tup) ?dangersl ?dangers2))
     (for-first-ans (match ?thru
                           (list 'tup ?tl ?t2 ?pos2 ?t3 ?t4 ?t5))
      (cond [(eq ?posl ?pos3) nil]
            [(eq ?posl ?t3) nil]
            [(eq ?posl ?t4) nil]
            [(eq ?posl ?t5) nil]
            [(eq ?tl ?t3) nil]
            [(eq ?tl ?t4) nil]
            [(eq ?tl ?t5) nil]
            [(eq ?tl ?pos3) nil]
            [(eq ?t2 ?t3) nil]
            [(eq ?t2 ?t4) nil]
            [(eq ?t2 ?t5) nil]
            [(eq ?t2 ?pos3) nil]
            [t (add '(unit-path-time ?unit ?posl ?pos3
                                    ?time ?dangers ?thru))])
      )))) )
;
```

```
;
(terpri)(princ "Define Octary unit-path-time's :")
(lisprule octary-path-cost-def ->
     (unit-path-time
           ?unit begin west-ford ?timel ?dangersl (tup ?tl ?t2))
  (for-first-ans
      (fetch
       '(unit-path-time ?unit west-ford end ?time2 ?dangers2
                        (tup ?t3 ?t4 ?t5 ?t6)))
   (for-first-ans (match ?time (plus ?timel ?time2))
    (for-first-ans (match ?dangers
                         (append '(tup) ?dangersl ?dangers2))
     (for-first-ans (match ?thru
                          (list 'tup ?tl ?t2 'west-ford
                               ?t3 ?t4 ?t5 ?t6))
      (add '(unit-path-time ?unit ?posl ?pos3
                          ?time ?dangers ?thru))
      )))) )
;
(terpri)(princ "Define Safe-units proposition")
(duclare safe-units (fun PROP ((lst UNIT) DANGER) ()))
(/:/:/: safe-units template /:/:/:
     ((safe-units ?units ?danger)
      (?units " are safe from " ?danger)))
;
(rule safe-units-when-first-unit-safe
      (<- (safe-units (tup ?unitl !& ?units) ?danger)
          (not (danger-to ?unitl ?danger))))
;
(rule safe-units-when-some-unit-safe
      (<- (safe-units (tup ?unitl !& ?units) ?danger)
          (and (danger-to ?unitl ?danger)
               (safe-units ?units ?danger))))
;
(terpri)(princ "Define Safe proposition")
(duclare safe (fun PROP ((lst UNIT) (lst DANGER)) ()))
(/:/:/: safe template /:/:/:
     ((safe ?units ?dangers)
      (?units " are safe from " ?dangers)))
;
(rule safe-when-no-danger
      (safe ?units (tup)))
;
(rule safe-for-one-or-more-dangers
      (<- (safe ?units (tup ?danger !& ?dangers))
          (and (safe-units ?units ?danger)
               (safe ?units ?dangers))))
;
```

```
;
(terpri)(princ "Defining Travel-same Proposition")
(duclare travel-same
         (fun PROP ((lst UNIT) PLACE PLACE (lst PLACE)) ()))
(/:/:/: travel-same template /:/:/:
     ((travel-same ?units ?start ?finish ?thru)
      (?units " can travel from " ?start
        " through " ?thru " to " ?finish)))
;
(rule travel-same-only-no-unit
      (travel-same (tup) ?start ?finish ?thru))
;
(rule travel-same-many-units
      (<- (travel-same !<?unit !& ?units> ?start ?finish ?thru)
          (and (unit-path-time
                     ?unit ?start ?finish ?time ?dangers ?thru)
               (travel-same ?units ?start ?finish ?thru))))
;
(terpri)(princ "Defining Travel Proposition")
(duclare travel (fun PROP ((lst UNIT) PLACE PLACE) ()))
(/:/:/: travel template /:/:/:
     ((travel ?units ?start ?finish)
      (?units " can travel from " ?start " to " ?finish)))
;
(rule travel-no-unit-rule
      (travel (tup) ?start ?finish))
;
(rule travel-units-rule
      (<- (travel (tup ?unit !& ?units) ?start ?finish)
          (and (unit-path-time
                     ?unit ?start ?finish ?time ?dangers ?thru)
               (travel-same ?units ?start ?finish ?thru))))
;
(terpri)(princ "Defining Safe-travel Proposition")
(duclare safe-travel (fun PROP ((lst UNIT) PLACE PLACE) ()))
(/:/:/: safe-travel template /:/:/:
     ((safe-travel ?units ?start ?finish)
      (?units " can travel safely from " ?start " to " ?finish)))
;
(rule safe-travel-no-units
      (safe-travel (tup) ?start ?finish))
;
(rule safe-travel-many-units
      (<- (safe-travel (tup ?unit !& ?units) ?start ?finish)
          (and (unit-path-time ?unit ?start ?finish
                               ?time ?dangers ?thru)
               (travel-same ?units ?start ?finish ?thru)
               (safe (tup ?unit !& ?units) ?dangers))))
;
(terpri)
(princ "Tank.1 Loaded")
(terpri)
```