

TEC-0007

AD-A252 556



2



US Army Corps
of Engineers
Topographic
Engineering Center

Intelligent TDA Systems Using Neural Networks Phase I Final Report

Richard H. Stottler
Andrea L. Henke

DTIC
ELECTE
JUL 09 1992
S A D

Stottler Henke Associates, Inc.
916 Holly Road
Belmont, CA 94002

May 1992

Approved for public release; distribution is unlimited.

92 17704

U.S. Army Corps of Engineers
Topographic Engineering Center
Fort Belvoir, Virginia 22060-5546

T

E

C



92-17704



Destroy this report when no longer needed.
Do not return it to the originator.

The findings in this report are not to be construed as an official
Department of the Army position unless so designated by other
authorized documents.

The citation in this report of trade names of commercially available products does not
constitute official endorsement or approval of the use of such products.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE May 1992		3. REPORT TYPE AND DATES COVERED Technical Report October 1991 - May 1992	
4. TITLE AND SUBTITLE Intelligent TDA Systems Using Neural Networks Phase I Final Report				5. FUNDING NUMBERS	
6. AUTHOR(S) Richard H. Stottler Andrea L. Henke					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Stottler Henke Associates, Inc. 916 Holly Road Belmont, CA 94002				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) U.S. Army Topographic Engineering Center Fort Belvoir, VA 22060-5546				10. SPONSORING/MONITORING AGENCY REPORT NUMBER TEC-0007	
11. SUPPLEMENTARY NOTES Effective 1 October 1991, the U.S. Army Engineer Topographic Laboratories (ETL) became the U.S. Army Topographic Engineering Center (TEC).					
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.				12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) The overall objective of Phase I was to develop a prototype intelligent Tactical Decision Aids (TDA) neural network system through the use of an expert system to neural network translator. From the resulting prototype network, the performance of the complete TDA system could be determined. The Phase I results absolutely demonstrated the feasibility of this concept and cleared the way for development of a complete system in Phase II. Current TDA systems require too much expertise from the user and run too slowly for use in tactical situations, when time is in short supply. In tactical situations, terrain is an extremely important factor and represents one of the main inputs to these systems. The Airborne Avenues of Approach (AAA) Planning problem was chosen. While initial route planning is performed prior to the start of the mission, a need exists to automate and speed the process. Additionally, a system which could meet the rigorous real-time requirements of an on-board AAA re-planner would prove enormously beneficial in many time-critical tactical situations.					
14. SUBJECT TERMS Helicopter Path Planning, Helicopter Avenues of Approach, Airborne Avenues of Approach, Artificial Intelligence, Tactical Decision Aids, Translation Neural Networks, Expert Systems				15. NUMBER OF PAGES 72	
				16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UNLIMITED		

Table of Contents

<u>Section</u>	<u>Page</u>
1.0 PREFACE	v
2.0 SUMMARY	1
3.0 PROJECT OBJECTIVES	2
3.1 Identification and Significance of the Problem	2
3.1.1 Airborne Avenues of Approach	3
3.1.2 Characteristics of Expert Systems	3
3.1.3 Characteristics of Neural Networks	4
3.1.4 Opportunity	5
3.1.5 Innovations	6
3.2 Statement of Objectives	6
4.0 WORK DESCRIPTION	7
4.1 Approach	7
4.2 Task Descriptions	7
4.2.1 Identify Tactical Decision Aids (TDA) Problem	7
4.2.2 Knowledge Engineering	8
4.2.3 Translate Expert System to Neural Network Representation	8
4.2.3.1 Update Prototype Translator	8
4.2.3.2 Develop Additional Translation Strategies ...	9
4.2.3.3 Implement Additional Translation Strategies .	9
4.2.4 Design and Implement Demonstration Application ...	9
4.2.5 Design Phase II Software Architecture	9
4.2.6 Determine Performance of Final System	9
4.2.7 Prepare Final Report	9
5.0 TECHNICAL RESULTS AND PHASE I ACCOMPLISHMENTS	11
5.1 Summary of Results	11
5.2 TDA Problem Selection and Analysis	11
5.3 Knowledge Engineering Results	12
5.3.1 Knowledge Gathering	12
5.3.2 Knowledge Implementation	15
5.4 Translation Results	16
5.4.1 Translation Strategies	16
5.4.1.1 Expert System Concepts	16
5.4.1.2 Translatable Concepts	17
5.4.1.3 Untranslatable Concepts	17
5.4.1.4 The Logic Network Representation	18
5.4.1.5 The Neural Network Representation	19
5.4.1.6 Additional Strategies	19
5.4.2 Architecture of the Translator	20
5.4.3 Translator Updating Results	22
5.4.4 Translator Performance Results	22
5.5 Phase II Software Architecture	23

5.6	Phase II Predicted Performance	25
6.0	TECHNICAL FEASIBILITY	27
7.0	FUTURE RESEARCH AND DEVELOPMENT	28
7.1	Future Development	28
7.1.1	Helicopter Avenues of Approach TDA Development ...	28
7.1.2	Improvements to the Translator	29
7.2	Future Research	30
7.2.1	Adaptability	31
7.2.2	Fault Tolerance	30
7.2.3	Generalization	31
7.2.4	Neural Structuring	31
7.2.4	Optimizations	32
8.0	CONCLUSIONS.....	34
APPENDIX A	PHASE II DESIGN	35
APPENDIX B	COMPLETE TRANSLATION STRATEGIES	40
B.1	Rule to Logic Network Translation	40
B.1.1	Logic Network Description	40
B.1.2	Rule to Logic Network Translation Strategies	43
B.2	Logic Network to Neural Network Translation	45
B.2.1	Neural Network Target Representation	45
B.2.2	Logic Network to Neural Network Translation	45
B.3	Additional Translation Strategies	52
B.3.1	Rating Summation	52
B.3.2	Route Planning	53
APPENDIX C	DESCRIPTION OF HELICOPTER PROOF OF CONCEPT PROTOTYPE	56
APPENDIX D	EXPERT SYSTEM RULE-BASE	58

Accession For	
NTIS CRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	

List of Figures

<u>Figure</u>	<u>Page</u>
5-1 Translator Architecture	21
5-2 Phase II Software Architecture	24
A-1 Phase II Software Architecture	35
A-2 Landform Identification Neural Network Layers	36
A-3 Polygon Combiner	37
A-4 AAA Planner Expert System Development	37
A-5 Expert System to Neural Network Translation	37
A-6 AAA Planner Configuration	38
B-1 Gate Truth Tables	40
B-2 Simple Antecedent Logic Network	42
B-3 Simple Consequent Logic Network	42
B-4 Complex And Antecedent Logic Network	43
B-5 Complex Or Antecedent Logic Network	44
B-6 Real And Configuration	47
B-7 Real Or Configuration	47
B-8 Real Not Configuration	48
B-9 Real Then Configuration	48
B-10 Real Unknown Configuration	49
B-11 Binary And Configuration	50
B-12 Binary Or Configuration	50
B-13 Binary Not Configuration	51
B-14 Binary Then Configuration	51
B-15 Before Translation	52
B-16 After Translation	52
B-17 Route Planner Fragment	54
B-18 Binary Min Gate	55
B-19 Same Gate	55
B-20 Sum Gate	55
C-1 Polygon Map Display	56
C-2 Route Display	57

1.0 PREFACE

This report was prepared under Contract DACA76-91-C-0026 for the U.S. Army Topographic Engineering Center, Fort Belvoir, Virginia 22060-5546 by Stottler Henke Associates, Inc., Belmont, California 94002. The Contracting Officer's Representative was James A. Shine.

2.0 SUMMARY

The overall objective of Phase I was to develop a prototype intelligent Tactical Decision Aids (TDA) neural network system through the use of an expert system to neural network translator. From the resulting prototype network, the performance of the complete TDA system could be determined. The Phase I results absolutely demonstrated the feasibility of this concept and cleared the way for development of a complete system in Phase II.

Current TDA systems require too much expertise from the user and run too slowly for use in tactical situations, when time is in short supply. In tactical situations, terrain is an extremely important factor and represents one of the main inputs to these systems.

Our approach was to first choose a particular domain. The Airborne Avenues of Approach (AAA) Planning problem is a significant one in the TDA class of problems. While initial route planning is performed prior to the start of the mission, a need exists to automate and speed the process. Additionally a system which could meet the rigorous real-time requirements of an on-board AAA re-planner would prove enormously beneficial in many time-critical tactical situations.

The relevant knowledge was gathered and the system designed. We implemented the prototype solution with expert system technology. We then completely translated that system to a neural network representation using our automatic translator. The network's performance was evaluated using test cases and a neural network simulator. Additionally, the performance of the completed Phase II system was estimated. The time to completely plan a route in a 100 mile by 100 mile area was conservatively estimated to be 3 seconds with a \$4000 specialized board and a 486-based PC.

There are many potential applications of this research. The most direct application will be for Airborne Avenues of Approach Planning. This technology could be applied to many route planning problems including Avenues of Approach for ground units. The techniques used here would benefit any application involving the intelligent evaluation of a large number of items. Examples include threat evaluation and image understanding. Finally, the generic translator can be used to allow almost any expert system to be translated to a neural network and thus receive the combined benefits of each technology.

3.0 PROJECT OBJECTIVES

The overall objective of Phase I was to develop a prototype intelligent Tactical Decision Aids (TDA) neural network system. This was accomplished by developing a prototype expert system, and converting it to a neural network representation using Stottler Henke Associates, Inc.'s (SHAI) automatic expert system to neural network translator. From the resulting prototype network, the performance of the complete TDA system could be determined. This includes the required number of neurons and interconnections, and the processing speed required to meet real-time constraints. The Phase I tasks absolutely demonstrated the feasibility of this concept and cleared the way for development of a complete system in Phase II. The Phase II Design is given in Appendix A.

3.1 Identification and Significance of the Problem

Current TDA systems require too much expertise from the user and run too slowly for use in tactical situations, when time is in short supply. In order for the system to require less user expertise, raw data must be accessed directly by the systems, without any intervention or processing by the user. In tactical situations, terrain is an extremely important factor and represents one of the main inputs for the systems. Terrain data can be accessed directly from the Geographic Information System (GIS) which stores it along with other data, requiring much less involvement from the user.

The opportunity presented by this project is to apply intelligent methods which utilize both terrain information stored in a GIS, and other information available in real-time such as weather, field observations, and remote sensor system data. Such an intelligence can be easily approximated with expert system technology. However, the expert system will run too slowly to meet the rigorous real-time constraints of a tactical situation. It is also very difficult to tune the approximation to a completely correct solution. Additionally, expert systems are not fault-tolerant, adaptable, or generalizing. But, if the expert system approximation could be transformed into a neural network and its performance tuned through training, then the network would possess the high degree of intelligence easily implemented with expert system technology, run in real-time on neural network hardware, and realize other benefits of neural networks such as fault-tolerance, adaptation, and the ability to generalize.

Of course for the network to process the needed information most efficiently, it should also be in a parallel form. This is especially true for the terrain data, since it drives the decision processes of the TDA systems and also represents a very large volume of information. If the terrain data is available before the TDA systems must operate, the GIS information can also be converted

to a neural form so both the inference and the information upon which it operates are parallelized.

A GIS represents terrain data as a set of polygons, each of which has borders in two- or three- dimensional space. Additionally, associated with each polygon are attributes of the piece of terrain the polygon represents. The polygons can be represented in the artificial intelligence (AI) knowledge representation of a frame in a straightforward manner. These frames can then be linked to the expert system and the entire package translated to a neural network for parallel processing. Such a translation could occur almost as fast as the data can be transferred from the GIS.

3.1.1 Airborne Avenues of Approach

Airborne Avenues of Approach (AAA) Planning is a significant problem in the TDA class of problems. While initial route planning is performed prior to the start of the mission, a need exists to automate and speed the process. Additionally a system, which could meet the rigorous real-time requirements of an on-board AAA replanner, would prove enormously beneficial in many time-critical tactical situations.

The AAA Planner needs the high level of intelligence representable with expert system technology both from a correctness and from a understandability standpoint. Such a system also requires enormous data processing which indicates a need for parallelization to achieve the required speed. In addition, as in any fielded airborne, tactical system, fault-tolerance is extremely important. The AAA Planner is an ideal candidate for the use of SHAI's expert system to neural network translation technology.

3.1.2 Characteristics of Expert Systems

Expert systems represent knowledge as an explicit collection of facts, rules, and frames and provide inference procedures for manipulating this information. Because the facts, rules, and frames are not encoded as programs, knowledge can be added to the expert system knowledge-base or changed without affecting other existing knowledge. In addition, because the knowledge is declared explicitly rather than embedded in application procedures, it is straightforward to understand and can be used for many diverse purposes. This eliminates the need and potential problem of storing facts multiple times, once for each type of application. A declarative representation allows knowledge to be extended by a reasoning process and accessed by introspective programs, enabling the system to answer questions about what it knows.

The major drawback of expert systems is that reasoning over declarative information tends to be inefficient. This is because the knowledge is represented independent of procedures for

utilizing it, and because expert systems are generally run on serial single processor machines. In addition, altering an inference engine and/or an expert system to directly run on a parallel processor machine is extremely difficult. Devising a general method for dividing a problem into sub-problems which can be solved on multiple processors and then recombining their solutions is not a well-understood process.

Several other shortcomings arise because of the static nature of expert systems. Expert systems do not exhibit adaptable behavior. For example, if an expert system derives an answer and receives feedback that the answer is incorrect, we would like the expert system to change its behavior for the future. In addition, expert systems are unable to generalize from a set of examples in an inductive manner. While machine learning offers great potential in both these areas, it is still very much a difficult research topic. Human intervention is still required to modify an expert system's characteristics.

Expert systems are intolerant to variations in input. Consider, for example, the following expert system rule:

if A and B and C then D

The designer of the expert system may have assumed that full knowledge of the truth of A, B, and C would be available at the time this rule was checked. If A, B and C are available and true, D can be concluded. It may be the case, in some unusual circumstance, that only two of these three antecedents are known. While we may prefer the expert system to compromise and conclude that D is almost true, it is unable to conclude anything.

Finally, expert systems are not fault tolerant during hardware failure. As software, an expert system usually will crash completely or produce unreliable results after a hardware failure.

3.1.3 Characteristics of Neural Networks

Neural networks are composed of simple analog processing elements. Each processing element has any number of inputs and a single output. The output value is computed according to the following strategy:

- 1) Perform a weighted sum of the inputs,
- 2) Perform a simple function on that sum.

The output of a processing element is connected to the inputs of many other processing elements. Each connection has a weight associated with it which is used in the weighted sum computation.

Some of the inputs to processing elements are connected to the external world, as are some of the outputs. A neural network can

learn if it is presented a set of inputs and a set of desired outputs. The learning results by slightly changing the neural network's weights according to a learning algorithm.

Neural networks are a simple, but powerful parallel processing paradigm, offering the potential for a very large number of processing elements. Neural networks can be implemented in hardware or simulated in software. Implementation of a neural network on a parallel processing machine is straightforward. Implementation of a neural network is generally accomplished by dividing the processing elements among the available processors. Each processor simulates the functioning and interconnections of the processing elements allocated to it. By properly allocating the processing elements, full utilization of the multiple processors is possible.

Unlike expert systems, neural networks are adaptable and fault tolerant and can generalize solutions to problems. The primary shortcoming of neural networks is that they are difficult to design and understand. This is because the knowledge they use to solve problems is represented by interconnection weights, which are un-intuitive.

3.1.4 Opportunity

While expert systems are among the most common and successful AI systems, their slowness and fragility prevent their application directly to the Airborne Avenues of Approach Planner. However, by translating a rule and frame based AAA Planner into a neural network representation, performance is dramatically improved through parallelization and we derive the combined benefits of expert systems and neural networks. The performance improvement is dramatic enough to permit operation of the AAA Planner on-board the helicopter. The resulting system is adaptable, fault tolerant and easy to understand because of its expert system origin. The neural network representation can be implemented on parallel processing machines or neural network hardware.

Currently, neural network hardware with a large number of processing elements is available and orders of magnitude improvements in this number can be expected periodically. The combined processing power of these elements is much greater than that of conventional super computers. For example, while the Cray 2 can process 35 million connections per second, a single low-cost chip from Intel, the 80170NX, can process 2 billion connections per second. Expert systems can take full advantage of this power with the automatic expert system to neural network translation facility now developed.

The opportunity is to greatly improve on-board tactical decision making through the use of terrain data and other environmental information processed with an intelligent AAA neural

network developed through the use of an expert system translator and training, which tunes the resulting network's performance. The expert system origin allows the final AAA neural network to possess a degree of intelligence impossible from the use of neural network technology alone.

3.1.5 Innovations

Significant innovations were applied to this project. First, an Airborne Avenues of Approach Planner was developed utilizing standard AI techniques. The concept of translating this into a neural network is itself innovative. Few other researchers are working on the problem of translating implemented expert systems to a neural network representation. Furthermore, each binding for each rule-object combination can be represented in parallel. Finally, the combination of all the techniques to produce a working expert system to neural network translator is not only innovative, but many would have thought it impossible.

3.2 Statement of Objectives

Specifically, there were three technical objectives all of which were met. A fourth objective, to improve the network's performance through training, was considered but later dropped because the network's performance did not need improvement.

1. Develop an intelligent TDA system using conventional expert system technology.
2. Translate the expert system into a neural network.
3. Determine the performance of the complete TDA system, implemented in neural network hardware.

4.0 WORK DESCRIPTION

4.1 Approach

To achieve the primary project objective of developing an intelligent system which aids in tactical decisions based on both terrain data stored in a GIS and other environmental information, we identified a number of tasks to be carried out over the Phase I period. These tasks are listed below:

Phase I Tasks

1. Identify the TDA problem.
2. Perform knowledge engineering.
3. Translate the expert system to a neural network representation using the automatic translator.
4. Demonstrate the Application
5. Design the Phase II System
6. Determine the Performance of the Final System
7. Write the Phase I Final Report.

Our approach was to first identify the TDA problem. We chose a particular domain, determined what information was available and what was considered a correct decision based on this information. Second, the relevant knowledge was gathered and the system designed. Third, we implemented the prototype solution with expert system technology. We then translated that system to a neural network representation using the automatic translator. The network's performance was evaluated using test cases and a neural network simulator. Additionally, the performance of the completed Phase II system was estimated. Finally, the effort and results were documented in this Final Report.

4.2 Task Descriptions

4.2.1 Identify Tactical Decision Aids (TDA) Problem

This task consisted of a number of steps. First, the particular domain was identified. Possibilities included supply facilities location, sensor system site allocation, main battle unit movement corridor identification, integrated air defense planning, integrated defensive fire support planning, Joint Surveillance and Target Attack Radar System (JSTARS) patrolling patterns, smart weapons deployment, and Airborne Avenues of Approach (AAA). Then, performance criteria were identified. This

involved answering the following questions. What constitutes correct decisions based on the terrain and other information available? What are the real-time constraints? What information is available on which to base decisions? Will the network be directly connected to this source of information? Is the source of information hardware parallel in nature? Additionally, sources of domain knowledge had to be identified and training sets of data had to be gathered for testing and training.

4.2.2 Knowledge Engineering

This task included both knowledge acquisition and the design of the expert system. The following questions had to be answered: What conditions will the TDA face? What factors must be considered? Using the answers to these questions and considering that the expert system would be translated to a neural network, the expert system was designed. Considerations included how best to represent the knowledge to take advantage of the particular benefits resulting from the neural network translation.

For example, each polygon of the terrain information was represented as a frame. The polygon frame included slots to hold the spatial information such as the polygon border information and adjacent polygons, ground type, vegetation, and other information which could be used to make intelligent decisions.

The knowledge-based solution was implemented using Intellicorp's KAPPA expert system development tool. The implementation contained rules for intelligent tactical decision making. For example, when determining the best terrain for a helicopter avenue of approach, several factors were considered. The terrain should be shielded from enemy weapons systems. Shadows cast by the helicopter onto the terrain should be difficult to spot from the air. Preferably the terrain should be free from enemy ground units as well.

4.2.3 Translate Expert System to Neural Network Representation

The expert system was translated to a neural network representation using the automatic translator. Some procedural, numeric code needed to be approximated by small networks which were designed and translated separately from the main expert system network. Additionally, the inputs and outputs to the translated network were connected to a simple interface for demonstration purposes. The translation task consisted of a number of steps.

4.2.3.1 Update Prototype Translator

The translator was written in the summer of 1990. It needed to be updated to 1992 software standards. The intervening two years produced two major updates of KAPPA. The translator needed

to be ported across these updates. In addition, certain simplifications in the operation of the translator were developed.

4.2.3.2 Develop Additional Translation Strategies

Two additional types of translation were added to the translator. Both were particular to this application. One was to allow the summation of the polygon ratings inside of the network instead of a virtual processor (see Appendix B for a description of translation elements and strategies). The other was to use the ratings to perform route planning within the neural network. An additional step was to assemble the input vectors for each polygon separately from running the network for that polygon. This way the vector could be assembled ahead of time, which is closer to how the final Phase II system would operate.

4.2.3.3 Implement Additional Translation Strategies

The additional translation strategies were implemented in either Kappa or C, whichever was most expedient for each. The code to create the vector assembler was written in Kappa as was the code to translate the summations. The code to translate the route planning algorithm was written in C. The neural network simulations were all in C.

4.2.4 Design and Implement Demonstration Application

A demonstration AAA Planner was designed and implemented. This included a user interface to allow the Planner to be demonstrated. The purpose of this step was to show the flavor of the final Phase II system and prove that the intelligent neural network was possible.

4.2.5 Design Phase II Software Architecture

Based on performance and hardware considerations, the AAA Planner to be implemented in Phase II was designed. This design is carefully documented and explained in Section 5.5.

4.2.6 Determine Performance of Final System

The performance of the neural network prototype was evaluated through a simulation testing phase. The simulation allowed us to determine what percentage of the time the network performs correctly and in which situations, and whether decisions can be reached with the existing neural network hardware and real-time constraints. From this prototype, performance criteria of the completed system implemented in neural network hardware can also be estimated. The number of neurons created and the hardware available to simulate them was a major factor in the calculation of the expected performance.

4.2.7 Prepare Final Report

This final report documents the work performed in Phase I and lays the groundwork for Phase II. Specifically, the results of the evaluation of the prototype and the estimated performance of the completed Phase II system are detailed.

5.0 TECHNICAL RESULTS AND PHASE I ACCOMPLISHMENTS

5.1 Summary of Results

The AAA Planner was selected as the TDA problem solution for implementation and translation. Knowledge sources consisting primarily of Army documents were identified. Knowledge was extracted and implemented in IntelliCorp's Expert System Building Tool, Kappa. This expert system was completely translated into a neural network representation using both generic and AAA Planner specific strategies. Based on this Phase I experience, a Phase II system was designed and its performance was estimated. The time to completely plan a route in a 100 mile by 100 mile area was conservatively estimated to be 3 seconds with a \$4000 specialized board and a 486-based PC.

5.2 TDA Problem Selection and Analysis

Although the AAA Planner was ultimately chosen as the best TDA, it was not our only possibility. The other possibilities included supply facilities location, sensor system site allocation, main battle unit movement corridor identification, integrated air defense planning, integrated defensive fire support planning, Joint Surveillance and Target Attack Radar System (JSTARS) patrolling patterns, and smart weapons deployment.

TEC suggested that we choose a battlefield domain for the expert system implementation. Given this guidance and our own domain knowledge we eliminated supply facilities location and sensor system site allocation, leaving the five other domains. We researched various tactical domains and identified possible sources of knowledge for particular domains and narrowed our choices down to fire support, defense setup against tanks, and the ground avenues of approach.

We chose the ground avenues of approach domain and began work using terrain information gathered during a previous project. We developed rules which considered slope, elevation, soil type, weight bearing ability of the soil, tree density, vegetation, and type of vehicle. The rule base evaluated the traversability of polygons to certain types of vehicles. We also began formulating the route planning method used by the ultimate AAA Planner.

We then attended a meeting at TEC to discuss the particular direction and enhancements of the ground avenues of approach planner. At that time we discovered much work had already been done in the ground avenues of approach domain and that a need was anticipated for a helicopter or airborne avenues of approach planner. We identified relevant Army documents that covered this domain through a former Ft. Rucker contractor employee

5.3 Knowledge Engineering Results

5.3.1 Knowledge Gathering

Helpful Documents

The following documents were very helpful in providing detail on planning airborne avenues of approach:

- TC 1-201 Tactical Flight Procedures (especially chapter 6)
Training Circular NO. 1-201
- FM 1-202 Environmental Flight (provided tactical information about 4 different environments, mountain regions section was especially helpful)
Field Manual NO. 1-202
Headquarters Department of the Army, Washington DC
23 February 1983

Potentially Useful Related Documents

The following documents were referenced by the above documents and may provide helpful detailed information for future work:

- FM 1-402 Threat equipment
- AR 21-33 Terrain Analysis (Army Regulations)
- FM 1-101 Aircraft Battlefield Counter-measures and Survivability
- FM 1-204 Night Flight Techniques and Procedures

Other Documents

The following documents focussed either on general air/land battle doctrine without the detail necessary for planning purposes or on the training techniques required for learning to fly a helicopter.

- FC 1-214 AirCrew Training Manual, Attack Helicopter, AH-64
Field Circular NO 1-214
Headquarters Aviation Center, Fort Rucker, AL
31 May 1986
- FM 1-113 Assault Helicopter Battalion
Field manual No 1-113
Headquarters, Department of the Army, Washington, DC
28 October 1986
- TC 1-212 AirCrew Training Manual, Utility Helicopter, UH-60
Training Circular NO 1-212
Headquarters, Department of the Army, Washington, DC
3 October 1988
- FM 1-100 Doctrinal Principles for
Army Aviation In Combat Operations
Field Manual No. 1-100
Headquarters, Department of the Army, Washington, DC
28 February 1989

FM 1-116 Air Cavalry Troop
Field Manual NO 1-116
Headquarters, Department of the Army, Washington, DC
14 August 1986

FM 90-4 Air Assault Operations
Field Manual NO 90-4
Headquarters, Department of the Army, Washington, DC
16 March 1987

FM 1-112 Attack Helicopter Battalion
Field Manual No 1-112
Headquarters, Department of the Army, Washington, DC
14 July 1986

Knowledge Used in the Expert System Application

Definitions:

Terrain flight - low-level, contour and nap of the earth (NOE)
METT-T - Mission, Enemy, Terrain and weather, Troops, Time available
Low level flight - 50-200 feet above ground level (AGL)
NOE and contour flight altitude - 0-50 feet AGL
Nontactical flight altitude - 500 feet AGL
3 types of desert - mountain, rocky, and sandy (most arid regions are rocky)

Knowledge:

Threats:

- Keep highest terrain and thickest vegetation between threat and aircraft, if not possible, keep terrain behind helicopter
- Avoid air defense weapons and ground units
- Use friendly side of terrain features
- Use terrain which provides cover from visual observation or electronic detection
- Threats have trouble with rugged, swampy and heavily vegetated areas
- Use areas inaccessible to wheeled or tracked vehicles
- In gently rolling areas, use low terrain such as streambeds
- In arid and open areas use streambeds or depressions where there may be trees

Rotor Wash:

- Avoid snow and dust

Shadows:

- Avoid large bodies of water
- Use heavily vegetated rather than open terrain

Path Planning:

- Plan routes which provide recognizable checkpoints
- Fly in the lowest part of a valley
- Avoid routes which restrict maneuverability and channelize movement into a small area
- Avoid deep river valleys or gorges
- Do not follow linear, manmade features such as roads or canals
- Avoid built-up areas
- Avoid manmade obstacles such as wires and towers
- Avoid visibility restrictions such as fog, clouds, smoke
- Avoid poor weather areas
- Preselect alternate routes
- Avoid areas originating or targeted for friendly fire

Additional Knowledge of Interest

Wind:

(see FM 1-202, pp. 4-2 through 4-9, 4-19, 4-33, 4-36)

- Avoid turbulence
- Use updrafts
- Avoid flight near abrupt changes in terrain
- Sun heats the land and creates wind turbulence
- In the desert, avoid sand and dust storms because of damage caused

Personnel:

- Personnel factors may influence selection of terrain flight techniques: crew rest, proficiency, mission-oriented protective posture (MOPP)
- Terrain flight in mountains causes greater stress and fatigue

Multi-helicopter Operations:

- Spacing between helicopters varies depending on environment: In snow, 5-10 seconds separation; In landing zone, 15-30 seconds separation
- In snow, avoid narrow valleys or crevices with multiple helicopters

Altitude and Velocity Selection:

- Mission may dictate altitude (eg. M-56 aerial mines must be dispersed at 100 ft AGL)
- Fly NOE or Contour when within range of enemy's weapons
- Always fly at highest terrain flight altitude for specific condition to reduce navigation difficulty and minimize fatigue
- NOE flight over snow leaves a trail
- NOE flight over desert may leave dust signature or shadow
- Weather may prohibit visual flight

- Cold weather: If nontactical, fly high; fly > 40 knots to minimize rotor wash
- Mountains: Fly as fast as possible; NOE may be more dangerous than contour due to turbulence and terrain features
- Use NOE and contour in unfamiliar terrain where enemy detection is likely

Checkpoint Selection:

- Identify key terrain features to use as checkpoints

Plotting:

- Plot selected landing zones, ambush and/or firing positions, aircraft control points and known or suspected enemy positions
- Terrain is evaluated along the route for hazardous conditions

Miscellaneous:

- Use artillery delivered smoke in sparse areas
- Use indirect fires to suppress threat weapons
- Underfly wires
- Do not underfly wires over snow
- Cross over wires at poles or midpoint
- Cross under wires near poles; clearance shrinks with rising temperature
- Snowstorms and wind can change the appearance of an area
- Never turn tail toward enemy
- Avoid obvious avenues of approach into enemy territory
- Watch for wires stretched across narrow canyons
- Dash across open fields at the narrowest point
- Cross ridges at the lowest point
- Cross peaks at the lowest point
- Dash down the forward slope to the nearest concealment after crossing a ridge line

5.3.2 Knowledge Implementation

The tactical knowledge had to be converted into a form amenable to implementation as rules or procedures. In particular, certain concepts had to be defined. For example, in the phrase "large bodies of water", "large" had to be mathematically defined. Polygons were defined to have a Length (longest possible dimension) and an AverageWidth (Area divided by Length). "Large" was defined to be an AverageWidth greater than 50 meters. In addition, the concept of favoring or avoiding certain kinds of terrain was handled by having rules alter a Rating for each polygon. So the rule to avoid large bodies of water became:

```
[p|Polygons]
```

```
IF
```

```
  p:CoveredBy = Water And p:Width > 50
```

THEN

p:Rating = p:Rating - 4

A complete listing of the rules and methods used to calculate some of the polygon features is given in Appendix D.

5.4 Translation Results

5.4.1 Translation Strategies

5.4.1.1 Expert System Concepts

One of the primary objectives of the original translator research was to design strategies for converting a large percentage of expert system concepts to a neural network representation. The features common to a large class of expert systems include rules, frames (objects) and inference mechanisms. These concepts are the origin of the translation and the input to the translation strategies. Additionally, procedural code in the expert system must be allowed for.

The most general form of a rule is a simple IF-THEN statement. More complex rules employ any number of connectives, user-defined functions, and comparison operators. Examples of various types of rules are given below:

Simple rule	...	If A Then C
Connectives: And, Or, Not	...	If A Or B Then C
User-defined functions	...	If f(A) Then C
Comparison functions	...	If A > B Then C

The other commonly supported knowledge representation is the object-oriented frame. A frame object consists of slots representing object attributes, and slot values which can typically be numeric, textual, or boolean. Objects may be semantically connected into networks and referenced in the antecedent and consequent of rules. During reasoning, an expert system examines and sets slot values. In the following example, the CoveredBy slot of the object, Polygon, would be examined during inference and the Rating slot would be modified by the rule.

Object: Polygon

<u>Slots</u>	<u>Values</u>
CoveredBy	Vegetation
Trafficable	TRUE
OwnedBy	Enemy
AAWCoverage	FALSE
Rating	10

```
If Polygon:CovetedBy = Vegetation  
Then Polygon:Rating = Polygon:Rating + 2;
```

The reasoning mechanisms generally supported in an expert system are forward and backward chaining, with typical search strategies of depth first, breadth first and best first.

5.4.1.2 Translatable Concepts

After reviewing the expert system representations and inference strategies described in the previous section, we resolved to translate slot and rule structures, including complex rules with connectives and comparison operators and rules containing objects and slots. After examination of numerous expert system applications, we established the fact that in a well-designed expert system, the order of evaluation of rules during inference should have no bearing on the ultimate outcome of the inference. That is, the choice of forward chaining versus backward chaining, enhanced with any of the search strategies, should not have an impact on the answer produced by the reasoning process, only on the efficiency with which the answer is found. Therefore, if rules are translated and thus effectively parallelized, the result of activating them simultaneously should continue to produce the same answer. Inference is therefore translatable. Additionally, by selecting intelligent ranges of slot values and representing each range with a neuron, we found that most slots are also translatable.

5.4.1.3 Untranslatable Concepts

The portions of an expert system that we cannot translate include procedural programming code as in user-defined functions and methods, and the user interface code. These untranslatable components are called expert system fragments. The objects and slots that are set as input by the user are not normally translated. An exception was made for the AAA application so that they can be translated and hooked to a virtual processor which assembles the input vector for the neural network from the input objects and slots set by the user of the AAA Planner.

Although the fragments are not translated into neurons, they are nevertheless parallelized. As mentioned previously, we devised the concept of a virtual processor which in effect represents and executes an untranslated fragment. Virtual processors are integrated with neurons as the output of translation. Each virtual processor can then be run on a separate processor, maintaining the parallel nature of the translated system. The concept of virtual processors will be further clarified in the following sections when we demonstrate the techniques for creating virtual processors.

5.4.1.4 The Logic Network Representation

In devising strategies for expert system to neural network translation, we sought to create very general techniques with wide applicability to a number of expert system and neural network tools. For this reason, we needed a generic knowledge representation independent of any particular expert system tool or neural network simulation. This representation is the logic network.

The logic network is a complete symbolic, object-oriented, parallel representation of the rules and objects present in the expert system. From the logic network, a neural network can be constructed. The logic network is independent of a particular neural network paradigm.

The elements of the logic network correspond to elements of the rules. There are three major types of logic network elements: gates, predicates and virtual processors. The gates represent And, Or, Not, the Unknown function, and the Then indicator in a rule consequent. Predicates represent particular slot values or ranges of slot values for a particular slot for a particular object. Virtual processors represent the untranslatable fragments of a rule, including user-defined functions, user-interface procedures and the objects and slots used in the user-interface.

Gates and predicates input and output logical values of true, false and unknown. Each gate accepts inputs, computes the appropriate logical operation on the inputs, and produces an output. A predicate simply passes its input value through as output. Predicates may be used for input, output or intermediate functions in the logic network.

Virtual processors represent the untranslatable components of an expert system. Because we chose to translate as much of the logic of an expert system as possible to neural form, we decided to make virtual processors as simple (and unintelligent) as possible. A virtual processor cannot decide on its own under what conditions to execute its code fragment. Instead, it examines an associated predicate, called a trigger predicate. When the trigger is turned on, the virtual processor executes. To prevent the virtual processor from wastefully executing over and over, we added extra logic to turn the trigger predicate off. Extra logic was also required to prevent unnecessary evaluation of conjuncts and disjuncts in an antecedent. Consider for example, the antecedent

If A And B And C

If conjunct A is examined and found to be false, there is no need to evaluate conjuncts B and C. Expert systems typically employ this efficiency measure. Since we wanted our translated expert

system to perform identically, we accommodated this in the logic network.

Each rule in the expert system knowledge base is mapped into a logic network form. This form contains a Then gate whose input is connected to the output of the logic network representation of the antecedent and whose output is connected to the input of the logic network of the consequent. The logic network initially associated with a rule is considered a template. For each variable the rule ranges over, the template must be instantiated. Thus, every potential instantiation of a rule is a separate logic network. The details of the expert system to logic network translations and the method of simulating the logic network are given in Appendix B.

5.4.1.5 The Neural Network Representation

Because the logic network is a parallel representation, conversion to neural form is straightforward. Each element of the logic network has a corresponding neural configuration. For the current implementation of the translator, neurons are real-valued, employing the logistic function. Translation to neural form is accomplished by iterating through the set of logic networks, converting each logic network element to its neural configuration and establishing the positive and negative connections between the neurons. Virtual processors are tightly integrated with the resulting neural network. However, execution of the virtual processor is triggered by a positive output from a neuron rather than a predicate.

As an example of the translation from a logic network element to neural form, each AndGate in a logic network becomes a set of four neurons organized identically with the same weights. As a unit, the four neurons with their biases and connection weights, implement the logic of an AndGate with two inputs and one output. Further details of the neural configurations are given in Appendix B.

5.4.1.6 Additional Strategies

There were three additional translation strategies implemented for the AAA Planner application. These strategies translated Rating Summation, Input Vector Assembly, and Route Planning to a neural network form.

5.4.1.6.1 Rating Summation

The first step in the implementation is to identify virtual processors that sum a polygon's rating. These are always associated with a rule consequent. Next, the code identifies the value which is added to (or subtracted from) the Rating. The translator creates a connection from the trigger neuron which

normally would cause the summing virtual processor to execute. That connection is weighted with the value that would be added to the rating and attached to a summing neuron for each polygon.

5.4.1.6.2 Input Vector Assembler

The translator first identifies virtual processors which convert frame information to a neural input vector representation. The virtual processors are all placed in a list which the vector assembler can use, prior to running the neural network. This way the vector could be assembled ahead of time, which is more efficient.

5.4.1.6.3 Route Planning

The overall goal of the route planning translation is to plan a route through a series of polygons using a neural network. This network finds a minimum cost route to all polygons from a starting polygon. The input to the translation process is a list of polygons. With each Polygon is a list of the adjacent ones. The output is a neural network which calculates a minimum cost route. The inputs to the created neural network are the cost to traverse the polygon and the polygon in which to start the route. The output is the minimum cost route to all polygons from the starting one.

Translation is a two-step process. The first step is to convert the network of polygons (each polygon is 'attached' to its neighbors) to a network of gates. Each polygon maps to a certain set of gates based on the number of neighbors it has. A gate has multiple inputs and one output which computes some simple function of the inputs. The second step is to convert the gate network to a network of neurons. Each gate maps to a set of neurons which perform its function. More detail is given in Appendix B.

5.4.2 Architecture of the Translator

The translation process is separated into several well-defined components to simplify the activities and representations required in each process. This also permits greater flexibility to swap different versions of components in and out. The overall architecture of the prototype translation system is shown below.

The Translator is made up of both generic and application specific components. The generic component, the Generic Translator, converts any Kappa Knowledge base into a system of virtual processors and a neural network. The Generic Translator processes rule structures and produces a set of corresponding logic network templates. These templates are then instantiated, creating the logic networks and the expert system environment required to run the virtual processors. The Logic Net Translator converts the logic networks into a generic neural network representation. This

process is described in more detail in Appendix B. This system can be run as is or further translated with the AAA Planner Specialized Translator. The Specialized Translator converts all of the virtual processors into neural network components, thereby creating a neural network only application. To accomplish this, it creates an Input Vector Assembler which assembles input vectors for the neural network from the polygon information.

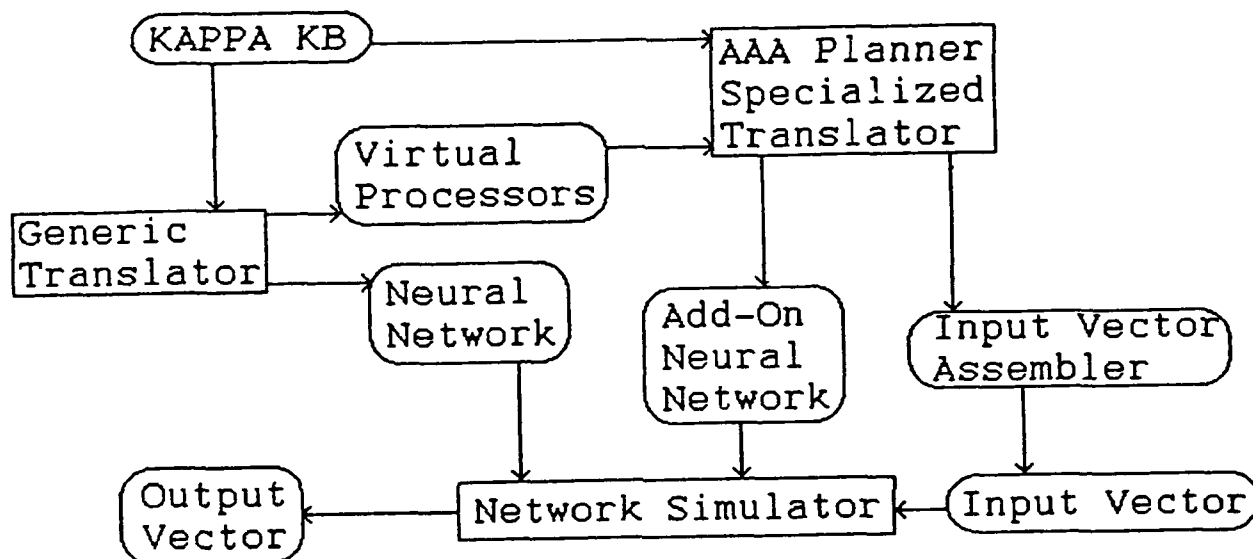


Figure 5-1. Translator Architecture

The developer interaction with the translator consists of starting the various processes. In the current translator, the developer interface consists of KAPPA-provided functionality to alter slot values and start methods and functions. The important functions are `Reset()`, `Translate()`, `RunSystem()`, `AddToNetwork(file)`, `AssembleInputVectors()`, and `RunCombinedNetworks`. `Reset()` clears out old translation objects. It should be called before `Translate()` to ensure a clean environment. `Translate()` is the generic translator. It translates `test.rul` (a KAPPA rulebase), `test.ins` (a KAPPA instances file), and `test.cla` (a KAPPA class file) into a neural network and associated virtual processors. `RunSystem()` runs the system of virtual processors and neural network created by `Translate()`. `AddToNetwork(file)` makes a new add-on network which sums polygon ratings and adds input Virtual Processors to a list used by `AssembleInputVectors()`. `AssembleInputVectors()` assembles neural network input vectors for polygons in file, `vecti`, where `i` varies from 1 to the number of polygons. `RunCombinedNetworks()` runs the polygon evaluation network on each input vector then runs the route planner neural network.

5.4.3 Translator Updating Results

The original translator was a combination of Kappa 1.1 and C code. The C code was linked into the Kappa executable at compile time. To bring the translator up to Kappa 1.3, the executable had to be remade. This occurred without difficulty. Some behavior differences were present in the interpretation of some Kappa statements. These led to errors which were tracked down and re-coded to avoid the error.

5.4.4 Translator Performance Results

The performance of the translator is impressive. One hundred percent of the rules and inferencing was translatable from the rule base and expert system building tool. Numeric, symbolic, boolean, and object reference slots are all translated. Using the simple slot translation strategies previously implemented, 60% to 75% of all slots in a typical expert system are translated and therefore eliminated. In the AAA Planner, the remaining untranslated slots (except for those that served as input to or output from the expert system) were translated using the additional translation techniques described above.

The AAA Planner had 34 rules and 19 slots. The translated system had no rules and no slots except to store input from or output to the user. This total translation was made possible by developing translations of the Rating summing and route planning.

It appears that most, if not all, knowledge bases are translatable. Of the six knowledge bases we have examined and translated so far, all would be translated to the above mentioned standards. The knowledge bases included three examined before the strategies were developed and three after. The great majority of slots were used exclusively in comparison or assignments involving explicit values. Very few functions or other procedural code was used.

A broad class of problems can be translated using our strategies. For example, recursive rules will translate correctly. Slot types of numeric, symbolic, boolean, and object reference are all translatable. The best increases in speed as a result of parallelization come about when an expert system uses a large number of rules and/or objects, as in the large number of polygons in the AAA Planner. The major prerequisites for translation are that rule ordering is unimportant and that little procedural code (function calls, methods, user interface, etc.) is used. While such systems can be translated and function correctly, not all of the translated system is neurons; virtual processors would also be required. To completely eliminate the virtual processors requires that few function calls are used and that those be mainly numeric calculations for which a neural approximation is acceptable (as in graph navigation algorithms or summing polygon factors).

There are two options when translating an expert system. One is to have the target network represent a single copy of the rules. This network must then be applied to each object the rules range over. The other is to create a copy for each of the rule set neurons for each object, so the data for each object is examined in parallel. Which choice is exercised depends on the number of objects, the number of neurons created, and the target hardware. For example, with the AAA Planner on a serial processor, we chose the first alternative.

The number of neurons created by the translation per object is given by the following formula, number of neurons per object = $K * \text{number of rules}$, where K is a constant which depends upon rule complexity. K is generally a number between 10 and 25. For the AAA Planner, 31 rules became 666 neurons per object leading to a value for K of about 21, which corresponds to the fact that the rules were relatively complex. Optimizations performed on the network could reduce the number of neurons at least by half.

The translator produces sparsely connected networks. In fact the number of connections is linear with the number of neurons instead of proportional to the square of the number of neurons. The AAA Planner translated into 666 neurons and 1447 weights and biases.

The success of the expert system to neural network translation offers great potential benefits to both the military and private sector. The importance of expert systems as independent intelligent entities has already been recognized. The translation and deployment of an expert system in neural network form and on a parallel processor will allow adaptable, fault tolerant behavior during unforeseen events and inputs, and permit high speed performance.

5.5 Phase II Software Architecture

The Phase II Software Architecture was designed to optimize real-time performance. This was accomplished by maximizing the amount of computation that could be performed ahead of time. In the architecture shown below, processes above the dotted line are performed ahead of time, off-line, and the processes below the dotted line occur in real-time, on-board.

Elevation grid data is processed through a Landform Identification Module, producing terrain polygons which correspond to hills, valleys, and other terrain features. This information must be combined with overlay data such as vegetation and soil types, to produce the set of polygons and associated information used by the AAA Planning Expert System. These are put through the translator to produce a neural network and input vector, which

represents the polygon information. On-board the helicopter, the neural network processes the input vector and produces a recommended path in a matter of seconds. During flight, update information can be written into the input vector and the route can be replanned, again, in a matter of seconds.

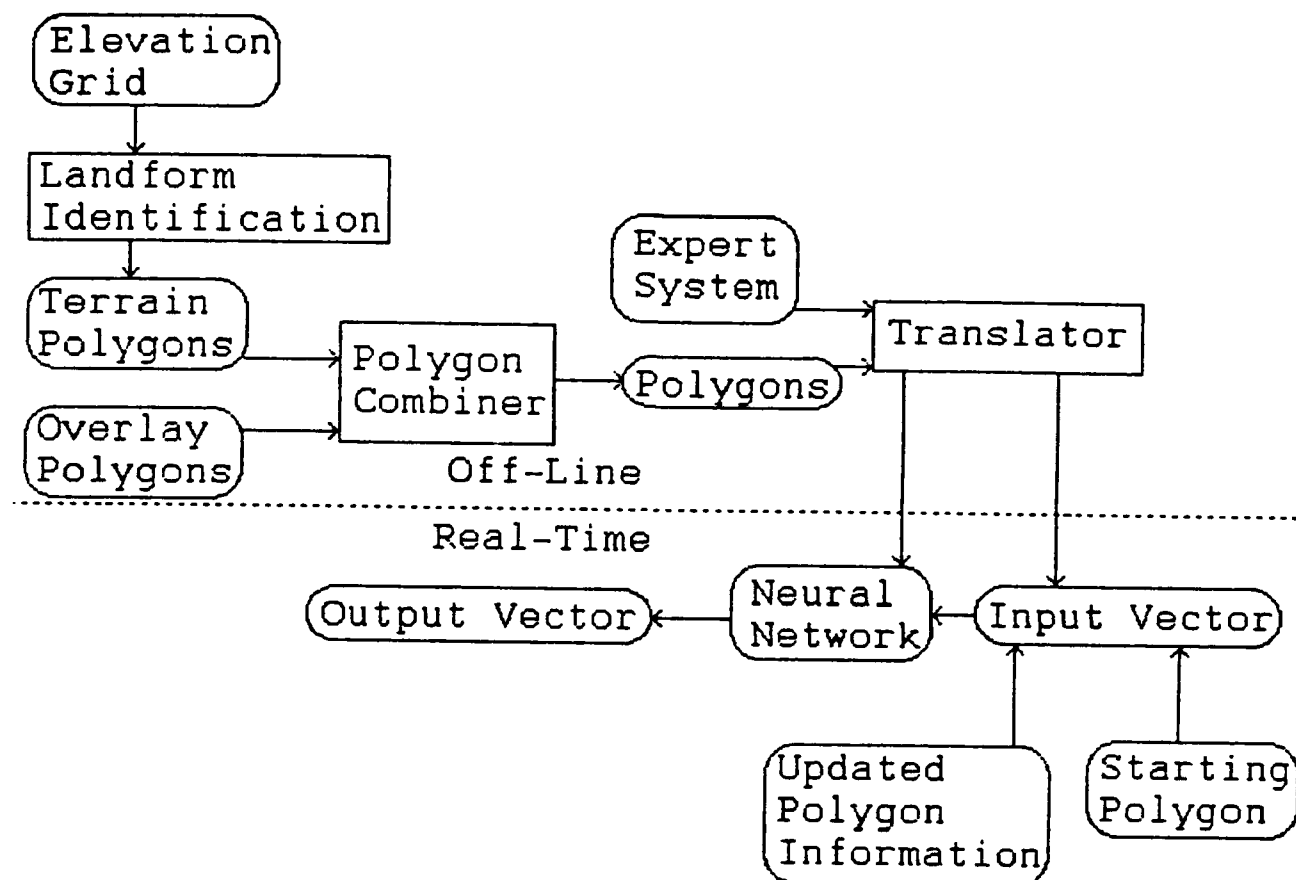


Figure 5-2. Phase II Software Architecture

This system is able to achieve performance impossible even with super-computer power by taking advantage of two facts. One is that the landform polygons do not change even if information about them does. This permits extensive preprocessing. The second is that inexpensive, specialized neural network processors exist which outperform super-computers by two to three orders of magnitude.

Onboard the helicopter, the user of the AAA Planner would add updated information on any polygon by editing its information, perhaps through a touch sensitive screen. The system could also automatically keep track of which polygon the helicopter is currently in through the use of Global Positioning System (GPS) data. This information would be translated and written into the

input vector. The network would process this vector, in effect planning optimum routes to every polygon from the current location. By specifying the desired destination, the optimal route could be displayed. An updated optimal route could be calculated and displayed every ten seconds or less, depending on the chosen hardware configuration as calculated in the next section.

5.6 Phase II Predicted Performance

The smallest terrain features of interest in the AAA problem have linear dimensions of roughly 1/2 mile or 1/4 square miles of area. This is roughly equivalent to 640,000 square meters or 64 grid points in a 100-meter grid. The following discussions follow rough, order-of-magnitude calculations.

A rule-base of the required complexity for polygon evaluation would translate into approximately 1000 neurons or less. At most each neuron would average three connections, so at most 3000 connections would need to be processed per polygon. A 100 mile by 100 mile square of interest should contain at most 40,000 polygons. Therefore, at most 120 million connections would need to be processed to evaluate each polygon. Route planning requires 15 neurons per polygon. Therefore, at most 1.8 million connections (40,000 polygons x 15 neurons per polygon x 3 connections per neuron) would need to be processed for the route planning portion of the network.

There are several neural network processing options with a range of costs and performances. One option is to use the PC compatible Loral P-board, built around Digital Signal Processing technology. One P-board costing about \$4000 today can process 41.3 million connections per second. The time to perform the required processing is then 3.0 seconds. One benefit of the P-board is that it is a general purpose floating point accelerator board which could be used for other purposes. Another benefit is that all hardware required is already commercially available, and dropping in price. Last July (1991) a single P-board was \$9000. If further processing power was required, an additional P-board could be purchased and used effectively.

On the other side of the price and performance scale is the CNAPS Server by Adaptive solutions, compatible with UNIX Ethernet systems. At \$55,000 it provides processing of 5 billion connections per second for a processing time of 0.024 seconds. The CNAPS is also a general purpose accelerator which could perform other tasks, although it only uses integer arithmetic.

Intel has developed an analog neural network processing chip called the 80170NX which can process 2 billion connections per second. The chip processes the connections for 64 neurons every 3 microseconds. To simultaneously represent the 1000 neurons could take as many as 20 chips. Each polygon could be processed every 3

micro seconds for a time of 0.12 seconds. The 600,000 neurons associated with the path planning portion of the network would require at most 0.06 seconds for a total time of 0.18 seconds. At \$540 per chip, the cost for the chips would be \$10,800. In addition, some simple A/D and D/A conversion boards would have to be developed using existing chips, or purchased outright.

Another alternative is to use a single 80170NX chip and perform the calculations of each of the 20 chips in the above option, in turn. The software driving the hardware would be more complex, the memory requirements would be greater, and the specialized hardware would have to be faster. But the total cost of the hardware would be less. To process the polygons would take 20 times as long, 2.4 seconds for all of them. The time to process the route planning neurons would still be 0.06 seconds as before, leading to a total time of 2.5 seconds.

The advantages of using either 80170NX approach is that the chip has spare processing power that we are not using. The processing time is independent of the number of connections per neuron if that number remains small. For example, if instead of three connections per neuron, we used 30 connections, the processing time with the same hardware would be nearly identical. This situation might develop if we introduced neural network training which increased the strength of previously zero-valued connections.

The translation algorithms can be easily changed to use different types of neurons. This allows us to delay the choice of hardware until late in the projects, if desired. Actually, all of the hardware options described above use real-valued, sigmoid function neurons, so no alteration of the translation would be necessary. Translation changes would only be required to take advantage of some as-yet-unknown option. If the final hardware choice required specialized hardware development, more time would be required.

6.0 TECHNICAL FEASIBILITY

Assuming that terrain polygons are available, the use of an intelligent neural network to process them has been shown to be feasible in Phase I. To determine the practicality requires looking at costs. Since it is impossible to guess the costs of hardware two years from now when the system is fielded, we will use today's hardware costs. It should be kept in mind that hardware costs are still dropping rapidly.

The Loral P-board option requires the \$4000 P-board and approximately \$2000 for a portable 486 machine. The total per system hardware cost is then \$6000.

The 20 80170NX chip option requires \$10,800 for the chips. The entire hardware requirements, including the portable 486 machine, could be produced for roughly \$15,000.

The single 80170NX chip option requires a single \$540 80170NX. The entire hardware requirements, including the portable 486 machine and chip interface hardware, could be produced for roughly \$6000.

One requirement for the final Phase II system which was not tested in Phase I was the conversion of the elevation grid data into terrain polygons. This was to avoid duplication of effort. We would use techniques developed in the other effort for our Phase II project, or develop our own techniques. One task which could be performed by a neural network is to group areas by curvature. Areas of positive curvature form valleys and ravines, while areas of negative curvature form hills and ridges. Curvature can be easily calculated at each grid point based on the data of the surrounding points. Smoothing with surrounding curvature values could be employed to prevent small bumps and depressions from becoming their own polygons. If these calculations are performed by a neural network, edge detection is simplified for the polygons by having a neural layer identify points with areas of opposite curvature on either side.

7.0 FUTURE RESEARCH AND DEVELOPMENT

There is an enormous amount of both research and development that remains to be performed. Phase I has laid a firm foundation by proving that the translation of the AAA Planner from an expert system to a neural network is possible. The AAA Planner and its associated neural network must be further developed to take advantage of this opportunity. Research should be performed to allow even greater advantage of that translation.

7.1 Future Development

Future development includes work on the AAA TDA and improvements to the translator.

7.1.1 Helicopter Avenues of Approach TDA Development

In Phase II, the complete AAA Planner using neural network hardware will be implemented, tested, installed and field-tested on-board a helicopter. A major part of this development will be on the originating expert system. Domain experts must be identified, interviewed and knowledge engineered. These experts will likely consist of personnel from Fort Rucker, Alabama. Most importantly, examples of helicopter path planning must be acquired. If possible, watching the process in an actual tactical setting would be ideal. The Phase I prototype was based on simplified knowledge. The Phase II system must be based on a much more accurate and comprehensive representation of the AAA planning knowledge.

The Phase I prototype assumed that terrain polygons had already been identified. An automatic way of converting the elevation grid data into terrain polygons must be developed. This may be provided by another SBIR project or may be developed as part of the Phase II effort. One approach, based on curvature, is described in Appendix A.

The Phase I prototype's interface was the minimal required to demonstrate the application. The final system's user interface must meet the rigid requirements of a high-pressure, tactical environment. Where the Phase II interface development was a negligible fraction of the development effort, the Phase II interface work will be a considerable portion of the project. Touch-sensitive screens should be investigated as one possible input/output device, especially for specifying current position or desired target area. The interface will go through several iterations in simulated cockpits with helicopter crews to ensure that it enhances, not detracts from, their abilities.

The Phase I effort could not include specialized neural network hardware. Several options have already been identified. These and others will be investigated in Phase II. Because of their low cost and the flexibility of the translator, several

different neural network processors may be acquired and evaluated in detail. The translator is an inherent component of the Phase II effort. Therefore, development will proceed on it in parallel with the AAA Planner, as described in the next section. The Phase II design is described in Appendix A.

7.1.2 Improvements to the Translator

To improve the network resulting from translation of the AAA Planner expert system, the translator itself must be improved so that it is sophisticated, practical, and tunable. Currently, the translation process emphasizes the correctness and parallelization of the final network. The neural network behaves exactly as the expert system does. This was necessary for a convincing proof of the feasibility of translating an expert system. To seize upon the Phase I research, the user should also be able to maximize adaptability, generalization, and fault-tolerance of the final neural network. Certainly the size or speed of the network on various platforms could also be optimized.

Both binary and real valued neurons with various output functions and various topology types must be allowable targets. Fuzzy Logic Networks must be supported. If a conversion to Conjunctive Normal Form (CNF) proves useful it should be implemented. This would allow a more neural-like network topology. A better set of slot conversion strategies should be implemented. Lists can be implemented with a set of neurons corresponding to the possible members of the list. Assignments to one slot from another slot could be handled with a more intelligent analysis. Comparisons between two slots in the antecedent should be supported instead of just between a slot and an explicit value. Certain functions and methods could be translated into a network framework, including certain classes of user-defined functions and methods. A more generic representation for numbers could be implemented involving a binary representation or by making use of input neurons.

Optimizations on the network can be performed which eliminate neurons and thereby reduce the number of layers. Multivariable rule translation was designed but needs to be implemented. The translation must occur rapidly and should be implemented in a highly portable language such as C.

Functions normally associated with Knowledge Base Management Systems can be performed on the rule-base. These include optimizations such as the removal of redundant rules, combining related rules, and checking for correctness, consistency or order independence. Optimizations can also be added in the logic network to neural network translation process. Certain logic network elements commonly appear together and these can be better translated as a group, instead of individually.

7.2 Future Research

Further research into adaptability, fault tolerance, generalization and other optimizations will benefit the AAA planner. These research ideas are discussed in the following sections.

7.2.1 Adaptability

Research must be performed to determine the best way to make the neural network easily adaptable. Two options should be investigated. The simplest option is to associate another neural network with the translated neural network. This neural network shares the translated network's input and output neurons, initially adding effectively nothing to the output. The translated network's weights are static. All training is performed with the associated network's weights. The advantage to this approach is that it is straightforward and flexible, easily allowing multiple network paradigms. The disadvantage is that extra neurons are required and the number of extra neurons and their topology must somehow be decided upon.

The second option is to change the translation process so that the resulting network is more adaptable. Currently, the structure of the network mirrors the structure of the rule base. The weights are set very high to guarantee logical behavior. For adaptable network behavior, the weights need to be relaxed considerably. The degree to which they should be relaxed depends on many factors and should be somewhat under a user's control. For example, the weights could be relaxed a minimal amount so that correct behavior is always guaranteed. They could be relaxed so that sometimes values are produced outside of allowable ranges. In the extreme case, they could be relaxed until values produced fall in incorrect ranges, thus producing predictable errors.

An alternative approach for adaptation is to alter the topology of the network. One strategy is to convert the logic network into Conjunctive Normal Form (CNF). This could then be converted into a neural network with a relatively small number of layers. Such a network would likely be more amenable to adaptation than the networks which are currently produced.

7.2.2 Fault Tolerance

Research also needs to be performed to allow greater fault tolerance of the produced network. Currently, a broken connection in the network corresponds to altering a particular expert system rule for a particular object. Alteration of a rule is certainly better than failure of the entire expert system when a wire in its processor breaks. However, this is not as fault tolerant as many neural networks. Fault tolerance can be achieved in three ways. The first way would be, in hardware, to triple each connection and

divide each weight by three. This would mean that the loss of any one connection would have no effect, but the loss of a single neuron would still cause erroneous, but not disastrous, results. If this process is taken a step further, each neuron could be tripled; this requires nine times the number of connections but no single loss would cause incorrect behavior. The disadvantage is, of course, the extra neurons and connections required. Although further research is required to determine a more exact number of extra neurons and connections for more fault tolerant behavior, tripling the current number would seem to be the minimal amount necessary.

As with adaptation, another solution exists to the problem of increasing fault tolerance - change the structure of the network produced by the translation. This might entail converting the logic network to CNF and then completely connecting adjacent layers. Research would be required to determine proper weights on the added connections. One solution is to use training to set those weights.

7.2.3 Generalization

The generalization capabilities of our translated networks have not been investigated. Currently, the network accepts inputs within 0.1 of 0, 1, and 0.5, corresponding to true, false, and unknown, and produces outputs in these same allowed ranges. Intermediate values would tend to be interpreted as falling in the closest allowed range, which may or may not be the best interpretation. Research should be performed to take advantage of values outside the allowed ranges. Investigating Fuzzy Logic (FL) would be appropriate, since currently the network implements FL for Ors, Ands, and Nots in the expert system for neural values near 0, 0.5, and 1. Additionally, FL systems are becoming common in many actual hardware controllers.

7.2.4 Neural Structuring

The above three topics (adaptability, fault tolerance, and generalization) can all be considered part of a more general topic, which is how to make the translated network behave more like a neural network and less like an expert system. This would also aid the process of designing a neural network for some task by writing an expert system and translating.

As mentioned previously, one way to make the network more neurally structured is to convert the logic network to CNF, then translate this logic network into a small number of layers. Many connections would need to be added to fully connect adjacent layers since the translation process creates many 0 weights. The weights of these added connections could be calculated or perhaps set with training.

In performing the research, two opposing concepts must be balanced. We must allow for specific neural paradigms for current translation, while staying general to easily take advantage of future discoveries of new paradigms.

7.2.5 Optimizations

There are optimizations associated with each of the three primary software modules in the translator. Type 1 optimizations are performed on logic network templates during the rules to logic network templates translation. Type 2 optimizations occur during template instantiation on the instantiated logic networks. Type 3 optimizations are performed on the neural networks during the logic network to neural network translation process. The various optimization strategies are outlined below.

Type 1 Optimizations

Type 1 optimizations actually may occur on the generic rules, during the translation from rules to logic network process, or on the resulting logic network. Optimizations can be performed based on the type of inference desired in the final system. For example, there may be different answers to the following questions: Is it important to fire all true rules or only true rules whose antecedent contains slot names on an agenda? Must virtual processors only execute once and in the order dictated by the logic or can they also execute at once?

There are various optimizations normally associated with Knowledge Base Management Systems (KBMSs). These include combining and eliminating rules. Certain gates often appear together. These could be combined into new types of gates which would improve the final neural representation. Rules or the logic network could be translated into Conjunctive Normal Form (CNF) since all logical systems have an equivalent CNF representation. This transformation combined with new types of gates could lead to a network with very few layers.

Type 2 Optimizations

Type 2 optimizations are performed during the logic network instantiation process and on the instantiated logic networks. The optimizations are the same as Type 1 but are only possible after the logic network has been instantiated. This would primarily include rules whose antecedent (or consequent) ranges over a different set of objects than the consequent (or antecedent).

Type 3 Optimizations

Type 3 optimizations occur during logic network to neural network translation and on the resulting generic neural network. These optimizations include eliminating neurons to reduce the

number of layers; adding neurons to increase the fault tolerance and robustness of the network; changing network structure or weights for increased adaptability, generalization, speed, or logical correctness; and increasing the Fuzzy Logic properties of the network.

In performing the optimizations there are many considerations. One is that the above mentioned goals are often contradictory. The type of neurons must be considered. The target neural paradigm is important and several must be supported. Finally, whether a 2- or 3-valued logic is being used has a dramatic effect. Whereas most gates require 4 neurons with a 3-valued logic, only one would be required for a 2-valued one.

8.0 CONCLUSIONS

The Phase I results absolutely demonstrated the feasibility of developing a prototype intelligent Tactical Decision Aids (TDA) neural network system through the use of an expert system to neural network translator. We implemented the prototype solution with expert system technology. We then completely translated that system to a neural network representation using our automatic translator. The network's performance was evaluated using test cases and a neural network simulator. Additionally, the performance of the completed Phase II system was estimated. The time to completely plan a route in a 100 mile by 100 mile area was conservatively estimated to be 3 seconds with a \$4000 specialized board and a 486-based PC. These results clear the way for development of a complete system in Phase II.

There is an enormous amount of both research and development that remains to be performed. Phase I has laid a firm foundation by proving that the translation of the AAA Planner from an expert system to a neural network is possible. The AAA Planner and its associated neural network must be further developed to take advantage of this opportunity. Research should be performed to allow even greater advantage of that translation.

There are many potential applications of this research. The most direct application will be for Airborne Avenues of Approach Planning. This technology could be applied to many route planning problems including Avenues of Approach for ground units. The techniques used here would benefit any application involving the intelligent evaluation of a large number of items. Examples include threat evaluation and image understanding. Finally, the generic translator can be used to allow almost any expert system to be translated to a neural network and thus receive the combined benefits of each technology.

APPENDIX A PHASE II DESIGN

The Phase II Software was designed to optimize real-time performance. This was accomplished by maximizing the amount of computation that could be performed ahead of time. In the architecture shown below, processes above the dotted line are performed ahead of time, off-line, and the processes below the dotted line occur in real-time, onboard.

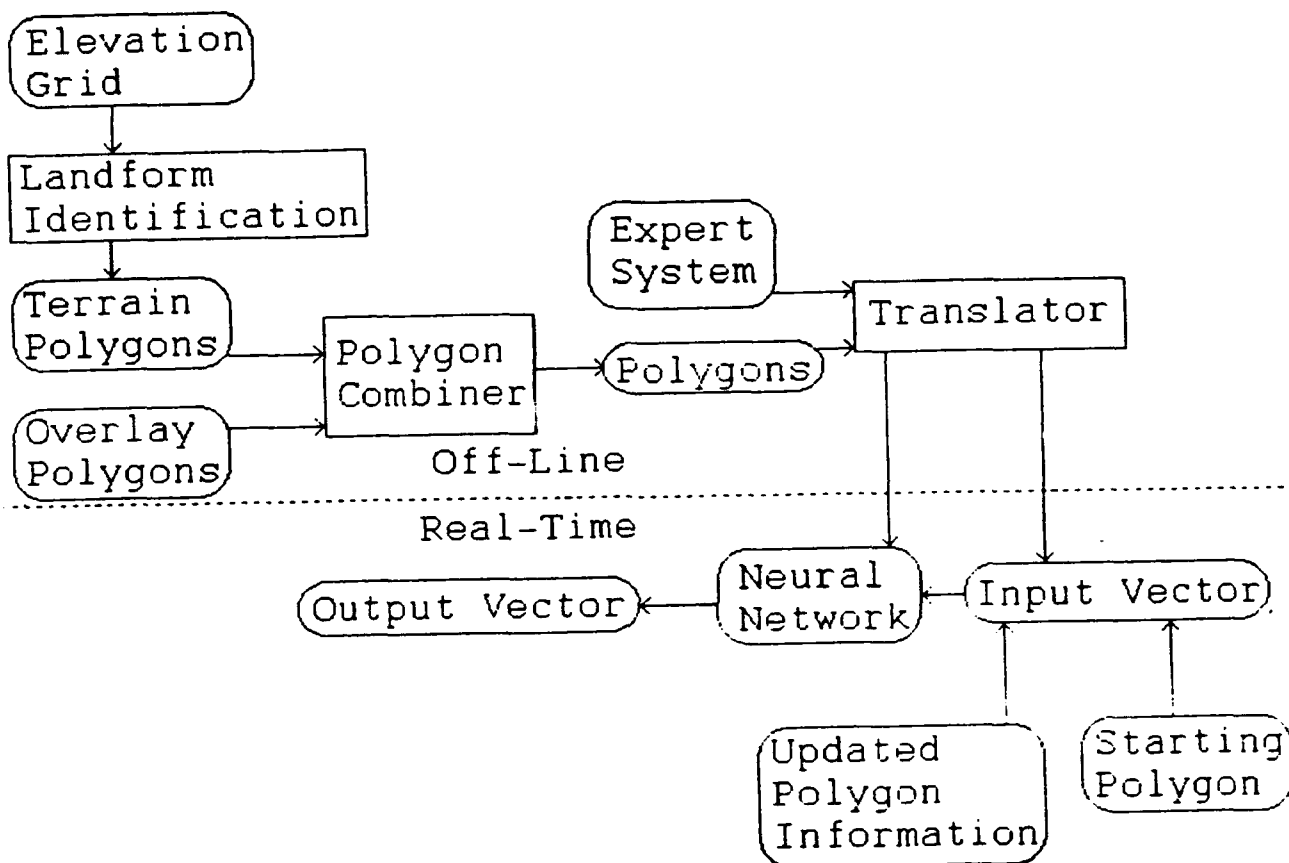


Figure A-1. Phase II Software Architecture

Elevation grid data is processed through a Landform Identification Module, producing terrain polygons which correspond to hills, valleys, and other terrain features.

The input to the landform identification layer is the gridpoint elevations. Each gridpoint corresponds to one input neuron. The first layer, curvature calculations, contains a neuron for each grid point. It receives the elevation of the gridpoint it corresponds to, and its neighboring gridpoints, and calculates an average curvature.

A Smoothing Layer neuron receives, from the previous layer, the calculated curvature from one or a small group of neurons. It

also receives input from other neurons in the Smoothing layer. It sums these inputs, and outputs the average to its neighbors. In this way large areas of same signed curvature develop a momentum of activation signals which tend to swamp small areas of opposite curvature. Two large areas of different curvature cancel each other at their border, accentuating that border.

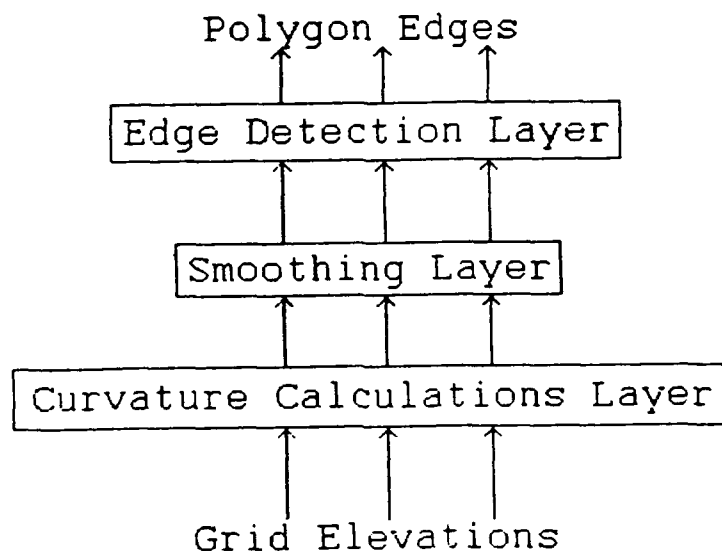


Figure A-2. Landform Identification Neural Network Layers

The smoothed curvature calculations are output to the Edge Detection Layer. A single Edge Detection Neuron receives the smoothed curvature results from a small group of adjacent neurons. Edge Detection neurons are activated when their inputs are different in sign so that neurons corresponding to a border between different areas of curvature are activated. These detected edges are output to form the terrain polygons.

The terrain polygons must be combined with overlay data such as vegetation, soil types, etc. to produce the set of polygons and associated information used by the AAA Planning Expert System. This combination is best performed by a GIS as shown in Figure A-3.

During the Phase II effort, SHAI will develop the AAA Planning Expert System as shown in Figure A-4. This knowledge base will be alterable by Army personnel, if required. This knowledge base will be translated initially into a template Neural Network. This template can be instantiated for each polygon or run on each polygon separately.

The altered terrain polygons and the neural network template are put through a second translator to produce a neural network and input vector, which represents the polygon information, as shown in

Figure A-5. The network can be broken into the network which evaluates each polygon and the network which plans the route.

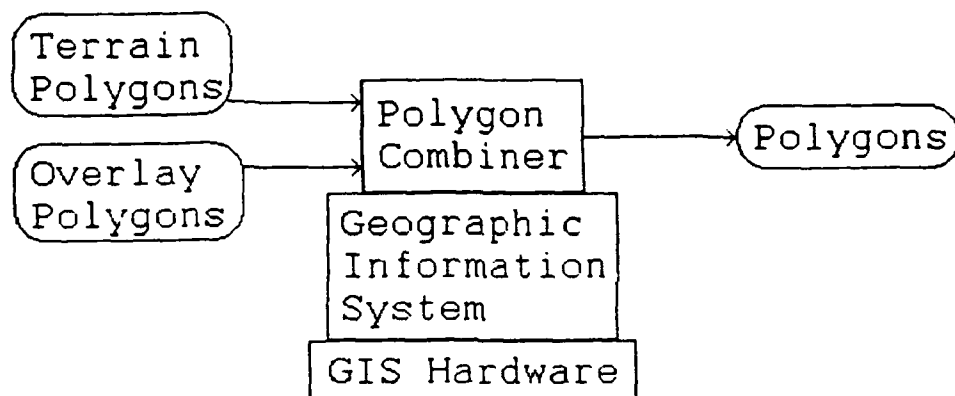


Figure A-3, Polygon Combiner

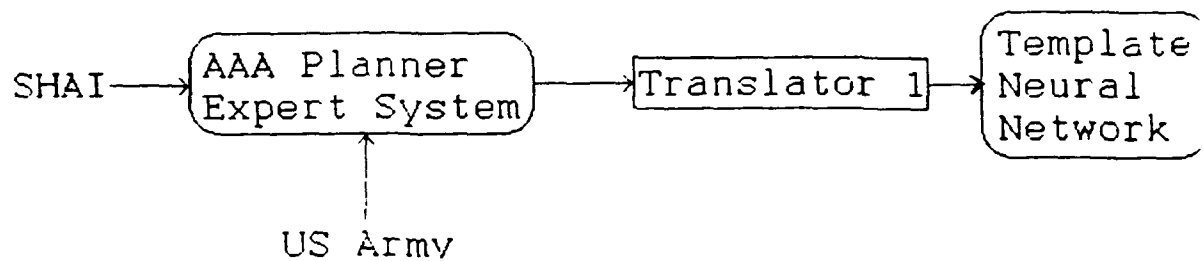


Figure A-4, AAA Planning Expert System Development

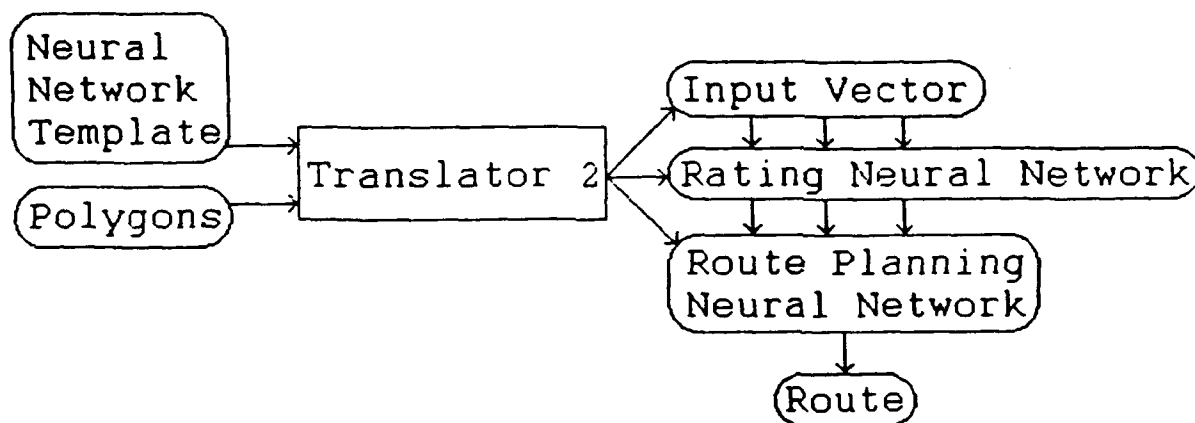


Figure A-5. Expert System to Neural Network Translation

The polygon neural network will be a faithful representation of the evaluation neural network, perhaps altered with training, if that is considered beneficial. The route planning neural network is developed from a routing algorithm and the polygon structure. The Phase I routing neural network ignored distances and fuel consumption when navigating polygons. The Phase II system will take these into account. The cost of traversing a polygon can be dependent on the direction and distance of that particular traversal. The Routing neural network will calculate the least cost route to every polygon from the current polygon.

The helicopter crew would access the AAA Planner through a user-friendly interface. Figure A-6 shows the configuration. On-board the helicopter, the user of the AAA Planner would add updated information on any polygon by editing its information, perhaps through a touch sensitive screen. The desired polygon is selected by touching it. The attribute of the polygon could also be selected for updating by touch. In the case of a small number of possible values for the selected attribute, the value itself could be touch-selected from a menu. Otherwise, the value would be keyed in. Updated information may also be available directly from the helicopter's avionics. For example, the system could automatically keep track of which polygon the helicopter is currently in through the use of GPS data.

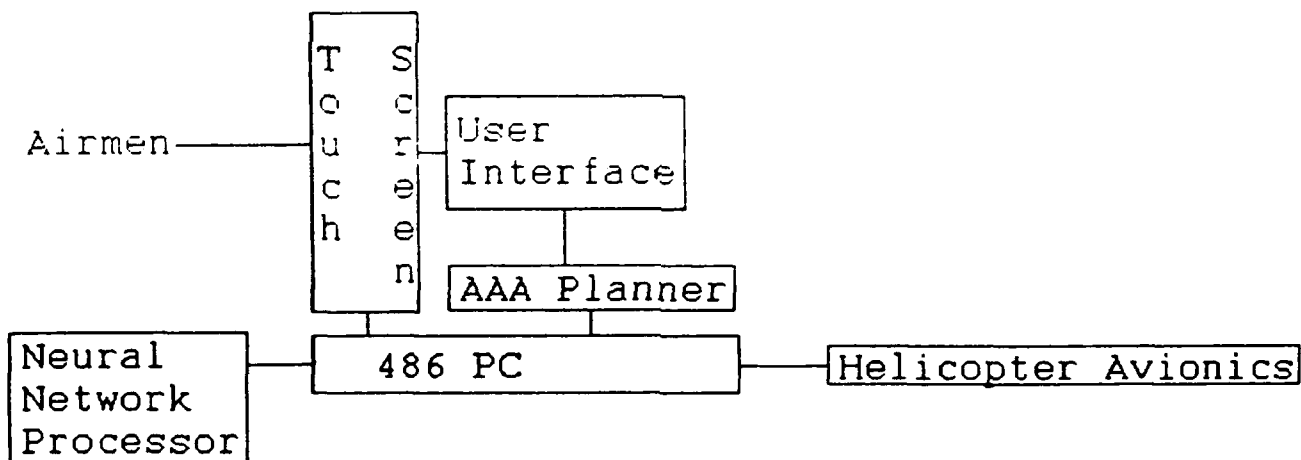


Figure A-6. AAA Planner Configuration

The updated information for polygons would be translated and written into the input vector. The neural network would process this vector, in effect planning optimal routes to every polygon from the current location. By specifying the desired destination, again by touch-selection, the optimal route could be displayed. An updated optimal route could be calculated and displayed every ten seconds or less, depending on the chosen hardware configuration.

The system is able to achieve this unprecedented level of performance by taking advantage of two facts. One is that the landform polygons do not change even if information about them does. A given polygon will always be the same shape, be in the same location, and have the same neighbors. This permits extensive, off-line preprocessing. The second is that inexpensive, specialized neural network processors exist which outperform even supercomputers by more than two orders of magnitude.

APPENDIX B COMPLETE TRANSLATION STRATEGIES

B.1 Rule to Logic Network Translation

B.1.1 Logic Network Description

Logic Networks consist of gates, predicates and virtual processors. Gates and predicates input and output logical values of false, true, and unknown (F, T, and U).

Gates take inputs from other gates or predicates. Each gate has one output connected to one or more gates or predicates. The types of gates are ThenGate, UnknownGate, AndGate, OrGate, PredicateAssignmentGate, and NotGate. The functions of the gates are described in the following truth tables in Figure B-1.

<u>ThenGate</u> (1 input)						<u>UnknownGate</u> (1 input)					
Input			Output			Input			Output		
T			T			T			F		
U			U			U			T		
F			U			F			F		

<u>AndGate</u> (2 inputs)						<u>OrGate</u> (2 inputs)					
			Second Input						Second Input		
			T	U	F				T	U	F
			Outputs						Outputs		
F	I	T	T	U	F	F	I	T	T	T	T
i	n					i	n				
r	p	U	U	U	F	r	p	U	T	U	U
s	u					s	u				
t	t	F	F	F	F	t	t	F	T	U	F

<u>PredicateAssignmentGate</u> (any # of + or - inputs)						<u>NotGate</u> (1 input)					
			Negative Input			Input			Output		
			T	U	F	T			F		
			Outputs			U			U		
P	I	T	UD	T	T	F			T		
o	n										
s	p	U	F	U	U						
i	u										
t	t	F	F	U	U						
i											
v			UD = Undecided								
e											

Figure B-1. Gate Truth Tables

The PredicateAssignmentGate actually has a number of inputs divided into two sets - positive and negative inputs. If any positive input is true then the gate outputs true, if any negative input is true, the gate outputs a false. If both a positive and negative input are "on", then the output is undefined. The translation is such that this never occurs.

Predicates come in three classes. An input predicate's value is set by a virtual processor. Trigger predicates and hidden predicates behave identically to each other. The output of a predicate of either type is the same as its input. The value of a trigger predicate is monitored by a virtual processor. When that predicate's value is True, the virtual processor monitoring it executes.

Virtual Processors can read and set predicates' output. They consist of procedural statements and user defined methods and functions. Execution begins when an associated (trigger) predicate is true. The final statements set input predicates.

The logic network can be simulated in the following way:

- Initialize all predicates to unknown.
- Loop through the following actions until no virtual processors execute.
 - Simulate the network consisting of the interconnected predicates and gates starting with the input predicates and working upward toward the trigger predicates.
 - For each trigger predicate which is true, execute the associated virtual processor.

The graphical logic network representations of typical rules are given in the following figures. Predicates are represented by circles, gates are labeled rectangles, and virtual processors are elongated hexagons. Actual inputs to gates are indicated by solid lines, virtual connections are indicated by dashed lines.

Simple Antecedent

The logic network representation of a simple antecedent of a rule is given below. This representation corresponds to an antecedent which contains untranslatable items such as function calls or input/output slots. A typical example is the antecedent, If Ravine?(p:Depth,p:Width).

The primary element of this logic network is the virtual processor. The virtual processor is virtually connected to three predicates, the trigger predicate (left), the value predicate (above) and the continue predicate (below). All predicate values are initially set to unknown. The value of the trigger predicate

depends upon the output of the AndGate, whose value is dependent upon an implicit input of true and the output of the UnknownGate. The value of the UnknownGate is obtained from the continue predicate. If the virtual processor has not executed, the continue predicate will have a value of unknown. This, when input to the UnknownGate, produces true. This value and the implicit true value are input to the AndGate to produce a true value, which becomes the value of the trigger predicate. The virtual processor examines its trigger predicate to determine if it should execute. If execution is triggered, the virtual processor sets its value predicate to the result of the execution, and its continue predicate to true. The continue predicate indicates that the virtual processor has completed execution. The next time through the simulation, the trigger predicate will be false, thus preventing multiple executions of the virtual processor.

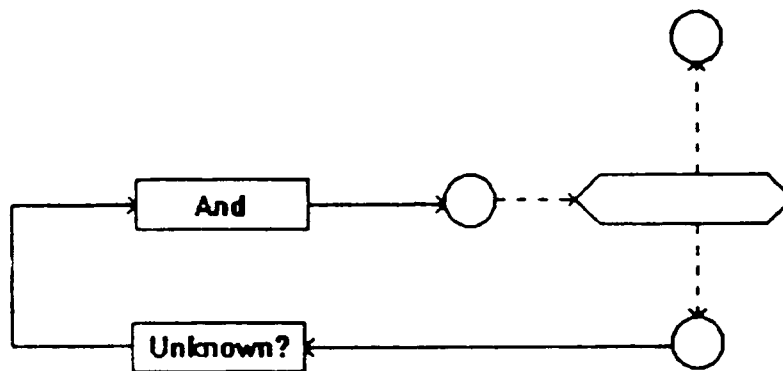


Figure B-2. Simple Antecedent Logic Network

Simple Consequent

The simple consequent is very similar to the simple antecedent, with two exceptions. It contains a ThenGate, whose input would come from the value predicate of the simple antecedent. It also does not have a value predicate.

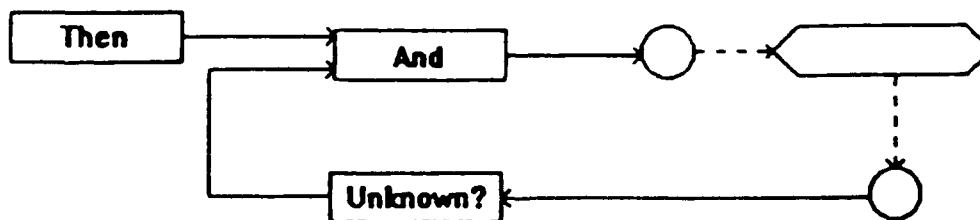


Figure B-3. Simple Consequent Logic Network

Complex And Antecedent

Pictured below is the logic network corresponding to an antecedent containing an And connective. It is similar to the structure for the simple antecedent, except that it has an additional AndGate. The circle labeled "Left Side", could be replaced with the logic network structure for a simple antecedent, or any of the complex antecedents. This structure, for example, could correspond to the antecedent,

If Ravine? (p:Depth,p:Width) And Long(p:Length,p:Width)
The virtual processor in the drawing below would represent the right side of the And, Long(P:Length,p:Width).

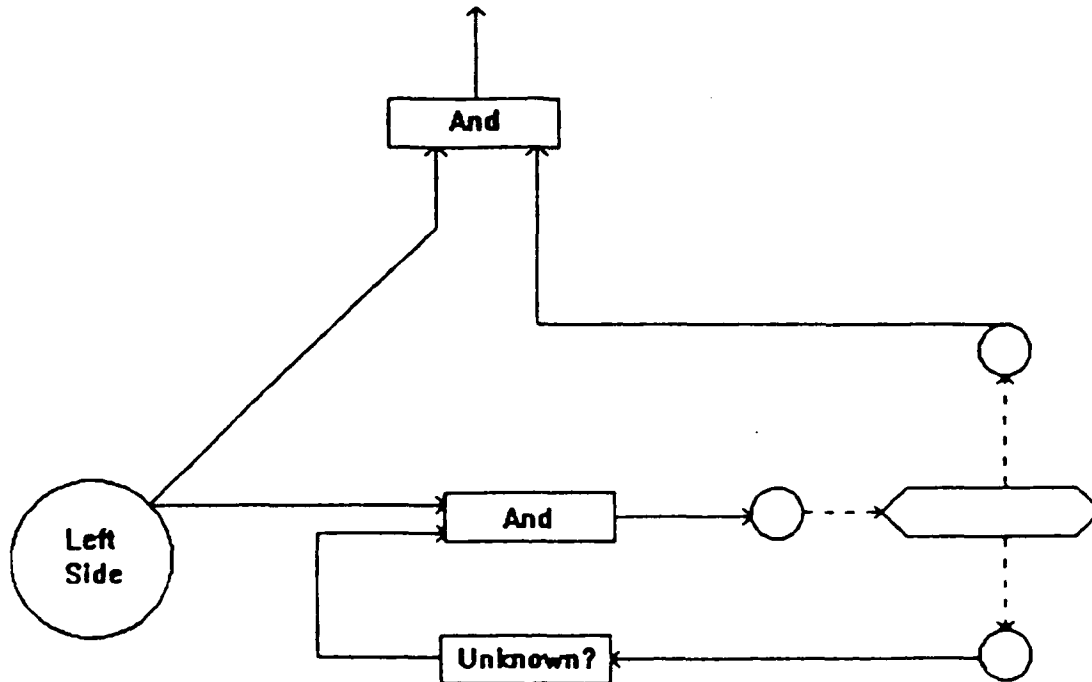


Figure B-4. Complex And Antecedent Logic Network

Complex Or Antecedent

The logic network structure for an antecedent containing the Or connective is very similar to that of the complex And antecedent. Note the added presence of a NotGate and an OrGate rather than an AndGate.

B.1.2 Rule to Logic Network Translation Strategies

The translation from a rule representation to the logic network representation occurs in two steps. The first is to translate the rule base into templates. Each template is then instantiated (copied) for each object that the template's originating rules range over.

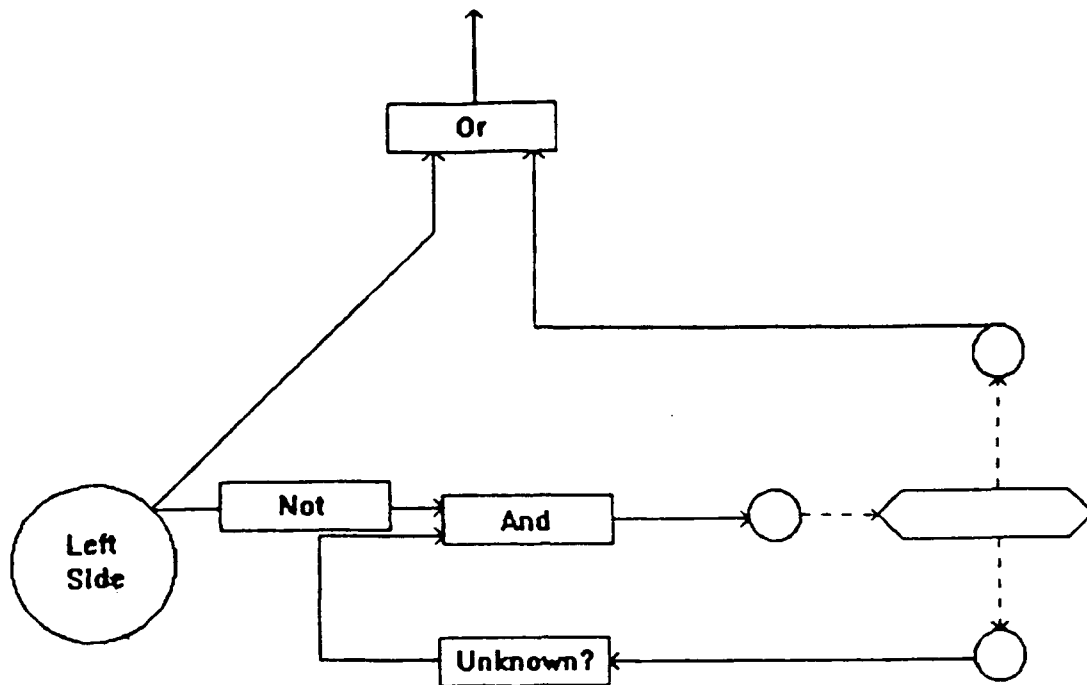


Figure B-5. Complex Or Antecedent Logic Network

In translating rules to logic network templates, each rule translates into a ThenGate whose input is connected from the translated antecedent and whose output is connected to the translated consequent.

Slot value comparisons are generally translated into predicates. If a rule's consequent sets a slot that appears in another translated rule's antecedent as part of a comparison operation, a connection is required from the first rule's consequent logic network to the predicate which represents the slot value comparison. This process has the effect of combining logic networks into larger ones so that any one self-contained logic network is the result of several inter-related rules. Additionally, logic networks are combined when the same slot value comparison appears in the antecedent of separate rules. Both appearances are translated into exactly the same predicate.

Slots were translated if every appearance of the slot name appeared in a comparison or assignment to an explicit value. Therefore, slots were not translated if they appeared anywhere in a function call or were compared to or assigned to or from another slot. These restrictions are not required in a more extensive implementation. A more extensive implementation could translate slot to slot comparisons in a number of ways. One way is to keep track of the possible values a slot can store based on assignments to it. Another method for use with numeric slots is to store the slot in an input real-valued neuron or in a binary representation. Either would allow comparisons using neuron-implemented comparison functions (<, =, etc.).

B.2 Logic Network to Neural Network Translation

B.2.1 Neural Network Target Representation

There are two types of elements in a neural network: neurons and virtual processors. Virtual processors behave identically to their logic network counterparts, except that they read and set the outputs of neurons instead of predicates. There are three types of neurons - input, output and hidden. Output neurons correspond to trigger predicates. A virtual processor is attached to each one. Each input neuron is set by a virtual processor. Output and hidden neurons behave identically in that each computes its value by performing its output function on the weighted sum of its inputs. Input neuron outputs are not computed but simply set by virtual processors.

Either binary or real-valued neurons could have been used in the network. Because we are translating a three-valued logic, 2 binary neurons are required to represent one truth value. If real-valued neurons are used, either two or one unit can be used. We developed translation strategies for both neuron types.

In our translator, the neurons are real-valued with a logistic activation function. Their outputs must be between 0 and 1. Furthermore, the network is designed such that the outputs will fall into three ranges corresponding to F, U, and T, respectively. The three ranges are 0 - 0.1, 0.4 - 0.6, and 0.9 - 1.

To simulate the network of neurons and virtual processors the following actions are taken.

- All neurons are initialized to 0.5 output (unknown)
- The following two steps are repeated until no virtual processors execute
 - Simulate the network
 - Run all the applicable virtual processors

B.2.2 Logic Network to Neural Network Translation

Real-valued neurons have three ranges defined to correspond to the three possible truth values. These ranges are defined in terms of constants. The translation strategies from logic network to neural network are also defined in terms of these constants to facilitate experimentation in future work. The constants are defined below.

Truth Value	Range
False	0 -> a
Unknown	b -> c
True	d -> 1

In order to facilitate logic network to neural network translation it is convenient to keep an ordered relationship among the binary sums of truth values. For example, it is convenient for the sum of any true and unknown value to be less than the sum of any two true values. This constrains the values of the constants that define the ranges. These constraints are shown below.

Constraint	Meaning
$1 + c < 2d$	true + unknown < true + true
$2c < 1 + b$	unknown + unknown < true + unknown
$c + a < 2b$	unknown + false < unknown + unknown
$2a < b$	false + false < unknown + false

A set of values which we used which satisfy these constraints are $a = .1$, $b = .4$, $c = .6$, and $d = .9$.

Each predicate translates into one neuron. Each virtual processor that monitors a trigger predicate will instead monitor an associated neuron. Each virtual processor which sets a predicate, instead sets the corresponding neuron.

Each gate is translated into a group of neurons which computes the appropriate output. The neural translation for each gate is shown below. The weights and biases are functions of the range constants as well as of two other constants, m and M . These constants are used to force logical values toward 0, 0.5 and 1. They currently are defined to produce large weights and biases ($m = 10$, $M = 50$), but should be redefined as smaller numbers to facilitate adaptation. This would be one subject of future research.

The neural configurations corresponding to the various logic network gates are given below. The circles indicate neurons and the solid lines, connections. Charts are given showing the biases and weights for each configuration.

And: Real Neuron Configuration

And Gate Translation

Weights			Biases	
From	To	Weight Equation	Neuron	Bias
N5	N2	M	N1	$-m/2$
N5	N3	M	N2	$-M(a+b)/2$
N6	N3	M	N3	$-M(1+c+2d)/2$
N6	N4	M	N4	$-M(a+b)/2$
N2	N1	$m/4$		
N3	N1	m		
M4	N1	$m/4$		

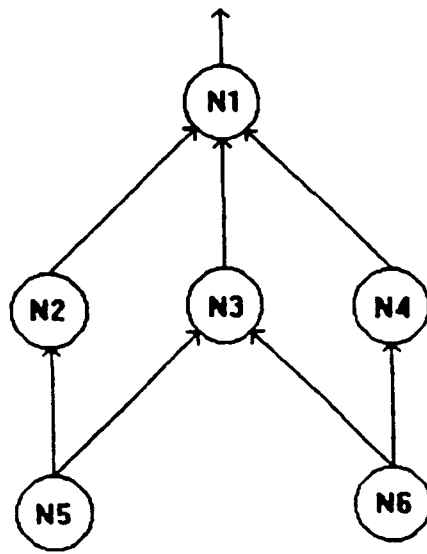


Figure B-6. Real And Configuration

Or: Real Neuron Configuration

Or Gate Translation
Weights

Biases

From	To	Weight	Equation	Neuron	Bias
N5	N2	M		N1	$-m/2$
N5	N3	M		N2	$-M(c+d)/2$
N6	N3	M		N3	$-M(2a+b)/2$
N6	N4	M		N4	$-M(c+d)/2$
N2	N1	m			
N3	N1	$m/2$			
N4	N1	m			

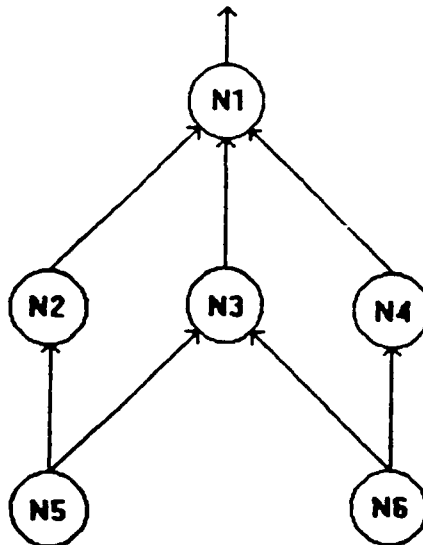


Figure B-7. Real Or Configuration

Not: Real Neuron Configuration



Figure B-8. Real Not Configuration

Then: Real Neuron Configuration



Figure B-9. Real Then Configuration

Unknown: Real Neuron Configuration

Unknown? Gate Translation
Weights

Biases

From	To	Weight Equation	Neuron	Bias
N4	N2	$-2m$	N1	$-1.5m$
N4	N3	$2m$	N2	$m(c+d)$
N2	N1	m	N3	$-m(a + b)$
N3	N1	m		

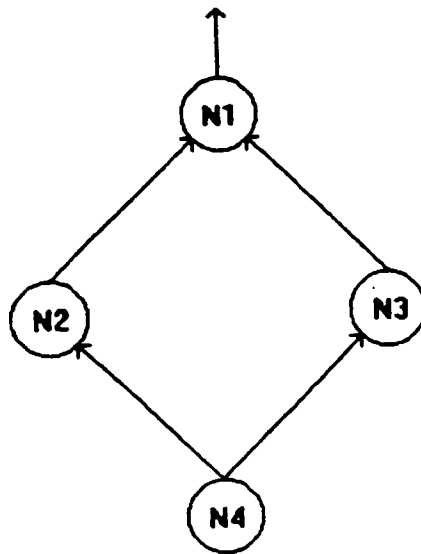


Figure B-10. Real Unknown Configuration

The above weights and biases assume the logistic activation function, $O(s,t) = 1/(1 + \exp(-(s + t)))$, where s = weighted sum of the neuron's inputs and t = threshold, or bias, a constant adjusted during learning. If a different activation function were used, the weights of the translated system would need to change, but the translation concepts would be the same.

Translation strategies were also developed for binary neurons. With binary neurons, two neurons are required to represent one predicate. One neuron represents Known/Unknown the other represents True/False. The allowed values for a set of two neurons is shown below.

T/F	K/U	
1	1	True
0	1	False
0	0	Unknown
1	0	Unallowed

The advantage of using binary neurons over real-valued ones is that binary neurons are faster than real-valued (integer vs real-number arithmetic) especially with feedback connections. The disadvantage is that with this scheme of values, the network could be harder to train than the real-valued counterpart.

And: Binary Neuron Configuration

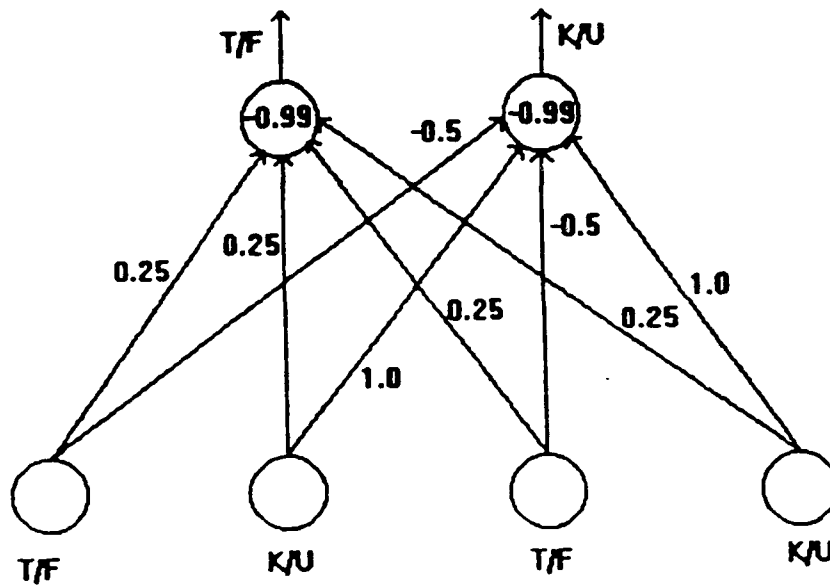


Figure B-11. Binary And Configuration

Or: Binary Neuron Configuration

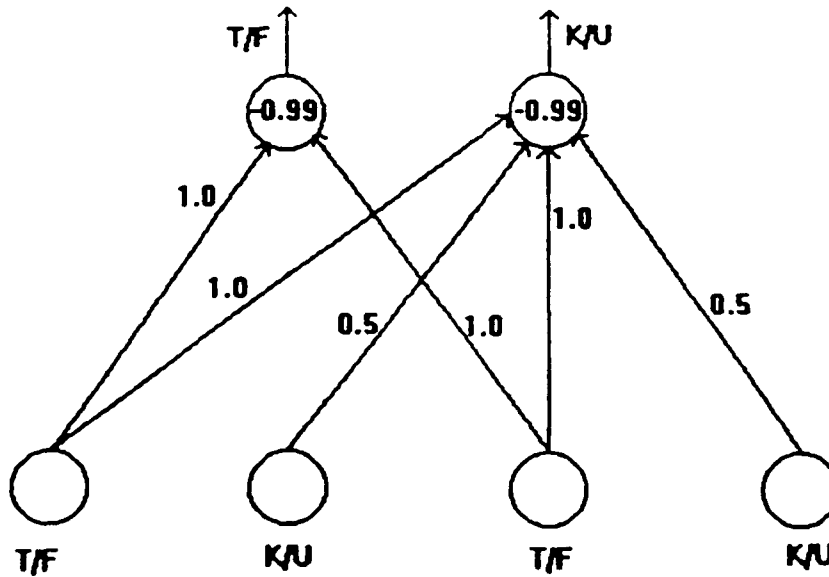


Figure B-12. Binary Or Configuration

Not: Binary Neuron Configuration

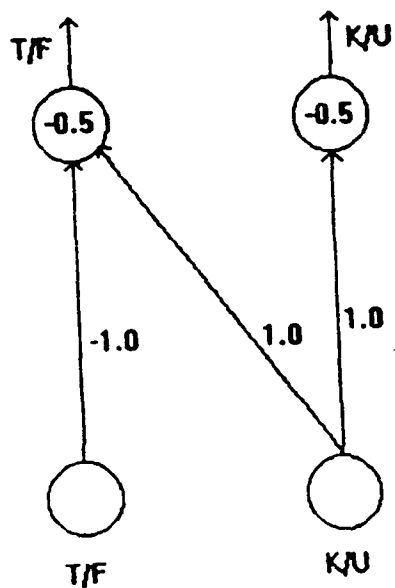


Figure B-13. Binary Not Configuration

Then: Binary Neuron Configuration

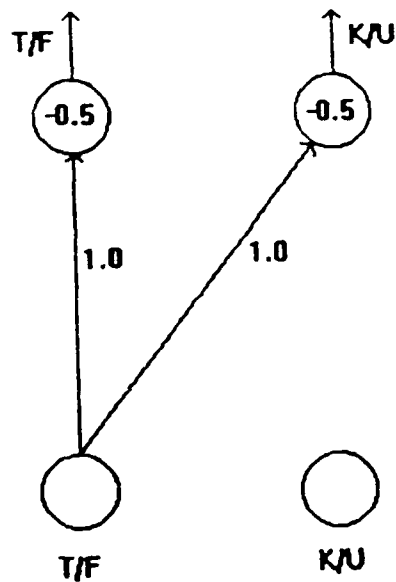


Figure B-14. Binary Then Configuration

B.3 Additional Translation Strategies

B.3.1 Rating Summation

The first step is to identify virtual processors that sum a polygon's rating. These are always associated with a rule consequent. Next, we identify the value which is added to (or subtracted from) the Rating. The translator then finds the neuron which triggers activation of the virtual processor as shown in figure B-15. That triggering association is broken and a connection from the trigger neuron is created. That connection is weighted with the value that would be added to the rating and attached to a summing neuron for each polygon as shown in Figure B-16.

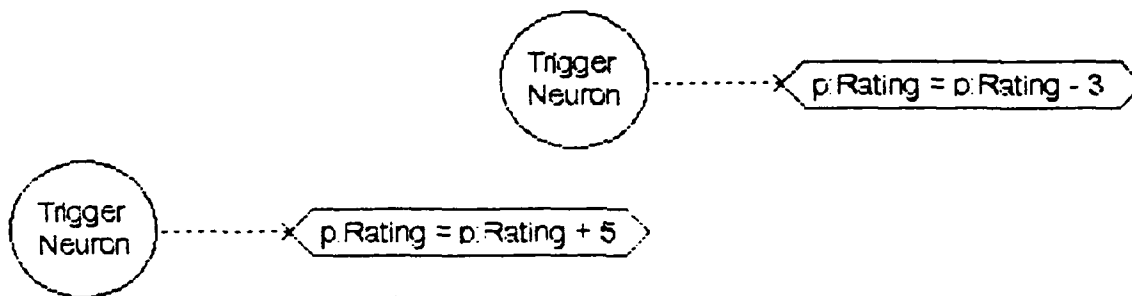


Figure B-15, Before Translation

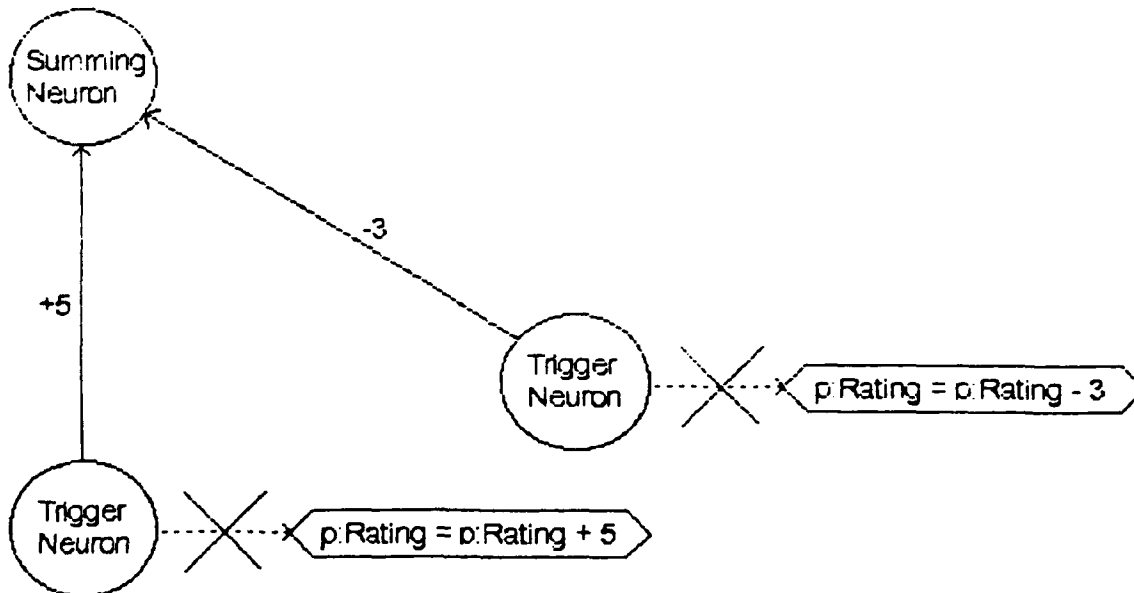


Figure B-16. After Translation

The summing Neuron simply sums its input and passes the rating on to the next stage.

B.3.2 Route Planning

The overall goal of the route planning translation is to plan a route through a series of polygons using a neural network. This network finds a minimum cost route to all polygons from a starting polygon. The input to the translation process is a list of polygons. With each polygon is a list of the adjacent ones. The output is a neural network which calculates a minimum cost route. The inputs to the created neural network are the cost to traverse the polygon and the polygon in which to start the route. The output is the minimum cost route to all polygons from the starting one.

The translation is a two-step process. The first step is to convert the network of polygons (each polygon is 'attached' to its neighbors) to a network of gates. Each polygon maps to a certain set of gates based on the number of neighbors it has. A gate has multiple inputs and one output which computes some simple function of the inputs. The second step is to convert the gate network to a network of neurons. Each gate maps to a set of neurons which perform its function. These two steps are described in more detail below.

In the first step, each polygon is converted into a network of the following components:

- 1 Starting Polygon (Network Input) node which can be set to 0 (this is the starting polygon) or infinity (this is not the starting polygon).

- 1 Local Cost (Network Input) node which can be any positive number. This input gives the cost to traverse the polygon.

- N Which Neighbor (Network Output) nodes whose names are the name of the corresponding polygon concatenated with the names of the corresponding neighbor polygons. N is the number of neighbors. This network output gives which neighbor the minimum path comes from.

- N Input Neighbor Cost lines (N is different for each polygon) corresponding to the polygon's N Neighbors. These connections originate with the networks of the neighboring polygons.

- 1 Min Gate with N+1 inputs. The inputs are the N Input Neighbor Cost lines and the Starting Polygon Network Input Node. A Min Gate outputs the minimum value of any of its inputs.

- N SameGates. Each gate has two inputs, an Input Neighbor cost

line and the Min Gate output. The output of each SameGate is connected to a Which Neighbor node. A SameGate outputs a 0 if its two inputs are the same and a positive value if they are different.

1 Sum Gate - 2 inputs, the output of the Min Gate and the Local Cost Node. The output of the Sum Gate is used as an Input Neighbor cost line for neighborin polygons and represents the minimum cost for the minimum path from the starting polygon to this one.

Figure B-17 is an example for one polygon.

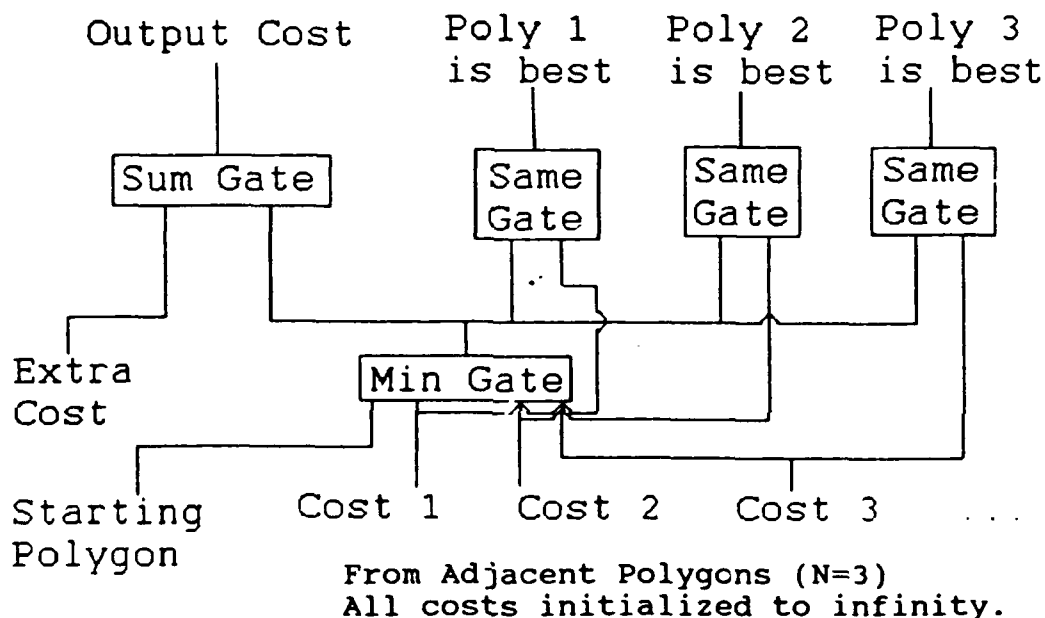


Figure B-17. Route Planner Fragment for One Polygon

The second step is the translation from the gate network to a neural network. A Network Output Node can be translated to a single output neuron. A Network Input node can be translated into an input neuron. A Min Gate must first be converted into a network of binary min gates. The translation of Binary Min Gates, Same Gates, and Sum Gates is given by the following figures.

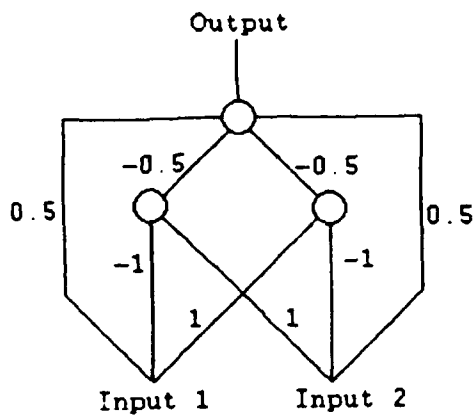


Figure B-18.
Binary Min Gate

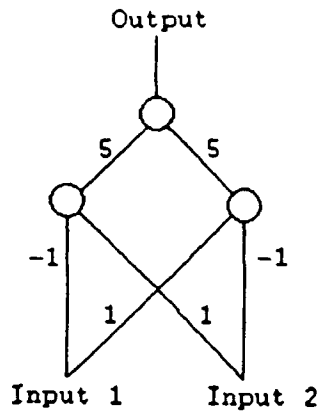


Figure B-19.
Same Gate

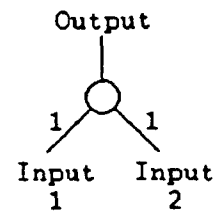


Figure B-20.
Sum Gate

APPENDIX C PROTOTYPE DESCRIPTION

This section will describe the prototype as a logical ordering of events.

A rule-base was created which rates polygons as to their desirability for helicopter traversal based on information about that polygon. This rule-base could be easily changed and the succeeding steps repeated.

The rule-base was translated into a network of neurons and virtual processors using the generic translation strategies. That network was then translated into a network of only neurons through the use of specialized strategies.

The route planning neural network was created by first reading the polygon information from a file. A connectivity map was automatically generated which specifies which polygons border each other. That connectivity map is translated into a neural network which finds the least cost route from an input vector of costs.

After the polygons are read in, they can be displayed in the prototype as in Figure C-1.

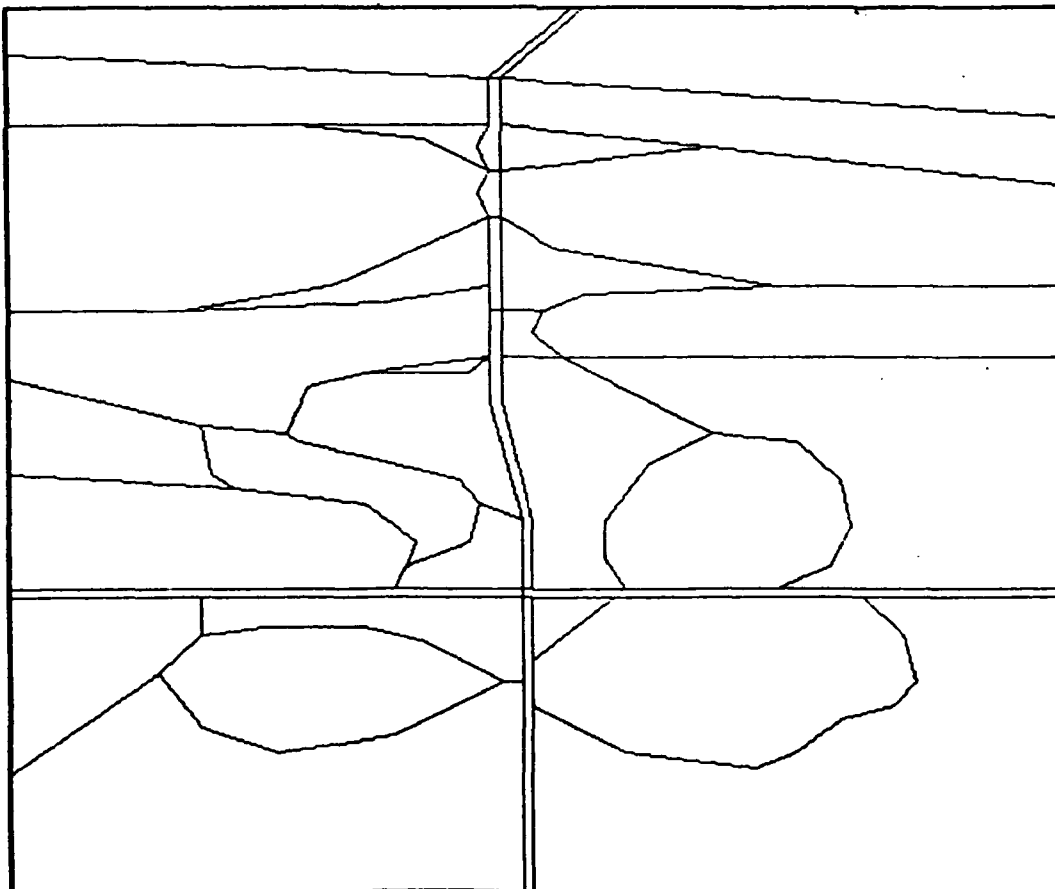


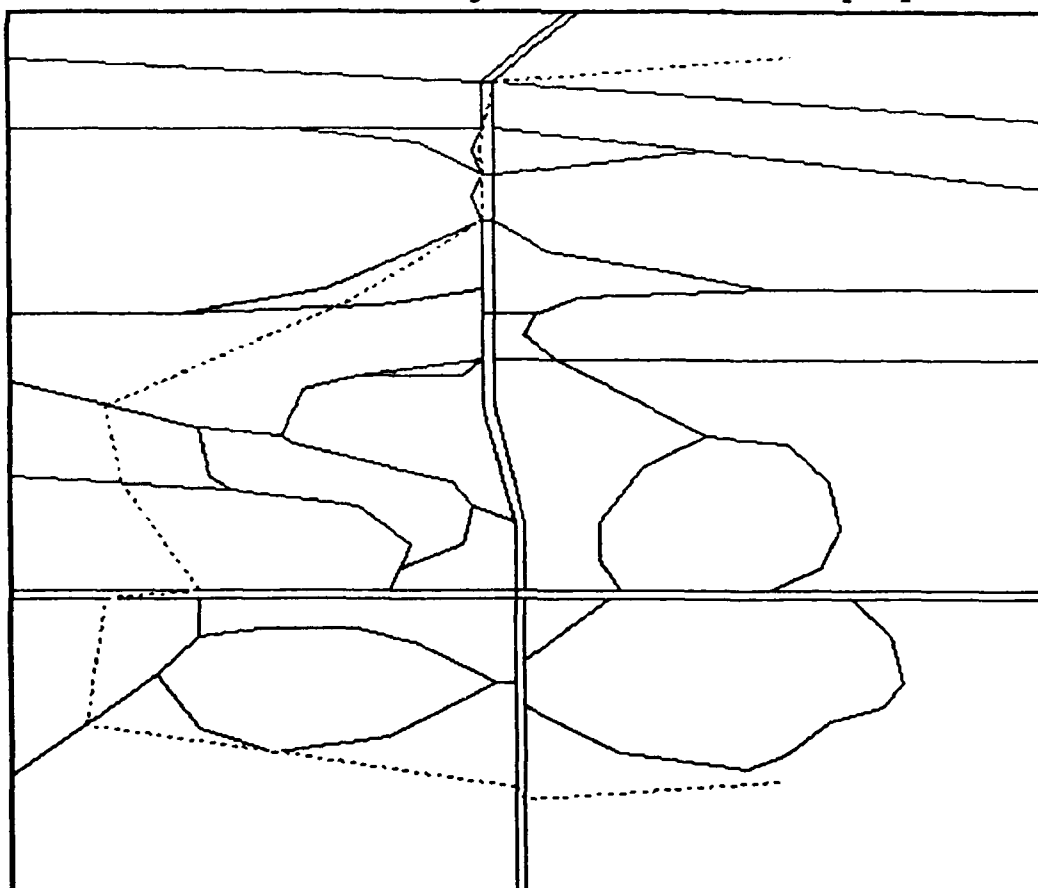
Figure C-1. Polygon Map Display

The polygon information then goes through a preprocessing step. To calculate Linearity, the points of a polygon are treated as random and the covariance between the X and Y coordinates of the points is calculated. Some adjustments must be made to allow for the average slope of the polygon's points in the X-Y plane. To determine Length, the code calculates every possible distance between the points of the polygon and uses the maximum. The points which give that maximum distance are saved for use by the area calculation method.

At this point the polygon's information might be edited to correct or update the original information which was read in. In preparation to run the network, the input vector is assembled from the polygon's information.

All of the above steps have occurred ahead of time, some taking significant processing time. The following steps would all occur at demonstration time. Any polygon can be re-edited and only that new information is written over the old information in the input vector. Then the neural network is run. The first stage of the network rates the factors associated with each polygon. The next stage sums those ratings into a cost. The last stage of the network uses those costs to plan a minimum cost route to every polygon from the current one. Finally, this route can be displayed as in Figure C-2.

Figure C-2. Route Display



APPENDIX D KAPPA KNOWLEDGE BASE

Below is a listing of the prototype AAA Planner's knowledge base. The knowledge base was implemented in KAPPA and includes rules, methods called in rules, and slots.

```

/*****
**** RULE: AAWRule
*****/
MakeRule( AAWRule, [p|Polygons],
Identity ( p:AAWCoverage ) != TRUE ,
p:Rating = p:Rating - 10 );

/*****
**** RULE: AAWNotRule
*****/
MakeRule( AAWNotRule, [p|Polygons],
Identity ( p:AAWCoverage ) != FALSE ,
p:Rating = p:Rating + 10 );

/*****
**** RULE: EGU
*****/
MakeRule( EGU, [p|Polygons],
Identity ( p:EnemyGroundUnits ) != TRUE ,
p:Rating = p:Rating - 5 );

/*****
**** RULE: EQUNot
*****/
MakeRule( EQUNot, [p|Polygons],
Identity ( p:EnemyGroundUnits ) != FALSE ,
p:Rating = p:Rating + 5 );

/*****
**** RULE: Enemy
*****/
MakeRule( Enemy, [p|Polygons],
Identity ( p:Owned ) != Threat ,
p:Rating = p:Rating - 5 );

/*****
**** RULE: Friendly
*****/
MakeRule( Friendly, [p|Polygons],
Identity ( p:Owned ) != Friendly,
p:Rating = p:Rating + 5 );

/*****
**** RULE: TrafficableNot
*****/
MakeRule( TrafficableNot, [p|Polygons],

```

```

p:Owned != Enemy And
  Identity ( p:Trafficable ) != FALSE ,
p:Rating = p:Rating + 2 );

/*****
**** RULE: Trafficable
*****/
MakeRule( Trafficable, [p|Polygons],
  p:Owned != Enemy And p:Trafficable != TRUE ,
  p:Rating = p:Rating - 2 );

/*****
**** RULE: CoveredByDustSnow
*****/
MakeRule( CoveredByDustSnow, [p|Polygons],
  Identity ( p:CoveredBy ) != Snow Or p:CoveredBy != Dust,
  p:Rating = p:Rating - 2 );

/*****
**** RULE: CoveredByWater
*****/
MakeRule( CoveredByWater, [p|Polygons],
  p:CoveredBy != Water And Identity ( p:Width ) > 50,
  p:Rating = p:Rating - 4 );

/*****
**** RULE: CoveredByVegetation
*****/
MakeRule( CoveredByVegetation, [p|Polygons],
  p:CoveredBy != Vegetation,
  p:Rating = p:Rating + 1 );

/*****
**** RULE: Forest
*****/
MakeRule( Forest, [p|Polygons],
  Identity ( p:Forest != Heavy ) And p:CoveredBy != Snow,
  p:Rating = p:Rating - 4 );

/*****
**** RULE: Checkpoints
*****/
MakeRule( Checkpoints, [p|Polygons],
  Identity ( p:CheckPoints ) > 0,
  p:Rating = p:Rating + 1 );

/*****
**** RULE: ManMade
*****/
MakeRule( ManMade, [p|Polygons],
  Identity ( p:ManMade ) != TRUE ,
  p:Rating = p:Rating - 2 );

```

```

/*****
**** RULE: ConstructionDensitySub
*****/
MakeRule( ConstructionDensitySub, [p|Polygons],
Identity ( p:ConstructionDensity ) #= Suburban,
p:Rating = p:Rating - 1 );

/*****
**** RULE: ConstructionDensityUrban
*****/
MakeRule( ConstructionDensityUrban, [p|Polygons],
p:ConstructionDensity #= Urban,
p:Rating = p:Rating - 2 );

/*****
**** RULE: ConstructionDensityMetro
*****/
MakeRule( ConstructionDensityMetro, [p|Polygons],
p:ConstructionDensity #= Metropolitan,
p:Rating = p:Rating - 3 );

/*****
**** RULE: ConstructionDensityUnin
*****/
MakeRule( ConstructionDensityUnin, [p|Polygons],
p:ConstructionDensity #= Uninhabited,
p:Rating = p:Rating + 2 );

/*****
**** RULE: VisibilityGood
*****/
MakeRule( VisibilityGood, [p|Polygons],
Identity ( p:Visibility ) #= Good,
p:Rating = p:Rating + 1 );

/*****
**** RULE: VisibilityPoor
*****/
MakeRule( VisibilityPoor, [p|Polygons],
p:Visibility #= Poor,
p:Rating = p:Rating - 5 );

/*****
**** RULE: Visibility
*****/
MakeRule( Visibility, [p|Polygons],
p:Visibility #= Fair,
p:Rating = p:Rating - 2 );

/*****
**** RULE: FFire
*****/

```

```

MakeRule( FFire, [p|Polygons],
  Identity ( p:FriendlyFire ) #= Originating Or p:FriendlyFire #=
    Targeted,
  p:Rating = p:Rating - 4 );

  /*****
  **** RULE: ObstructionsHigh
  *****/
MakeRule( ObstructionsHigh, [p|Polygons],
  Identity ( p:ObstructionDensity ) #= High,
  p:Rating = p:Rating - 2 );

  /*****
  **** RULE: ObstructionsNone
  *****/
MakeRule( ObstructionsNone, [p|Polygons],
  p:ObstructionDensity #= None,
  p:Rating = p:Rating + 2 );

  /*****
  **** RULE: NoGorges
  *****/
MakeRule( NoGorges, [p|Polygons],
  Identity ( p:AverageDepth ) > 50 And Identity ( p:Width < 2 *
    p:AverageDepth ) #= TRUE ,
  p:Rating = p:Rating - 6 * p:AverageDepth /
    p:Width );

  /*****
  **** RULE: LinearityPoor
  *****/
MakeRule( LinearityPoor, [p|Polygons],
  Identity ( p:Linearity ) >= 0.92 And p:Linearity < 0.97,
  p:Rating = p:Rating - 2 );

  /*****
  **** RULE: LinearityBad
  *****/
MakeRule( LinearityBad, [p|Polygons],
  p:Linearity >= 0.97,
  p:Rating = p:Rating - 5 );

  /*****
  **** RULE: AridStream
  *****/
MakeRule( AridStream, [p|Polygons],
  Identity ( p:ClimateType ) #= Arid And p:CovredBy #= Water,
  p:Rating = p:Rating + 2 );

  /*****
  **** RULE: GentleDepression
  *****/

```

```

MakeRule( GentleDepression, [p|Polygons],
  Identity ( p:Depression ) #= TRUE And Identity ( p:Width > 4 *
p:AverageDepth ) #= TRUE ,
  p:Rating = p:Rating + 1 );

```

```

/*****
**** RULE: AridDepression
*****/
MakeRule( AridDepression, [p|Polygons],
  p:ClimateType #= Arid And p:Depression #= TRUE,
  p:Rating = p:Rating + 1 );

```

```

/*****
**** CLASS: Polygons
*****/
MakeClass( Polygons, Belvoir );

```

```

/***** METHOD: DrawPoly *****/
MakeMethod( Polygons, DrawPoly, [],
  Let [len LengthList( Self:PointsX )]
  If ( len > 0 )
  Then {
    MoveTo(
      GetNthElem( Self:PointsX, 1 ),
      GetNthElem( Self:PointsY, 1 ) );
    For i [2 len ]
      LineTo(
        GetNthElem( Self:PointsX, i ),
        GetNthElem( Self:PointsY, i ) );
    LineTo(
      GetNthElem( Self:PointsX, 1 ),
      GetNthElem( Self:PointsY, 1 ) );
  } );

```

```

/***** METHOD: CalculateLength *****/
MakeMethod( Polygons, CalculateLength, [],
  {
    ClearList( Self:EndPoints );
    Self:Length = 0;
    Let [N LengthList( Self:PointsX )]
    For i [1 N ]
      Let [S i + +1]
      For j [S N ]
        Let [d Distance(
          GetNthElem( Self:PointsX, i ),
          GetNthElem( Self:PointsX, j ),
          GetNthElem( Self:PointsY, i ),
          GetNthElem( Self:PointsY, j ) )];
        Self:EndPoints Add d;
    Self:Length = Self:Length + Self:EndPoints;
  } );

```

```

j ) )]
{
If ( d > Self:Length )
Then {
Self:Length = d;
SetValue( Self:EndPoints, i, j );
};
};

) );

/***** METHOD: CalculateArea *****/
MakeMethod( Polygons, CalculateArea, [xslot yslot ],
{
Self:Area = 0;
Self:PCATopLength = 0;
Let [p1 GetNthElem( Self:EndPoints, 1 )]
[p2 GetNthElem( Self:EndPoints, 2 )]
[n LengthList( Self:xslot )]
Let [m ( GetNthElem( Self:yslot, p2 ) -
GetNthElem( Self:yslot, p1 ) ) /
( GetNthElem( Self:xslot, p2 ) -
GetNthElem( Self:xslot, p1 ) )]
Let [b GetNthElem( Self:yslot, p1 ) - m *
GetNthElem( Self:xslot, p1 )]
{
For i [p1 ( ( p2 - 1 ) ) 1]
Let [x1 GetNthElem( Self:xslot, i )]
[y1 GetNthElem( Self:yslot, i )]
[x2 GetNthElem( Self:xslot,
i + +1 )]
[y2 GetNthElem( Self:yslot,
i + +1 )]
{
Self:Area = Self:Area +
TrapezoidArea( Self:PCATopLength, x1,
y1, x2, y2, m, b, Self, PCATopLength
);
};
Self:PCATopLength = 0;
Self:PCAi = p2;
While (( Self:PCAi != p1 ))
{
Let [x1 GetNthElem( Self:xslot, Self:PCAi )]
[y1 GetNthElem( Self:yslot, Self:PCAi )]
[x2 GetNthElem( Self:xslot,
If ( Self:PCAi < n )
Then ( Self:PCAi +
+1 )
Else 1 )]
[y2 GetNthElem( Self:yslot,
If ( Self:PCAi < n )
Then ( Self:PCAi +

```

```

                                +1 )
                                Else 1 )]
    {
        Self:Area =
            Self:Area +
            TrapezoidArea( Self:PCATopLength, x1,
                y1, x2, y2, m, b, Self,
PCATopLength );
    };
    Self:PCAi = Self:PCAi + 1;
    If ( Self:PCAi > n )
        Then Self:PCAi = 1;
    };
    Self:Area = Abs( Self:Area );
} );

/***** METHOD: CallCalculateArea *****/
MakeMethod( Polygons, CallCalculateArea, [],
    SendMessage( Self, CalculateArea, PointsX, PointsY ) );

/***** METHOD: CalculateLinearity *****/
MakeMethod( Polygons, CalculateLinearity, [slotname ],
    Let [N LengthList( Self:PointsX )]
        Let [XBar Average( Self:PointsX )]
            [YBar Average( Self:PointsY )]
            {
                Self:PCLStdX = 0;
                EnumList( Self:PointsX, x,
                    Self:PCLStdX = Self:PCLStdX +
                        ( x - XBar ) ^ 2 );
                Self:PCLStdX =
                    Sqrt( Self:PCLStdX );
                Self:PCLStdY = 0;
                EnumList( Self:PointsY, y,
                    Self:PCLStdY = Self:PCLStdY +
                        ( y - YBar ) ^ 2 );
                Self:PCLStdY =
                    Sqrt( Self:PCLStdY );
                Self:PCLCovar = 0;
                Self:PCLStdXprime = 0;
                For i [1 N ]
                    Let [x GetNthElem( Self:PointsX, i )]
                        [y GetNthElem( Self:PointsY, i )]
                        Let [xprime XBar + y - YBar + 1.4142
                            *
                                ( Self:PCLStdY * x -
                                    Self:PCLStdX * y
                                    -
                                        Self:PCLStdY * XBar
                                        +
                                            Self:PCLStdX * YBar )

```

```

                                /
                                Sqrt(
                                    Self:PCLStdX ^ 2
                                    +
                                    Self:PCLStdY ^
                                    2 )]
                                {
                                    Self:PCLStdXprime =
                                        Self:PCLStdXprime +
                                        ( xprime - XBar ) ^ 2;
                                    Self:PCLCovar =
                                        Self:PCLCovar +
                                        ( xprime - XBar ) *
                                        ( y - YBar );
                                };
Self:PCLStdXprime =
    Sqrt( Self:PCLStdXprime );
Self:Linearity =
    ( Self:PCLCovar / Self:PCLStdY /
      Self:PCLStdXprime ) ^ 2;
} );

/***** METHOD: CalculateWidth *****/
MakeMethod( Polygons, CalculateWidth, [],
    Self:Width = Self:Area / Self:Length );

MakeSlot( Polygons:PointsX );
SetSlotOption( Polygons:PointsX, MULTIPLE );
ClearList( Polygons:PointsX );

MakeSlot( Polygons:PointsY );
SetSlotOption( Polygons:PointsY, MULTIPLE );
ClearList( Polygons:PointsY );

MakeSlot( Polygons:Label );

MakeSlot( Polygons:EdgeAdjacent );
SetSlotOption( Polygons:EdgeAdjacent, MULTIPLE );
ClearList( Polygons:EdgeAdjacent );

MakeSlot( Polygons:PointAdjacent );
SetSlotOption( Polygons:PointAdjacent, MULTIPLE );
ClearList( Polygons:PointAdjacent );

MakeSlot( Polygons:Length );
SetSlotOption( Polygons:Length, VALUE_TYPE, NUMBER );

MakeSlot( Polygons:EndPoints );
SetSlotOption( Polygons:EndPoints, MULTIPLE );
ClearList( Polygons:EndPoints );

MakeSlot( Polygons:Owned );

```


SetSlotOption(Polygons:Owned, ALLOWABLE_VALUES, Friendly, Threat, None);

MakeSlot(Polygons:AAWCoverage);
SetSlotOption(Polygons:AAWCoverage, VALUE_TYPE, BOOLEAN);

MakeSlot(Polygons:EnemyGroundUnits);
SetSlotOption(Polygons:EnemyGroundUnits, VALUE_TYPE, BOOLEAN);

MakeSlot(Polygons:Rating);
SetSlotOption(Polygons:Rating, VALUE_TYPE, NUMBER);
Polygons:Rating = 0;

MakeSlot(Polygons:Trafficable);
SetSlotOption(Polygons:Trafficable, VALUE_TYPE, BOOLEAN);

MakeSlot(Polygons:CoveredBy);
SetSlotOption(Polygons:CoveredBy, ALLOWABLE_VALUES, Dust, Snow, Water, Rocks, Vegetation);

MakeSlot(Polygons:Forest);
SetSlotOption(Polygons:Forest, ALLOWABLE_VALUES, None, Slight, Medium, Heavy);

MakeSlot(Polygons:CheckPoints);
SetSlotOption(Polygons:CheckPoints, VALUE_TYPE, NUMBER);
SetSlotOption(Polygons:CheckPoints, MINIMUM_VALUE, 0);

MakeSlot(Polygons:ConstructionDensity);
SetSlotOption(Polygons:ConstructionDensity, ALLOWABLE_VALUES, Uninhabited, Rural, Suburban, Urban, Metropolitan);

MakeSlot(Polygons:ObstructionDensity);
SetSlotOption(Polygons:ObstructionDensity, ALLOWABLE_VALUES, High, Low, None);

MakeSlot(Polygons:Visibility);
SetSlotOption(Polygons:Visibility, ALLOWABLE_VALUES, Good, Fair, Poor);

MakeSlot(Polygons:ClimateType);
SetSlotOption(Polygons:ClimateType, ALLOWABLE_VALUES, Arid, Tropical, SubTropical, Normal, Artic);

MakeSlot(Polygons:ManMade);
SetSlotOption(Polygons:ManMade, VALUE_TYPE, BOOLEAN);

MakeSlot(Polygons:FriendlyFire);
SetSlotOption(Polygons:FriendlyFire, ALLOWABLE_VALUES, None, Originating, Targeted);

```

MakeSlot( Polygons:AverageDepth );
SetSlotOption( Polygons:AverageDepth, VALUE_TYPE, NUMBER );
SetSlotOption( Polygons:AverageDepth, MINIMUM_VALUE, 0 );

MakeSlot( Polygons:Linearity );
SetSlotOption( Polygons:Linearity, VALUE_TYPE, NUMBER );
SetSlotOption( Polygons:Linearity, IF_NEEDED, CalculateLinearity );

MakeSlot( Polygons:Depression );
SetSlotOption( Polygons:Depression, VALUE_TYPE, BOOLEAN );

MakeSlot( Polygons:PCLStdX );

MakeSlot( Polygons:PCLStdY );

MakeSlot( Polygons:PCLCovar );

MakeSlot( Polygons:PCLStdXprime );

MakeSlot( Polygons:Width );

MakeSlot( Polygons:CostStack );
SetSlotOption( Polygons:CostStack, MULTIPLE );
SetSlotOption( Polygons:CostStack, VALUE_TYPE, NUMBER );
ClearList( Polygons:CostStack );

MakeSlot( Polygons:WhoStack );
SetSlotOption( Polygons:WhoStack, MULTIPLE );
SetSlotOption( Polygons:WhoStack, VALUE_TYPE, OBJECT );
SetSlotOption( Polygons:WhoStack, ALLOWABLE_CLASSES, Polygons );
ClearList( Polygons:WhoStack );

MakeSlot( Polygons:Cost );

MakeSlot( Polygons:Who );

MakeSlot( Polygons:PotentialCost );
SetSlotOption( Polygons:PotentialCost, VALUE_TYPE, NUMBER );

MakeSlot( Polygons:PotentialWho );
SetSlotOption( Polygons:PotentialWho, VALUE_TYPE, OBJECT );
SetSlotOption( Polygons:PotentialWho, ALLOWABLE_CLASSES, Polygons );

MakeSlot( Polygons:Updated );
SetSlotOption( Polygons:Updated, VALUE_TYPE, BOOLEAN );

MakeSlot( Polygons:Area );

MakeSlot( Polygons:PCATopLength );

MakeSlot( Polygons:PCAI );

```