MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS 1963 A

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/CI/NR 85-17T	GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Generation of Flight Paths Using Heuristic Search		5. TYPE OF REPORT & PERIOD COVERED THESIS/DISSERTATION
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Carl Steven Lizza		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS AFIT STUDENT AT: Wright State Univ		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS AFIT/NR WPAFB OH 45433		12. REPORT DATE 1984
		13. NUMBER OF PAGES 100
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASS
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

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

18. SUPPLEMENTARY NOTES

APPROVED FOR PUBLIC RELEASE: IAW AFR 190-1

LYNN E. WOLAVER 23 Feb 85
Dean for Research and
Professional Development
AFIT, Wright-Patterson AFB OH

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

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

ATTACHED

GENERATION OF FLIGHT PATHS

USING HEURISTIC SEARCH
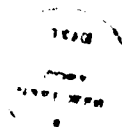
A thesis submitted in partial fufillment
of the requirements for the degree of
Master of Science

By

CARL STEVEN LIZZA
B.S., Ohio State University, 1977

1984
Wright State University

WRIGHT STATE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

November, 1984

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY

__Carl Steven Lizza__ ENTITLED __Generation Of Flight Paths Using Heuristic

Search__ BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF __MASTER OF SCIENCE__ .

_____
Thesis Director

_____
Chairman of Department

Committee on
Final Examination

_____

_____

_____

_____
Dean of the School of Graduate Studies

ABSTRACT

Lizza, Carl Steven. M.S., Department of Computer Science, Wright State University, 1984. Aircraft Route Planning in a Hostile Enemy Environment Using Heuristic Search.

This thesis examines a solution to the problem of developing an effective, computer algorithm to route aircraft through hostile enemy defenses. The problem was proposed to assist in studies of the impact of varied, on-board countermeasures upon pre-planned aircraft routes. The quality of a route can be defined in terms of the cost of interaction between the aircraft and threats encountered along the path, and the length of the path. Current methods of automated routing are either inefficient or produce unsatisfactory results. Given a more general description of the problem, this thesis details a solution for developing routes by using the artificial intelligence technique of heuristic search in an A* algorithm. The algorithm uses heuristics based upon estimates of the degree of threat interactivity and distance from a point to the goal. Due to the nature of the heuristic, the A* algorithm cannot guarantee development of a least-cost path. However, the heuristic may be adjusted to insure reasonably good results in most cases. Test cases confirm improvements in efficiency while maintaining solution quality comparable to previous methods. The flexibility of the algorithm allows applicability to the specific aircraft routing problem and to other route planning applications.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF TABLES

LIST OF TABLES (Continued)

To Gretchen and Tony

For their loving tolerance and invaluable support

during these long hours and each passing year.

I. INTRODUCTION

This thesis topic was proposed by Mr. William McQuay of the Air Force Wright Aeronautical Laboratories, Wright Patterson AFB, Ohio. The proposal was to develop a system that would create aircraft penetration routes through a hostile enemy environment. Given the ability of an aircraft to carry various configurations of threat-countering devices, the system would enable a planner to evaluate the effect of different configurations upon route length, threat interaction, fuel consumption, etc. Clearly, this problem is characteristic of a general class of problems involving developing routes through obstacles. For example, building a road as short as possible while trying to avoid hills, lakes, etc. Or, if the obstacles can be considered impenetrable, a robot moving around a room of furniture. Therefore, it was decided to attempt to develop a solution to a more general problem description while maintaining applicability to the specific task of aircraft routing.

This paper describes methods currently used to solve the specific problem of aircraft routing, proposes alternative solutions using artificial intelligence techniques, defines a generalized problem space, then presents the implementation of a heuristic search algorithm. Finally, the algorithm is tested with the results analyzed and conclusions presented.

1

node to node n is known when node n is generated if n is not the successor of any other nodes. However, the actual cost of the path from node n to the goal node can only be estimated. It is the function of the heuristic to estimate this cost, designated h*(n). A new evaluation function, f*, can now be defined as:

$$f*(n) = g*(n) + h*(n).$$

A **GRAPHSEARCH** algorithm which employs this evaluation function is called algorithm A. The term g* is the cost of the current best path from the start node to node n. The h* term represents the estimate of the cost from node n to the goal. If h*(n) = 0 for all n, and g*(n) = cost of n for all n, then algorithm A is a breadth-first search. Further, if h*(n) ⩽ h(n) for all n (h* is optimistic), then algorithm A will find an optimal, or least cost, path to the goal. This algorithm is called an A* (A-star) algorithm. It has been proven (Nilsson[3]) that the A* algorithm is admissible, that is, it always finds an optimal path when the heuristic is optimistic.

The importance of the heuristic function can now be examined. If the heuristic function h*(n) = h(n) for all n, then the function is a perfect estimator and the algorithm will perform as a depth-first search along the optimal path. And, as stated above, if h*(n) = 0 for all n, then the algorithm performs as a breadth-first search. So, the closer h*(n) estimates h(n), the closer the algorithm will act as a depth first search over minimum breadth. That is, fewer nodes will need to be expanded.

A heuristic estimate which improves as successors are generated along the same path is called consistent. With a consistent heuristic, the A* algorithm will be optimal in terms of efficiency, i.e. it expands the fewest possible nodes. When the heuristic is inconsistent, i.e. the estimate does not

of artificial intelligence techniques. Heuristics are utilized in step 8 of **GRAPHSEARCH** to 'intelligently' order the **OPEN** list, where the most promising node is placed at the head of the list. These heuristics are a means of estimating the 'solution potential' of a problem state. For example, Figure 3.3 depicts the familiar fifteen-puzzle. The problem is to arrange the tiles in sequential order, left-to-right, top-to-bottom. A simple heuristic may be: count the number of tiles already in the correct position. A more complicated
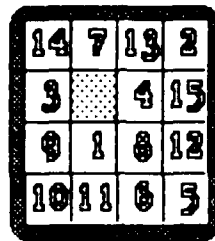
Figure 3.3
15-Puzzle

heuristic may be : estimate = $(C_1$ * count of the number of tiles already in the correct position) + $(C_2$ * the distance of tiles from their correct position) + $(C_3$ * the count of the sets of adjacent tiles whose correct locations are reversed), where C_n are weighted coefficients.

An evaluation function, f, is defined as:

$$f(n) = g(n) + h(n)$$

where $g(n)$ is the actual cost of the best path from the start node to node n, and $h(n)$ is the actual cost of the least cost path from node n to the goal node. So, $f(n)$ is the actual cost of the path from the start node to the goal node which passes through node n. The least cost of the path from the start

solution, if one exists, but the solution with the least cost. Due to the nature of the tree structure, the number of nodes at each level increases exponentially. This search scheme is therefore extremely expensive in terms of memory and CPU requirements.

In an exhaustive, breadth-first search, as outlined above, every terminal node that is a goal node will yield a solution path. Essentially the technique of SNOOPER is goal directed, breadth-first search. It is 'goal-directed' since the goal node is the root of the tree. It is an exhaustive search in that every node generated is expanded. Consider a problem presented to SNOOPER with a grid 100 by 50 units in size. SNOOPER generates 16 possible moves at every point (eight directions, two altitudes), except for edges. Therefore, there are slighty fewer that 100 * 50 * 16, or 80 000, nodes generated. And, many of these node are duplicates requiring the associated overhead of step 7 in the **GRAPHSEARCH** algorithm! Granted, an optimal solution is found for every point on the grid as a starting point. Indeed, in the route planning application, multiple starting points to a single goal may be desireable. But if only a single, best path is required, A* search provides a more efficient alternative.

A* SEARCH

The A* algorithm is an artificial intelligence technique that attempts to combine the benefits of depth-first and breadth-first searches by guiding the direction of the search pattern using heuristics. Heuristics may be defined as the application of task-dependent knowledge to minimize the size of the tree which must be created. In this sense, heuristics are part of the 'intelligence'

7   Order these members of M according to some arbitrary scheme or according to heuristic merit.

8   Add the 'best' member of M to OPEN.

This modification will produce the same solution as the FPG model. That is, the FPG model is essentially a depth-first search where the first iteration always terminates in a solution.

In general, though, depth-first searching can spend a great amount of effort exploring fruitless paths if few paths lead to a goal node. An alternative method is breadth-first search.

BREADTH-FIRST SEARCH

The breadth-first search is characterized by a search pattern that expands the least cost nodes at a level, before proceeding downward to the next level. The GRAPHSEARCH procedure can be modified to perform as a breadth-first search with the following modifications:

3 Set K = ∞.
   LOOP: If OPEN is empty, exit with best as solution

5   If n is a goal node and its cost is the lowest yet found, save n as best and set K to the cost of best.

7   Add members of M with cost less than K to OPEN.

8   Reorder OPEN according to cost.

It can be shown that breadth-first search is not only guaranteed to find a
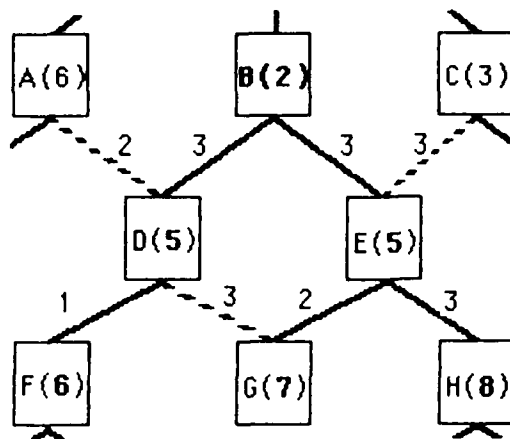
Figure 3.2

Ancestor Pointers Following B-node Insertion

DEPTH-FIRST SEARCH

The depth-first search technique is characterized by a search pattern through the tree that proceeds downward along a single path until a terminal node is encountered (or an arbitrary 'depth bound' is reached). If that terminal node is a goal node then the solution is found. Otherwise, the pattern 'backs up' a level, selects the most promising choice, and proceeds down that path until a terminal node is reached. This process continues until a goal node is reached (success), or all nodes have been examined (failure).

In this aircraft route planning application, one can determine by inspection that every path in the graph will terminate in a goal node. This can be guaranteed by not generating nodes which themselves cannot generate successors. Given this, and not considering duplicate generation of a node, steps 7 and 8 of **GRAPHSEARCH** can be modified as follows:

D is not the parent of G in the search tree (although G is a successor of D)
since its cumulative cost is higher than that of node E. Assume that node B is
expanded with a cumulative cost of two and successors D and E (Figure 3.2).
Node B represents a 'better' path to node D than does node A. Therefore, the
parent pointer from node D must be changed to node B. But since node D has
successors, the new cost must be propogated likewise. This process
continues recursively until all successors of B are updated. Figure 3.2
shows the new costs and a pointers after propogating all cost changes.
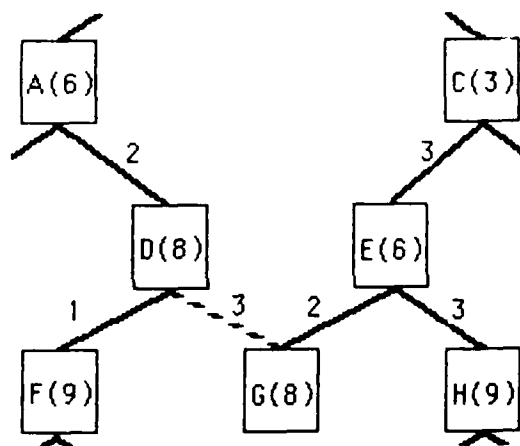
Figure 3.1

Ancestor Pointers Prior to B-node Insertion

already in G (i.e., not already on either **OPEN** or **CLOSED**).
Add these members of **M** to **OPEN**. For each member of **M** that
was already on **OPEN** or **CLOSED**, decide whether or not to
redirect its pointer to **n**. For each member of **M** already on
CLOSED, decide for each of its descendants in **G** whether or
not to redirect its pointer. (See following discussion
regarding redirection of pointers.)

8   Reorder the list **OPEN**, either according to some arbitrary scheme
or according to heuristic merit.

9 Go **LOOP**

The **OPEN** list contains those nodes that have been generated as
successors of previous nodes, but not yet selected for expansion. The
CLOSED list contains those nodes which have already been expanded. The
explicit search tree, **G**, collects each possible path to a generated node. A
single distinguished path can be traced backwards from any node through
single ancestor pointers maintained in step 7. Therefore, when the algorithm
terminates successfully by selecting the goal node from **OPEN**, the generated
solution path can be traced backwards from that node along its distinguished
path.

In a search graph, a single node may be reached by different paths (with
different costs) from the start node. When this occurs, it is necessary to
decide whether the parent pointer should be changed to select a parent with a
lower cost. For example, Figure 3.1 shows a graph with cumulative cost
from the start node indicated in parenthesis and the arc cost between nodes
indicated beside the arc. The dashed line between nodes G and D indicate that

successors. Each successor may have successors and so itself be the root of another graph. If a node has no successors, then it is a terminal node. This terminal node is a goal node if its state matches the goal state. Otherwise, it represents a path through the graph which failed to terminate at a solution. Each node points to that single parent node which represents the 'best' path to that node from the start node.

A search tree is a specific case of a search graph where a node can only be the successor of a single node.

A general, graph searching procedure, as outlined by Nilsson[3], is as follows:

procedure GRAPHSEARCH

1 Create a search graph, **G**, consisting solely of the start node, **s**. Put **s** on a list called **OPEN**.

2 Create a list called **CLOSED** that is initially empty.

3 **LOOP:** If **OPEN** is empty, exit with failure.

4   Select the first node on **OPEN**, remove it from **OPEN**, and put it on **CLOSED**. Call this node **n**.

5   If **n** is a goal node, exit successfully with the solution obtained by tracing a path along the pointers from **n** to **s** in **G**.

6   Expand node **n**, generating the set, **M**, of its successors that are not ancestors of **n**. Install these members of **M** as successors of **n** in **G**.

7   Establish a pointer to n from these members of **M** that were not

III. REVIEW OF SEARCH TECHNIQUES

STATE SPACE SEARCH

A state space is a formal description of a problem which consists of all possible configurations of the relevant objects. It is not necessary to enumerate all possible configurations explicitly. These configurations, or states, can be described by defining the set of objects within the problem space, and a set of operators or rules which, when applied to one or more of the objects, creates a new state. It is necessary to define one or more of these states as initial states, and likewise as define goal states. The notion of state space search is a process that begins with an initial state, and iteratively applies rules to move through the state space until a goal state is reached.

A search graph consists of nodes characterizing individual states within the state space. These nodes consist of a single parent node pointer, and multiple successor node pointers. The nodes of the graph are connected by arcs which each have an associated cost. This cost may not be constant over all arcs. For example, assume city a is a node in a graph with cities b and c as successor nodes, and arc cost is based upon distance between cities. Therefore it is possible to go from city a to cities b or c in one move. Yet the distance, and hence the arc cost, to city b may be greater than the distance to city c.

The search graph consists of a root node, the starting state, and its

techniaue does, however, hold promise of perhaps being less costly than heuristic search since the problem space may be significantly reduced through planning.

Since the SNOOPER approach is essentially exhaustive search, and the FPG approach resembles a depth-first search, it seemed feasible that a workable solution could be developed using heuristic search. It is possible to construct the heuristic search algorithm to guarantee finding an optimal solution given an accurate heuristic. Therefore, since heuristic search is clearly related to previous techniques, and it might possibly build an optimum path, it is the approach that was adopted for this solution and is described in detail in the following chapter.

spectrum of computational efficiency. That is, the FPG model is computationally very inexpensive since it examines a very limited number of alternatives. While the SNOOPER model examines every possible alternative at the expense of computer resources.

PROPOSED SOLUTION TECHNIQUES

Instead of building an entire search graph for a problem, artificial intelligence techniques strive to build as small of a subset of that graph as possible to locate a solution. Two techniques of 'cutting-down' the size of the graph are hierarchic planning and heuristic search.

A hierarchic planner, such as STRIPS[10], will iteratively develop solutions at an abstract level of detail, then pass the solution to the next level of detail where it can be used as an plan. This process continues until all details of the problem space are incorporated into the solution. The algorithm for this problem may begin by dividing the grid into several large boxes. Use some scheme for assigning a value to each box. Select the 'best' boxes from start to goal as a first-level plan. Then, divide each box into smaller boxes where each then becomes a separate problem space. This would proceed iteratively until an arbitrary limit is reached. The sub-plan may even be used to efficiently guide a heuristic search within sub-areas. Or, if the sub-area is small enough, an exhaustive search may become feasible. The single greatest drawback to this scheme is that the quality of the solution will be extremely dependent upon that first-level plan. This may necessitate developing and exploring alternate plans at the higher levels. Likewise, there can be no guarantee of finding an optimal solution. This

distance travelled. For this model, a grid is established with a fixed goal point but with an indefinite starting point. The aircraft is assumed to move between points on the grid parallel to either axis or at a 45 degree angle. Therefore, except for boundary points, there are eight possible directions of travel from any point. SNOOPER also permits discrete changes in altitude, high or low, to create 16 possible moves (except for boundary points). Starting at the goal, the algorithm iteratively examines every possible move from each point, calculating the cost of the move in terms of threat interaction and length. The result of these cost assignments is a penalty matrix from which an optimum route may be traced in a stepwise fashion backwards from the goal to any starting point on the grid. That is, every point on the grid, except for the goal, is a potential starting point. Figure 2.1 shows a simple example of a penalty matrix, and another representation of the same information using a tree structure.
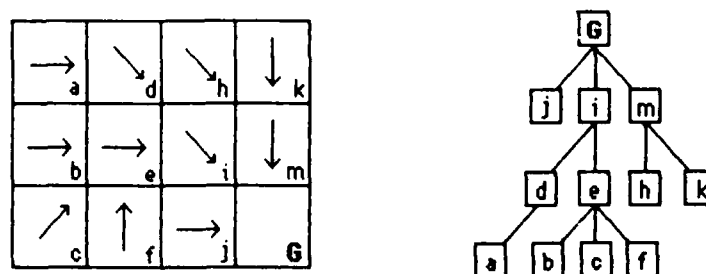
Figure 2.1

Penalty Matrix vs. Tree Representation

This scheme has several distinct features. As stated before, it will locate an optimum path from any starting point to a fixed goal. The cost of this feature, in computing resources, will be extremely large compared with the FPG model. In fact, these two models may represent the extremes of the

in the x-direction when the leg is projected on the nominal path. That is, each possible leg from a given point maintains the same 'forward progress' towards the goal. This is due to the assumption that the aircraft will fly at maximum speed on outward legs, and at minimum speed along the nominal path. A fixed number of possible legs are generated at each point within this path deflection angle. The algorithm then 'looks' outward along each proposed leg to detect threat interactions. Once detected, the type of each threat can be referenced in a table and the effect of each threat on the leg can be accumulated. The algorithm continues, selecting the shortest leg at each point with the least threat interaction until the goal is reached.

An extremely important feature of this technique is the existence of threats is 'discovered' by the algorithm only when they are encountered while evaluating possible legs. Effectively, the threats 'pop-up' in front of the aircraft which then makes a path adjustment to avoid them. Therefore, this scheme is especially applicable as an on-board flight path generator. Given that only limited information is available, it develops as good a route as possible. As a pre-flight planner though, assuming that the location of threats is known, the FPG model will develop relatively poor routes compared with other techniques which make effective use of the additional information.

STRATEGY NEEDED OVER OPTIMUM PENETRATION ROUTES (SNOOPER) MODEL

The SNOOPER model[4] makes use of the total knowledge of the threat environment to develop optimum routes using dynamic programmimg techniques. As in the previous model, optimum primarily means the path with the smallest cost in terms of threat interaction, and secondarily in distance

II. BACKGROUND

The advent of computer generated aircraft routes has been severely
hampered by the computational complexity of the current algorithms and the
scale of the problem. As a result, routing is often done by 'hand' using a
map, some pins, and a string. Perhaps the most useful advance has been in
the application of computer graphics to assist in this manual process. There
are two examples of aircraft route generation algorithms, which have
influenced the solution to be described in this thesis.

FLIGHT PATH GENERATION (FPG) MODEL.

The Flight Path Generation model[6] defines a problem space as a grid,
or corridor, aligned along an x-y axis. Given a starting point on the left edge
and a goal, or target, along the right edge, the FPG model 'flys' the aircraft
at a constant altitude within the corridor from the starting position to the
goal. The flight path is composed of a sequence of flight legs. A nominal
flight path is defined as a straight line from the start to the goal. The length
of a leg parallel to the nominal flight path is equal to the distance that the
aircraft can 'look-ahead' to detect threats. This 'look ahead' distance is
called the awareness radius. Given maximum and minimum speeds for the
aircraft, a path deflection angle is computed using the formula:

$$(\cos^{-1}(\text{minimum speed} / \text{maximum speed}))$$

Within this angle a leg can deviate in direction, yet cover the same distance

2

improve between a node and its successor, then the A* algorithm can search
exponentially. The goal of a proposed modification (Martelli [5]) to the A*
algorithm is to improve the efficiency with an inconsistent heuristic. The
essence of the technique is to recognize when the heuristic is inconsistent,
and to temporarily ignore it. This modified **GRAPHSEARCH** algorithm follows:

procedure **MODIFIED_GRAPHSEARCH** (* changes are <u>underlined</u> *)

1 Create a search graph, **G**, consistong solely of the start node, **s**.
   Put **s** on a list called **OPEN**. <u>Set F = 0</u>.

2 Create a list called **CLOSED** that is initially empty.

3 **LOOP**: if **OPEN** is empty, exit with failure.

4 <u>If the first node on **OPEN** has an f-value < F then select the node
      from **OPEN** with the smallest g-value. Otherwise</u>, select the
      first node on **OPEN**, remove it from **OPEN**, and put it on
      CLOSED. Call this node n and <u>set F = the f-value of n.</u>

5 If n is a goal node, exit successfully with the solution obtained by
      tracing a path along the pointers from n to s in **6**.

6 Expand node **n**, generating the set, **M**, of its successors that are
      not ancestors of n. Install these members of **M** as
      successors of **n** in **6**.

7 Establish a pointer to n from these members of **M** that were not
      already in **G** (i.e., not already on either **OPEN** or **CLOSED**).
      Add these members of **M** to **OPEN**. For each member of **M** that
      was already on **OPEN** or **CLOSED**, decide whether or not to
      redirect its pointer to **n**. For each member of **M** already on

CLOSED, decide for each of its descendants in **G** whether or not to redirect its pointer. (See following discussion regarding redirection of pointers.)

8   Reorder the list **OPEN** according to f-value. In the case of ties, goal nodes are to be placed first.

9 Go **LOOP**

This modified A* search algorithm is the method chosen for implementation as the Path-Finder program presented in the next chapter. The advantage of the modified algorithm is that it: (1) can be proven to be admissible with a consistent heuristic; (2) expands fewer nodes than the A* algorithm when presented with an inconsistent heuristic. Regarding efficiency, the second claim is based upon an ordered search algorithm which differs slightly from the **GRAPHSEARCH** algorithm. The ordered search algorithm will return a duplicately generated node to the **OPEN** list. It will therefore be counted as an expansion more than once. The **GRAPHSEARCH** algorithm expands the node only once, choosing instead to recursively propogate an improved path through the successors. The effect of this difference will be noted during the discussion of test results.

IV. PATH-FINDER SYSTEM IMPLEMENTATION

The problem space is organized as a grid of arbitrary units and size, with the origin of the axis in the lower left corner. A point on the y-axis is designated as the starting point, and a point on the opposite boundary is designated as the goal. Threats are located throughout the grid and are grouped by type. Each threat type is characterized by a radius, and a probability of damage (Pd). The Pd value, normally 0 < Pd < 1.0, shall be treated as a fractional part of a maximum threat cost rather than as a true probability. This aspect is discussed in a later section. In previous implementations, such as FPG and SNOOPER, the threats are modelled as as series of concentric circles, each divided into various sections. Each section can then be given a specific Pd which accounts for 'time in threat', area of intersection, etc. Although this is a realistic model for actual aircraft route planning, for simplicity, this problem will consider an entire threat with a single Pd. Other threat modelling schemes are easily incorporated into the design as needed by a particular problem domain. Another detail, changes in altitude, is excluded from the problem, also for simplicity. Altitude changes would simply multiply the number of possible legs at each point and require a different threat model with altitude specific information. Again, these features are easily incorporated as domain specific requirements. The FPG model used minimum and maximum speeds to compute the angle of path

deviation which will permit equivalent progress towards the goal for each leg. Rather than include speed of the object as part of the problem domain, the path deflection angle will be included as an input parameter. Also, all legs will be of equal length (except for perhaps the last one).

LANGUAGE AND HARDWARE

The nature of artificial intelligence programs makes a recursive language a necessity for tree manipulation, etc. The language chosen for this implementation is PASCAL, due to its wide acceptance and availability. PASCAL also lent itself to design and testing of program modules on a micro-computer. Final implementation and testing were performed on a DEC VAX 11/780 mini-computer. Off-line plots were produced on an APPLE MACINTOSH micro-computer.

PATH-FINDER ALGORITHM.

The entire PASCAL source for PathFinder is presented in Appendix 1. The general program design and data structures closely follows the version of the **GRAPHSEARCH** algorithm presented in Rich[1] with the Martelli modifications described in the previous chapter. The following sections detail some of the domain dependent details of the implementation.

LEG GENERATION

The information required to generate possible legs from a given point location is the path deflection angle, length of a leg, and the number of legs to be generated. These three factors are are all input by the user, and fixed

throughout the execution of a case. The path deflection angle describes the deviation from the nominal flight path (a straight line from the start to the goal). The path deflection angle is evenly subtended into the specified number of leg deflection angles according to the number specified as legs-per-arc. From this information, the coordinates of the endpoints are easily computed.

Leg Deflection Angle

Path Deflection Angle

parallel to x-axis
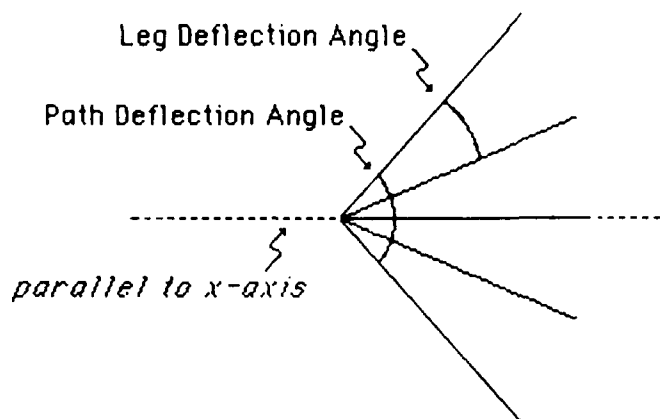
Figure 4.1
Leg Generation Diagram

If the distance from the given point to the goal is less than the leg length, then a single leg is generated joining the two points. This is the only case where a leg is generated of length less than the leg length parameter.

Boundaries are established at the top and bottom edges of the grid, and 'looking backwards' from the goal along the path deflection angle. The effect of these boundaries is to eliminate the possibility of generating a leg either going outside the grid, or whose successors could not reach the goal due to the limitation of the path deflection angle.
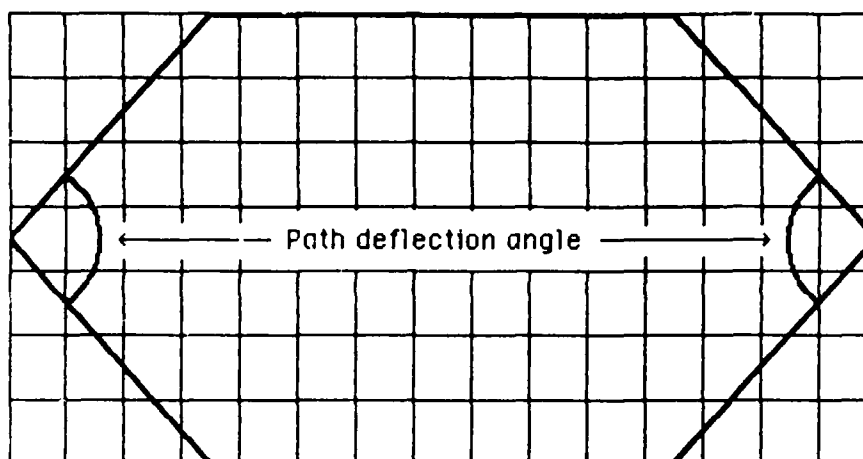
Figure 4.2

Leg Generation Boundaries

COMPUTATION OF ACTUAL LEG COSTS

The computation of actual cost over a leg is determined by the formula:

$$g = g\_coeff * ((g\_length\_coeff * leg\_length) +$$

$$(g\_threat\_coeff * threat\_cost\_of\_leg))$$

By adjusting these coefficients the definition of optimal path is altered since these coefficients represent the relative importance of path length versus threat avoidance. As a result, given the same problem domain with different coefficients, the algorithm may produce different paths.

Three coefficients are somewhat redundant, though they facilitated early testing. Usually, the g_coeff and g_length_coeff should be set to one and the g_threat_coeff manipulated to define optimality. A description of the threat cost assignment scheme is presented in a later section.

COMPUTATION OF ESTIMATED COST-TO-GOAL

The heuristic estimate of cost-to-goal is computed over a direct path from the given location to the goal by the formula:

$$h* = h\_coeff * ((h\_length\_coeff * distance\_to\_goal) +$$
$$(h\_threat\_coeff * threat\_cost\_to\_goal))$$

where the distance_to_goal is the straight-line distance from the point to the goal. This formula bases the heuristic estimate upon the same factors as actual cost. The number of coefficients provides the flexibility to create new heuristics simply by alterering their relative values. The h_coeff coefficient, if set to zero, effectively disables the heuristic creating a breadth-first search. Other values for h_coeff can alter the relationship between actual and estimated costs, g* and h*, without affecting the relationship between length and threat costs in the heuristic. This relationship between g* and h* is important since it defines the accuracy of the heuristic at estimating actual cost. As stated earlier, the more accurate the heuristic, the more efficient the algorithm will be at finding a solution.

ACTUAL THREAT COST COMPUTATION

Computation of actual threat cost involves accumulating the Pd's of all threats intersected by a leg. This method, the arithmetic sum, assumes that the Pd, in spite of the name, represents a fractional part of a maximum cost (represented by the threat coefficient). The arithmetic sum method accumulates cost as:

$$cost = \sum Pd's \text{ of intersected threats}$$

It is quite generic since it assigns a simple cost to a threat, and it gives a

clear threat value that is easily traced in graphic output. In order to define a 'solid' threat, one that 'must' be avoided, there are two methods. These solid obstacles could represent terrain contours which an aircraft certainly should attempt to avoid. One method is to simply assign a very high Pd to the solid threat, such as 1000. The algorithm may still find a path that intersects a solid threat, if no clear path exists, but the resultant threat cost for the path will reflect the very high cost. An alternative is to assign a threat cost of one and to modify the threat_cost evaluation routine to return a very high value as a result of an intersection of a threat of Pd equal to one. Another simple modification to the develop_path routine, recognizing this high value, could discard the leg entirely. As a result, the algorithm will fail if it cannot find a path.

Testing for the intersection of threats by a leg involves computing the perpendicular distance from the center of the threat to a line defined by the leg. If that distance is less than the radius of the threat then it must be verified that the threat actually intersects the line segment (leg). If the threat lies between the endpoints, yet intersects the leg, then the distance from either endpoint to the threat center must be less than the hypoteneuse of a right triangle with sides equal to the length of the leg and the distance from the opposite endpoint to the threat center. In Figure 4.3, the length of a must be less than the length of a', and likewise for the segments from the opposite endpoint. Other tests must be performed to insure that a threat intersection is not counted twice if it occurs over more than one leg of the same route. This is done by excluding a threat intersection from the cost for a leg if it covers the left-most endpoint since it must have been counted on

the prior leg. This is not true for the initial leg, however, whose left-most endpoint is the starting point which may lie within a threat which needs to be counted. If an entire leg is within a single threat, the left-most exclusion rule will also exclude the threat.
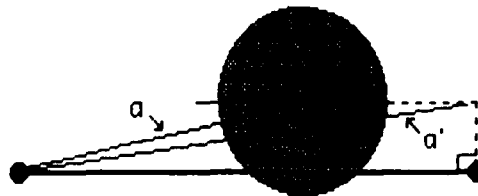
Figure 4.3
Intersection of a Leg by a Threat

HEURISTIC THREAT COST COMPUTATION

The computation of heuristic threat cost also involves the same formula for the arithmetic sum of Pd's.

The goal of the heuristic is to quantify the 'merit' of proceeding along a particular path. It must therefore measure expected interaction with threats between the current location and the goal. One heuristic is to accumulate the Pd's of all threats which actually intersect that direct path to the goal. The idea is that this value may represent the worst case cost of the path to the goal from the given point. Therefore, since the actual cost will likely be less, the h-threat coeficient should be less than the g-threat coefficient.

Another type of heuristic is to create a box around the projected path and accumulate the Pd's of all threats within the box. The reasoning is to consider all threats within some area of manueverability around a straight path to the goal. Again, this value may be even greater than the first method, so it is

reasonable to assume that a larger difference between the two threat

coefficients will be necessary. A box width equal to twice the leg length was

chosen for the test cases presented in the next chapter. Necessary code

changes for this heuristic are detailed in Appendix B.

V. USER'S GUIDE FOR PATH-FINDER

A separate program has been created to facilitate creation of input data sets for the Path-Finder program. A complete listing for this program, Threat-Builder, is included as Appendix C. Threat-Builder queries the user for input parameters to describe the threat domain, such as, grid size, start and goal locations, threat characteristics and various threat densities. A total threat density describes the fractional part of the total grid area to be covered by threats. That is, after all threats are generated, the sum of their areas is related to the total grid area by the total threat density. Since threats may overlap, a density of 1.0, for example, does not imply that the entire grid is covered. Each threat to be generated has an associated density which describes its fractional part of the total threat density. The sum of these individual threat densities must equal one.

Threat-Builder does not use a system random number generator. Rather, it employs it's own random number generator to simplify reproducing threat grids over various computer systems. The program outputs a file of records formatted for input to Path-Finder. The format for these records appears in Figure 5.1.

Path-Finder is structured to accept input parameters without regard to order, and to allow multiple case executions against the same threat grid using different parameters. A threat grid is defined by the 'G' (grid) and 'T'

(threat) records. These may not be altered between multiple cases. The remaining parameters may be altered between cases: 'P' (endpoints), 'L' (leg characteristics), 'C' (coefficients). Distinct cases are delimited by an 'R' (run case) record. Figure 5.2 illustrates a sample input data set. Figure 5.3 is a sample output plot from Path-Finder. Threats are shaded according to the value of the Pd, i.e. higher Pd threats are slightly darker. Figure 5.4 is a sample output listing from Path-Finder.

Record Formats for Path-Finder Input:

G $\langle$x_limit$\rangle$ $\langle$y_limit$\rangle$ —Define grid size

T $\langle$type$\rangle$ $\langle$radius$\rangle$ $\langle$Pd$\rangle$ $\langle$quantity$\rangle$ —Define threat
    $\langle x_1 \rangle$ $\langle y_1 \rangle$ —Threat coordinates
    $\langle x_2 \rangle$ $\langle y_2 \rangle$
    $. . .$
    $\langle x_n \rangle$ $\langle y_n \rangle$

P $\langle$start$\rangle$ $\langle$goal$\rangle$ —Start/goal y-coordinates

L $\langle$leg_length$\rangle$ $\langle$legs_per_arc$\rangle$ $\langle$path_arc(degrees)$\rangle$
                                        —Leg parameters

C $\langle$g$\rangle$ $\langle$g_threat$\rangle$ $\langle$g_length$\rangle$ $\langle$h$\rangle$ $\langle$h_threat$\rangle$ $\langle$h_length$\rangle$
                                        —Define g/h coefficients

R   —Run case

Figure 5.1

Path-Finder Input Formats

```
G  100  50
T  1  5  0.5  4
        73.3343 16.0923
        8.3552   23.0032
        10.0231 44.7446
        81.0377 34.9732
T  2  10  0.25  4
        20.7748  35.9103
        93.0213  17.4902
        33.2993   9.2618
        51.0289  21.7718
P  12.5  12.5
L  7.5  5  90
C  1  50  1  1  1  1
R
C  1  60  1  1  1  1
R
```

Figure 5.2

Sample Path-Finder Input
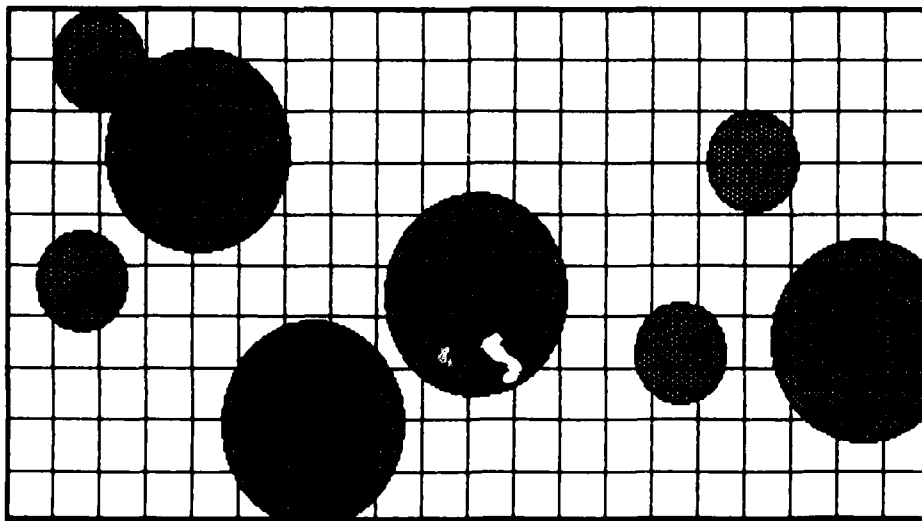
Figure 5.3

Threat Grid from Sample Input

```
*** Path Finder Results ***

Grid limits:   50   25

Start coordinates:   0.00, 12.00
Goal  coordinates:  50.00, 12.00

Leg generation options:
     Leg length:  5.00    Legs per arc:  5    Path arc: 90.00

G coefficients (g - threat - length):   1.00 -  20.00 -   1.00
H coefficients (h - threat - length):   1.00 -   1.00 -   1.00

Threats:
     Category:   1  Radius:  4.00  Pd:  0.30  Quantity:    9

     27.0782,   0.6397      18.1473, 15.9954      15.3320,   6.8943
     11.1130,   3.7563      26.8768, 15.5170      31.6666,   1.5026
     10.9314, 18.3048       35.2768, 16.6561      29.2145, 13.3045


     Category:   2  Radius:  3.00  Pd:  0.10  Quantity:   16

     39.9124, 12.8887        5.9448,  4.8130      26.0422, 10.1223
     27.8412,  3.3772       36.6348,  9.5287      44.1223,  0.7938
     27.1591, 18.5299       16.5070, 20.1099      15.5838,  0.7977
     19.2139, 15.6223       32.3776,  1.2539      38.9618, 22.8779
     20.5093, 21.0705       24.9695,  8.6735      35.4477, 17.6693
     38.9557, 21.0560

Path from goal:
     50.0000, 12.0000
     45.9293, 10.0866
     42.3937,  6.5510
     37.7743,  4.6376
     33.1549,  6.5510
     28.1549,  6.5510
     23.1549,  6.5510
     19.6194, 10.0866
     15.0000, 12.0000
     10.0000, 12.0000
      5.0000, 12.0000
      0.0000, 12.0000

Statistics:
     Nodes expanded   :        75
     CPU time (msecs):        8740
     Conflict cost    :        0.10
     Path length      :       54.50
     Martelli count   :         5
```

Figure 5.4

Sample Path-Finder Output Listing
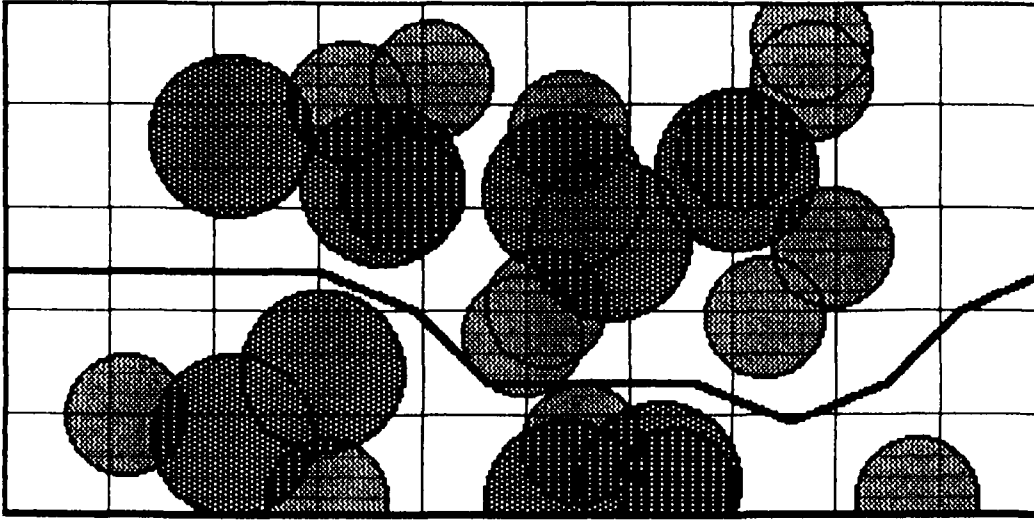
```
Path Length   =   54.50
Leg  Length   =    5.00
Conflict Cost =    0.10
```

Figure 6.7
Path Generated – Test Grid 4

DISCUSSION OF TEST GRID 4

This grid was developed to test the program against a different size grid and threat configuration. Note the jagged path near the goal due to the inability of the leg generator to create a leg at the exact angle necessary for a straight path. Table 6.6 reveals that the same path, with approximately the same node expansions, was developed by each coefficient configuration. This would seem to indicate that the solution was relatively 'easy' for the algorithm to find. Examining the output plot, Figure 6.7, bears out this statement since the solution path is rather obvious and an alternative path, with the same costs, is not readily apparent.

SEED: 31583

TOTAL THREAT DENSITY: 0.75

THREAT DATA: (radius / Pd / density)
4 / 0.3 / 0.5   3 / 0.1 / 0.5

LEG LENGTH: 5   LEGS-PER-ARC: 5   PATH DEFLECTION ANGLE: 90

| COEFFICIENTS | | | | | | | | CONFLICT | PATH |
G	I	L	H	I	L	NODES	CPU	COST	LENGTH
1	20	1	1	1	1	75	9	0.10	54.50
1	30	1	1	1	1	72	9	0.10	54.50
1	40	1	1	1	1	71	8	0.10	54.50
1	50	1	1	1	1	73	9	0.10	54.50
1	60	1	1	1	1	73	9	0.10	54.50
1	70	1	1	1	1	73	9	0.10	54.50
1	80	1	1	1	1	73	9	0.10	54.50

Table 6.4

Results for Test Grid 4
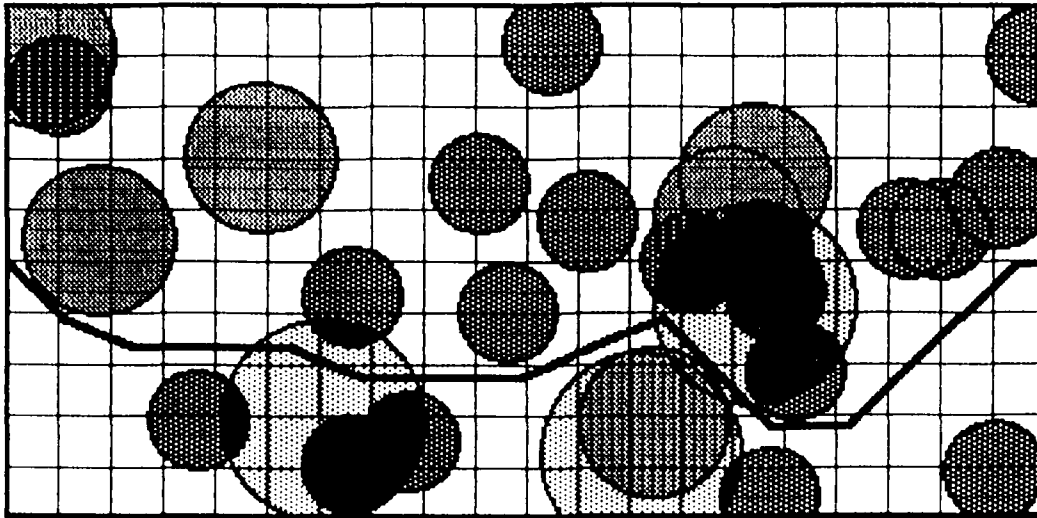
Path Length   = 115.46
Leg Length   =   7.50
Conflict Cost =   0.20

Figure 6.5
Path Generated – Test Grid 3.3
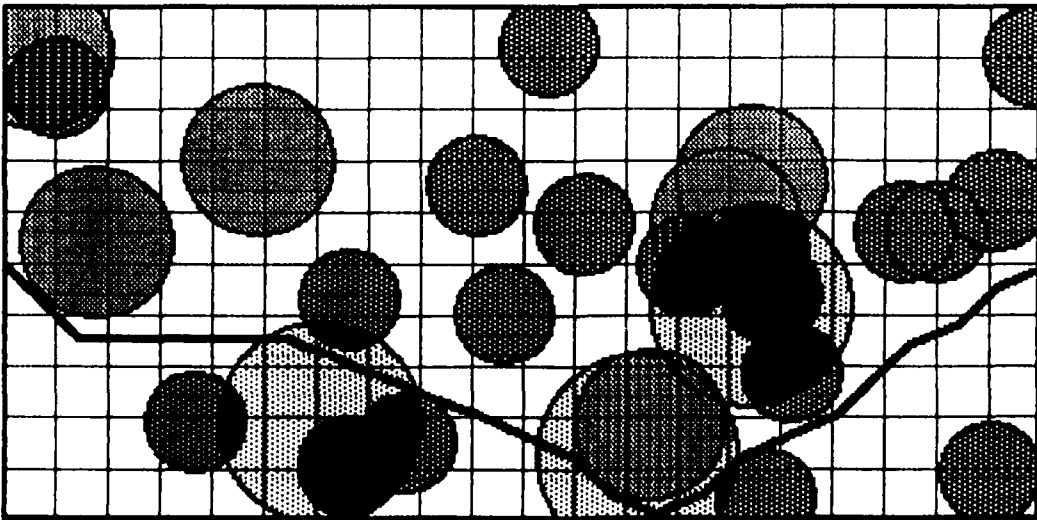
Path Length   = 114.96
Leg Length   =   5.00
Conflict Cost =   0.20

Figure 6.6
Path Generated – Test Grid 3.4
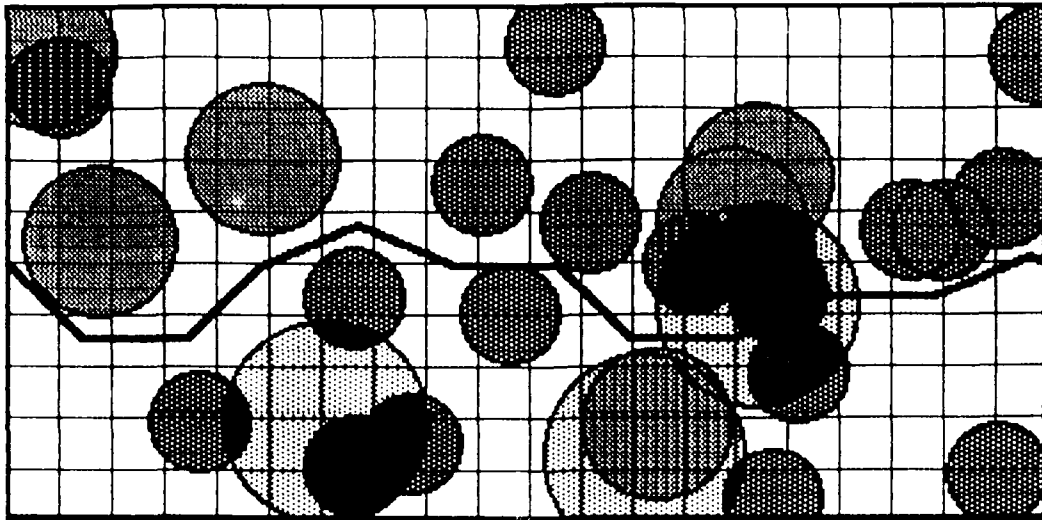
Path Length   = 111.92
Leg Length   =   10.00
Conflict Cost =   0.60

Figure 6.3
Path Generated — Test Grid 3.1

Path Length   = 120.62
Leg Length   =   10.00
Conflict Cost =   0.20

Figure 6.4
Path Generated — Test Grid 3.2

SEED: 46137

TOTAL THREAT DENSITY: 0.75

THREAT DATA: (radius / Pd / density)
      5 / 0.4 / 0.4   7.5 / 0.25 / 0.3   10 / 0.1 / 0.3

LEG LENGTH: 10   LEGS-PER-ARC: 5   PATH DEFLECTION ANGLE: 90

| COEFFICIENTS | | | | | | | | CONFLICT | PATH |
G	I	L	H	I	L	NODES	CPU	COST	LENGTH
1	20	1	1	1	1	436	89	0.50	111.92
1	30	1	1	1	1	468	94	0.20	120.62
1	40	1	1	1	1	312	52	0.20	120.62
1	50	1	1	1	1	269	41	0.20	120.62
1	60	1	1	1	1	343	57	0.20	120.62
1	70	1	1	1	1	357	59	0.20	120.62
1	50	1	1	1	0	1238	260	0.20	120.62
1	70	1	1	1	0	1058	225	0.20	120.62
1	50	1	0	0	0	1253	243	0.20	120.62
0	0	0	1	50	1	12	0.6	1.90	108.57

LEG LENGTH: 7.5

1	50	1	1	1	1	923	333	0.20	115.46

LEG LENGTH: 5

1	50	1	1	1	1	6189	9185	0.20	114.96

Table 6.3

Results for Test Grid 3

Figures 6.4 best solution path created with a leg length of 10. Notice that the path is more circuitous than Figure 6.3 as path length is sacrificed to avaoid threat encounters. Figures 6.5 and 6.6 represent the paths discovered using the coefficients 1-50-1-1-1-1 and leg length of 7.5 and 5, respectively. Note the significant increse in node expansions, 269 to 923 to 6189, as the leg length is decreased from 10 to 7.5 to 5. These increased expansions reflect the exponential growth in the size of the search tree. The solution improved, though, by avoiding a large threat which could not be avoided with a longer leg length. The advantage, intuitively, is that the shorter leg lengths increase the maneuverability. Also, the length 5 solution must never be worse than length 10 solution. This is true because every length 10 leg can be duplicated by two, length 5 legs.

DISCUSSION OF TEST GRID 3

This test followed the first examples with the suggestion that the coefficient configuration, 1-n-1-1-1-1, where n is a multiple of ten, produces reasonably good results. This trend serves as the foundation for all the remaining tests.

Several interesting points can be made about this test series. It is obvious that the length component of the heuristic estimate is quite beneficial to reducing the size of the search. The cases where the heuristic estimate is 'turned-off' (h-coefficients set to zero) gives some idea of the maximum search length. This is because the search becomes breadth-first without the heuristic. This is the approximate size of the effort that the SNOOPER model will invest in a solution. The number of nodes expanded are not equal since each case will not necessarily expand the same nodes in the same order due to the value of the g-coefficients. Also, unlike SNOOPER, the algorithm terminates on the first solution encountered. The breadth-first search with the g-coefficients, 1-50-1, is 1249 nodes. The improved A* algorithm only required 269 nodes to discover the same solution. This clearly demonstrates the benefit of heuristic search over uninformed search. Finally, the case with the g-coefficients set to zero gives an idea of the effort that the FPG model would invest in its solution. The path found by this case is not the same as would be found by FPG, however. The similarity is in the number of nodes expanded and therefore, the relative effort expended. Figure 6.3 illustrates the path discovered using a low threat-length ratio. Using an algorithm similar to the FPG method (built during the early stages of this thesis), exactly the same path is discovered!
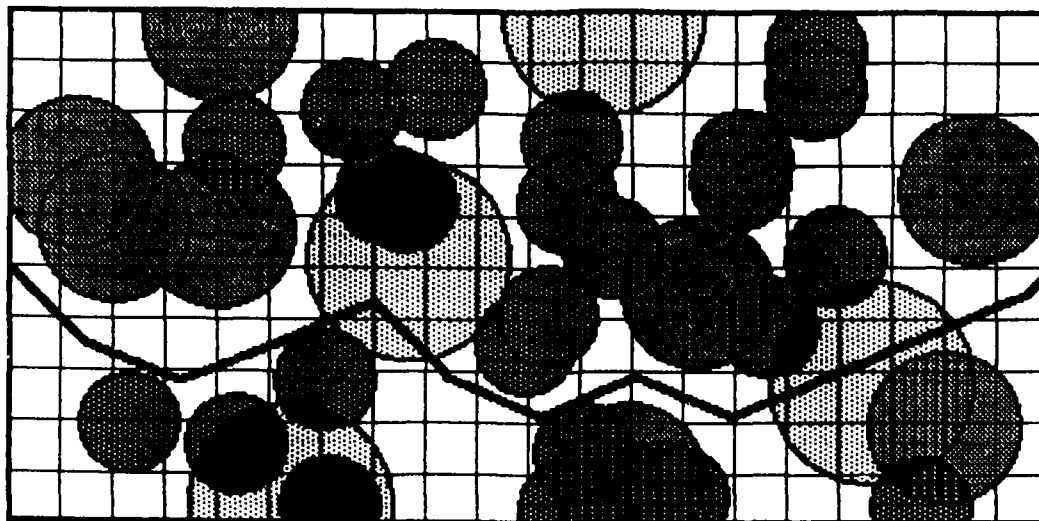
```
Path Length   =  113.80
Leg  Length   =   10.00
Conflict Cost =    0.20
```

Figure 6.1
Path Generated — Test Grid 1
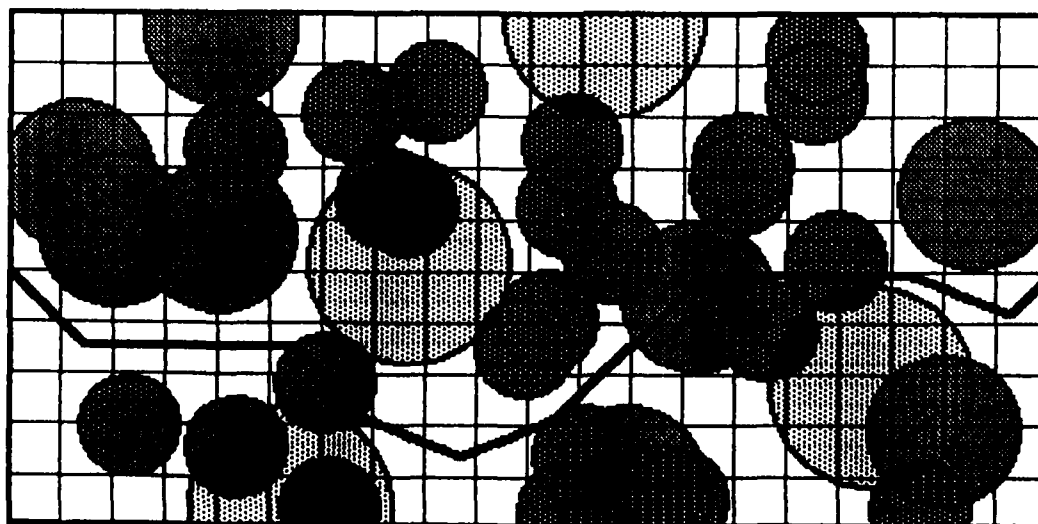

```
Path Length   =  115.54
Leg  Length   =   10.00
Conflict Cost =    0.45
```

Figure 6.2
Path Generated — Test Grid 2

SEED: 31583

TOTAL THREAT DENSITY: 1.0

THREAT DATA: (radius / Pd / density)
5 / 0.1 / 0.4   7.5 / 0.25 / 0.3   10 / 0.4 / 0.3

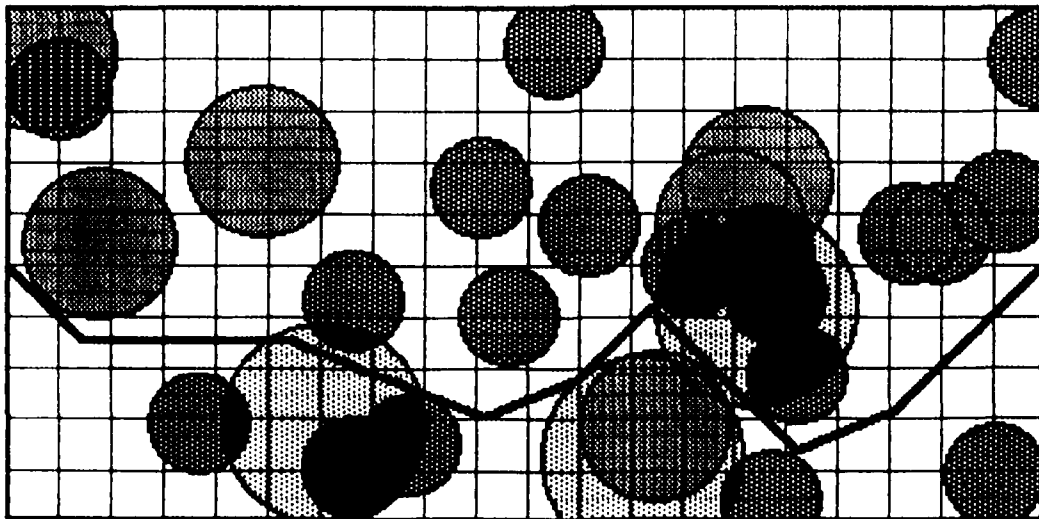LEG LENGTH: 10   LEGS-PER-ARC: 5   PATH DEFLECTION ANGLE: 90

| COEFFICIENTS | | | | | | | | CONFLICT | PATH |
G	I	L	H	I	L	NODES	CPU	COST	LENGTH
1	30	1	1	1	1	352	73	0.55	109.00
1	40	1	1	1	1	967	241	0.55	109.00
1	50	1	1	1	1	473	108	0.55	109.00
1	60	1	1	1	1	449	101	0.55	109.00
1	70	1	1	1	.	396	85	0.45	115.54
1	30	2	1	1	1	1465	314	0.60	107.48
1	50	2	1	1	1	1403	322	0.55	109.00
1	70	2	1	1	1	1236	302	0.55	109.00
1	30	3	1	1	1	1809	343	0.70	106.14
1	50	3	1	1	1	1633	335	0.60	107.48
1	70	3	1	1	1	1600	341	0.55	109.00
1	50	1	1	10	1	247	46	0.60	107.48
1	50	1	1	30	1	305	41	0.55	109.00
1	50	1	1	50	1	991	164	0.55	109.00
1	50	1	1	70	1	1231	242	0.55	109.00

Table 6.2

Results for Test Grid 2

SEED: 31583

TOTAL THREAT DENSITY: 1.0

THREAT DATA: (radius / Pd / density)
    5 / 0.4 / 0.4   7.5 / 0.25 / 0.3   10 / 0.1 / 0.3

LEG LENGTH: 10   LEGS-PER-ARC: 5   PATH DEFLECTION ANGLE: 90

| | COEFFICIENTS | | | | | | | CONFLICT | PATH |
G	I	L	H	I	L	NODES	CPU	COST	LENGTH
1	30	1	1	1	1	100	16	0.20	113.80
1	40	1	1	1	1	82	12	0.20	113.80
1	50	1	1	1	1	80	12	0.20	113.80
1	60	1	1	1	1	89	14	0.20	113.80
1	70	1	1	1	1	81	12	0.20	115.24
1	30	2	1	1	1	1157	266	0.20	113.80
1	50	2	1	1	1	742	171	0.20	113.80
1	70	2	1	1	1	510	109	0.20	113.80
1	30	3	1	1	1	1516	320	0.75	106.56
1	50	3	1	1	1	1235	280	0.20	113.80
1	70	3	1	1	1	1011	233	0.20	113.80
1	50	1	1	10	1	27	3	0.20	116.13
1	50	1	1	30	1	128	15	0.20	116.13
1	50	1	1	50	1	43	4	1.10	109.00
1	50	1	1	70	1	143	18	0.45	116.13
1	50	1	1	1	0	793	182	0.20	113.80
1	50	1	1	10	0	666	147	0.20	113.80
1	50	1	1	30	0	337	56	0.20	113.80
1	50	1	1	50	0	68	7	0.20	116.13
1	50	1	1	70	0	66	6	0.85	113.32

Table 6.1

Results for Test Grid 1

'jaggies' is introduced. That is, due to the fixed length of each leg, the path is often jagged where it 'sets itself up' to maneuver through an opening. For example, the jagged path through the first, small threat is likely necessary to penetrate the next opening in preparation for the final manuevers. This problem might be reduced by using shorter legs, or increasing the number of legs and the path deflection angle.

As stated earlier, test grids 1 and 2 are identical in layout but different in Pd assignments. Note that in test grid 1, the solution path favored the larger, lower Pd threats. And, in grid 2, the smaller threats were selected since they have the smaller Pd values.

coefficients, the more threat avoidance will be favored over short paths. Likewise, the less this ratio, the more the algorithm is likely to 'sacrifice' a threat interaction in favor of a shorter path. This behavior, though, is not indicative of every case tested.

There is a marked decrease in efficiency when the g-length coefficient is increased relative to the g-threat coefficient such that the same solution is discovered. This change demonstrates the sensitivity of the algorithm to the relationship between these two coefficients.

Some rather curious results occured when the heuristic coefficient for threat is increased while the leg coefficient is held to one. The algorithm seems to be misguided into selecting a poor solution. This shows the importance of the accuracy of the heuristic relative to both efficiency and quality of solution. Finally, it would seem reasonable to consider a heuristic that does not consider length as a factor. This is tested by setting the h-length coefficient to zero. The quality of the solutions are consistent with their counterparts of h-length equal to one. However, there is still an increase in the number of nodes expanded.

The results for test grid 2 again illustrate the lower bound upon the threat-length coefficient ratio which will produce results consistent with the primary goal of minimal threat interaction. Also the upper threshold is demonstrated by the improvement in the solution path as the coefficients increase to 1-90-1-1-1-1. When the h-threat coefficients are manipulated as they were in the previous example, this test does not have as marked fluctuations in solution quality.

In examining the output plots, especially Figure 6.2, the problem of

VAX PASCAL 'clock' function. Conflict cost is the sum of the Pd's of threats intersected along the path. Finally, path length is expressed in grid units.

Coordinate reference lines are drawn in each output plot every five units. Each of the output plots is related by path length and leg length to a set of coefficients associated results table. For example, the output plot in Figure 6.1, Path Generated – Test Grid 1, represents every case in Table 6.1, Results for Test Grid 1, with a path length of 113.80.

DISCUSSION OF TEST GRIDS 1 AND 2

These threat grids are identical except for the Pd values assigned to the threats. Grid 1 used the Pd values, 0.4 – 0.25 – 0.10, for the small, medium and large radius threats, respectively. Grid 2 reversed the Pd values of the small and large threats.

The results for test grid 1 illustrate a couple of trends that were evident in many later tests. Examining the results in Table 6.1, the balance in importance represented by the g_threat and g_length coefficients, clearly affect the solution. For example, coefficients 1-60-1-1-1-1 produce the apparent 'best' solution of path length 113.80 with a conflict cost of 0.20. However, a slight shift of the relationship to 1-70-1-1-1-1, produces a slightly longer path, 115.24, with the same conflict cost. A similar case exists between coefficients 1-30-3-1-1-1 and 1-40-3-1-1-1. In this second case, the higher g-threat coefficient caused an improvement in the solution in terms of conflict cost, the primary goal. This behavior is quite explainable intuitively. These coefficients represent the relative importance of conflict cost vs. path length. The greater the ratio between the g-threat and g-length

VI. PATH-FINDER TEST RESULTS

The test results presented in this chapter can be divided into to sets.
The first set, test grids 1 through 4, were the developed during the second
phase of system testing using the 'straight-line' heuristic in the improved A*
algorithm. The primary goal of this phase of testing was to examine the effect
of coefficients upon the overall goal of minimum conflict cost over the
shortest possible path length. A second set of test results, test grids 5
through 10, were run as a final series of tests using a reasonable set of
coefficients. This series of tests were run using both heuristics mentioned in
the previous chapter. The details will be discussed in the applicable
sections.

DESCRIPTION OF TABLES AND GRAPHS

Each table includes the necessary data to recreate the threat grid using
the Threat-Builder program. The coefficients, G I L , represent the
g_coeff, g_threat_coeff, and the g_length_coeff, respectively. Similarly for
the h-coefficients. In the following text these coefficients will be expressed
as: G-T-L-H-T-L (for example: 1-50-1-1-1-1). The nodes column represents
the total number of nodes expanded during the execution of the case. The CPU
time listed, in seconds, is an approximation of the elapsed CPU time
following data input, until the results are output. It is computed using the

30

ORIENTATION OF TEST GRIDS 5 - 10

This series of tests were run as 'final' test cases against the algorithm. The results for each test were compiled using three, different heuristics. The first heuristic is the same 'straight-line' heuristic and improved A* algorithm of the previous tests. This heuristic will be termed 'Improved A*' in the result tables. The second heuristic is the 'box' heuristic discussed in the last chapter. Early tests against this heuristic produced the same results as the first heuristic. Believing this to be due to the effects of the Martelli improvement, this heuristic is run using the standard A* algorithm and identified as 'A* w/box' in the results tables. Finally, in order to ccmpare the results with the straight-line method, the last heuristic is the straight-line heuristic, also without the Martelli improvement to the algorithm. This will be listed as 'A* w/straight-line' in the tables.

A series of coefficients, known to produce reasonable results, are run against each heuristic for a test grid to examine performance. Additional cases are run where results appear interesting. Both heuristic (informed) search, and a limited, breadth-first (uninformed) search are tested.

DISCUSSION OF TEST GRID 5

This test case is extremely interesting due to the threat density of 2.0. This density, with the given threat characteristics, create a grid with 75, overlapping threats. Examining the output graph, Figure 6.8, clearly shows the impact of this threat density. It is also apparent that determining a 'good' path manually might be very difficult in this case.

Table 6.5 displays the results using the improved A* search and

straight-line heuristic. Notice that all of the coefficients tested with a leg length of 10 yielded exactly the same solution with only slightly different numbers of node expansions. This pattern will prevail in all the test cases which follow. A couple of cases were tested with relatively high g-threat values, 100 and 500, to see if a slightly longer path might be produced as it was in test grid 1. However, the characteristics of this grid are such that this phenomena does not occur. Another interesting feature is that the informed search, coefficients 1-x-1-1-1-1, expanded more nodes as the g-threat coefficient increased. And, the uninformed search, h-coefficients set to zero, expanded fewer nodes as the g-threat coefficient increased. Again, this pattern will dominate in the following test grids. In all the tests, only the improved A* algorithm was selected for a run with leg length of 5 due to the large CPU requirement. In this test, the length 5 case required only 1650 seconds, or slightly more than 27 minutes. This is relatively fast compared with some of the tests which follow.

Comparing the results in Tables 6.6 and 6.7 show that the box heuristic, provides a slightly better estimate of actual cost than the straight-line heuristic. This is evident from the decrease in nodes expanded in the box heuristic cases.

The claim was made in the discussion of the Martelli improvement to the A* algorithm, that the improvement would guarantee that fewer nodes are expanded when an inconsistent heuristic is used, compared with the standard A* algorithm. That the heuristics used here are inconsistent can be seen intuitively or be verified by a non-zero 'Martelli count' in the output listing. A question is raised, then, concerning this claim when the node count of the leg

length 7.5 cases are checked in Tables 6.5 and 6.7. Remember that these are the same straight-line estimates, with and without the Martelli improvement, respectively. The same coefficients and therefore the same heuristc estimator is used in each case. The node count for the improved algorithm is slightly greater than for the standard algorithm, a violation of the claim! This happens since nodes are not doubly counted as 'expanded' when duplicates are discovered. The algorithm should still be more efficient, based upon claim, although this node count may be misleading.

This test included cases run with different leg generation parameters for number of legs and path deflection angle. The first cases tested were generating seven legs over 110 degrees, and generating nine legs over 130 degrees. Neither of these cases includes the same possible legs generated by the original, length-five test. It is a bit surprising that there is not a significant improvement in the solution quality in either case. In fact, while the seven-leg case is a slight improvement in conflict cost, the nine-leg case fails to improve the conflict cost at all. And, it builds a longer path than the original solution set! In addition, the nine-leg test used almost four hours of CPU time finding a solution worse than the five-leg test did in less than a minute! In order to test for improvement when the original legs are included, a case was run with seven legs over 135 degrees. This seven-leg test generated exactly the same path as its five-leg counterpart and, it used ten times the CPU time! Also, it failed to find a path as 'good' as the other seven-leg case. This indicates that there is a great deal of sensitivity to leg generation characteristics relative to solution quality.

The output graphs, Figures 6.8 , 6.9, and 6.10, depict the solution paths

generated at leg lengths 10, 7.5 and 5, respectively. Of interest is how the increased maneuverability of shorter leg lengths allow the algorithm to discover paths which 'squeeze through the cracks'. There is an especially significant improvement in the conflict cost when the leg length is changed from 10 to 5. The last two graphs, Figures 6.11 and 6.12, reflect the paths generated with the seven and nine-leg characteristics described above.

SEED: 143954

TOTAL THREAT DENSITY: 2.0

THREAT DATA: (radius / Pd / density)
5 / 0.5 / 0.4   7.5 / 0.25 / 0.3   10 / 0.1 / 0.3

LEG LENGTH: 10   LEGS-PER-ARC: 5   PATH DEFLECTION ANGLE: 90

| COEFFICIENTS | | | | | | | | CONFLICT | PATH |
G	I	L	H	I	L	NODES	CPU	COST	LENGTH
1	30	1	1	1	1	181	46	2.15	108.87
1	40	1	1	1	1	185	47	2.15	108.87
1	50	1	1	1	1	189	48	2.15	108.87
1	60	1	1	1	1	193	49	2.15	108.87
1	70	1	1	1	1	196	50	2.15	108.87
1	30	1	0	0	0	756	192	2.15	108.87
1	50	1	0	0	0	498	118	2.15	108.87
1	70	1	0	0	0	420	95	2.15	108.87

LEG LENGTH: 7.5

| 1 | 50 | 1 | 1 | 1 | 1 | 623 | 233 | 1.95 | 115.43 |

LEG LENGTH: 5

| 1 | 50 | 1 | 1 | 1 | 1 | 2310 | 1650 | 1.65 | 112.58 |

LEG LENGTH: 10   LEGS-PER-ARC: 7   PATH DEFLECTION ANGLE: 110

| 1 | 50 | 1 | 1 | 1 | 1 | 597 | 392 | 2.00 | 109.77 |

LEG LENGTH: 10   LEGS-PER-ARC: 7   PATH DEFLECTION ANGLE: 135

| 1 | 50 | 1 | 1 | 1 | 1 | 792 | 597 | 2.15 | 108.87 |

LEG LENGTH: 10   LEGS-PER-ARC: 9   PATH DEFLECTION ANGLE: 130

| 1 | 50 | 1 | 1 | 1 | 1 | 3625 | 13303 | 2.15 | 119.33 |

Table 6.5

Results for Test Grid 5 — Improved A*

SEED: 143954

TOTAL THREAT DENSITY: 2.0

THREAT DATA: (radius / Pd / density)
5 / 0.5 / 0.4   7.5 / 0.25 / 0.3   10 / 0.1 / 0.3

LEG LENGTH: 10   LEGS-PER-ARC: 5   PATH DEFLECTION ANGLE: 90

| | COEFFICIENTS | | | | | | | CONFLICT | PATH |
G	I	L	H	I	L	NODES	CPU	COST	LENGTH
1	30	1	1	1	1	176	45	2.15	108.87
1	40	1	1	1	1	181	47	2.15	108.87
1	50	1	1	1	1	188	48	2.15	108.87
1	60	1	1	1	1	189	49	2.15	108.87
1	70	1	1	1	1	195	50	2.15	108.87

LEG LENGTH: 7.5

1	50	1	1	1	1	603	225	1.95	115.43

Table 6.6

Results for Test Grid 5 – A* w/box

<u>SEED</u>: 143954

<u>TOTAL THREAT DENSITY</u>: 2.0

<u>THREAT DATA</u>: (radius / Pd / density)
5 / 0.5 / 0.4   7.5 / 0.25 / 0.3   10 / 0.1 / 0.3

<u>LEG LENGTH</u>: 10   <u>LEGS-PER-ARC</u>: 5   <u>PATH DEFLECTION ANGLE</u>: 90

<u>COEFFICIENTS</u>								<u>CONFLICT</u>	<u>PATH</u>
<u>G</u>	<u>I</u>	<u>L</u>	<u>H</u>	<u>I</u>	<u>L</u>	<u>NODES</u>	<u>CPU</u>	<u>COST</u>	<u>LENGTH</u>
1	30	1	1	1	1	181	46	2.15	108.87
1	40	1	1	1	1	185	47	2.15	108.87
1	50	1	1	1	1	189	48	2.15	108.87
1	60	1	1	1	1	193	49	2.15	108.87
1	70	1	1	1	1	196	50	2.15	108.87

<u>LEG LENGTH</u>: 7.5

| 1 | 50 | 1 | 1 | 1 | 1 | 619 | 232 | 1.95 | 115.43 |

Table 6.7

Results for Test Grid 5 - A* w/straight-line
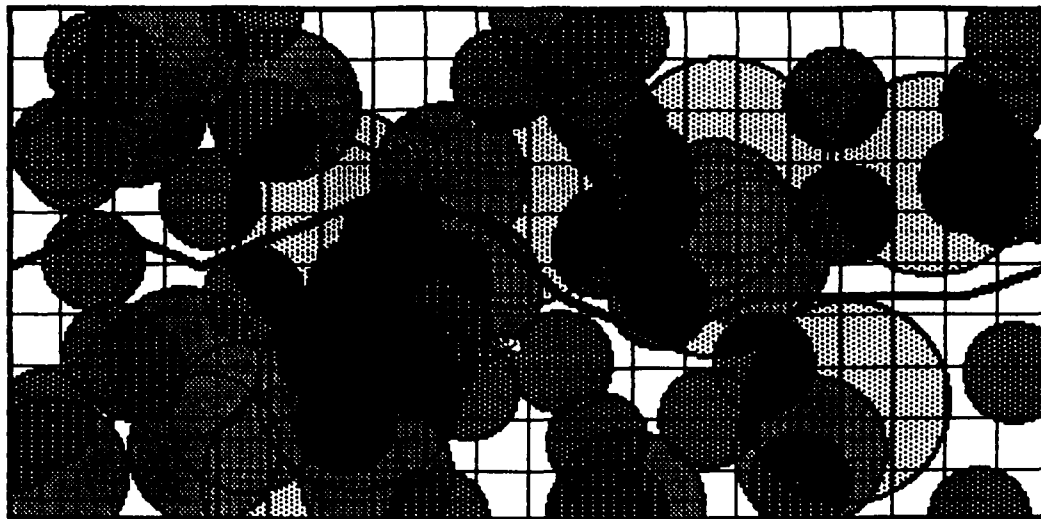
Path Length = 108.87
Leg Length = 10.00
Conflict Cost = 2.15

Figure 6.8
Path Generated – Test Grid 5.1

Path Length = 115.43
Leg Length = 7.50
Conflict Cost = 1.95

Figure 6.9
Path Generated – Test Grid 5.2
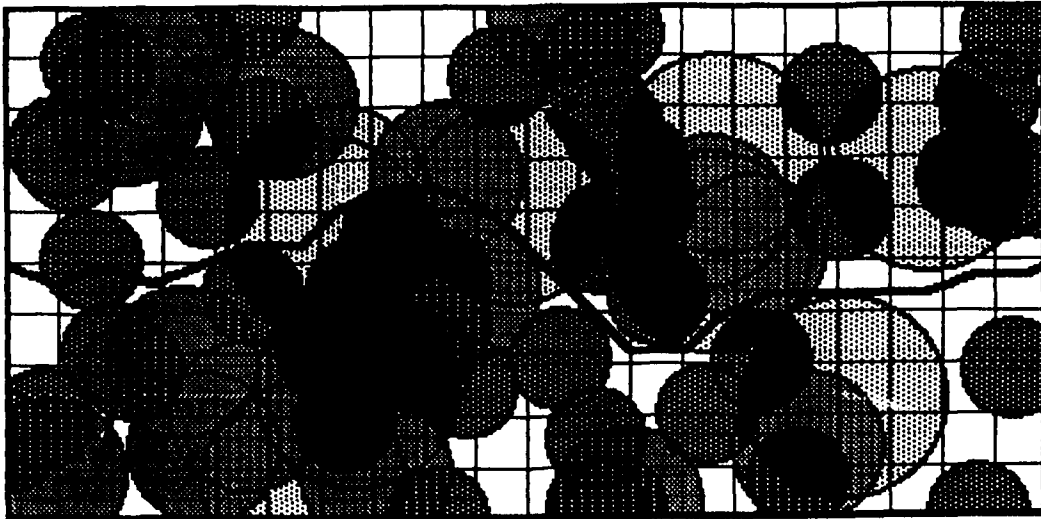
Path Length   = 112.58
Leg Length   =   5.00
Conflict Cost =   1.65

Figure 6.10
Path Generated – Test Grid 5.3

Path Length   = 109.77
Leg Length   =   10.00
Conflict Cost =   2.00

Figure 6.11
Path Generated – Test Grid 5.4
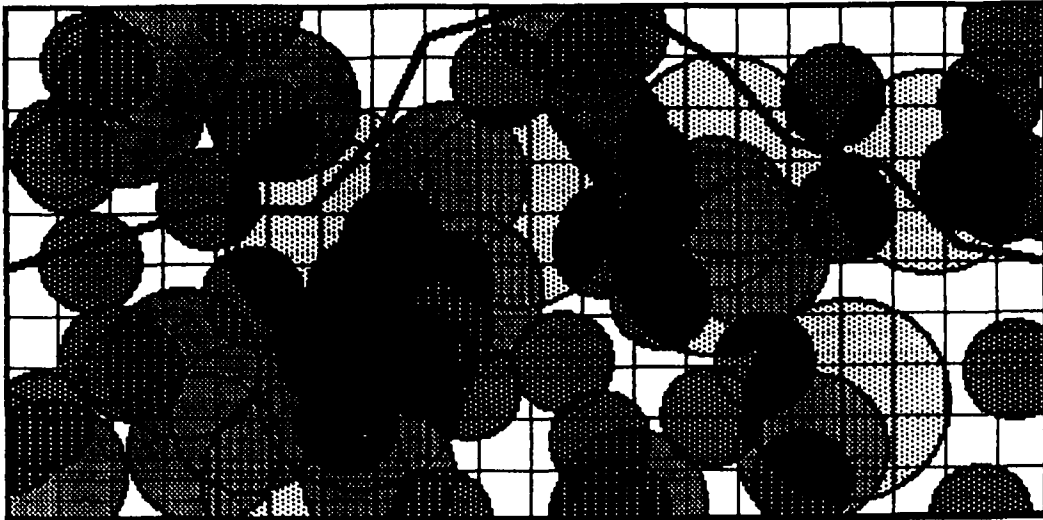
Path Length   = 119.33
Leg Length   =   10.00
Conflict Cost =   2.15

Figure 6.12
Path Generated - Test Grid 5.5

DISCUSSION OF TEST GRID 6

This test grid is generated using a total threat density of 1.5. Though this density is less than the previous test, the nodes expanded in each case is more than five times greater. This is an indication that the 'difficulty' of the problem is related to the arrangement rather than the quantity of threats. In fact, the CPU requirements are so extensive that the length 5 case could not execute within the four-hour CPU limit imposed upon test cases. Where the length 7.5 case of test grid 5 required only 23 seconds, the same case in this test required over 63 minutes!

Figures 6.13 and 6.14 represent the solution paths created at leg length 10 and 7.5. Notice the radical difference in the early part of the paths. A careful examination of the length 10 plot will reveal how the length of the leg would not permit an equivalent 'southern' route chosen for length 7.5.

SEED: 21738

TOTAL THREAT DENSITY: 1.5

THREAT DATA: (radius / Pd / density)
5 / 0.5 /0.4   7.5 / 0.25 / 0.3   10 / 0.1 / 0.3

LEG LENGTH: 10   LEGS-PER-ARC: 5   PATH DEFLECTION ANGLE: 90

COEFFICIENTS								CONFLICT	PATH
G	T	L	H	T	L	NODES	CPU	COST	LENGTH
1	30	1	1	1	1	991	287	2.20	113.40
1	40	1	1	1	1	1008	286	2.20	113.40
1	50	1	1	1	1	1020	286	2.20	113.40
1	60	1	1	1	1	1024	285	2.20	113.40
1	70	1	1	1	1	1021	282	2.20	113.40
1	30	1	0	0	0	1750	434	2.20	113.40
1	50	1	0	0	0	1601	422	2.20	113.40
1	70	1	0	0	0	1475	399	2.20	113.40

LEG LENGTH: 7.5

| 1 | 50 | 1 | 1 | 1 | 1 | 4782 | 4102 | 2.10 | 111.91 |

Table 6.8

Results for Test Grid 6 – Improved A*

SEED: 21738

TOTAL THREAT DENSITY: 1.5

THREAT DATA: (radius / Pd / density)
5 / 0.5 /0.4   7.5 / 0.25 / 0.3   10 / 0.1 / 0.3

LEG LENGTH: 10   LEGS-PER-ARC: 5   PATH DEFLECTION ANGLE: 90

COEFFICIENTS								CONFLICT	PATH
G	I	L	H	I	L	NODES	CPU	COST	LENGTH
1	30	1	1	1	1	968	280	2.20	113.40
1	40	1	1	1	1	984	280	2.20	113.40
1	50	1	1	1	1	1000	282	2.20	113.40
1	60	1	1	1	1	1019	288	2.20	113.40
1	70	1	1	1	1	1018	285	2.20	113.40

LEG LENGTH: 7.5

1	50	1	1	1	1	4749	4123	2.10	111.91

Table 6.9

Results for Test Grid 6 – A* w/box

SEED: 21738

TOTAL THREAT DENSITY: 1.5

THREAT DATA: (radius / Pd / density)
     5 / 0.5 /0.4   7.5 / 0.25 / 0.3   10 / 0.1 / 0.3

LEG LENGTH: 10   LEGS-PER-ARC: 5   PATH DEFLECTION ANGLE: 90

	COEFFICIENTS							CONFLICT	PATH
G	I	L	H	I	L	NODES	CPU	COST	LENGTH
1	30	1	1	1	1	991	286	2.20	113.40
1	40	1	1	1	1	1008	285	2.20	113.40
1	50	1	1	1	1	1018	284	2.20	113.40
1	60	1	1	1	1	1024	284	2.20	113.40
1	70	1	1	1	1	1021	281	2.20	113.40

LEG LENGTH: 7.5

1	50	1	1	1	1	4749	4123	2.10	111.91

Table 6.10

Results for Test Grid 6 – A* w/straight-line
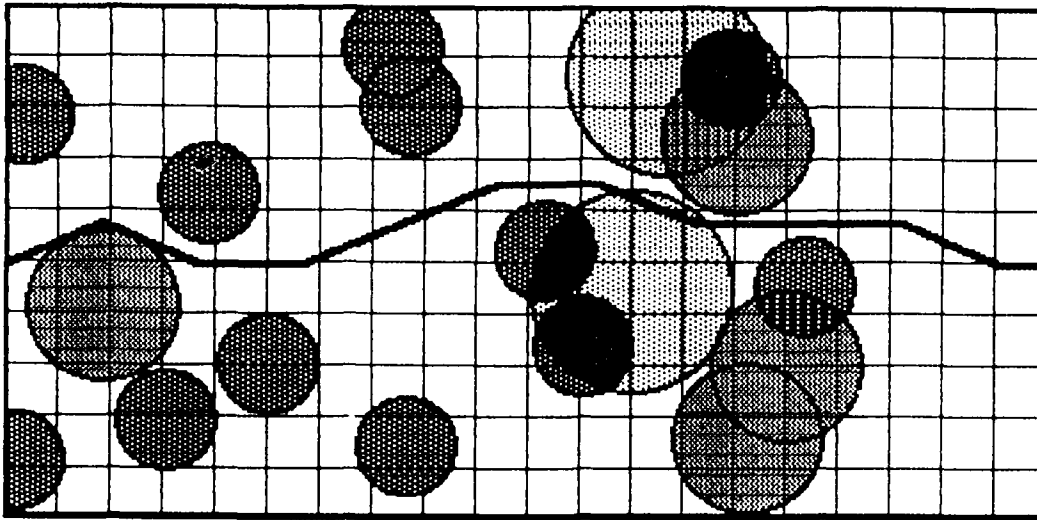
Path Length   = 104.57
Leg Length   =   10.00
Conflict Cost =   0.10

Figure 6.19
Path Generated — Test Grid 9.1
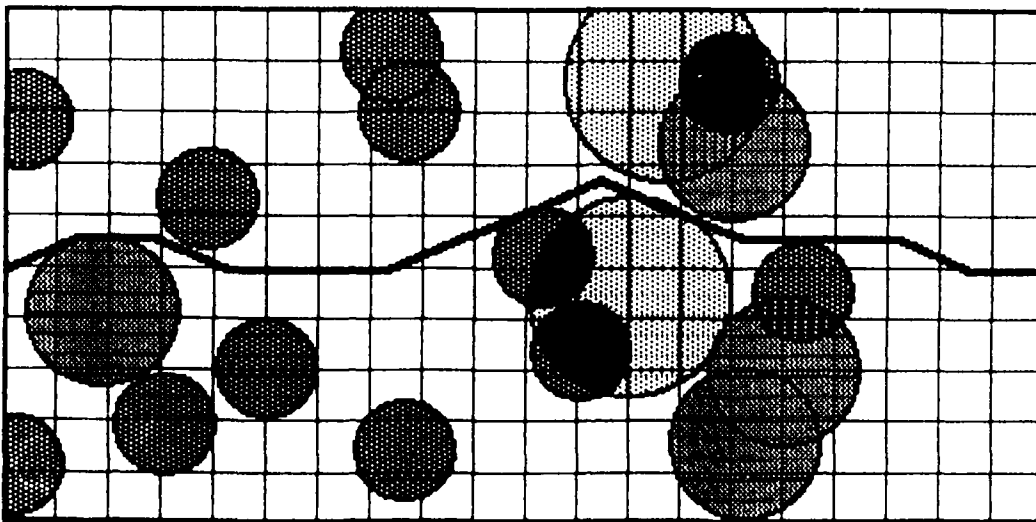
Path Length   = 104.57
Leg Length   =   7.50
Conflict Cost =   0.10

Figure 6.20
Path Generated — Test Grid 9.2

SEED: 28547

TOTAL THREAT DENSITY: 0.50

THREAT DATA: (radius / Pd / density)
    5 / 0.5 /0.4   7.5 / 0.25 / 0.3   10 / 0.1 / 0.3

LEG LENGTH: 10   LEGS-PER-ARC: 5   PATH DEFLECTION ANGLE: 90

COEFFICIENTS								CONFLICT	PATH
G	I	L	H	I	L	NODES	CPU	COST	LENGTH
1	30	1	1	1	1	47	4	0.10	104.57
1	40	1	1	1	1	64	6	0.10	104.57
1	50	1	1	1	1	74	7	0.10	104.57
1	60	1	1	1	1	85	8	0.10	104.57
1	70	1	1	1	1	99	10	0.10	104.57

LEG LENGTH: 7.5

1	50	1	1	1	1	293	44	0.10	104.57

Table 6.19

Results for Test Grid 9 — A* w/straight-line

SEED: 28547

TOTAL THREAT DENSITY: 0.5

THREAT DATA: (radius / Pd / density)
        5 / 0.5 / 0.4   7.5 / 0.25 / 0.3   10 / 0.1 / 0.3

LEG LENGTH: 10   LEGS-PER-ARC: 5   PATH DEFLECTION ANGLE: 90

| COEFFICIENTS | | | | | | | | CONFLICT | PATH |
G	I	L	H	I	L	NODES	CPU	COST	LENGTH
1	30	1	1	1	1	43	3	0.10	104.57
1	40	1	1	1	1	50	4	0.10	104.57
1	50	1	1	1	1	68	6	0.10	104.57
1	60	1	1	1	1	80	7	0.10	104.57
1	70	1	1	1	1	90	9	0.10	104.57

LEG LENGTH: 7.5

| 1 | 50 | 1 | 1 | 1 | 1 | 256 | 37 | 0.10 | 104.57 |

Table 6.18

Results for Test Grid 9 – A* w/box

DISCUSSION OF TEST GRID 9

The threat density of 0.50 creates only 18 threats in the grid. Given this,

and their position, solutions are found quite efficiently. For example, the

breadth-first search required 1506 nodes to find the same solution as the

heuristic search found in 47 nodes! There would also seem to be only one

'good' solution since the same one was found regardless of leg length. That

is, without changes in the path deflection arc to permit the negotiation of the

last threat gap (see Figure 6.19), the solution found is likely the best.

SEED: 28547

TOTAL THREAT DENSITY: 0.5

THREAT DATA: (radius / Pd / density)
   5 / 0.5 /0.4   7.5 / 0.25 / 0.3   10 / 0.1 / 0.3

LEG LENGTH: 10   LEGS-PER-ARC: 5   PATH DEFLECTION ANGLE: 90

| | COEFFICIENTS | | | | | | | CONFLICT | PATH |
G	I	L	H	I	L	NODES	CPU	COST	LENGTH
1	30	1	1	1	1	47	4	0.10	104.57
1	40	1	1	1	1	64	6	0.10	104.57
1	50	1	1	1	1	74	7	0.10	104.57
1	60	1	1	1	1	85	8	0.10	104.57
1	70	1	1	1	1	99	10	0.10	104.57
1	30	1	0	0	0	1506	276	0.10	104.57
1	50	1	0	0	0	1224	251	0.10	104.57
1	70	1	0	0	0	1050	218	0.10	104.57

LEG LENGTH: 7.5

| 1 | 50 | 1 | 1 | 1 | 1 | 295 | 44 | 0.10 | 104.57 |

LEG LENGTH: 5

| 1 | 50 | 1 | 1 | 1 | 1 | 1651 | 712 | 0.10 | 104.57 |

Table 6.17

Results for Test Grid 9 - Improved A*
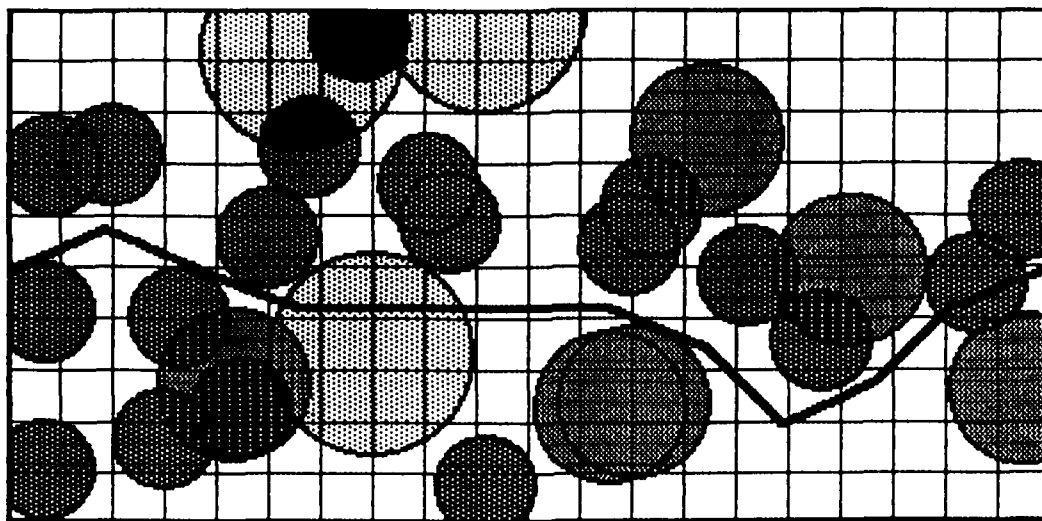
Path Length   = 110.43
Leg Length   =   10.00
Conflict Cost =   0.60

Figure 6.17
Path Generated – Test Grid 8.1
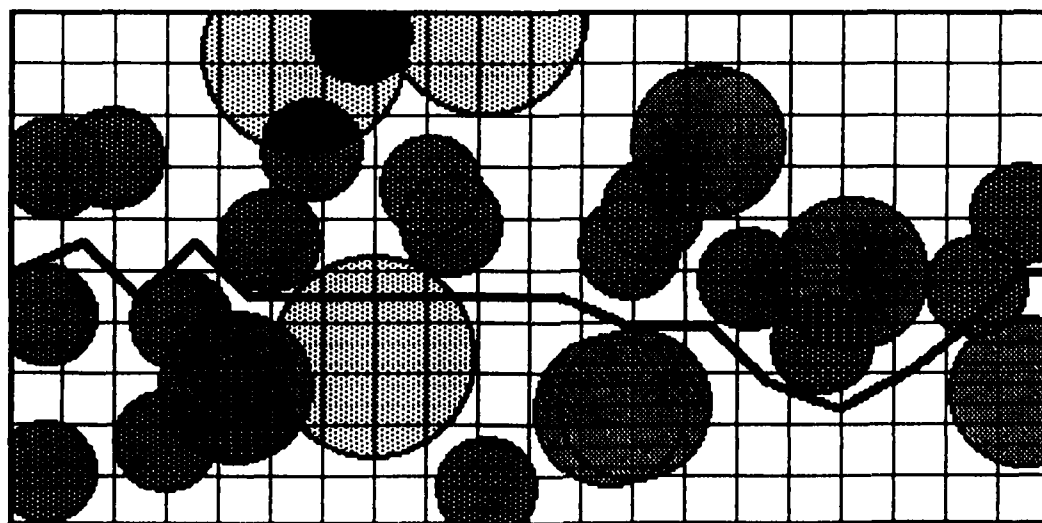
Path Length   ≈ 115.46
Leg Length   ≈   7.50
Conflict Cost ≈   0.60

Figure 6.18
Path Generated – Test Grid 8.2

SEED: 120775

TOTAL THREAT DENSITY: 0.75

THREAT DATA: (radius / Pd / density)
    5 / 0.5 /0.4   7.5 / 0.25 / 0.3   10 / 0.1 / 0.3

LEG LENGTH: 10   LEGS-PER-ARC: 5   PATH DEFLECTION ANGLE: 90

		COEFFICIENTS						CONFLICT	PATH
G	I	L	H	I	L	NODES	CPU	COST	LENGTH
1	30	1	1	1	1	269	39	0.60	110.43
1	40	1	1	1	1	313	46	0.60	110.43
1	50	1	1	1	1	315	46	0.60	110.43
1	60	1	1	1	1	316	46	0.60	110.43
1	70	1	1	1	1	319	46	0.60	110.43

LEG LENGTH: 7.5

1	50	1	1	1	1	1938	810	0.60	106.82

Table 6.16

Results for Test Grid 8 - A* w/straight-line

SEED: 120775

TOTAL THREAT DENSITY: 0.75

THREAT DATA: (radius / Pd / density)
         5 / 0.5 / 0.4   7.5 / 0.25 / 0.3   10 / 0.1 / 0.3

LEG LENGTH: 10   LEGS-PER-ARC: 5   PATH DEFLECTION ANGLE: 90

| COEFFICIENTS | | | | | | | | CONFLICT | PATH |
G	I	L	H	I	L	NODES	CPU	COST	LENGTH
1	30	1	1	1	1	255	37	0.60	110.43
1	40	1	1	1	1	325	49	0.60	110.43
1	50	1	1	1	1	383	60	0.60	110.43
1	60	1	1	1	1	385	61	0.60	110.43
1	70	1	1	1	1	442	69	0.60	110.43

LEG LENGTH: 7.5

1	50	1	1	1	1	1844	765	0.60	115.46

Table 6.15

Results for Test Grid 8 – A* w/box

DISCUSSION OF TEST GRID 8

This test is created with a threat density of 0.75. Execution times are much less than in previous tests. The output plots are roughly identical paths except for the 'jaggies' at the beginning of the length 7.5 path, Figure 6.18. This is obviously due to the necessity for correct positioning prior to maneuvering through the first pair of small threats. It is an interesting case since it is the first to demonstrate that a shorter leg length (one not a subset of the longer length) can be a detriment instead of an improvement.

SEED: 120775

TOTAL THREAT DENSITY: 0.75

THREAT DATA: (radius / Pd / density)
5 / 0.5 /0.4   7.5 / 0.25 / 0.3   10 / 0.1 / 0.3

LEG LENGTH: 10   LEGS-PER-ARC: 5   PATH DEFLECTION ANGLE: 90

G	I	L	H	I	L	NODES	CPU	CONFLICT COST	PATH LENGTH
1	30	1	1	1	1	269	39	0.60	110.43
1	40	1	1	1	1	318	47	0.60	110.43
1	50	1	1	1	1	317	46	0.60	110.43
1	60	1	1	1	1	318	46	0.60	110.43
1	70	1	1	1	1	321	46	0.60	110.43
1	30	1	0	0	0	2007	337	0.60	110.43
1	50	1	0	0	0	1809	352	0.60	110.43
1	70	1	0	0	0	1822	362	0.60	110.43

LEG LENGTH: 7.5

| G	I	L	H	I	L	NODES	CPU	CONFLICT COST	PATH LENGTH
1	50	1	1	1	1	1938	806	0.60	115.46

Table 6.14

Results for Test Grid 8 — Improved A*
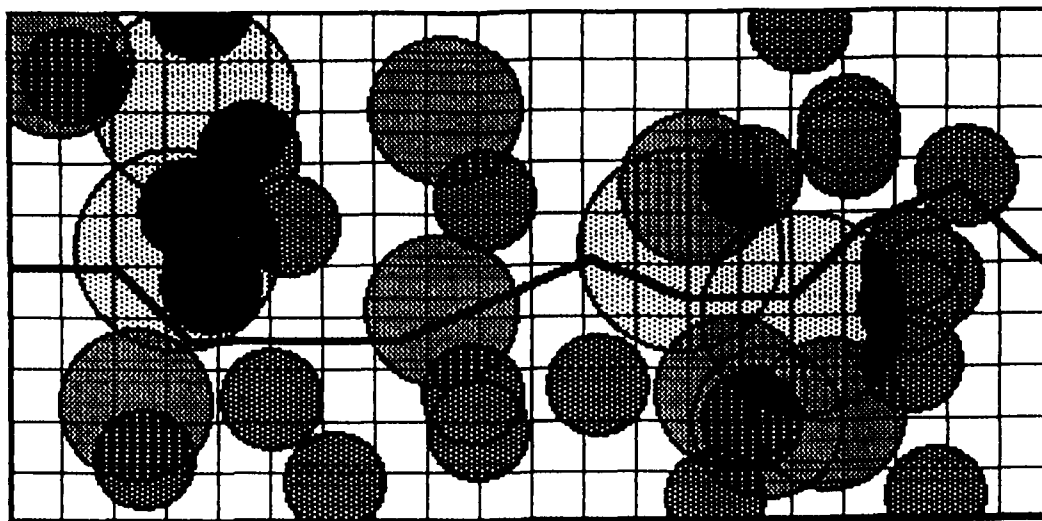
```
Path Length   =  111.92
Leg  Length   =   10.00
Conflict Cost =    1.05
```

Figure 6.15
Path Generated – Test Grid 7.1
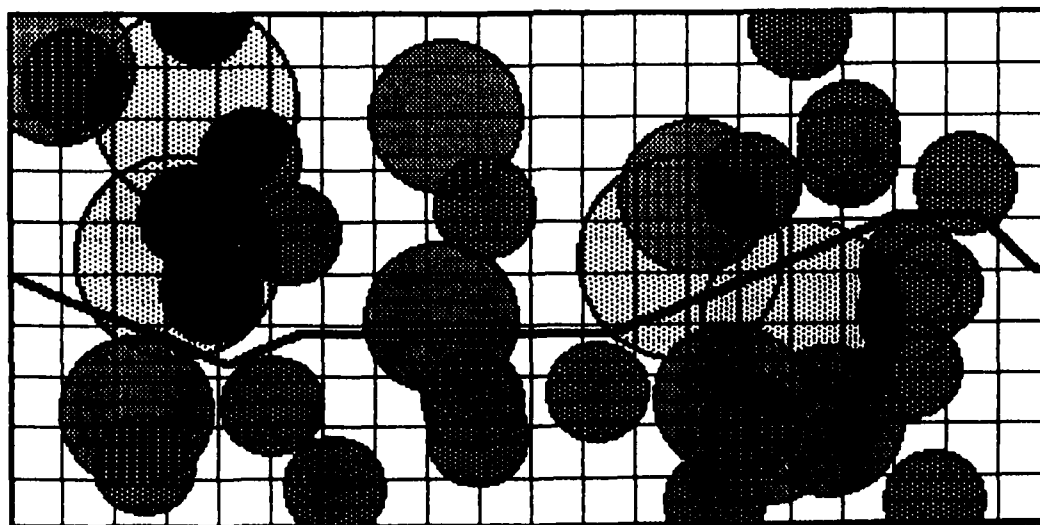

```
Path Length   =  106.82
Leg  Length   =    7.50
Conflict Cost =    1.05
```

Figure 6.16
Path Generated – Test Grid 7.2

SEED: 89204

TOTAL THREAT DENSITY: 1.0

THREAT DATA: (radius / Pd / density)
5 / 0.5 /0.4   7.5 / 0.25 / 0.3   10 / 0.1 / 0.3

LEG LENGTH: 10   LEGS-PER-ARC: 5   PATH DEFLECTION ANGLE: 90

		COEFFICIENTS						CONFLICT	PATH
G	I	L	H	I	L	NODES	CPU	COST	LENGTH
1	30	1	1	1	1	846	244	1.05	111.92
1	40	1	1	1	1	864	248	1.05	111.92
1	50	:	1	1	1	886	258	1.05	111.92
1	60	1	1	1	1	900	262	1.05	111.92
1	70	1	1	1	1	913	265	1.05	111.92

LEG LENGTH: 7.5

1	50	1	1	1	1	3287	2662	1.05	106.82

Table 6.13

Results for Test Grid 7 – A* w/straight-line

SEED: 89204

TOTAL THREAT DENSITY: 1.0

THREAT DATA: (radius / Pd / density)
5 / 0.5 / 0.4   7.5 / 0.25 / 0.3   10 / 0.1 / 0.3

LEG LENGTH: 10   LEGS-PER-ARC: 5   PATH DEFLECTION ANGLE: 90

| | COEFFICIENTS | | | | | | | CONFLICT | PATH |
G	I	L	H	I	L	NODES	CPU	COST	LENGTH
1	30	1	1	1	1	817	231	1.05	111.92
1	40	1	1	1	1	846	240	1.05	111.92
1	50	1	1	1	1	872	250	1.05	111.92
1	60	1	1	1	1	894	257	1.05	111.92
1	70	1	1	1	1	906	260	1.05	111.92

LEG LENGTH: 7.5

1	50	1	1	1	1	3256	2734	1.05	106.82

Table 6.12

Results for Test Grid 7 – A* w/box

SEED: 89204

TOTAL THREAT DENSITY: 1.0

THREAT DATA: (radius / Pd / density)
  5 / 0.5 / 0.4   7.5 / 0.25 / 0.3   10 / 0.1 / 0.3

LEG LENGTH: 10   LEGS-PER-ARC: 5   PATH DEFLECTION ANGLE: 90

| | COEFFICIENTS | | | | | | | CONFLICT | PATH |
G	I	L	H	T	L	NODES	CPU	COST	LENGTH
1	30	1	1	1	1	846	237	1.05	111.92
1	40	1	1	1	1	864	243	1.05	111.92
1	50	1	1	1	1	886	252	1.05	111.92
1	60	1	1	1	1	900	256	1.05	111.92
1	70	1	1	1	1	913	258	1.05	111.92
1	30	1	0	0	0	1631	346	1.05	111.92
1	50	1	0	0	0	1432	342	1.05	111.92
1	70	1	0	0	0	1391	349	1.05	111.92

LEG LENGTH: 7.5

| 1 | 30 | 1 | 1 | 1 | 1 | 2659 | 1933 | 1.05 | 106.82 |
| 1 | 50 | 1 | 1 | 1 | 1 | 3284 | 2656 | 1.05 | 106.82 |

Table 6.11

Results for Test Grid 7 – Improved A*

DISCUSSION OF TEST GRID 7

The results for this test are consistent with those of the previous tests. Also consistent with the previous test is the inability to execute a case with leg length of 5 within the four-hour CPU limit. This is true even though the threat density has decreased to 1.0! The output plots, Figures 6.15 and 6.16, are similar except for the more direct path the 7.5 route takes through the threats prior to the goal. An additional execution of a 7.5 case is included with the coefficients 1-30-1-1-1-1. A small improvement is evident between this and the 1-50-1-1-1-1 case with a leg length of 10. A considerably greater improvement is obtained when the leg length is 7.5, i.e. 2659 nodes with 1-30-1-1-1-1 vs. 3284 nodes with 1-50-1-1-1-1.
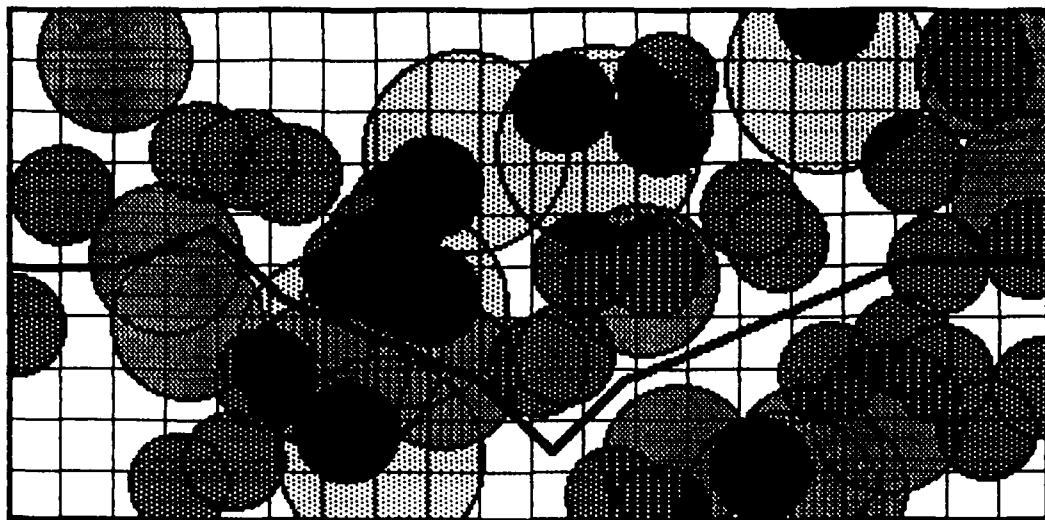
Path Length   = 113.40
Leg Length   =   10.00
Conflict Cost =   2.20

Figure 6.13
Path Generated – Test Grid 6.1
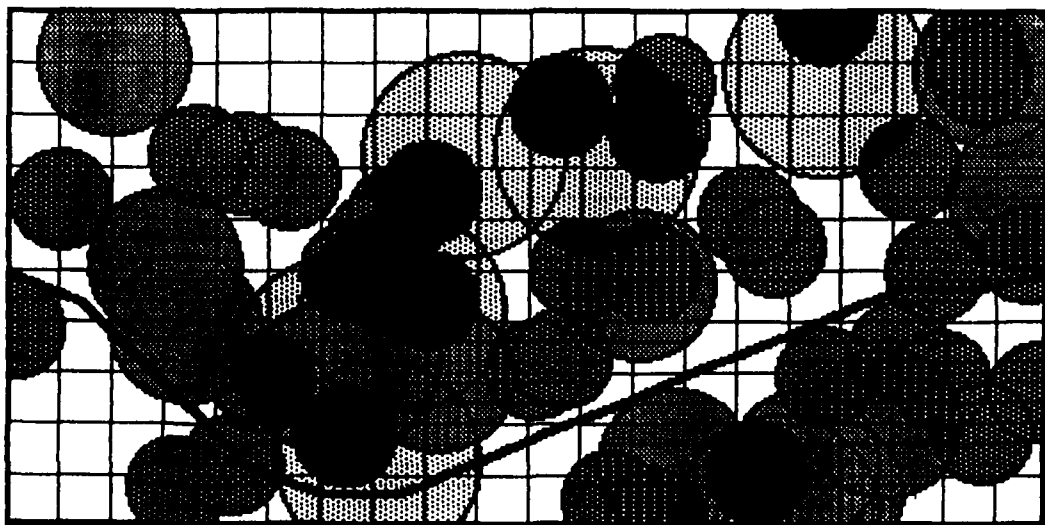
Path Length   = 111.91
Leg Length   =   7.50
Conflict Cost =   2.10

Figure 6.14
Path Generated – Test Grid 6.2
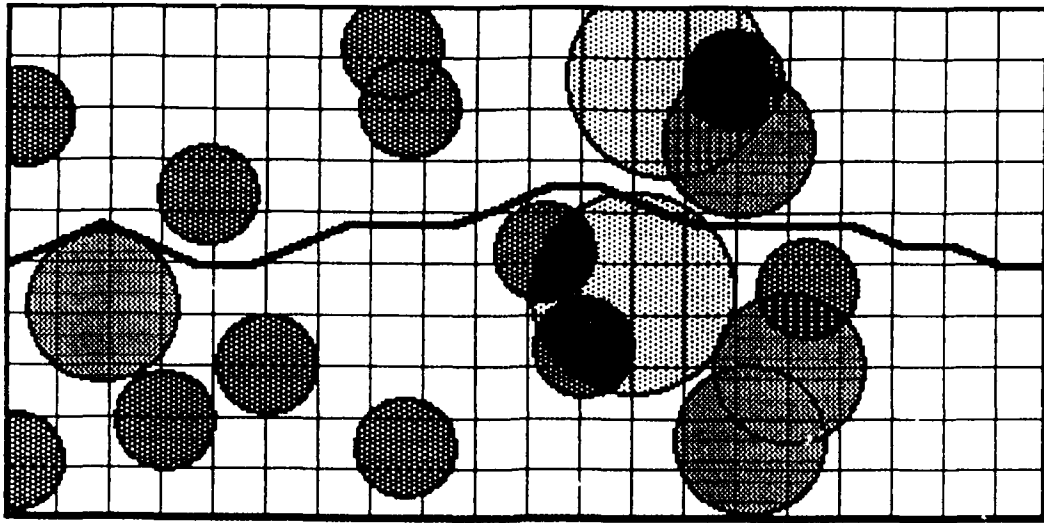
Path Length   = 104.57
Leg Length   =   5.00
Conflict Cost =   0.10

Figure 6.21
Path Generated - Test Grid 9.3

DISCUSSION OF TEST GRID 10

This grid, like the previous one, is also a density 0.50 test. It also develops roughly the same path regardless of leg length. The significant difference with this test is that it behaves exactly the opposite of all previous tests in this group (test grids 5 through 10) concerning the value of the g-threat coefficient. In the previous tests, increasing the value of this coefficient decreased the performance of the algorithm, i.e. it required greater nodes. In this test, the performance improved as this coefficient increased. Additional cases were run with g-threat coefficients of 100 and 500 with consistent results. This domain-dependent phenomenon hampers efforts to discover the 'perfect' set of coefficients.

There may be some question about the merit of the heuristic if the g-values are set very high, for example, using the g-coefficients 1-500-1. When the h-coefficients are 1-1-1 with this set of g-values, the solution requires 72 nodes. Without a heuristic, h-values set to zero, the same solution required 1130 nodes! A possible explanation may be that even though the heuristic value is relatively small compared to actual cost, it may be quite useful as a guidance tool by breaking ties.

The output plots, Figures 6.22 through 6.24, show that the cases with leg length less than 10 were able to maneuver a much more direct path around the last mid-sized threat. This results in a much shorter path length while maintaining the same conflict cost.

SEED: 39306

TOTAL THREAT DENSITY: 0.5

THREAT DATA: (radius / Pd / density)
   5 / 0.5 /0.4   7.5 / 0.25 / 0.3   10 / 0.1 / 0.3

LEG LENGTH: 10   LEGS-PER-ARC: 5   PATH DEFLECTION ANGLE: 90

		COEFFICIENTS						CONFLICT	PATH
G	T	L	H	T	L	NODES	CPU	COST	LENGTH
1	30	1	1	1	1	226	26	0.00	115.54
1	40	1	1	1	1	196	20	0.00	115.54
1	50	1	1	1	1	181	18	0.00	115.54
1	60	1	1	1	1	89	7	0.00	115.54
1	70	1	1	1	1	81	6	0.00	115.54
1	100	1	1	1	1	73	52	0.00	115.54
1	500	1	1	1	1	72	49	0.00	115.54
1	30	1	0	0	0	1827	301	0.00	115.54
1	50	1	0	0	0	1739	320	0.00	115.54
1	70	1	0	0	0	1723	331	0.00	115.54
1	500	1	0	0	0	1130	195	0.00	115.54

LEG LENGTH: 7.5

| 1 | 50 | 1 | 1 | 1 | 1 | 137 | 17 | 0.00 | 111.01 |

LEG LENGTH: 5

| 1 | 50 | 1 | 1 | 1 | 1 | 298 | 54 | 0.00 | 109.49 |

Table 6.20

Results for Test Grid 10 – Improved A*

SEED: 39306

TOTAL THREAT DENSITY: 0.5

THREAT DATA: (radius / Pd / density)
    5 / 0.5 / 0.4   7.5 / 0.25 / 0.3   10 / 0.1 / 0.3

LEG LENGTH: 10   LEGS-PER-ARC: 5   PATH DEFLECTION ANGLE: 90

	COEFFICIENTS							CONFLICT	PATH
G	I	L	H	I	L	NODES	CPU	COST	LENGTH
1	30	1	1	1	1	220	25	0.00	115.54
1	40	1	1	1	1	191	20	0.00	115.54
1	50	1	1	1	1	177	17	0.00	115.54
1	60	1	1	1	1	89	7	0.00	115.54
1	70	1	1	1	1	81	6	0.00	115.54

LEG LENGTH: 7.5

1	50	1	1	1	1	136	17	0.00	111.01

Table 6.21

Results for Test Grid 10 – A* w/box

SEED: 39306

TOTAL THREAT DENSITY: 0.50

THREAT DATA: (radius / Pd / density)
        5 / 0.5 /0.4   7.5 / 0.25 / 0.3   10 / 0.1 / 0.3

LEG LENGTH: 10   LEGS-PER-ARC: 5   PATH DEFLECTION ANGLE: 90

| COEFFICIENTS | | | | | | | | CONFLICT | PATH |
G	I	L	H	I	L	NODES	CPU	COST	LENGTH
1	30	1	1	1	1	226	26	0.00	115.54
1	40	1	1	1	1	196	20	0.00	115.54
1	50	1	1	1	1	181	18	0.00	115.54
1	60	1	1	1	1	89	7	0.00	115.54
1	70	1	1	1	1	81	6	0.00	115.54

LEG LENGTH: 7.5

1	50	1	1	1	1	137	17	0.00	111.01

Table 6.22

Results for Test Grid 10 - A* w/straight-line
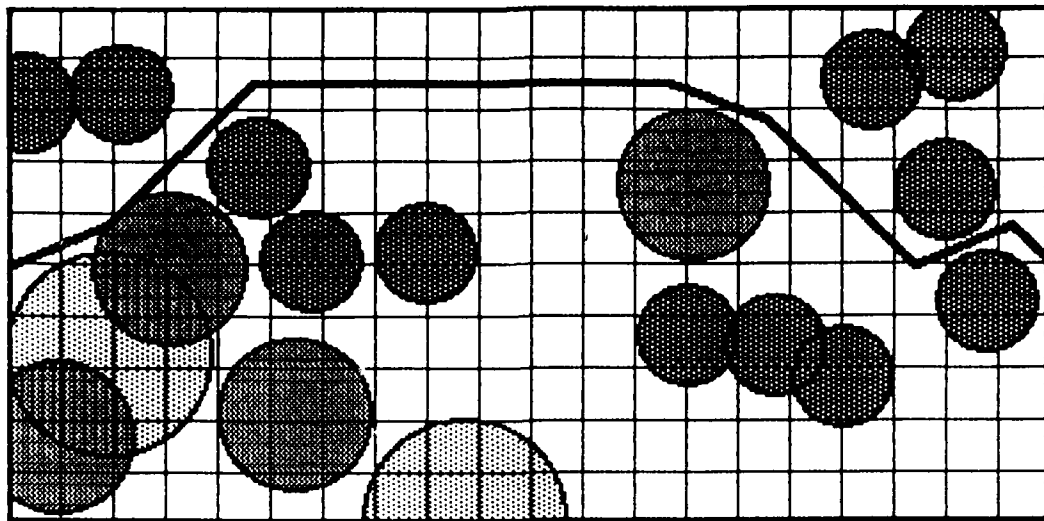
Path Length = 115.24
Leg Length = 10.00
Conflict Cost = 0.00

Figure 6.22
Path Generated – Test Grid 10.1
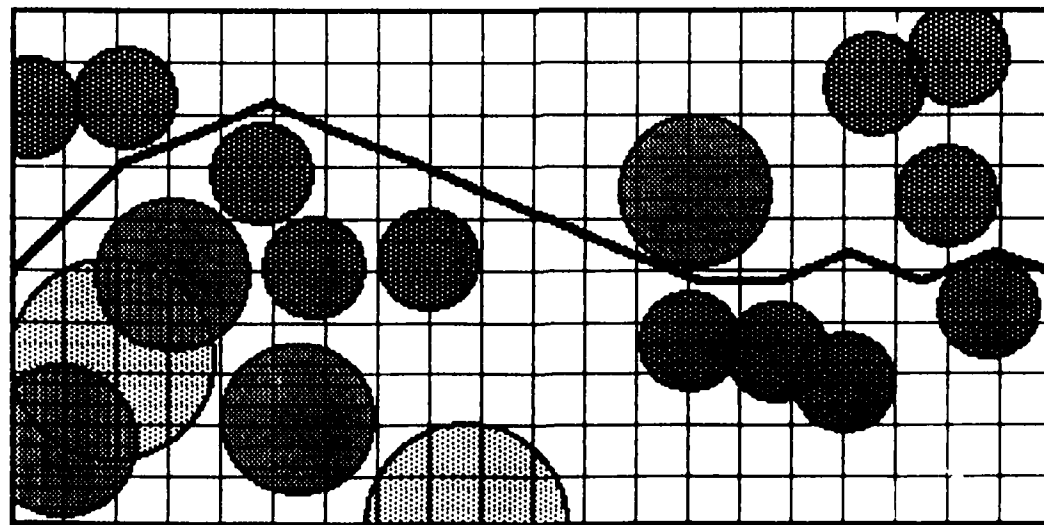
Path Length = 111.01
Leg Length = 7.50
Conflict Cost = 0.00

Figure 6.23
Path Generated – Test Grid 10.2
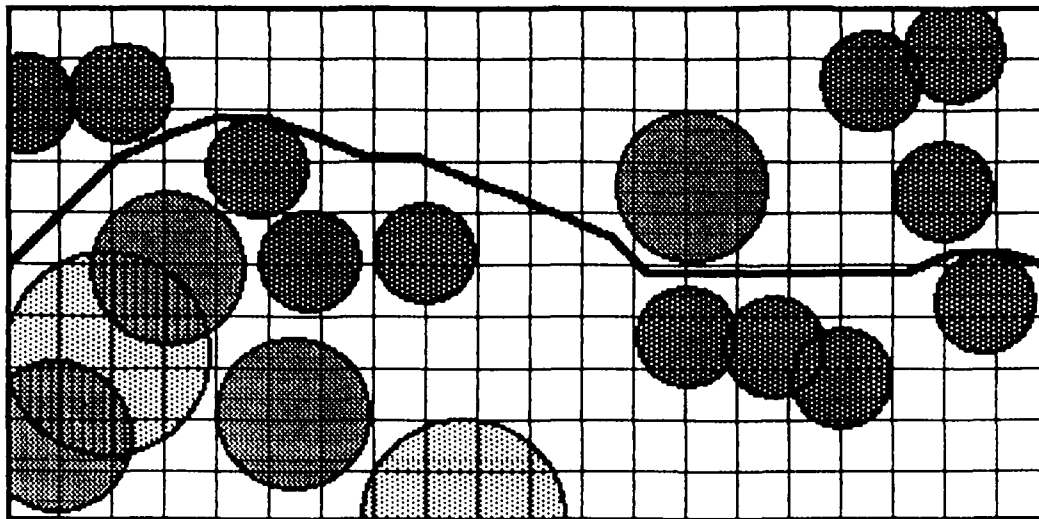
```
Path Length   =  109.49
Leg  Length   =    5.00
Conflict Cost =    0.00
```

Figure 6.24
Path Generated – Test Grid 10.3

VII. CONCLUSIONS

This thesis has clearly demonstrated the applicability of artificial intelligence techniques in the area of automated route planning. Certainly, these techniques have long been applied to route planning. For example, moving robots through a room or solving the travelling salesman problem of finding the shortest route which permits him to visit all cities, but only once.

One benefit of this thesis is the application of these techniques to a 'real-world' problem where current solutions are less than successful. In that sense, this thesis is quite successful. It is especially attractive in a strategic route planning arena where routes are created, using complete knowledge bases, prior to the execution of the route. In a tactical situation, where the immediate problem of avoiding the obstacle immediately ahead is paramount, then the FPG solution is most desireable, due to its speed of execution. An interesting alternative to this tactical problem would be to combine the two methods. Certainly threats or obstacles that weren't known *a priori* are going to be encountered, for example, a thunderstorm during an aircraft flight. The FPG model can quickly determine avoidance procedures based upon 'seeing' the new obstacle, while the more time-consuming Path-Finder model could re-route the remainder of the path given the new information and prior knowledge of future obstacles.

It is difficult to justify the expense of the SNOOPER model in virtually

every situation. Only if it is a necessary feature of the problem that a starting point, or even a few starting points, not be defined. An aspect of the A* search that has not been mentioned before is that it can be allowed to continue to find paths beyond the first success. The Path-Finder program could be easily modified to find all paths from a given starting point to a goal. The method would be to continue letting it expand goal nodes, saving the pointers so the path may be traced, until some desired condition is reached. And, it will generate these paths in order of least cost! Certainly, though, most of these paths will be partially coincident or intersecting. But, another algorithm might be employed to discard undesireable paths. The point is, the A* search provides the flexibility to attune the algorithm to specific requirements. This includes the inclusion of requirements such as altitude changes, complex threat modelling, and counter-threat capabilities.

This implementation is not without fault, though. During the initial tests, it was sincerely hoped that a 'magic' set of coefficients could be discovered. Unfortunately, that hope was not fulfilled. Yet, it is clear that there is a reasonably small set of coefficients that will give credible results in at least all the cases tested here. Given that the primary goal of minimum threat interaction is probably of paramount importance compared to path length, a very high value of the g-threat coefficient will 'guarantee' a solution of minimum conflict cost. This method, as the results indicate, may result in missing a shorter path with the same conflict cost. This problem of coefficients is a drawback to the current implementation, though not a serious one. Perhaps an idea for a different heuristic could eliminate the problem entirely.

APPENDIX A

```pascal
program path_finder (input, output);

const
    max_threat_cats = 5;
    max_threats = 100;
    max_legs = 15;
    pi = 3.141592654;
    deg_rad = 0.01745329252;

type

    list_type = (open_list, closed_list);

    coord_pair = record
        x_coord, y_coord : real;
    end;

    threat_rec = record
        category, n_centers : integer;
        radius, pd : real;
        center : array[1..max_threats] of coord_pair;
    end;

    node_ptr = ^node_rec;
    node_rec = record
        endpt : coord_pair;
        g_value, h_value, f_value : real;
        link, parent : node_ptr;
        child : array[1..max_legs] of node_ptr;
    end;

var

(* global variables *)

    x_limit, y_limit, legs_per_arc, nr_threats, nodes_exp, cpu_time,
        Martelli_count : integer;

    path_arc, leg_arc, leg_length, g_coeff, g_threat_coeff, g_length_coeff,
        h_coeff, h_threat_coeff, h_length_coeff : real;

    first_run : boolean;

    start, goal : coord_pair;

    threat : array[1..max_threat_cats] of threat_rec;
```

```
    open, closed, path_ptr : node_ptr;



function distance (point_a, point_b : coord_pair) : real;

(* compute the distance between the given points *)

begin
   distance := sqrt((point_a.x_coord - point_b.x_coord) ** 2 +
                    (point_a.y_coord - point_b.y_coord) ** 2);
end;




function flt_eq (a, b : real) : boolean;

(* evaluate approximate equality of two real numbers *)

begin
   if abs(a - b) < 0.00001 then
       flt_eq := true
   else
       fl_eq := false;
end;




procedure clear_list(list : node_ptr);

(* destroy given list *)

var
   next, old : node_ptr;

begin
   next := list;
   while next <> nil do
      begin
      old := next;
      next := next^.link;
      dispose(old);
      end;
end;




function threat_eval (point_a, point_b : coord_pair) : real;
```

```
(*  evaluate threat cost between given points  *)

label
   1000;

var
   i, categ : integer;
   cost, point_to_line, a, b, w, z, dist_ab, dist_ac, dist_bc : real;

begin
   cost := 0.0;
   dist_ab := distance(point_a, point_b);

   if dist_ab = 0.0 then    (* true when goal node generated *)
      goto 1000;

   (*  compute coefficients for distance from a point to a line  *)
   a := point_a.x_coord - point_b.x_coord;
   b := point_a.y_coord - point_b.y_coord;
   z := sqrt(a ** 2 + b ** 2);
   w := a * point_a.y_coord - b * point_a.x_coord;

   for categ := 1 to nr_threats do
      with threat[categ] do
         for i := 1 to n_centers do
            begin
            point_to_line := abs(b * center[i].x_coord - a *
                                 center[i].y_coord + w) / z;
            dist_ac := distance(point_a, center[i]);
            dist_bc := distance(point_b, center[i]);

            if point_to_line < radius then

               (* doe_ threat cover an endpoint *)
               if ((dist_bc < radius) and (dist_ac > radius)) or
                  ((point_a.x_coord = 0.0) and (dist_ac < radius)) then
                     cost := cost + pd

               (* is it within a box around the line segment *)
               else if (dist_bc <= sqrt(dist_ac ** 2 + dist_ab ** 2)) and
                       (dist_ac <= sqrt(dist_bc ** 2 + dist_ab ** 2)) and
                       (dist_ac > radius) then
                          cost := cost + pd;
            end;

1000:
   threat_eval := cost;

end;
```

```
procedure input_data;

(*   no input editing is performed in this version... *)

var
   ch : char;
   point : real;
   i : integer;
   execute_flag : boolean;

begin
   execute_flag := false;
   while (not eof) and (not execute_flag) do
      begin
      read(ch);

      if (ch = 'G') and (first_run) then
         readln(x_limit, y_limit)

      else if (ch = 'P') and (first_run) then
         begin
         read(point);
         start.x_coord := 0.0;
         start.y_coord := point;
         readln(point);
         goal.x_coord := x_limit;
         goal.y_coord := point;
         end

      else if (ch = 'T') and (first_run) then
         begin
         nr_threats := succ(nr_threats);
         with threat[nr_threats] do
            begin
            readln(category, radius, pd, n_centers);
            for i := 1 to n_centers do
               with center[i] do
                    readln(x_coord, y_coord);
            end

      else if ch = 'L' then
         begin
         readln(leg_length, legs_per_arc, path_arc);

         (*  convert input to radians  *)
         path_arc := path_arc * deg_rad;
         leg_arc := path_arc / (legs_per_arc - 1);
         end
```

```
            else if ch = 'C' then
                readln(g_coeff, g_threat_coeff, g_length_coeff,
                        h_coeff, h_threat_coeff, h_length_coeff)
                end

            else if ch = 'R' then
                begin
                readln;
                execute_flag := true;
                end;
end;



procedure develop_path;

(*  main driver for path finding algorithm  *)

label
    999;

var
    f_limit : real;
    child_ctr : integer;
    on_open, on_closed, failure : boolean;
    best, sucr, old : node_ptr;



function generate_legs (point : coord_pair) : node_ptr;

(*  return list of successor legs from given point  *)

 label
    1000;

 var
    i : integer;
    leg_angle : real;
    first, next, prev : node_ptr;

begin
    if distance(point, goal) <= leg_length then
        begin
        new(first);
        with first^ do
            begin
            endpt.x_coord := goal.x_coord;
            endpt.y_coord := goal.y_coord;
```
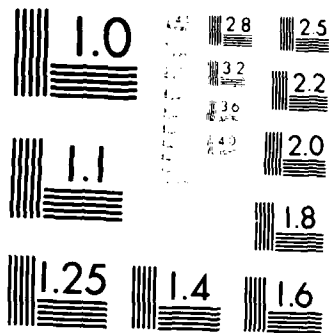
MICROCOPY RESOLUTION TEST CHART

```
            link := nil;
            goto 1000;
            end;

    prev := nil;
    first := nil;
    leg_angle := -(path_arc / 2.0) - leg_arc;

    for i := 1 to legs_per_arc do
        begin
        new(next);
        with next^.endpt do
            begin
            leg_angle := leg_angle + leg_arc;
            x_coord := point.x_coord + leg_length * cos(leg_angle);
            y_coord := point.y_coord + leg_length * sin(leg_angle);

            (* is the new point within the boundaries? *)
            if (y_coord < 0.0) or (y_coord > y_limit) or
              (abs(arctan((goal.y_coord - y_coord) /
              (goal.x_coord - x_coord))) > path_arc / 2.0) then
                    dispose(next)

            else
                begin
                if first = nil then
                    first := next
                else
                    prev^.link := next;
                next^.link := nil;
                prev := next;
                end;
            end;
        end;

1000 :
    generate_legs := first;
end;



function global_est (point : coord_pair) : real;

(* compute estimate of global cost between given point and goal *)

begin
    if h_threat_coeff <> 0 then
        global_est := h_coeff * (h_threat_coeff * threat_eval(point, goal) +
                                    h_length_coeff * distance(point, goal))
    else
```

```
            global_est := h_coeff * h_length_coeff * distance(point, goal);
end;



function local_cost (point_a, point_b : coord_pair) : real;

(*  compute actual cost along the leg from point a to b  *)

begin
    if g_threat_coeff <> 0 then
        local_cost := g_coeff * (g_threat_coeff * threat_eval(point_a, point_b)
                                + g_length_coeff * distance(point_a, point_b))
    else
        local_cost := g_coeff * g_length_coeff * distance(point_a, point_b);
end;



procedure insert_node_into_open (new : node_ptr);

var
    prev, next : node_ptr;
    inserted : boolean;

begin
    if open = nil then
        begin
        (*  put it on top  *)
        new^.link := open;
        open := new;
        end

    else if open^.f_value >= new^.f_value then
        begin
        (*  put it on top  *)
        new^.link := open;
        open := new;
        end

    else
        begin
        (*  search through list for correct spot *)
        inserted := false;
        next := open^.link;
        prev := open;
        while (not inserted) and (next <> nil) do
            if next^.f_value >= new^.f_value then
                begin
                new^.link := next;
```

```
                    prev^.link := new;
                    inserted := true;
                    end
                else
                    begin
                    prev := next;
                    next := prev^.link;
                    end;

            if not inserted then
                begin
                new^.link := nil;
                prev^.link := new;
                end;
            end;
    end;
end;




procedure reorder_open_list;

(*  reorder open list following propogation of better parent *)

var
    prev, curr, reorder_list : node_ptr;

begin
    prev := open;
    curr := open^.link;
    reorder_list := nil;

    (*  remove all out-of-place entries to reorder_list *)
    while curr <> nil do
        begin
        if curr^.f_value < prev^.f_value then
            begin
            prev^.link := curr^.link;
            curr^.link := reorder_list;
            reorder_list := curr;
            end
        else
            prev := curr;
            curr := prev^.link;
            end;

    while reorder_list <> nil do
        begin
        curr := reorder_list;
        reorder_list := curr^.link;
```

```
        insert_node_into_open( curr );
        end;
end;



procedure propogate_new_g ( old : node_ptr;  difference : real );

(* propogate improved g-value through successors of old *)

var
    i : integer;

begin
    i := 1;
    while old^.child[i] <> nil do
        begin
        if old^.child[i]^.parent = old then

            (* old is this child's parent *)
            with old^.child[i]^ do
                begin
                g_value := g_value - difference;
                f_value := g_value + h_value;
                propogate_new_g( old^.child[i], difference );
                end

        else if old^.g_value < old^.child[i]^.parent^.g_value then

            (* old is an improvement over current parent of this child *)
            with old^.child[i]^ do
                begin
                g_value := g_value - parent^.g_value + old^.g_value;
                f_value := g_value + h_value;

                (* propogate to successors with difference in values *)
                propogate_new_g( old^.child[i], parent^.g_value -
                                    old^.g_value );
                parent := old;
                end;

        i := succ( i );
        end;
end;



procedure select_best_g;

(* search open for node with lowest g_value node within f_limit bound *)
```

```
(*    -part of the Martelli improvement                          *)

var
   prev, next : node_ptr;
   best_g : real;
   within_limit : boolean;

begin
   Martelli_count := succ(Martelli_count);
   best := open;
   next := open;
   prev := nil;
   within_limit := true;
   best_g := open^.g_value;

   while (next^.link <> nil) and (within_limit) do
      if next^.link^.f_value < f_limit then
         begin
         if next^.link^.g_value < best_g then
            begin
            prev := next;
            best := next^.link;
            best_g := best^.g_value;
            end;
         next := next^.link;
         end
      else
         within_limit := false;

   If prev = nil then               (* top of open was best *)
      open := open^.link
   else
      prev^.link := best^.link; (* remove best from open list *)
end;

procedure update_old_node (new, old : node_ptr;  mode : list_type);

(* clean-up values/pointers for duplicate node, then trash it *)

var
   difference : real;

begin
   best^.child[child_ctr] := old;
   child_ctr := succ(child_ctr);
   if new^.g_value < old^.g_value then
      begin
      old^.parent := best;
      difference := old^.g_value - new^.g_value;
      old^.g_value := new^.g_value;
```

```pascal
        if mode = closed_list then
                begin
                propogate_new_g(old, difference);
                reorder_open_list;
                end;

            with old^ do
                f_value := g_value + h_value;
            end;
    end;



    procedure search_list (mode : list_type;
                           var sucr : node_ptr; var : found : boolean);

    (*  search named list for duplicate of sucr node *)

    var
       indx : node_ptr;

    begin
       if mode = open_list then
          indx := open
       else
          indx := closed;
       found := false;

       while (indx <> nil) and (not found) do
          if (flt_eq(sucr^.endpt.x_coord, indx^.endpt.x_coord)) and
             (flt_eq(sucr^.endpt.y_coord, indx^.endpt.y_coord)) then
                begin
                found := true;
                update_old_node(sucr, indx, mode);
                old := sucr;
                sucr := sucr^.link;
                dispose(old);
                end
          else
                indx := indx^.link;
    end;



    begin  (*  main section of develop path *)

    (*  create start node on open *)
       new(open);
       with open^ do
```

```
begin
endpt.x_coord := start.x_coord;
endpt.y_coord := start.y_coord;
g_value := 0.0;
h_value := 0.0;
f_value := 0.0;
link := nil;
end;

failure := true;
closed := nil;
f_limit := 0.0;
nodes_exp := 0;
cpu_time := clock; (* system dependent *)
Martelli_count := 0;

repeat
    if open = nil then (* failed *)
        goto 999;

    nodes_exp := succ(nodes_exp);
    if open^.f_value < f_limit then
        select_best_g (* Martelli improvement *)
    else                  (* best node is on top    *)
        begin
        best := open;
        open := best^.link;
        f_limit := best^.f_value;
        end;

    best^.link := closed;
    closed := best;
    if best^.endpt.x_coord = x_limit then (* at goal *)
        begin
        failure := false;
        goto 999;
        end;

    child_ctr := 1;
    sucr := generate_legs(best^.endpt);
    while sucr <> nil do
        begin
        sucr^.parent := best;
        sucr^.g_value := best^.g_value +
                         local_cost(best^.endpt, sucr^.endpt);

        (* search open list for duplicate node *)
        search_list(open_list, sucr, on_open);

        (* if not on open, check closed... *)
```

```
            search_list(closed_list, sucr, on_closed);
            if (not on_open) and (not on_closed) then
                begin
                best^.child[child_ctr] := sucr;
                child_ctr := succ(child_ctr);
                with sucr^ do
                    begin
                    h_value := global_est(endpt);
                    f_value := g_value + h_value;
                    child[1] := nil;
                    end;
                old := sucr^.link;
                insert_node_into_open(sucr);
                sucr := old;
                end;

            best^.child[child_ctr] := nil;
            end;
    until false; (* forever *)

999 :
    cpu_time := clock - cpu_time; (* system dependent *)
    if failure then
        path_ptr := nil
    else
        path_ptr := best;
end;



procedure output_results;

var
    i, j, k, indx : integer;
    path_length, conflict_value : real;
    prev, curr : node_ptr;

begin
    page;
    writeln('*** Path Finder Results ***');
    writeln;

    writeln('Grid limits: ', x_limit : 4, y_limit : 4);
    writeln;

    writeln('Start coordinates: ', start.x_coord : 6 : 2, start.y_coord : 6 : 2);
    writeln('Goal coordinates: ', goal.x_coord : 6 : 2, goal.y_coord : 6 : 2);
    writeln;

    writeln('Leg generation options: ');
```

```pascal
writeln(' ':5, 'Leg length:', leg_length:5:2, ' Legs per arc:',
        legs_per_arc:2, ' Path arc:', path_arc / deg_rad:5:2);
writeln;

writeln('G coefficients (g - threat - length):', g_coeff:6:2, '-',
        g_threat_coeff:6:2, '-', g_length_coeff:6:2);
writeln('H coefficients (h - threat - length):', h_coeff:6:2, '-',
        h_threat_coeff:6:2, '-', h_length_coeff:6:2);
writeln;

writeln('Threats:');
for i := 1 to nr_threats do
  with threat[i] do
    begin
    writeln(' ':5, 'Category:', category:3, ' Radius:', radius:5:2,
            ' Pd:', pd:5:2, ' Quantity:', n_centers:4);
    writeln;
    for j := 1 to (n_centers div 3) + 1 do
        begin
        for k := 1 to 3 do
            begin
            indx := (j - 1) * 3 + k;
            if indx <= n_centers then
                with center[indx] do
                    write(' ':3, x_coord:8:4, ',', y_coord:8:4);
            end;
        writeln;
        end;
    writeln;
    end;

writeln('Path from goal:');
prev := nil;
curr := path_ptr;
conflict_value := 0.0;
path_length := 0.0;
while curr <> nil do
    begin
    with curr^.endpt do
        writeln(' ':3, x_coord:8:4, ',', y_coord:8:4);
    if prev <> nil then
        begin
        path_length := path_length + distance(curr^.endpt, prev^.endpt);
        conflict_value := conflict_value + threat_eval(curr^.endpt,
                                                prev^.endpt);
        end;
    prev := curr;
    curr := curr^.parent;
    end;
writeln;
```

```
      writeln('Statistics:');
      writeln(' ':3, 'Nodes expanded :', nodes_exp);
      writeln(' ':3, 'CPU time(msecs): ', cpu_time);
      writeln(' ':3, 'Conflict cost  :', conflict_value : 10 : 2);
      writeln(' ':3, 'Path length    :', path_length : 10 : 2);
      writeln(' ':3, 'Martelli count :', Martelli_count);
      writeln;
      writeln;
   end;



begin (*  main section of path_finder  *)

   first_run := true;
   while not eof do
      begin
      input_data;
      develop_path;
      output_results;
      clear_list(open);
      clear_list(closed);
      first_run := false;
      end;

end. (*  end of path_finder  *)
```

APPENDIX B

```
(*  define a main-level TYPE: *)

   mode = (LOCAL, GLOBAL);
   . . .
   . . .

(*  all following code changes are underlined... *)

function threat_eval (point_a, point_b : coord_pair; eval_mode : mode) : real;

(*  evaluate threat cost between given points *)

label
   1000;

var
   i, categ : integer;
   cost, point_to_line, a, b, w, z, dist_ab, dist_ac, dist_bc : real;
   prox_dist : real;

begin
   cost := 0.0;
   dist_ab := distance(point_a, point_b);

   if dist_ab = 0.0 then    (* true when goal node generated *)
      goto 1000;

   (*  compute coefficients for distance from a point to a line *)
   a := point_a.x_coord - point_b.x_coord;
   b := point_a.y_coord - point_b.y_coord;
   z := sqrt(a ** 2 + b ** 2);
   w := a * point_a.y_coord - b * point_a.x_coord;

   for categ := 1 to nr_threats do
      with threat[categ] do
         begin
         if mode = LOCAL then
            prox_dist := radius
         else
            prox_dist := leg_length;


         for i := 1 to n_centers do
            begin
            point_to_line := abs(b * center[i].x_coord - a *
                              center[i].y_coord + w) / z;
```

```
              dist_ac := distance(point_a, center[i]);
              dist_bc := distance(point_b, center[i]);

              if point_to_line < prox_dist then

                  (*  does threat cover an endpoint  *)
                  if ((dist_bc < prox_dist) and (dist_ac > prox_dist)) or
                     ((point_a.x_coord = 0.0) and (dist_ac < prox_dist)) then
                          cost := cost + pd

                  (*  is it within a box around the line segment  *)
                  else if (dist_bc <= sqrt(dist_ac ** 2 + dist_ab ** 2)) and
                          (dist_ac <= sqrt(dist_bc ** 2 + dist_ab ** 2)) and
                          (dist_ac > prox_dist) then
                              cost := cost + pd;
              end;
          end;
1000:
   threat_eval := cost;

end;


...
...


function global_est (point : coord_pair) : real;

(*  compute estimate of global cost between given point and goal  *)

begin
   if h_threat_coeff <> 0 then
       global_est := h_coeff * (h_threat_coeff *
                               threat_eval(point, goal, GLOBAL) +
                               h_length_coeff * distance(point, goal))
   else
       global_est := h_coeff * h_length_coeff * distance(point, goal);
end;




function local_cost (point_a, point_b : coord_pair) : real;

(*  compute actual cost along the leg from point a to b  *)

begin
   if g_threat_coeff <> 0 then
       local_cost := g_coeff * (g_threat_coeff *
                               threat_eval(point_a, point_b, LOCAL) +
                               g_length_coeff * distance(point_a, point_b))
   else
```

```
        local_cost := g_coeff * g_length_coeff * distance(point_a, point_b);
end;

    . . .
    . . .

(* Modify the output_results subroutine: *)
    . . .
    . . .

        conflict_value := conflict_value +
                        threat_eval(curr^.endpt,prev^.endpt, LOCAL);

    . . .
    . . .

(* End of changes... *)
```

APPENDIX C

```pascal
program threat_bldr (input, output, datafil);

const
    pi = 3.1415927;

var
    i, j, nr_threats, count, legs_per_arc : integer;

    seed, total_density, threat_density,
        radius, pd, x_coord, y_coord,
        g_coeff, g_threat_coeff, g_length_coeff,
        h_coeff, h_threat_coeff, h_length_coeff,
        x_limit, y_limit, grid_area, start, goal,
        path_angle, leg_length : real;

    datafil : text;


function rand (var seed : real) : real;
begin
seed := ((25173 * seed) + 13849) mod 65536;
rand := seed / 65536;
end;


begin  (* main section of threat_bldr *)

rewrite(datafil);

writeln('Enter seed:');
readln(seed);

writeln('Enter x and y grid limits...');
readln(x_limit, y_limit);
grid_area := x_limit * y_limit;
writeln(datafil, 'G', x_limit, y_limit);

writeln('Enter number of threat categories...');
readln(nr_threats);
writeln('Enter total threat density...');
readln(total_density);
for i := 1 to nr_threats do
    begin
    writeln('For threat category:', i : 3, ' enter radius, pd, density...');
    readln(radius, pd, threat_density);
```

```
                count := trunc(total_density * threat_density * grid_area /
                          (pi * radius ** 2));
            writeln(datafil, 'T', i, radius, pd, count);
            for j := 1 to count do
                begin
                x_coord := x_limit * rand(seed);
                y_coord := y_limit * rand(seed);
                writeln(datafil, x_coord, y_coord);
                end;
            end;

    writeln('Enter start and goal coordinates...');
    readln(start, goal);
    writeln(datafil, 'P', start, goal);

    writeln('Enter leg length, legs per arc, and path deflection angle...');
    readln(leg_length, legs_per_arc, path_angle);
    writeln(datafil, 'L', leg_length, legs_per_arc, path_angle);

    writeln('Enter g coefficients (g - threat - length):');
    readln(g_coeff, g_threat_coeff, g_length_coeff);
    write(datafil, 'C', g_coeff, g_threat_coeff, g_length_coeff);
    writeln('Enter h coefficients (h - threat - length):');
    readln(h_coeff, h_threat_coeff, h_length_coeff);
    writeln(datafil, h_coeff, h_threat_coeff, h_length_coeff);

    writeln(datafil, 'R');

    end.
```

BIBLIOGRAPHY

[1] Rich, E., _Artificial Intelligence_, McGraw-Hill, 1983

[2] Barr, A. & E.A. Fiegenbaum, _Handbook of Artificial Intelligence_, Kaufman, 1981

[3] Nilsson, N., _Principles of Artificial Intelligence_, Tioga, 1980

[4] McLaughlin R.G., "Description and Use of SNOOPER III, A Model for Determining the Strategy Needed Over Optimum PEnetration Routes", Cornell Aeronautical Laboratory, 1971

[5] Martelli, A., "On the Complexity of Admissible Search Algorithms", _Artificial Intelligence_, Vol 8, 1977

[6] Grove, D., "Documentation for the FPG Model", University of Dayton, 1980

[7] Gelperin, D., "On the Optimality of A*", _Artificial Intelligence_, Vol 8, 1977

[8] Hart, P.E., N.J. Nilsson & B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", _IEEE Transactions on SSC_, Vol 4, 1968

[9] Hart, P.E., N.J. Nilsson & B. Raphael, "Correction to 'A Formal Basis for the Heuristic Determination of Minimum Cost Paths' ", _SIGART Newsletter_, Vol 37, 1972

[10] Fikes, R.E. & N.J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving", _Artificial Intelligence_, Vol 2, 1971

END

FILMED

5-85

DTIC