AN EXPERT DATABASE SYSTEM FOR THE OVERLAND SEARCH PROBLEM

by

Oliver Günther

Memorandum No. UCB/ERL M85/66

7 August 1985

AN EXPERT DATABASE SYSTEM FOR THE OVERLAND SEARCH PROBLEM

by

Oliver Günther

Memorandum No. UCB/ERL M85/66

7 August 1985

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

An Expert Database System for the Overland Search Problem

by

Oliver Günther

Department of Electrical Engineering and Computer Sciences

Computer Science Division

University of California at Berkeley

Berkeley, CA 94720

ABSTRACT

This paper describes a heuristic system to solve the overland search problem. The task in this problem is to find the best path between two points on a geographic map. In this paper the map is stored in an INGRES database. It is represented by a set of adjacent polygons. The cost of moving between two points within the same polygon is proportional to the distance between the two points. First, we present a basic system that performs a greedy path search to find the best path. This system is implemented in QUEL**, an extension of the INGRES query language QUEL. In order to speed up the path search we then suggest the following improvements. First, we propose to utilize knowledge about the area to perform a hierarchical decomposition of the given problem. Second, we propose to perform an initial coarsening of the given map. On a coarsened map, the system will be able to find an approximation of a solution path quickly. We propose several coarsening strategies, including two that utilize quadtrees.

This research was sponsored by the Naval Electronics Systems Command Contract N39-82-C-0235.

1. INTRODUCTION

Every car driver knows one of the most common instances of the overland search problem: to find the fastest route from his current location (say San Francisco Airport) to a given destination (like the restaurant "Chez Panisse" in Berkeley).

This overland search problem has recently received a lot of attention [Harmon84, Isik84, Kuan84, Kung84, Meystel84, Parodi84]. Besides the practical relevance of the problem, there are several reasons for the increasing interest. The overland search problem is a challenging application for expert systems [Hayes-Roth83] and heuristic search [Pearl84]. Furthermore, the problem raises interesting data management questions due to the enormous amount of data that is encoded in geographic maps. For example, one has to choose appropriate data and storage structures. If a database is used to hold the map data, one has to design an interface between the database and the path search system.

We believe that a database management system (DBMS) is a practical necessity. As proposed in [Kung84], we also move the search portion of the expert system into the DBMS. All proposed functions can be implemented in a classical query language like QUEL [Stonebraker76] that has been extended in order to allow spatial operations [Stonebraker83, Guttman84], and more sophisticated control structures for efficient rule processing [Stonebraker85a, Stonebraker85b]. This approach is in contrast to LISP-based approaches which face difficulties in performing an efficient data management [Butler85, Kung85].

The remainder of this paper is organized as follows. Section 2 gives a more concise definition of the overland search problem and introduces the abstraction of the problem that is employed in this paper. In this abstraction, the map is represented by a set of adjacent polygons. The cost of moving between two points within the same polygon is proportional to the distance between the two points. Section 3 outlines a simple approach how to transform the actual input data into this polygon representation. Section 4 describes a heuristic system of rules for the greedy path search and discusses its implementation. The

following two sections discuss several improvements to this system. Section 5 describes a knowledge-based approach for an initial hierarchical decomposition of the overland search problem. Section 6 presents the concept of map coarsening; on a coarsened map the system will be able to find a rough approximation of a solution path quickly. Several coarsening strategies are presented, including two that utilize quadtrees. Section 7 is a summary of our conclusions.

2. THE OVERLAND SEARCH PROBLEM

In the overland search problem one tries to find the best path between two given points in a given environment. The solution to such an overland search problem does not only depend on the starting point and destination point. It also depends on the kind of vehicle used, on the road conditions, on the time of the day, on the cost function to be used, and so on. For example, some vehicles are restricted to travel on roads, whereas other vehicles can choose an overland route. The time to cross Golden Gate Bridge during rush hour is much greater than during off-peak hours. To minimize fuel consumption one will often choose a different route than to minimize travelling time.

Therefore, an instance of the overland search problem is formally given by a starting point and a destination point in the plane, and a scalar cost function. The cost function is completely defined over all points in the plane and indicates the marginal cost of moving the vehicle under the given conditions from a given point into a given direction. The cost function incorporates all data about the vehicle, the road conditions, the time of the day, the minimization criterion, and so on.

For practical purposes, the cost function can be thought of as a grey-level map where the grey level of an area reflects the marginal cost to move the given vehicle in this area. It is a separate problem how to obtain this map quickly. A simple approach is outlined in section 3.

This paper deals with the following abstraction of the overland search problem. Given is a partition of the plane into polygons where the polygons may be concave and may have holes. The cost of moving within the polygon P one unit of distance is given by a cost coefficient C_P. The direction of the vehicle's movement does not matter as long as it stays inside the same polygon. Clearly, this polygon partition of the plane is a discretized version of the grey-level map described above.

This approach is in contrast to the approaches of [Kung84] and [Parodi84] where the map is assumed to be discretized and processed into a grid-like graph. The graph has edges and associated costs for each pair of points that are neighbors in the grid. For a discussion of these and other ways to represent a map see [Meystel84].

3. MAP MAKING

The map making process is not directly part of the overland search problem: it is a pre-processing step. This section will outline a simple approach how to transform the actual input data first into a grey-level map, and then into the polygon representation described above. This polygon representation serves as input for the algorithms in sections 4 and 6.

Suppose one is given several special-purpose maps of an area, such as a topographic map, soil and vegetation maps for the current season, a traffic map for the current time of the day, and so on. All these maps represent features that are significant for a route planner. The union of these features represents the current state of the area where the route planning process will take place. The first task is to produce a cost function that takes all relevant special-purpose maps into account and represents the composition of those maps. In other words, one would like to have a grey-level map that is a sufficiently good approximation of the composition of the relevant special-purpose maps.

It seems that this approach is in fact a good approximation to the way a human reads a map. Given several special-purpose maps of an area, most human map readers will first

conceptually compose those maps into a grey-level map. Then they perform the actual path search on this conceptual composed map.

The grey-level map can be obtained as follows. We only give a short outline of the steps of the algorithm. The details are beyond the scope of this paper and are not discussed further.

(1) *Picture Sampling and Quantization:* All available special-purpose maps are transformed into grey-level maps. Then the grey levels are transformed into a number representation. The numbers indicate how expensive it is to move the given vehicle in the corresponding areas. The discretized maps are stored as arrays of these numbers. This step has to be done only once for each special-purpose map.

(2) *Map Selection and Overlay:* The maps that are significant for the given situation are selected and overlaid. The number representation of the resulting composed map is determined by the (maybe weighted) average of the corresponding numbers of the single layers.

(3) *Picture Segmentation*: Finally, the composed map has to be transformed into a polygon representation as described above. This can be done by a split-and-merge algorithm as described by Horowitz and Pavlidis [Horowitz76].

(3.1) *Split*: First the map is recursively split into smaller regions until each component map is *homogeneous*. Thus, a criterion must be chosen for deciding that a region is homogeneous. One such criterion requires the definition of several disjoint cost ranges whose union is the set of real numbers. Then a region is homogeneous if the numbers that represent the grey levels in the region are all in the same cost range. Another criterion is that the standard deviation of the grey levels of the region (taken over all pixels) is below a given threshold.

(3.2) *Merge*: At the end of the splitting step all regions are homogeneous. However, the regions are not necessarily maximal homogeneous regions. Therefore, a merge step is performed. Each region checks its neighbor regions one by one, if the union region would still be homogeneous. If yes, the two regions merge.

(3.3) *Polygon Retrieval*: The polygon representation of the map is determined by the maximal homogeneous regions yielded by the merging step. The cost coefficient of a polygon is the average of the numbers that represent the grey levels of the pixels in the polygon.

Due to the merging step (3.2), the polygons that result from this map making process may be concave and may have holes. This is not very good for path finding applications because most path finding techniques will do better if the polygons are "simple" what may mean convex, hole-free, or few vertices.

On the other hand it is desirable to keep the number of polygons in the map representation low, because this number will have a direct impact on the running time of the algorithm. There is a trade-off to be made between the number of polygons and the complexity of their shapes. It seems like the only way to do this is to find out by practical implementation experience.

If the granularity of a map seems too fine (i.e. the representation has too many polygons), a map coarsening may be performed to decrease the number of polygons. This is done by redefining the criterion for homogeneity and applying this criterion to the given map. In this paper we discuss two ways of doing that. One way is to define cost ranges. The coarsening algorithm will then merge all adjacent polygons with a cost coefficient in the same range. Another way is to specify a (possibly large) threshold for the standard deviation of grey levels in a homogeneous region. Then the coarsening algorithm merges adjacent polygons if the standard deviation of the grey levels of their union region is below the threshold. This kind of coarsening does usually not yield a unique result map. Both approaches are discussed in section 6.

On the other hand, it might be the case that the map representation contains many polygons with an "irregular" shape (i.e. they are convex, have holes, and so on). In this case, one might be willing to accept a higher number of polygons if the shape of the polygons was simpler. For this purpose one may perform a polygon simplification step that

decomposes the polygons into simpler components. It turns out that a trade-off has to be made between the running time of such an algorithm and the number of yielded components. Algorithms that yield a minimum number of components are usually hyperquadratic or even exponential. For example, the problem of decomposing a polygon with holes into a minimum number of convex polygons has recently been proven NP-hard [Keil83]; if the original polygon does not have holes, the same problem can be solved in polynomial, however hyperquadratic, time [Chazelle79, Keil83]. It is an area of future research to develop appropriate heuristics that do a faster decomposition without increasing the number of yielded components too much above the optimum. As of yet, there are only few heuristics known that address these problems.

4. A RULE SYSTEM

This section presents a system of *path generation rules*. This system is geared towards simulating the path finding strategy of a human map reader. As our experiments have shown, most human map readers perform a best-first strategy. They are looking for the cheapest nearby areas that lead them closer to their destination and continue from there. This strategy can be simulated by the following heuristical approach.

In the given abstraction of the overland search problem the map is represented by a set of adjacent polygons. The cost of moving between two points within the same polygon is proportional to their distance. The solution path will be represented by a list of path nodes. The greedy path search retrieves this path, node by node. In order to to find the *next path node* from the *current path node* the following algorithm FINDNEXT is performed.

4.1. Path Generation Rules

Let π_{cur} denote the current path node, and Ψ_{cur} the current polygon. The *current polygon* has been determined when π_{cur} was determined. It is the polygon that contains the current path node π_{cur} and that will contain the path segment from π_{cur} to the next path node π_{nex}. Starting from π_{cur}, FINDNEXT looks towards the destination Ω and scans the boundary segments of Ψ_{cur} that lie within the optic angle. The *optic angle* is the angle that is rooted at π_{cur} and halved by the axis (π_{cur}, Ω). Its size is to be defined by the user; typical sizes are between 30 and 120 degrees. For an example see figure 1.

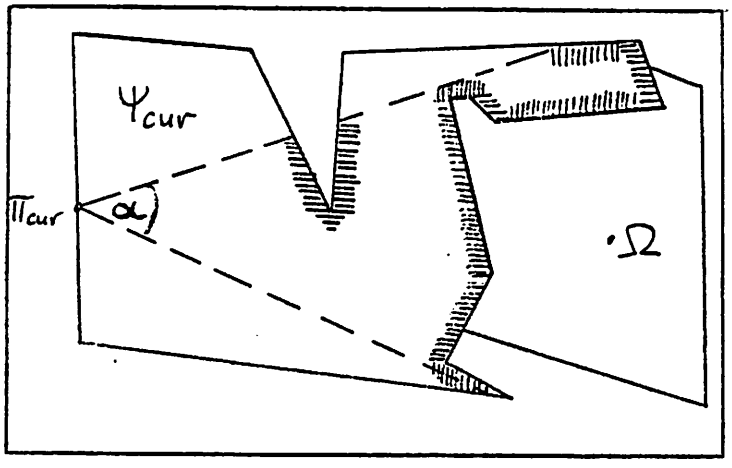


Figure 1: α is the optic angle, and the boundary segments of Ψ_{cur} that lie within the optic angle are shaded.

FINDNEXT selects the cheapest polygon Ψ_{nex} among all feasible polygons (except Ψ_{cur}) that are adjacent to any of those boundary segments. A polygon is *feasible* if its cost coefficient is below a given threshold which depends on the vehicle. If Ψ_{nex} is no more expensive than Ψ_{cur} then the next path node π_{nex} will be the closest point within the optic angle that lies on the boundaries of Ψ_{nex} and Ψ_{cur}. If Ψ_{nex} is more expensive than Ψ_{cur} but still feasible, then FINDNEXT prefers Ψ_{cur} over Ψ_{nex} and stays in Ψ_{cur} as long as it leads closer to Ω. Then the next path node π_{nex} will be the point that is within the optic angle, lies on the boundaries of Ψ_{nex} and Ψ_{cur}, and is closest to the destination Ω. In any case, Ψ_{nex} is the new current polygon.

If all the polygons within the optic angle are infeasible then there is some obstacle in the way. In this case, FINDNEXT asks the user to provide a larger optic angle. If the user provides one, FINDNEXT repeats the above procedure with the larger optic angle. Otherwise, FINDNEXT repeats the procedure with an optic angle of 360°, i.e. all adjacent polygons are taken into consideration. If this does still not yield a new path node, FINDNEXT gives up. Note that a larger optic angle makes FINDNEXT less greedy: it does not only take a rather direct route into consideration but also considers routes that might lead to the destination less directly.

In order to avoid endless loops, FINDNEXT checks each pair (π_{nex}, Ψ_{nex}) whether it has occurred previously. If yes, FINDNEXT is about to enter an endless loop. In this case, the new path node is rejected and FINDNEXT continues from the current path node with the next cheapest adjacent polygon within the optic angle.

If a new path node π_{nex} has been found and validated then FINDNEXT checks if the straight line between π_{nex} and the old path node π_{cur} is completely within the current polygon Ψ_{cur}. If Ψ_{cur} is not convex there might be some obstacles (in most cases, more expensive polygons) in the way. In this case, FINDNEXT constructs detour paths around those other polgons along the boundary of Ψ_{cur}. These detour paths induce additional path nodes and lie completely within Ψ_{cur}.

Finally, for each new path node π_{nex}, FINDNEXT checks if π_{nex} is in the same polygon as the destination Ω. If yes, FINDNEXT is done: FINDNEXT draws the straight line between π_{nex} and Ω, and refines it according to the detour rule presented above.

FINDNEXT can be described more concisely by the rules on the following page. As above, π_{cur} is the current path node, π_{nex} the next path node, Ψ_{cur} the current polygon, Ω the destination. Furthermore, $(\alpha_0, \ldots, \alpha_k)$ is the user defined increasing sequence of optic angles, and G is the upper cost bound for a feasible polygon. i is initialized as 0.

(1) IF π_{cur} and Ω are in the same polygon

THEN π_{nex} is Ω, execute rule (7), and STOP

ELSE

(2) IF the cheapest adjacent polygon within angle α_i (looking towards Ω) is no more expensive than Ψ_{cur}

THEN Ψ_{nex} is this polygon, π_{nex} is the closest point of Ψ_{nex} that lies within angle α_i; continue with rule (6)

ELSE

(3) IF the cheapest adjacent polygon within angle α_i (looking towards Ω) is more expensive than Ψ_{cur} but has cost $\leqslant G$

THEN Ψ_{nex} is this polygon, π_{nex} is the point on the common boundary of Ψ_{cur} and Ψ_{nex} that lies within angle α_i and is closest to Ω; continue with rule (6)

ELSE IF i < k THEN { i := i + 1; continue with rule (2) }
ELSE continue with rule (4).

(4) IF the cheapest adjacent polygon is no more expensive than Ψ_{cur}

THEN Ψ_{nex} is this polygon, π_{nex} is the closest point of Ψ_{nex}

ELSE

(5) IF the cheapest adjacent polygon is more expensive than Ψ_{cur} but has cost $\leqslant G$

THEN Ψ_{nex} is this polygon, π_{nex} is the point on the common boundary of Ψ_{cur} and Ψ_{nex} that is closest to Ω

ELSE FAILURE. STOP.

(6) IF the pair (π_{nex}, Ψ_{nex}) has occurred previously

THEN discard this new path node π_{nex}; continue with the rule and the optic angle that yielded this path node and take the next cheapest polygon for Ψ_{nex} instead.

(7) IF the straight line (π_{cur}, π_{nex}) intersects polygons other than Ψ_{cur}

THEN construct detour paths along the boundary of Ψ_{cur}.

Figures 2 and 3 give examples for the application of these rules. Note, that except the start and the destination point, all path nodes will lie on polygon boundaries.

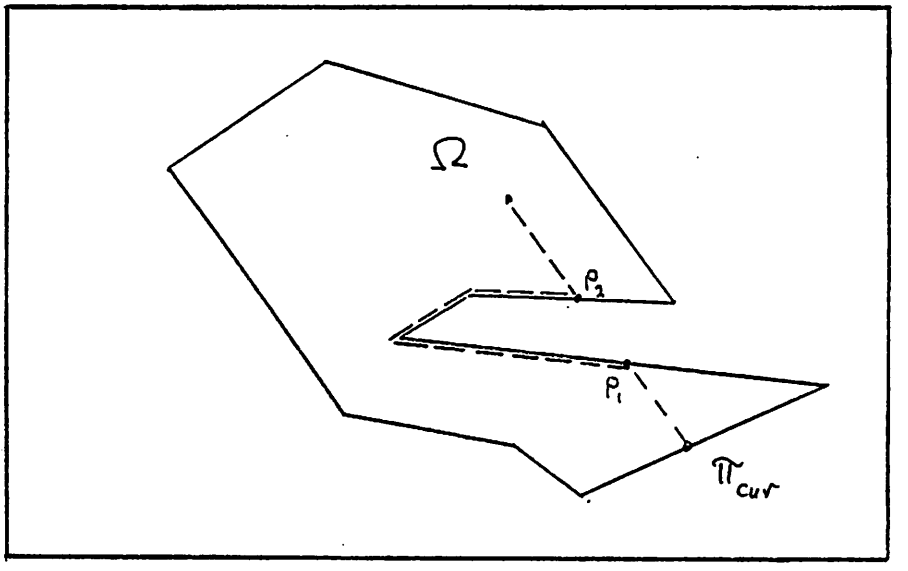


Figure 2: Rule 1 yields the straight line between π_{cur} and Ω ; rule 7 yields the detour path from point P_1 to point P_2.

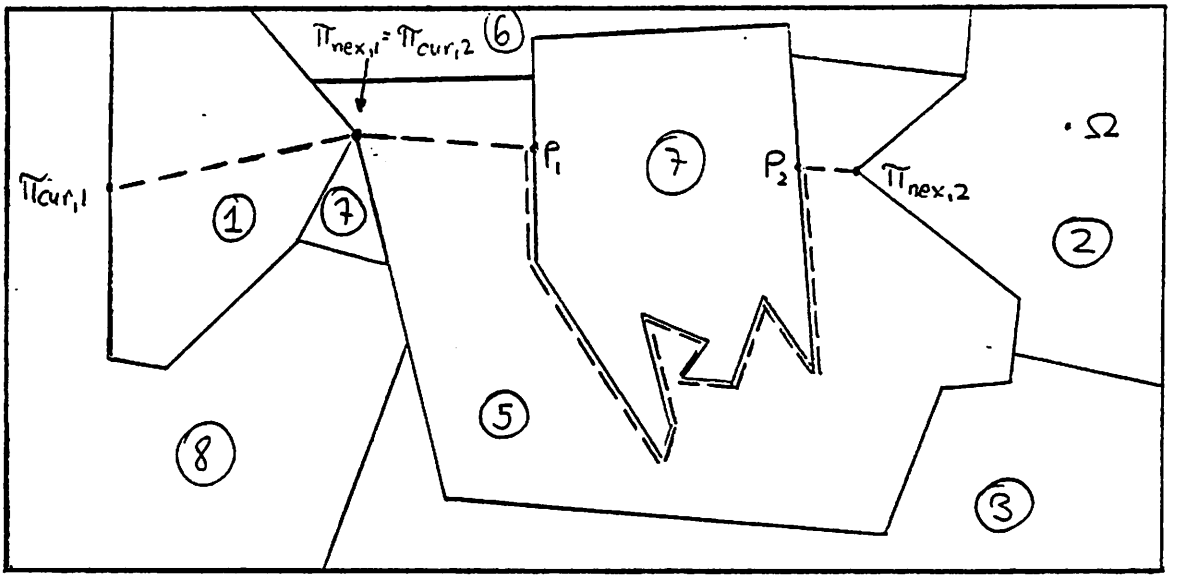


Figure 3: The circled numbers denote the cost coefficients of the polygons, the optic angle is 90°. Initially, the polygon with cost coefficient 1 is the current polygon, and $\pi_{cur,1}$ is the current path node. First, rule 2 starts at point

$\pi_{cur,1}$ and searches for polygons within the optic angle that have a cost coefficient of less than 1. As this does not succeed, rule 3 is activated. Rule 3 finds the polygon with cost coefficient 5, and yields the straight line from $\pi_{cur,1}$ to $\pi_{nex,1}$. The polygon with cost coefficient 5 is now the current polygon. Then rule 2 starts from point $\pi_{nex,1} = \pi_{cur,2}$ and finds the polygon with cost coefficient 2. It yields the straight line from $\pi_{cur,2}$ to $\pi_{nex,2}$. Finally, rule 7 yields the detour path from point P_1 to point P_2. This example shows that rule 7, that takes care of the detour paths, is in itself very clumsy. It might yield very strange paths if there are many non-convex polygons that have irregular boundary shapes.

4.2. Implementation of Path Generation Rules

The map is stored in an INGRES [Stonebraker76] database. The rules are coded in QUEL**, an extension of the query language QUEL [Stonebraker76] that allows abstract data types [Stonebraker83, Guttman84], and more sophisticated control structures for efficient rule processing [Stonebraker85a, Stonebraker85b]. This system makes especially use of data types for spatial objects (polygons, lines, etc.) and for program code. The current implementation is written in EQUEL/FORTRAN [RTI84]. The above extensions are simulated by EQUEL/FORTRAN programs.

The control structures used are the EXECUTE* command and the EXECUTE-UNTIL command.

EXECUTE* (*QUEL-COM*) WHERE (*WA*)

executes the command sequence *QUEL-COM* over and over again until it fails to modify the database or until the qualification *WA* is false. Note that *QUEL-COM* is an object of the abstract data type *QUEL-Code*.

EXECUTE-UNTIL (*QUEL-COM*)

tries to execute the commands in the command sequence *QUEL-COM* one after another until one command does actually have some effect on the database. This command will be the only command that will be executed.

The EXECUTE-UNTIL command is applied to the set of rules: EXECUTE-UNTIL (*RULES*). Each command in *RULES* represents one of the rules (1), (2), (3), (4), and (5). Hence, only one rule applies to a current point. This means, however, that only one path is alive at a given time which is necessary to maintain computational efficiency. Considering computational efficiency it does not make sense to keep alive an exponential number of paths.

Note that explicit QUEL-statements are needed only for the rules (1), (2), (3), (4), and (5). Rule (6) is implied by the 'stay-in' operator (see below), rule (7) is covered in the WHERE clauses.

The solution uses the concept of abstract data types to implement the data types 'polygon', 'line', 'path', and 'point'. The following operations are defined on those types.

><	...	stay-in: line-1 >< pol-1 returns a path of line segments with endpoints in common with line-1 which stays inside pol-1
@	...	adjacent; this operator is simulated by a relation ADJACENT (pol-1,pol-2)
//	...	closest
()	...	in
&	...	concatenation
!!	...	intersection
#(X,α)	...	area covered by angle α that is rooted at X and halved by (X,Ω)
line(X,Y)	...	line from point X to point Y
length(P)	...	length of path P

The solution uses the following relations:

MAP (polygon,cost)	...	polygons of the map
PATHS (path,cost,cur-poly,cur-pt,next-poly,next-pt)	...	paths
OLD (node,next-poly)	...	pairs (π_{nex} ,Ψ_{nex}) that have occurred previously

It follows the code of the MAIN LOOP and of RULE 2. The complete code is listed in appendix A.

```
range of m,m1,m2,m3 is MAP
range of p is PATHS
range of o is OLD

MAIN LOOP:

execute* (RULES; append to OLD (node = p.next-pt,next-poly = p.next-poly))
where p.cur-pt != finish

RULE 2:

replace p (path = p.path & (line(p.cur-pt,p.next-pt) >< p.cur-poly),
        cost = p.cost + m1.cost * length(line(p.cur-pt,p.next-pt) >< p.cur-poly),
        cur-poly = p.next-poly,
        next-poly = m3.polygon,
        cur-pt = p.next-pt,
        next-pt = p.next-pt // (#(p.next-pt,α_i) !! m3.polygon)

where m1.polygon = p.cur-poly
and   m2.polygon = p.next-poly
and   m3.cost = min (m.cost
                where m.polygon @ (p.next-poly !! #(p.next-pt,α_i))
                and   m.cost ≤ m2.cost)
and   any (o.node where o.node = p.next-pt // (#(p.next-pt,α_i) !! m3.polygon
                  and o.next-poly = m3.polygon)) = 0
```

So far we ran one experiment to test the performance of the path generation rules. The map is a topographic map of a 36 square mile area of Santa Cruz county, California. This area is extremely mountainous with deep valleys and an extensive river system. The map is represented by a 420 adjacent polygons; the adjacency relation has 2039 tuples. The starting point Λ and the destination point Ω are at opposite corners of the map. The sequence of optic angles was (90°,180°). The result path has 1928 nodes. The path search took 1017 CPU seconds of INGRES time, and 352 CPU seconds of FORTRAN time.

Hence, in this example the path search takes about 23 minutes of CPU time. For practical applications, this is not acceptable. The main reasons for this high running time are the necessity of complicated queries in order to retrieve certain boundary segments, and the expensive check which boundary segments lie in the optic angle.

In order to speed up the search we suggest the following improvements. First, we propose to utilize knowledge about the area to perform a hierarchical decomposition of the given problem. Second, we propose to perform an initial coarsening of the given map. On a coarsened map, a system will be able to find an approximation of a solution path quickly. From there, the path can be refined further. These suggestions are discussed in more detail in the following two sections.

5. HIERARCHICAL PROBLEM DECOMPOSITION

5.1. Path Decomposition Rules

In order to speed up the path search, we propose the hierarchical decomposition of a given overland search problem by rules which utilize knowledge about the area. A common example is the following:

> If one wants to go by car from any point in Berkeley to any point in San Francisco as fast as possible then one has to use the Bay Bridge.

These *path decomposition rules* will usually be used *before* one resorts to the path generation rules presented in section 4.

The system first uses the available local knowledge to construct path decomposition rules. The application of these rules yields a hierarchical decomposition of the given overland search problem. These rules are applied until none of the rules can be applied to any of the yielded subproblems, i.e. one does not have any local knowledge about these subproblems.

For example, suppose a car driver wants to drive as fast as possible from his apartment on Blake Street in Berkeley to "Mc Donald's" in San Francisco. The only local knowledge he has can be stated in the following two rules:

> (1) If one wants to go by car from any point in Berkeley to any point in San Francisco as fast as possible then one has to use the Bay Bridge.

(2) If one wants to go by car from Blake Street to the Bay Bridge as fast as possible then one has to take Ashby Street.

The application of these two rules yields three subproblems: how to get from Blake to Ashby, how to get from Ashby to the Bay Bridge and how to get from the Bay Bridge to "Mc Donald's".

One does not have any local knowledge that is applicable to any of these subproblems. Instead one has to resort to a map that approximates the cost function and to a more general rule system like the one described in section 4.

5.2. Implementation of Path Decomposition Rules

In an actual implementation, the number of path decomposition rules will typically be at least several hundreds. One therefore encounters two problems when implementing path decomposition rules. The first problem is to find applicable rules for a given situation sufficiently fast. The second problem arises if there are several rules that all apply to the current situation: one rule has to be selected.

These problems are addressed in [Stonebraker85b]. This paper proposes a mechanism that extends a relational database system like INGRES by some inferencing capabilities. When a retrieve command cannot be satisfied using only stored data, the data manager determines if a rule in the knowledge base can be used to reformulate the query. In this way, one works from the desired data toward database facts which must be ascertained using backward chaining. The paper also proposes a mechanism to support priorities among rules in case several rules are applicable to a certain situation. All priorities must be defined explicitly by the user. If a priority has been followed and it leads to a dead end the system will perform backtracking and apply the rule with the next lower priority.

With these new features, QUEL could process a query like

```
retrieve DEMAND PRIORITY (ROUTE.routing)
where ROUTE.start = "2423 Blake St., Berkeley"
and ROUTE.dest = "1122 Broadway, San Francisco"
```

as follows. The data manager finds that this query cannot be satisfied using only stored data. The keyword DEMAND indicates that backward chaining should be performed in this case. The backward chaining uses the rules in the knowledge base, thereby performing a hierarchical decomposition of the route. If several rules apply to a given situation, the keyword PRIORITY causes the data manager to consult the user-defined priorities to resolve the conflict. Eventually there are no more rules applicable. The data manager collects the route pieces that were yielded by the hierarchical decomposition and passes them to the user in the object "ROUTE.routing". For a more detailed discussion of these issues see [Stonebraker85b].

6. MAP COARSENING

The comprehension of most human map readers often involves an initial conceptual coarsening of the given map. The coarsened version of the map allows to find an approximation of a solution path quickly. From this first approximation the human reader will continue with further refinements of this path, taking more and more details into account.

This approach is also a hierarchical decomposition of the problem. Instead of applying path decomposition rules that are based on local knowledge, the map is coarsened. On the coarsened map it is much easier for the search heuristics to recognize larger areas where the cost of moving is relatively low. The resulting "main road" paths are then subject to further refinements by means of a less coarsened map.

Instead of such a top-down approach one could also solve the problem in a bottom-up manner. The bottom-up approach first finds best paths from the start and destination points to any main road, then searches a coarser map for the best path between the two chosen points on the main roads.

This approach of coarsening the map is a generalization of the hierarchical algorithm described by Kung et al. [Kung84]. Their approach requires the available maps to be ordered in a tree. The leaf maps are detailed maps like plat maps, the higher levels contain

coarser maps like a state highway map or a USA map. To solve an overland search problem, one could use this tree in a bottom-up or a top-down manner, in a similar way as described above.

In the given abstraction of the overland search problem, the map coarsening will be done by merging adjacent polygons. Thereby the number of polygons will be substantially reduced and the path finding algorithm will yield a path much faster than on the original map.

Map coarsening is done by defining a new criterion for homogeneity, and applying this new criterion to the given map. One possibility would be to specify several disjoint cost ranges whose union is the set of positive real numbers, $[0,\infty)$, such as, for example, the following four ranges: $[0,10)$, $[10,30)$, $[30,55)$, $[55,\infty)$. Then the union of several polygons is considered to be homogeneous if all of their cost coefficients are in the same range. The coarsening algorithm will merge all adjacent polygons with a cost coefficient in the same range. The cost coefficient of a new polygon is, for example, the average of the cost coefficients of its component polygons.

Note that this range merging has the following nice feature. Depending on the ranges, the path finding algorithm performed on the coarsened map will yield paths of different characteristics. Suppose, for example, that a vehicle can only manage areas with a cost coefficient less than 100. Let us consider two coarsenings of a given map: coarsening A, coarsened according to the ranges $[0,100)$, $[100,\infty)$; and coarsening B, coarsened according to the ranges $[0,50)$, $[50,80)$, $[80,90)$, $[90,100)$, $[100,\infty)$. The path found on coarsening A might have long pieces in areas with cost coefficients close to 100, because here, FINDNEXT cannot distinguish between areas of cost 1 and areas of cost 99; on the other hand this path might be rather direct. The path found on coarsening B will try to avoid areas with cost coefficients between 80 and 100; it might, however, be a longer path.

Both coarsenings reflect different planning goals. Coarsening A reflects the goals of a planner who is looking for a feasible path that is as direct as possible. On the other hand,

coarsening B reflects the goals of a planner who would like to avoid areas that are close to the feasibility limit and is willing to accept a longer path for that reason. This feature allows the user to include his personal planning goals which is much harder to express in the rather inflexible concept of a scalar cost function. It is a well-known fact in artificial intelligence that the *optimal* solution (like the path with the minimum cost on the original map) is not always what the user really wants. A cost function like the one proposed in this paper is somewhat artificial in most cases; very often it will not be able to capture all the aspects that a human would take into consideration. Therefore, it might well be that the human would choose a solution that has a suboptimal cost, due to aspects that are apparently not reflected well enough in the cost function. In our application, that would mean that an expert might intuitively prefer a route A to a route B although route B has a lower total cost (according to the cost function) than route A.

An experienced user could now perform several coarsenings with different cost ranges, let the system find an optimal path for each of the coarsened maps, and then choose the path that seems most appropriate to him. The selection of appropriate cost ranges is a matter of practical experience with the system and has to be done by an experienced user.

A different coarsening strategy is the coarsening by standard deviation. Here, the user specifies a (possibly large) threshold for the standard deviation of grey levels (taken over all pixels) in a homogeneous region. Then the coarsening algorithm merges adjacent polygons if the standard deviation of the grey levels of their union region is below the new threshold. This kind of coarsening does usually not yield a unique result map. The result is sensitive to the order in which the polygons are considered for merging.

In both coarsening strategies, the coarsening is done by merging adjacent polygons. One way to do this is to retrieve the boundary segments of the original map that will *not* appear on the coarsened map. For each such boundary segment the two adjacent polygons are to be merged. This strategy is presented in section 6.1. Simpler strategies are possible

if the original map is given in a quadtree data structure. In this case one bottom-up traversal of the quadtree will yield the coarsened map. This case is discussed in section 6.2.

6.1. Map Coarsening by Direct Polygon Merging

In this section we only consider the coarsening by cost ranges. The coarsening by standard deviation could be performed in a similar manner, but it would not be fast enough for practical applications. There are two reasons for that. First, to do that it is necessary to compute the standard deviation of grey levels in the union of two given polygons. This has to be done once per pair of polygons that is considered for merging. However, to do this requires information about the area of the two polygons. If this information is not stored the computational overhead will be tremendous. Second, the result of the map coarsening by standard deviation is sensitive to the order in which the polygons are considered for merging. This fact yields additional computation overhead.

6.1.1. The Algorithm MERGEPOL

In order to merge adjacent polygons whose cost coefficients are in the same range, the following algorithm MERGEPOL is executed. MERGEPOL builds a graph that has one vertex per polygon of the original map, and initially no edges. For each pair of polygons (P_1 , P_2) such that P_1 and P_2 are adjacent in the original map, MERGEPOL considers the two cost coefficients of P_1 and P_2. If both coefficients are in the same cost range then the edge (P_1 , P_2) is inserted into the graph. The resulting graph *MERGE* contains one vertex per polygon and one edge between each pair of vertices whose corresponding polygons are to be merged.

Then MERGEPOL finds the connected components of *MERGE*. Each connected component corresponds to a polygon of the coarsened map and vice versa. MERGEPOL merges all polygons whose corresponding vertices are in the same connected component. The merging is done pairwise. A pair of polygons can be merged by concatenating the boundaries of the two polygons in the following manner.

Let $(\pi_1 , \pi_2 , \ldots , \pi_n , \pi_1)$

and $(\rho_1 , \rho_2 , \ldots , \rho_m , \rho_1)$

be the boundary point sequences of the two adjacent component polygons P_1 and P_2, respectively. W.l.o.g., let

$$(\pi_1 = \rho_m) \quad \text{and} \quad (\pi_n = \rho_1)$$

Hence, the boundary segment

$$(\pi_1 , \pi_n) = (\rho_1 , \rho_m)$$

is the one that is shared by the two polygons. Then

$$(\pi_1 , \pi_2 , \ldots , \pi_n = \rho_1 , \rho_2 , \ldots , \rho_m = \pi_1)$$

is a boundary point sequence of the result polygon $P_1 \bigcup P_2$. Note that the boundary may contain several points more than once. This is the case if P_1 and P_2 have more than one boundary segment in common. Then $P_1 \bigcup P_2$ might even have holes. Figure 4 gives an example.

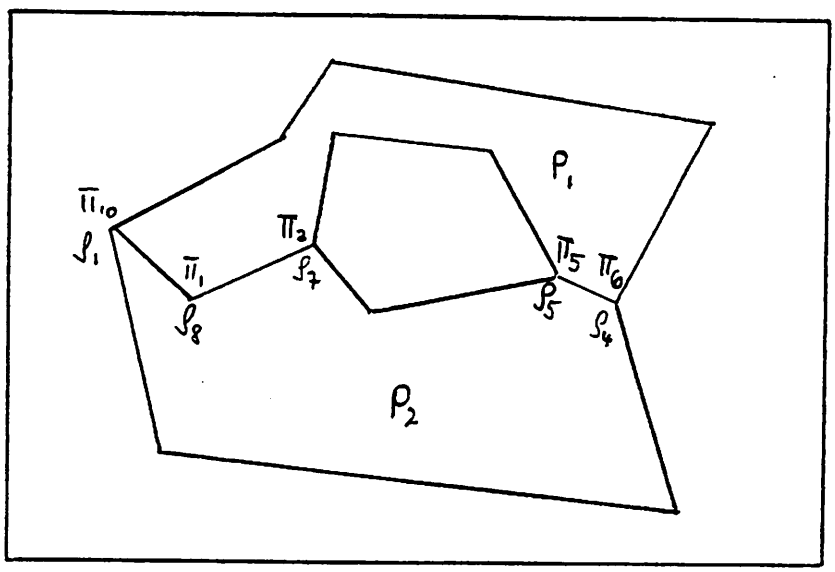


Figure 4: The polygons P_1 and P_2 have three edges in common: edge $(\pi_{10} , \pi_1) = (\rho_1 , \rho_8)$, edge $(\pi_1 , \pi_2) = (\rho_8 , \rho_7)$, and edge $(\pi_5 , \pi_6) = (\rho_5 , \rho_4)$. Hence, the boundary of the polygon $P_1 \bigcup P_2$,

$$(\pi_1 , \pi_2 , \ldots , \pi_{10} = \rho_1 , \rho_2 , \ldots , \rho_8 = \pi_1)$$

contains three points twice: points $\pi_2 = \rho_7$, $\pi_5 = \rho_5$, and $\pi_6 = \rho_4$. The polygon $P_1 \bigcup P_2$ has a hole.

As discussed in section 3, it is not very convenient to have polygons with complicated boundaries like this one. They tend to make the search for a minimum cost path more difficult. Furthermore, path generation rules often include some rule that makes the path follow some polygon boundary (such as rule 7). In this case, the result path might have a very undesirable shape. We therefore suggest to decompose these polygons into components without holes. In many cases, this can be done by $k+1$ cuts where k is the number of holes. We are currently working on a more detailed analysis of this kind of polygon decomposition. A simple example with k=3 is given in figure 5.

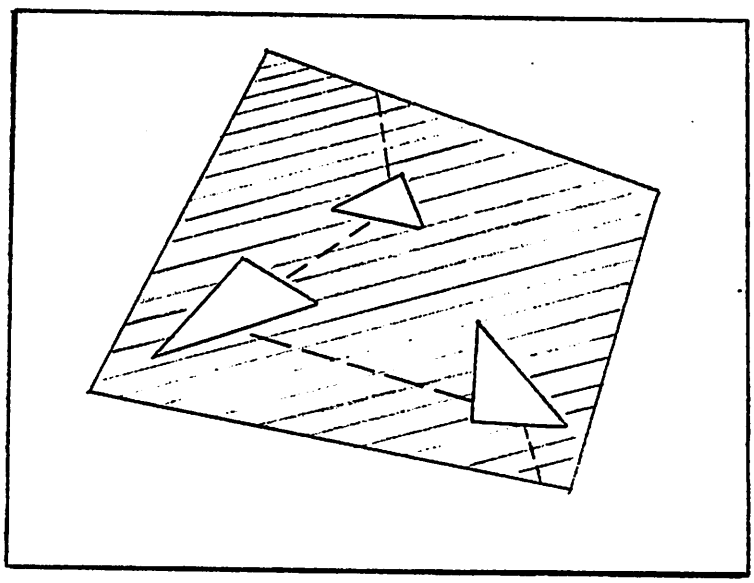


Figure 5: The shaded polygon which has three holes can be decomposed into two polygons without holes by the three dashed cuts.

The time for the map coarsening by direct polygon merging is about linear in the number of adjacencies, hence at most about quadratic in the number of polygons on the map.

6.1.2. Implementation of MERGEPOL

The algorithm MERGEPOL can be implemented in a very straightforward manner if the following relations are available and initialized with the data of the map.

MAP (polygon,cost)
ADJACENT (pol-1,pol-2)

First, MERGEPOL retrieves pairs of adjacent polygons whose cost coefficients are both in the same cost range, i.e. the edges of the graph *MERGE*. For that purpose one has to perform the following retrieve command for each cost range R_i. Let *lower* and *upper* be the limits of the cost range R_i.

range of ma1, ma2 is MAP
retrieve into MERGEDGE (ADJACENT.all)
where (ADJACENT.left-pol = ma1.polygon and ADJACENT.right-pol = ma2.polygon
and (ma1.cost $\geq$ *lower* and ma1.cost $<$ *upper* and ma2.cost $\geq$ *lower* and ma2.cost $<$ *upper*)

Then MERGEPOL finds the connected components of *MERGE*, merges the polygons in the same connected component one by one, and builds the boundaries of the result polygons. This is done by a straightforward implementation of a labeling algorithm like the one in [Papadimitriou83], and of the boundary concatenation presented in section 6.1. Finally, it builds the adjacency relation of the coarsened map.

So far we ran one experiment to test the performance of the map coarsening by polygon merging. The original map is a topographic map of a 36 square mile area of Santa Cruz county, California. It is represented by 420 adjacent polygons; the adjacency relation has 2039 tuples. We chose the following cost ranges: [0,30), [30,100), and [100,∞). Of the 420 polygons, the cost of 174 polygons is in the first range, the cost of 149 polygons in the second, and the cost of 97 polygons in the third.

First the program retrieves the relation MERGEDGE and finds the connected components of *MERGE*. This takes 930 CPU seconds INGRES time and 21 CPU seconds FORTRAN time. Then it performs a pairwise polygon merging by boundary concatenation; finally it builds the new adjacency relation. This takes 2700 CPU seconds INGRES time and 1897 CPU seconds FORTRAN time. Hence, the coarsening takes altogether 5548 seconds or about one and a half hour of CPU time. The resulting map is represented by 143 adjacent polygons. The new adjacency relation has 1293 tuples.

6.2. Map Coarsening by Quadtrees

So far the map was represented by a set of adjacent polygons, such that the cost of moving between two points within a polygon is proportional to their distance. The shape of the polygons was allowed to be arbitrary; even holes were allowed. This section considers a special case of this model that allows the representation of a map as a quadtree [Samet84]. The polygons are all squares, and their edge lengths are powers of 2. Each square of the map corresponds to one leaf of the quadtree. The square partition is obtained by a recursive subdivision of the map into four equal-size quadrants until the area in each square is homogeneous. For an example for such a partition and its corresponding quadtree see figure 6. The letters and numbers in figure 6 will be explained later.

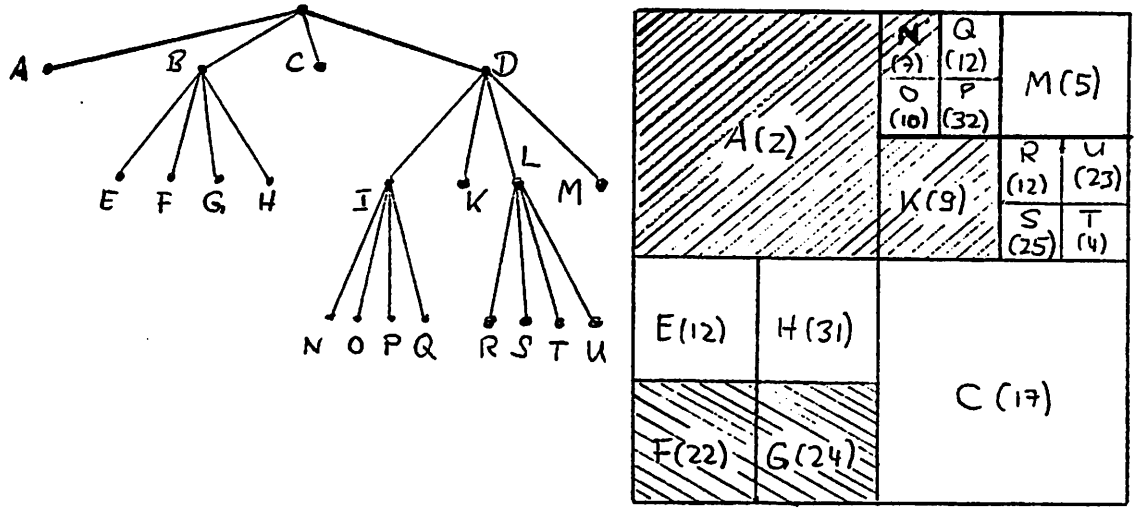


Figure 6: A partition and its corresponding quadtree.

Quadtrees are useful to implement the map making process described in section 3. In this process, several special-purpose maps are overlaid. The resulting map represents a cost function which incorporates all data about vehicle and terrain that has to be taken into account. If the special-purpose maps are given as quadtrees, then the quadtree of the result map is obtained by overlaying these quadtrees. The overlay operation is a slight variation of the union operation for which there are fast and simple algorithms [Hunter79]. The resulting quadtree has a node at every position at which any of the input quadtrees has a node. Its leafs represent the homogeneous subareas of the composed map. The cost

coefficients of each subarea are determined by the (maybe weighted) average of the cost coefficients of the corresponding subareas in the single layers.

Quadtrees are also useful to implement the map coarsening. One bottom-up traversal of the tree is sufficient. Hence, the time for the coarsening is about linear in the number of blocks. On the other hand, the time for coarsening by direct polygon merging is about linear in the number of *adjacencies*, which might be about quadratic in the number of polygons. However, the number of blocks might exceed the number of polygons in a simple polygon representation by far. It is therefore not immediately obvious which data structure might be better for the map coarsening. Both ways have to be implemented to allow a comparison.

6.2.1. Coarsening by Cost Ranges

If the original map is given in a quadtree one can apply a much more natural strategy to do map coarsening. Coarsening by ranges can be done bottom-up in the quadtree. Each leaf in the quadtree is assumed to have an associated cost coefficient. Each quadruple of leaf nodes is checked if all four associated cost coefficients are in the same range. If yes, the four leaf nodes are deleted, thereby making their ancestor node a leaf node. Its associated cost coefficient is the average of the cost coefficients of its four former descendants. This strategy requires only one bottom-up traversal through the quadtree.

If the granularity of the resulting map seems too fine, i.e. the number of polygons is still high, one might even merge adjacent quadrants which have different ancestors. This may be done if the cost coefficients of the quadrants are in the same cost range. The merging may continue until there are no more pairs of adjacent polygons on the map whose cost coefficients are in the same cost range. A detailed description of this kind of merging can be found in [Horowitz76].

Note, however, that the resulting partition may no longer be representable by a quadtree. Hence, this kind of coarsening should only be performed if the quadtree data

structure is no more required. An example is given in figure 6. Suppose, a bottom-up coarsening with the cost ranges [0,10), [10,20), [20,30), [30,40), [40,∞) yielded the quadtree and the corresponding polygon partition that are given in figure 6. The letters denote the polygons, and the circled numbers are their corresponding cost coefficients. Then it is possible to coarsen the polygon partition by merging the squares A, N, and K, as well as the squares F and G because their corresponding cost coefficients are in the same cost ranges. The resulting polygon partition can no longer be represented by a quadtree.

Note that the polygon partition resulting from this coarsening by ranges is unique for a given input quadtree and given cost ranges.

6.2.2. Coarsening by Standard Deviation

Given a quadtree of the original map, one may also apply a different coarsening strategy: coarsening by standard deviation. However, this is only possible if the construction of the original quadtree used the standard deviation of grey levels as criterion for homogeneity. In this case, the standard deviation of grey levels in each quadrant is below a given threshold t.

Let Q_t be the quadtree obtained from the original map by applying the threshold t. If t^* is greater than t then the map represented by the quadtree Q_{t^*} is a coarsened version of the map represented by Q_t. The problem is, how to obtain quickly a quadtree Q_{t^*} for a given coarsening parameter t^* and a given quadtree Q_t ($t \ll t^*$). Of course, one does not want to repeat the whole process of building a quadtree in order to obtain Q_{t^*}.

This kind of coarsening can be performed easily with one bottom-up traversal of the quadtree Q_t. Each leaf of the quadtree Q_t is required to store the mean and the standard deviation of the grey levels of the associated map area (taken over all pixels). Let X, Y, Z, and W be four sibling leafs in Q_t and let A be their ancestor. Furthermore, let σ_x, σ_y, σ_z, σ_w, σ_a be their standard deviations, and $\bar{x}$, $\bar{y}$, $\bar{z}$, $\bar{w}$, $\bar{a}$ their means, correspondingly. Because the number of pixels in the areas corresponding to X, Y, Z, and W is all the same

(namely one quarter of the number of pixels in the area corresponding to A), the following equation holds.

$$\sigma_a^2 = \frac{1}{4}\left(\sigma_x^2 + \sigma_y^2 + \sigma_z^2 + \sigma_w^2\right) + \frac{3}{16}\left(\bar{x}^2 + \bar{y}^2 + \bar{z}^2 + \bar{w}^2\right) - \frac{1}{8}\left(\overline{xy} + \overline{xz} + \overline{xw} + \overline{yz} + \overline{yw} + \overline{zw}\right)$$

Hence, one can just perform a bottom-up traversal of the quadtree Q_t, compute for each quadrupel of leafs the standard deviation of their ancestor (according to the above equation) and check if it is no more than t^*. If yes, the four leafs are deleted, thereby making their ancestor a leaf. Its standard deviation has just been computed, its mean is the mean of the means of its four descendants. The resulting quadtree is the quadtree Q_{t^*}, and it represents the coarsened map. Again, the result of this coarsening strategy is unique, because only sibling leafs are merged.

After this coarsening it is also possible to merge quadrants that have different ancestors. This may be done if the union of the quadrants is still homogeneous. The merging may continue until all yielded polygons are maximal homogeneous regions. This merging process does not yield a unique result; the result is sensitive to the order in which the polygons are considered for merging. In their "split-and-merge" algorithm, Horowitz and Pavlidis [Horowitz76] solve the problem by imposing an arbitrary order on the polygons. Again, the resulting map may no more be representable as a quadtree. Hence, this kind of coarsening should be performed only if the quadtree data structure is no more required.

7. CONCLUSIONS

We presented some solution strategies for the overland search problem. The resulting system is based on rules and the utilization of stored knowledge about the area. Our experiences with using an extended query language for the implementation are very good. The data management is becoming a lot easier. Due to the extensions, we did not encounter major problems when programming the algorithms and rule constructs.

Our future research will focus on the implementation of the quadtree approach and on a more detailed investigation of the hierarchical problem decomposition proposed in section 5. Furthermore, we are interested in heuristic algorithms to decompose polygons into simpler components. We especially intend to work on fast heuristics for decomposition into convex components, as well as on algorithms to decompose polygons into components without holes, and into components whose number of vertices is below a given threshold.

Acknowledgements

I would like to thank my advisor, Mike Stonebraker, for introducing me to this area of research and for his continuing support and encouragement. I have also benefited greatly from many discussions with Ru-Mei Kung and from the support of ESL Inc., Sunnyvale, Ca. where the system was implemented.

References

[Aho74] Aho, A., Hopcroft, J., and Ullman, J., "The Design and Analysis of Computer Algorithms", Addison Wesley, Reading, Mass., 1974.

[Butler85] Butler, M., private communication, 1985.

[Chazelle79] Chazelle, B., and Dobkin, D., "Decomposing a Polygon into its Convex Parts", Proc. 11th Annual ACM Symposium on Theory of Computing, pp. 34-48, 1979.

[Guttman84] Guttman, A., "New Features for a Relational Data Base System to Support Computer Aided Design", PhD Thesis, University of California, Berkeley, June 1984.

[Harmon84] Harmon, S.Y., Gage, D.W., Aviles, W.A., and Bianchini, G.L., "Coordination of Intelligent Subsystems in Complex Robots", Proc. The First Conference on Artificial Intelligence Applications, IEEE, Dec. 1984.

[Hayes-Roth83] Hayes-Roth, F., et al. (eds.), "Building Expert Systems", Addison Wesley, Reading, Mass., 1983.

[Horowitz76] Horowitz, S.L., and Pavlidis, T., "Picture Segmentation by a Tree Traversal Algorithm", J. ACM 23, 2, pp. 368-388, April 1976.

[Hunter79] Hunter, G.M., and Steiglitz, K., "Operations on Images Using Quadtrees", IEEE Trans. Pattern Anal. Mach. Intell. 1, 2, pp. 145-153, April 1979.

[Isik84] Isik, C., and Meystel, A., "Knowledge-Based Pilot for an Intelligent Mobile Autonomous System", Proc. The First Conference on Artificial Intelligence Applications, IEEE, Dec. 1984.

[Keil83] Keil, J.M., "Decomposing Polygons into Simpler Components", Ph.D. thesis, Dept. Computer Science, U. of Toronto, 1983.

[Kuan84] Kuan, D., et al., "Automatic Path Planning for a Mobile Robot Using a Mixed Representation of Free Space", Proc. The First Conference on Artificial Intelligence Applications, IEEE, Dec. 1984.

[Kung84] Kung, R., et al., "Heuristic Search in Data Base Systems", Proc. 1st International Conference on Expert Database Systems, Kiowah, S.C., October 1984.

[Kung85] Kung, R., private communication, 1985.

[Meystel84] Meystel, A., "Automated Map Transformation for Unmanned Planning and Navigation", Proc. 9th W.T. Pecora Memorial Remote Sensing Symposium, IEEE, Oct. 1984.

[Papadimitriou83] Papadimitriou, C. and Stieglitz, K., "Combinatorial Optimization", Prentice Hall, 1983.

[Parodi84] Parodi, A., "A Route Planning System for an Autonomous Vehicle", Proc. The First Conference on Artificial Intelligence Applications, IEEE, Dec. 1984.

[Pearl84] Pearl, J., "Heuristics: Intelligent Search Strategies for Computer Problem Solving", Addison Wesley, Reading, Mass., 1984.

[RTI84] Relational Technology Inc., "INGRES/EQUEL/FORTRAN User's Guide", Version 3.0, VAX/VMS, Oct. 1984.

[Samet84] Samet, H., "The Quadtree and Related Hierarchical Data Structures", Computing Surveys, Vol. 16, No. 2, pp. 187-260, 1984.

[Stonebraker76] Stonebraker, M., et al., "The Design and Implementation of INGRES", ACM Transactions on Database Systems, Vol. 1, No. 3, pp. 189-222, Sep. 1976.

[Stonebraker83] Stonebraker, M., "Application of Abstract Data Types and Abstract Indices to CAD Data", Proc. Engineering Applications Stream of the ACM-SIGMOD International Conference on Management of Data, San Jose, Ca., May 1983.

[Stonebraker85a] Stonebraker, M., et al., "Extending a Data Base System with Procedures", unpublished manuscript, University of California, Berkeley, Ca., 1985.

[Stonebraker85b] Stonebraker, M., "Triggers and Inference in Data Base Systems", Proc. 1985 Islamoora Conference on Expert Systems, Islamoora, Fla., Feb. 1985 (to appear as Springer Verlag book, edited by M. Brodie).

Appendix A: Code for the Path Generation Rules

The solution uses the concept of abstract data types to implement the data types 'polygon', 'line', 'path', and 'point'. The following operations are defined on those types.

><	...	stay-in: line-1 >< pol-1 returns a path of line segments with endpoints in common with line-1 which stays inside pol-1
@	...	adjacent; this operator is simulated by a relation ADJACENT (pol-1,pol-2)
//	...	closest
()	...	in
&	...	concatenation
!!	...	intersection
#(X,α)	...	area covered by angle α that is rooted at X and halved by (X,Ω)
line(X,Y)	...	line from point X to point Y
length(P)	...	length of path P

The solution uses the following relations:

MAP (polygon,cost)	...	polygons of the map
PATHS (path,cost,cur-poly,cur-pt,next-poly,next-pt)	...	paths
OLD (node,next-poly)	...	pairs (π_{nex} ,Ψ_{nex}) that have occurred previously

It follows the code.

```
range of m,m1,m2,m3 is MAP
range of p is PATHS
range of o is OLD
```

INITIALIZATION:

```
append to CAND (path = line(start,start),
                cost = 0,
                cur-poly = m.polygon,
                cur-pt = start,
                next-poly = m.polygon,
                next-pt = start)
where start () m.polygon

append to OLD (node = start, next-poly = p.next-poly)
```

MAIN LOOP:

```
execute* (RULES; append to OLD (node = p.next-pt, next-poly = p.next-poly))
where p.cur-pt != finish
```

RULES:

```
execute-until {

{RULE 1}

replace p (path = p.path & (line(p.cur-pt,p.next-pt) >< p.cur-poly)
                  & (line(p.next-pt,finish)   >< p.next-poly),
        cost = p.cost + m1.cost * length(line(p.cur-pt,p.next-pt) >< p.cur-poly)
                  + m2.cost * length(line(p.next-pt,finish)   >< p.next-poly),
        p.cur-pt = finish)

where m1.polygon = p.cur-poly
and   m2.polygon = p.next-poly
and   finish () p.next-poly

{RULE 2}

replace p (path = p.path & (line(p.cur-pt,p.next-pt) >< p.cur-poly),
        cost = p.cost + m1.cost * length(line(p.cur-pt,p.next-pt) >< p.cur-poly),
        cur-poly = p.next-poly,
        next-poly = m3.polygon,
        cur-pt = p.next-pt,
        next-pt = p.next-pt // (#(p.next-pt,α0) !! m3.polygon)

where m1.polygon = p.cur-poly
and   m2.polygon = p.next-poly
and   m3.cost = min (m.cost
                where m.polygon @ (p.next-poly !! #(p.next-pt,α0))
                and   m.cost ≤ m2.cost)
and   any (o.node where o.node = p.next-pt // (#(p.next-pt,α0) !! m3.polygon
                and o.next-poly = m3.polygon)) = 0

{RULE 3}

replace p (path = p.path & (line(p.cur-pt,p.next-pt) >< p.cur-poly),
        cost = p.cost + m1.cost * length(line(p.cur-pt,p.next-pt) >< p.cur-poly),
        cur-poly = p.next-poly,
        next-poly = m2.polygon,
        cur-pt = p.next-pt,
        next-pt = (p.next-poly !! #(p.next-pt,α0) !! m2.polygon) // finish)

where m1.polygon = p.cur-poly
and   m2.cost = min (m.cost
                where m.polygon @ (p.next-poly !! #(p.next-pt,α0))
                and   m.cost ≤ G )
and   any (o.node where o.node = (p.next-poly !! #(p.next-pt,α0) !! m2.polygon) // finish
                and o.next-poly = m3.polygon) = 0

**********************************************************************************************
Insert here the code for RULE 2 and RULE 3 where α0 is replaced by αi  (i = 1 , . . . . , k).
**********************************************************************************************
```

```
{RULE 4}

replace p (path = p.path & (line(p.cur-pt,p.next-pt) >< p.cur-poly),
        cost = p.cost + m1.cost * length(line(p.cur-pt,p.next-pt) >< p.cur-poly),
        cur-poly = p.next-poly,
        next-poly = m3.polygon,
        cur-pt = p.next-pt,
        next-pt = p.next-pt // m3.polygon

where m1.polygon = p.cur-poly
and   m2.polygon = p.next-poly
and   m3.cost = min (m.cost
                where m.polygon @ p.next-poly
                and   m.cost ≤ m2.cost)
and   any (o.node where o.node = p.next-pt // m3.polygon
                  and o.next-poly = m3.polygon) = 0

{RULE 5}

replace p (path = p.path & (line(p.cur-pt,p.next-pt) >< p.cur-poly),
        cost = p.cost + m1.cost * length(line(p.cur-pt,p.next-pt) >< p.cur-poly),
        cur-poly = p.next-poly,
        next-poly = m2.polygon,
        cur-pt = p.next-pt,
        next-pt = (p.next-poly !! m2.polygon) // finish

where m1.polygon = p.cur-poly
and   m2.cost = min (m.cost
                where m.polygon @ p.next-poly
                and   m.cost ≤ G )
and   any (o.node where o.node = (p.next-poly !! m2.polygon) // finish
                  and o.next-poly = m2.polygon) = 0


{Algorithm gives up}

replace p (path = p.path & line(p.cur-pt,finish),
        cost = ∞,
        cur-pt = finish)

} {end of RULES}
```

Appendix B: Building a Map from its Boundary Segments

During our investigations about efficient algorithms for the map coarsening by ranges we also considered a constructive approach as follows. The algorithm selects the boundary segments that will appear on the coarsened map, and builds the coarsened map from these boundary segments. We found that this algorithm is more complicated and more time-consuming than the one presented in section 6.1. We present it in the appendix because we think that this algorithm is interesting in its own right.

The map is built by executing the procedure BUILDMAP for each cost range R_i. For each boundary segment in the original map, BUILDMAP considers the cost coefficients of the two polygons that are adjacent to this segment. BUILDMAP finds the segments where exactly one of the two coefficients is in R_i. Let LST_i be the set of those segments. Note that the segments where both coefficients are in R_i are not part of the coarsened map.

From the line segments in LST_i, BUILDMAP produces the vertex and edge lists of the graph G_i that is formed by these line segments. Then BUILDMAP finds the connected components of G_i. Let $LST_{i,j}$ be the subset of line segments in LST_i that belong to the j-th connected component. From the connected components BUILDMAP constructs the polygons that are bounded by the line segments in the graph. Note that each connected component corresponds to only one polygon but not necessarily to a simple one. Figure B1 shows a connected component whose corresponding polygon is not simple. Its boundary contains some points more than once. The shaded areas have a cost coefficient in R_i.

It seems obvious that the boundary of the polygon can be retrieved by tracing along the line segments. It turns out, however, that there is no one-pass boundary tracing algorithm without backtracking that succeeds in every instance of the problem. The reason is that there is no decision criterion how to continue at points that are in the boundary more than once. For example: in figure B1, at point π_1 coming from line segment ls_1, it will not work out to go left, i.e. to continue with line segment lsa_1. In order to obtain the whole boundary in one pass one has to go right. However, at point π_2 coming from line segment

ls_2, one has to go left and continue with line segment lsa_2. Otherwise the small loop at the bottom would be omitted. For a boundary tracing algorithm both situations are identical; there is no decision criterion what to do.

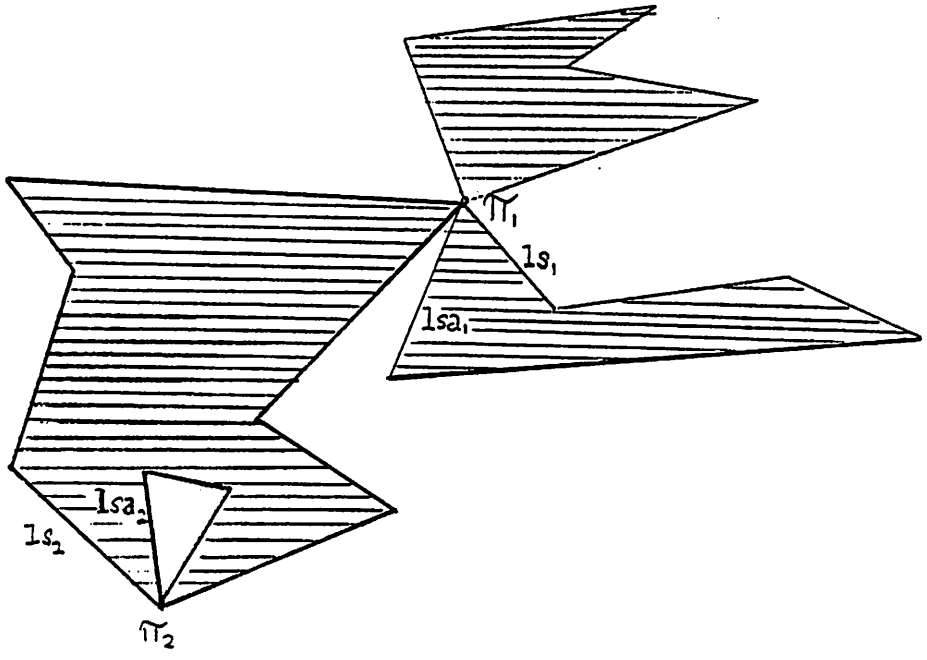


Figure B 1: A connected component whose corresponding polygon is not simple.

We therefore resort to an algorithm that traces the polygon boundaries in such a way that the area whose cost coefficient is in R_i is on the left hand sides of the boundary segments as they are traced. If the algorithm comes to a point with several choices to continue (i.e. there are several adjacent boundary segments that have not been traced yet), it chooses the leftmost one. This method will often yield several polygons from the same connected component. The "go left"-rule, however, has the following advantage. If it yields two or more polygons from the same connected component, none of them lie in the interior of another one. The importance of this observation will become clear later on.

With these considerations in mind, the boundary retrieval can be performed as follows. BUILDMAP first selects a connected component and an arbitrary line segment ls in the component's corresponding line segment set $LST_{i,j}$. BUILDMAP removes ls from $LST_{i,j}$. Let Λ_{ls} be its starting point and Ω_{ls} its end point such that the polygon whose cost coefficient is in R_i is on ls's left hand side.

To find the next boundary segment, BUILDMAP scans $LST_{i,j}$ for line segments that are adjacent to Ω_{ls}. If there is none then the boundary of a polygon is complete; the boundary is stored and BUILDMAP picks an arbitrary segment from $LST_{i,j}$ to continue with the next boundary. If there is one adjacent segment then this one is the next boundary segment. If there are several adjacent segments then BUILDMAP selects the one where the angle between the new segment and ls is minimal ("go left"-rule). For example, in figure B1, from line segment ls_1 line segment lsa_1 would be selected, from line segment ls_2 it would be line segment lsa_2. The new boundary segment is removed from $LST_{i,j}$ and BUILDMAP continues from there the same way to find the following boundary segment.

If $LST_{i,j}$ is eventually empty then BUILDMAP continues with the next connected component the same way.

After all connected components in G_i have been processed, BUILDMAP checks all polygons retrieved and finds the ones that are just holes of other polygons. A polygon P_{in} is a hole of P_{out} if it lies completely in the interior of P_{out}, if it does not lie in the interior of any other polygon that lies in P_{out} and if the cost coefficient of the area $P_{out} - P_{in}$ is in R_i. Note that the boundaries of P_{in} and P_{out} have to be disjoint. For example, in figure B1 the loop at the bottom is not considered to be a hole.

In order to find all such pairs (P_{out}, P_{in}), BUILDMAP first checks each pair of polygons (P_1, P_2) if P_1 and P_2 are in different connected components of G_i and if P_2 lies completely within P_1. (As pointed out above, if P_1 and P_2 both belonged to the same connected component none of them could be a hole of the other.) Let LCW_i be the set of such pairs. Then BUILDMAP deletes the pairs in LCW_i that are just transitive closure results of

other pairs in LCW_i. Let $LCWX_i$ be the set of remaining pairs.

Then BUILDMAP checks each pair (P_1,P_2) in $LCWX_i$ *one by one* if there is another pair (P_1^*,P_2^*) in $LCWX_i$ such that $P_1 = P_2^*$. If yes, BUILDMAP continue with the next pair. If no, then P_2 is a hole of P_1. The pair (P_1,P_2) is added to the set $HOLE_i$ and deleted from $LCWX_i$. Furthermore, all pairs (P_2,Q) in $LCWX_i$ have to be deleted because P_2 cannot be an outer polygon anymore. This process is repeated until $LCWX_i$ is eventually empty.

For each polygon P_{out} that is not a hole, BUILDMAP retrieves all pairs

$$(P_{out},P_{in,1}), \ldots, (P_{out},P_{in,k})$$

from the set $HOLE_i$.

Finally, BUILDMAP builds the boundary of the difference polygon

$$P_{out} - P_{in,1} - \ldots - P_{in,k}$$

and stores it in the polygon set of the coarsened map. The boundary of the difference polygon can be obtained by concatenating the boundaries of the outer polygon and its holes in the following manner. Let

$$(\pi_1, \pi_2, \ldots, \pi_n, \pi_1)$$

and

$$(\rho_1, \rho_2, \ldots, \rho_m, \rho_1)$$

be the boundary point sequences of an outer polygon P_{out} and one of its holes P_{in}, respectively. Then

$$(\pi_1, \pi_2, \ldots, \pi_n, \rho_1, \rho_2, \ldots, \rho_m, \rho_1, \pi_n, \pi_1)$$

is a boundary point sequence of the difference polygon. Note that the boundary contains several points more than once. The polygon might even be self-intersecting because this concept introduces two artificial edges, the edges (π_n, ρ_1) and (ρ_1, π_n). Figure B2 gives an example.

The algorithm BUILDMAP can be described more concisely as follows.

(1) For each boundary segment consider the cost coefficients of the two polygons that are adjacent to this segment. Find the segments where exactly one of the two coefficients is in R_i. Let LST_i be the set of those segments.

(2) Build the vertex list and the edge list of the graph G_i that is formed by the segments in LST_i.

(3) Find the connected components of G_i.

(4) From these connected components, find the polygons whose cost coefficient is in R_i, using the "go left"-rule. Let $POLY_i$ be the set of all those polygons.

(5) Determine the relation

$$HOLE_i = \{ (a,b): a,b \in POLY_i, \text{ b is a hole of a } \}$$

$$= \{ (a,b): a,b \in POLY_i, \text{ b lies completely within a,}$$

$$\text{there is no } c \in POLY_i \text{ such that c lies within a and b lies within c,}$$

$$\text{the cost coefficient of the area in a - b is in } R_i \}$$

by determining first

$$LCW_i = \{ (a,b): a,b \in POLY_i, \text{ b lies completely within a,}$$

$$\text{the boundaries of a and b belong to different connected components of } G_i \},$$

then

$$LCWX_i = \{ (a,b): (a,b) \in LCW_i, \text{ there is no } x \in POLY_i: (a,x) \in LCW_i, (x,b) \in LCW_i \}$$

Clearly it is $HOLE_i \subseteq LCWX_i \subseteq LCW_i$.

(6) For each polygon P_{out} in $POLY_i$ such that there is no $X \in POLY_i : (X, P_{out}) \in HOLE_i$ build the polygon corresponding to

$$P_{out} - \bigcup_{(P_{out}, P_{in}) \in HOLE_i} P_{in}$$

and add it to the set that includes all polygons of the coarsened map.

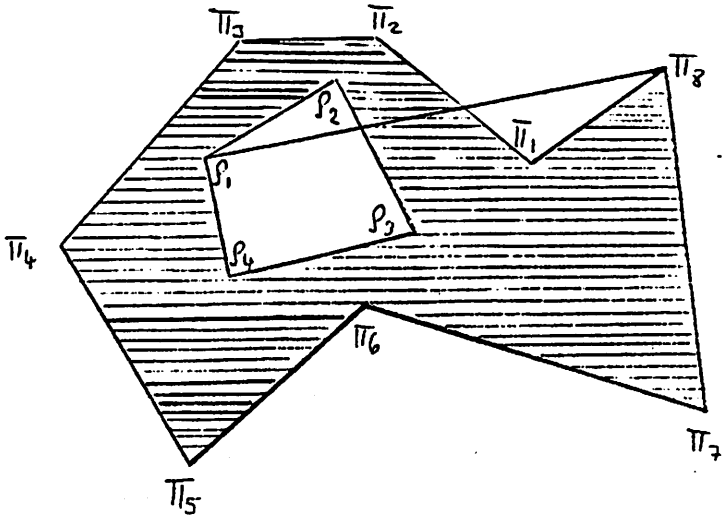
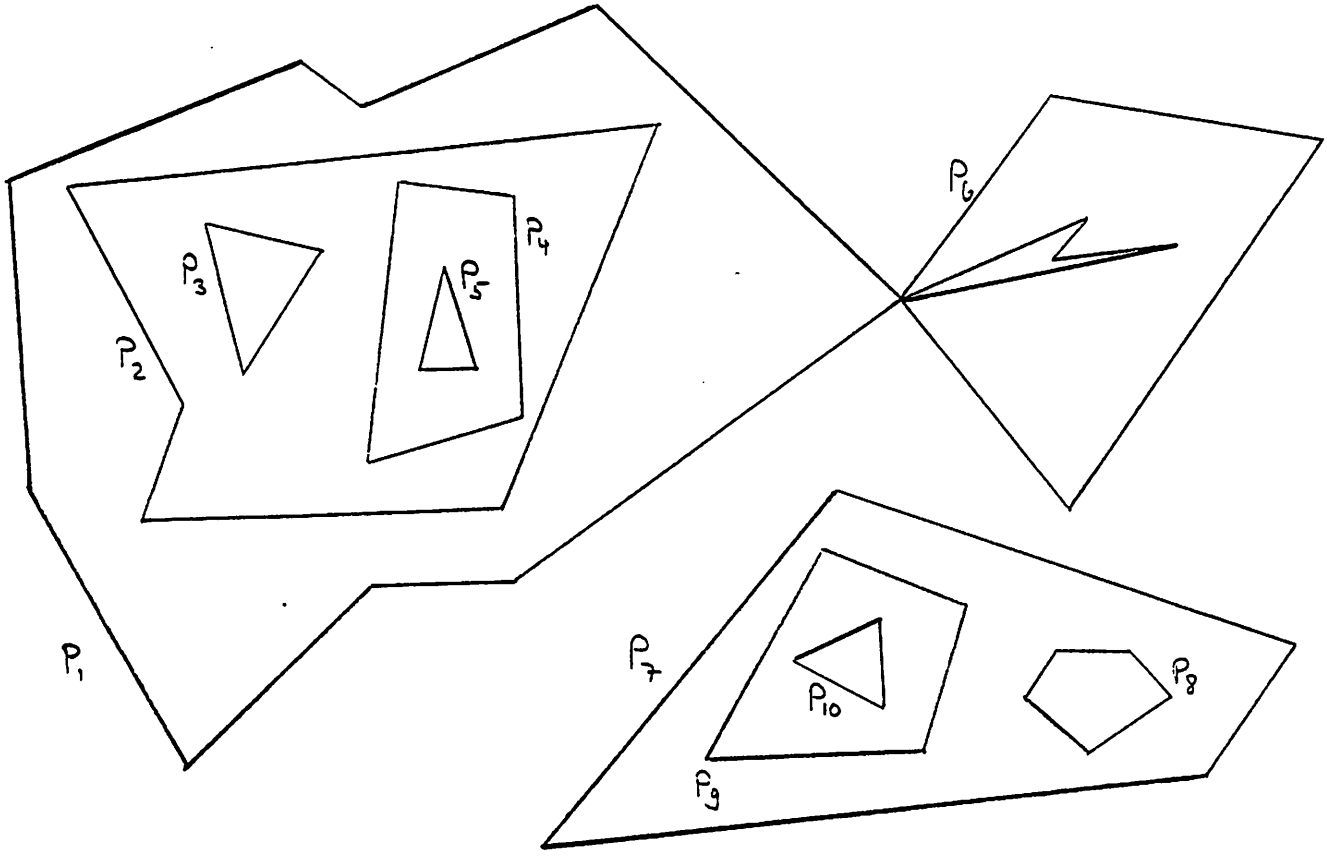


Figure B2: P_{out}, P_{in}, and P_{out} - P_{in} (shaded). P_{out} - P_{in} is self-intersecting because of the two artificial edges (π_8,ρ_1) and (ρ_1,π_8).

Let us give a brief example for steps (5) and (6) of the algorithm BUILDMAP. Suppose step (4) yields the following ten polygons.



From there one obtains

$$LCW_i = \{ (P_1,P_2), (P_1,P_3), (P_1,P_4), (P_1,P_5), (P_2,P_3), (P_2,P_4), (P_2,P_5), (P_4,P_5), (P_7,P_8), (P_7,P_9), (P_7,P_{10}), (P_9,P_{10}) \}$$

and

$$LCWX_i = \{ (P_1,P_2), (P_2,P_3), (P_2,P_4), (P_4,P_5), (P_7,P_8), (P_7,P_9), (P_9,P_{10}) \}$$

and

$$HOLE_i = \{ (P_1,P_2), (P_4,P_5), (P_7,P_8), (P_7,P_9) \}$$

If polygons with holes are allowed, step (6) yields six polygons.

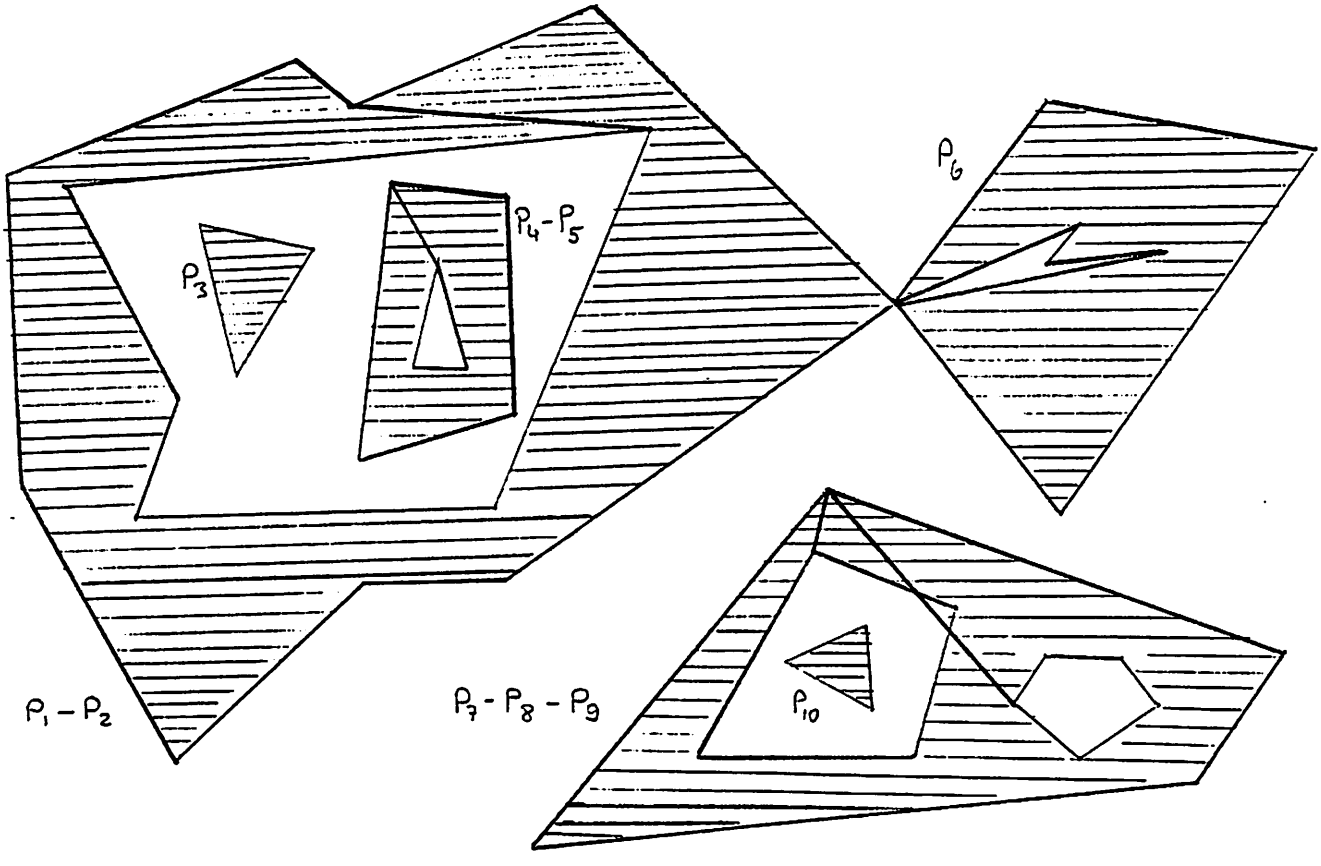


Figure B 4: $P_1 - P_2$, P_3, $P_4 - P_5$, P_6, $P_7 - P_8 - P_9$, P_{10}