Robotica (2001) volume 19, pp. 125–135. Printed in the United Kingdom © 2001 Cambridge University Press

A rapid method for planning paths in three dimensions for a small aerial robot

M. Williams and D.I. Jones

School of Informatics, University of Wales, Dean Street, Bangor, Gwynedd LL57 1UT, UK

(Received in Final Form: July 17, 2000)

SUMMARY
This paper describes a path planning method for a small autonomous aerial vehicle to be used for inspecting overhead electricity power lines. A computational algorithm is described which converts a standard three dimensional array representation of one or more obstacles in the vehicle's environment into an octree and a connectivity graph. This achieves a significant reduction in computer memory usage and an increase in execution speed when the graph is searched. Path planning is based on a three-dimensional extension of the distance transform. Test results demonstrate rapid and effective operation of the planner within different workspaces.

KEYWORDS: Aerial robot; Path planning; Three dimensions; Distance transform.

1. INTRODUCTION

Planning paths is a central requirement of mobile robotics. It is essential that the vehicle is able to move between two points without causing damage to itself or its surroundings and without an impractical limitation being imposed on its speed of movement. Many methods have been proposed to achieve this although most of the literature describes path planning methods for ground based, that is, two-dimensional environments. The application considered here is different being concerned with path planning methods for a small autonomous aerial vehicle under consideration for use in video inspection of overhead power lines. The primary requirement is to plan paths in three-dimensions quickly and reliably. In particular, should an obstacle be discovered on the nominal flight path, an alternative path must be produced from the vehicle's current position to a position of safety and thence to re-join the nominal path once the obstacle has been passed. The new path need not be particularly distance or time efficient but it must be generated rapidly and be safe. This paper describes a three-dimensional path planner, based on the distance transform and an octree and connectivity graph representation of the vehicle's environment, which meets these goals. Its main contribution is a rapid and efficient computational method that allows path planning to be implemented with the limited resources on-board the vehicle.

In the United Kingdom alone, the electricity companies together own about 150,000 km of overhead distribution lines that must be inspected at regular intervals. Routine inspection is often performed from a helicopter but there are advantages to using an unmanned aerial vehicle instead. The concept and requirements for a "Robot Inspector of Power Lines" based on a small, remotely piloted helicopter (RPH) such as 'Sprite' (see Figure 1) has been described by Jones & Earp.[1] Many of the sub-systems needed to realise the concept already exist but it is apparent that one of the major barriers to adoption of this technology is the regulatory requirement for air-worthiness. Thus the Civil Aviation Authority (CAA) requires that the reliability and safety of the vehicle and its support system are such that no more than a given number of fatal accidents occur per million flying hours, typically 3 to 10 per million hours. A more detailed account of the issues involved and the motivation for this work are to be found in the paper by Williams *et al.*[2] Conventionally, it is the pilot's function to recognise potential hazards to the aircraft or hazards which the aircraft itself may cause and hence to take appropriate avoiding action. The work reported here is part of a programme of research investigating whether it is possible to automate this function, at least in part, by the use of machine vision. Specifically, if an unexpected static obstacle is detected on the flight path, path planning is required to plot an avoiding course and then return to routine inspection of the overhead line once past the obstacle. It is essential that this be done on-board the RPH because it is a function that must be retained even during failure of the communications link to the ground station.

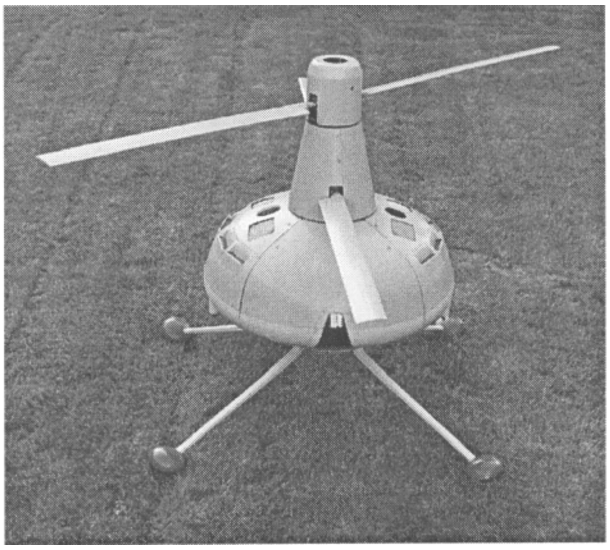
Fig. 1. Sprite miniature rotorcraft.

Automatic navigation and obstacle detection for manned helicopters is an established research topic (e.g. Bhanu *et al.*),[3] usually in a military context – typically for 'nap-of-the-earth' flight or to aid an incapacitated pilot. These systems often use a number of different sensors to obtain information about the airspace around the helicopter including millimetre wave radar and laser ranging systems. Generally, the sophistication, expense and size of such systems preclude their use in a small RPH and it is necessary to seek a relatively low cost solution which is compatible with commercial budgets and appropriate to the limited physical volume and electrical power available. Similar design constraints apply to the path planner – unlike the hostile and rapidly changing battlefield, the civil environment considered here is relatively benign but the limitations on sensors and computing power are severe. Many different approaches to path planning have been proposed such as the potential field, cell decomposition and roadmap methods; a comprehensive survey is to be found in Latombe's book.[4] However, the method which best matches our requirements is based on the distance transform which is extended here to the case of three dimensions. The concept of the distance transform and its suitability for this application is described in the next section. Section 3 is a detailed discussion of the algorithm for constructing the connectivity graph, which is crucial to minimising computer memory and achieving rapid path computation. Section 4 discuses how paths are derived from the graph and discusses the data structures and implementation details of the code. The method was tested by simulation and the results presented in Section 5 show its effectiveness.

2. PATH PLANNING WITH THE DISTANCE TRANSFORM

2.1 Principle of the distance transform approach

The distance transform was first presented by Jarvis and Byrne.[5] It is a relatively simple concept that is best grasped by means of a graphical example. Suppose that the two dimensional matrix in Figure 2(a) shows the occupancy of an environment where the black cells of the matrix are obstacles, the white cells are freespace, S marks the start cell and G the goal cell. The distance transform is calculated using the connectivity of the matrix structure. Starting at the goal cell, which has a distance transform of zero, successive connected cells are incrementally labelled with the next positive integer. This process can be thought of as an expanding wave centred on the goal location that flows through the matrix structure. On completion, all free-space cells that can be reached from the goal have been labelled with a distance transform value; thus starting at **any** labelled free space cell guarantees that a path exists to the goal. Figure 2(b) shows the distance transform generated for Figure 2(a). Having computed the distance transform, a simple descent search is used to produce a cell sequence that is then interpolated and smoothed to give an actual motion path. In Figure 2(b), a number of paths from cell 'S' to cell 'G' are possible, one of which is shown in Figure 2(c).

2.2 Application of the distance transform in path planning

In recent years, several authors[6–8] have used the distance transform as a basis for two-dimensional path planning and exploration algorithms.

Shin[6] presents a fast motion planning algorithm for a mobile robot using the distance transform, although it is used in a different way to that described here. Shin uses the distance transform to measure how close an obstacle is to a free space location and the result is conceptually similar to applying a potential function to the workspace. Once the distance transform has been calculated, path planning can be performed. To simplify the process of path planning it is common to 'reduce' the robot to a point[4] which usually requires the obstacles to be grown by at least the same amount as the robot was reduced. However, this is not necessary when using the distance transform because the distance from a freespace cell to the nearest obstacle is implicit to the transform. By using a guard space of at least the amount by which the robot was reduced, an effective safety margin can be placed around obstacles.

Zelinsky[7] presents an exploration algorithm using the distance transform where the robot's environment is initially totally unknown. Zelinsky's implementation uses the distance transform in the same mode as presented here. The unknown area is treated as "free-space" until an obstacle is found using tactile sensors located on the robot. Using the sensed data, a local map of the environment is constructed but the exploration concludes once the goal has been

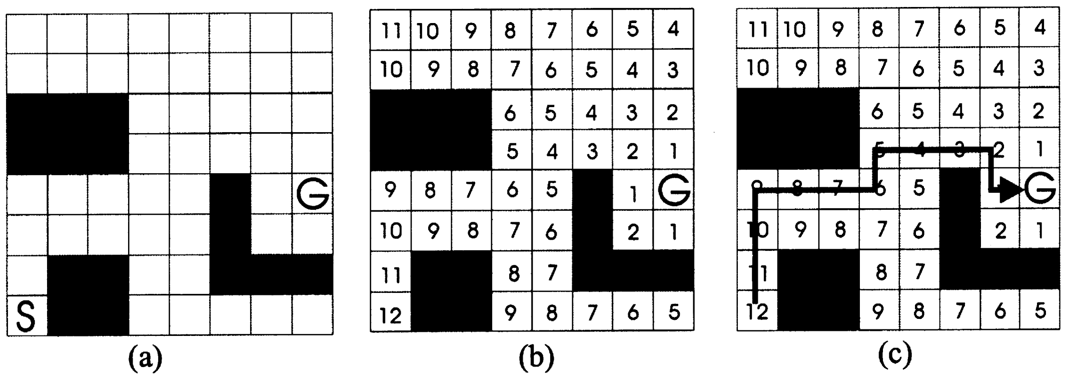

Fig. 2. Graphical example of path planning using the distance transform.

achieved. In other applications of the distance transform,[9] it is necessary to discover the full details of the environment.

Jarvis has been an exponent of the distance transform producing several papers on its application such as the development of an all-terrain vehicle using sensor-fusion-based navigation.[8] This vehicle uses a number of sensors such as flux gate compasses, the differential global positioning system, range sensors to construct an environmental map and also explores the possibility of using stereopsis ranging and natural landmark based localisation. Jarvis claims that the distance transform path planning methodology is ideal for supporting an outdoor navigation project, its advantages being that it can model various types of terrain in terms of navigability costs and allow a variety of path planning modes (point-to-point, search all space, etc).

The simple concept and previous applications of the distance transform are strong indications of its suitability for this application. However, there is little or no reference in the literature for its use for planning in three dimensions. In fact, extending the two dimensional method to three dimensions is straightforward in principle but a simplistic extension would result in a dramatic increase in computer memory requirements and program run time. For our application rapid path planning is essential so calculation of the distance transform and subsequent derivation of a path must be implemented with care.

The method depends, of course, on a 3D map of the environment being available. As explained by Williams *et al.*,[2] several different methods for building a workspace were considered. Although active systems such as millimetre wave radar have the advantage of producing range information, cost and off-the-shelf availability predominated in the decision to use a vision system as the primary sensor. Motion stereo and optical flow techniques are used to update a map of the environment.

2.3 *Octrees and quadtrees*

For efficient computation and use of memory a spatial decomposition of the environment map into a searchable representation is vital. Several decomposition schemes exist whose properties are discussed by Samet.[10] Here, use is made of octrees – an *octree*[4] is an ordered tree composed of labelled nodes having eight child nodes each. Construction of the octree is performed recursively on workspace sizes of $2^n \times 2^n \times 2^n$ ($n = 2, 3, 4 \ldots$), allowing integer division for improved computational speed. The structure of the resultant tree depends on the pattern of occupancy of the workspace. For the sparsely populated workspace anticipated in this application, the tree structure would be relatively small, resulting in a substantial reduction in memory requirement compared to the matrix representation. Figure 3 shows the node labelling convention for a 6-connected octree (node 7 is occluded). Connections between neighboring nodes are limited to the *city-block* (or '6-connected') method so that each internal node has a neighbor on each face of the cuboid cell. An alternative connection method is the *chessboard* method which allows a cell to have 26 neighbours. This can, however, produce diagonal paths that clip obstacles so, because this applica-

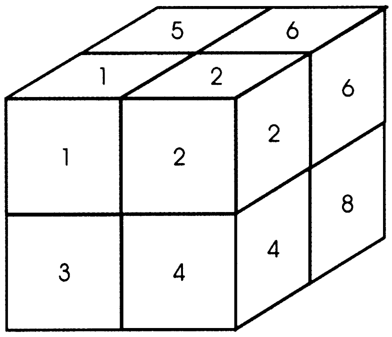

Fig. 3. Octree node labelling convention.

tion requires a conservative approach, the city-block method is used despite the fact that it can yield non-optimal paths.[11]

Continuing with the explanation of the principle by means of the 2D example, the workspace of Figure 2(a) is converted into a *quadtree* (which has only 4 child nodes per branch) by examining whether successive sub-divisions are all freespace ('white'), all occupied ('black') or contain both types of cell ('grey'). One of these three values is assigned to each node during the decomposition procedure until the final resolution level is reached. The recursive algorithm starts with the current quadrant set to the whole workspace:

```
Function generate tree
{
    Scan the current quadrant
    If quadrant contains both free and obstacle cells
    {
        Mark node as 'grey'
        Divide quadrant into sub-quadrants
        Label each new sub-quadrant as a child node
        For each sub-quadrant
        {
            Call generate tree
        }
    }
    Else
    {
        If quadrant is all free
        {
            Mark node as 'white'
        }
        Else
        {
            Mark node as 'black'
        }
    }
    Return
}
```

Note that, if the root node is marked 'white' then no path planning is required and if it is marked 'black' then no path is possible.

Figure 4 shows the first sub-division. Each node is labelled with a location code that gives its position in the tree and its position with respect to its siblings. This produces the four children of the quadtree's root node, as shown in Figure 7. Figure 5 shows the workspace after the second decomposition. The number of digits in the label (N) gives its depth within the tree (the root node is at depth zero) where the first (N-1) digits are the parent node's label. For example, node 431 in Figure 6 is a depth three node, whose parent is node 43 and grandparent is node 4. Figure 6 shows the workspace after decomposing down to cell resolution and Figure 7 shows the complete quadtree for this example.

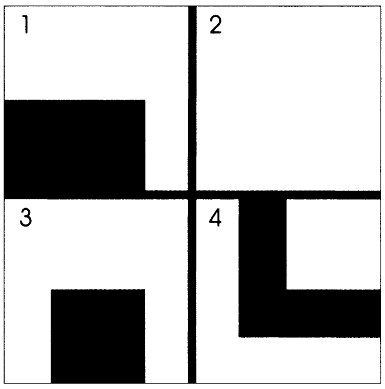

Fig. 4. First decomposition of workspace for quadtree representation.

Quadtrees and octrees convert a matrix structure into a memory efficient tree structure but the key to rapid path computation is to create a graph consisting of only freespace nodes whose adjacency is denoted by the graph's edges. This is examined in detail in Section 3.

2.4 Memory requirements

The Quadtree Complexity Theorem[12] states that, except for pathological cases, the number of nodes in the quadtree representation of a region is proportional to the perimeter of the region. Given a square (or cube for three-dimensional data) region of side length $\boldsymbol{l}$ units, the matrix representation requires $\boldsymbol{l}^2$ elements whereas the number of nodes in a quadtree representation is proportional to $\boldsymbol{l}$. The Quadtree Complexity Theorem also holds for 3-dimensional data[13] where the perimeter is replaced by the surface enclosing the workspace volume, so for a cube the number of nodes is proportional to $\boldsymbol{6\,l}^2$.

A three dimensional matrix representation stores the information for the distance transform in a multidimensional matrix of integers (two bytes per matrix cell). The data structure shown in Figure 8 requires 50 bytes of memory per node. So comparing the memory requirements for the two representations gives:

$$\text{Memory}_{\text{matrix}} = 2.\boldsymbol{l}^3 \text{ bytes}$$

$$\text{Memory}_{\text{octree}} = 50.6.\boldsymbol{l}^2 \text{ bytes}$$

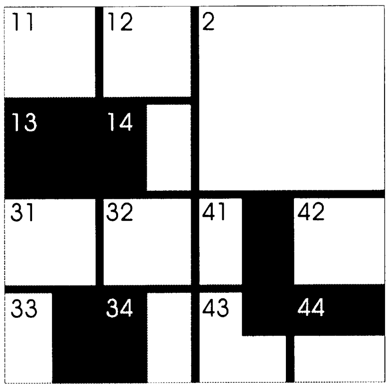

Fig. 5. Next level of decomposition of workspace.

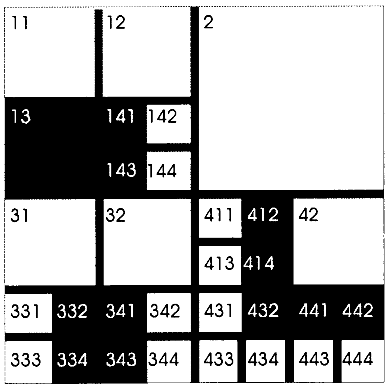

Fig. 6. Final decomposition at cell level of resolution.

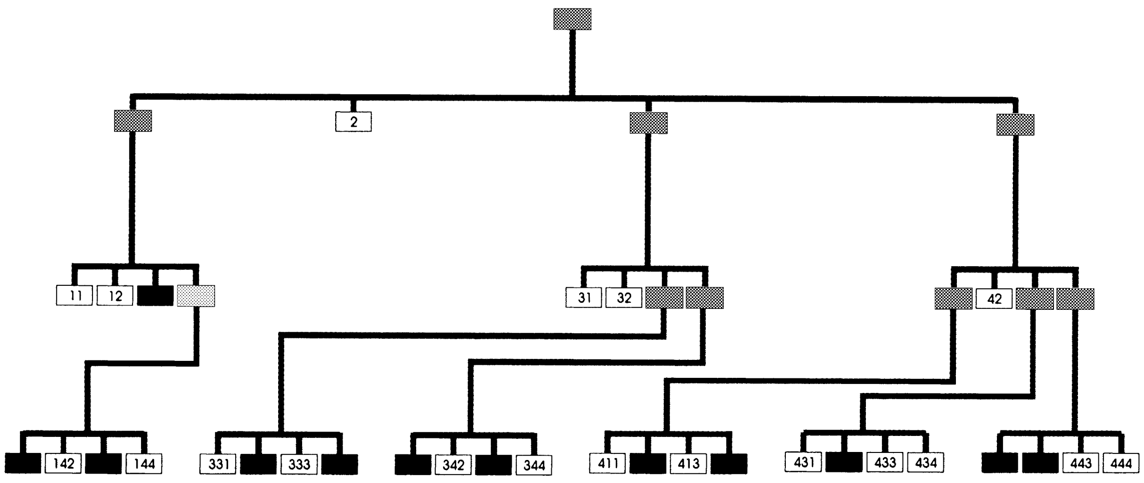

Fig. 7. Complete quadtree for the example workspace.

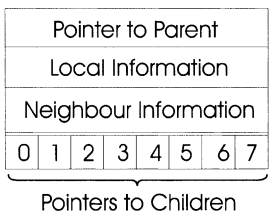

Fig. 8. Node data structure for a connectivity graph derived from an octree.

Figure 9 shows the difference in memory requirement as the size of the workspace increases. The solid line shows the amount of memory required to store a matrix representation of the workspace and the dashed line shows the memory requirement to store the same amount of data in an octree representation. For small size workspaces, the matrix representation is more efficient because of the lower overhead associated with each node but, at perimeter sizes of over 150 cells, the octree representation is superior.

The execution time of most algorithms that operate on a quadtree representation is proportional to the number of nodes in the tree. The Quadtree Complexity Theorem states that, in general, the execution time of a quadtree-based algorithm on a *d*-dimension problem is similar to a matrix-based algorithm on a (d-1)-dimension problem. Thus, octrees and quadtrees act as dimension-reducing devices.

3. CONNECTIVITY GRAPH CONSTRUCTION

The octree must be transformed into an undirected *connectivity graph* containing information about the adjacency of freespace nodes before the distance transform can be applied. Establishing the connectivity requires three stages: initial neighbour assignments followed by two further node-linking passes through the structure. The 2D example will again be used to demonstrate the principles involved.

3.1 Stage 1: Initial neighbour assignments

Construction of the connectivity graph starts by forming links only between nodes that share a common parent. The children of every node in the quadtree (Figure 7) are examined in turn and a link made between all non-black nodes whose corresponding cells (see Figures 4 to 6) share a common boundary. The result is a number of intra-child-group connections as shown in Figure 10. Valid connections are stored as a pointer-driven list of Neighbour Information within each node's data structure (see Figure 8); links to black nodes never appear in the list.

3.2 Stage 2: Update connections

Two further types of connection are established at Stage 2. First, connections are made between the white *children* of a grey node and any neighbouring white node. An example of this in Figure 5 is the connection of cell 12 (a child of grey node 1) to white node 2. Secondly, all valid connections between nodes that lie at the same depth but do *not* share a common parent are added to the Neighbour Information list. An example of this in Figure 6 is the connection of cell 342 (a child of grey node 34) with cell 431 (a child of its neighbour grey node 43).

Traversal of the tree is accomplished in depth-first fashion by the recursive function *diver*, beginning at the root node, and applying the connection *update1* when the conditions stated above are encountered.

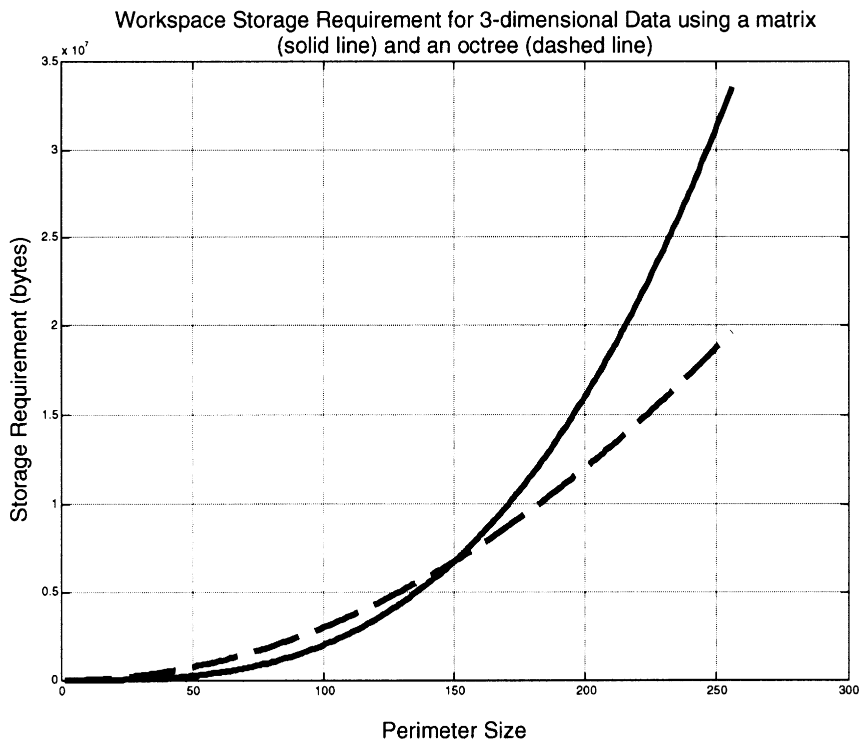

Fig. 9. Relationship between computer memory requirement and workspace size for matrix and octree representations.

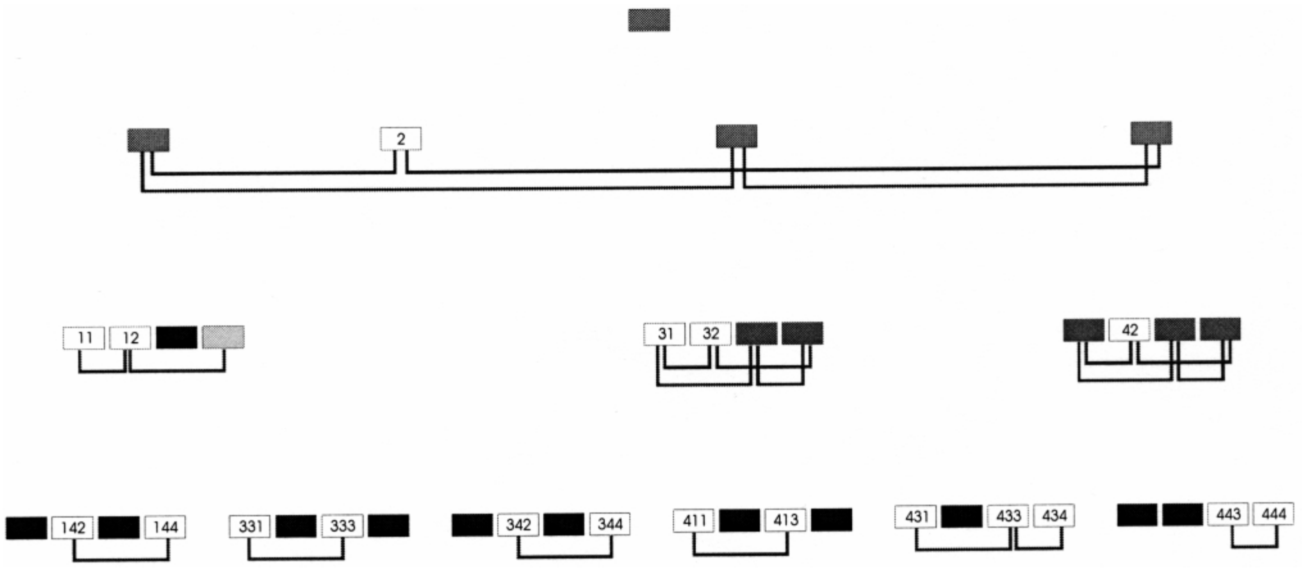

Fig. 10. First stage of building the connectivity graph showing intra-child-group connections.

```
Function diver(current_node)
{
        For n = 1:8
        {
                temp_node = nth child of the current node
                If temp_node is a grey node
                {
                        diver(temp_node)
                }
                update1(temp_node)        /* call to the
                        connection function */
        }
        Return
}
```

The *update1* function is called repeatedly during Stage 2 and the key to fast execution is the node labelling scheme which selects *only* those pairs of nodes which correspond to spatially adjacent cells to be tested for a freespace-to-freespace connection. The logical rules are described in Appendix A. The tables IA and IB give the linking rules based on the final digits of the current node (C) and the neighbour node (N). For example if the current node is 34 (X = 34) and the neighbour node is 43 (Y = 43) then C = 4 and N = 3. Using table 1B, X2 (child 2 of the current node, in this case 342) would be linked to Y1 (child 1 of the neighbour node, in this case 431) and X4 (344) would be linked to Y3 (433).

Table I. Truth tables used to determine adjacency relationships in the connection function *update1*.

Truth Table 1A

C \ N	1	2	3	4
1		X2 – Y X4 – Y	X3 – Y X4 – Y	
2	X1 – Y X3 – Y			X2 – Y X4 – Y
3	X1 – Y X2 – Y			X2 – Y X4 – Y
4		X1 – Y X2 – Y	X1 – Y X3 – Y	

Truth Table 1B

C \ N	1	2	3	4
1		X2 – Y1 X4 – Y3	X3 – Y1 X4 – Y2	
2	X1 – Y2 X3 – Y4			X2 – Y1 X4 – Y2
3	X1 – Y3 X2 – Y4			X2 – Y1 X4 – Y3
4		X1 – Y3 X2 – Y4	X1 – Y2 X3 – Y4	

Blank spaces indicate that no connections are made

```
Function update1(current_node)
{
    X = current node's label
    C = final digit of X
    For Each node in the current node's Neighbour
    Information List
    {
        Y = neighbour node's label
        N = final digit of Y
        If current node is grey AND neighbour node is
        white
        {
            Apply truth-table IA to link white or grey
            children of the current node to the neigh-
            bour node.
        }
        Else If both current node AND neighbour node
        are grey
        {
            Apply truth-table 1B to link the white or
            grey children of the current node to the
            white or grey children of the neighbour
            node.
        }
        Else            /* current node is white
                        AND neighbour node is
                        grey OR both current node
                        AND neighbour node are
                        white */
        {
            Do nothing  /* the first case is dealt with
                        in update2; the second case
                        does not occur because such
                        links will already have been
                        made */
        }
    }
    Return
}
```

The dotted lines in Figure 11 show the connections for the example workspace after applying the *update1* algorithm.

3.3 Stage 3: Update connections

Stage 3 connects nodes at different depths in the tree. The links that were made during the two previous stages are retained and the function *diver* is applied once more to traverse the tree but with the *update1* function replaced by the *update2* function, which again relies on the node labelling scheme for fast execution.

```
Function update2(current_node)
{
    X = current node's label
    C = final digit of X
    P = Parent node of X
    If current node has children
    {
        Switch (C)
        {
            Case 1:
                X2 linked P2; X3 linked P3; X4
                linked P2 and P3
            Case 2:
                X1 linked P1; X3 linked P1 and P4;
                X4 linked P4
            Case 3:
                X1 linked P1; X2 linked P1 and P4;
                X4 linked P4
            Case 4:
                X1 linked P2 and P3; X2 linked P2;
                X3 linked P3
        }
    }
    Return
}
```

The dashed lines in Figure 11 show the Stage 3 connections for the example workspace. Note that Figure 11 now contains only the links between freespace nodes which are

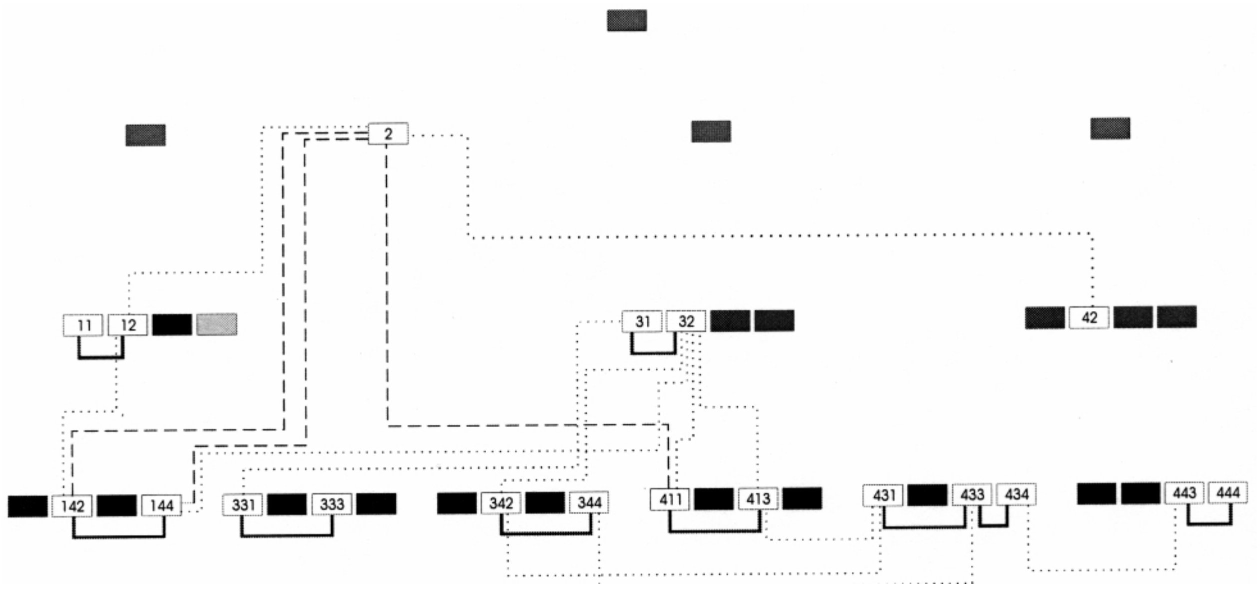

Fig. 11. Final connectivity graph after stages 2 and 3; note that links exist only between white nodes.

needed for path planning; the others are redundant once the connectivity graph is complete and are omitted. Whilst it would be possible to search this graph directly for a path, a much more efficient guided search can be performed using distance transform annotation.

4. THE DISTANCE TRANSFORM AND PATH PLANNING

4.1 *Distance transform*

Applying the distance transform (DT) to the connectivity graph is straightforward. The algorithm to generate the distance transform uses queues as data structures as follows:

```
Function distance transform
{
    Create empty queue
    Add goal node to queue
    Mark the goal node with a DT value of zero
    While the queue is occupied
    {
        Remove head of queue and make it the current node
        Copy all unlabelled neighbours of the current node onto the tail of the queue
        Mark all unlabelled neighbours of the current node with a DT value of one greater than the current node
    }
    Return
}
```

In this way, every node that can be reached from the goal node will be labelled with a DT value and those that cannot will remain unmarked. In particular, if the start node is unmarked, the graph is not connected and the start node is not reachable from the goal so it is immediately known that no feasible path exists (Figure 12).

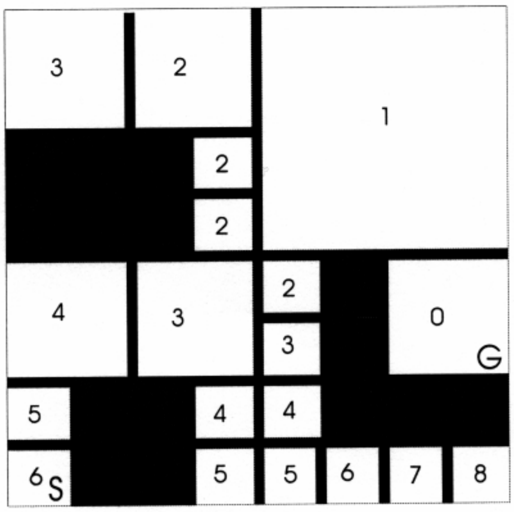

Fig. 12. Distance transform applied to the sub-divided work-space.

4.2 *Path planning*

The path planner first examines the start node's neighbours to find the one with the lowest DT value, which is made the current node. In turn, its neighbours are examined for the lowest DT value and this process continues until the goal node is reached. If two nodes are found with the same distance transform value, then the first one found is used. Whilst it would be possible to resolve the selection by weighting the distance transform with another metric (for example, the area covered by the node), this could lead to production of a false minimum and possible failure of the planner.

Once a list of nodes has been produced, path planning is complete and it is necessary to generate the motion plan which controls the vehicle's motion through physical space. Optimal methods of utilising the distance transform information have been described, such as that proposed by Chen *et al.*[14] who introduced a framed-quadtree structure which combines the accuracy of grid-based planning techniques with the efficiency of quadtree techniques. The method used here is simply to link the centres of the volumes represented by the nodes by straight line segments and to interpolate the corresponding world coordinates that the vehicle should pass through. In a dynamic environment, path planning is a continual process. One advantage of the octree representation is that the whole structure does not need to be re-built every time. If the apparent movement of the obstacle is confined to one sub-tree then only that sub-tree needs to be re-built. Zelinsky[7] uses a distance transform partial-update algorithm to achieve this.

4.3 *Implementation details*

The current implementation of the algorithm uses the Borland C++ 5.01 compiler running under Microsoft Windows 95. This environment was chosen as it fits in with our overall project, of which the path planner is only one function. The algorithm runs on a Pentium 166 MHz with 64 MB of memory. The Borland compiler includes support for templates or parameterised types. Templates are container data structures. For example, an array is a container that can be used to store collections of data of different types. The Borland environment includes other templates such as sets, queues and vectors which have proved to be very useful in the development of the code as they provide a standard method of handling complex data types such as the queue structure described in Section 4.1.

5. SIMULATION RESULTS

In this section, simulation results are presented which show how the path planning algorithm copes with workspaces of increasing complexity. Testing was performed on hand-made synthetic data representing different 3D workspaces stored in matrix format. The test set ranges from a completely empty workspace to a relatively dense work-space containing several obstacles and includes a case where a path is not possible. The test workspaces are $32 \times 32 \times 32$ (32768) cells in size. Because the simulation was performed under a multitasking operating system, it was necessary to run each case about 15 times and to record

Table II. Measured execution times for the five examples.

	Test 1	Test 2	Test 3	Test 4	Test 5
Execution time	30 ms	50 ms	60 ms	50 ms	100 ms

the minimum run time so that errors due to operating system overhead were minimised.

The first test is performed on a completely empty workspace in order to determine a reference processing time, as shown in Table II. An empty workspace requires neither sub-division nor path planning so the computation time is almost entirely due to scanning the workspace for obstacles.

The second test uses a simple workspace containing a few large obstacles as illustrated in Fig. 13, where the basic tree-building operations contribute substantially to the processing time. As shown in Table II, this has almost doubled.

The third test, as illustrated in Fig. 14, considers a workspace with a number of small obstacles. As shown in Table II, the processing time increases once more because the neighbourhood building operations for this type of environment are relatively complex.

The fourth test considers an environment in which no path is possible as shown in Fig. 15. This test demonstrates that the method can quickly identify when this is the case (see Table II).

The final test considers combinations of large and small obstacles producing a complex path, as shown in Fig. 16. Table II shows that this has the highest computation time, but nevertheless it is only just over three times the "empty workspace" case.

Overall, the results show that the proposed method can deal with all types of workspace. It is essential for this application that the distance transform algorithm always shows when no path is possible. Other path planning algorithms, such as the basic potential field method, can spend long periods in an iterative search for a non-existent path. Together, workspace decomposition, neighbourhood generation and the distance transform provide a method for path planning which is at least no worse than other methods; for the type of environment applicable to a RPH it is usually significantly better.

6. CONCLUSIONS

This paper has described a method for path planning in three dimensions which is both rapid and safe and well suited to its intended application of vision based collision avoidance for a small RPH. The importance of careful implementation has been emphasised. The examples show that, using octree representation and a distance transform metric, the pointer-based connection algorithms developed here allow paths to be planned in complex workspaces well within the limits of memory and computational power of an inexpensive processor.

The planning method used is a simple descent of the distance transform from the Start to the Goal. It is possible to include a measure of workspace quality within the distance transform in order to weight certain areas so that they are avoided. In ground based applications this weighting can be thought of as a traversability factor that penalises areas of unsuitable ground – such as uneven, boulder-strewn ground – in favour of clear areas of flat ground. An analogous situation for a UAV could be weighting an area that is dangerous to fly into such as near airfields or in areas where there may be large downdrafts such as valleys. Whilst this could produce safer paths, the additional weighting term could produce false minima in the modified distance transform values so introducing any preferential path factor must be set against the possibility of algorithm failure.

The method has been used to demonstrate path planning on a laboratory test rig[2] where the internal map containing

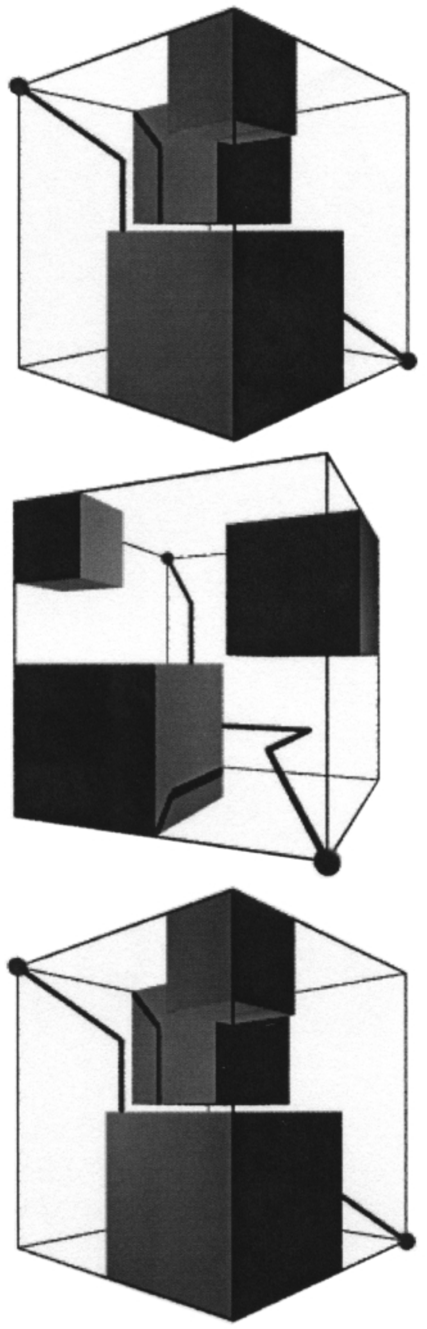
Fig. 13. TEST 2: Three views of a simple workspace showing the path from start (top) to goal (bottom).

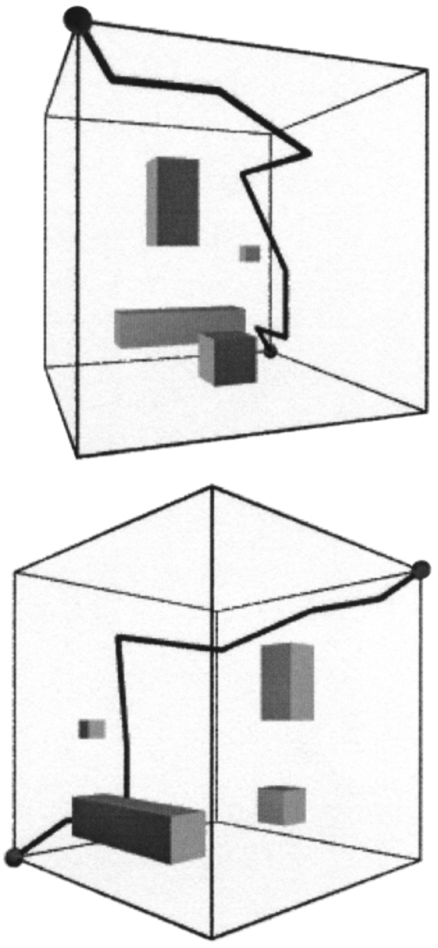

Fig. 14. TEST 3: A workspace containing small, well-spaced obstacles, viewed from two different angles. Note that the path does not go directly from Start to Goal despite the relatively sparse obstacles.

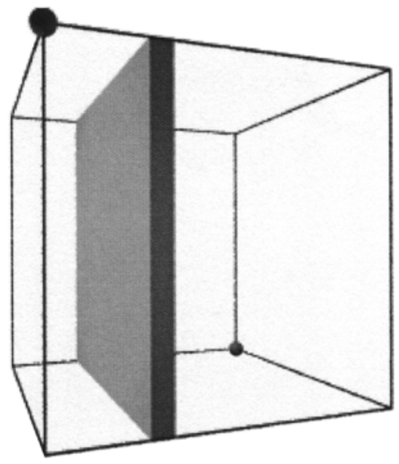

Fig. 15. TEST 4: Workspace where no path is possible.

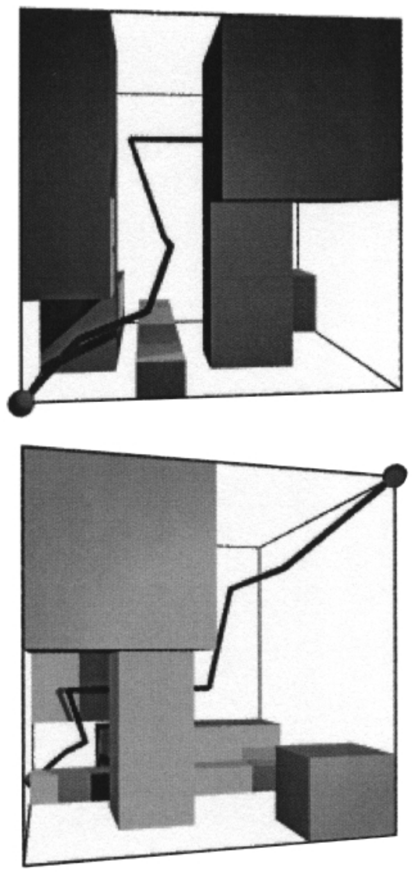

Fig. 16. TEST 5: Complex environment with a combination of large and small obstacles, viewed from two different angles.

information about the obstacle locations is updated by means of a machine vision system. The major difficulty is obtaining estimates of the distance of objects from the vehicle. The optical flow methods currently under investigation for this purpose are far more computationally intensive and less robust than the path planner and remain the major hurdle to be overcome.

Acknowledgements

The authors wish to thank EA Technology Ltd and the Engineering & Physical Sciences Research Council for supporting this work. Particular thanks are due to Mr G.K. Earp for his advice and encouragement.

References

1. D.I. Jones and G.K. Earp, "Requirements for aerial inspection of overhead electrical power lines", *Proceedings of the 12th International Conference on Remotely Piloted Vehicles*, Bristol, England (1996) Paper 4.
2. M. Williams, D.I. Jones & G.K. Earp, "Obstacle avoidance during aerial inspection of power lines", *Proceedings 2nd International Workshop on European Scientific and Industrial Collaboration*, Newport, Wales (1999) pp. 61–68.
3. B. Bhanu, S. Das, B. Roberts and D. Duncan, "A system for obstacle detection during rotorcraft low altitude flights", *IEEE Transactions on Aerospace and Electronic Systems* **32**(3), 875–897 (1996).
4. J.C. Latombe, *Robot Motion Planning* (Kluwer Academic Publishers, 1991).

5. R.A. Jarvis and J.C. Byrne, "Robot navigation: Touching, seeing and knowing", *Proceedings of the 1st Australian Conference on Artificial Intelligence* (1986).
6. D. Shin, "A fast motion planning algorithm for a mobile robot using a distance transformation image", *Systems and Computers in Japan* **25**(5), 88–99 (1994).
7. A. Zelinsky, "A mobile robot exploration algorithm", *IEEE Transactions on Robotics and Automation* **8**(6), 707–717 (1992).
8. R.A. Jarvis, "An all-terrain intelligent autonomous vehicle with sensor-fusion-based navigation capabilities", *Control Engineering Practice* **4**(4), 481–486 (1996).
9. S. Hert, S. Tiwari and V. Lumeksly, "A terrain-covering algorithm for an AUV", *Autonomous Robots* **3**, 91–119 (1996).
10. H. Samet, "An overview of quadtrees, octrees, and related hierarchical data structures". In: *Theoretical Foundations of Computer Graphics and CAD* (R.A. Earnshaw, Ed.) (Springer-Verlag, 1988) **vol. F40** of *NATO ASI*, pp. 51–68.
11. A. Klinger, "Patterns and Search Statistics". In: *Optimising Methods in Statistics*, Academic Press (Ed. J.S. Rustagi, 1971) (Academic Press, 1971) pp. 303–337.
12. G.M. Hunter and K. Steiglitz, "Operations on Images Using Quad Trees", *IEEE Transactions on Pattern Analysis and Machine Intelligence* **1**(2), 145–153 (1979).
13. D. Meagher, "Octree encoding: a new technique for the representation, the manipulation, and display of arbitrary 3-d objects by computer", *Technical Report IPL-TR-80–111* (Image Processing Laboratory, Rensselaer Polytechnic Institute, Troy, New York, 1980).
14. D.Z. Chen, R.J. Szczerba and J.J. Uhran, "A framed-quadtree approach for determining Euclidean shortest paths on a 2-D environment", *IEEE Transactions on Robotics and Automation* **13**(5), 668–681 (1997).

APPENDIX A. ADJACENCY RELATIONSHIPS

The adjacency relationships used by the connection function *update1* are described for the quadtree case – the rules for an octree are no different in principle but are considerably longer. Consider Figure A1 which shows the decomposition of four cells (1, 2, 3 and 4) into 16 sub-cells, whose corresponding nodes lie at the same depth in the quadtree.

Evidently there are only 16 permutations of sub-cells (12–21, 21–12, 14–23, 23–14 . . . etc.) which share a common boundary with sub-cells of another parent, i.e. they are *adjacent* sub-cells. The node labelling scheme ensures that *update1* applies the tests for a freespace-freespace connection to adjacent cells *only.* In fact, as shown in truth tables 1A and 1B, any invocation of *update1* applies two tests at most. The truth tables are generic and applicable at any depth within the tree.

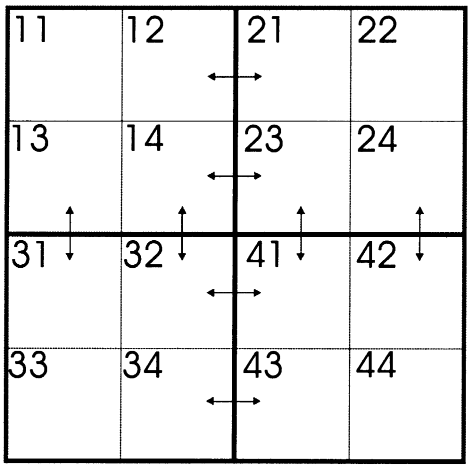

Fig. A1. Illustration of adjacency relationships.