A Dynamic-Programming Heuristic for Regular Grid-Graph Partitioning

W.W. Donaldson † and R.R. Meyer †

† Department of Computer Sciences, University of Wisconsin - Madison, WI

November 15, 2000

Abstract

Previous researchers have demonstrated that striping heuristics produce very good (and, in some cases, asymptotically optimal) partitions for regular grid graphs. These earlier methods differed in the domains of application and the stripe-height selection process, and did not have polynomial run-time guarantees.

In this paper, we transform the stripe selection problem for general grid graphs into a shortest path problem. The running time for the entire process of transforming the problem and solving for the shortest path is polynomial with respect to the length of input for the original problem. Computational results are presented that demonstrate improved solution quality for general domains.

1 Introduction

1.1 Statement of Problem

Given a graph G = (V,E) and a number of components P, the graph partitioning problem (with uniform node and edge weights) requires dividing the vertices into P groups of specified size such that the number of edges connecting vertices in different groups (cut edges) is minimized. (Typically, these problem constraints are motivated by load-balancing considerations that require the P groups to be equal or nearly-equal-sized.) This problem is known to be NP-Complete [7]. In this paper, a restricted class of graphs is studied, namely regular grid graphs. Figure 1 shows an example of a regular grid graph. The vertices lie at lattice points of a rectangular grid and are connected only to points adjacent on the lattice. (Note that the overall domain need not be rectangular; it may, for example, be a grid approximation to a torus.) Graph partitioning of large regular grid graphs arises in the context of minimizing interprocessor communication subject to load balancing in parallel computation for a variety of problem classes including the solution of PDEs using finite difference schemes [13], computer vision [12], and database applications [8]. Given this application context, we can think of the problem as that of optimal assignment of tasks represented by the nodes or, in the transformed problem below, assignment of cells to processors.

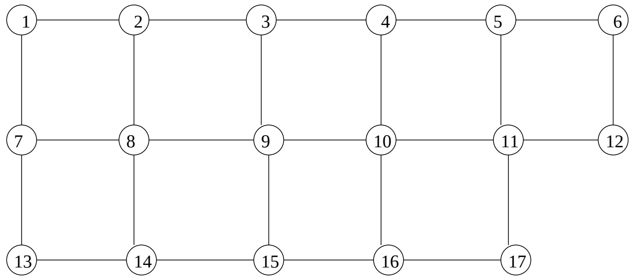


Figure 1: A grid graph

1.2 Transformation to Cell Domain Format

Christou-Meyer [3] consider another way of formulating this problem, which is useful in terms of generating the lower bounds on the optimal value considered below. Figure 2 gives an example of how the original graph is transformed into a domain of square cells. Each node is mapped into a unit square and each edge is mapped into an edge of the square. Additional "boundary" edges are added as needed to complete squares. (In applications, it is assumed that geometrically adjacent nodes are connected by an edge; we assume this property below.)

For the transformed problem, instead of counting cut edges, the sum of the perimeters for the associated components is minimized. Formally, the problem to be solved is:

minimize $\sum_i \ \mathbf{Per}(C_i) \quad \mathrm{i} = 1 \ldots \mathrm{P}$

s.t.

each cell is assigned to one component, and

each component is assigned a given number of cells,

where $\mathbf{Per}(C_i)$ equals the perimeter for component C_i. (Although in the graph literature components are normally considered to be connected, we do not assume this connectedness property in this paper.) In most of the discussion below, we assume that P divides the total number of cells,

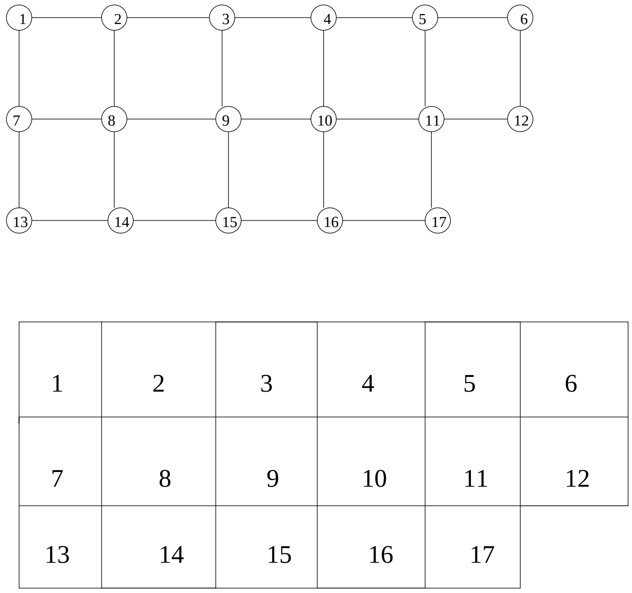


Figure 2: The original graph and the corresponding cell domain.

and this ratio is the given “area” for each component. (This reflects the desired load balancing in parallel computing applications.)

The relationship between cut edges in a partition of the graph and the total perimeter of the corresponding partition of the cell domain is:

$$\text{cut edges} = (\text{total perimeter - perimeter of the boundary of the domain})/2$$

This follows from the observations that the domain boundary edges are always “perimeter” edges but do not correspond to graph edges; furthermore, each cut edge contributes two to the total perimeter. Thus minimizing perimeter is equivalent to minimizing cut edges.

Figure 3 shows an example of this relationship. In this case, P is 6 and the specified component sizes are 3,3,3,3,2,3. (In this example, P does not divide the number of cells, so not every component is the same size; we consider below extensions to our basic approach that allow this generalization.) The number of cut edges in the graph equals 14. The sum of the perimeters for the components in the cell domain is 46. The perimeter for the boundary of the cell domain is 18, so the number of cut edges is (46-18)/2. The bottom partition is also an example of a stripe-form solution (to be defined formally below; informally this means that components are confined to horizontal bands except possibly for “overflows” at the ends of the bands, which do not occur in this instance). This partition is also an optimal solution since it is easily shown to yield the lower bound available from [14].

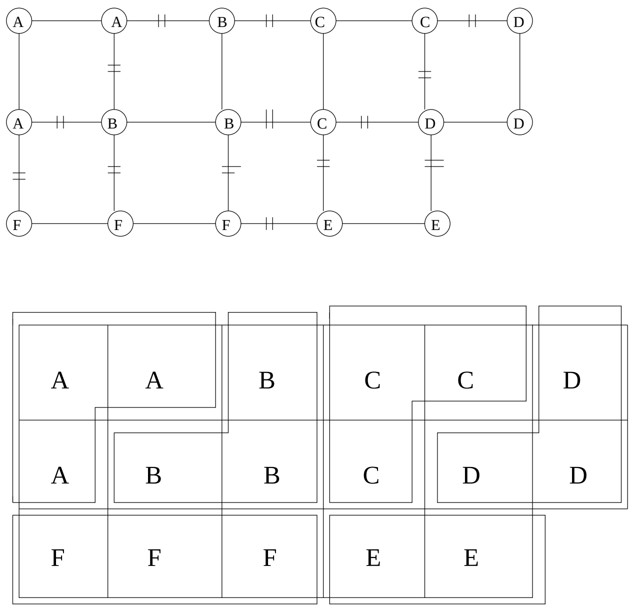


Figure 3: The top partition of the figure shows the cut edges of a partition of the original graph. The bottom figure shows the corresponding cell-domain partition.

1.3 Organization of the Paper

We develop below a heuristic for partitioning regular grid graphs. This heuristic is based on the construction of an optimal set of stripe heights for stripe-form solutions. Both theory and computation have established that high quality (and, in some cases, asymptotically optimal) solutions can be constructed via striping.

Topics are developed in the following order:

- Definitions
- A method for partitioning the problem into subproblems
- Showing that the number of stripe-based solutions is generally a super-polynomial function of the problem dimension, hence a brute force method is not feasible
- Transforming the problem into a shortest-path problem
- Numerical results
- Concluding remarks and directions for further research

2 Terms and Definitions

The following example, figure 4, is presented in order to demonstrate concepts and definitions that will appear in the paper.

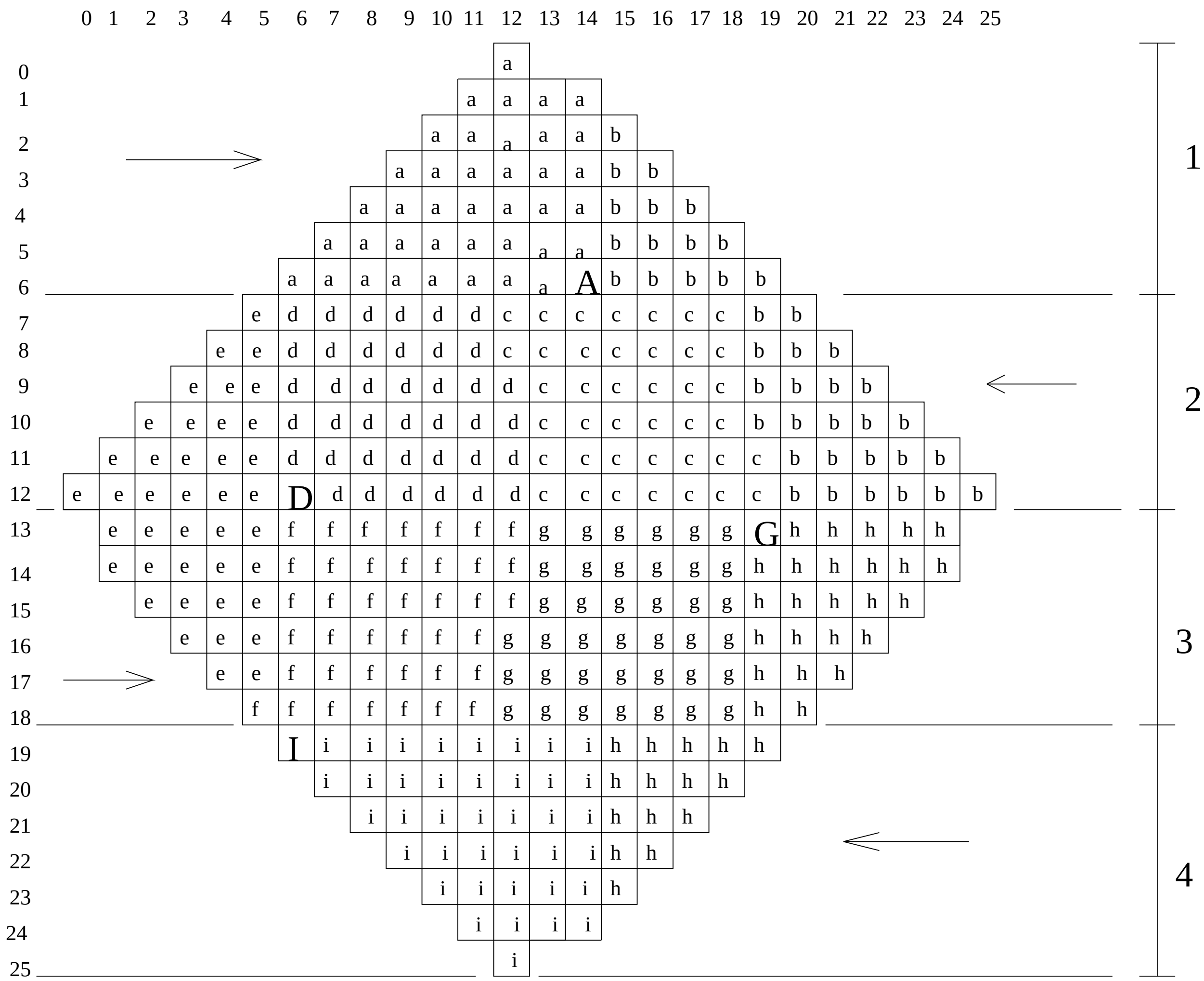


Figure 4: An assignment of a grid graph

In the grid, there are 360 cells, which are to be divided equally among 9 processors (in the feasible solution shown, the corresponding components are labelled a-i). (If different areas (number of cells) are specified for the components (as in figure 3) we assume that the assignment procedure completes the assignment of components sequentially in the order in which sizes are specified. As discussed below, it is possible to modify the algorithm to take advantage of different component sizes.) To obtain the feasible solution shown, the 26 rows of the grid (numbered 0 to 25) were first partitioned into the four stripes indicated in the figure. (As will be shown below, there are better selections of stripe heights, but this feasible solution serves to illustrate in a simple manner some basic concepts.) For illustrative purposes, a simple "fill" procedure, for assigning cells to components, was then used that allowed at most one component to "overflow" from one stripe into the next stripe. Cells are assigned column-wise, top to bottom within stripe i , and then the assignment procedure is continued (still column-wise, but going in the other horizontal direction) in stripe *i+1*. This process was termed "snake fill" in [3], because of the manner in which it snakes through the domain. In contrast to the simple example in figure 3, stripe heights must be chosen carefully in this instance in order to obtain a good solution. The total perimeter of this partition is 294.

Definitions - A row in a grid is said to be an **end-row** of a stripe if it is the last row in the stripe (in figure 4, rows 6, 12, 18, and 25 are end-rows). A **begin-row** is defined similarly.

A **begin-end (row) pair** defines a **stripe** as the collection of cells with corresponding row indices (in figure 4, stripe 1 is defined by the begin-end-row pair (0,6)).

An **optimal stripe set** (with respect to a given fill procedure) is a sequence of stripes that yields minimum total perimeter relative to other feasible sets of stripes, assuming the given fill procedure is utilized for each stripe set. (Implicit in this definition is the property the combination of a fill procedure with a set of stripe heights uniquely determines a feasible partition. This property will be discussed further below in conjunction with restrictions on the fill procedures that we consider.)

In section 8, figure 13 shows a four-stripe partition that is optimal (with respect to a more complex fill procedure) for the grid in figure 4. This latter solution has perimeter 282.

Definitions - A subset of the grid is said to be **filled** if all of its cells have been assigned.

Within a stripe, a **divider** cell is the last cell assigned by the fill procedure to complete a particular "last" component associated with that stripe. (In 4, the uppercase A, D, G, and I are divider cells. For this example, the fill procedure is such that the divider cells correspond to the last components that can fit entirely within each stripe. However, this need not be the case, and in the implementation below, we utilize more complex fill procedures in which the divider cells occur in other components.) The divider cell concept is a data structure that eliminates the need to keep track of the individual cells in the stripe "U-turn region" (defined below) in which components may cross stripe boundaries. (The methodological ideas developed below are easily extended to more general approaches in which the U-turn region is allowed to have an arbitrary configuration.)

A begin-end-row pair is **valid** if the corresponding stripe (termed a valid stripe) contains a divider cell. (Note that the validity of a begin-end pair is equivalent to having a sufficient number of cells added by the appending stripe to allow the completion of the assignment of at least one component. In the discussion below, (b_i, e_i) will designate a valid begin-end pair unless stated otherwise.) A set of end-rows (r_1, r_2, ..., r_k) is said to be a **feasible sequence of end-rows** if:

for i $\geq$ 0, (r_i+1, r_{i+1}) is a valid stripe (where r_0 = -1).

The corresponding sequence of begin-end-row pairs is ((0,r_1), (r_1+1,r_2),..., (r_{k-1}+1,r_k)) (note: the use of valid-stripe sequences implies the existence of a divider cell within each stripe).

We now define the key generic property that we assume for fill procedures. This property makes it possible to construct an optimal stripe set for such a fill procedure via a polynomial-time method.

Definition - A fill procedure is said to be **stripe quasi-independent (SQI)** if, for each stripe i implied by a feasible sequence of end-rows, the divider cell position for the stripe is uniquely determined by the given fill procedure and the begin-end pair for the stripe, and is independent of prior stripes in the sequence. Moreover, we assume that the fill procedure completes the assignment of the cells up to and including the divider cell in stripe i, before assigning any other cells in the domain. (See figure 7.) We will say that an assignment is of **stripe-form** if it may be obtained by applying an **SQI** fill procedure to a feasible sequence of end-rows. The **U-turn** region for stripe i consists of those cells within stripe i that "follow" the divider cell and hence are not assigned until stripe *i+1* is considered. (Given a stripe and a direction of assignment, we can number the cells in the stripe sequentially, column-wise, proceeding from top to bottom in each column. With such a numbering, the U-turn region (if non-null) will be comprised of the last k cells of the stripe for an appropriate k. In the simplest fill procedures, k will always be less than the size of a component. However, better results may be obtained by using more complex fill procedures that allow values of k larger than the component size. For example, note that a U-turn region consisting of a single "spike" column may lead to a component with poor perimeter when the component assignment is completed in the next stripe, since the total perimeter for the component will include a large contribution from the spike. This situation may be alleviated by having a larger U-turn region consisting of the spike column plus the cells of the previous component. In this case, the fill procedure may then handle the U-turn region by using a row-fill method within the U-turn region for the component that fits entirely within the U-turn region, in essence moving assignments from what would have been spike positions to cells in the row (or rows) immediately above the next stripe so that the next component may be spike-free and have a good perimeter.) Those cells required in stripe *i+1* to complete the assignments for components only partially assigned within the U-turn region of stripe i are called **overflow**.

In figure 4, the upper-stripe regions corresponding to the cells assigned to components b, e, and h are the U-turn regions.

We next consider definitions relevant to the "state graph" used in the shortest-path problem developed below. These definitions provide insight into the key ideas needed to construct optimal stripe sets.

Definition - A **vertex** in the state graph is uniquely defined by a begin row, an end row, and direction of assignment within that stripe (left-to-right or right-to-left). In the figures to follow, such a vertex is defined by (b_i, e_i, d_i). An ordered triple is a valid vertex label if and only if b_i and e_i correspond to a stripe determined within a feasible sequence of end rows. For a stripe-form solution, define **perim** **(b_{i-1},b_i,e_i,d_i)** to be the perimeter of the components falling within the region comprised of:

(1) the U-turn region of stripe *i-1* (if any), and

(2) the non-U-turn region of stripe i.

(In the construction of the state graph, perim(b_{i-1}, b_i, e_i, d_i) is the length assigned to the edge connecting (b_{i-1}, e_{i-1}, d_{i-1}) and (b_i, e_i, d_i) and represents the perimeter increment corresponding to appending stripe i. The **SQI** property guarantees that this length is determined solely by nodes (b_{i-1}, e_{i-1}, d_{i-1}) and (b_i, e_i, d_i) and is independent of the other nodes (stripes). Note also that the perimeter increment corresponds to the perimeter of at least one complete component and only involves perimeter contributions corresponding to complete components.)

3 An Overview of Striping Algorithms

Yackel-Meyer [15], Christou-Meyer [3], and Martin [11] showed that striping techniques produced very good (and, for some problem classes, asymptotically optimal) partitions when applied to regular grid graphs. Any algorithm based on striping can be broken into two main parts:

1) selecting stripe-heights, and

2) assigning the cells within a stripe ("fill procedures").

Yackel-Meyer and Christou-Meyer use genetic algorithms for generating stripe heights for a general class of grid graphs. Martin considers rectangular domains and transforms the original problem into a knapsack problem [9]. Both methods for selecting stripe heights have limitations.

In general, in the worst case Christou-Meyer requires at least super-polynomial (a constant greater than one whose exponent is raised to a fractional power) time for discovering an optimal solution. Martin's approach can only be applied to a restricted set of problems with rectangular domains and equal-sized components. (It is not known what the computational complexity is for Martin's method; but, since the problem is formulated as a knapsack, it is expected to be NP-Complete.) The Donaldson-Meyer method described below eliminates both of the barriers when selecting stripe heights.

Within-stripe assignment fill procedures have been addressed extensively. For rectangular grids, we have shown under certain conditions that the Christou-Meyer fill procedure [3] produces locally optimal solutions. We have also developed more complex assignment procedures that produce locally optimal solutions under more relaxed conditions. Proofs of these last two properties can be found in [5] and [6].

The basic idea of a stripe-based fill procedure is that components with nearly minimal perimeter can be obtained by applying column-wise assignment in the interior of stripes of appropriate height, and then using special procedures in the remainder of the stripe (the U-turn and overflow regions) to deal with components that cross stripe boundaries. (Stripes that are "too tall" or "too short" will result in components with large perimeters when such fill procedures are used.) This paper focuses on the development of methods that guarantee optimal choices of stripe heights assuming that an SQI fill procedure is utilized .

4 Defining Subproblems

4.1 Identifying Subproblems

In this section, a method for breaking up the original problem into similar but smaller subproblems will be presented. This is a the key step in the algorithms that will be presented later and corresponds to the construction of the nodes in the state graph.

Definition - Let (b_i, e_i) be a valid begin-end pair, let d_i be the direction of assignment within this stripe, and let c_i be the divider cell appearing within this stripe. We say that b_i, e_i, and d_i define the valid **subproblem** of determining an optimal partition of the domain whose "last" cell is c_i.

In figure 5, which illustrates a the domain of a subproblem, the definition of divider cell implies that an integral number of components can be assigned to the cells in the union of regions A and

B, where c_i denotes the divider cell in stripe i.

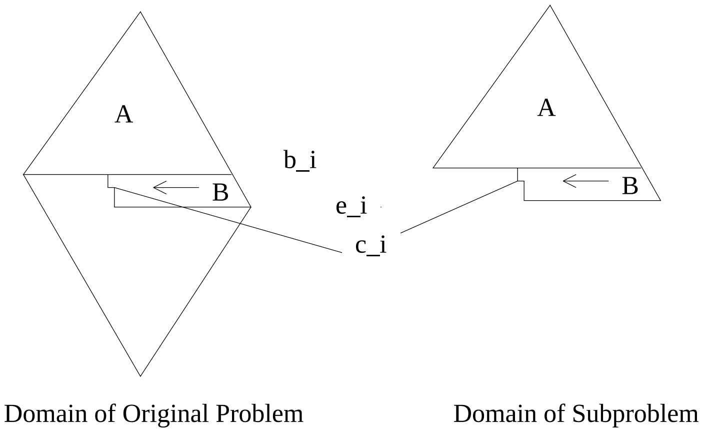


Figure 5: Subproblem example

(If the graph is symmetric, then only b_i and e_i are needed to uniquely define a subproblem; otherwise direction of assignment becomes a factor. We focus on the non-symmetric case in this paper.) In the non-symmetric case in figure 6, note that the assignments for the last stripes are in opposite directions. Although both subproblems in the figure correspond to the same begin-row end-row pair, the configurations (U and U') of remaining U-turn regions are different. This difference will generally lead to different perimeter increments when the next stripe is added to the subproblem, hence it is necessary to include direction information to identify a subproblem.

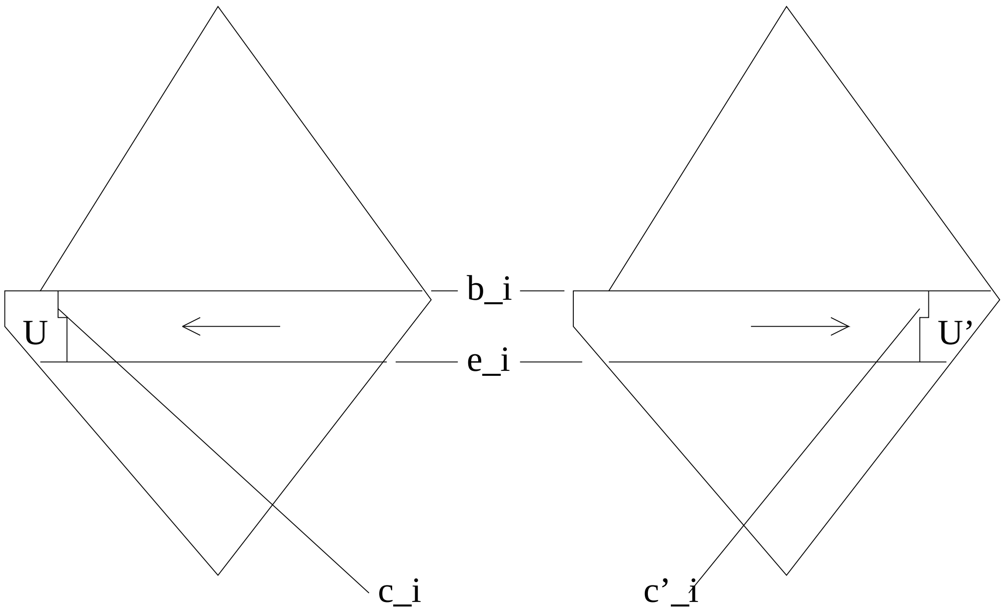


Figure 6: Direction of assignment affects the configuration of U-turn regions.

In the discussion to follow, we consider only stripes that contain a divider cell so that they correspond to valid subproblems.

4.2 Selecting a Divider Cell

In the arguments to follow, we assume an **SQI** fill procedure so that the choice of divider cell c_i for a given stripe (b_i,e_i,d_i) is independent of the height of previous stripes. Figure 7 indicates how a begin-row, end-row, and direction of assignment determine a domain that is independent of previous stripes and subproblems.

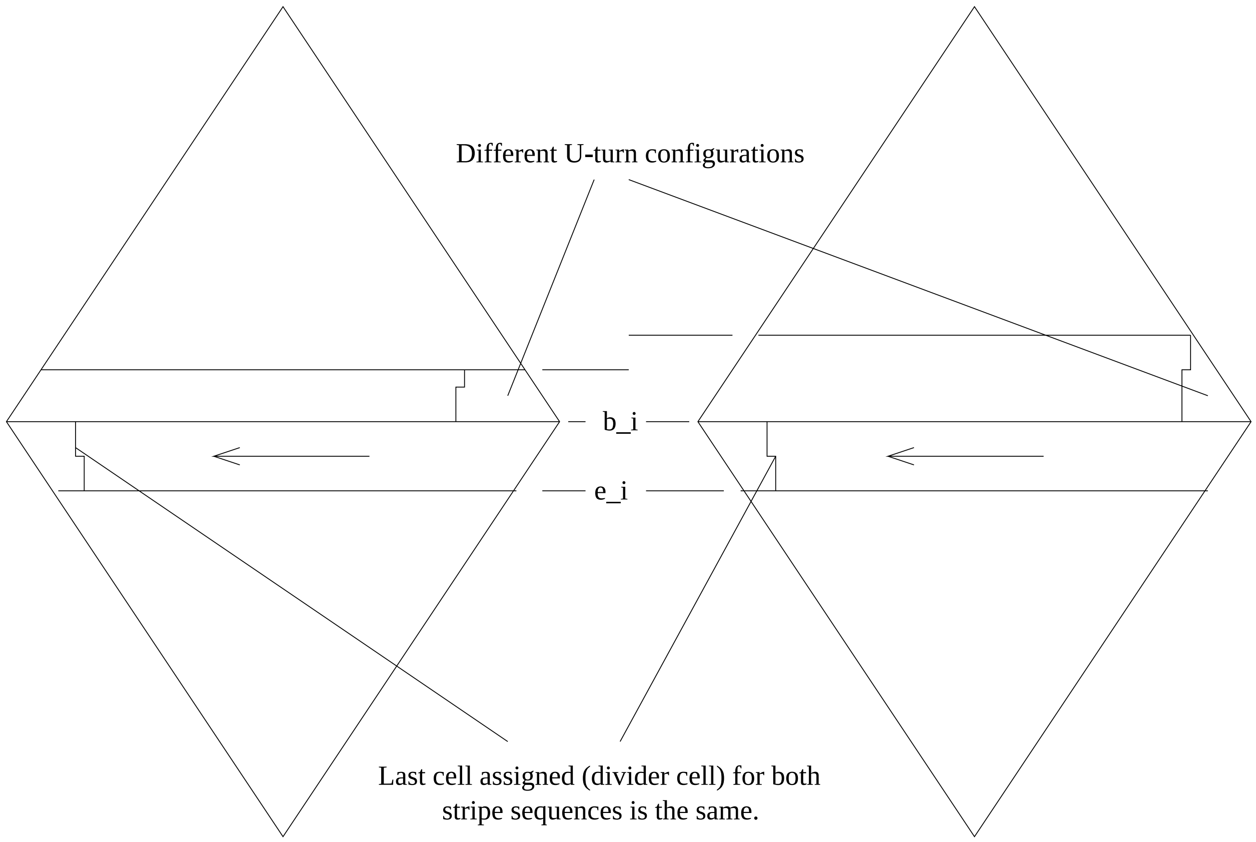


Figure 7: The final begin-row, end-row, and direction uniquely determine a domain that is independent of the preceding stripes.

This deterministic way of uniquely defining subproblems via the triple (b_i, e_i, d_i) allows the use of efficient dynamic-programming techniques for coordinating subproblem results and eliminates redundant calculations. This approach to defining subproblems also provides a useful way of identifying vertices and calculating edge weights in the corresponding shortest-path problem (see section 9.2,"Description of a State Graph").

Subproblems:				Perimeters:	
Sub. No.	b_i	e_i	d_i	Stripe Increment	Subproblem Perimeter
1	0	6	1	32	32
2	7	12	0	98	130
3	13	18	1	90	220
4	19	25	0	74	294

Table 1: a subset of the dynamic-programming data for figure 4

Consider the partitioned grid in figure 4. Examples of the two ways that a portion of the subproblem data can be presented are shown in table 1 and figure 8. Table 1 contains the perimeter data as organized by the dynamic-programming implementation. Figure 8 presents the same data in the form of a weighted graph, with vertices corresponding to subproblems and edge weights equal to the perimeters for the appending stripes. The total length of the path is the total perimeter of the corresponding partition, which is 294 (this is also the final subproblem perimeter shown in the far right-hand column in table 1). This is not an optimal partition. An optimal stripe-based partition, with perimeter equal to 282, is shown in figure 13. This latter partition would thus correspond to a larger DP table and another path from the unassigned node with different intermediate nodes and total length 282.

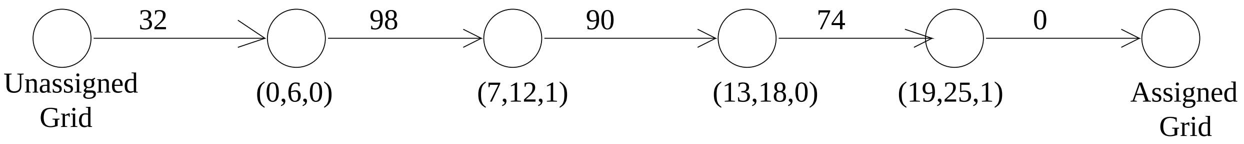


Figure 8: Path data corresponding table 1

5 Grid Partitioning as a Shortest-Path Problem

Constructing a feasible sequence of end-rows that partitions a grid is equivalent to determining a path from an origin (unassigned grid) to a destination (assigned grid) in a "state" graph (to be formally defined later). The graph in figure 8 shows a portion of a state graph. For a given sequence of end-rows (r_1, r_2, ... ,r_n), we know for any consecutive end-rows, r_i, r_{i+1}, that $(r_i{+}1,r_{i+1},d_{i+1})$ is reachable from $(r_{i-1}+1,r_i,d_i)$ if and only if the begin-end-row pair $(r_i + 1,\ r_{i+1})$ is a valid stripe and $d_i \neq d_{i+1}$.

This approach should be contrasted with the alternative in which the nodes of the state graph equal twice the number of valid end-row sequences. In this alternative state graph, vertices would have the form (d_1, r_1, r_2, ... ,r_i), where d_1 is the initial fill direction and the $\{r_i\}$ form a valid end-row sequence. It will be proved under very limited assumptions that the number of vertices in the state graph corresponding to this second method is at least super-polynomial. The Donaldson-Meyer (**DM**) method uses the SQI property to reduce the number of vertices in the state graph to twice the number of valid begin-end pairs, which is polynomial with respect to input size.

6 Properties of Subproblems

6.1 Relationship between Subproblems and Stripes

A stripe is defined by a begin-end row pair and direction of assignment. As stated earlier, such a triple (b_{i-1},e_{i-1},d_{i-1}) also uniquely defines a subproblem and the U-turn region appearing within stripe *i-1*. By the same reasoning, (b_i,e_i,d_i) determines the U-turn region for stripe *i*. We now have a mechanism for defining the cells to be considered in the perimeter increment computation–namely those cells that are appended to subproblem (b_{i-1},e_{i-1},d_{i-1}) in order to get subproblem (b_i,e_i,d_i) (note: this region does not include the cells in the U-turn region for stripe i). Those cells appearing within this "appending" stripe region that correspond to an integral number of processors may be assigned independently of all other cells. A feasible solution of the problem can then be thought of as a sequence of "appending" stripes.

6.2 Construction of a Stripe-based Solution

Perimeter is computed in our approach incrementally, stripe-by-stripe as follows (see figure 9): perimeter in the first stripe (0,e_1,d_1) (not including its U-turn region in U_1) is computed, then the perimeter for (b_2,e_2,d_2) + U_1 - U_2, is added, etc. Figure 9 shows a fourth iteration of this process.

6.3 The Existence of Common Subproblems

Figure 10 shows two different sequences of stripe-heights used to partition the given grid. Both cases contain the subproblem ending with the stripe (b_5,e_5,d_5).

When the same subproblem appears in several other larger problems, the overall problem is said to possess the property of common subproblems. With respect to the state-graph viewpoint,

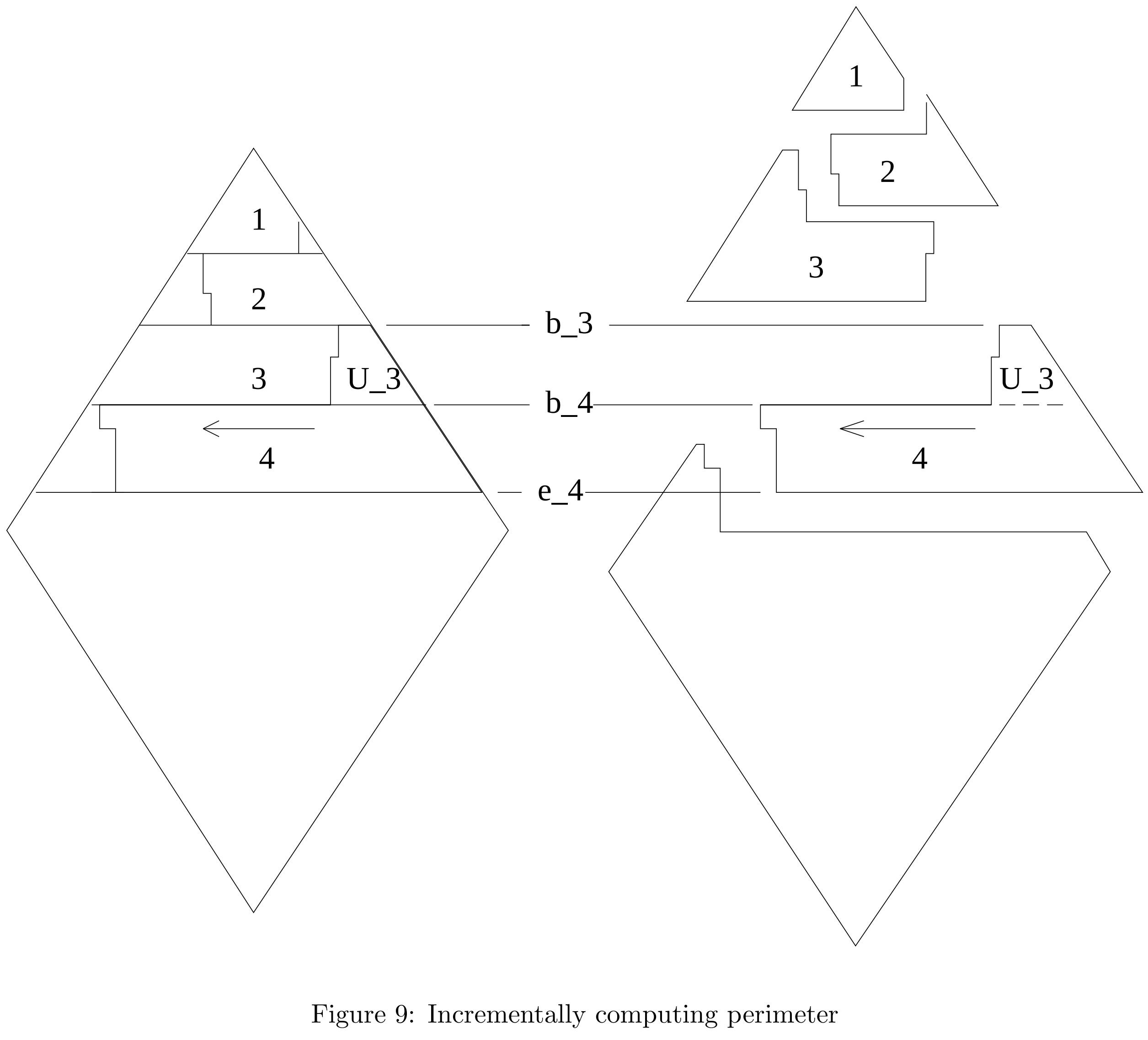


Figure 9: Incrementally computing perimeter

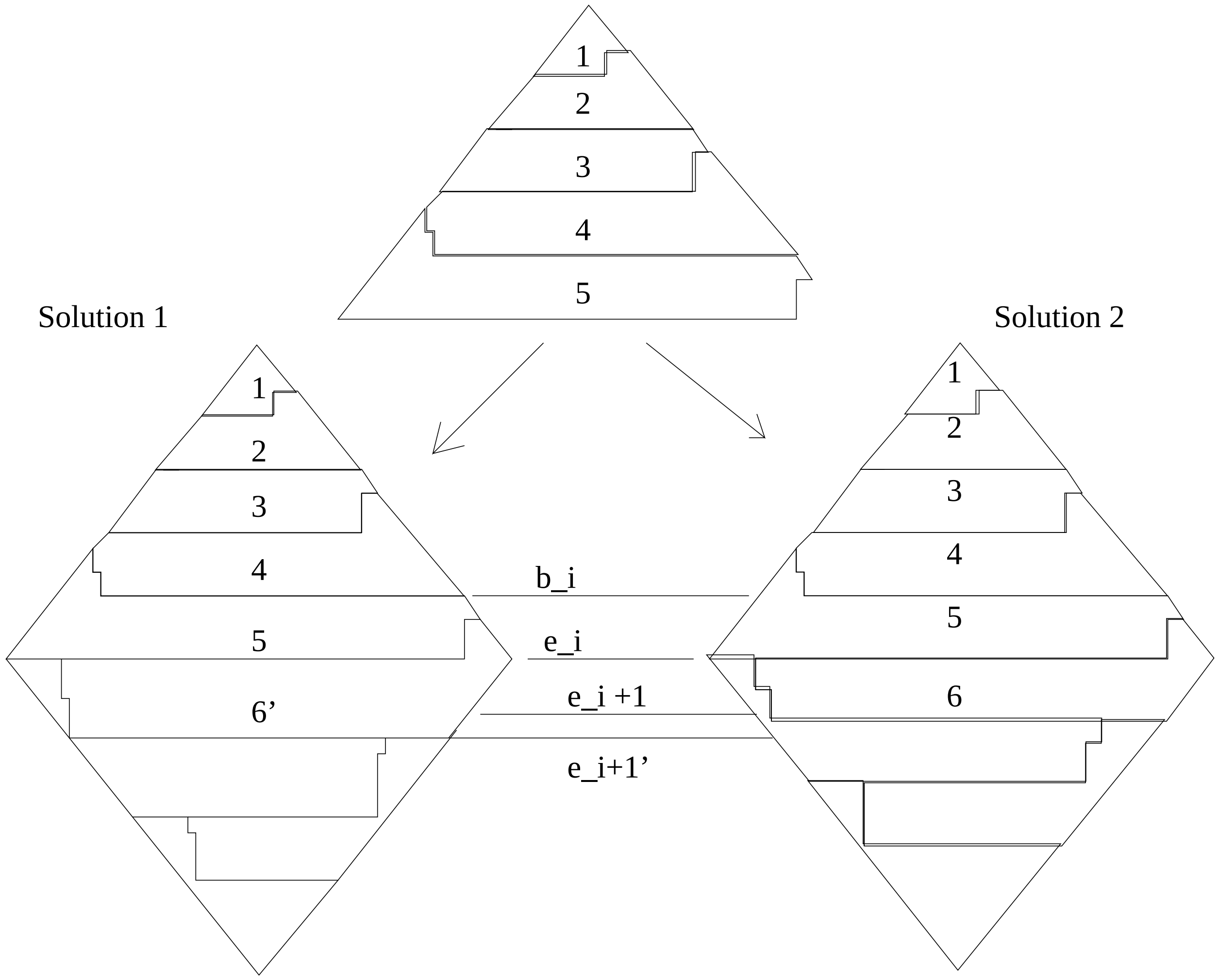


Figure 10: Feasible solutions with the same subproblem (stripes 1-5).

a subproblem is a node and this node may appear in multiple paths.

6.4 Optimal Substructure within a Solution

Cormen, Leiserson, and Rivest [4] state that a problem exhibits "optimal" substructure if an optimal solution can be decomposed into subproblem solutions each of which is optimal.

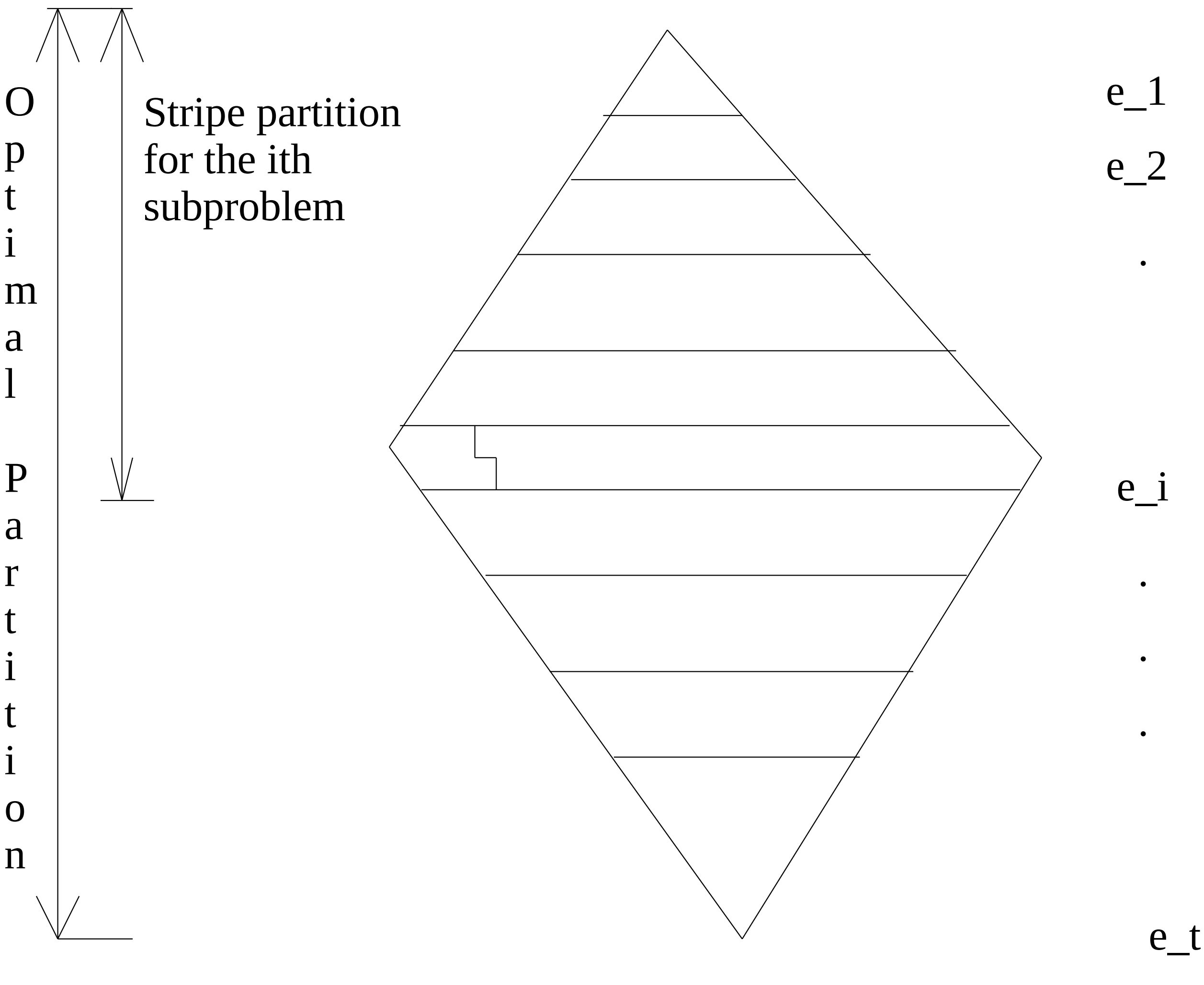


Figure 11: Optimal Substructure

Given a grid, assume that the sequence of end-rows (e_1,e_2, ... e_{t-1}, e_t) produces an optimal solution. Consider the subproblem ending at stripe (b_i,e_i,d_i) (see figure 11). For this subproblem, if the subsequence (e_1,e_2, ... e_{i-1}, e_i) was not optimal for the corresponding subproblem, then a better stripe partition for this subproblem (e_1',e_2', ... , e_{i-2}',e_{i-1}, e_i) (notice that the last stripe must have the same begin-row end-row pair for any solution for the subproblem) could be substituted into the larger solution. Because of the **SQI** property, this new sequence would produce a better overall solution, which would contradict the fact that we previously had an optimal solution. From a state-graph viewpoint, the shortest path has the property that each sub-path (beginning at the the start node) is also a shortest path.

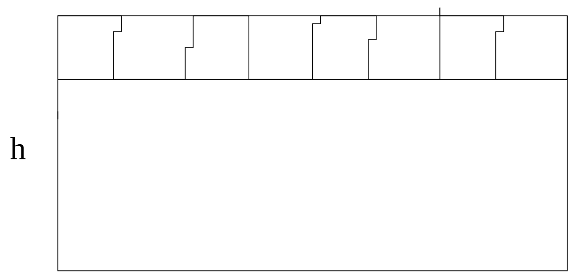
7 Stripe-Height Selection

In this section, we focus on the lower bound of super-polynomiality on the number of stripe-form solutions. Given an MxN rectangular grid, Martin [11] converts the original problem into a knapsack problem and uses knapsack software [10] for solving the reformulated problem. This approach will produce the best stripe-based solution, under the following assumptions. In addition to the rectangular grid assumption, Martin [11] assumes that MN/P is an integer and that only stripes containing an integer number of components are allowed. Under these constraints, there is no overflow into the next stripe and the perimeter in each stripe is independent of the other stripes, and independent of its position within the grid (see figure 12). Therefore, for a stripe of a given height, there will be a unique perimeter associated with that height, assuming that it is a "permissible" height corresponding to an integral number of components in the stripe. The problem can then be thought of as the selection of a set of heights totaling to M with minimum total perimeter. With respect to the knapsack problem, the stripe heights correspond to the constraint coefficients and the stripe-component perimeter totals correspond to the objective coefficients. Optimal solutions of the knapsack problem correspond to optimal stripe partitions (assuming column-wise assignment and permissible stripes heights) of the rectangular domain.

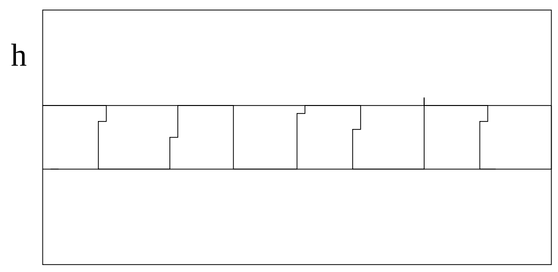
For the CM method of [3], the authors relax many of these restrictions (integral number of components per stripe and rectangular grids). Since CM is a genetic algorithm (GA), it doesn't in general produce guaranteed optimal stripe partitions, but does generate high-quality partitions. The number of different feasible solutions that are tried by the GA can be limited to guarantee a polynomial running time, but no quality guarantee can in general be associated with the result.

Whereas the type of problem that Martin [11] studied could be represented by three numbers (the length of the input being the logs of the dimensions of the rectangular domain and P), we assume that a full adjacency matrix or adjacency list is required to represent a general grid graph. This assumption implies that that the length of input is bounded below by (V + E). Since E is O(V), we have that the length of input $\Omega(V) = \Omega$(number of cells in the grid).

That raises the question of whether or not it is possible, in a reasonable amount of time (i.e.,



Same sequence of components in another stripe.



Same sequence of components in stripe 1.

Figure 12: Rectangular domain case illustrating Martin assumption that stripe perimeter depends only on stripe height (not position within domain) .

polynomial time), to evaluate the perimeter for all possible stripe-height sequences (assuming a given fill procedure). In this and later sections, it is assumed that the stripes will run perpendicular to the major axis of the grid, assumed to be the length M. (In practice, striping should be performed relative to both domain orientations, with the better results then being chosen.) By making this assumption, bounds are then expressed in terms of the size of the original problem.

The Christou-Meyer algorithm [3] uses the notion of **base heights**. The base height, **k**, for a given grid is equal to the floor of the square root of the area to be assigned to a processor. For this discussion, we will make the following assumptions:

1. The number of rows in the grid, M, is greater than $(k+1)^2$.
2. $k \geq 2$.
3. Stripes are perpendicular to the major axis of the grid.
4. Any set of consecutive k-1 rows contains at least one divider cell.

The first assumption is stronger than that of Christou-Meyer [3] and is needed soley for the bound in this section (not for the algorithm). The second and fourth assumptions are required to guarantee a range of valid stripe heights in the proof. The third assumption is needed to bound the cells in the grid by a function of M.

Christou and Meyer [3] showed that there exist non-negative integers α and β such that

$$M = \alpha(k) + \beta(k+1)$$

This sequence of α k's and β (k+1)'s will now be called a **base feasible solution**.

In order to discover a lower bound on the total number of valid stripe-height sequences, we need only consider sequences for which every stripe height falls within the following range:

$$[\ k\text{-}1,\ k,\ k+1,\ k+2].$$

With these assumptions, we now have the following theorem:

Theorem 1 *The number of feasible stripe-form solutions is* $\Omega\ (3^{M^{0.5}/4})$.

Proof

In order to prove this result, we need only look at those base feasible solutions that contain $S = \alpha + \beta$ stripes.

Assume that $\alpha > \beta$ (if $\beta > \alpha$ a similar argument with β replacing α can be given). The stripes associated with α are of height k. Let's further divide the α stripes into two approximately equal groups, so that the smallest group is of size at least $\alpha/2$ - 1. Call this group of stripes the **independent stripes**. The remaining stripes of height k along with all the β stripes will be called the **dependent stripes**.

For any stripe within the set of independent stripes, the height may be increased/decreased by one or may remain the same (a total of three possible heights), with respect to the base height, provided the opposite change is made to a corresponding dependent stripe. Thus the number of feasible solutions is at least

$$3^{\alpha/2}/3 \geq 3^{S/4}/3 \geq 3^{M/(k+1)4}/3 \geq 3^{M^{0.5}/4}/3$$

The second inequality follows from the fact that we have:

$$\begin{aligned} \text{M} &= \alpha(\text{k}) + \beta(\text{k+1}) \\ \text{M/(k+1)} &= \alpha(\text{k})/(\text{k+1}) + \beta(\text{k+1})/(\text{k+1}) \\ &= \alpha(\text{k})/(\text{k+1}) + \beta \\ &\leq \text{S} \end{aligned}$$

The third inequality follows from the assumption that M $\geq$ (k+1)^2

□

To convert this bound into the units of the input, note that

$$\text{M} \leq \text{(number of cells in grid)} \leq M^2$$

The number of stripes is super-polynomial with respect to the number of cells in the grid, since

$$3^{M^{0.5}/4}/3 \geq 3^{(no.of cells)^{0.25}/4}/3$$

From this result, we know that there are at least a super polynomial number of feasible solutions and that an exhaustive search it is not practical.

In [2], an upper bound of 2^{M-1} is given, but this bound allows stripes that may not contain enough cells to complete the assignment of a processor, and hence do not conform to the stripe-form solutions considered here.

8 An Example Illustrating the Methods

The example in figure 13 will be used to demonstrate both the the shortest-path approach and the dynamic-programming approach. The solution shown is stripe-fill optimal, in contrast to the non-optimal solution of figure 4, an alternative stripe-based partition of the same domain. (It is also easy to see that this solution is locally optimal (relative to two-cell swaps) in the context of general (not necessarily stripe-form) partitions. However, it is not globally optimal, since the positions of the **1**'s in the central column may be interchanged with the **2**'s to improve the total perimeter. This illustrates the potential for improved solutions based upon even more sophisticated fill procedures or post-processing procedures that focus on improving component boundaries. This topic is discussed further in the concluding section.)

```
                        1
                      1 1 2 2
                    1 1 1 2 2 2
                  1 1 1 1 2 2 2 2
                1 1 1 1 2 2 2 2 2 2
              1 1 1 1 1 2 2 2 2 2 2 2
            1 1 1 1 1 1 2 2 2 2 2 2 2 2
          1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2
        1 1 1 1 1 1 1 1 2 4 4 4 4 4 3 3 3 3
----------------------------------------------------
      6 6 6 5 5 5 5 5 5 4 4 4 4 4 4 3 3 3 3 3
    6 6 6 6 5 5 5 5 5 5 5 4 4 4 4 4 3 3 3 3 3 3
  6 6 6 6 6 5 5 5 5 5 5 5 4 4 4 4 4 4 3 3 3 3 3 3
6 6 6 6 6 6 5 5 5 5 5 5 5 4 4 4 4 4 4 3 3 3 3 3 3 3
  6 6 6 6 6 5 5 5 5 5 5 5 4 4 4 4 4 4 3 3 3 3 3 3
  6 6 6 6 6 6 5 5 5 5 5 5 4 4 4 4 4 4 3 3 3 3 3 3
----------------------------------------------------
    6 6 6 6 6 7 7 7 7 7 7 7 7 8 8 8 8 8 8 8 8 8
      6 6 6 7 7 7 7 7 7 7 7 8 8 8 8 8 8 8 8 8
        6 6 7 7 7 7 7 7 7 7 8 8 8 8 8 8 8 8
          6 7 7 7 7 7 7 7 7 8 8 8 8 8 8 8
            7 7 7 7 7 7 7 7 8 8 8 8 8 8
----------------------------------------------------
              9 9 9 9 9 9 9 9 9 9 9 8
                9 9 9 9 9 9 9 9 9 9
                  9 9 9 9 9 9 9 9
                    9 9 9 9 9 9
                      9 9 9 9
                        9
```

Figure 13: A stripe-fill optimal solution

Subproblem			Subproblem Data			
			Stripe	Total		
b_i	e_i	d_i	Perim	Perim	b_{i-1}	e_{i-1}
0	8	0	34	34	0	0
0	8	1	36	36	0	0
0	9	0	36	36	0	0
0	9	1	36	36	0	0
0	10	0	68	68	0	0
0	10	1	68	68	0	0
9	14	0	118	154	0	8
9	14	1	120	154	0	8
10	14	0	124	160	0	9
10	14	1	124	160	0	9
11	19	0	124	192	0	10
11	19	1	124	192	0	10
15	18	0	34	188	9	14
15	18	1	34	188	9	14
15	19	0	34	188	9	14
15	19	1	34	188	9	14
19	25	0	98	286	15	18
19	25	1	98	286	15	18
20	25	0	94	282	15	19
20	25	1	94	282	15	19

Table 2: Subset of perimeter data for stripe partitioning the grid in figure 13

9 A State-Graph Representation of Grid Graph Partitioning

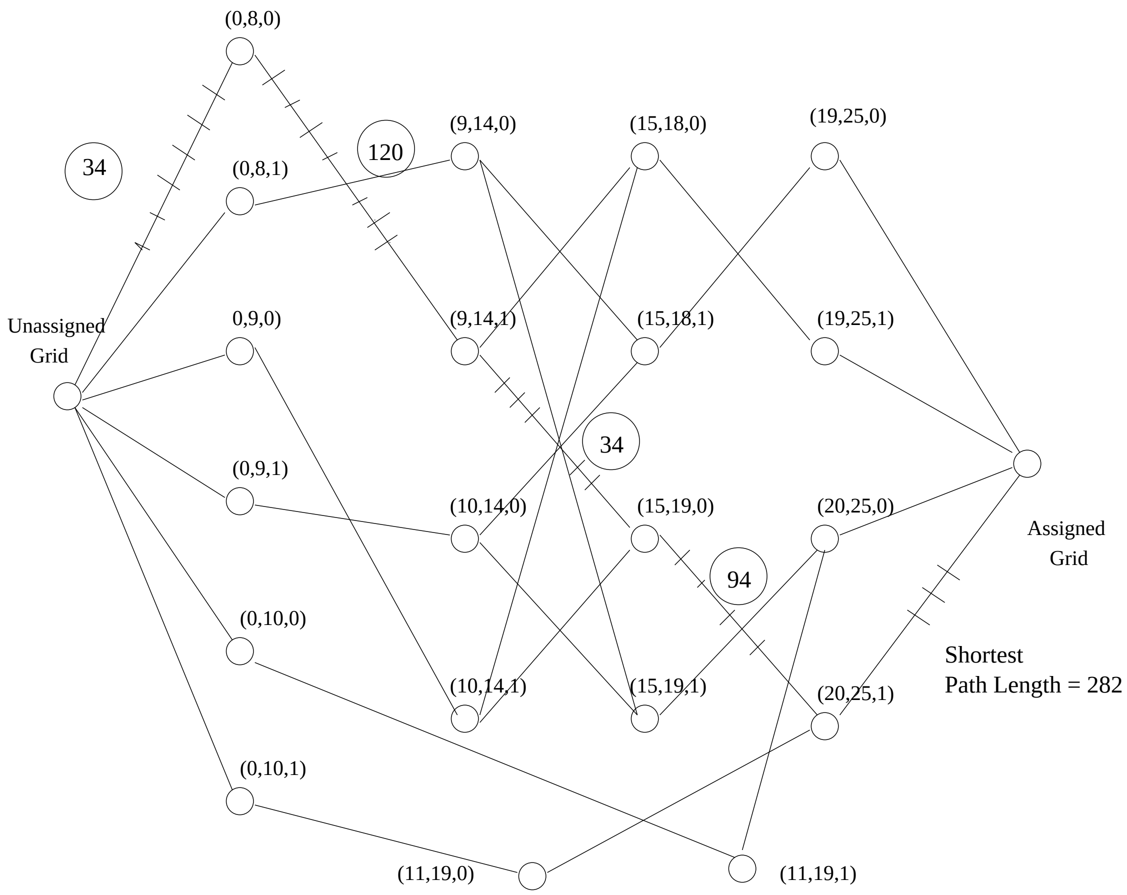


Figure 14: Partial state-graph (shortest path shown via hatched edges and edge lengths circled)

9.1 Background

An optimal partition of a grid graph can be thought of as a sequence of valid stripes, where each stripe is added incrementally. After each stripe is added, a new valid subproblem is generated. The discovery of an optimal solution can be thought of as going from the initial unassigned state to the final fully assigned state in an optimal fashion.

If a solution for a (sub)problem is represented by the sequence of stripe-defining end-rows (r_1, r_2, ...,r_n), the previous theorem showed that the number of such states is at least super-polynomial.

However, it was shown above that the **SQI** fill property implies that a begin-end-row pair and direction of assignment can be used to define a subproblem. Thus, instead of the sequence (r_1, r_2, ...,r_n) , all that is required is the begin-end pair corresponding to the current stripe. When this representation is used, the size of the state graph becomes polynomial.

Figure 14 shows the state graph for the data contained in table 2. The "unassigned grid" (source node) is connected to nodes that correspond to an initial valid stripe height. Each of these nodes is then connected to nodes whose corresponding subproblems represent a valid stripe extension, etc. The path containing the cross-hatched edges corresponds to an optimal partition of the original grid graph.

9.2 Properties of the State Graph

In the **state graph**, the number of vertices equals the total number of valid subproblems. An edge connects two vertices v_i and v_{i+1} if and only if a valid stripe can be appended to the subproblem defined by v_i to get the subproblem defined by v_{i+1}.

Let the ordered triple (b_i, e_i, d_i) represent a given subproblem(vertex). At most, there are M different choices for b_i. Once b_i has been determined, the total number of valid stripe heights (call this number h(b_i), where h(b_i) $\leq$ M) is an upper bound on the number of different possible values that e_i may assume. Let h = max(h(b_i)), over all eligible values of b_i (note that h may also be bounded by the size of the largest component, since no stripe should be taller than that value). For any given stripe, there are only two directions of assignment, hence we get the following bound:

$$\text{total number of vertices (subproblems)} \leq 2\text{Mh} \leq 2M^2$$

The inequality comes from the fact that under certain circumstances not all M rows may be begin rows and not all begin-end-row pairs, (b_i, b_i + k), k = 0, ..., h-1, represent a valid subproblem. In the shortest-path problem, there will be two additional vertices: a "source" (unassigned grid) and a "sink" (assigned grid).

For a given subproblem, an upper bound on the number of different stripes that may be appended to that subproblem equals the maximum number of valid stripe heights, h. It follows that:

$$\begin{aligned}\text{total number of edges} &\leq \text{the total number of vertices * h}\\ &\leq 2\text{M}h^2\end{aligned}$$

When constructing the state graph, the number of edges can be reduced by the observation that only the optimal preceding edge needs to be kept for each node.

Recall that the increment to the perimeter for the components assigned in a U-turn region and an appending stripe becomes the weight of the edge connecting vertices (b_{i-1},e_{i-1},d_{i-1}) and (b_i, e_i,

d_i), denoted as **perim(b_{i-1}, b_i, e_i, d_i)**. (In a slight abuse of notation, we use perim$(0, 0, e_1, d_1)$ to denote the perimeter of the first stripe (not including the first U-turn region).)

Figure 15 shows a portion of a state graph, under the assumption that the initial stripe always is assigned left to right. We make this assumption below to reduce the complexity of the proofs, but the extension to the bi-directional case illustrated in figure 14 is straightforward.

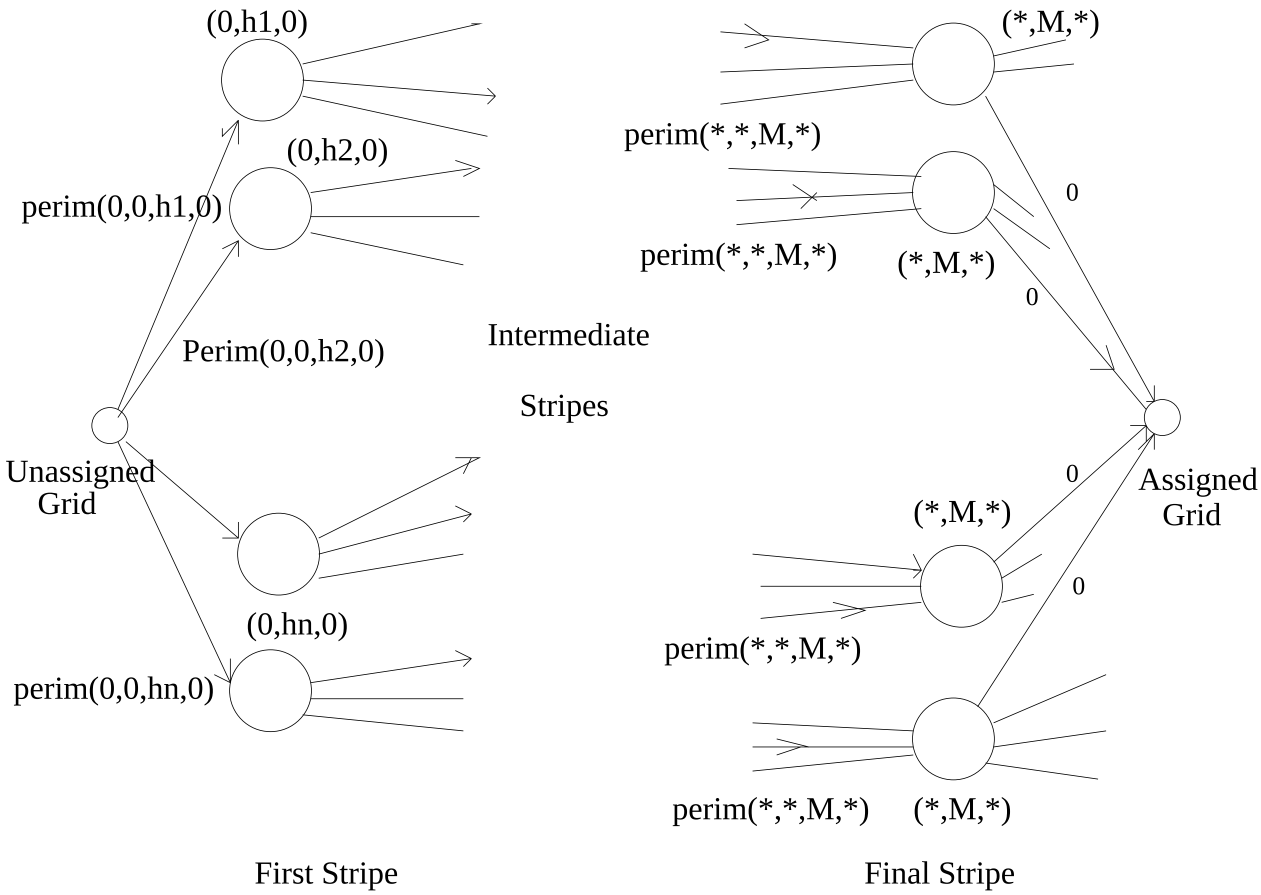


Figure 15: State graph

9.3 Proof of Optimality

We must now show that the path corresponding to an optimal striping is contained within the graph.

Theorem 2 *The previously defined graph contains a path corresponding to every feasible sequence of end-rows. Conversely, every path beginning at node 0 corresponds to a feasible sequence of end-rows.*

Proof - (by induction)

Base Case - A sequence of length one. Given that (e_1) is a feasible sequence, $(0,e_1)$ is a valid stripe. It follows that vertex $(0,e_1,0)$ and the corresponding edge (source vertex)-$(0,e_1,0)$ exists within the graph.

Induction Hypothesis - Given a feasible sequence of end-rows of length n, there exists a path in the graph from the unassigned grid node to the node corresponding to the last stripe in the sequence.

Induction Step - Given a sequence of n+1 valid stripes, by the induction hypothesis, there exists a path from the unassigned grid node to the node corresponding to the nth stripe. Since all n+1 stripe heights are valid, there must be an edge from the node corresponding to the nth stripe to the node corresponding to the last stripe.

The converse follows directly from the way the state graph was constructed.

□

Lemma 1 *The shortest path in the transformed graph corresponds to an optimal stripe-based solution.*

Proof (by contradiction) - From theorem 2 , we know that transformed graph contains the path for an optimal solution. The value of the perimeter for the optimal striping must equal the shortest path in the graph. For if this were not the case, then the stripe heights corresponding to the shorter path would produce a better solution in the original grid graph, which would be a contradiction.

□

Every path within a state graph represents a valid sequence of stripe heights. Any shortest path for a state graph then represents a **stripe-fill optimal solution(SFO)** that is optimal with respect to all stripe-form solutions associated with the given fill procedure.

10 A Dynamic-Programming Approach

10.1 Background

Earlier, it was shown that an optimal stripe-based partition of a grid graph possessed two qualities: optimal substructure and common subproblems. These are two properties that are present when dynamic programming is applicable. For the remainder of this this section, an optimal solution will be dissected to provide intuition as to why a recurrence can model the optimal stripe-based solution.

Table 2 contains the striping information for a small example. There are four different last stripes in this case. The optimal solution corresponds to the best of these four different possibilities, node (20, 25, 1). The perimeter associated with this node therefore satisfies:

optimal perimeter	= incremental perimeter corresponding to
	last stripe in an optimal solution
	+ perimeter for subproblem with this last stripe
	removed)
Perim(20, 25, 1)	= 94 + Perim(15,19,0)

Note that the value of Perim(15,19,0) is also optimal for the corresponding subproblem, since it represents the optimal solution of that subproblem obtained by considering all possible edges leading to the node (15,19,0). Similar arguments apply to the other nodes on the path leading back to the unassigned grid node

10.2 A Recurrence Relation for Grid-Graph Partitioning

This section is devoted to formally defining a dynamic programming recurrence that models the optimal solution. The proof that the recurrence models an optimal partition closely follows the proof for the shortest-path solution and can be found in [6].

We will now define a recurrence that may be used to obtain the perimeter for subproblem defined by (i,j,d), given the perimeter of smaller subproblems. This value will be stored in an array Perim(i,j,d). We need the following definitions for the recurrence.

Definitions

d = direction of assignment

V = number of cells in the grid

VALID-STRIPE = the set of begin-end-row pairs, (b_i, e_i), such that (e_i - b_i + 1) is less than or equal to h (the maximum allowable stripe height) and the corresponding rows contain a divider cell.

$$Perim(i,j,d) = \begin{cases} 5V & \text{if (i,j) is not in VALID-STRIPE} \\ perim(0,i,j,d) & \text{if (i = 0) and ((i,j) is in VALID-STRIPE)} \\ min_b(Perim(b,i-1,(d-1)mod2)+ & \\ \quad f(b,i,j,d)) & \forall \text{ b s.t. (b,i-1) is in VALID-STRIPE} \end{cases}$$

where

$$f(b,i,j,d) = \begin{cases} 1 & \text{if Perim(b,i-1,(d-1) mod 2)} \geq 5 * \text{V} \\ perim(b,i,j,d) & \text{otherwise} \end{cases}$$

Time Analysis -

The dynamic-programming array, **Perim**, is M x M, where M is the number of rows in the grid. For this structure, certain facts follow:

1. For each row of Perim, the number of cells that contain non-trivial stripe data is less than or equal to the maximum valid stripe height,h.
2. For the entire array Perim, it then follows that the total number of cells which will contain non-trivial stripe data is less than or equal to h*M.

Each cell in the dynamic-programming array represents a unique subproblem within the grid. In order to determine the corresponding entry in the dynamic-programming array the following must be done. (This time analysis contains details of the actual implementation. In particular, it references the nine alternative assignment techniques (based on various combinations of column-wise and row-wise assignments) that are evaluated per U-turn region. After all nine are evaluated, the best is selected, see [6] for more details. An alternative technique based on a recursive application of the general approach to the U-turn regions is considered in the final section of this paper. Also, we will define the variable N to be the length of the minor axis and $\bar{V}$ = min(V, hN).)

1. Create the array of cells to be assigned when appending a stripe. There will be a different array for each allowed b_i.
   Time = O(h$\bar{V}$)
2. For each stripe-subproblem combination, compute nine alternative assignments of the cells of the U-turn/overflow region.
   Time = O(h$\bar{V}$)
3. For each stripe-subproblem and each stripe assignment, calculate the total perimeter. (This can be done by looking at each assigned cell and adding this amount to the perimeter for the subproblem.)
   Time = O(h$\bar{V}$)

The total work for each element in Perim is thus O(h$\bar{V}$). The total amount of work to fill in the non-trivial cells for the dynamic-programming array is O(Mh$^2\bar{V}$). The amount of time to initialize the dynamic-programming array is O(M^2). The overall time becomes O(M^2 + Mh$^2\bar{V}$).

How does this relate to the length of input? It was assumed that to represent this problem that an adjacency matrix or list is required. This would force the length of input to be at least equal to the number of vertices in the grid. Assuming a binary representation, that would force the length of input to be Ω(V).

11 Implementation

11.1 Software and Hardware Issues

All coding was done using the C programming language, which was compiled using g++, a GNU project C++ compiler [1]. All software development was done on a Pentium Pro 200Mhz machine with 64MB of RAM. All results were generated using this Pentium machine and an Ultra Enterprise 6000 Server.

12 Results

Table 3 contains the results for several non-rectangular grids comparing METIS/Pmetis/Kmetis (see [6]), CM (the genetic algorithm version, again see [6]), and the DM algorithm. For the actual partitions produced by the DM methodology, see figures 16, 17, and 18. Note that DM substantially outperforms METIS in all cases, and ties or outperforms CM except for the first test

Shape	Cells	Comps	METIS error bound %	CM error bound %	DM error bound % (h bound, max observed height)	DM - CM Relative Improvement(%)
Diamond	4019	16	25.98	16.40	17.19 (30,20)	-4.82
Diamond	4019	64	22.07	13.37	10.25 (20,15)	23.34
Ellipse	823	16	14.58	8.33	8.33 (20,7)	0
Ellipse	823	64	5.37	5.36	3.58 (20,4)	33.21
Torus	7696	16	30.97	11.50	10.51 (40,28)	8.61
Torus	7696	64	22.59	11.00	8.38 (20,14)	23.82

Table 3: Comparison of performance: DM, CM and METIS.

problem. For this problem CM did better for the following reason: under certain conditions, CM handles assignments for the overflow from the U-turn regions differently than by DM. We discuss below a recursive procedure for handling these assignments that should outperform the heuristics of both CM and DM. Upper bounds (h) on stripe heights were used to reduce the amount of computing. The bounds were set higher than the maximum observed heights in the solutions.

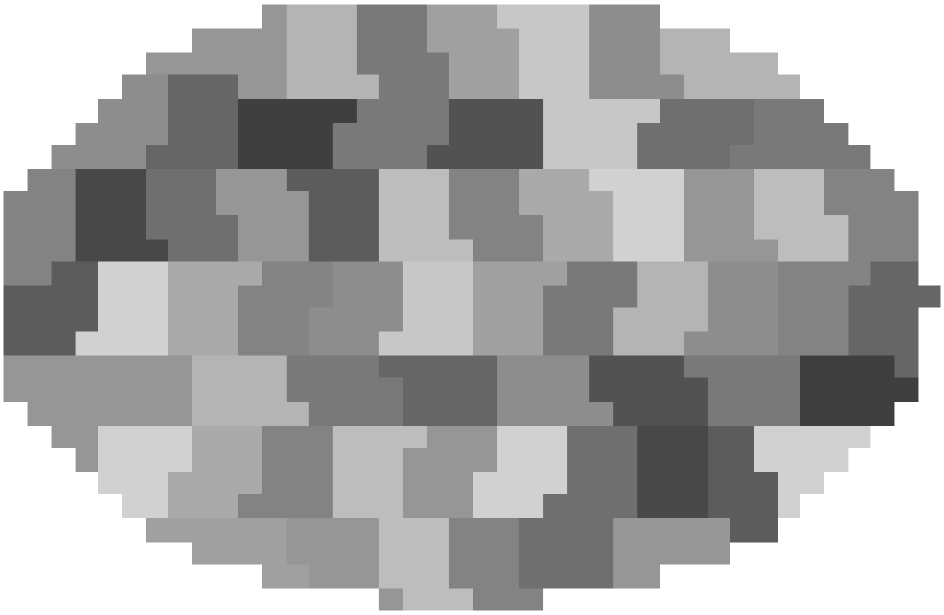


Figure 16: The DM partition of the 823-cell ellipse into 64 parts.

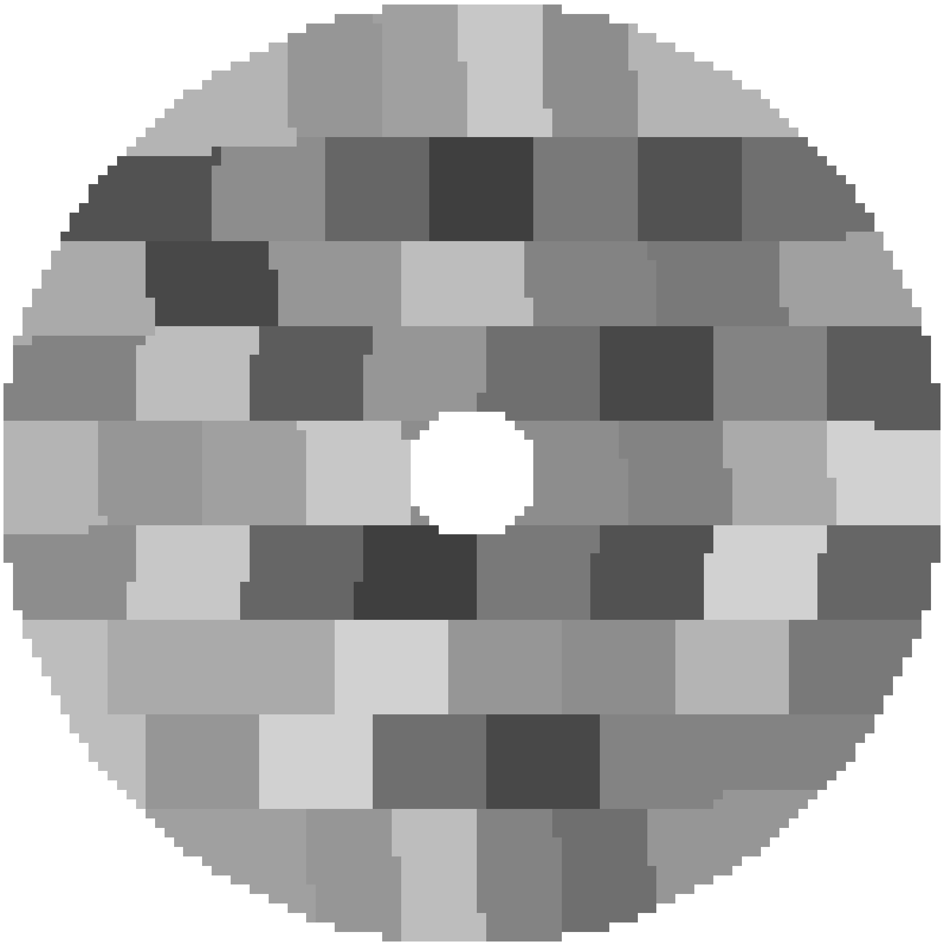


Figure 17: The DM partition of the 7696-cell torus into 64 parts.

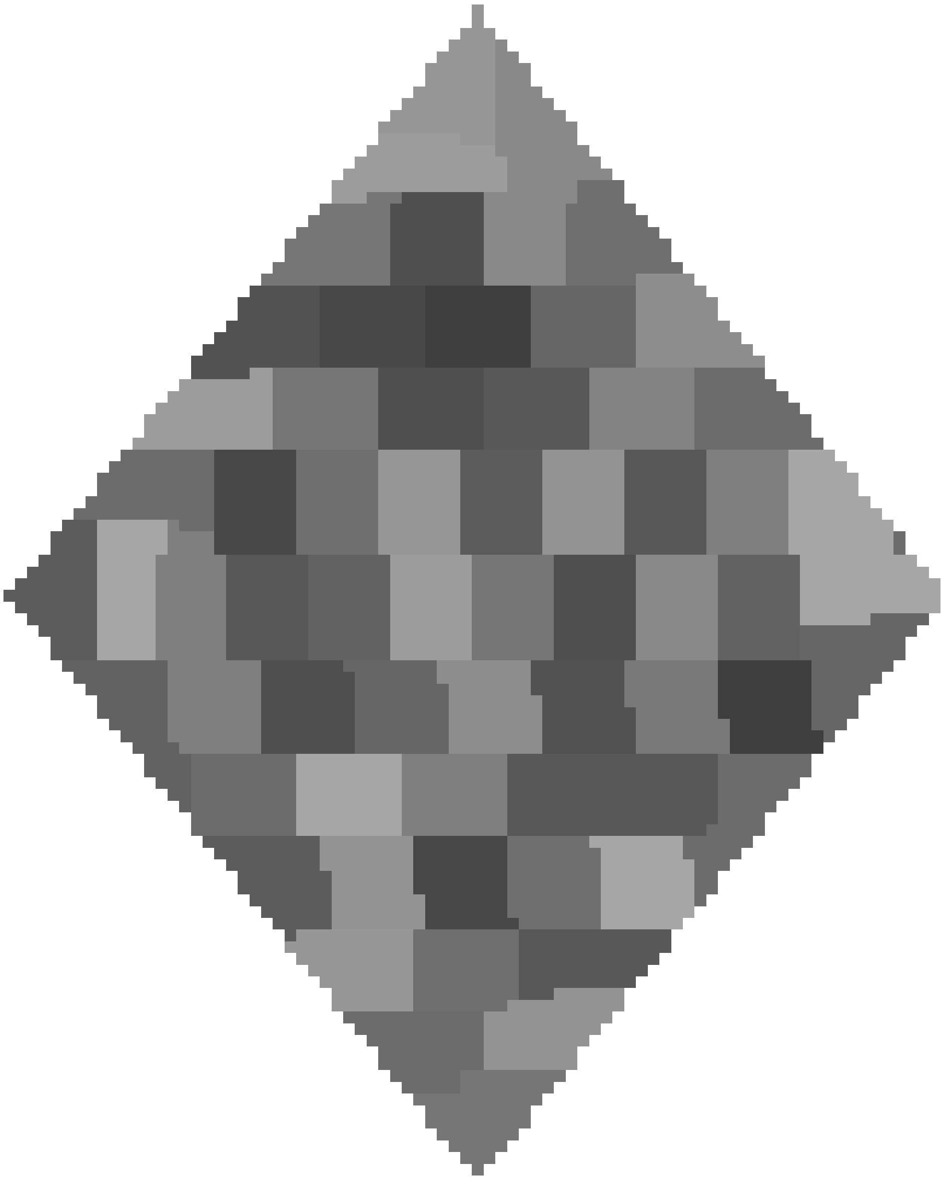


Figure 18: The DM partition of the 4019-cell diamond into 64 parts.

13 Conclusions

From this research three main results have emerged. Theorem 1 demonstrates that a brute-force approach has a running time which is at least super-polynomial.

The second major contribution of this research is the decomposition of the original problem into smaller problems based on the principles of **optimum substructure** and **common subproblems**. From this decomposition, two concepts have emerged. The first is the idea of a **stripe-quasi-independent(SQI)** fill procedure. A SQI fill procedure allows the perimeter value for subproblem (e_{i-1}, e_i) to be computed as the perimeter value for the subproblem (e_{i-2}, e_{i-1}) plus the perimeter for the components appearing in the corresponding u-turn region for stripes i-1 plus the perimeter for the components assigned in stripe i that fall outside of the u-turn region for stripe i. The second concept is a **stripe-fill optimal solution(SFO)**. A SFO optimal solution is one that is optimal with respect to the set of feasible solutions obtained by:

2) specifying an SQI fill procedure.

2) applying the fill procedure to a valid sequences of end-rows.

The third major contribution of this research is the organization of the solution set. We have developed a dynamic programming heuristic that exploits the property of **common substructure**. Through the use of dynamic programming, the set of solutions for the subproblems can be organized in such a manner so as to eliminate any duplication of work for solving subproblems. As a result of this approach, we have discovered a heuristic that will produce a stripe-fill optimal solution in polynomial time.

Future directions for research include various refinements and extensions of the approach in the 2-D case as well as generalization to the 3-D case. The most troublesome subregions with respect to perimeter contributions are the U-turn and overflow regions. In order to improve on the perimeter contributions from these regions, a recursive approach could be taken. That is, the stripe-based approach used for the domain as a whole could be directly applied to those subdomains, in effect allowing a completely different striping mode to be used for the subdomains. (The striping for a subdomain could even be done vertically as opposed to horizontally.) Another level of recursion could even be performed for new U-turn regions generated within these subdomains by their modified striping. Another approach for avoiding bad components in the U-turn region would be to keep a record of the component sizes used thus far, and to determine the mix of component sizes that best fits (in the sense of minimizing overflow) a newly added stripe (a record of used or remaining component sizes could be carried along with the other node information in the shortest path approach). A variety of heuristics for post-processing the optimal stripe-form solution are under consideration. One method would use a modification of Kernighan-Lin heuristic focussing on the perimeter cells in U-turn regions and any stripes whose components are not of good quality, employing swap selection priorities determined by perimeter contributions. To extend the stripe-selection process to 3-D grid graphs only requires a modification as to how a subproblem is to be defined. The basic notion of focussing on stripe-form solutions generalizes in an obvious way to solutions obtained by assigning within "horizontal" slices of appropriately chosen widths, with special approaches for U-turn and overflow regions at the ends of slices.

References

[1] ANONYMOUS. http://gcc.gnu.org.

[2] I. Christou. *Distributed Genetic Algorithms for Partitioning Uniform Grids.* PhD thesis, University of Wisconsin - Madison, August 1996.

[3] I. T. Christou and R. R. Meyer. Optimal equi-partition of rectangular domains for parallel computation. *Journal of Global Optimization*, 8:15–34, January 1996.

[4] T. H. Cormen, C. E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* McGraw-Hill, New York, 1990.

[5] W.W. Donaldson. Locally Optimal Striping for Rectangular Grids. Manuscript, 1997.

[6] W.W. Donaldson. *Grid-Graph Partitioning.* PhD thesis, University of Wisconsin - Madison, May 2000.

[7] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, 1979.

[8] S. Ghandeharizadeh, R.R. Meyer, G. Schultz, and J.Yackel. Optimal balanced partitions and a parallel database application. *ORSA Journal on Computing*, 4:151–167, 1993.

[9] Udi Manber. *Introduction to Algorithms: A Creative Approach.* Addison-Wesley, 1989.

[10] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations.* Wiley, 1990.

[11] W. Martin. Fast equi-partitioning of rectangular domains using stripe decomposition. *Discrete Applied Mathematics*, 82:193–207, 1998.

[12] R. J. Schalkoff. *Digital Image Processing and Computer Vision.* John Wiley & Sons, 1989.

[13] J. Strikwerda. *Finite Difference Schemes and Partial Differential Equations.* Wadsworth & Brooks, 1989.

[14] J. Yackel, R. R. Meyer, and I. Christou. Minimum-perimeter domain assignment. *Mathematical Programming*, 78:283–303, 1997.

[15] J. Yackel, R. R. Meyer, and I. Christou. Minimum perimeter domain assignment. *Mathematical Programming*, 78:283–303, 1997.