A Fast Algorithm for Performance-Driven Module Implementation Selection *

Edward Y.C. Cheng and Sartaj Sahni
Department of Computer and Information Science and Engineering
University of Florida
Gainesville, FL 32611-6120
{yccheng, sahni}@cise.ufl.edu

March 2, 1999

Abstract

We develop an $O(p \log n)$ time algorithm to obtain optimal solutions to the p-pin n-net single channel performance-driven implementation selection problem in which each module has at most two possible implementations (2-PDMIS). Although Her, Wang and Wong [1] have also developed an $O(p \log n)$ algorithm for this problem, experiments indicate that our algorithm is twice as fast on small circuits and up to eleven times as fast on larger circuits. We also develop an $O(pn^{c-1})$ time algorithm for the c, $c > 1$, channel version of the 2-PDMIS problem.
Keywords and Phrases: channel density, net span constraint, performance-driven module implementation selection, complexity.

1 Introduction

In the channel routing problem, we have a routing channel with modules on the top and bottom of the channel, the modules have pins and subsets of pins define nets. The objective is to route the nets while minimizing channel height. Several algorithms have been proposed for channel routing (see, for example, [2].)

*This research was supported, in part, by the Army Research Office under grant DAA H04-95-1-0111.

When the modules on either side of the channel are programmable logic arrays, we have the flexibility of reordering the pins in each module; any pin permutation may be used. The ability to reorder module pins adds a new dimension to the routing problem. Channel routing with rearrangeable pins was studied by Kobayashi and Drozd [3]. They proposed a three step algorithm (1) permute pins so as to maximize the number of aligned pin pairs (a pair of pins on different sides of the channel is aligned iff they occupy the same horizontal location and they are pins of the same net), (2) permute the nonaligned pins so as to remove cyclic constraints, and (3) while maintaining an acyclic vertical constraint graph, permute unaligned pins so as to minimize channel density. Lin and Sahni [4] developed a linear time algorithm for step (1) and Sahni and Wu [5] showed that steps (2) and (3) are NP-hard. Tragoudas and Tollis [6] present a linear time algorithm to determine whether there is a pin permutation for which a channel is river routable. They also showed that the problem of determining a pin permutation that results in minimum density is NP-hard in general, and they developed polynomial time algorithms for the special case of channels with two terminal nets and channels with at most one terminal of each net being in each module.

Variants of the channel routing with permutable pins problem have also been studied [7, 8, 9, 10]. In these variants restrictions are placed on the allowable pin permutations for each module. Restrictions may arise, for example, because the module library contains only a limited set of implementations of each module [7]. Another variant, considered by Cai and Wong [8] permits the shifting of modules and pins so as to minimize channel density. Extensions to the case when over the cell routing is permitted are considered in [9] and [10].

The variant of the channel routing with permutable pins problem that we consider in this paper is the performance-driven module implementation selection (PDMIS) problem formulated by Her, Wang and Wong [1]. In the k-PDMIS problem, we are given two rows of modules with a routing channel in between, up to k possible implementations for each module (different implementations of a module differ only in the location of pins, the module size and pin count are the same); and a set of net span constraints (the span of a net is the distance between its leftmost and rightmost pins). A *feasible* solution to a k-PDMIS instance is a selection of module implementations so that all net span constraints are satisfied. An *optimal* solution is a feasible solution with minimum channel density.

Figure 1(a) shows a routing channel with two modules on either side of the routing channel. Assume that each module has two implementations and that the pin locations for the second implementation of each module are as in Figure 1(b). The net span constraints of the five nets are 4,4,1,1 and 6, respectively. This defines an instance of the 2-PDMIS problem. Using the implementations of Figure 1(a), the net spans are 5,3,1,1 and 6, respectively. The span constraint of net 1 is violated. If each module is implemented as in Figure 1(b), the net spans are 1,5,1,1 and 4, respectively. This time, the span constraint of net 2 is violated. If we implement the modules as in Figure 1(c) (i.e., for modules 1 and 2 use the implementations of Figure 1(a) and for modules 3 and 4, use the implementations of Figure 1(b)), the net spans are 4,4,1,1 and 6, respectively. Now, the net span constraints are satisfied for all nets. The channel density when module implementations are selected as in Figure 1(c) is 5. Selecting module implementations as in Figure 1(d), we obtain a feasible solution whose density is 3.

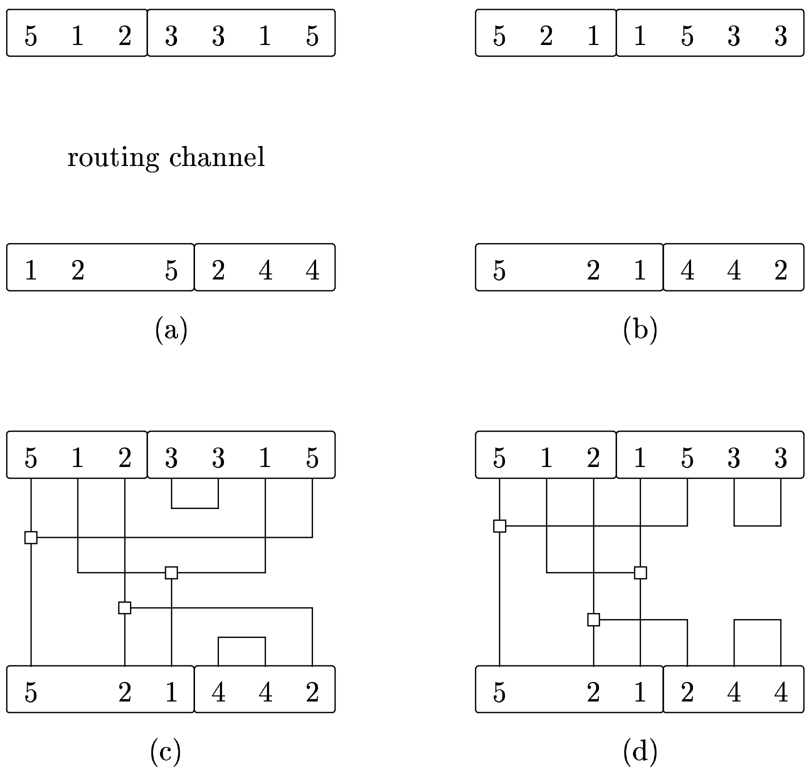

Figure 1: An example PDMIS problem. (a) first implementation; (b) second implementation; (c) selections that satisfy the net span constraints; (d) selection with better density

Her, Wang and Wong show that the k-PDMIS problem is NP-hard for every $k \geq 3$. For the 2-PDMIS problem, they develop an $O(p \log n)$ algorithm to find an optimal solution. In this paper, we develop an alternative $O(p \log n)$ algorithm to find an optimal solution to the 2-PDMIS problem. Experiments indicate that our algorithm is twice as fast on small circuits and up to eleven times as fast on larger circuits.

We begin, in Section 2, by providing an overview of the $O(p \log n)$ algorithm of [1]. Then, in Section 3, we describe our $O(p \log n)$ algorithm. In Section 4, we develop an $O(pn^{c-1})$ algorithm for the c, $c > 1$, channel 2-PDMIS problem. Experimental results using the single channel 2-PDMIS algorithm are presented in Section 5.

2 The Known $O(p \log n)$-Time Algorithm

Her, Wang and Wong [1] show how to transform an instance P of 2-PDMIS with net span constraints and a constraint, d, on channel density into an instance S of the 2-SAT problem (each instance of the 2-SAT problem is a conjunctive normal form formula in which each clause has at most two literals.) The 2-SAT instance S is satisfiable iff the corresponding 2-PDMIS instance has a feasible solution with channel density $\leq d$. The size of the constructed 2-SAT formula S is $O(p)$, where p is the total number of pins in the modules of P. Since the channel density of the optimal solution is in the range $[1, n]$, where n is the total number of nets, a binary search over d can be used to obtain an optimal solution in $O(p \log n)$ time.

Her, Wang and Wong [1] use one boolean variable to represent each module. The interpretation is, variable x_i is true iff implementation 1 of module i is selected. The steps in the 2-PDMIS algorithm of [1] are:

1. Construct the 2-SAT formula C_{span} such that C_{span} is satisfiable iff the given 2-PDMIS formula has a feasible solution. This is done by constructing a 2-SAT formula for each net and then taking the conjunction of these instances. For each net j, the leftmost and rightmost modules on the top row and bottom row are identified. These (at most four) modules are the critical modules for net j as the span of net j is determined solely by these modules. A 2-SAT formula involving the boolean variables that represent these critical modules is constructed. This 2-SAT formula has the property that truth value assignments satisfy the 2-SAT formula

iff the corresponding module implementations cause the net span constraint for net j to be satisfied.

2. Construct a 2-SAT formula C_{den} using a density constraint d. C_{den} is satisfiable only by module implementation selections which result in a channel density that is $\leq d$. To construct C_{den}, partition the channel into a minimum number of regions such that no region contains a module boundary in its interior; for each region, construct a 2-SAT formula so that the density in the region is $\leq d$ whenever the 2-SAT formula is true (this 2-SAT formula involves only the module in the top row of the region and the one in the bottom row); take the conjunction of the region 2-SAT formulae.

3. Determine if the 2-SAT formula $C_{span} \wedge C_{den}$ is satisfiable by using the strongly connected components method described in [11]. This requires that we first construct a directed graph from $C_{span} \wedge C_{den}$.

4. Repeat steps 2 and 3 performing a binary search for the minimum value of d for which $C_{span} \wedge C_{den}$ is satisfiable.

As shown in [1], the size of $C_{span} \wedge C_{den}$ is $O(p)$; step 1-3 take $O(p)$ time; and the overall complexity is $O(p \log n)$.

3 Our $O(p \log n)$-Time Algorithm

Our algorithm is a two stage algorithm that does not construct a 2-SAT formula. In the first stage, we construct a set of $2m$ "forcing lists", where m is the number of modules. $L[i]$ is a list of module implementation selections that get forced if the first implementation of module i, $1 \leq i \leq m$ is selected; $L[m+i]$ is the corresponding list for module i when the second implementation of module i is selected. By forced, we mean that unless the module implementations on $L[i]$ ($L[m+i]$) are selected whenever the first (second) implementation of module i is selected, we cannot have a feasible solution that also satisfies the given density constraint. In the second stage, we use the limited branching method of [12] and the forcing lists constructed in stage 1 to obtain a module implementation selection that satisfies the net span and density constraints (provided such a selection is possible). To find an optimal solution, we use binary search to determine the smallest density constraint for which a feasible solution exists.

3.1 Stage 1

In stage 1, we construct the forcing lists $L[1..2m]$. If the selection of implementation 1 of module i requires that we select implementation 1 of module j, we place j on the list $L[i]$; if the selection of implementation 1 of module i requires that we select implementation 2 of module j, we place $m+j$ on $L[i]$. Similarly when the selection of implementation 2 of module i requires a particular implementation be selected for module j, we place either j or $m+j$ on $L[m+i]$. To assist in the construction of the forcing lists, we use another array $C[1..m]$ with $C[i]=0$ if no implementation of module i has been selected so far; $C[i]=1$ if the first implementation of module i has been selected; and $C[i]=2$ if the second implementation has been selected.

First, we construct the forcing lists necessary to ensure the net span constraints. For each net i for which a net span constraint is specified, identify the leftmost and rightmost modules, in each module row, that contain net i (see Figure 2). There are at most four such modules: leftmost module with net i in the top module row (module u of Figure 2), leftmost in the bottom module row (w), rightmost in top row (v) and rightmost in bottom row (x). The span of net i is determined by a pair of these critical modules. One module in this pair is a leftmost critical module and the other is a rightmost critical module. So, there are at most four module pairs to consider (for the example of Figure 2, these four pairs are (u,v), (w,v), (u,x) and (w,x)).

When a critical module pair is considered, let A denote the implementation of the left module (of the pair) in which the leftmost pin of net i is to the right of the leftmost pin of net i in the other implementation (ties are broken arbitrarily). Let A' denote the other implementation of the left module. Let B denote the implementation of the right module for which the rightmost pin of net i is to the left of the rightmost pin of net i in the other implementation (ties are broken arbitrarily). Let B' denote the other implementation of the right module. In the example of Figure 2, consider the critical module pair (u,x), u is the left module and x is the right module. The second implementation of u is A and its first implementation is A'; the first implementation of x is B and its second implementation is B'. There are four ways in which we can select the implementations of the modules u and x: (A,B), (A,B'), (A',B) and (A',B'). For each of these four selections, we can determine the span of net i and classify the selection as feasible (i.e., does not violate the net span constraint) or infeasible. Notice that if the selection (A, B) violates the net span constraint for net i, then each of the remaining three selection pairs also violates the net span constraint for this net.

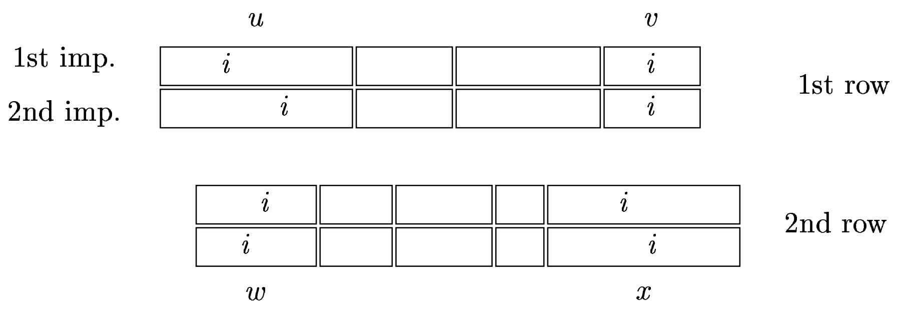

Figure 2: Critical modules of net i

We have the following possibilities:

Case 1: [No selection is infeasible.] All four selections are feasible. In this case no addition is made to the forcing lists.

Case 2: [Exactly one selection is infeasible.] The infeasible selection must be (A', B') and the other three selections are feasible. Now, the selection of A' forces us to select B and the selection of B' forces us to select A. Therefore, we add B to the forcing list for A' and A to that for B'. To add B to the forcing list of A' (and similarly to add A to the list of B'), we first check C to determine if an implementation for the module corresponding to A' has already been selected. If no implementation has been selected, we simply append B to the list for A'. If the implementation A has been selected, then we do nothing. If the implementation A' has been selected, then the implementation B is forced and we run the function *Assign* (L, C, B) of Figure 3 which selects implementation B as well as other implementations that may now be forced. This function returns the value False iff it has determined that no feasible solution exists.

Case 3: [Exactly two selections are infeasible.] This can arise in one of two ways (a) (A, B) and (A, B') are feasible and (A', B') and (A', B) are infeasible and (b) (A, B) and (A', B) are feasible and (A', B') and (A, B') are infeasible. In case (a), we must select implementation

```
Algorithm Assign $(L, C, M)$
  /* Select implementation $M$ and related modules */

  if $M$ is selected then
    return True;
  if $M'$ is selected then
    return False;
  /* $M$ is undecided */
  Mark $M$ selected in $C$;
  for each $X \in L[A]$ do
    if not Assign $(L, C, X)$ then
      return False;
  end for
  Remove $L[M]$ and $L[M']$;
  return True;
```

Figure 3: Function *Assign*

A. This is done by executing *Assign* (L, C, A). In case (b), we must select implementation B; so, we perform *Assign* (L, C, B).

Case 4: [Exactly three selections are infeasible.] Now (A, B) is the only feasible selection and we perform *Assign* (L, C, A) and *Assign* (L, C, B).

Case 5: [All four selections are infeasible.] In this case, the 2-PDMIS instance has no feasible solution.

Once we have constructed the forcing lists for the net span constraints, we proceed to augment these lists to account for the channel density constraint. Of course, this augmentation is to be done only when we haven't already determined that the given 2-PDMIS is infeasible. Our strategy to augment the forcing lists to account for the density constraint begins by partitioning the routing channel into regions such that no module boundary falls inside of a region (see Figure 4).

To ensure that the channel density is $\leq d$, we require that the density in each region of the channel be $\leq d$. This can be done by examining each channel region. Let T be the module on the top row of the channel region and B the module on the bottom row. The density in this channel region is completely determined by the nets that enter this region from its left or right and by the implementations of T and B. Let T_1, T_2 (B_1, B_2) denote the two possible implementations of T

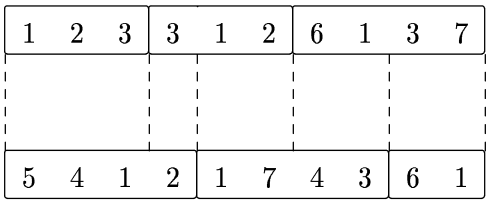

Figure 4: Partition a routing channel into regions

(B). We have four possible implementation pairs (T_1, B_1), (T_1, B_2), (T_2, B_1) and (T_2, B_2). We can determine which of these four implementation pairs are infeasible (i.e. result in a channel region density $\geq d$) and use a case analysis similar to that used above for net span constraints. The cases are:

Case 1: [None are infeasible.] Do nothing.

Case 2: [Exactly one is infeasible.] Suppose, for example, only (T_1, B_2) is infeasible. We need to add B_1 to the forcing list for T_1 and T_2 to the list for B_2. This is similar to case 2 for net span constraints.

Case 3: [Exactly two are infeasible.] This can happen in one of six ways. If the feasible pairs are (T_1, B_2) and (T_2, B_1), then T_1 forces B_2, B_2 forces T_1, T_2 forces B_1 and B_1 forces T_2. The remaining five cases are similar.

Case 4: [Exactly three are infeasible.] There are four ways this can happen. For example, if (T_1, B_1) is the only feasible pair, then implementations T_1 and B_1 must be selected. The remaining three cases are similar.

Case 5: [All four are infeasible.] The 2-PDMIS instance with density constraint d has no feasible solution.

3.2 Stage 2

If following stage 1 we have not determined that the 2-PDMIS instance is infeasible, stage 2 is entered. If no nonempty forcing list remains, all implementations of the modules for which no implementation has been selected result in feasible solutions. When nonempty forcing lists remain, we use the limited branching method of [12] to make the remaining module implementation selections. In this method, we start with a module i whose implementation is yet to be selected. For this module, we try out both implementations, in parallel, following the forcing lists $L[i]$ and $L[m+i]$, respectively. This is equivalent to running *Assign* (L, C, i) and *Assign* $(L, C, m+i)$ in parallel and terminating when either (a) both return with value False or (b) one (or both) return with value True. When (a) occurs, we have an infeasible solution. When (b) occurs, the selections made by the branch that returns True are used. Note that the parallel execution of *Assign* (L, C, i) and *Assign* $(L, C, m+i)$ is actually done via simulation by a single processor; this processor alternates between performing one step of *Assign* (L, C, i) and one of *Assign* $(L, C, m+i)$ and stops when one of the two conditions (a) or (b) occur. In case of (b), we proceed with the next module with unselected implementation.

3.3 Implementation Details

To implement stage 2, we need two copies of the implementation selection array C; one copy for each parallel execution branch. Call these copies C_1 and C_2. Although both are identical at the start of *Assign* (L, C_1, i) and *Assign* (L, C_2, i), C_1 and C_2 may differ later. When the execution of these two branches terminates, we need to set the C_i corresponding to the unselected branch equal to that of the selected branch. This is done efficiently by maintaining two lists Δ_1 and Δ_2 of changes made to C_1 and C_2 since the start of the two branches. Then, if C_1 is selected, we can use Δ_2 to first convert C_2 back to its initial state and then use Δ_1 to convert it from the initial state to C_1. If C_2 is selected, a similar process can be used to convert C_1 to C_2. The time need for this is $|\Delta_1| + |\Delta_2|$ rather than $|C_1| = |C_2| = m$ (as would be the case if we simply copy C_1 to C_2 or C_2 to C_1).

Further, since the forcing lists are shared by two branches, these branches should not modify the forcing lists. Therefore the simulation of *Assign* omits the steps that remove forcing lists. Finally, to efficiently simulate two parallel executions of *Assign*, we need to convert the recursive version of Figure 3 into an iterative version. Our iterative code which simulates the parallel execution of two *Assign* branches employs two queues Q_1 and Q_2. A high level description of the code is given in Figures 5, 6 and 7.

```
Algorithm Satisfy $(L, C_2)$
  /* Test whether $L$ is satisfiable */
  Copy $C_2$ into $C_1$;
  for $i = 1$ to $m$ do
    if $C_1[i] == 0$ then /* $i$ is undecided */
      if $L[i]$ is empty then
        $C_1[i] = C_2[i] = 1$; /* select first implementation */
      else if $L[m+i]$ is empty then
        $C_1[i] = C_2[i] = 2$; /* select second implementation */
      else
        EnQueue $(Q_1, i)$;
        EnQueue $(Q_2, m+i)$; /* $m+i$ represent the 2nd implementation of module $i$ */
        while $Q_1$ not empty and $Q_2$ not empty do
          $a$ = DeQueue $(Q_1)$;
          $b$ = DeQueue $(Q_2)$;
          if $a$ is rejected in $C_1$ and $b$ is rejected in $C_2$ then
            return False;
          else if $a$ is rejected in $C_1$ then
            EnQueue $(Q_1, a)$;
            if not Search $(L, Q_2, C_2, \Delta_2, b)$ then
              return False;
          else if $b$ is rejected in $C_2$ then
            EnQueue $(Q_2, b)$;
            if not Search $(L, Q_1, C_1, \Delta_1, a)$ then
              return False;
          else
            if $a$ is undecided in $C_1$ then
              Add List $L[a]$ into $Q_1$;
              Insert $a$ into $\Delta_1$;
              Mark $a$ selected in $C_1$;
            if $b$ is undecided in $C_2$ then
              Add List $L[b]$ into $Q_2$;
              Insert $b$ into $\Delta_2$;
              Mark $b$ selected in $C_2$;
        end while /* $Q_1$ not empty and $Q_2$ not empty */
        if $Q_1$ is empty then
          Undo $(C_2, \Delta_2, C_1, \Delta_1)$; /* make $C_2 = C_1$ */
        else /* $Q_2$ is empty */
          Undo $(C_1, \Delta_1, C_2, \Delta_2)$; /* make $C_1 = C_2$ */
      end if /* $L[i]$ is empty */
    end if /* module $i$ is undecided */
  end for
  return True;
```

Figure 5: Function *Satisfy*

```
Algorithm Search $(L, Q, C, \Delta, x)$
  /* Select module $x$ and modules in $Q$ and related modules, update list $\Delta$ */
  Mark $x$ selected in $C$;
  Insert $x$ into $\Delta$;
  Add List $L[x]$ into $Q$;
  while $Q$ not empty do
    $y$ = DeQueue ($Q$);
    if y is rejected in $C$ then
      return False;
    else if y is undecided in $C$ then
      Add List $L[y]$ into $Q$;
    else /* y is rejected in $C$ */
      Insert $y$ into $\Delta$;
      Mark $y$ selected in $C$;
  end while
  return True;
```

Figure 6: Function *Search*

```
Algorithm Undo $(C_1, \Delta_1, C_2, \Delta_2)$
  /* make $C_1 = C_2$ by using delta lists */
  for each $x \in \Delta_1$ do
    Mark $x$ undecided in $C_1$;
  for each $x \in \Delta_2$ do
    Mark $x$ selected in $C_1$;
```

Figure 7: Procedure *Undo*

3.4 Time Complexity

To construct the net span constraints' portion of the forcing lists, we must identify the up to four critical modules of each net and establish the forcing constraints for each of the up to four critical module pairs that determine the net span. The critical modules for all nets can be determined in $\Theta(p)$ time by making a left to right sweep of the modules, keeping track, for each net i, of the first and last modules in the top and bottom module row that contain net i. Since all pin locations and module boundaries are integers, the modules can be sorted in left to right order in linear time using bin sort [13]. Each net's contribution to the forcing lists can now be determined in $\Theta(1)$ time. Therefore, representing each $L[i]$ as a chain, the net span constraints' contribution to the $L[i]$s can be determined in $\Theta(p+n) = \Theta(p)$ time.

To construct the portion of $L[i]$ that results from the channel density constraint, we partition the channel into regions by performing a left to right sweep of the modules and using the module end points as region boundaries. The number of channel regions is, therefore, $\Theta(m)$. In our implementation, we scan the channel four times to compute the maximum density of each region for each of the four possible implementations of the module pair that bounds the region. This takes $\Theta(p)$ time. Once we have the densities of each region we can, given the density constraint, construct the forcing lists $L[1..2m]$ in $\Theta(m)$ time. Notice that on succeeding iterations of the binary search for an optimal solution, only the contribution to L from the density constraint may change. The new contribution to L can be determined without recomputing the densities of each region.

The limited branching method of stage 2 uses two queues Q_1 and Q_2. The time needed to add (EnQueue) or delete (DeQueue) an element to/from a queue is $\Theta(1)$ [13]. In each iteration of the **for** loop of Figure 5, the time spent following the successful branch equals that spent following the unsuccessful branch and the time needed to make C_1 and C_2 identical (i.e., the cost of the *Undo* operation) is, asymptotically, no more than the time spent following the successful branch. The time spent following all successful branches is no more than the size of the forcing lists because no forcing list is examined twice. Therefore, the stage 2 time is $O(p)$.

The binary search for the minimum density solution iterates $O(\log n)$ times. Therefore, our algorithm finds an optimal solution to the 2-PDMIS problem in $O(p \log n)$ time.

Comparing our algorithm to that of [1], we note that our algorithm has the potential of identifying infeasible 2-PDMIS instances quite early; that is, during the construction of the forcing lists. Although infeasibility resulting from the critical modules of a single net being too far apart are detected immediately by both algorithms, our algorithm also can quickly detect infeasibility resulting from forced selections during stage 1. The algorithm of [1] does not do this. Because of the calls to *Assign* made during stage 1, the size of the forcing lists to be processed in stage 2 is often significantly reduced. As a result, the limited branching operation is often applied to much smaller data sets than the 2-SAT graph on which the strongly connected component algorithm is applied in [1]. These factors contribute to the observed speedup provided by our algorithm relative to that of [1].

4 Multichannel 2-PDMIS Problem

In the multichannel 2-PDMIS problem, we have $c+1$, $c>1$ rows of modules. Each module has pins on its upper and lower boundaries, each module has two possible implementations, there is a routing channel between every pair of adjacent rows, and net span bounds are provided for every channel [1]. Although Her, Wang and Wong [1] develop a heuristic for the general multichannel PDMIS problem, they do not consider polynomial time algorithms for the multichannel 2-PDMIS problem.

For any fixed channel density tuple $(d_1, d_2, \ldots, d_c)$ for the c routing channels, we can develop the forcing lists in $O(p)$ time, where p is the total number of pins. These lists are developed using ideas similar to those used in Section 3. Then, using the limited branching method of Section 3, we can determine, in $O(p)$ time, whether it is possible to select module implementations so that the channel densities do not exceed $(d_1, d_2, \ldots, d_c)$ and so that the net span bounds are satisfied. Thus, the method of Section 3 is easily extended to obtain an $O(p)$ feasibility test for $(d_1, d_2, \ldots, d_c)$. Since there are $O(n^c)$ possible density vectors (n is the number of nets), the c channel 2-PDMIS problem can be solved by trying out all $O(n^c)$ tuples in $O(pn^c)$ time.

We can reduce this time to $O(pn^{c-1})$ as follows. When $c=2$, first determine the least y such that $(\frac{n}{2}, y)$ is a feasible channel density tuple. This is done using a binary search on d_2 and takes $O(\log n)$ feasibility tests, each test taking $O(p)$ time. We can ignore tuples (d_1, d_2) with $d_1 < \frac{n}{2}$ and $d_2 < y$ because these tuples are infeasible, and we can ignore tuples (d_1, d_2) with $d_1 > \frac{n}{2}$ and $d_2 \geq y$ because these are inferior to $(\frac{n}{2}, y)$. Therefore, the search for a better tuple than $(\frac{n}{2}, y)$ may be limited to the regions $d_1 < \frac{n}{2}$ and $d_2 \geq y$, and $d_1 > \frac{n}{2}$ and $d_2 < y$. These two regions (Figure 8) may now be searched recursively. For example, to find the best tuple in the region $d_1 < \frac{n}{2}$ and $d_2 \geq y$, find the least z such that $(\frac{n}{4}, z)$ is feasible. Now search the two regions $d_1 < \frac{n}{4}$ and $d_2 > z$, and $d_1 > \frac{n}{4}$ and $d_2 < z$. for a better tuple than $(\frac{n}{4}, z)$.

The worst-case number of feasibility tests for the above search strategy is given by the recurrence

$$N(n) = 2N(\frac{n}{2}) + \log n, \quad n \geq 2$$

and $N(1) = 1$. The solution to this recurrence is $N(n) = O(n)$. Since each feasibility test takes $O(p)$ time, the 2-channel 2-PDMIS problem can be solved in $O(pn)$ time.

By doing an exhaustive search on the densities of $c-2$ channels and using the above technique for the remaining 2 channels (i.e., for each choice of densities for $c-2$ channels, find the overall best choice for the c channels as above), we can solve the c-channel 2-PDMIS problem in $O(p \cdot n^{c-2} \cdot n) = O(pn^{c-1})$ time.

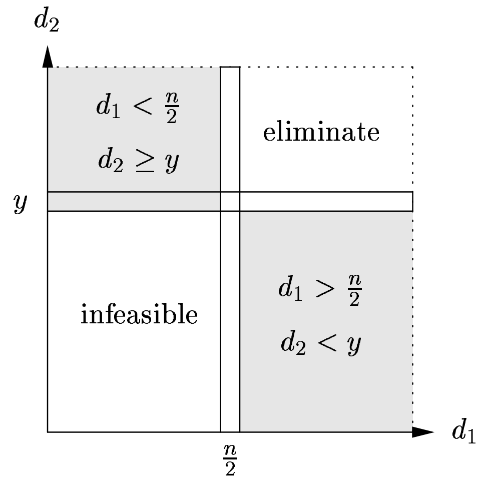

Figure 8: The two regions to be searched recursively after the binary search

5 Experimental Results

We implemented our algorithm as well as that of Her, Wang and Wong [1] in C and measured the run time performance of the two algorithms on a SUN SPARCstation 5. Our first data set consists of benchmark channels used in [1]. We partitioned the top row and bottom row of the channel into intervals and consider these intervals as "modules", and assume each module has two implementations. Table 1 gives the characteristics of these circuits as well as the time, in seconds, taken by the two algorithms. The optimal densities given in Table 1 differ from those reported in [1] because the partitioning of the top and bottom rows of pins used by us is different from that used in [1]. The speedup provided by our algorithm ranges from 1.67 to 2.20. Our second data set consists of circuits designed to minimize the size of the forcing lists constructed in stage 1. The characteristics of these circuits as well as the performance of the two algorithms on these two circuits are given in Table 2. Our algorithm is 9 to 11 times as fast on these circuits.

Table 1: Running time for benchmark channels

Channel	n	m	p	Optimal density	Time/Second		Speedup
					[1]	Our	
ex1	21	19	74	12	0.0022	0.0010	2.20
ex3a	44	36	158	14	0.0046	0.0023	2.00
ex3b	47	24	158	16	0.0035	0.0021	1.67
ex3c	54	23	178	18	0.0039	0.0023	1.70
ex4b	54	28	192	17	0.0045	0.0024	1.88
ex5	64	40	190	18	0.0042	0.0025	1.68

Table 2: Running time for generated channels

Channel	n	m	p	Time/Second		Speedup
				[1]	Our	
w32x32	64	66	192	0.0425	0.0046	9.24
w64x64	128	130	384	0.0999	0.0105	9.51
w128x128	256	258	768	0.2275	0.0225	10.11
w256x256	512	514	1536	0.5130	0.0487	10.53
w512x512	1024	1026	3072	1.1755	0.1066	11.03
w1024x1024	2048	2050	6144	2.6150	0.2309	11.33
w2048x2048	4096	4098	12288	5.6700	0.4886	11.60
w4096x4096	8192	8194	24576	12.0500	1.0280	11.72
w8192x8192	16384	16386	49152	24.8800	2.1260	11.70

6 Conclusion

We have developed an $O(p \log n)$ time algorithm for the single channel 2-PDMIS problem and an $O(pn^{c-1})$ time algorithm for the c, $c > 1$, channel 2-PDMIS problem. Experiments indicate that our single channel algorithm is substantially faster than the single channel algorithm of [1]. The heuristic proposed in [1] for the k-PDMIS problem, $k > 2$, uses the algorithm for the 2-PDMIS problem. By using our 2-PDMIS algorithm, the k-PDMIS heuristic of [1] will also run faster.

References

[1] T. W. Her, Ting-Chi Wang, and D. F. Wong. Performance-driven channel pin assignment algorithms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(7):849–857, July 1995.

[2] Takeshi Yoshimura and Ernest S. Kuh. Efficient algorithms for channel routing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-1(1):25–35, January 1982.

[3] H. Kobayashi and C. E. Drozd. Efficient algorithms for routing interchangeable terminals. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-4(3):204–207, 1985.

[4] Li-Shin Lin and Sartaj Sahni. Maximum alignment of interchangeable terminals. *IEEE Transactions on Computers*, 37(10):1166–1177, October 1988.

[5] Sartaj Sahni and San-Yuan Wu. Two NP-hard interchangeable terminal problems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 7(4):467–472, April 1988.

[6] Spyros Tragoudas and Ioannis G. Tollis. River routing and density minimization for channels with interchangeable terminals. *Integration, the VLSI Journal*, 15:151–178, 1993.

[7] T. W. Her and D. F. Wong. Module implementation selection and its application to transistor placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(6):645–651, June 1997.

[8] Yang Cai and D. F. Wong. On shifting blocks and terminals to minimize channel density. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(2):178–186, February 1994.

[9] C. Y. Hou and C. Y. Chen. A pin permutation algorithm for improving over-the-cell channel routing. In *Proceedings of the 29th Design Automation Conference*, pages 594–599, Anaheim, California, 1992.

[10] T. W. Her and D. F. Wong. On over-the-cell channel routing with cell orientations consideration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(6):766–772, June 1995.

[11] C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization – Algorithms and Complexity.* Prentics-Hall, Englewood Cliffs, New Jersey, 1982.

[12] S. Even, A. Itai, and A. Shamir. On the complexity of timetable and multicommodity flow problems. *SIAM Journal on Computing*, 5(4):691–703, December 1976.

[13] Sartaj Sahni. *Data Structures, Algorithms, and Applications in C++.* McGraw Hill, Boston, Massachusetts, 1998.