ISPD2020: Hill Climbing with Trees

Detail Placement for Large Windows

Mohammad Khasawneh and Patrick H. Madden

SUNY Binghamton Computer Science Department

ACM Digital Library

YouTube Video

The VLSI Physical Design Flow

- Global Placement
  - Usually analytic, recursive bisection, or annealing
- Legalization
  - Tetris-derived, matching, flow based
- Detail Placement
  - Predominantly small-window branch-and bound
  - Some use of mixed integer programming

Detail Placement Window Sizes

Optimization windows for detail placement are typically small; large windows result in unacceptable run times, using conventional methods.

Cells	**Number of Permutations**
4	24
5	120
6	720
16?	2.092279e+13
100?	9.332622e+157

Expanding the Window Size

The focus of this work is on expanding

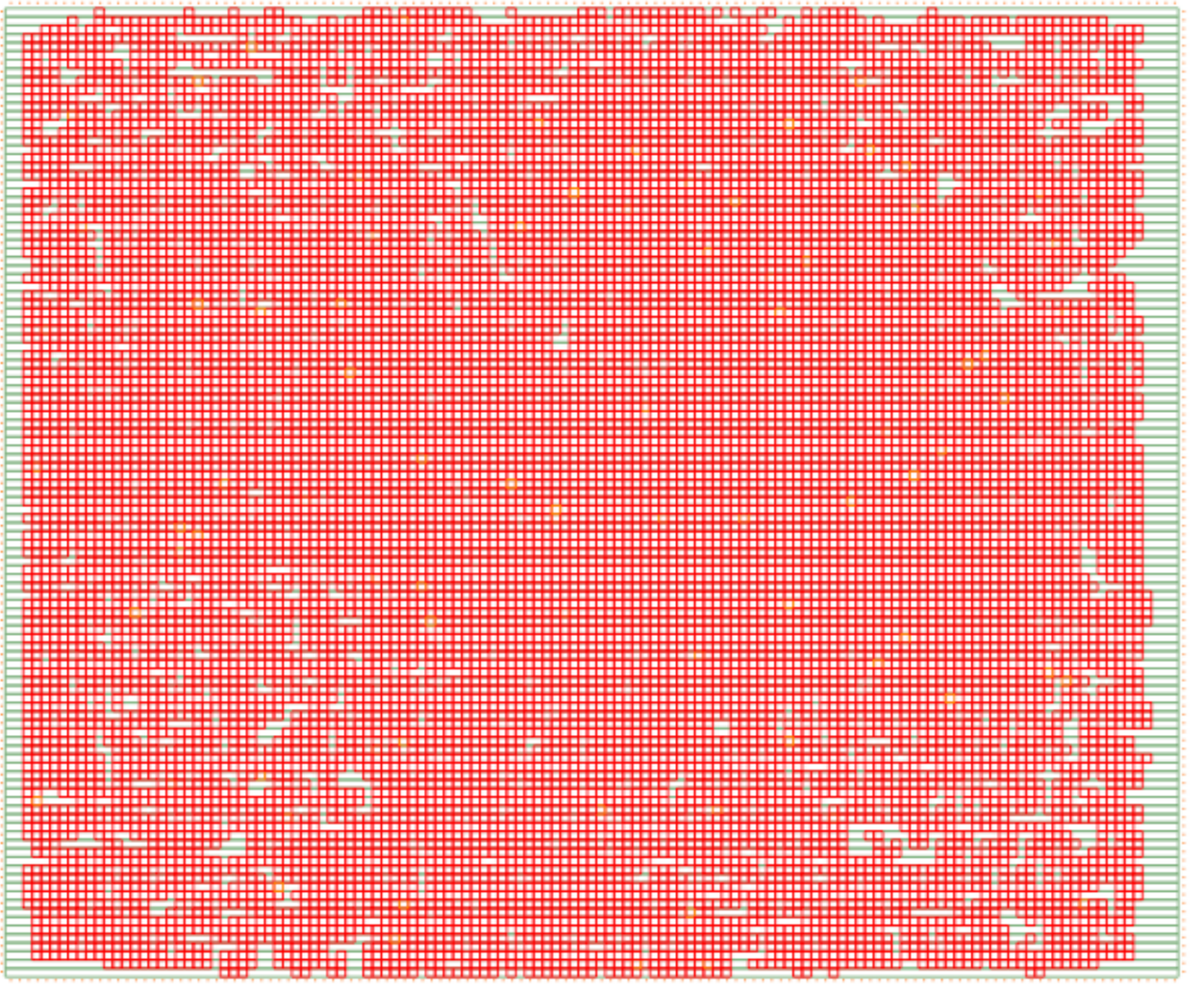
Most sliding window optimizations are six to eight cells. Larger windows are too expensive computationally, with traditional methods -- this results in optimization opportunities being missed.

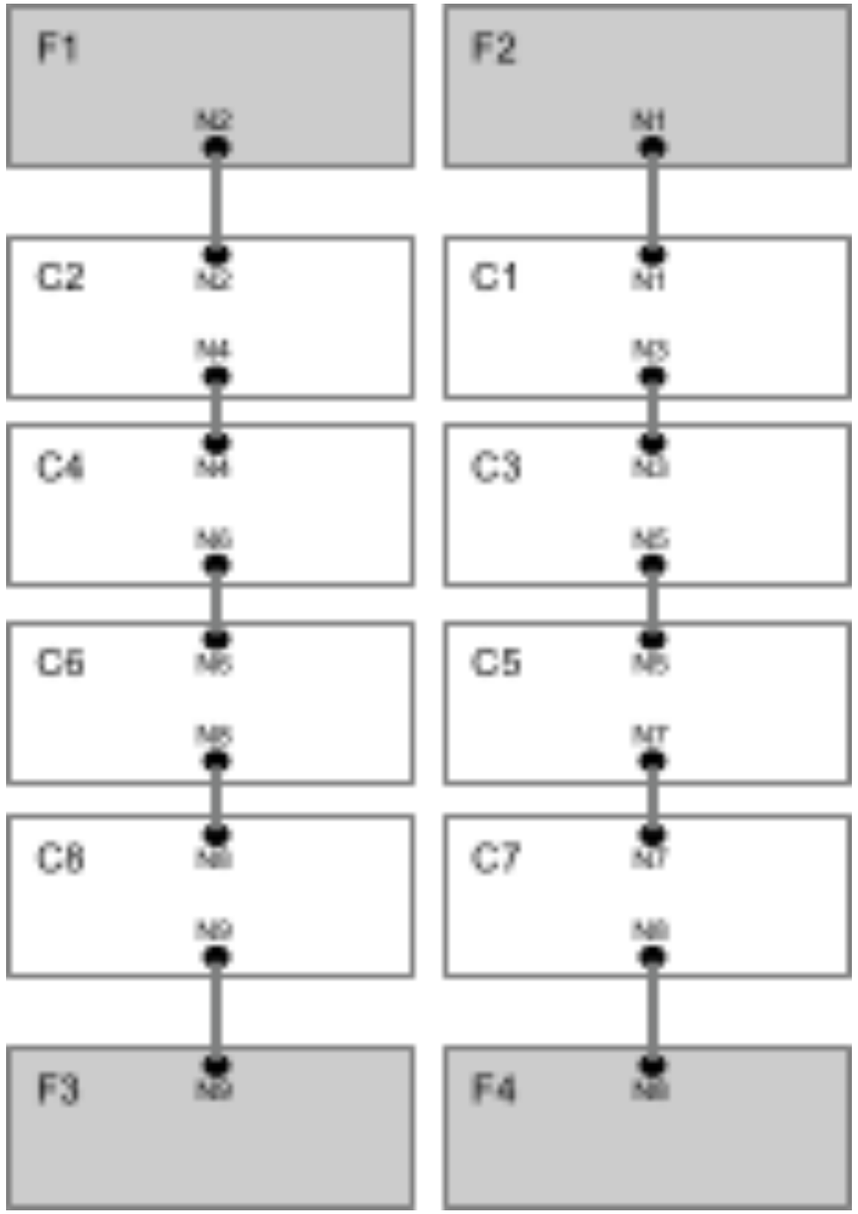
Modern designs have tens or hundreds of thousands of standard cells. Detailed placement involves running a small optimization window across the entire design, working on a handful of cells at a time.

Much of the motivation for this work is from close inspection of the PEKO benchmarks; placement tools were often 50% to 200% away from optimal, despite the placements being "good" globally. There was obvious quality loss at the local level, and many opportunities to reduce wire length were missed.

Peko01 Optimal placement: Wire length 814368

Placement by NTUPlace3: Wire length 1814304



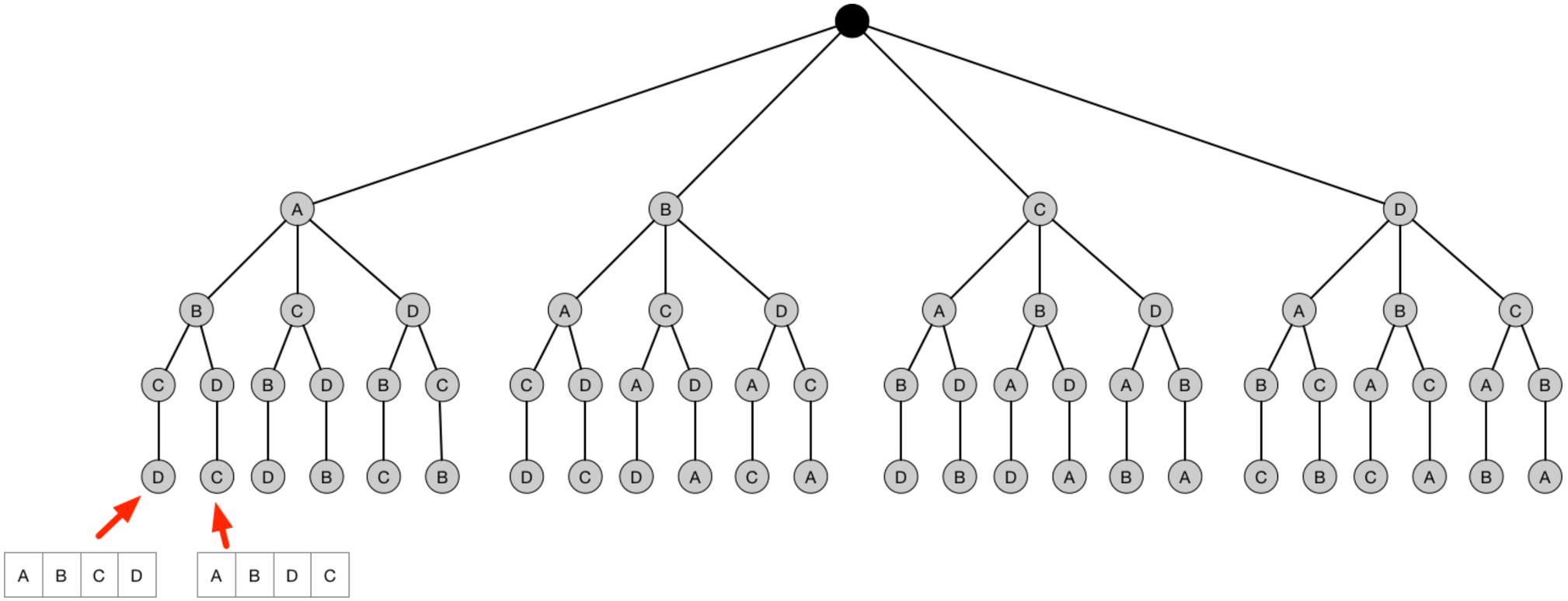


For detail placement, the window size can have a huge impact on how much improvement is possible. In this example, there's an obvious better solution -- but unless the window contains all eight movable cells, we can't reach it.

A window of size two, or four, has no incentive for moving towards the optimal configuration. "Twists" of nets were commonplace in the PEKO benchmark results.

Traditional Methods -- Branch and Bound

A simple problem with four cells has twentyfour configurations; these are commonly explored in depth-first manner, with the potential to prune the solution space when incremental wire lengths exceed a known complete solution.



Recommended Background on Traditional Methods

Optimal Partitioners and End-Case Placers for Standard-Cell Layout, Caldwell, Kahng, and Markov, ISPD 1999

A good reference covering enumeration, gray codes, and branch-and-bound methods.

Newer Methods: Integer Linear Programming

Optimal Placement by Branch-and-Price, Ramachandaran et. al., ASPDAC 2005

Window sizes up to six-by-six, each taking about a half hour. Optimization restricted to square cells (PEKO-style).

Newer Methods: Integer Linear Programming

Mixed Integer Programming Models for Detailed Placement, Li & Koh, ISPD 2012

Smaller two-by-ten windows, in a few seconds each. Support for white space, and cells of different size.

Hill Climbing with Trees

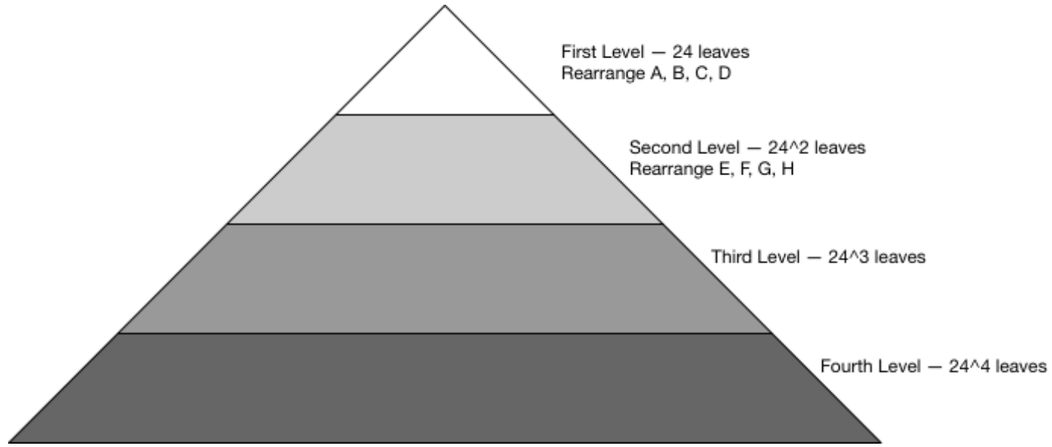
Our new approach is influenced by the hill climbing work of Kernighan & Lin, and by the data structure efficiency of Fiduccia & Mattheyses. The key hill climbing insight is that it's essential for an optimization approach to be able to climb out of a local minima, and pass through parts of the solution space that have higher cost, to reach a better minima elsewhere.

KL and FM push through the solution space by locking vertices, and making moves that will degrade solution quality. In our approach, we maintain multiple tree branches of the search tree -- following them, even if they're higher cost than a known "good" branch.

A Small Placement Example

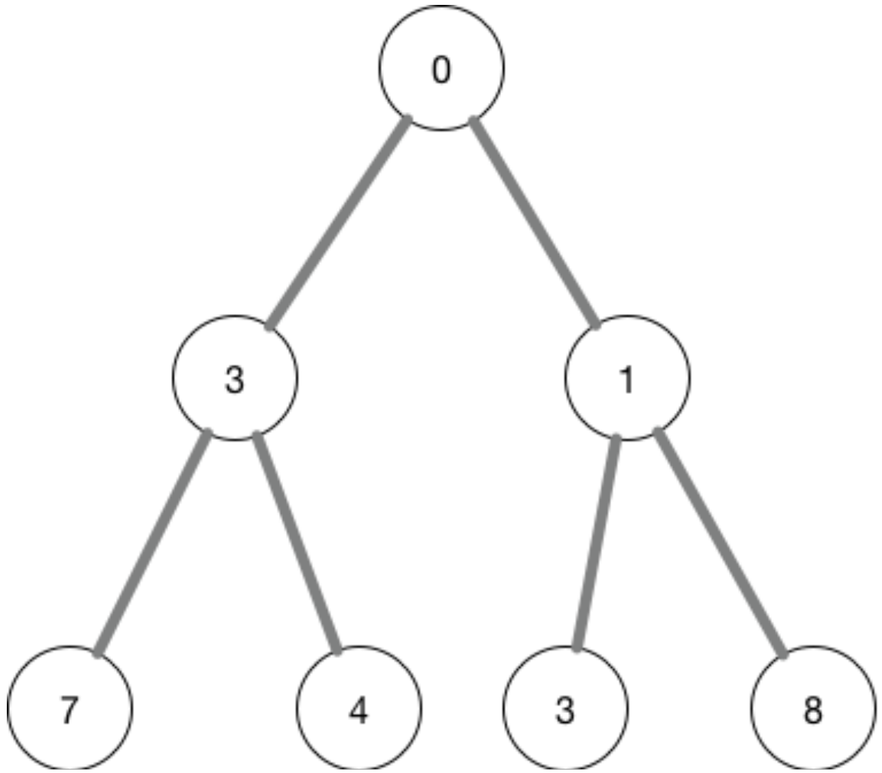
If we have a 4x4 grid of cells, and restrict them to their original rows, there are 24 permutations for the first row, 24 for the second, and so on. Collectively, there are 24^4 different arrangements.

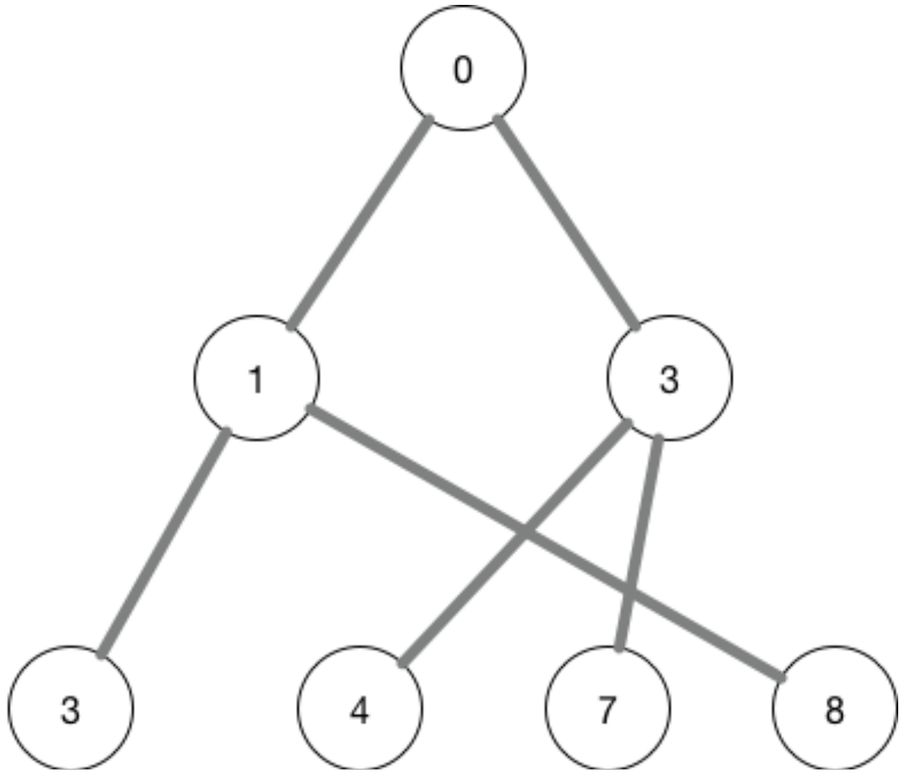
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P



Arranging the Search Tree

We'll explore the solution space in tree-like fashion -- but rearranging the tree so that low cost branches are to the "left" helps with understanding the key ideas. For the following slides -- assume the trees are arranged as shown.

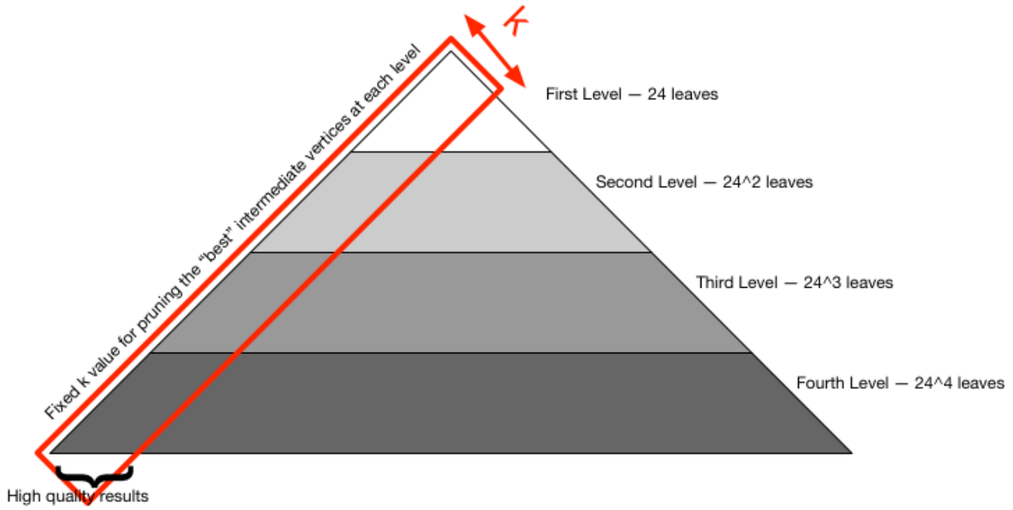




A Thought Experiment....

Suppose we build a brute-force tree, covering every possible permutation of each row -- the "best" solution would be the lower left-most corner. If we trace back up the tree from that solution -- we can expect that the parents would also be towards the left.

Not necessarily the absolute left -- but likely within a relatively few number of spots away. We prune the solution space to *k* vertices at each level -- if the nodes of an optimal solution are within *k*, our heuristic can find it. In practice, we find very good solutions.



Hill Climbing with Trees Pseudocode

In our implementation, we precompute many permutations partial bounding boxes for nets. The core of our approach resembles breadth-first search: possible permutations for row m are combined in an all-pairs manner with leaves from row $m-1$.

We prune, keeping the best k solutions at each level; this prevents an explosion of the solution space. Wire lengths (and precise positions of each cell) can be determined by tracing back up the tree.

```
hcwt_dp(list cells, int delta, int k)
{
  levels = sort_and_group_cells(cells);
  generate_permutations(levels, delta);
  generate_partial_boundingboxes();

  heap prior_heap = new_heap(1);
  heap_insert(prior_heap, root_node);

  foreach (L in levels)
  {
    heap current_heap = new_heap(k);
    foreach (node parent in prior_heap)
    {
      foreach (permutation P for L)
      {
        node nn = combine(P, parent);

        // Heaps have a maximum size
        // of k elements; high cost
        // solutions are pruned
        heap_insert(current_heap, nn);
      }
    }
    prior_heap = current_heap;
  }

  // The best node in the last heap
  node best = heap_best(prior_heap);
  if (best->cost < 0)
  {
    trace_back_and_deploy(best);
  }
}
```

Comparability -- A Key Element of the Approach

In our approach, we place *all* the cells in a row simultaneously. Because each row has the same set of cells, and the same connected nets, wire lengths can be compared meaningfully.

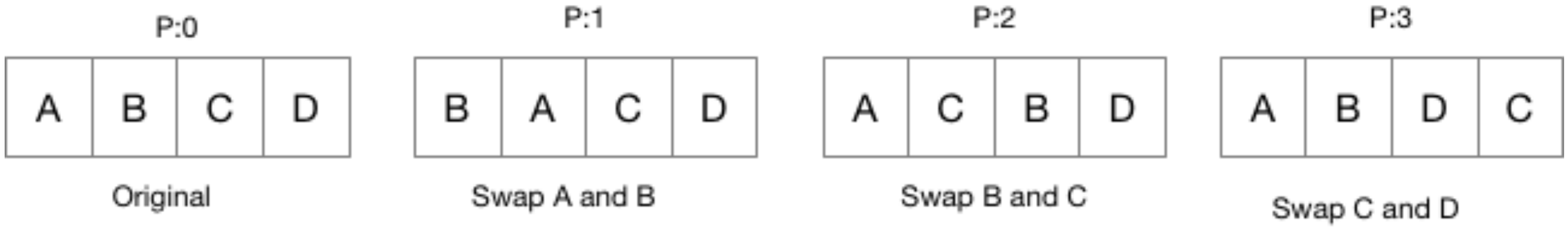
With branch and bound, each tree vertex represents different sets of cells and different nets. No meaningful comparison is available.

We can prune to *k* vertices **because** we place rows of cells as a group.

Partial Permutations

In practice, the incoming placement we perform detail placement on is good quality; if the cells are initially ABCDEFG, it's extremely unlikely that an optimal placement is GFEDCBA.

Rather than considering every possible permutation, we restrict consideration to permutations that are relatively similar to the original positions. We compare the index order of each cell, and assign a cost for changes as a Delta value. A Delta of 2 is a pair-wise swap.



An Example

In the next slide, we show the tree created by our approach, on a set of cells extracted from the benchmark IBM01. The window is roughly six standard cells wide, by twelve rows -- the number of cells in each row varies, because the cell widths vary.

We expand out the tree in brute-force manner -- pruning each level of the tree to sixteen nodes. Incremental wire lengths are used to order the cells; this is were the thought experiment plays a critical role. We can expect the best solutions to be clustered towards the left -- but the best solution may not always be the absolute leftmost.

By keeping multiple branches alive (as compared to a simple greedy heuristic), our approach explores more of the solution space, looking in the areas that are most likely to have a high quality solution.

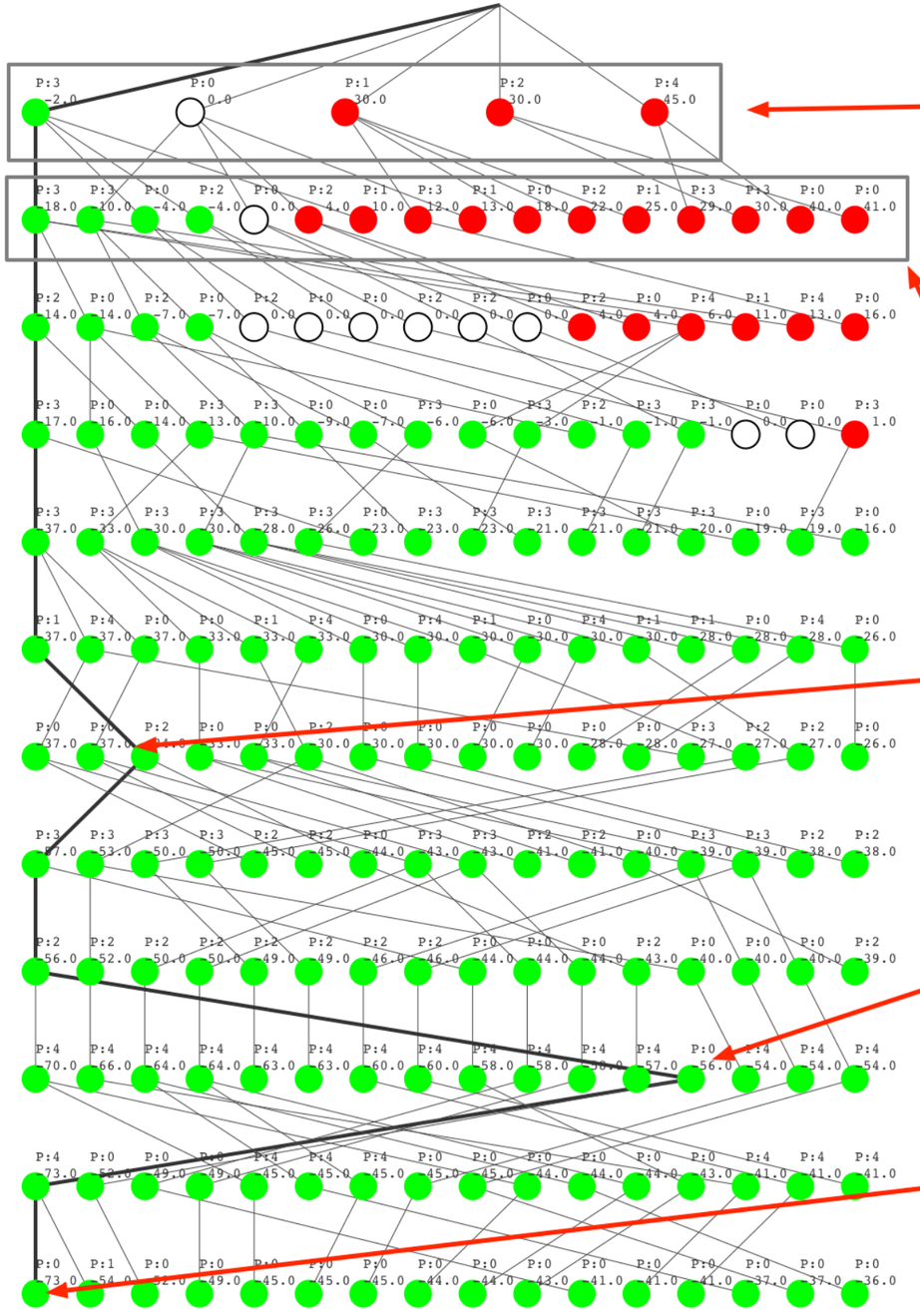
```
hcwt_dp(list cells, int delta, int k)
{
  levels = sort_and_group_cells(cells);
  generate_permutations(levels, delta);
  generate_partial_boundingboxes();

  heap prior_heap = new_heap(1);
  heap_insert(prior_heap, root_node);

  foreach (L in levels)
  {
    heap current_heap = new_heap(k);
    foreach (node parent in prior_heap)
    {
      foreach (permutation P for L)
      {
        node nn = combine(P, parent);

        // Heaps have a maximum size
        // of k elements; high cost
        // solutions are pruned
        heap_insert(current_heap, nn);
      }
    }
    prior_heap = current_heap;
  }

  // The best node in the last heap
  node best = heap_best(prior_heap);
  if (best->cost < 0)
  {
    trace_back_and_deploy(best);
  }
}
```



The first level contains five cells; there are five permutations with δ=2; the original configuration, and four nearest-neighbor swaps.
Permutation three swaps the third and fourth cells, obtaining a net gain of two units (ignoring the length of nets connected to cells in later levels).

The second level contains four cells, and there are four permutations. Each permutation is combined with the prior level — resulting in twenty combinations, of which only the lowest cost sixteen are kept; the pruning prevents exponential tree growth.

In this level, the third best solution is preserved; a simple greedy algorithm would normally prune this option. The configuration here returns to being best one step later.

In this level, the thirteenth best solution is preserved; as happened earlier, the configuration returns to being one of the best.

In the last row, permutation zero is the best in almost all branches; cells in this level have numerous connections to other cells, and are less likely to move. The best solution found reduces wire length by 73 units; in many, but not all cases, a greedy choice was made.

Experimental Results

In the first set of experiments, we utilize only our hill climbing method to optimize results of NTUPlace3 on the PEKO benchmarks, immediately after legalization. In thirteen of the eighteen benchmarks, we obtain higher wire length reductions than conventional methods, with low run times.

When combined with the detail placement techniques of NTUPlace3, we obtain a further 3% gain.

	Legal	NTUPlace3 DP		Ours		
PEKO	HPWL	HPWL	%	HPWL	%	t(s)
Peko01	1975456	**1814304**	**8.16**	1838464	6.93	15.1
Peko02	3071136	2913696	5.13	**2877760**	**6.30**	25.45
Peko03	3561952	3372608	5.32	**3310592**	**7.06**	30.08
Peko04	3732352	3527872	5.48	**3460160**	**7.29**	34.52
Peko05	4111232	3945472	4.03	**3859040**	**6.13**	39.14
Peko06	4109440	**3771072**	**8.23**	3777152	8.09	40.17
Peko07	6077760	**5591200**	**8.01**	5641888	7.17	55.03
Peko08	7015776	6655936	5.13	**6524544**	**7.00**	66.53
Peko09	7296128	6783456	7.03	**6697440**	**8.21**	71.94
Peko10	9255936	8636224	6.70	**8530624**	**7.84**	95.18
Peko11	9458784	8834496	6.60	**8708416**	**7.93**	92.34
Peko12	9642048	9010848	6.55	**8939200**	**7.29**	99.67
Peko13	11373088	10674464	6.14	**10463424**	**8.00**	116.91
Peko14	19358304	17712672	8.50	**17650816**	**8.82**	182.22
Peko15	25146560	**23028000**	**8.42**	23127104	8.03	228.64
Peko16	23939712	22277824	6.94	**21999360**	**8.11**	252.43
Peko17	23324896	21944640	5.92	**21441856**	**8.07**	274.93
Peko18	28354592	**25889568**	**8.69**	26031264	8.19	266.72
		Average:	6.72	Average:	**7.58**	

Experimental Results -- *k*

When k is 1, our approach resembles a simple greedy heuristic; if k is infinity, we would be performing exponential time brute-force search.

Two important points to note. First, with modest values of k, we can obtain from two to four times as much improvement, compared to a greedy heuristic -- and this *after* NTUplace3 has exhausted the wire length gains available with their detail placement flow. Second, there are diminishing gains as k increases -- while it's impossible to determine the optimal placement gains, it seems likely that there is much left of the table.

	Peko18				IBM18			
	horiz.		vert.		horiz.		vert.	
k	gain	t(s)	gain	t(s)	gain	t(s)	gain	t(s)
1	108960	0.46	87104	1.25	11477	1.1	8613	1.13
2	142144	0.47	109920	1.3	14397	1.14	9986	1.17
4	174720	0.55	126400	1.38	18882	1.23	13574	1.25
8	186816	0.68	132000	1.51	23410	1.37	16831	1.4
16	191360	0.89	133440	1.74	26734	1.65	19770	1.69
32	192864	1.29	133664	2.23	29683	2.2	21882	2.25
64	193184	2.02	133760	2.94	32325	3.23	23438	3.27
128	193376	3.27	133760	4.26	34121	5.24	24622	5.26
256	193408	5.64	133760	6.52	35778	9.05	25380	9.08
512	193408	9.62	133760	9.55	37114	16.36	26093	16.24
1024	193440	17.16	133760	15.4	37926	30.49	26387	30.26

Experimental Results -- *Delta*

We use partial permutations, restricting how much any row can change, with a parameter $Delta$. The results may be a bit counterintuitive -- larger values of $Delta$ degrade results. What appears to be happening is increased numbers of permutations causes a massive increase in the size of the solution space -- with our heuristic seeking to find the needles in much bigger haystacks.

From prior experiments using branch-and-price ILP, we observed that major changes to placements were uncommon. Having a high $Delta$ value increases the size of the solution space, increases run times, and degrades results -- small values work best!

	Peko18				IBM18			
δ	horiz. gain	t(s)	vert. gain	t(s) t(s)	horiz. gain	t(s)	vert. gain	t(s) t(s)
2	201440	18.87	201312	18.47	21853	22.52	16408	23.41
4	220288	74.65	230144	75.67	12181	77.27	8520	84.46
6	203488	209.89	220160	216.09	7908	211.03	5820	234.98
8	188064	480.15	208064	506.75	6143	473.28	4323	539.46
10	182016	871.16	201568	950.41	5116	828.08	4073	992.45

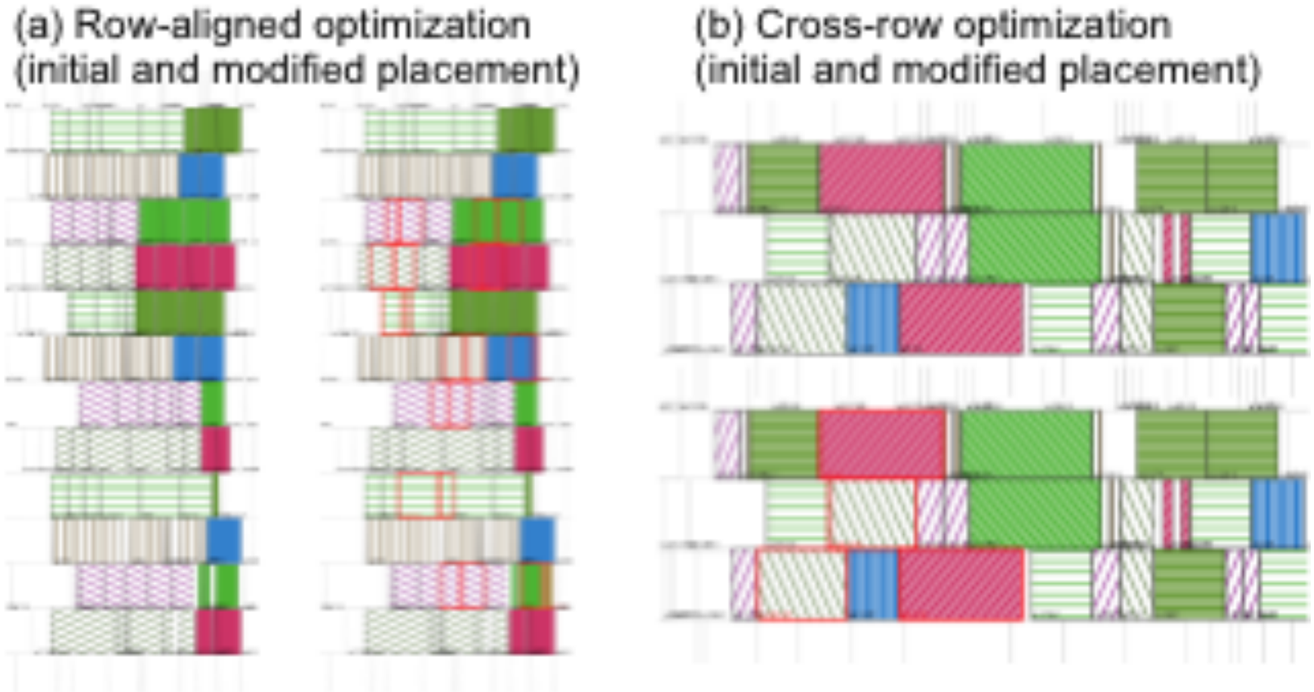
Experimental Results -- Window Sizes

We swept a variety of window sizes, covering completed and optimized placements with a single pass. Run times were modest in all cases; the amount of gain obtain varies, but in practice, windows of 4x8 or 6x8 work well. These windows are substantially larger than what has been typical practice.

	Peko18					IBM18				
	horiz.		vert.			horiz.		vert.	
size	gain	t(s)	gain	t(s)	size	gain	t(s)	gain	t(s)
2x2	127328	0.51	120800	0.5	2x2	25010	0.71	34093	0.74
2x3	152544	1.25	124128	0.52	2x3	25586	0.98	34859	0.94
2x4	145920	1.2	125792	0.57	2x4	25885	1.4	35501	1.44
2x8	147456	1.75	129888	1.34	2x8	24770	3.42	36240	3.16
2x12	152704	3.29	129696	2.05	2x12	24458	4.58	35570	3.76
2x16	150368	3.98	131104	2.43	2x16	23575	5.17	33950	4.02
2x20	151808	4.49	130944	2.67	2x20	22312	5.57	31711	4.18
4x2	131936	1.21	181024	0.59	4x2	27016	1.44	36895	1.7
4x3	137024	1.5	186048	0.84	4x3	26644	2.56	37764	2.78
4x4	140736	1.8	187904	1.52	4x4	27162	3.55	38017	3.89
4x8	133760	4.26	193376	3.31	4x8	24622	5.3	34121	5.24
4x12	137760	5.24	193504	3.73	4x12	19861	5.93	28759	5.65
4x16	134368	5.5	194848	3.95	4x16	16346	6.29	23624	5.89
4x20	132928	5.76	193024	4.08	4x20	12788	6.54	19583	6.04
6x2	196384	1.48	189856	1.44	6x2	27866	2.57	37046	2.94
6x3	199520	2.23	194176	2.18	6x3	28905	3.98	36989	4.35
6x4	200608	3.75	196512	3.88	6x4	27625	4.79	35600	4.96
6x8	200416	5.35	201184	5.27	6x8	19592	6.02	24923	5.87
6x12	199520	5.82	199840	5.68	6x12	12405	6.54	17351	6.2
6x16	195520	6.05	198528	5.91	6x16	8984	6.86	12100	6.44
6x20	190560	6.19	194656	6.14	6x20	6233	7.09	9213	6.64
8x2	180032	1.79	164896	1.8	8x2	25682	3.61	37049	4.08
8x3	182144	3.8	168608	3.65	8x3	26015	4.81	35474	5.21
8x4	183648	4.58	170304	4.4	8x4	23490	5.43	31700	5.63
8x8	183616	5.7	172000	5.28	8x8	12739	6.46	17730	6.39
8x12	177344	6.17	164256	5.56	8x12	6743	6.97	10677	6.79
8x16	168512	6.31	159328	5.72	8x16	3543	7.34	7528	7.09
8x20	161984	6.47	150144	5.81	8x20	2399	7.63	4954	7.37

An Example

Here's a small example of detailed placement problems extracted from IBM01. Our approach can operate on a row-by-row basis, or across rows (permuting cells with equal width). We highlight the cells that have been moved, obtaining a wire length reduction that was missed by the NTUPlace3 detail placement code.



Hill Climbing With Trees - Summary

- Breadth-first tree construction
- Place *sets* of cells a row at a time, limiting the permutations
- We achieve comparability between tree branches because the cells are placed as sets. This enables pruning to *k* branches.
- Hill climbing by keeping *multiple* tree branches; like KL and FM partitioning heuristics, we explore higher-cost configurations.
- In some sense, we're not just hill climbing, but going over the mountain, looking for good solutions in between the ups and downs.
- Run times are near linear, rather than exponential.
- Technique seems complimentary to conventional methods; there are benefits to using both.

Like to Know More?

More details in the paper!

Or, contact the research group at optimal.cs.binghamton.edu

Thanks!