Dynamic Programming for the Masses

Patrick H. Madden
SUNY Binghamton CSD
pmadden@acm.org

Overview

- Motivation
- NP Complete Problems
- Dynamic Programming
  - Integer 0-1 Knapsack Problem
  - Removing the Integer Constraint
  - Pareto Optimality in General
- Making DP More Accessible
  - Enumeration sieve -- a hybrid of enumeration, dynamic programming, and greedy approaches

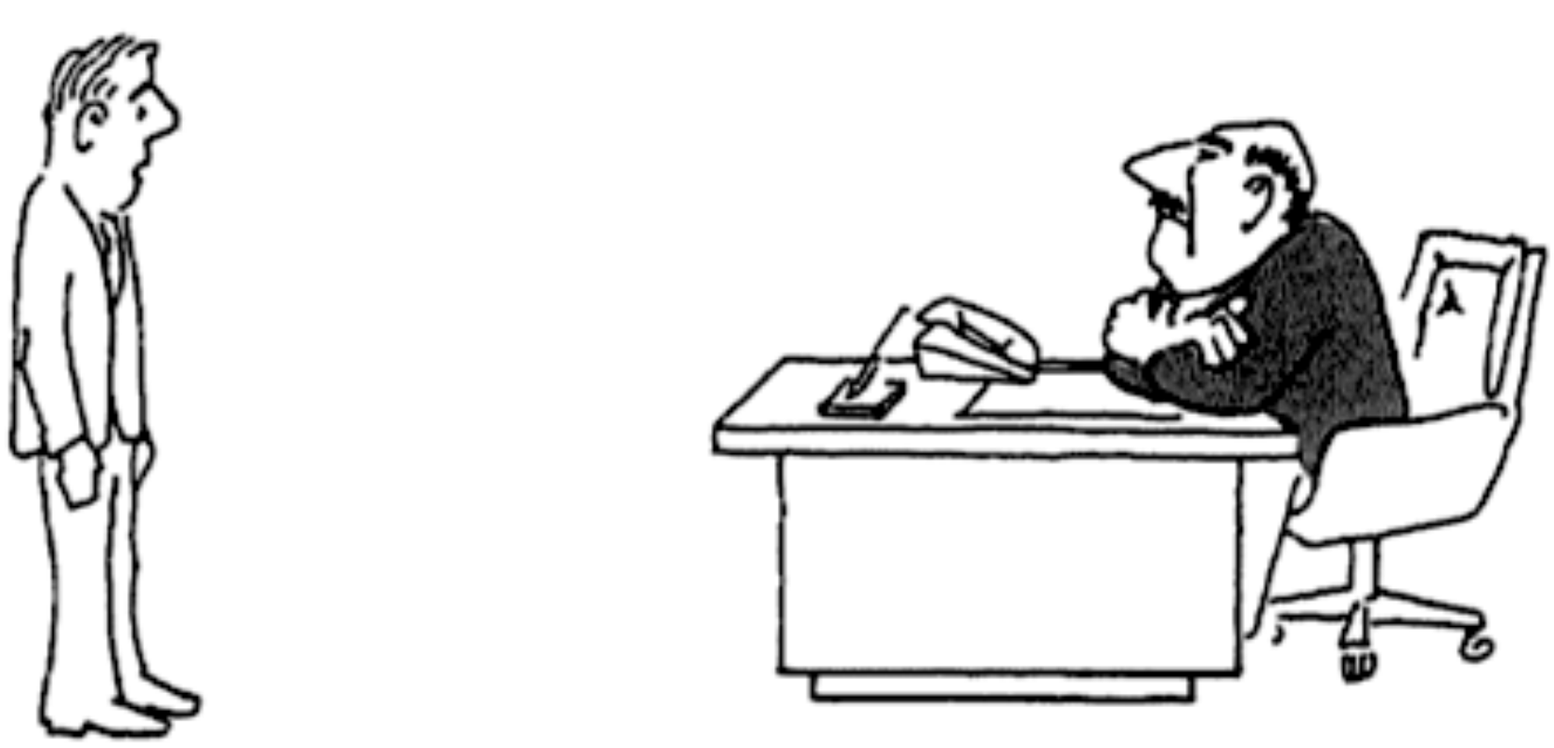
Motivation

In the chip design world, many of the problems are combinatorial in nature, NP-Complete, and problem sizes have been scaling with Moore's Law. This results in a lot of scrambling to find good heuristics.

Dynamic Programming is one of many techniques to attack difficult problems. It can be conceptually challenging to apply. The work discussed here is intended to make it more widely accessible.

Hard Combinatorial Problems

Image from Garey & Johnson, Computers and Intractability



"I can't find an efficient algorithm, I guess I'm just too dumb."

Hard Combinatorial Problems

Image from Garey & Johnson, Computers and Intractability

"I can't find an efficient algorithm, because no such algorithm is possible!"

Hard Combinatorial Problems

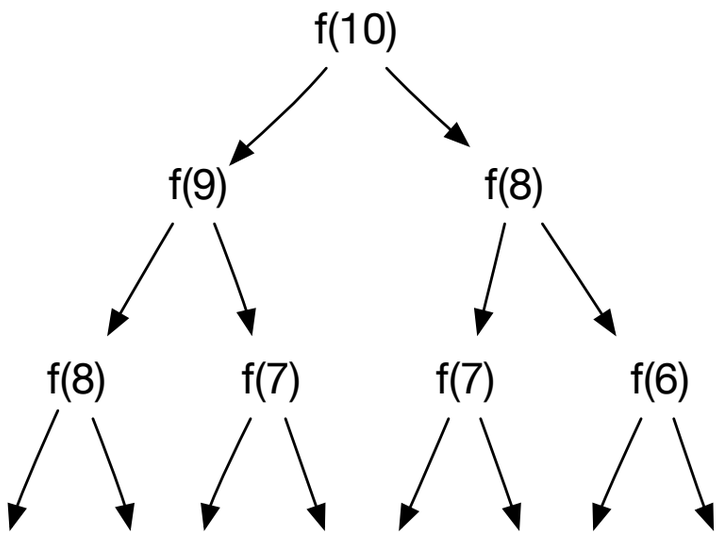
Image from Garey & Johnson, Computers and Intractability

"I can't find an efficient algorithm, but neither can all these famous people."

Making Things Tractable

A Simple Example to illustrate Dynamic Programming

Fibonacci f(n) = f(n - 1) + f(n-2)



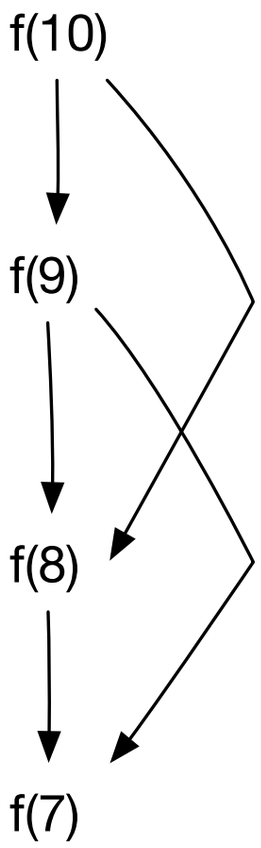
Key observation:
We repeatedly calculate the same value. There are many opportunities to prune the tree….

There can be an exponential number of leaves…. this grows quickly!

Making Things Tractable

```
int fib(int n)
{
    if ((n == 0) || (n == 1))
      return 1;
    return fib(n - 1) + fib(n - 2);
}
```

The brute-force tree is embedded in an array



Simple improvement with Dynamic Programming

Keep track of what is computed, and solve things only once!

We go from exponential time to linear! A massive savings!

Fibonacci - Dynamic Programming

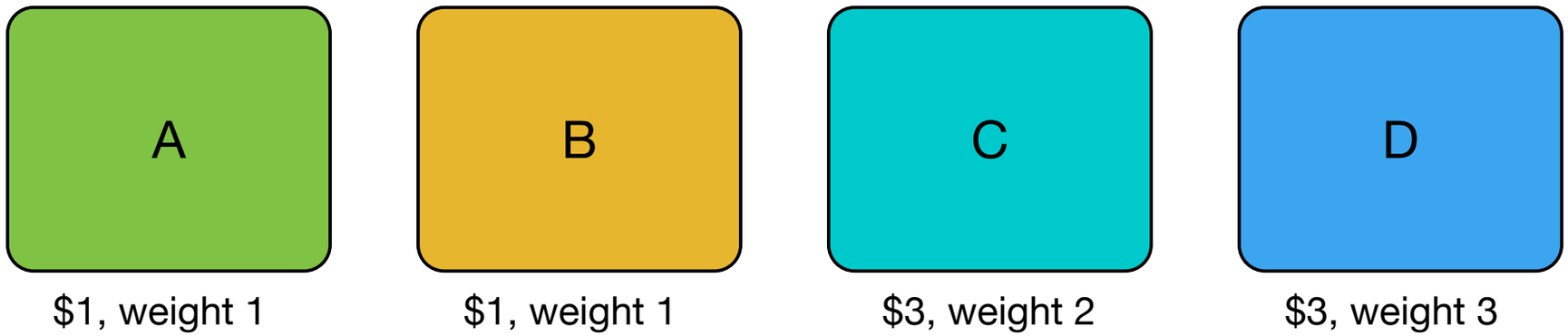
```
static int dp_table[MAX]; // Initialize to -1
int fib_dp(int n)
{
    if ((n == 0) || (n == 1))
      return 1;
    if (dp_table[n] == -1)
      dp_table[n] = fib_dp(n - 1) + fib_dp(n - 2);
    return dp_table[n];
}
```

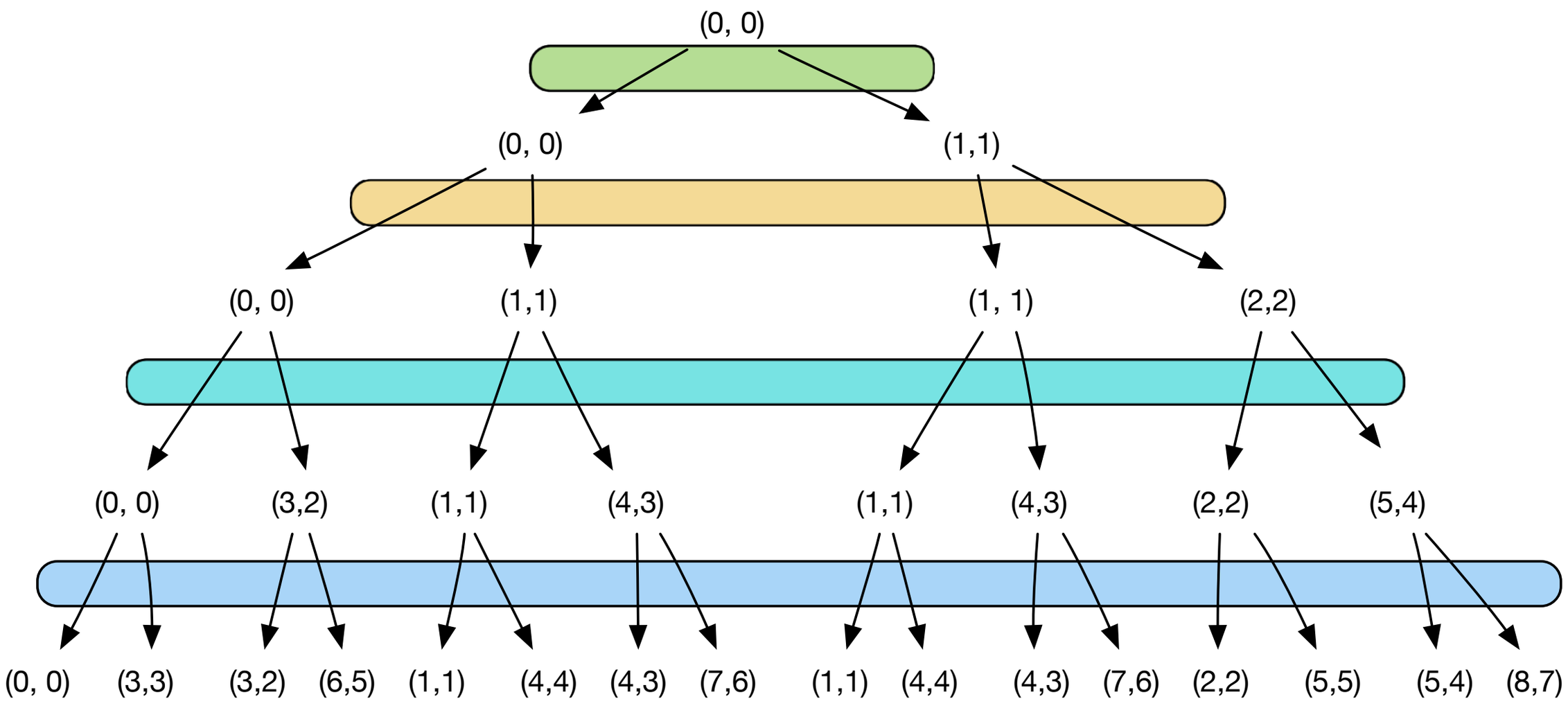
The 0-1 Knapsack Problem

A More Complex Example

Suppose we have a set of items A, B, C, D.... Each item has a weight and value associated with it.

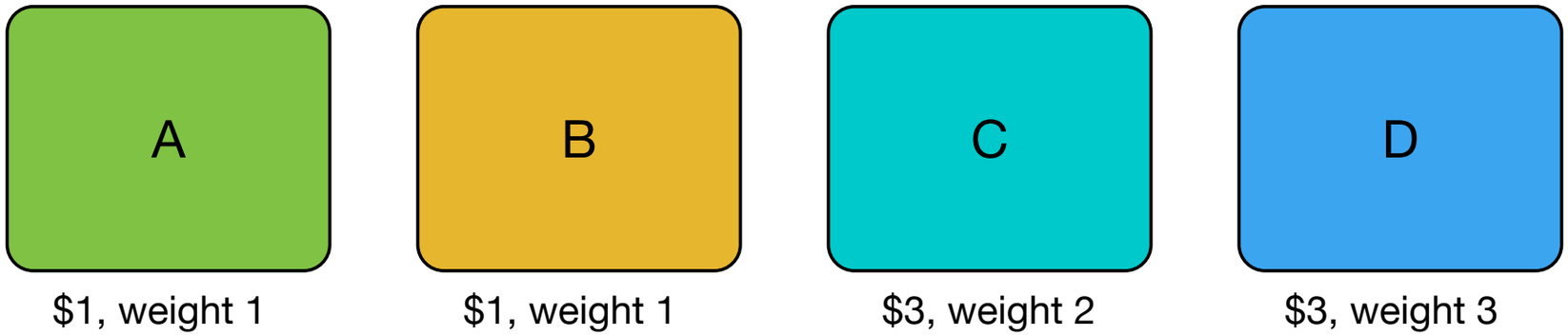
The objective? Select a subset of items to maximize total value, subject to a total weight constraint.

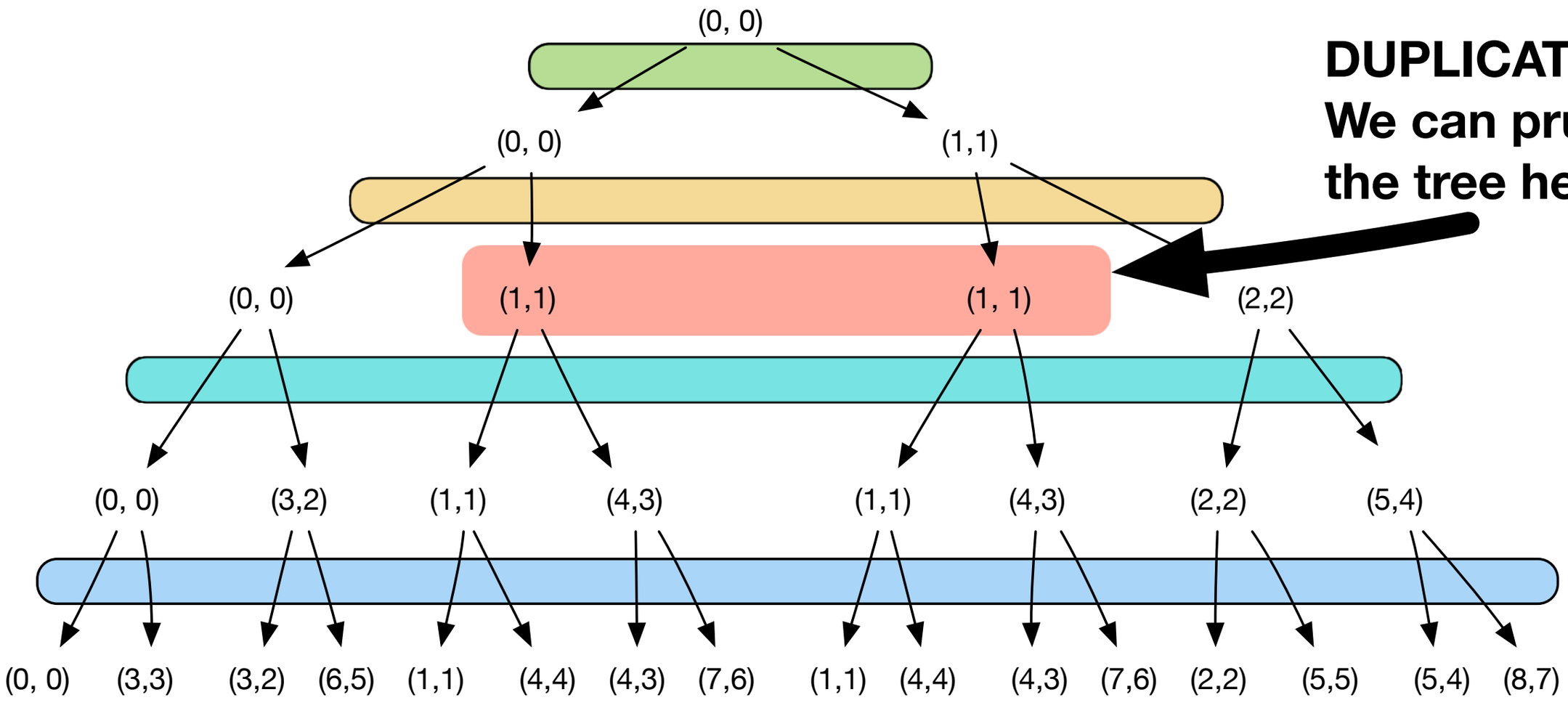

A
B
C
D
$1, weight 1
$1, weight 1
$3, weight 2
$3, weight 3


(0, 0)
(0, 0)
(1,1)
(0, 0)
(1,1)
(1, 1)
(2,2)
(0, 0)
(3,2)
(1,1)
(4,3)
(1,1)
(4,3)
(2,2)
(5,4)
(0, 0)
(3,3)
(3,2)
(6,5)
(1,1)
(4,4)
(4,3)
(7,6)
(1,1)
(4,4)
(4,3)
(7,6)
(2,2)
(5,5)
(5,4)
(8,7)

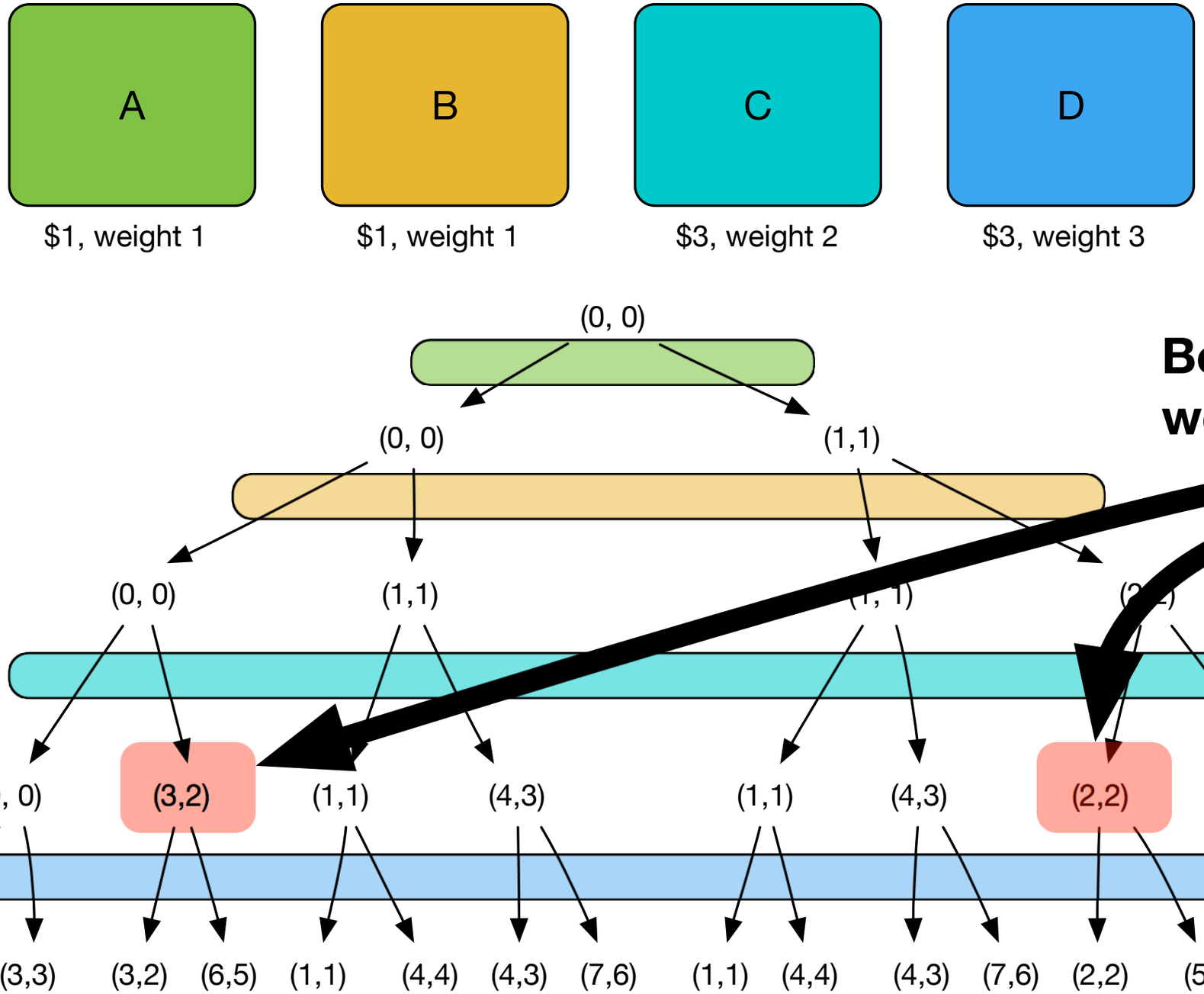
Brute Force Enumeration

- Easy to conceptualize, but grows exponentially
- Practical for twenty-ish items (CPU is cheap!)
- Dynamic programming can dramatically reduce the solution space
- Note the integer weights. This is important for the standard DP approach.





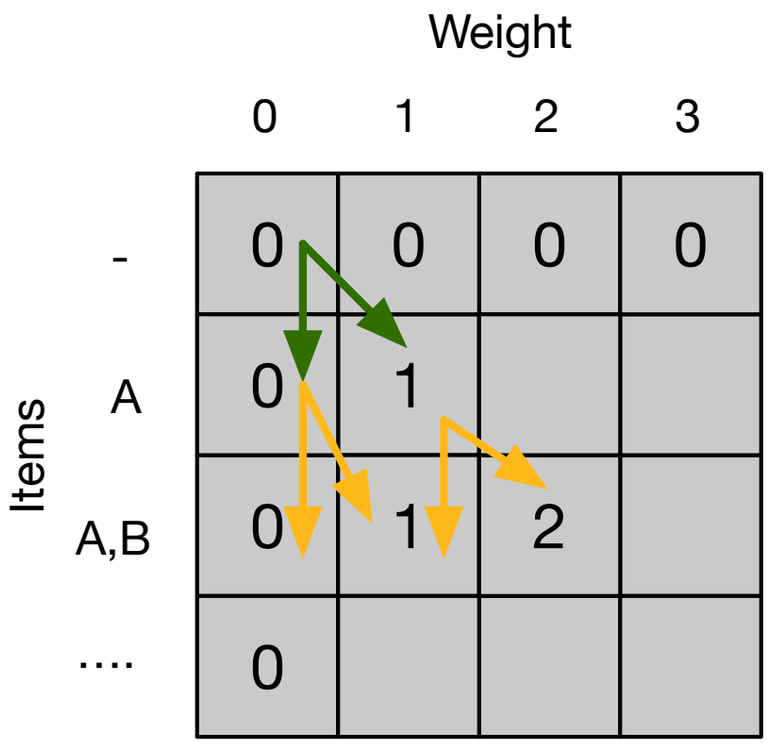
DUPLICATE!
We can prune the tree here!


A
B
C
D
$1, weight 1
$1, weight 1
$3, weight 2
$3, weight 3
(0, 0)
(0, 0)
(1,1)
(0, 0)
(1,1)
(1, 1)
(0, 0)
(3,2)
(1,1)
(4,3)
(1,1)
(4,3)
(2,2)
(5,4)
(0, 0)
(3,3)
(3,2)
(6,5)
(1,1)
(4,4)
(4,3)
(7,6)
(1,1)
(4,4)
(4,3)
(7,6)
(2,2)
(5,5)
(5,4)
(8,7)
Better solution with weight of 2

Integer 0-1 Knapsack

```
// Initialize dp_table entries to 0, values, weights
// as needed.  Items are indexed starting with 1
static int dp_table[ITEMS+1][MAX_WEIGHT + 1];
static int values[ITEMS+1];
static int weights[ITEMS+1];
int knapsack()
{
  for (int i = 1; i <= ITEMS; ++i)
  {
    for (int w = 0; w <= MAX_WEIGHT; ++w)
    {
      dp_table[i][w] = dp_table[i - 1][w];
      if (w >= weights[i])
      {
        // Find the value for this weight if we select the item
        int newvalue = dp_table[i-1][w - weights[i]] + values[i];
        if (newvalue > dp_table[i][w])
          dp_table[i][w] = newvalue;
      }
    }
}
```

The brute-force tree is embedded in the matrix



The brute-force tree maps into a simple matrix.

One column for each of the W+1 possible weights.

.... One row, each time we consider a new item.

Collisions in the matrix allow easy detection of duplicates, or instances where one solution dominates another.

If Integer Knapsack is NP-Complete, How Can We Solve It?

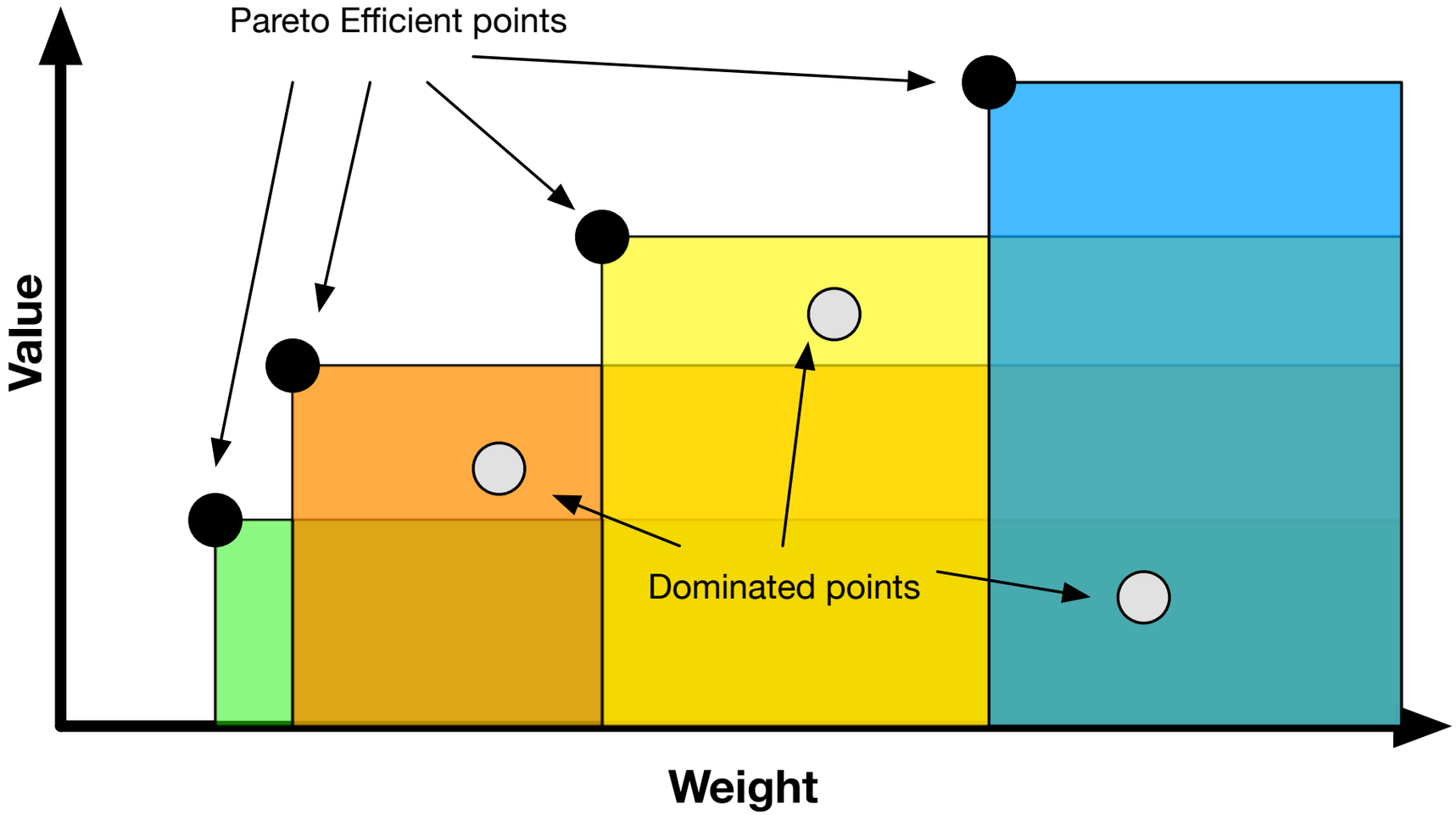
- The standard DP algorithm is pseudo-polynomial, $O(n * W)$
  - Weights are integers, and complexity depends on maximum weight W
  - Collisions in the matrix allow duplicate removal, solution space pruning

What If We Don't Have Integers?

The array/matrix structures used in dynamic programming make it very easy to find solutions that duplicate, overlap, or dominate one another. But if we don't have integer indices, life gets more tricky.

What we are looking for is the dominance of one solution over another (is solution A as good as, or better in every way, compared to a solution B.) This is Pareto dominance, Pareto optimality, Pareto efficiency...

Pareto Dominance



Pareto Dominance

- The importance of Pareto dominance for dynamic programming can't be stressed enough!
- It may be impossible to compare any two partial solutions directly
- but the *set* of solutions will usually have a smaller *subset* that dominates. We only need to keep track of the subset.
- With integer knapsack, the size of that subset is guaranteed to be no larger than W

Dynamic Programming with Lists

By removing the integer constraint, pruning the solution space becomes a little bit more work -- but we are no longer forced to have a $O(n \times W)$ matrix.

Instead, we build a tree level by level, using Pareto dominance to prune as we go. There is a potential explosion of the solution space -- but this rarely happens in practice.

Finding the Pareto Dominating Set (quickly!)

The obvious way is $O(n^2)$, but there's a much more elegant, much more clever divide-and-conquer approach that is $O(nlog^k n)$, for k-dimensional spaces. A bit tricky to implement, but worth the effort! A classic 1980 CACM paper:

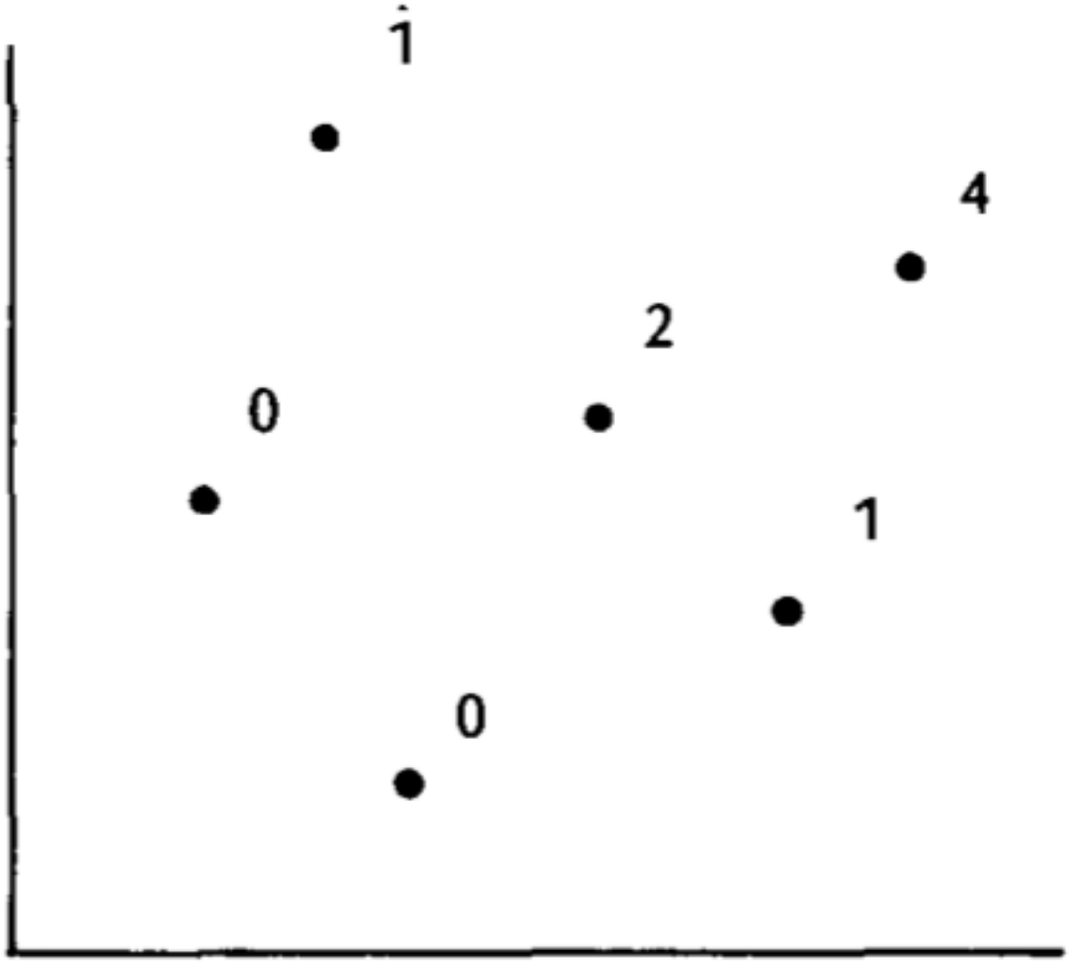
Programming Techniques
R. Rivest Editor

Multidimensional Divide-and-Conquer

Jon Louis Bentley
Carnegie-Mellon University

Empirical Cumulative Distribution Function

Fig. 2. With each point is associated its rank.



The J. L. Bentley paper doesn't mention Pareto dominance, so it won't come up easily in a search. But the ECDF is doing the same thing -- and we can get the total number of points any other point dominates quickly.

More than one Bentley!

If you use Google to find the Bentley paper -- be careful! There is heuristic approach by P. J. Bentley, published in 1998, which is different from the exact algorithm by J. L. Bentley, published in 1980!

Hard Combinatorial Problems

Image from Garey & Johnson, Computers and Intractability

"I can't find an efficient algorithm, but neither can all these famous people."

A Polynomial-Time Solution is not Guaranteed

But.... we might not care

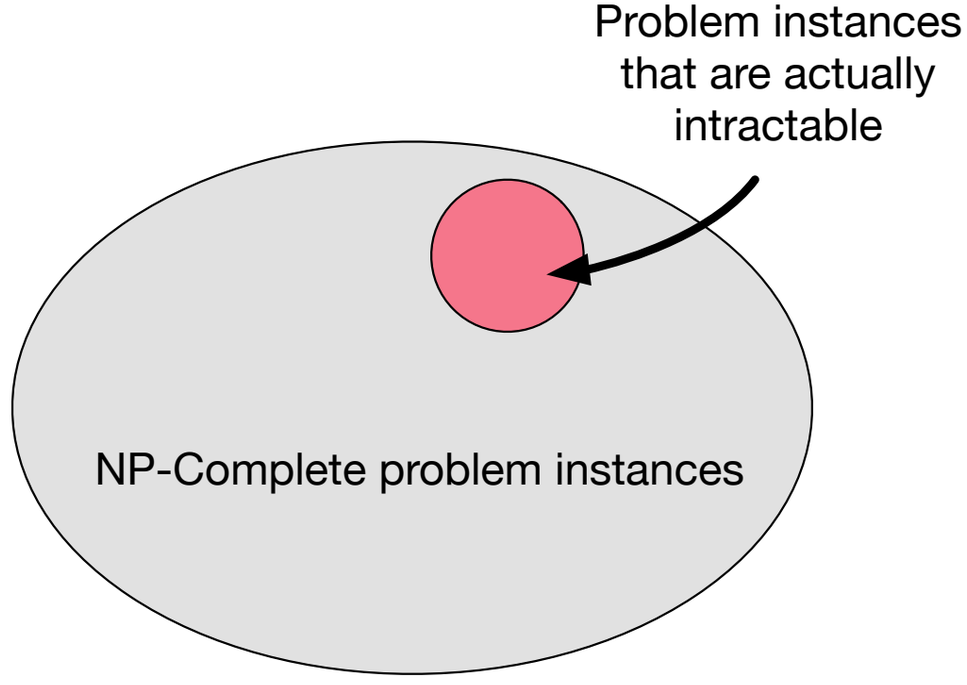
A problem in the NP-Complete class has no known polynomial time algorithm that can solve *all* instances of the problem. But in practice, many problems are not pathological in nature. Boolean satisfiability problems, for example, are routinely solved in logic synthesis.

An algorithm that can solve all problem instances, even pathological ones, would be a great **theory** advance.

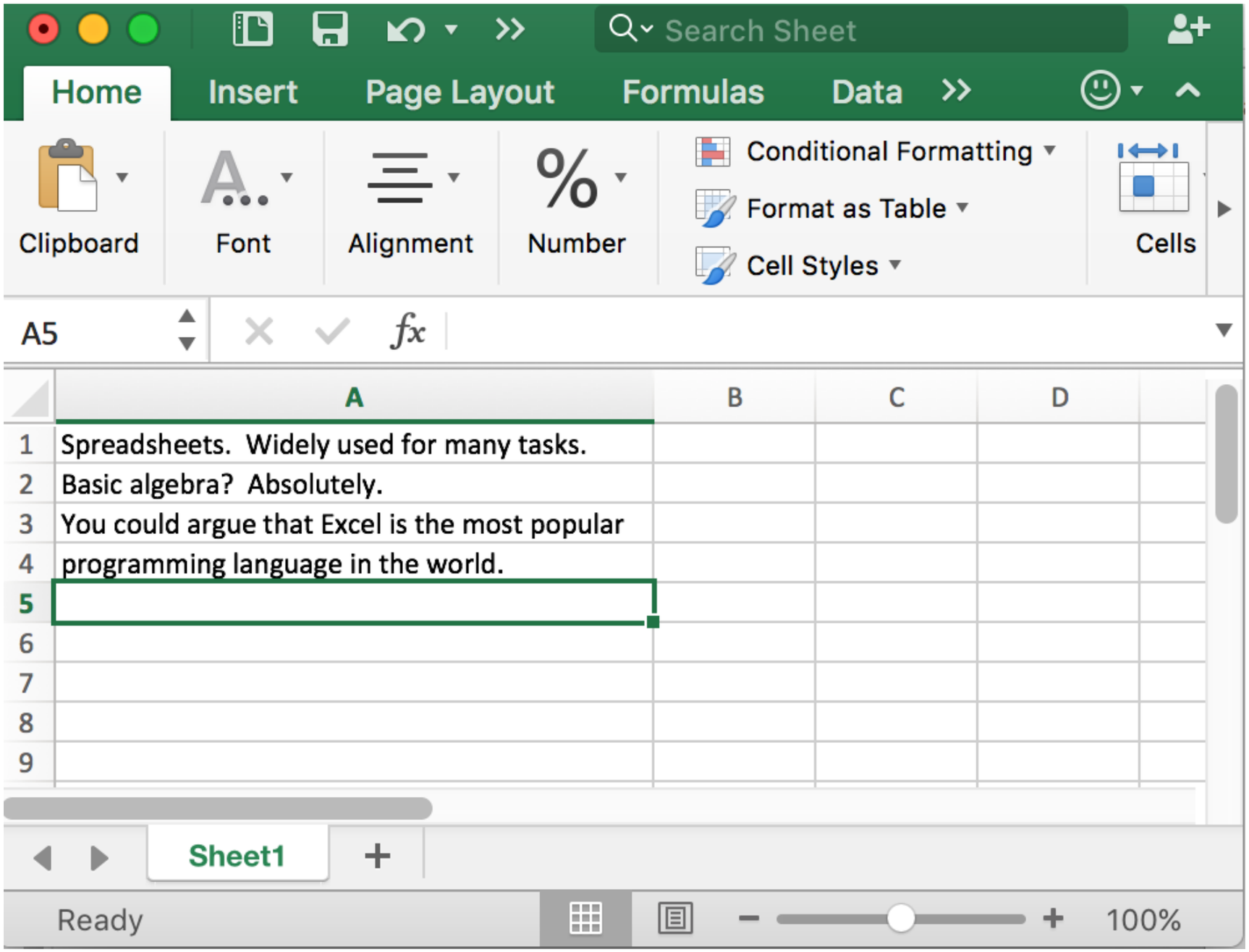
Solving a single instance at hand, however, is of **practical value**.

Not All NP-Complete Problem Instances are Actually Hard

And you never know which is which until you try



Tools for the Masses



Tools for the Masses

Modern spreadsheets made basic algebra easily available to a massive audience; the first spreadsheets were the "killer apps" that kickstarted the personal computing revolution.

- Free-form definition of cells, and relationships
- The spreadsheet performs a topological sort, and then crunches the numbers
- The flexibility is a key feature

A "Tree-First" Perspective

Explore the solution space in breadth-first manner, brute-force enumeration style. Use lists to maintain the solutions at each level of the tree.

The brute-force tree is easy to grasp intellectually, and removes any requirement to map things into an array or matrix. Branching is unconstrained.

The power of dynamic programming comes from it's ability to prune the solution space. Bentley's algorithm allows k-dimensional dominance to be determined quickly and easily; no integers required!

Real World Applications

Individual transistors and gates can be resized.

Bigger transistors are faster -- but consume more power.

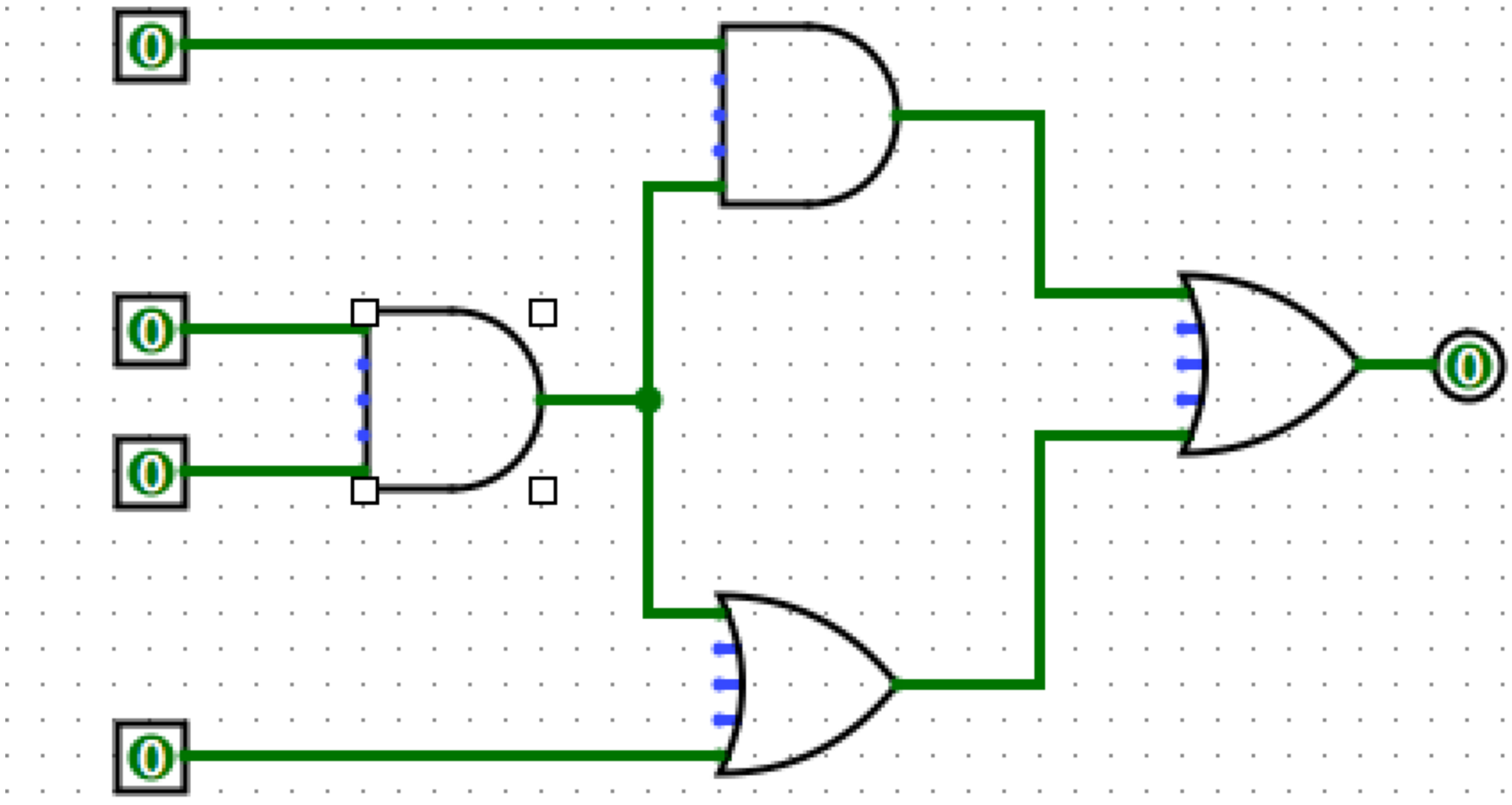
Benefit of increasing the size of inverters A, B, C, D?

A: 1 unit more power, 1 unit faster

B: 1 unit more power, 1 unit faster

Yes... This is exactly the Knapsack problem

The Real World is Poorly Behaved



The Real World is Poorly Behaved

- Sizing one chain of inverters is like the knapsack problem; easy, but very limited.
- Real circuits have fan-in and fan-out
- Electrical delay is non-trivial to compute
- Each Boolean gate might have a number of different sizes and configurations
- With multiple voltage levels on a circuit, some configurations cannot be allowed
- The numerical freedom of a spreadsheet is required. Every circuit is weird in some way.

Enumeration Sieve

Dynamic Programming (and more) for the Masses

Because every circuit is unique, we focused on a general purpose solver. The foundation is brute-force enumeration, with the ability to prune the search space in a variety of ways.

Problems are defined using a simple, human-readable language. Specify the decisions available, and how they interact. The solver constructs the tree, and prunes the solution space. *At present, pruning operations must be user-specified.*

Enumeration Sieve

- If the solution space is small (a few hundred million combinations), just use brute force.
- If greedy choice is available, or hard constraints are known, prune with these.
- If appropriate, use k-dimensional Pareto dominance.
- If a good heuristic is available, use that -- perhaps keeping the top m solutions for a large value of m. Keeping large numbers of partial solutions allows the solver to degrade gracefully on difficult problems.
- The order in which decision variables are considered, and the methods used to prune, are user-specified, and can change freely.

Enumeration Sieve Language

```
/* Knapsack example
 * Three items
 * A wt 6, value 7
 * B wt 5, value 5
 * C wt 5, value 5
 */
block item_a
{
        float weight;
        float value;

        /* Dominance for higher, lower weight */
        pareto (high value, low weight);

    option include_a
        {
                weight = 6;
                value = 7;
        }
        option reject_a
        {
                weight = 0;
                value = 0;
    }
}

block item_b
{
        float weight;
        float value;

        /* Dominance for higher, lower weight */
        pareto (high value, low weight);

    option include_b
    {
            /* Can only include if there are 5 weight available */
        constrain item_a.weight <= 5;

        weight = item_a.weight + 5;
        value = item_a.value + 5;
    }
```

There's An App for That

To make the tool widely available, we've developed an iOS app, which will be made available shortly. Stand-alone command line solvers, and linkable libraries, are not too far off.

```
/* Knapsack example
 * Three items
 * A wt 6, value 7
 * B wt 5, value 5
 * C wt 5, value 5
 */

block group_a
{
  float weight;
  float value;
  option include_a
  {
    weight = 6;
    value = 7;
  }
  option reject_a
  {
    weight = 0;
    value = 0;
  }
}

block group_b
{
  float weight;
  float value;
  option include_b
  {
    weight = 5 + group_a.weight;
    value = 5 + group_a.value;
  }
  option reject_b
  {
    weight = group_a.weight;
    value = group_a.value;
  }
}


block group_c
{
  float weight;
  float value;
  option include_c
  {
    weight = 5 + group_b.weight;
    value = 7 + group_b.value;
  }
  option reject_c
  {
    weight = group_b.weight;
    value = group_b.value;
  }
}
```

```
/* Demonstration enumeration sieve input file; four gates, each
 * of which has high and low voltage threshold options, and two
 * different driver sizes.  The language is generic--if you want
 * to add in parameters for slew, best-case/worst-case, etc.,
 * the solver can currently handle it.
 *
 * The circuit structure that A fans out to B and C, and then
 * D catches the output of B and C.  Very simple circuit, just
 * for demo purposes.
 *
 * The delays along the line are a wire delay, plus the delay
 * from the previous gate; for D, which has two inputs, we take
 * the maximum delay from B and C.
 *
 * Total power consumed is a bit tricky--as it's cumulative,
 * we just add up each gate as we go along -- so even though
 * B and C are in parallel, the power consumed at gate C
 * is just the power from C itself plus the power at B.  You
 * can think of this as power(A) + power(B) + power(C) + power(D).
 *
 * Essentially, we're doing exhaustive enumeration, but the "keep"
 * lines restrict how many solutions we preserve to the next
 * level.  We're using the basic approach on our multi-row placement
 * legalizer, and it's working very well.
 */

block gate_a
{
    float delay;
    float power;
    float powerlevel;

    option lp_narrow
    {
        power = 5 ;
        delay = 10;
        powerlevel = 0;
    }

    option hp_narrow
    {
        power = 10 ;
        delay = 7;
        powerlevel = 1;

    }

    option lp_wide
    {
        power = 8 ;
        delay = 8;
        powerlevel = 0;
    }

    option hp_wide
    {
        power = 15 ;
        delay = 4;
        powerlevel = 1;
    }
}


block gate_b
{
    float delay;
    float power;
    float powerlevel;

    option lp_narrow
    {
        power = 5  + gate_a.power;
        delay = 10 + gate_a.delay;
        powerlevel = 0;
    }

    option hp_narrow
    {
```

Thanks!

Dynamic programming has been a key part of many of our research efforts. The enumeration sieve work is an attempt to make the technique easy to use -- especially to people without a computing background.

Questions? Please contact me at pmadden@acm.org