CAD of VLSI

Tutorial # 9

Optimization Methods in CAD of VLSI
Introduction

• **Computer Aided Design** = Automation of the design process

• Design process consists of stages in each of which we solve *optimization problem*
  – Usually: minimize area, delay, power under different constrains

• Good design automation:
  – good enough (possibly optimal) solution of optimization problem
  – fast enough solution of optimization problem

• So, the main question in design automation is:

  *How to solve optimization problem in fast and exact way?*
What is optimization

- Optimization, in general, is the process of decision making which leads to the best value of optimized objective.
- In mathematics, optimization (or *mathematical programming*) means minimizing or maximizing some function on certain (finite or infinite domain).
- If constraints are implied on function domain, then the optimization is *constrained*, otherwise – *unconstrained*.
- When number of possible values of objective function is finite, then it is *combinatorial optimization* problem.
- Optimization problem always satisfies one of the following:
  - Infeasible
  - Has global optimum
  - Unbounded
- In many cases, optimization problem has also one or more local optima.
Optimization domains

• If the objective function is continuous, then the optimization problem is *continuous optimization problem*. Examples from CAD:
  – find sizes of inverters in buffer so that buffer power is minimized
  – minimize width of interconnect channel under delay constraints

• Continuous optimization problems in many cases may be solved analytically or numerically

• Continuous problems with “good” properties may be provided to have a single optimum, i.e. each local optimum is also its global optimum.

• However, most of CAD optimization problems are *constrained combinatorial optimization* problems. Some examples:
  – Find the shortest path between two vertices in the graph
  – Find the placement of cells with minimum area
  – Find the smallest representation of Boolean function

• The solution of combinatorial optimization problem can be obtained by simply checking all possible values of objective function, but usually the number of possibilities is very large (exponential)
Continuous optimization: convex vs. non-convex problems

- Continuous problems with “good” properties are called *convex*
  - Has single minimum
  - Efficiently solvable analytically or numerically (descent methods)

- *Non-convex* problems are hard to solve exactly
  - Usually solved numerically, using heuristic methods
Analytical vs. numerical solution

- **Analytical solution**: set of equations and inequalities are solved
  - Obtained solution is exact
  - For example: least squares problem

- If problem is hard or impossible to solve analytically, then analytical investigation of the problem can help to develop numerical solution

- **Numerical solution**: non-exact. Usually differs from exact up to predefined accuracy
  - Example: steepest descent method, Newton method
Discrete optimization: order of growth

- Optimization domain of discrete problem depends on problem size
- For practicality, we want polynomial-time algorithms

<table>
<thead>
<tr>
<th>Function</th>
<th>10</th>
<th>20</th>
<th>$10^2$</th>
<th>$10^3$</th>
<th>$10^6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T(n)=n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T(n)=3n</td>
<td>30</td>
<td>60</td>
<td>$3 \times 10^2$</td>
<td>$3 \times 10^3$</td>
<td>$3 \times 10^6$</td>
</tr>
<tr>
<td>T(n)=nlogn</td>
<td>10</td>
<td>26</td>
<td>$2 \times 10^2$</td>
<td>$3 \times 10^3$</td>
<td>$6 \times 10^6$</td>
</tr>
<tr>
<td>T(n)=n^2</td>
<td>$10^2$</td>
<td>$4 \times 10^2$</td>
<td>$10^4$</td>
<td>$10^6$</td>
<td>$10^{12}$</td>
</tr>
<tr>
<td>T(n)=n^3</td>
<td>$10^3$</td>
<td>$8 \times 10^2$</td>
<td>$10^6$</td>
<td>$10^9$</td>
<td>$10^{18}$</td>
</tr>
<tr>
<td>T(n)=2^n</td>
<td>$10^3$</td>
<td>$10^6$</td>
<td>$10^{30}$</td>
<td>$10^{301}$</td>
<td>&gt; $10^{500}$</td>
</tr>
<tr>
<td>T(n)=n!</td>
<td>$3 \times 10^6$</td>
<td>$2 \times 10^{18}$</td>
<td>$9 \times 10^{157}$</td>
<td>&gt; $10^{500}$</td>
<td>&gt; $10^{500}$</td>
</tr>
</tbody>
</table>
Discrete optimization: \( P \) vs. \( NP \)

- **\( P \)** – “poly-find” – a class of problems solvable in polynomial time. Examples:
  - finding shortest path in a graph, finding minimum spanning tree
- **\( NP\)–complete** – “poly-verify” – a class of problems, which are not in \( P \), but verifiable in polynomial time. Usually represent decision problems. Examples:
  - Are there any Hamiltonian paths* in given graph?
  - Are there any variable assignments that satisfy given SOP?
- **\( NP\)–hard** – at least as hard as \( NP\)–complete. Usually represent optimization versions of appropriate decision problems.
  - What is the minimum length Hamiltonian path* in given graph? (Travelling Salesperson Problem - \( TSP \))

(*) Hamiltonian path is a path in an undirected graph that visits each vertex exactly once.
Exact and heuristic solutions

• **Bad news:** most of CAD problems are: **NP-complete or NP-hard**
  – Impossible to find exact solution by exhaustive search

• We use **heuristic methods** to get to optimal solution as close as possible
  – Heuristics are "rules of thumb", educated guesses, intuitive judgments or simply common sense
  – Sometimes can result in optimal solution!
  – Example: for TSP, start from minimum spanning tree, then try to convert it to Hamiltonian path
Optimization strategies for hard problems

• We cannot afford spending exponential time to solve the problem.
• Solving the problem with high quality, i.e. use good heuristics.
  – Trade-off quality for run-time.
  – Might need to solve the problem at different steps, but have different requirements of solution quality at run-time.
  – Common case: approximate algorithms (of simpler complexity) that guarantee that the result is within some margin of the optimum.
• Solving a simpler (or restricted) version of the problem.
  – Reveal insight of the general problem.
  – Heuristic for solving original problem.
Discrete optimization methods

• **Exact methods:**
  – Exhaustive search
  – Backtracking with branch-and-bound
  – Dynamic programming – *will be shown independently*

• **Heuristic methods:**
  – Greedy approach
  – Local search
  – Divide-and-conquer
  – Taboo search
  – Genetic algorithms
  – Simulated annealing – *will be shown independently*
Example: a bin-packing problem

• We have a collection of items $T = \{t_1, t_2, t_3, \ldots, t_n\}$.
• Every item $t_k$ has an integer size $s_k$.
• There is a set of $m$ bins called $B$, each with a fixed integer size $b$. - Can hold items if the sum of their sizes is $b$ or less.
• GOAL: Pack all items, using a minimum number of bins.
• The problem is NP-hard.
Motivation - standard cells

Cell based design:

Motivation - floorplanning

Floorplanning is a 2D packing problem!

Functional units: 

Compact layout:
Motivation – more

There are more interesting variations:

3D packing problem!

- Component fitting for high density.
- Multi-layer routing.
- Resource allocation.
Bin Packing problem

- We have bins with maximum size of 6 units. \( b = 6 \)
- We have items with sizes: \( \{1, 4, 2, 1, 2, 3, 5\} \) \( s_{1\ldots7} = t_{1\ldots7} \)
- The **cost function** is simply the number of bins in \( B \).
- We will demonstrate several optimization methods with this simple problem.
Exhaustive search

- Generate all possible combinations, decide which ones are feasible, find one with minimum cost
- Optimal but slow - traverses the whole search-tree (exponential)

Exact methods:
- Exhaustive search
- Backtracking with branch-and-bound
- Dynamic programming
Exhaustive search by backtracking

- Systematic method of traversing a graph recursively.
- **Backtracking:**
  - A *recursive* way for doing an exhaustive search.
  - Start with a partial solution with as many as possible unspecified variables.
  - Systematically assign values to variables.
    - Try all allowed values for variable $k+1$, given a choice of the first $k$ variables.
  - Until a feasible solution is found or until a “dead-end”.
  - Go back to an earlier partial solution and keep on trying.
Backtracking on our example

- The items: \{1, 4, 2, 1, 2, 3, 5\}
- Item 1\(_{(1)}\) put in bin 1
- Item 2\(_{(4)}\) has 2 options, bin 1 or bin 2
- If items 1\(_{(1)}\) and 2\(_{(4)}\) are in same bin, then item 3\(_{(2)}\) has only one option, bin 2
- If items 1\(_{(1)}\) and 2\(_{(4)}\) are in separate bins, then item 3\(_{(2)}\) has 3 options
- and so on …
- This creates only feasible solutions
Back-tracking pseudo code

```c
float best_cost;
solution_element val[n][m], best_solution[m];

backtrack (int k) {
    float new_cost;
    if (k == n) {
        new_cost := cost(val);
        if (new_cost < best_cost) {
            best_cost := new_cost;
            best_solution := copy(val);
        }
    } else {
        foreach (el[] in allowed(val,k)) {
            val[k][] := el[];
            backtrack(k+1);
        }
    }
}

main () {
    best_cost := infinity;
    backtrack(0);
    report(best_solution);
}
```

- We have a collection of items \( T=\{t_1, t_2, t_3, \ldots, t_n\} \).
- Every item \( t_k \) has an integer size \( s_k \).
Branch and Bound

- Branch & bound is refinement of backtracking
- To avoid complete enumeration of all possible solutions, we want to prune the tree (cut out some parts)
- For each branch, let's compute a lower bound for all solutions in the sub-tree that grows from it
- If that bound is higher cost than the best solution we found so far, we can skip (prune) the sub-tree!
  - Algorithm is still exponential, but can be much less on average
  - Branching selection heuristics: try to visit the most promising solutions early!
    It would facilitate pruning and save time (but no effect on exactness of solution)
Branch & bound pseudo code

```plaintext
float best_cost;
solution_element val[n], best_solution[n];

b&b (int k) {
    float new_cost;
    if (k == n) {
        new_cost := cost(val);
        if (new_cost < best_cost) {
            best_cost := new_cost;
            best_solution := copy(val);
        }
    } else if (lower_bound_cost(val,k) >= best_cost
               return
    } else {
        foreach (el in allowed(val,k)) {
            val[k] := el;
            backtrack(k+1);
        }
    }
}

main () {
    best_cost := infinity;
b&b(0);
    report(best_solution);
}
```
Illustration:
A toy problem with four possible solutions
The cost of a leaf is the sum of the arcs on the path to it.
Assume the bound is set to the lowest cost found.
(initialized to infinity).
After visiting the left subtree, discovering a solution of cost 4, we can prune the whole subtree on the right (c), since we know that all its solutions would cost at least 6!
A “sharp” /”tight” bounding function can save time
– It should also be quick to compute.
– What about an inaccurate, incorrect bound?
Branch and Bound on our example

- The items: \{1, 4, 2, 1, 2, 3, 5\}
- Once we found the solution with 3 bins: 
  - We can prune all those reaching 4!

\[
\begin{array}{c|c}
 s_k & k = \# \text{ of item} \\
 \hline
 0 & \\
 1 & 1 \\
 4 & 2 \\
 2 & 3 \\
 5 & 7 \\
\end{array}
\]

\[
\begin{align*}
\{1, 5\} & \{4, 2\} \{2, 1, 3\} \Rightarrow \text{cost is 3}
\end{align*}
\]
**Divide-and-conquer approach**

- **Divide** the problem into smaller (simpler) sub problems.
- **Conquer** the problems by solving them recursively.
  - Keep partitioning until sub problems are easy enough.
- **Combine** the solutions of the sub problems into the solution for the original problem.
- In our example:
  - Let’s interpret “easy enough” = sub problem fits in 1 bin.
- Partition: \{1, 4\} \{2, 1\} \{2, 3\} \{5\}
Greedy approach (heuristic)

- A greedy algorithm always makes the choice that looks best at the moment.
- This is a simple and fast heuristic
- Usually doesn’t provide global minimum
- In our case: let’s try first fit algorithm - placing each item into the first bin in which it will fit
Greedy approach (heuristic)

- A greedy algorithm always makes the choice that looks best at the moment.
- This is a simple and fast heuristic
- Usually doesn’t provide global minimum
- In our case: let’s try **first fit algorithm** - placing each item into the first bin in which it will fit

\{ 1, 4, 1 \} \{ 2, 2 \} \{ 3 \} \{ 5 \}
Greedy approach (heuristic)

- Now try **first fit increasing heuristic**: first sort the list of elements into *increasing* order and then place each item into the first bin in which it will fit

\[
\{ 1, 1, 2, 2 \} \{ 3 \} \{ 4 \} \{ 5 \}
\]
Greedy approach (heuristic)

- Finally try **best fit decreasing heuristic**: first sort the list of elements into *decreasing* order and then try to find bin with minimum free space that still can include given item

  \{ 5, 1 \} \{ 4, 2 \} \{ 3, 2, 1 \} – optimal solution!
Local search

- **Local search** is the method that starts from feasible solution $f$ and looks for better solution in a neighborhood $N(f)$
  1. Start from feasible solution
  2. Generate neighborhood of the solution
  3. Choose the solution from the neighborhood with:
     - better cost – *first improvement* strategy, or
     - the best cost – *steepest descent* strategy
  4. go to 2

- Non-intelligent generation of neighborhood can lead to its huge size
  - Usually will use some heuristic to do this

- Main disadvantage: can stuck in a local minimum

- Solutions:
  - repeat local search from some initial points
  - adapt the size of neighborhood during local search
Local search

• In our case:
  – generate neighborhood of given solution by splitting the bin with minimum occupation (whenever possible), and then apply best fit heuristic
Taboo search

- Taboo search is improved version of local search
- Allows movement to the feasible solution with worse cost
  1. Start from feasible solution
  2. Generate neighborhood of the solution
  3. Choose the solution from the neighborhood, which is not in taboo list
  4. If its cost is better than known so far, save it
  5. Save the solution in taboo list
  6. go to 2

- Taboo list is the list of k last visited feasible solutions (which are taboo), to avoid cycles of length ≤ k

Mutations are not necessarily the bad-guys!
Taboo search

• In our case:
  – can generate solutions with the cost (i.e. number of occupied bins) worse than known so far
  – For example, totally split one of bins, then merge some of bins

These solutions are added to taboo list
Genetic Algorithms

- Instead of repetitively transforming a single current solution into a next one by the application of a move, the algorithm simultaneously keeps track of a set of feasible solutions, called the population.
- Start with several feasible solutions (init):
  - \( S_1 = \{ 1, 4 \} \{ 2, 1 \} \{ 2 \} \{ 3 \} \{ 5 \} \)
  - \( S_2 = \{ 1, 2 \} \{ 4, 1 \} \{ 2, 3 \} \{ 5 \} \)
- Obtain new solutions (children) by mutations of parents e.g.
  - From \( S_1 \) obtain \( S_3 = \{ 1, 4 \} \{ 2, 1 \} \{ 2, 3 \} \{ 5 \} \)
  - From \( S_2 \) obtain \( S_4 = \{ 1 \} \{ 2 \} \{ 4, 1 \} \{ 2, 3 \} \{ 5 \} \)
- Obtain new solutions by crossovers of existing solutions.
  e.g. take some members of \( S_1 \) and some of \( S_2 \) such that the generated solutions have required characteristics:
  - From \( S_1 \) and \( S_2 \) obtain \( S_5 = \{ 1, 2, 1 \} \{ 4 \} \{ 2, 3 \} \{ 5 \} \)
  - Eliminate some solutions to reduce population (survival of the fittest)
  - e.g. select subset of solutions - the ones requiring fewest bins (\( S_2, S_3, S_5 \)).
- Repeat procedure till a good solution is found.
Summary

• In this lesson we studied:
  – what is optimization problem
  – kinds of optimization problems, especially in CAD of VLSI
  – methods for solving optimization problems:
    • exact: backtracking, branch and bound, divide and conquer
    • heuristic: greedy, local search, taboo search, genetic

• Two more methods to come:
  – Dynamic programming (exact)
  – Simulated annealing (heuristic)