The Partitioning Problem in VLSI Design

The **partitioning problem** involves dividing a circuit into two balanced subcircuits to minimize the number of connections (nets) cut between them. It is commonly solved using **iterative improvement algorithms**, where an initial solution is gradually refined based on certain “goodness” criteria.

1. Iterative Improvement Method

This general method starts with an initial partition and repeatedly generates new solutions. Each new partition is compared with the previous one using a *goodness measure* (like the number of cut nets). Greedy algorithms only accept better solutions, while non-greedy (hill-climbing) methods may accept worse ones temporarily to escape local minima.

2. Kernighan–Lin (KL) Algorithm

The KL algorithm treats the circuit as a **graph**, where gates are vertices and connections are edges.
Steps:

- Randomly divide gates into two equal groups.
- Calculate **cut** (edges between groups) and **uncut** edges.
- Evaluate possible swaps of gate pairs between partitions to reduce cuts.
- Record and apply the sequence of swaps that yields the minimum cut.
  This process repeats until no improvement occurs.
  While effective, KL has high computational cost (about $O(n^4)$ in basic form).

3. Fidducia–Mattheyses (FM) Algorithm

FM is an optimized version of KL:

- Works directly with **nets** instead of graph edges.
- Allows slight imbalance between partitions.
- Uses faster single-gate moves instead of pair swaps.
- Measures quality by the number of **cut nets**, not edges.
  It produces similar or better results more efficiently.

4. Simulated Annealing (SA)

Simulated Annealing is a probabilistic optimization method inspired by thermodynamics:

- Starts with a random partition and temperature value.
- Random swaps are attempted; better ones are always accepted.
- Worse ones are accepted probabilistically based on the current temperature (to avoid local minima).
- The temperature gradually decreases, reducing acceptance of worse solutions.
  Although **slow**, SA often finds **near-optimal global solutions**.

In short:

The document explains three key algorithms for VLSI **partitioning optimization**:

- **Kernighan–Lin (KL):** classical, pair-based, accurate but slow.
- **Fidducia–Mattheyses (FM):** faster, net-based refinement.
- **Simulated Annealing (SA):** probabilistic, global optimization approach.

The Partitioning Problem

1. Iterative Improvement

The partitioning problem is the problem of breaking a circuit into two subcircuits. Like many problems in VLSI design automation, we will solve this problem by a method known as *Iterative Improvement*. To apply the iterative improvement technique, we need to be able to do several things. These are listed below.

1. We need to be able to generate an *initial solution* to the problem. (This is usually trivial.)

2. We need to have a set of criteria to determine whether or not a solution is *acceptable*. Certain solutions to a problem are simply not acceptable. For instance, a solution to the partitioning problem that puts all gates into one subcircuit and leaves the other subcircuit empty is unacceptable, because it simply gives you what you started with.

3. We need a way of evaluating a solution to determine its *goodness*. Goodness is not measured on an absolute scale, but on a relative one. The goodness of a solution on an absolute scale is not nearly so important as being able to compare two solutions to determine which is "better." It is possible to have several different goodness criteria. A solution that is very good with respect to one criterion might be very bad with respect to another. In general it is difficult to specify simple, measurable criteria that lead to a circuit that is "good" in some global sense.

4. We need one or more techniques for generating new solutions from existing solutions. The techniques must take the acceptability criteria (see 2 above) into account so that only acceptable solutions are generated.

The basic idea of iterative improvement is to start with an initial solution, and generate new solutions iteratively until we have a solution that is as optimal as we can get it. Optimality is measured with respect to the goodness criteria. It is important to note that most iterative improvement techniques *do not* produce optimal solutions, but some of them come quite close. Most iterative improvement techniques are *greedy*. When a new solution is generated from an existing solution we have two choices. We can discard the old solution and keep the new one, or we can discard the new solution, and try some other technique for generating a new solution (or stop the process entirely!) In a greedy algorithm, the new solution is accepted only if it is better than the old one (with respect to the goodness criterion). Non-greedy methods (sometimes known as hill-climbing algorithms) will sometimes accept a solution that is worse than the existing solution. The reason that hill-climbing algorithms are used is to avoid getting trapped in a local minimum. If solution A can be created from solution B in one step, then solution B is called a *neighbor* of solution A. It is possible for a non-optimal solution to be better than all of its neighbors. When this occurs, we say that the solution is a *local minimum*. A hill-climbing algorithm can sometimes climb out of a local minimum, and find an even better solution by temporarily accepting a solution that is worse than the existing solution.

2. The Kernighan-Lin Algorithm

The most basic approaches to the partitioning problem treat the circuit as a graph. This is true for the first, and most famous partitioning algorithm, called the Kernighan-Lin algorithm. This algorithm was originally designed for graph partitioning rather than circuit partitioning, so to apply the algorithm, one must first convert the circuit into a graph. To do this, each gate is treated as a vertex of the graph. If two gates are directly connected by a net, then an edge is placed between the corresponding vertices of the graph. Figure 1 shows the relationship between a circuit and the corresponding graph.

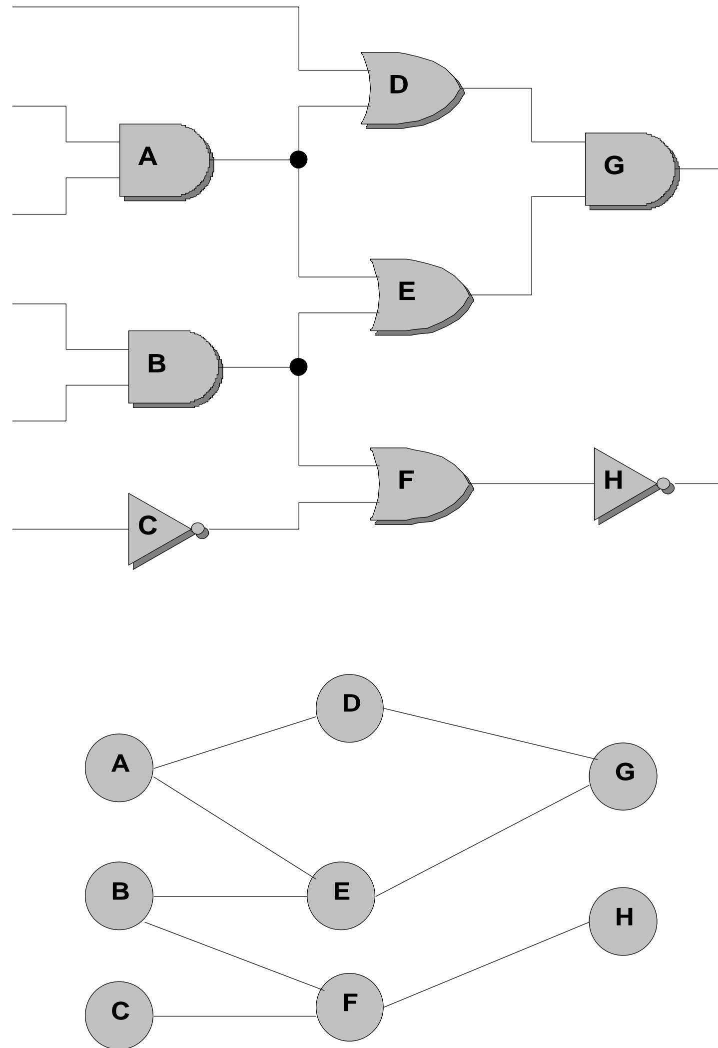


Figure 1. Converting Circuits to Graphs.

Although an approximation of a circuit partitioning can be obtained by partitioning the corresponding graph, graph partitioning and circuit partitioning are two different things. In general, algorithms that treat circuits as circuits will out-perform those that treat circuits as graphs.

The Kernighan-Lin algorithm works as follows.

1. The initial partition is generated “at random.” We create two subcircuits S1, and S2. If the circuit has n gates, the first n/2 are assigned to S1, and the rest are assigned to S2. Because the gates in a circuit description appear in what is essentially a random order, the initial partition appears to be random.

2. A solution is acceptable only if both subcircuits contain the same number of gates. We will assume that the number of gates is even. The algorithm can be “tweaked” to handle an odd number of gates.

3. The goodness of a solution is equal to the number of graph edges that are cut. Suppose the edge (V,W) exists in the graph derived from the circuit. (Recall that V and W are both gates.) There are two possibilities. V and W can be in different subcircuits, or they can be in the same subcircuit. If V and W are in different subcircuits, we say that the edge (V,W) is *cut*. Otherwise we say that (V,W) is *uncut*.

4. The technique for generating new solutions from old solutions is to select a subset of gates from S1, and a subset of gates from S2 and swap them. To maintain acceptability, we always select two subsets of the same size.

Selecting a subset of gates.

Much of the Kernighan-Lin algorithm focuses on selecting gates one at a time, so much so, that it is easy to get the wrong impression about the algorithm. The Kernighan-Lin algorithm *does not* swap gates one at a time. Although the algorithm does many single-gate swaps, these are all done on a temporary basis. The objective of doing these swaps is to find a *subset* of gates to swap on a permanent basis.

The process of selecting a subset of gates is quite lengthy. The first step in this process is to compute the cut and uncut counts for each vertex. For each vertex V in the circuit we compute two values, C_V and U_V. C_V is the number of edges attached to V that are cut, while U_V is the number of edges attached to V that are uncut. The sum, C_V+U_V, is the total number of edges attached to V, which is also known as the *degree* of V.

Let us suppose that K is the number of edges that are cut in the current partition. If we move vertex V from one partition to the other, the change in K will be $I_V=C_V-U_V$. In other words, after moving V from one partition to the other, there will be the new number of edges cut will be $K-I_V$ cut edges in the circuit. This is because, after moving V, any edge attached to V that was cut will no longer be cut, and any edge that was uncut will become cut. The number I_V is generally called the *improvement* with respect to V. A positive improvement is good, while a negative improvement is bad.

The Kernighan-Lin algorithm uses a series of pair-swaps to determine the set of gates to be swapped between partitions. Let V and W be vertices such that V is in one partition and W is in the other. The improvement $I_{(V,W)}$ that is obtained from swapping V and W is I_V+I_W-2 if V and W are connected to one another and, and I_V+I_W otherwise. It is obvious how the second formula was obtained. The first formula reflects the fact that the edge (V,W) between V and W is counted in the cut count of both V and W. When we swap the pair, the improvement (before subtracting 2) gives credit for uncutting (V,W) twice, once for each vertex. Since V and W still end up in different partitions, the status of the edge

(V,W) is never changed to uncut, so we have to subtract out the credit that was given for uncutting it.

Once we have computed the combined improvement for all pairs of gates (there are $n^2/4$ such pairs), we can proceed with swapping pairs of gates. Note that these swaps are *tentative swaps*, no actual movement of gates takes place. Before proceeding, we compute the total number of cut nets in the circuit, and record that count as the "zero swaps" count.

We sort the list of gate pairs into descending sequence, and select the pair, (V,W), with the greatest improvement. We then perform a tentative swap of V and W. We then reduce the total number of cut nets by $I_{(V,W)}$, and we must also update the improvement I_X for any gate X attached to V or W. If the edge (V,X) was cut, we must reduce I_X by one. If the edge (V,X) was uncut, we must increase I_X by one. We must also update the improvement $I_{(X,Y)}$ of any pair of gates containing X, and possibly reposition the pair (X,Y) in the list of sorted pairs of gates.

After completing these steps, we mark V and W as having been swapped (even though we haven't actually moved them yet), and delete any pair of gates containing either V or W from the list of sorted pairs of gates. Once a gate has been swapped with another gate, we won't swap it a second time. We record the pair (V,W) along with the new total number of cut nets we obtained after swapping V and W.

After completing these steps, we go back to the list of sorted pairs of gates, and select the pair with the largest improvement, repeating the steps a second time. We continue selecting gate pairs until the list of available pairs is exhausted. This will cause every gate to be *tentatively* swapped with a gate in the other partition. If we had actually performed the swaps, we would now be back where we started. If there are *n* gates in the circuit, we now have a list of *n*/2 tentative swaps, along with a total cut count for each pair.

To determine the set of gates to be swapped, we search the list of tentative swaps for the minimum total cut count. When we find the minimum, we compare that number to the zero-swaps-count. If the minimum is less than the zero-swaps-count, then we actually perform the swaps from the start of the list of tentative swaps up to, and including, the pair that gave the smallest total cut count. If the smallest total cut count is greater than or equal to the zero-swaps-count, we terminate the entire algorithm.

The Iterative Improvement Process

As complicated as the steps in the previous section are, they represent *one iteration* of the Kernighan-Lin iterative improvement algorithm. After swapping a set of gates, we go back and do the whole thing again, recomputing improvements, counting cut nets, and so forth. (This algorithm has several obvious optimizations.) The entire algorithm stops when an iteration produces no improvement. This is typical of iterative-improvement algorithms.

An Example

To give an example of the Kernighan-Lin algorithm, consider the graph given above, with an initial partition as indicated in xxx.

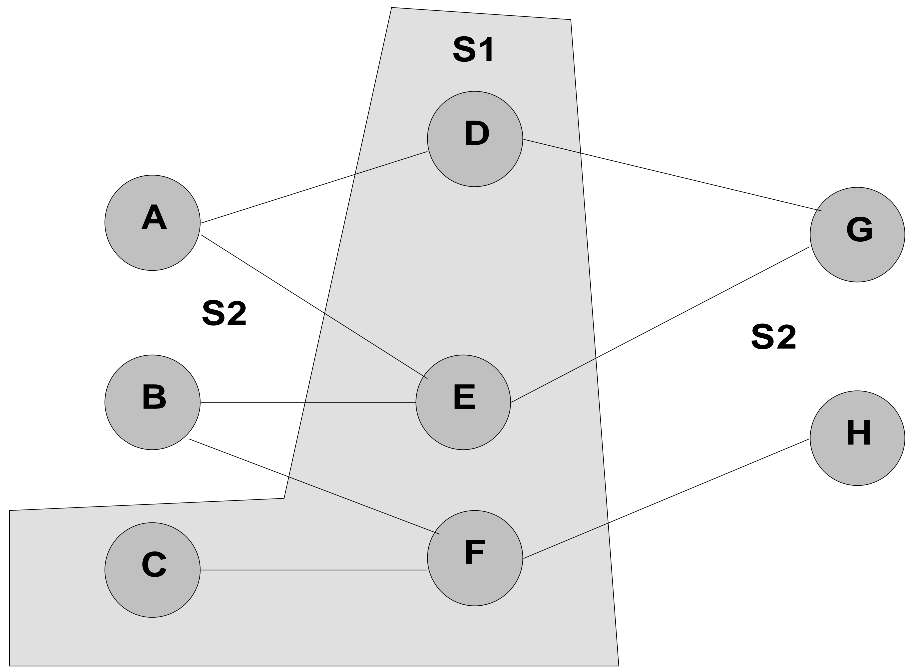


Figure 2. An Initial Partition.

Figure 3 lists the Cut-Count, Uncut-Count, and Improvement for each gate.

Gate	Cut-Count	Uncut-Count	Improvement
A	2	0	2
B	2	0	2
C	0	1	-1
D	2	0	2
E	3	0	3
F	2	1	1
G	2	0	2
H	1	0	1

Figure 3. Gate Improvements, Step 1.

The pair improvements for each pair of gates are given in Figure 4.

Pair	Improvement
A,C	2+(-1)-0=1
A,D	2+2-2=2
A,E	2+3-2=3
A,F	2+1-0=3
B,C	2+(-1)-0=1
B,D	2+2=4
B,E	2+3-2=3
B,F	2+1-2=2
G,C	2+(-1)-0=1
G,D	2+2-2=2
G,E	2+3-2=3
G,F	2+1-0=3
H,C	1+(-1)-0=0
H,D	1+2-0=3
H,E	1+3-0=4
H,F	1+1-2=0

Figure 4. Pair Improvements, Step 1.

There are two pairs with maxim improvement in Figure 4, (B,D) and (H,E). The total number of cut nets is 7. Let us select the pair (B,D) and make a tentative swap. Figure 5 shows the initial state of the list of tentative swaps.

Pair	Cut Count
Zero Swaps	7
(B,D)	3

Figure 5. First Tentative Swap.

The changes in the gate improvements are given in Figure 6.

Gate	Cut-Count	Uncut-Count	Improvement
A	1	1	0
C	0	1	-1
E	2	1	1
F	1	2	-1
G	1	1	0
H	1	0	1

Figure 6. Gate Improvements, Step 2.

The changes in the list of pairs is given in Figure 7.

Pair	Improvement
A,C	0+(-1)-0= -1
A,E	0+1-2= -1
A,F	0+(-1)-0= -1
G,C	0+(-1)-0= -1
G,E	0+1-2= -1
G,F	0+(-1)-0= -1
H,C	1+(-1)-0= 0
H,E	1+1-0= 2
H,F	1+(-1)-2= -2

Figure 7. Pair Improvements, Step 2.

Now, the only pair that has a positive improvement is (H,E), so we select that pair, and perform a tentative swap between them, obtaining the tentative swap list given in Figure 8.

Pair	Cut Count
Zero Swaps	7
(B,D)	3
(H,E)	1

Figure 8. Second Tentative Swap.

The second tentative swap gives the changes to the gate-improvements and pair-improvements listed in Figure 9 and Figure 10.

Gate	Cut-Count	Uncut-Count	Improvement
A	0	2	-2
C	0	1	-1
F	0	3	-3
G	0	2	-2

Figure 9. Gate Improvements, Step 3.

Pair	Improvement
A,C	(-2)+(-1)-0= -3
A,F	(-2)+(-3)-0= -5
G,C	(-2)+(-1)-0= -3
G,F	(-2)+(-3)-0= -5

Figure 10. Pair Improvements, Step 3.

At this point, there are no more positive improvements to be made. Some enhanced versions of the Kernighan-Lin algorithm are capable of detecting this condition and stopping at this point. We’re not that smart yet, so we will persevere by swapping gates A and C, giving the tentative swap list of Figure 11.

Pair	Cut Count
Zero Swaps	7
(B,D)	3
(H,E)	1
(A,C)	4

Figure 11. Third Tentative Swap.

As Figure 11 shows, we are now moving backwards. The new improvements are given in Figure 12 and Figure 13. Since there is only one pair left, (G,F), there is no confusion about which we should select. In fact, it is not necessary for us to compute the new improvements for this pair, because we know that the final swap must return us to the original cut-count of 7.

Gate	Cut-Count	Uncut-Count	Improvement
F	1	2	-1
G	0	2	-2

Figure 12. Gate Improvements, Step 4.

Pair	Improvement
G,F	(-2)+(-1)-0= -3

Figure 13. Pair Improvements, Step 4.

After the final swap of G and F, we end up with the complete tentative swap list given in Figure 14.

Pair	Cut Count
Zero Swaps	7
(B,D)	3
(H,E)	1
(A,C)	4
(G,F)	7

Figure 14. Fourth Tentative Swap.

After creating the tentative swap list, we trace the list from beginning to end searching for the minimum cut count. As Figure 14 shows, the minimum cut count is obtained after swapping (B,D) and (H,E). The minimum cut-count is less than the zero-swaps-count, so we go ahead and make the swaps, giving the improved partitioning shown in Figure 15.

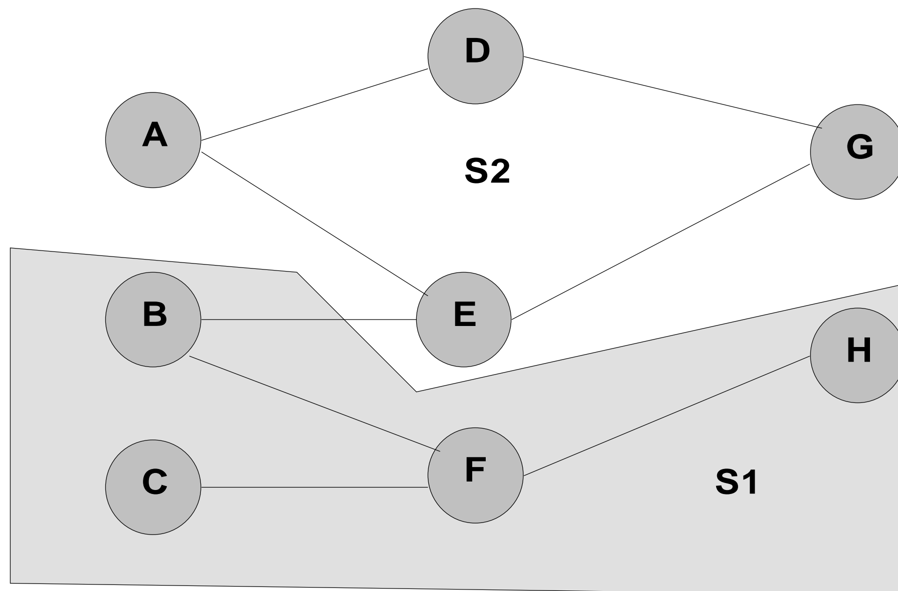


Figure 15. The Final Partition.

The partitioning of Figure 15 is optimal, and a second iteration (which will actually be performed by the algorithm) will show no improvement.

Performance

The Kernighan-Lin algorithm is probably the most intensely studied algorithm for partitioning. There have been many improvements to the algorithm to improve its running time. The unimproved running time is actually quite bad due to the need to examine $n^2/4$ pairs of gates. If we simplistically search through the list every time we wish to swap a new pair of gates, the total running time of each iteration would be $n^3/4$. This could lead to a total running time of $O(n^4)$ or worse. Needless to say, there has been considerable interest in algorithms that deal with individual gates rather than pairs of gates.

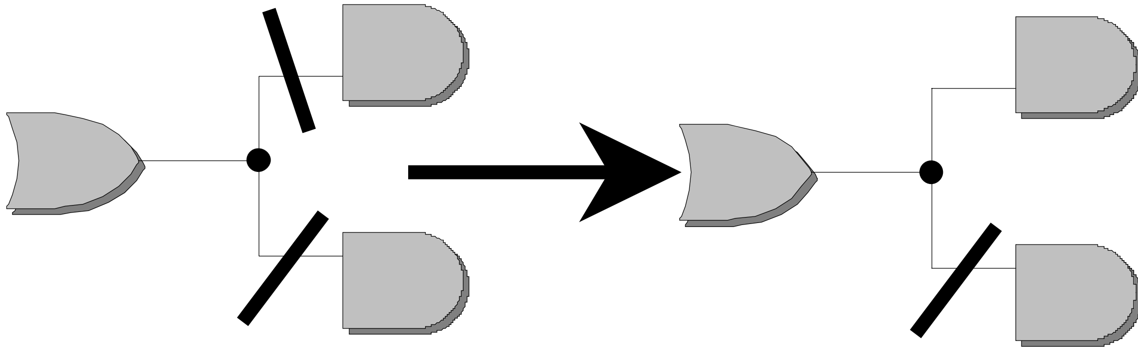
3. The Fidducia-Mattheyses Algorithm

The Fidducia-Mattheyses (FM) algorithm is a cleaned-up version of the Kernighan-Lin algorithm. The difference in the two algorithms lies in the methods used to generate new solutions from old. Although there are significant differences in the process, the basic plan is still the same: start with an initial partition, tentatively swap all gates from one partition to the other, find the point at which the minimum number of nets is cut, and make all swaps up to that point permanent.

The rules for acceptable intermediate solutions are relaxed somewhat to permit a slight unbalancing of the partition. If there are an odd number of gates in the circuit, then it is permissible for one partition to be one unit larger than the other. If there are an even number of gates in the circuit, then it is permissible for one partition to be two gates

larger than the other. In the case of an even number of gates, the final partition, after making tentative swaps permanent, must be balanced.

In addition, the *goodness* of a solution is measured with respect to the number of *nets* cut not with respect to the number of edges in a derived graph. This makes a big difference when more than two gates are attached to a net. In the KL algorithm moving from a state where both fanout branches of a net are cut to a state where only one fanout branch is cut is considered an improvement. FM will not consider this to be an improvement, because the (single) net remains cut. See below.



In KL, the two fanout branches in the above diagram will be converted into two edges in the derived graph. In FM, the single net will be considered alone, so the above situation is an improvement in KL, but not in FM.

4. Simulated Annealing

In many optimization problems, it is possible to get caught in a local minimum. In other words, the algorithm will produce a “minimal” solution that is not even close to minimal, but be unable to find the true minimum, because the only way to get there is to temporarily create a solution that is *worse* than the one the algorithm started with. Computer algorithms are notoriously bad at doing “bad” things for the purpose of “getting something even better.” One algorithm that was specifically designed to get around this problem is Simulated Annealing. Simulated Annealing(SA) is a general iterative improvement algorithm that can be used for many different purposes. In SA, it is necessary to consider several thousand or even several million states, so computing a new state from an old state must be very efficient. In partitioning, SA starts with a random partition, just as the two previous algorithms. A new state is computed by selecting a gate at random from each of the two subsets, and swapping them. As before, the swap remains tentative, until the quality of the new partitioning is computed. The number of nets cut is the measure of goodness. If the new state is better than the old state, it is accepted and the swap is made permanent. If the new state is worse than the old state, it might be accepted and it might not. The SA algorithm operates in a series of distinct phases called “temperatures.” An actual temperature value is assigned to each phase. The algorithm begins with temperature set to a high value, and proceeds to lower and lower temperatures. A predetermined number of moves is attempted at each temperature. When a bad move is attempted, the algorithm computes an acceptance value that is based on temperature and on the “badness” of the solution. This acceptance value is compared to a random number to determine whether the move will be accepted. The random number is

used to guarantee that there is always a non-zero probability that *any* bad move will be accepted. The higher the temperature, the more likely it is that a particular bad move will be accepted, and at a given temperature, the worse the move, the less likely it is to be accepted. In most cases the acceptance function is computed using the following function, where δs is the change in the quality and T is the current temperature. For bad moves this function will produce a value between 0 and 1. A random number between 0 and 1 is generated and if the quality measure is larger than the generated random number, the bad move is accepted. Recall that in partitioning, negative values of δs are good and positive values are bad.

$$e^{-\frac{\delta s}{T}}$$

There are several parameters that must be determined experimentally. The first of these is the starting temperature. This is usually chosen so that many bad moves will be accepted. The second is the cooling schedule. This is the series of temperatures that will be used. The change in temperature is seldom uniform. Usually temperature is changed in large steps at high temperatures and in small steps at lower temperatures. A significant amount of experimentation is usually necessary to determine the best cooling schedule. The final parameter that must be determined is the number of moves to be made at each temperature. This number of moves is generally based both on temperature and on the number of gates in the input. One reasonable choice for a number of moves is 500 times the number of gates in the input.

Simulated annealing generally does a very good job, but runs very, very slowly.

SIMULATED ANNEALING

KEY TERMS

- Floorplanning
- Floorplanning cost – Area + Wirelength cost
- Floorplan optimization – Make the best use of available floorspace

There are two popular approaches to find a desired floorplan:

- Simulated annealing
- Analytical approach

SIMULATED ANNEALING

- Simulated annealing (SA) is probably the most popular method for floorplan optimization
- simulated annealing-based floorplanning relies on the representation of the geometric relationship among modules

SIMULATED ANNEALING APPROACH

- To apply simulated annealing for floorplan design, it needs to first encode a floorplan as a solution , called a floorplan representation
- which models the geometric relation of modules in a floorplan
- a unique solution structure that guides the search of simulated annealing to find a desired floorplan in the solution space

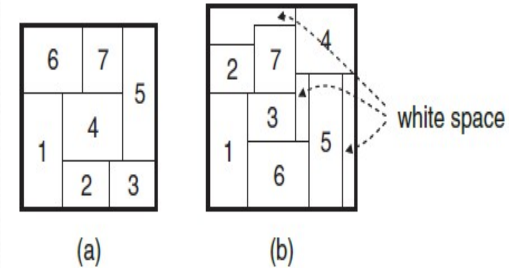


Fig1: (a) an optimal floorplan in terms of area.

(b) a non-optimal floorplan

SIMULATED ANNEALING BASICS

- **Objective - To search for a globally optimal floorplan solution with the lowest cost.**
- Cost of a floorplan -vertical axis
- state of a floorplan -horizontal axis
- initial solution S
- For a greedy approach-simulated annealing adopts a hill-climbing technique (uphill move capability)

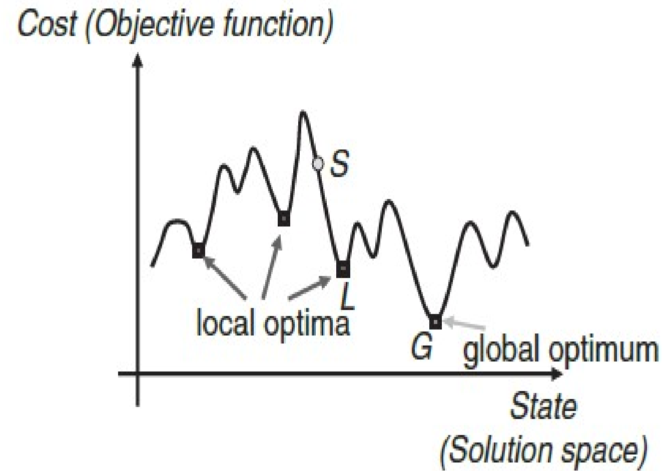


Fig2: Illustration of simulated annealing

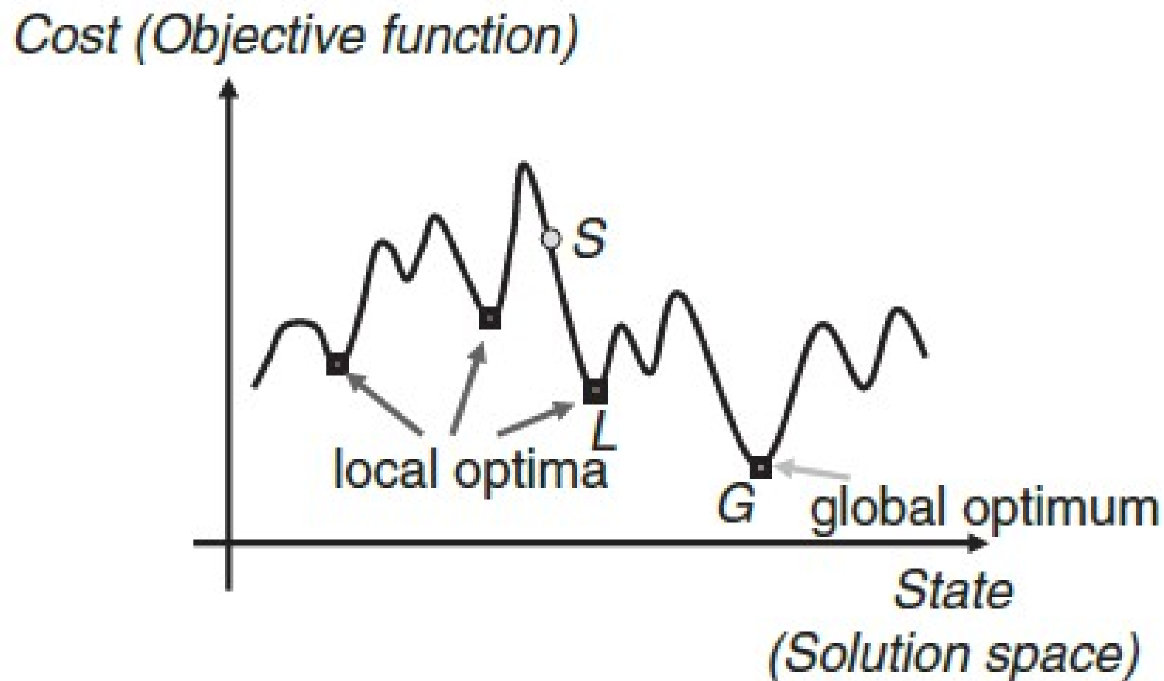


Fig: Illustration of simulated annealing

SIMULATED ANNEALING BASICS

- The probability of accepting a neighboring solution depends on two factors:
  - (1) the magnitude of the uphill move (cost difference)
  - (2) the search time (annealing temperature)
- To implement this idea, the probability of accepting a new solution **S′** is defined by-

$$\text{Prob}(S \rightarrow S') = 1 \text{ if } C < 0 \text{ (down-hill move)}$$
$$= e^{\Delta c/t} \text{ if } \Delta c > 0 \text{ (up-hill move)}$$

- Where, $\Delta c = \text{cost}(S') - \text{cost}(S)$, and T is the current temperature
- Initially, we are assigned a high temperature. As the annealing process goes by, the temperature is typically decreased by a fixed ratio

Simulated Annealing Algorithm for Floorplanning

```
Get an initial floorplan S;   S_Best = S;
Get an initial temperature T > 0;
while not "frozen" do
    for i = 1 to k do
        Perturb the floorplan to get a neighboring S' from S;
        ΔC = cost(S') − cost(S);
        if ΔC ≤ 0 then     // down-hill move
            S = S';
        else     // uphill move
.           S = S' with the probability e^(−ΔC/T);
.       end if
.       if cost(S_Best) > cost(S) then
.           (S_Best) = S;
.       end if
.   end for
.   T = rT;     // reduce temperature
. end while
. return S_Best;
```

There are four basic ingredients for simulated annealing

1. **solution space**
2. **neighborhood structure**
3. **cost function (objective function)-optimization goal**
4. **annealing schedule-captures the temperature change during the annealing process**

Input $Netlist_1$, $Netlist_2$, …, $Netlist_k$

Create new **Floorplan** for each design

Evaluate **Total Costs** of all designs (*frames, congestion, area, wirelength*)

Floorplans **Accepted**?

Yes

Final Floorplan

No

Apply **Moves** (*block, compaction, matching*)

Simulated Annealing

Any Question?

THANK YOU