Design and implementation of move-based partitioners

Andrew Caldwell, Andrew Kahng and Igor Markov

Abstract

In this report we develop the idea that metaheuristics for VLSI move-based partitioning, such as *Kernighan-Lin* passes and simulated annealing, can be built from several reusable components with well-defined interfaces and clear implementation options. For example, the classic net-cut bipartitioning algorithm by Fiduccia and Mattheyses can be structured for efficient implementation with many explicit degrees of freedom by using four components: *incremental evaluator, gain container, move manager* and *pass-based partitioner proper*. We discuss implementation choices and tradeoffs as well as inheritance hierarchies, internal data structures, optimizations, etc.

Contents

Basic Definitions

0.1 We consider balanced k-way partitioning of an edge- and vertex-weighted hypergraph with N vertices. This optimization problem seeks to minimize a given cost function $c(\cdot)$ whose arguments are *partitionings*, i.e. assignments of vertices to k partitions.

The returned solution must satisfy given constraints, in particular, certain vertices can be fixed in some partitions (fixed constraints) and the total weight of vertices in each partition may be limited (balance constraints).[1] In other words, the cost function $c(\cdot)$ is minimized over the set of *feasible solutions* S_f, which is a subset of the set of all possible solutions S.

0.2 Variant partitioning formulations can be addressed by slight modifications of S_f and S:

- In multi-balance partitioning, each vertex is assigned w weights, and independent balance constraints are established for each of the w weights. S_f is restricted to partitionings that meet balance constraints for all weights.
- Kring-Newton replication entails the space S of all partitionings with vertex replications allowed. S_f is restricted from S by balance constraints and limits on replication.

0.3 A *move structure* is assumed on S, i.e., a set of *moves* M_s at each $s \in S$, such that for $m \in M_s$, $m \cdot s = s' \in S$ (i.e., the moves can be applied to the solutions).[2]

[1] Numerous other constraints exist.

[2] The trivial move and the inverse of each move are also moves. The latter can be used in the undo/redo operations of Section 6.

0.4 We distinguish *generic* moves M^g which can be applied to any solution $s \in S$. In other words, M^g is a subset of M_s for any $s \in S$.[3] Moves which are not generic typically have *preconditions*, i.e., can only be applied to some solutions.[4]
0.5 Given a solution $s \in S$ and a move $m \in M_s$, we say that m is *illegal* iff $s \in S_f$ and $m \cdot s \in S \setminus S_f$.[5] Otherwise m is *legal*.[6]
0.6 The *gain of move m from solution s* is defined as $gain_s(m) = c(m \cdot s) - c(s)$.

1 Top-Down Structure of This Document

We can distinguish the following reusable components for design of VLSI partitioning heuristics. The second through fifth components are the four structural components of a traditional pass-based (KLFM-style) partitioner.

Common Partitioner Interface Describes input and output to partitioners without mentioning internal structure and implementation details. Invariants that need to be supported are stated.

Initial Solution Generator A stand-alone component, generating partitionings that satisfy given constraints. Such partitionings can be randomized, possibly having better than average costs.

Incremental Cost Evaluator A stand-alone component computing the cost of a given partitioning and efficiently updating the cost after one move.

Legality Checker A stand-alone component that can verify if a partitioning satisfies a given constraint as well as efficiently determine the legality of a move.

Gain container (for pass-based partitioners only) A container defined by simple interface (i.e. without knowing the KL algorithm). Optimized for efficient storage, quick retrieval and update of gains of moves from the current solution. Does not perform updates by itself and is not aware of the incremental cost evaluator.

Move manager A stand-alone component capable of choosing and applying the one (best, random etc) move. Can perform undo/redo operations on request. If used in pass-based partitioners, incrementally updates gain container.[7] Must report "status information" after each move (e.g. current cost and how each partition is filled). May be controlled by the caller via parameter updates before every move selection (e.g. temperature for simulated annealing).

Accelerated move manager A stand-alone component similar to regular Move Manager, except that it can perform complete passes in one call. Useful when moves are cheap, e.g. when no gain container is needed, as in the AGREED algorithm.

***Pass-based* partitioner (proper)** A stand-alone component implementing *common partitioning interface* that solves partitioning problems by applying incrementally improving passes to initial solutions (possibly randomly chosen) and determining the best solution. A pass consists of moves, chosen and applied by the move manager. Within a pass, a partitioner can request that move manager undo/redo some of the moves.

***Simulated annealing* partitioner** A stand-alone component implementing *common partitioning interface* that solves partitioning problems by applying moves that are first randomly selected and accepted with probability depending on their gain. The gains are computed only for a small number of randomly selected moves, therefore the move manager does not need a gain container to maintain and update gains of all possible moves.

Exhaustive search (small) partitioner A stand-alone component implementing *common partitioning interface* that solves partitioning problems by exhaustive search over feasible solutions with specialized algorithms. Only applicable to very small, potentially very constrained, problem instances where other partitioners may not easily find the optimal solutions.

Dereplicating partitioner A stand-alone component implementing *common partitioning interface* that solves partitioning problems by first assuming that every vertex can go into any partitioning and then greedily removing the alternatives leading to best available solution costs.

[3] Generic moves often allow for additional optimization by storing them in tables, see Section 5.
[4] For example, all single-vertex partition-to-partition moves are generic. So is "uncutting" a net, i.e., putting all of net's vertices into one partition. "Unreplicating" moves in Kring-Newton replication can only be applied to replicated vertices, thus are not generic.
[5] Some heuristics make illegal moves.
[6] If the infeasibility of a solution can be quantified, one can choose moves which will decrease it, eventually leading to a feasible solution. E.g., if one partition is overfilled and the other is underfilled, one moves cells from the former to the latter until the balance is within reasonable boundaries. Similar examples appear in Kring-Newton replication.
[7] Must use incremental algorithms itself, oftentimes independently of evaluator.

In addition to general definitions above, for each component, we discuss "what it is/does", "what it is not/does not do", and describe requirements, interface and implementation details.

2 Common Partitioner Interface

A move-based partitioner is a stand-alone component that solves partitioning problems by iterative improvement and is described by the following model.

2.0.7 Input

- `PartitioningProblem`
  - `HGraph`
  - `PartitioningBuffer` which contains initial solutions
  - Context information, including terminal locations, balance targets and tolerances, etc.
- Parameters
  - Move/Pass/Time limits
  - `Verbosity` level
  - Algorithm-specific options

2.0.8 Output

- New `Partitionings` in the `PartitioningBuffer`
- Which of the solutions is the best (meets the constraints and has the lowest cost)
- Status information (moves made, time used, etc.)

2.0.9 Invariants

- Partitioners must NOT modify `PartitioningProblem` except for the `PartitioningBuffer`.[8]
- Partitioners must iteratively improve all initial solutions passed.

3 Incremental Cost Evaluator

3.1 What it is/does

- A stand-alone component which maintains and provides on request the current total cost of hypergraph and the costs of single nets.[9].
- Maintains "internal state", defined by internal data structures, to facilitate efficient cost updates after single moves.

3.2 What it is not/does not do

- Does not support the concept of gains.[10]

3.3 Requirements

- Must be able to efficiently[11] update the internal state after one move and efficiently compute the cost of the current state.
- For some pass-based partitioners only and not for simulated annealing,[12] can only return values from a small, often pre-defined, set. Further, must report the maximal and minimal value each net can ever return for the given partitioning instance.

[8] In particular, partitioner cannot change `HGraph`. Such changes can be done in the code calling the partitioner.
[9] "local" evaluators are not in the scope of this document, see ~projects/OPTILE
[10] which makes it usable for algorithms other than FM, e.g. simulated annealing.
[11] typically, in constant time.
[12] Required for efficient implementation of prioritizer in gain container. See Section 5 for details.

3.4 Interface

- Initialization (of internal structures and data caches) with a hypergraph and a partitioning solution.
- Reporting current cost (total or of one net) without changing internal state.
- A complete change of internal state (re-initialization) for all vertices and nets
- An incremental change of internal state (for all nets whose cost is affected or for a given net) due to one elementary move without updating the costs.[13]
- Shortcut for simultaneously changing internal state (incrementally) and updating costs.

3.5 Implementation details

3.5.1 The incremental interface can be used even by evaluators which do not lend themselves to incremental computation. At minimum, they will be able to update their internal state incrementally, e.g., with net tallies (see below), possibly allowing for cost precomputation and fast table lookup. The latter may reasonable for an MST- or bounding box-based cost evaluator.
3.5.2 Depending on the algorithm used by the evaluator, the return values may be discrete (e.g., small integers) or not (e.g., floating point), ditto for small and large ranges. On the other hand, depending on whether the values are expected to be discrete in a small range, scaling followed by rounding and type conversion may be needed. Thus, to support reuse in different applications, we say that evaluator components can provide two cost computation methods – one *natural* to the algorithm, the other implemented as scaling and conversion of natural values.
3.5.3 Adding a new move type (esp. if not generic) may or may not require new evaluator implementation.[14]
3.5.4 In the next subsection we distinguish some *generic features* common to many incremental evaluators of practical interest. To ensure code reuse between different implementations, a hierarchy of incremental evaluators can be created. Generic features will then be inherited by specific evaluator classes from common base classes.[15]

3.6 Generic Features

3.6.1 Maintaining *net tallies*, i.e., the numbers of vertices in each partition for each net, is useful for incremental evaluation of partitioning cost functions (e.g. net-cut, bounding box etc.). Net tallies can be quickly updated after a single-vertex partition-to-partition move. Net tallies should be implemented by an evaluator class from which other classes that use net tallies will be derived.
3.6.2 Further net cost computation can be aided by additional incrementally maintained data. Examples are (a) the maximal number of vertices in any one tally for each net and (b) *net vectors* maintaining one bit per tally/partition according to whether the latter is empty.
3.6.3 If the net cost only depends on net vector value, one can precompute costs for all (or the most frequent) values and store them in a table for fast lookup.[16] Such tables will typically be created at evaluator initialization.[17]
3.6.4 In placement-related partitioning formulations, certain vertices of the hypergraph (terminals) are not assigned to any partition and only exist to bias the cost of the nets incident to them. To simplify net cost computation[18] it is often convenient to *propagate* such vertices to one or more partitions as fixed vertices.[19] While terminal propagation is typically done *before* iterative partitioning, some evaluator

[13] Changing the state of one net will most likely make the overall state of the evaluator inconsistent. This can be useful, however, for "what-if" cost lookups when a chain of incremental changes returns to the original state.
[14] For example, "uncutting net" moves can synthesized from single-vertex partition-to-partition moves and possibly do not require a new evaluator (the standard evaluator will be asked to change its state several times and then compute cost). On the other hand, "replicating" and "unreplicating" moves used in Kring-Newton replication cannot be expressed in single-vertex partition-to-partition moves and need to be explicitly supported by an evaluator.
[15] Multiple inheritance may or may not be needed from classes implementing *generic features*. A "top-most" base class may be used to implement the incremental evaluator interface described above in pure virtual methods. No virtualization overhead should be incurred by concrete incremental evaluator classes since the precise class will always be known at compile-time.
[16] The size of the table is crucial for its feasibility. Net vectors often have size $O(k)$ (k is the number of partitions). Some variants have size $O(2^k)$, e.g. for MST cost evaluators.
[17] There are at least two main regimes. *Generic* net vectors are typically used for nets with all incident vertices assigned to partitions; one generic net vector can be used for all such nets. In contrast, *non-generic* net vectors may be used for nets with some incident vertices not assigned to any partition; a separate net vector may be needed for each such net. Tables used to store net vectors may be initialized on demand, during the move-based partitioning.
[18] And to guarantee the discreteness of *gain values*; see Section 5.
[19] For example, a fixed vertex can be represented by an equivalent of, say, 10 regular vertices in a tally, which can be useful for evaluators sensitive to the the biggest tally.

implementations achieve run-time optimizations by recognizing propagated terminals and caching part of the cost computation for nets incident to them (see next subsection).

3.7 Cost-function specific details

3.7.1 For the bounding box evaluator, it is convenient to maintain a pair of net tallies for x and y dimensions, each accompanied with a net vector and a precomputed cost table.
3.7.2 Net cost computation for each net can be optimized by recognizing fixed vertices, e.g. terminals or "locked" vertices, when the evaluator is initialized. Then, portions of net cost computation can be performed once at initialization, results cached, and used every time the net cost is updated.[20]

4 Legality Checker and Initial Solution Generator

4.1 Legality checker is a stand-alone reusable component representing partitioning constraints. Generators of initial solutions satisfying such constraints naturally accompany legality checkers.

4.1 Legality Checker

4.1.1 Given a partitioning, a legality checker can verify if the partitioning satisfies certain constraint.
4.1.2 Similarly to incremental cost evaluator, legality checker can be applied incrementally to test the legality of a move (e.g., assuming that the constraint is satisfied by the current partitioning, will be still satisfied after a given move). The "cost" returned by a legality checker for a move is boolean value, true iff the move is legal.
4.1.3 Legality checker(s) will be normally applied to accept or reject moves selected on the basis of their cost.

4.2 Initial Solution Generators

4.2.1 Initial solution generators create partitionings satisfying given balance constraints, typically randomized, often with better than average costs.
4.2.2 Initial solution generators are parallel to legality checkers in the sense that for every legality checker, there will normally be a generator of solutions satisfying the corresponding partitioning constraint.
4.2.3 Unlike legality checkers, efficient[21] initial solution generators often can not be easily composed to produce solutions satisfying multiple constraints.

5 Gain Container

5.1 What it is/does

- Is specific to pass-based partitioners.
- Stores all moves currently available and prioritizes them by the immediate effect each would have on the total cost, i.e., the gain.
- A container of move/gain pairs is defined by simple interface, i.e., not in terms of a particular algorithm (e.g., FM).
- Allows quickly updating each move/gain pair after a single move.
- Allows finding the move with highest gain, possibly subject to various constraints such as a given source or destination partition etc.
- May provide support for *tie-breaking schemes* in order to choose the best move among the moves with highest gain.

[20]For example, precomputing with caching, and using *PartitionIDs* bit-vectors, are two alternatives for k-way net cut computation without net tallies:

- If the net has fixed vertices, fix their partitions (if they cannot be all in one partition, the net cut is 1). Assume that each movable vertex is in exactly one partition. If the first movable vertex is in one partition with all terminals, check whether all other movable vertices are in the same partition.
- Any vertex can be in any number of partitions and is represented by a *PartitionIDs* bit-vector having 1s in for the partitions that have this vertex and 0 for those that do not. The net is cut if and only if the bit-wise "and" of all *PartitionIDs* on a given net is 0. The computation can be sped up by checking for 0 after every "and" operation, because during incremental improvement of a randomly chosen initial solution, most of the nets are cut.

Our experiments show that using *PartitionIDs* bit-vector is slightly faster, even though this computation is not incremental.
[21]Typically, legal initial solutions must be generated in linear time.

5.2 What it is not/does not do

- Does not perform updates by itself.
- Is not aware of incremental cost evaluator.
- Is not aware of move manager or how the gains are interpreted.
- Does not determine legality of moves.[22]
- Is not used in simulated annealing.

5.3 Requirements

- Implementations described here require that gains take values from a small, discrete, set.
- Requires that the maximum possible gain of a move can be determined given an instantiated cost evaluator.
- For any move, provide a test to see if the move is in the container (i.e., if there is a move/gain pair with this move).
- For any move with computed gain, allow for constant-time addition of the corresponding move/gain pair to the gain container.
- For any move in the gain container, allow for constant-time gain lookup and updating to a given value.
- Allow finding the move with highest gain, possibly subject to various constraints, in constant time.

5.4 Interface

- Add a move/gain pair with computed gain.
- Find the move/gain pair, given a move.
- Get the gain for a move/gain pair.
- Set the gain for a move/gain pair (e.g. to update).
- Remove a move/gain pair.
- Find the highest gain move.
- Invalidate current highest gain move to request the next highest gain move.[23] Typically applied if the current highest gain move appears illegal.
- Invalidate current highest gain bucket to access the next highest gain bucket.
- Some GainContainers may support finding the *first* move/gain pair with a given property.
  Sample properties: moves of a given vertex, moves to or from a given partition.

5.5 Implementation Details

5.5.1 The primary components of Gain Containers are *repositories* and *prioritizers.*

Repository for gain/move pairs handles allocation and deallocation of move/gain pairs, fast gain lookups given a move, and gain updates given a move and a new gain value.

Prioritizer chooses the move with highest gain. In addition, may be able to choose choose best-gain moves among moves with certain properties, such as a particular destination or source partition. Is responsible for tie-breaking schemes.

5.5.2 We say that some moves stored in the repository are *prioritized* when they participate in the prioritizer's data structures.
5.5.3 Not prioritizing the moves affecting a given cell is commonly referred to as "locking" the cell. FM-type heuristics routinely lock a cell after it is moved, however, cells may be locked and unlocked by additional heuristics.[24]
5.5.4 There are two general organizations of repositories and prioritizers:

[22] This makes gain containers designed for common partitioning formulations equally fit for multi-balance partitioning and other formulations. Also see Section 4.
[23] Note that this does not remove the move from the Gain Container.
[24] For example, cells not incident to any cut nets may be locked in the beginning of the pass if the goal is only cut refinement.

- Storage units for each move begin in the repository, then are removed and added to the prioritizer. When a move is removed from the prioritizer (i.e., the node it corresponds to is "locked"), it is returned to the repository.
- Storage units are permanently owned by the repository. The prioritizer only references them (either explicitly with pointers, or by embedding its data structures, e.g. linked lists, into the repository).

5.5.5 When gains are small integers, the prioritizer is most naturally implemented with bucketing. Typically, this entails an array of size $maxGain * 2 + 1$ of doubly-linked lists. Each list contains all moves of a given gain and is known as a *bucket*.[25]
5.5.6 Support for choosing highest gain nodes

- When the caller (typically move manager) identifies the current highest gain move as illegal (typically using legality checkers), it invalidates the highest gain move in the gain container so that the next highest gain move can be requested and checked for legality. A whole bucket can be invalidated as well.
- The Gain Container maintains a counter with the index of the highest gain bucket with at least one move in it, and a counter with the index of the highest gain non-empty bucket currently being considered for the purpose of finding the highest gain move.
- Gain Container maintains a pointer to the highest gain element in the highest gain bucket of those that have not been invalidated.
- The Gain Container supports two functions which change the index of the highest gain bucket and the move pointers used to find the move to take.
  - InvalidateMove: removes from consideration just the move which previously had highest gain.
  - InvalidateBucket: removes from consideration the whole bucket containing the previously highest gain move.
- The functions which update the gains reset the counters and pointers.

5.5.7 Another design aspect for both the repository and the prioritizer is the support for selected *properties*.

- To link move/gain pairs with certain properties, gain container implementations often have embedded lists. This is compatible with storing move/gain pairs in a table and can be used to co-implement the repository and the prioritizer. The list and the table can support associations by independent properties, e.g. the lists can link moves of cells on same nets, while the table can be indexed by the destination partition of moves.
- Another way of linking move/gain pairs with important properties is to implement the repository as several lower-level containers. For example, all moves sharing the source partition can be stored together in a table, indexed by the destination partition. Such lists need to handle insertion and deletion with one argument, which implies doubly-linked lists.
- Prioritizer can also be implemented with several separate lower-level components, e.g. bucket structures. This allows to query for moves with highest gain among moves with certain property (e.g., a given source partition).

5.5.8 Gain container can be *randomized* by changing tie breaking in the prioritizer. The simplest and safest randomization is achieved by populating gain container with move-gain pairs in random order.[26]

5.6 Repository Implementations

5.6.1 While this subsection primarily applies to the move structure of FM-type heuristics (i.e. single-vertex partition-to-partition moves), similar implementation options (possibly, with different preferences) are available for some other move structures.
5.6.2 The two proposed implementations are:

- One or several linked lists of move/gain pairs that are not currently prioritized, and a $N \times k$ array of pointers to them to provide fast lookup by move.
  - No *embedded* linked lists are required (simpler implementation).

[25] For cost functions where gains are not small integers and with look-ahead levels, a bucketing approach is not possible, and a tree structure, hash-table or a priority queue is more appropriate.

[26] Randomized buckets have been considered in the literature which equiprobably return one of their elements. They may conflict with certain algorithmic details, e.g. CLIP FM, and typically do not provide convincing improvement.

  - Move/gain pairs are represented by arbitrarily and separately allocated nodes of linked lists. This implies system memory overhead for allocation and usually does not provide good processor cache locality as move/gain pairs affected by the same moves are not necessarily neighbors in memory.
- Implementation built with a $N \times k$ array of move/gain pairs.
  - Linked lists can be re-threaded through the 2-D array without requiring node-by-node allocation.
  - The vertexId can be removed from the move/gain pair in this scheme, lowering the memory requirements. The vertexId can be calculated given the pair's address, the start address of the container, N and k.
  - This implementation ensures the move/gain pair will only be allocated and de-allocated once, without the need for maintaining a list of move/gain pairs not prioritized.[27]
  - No extra array of pointers is necessary for fast lookup of move/gain pairs by move.
  - Ordering the array well may improve processor cache locality (thus speed-up) by ensuring that move/gain pairs corresponding to the same vertex[28] are neighbors in memory.

5.6.3 Within an FM-specific implementation, our preference is for the latter option. It should provide lower memory usage and overall faster access time. It may have problems for some move structures (i.e., KL has k^2 moves, which we would not want to pre-allocate). We are not aware of a published account of this structure. It does not appear in any of the source code we reviewed.

5.7 Prioritizer Implementation

5.7.1 In this section we assume that prioritizer is implemented in the context of FM algorithm with several arrays of buckets of linked-lists of move/gain pairs.[29] However, the implementation options are likely to apply to other move structures as well, especially if they include single-vertex partition-to-partition moves.
5.7.2 The prioritizer consists of k bucket arrays, one for each destination partition (a k-component *destination-centric* prioritizer[30]).
5.7.3 The current maximum-gain move in each array is stored to enable efficient finding of the overall maximum. We may wish to build a priority queue or similar structure for finding finding the overall maximum.
5.7.4 Each of the above repository designs requires its own prioritizer implementation.
5.7.5 If the repository consists of linked lists of move/gain pairs:

- Prioritizer has k arrays of lists of move/gain pairs. This should be implemented using an array of STL lists, templated by the move/gain pair class.
- Updating a move/gain pair's gain is implemented as:
  - Find the move/gain pair using the vector of pointers in the repository.
  - Remove the pair from the linked list (its bucket).
  - Add the move to the bucket (usually at the head) corresponding to the new gain in the array for the destination of the move.

5.7.6 If the repository consists of an $N \times k$ array of node/gain pairs:

- Prioritizer has k arrays of pointers to move/gain pairs. This cannot be implemented as an array of STL lists. Rather, the gain/node pairs have pointers to the next/previous pairs in their buckets.
- Updating a move/gain pair's gain is implemented as:
  - Find the move/gain pair by computing the offset from the repository.
  - Remove the move from the linked list.
  - Add the move to the bucket (usually at the head) corresponding to the new gain in the array for the destination of the move.
  - Note that the list functions must be built into the move/gain pairs, rather than being functions of an STL list.

[27] Removing a move/gain pair from the lower-level container is slightly cheaper, as there is no need to add it to a linked lists.
[28] More generally, move/gain pairs whose updates are caused by the same moves.
[29] which assumes very discrete gains and, most likely, no higher gain levels.
[30] See Appendix B for a more detailed description of prioritizer implementation choices.

6 Move Manager

6.1 What it is/does

- A stand-alone component capable of choosing and applying the best move (typically, the best *legal* move).
- Can perform undo/redo operations on request (mostly for pass-based partitioners).
- Must report relevant "status information" after each move (e.g. current cost and how each partition is filled).
- May be controlled by the caller with parameter updates before every move selection.
- Incrementally updates gain container after single moves.
- Is instantiated with one incremental evaluator and one gain container at once.
- Essentially, handles the move structure.

6.2 What it is not/does not do

- Does not evaluate cost of cost updates for individual nets (rather calls evaluator).
- Does not decide when to stop passes and perform undo/redo operations.
- Cannot be used with multiple gain containers or evaluators simultaneously.

6.3 Requirements

- Must use an incremental algorithm for gain update (the recommended algorithm is described below).
- Must efficiently deal with move legality.

6.4 Interface

- Choose one move (e.g., the best feasible) and apply it. Ensure all necessary updates (gain container, incremental evaluator). Return new "status info", e.g., total cost, partition balances, etc, and log the move (to facilitate undo/redo operations).
- Undo a given number of moves.
- (Possibly) get the current "status info".

6.5 Implementation details

6.5.1 Before initializing gain containers, need to initialize the evaluator.
6.5.2 Move manager will typically be parameterized (templated) by one gain container and one evaluator, both compatible with its move structure.
6.5.3 Since move manager will be called by partitioner which will not be templated, there needs to be an abstract base class defining all of the move manager's interface by pure virtual methods.
6.5.4 Before applying a move, must update gains in gain container "as incrementally as possible" and only using incremental cost computations (not gain computations) by evaluators. The recommended algorithm follows:

```
1. For each vertex actually moved (e.g. once for a single-vertex move)
2.   For each net incident to the vertex actually moved
3.     compute the old cost of the net
4.     compute the cost of the net with the vertex actually moved
5.     For each vertex incident to the net
6.       compute the hypothetical cost of the net for each possible move of the vertex,
         with and without the other vertex actually moved
7.       Use the 4 net costs for the net to update the gain of the vertex
```

Note: The gain of a vertex will be updated once for each net it shares with the vertex moved.

Note: Incrementally changing the internal state of the cost evaluator without computing the cost is useful in 6. as well as to return the evaluator into the original state after 6.

6.5.5 To enable undo/redo operations, all moves applied need to be logged by the move manager. For an undo request, some number of logged moves are reversed (most recent move first) and applied to the current partitioning. When the number of undo moves is large, it is more economical to not update the gain container and evaluator incrementally, but rather re-initialize them with the result of the undo operation. A redo request may be issued when it is cheaper to replay some moves of the pass forward, rather than undoing from the end of the pass.

7 Pass-Based Partitioner (Proper)

7.0.6 A pass-based partitioner applies incrementally improving passes to initial solutions (possibly, chosen at random) and determines the best outcome.
7.0.7 A pass consists of moves, chosen and applied by move manager.
7.0.8 After a pass, a partitioner can request that move manager perform undo/redo operations.
7.0.9 Partitioner decides when to stop a pass and the details of undo/redo operations on the basis of its control parameters and status information returned by move manager after each move.
7.0.10 Partitioner can have access to multiple combinations of incremental evaluators, move managers and gain containers, and can use them at different passes to solve a given partitioning problem. The incremental cost evaluator, move manager and gain container are determined by partitioner's control parameters.

7.1 What partitioner proper is not/does not do

- Is not required to support gain computation and update.
- Does not choose best moves or apply them.
- Does not perform undo/redo operations.
- Should not be parameterized (templated) by other classes.
- Is not aware of details of the move structure used (i.e., moves).[31]

7.2 Requirements

- Must work with standard solution buffers.
- Sufficient flexibility in parameter processing to accommodate various combinations of incremental evaluators, move managers and gain containers.
- Sufficient reporting of operating parameters, e.g., CPU time spent on various stages.

7.3 Interface

- Takes a `PartitioningProblem` and operational parameters on input.
- Writes solutions into `PartitioningProblem` and sets the number of the best solution.
- Prints operational parameters depending on verbosity value.

7.4 Implementation Details

7.4.1 The control parameters will need to use "registry classes"[32] for incremental evaluators, gain containers and move managers. FM-partitioner will have several nested big switches to instantiate the correct move combinations of incremental evaluators, gain containers and move managers (potentially a few hundred lines, every third instantiating a template — will take ages to compile).

8 Simulated Annealing Partitioner and Supporting Move Manager

8.0.2 Specifications very similar to those in Section 7. However, an SA partitioner does not necessarily use improving passes with deterministically chosen moves. The simulated annealing algorithm first randomly selects a move and accepts it with probability depending on the move's gain, moves are selected until one is accepted. This process is repeated multiple times, possibly with variations in parameters affecting move acceptance.

[31] It just needs to instantiate a relevant move manager with matching evaluator and gain container.

[32] Essentially, those are global `enums` conveniently wrapped into classes and having their values in their own namespaces. Will need to be updated with every new evaluator class, gain container etc just as the code that switches on them. Backwards compatibility is guaranteed if no values are removed from registry classes.

8.0.3 Since it is the move manager that selects and applies moves, the simulated annealing partitioner will need a specially designed move manager.
8.0.4 Gains are computed only for a small number of randomly selected moves (from scratch), therefore the move manager does not need a gain container to maintain and update gains of all possible moves.
8.0.5 *Simulated annealing* partitioner is not likely to request undo/redo operations, therefore the move manager does not necessarily have to log all moves it applies.
8.0.6 Algorithms for determining random move selection (e.g. types of moves that are more likely to be accepted) and initial values of parameters for acceptance probabilities (e.g., temperature) may be encapsulated into a separate component, which is likely to call move manager.

8.1 Control Parameters

include, but are not limited to

- target acceptance scheme
- temperature schedule scheme
- initial temperature
- initial target accept rate
- final target accept rate
- total number of moves
- number of iterated descents
- kick move size (fixed)
- history window size (used to find the next temp)

9 Exhaustive Search (Small) Partitioner

9.0.1 A stand-alone component implementing *common partitioning interface* that solves partitioning problems by exhaustive search over feasible solutions with specialized algorithms.
9.0.2 Strongly relies on incremental cost evaluation and efficient enumeration of the solution space with, e.g., Gray codes.
9.0.3 Only applicable to very small problem instances where other partitioners may not find the optimal solutions ("combinatorial" instances).
9.0.4 The implementation of an exhaustive search partitioner is influenced by how much smaller the subspace of feasible solutions S_f is compated to solution space S (i.e. whether it is faster to search for feasible solutions among all or generate feasible solutions directly).
9.0.5 Branch-and-bound algorithms seem especially suited for exhaustive search partitioners. A good solution produced by a heuristic partitioner can be used to reduce the search space from the very beginning. Bounding on partition balances can also be performed.

The idea of the algorithm is to assign cells to partitions one by one (i.e. this is not an iterative algorithm, as we do not have a complete partitioning most of the time). "Partial partitionings" are evaluated by inducing hypergraph over modules already assigned, i.e. unassigned modules are ignored. When more modules are placed, the cost and the partition fills can only increase, hence the branch-and-bound algorithm.
9.0.6 Initial solutions can be ignored by exhaustive search partitioner.
9.0.7 To comply with the output requirements of the *common partitioning interface*, exhaustive search partitioner can attempt to return as many different solutions with optimal cost, as initial solutions it has been given. Remaining solution slots can be filled by duplicating solutions.[33]
9.0.8 Since all solutions returned have the best cost, the "best solution number" required by *common partitioning interface* can be the number of the last solution that was not duplicated.[34]

10 Dereplicating Partitioner

10.0.9 A stand-alone component implementing *common partitioning interface* that solves partitioning problems by first assuming that every vertex can go into any partitioning and then greedily removing the alternatives in order to achieve the best available solution costs. Ties can be broken arbitrarily.

[33] This, for example, allows for one solution duplicated many times.
[34] This assumes that all different solutions are mentioned first, and then go duplicates.

10.0.10 The algorithm is driven by the need to "dereplicate" vertices, even if this increases solution cost, and stops when each vertex is assigned to one partition.
10.0.11 There are no undo/redo operations, and there is essentially one pass.[35]
10.0.12 A specialized Move Manager is required to support the move structure of "dereplications", as well as a specialized gain container.
10.0.13 Specialized cost evaluators can be constructed that correspond to classic cost functions, e.g. net-cut. In fact, some implementations of classic cost function evaluators are directly usable in dereplicating partitioner.
10.0.14 Initial solutions can be ignored by dereplicating partitioner.
10.0.15 To comply with the output requirements of the *common partitioning interface*, dereplicating partitioner can attempt to return as many different solutions with optimal cost, as initial solutions it has been given. Remaining solution slots can be filled by duplicating solutions.[33]
10.0.16 Since all solutions returned have the best cost, the "best solution number" required by *common partitioning interface* can be the number of the last solution that was not duplicated.[34]

Appendix A: Class Hierarchy for Incremental Evaluators

- `class NetCutWBits : public PartEvalInterface`
- `class NetTallies : public PartEvalInterface`
  - `class NetTalliesWCosts : public NetTallies`
    `class NetTalliesWConfigIds : public NetTalliesWCosts`
    - `class NetCutWConfigIds : public NetTalliesWConfigIds`
    - `class NetVecGeneric : public NetTalliesWConfigIds`
      - `class NetCutWNetVec : public NetVecGeneric`
      - `class BBoxWNetVec : public NetVecGeneric`

Appendix B: Prioritizer Implementation Options

There are 4 primary organization choices for arrays of buckets in the prioritizer:

- Single-component prioritizer
  - Advantage: easier (possibly faster) to find the vertex with the highest gain. Other options require finding the highest gain vertex from among multiple structures.
  - Disadvantage: does not easily allow specifying the source or destination of the move to be selected.
  - Disadvantage: memory intensive. The move/gain pairs must each store the destination of the move (but the need not store not the source, though our k-way partitioner does).
- k-component *source-centric* prioritizer
  - Advantage: easily allows for specifying the source of the move (useful for a "balancing phase").
  - Disadvantage: memory intensive. Each move/gain pair must store the destination of the move (but not necessarily the source).
  - Disadvantage: does not easily allow specifying the destination of the move to be selected.
- k-component *destination-centric* prioritizer
  - Advantage: easily allows for specifying the destination of the move to be selected.
  - Advantage: less memory required. Does not require each move/gain pair to store the destination of the move (and the source is never required in each move/gain pair).
  - Disadvantage: does not easily allow specifying the source of the move to be selected.
- $k(k-1)$-component *isotropic* prioritizer
  - Advantage: easily allows for specifying the source and/or destination of the move to be selected.
  - Advantage: less memory required. Does not require each move/gain pair to store the destination of the move (and the source is never required in each move/gain pair).

[35] Multiple solutions can be generated by randomly breaking ties or skipping the greedy option. The complexity of the algorithm is comparable to that of first $k-1$ passes of typical pass-based partitioners; fair comparison of solution quality would be against results of such $k-1$ passes.

  – Disadvantage: requires choosing among $k(k-1)$ "bests" to find the best move out of $k(k-1)$ prioritizer components (can be beaten by maintaining max gain with a change to every component's max gain).

What other people used:

- Original Laura Sanchis k-way partitioner used the $k(k-1)$ setup. Note that she *does* store both the source and destination in the move/gain pair though (totally unnecessary...there's not even a speed gain from doing this).
- QUAD used the same option as Laura Sanchis.
- Our k-way FM uses single-component prioritizer, apparently no other implementations do. It does store both the source and destination in each move/gain pair.
- Various 2-way FM implementations all use k-component *destination-centric* prioritizer.