FACULTY OF ENGINEERING AND ARCHITECTURE

Layout generation using simulated annealing and A star

Jorne De Schepper
Student number: 01902405

Supervisors: Prof. dr. ir. Guy Torfs, Prof. dr. Xin Yin
Counsellors: Prof. dr. ir. Johan Bauwelinck, Caro Meysmans

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Electrical Engineering - Electronic Circuits and Systems

Academic year 2023-2024

GHENT UNIVERSITY

PREFACE

Dankwoord

I would like to sincerely thank ir. Caro Meysmans for the many useful brainstorm sessions, amounting to important insights and for always being ready to offer assistance, regardless of the matter at hand. I also want to express my gratitude to Prof. dr. ir. Guy Torfs for his pristine advice and for continuously challenging me, hence, elevating my knowledge. Thanks to them, my masters dissertation has been completed, marking the pinnacle with which I could conclude my education in electrical engineering. This awesome topic fulfilled my passion for software design, while requiring my accumulated comprehension of electronics. The acquired insights will accompany me into my budding career as an electrical engineer.

Also a huge thanks goes to my classmates for creating such a fantastic atmosphere and for always providing insightful ideas and critically exchanging thoughts.

Additionally, I want to thank my parents for always granting me the freedom to pursue my diploma in my own determined and somewhat unorthodox manner. Lastly, I would like to thank Justine for her unwavering support and trust, and for always listening to my extensive explanations (ramblings) about my thesis and all other irrelevant topics.

Jorne De Schepper
23/05/2024

STATEMENT

This master's dissertation is part of an exam. Any comments formulated by the assessment committee during the oral presentation of the master's dissertation are not included in this text.

ABSTRACT

This master's dissertation is authored by Jorne De Schepper, supervised by Prof. dr. ir. Guy Torfs, Prof. dr. ir. Xin Yin and counselled by Prof. dr. ir. Johan Bauwelinck and ir. Caro Meysmans. The dissertations involves an automatic layout generator which employs simulated annealing and A-star and is submitted to obtain the academic degree of Master of Science in Electrical Engineering - Electronic Circuits and Systems.

Abstract – This master's dissertation proposes a design automation tool for analog or mixed-signal design in CMOS technology. This layout generator was developed with the goal of being used together with an IC designer. By introducing many design parameters, an optimal DRC-clean layout can be obtained tailored to the precise requirements. Provided a well designed schematic, the primitives are extracted and drawn according to the design rules. An optimization process employing simulated annealing, performs the routing and placement simultaneously, minimizing the weighted cost of the global area and parasitic resistance of the metal traces. This involves continuously introducing adaptations to the layout by executing actions. Moving primitives, overlapping interconnects, changing routing order are all examples of these actions which improve the cost. Each time the actions are performed, the routing is executed. An improved A* algorithm allows connecting all nets by following the path which adds the least parasitic resistance. The layout generator provides the engineer with relevant data about each obtained layout. Besides that, a calibration process is available to normalize the cost. This way the many design parameters can easily be adjusted. This is all demonstrated and verified by a typical analog circuit, the operational amplifier.

Keywords – Layout generation, analog integrated circuits, electronic design automation, CMOS layout, simultaneous placement and routing

Automatic layout generation with simulated annealing and A*

Jorne De Schepper - 01902405

Supervisors: Prof. dr. ir. Guy Torfs, Prof. dr. Xin Yin
Counsellors: Prof. dr. ir. Johan Bauwelinck, ir. Caro Meysmans

Abstract— **This extended abstract proposes a design automation tool for analog or mixed-signal design in CMOS technology. This layout generator was developed with the goal of being used together with an IC designer. By introducing many design parameters, an optimal DRC-clean layout can be obtained tailored to the precise requirements. Provided a well designed schematic, the primitives are extracted and drawn according to the design rules. An optimization process employing simulated annealing, performs the routing and placement simultaneously, minimizing the weighted cost of the global area and parasitic resistance of the metal traces. This involves continuously introducing adaptations to the layout by executing actions. Moving primitives, overlapping interconnects, changing routing order are all examples of these actions which improve the cost. Each time the actions are performed, the routing is executed. An improved A* algorithm allows connecting all nets by following the path which adds the least parasitic resistance. The layout generator provides the engineer with relevant data about each obtained layout. Besides that, a calibration process is available to normalize the cost. This way the many design parameters can easily be adjusted. This is all demonstrated and verified by a typical analog circuit, the operational amplifier.**

Keywords— **layout generation, analog integrated circuits, electronic design automation, CMOS layout, simultaneous placement and routing**

I. Introduction

ELECTRONIC design automation (EDA) has always been pursued for analog integrated circuit (IC) design. As scaled technologies become more and more complex, the constraints on the layout increase too. The Design Rules Check (DRC) deck for 28nm technologies consist of 10000 rules, therefore automating layout design can aid an IC designer by drastically cutting down the development time.

Aside from the digital counterpart, the design automation of analog complementary metal-oxide-semiconductor (CMOS) devices is less adopted, because fully automated designs limit the control of the IC engineer, which is required for these cutting edge designs. The performance requirements of these layouts is too high and the EDA tools do not reach these high specifications. Therefore this thesis aims to provide a layout generator which allows total control by the IC designer. This way the resulting layout can be tailored to the requirements of the schematic.

The first step in generating an IC layout requires identifying the primitives, such as a differential pair or a current mirror, among the various device combinations. What follows is the creation of the floor plan of these primitives, based of their specific design parameters. The primitive layout should incorporate optimization techniques like interdigiting structures, merging devices and improving matching by including symmetry. The final and most influential step involves carefully routing and placing the primitive devices to ensure the layout induces minimal parasitic effects and meets the design specifications.

Many layout generators have been developed over the years employing various approaches. In [5], [9], [8] a template-and-grid based structure was utilised which involves primitive templates specific to a technology with predefined placement grids. Two equal primitives in two different technologies will have the same absolute coordinates but different placement grids. This ensures that layout objects are placed consistently. After an optimal placement, the routing is performed on-grid, symmetrically across the templates containing the devices.

BAG2 [2] introduces a generative approach. Rather than designing a single circuit instance, the designer encapsulates their methodology in the form of an executable circuit generator. This high level generator is able to produce schematics and layouts from given input specifications. Utilizing these generators, designers can implement fully automated design iteration loops.

In [4], [3] an optimization based method is developed in these open-source, fully automatic layout generators. From an unannotated netlist, a hierarchical representation is obtained. From a list of primitives templates, the optimal one is chosen and placed via an analytical method which takes into account the many constraints, which are captured from the netlist and the design rules. This method minimizes parasitics and utilizes symmetry and other layout conventions. The routing employs an integer linear programming method to find the optimal path.

In contrast to current state-of-the-art layout generators, this work proposes a layout generator which optimizes both placement and routing simultaneously. The tool is meant to be used aside an IC designer, allowing much outside control. The schematic of an operational amplifier (opamp) will be used to generate a layout and simulations are performed in Virtuoso.

Section II discusses the preliminaries and general concepts employed by this layout generator. The third section III explores the developed method to detect, place and route primitive cells. Then results are displayed in section IV and conclusions are addressed in section V.

II. Preliminaries

This layout generator is an extension of IDcircuits, an inhouse developed tool to generate DRC-clean layouts of certain primitives. The codebases uses Python to interface with Cadence Virtuoso. A differential pair as generated by IDCircuits is shown in figure 1. To achieve optimal matching, the transistors are arranged symmetrically and enclosed by dummy devices. Additionally, a guard ring is added for proper isolation. Rails above and below the layout primitive serve as interfaces for the inputs and outputs of the primitive.

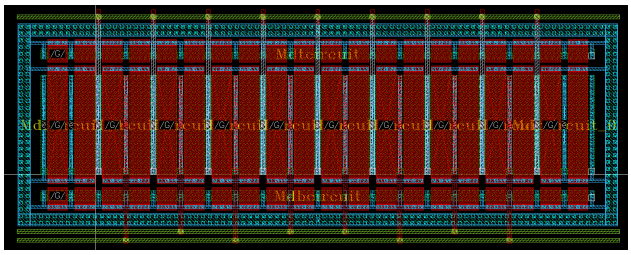


Fig. 1. Differential pair layout generated by IDcircuits

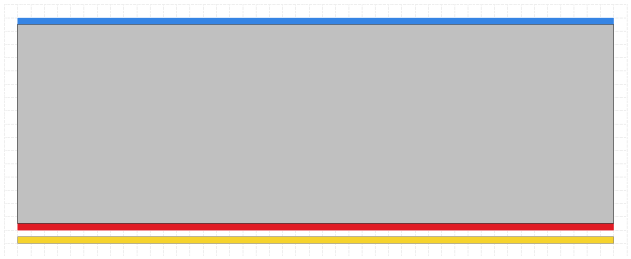


Fig. 2. Abstract representation of a differential pair primitive

In figure 2 the abstract representation of the same primitive is displayed. The active area and the primitive level routing are represented as a grey rectangle. The other rectangles represent the rails. This abstraction provides simplification while containing the important information for the placement and router.

As the grey rectangle is DRC-clean, the only restrictions on the layout are the relative placement of the primitive and the metal traces. Primitives with different bulk biasing have to be spaced apart and distinct traces and vias have to meet spacing requirements as well. The spacing of these traces is seen as the measurement unit for the grid structure, which is required for the routing. This means that the abstraction quantizes the primitive cell, with the smallest distance being 1: the metal spacing.

A. General search method

As the layout is obtained via an optimization algorithm a figure of merit is required to gauge potential layouts. However, it is essential to understand that the concept of an optimal design is heavily reliant on the considered circuit. Optimal could involve having shortest interconnects, the least surface area or minimised parasitics. The employed cost is calculated as a weighted sum of the total area and the parasitic resistance of rails, traces and vias. These weights should fit the circuit requirements.

The cost of the layout is being minimised by simulated annealing (SA) [7], a well known global optimization technique, which is able to avoid local minima. SA proposes a chaotic, completely random starting position and performs actions to lower the global temperature T. These actions perform random changes to the placement with the goal of decreasing the cost.

This algorithm 1 implements SA for both placement and routing. An initial placement is routed, after which the cost and amount of unrouted nets is calculated. Then actions alter this initial configuration to obtain a new circuit, which is also routed and evaluated. If the new layout has less unrouted nets, it is accepted as the current layout. Each time a new layout is adopted, the temperature decreases: $T_{i+1} = \alpha T_i$. If the current amount of unrouted nets is equal, the cost of the new layout is compared with the current one. A lower cost also means adopting the new layout. This process is continued until T reaches T_{end} or the iterations exceed the maximum amount.

To avoid local minima, uphill movement is introduced, by accepting a layout with a higher cost. A Boltzmann distribution 1 determines by chance whether uphill movement is tolerated.

$$\varepsilon < e^{-\Delta C/T_i} \tag{1}$$

If the difference in cost is small, while the current temperature is still quite high, the probability of uphill movement is high. As the solution converges, this change decreases. It should be noted that ε follows the normal distribution.

The choice of T_{start}, T_{end} is crucial to determine how the process converges. This interval and α determine how many improvements are required. These parameters have default values [1], but can be changed by the engineer.

Algorithm 1 Simulated Annealing for placement and routing

$T_0 \leftarrow 100$
$T_{end} \leftarrow 0.01$
$\alpha \leftarrow 0.95$
$i \leftarrow 1$

$P_0 \leftarrow$ Initial placement
$U_0 \leftarrow route(P_0)$ ▷ U: amount of unrouted nets
$C_0 \leftarrow cost(P_0)$
while $T_i > T_{end}$ and $i < 5000$ **do**
  $P_i = P_{i-1}$
  $performActions(P_i)$
  $U_i \leftarrow route(P_i)$
  $C_i \leftarrow cost(P_i)$

  Evaluate new placement P_i
  if $U_i < U_{i-1}$ **then**
    $T_i = \alpha T_{i-1}$ ▷ Accept new placement
  else if $U_i = U_{i-1}$ **then**
    if $C_i < C_{i-1}$ **then**
      $T_i = \alpha T_{i-1}$ ▷ Accept new placement
    else
      $\Delta C = C_i - C_{i-1}$
      if $\varepsilon < e^{-\Delta C/T_i}$ **then** ▷ $\varepsilon \sim U(0,1)$
        $T_i = \alpha T_{i-1}$ ▷ Accept new placement
      else
        $P_i = P_{i-1}$ ▷ Reset placement
      end if
    end if
  else
    $P_i = P_{i-1}$ ▷ Reset placement
  end if
  $i = i + 1$
end while

The actions require a random character and should increase the diversity of obtainable layouts. Once again the designer determines which actions are executed more frequently and also how many actions a single iteration can perform at a time.

B. Routing

The A* algorithm [6] is employed to connect primitives. This algorithm is able to find the shortest path, via a heuristic search, while avoiding obstacles. A grid is traversed by visiting the neighbours of the current node and assigning a score to them. The next step is determined by finding the visited node with the current lowest score. This score is the sum of the actual distance from the start to the node and an estimate between the node and the goal, which is calculated by a heuristic.

A* is very well known, however, out of the box it is not viable for routing integrated circuits. Multiple adjustments are required to allow A* to route an IC layout. The first limitation is that it connects only 2 single nodes. Realistic IC routing involves routing different nets, which often interconnect more than two primitives. Besides the routing of more than two rails, plain A-star is also not able to route whole rails. Therefore, a point on the rails to be routed will have to be selected.

An IC routing algorithm also has to deal with different constraints on the present obstacles and traces. Plain A* simply routes around present obstacles, which means the validation of neighbouring nodes will have to be extended to ensure DRC clean routing.

Aside from all the aforementioned limitations, the most limiting factor is the 2D grid. The fact that IC routing uses multiple layers, has implications on both the heuristic and exploration. An IC layout should strive for straight interconnections, avoiding zigzag routing as much as possible. A convention that is employed in (digital) IC routing is using even and odd layers for respectively vertical and horizontal movement or vice versa. This convention allows to find the optimal path faster.

3D routing requires a heuristic other than the Manhattan distance, due to the introduction of vias. The 'distance' between two metal layers is not the same as the distance between two nodes, due to the vias adding more resistance. Besides that, higher metal layers add less resistance for the unit distance, which means the distance between two cells changes too.

III. Implementation

A. Primitive detection

The first step is to detect the primitives present in the schematic. The detection logic revolves around recognising patterns associated with various primitives and determining which combinations of devices correspond with which specific primitive. A dictionary stores the nets connected to each terminal of each MOSFET, the specific transistor type and whether the gate contains a digital net.

The algorithm iterates over the dictionary and performs sequential checks on the devices. If a condition is fulfilled, the devices and transistor type are assigned to the corresponding primitive. Detected devices are removed from the iteration.

The first condition verifies whether the current device is a dummy. Dummy devices are single transistors which have the ground net or supply net at each terminal.

The next checks examine if the current device is the input device of a current mirror. The detection is based upon the presence of a diode-connected device, i.e., the gate and drain are shorted. All devices sharing the gate connection are included in the current mirror. An example of a current mirror and a dummy device are shown in figure 3.

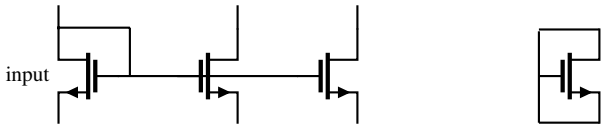


Fig. 3. Left: nmos current mirror bank; Right: nmos dummy device

Another current mirror variant that can be detected is shown in figure 4. This primitive is found when the transistor M0 is present, which source is connected to the drain of M1 and which drain is shorted with the gate of M1. The primitive consists of these two devices and the others which share M1's gate signal.

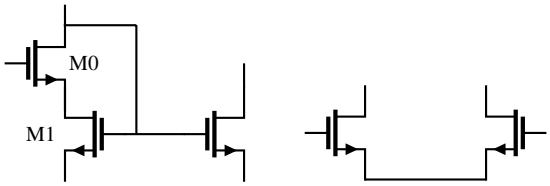


Fig. 4. Left: nmos current mirror variant; right: differential pair

The differential pair is the last dedicated circuit which can be detected and is shown in figure 4. If the previous conditions are not fulfilled this primitive is detected next. If devices of the same type share the same source connection, a differential pair is identified.

The remaining unassigned devices are estimated to be a cascode or switch. If the net at the gate is a digital net, the single transistor will be classified as a switch, if not a cascode. It is up to the engineer to verify these choices.

A final, but important addition to the detection algorithm, allows the layout generator to use cells with predefined layouts, e.g., an inverter. Such a predefined layout can be drawn by a designer or even IDcircuits. The designer has two choices to submit such a layout. One can adopt the abstraction convention and have rails above and below the cell, if the nets are known to the layout generator, this primitive behaves like any other. The other option requires introducing a primitive which has pins at predefined locations. This would then be a cell without rails, but with fixed pins which require routing.

B. Actions

The placement is done in a seemingly infinite, quantised space. Primitives are defined by bounding boxes and a list of `Rail` objects. These bounding boxes enable quick and simple detection of overlapping cells and allow for padding to take into account well spacing. Initial placement is performed by selecting a random position for each cell and verifying it does not collide with other cells. Then various actions can be performed to improve the initial layout. It should be noted that after each action the actual gridsize is changed to the enclosing bounding box, to limit memory occupation and increase routing performance.

The most obvious action is moving a cell. This involves choosing a cell and changing its position. Three variants ex-

ist. The first one limits the available space to the surrounding bounding box of all cells. This variant optimizes the layout in later stages, because only precise movement is allowed. The second variant allows each cell to be placed everywhere, by increasing the grid size by the dimensions of the cell that shall be moved. The final variant only influences cells with overlapping rails. It allows the cell to move horizontally along its rails, to improve the amount of overlap.

Another action is responsible for changing the order of the rails of a primitive. This action can swap two individual rails or change the position of a single one. The outer rails remain in its place if they overlap with other rails. This action is also able to flip a cell, by change the position of each rail to the other side of the primitive. A designer can specify if a cell has rails which do not allow this behavior. For example the tail current of a differential pair cannot be placed on the same side of a primitive as the other rails.

Besides rails, cells can be swapped as well. This action selects two cells at random. Based of the position of the cells an anchor point is determined, from which the new position can be calculated. This anchor is the corner of the cell closest to the center of the grid. The two cells simply move to their new position. If, due to the swap, two cells overlap, the cell at this location is moved away. It follows the direction that requires the least movement to avoid any overlap

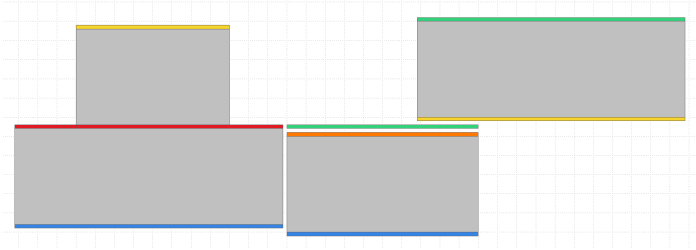


Fig. 5. Temporary placement; the two left cells are merged along the red net

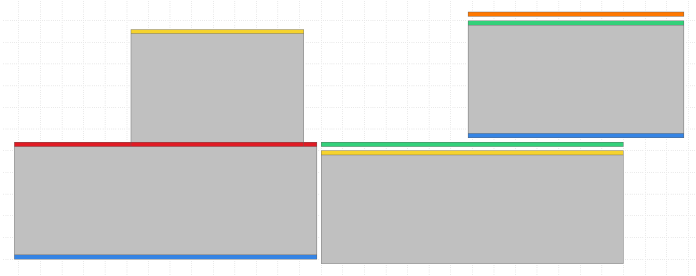


Fig. 6. The new layout is improved. The two cells on the right are swapped. Each one of them also underwent the swap rails action, both cases are displayed. The merge variant of the move action was performed on one of the cells with the merged rails.

The most influential action has been mentioned a few times, due to its impact on the other actions. It is the action that merges the rails of cells carrying the same net. This action has the most potential to lower the cost at the later stages of the algorithm, as the parasitic resistance is reduced a lot. Merging cells means that the two rails overlap, sharing the signal. The amount of cells that can merge is unlimited, as long as no shorts happen. Also a single rail can be shared by multiple cells and two cells can have more then one overlapping rail.

While this action is very useful, it comes with complex implementation. Balancing this action is difficult, as the actions should be very random and simple. However, this action, besides reducing the parasitic resistance, moves cells and swaps rails, which takes away the responsibility of the other actions.

The merging is done by analyzing the cells above and below the selected cell. If they have common nets, the potential decrease in cost is calculated. These cells are referred to as merge candidates. Three cases can be abstracted:

Case 1 The selected cell is not merged with other rails. The rails of the first candidate can be merged onto the selected cell without problems and case 3 is the next step. If not, case 2 is.

Case 2 When the selected cell is already merged with another cell, the cost reduction of the current merged rails is calculated. If this is lower than the new reduction in cost, the selected cell is unmerged and remerged with the first candidate. Either way, case 3 is considered.

Case 3 At this point the current cell is already merged on at least one side. If the other side is already merged, the algorithm ends. If not, the candidates are reanalyzed for the yet unmerged rails of the selected cell. The best candidate is then merged.

The two remaining actions influence the routing, one action changes the order in which the nets are being routed. The other one changes how many metal layers are available for the router to use. All these actions introduce a lot of creativity and diversity to the placement and routing of the cells. An example of some actions can be seen in figure 5 and 6.

C. Routing

Each iteration, the layout is routed after executing the actions. Before that happens, the cells are repositioned to ensure their position resembles indices in the routing grid. These positions are filled with certain values representing primitives or nets:

0 Unoccupied routeable space

1 Do not route, this represents space occupied by the primitives

2 This represents space occupied by the primitives, which allow interconnections above the devices

3-9 Placeholder values

10-999 These values represent traces carrying net, 10 represents the first net

1000- ... These numbers represent a via, 1000 represents the same net as 10 does for the traces

The original heuristic, the Manhattan distance, was interchanged for the added parasitic resistance. The sheet resistance for the current metal layer is multiplied by the distance covered in that layer. The resistance of the vias has to be added for the difference in metal layers and for the turns. This heuristic determines how the grid is explored and is crucial in efficient exploring and performance.

When two rails are being connected, custom logic determines the optimal point on each rail, which will be connected by the improved A* algorithm. This is based on the relative positions of both cells. Most of the time, the outer points on both rails which are closest together are selected. However, some cases require the selection of a specific point on the rail.

The next limitation which requires custom logic is when a net connects more than two primitives. A general search method is required to solve this problem. The layout generator employs a greedy algorithm, that approaches optimal routing, while keeping the amount of executions of the routing algorithm at a minimum. This method connects all the rails sequentially, based on which rail has the current shortest distance to the present traces of this net. This current shortest distance is estimated by the heuristic. This will not always provide an optimal result, but it is the task of SA to make sure the correct cells are close together in which case this algorithm has optimal performance.

Figure 7 demonstrates how the greedy algorithm operates. The red connection is made first, as it connects the two rails of the blue net via the path, which has the lowest cost. Then the remaining rails are routed onto the present traces, based on the respective routing cost.

To deal with the additional constraints, an extensive validation is required before exploring a node. This involves checking the following conditions in order on the neighbours of the selected node.

- Is the neighbour colliding with a primitive
- Is the neighbour not free space or metal with the same net
- Is the neighbour outside of the grid boundaries

The order is established to ensure the most frequent failing condition is verified first, to reduce execution time. The constraints are validated in the method `validMove`, employed by the final improved A* algorithm 2.

The A* algorithm is the most executed part of the layout generator. Each iteration of SA requires many calls to the router. Therefore, a lot of time went into optimizing the algorithm. However, as a larger schematic contains more rails and more nets, the amount of time the router is executed increases. Besides that, the routing algorithm also scales bad for larger grids, as it takes a longer time to route cells which are further apart.

The most effective way to enhance the performance of the layout generator is consistently performing actions that lower the cost. Hence, the amount of iterations is minimised. If the actions do not lower the cost, the calls to the router do not amount to anything. This can be improved upon by the designer, by choosing an optimal configuration for the framework.

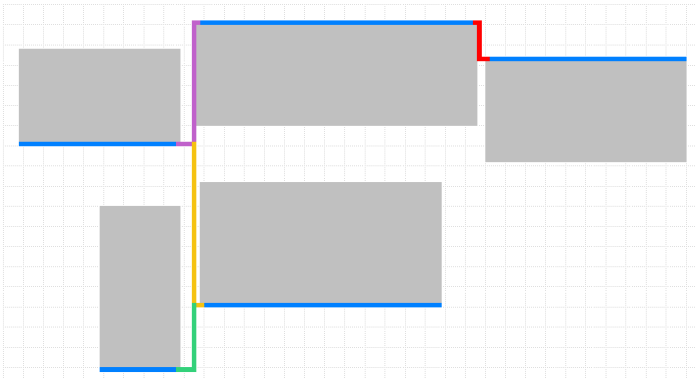


Fig. 7. Greedy routing on the blue net. Order of adding traces: red, purple, yellow, green

Algorithm 2 Improved A* algorithm

function AstarImproved ($grid, start, goal, netValue$)
  $moves \leftarrow [[(1,0,0),(1,0,0),(0,0,1),(0,0,-1)],$
  $[(0,1,0),(0,-1,0),(0,0,1),(0,0,-1)]]$

  if $grid.layers = 1$ **then**
    $moves \leftarrow [[(0,1,0),(0,-1,0),(1,0,0),(-1,0,0)],$
    $[(0,1,0),(0,-1,0),(1,0,0),(-1,0,0)]]$
  end if
  $closedSet = set()$
  $cameFrom = Dict[(int,int,int)] = (int,int,int)$
  $gScore = Dictionary[start] : 0$ ▷ Default value is infinity
  $fScore = Dictionary[start] : heuristic(start, end)$
  $openList = [(fScore[start], start)]$
  while $openList$ not empty **do**
    $curr = openList.pop()[1]$
    if cur in $closedSet$ **then**
      continue
    end if
    if $cur = end$ **then**
      return `reconstructPath(`cur`)`
    end if
    for dx, dy, dz in $moves[cur.z \mod 2]$ **do**
      $neighbour = (cur.x + dx, cur.y + dy, cur.z + dz)$
      if $neighbour$ in $closedSet$ **then**
        continue
      end if
      if $\neg$`validMove(`$neighbour$`)` **then**
        continue
      end if
      $tentativeG = gScore[cur] + dist(cur[0], dz)$
      if $tentativeG > gScore[neighbour]$ **then**
        continue
      end if
      $cameFrom[neighbour] = cur$
      $gScore[neighbour] = tentativeG$
      $fScore[neighbour] = tentativeG + heuristic(neighbour, end)$
      $openList.push((fScore[neighbour], neighbour))$
    end for
  end while
  return []
end function

IV. Results and Usage

With the many degrees of freedom and the huge variety in available circuits, the results of the layout generator can differ a lot. A simplified operational amplifier circuit was studied to gauge the influence of the parameters. Figure 8 shows an optimal configuration obtained for the default annealing parameters. To balance the influence of the area and parasitic resistance a fast calibration method is provided. By finding the cost for the optimal area and the optimal parasitic resistance, both contributions to the cost can be normalised. These values are found by first executing the algorithm without the router. Then this result is

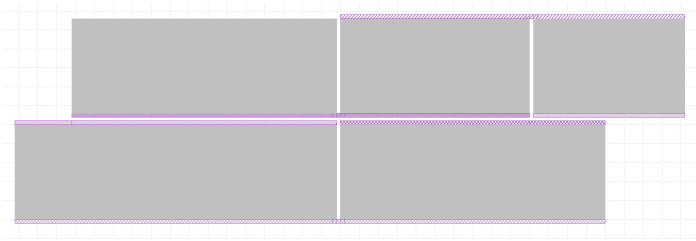


Fig. 8. Optimal result of the layout generator for simplified opamp circuit

used as the initial placement for a second run of the algorithm, which only takes into account the parasitic resistance.

The recommended method to find fitting values for the weights of the actions is by doing manual adjustments based on previous runs. Each run provides the engineer with data for these actions which can be used as feedback. In general for each iteration of the simulated annealing, the current cost and temperature is stored, together with the performance of each action.

However, the annealing parameters are more influential on the behavior of the layout generator. By executing the algorithm multiple times for different configurations, it is clear that for $\alpha = 0.98$, the algorithm performs the best. The distributions of the cost for different values for α are shown in figure 9. The cost is the lowest on average and the difference between the worst and best results is also the smallest. The downside is that the results require a very high number of iterations to reach T_{end}.

The temperature interval $[T_{start}, T_{end}]$ can also be tweaked to obtain different results. A smaller T_{end} will obtain very precise results, but less uphill movement is allowed in later stages. A higher T_{start} will mean the early stage is very random as all the uphill movement will be allowed for these high temperatures. Changing the ratio of the start and end temperature will change the amount of improvements required, as well as the behavior of dealing with the local minima. The designer is tasked with configuring the layout generator, as its specifics are contingent upon the circuit being designed.

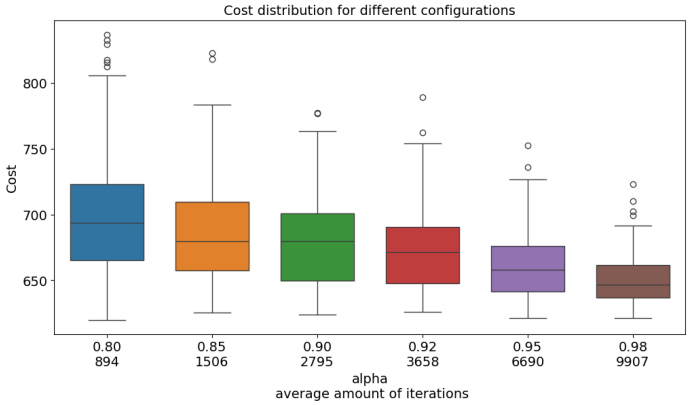


Fig. 9. Distribution of the cost for different α

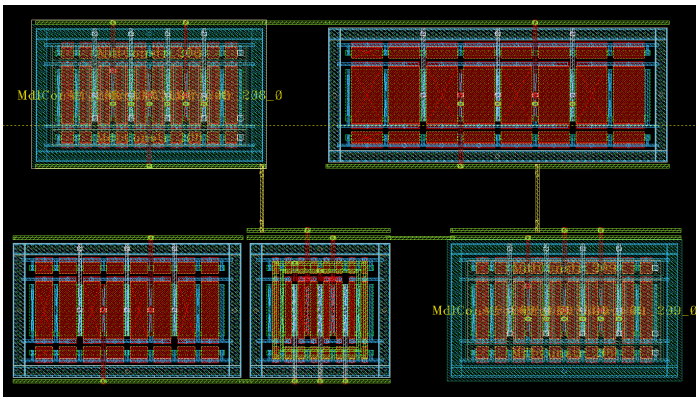


Fig. 10. Final layout of the operational amplifier

A final layout is shown in figure 10 and displays an optimal DRC-clean layout of an operational amplifier in Virtuoso. The spacing for primitives with different bulk biasing and the constraints on the rails of the differential pair are taken into account.

V. Conclusion

Overall, this layout generator achieves precise routing and placement via simulated annealing and an improved A* algorithm. An operational amplifier was extensively studied to evaluate the performance of this tool, which extends the capabilities of IDcircuits, a python framework for automatic layout generation. By simplifying the available primitives, an optimization procedure was developed to simultaneously place and route these primitives. As the layout generator is meant to be used together with a designer, the framework allows a script and a configuration file to control the resulting layouts. The influence of these parameters is clarified and a calibration method is provided to accurately gauge the layouts based on the enclosing area and parasitic resistance.

References

[1] Noraziah Adzhar and Shaharuddin Salleh. “Simulated Annealing Technique for Routing in a Rectangular Mesh Network”. In: *Modelling and Simulation in Engineering* 2014 (Dec. 2014).

[2] Eric Chang et al. “BAG2: A process-portable framework for generator-based AMS circuit design”. In: *2018 IEEE Custom Integrated Circuits Conference (CICC)*. 2018, pp. 1–8.

[3] Hao Chen et al. “MAGICAL: An Open-Source Fully Automated Analog IC Layout System from Netlist to GDSII”. In: *IEEE Design & Test* 38.2 (2021), pp. 19–26.

[4] Tonmoy Dhar et al. “ALIGN: A System for Automating Analog Layout”. In: *IEEE Design & Test* 38.2 (2021), pp. 8–18.

[5] Jaeduk Han et al. “LAYGO: A Template-and-Grid-Based Layout Generation Engine for Advanced CMOS Technologies”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 68.3 (2021), pp. 1012–1022.

[6] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107.

[7] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. “Optimization by Simulated Annealing”. In: *Science* 220.4598 (1983), pp. 671–680.

[8] Ricardo Martins, Nuno Lourenço, and Nuno Horta. “LAYGEN II—Automatic Layout Generation of Analog Integrated Circuits”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32.11 (2013), pp. 1641–1654.

[9] Taeho Shin et al. “LAYGO2: A Custom Layout Generation Engine Based on Dynamic Templates and Grids for Advanced CMOS Technologies”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42.12 (2023), pp. 4402–4412.

Contents

List of Figures

List of Tables

LIST OF ALGORITHMS

1 INTRODUCTION

Electronic design automation (EDA) has always been pursued for analog integrated circuit(IC) design. As scaled technologies become more and more complex, the constraints on the layout increases too. The Design Rules Check (DRC) for 28nm technologies consist of 10000 rules, therefore automating layout design can aid an IC designer by drastically cutting down the development time.

Aside from the digital counterpart, the design automation of analog complementary metal-oxide-semiconductor (CMOS) devices is less adopted, because fully automated designs limit the control of the IC engineer, which is required for these cutting edge designs. The performance requirements of these layouts is too high and the EDA tools do not reach these high specifications. Therefore this thesis aims to provide a layout generator which is scripted by an IC designer. This way the resulting layout can be tailored to the requirements of the schematic, because a lot of outside control is introduced.

The first step in generating an IC layout requires detecting the devices inside a schematic and selecting the combination of devices which sum up so called primitives, like a differential pair or current mirror. What follows is the creation of the floor plan of these primitives, based of their specific design parameters. The base block design should incorporate optimization techniques like interdigiting structures, merging devices and improving matching by including symmetry. The final and most influential step involves carefully routing and placing the primitive devices to ensure the layout induces minimal parasitic effects and meets the design specifications.

In literature many different layout generators have been proposed, each one employing different concepts and algorithms. One of the types of layout generators are template-and-grid-based [6],[12], [11]. They use primitives specific to the technology which have predefined placement grids. These placement grid consists of regularly spaced points or cells, where each cell corresponds to a specific area on the chip. By aligning template instances with the grid, the generator ensures that layout objects are placed consistently. The routing is performed on-grid, symmetrically across the templates containing the devices. The placement is done by using the knowledge of previously designed layouts to define relative placement constraints. The routing is done as the last step utilizing an appropriate optimization engine.
LAYGEN II specifically extracts primitives from a thorough list of information and requirements of the nets. And extracts a so called nonslicing B*-tree to perform the placement. By utilizing a packing algorithm a position for the templates is derived. The routing step is performed using an optimization based algorithm called an evolutionary algorithm.

BAG2 [3] introduces a generative approach. Rather than designing a single circuit instance, the designer encapsulates their methodology in the form of an executable circuit generator. This high level generator is able to produce schematics and layouts from given input specifications. Utilizing these generators, designers can implement fully automated design iteration loops.

ALIGN [5], and MAGICAL [4] are optimization based layout generators. Both are developed in an open-source environment. ALIGN has a very large pool of primitive cells which each have multiple layouts. ALIGN fully automates the process by accepting an unannotated net list as input. By incorporating machine learning models it annotates the netlist, and generates the specific constraints relative to the circuit and technology. Based of the

generated constraints and netlist, the corresponding primitives are chosen, placed and routed. Implementing multiple optimization techniques to perform the placement and routing for advanced technologies employing FINFET devices, like a sequence-pair method and A-star routing.

MAGICAL is another fully automated IC layout framework. By extracting layout constraints, concerning for example symmetry, of a given netlist the parameters for placement and routing are found. The analog placement uses an analytical framework to satisfy the generated constraints and an integer linear programming (ILP) algorithm for the global routing. The detailed routing is performed with an adaptation of the A* search algorithm. This way they generate high performing, symmetrical analog layouts.

This thesis presents a layout generator which produces layouts for single-ended circuits. The layout generator combines the routing and placement, by utilizing a generative and optimization based approach. Simulated Annealing (SA) performs the placement of primitives based of specific technology design rules, while an improved A-star(A*) algorithm takes care of the routing simultaneously. The preliminaries such as environment, tools and testing circuit required for this layout generator are described in chapter 2. Then the main concepts of the proposed design are discussed in chapter 3. Chapter 4 dives into the actual implementation, describing the architecture and algorithms which make up the placer and router. Chapter 5 discusses the performance and usage of the layout generator. A conclusion and future works conclude this thesis in chapter 6.

2 ENVIRONMENT

This chapter delves into the required tools and concepts to convert a schematic into an optimal layout.

To evaluate the performance of the developed algorithms, two sets of schematics were utilised. The first set consists of randomly generated schematics, which may not accurately reflect realistic functional circuits. These schematics were employed to test the various tools for a wide range of inputs. However, caution is necessary to prevent the unrealistic nature of some inputs from leading to unnecessary assumptions and optimizations.

The second set comprises of an operational amplifier (opamp). This circuit was employed to gauge the performance of the layout generator. The influence of the design parameters were explored by analyzing many layouts for different configurations.

While the developed tools and concepts are technology-agnostic, the IHP SG13 technology was used during the development of the layout generator. Hence, when applying the framework to other technologies, it is expected that slightly different configurations of the algorithms may be required to account for the differences in design rules.

To programmatically interface with Virtuoso, the SKILL language was developed by Cadence. However, most engineers are not familiar with using this scripting language. Luckily Skillbridge exists, which serves as a bridge between Python [13] and Skill, allowing full control over Virtuoso in Python.

The layout generator is not a stand alone project, as it builds further upon IDcircuits, an inhouse developed tool to generate layouts of certain primitives. This thesis aims to extend IDcircuits to automatically detect, place and route these primitives. These primitives are for example a differential pair or a current mirror. These can be generated given a set of primitive-specific design parameters and the resulting layouts are DRC clean. This thesis does not aim to do routing and placement on device level, thus primitives are seen as a black box.

Figure 2.1 shows the layout of a differential pair, as generated by IDCircuits. To achieve optimal matching, the transistors are arranged symmetrically and enclosed by dummy devices. Additionally, a guard ring is added for proper isolation. Rails above and below the layout primitive serve as the inputs and outputs of the primitive. In the following discussion an abstract representation of the layout primitive is used, as shown in Figure 2.2. The active area and the primitive level routing are simplified as a grey rectangle. The other rectangles represent the rails to interface with the transistors. Before chapter 4, the colours of the rails represent a certain net. When the results and usage is discussed the colours represent a metal layer.

The employed operation amplifier is a single-ended, two-stage, operational amplifier. The corresponding schematic can be seen in figure 2.3. The transistor dimensions can be seen on the figure, where the labels represent the width/length. These dimensions were obtained, using pre-computed lookup tables [9]. This design provides a good starting point for the layout generator as it contains 5 simple building blocks: a NMOS differential pair, 2 NMOS current mirror, 2 PMOS current mirrors.

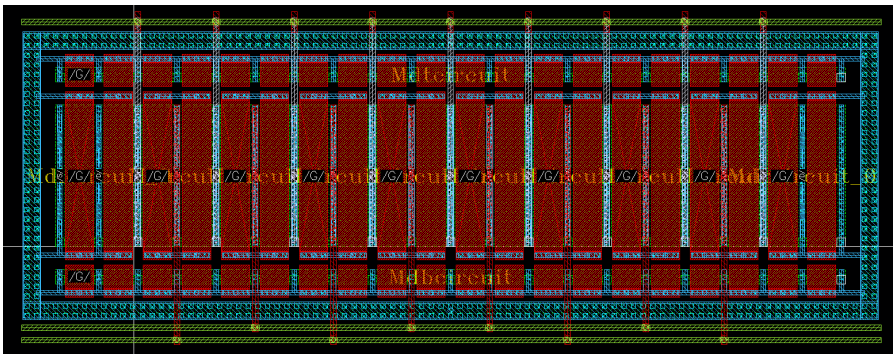


Figure 2.1: Layout of a differential pair primitive

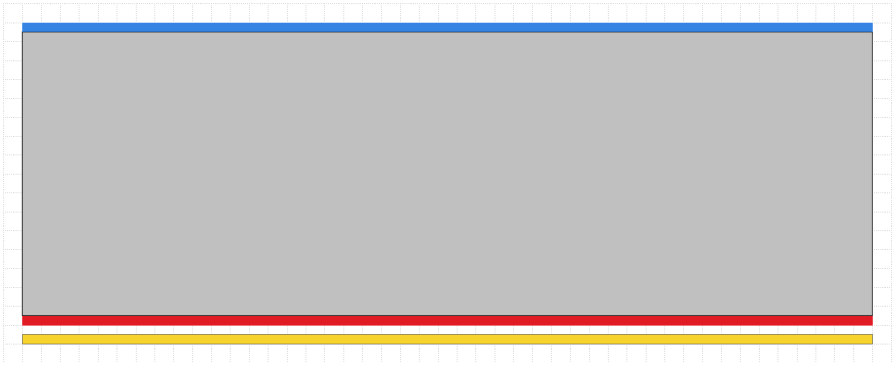


Figure 2.2: Representation of the abstracted differential pair primitive

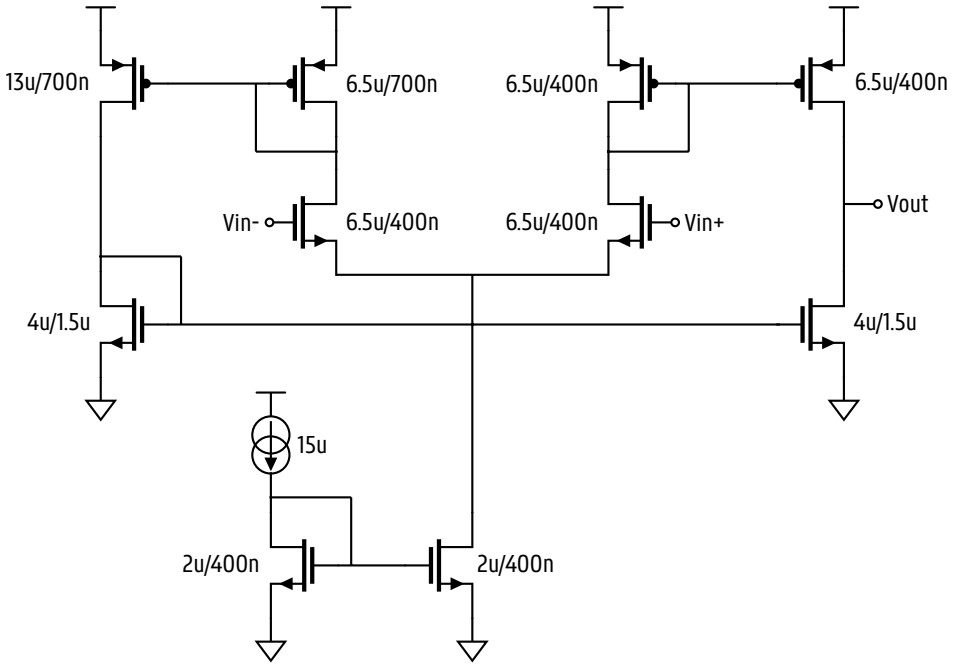


Figure 2.3: Schematic and sizing of the operational amplifier

3 PROPOSED CONCEPTS

This chapter explores the proposed concepts of the primitive detection, and the simultaneous optimization of placement and routing. As a starting point, a schematic with proper sizing is assumed.

3.1 Primitive detection

The first step is to detect the primitives present in the schematic. The detection logic revolves around recognising the patterns associated with various primitives and determining which combinations of devices correspond with which specific primitive. A dictionary stores the nets connected to each terminal of each MOSFET, the specific transistor type and whether the gate contains a digital net. This way all connections between the devices can easily be retrieved. To demonstrate how this works the opamp is redrawn in figure 3.1, the corresponding net of each wire is annotated.

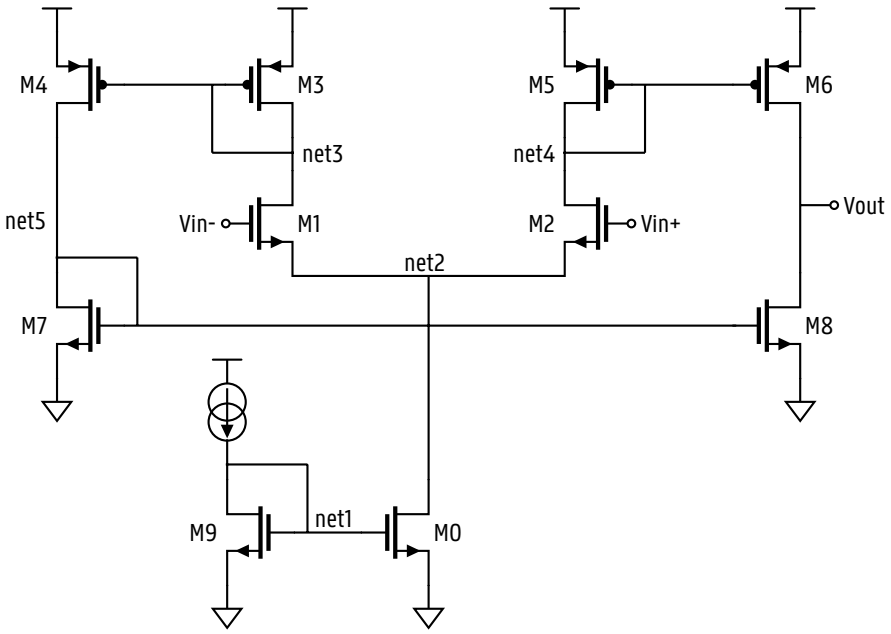


Figure 3.1: Opamp schematic with annotated nets

```
devices = {
M1:{type:nmos, S:net2, G:Vin-, D:net3},
M2:{type:nmos, S:net2, G:Vin+, D:net4},
M3:{type:pmos, S:Vdd, G:net3, D:net3},
M4:{type:pmos, S:Vdd, G:net3, D:net5},
M5:{type:pmos, S:Vdd, G:net4, D:net4},
M6:{type:pmos, S:Vdd, G:net4, D:Vout},
M7:{type:nmos, S:gnd, G:net5, D:net5},
M8:{type:nmos, S:gnd, G:net5, D:Vout},
M9:{type:nmos, S:gnd, G:net1, D:net1},
M10:{type:nmos, S:gnd, G:net1, D:net2}}
```

The `devices` dictionary stores for each device its type and the nets at each terminal. Two extra utility methods are employed: `findNetAtTerminal`, `findDevicesAtNet` of which the pseudocode is shown in 1.

Algorithm 1 findNetAtTerminal, findDevicesAtNet

```
function FINDNETATTERMINAL(net, terminal, devices)
    result ← []
    for all name, nets in devices do
        if nets[terminal] = net then
            result.append(name)
        end if
    end for
    return result
end function
function FINDDEVICESATNET(net, devices)
    result ← Dictionary[str, List]
    for all name, nets in devices do
        terms ← []
        for all term, curNet in nets do
            if curNet = net then
                terms.append(term)
            end if
        end for
        result[name] = terms
    end for
    return result
end function
```

The current implementation only detects differential pairs and 2 variants of current mirrors, as other primitives are not available yet in IDCircuits. Nonetheless, the algorithm could easily be extended to include more advanced primitives. Primitives which could not be detected, are either classified as a cascode or a switch, depending on the nature of the gate voltage.

The algorithm for the primitive detection is shown in 2.

Algorithm 2 Detection of primitives

```
function detectPrimitive(devices)
    unassigned ← devices.keys
    primitives ← []
    for all name, type, nets in devices do
        if ¬(name in unassigned) then
            continue
        else if nets[S] = nets[G] = nets[D] then
            primitives.append((type, Dummy, [name]))
            unassigned.remove([name])
        else if nets[G] = nets[D] then
            atGate ←findNetAtTerminal(nets[G], G, devices)
            primitives.append((type, CurrentMirror, [name, atGate]))
            unassigned.remove([name, atGate])
        else if findNetAtTerminal(nets[D], S) = findNetAtTerminal(nets[G], D) then
            atGate ←findNetAtTerminal(nets[G], G, devices)
            atDrain ←findNetAtTerminal(nets[D], S, devices)
            primitives.append((type, CurrentMirrorVariant, [name, atGate, atDrain]))
            unassigned.remove([name, atGate, atDrain])
        else
            atSource ←findNetAtTerminal(nets[S], S, devices)
            if length(atSource) = 1 then
                primitives.append((type, DifferentialPair, [name, atSource]))
                unassigned.remove([name, atSource])
            end if
        end if
    end for
    for all name in unassigned do
        if devices[name][G] has digital nature then
            primitives.append((type, Switch, name))
        end if
        primitives.append((type, Cascode, name))
    end for
    return primitives
end function
```

The algorithm iterates over the dictionary and performs sequential checks on the devices. If a condition is fulfilled, the devices and transistor type are assigned to the corresponding primitive. Detected devices are removed from the iteration.

The first condition verifies whether the current device is a dummy. Dummy devices are single transistors which have the ground net or supply net at each terminal.

The next primitive being detected is a standard current mirrors. This is based upon the presence of a diode-connected device, i.e., the gate and drain are shorted. All devices sharing the gate connection are included in the current mirror. An example of a current mirror bank is shown in figure 3.2.

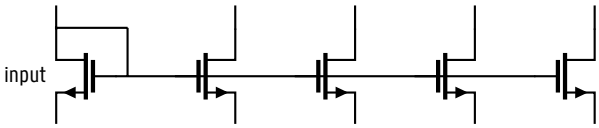


Figure 3.2: Current mirror bank

Another current mirror variant that can be detected is show in figure 3.3. This primitive is found when the transistor M0 is present, which source is connected to the drain of M1 and which drain is shorted with the gate of M1. Then the primitive consists of these two devices, together with the other transistors carrying the gate of M1 at their own gate.

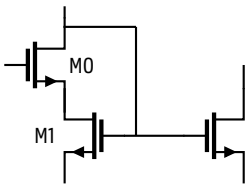


Figure 3.3: Cascoded current variant

The differential pair is the last dedicated circuit which can be detected and is shown in figure 3.4. If the previous conditions are not fulfilled this primitive is detect next. Devices of the same type, which share the same source connection are classified as a differential pair.

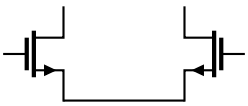


Figure 3.4: Differential Pair

The remaining unassigned devices are estimated to be a cascode or switch. If the net at the gate is of a digital nature, the signle transistor will be classified as a switch, if not a cascode. It is up to the engineer to verify these and all the other detected primitives.

A final, but important addition to the detection algorithm, allows the layout generator to use cells with predefined layouts, e.g., an inverter. Such a predefined layout can be drawn by a designer or even IDcircuits. The designer has two choices to submit such a layout. One can adopt the abstraction convention and have rails above and below the cell, if the nets are known to the layout generator, this primitive behaves like any other. The other option requires introducing a primitive which has pins at predefined locations. This would then be a cell without rails, but with fixed pins which require routing.

3.2 Placement and Routing

Having detected the different subblocks inside a circuit, placing and routing these primitives is the next step. The final layout should be DRC clean and as optimal as possible. Optimal could involve having shortest interconnects, the least surface area or minimised parasitics. However, it is essential to understand that the targeted performance metric depends heavily on the considered circuit. Hence, the IC designer must be allowed to tailor the algorithm to optimize for the desired performance metric. The main objective of the layout generator is to provide the engineer with quick and proficient starting point, which allows easy finetuning.

As the layout is obtained via an optimization algorithm a figure of merit is required to gauge the performance of a specific configuration. This figure of merit is a cost function, where a lower cost corresponds to an optimal design. It is calculated as a weighted sum of the total area of the layout and total added parasitic resistance of the rails, traces and vias. The second comes down to minimizing the routing or choosing the path with least resistance between two rails.

The final product will be discussed in depth. The algorithms went trough multiple iterations, each one improving on previous designs to reach the final goal of generating an optimal, DRC-clean IC layout. This involves routing and placing layouts of primitives on a grid with multiple layers, up to 4. The subblock routing happens on a base layer, below the routeable layers, with the rails of these subblocks exposed on the first available layer. The aforementioned grid is assumed infinite, for the cells can be placed everywhere. The pitch of this grid is defined as 1. This unit distance is the smallest size of anything in the layout, meaning that the minimal width of a trace is 1 and the smallest spacing between different primitives is 1 as well. This results in a quantised layout, which is necessary to limit memory usage and computation time. A smaller pitch would mean increasing the total grid size.

This quantization also influences the design rules. It is crucial that the placement and routing obliges to these technology-specific constraints. As the primitive layouts, drawn by IDcircuits, are DRC-clean, the placement and routing only needs to consider the relative placement of the primitives, metal traces, and vias. All these design rules require to be quantised, to correspond with the pitch. Hence, the unit distance 1 is translated to the minimal spacing between two metal traces, as this is the most stringent constraint.

The constraints for relative placement of vias and traces depend on the specific metal layer. This is disadvantageous for the routing, as this would mean that different layers behave on different pitches and that the unit distance 1 is not equal for these layers. To avoid this, the layout generator assumes that 4 metal layers with the same relative spacing and minimal width are available. If this is not the case, the amount of metal layers can be easily reduced.

The relative placement between subblocks also has to oblige to design rules. Primitives can be placed next to each other if their bulk is biased to the same potential and the devices inside the primitive have the same type (N or P type devices). If this is not the case, the cells have to be spaced apart correctly.

Besides relative placement, it is obviously required to avoid shorts and open or unrouted nets. It is the task of the router to make sure no metal traces carrying different signals touch each other and that each net is routed to avoid opens. If it does happen, the placer can detect these erroneous layouts.

To give an idea of what a general layout of the opamp looks like, one could analyze figure 3.5. Here, the spacing between the N and P cells is clear. This layout is definitely not optimal, but it displays a DRC clean layout. For the remainder of this thesis all cells are assumed to be n-type, unless specified. The examples shown will demonstrate the operation of the algorithms and do not resemble actual circuits. Most examples are visualised with simplified primitives of the opamp circuit.

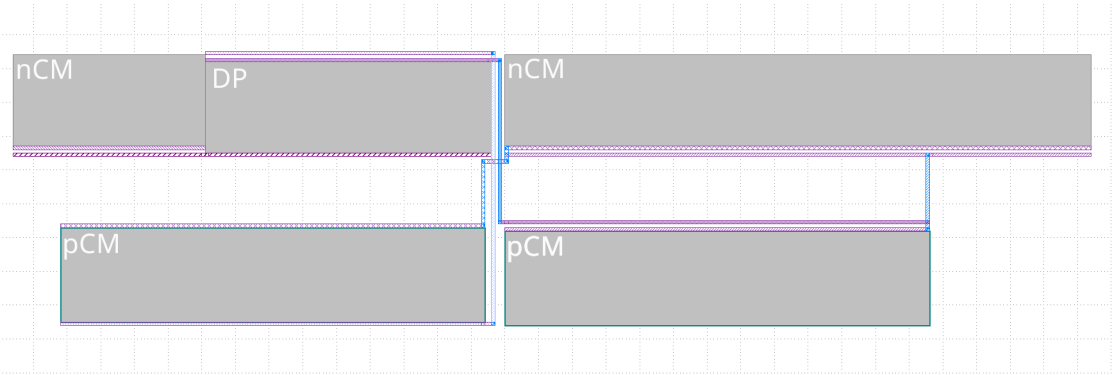


Figure 3.5: DRC clean layout for the opamp circuit.

3.2.1 Placement via Simulated Annealing

Before going in the details of the placer, it is important to explain the main method used to solve the routing and placing problem. The optimization process should explore and narrow down the solution space to obtain a good design. A viable method for solving complex problems is a genetic algorithm. This would involve changing the same layout once, N times, providing N different new layouts. Then from these N designs, the layout with the lowest cost is selected and the previous step happens anew, where something is changed about this new circuit N times. This way the layout would keep on improving and this would narrow down the solution space. After a certain amount of iterations, the designer would be happy with the outcome.
However this method is not that good at detecting local minima, hence reaching a suboptimal solution. Another problem with this method is that the many iterations take a lot of time, because the router has to be executed a lot. This approach could be possible if the routing and placement was seen as two sequential problems. This thesis however gauges the routing and placement of a layout at the same time, with correlated algorithms for routing and placement.

A method that does deal well with detecting and avoiding local minima is Simulated Annealing(SA) [10]. This heuristic algorithm is based of the controlled cooling down of metal, called annealing. Hot metal has a high entropy and a very chaotic, randomised crystal structure. By gradually cooling down, the atoms move into a more structured lattice, reducing entropy and randomness. However if a part of this lattice was not perfectly aligned with the rest, reheating the metal a bit before cooling it down again, allows the lattice to realign itself.

The placer employs simulated annealing to find an optimal solution, starting from a random chaotic position. Applied to layout generation this comes down to the following. At the start, the cells are placed at a random spot in the infinite grid. Then this configuration is routed and the corresponding cost is calculated. By changing the configuration, one tries to reduce this cost until it reaches a point low enough. Changing the configuration is done by executing an 'action' in the hope of lowering the cost. To avoid local minima, sometimes a worse cost can be accepted and uphill movement is allowed. Algorithm 3 shows pseudocode used by this layout generator. The initial values of the parameters are the default values, inherited by [1]. Some utilised functions are not yet specified, for they are explained further on.

Algorithm 3 Placement algorithm based of simulated annealing

Initialize parameters
$T_{start} \leftarrow 100$
$T_{end} \leftarrow 0.01$
$T_0 \leftarrow T_{start}$
$\alpha \leftarrow 0.95$
$i \leftarrow 1$
$maxIter \leftarrow 5000$

Initialize layout
$P_0 \leftarrow$ Initial placement
$U_0 \leftarrow route(P_0)$ ▷ $route()$ returns amount of unrouted nets U_0
$C_0 \leftarrow cost(P_0)$

while $T > T_{end}$ and $i < maxIter$ do
    $P_i = P_{i-1}$
    $performActions(P_i)$
    $resizeGrid(P_i)$
    $U_i \leftarrow route(P_i)$
    $C_i \leftarrow cost(P_i)$

    Evaluate new placement P_i based of amount of unrouted nets U_i and cost C_i
    if $U_i < U_{i-1}$ then
        $T_i = \alpha T_{i-1}$ ▷ Accept new placement
    else if $U_i = U_{i-1}$ then
        if $C_i < C_{i-1}$ then
            $T_i = \alpha T_{i-1}$ ▷ Accept new placement
        else
            $\Delta C = C_i - C_{i-1}$
            if $\varepsilon < e^{-\Delta C/T_i}$ then ▷ $\varepsilon \sim U(0,1)$
                $T_i = \alpha T_{i-1}$ ▷ Accept new placement
            else
                $P_i = P_{i-1}$ ▷ Reset placement
            end if
        end if
    else
        $P_i = P_{i-1}$ ▷ Reset placement
    end if
    $i = i + 1$
end while

The mentioned parameters, which track the progress of the algorithm, are related to temperature. T is the global temperature of the system. The temperature at the ith iteration is denoted as T_i. This temperature lowers every time the cost lowers or a configuration with a worse cost is accepted. T_{end} is the ending temperature, when T, reaches this value, the layout generation is finished. α indicates how quickly the global temperature changes. Each time the temperature lowers, the new temperature is calculated as follows: $T_{i+1} = (1-\alpha)T_i$. To avoid local minima, uphill movement is introduces, by accepting a layout with a higher cost. A Boltzmann distribution 3.1 determines by chance whether uphill movement is tolerated.

$$P(\Delta Cost) = exp\left(\frac{-\Delta Cost}{T_i}\right) \tag{3.1}$$

When the cost is worse than the cost of the current layout, this distribution determines if the uphill movement is allowed. It does this according to the random chance that the distribution is larger than a random uniform distribution $P(\Delta Cost) > \varepsilon \sim U(0,1)$.
The odds of uphill movement depend on the change in cost and the current temperature, this means the lower the global temperature gets, the less a worse configuration is accepted. Or in other words only configurations which are slightly higher in cost can be accepted for lower temperatures to gradually find the optimal layout.

The driving power behind simulated annealing are the actions, which modify the layout of the current generation to obtain the layout of the next generation. Each SA iteration, a specific action is chosen at random. The most obvious action is moving a cell to a new position. An action should be simple and flexible, for SA relies on randomness to find an optimal solution. An action too specific will not perform good enough, because it would only lower the cost for the specific purpose. For example if one would design an action that moves a cell 10 units to the left. This action is too specific and will only work when a cell should be moved 10 places to lower the cost. This example is an exaggeration, but it points out that the actions should empower the chaotic, random character of SA.

The reason actions should be simple is that it should be easy to adapt actions and add other actions. Actions should interact well with each other and too much edgecases may lead to faulty configurations. This can be extended to everything that encompasses the layout generator and not always was it possible to achieve a simple design for the actions.

The design and concept of an action is crucial, yet its impact is reliant on the current placement and implementation. As the global temperature gets lower, SA needs more time to find a fitting action. The toleration on the cost tightens and everything needs to be more precise. In scenarios where only a couple of viable actions exist, it is luck dependent whether the cost lowers. In most of these scenarios uphill movement is required to broaden the solution space. In these later stages, some actions are more useful. These tend to be the actions with less impact on the cost. The opposite is also true, moving cells is for example an action that can be very powerful at the start of SA. That is why each action has its own weight, which influences how often it is chosen.

Another example why the implementation of the actions is crucial is the following situation. The layout generator can reach a point where a single action cannot further lower the cost, whilst the layout itself is not optimal yet. An example of this can be seen in figure 3.6, for this example the available actions are moving cells and changing the position of rails. The routing is done on a single layer as this is the most optimal.
The only improvement to the layout is having a shorter interconnection for the yellow net, which requires changing the position of cell 1 and cell 2, the optimal layout is shown in figure 3.7. However, moving any of the cells to another location in order to make space for the other, increases the cost significantly. With the given actions the optimal solution cannot be reached. The solution is to allow executing 3 actions at the same time to change the locations of the two cells. Another solution would be to swap the cells, which is explained further on.

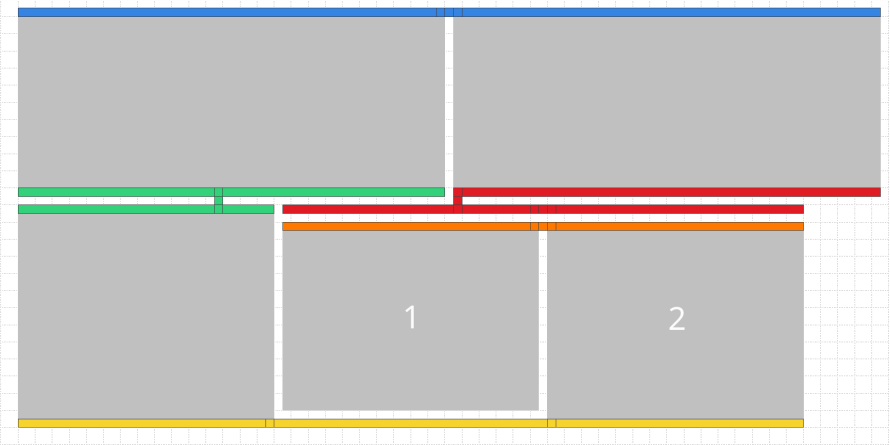


Figure 3.6: Placement where swapping and moving cells does not amount to a lower cost

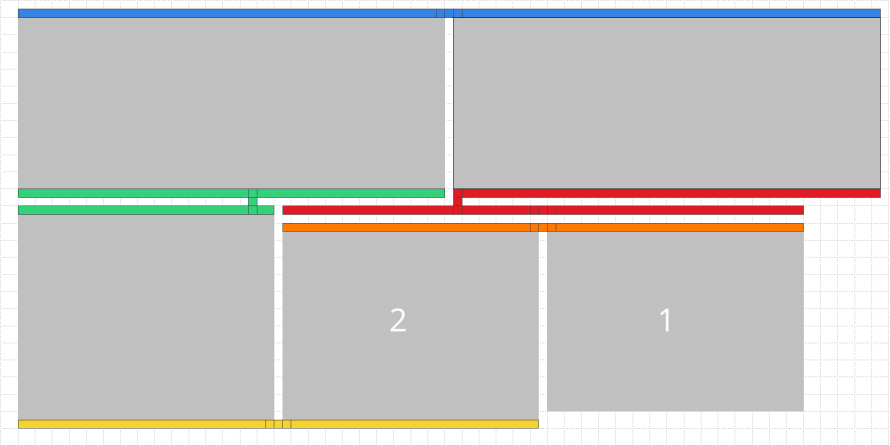


Figure 3.7: Actual optimal configuration

Therefore each iteration a random amount of actions were performed, to introduce more creativity in the placement algorithm. The maximum amount of actions per iteration is controlled by the IC designer. It is important not to choose this value too high, because this increases the chance that a bad action is executed. If for example five actions are executed and four of them lower the cost, but one of them is very bad and the resulting cost is increased or even worse a rail is not routed, these other four good actions are not propagating to the next iteration. To avoid this, choose this parameter as low as possible, to still avoid problems like seen in figure 3.6, where a minimum amount of actions are required. The pseudocode for the method $performActions$ is shown in algorithm 4.

Algorithm 4 perform Actions

```
Initialize weights and actions
allActions ← [moveCell, swapRail, ...]
weights ← [1, 1, ...]
maxActions ← 2
function performActions(P)
    n ← random(0, maxActions)
    actions ← choice(n, allActions, weights)   ▷ choose n values from array with given weights
    succes ← False
    while ¬succes do
        for all action in actions do
            if action(P) then
                succes ← True
            end if
        end for
        if succes then
            break
        end if
        n ← random(0, maxActions)
        actions ← choice(n, allActions, weights)   ▷ If all actions failed, choose actions
    end while
end function
```

3.2.2 Routing with A*

The actions of the simulated annealing take care of the placement of the cells. After the action(s) are executed, the nets are routed before the cost is calculated. This section focusses on the general algorithm for the routing of the subblocks and the individual rails. An optimal algorithm to connect two points in a grid where many obstacles are present is the astar(A*) algorithm. This algorithm is an extension of Dijkstra's, but it performs better, due to the search being directive and aware of the goal at any time by employing a heuristic. Algorithm 5 explains the working of A* as proposed by [8].

The algorithm explores a 2D grid or graph, while keeping track of two metrics for each visited node: the current cost from the starting point to the node and the estimated cost to the endpoint. The current cost is accurate as the path from start to the point is aware of all the obstacles in between. Line 17 of the pseudocode calculates this value, the so called gScore. On the other hand, the estimation of the cost between the current node and the goal follows a heuristic function. For plain A* and 2D routing the heuristic is just the Manhattan distance between each node and the goal, if the movement is restricted to the cardinal directions. This means that each movement has a distance of 1, which means that the gScore is always updated by 1. The fScore is then the sum of the estimated cost to the goal and the current gScore. By exploring the neighbours of the node with the current lowest fScore and recalculating the two metrics for these nodes, the shortest path is found. The only time the distance between a node and its neighbour is not 1, is when the neighbour is an obstacle. Then, an infinite gScore is obtained.

A-star is very well known and well documented, however out of the box it is not viable for routing integrated circuits. Multiple adjustments are required to allow A* to route an IC layout. The first limitation of A* is that it connect 2 single nodes. Realistic IC routing involves routing different nets, which often interconnect more than two primitives. Besides the routing of more than two rails, plain A-star is also not able to route whole rails. Therefore, a point on the to be routed rails will have to be selected. Besides dealing with the rails of the cells, the routing algorithm also has to deal with different constraints on the present obstacles.

An IC routing algorithm also has to deal with different constraints on the present obstacles and traces. Traditional A* simply routes around present obstacles, which means the validation of neighbouring nodes will have to be extended to ensure DRC-clean routing.

Aside from all the aforementioned limitations, the most limiting factor is the 2D grid. The fact that IC routing uses multiple layers, has implications on both the heuristic and exploration. However, an IC layout should strive for straight interconnections, avoiding zigzag routing as much as possible. A convention that is employed in (digital) IC routing is using even and odd layers for respectively vertical and horizontal movement or vice versa. This convention allows to find the optimal path faster.

Furthermore, 3D routing requires a heuristic other than the Manhattan distance, due to the introduction of vias. The 'distance' between two metal layers is not the same as the distance between two nodes, due to the vias adding more resistance. Besides that,higher metal layers add less resistance for the unit distance, which means the distance between two cells changes too.

Algorithm 5 original A* Algorithm

```
1:  function AStar(start, goal)
2:      openList ← {start}
3:      closedList ← {}
4:      gScore[start] ← 0
5:      fScore[start] ← heuristic(start, goal)
6:      while openList is not empty do
7:          current ← node in openList with lowest fScore
8:          if current = goal then
9:              return ReconstructPath(current)
10:         end if
11:         Remove current from openList
12:         Add current to closedList
13:         for each neighbour of current do
14:             if neighbour is in closedList then
15:                 continue
16:             end if
17:             tentative_gScore ← gScore[current] + distance(current, neighbour)
18:             if neighbour is not in openList or tentative_gScore < gScore[neighbour] then
19:                 cameFrom[neighbour] ← current
20:                 gScore[neighbour] ← tentative_gScore
21:                 fScore[neighbour] ← gScore[neighbour] + heuristic(neighbour, goal)
22:                 if neighbour is not in openList then
23:                     Add neighbour to openList
24:                 end if
25:             end if
26:         end for
27:     end while
28:     return failure
29: end function
30: function ReconstructPath(current)
31:     totalPath ← [current]
32:     while current in cameFrom.Keys do
33:         current ← cameFrom[current]
34:         Add current to totalPath
35:     end while
36:     return totalPath
37: end function
```

4 IMPLEMENTATION

As the main concepts for routing and placing primitives are explained, the full details of the layout generator can be discussed. The architecture of the code, all actions available to simulated annealing and the improved A-star routing algorithm will be explained. Finally, the abstraction is lifted and a real layout in Virtuoso is the result.

4.1 Architecture

The code architecture of the layout generator evolved a lot. The final architecture introduces multiple classes or instances to allow precise control of the primitives and to easily store and access the current state of the layout. By applying the single-responsibility principle, the functionality of the instances was kept modular and simple. Following this principle made adding new features easy and simplified the interaction between different instances.

4.1.1 Cell

The abstracted primitives are stored in the `Cell` class, whose attributes are shown in table 4.1. The dimensions of a `Cell` object are defined by the coordinate of the top left corner and the width and the length of the cell. The width of the cell is the total width and `w_si` denotes the width of the black box area of the primitive, represented in grey in figure 4.1. The rails of the primitive are stored in two arrays: `top`, `bot` as `Rail` objects. The `strictRails` variable decides how flexible the rails can be placed. Certain primitives, like the differential pair, do not allow mixing the initial rail configuration. The net containing the tail current should always be on the other side of the cell, compared to the two other nets containing the outputs of the differential pair. These types of cells have strict rail constraints and will have to be handled differently for certain actions.

Besides that the remaining characteristics of a primitive cell is a unique id, used for indexing cells and the bulk potential. For simplicity the latter has type `net`. The nets are stored as the string representing them, but the `net` type is used as a wrapper to make this more clear.

When a Cell is initialised, the required parameters are: `id`, `bulk`, `w`, `l`, `topNets`, `botNets`. The starting position is optional and cells are assumed to not have strict rails. The width that is passed during initialization is `w_si`. This is because `w` is calculated of the other parameters as it depends on the placement of the rails. `w` is calculated as follows.

$$w = w\_si + 2 \times length(top) - 1 + 2 \times length(bot) - 1$$

The `topNets` and `botNets` parameters are lists containing the nets above and below the cell, afterwards `top` and `bot` are initialised with the correct Rail instances. The Rail class stores the state of these rails, but they still rely on the cell class for information related to position. The order of the rails is determined by the two lists storing them. The first rail in `top` is the first rail at the top of the cell and the first rail in `bot` is the rail at the outside, below the cell. So the order is based of how far the rails are from the cell.

Cell	
id:	str
bulk:	net
w:	int
l:	int
w_si:	int
x:	int
y:	int
top:	List[Rail]
bot:	List[Rail]
strictRails:	Boolean

Table 4.1: Cell architecture

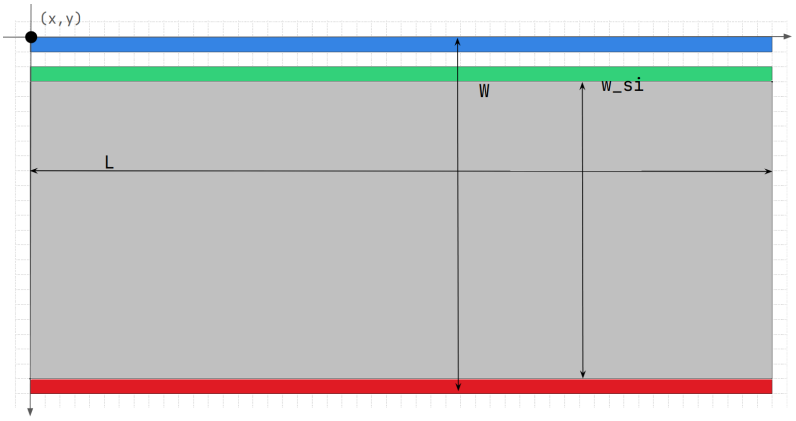


Figure 4.1: Cell representation

4.1.2 Bounding Box

As mentioned above the overlapping of cells is done by comparing the bounding boxes(bbox) of the cells. A `Bbox` object is defined by the top left coordinate and the bottom right coordinate of the bounding box, the variables are shown in table 4.2 and they are visualised in figure 4.2. A bounding box can be initialised by passing these 4 values or by passing a `Cell` object. When a cell is given, the bulk of the cell is also stored. A `Bbox` object utilizes algorithm 6 to detect collision. If two cells need to be spaced apart because a different biasing of the bulk, the bounding box can be easily padded and by applying the same collision algorithm, the two cells will be spaced apart correctly.

Bbox	
x1:	int
x2:	int
y1:	int
y2:	int
bulk:	net

Table 4.2: Bbox attributes, x1 < x2, y1 < y2

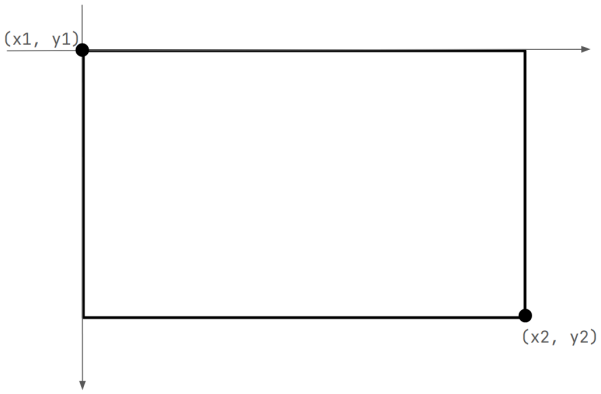


Figure 4.2: Coordinates of a bounding box

When passing a `Cell` object to the `Bbox`, the coordinates are converted as follows: `Bbox.x1 = Cell.x, Bbox.y1 = Cell.y, Bbox.x2 = Cell.x + Cell.l - 1, Bbox.y2 = Cell.y + Cell.w - 1`

Algorithm 6 Bounding box collision

```
function collide(Bbox_1, Bbox_2)
    if (Bbox_1.x2 < Bbox_2.x1) or (Bbox_1.x1 > Bbox_2.x2) then
        return False                                  ▷ No overlap in x dimension
    else if (Bbox_1.y2 < Bbox_2.y1) or (Bbox_1.y1 > Bbox_2.y2) then
        return False                                  ▷ No overlap in y dimension
    else
        return True                                   ▷ Overlaps for both axes
    end if
end function
```

4.1.3 Rail

The `Rail` object was mainly introduced to simplify the overlapping of rails and the swapping of the rails, its properties are listed in table 4.3. Rails are created upon the initialization of the `Cell` class. The following properties (`x1, x2, y, index, top, net`) are derived from the `parent` cell. `merged` keeps tracks if the rail is merged with other rails. If this is the case, the other `Rail` objects are stored in `connections` Merging rails refers to an action described later on. It boils down to multiple cells sharing the same rail. This reduces the cost a lot, as the amount of metal required to route the net is very small. This action was the main reason a `Rail` Class was required.

Rail	
x1:	int
x2:	int
y:	int
top:	Bool
net:	net
index:	int
parent:	Cell
merged:	Bool
connections:	List[Rail]

Table 4.3: Rail attributes, x1 < x2

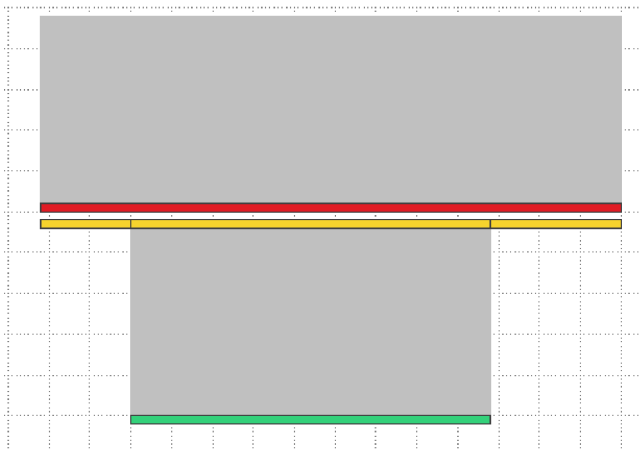


Figure 4.3: Demonstration of overlapping or merged rails

The `Rail` class provides a method to find the length of the rail that is overlapping with other cells. The pseudocode to do this is shown in algorithm 7. This value is required to correctly calculate the length of all the rails, in order to calculate the global length of the layout for the placement cost.

This instance does come with a downside. As the information of the rail depends for the most on a `Cell` object, the rails need to be updated each time the position of a primitive changes. To tackle this, the method `updateRails` was introduced, which needs to be called each time an instance of `Cell` is updated. The pseudocode for this method is shown in algorithm 8.

Algorithm 7 Length of overlapping rails

```
function OVERLAPLENGTH(rail)
    length ← 0
    for all connection in rail.connections do
        x1Max ← max(connection.x1, rail.x1)
        x2Min ← min(connection.x2, rail.x2)
        length = length + x2Min − x1Max
    end for
    return x1Max, length
end function
```

Algorithm 8 Update the parameters of the rails after any action is executed

```
function UPDATERAILS(cell)
    cell.w ← cell.w_si
    if (length(cell.top) > 0) then
        cell.w ← cell.w + 2 × length(cell.top) − 1
    end if
    if (length(cell.bot) > 0) then
        cell.w ← cell.w + 2 × length(cell.bot) − 1
    end if
    i ← 0
    for all rail in cell.top do
        e.index ← i
        e.top ← True
        e.y ← self.y + i × 2
        e.x1 ← self.x
        e.x2 ← self.x + self.l − 1
        i ← i + 1
    end for
    i ← 0
    for all rail in cell.bot do
        e.index ← i
        e.top ← False
        e.y ← self.y + self.w − i × 2 − 1
        e.x1 ← self.x
        e.x2 ← self.x + self.l − 1
        i ← i + 1
    end for
end function
```

4.1.4 Router and Grid

These three classes are everything required to represent the primitive. There are two large classes left. The first one is the `Grid` class, with the properties listed in table 4.4. The grid class contains the list of primitives and represents the total area containing the cells. Each action that involves more than two cells is implemented here, else it is implemented inside the `Cell` object. The specific implementation of the grid initialization and actions will be discussed in the following sections. The grid does not require any arguments on creation. However, to have correct behavior the individual `Cell` objects have to be passed to a `Grid` object using the `addCell` method. This method updates multiple values of the `Grid`. By passing all the cells, the aiding dictionaries `railsFromNet` and `cellsFromNet`, which allow finding specific `Rails` or `Cells` quickly, are initialised. Also the `nets`, `maxCellLength` and `maxCellWidth` are calculated.
The current allowed amount of metal layers is stored in the `layers` parameter. After the routing is complete the resulting traces are stored in `traces`.

Grid	
gridWidth:	int
gridLength:	int
cells:	List[Cell]
nets:	List[net]
maxCellLength:	int
maxCellWidth:	int
traces	List[Trace]
layers:	int
railsFromNet:	Dictionary[net, List[Rail]]
cellsFromNet:	Dictionary[net, List[Cell]]

Table 4.4: Grid properties

The last class is the `Router` class which controls the algorithms. Its first argument is the corresponding `Grid` object that contains the to be routed and placed primitives. It executes the annealing algorithm and contains wrappers for the routing algorithms and the different actions. This class is also responsible for storing the previous best iteration and determining the cost of the current iteration. The second and last argument is the path of a configuration file, which contains all the tunable parameters of the layout generator.
Besides the annealing algorithm explained in the previous chapter, this class contains only wrapper methods to provide simple execution of the placement and routing. The specifics of the actions and routing algorithms will be explained in the following sections.

4.2 Initializing Grid and Router

In earlier versions the `Grid` class contained an actual 2D list, where each value represented a 1 by 1 unit cell. The grid contained zero for empty space, non-zero integers represent the nets or cells. This grid was used for the routing and the actions/placement. Collision detection of the cells was also performed by checking the value of the list corresponding to the specific cell. As mentioned earlier the use of bounding boxes was adopted quickly, to increase performance and scalability. From this point on the data structures for placement and routing were split.

The placement uses the `Bbox`, `Cell`, `Rail` and `Grid` objects for implementing the actions. The initialization of these objects were described earlier, except for the latter. The initial sizing of the grid and placement of the primitives are the first step in the annealing algorithm. This requires placing the subblocks at random on a seemingly infinite grid. An initial placement of a simplified opamp circuit can be seen in figure 4.4.

By default, the initial grid width is twice as large as the sum of widths of the cells and the same is true for the grid length. This way a nice trade-off between initial performance and randomness is obtained. A larger initial grid size significantly increases the execution time due to the much larger routing time.

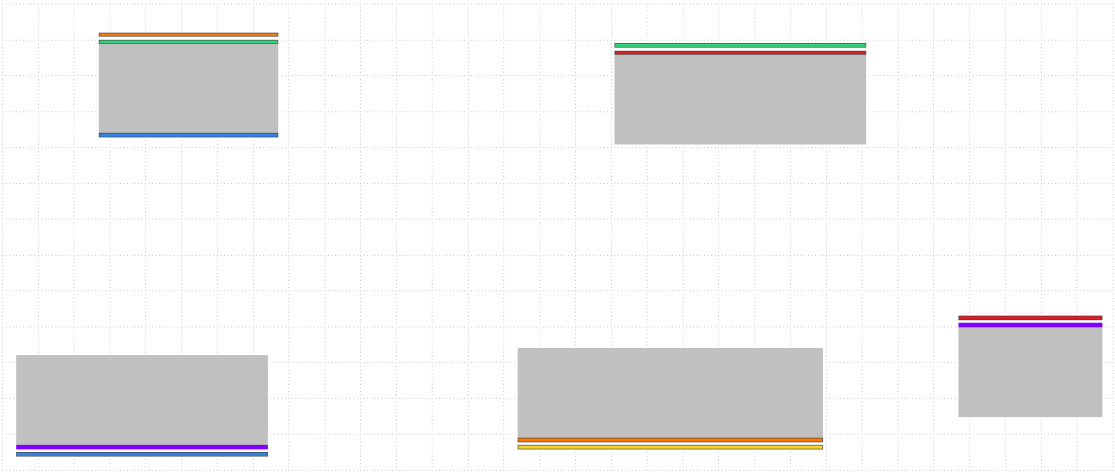


Figure 4.4: Random placement of cells

After the `Grid` was initialised by adding all the primitives as `Cell` objects, the grid sizing is calculated as mentioned earlier. Following this, algorithm 9 is used to place the primitives randomly in the grid. The `collision` method of the `Bbox` class ensures proper spacing of the cells.

Next, the initial placement is routed and therefore the data structure for the router has to be initialised. The state of the router is stored in a numpy [7] array of integers with dimensions `layers x gridWidth x gridLength`. The meaning of each integer value is shown below. The grid represents the placement of the primitives and the routing can explore the grid where the values are zero or equal to the value of the being routed net. The position of the primitives directly correspond to the indices on this routing grid, because of the quantization of the traces.

0: Unoccupied space

1: Do not route, this represents space occupied by the primitives.

2: This represents space occupied by the primitives, which allow interconnections above the devices.

3-9: Placeholder values

10-999: These values represent a certain net. 10 represents the first value in the `nets` property of a `Grid`

1000- ... : These numbers also represent a net, where 1000 refers to the same net as 10 previously those. These values are used for when the net goes up or down a layer, this is to be able to space vias accordingly.

Algorithm 9 Initialize placement

```
function GRID.INITPLACEMENT()
    for all cell in Grid.cells do
        i ← 0
        while i < 100 do
            cell.x = random(0, gridLength − cell.l)
            cell.y = random(0, gridWidth − cell.w)
            if (grid.checkOverlap(cell)) then
                break
            end if
            i ← i + 1
        end while
        if i = 100 then                          ▷ after 100 tries resize the grid
            increase grid for more space
        end if
    end for
end function
function GRID.CHECKOVERLAP(curCell)
    curBbox = Bbox(curCell)
    for all cell in Grid.cells do
        if cell.id = curCell.id then continue
        end if
        bbox = Bbox(cell)
        if ¬(cell.bulk = curCell.bulk) then
            bbox.extend(padding)          ▷ padding is the design rule spacing for distinct wells
        end if
        if collision(curBbox, bbox) then
            return False
        end if
    end for
    return True
end function
```

Before the routing grid is filled with these values, one more algorithm is performed on the cells and grid. Because the grid is infinite the routing should also be able to go at the edge of the grid. Therefore the gridsize and cell positions are optimised before the routing. This is also useful to limit memory usage. If the grid remains at the initial size, which is way larger than the final layout, the memory occupation is much larger than when the gridsize would dynamically scale according to the current layout size.

Another reason to resize the grid and rearrange the cell positions is due to the actions of SA. When a cell is moved, outside the current gridsize, the gridsize needs to change. Also a cell can be moved anywhere, which means the position of the cell can be negative. Because these positions correspond to indices in the routing grid, the cells have to be repositioned such that each cell has positive integer coordinates. An example of this method is shown in figures 4.5. These figures display an initial placement, the position after the actions are performed and the final resized grid.

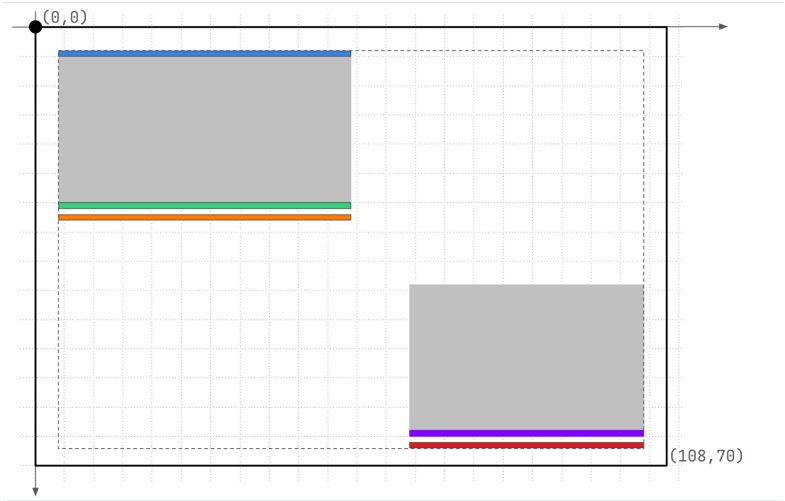
The pseudocode for the method `resetDimensions` is shown in 10. The new dimensions are found by iterating over all cells and finding the minimal and maximal values for both directions. In the figures, this is highlighted by the dotted bounding box. Then the coordinates of the cell is accounted for by the appropriate amount, such that the each coordinate is a positive value. Because each cell changes position, it is required to always update the rails of the cells.

This algorithm can also be used to add padding to the grid, which is useful for more complicated schematics, that require more routeable space at the edge of the grid. This algorithm can move all the cells more to the center, creating more space. This feature will also be useful for certain actions. By default 4 units are added at each side of the grid.

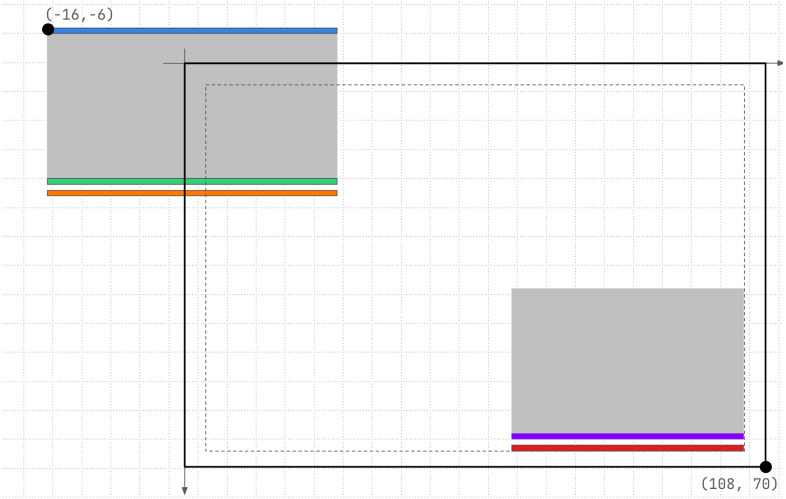
Algorithm 10 Reset Dimensions

function GRID.RESETDIMENSIONS(px, py)
    $xMin \leftarrow$ `Grid.gridWidth`
    $yMin \leftarrow$ `Grid.gridLength`
    $xMax \leftarrow 0$
    $yMax \leftarrow 0$
    for all $cell$ in `grid.cells` do
        $xMin \leftarrow min(xMin, cell.x)$
        $yMin \leftarrow min(yMin, cell.y)$
        $xMax \leftarrow max(xMax, cell.x + cell.w - 1)$
        $yMax \leftarrow max(yMax, cell.y + cell.l - 1)$
    end for
    `Grid.gridLength` $\leftarrow xMax - xMin + 2 * px$
    `Grid.gridWidth` $\leftarrow yMax - yMin + 2 * py$
    for all $cell$ in `grid.cells` do
        $cell.x \leftarrow cell.x - (xMin - px)$
        $cell.y \leftarrow cell.y - (yMin - py)$
        $cell$`.updateRails()`
    end for
end function

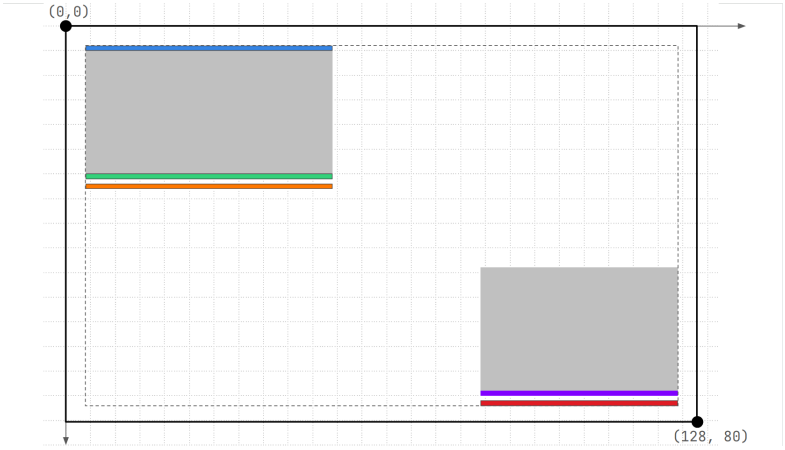
Now that the data structures for the placement and routing is fully explained, the actions for the simulated annealing and the improved A* algorithm for the routing can be explained.



(a) Initial placement



(b) One of the cells is moved



(c) The positions of the cells are adjusted and the grid is resized

Figure 4.5: Demonstration of grid resizing and changing the cell position before the routing

4.3 Actions

This section discusses in depth the implementation of each action. Some actions were mentioned earlier, like swapping rails, moving cells and merging rails. The introduction of the rail merging action required some modification to the other actions. Because, when two rails are merged together, the parent cells are not as flexible anymore as their position is reliant on the overlap. Actions, such as the move cell action, should behave differently for merged cells. As many edge cases exist it is too much to take care of them all. The actions are developed as general as possible to work in most situation. The edge cases which are taken care of are demonstrated.
Sometimes the design rules are violated, to make sure this does not propagate to the final solution a sanity check is introduced after the routing. This checks for overlapping cells and shorted nets. If one of these rules is violated, the current cost becomes infinite, making sure this configuration does not propagate.

4.3.1 Dealing with collision

Some actions require moving a cell to a certain position, which could result in two cells colliding. To solve this issue, a method was introduced which 'evacuates' cells from a certain bounding box. This ensures that the cell can safely move to that specific position. The reason its referred to as evacuating a cell, is due to this method taking the shortest possible direction to avoid collision. This may not be the best way to move a cell out of the way, but it is the most consistent. An earlier version implemented a different method, where depending on the relative positions of the cells a fixed direction was chosen to move the colliding cell(s). This method however was very inconsistent and too complex.

A figure explaining the logic behind finding the evacuation path is shown in 4.6. Because of an action, Cell 1 is moved to that specific position, which makes it collide with Cell 2. Next, for each cardinal direction an escape path is considered. The length for each path is calculated based on the two bounding boxes. Consequently, the path with the shortest distance is chosen. In the example, the shortest escape path has distance of 10. This means cell 2 is moved 10 spaces plus the correct padding (1) to the right.

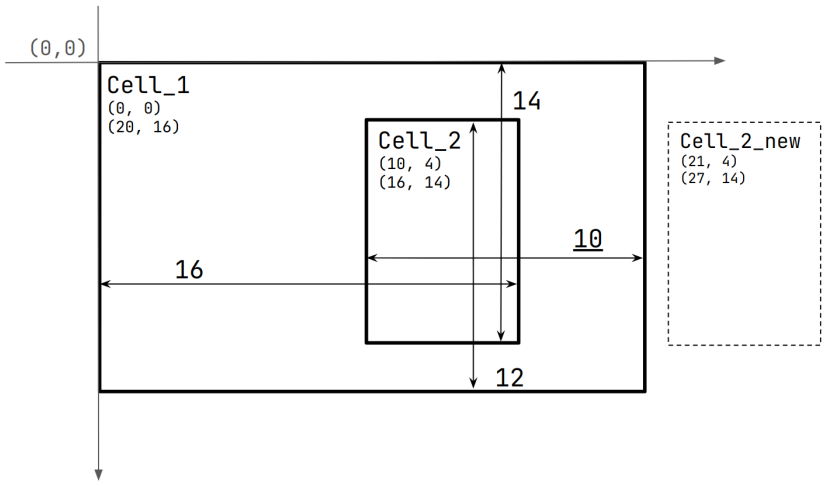


Figure 4.6: Demonstration of `evacuateCell`. Cell 2 should be moved out of the way. The distances for each cardinal direction is displayed. Moving to the right is the best option, as it is distance 10 away. The top right and bottom left coordinates are displayed for each cell.

Now this method, called `findExit`, provides the point of the cell which should be moved, together with the evacuation distance and whether the movement is horizontal or vertical. For the given example, these values are the following: horizontal movement, distance 10 and the coordinate is 10. Another method handles this information to actually move the cell. The reason that this is handled by a different method is that it is not straightforward, as another cell could be present at the new location, meaning the collision is just transferred to a different combination of cells.

A possible solution for this would be to recursively evacuate cells, until everything has a new distinct position. But this could amount in very hectic movement or even worse infinite recursion. This problem could be tackled by simply trying the recursive approach and if a recursion limit is reached, the action causing the collision is simply canceled.

However, a different approach was employed that only required minimal evacuations. Instead of only moving a single cell, all the cells past the anchor point where moved, basically inserting empty space for the new cell. When applied to the previous example this comes down to the following. If a third colliding cell was added below cell 2, outside of cell 1, it would also be moved 11 spaces to the right. This method seems worse as it will increase the area most of the time, but on the other hand it will retain the relative placement between the cells, which will result in similar routing.

An example of how a single obstacle is handles is displayed in figure 4.7. In this case all cells that are present to the right of the anchor and in between the dotted horizontal lines are moved the evacuation distance to the right. Figure 4.8 shows an example with an additional colliding cell. The algorithm then behaves differently. At first it selects the cell with the longest evacuation distance. Then all the cells to the right of the anchor, for the entire width of the grid, are moved that distance to the right. Each time a cell is evacuated the collided cells are reevaluated. This is necessary as moving cell 2, evacuated the third cell as well, which is very beneficial.

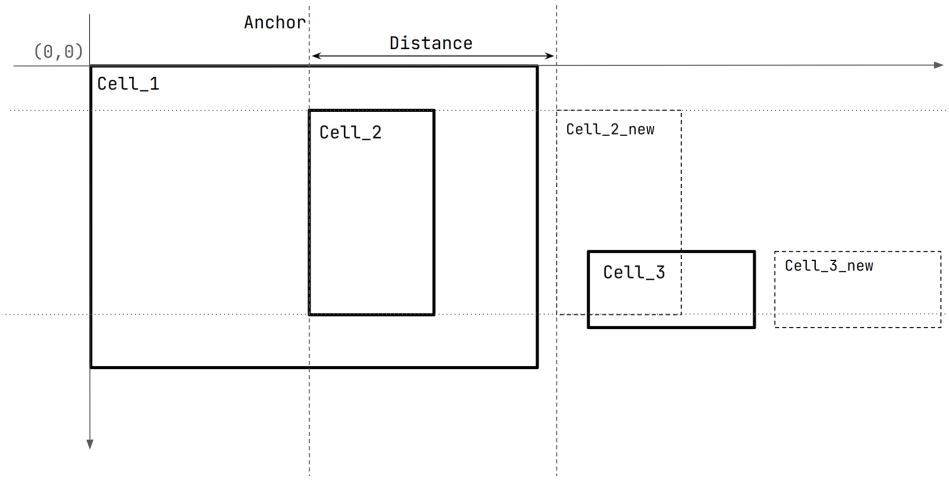


Figure 4.7: Evacuation of a single cell, only moves the section containing the cell. Cell 3 retains the same position relative to cell 2

It is not always the case that the largest exit distance evacuates other colliding cells. An example requiring two distinct evacuations is shown in figure 4.9. Cell 2 has the longest evacuation distance. It is moved according to the vertical anchor. Cell 3 is moved as well, because a part of its bounding box is present in the bounding box defined by the grid width and anchor. However, cell 3 still collides with cell 1. It is moved down according to the new evacuation distance and horizontal anchor. Resulting in its final position down right from the original one.

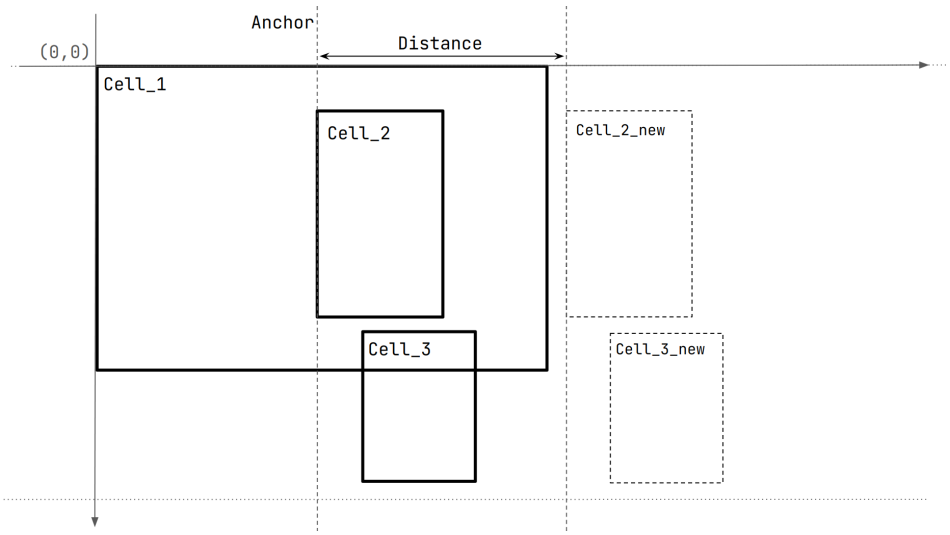


Figure 4.8: When multiple cells collide, the full width or length is considered when moving cells aside.

The explanation was mainly performed for horizontal movement, but it behaves the same for vertical evacuation. However, care must be taken that a vertical evacuation does not result in rails being unmerged.

Now the general concept is explained, the pseudocode for the multiple algorithms mentioned is shown in 11. The algorithm explains the main method `handleEvacuation`, which accepts a repositioned `Cell`, this would be cell 1 when looking at the previous examples. By default `mergesafe` is false, which restricts the evacuation to the x-axis.

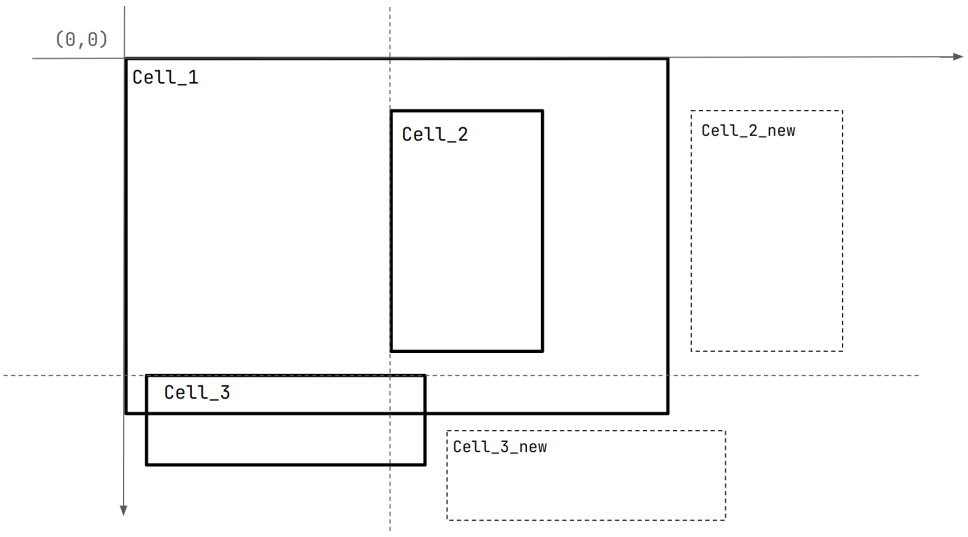


Figure 4.9: Two distinct evacuations are required, which results in cell 3 moving twice.

Algorithm 11 handle evacuation of cells

```
function GRID.HANDLEEVACUATION(cell, mergeSafe)
    bbox ← Bbox(cell)
    collisions ← Grid.cellsInBbox(bbox)
    if (length(collisions) = 1) then
        (Horiz, distance, anchor) ← findExit(bbox, collision[0], mergeSafe)
        Grid.moveGridSection(horiz, anchor, distance, collision[0].y, collision[0].y +
collision[0].l)
    else
        for all collision in collisions do
            (Horiz, distance, anchor) ← findExit(bbox, collision[0], mergeSafe)
            Grid.moveGridSection(horiz, anchor, distance, 0, grid.gridlength)
            collisions ← Grid.cellsInBbox(bbox)
        end for
    end if
end function
function GRID.CELLSINBBOX(curCell)
    curBbox ← Bbox(curcell)
    cells ← []
    for all cell in Grid.cells do
        bbox = Bbox(cell)
        if ¬(cell.bulk = curCell.bulk) then
            bbox.extend(padding)
        end if
        if collision(curBbox, bbox) then
            cells.append(cell)
        end if
    end for
    return cells
end function
```

4.3.2 Moving cells

Moving a cell is the most essential action and three variants of this actions exist. For high temperatures this action has the most merit, because random movement will likely improve the cost. However, as the temperature gets lower, the actions need to be more precise and moving a cell results most of the time in a worse position. The basic move action does not increase the gridsize before moving a cell. A second version does increase the gridsize based of the cell which will be moved. By passing the dimensions of the selected cell to the `resetDimensions` method, together with appropriate margins, the selected cell can be moved everywhere around the grid. Both actions can be seen in figures 4.10 and 4.11.

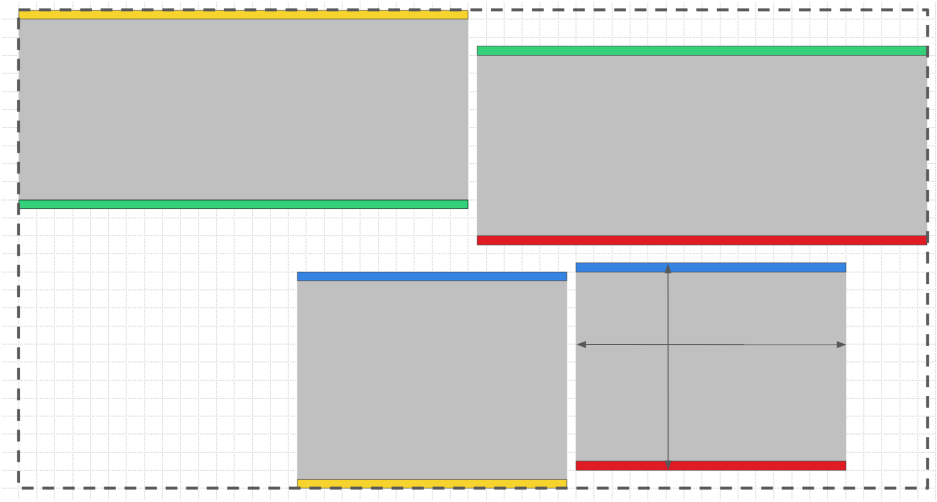


Figure 4.10: The cell in the bottom right is moved, but has very little space to displace to.

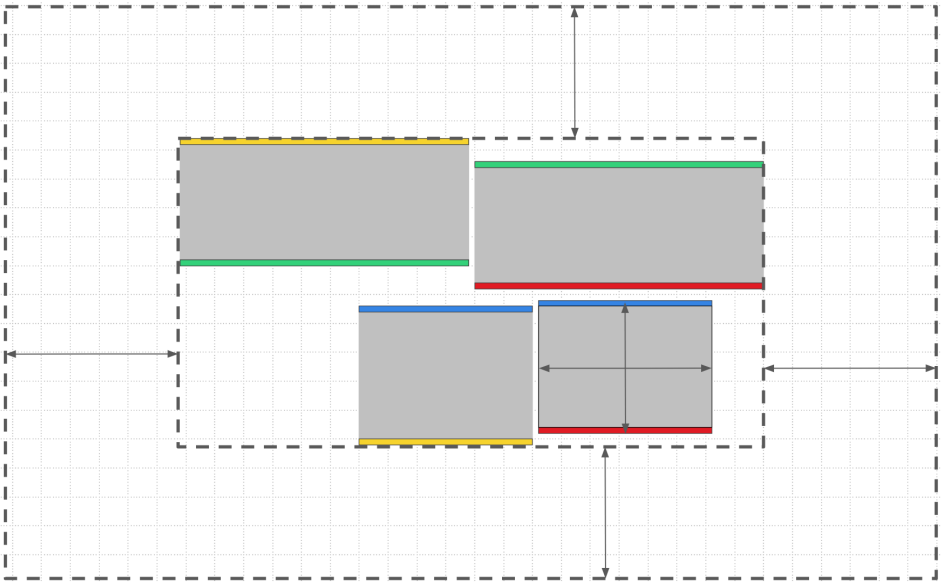


Figure 4.11: The cell in the bottom right is moved after a refit, the cell can be placed everywhere

If this second version would not be available, the cell movement would be very limited. In some cases this limited movement is good, as this increases the chance a precise move is found. But if a cell needs to moved to the other side, this will not happen. Both actions can be chosen by the placement algorithm. They are referred to as 'move cell' and 'refit and move cell'.

The pseudocode for the move actions is shown in algorithm 12. As without resizing the chance a new position is valid is small, the grid is automatically refit after a certain amount of iterations. This algorithm has similar behavior as the initial placement of the cells.

Algorithm 12 Basic move action and refit move action.

```
function GRID.MOVECELL(cell, refit)
    cell.unmerge()
    if (refit = True) then
        Grid.resetDimensions(cell.l + 5, cell.w + 5)
    end if
    i ← 0
    while i < 100 do
        cell.x = random(0, gridLength − cell.l)
        cell.y = random(0, gridWidth − cell.w)
        if (grid.checkOverlap(cell)) then
            break
        end if
        i ← i + 1
    end while
    if i = 100 then                          ▷ after 100 tries, change to refit variant
        Grid.moveCell(cell, True)
    end if
end function
```

The third version of the move action has to do with how this actions interacts with merged rails. As the two variants just described move the single cell away, the cells rails are unmerged, before its position is changed. When a move action is executed, the wrapper in the `Router` class selects a random cell. If this cell is merged with another cell, it is determined by chance which variant of the move function is executed on this cell. It chooses between the basic move action and a specific move action for merged cells. This algorithm is given in 13, the `updateMerge` function just recalculates the new overlap length with the algorithm explained earlier. This specific action allows the cell to move along the rail it is merged with, thus allowing only some horizontal movement. This way merged cells can be rearranged to align better and overlap more. This action allows the cell the move at most its own length to either the left or the right. This was added because previously once cells were merged they were not able to move sideways will staying merged, as the normal move action would always undo the merging of rails.

Algorithm 13 Move action for merged cells, which restrict to horizontal movement

```
function GRID.MOVECELLMERGED(cell, refit)
    i ← 0
    while i < 100 do
        cell.x = random(max(0, cell.x − cell.l), min(gridWidth, cell.x + cell.l))
        if (grid.checkOverlap(cell)) then
            break
        end if
        i ← i + 1
    end while
    if i = 100 then                                   ▷ after 100 tries, change to basic variant
        Grid.moveCell(cell, False)
    end if
    cell.updateOverlap()
end function
```

4.3.3 Swapping Rails

The second action: swapping rails, was the next step to lowering the cost even more. Together with moving the cells, a quite optimal solution could almost always be reached. These two actions already introduce a large amount of creativity to the layout generator, due to the very simple nature of these actions and them working in every situation (unless the overlapping rails are introduced).

This actions just changes the order of the rails of a cell and this consists of two cases. Again a cell is selected by the wrapper for this action and then one of the two is handled. The first case changes the position of a single rail and the other one swaps two rails. In figure 4.12 these two cases are demonstrated.

Figure 4.12: Example of both cases of the swap rail action; The left cell swaps the two rails and the right cell changes the position of the blue rail.

Swapping two rails is very easy, for the rails are stored in two arrays, one for the rails above the cell and one for the rails below. It suffices to check which arrays contains the selected rails and then swapping them correctly depending on the selected array.

Implementing the swap variant is not strictly necessary as it combines two sequential swap rail actions of case 1, which defeats the purpose of simulated annealing. It was added for the sake of simplicity and a faster result, but in hindsight, however, it is obsolete.

Changing the position of a single rail is a bit more complicated. To move a single rail, all available new positions are determined and one of these is selected. Figure 4.13 shows two examples of the first case and demonstrates how the available positions are determined. These positions will now be referred to as indices in the combined array of `cell.top` and `cell.bot` called `rails`.

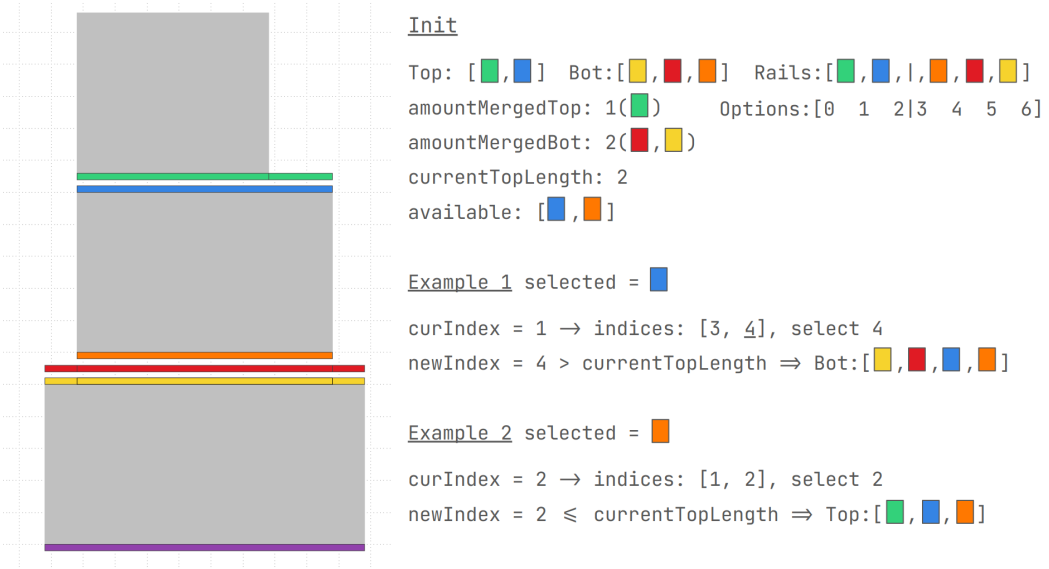


Figure 4.13: Demonstration of swapping rails. The determining of the indices list is visualised in detail

All available indices are determined first. This is denoted as options in the image and this goes from index 0 to the amount of rails incremented by one. Then based on how many rails are merged, certain indices are removed. As one rail is merged at the top (the green rail), index 0 is removed from the array. For the same reasoning but at the other side of the cell, 5 and 6 is removed. Then the available array is obtained.

In order to not insert the selected rail to its old location, the two options above and below this selected rail are removed. For the first example, the blue rail is selected. This means index 1 and 2 are removed from the options, which is the index of the selected element and the next one. For the second example, where the selected rail is not in `top`, this is index 3 and 4. Therefore, the rails below the cell, this comes down to `currentIndex + 1` and `currentIndex + 2`.

The result of the two examples are shown in figures 4.14 and the pseudocode for `findIndices` is shown in algorithm 14. The pseudocode for the actual action is shown in algorithm 15, explaining both cases.

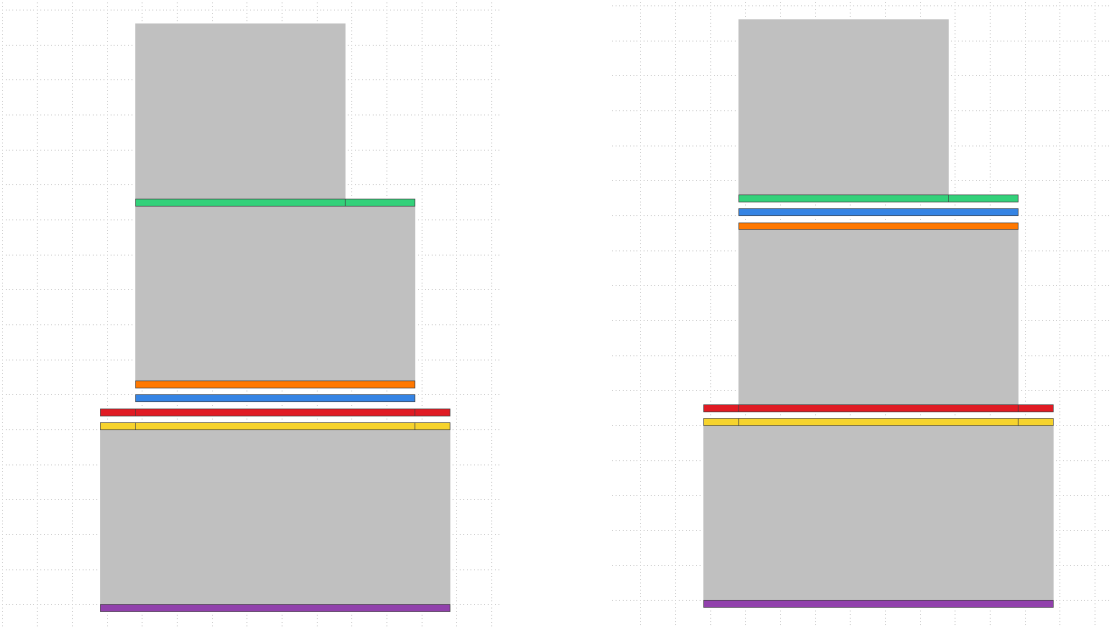


Figure 4.14: Example of both cases of the swap rail action

Algorithm 14 Custom logic which finds the available indices for the selected rail

function CELL.FINDINDICES($rail$)
    $rails \leftarrow$ `cell.top` $+ reverse($`cell.bot`$)$
    $currentIndex \leftarrow rails.index(rail)$
    $amountMergedTop$
    $amountMergedBot$
    $currentTopLength \leftarrow length($`cell.top`$)$
    $indices \leftarrow$ []
    for $i = amountMergedTop$ to $length(rails) + 1 - amountMergedBot$ do
        if $(rail.top$ and $\neg(i = currentIndex$ or $i = currentIndex + 1))$ then
            $indices.append(i)$
        else if $(\neg rail.top$ and $\neg(i = currentIndex + 1$ or $i = currentIndex + 2))$ then
            $indices.append(i)$
        end if
    end for
    return $indices$
end function

Algorithm 15 Swap Rail action for cells with no stricts rails

```
function CELL.SWAPRAIL(case)
    available ← []
    rails ← cell.top + reverse(cell.bot)
    currentTopLength ← length(cell.top)
    for all rail in rails do
        if ¬(rail.merged) then
            available.append(rail)
        end if
    end for
    if case = 1 or length(available) = 1 then
        rail ← selectRandom(available)
        indices ← findIndices(rail))          ▷ select indices as described and visualised above
        if length(indices) = 0 then
            return False
        end if
        newIndex = selectRandom(indices)
        if newIndex > currentTopLength then
            cell.assignRail(rail, length(rails) + 1 − newIndex, False)
        else
            cell.assignRail(rail, rail1.index, True)
        end if
    else if case = 2 then
        rail1 = selectRandom(available)
        rail2 = selectRandom(available)
        cell.assignRail(rail2, rail1.index, rail1.top)
        cell.assignRail(rail1, rail2.index, rail2.top)
    end if
    cell.updateRails()
    return True
end function
function CELL.ASSIGNRAIL(rail, newIndex, newTop)
    if rail.top = True then
        cell.top.remove(rail)
    else
        cell.bot.remove(rail)
    end if
    if newTop = True then
        cell.top.insert(rail, newIndex)
    else
        cell.top.insert(rail, newIndex)
    end if
end function
```

The swap rail action suffers the most from the constraints on cells and the addition of merging rails. The merged rails can obviously not be swapped as this would mean rails with different nets be will be merged creating shorts. The other problem is the cells with the strict rails, the wrapper function checks whether this is the case for the selected cell and executes an adapted variant of the algorithm. This adapted versions just executes the algorithm for swapping rails, but now `rails` only includes `bot` or `top` based of chance. The only difference is that there is a third case for the strict rails cells. To introduce enough randomness the two lists containing the cells can be interchanged, visually this comes down to flipping the cell.

This action can also cause shorts unwillingly as the dimensions of the cell can change if one of the sides of the cell becomes empty, this increases the width of the cell by one. An example is shown in figure 4.15, the rail containing the yellow net is swapped and without moving any cell, two nets are shorted. This is handled by performing the sanity check and the actions that cause this problem are rejected. The swap action can also be rejected for other reasons, that is why the algorithm returns a boolean.

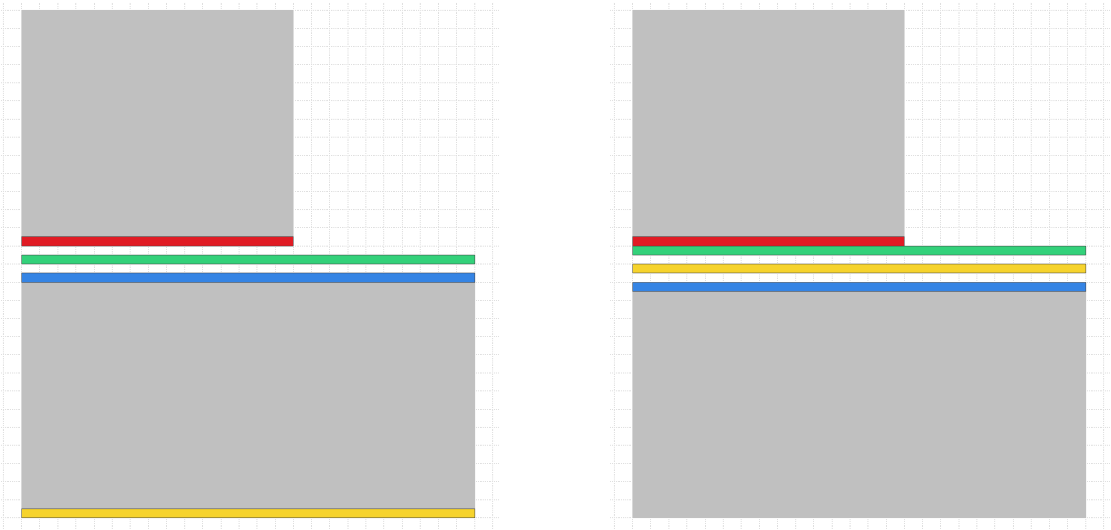


Figure 4.15: Short after swapping rails

4.3.4 Swapping Cells

Another action allows two cells to swap. This was added as a different solution for the problem mentioned in chapter 3, where a layout was shown which did not allow further improvement and required multiple actions. This action is straight forward and works as follows.

The wrapper function of the `Router` selects two cells at random. The anchor point of each cell is determined. This anchor point is the coordinate where the other cell swaps to and is required to derive its new position. This is fixed for each quadrant of the grid to have a consistent algorithm. This anchor point is the point of the bounding box closest to the center of the grid. In the example, shown in figure 4.16, the anchor is bottom right for cell 3 and bottom left for cell 5. When the cells swap, the anchor of the other cell is adopted and the cell is placed at that point.

This first example is trivial, because the two cells can just swap position. The result is shown in figure 4.17 . This is the simplest case, however this action is the first one to require the evacuation method stated earlier. The need for this is demonstrated in figure 4.18. In this example swapping cell 1 and cell 5, without taking into account the other cells, would result in collisions. The anchors and new bounding boxes are also shown as hatched versions of the cells. To avoid this overlap, it suffices to execute the `handleEvacuation` method. After using this method the resulting layout is shown in 4.19.

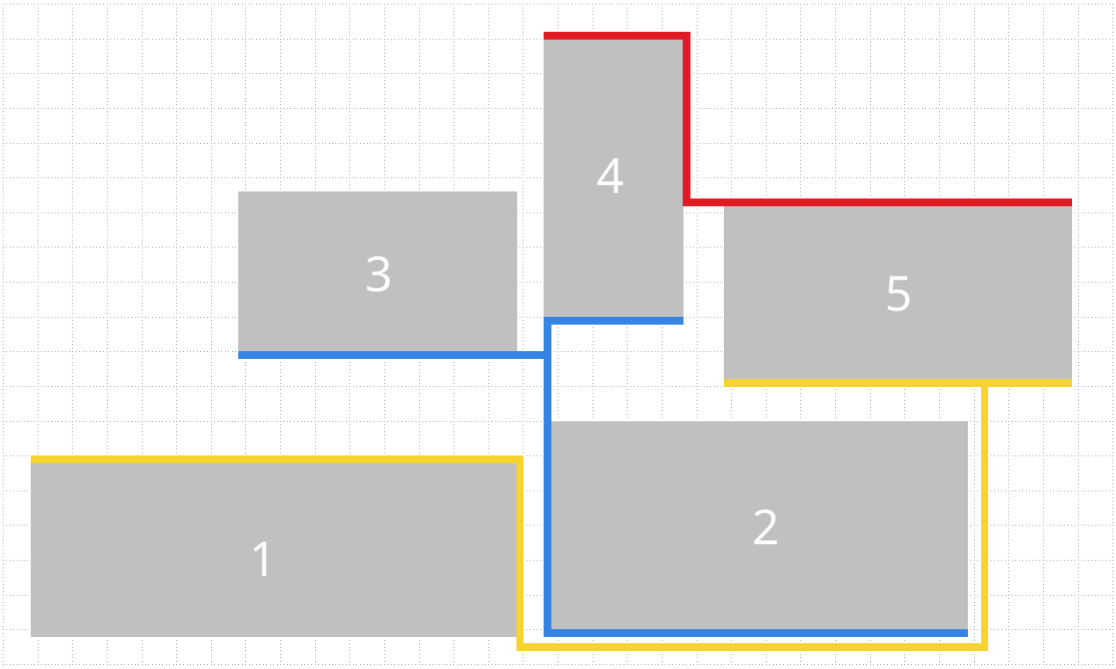


Figure 4.16: Initial configuration

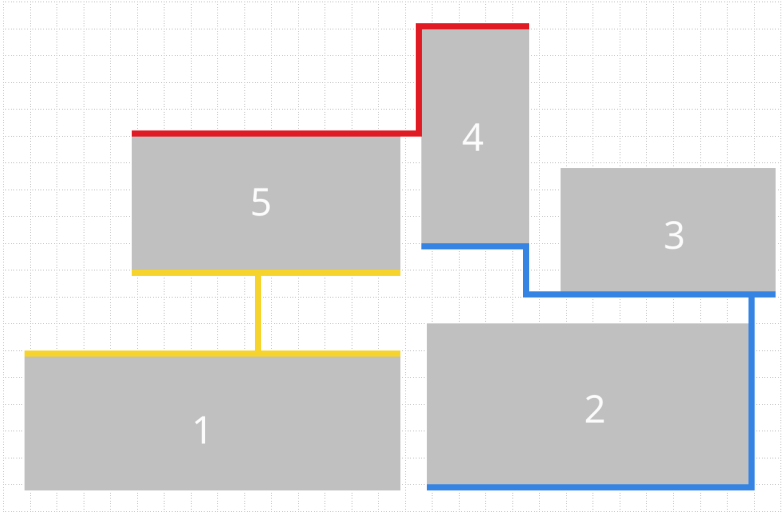


Figure 4.17: Simple swap action concerning cell 3 and 5

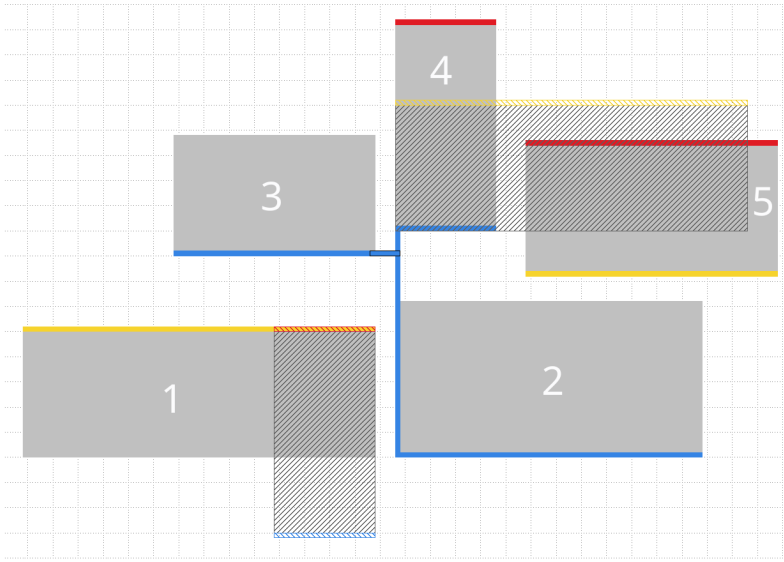


Figure 4.18: Anchor and new bounding boxes when swapping cell 1 and 4

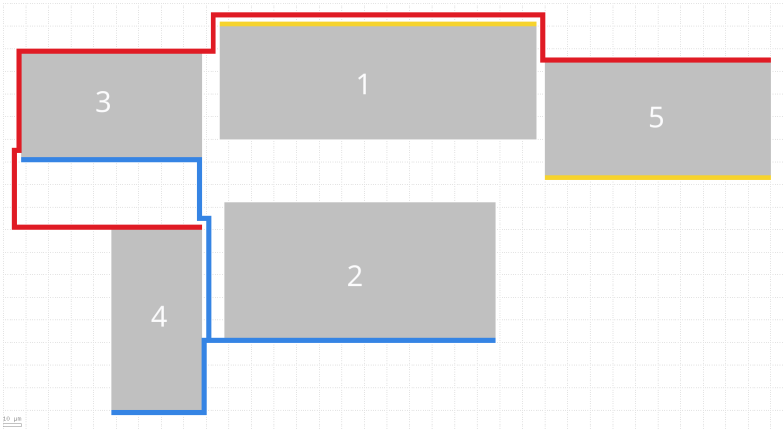


Figure 4.19: Cell 5 moved out of the way to make place for cell 1, cell 4 could move without any problems

Quadrant	cell.x	cell.y
1	anchor.x - cell.l	anchor.y - cell.w
2	anchor.x	anchor.y - cell.w
3	anchor.x - cell.l	anchor.y
4	anchor.x	anchor.y

Table 4.5: anchor of a cell after swapping with another cell based of the anchor of the other cell

Figure 4.20 displays how the anchor positions are determined and this is based on in which quadrant the center of each cell was. Table 4.5 shows how the new x and y of the cells are calculated.

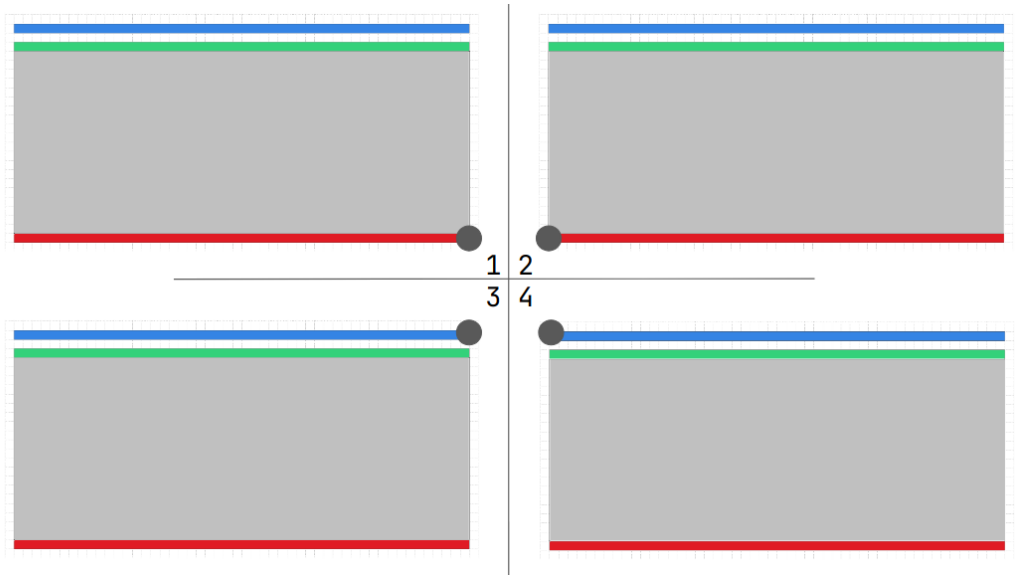


Figure 4.20: Different swap anchors based of the quadrants, always the point closest to the center of the grid. The crossing of the axes denotes the center of the grid.

4.3.5 Merging Cells

The most influential action is already discussed a lot because of the impact on the other actions. What follows is the discussion of the behavior of the action that merges the rails of cells carrying the same net. A basic example of this can be seen in figure 4.21. This action has the most potential to lower the cost at the later stages of the algorithm, as a lot of metal can be omitted. Merging cells just means that the two rails overlap, thus sharing the signal. The amount of cells that can merge is unlimited, as long as no shorts happen. Also a single rail can be shared by multiple cells and two cells can have more than one overlapping rail.

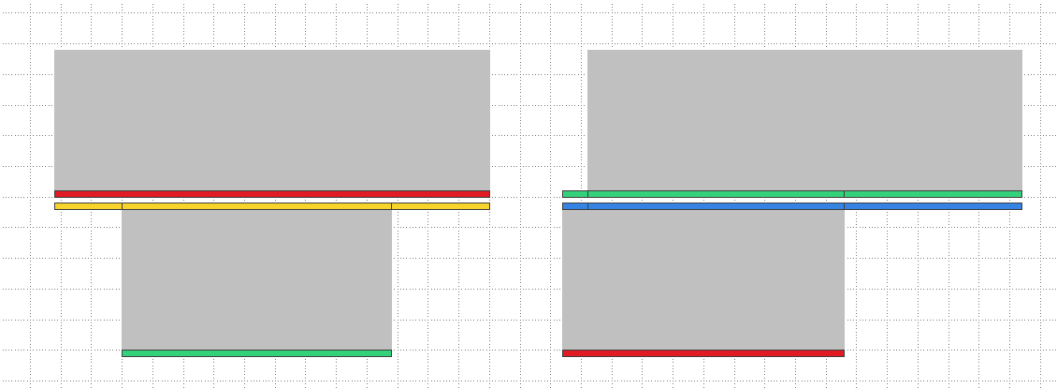


Figure 4.21: 2 examples of 2 cells merging

While this action is very useful, it comes with complex implementation. Balancing this action is also difficult, as the actions should be very random and simple. But this action could control the algorithm too much, rendering the other actions useless. The reason this action is so powerful is that it, besides reducing metal, moves cells and swaps rails. Again taking away the work of the simulated annealing.

The merging of rails is however a crucial step in reducing the parasitic resistance in IC design. Because the difficult nature of this move, it was briefly explored to not implement this as an action but as an extra step before the routing. This would allow the algorithm to do every available merge thus reducing the cost a lot. But due to previous mentioned reasoning, this was not done as it would be too influential.

To balance this action correctly it needs to be determined how much parasitic resistance can be reduced by a single merge action. An extreme example is shown in figure 4.22. This shows the combination of cells which overlap with the center cell, if each cell was considered. The actual implementation of this action performed at maximum two merges at once, one at each side of the cell, selected by the wrapper method of the `Router` class. When applying this to the example, the lower cell containing the red and green net would be merged below the cell and the subblock on the top right would overlap the rails for the blue and yellow net, above the selected cell.

Due to allowing one merge at each side, the cells above and below the selected subblock need to be analyzed and it has to be determined which of the neighbouring cells are most suitable to merge. An earlier version allowed the cell that was the closest to be analyzed first for a possibility of overlapping rails. If it had a common net, the corresponding rails would be merged. But the final version merges the cells that would decrease the cost the most. This was done in two steps. Of all the cells below and above the cell, the cell which would reduce the cost the most was merged. Then, available cells were recalculated and the cell with the remaining highest

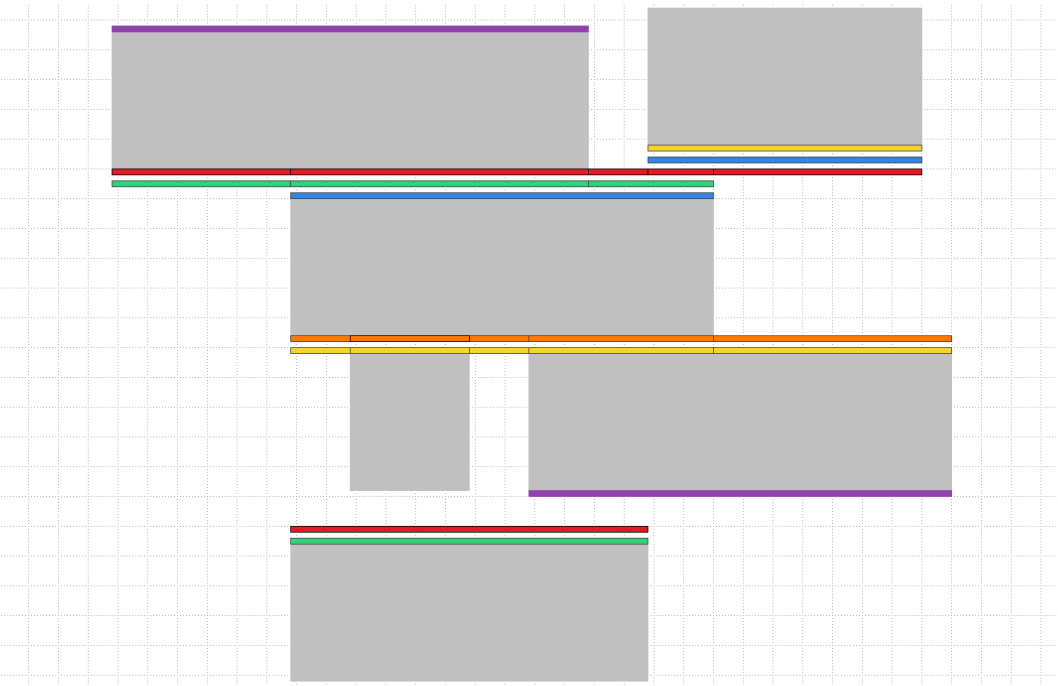


Figure 4.22: Merge example in an extreme case, showing off the correct way to merge all the cells in the vicinity of the selected cell in the center. The cell at the bottom lowers the parasitic resistance the most, but is not merged due to the final combination amounting to lower parasitic resistance.

cost reduction was merged. The merge algorithm does not care about the relevant placement of the rails, it just compares all the rails of two neighbouring cells.

This is not the most optimal way, as the optimal strategy would be to analyze the combination of all the neighbouring cells. But this does not matter, as the merge is part of the annealing process and it is the task of simulated annealing to make sure the cells most optimal for overlap are placed accordingly, such that the merging action can do optimal merges without being truly optimised.

The main algorithm for this will be explained soon. This method does the detection and validation of the selected rails and cells, the actual merging however is not explained as it is trivial. It comes down to moving the cells on top of each other and performing an evacuation check. This is necessary because the best cell for the merge can be anywhere above and below the selected cell. The selected cell will never move, so there is definitely potential of collision. Than the relevant parameters of the `Rail` objects are updated correctly.

The first steps involve detecting the cells above and below the cell by invoking the method `cellsInBbox`, then the cells with different bulk biasing and the cells that are already merged with the selected cell are omitted. That is what the method `findMergableNeighbours` does. Then the common rails are found between the available cells and the selected cell. This allows the construction of a list of candidates where each candidates is a list of the rails that can be merged between a cell and the selected cell. An example of the whole process is shown in figure 4.23. In this example we are considering cell 4. The list of cells available for merging is displayed. Cell 1 is excluded from this list as no rails are common with the selected cell. Cell 5 is excluded because it has a different bulk biasing and it will need to be moved down.

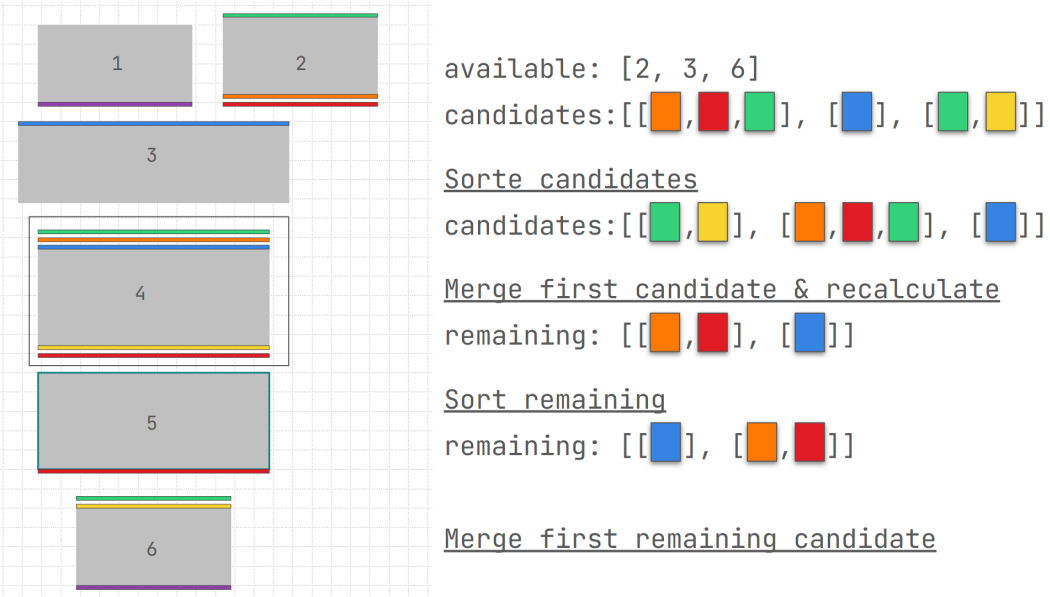


Figure 4.23: Demonstration of merge actions

The candidate list is sorted based on how much metal is overlapping with the selected cell. The first element of the sorted candidates are merged, which means cell 6 is merged onto cell 4, evacuating cell 5 in the process. This means the yellow and red net are not available anymore for overlap, therefore the remaining candidates have to be recalculated and resorted based on which side of the cell is not occupied. What follows is the merging of the first remaining candidate.

This method seems quite easy, but things become more complicated when other cells already have merged rails. Therefore it should be determined whether the first candidate is allowed to be merged and whether other options are available. This is done in three steps. These steps require knowledge about the current length of the merged rails, this can be calculated via the method `overlapLength`. Besides that, only the current cell is analyzed for any pre-existing merged rails. If the rails of the first candidate are merged with the selected cell, these rails are unmerged, regardless of the new merge being is less beneficial. This does not matter as the simulated annealing takes care of determining what is optimal.

Case 1 The selected cell is not merged with other rails. The rails of the first candidate can be merged onto the selected cell without problems and case 3 is the next step. If not go to case 2.

Case 2 When the selected cell is already merged with another cell, the cost reduction of the current merged rails is calculated. If this is lower than the new reduction in cost, the selected cell is unmerged and remerged with the first candidate. Either way, case 3 is considered.

Case 3 At this point the current cell is already merged on at least one side. If the other side is already merged, the algorithm ends. If not, the candidates are reanalyzed for the yet unmerged rails of the selected cell. The best candidate is then merged.

These three cases ensure that both sides of the cell allow overlap regardless of present merged rails. The main algorithm managing all this is explained in 16.

Algorithm 16 Overlap cells

```
function GRID.MERGE(selectedCell)
    available ← Grid.findMergableNeighbours(selectedCell)
    candidates ← []
    for all cell in available do
        common ← selectedCell.commonRails(cell)
        candidates.append(common)
    end for
    if length(candidates) = 0 then
        return False
    end if
    candidates.sort()        ▷ sort the candidates based on the length of overlap with the selected cell
    _, newMergeLength ← overlapLength(candidates[0])
    if currentMergeLength = 0 then                                              ▷ Step 1
        Grid.mergeCells(selectedCell, candidates[0])
    else if currentMergeLength < newMergeLength then                            ▷ Step 2
        Grid.unmerge(selectedCell)
        Grid.mergeCells(selectedCell, candidates[0])
    end if
    candidates.remove(candidates[0])
    Step3: find the unmerged rails and the free side of the selected cell
    for all candidate in candidates do
        if (candidate all not merged and on freeSide) then
            remainingCandidates.append(candidate)
        end if
    end for
    if length(remainingCandidates) = 0 then
        return False
    end if
    remainingCandidates.sort()
    Grid.mergeCells(selectedCell, remainingCandidates[0])
    return True
end function
```

4.3.6 Minor Actions

Two actions were added related to the routing algorithm, as this might improve the cost. One of these actions changes the order in which the nets are being routing. It could be for example that routing net2 before net1 uses different paths, which leads to shorter metal traces.

This action is optional as the order of the routed nets could be fixed by the designer. Thereby assigning a certain hierarchy of importance to the nets. Besides that nets could also be given an additional weight, which makes that net more important. Meaning that a change of the interconnects of the net has a larger influence on the cost. The final action could increase or decrease the amount of metal layers that were used by the router. This way the routing complexity could be gradually increased or decreased and in very simple cases the routing could also use a single metal layer without using the digital routing convention.

4.4 Overcoming Limitations of the routing algorithm

The proposed routing algorithm overcomes two limitations of the A* algorithm. It is able to connect more than 2 points, and it supports a 3D grid. First, the modifications with respect to the A* algorithm are described, after which the complete solution is showcased.

4.4.1 Point on rail

The first handled limitation is allowing the router to connect rails instead of single points. The actual algorithm does still only connect two points, but before the routing happens two points on each rail are chosen. This depends on the relative position of the two cells and the specific rails. A naive way of handling this, is choosing the points on each rail that are closest together. This would work fine if the primitives would not serve as obstacles. The implemented logic partially takes into account the present obstacles. Again, it is not the task of the algorithm to always behave optimally, it is the task of the annealing to place the cells that should be connected together close to each other. Only in this case the path should be optimal. Therefore, only the cells containing the specific rails are taken into account when analyzing obstacles in between rails.

There are many different cases. Some cases deliver specific points on each rail and other result in returning the points on the outside of both rails which are closest together. The latter is shown in figure 4.24. This example shows the simplest case: the two rails do not overlap on the x-axis. The case where rails do overlap will be explained next, but this example already shows the case where the two rails are not obstructed by a cell. In this scenario, the algorithm selects two random points on each rail with a common x coordinate. Since these points all yield the same path length, any pair can be chosen.

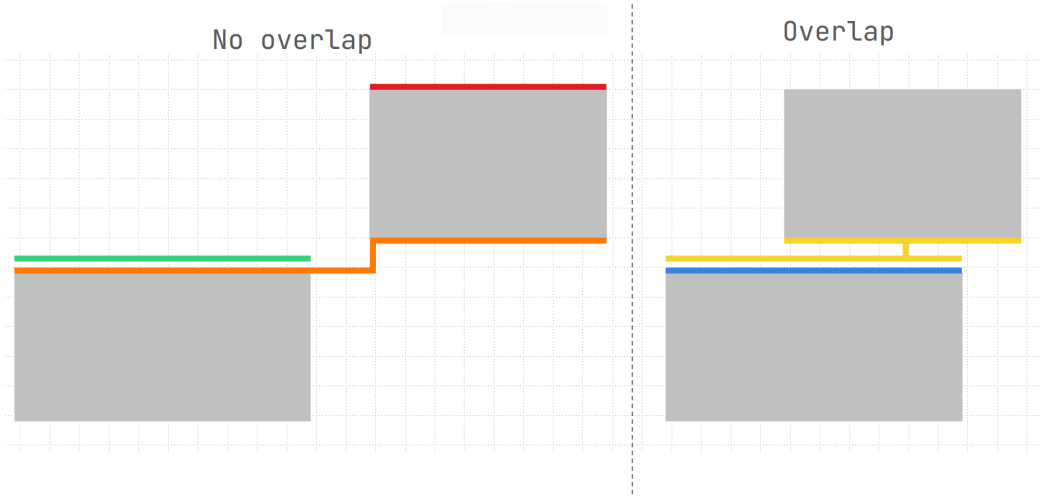


Figure 4.24: Two cases for selecting the correct point on a rail, the first example on the left has no overlapping x coordinates

The remaining cases are all variants of the type where both rails are above or below the cell. The only ones that do not fit this type is the opposite of the second example in figure 4.24. In that example, the two rails are not obstructed by the cells. The opposite case is when both rails are on the outside and are obstructed by both cells. In this case and the no overlap case, the method `findClosest` is used to determine the two points that should be routed.

This method determines the outer point on each rail that result in the shortest path. It accomplishes this by computing the cost of the four combinations of the outer points of the rails, via the heuristic. It returns the two points corresponding to the lowest cost.

The remaining cases have both rails on the same side of the cells. It is assumed from now on that cell 1 is above cell 2. Two of the remaining six cases use the `findClosest` method. These two cases can be seen in figure 4.25. These examples can be recognised, due to the cells fully overlapping on the x axis and the rail in the center is connected to the cell with the smallest length. Hence, the traces wrap around the larger cell.

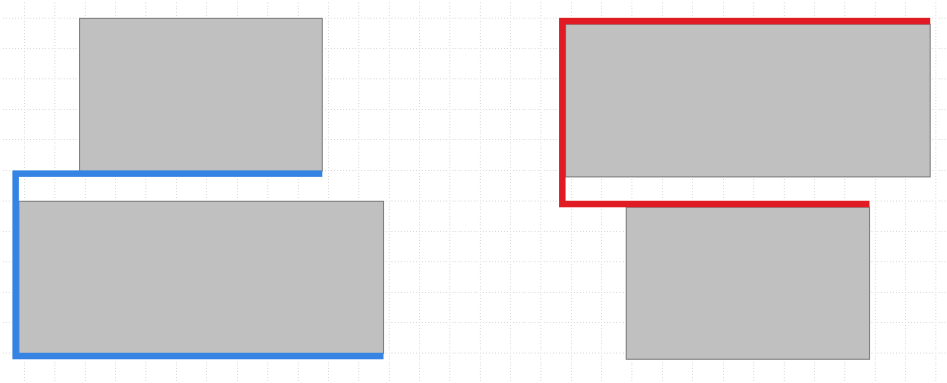


Figure 4.25: Two cases where the two points closest together are chosen, the rail in the middle is part of the smaller length cell and this cell is completely overlaps on the x axis.

The remaining cases allow direct calculation of the optimal points, because the path follows the side of one of the cells. This means the path goes from one of the outer points to the left of right and then up/down to the other rail. Two of the four cases are displayed in figure 4.26. The other two cases follow the same principle but now the cells are on the bottom. All these cases are handled in algorithm 17.

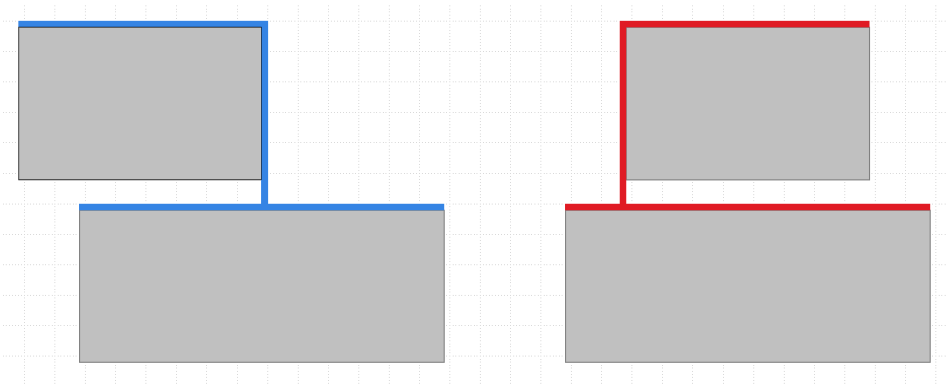


Figure 4.26: Two cases where the routing can go next to the cell with the smaller length.

Algorithm 17 Find points on two rails which are optimal to connect. rail1.y<rail2.y

```
function ROUTER.FINDPOINTS(rail1, rail2)
    overlapX, overlapL ← overlapLength(rail1, rail2)
    if overlapL = 0 then
        return findClosest(rail1, rail2)
    end if
    if ¬rail1.top and rail2.top then
        x ← random(overlapX, overlapX + L
        return (x, rail1.y, rail1.z), (x, rail2.y, rail2.z)
    end if
    if rail1.top and ¬rail2.top then
        return findClosest(rail1, rail2)
    end if
    if cell1.top then                                   ▷ only top, bot combinations left
        if cell1.x < cell2.x then
            if cell1.x + cell1.l > cell2.x + cell2.l then                    ▷ full overlap
                return findClosest(rail1, rail2)
            else
                return (rail1.x + rail1.l, rail1.y, rail1.z), (rail1.x + rail1.l + 1, rail2.y, rail2.z)
            end if
        else
            return (rail1.x, rail1.y, rail1.z), (rail1.x − 1, rail2.y, rail2.z)
        end if
    else
        if cell1.x > cell2.x then
            if cell1.x + cell1.l < cell2.x + cell2.l then                    ▷ full overlap
                return findClosest(rail1, rail2)
            else
                return (rail2.x + rail2.l + 1, rail1.y, rail1.z), (rail2.x + rail2.l, rail2.y, rail2.z)
            end if
        else
            return (rail2.x − 1, rail1.y, rail1.z), (rail2.x, rail2.y, rail2.z)
        end if
    end if
end function
function FINDCLOSEST(rail1, rail2)
    path1 ← heuristic((rail1.x, rail1.y, rail1.z), (rail2.x, rail2.y, rail2.z))
    path2 ← heuristic((rail1.x, rail1.y, rail1.z), (rail2.x + rail2.l − 1, rail2.y, rail2.z))
    path3 ← heuristic((rail1.x + rail1.l − 1, rail1.y, rail1.z), (rail2.x, rail2.y, rail2.z))
    path4 ← heuristic((rail1.x + rail1.l − 1, rail1.y, rail1.z), (rail2.x + rail2.l − 1, rail2.y, rail2.z))
    if path1 = min(path1, path2, path3, path4) then
        return (rail1.x, rail1.y, rail1.z), (rail2.x, rail2.y, rail2.z)
    else if path2 = min(path2, path3, path4) then
        return (rail1.x, rail1.y, rail1.z), (rail2.x + rail2.l − 1, rail2.y, rail2.z)
    else if path3 = min(path3, path4) then
        return (rail1.x + rail1.l − 1, rail1.y, rail1.z), (rail2.x, rail2.y, rail2.z)
    else
        return (rail1.x + rail.l − 1, rail1.y, rail1.z), (rail2.x + rail2.l − 1, rail2.y, rail2.z)
    end if
end function
```

4.4.2 Heuristic and pathing

A-star utilised the Manhattan distance as a heuristic, so the best path at this point is also the shortest path. But for routing an IC the parasitic resistance should be the figure of merit. The heuristic plays a big part in the performance of the router. A good heuristic allows fast exploration and precise, optimal path estimation.

The heuristic requires knowledge of the sheet resistance of the employed metal layers and the resistance of the vias in between them. The total estimated resistance is then calculated, by looking at which layers will be employed when using the best path. The main heuristic multiplies the difference in x coordinate by the sheet resistance of the lowest layer for horizontal movement. The same principle is used for the difference in y coordinate, but the cost of two vias between the lowest two layers is added. Because vertical movement implies the minimal use of two vias. Then, the difference in metal layers is determined and based of the highest metal layer, the cost for the vias are calculated. This is summarised in the following formula.

$$h(\Delta x, \Delta y, z1, z2) = abs(\Delta x) * R_{M_0} + abs(\Delta y) * R_{M_1} + sgn(abs(\Delta y)) * R_{V_0} + viaCost(z1, z2)$$

This function should execute fast, as this method has a very large amount of calls during each routing step. The method `viaCost` is described in algorithm 18

Algorithm 18 Calculate via cost

```
RVia ← [...]                          ▷ List containing via resistance, first elements refers to first via
function viaCost(z1, z2)
    zMax ← max(z1, z2)
    zMin ← min(z1, z2)
    totRes ← 0
    for i = zMin to zMax do
        totRes ← totRes + RVia[i]
    end for
    return totRes
end function
```

For simplicity the amount of metal layers used for the examples is four and the first four metal layers have sheet resistance 1, where each via has a resistance of 6.

As mentioned in the previous chapter, the heuristic is used to determine which cell is explored next. As it is used to estimate the current cost from each node to the goal. The other metric, the gScore, is the accurate cost of the current path: from the starting node to the current node. This is calculated by using the exact distance from node to node, so a step in a metal layer adds a cost of 1 and going up or down a layer adds a cost of 6. These values are arbitrary and employed for the results.

4.4.3 Greedy routing

When routing the layout of an IC, the routing problem comes down to a variant of the traveling salesman problem. This means finding an optimal solution is computationally expensive. Simulated annealing is a method that can be used to solve these kind of routing optimization problems. Then, the actions are focussed on deciding which path to take between two points. This layout generator utilizes the simulated annealing algorithm for the placement specific, without it influencing the router much.

When a net only interconnects two rails, it suffices to find one point on each rail and use this as the input for the improved A* algorithm to find the path. However, once a net connects more than two rails, the problem gets much harder. To solve it a greedy algorithm is proposed. This approach estimates the optimal routing well, while keeping the function calls to the routing algorithm at a minimum. The greedy algorithm connects all the rails sequentially onto the current interconnects, based on which rail has the current shortest distance to the existing traces. This current shortest distance is estimated by the heuristic of the A* algorithm. Again, this will not always provide the optimal result, but it is the task of the placer algorithm to make sure the cells are close together in which case this algorithm will perform correctly.

Figure 4.27 demonstrates how the blue net is being routed. The first step is to find the two rails which selected points are closest together, according to the heuristic. This is the red net. If the actual cost of the path is smaller than the second best heuristic, this path is accepted. For this example, this is the case. The heuristic estimation is the same as the actual implementation, for the heuristic calculates the cost of the optimal path.

After the first path is chosen, the process is repeated. The optimal connection between the current path and the remaining rails is found. This is done by finding the closest point between the rail and the currently routed net, by comparing each combination of points. Then the shortest path is determined and if the cost of the path again is lower than the second best heuristic, it is routed. This step is being repeated until every rail is connected.

If the actual path happens to be worse then the second best estimation of the heuristic, the next path is routed. If this implementation is better than the previous one, it is implemented. If not, it is compared to the third best estimation. If this estimation is better, that path is calculated. This process keeps on repeating and stops if the actual routing is cheaper than the next estimation; or a previous routing is cheaper than the current one; or until all rails are compared. The final algorithm executing this, is shown in 19, the method `findBestPath` is a helper function which can access the same properties as `greedy`.

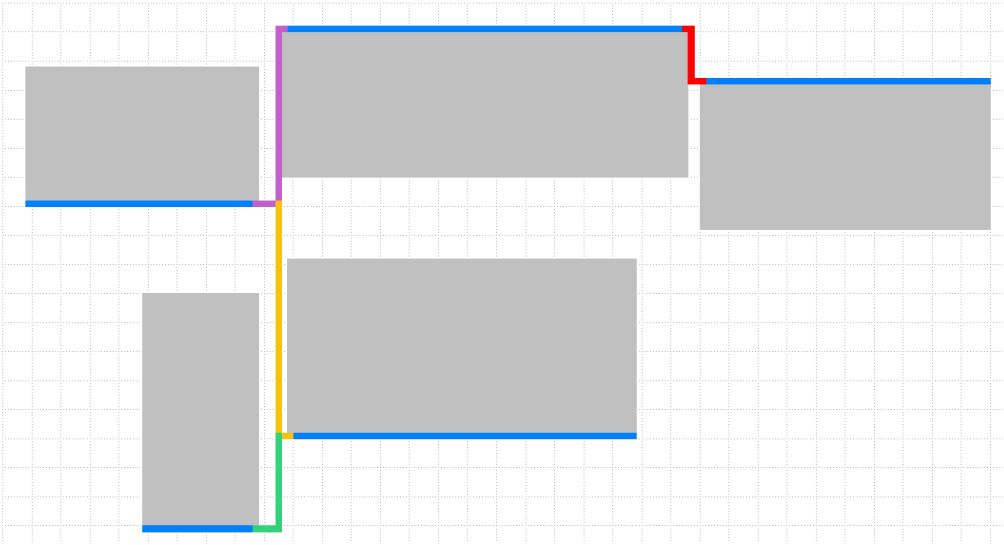


Figure 4.27: Example of routing multiple rails at once. The interconnects are added as follows: red, purple, yellow, green.

Algorithm 19 Greedy algorithm for routing multiple rails

```
function GREEDY(routeGrid, rails, netValue)
    pathCost ← Dictionary[(int, int), cost]    ▷ tuple contains indices of two rails, cost is a float
    for i = 0 : length(rails) − 1 do
        for j = i : length(rails) do
            p0, p1 = pointsOnRails(rails[i], rails[j])
            pathCost[(i, j)] =heuristic(p0.x − p1.x, p0.y − p1.y, 0, 0)
        end for
    end for
    findBestPath(pathCost)
    while rails.notAllMarkedDone do
        pathCost ← Dictionary[(int, int)] = cost
        for i = 0 : length(rails) − 1 do
            for j = i : length(rails) do
                p0, p1 = closestPoints(rails[i], rails[j])
                pathCost[(i, j)] =heuristic(p0.x − p1.x, p0.y − p1.y, 0, 0)
            end for
        end for
        findBestPath(pathCost)
    end while
end function
function findBestPath(pathCost)
    pathCost.sortByValue()
    curLowest ← inf
    curLowestIndices ← (0, 0)
    for all pair, optCost in pathCost do
        nextOptCost                                    ▷ obtain next cost in pathCost too
        bestPair ← pathCost.keys[0]
        path, cost ← findPath(pointsOnRails(rails[bestPair[0]], rails[bestPair[1]]), net)
        if cost > curLowest then
            rail1 = rails[curLowestIndices[0]]
            rail2 = rails[curLowestIndices[1]]
            path, cost ← findPath(pointsOnRails(rail1, rail2, net)
            routeGrid.addPath(path)
            rails.markDone(pair)
        else if cost < nextOptCost then
            routeGrid.addPath(path)
            rails.markDone(pair)
        else if cost < curLowest then
            curLowest ← cost
            curLowestIndices ← pair
        end if
    end for
end function
```

4.4.4 Valid move

Another responsibility of the router is to avoid shorting multiple nets and keeping 1 unit distance between distinct traces. This adds an extra layer of complexity to the router. Each time a neighbour is explored, it needs to be validated. This means that neighbours of the explored node cannot be next to another trace, as one of the constraints will be violated. Consequently, each explored neighbour, requires checking 8 neighbouring cells. Performance wise this makes the router slower, but it is required. The algorithm for `validMove` is shown in 20, this method can access the same parameters as the A* method.

Algorithm 20 Valid move

```
allowed ← set((int, int, int))
function validMove(node)
    for i = −1 to 1 do
        for j = −1 to 1 do
            if ¬(i = j) then
                col ← node.x + i
                row ← node.y + j
                if (node.z, row, col) in allowed then:
                    continue
                end if
                if routeGrid[0, node.y, node.x] = 1 then
                    return False
                end if
                if ¬(routeGrid[node.z, node.y, node.x] = 0 or netValue) then
                    return False
                end if
                if ¬(0 <= node.z < grid.layers and 0 <= row < grid.rows) and 0 <=
col < grid.cols) then
                    return False
                end if
            end if
            allowed.add((node.z, row, col))
        end for
    end for
    return True
end function
```

This algorithm has intensively been optimised. A number of checks are required which are also slow, but they are executed in a specific order and also allow premature exiting of the method, such that not all 9 cells have to be verified if one requirement is not met. The order is determined based on which check is violated the most. This way unnecessary statements are avoided. The first condition is a member check of a set storing the already verified nodes. The second condition checks if the node is above a primitive or not. The next condition verifies whether the current cell is free space or the same net. The last one ensures the index does not exceed the bounds of the routing grid.

4.5 Improved A*

The final A* routing algorithm is shown in 21.

Algorithm 21 Improved A* algorithm

```
function AstarRevisited(grid, start, goal, netValue)
    moves ← [[(1,0,0),(1,0,0),(0,0,1),(0,0,−1)],[(0,1,0),(0,−1,0),(0,0,1),(0,0,−1)]]
    if grid.layers = 1 then
        moves ← [[(0,1,0),(0,−1,0),(1,0,0),(−1,0,0)],[(0,1,0),(0,−1,0),(1,0,0),(−1,0,0)]]
    end if
    closedSet = set()
    cameFrom = Dict[(int,int,int)] = (int,int,int)
    gScore = Dictionary[start] : 0                          ▷ Default value of items is infinity
    fScore = Dictionary[start] : heuristic(start, end)
    openList = [(fScore[start], start)]
    while openList not empty do
        current = openList.pop()[1]
        if current in closedSet then
            continue
        end if
        if current = end then
            return reconstructPath(current)
        end if
        for dz, dy, dx in moves[current.z  mod 2] do
            neighbour = (current.x + dx, current.y + dy, current.z + dz)
            if neighbour not in closedSet then
                if validMove(neighbour) then
                    tentativeGScore = gScore[current] + distance(current[0], dz)
                    if tentativeGScore < gScore[neighbour] then
                        cameFrom[neighbour] = current
                        gScore[neighbour] = tentativeGScore
                        fScore[neighbour] = tentativeGScore + heuristic(neighbour, end)
                        openList.push((fScore[neighbour], neighbour))
                    end if
                end if
            end if
        end for
    end while
    return []
end function
```

To allow 3D exploration and to follow the routing convention, a list of moves is constructed. Moves contains two sets of moves, the first one is for movement on even layers(x-axis), the second one for movement on odd layers (y-axis). Most of the algorithm remains the same, but with optimised data structures and conditions. The algorithm itself uses a dictionary for the g and f scores. Python dictionaries (and sets) provide member checks on average in $O(1)$, worst case it is performed in $O(n)$, which is the average for member checks in Python lists. Besides that, the default value for the items in these dictionaries is infinity.

Similar to `validMove`, the if-statements are precisely ordered, mainly to execute `validMove` and `heuristic` as late as possible, for they are more expensive methods. The algorithm itself does not differ that much from the original one, but it is assisted by all the other methods described in this section, to overcome all the limitations of standard A*.

4.6 Performance and optimization

The A* algorithm is the most executed part of the layout generator. Each iteration of simulated annealing requires many calls to the router, therefore a lot of time went into optimizing the algorithm. A larger schematic contains more rails and more nets, thus the amount of time the algorithm is executed increases too. But the execution time itself also goes up for larger grids, as it takes longer to reach further cells.

What would increase the layout generator performance the most is always doing actions that lower the cost, this way the amount of iterations is low. If the actions do not lower the cost, the calls to the router were 'wasted' which can be costly. This can only be optimised by choosing the correct weights for the actions.

To make `validMove` and `A*` perform faster Cython [2] was explored, for the A* algorithm in a 2D grid, this improved performance, but for the last version it slowed the algorithm down slightly. Cython is meant to be used when large arrays are iterated over to perform calculations, as the data structures for the routing algorithm are mostly dictionaries which are constantly updated, it is not able to speed up the algorithm. The best option to do so would be to use a different programming languages which perform faster, to implement the improved A* method.

When looking at the memory performance of the layout generator, it is obvious that it depends on the size of the circuit. What is always stored is two objects of the router class, containing the information of all the cells for a specific configuration, based on the state of simulated annealing. These objects are always there and do not depend much on the size. However, the router requires the initialization of a grid, to be able to perform the router. This grid has to be reinitialised for each router, but previous grids are not stored. This grid does not scale optimally as it depends on the grid size which depends on the size of all the cells.
The A* algorithm minimises memory usage by employing dictionaries and sets to store relevant data.

Overall the memory occupation is not the performance concern of this layout generator, the main issue is the non-ideal scaling of execution time for larger circuits.

The only step that remains is exporting the python representation of the layout to Virtuoso. This would mean lifting the abstraction to make a realistic and functional IC layout. The primitives are already drawn after they are detected in order to perform the quantization and to know the dimensions and other specifications of them. These are all passed to a `Router` instance to generate a layout. When a result is obtained, the information about the placement and routing is passed back to the script which redraws the primitives. The previously drawn subblocks cannot be reused because the rails swap places. Utilizing all these methods, a fully operational layout is obtained in Virtuoso and the performance of the layout and the layout generator can be measured.

5 USAGE

With the many degrees of freedom and the huge variety in available circuits, the results of the layout generator differ a lot. Different circuits and setups will be explored to demonstrate the various outcomes. The reference circuit is the operational amplifier circuit. It is simple, which allow fast performance of the routing and it has many different good configurations. The realistic circuit includes using the strict rails for the differential pair and taking into account the bulk of the primitive. However, for testing the performance of the layout generator, these constraints will not be enforced. This allows more creativity and allows better gauging of the layout generator. This simplified configuration can be seen in figure 5.1, the showcased layouts will be annotated with the cost, the area and length are unit-less, as it is expressed by the pitch, which is 1.

What follows, is an in depth discussion of the influence of the parameters on the layout generator and how the best performance can be ensured. Finally, the performance of the circuit will be analyzed using a realistic layout.

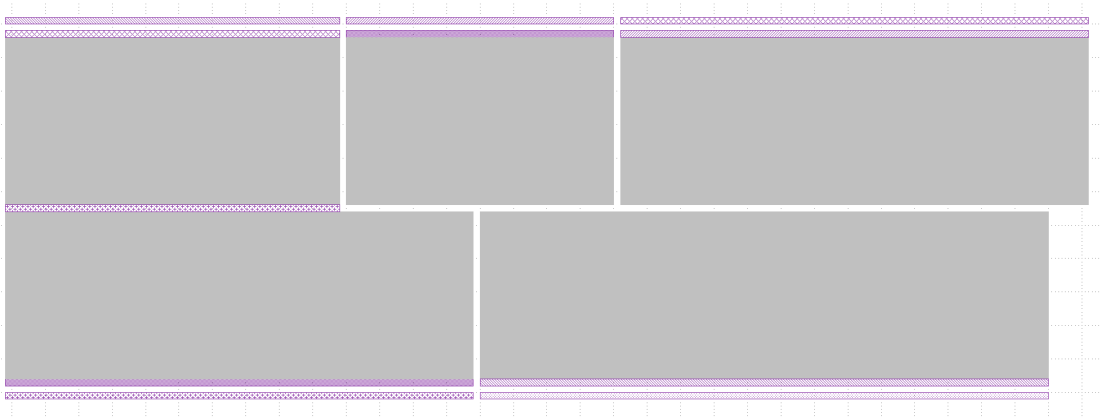


Figure 5.1: Subblocks of simplified opamp, showing optimal area configuration. Bulk potential and strict rail configuration of differential pair are not taken into account. Area: 8910

5.1 Calibration and determining parameters

5.1.1 Actions

While all the degrees of freedom provide much control over the generator, it can be difficult to find the correct combination of values to ensure good performance. Therefore, providing fitting default values is important. The default values of some parameters do not depend on the specific circuit. This the case for the weights of the actions, the default values for these can be seen in table 5.1. These default values are determined by trial and error, based on their importance and influence on the cost. In theory, simulated annealing should always reach

a decent layout independent of these weights, due to the random nature. However, finding the best actions will take longer depending for the exact values.

The recommended method to find good values for the weights is by doing manual adjustments based on previous runs. Each run will provide data for these actions which can be used as feedback. In general for each iteration of the simulated annealing, the current cost and temperature is stored. Besides that each executed action also provides the following information.

- Type of action
- current iteration (allows finding the current cost and temperature)
- cost lowered, cost increased and tolerated or cost increased and not tolerated

This allows to find when each action was executed and how influential each actions was. By analyzing the frequency of lowering the cost, it can be analyzed if the action is for example more useful for higher annealing temperatures. Based on that the designer should tweak the weights of the actions.

Action	weight
Move	100
Move and Refit	40
Swap rails	50
Swap cells	40
Merge	20
Swap routing order	5
Update layers	10

Table 5.1: Default weights for the actions of the placement algorithm

In contrary to the weights, finding the maximum amount of actions per iteration relies on the specific circuit. Therefore, it requires looking at multiple iterations of the algorithm. If the results are always similar or if only doing more actions at a time will improve the cost, this value should be incremented. The default value should be 2 or 3, depending on the circuit complexity as this influences how fast the layout generator converges. For a more complex circuit, a higher value is better.

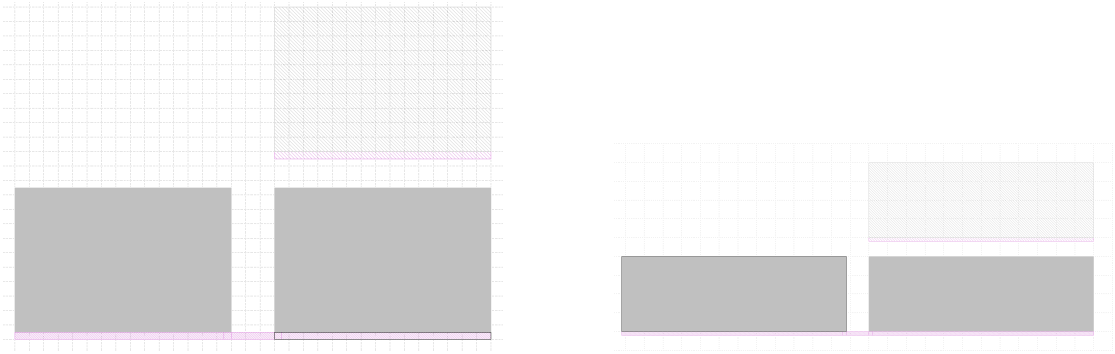
The amount of predefined constraints on the layout matter. If the solution space is small, it can be beneficial it to have a higher number of actions. To give an example, for the simpler opamp the maximum amount of actions per placement is 2 and for the actual opamp layout this is 3, due to the extra constraints. It should be noted that a higher number of actions result in larger execution time.

5.1.2 Calibration

To determine the weights for the cost function a calibration process is available. Firstly, this is necessary to allow the simulated annealing to always behave equally for the same parameters. To understand this, the Boltzmann distribution which allows uphill movement should be analyzed.

$$P(\Delta Cost) = exp\left(-\frac{\Delta Cost}{T_i}\right)$$

Depending on the change of the cost and the current temperature uphill movement will be accepted, i.e., when the temperature is high and the change in cost is low, a higher probability of acceptance is obtained. This implies that, in order to always have the same behavior, the difference in cost for the same action should be independent of the circuit. As the cost consists of the total trace resistance and the total area, the influence on these two values should be the same. The trace resistance, which is equivalent to the trace length, follows this principle. If an action brings a cell closer to the left, the length of the rail is reduced. This is independent of the circuit. However by moving the cell closer, the total area changes, which depends on the size of the layout. Figure 5.2 shows two layouts with two cells. The same action is performed for both cases. The difference in length is the same, but the difference in area is not. To solve this the area should somehow be normalised.

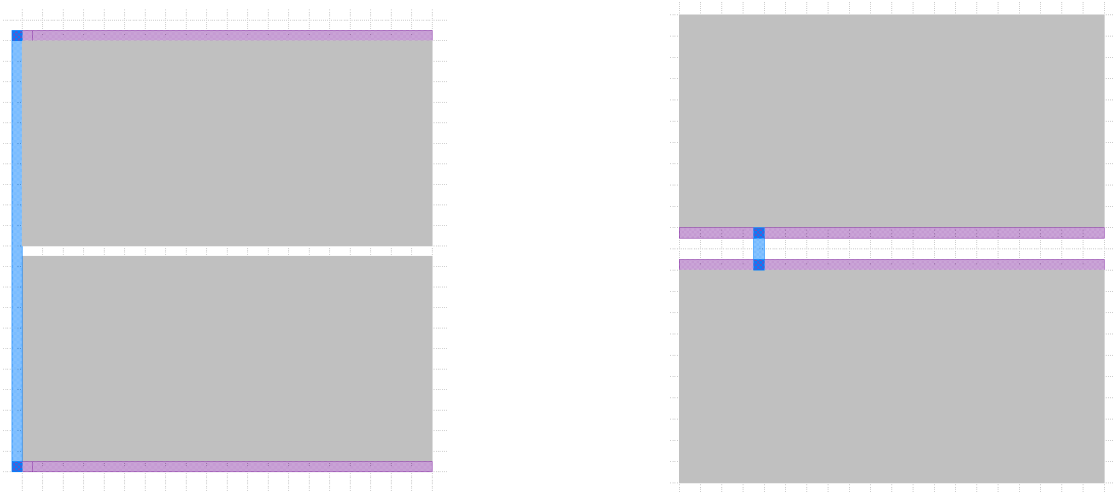


(a) Layout 1: ΔArea=2925, ΔLength=27

(b) Layout 2: ΔArea=1625 ΔLength=27

Figure 5.2: Two layouts underwent the same action: the cell on the right moved down to a new position adjacent the other cell, previous position is shown as a faded cell. The change in area and length is given.

The second reason to normalize the area is due to the cost not represent a correct figure of merit without rescaling, as the area will in most cases be much larger, compared to the length. Figure 5.3 shows two layouts; the sheet resistance of the traces is assumed 1. Both layouts almost have the same cost, but the configuration of layout 2 is clearly better. This is due to the area and length being unbalanced.



(a) Layout 1: 1846 = 1720 + 126

(b) Layout 2: 1845 = 1760 + 85

Figure 5.3: 2 possible layouts with similar cost (Area + Length)

This can be solved by correctly choosing weights for the area and length, but this would make using the weights very unintuitive. It would be much better if the cost and length are balanced when the weights are equal. To obtain this, the area needs to be normalised. The normalization factor is the ratio between the best possible length and the best possible area. This way the optimal area equals the optimal length. Applying this to the example in figure 5.3, the costs for the layouts would be 168 and 128 respectively, which reflects the change in length (39) and change in area (1).

However, this clashes with the first problem mentioned and the area is not being resized to the same value. But due to the rescaling the difference in cost is much smaller. Hence, the negative effect on the uphill movement stated earlier is much weaker.

Now to do this normalization the calibration is necessary. The calibration determines the 'best' area and 'best' length, to calculate the scaling factor. The first step is to find the optimal area. This can be found very quickly by performing the simulated annealing algorithm without routing the cells, which would slow down this process. The cost of the circuit only takes into account the area and the final area is this (almost) optimal area. For the simplified opamp this step can be seen in figure 5.1 and the optimal area is 8910.

After this the optimal length should be calculated. This takes a bit more time as the router should obviously be used. Now only the length or trace resistance should be considered for the cost. For simplicity and speed routing over the cells is allowed. To further speed up the process, the starting layout could be the layout obtained by step 1. The result for this can be seen in figure 6.2 and the length is 288. It should be noted that using the digital convention does increase the cost for the final results, due to the additional vias. However, the algorithm performs faster with it. This convention can be toggled by the designer.

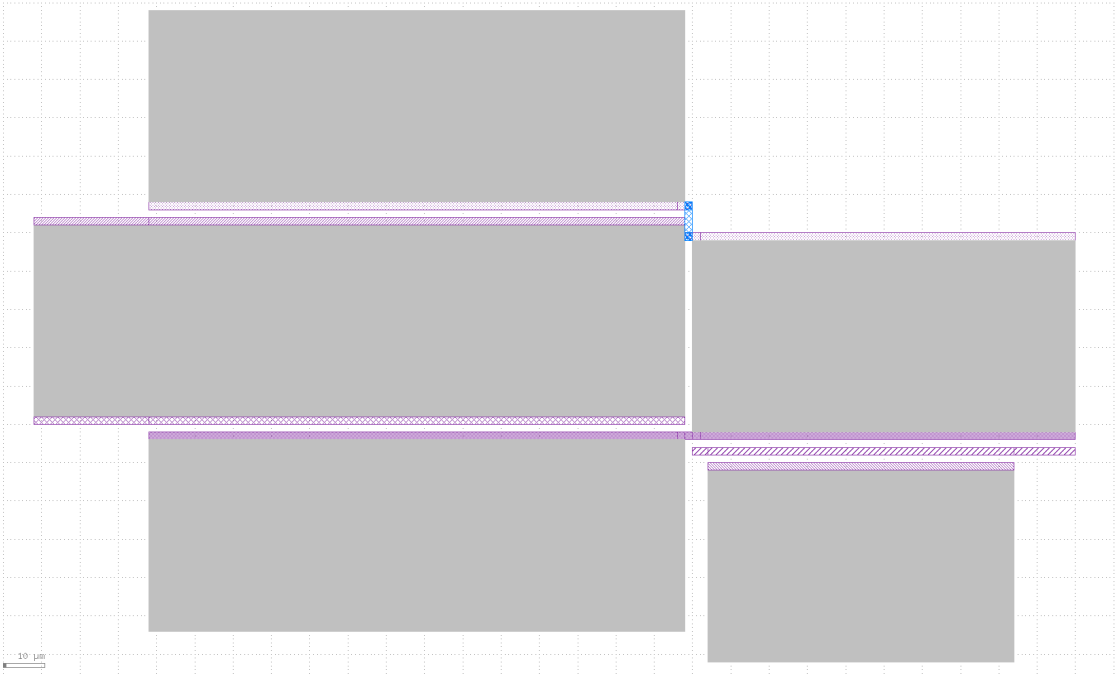


Figure 5.4: Simplified opamp with optimal length layout. Length: 288, Area: 11560

The final step just involves rescaling the area and the layout generator is calibrated. The weights now represent a correct scaling of the area and length. By default both weights are 1. Now an optimal layout for the simplified opamp can be generated. When using the default weights and the normalization calculated earlier, the following layout is obtained 5.5. The final cost is 622, which is close to twice the optimal length.

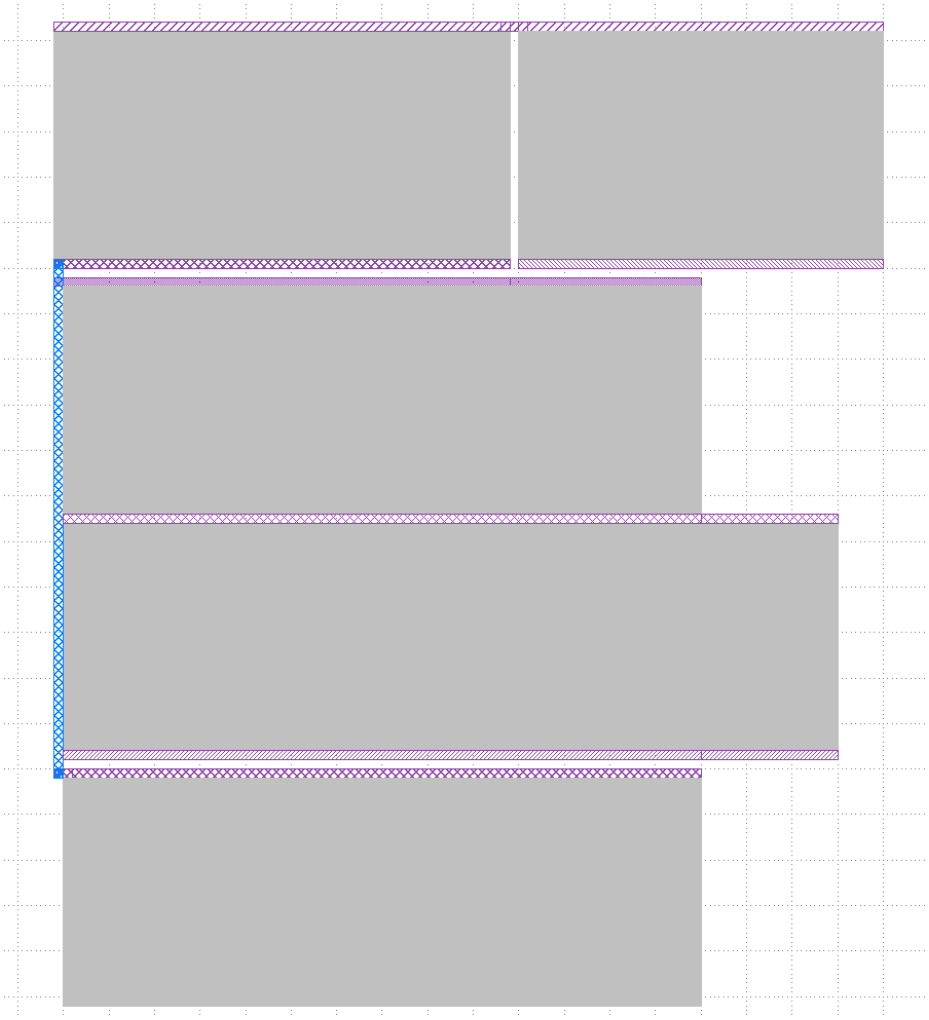


Figure 5.5: Simplified opamp with optimal layout. Cost: 622, Length: 305, Area: 9828

This calibration system is not perfect, as these values will almost never be actually optimal. The problem with the different annealing behavior also remains. But it provides the designer with proper default values for the cost function and greatly improves when comparing to the performance without the calibration.

5.1.3 Annealing parameters

The remaining parameters are the annealing parameter: T_{start}, T_{end} and α. T_{end} is the parameter that has the most influence on allowing uphill movement. As the denominator of the Boltzmann distribution is the current temperature. The idea is that, as T gets very close to T_{end}, less uphill movement is allowed, as only a slightly worse cost can be accepted. As T_{end} controls how low the temperature can get, it also controls how much uphill movement is allowed in the later stages.

Even more, when the ratio between T_{start} and T_{end} both is divided by 10 and the weights of the cost are multiplied by 10, the behavior of the layout generator will be the same as the Boltzmann distribution will not change.

If the layout generator does not seem to reach an optimal solution, incrementing T_{start} will allow more variation in the beginning with the same uphill movement in the end. Decreasing T_{end} on the other hand, enforces more precise moves at the end of the algorithm. The default values for these parameters are 100 and 0.01 respectively. For α, the default value is 0.95.

Changing how fast the algorithm reaches the end, means changing α or how far the start and end temperature are apart. When the solution is reached too fast, changing α is useful if the amount of uphill movement is sufficient. Else,it is best to change the temperature interval. There is no clear cut way to determine these values, but at least the default values will allow generating a decent layout.

Now the layout shown in Figure 5.5 uses these default values. However, the algorithm reaches a solution very fast as the circuit is quite simple and small. At a certain point the layout is just waiting for the temperature to be lowered but no optimal moves can be found as the layout is already quite good. This is visualised in figure 5.6, where 191 generated layouts were analyzed.

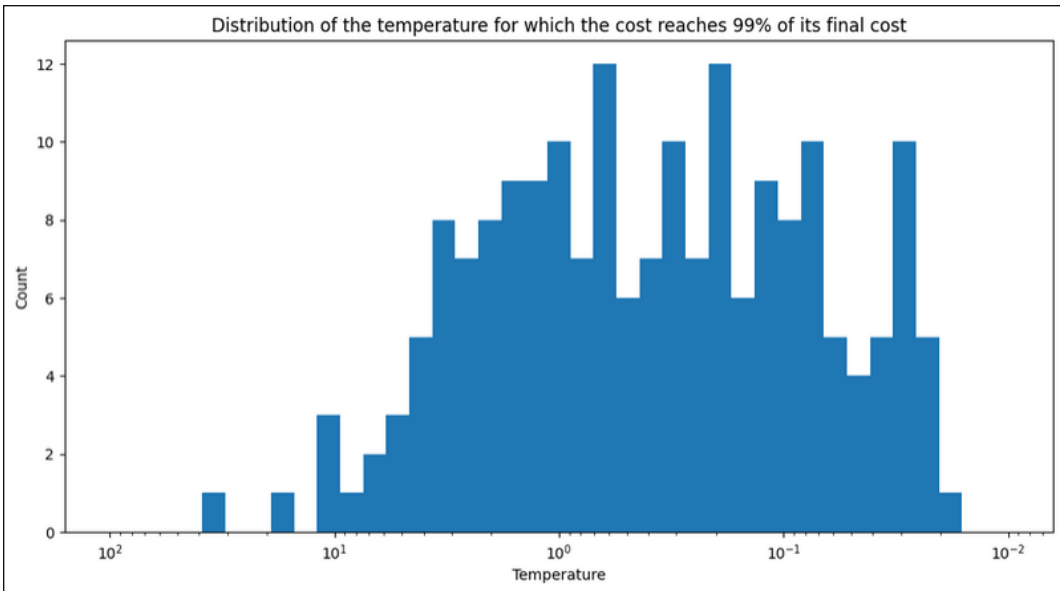


Figure 5.6

The majority of these layouts reach 99% of their final cost when the temperature is around halfway. On average the amount of iterations is 2579 when the cost reaches this point. The average amount of total iterations is 5900 and the average final cost is 669. This means the temperature interval is too large or α is too large. When looking at the progression of the cost of the 5 best and 5 worst layouts, it is clear that annealing converges fast. The cost evolution of both is shown in figures 5.7 and 5.8.

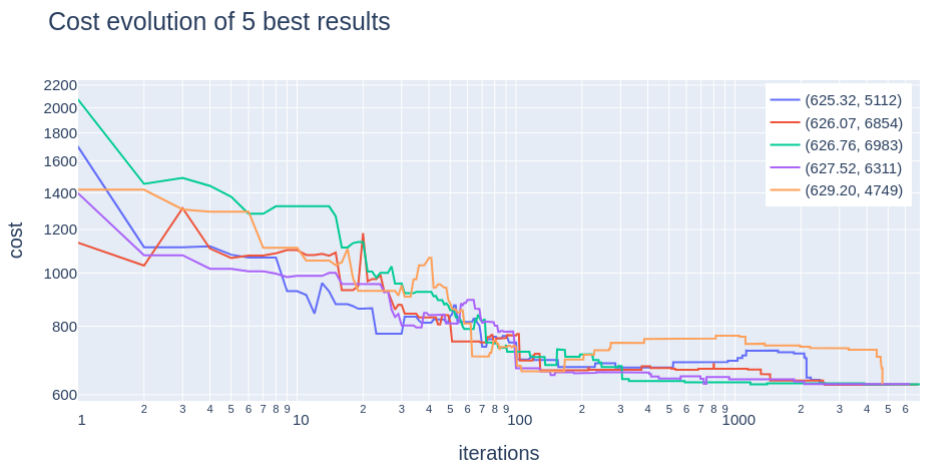


Figure 5.7: Cost evolution of five best results. Legend displays final cost and amount of iterations

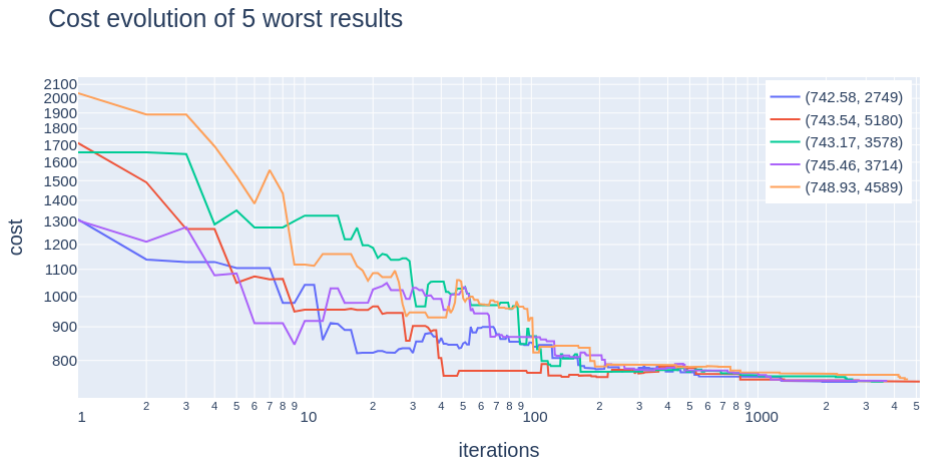


Figure 5.8: Cost evolution of five worst results. Legend displays finals cost and amount of iterations

It is clear that the way the solution is reached does not depend on the total amount of iterations or the starting position. The randomness of the algorithm is clearly shown as the convergence shows no clear pattern for good or bad results. These results also show no correlation when looking at the amount of uphill movement or when the cost reaches '99%' of the final cost. As stated above, the cost converges fast but it takes a long time to finish the algorithm. Next the influence of α will be explored.

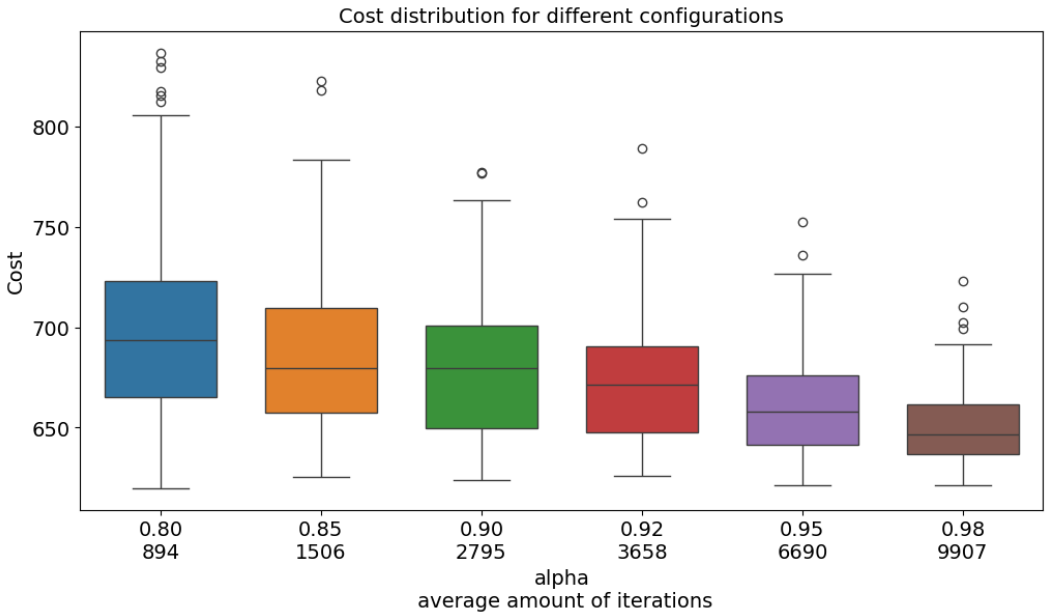


Figure 5.9: Cost distribution for different values of α

The same experiment was executed for the following values $\alpha \in [0.98, 0.95, 0.92, 0.90, 0.85, 0.80]$. Going higher or lower than these values means converging extremely slow or fast respectively, thus not providing feasible results. Figure 5.9 shows the distribution of the cost of each experiment.

It is clear that for most values the optimal solution (622) almost can be reached. The algorithm is more consistent if it runs longer as the largest two values provide on average very good results. The speed of convergence will be very similar for each experiment, but the algorithm will just finish sooner. The main disadvantage for using $\alpha = 0.98$ is that it takes on average 9907 iterations to finish. This value is not even a correct estimate as the maximum amount of iterations is 10000 and the annealing stops before reaching T_{end}. For $\alpha = 0.98$, the amount of improvements required to lower the order of magnitude of the temperature is very high, this is displayed in table 5.2.

α	x
0.8	10
0.85	14
0.90	22
0.92	28
0.95	45
0.98	114

Table 5.2: Improvements required to divide temperature by 10. This would mean finding x for $10 \times \alpha^x = 1$

Another observation is the similarity in layouts. The resulting costs seem discretised as they are grouped in adaptations of the same layouts. As the simplified opamp has not many building blocks the variation in cost is not that large.

Another almost optimal layout that was generated can be seen in figure 5.10. This design was generated multiple times by different configurations with various parameters. Another layout with almost similar cost was obtained even more and is shown in figure 5.11. These two layouts are quite alike, but the latter occurred much more. Only the frequency of the best layouts where counted. Because of the large number of simulations the results where often very close to one of the three optimal layouts.

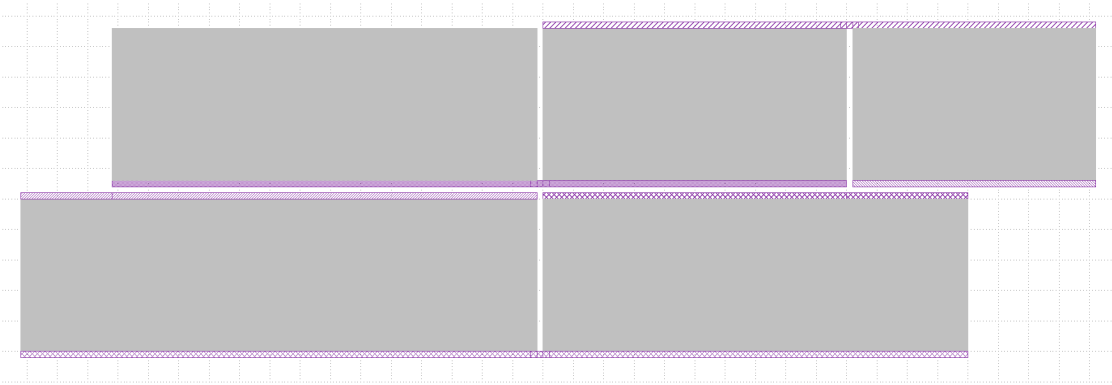


Figure 5.10: Optimal generated by the framework. Cost: 626, Length: 311, Area: 9735

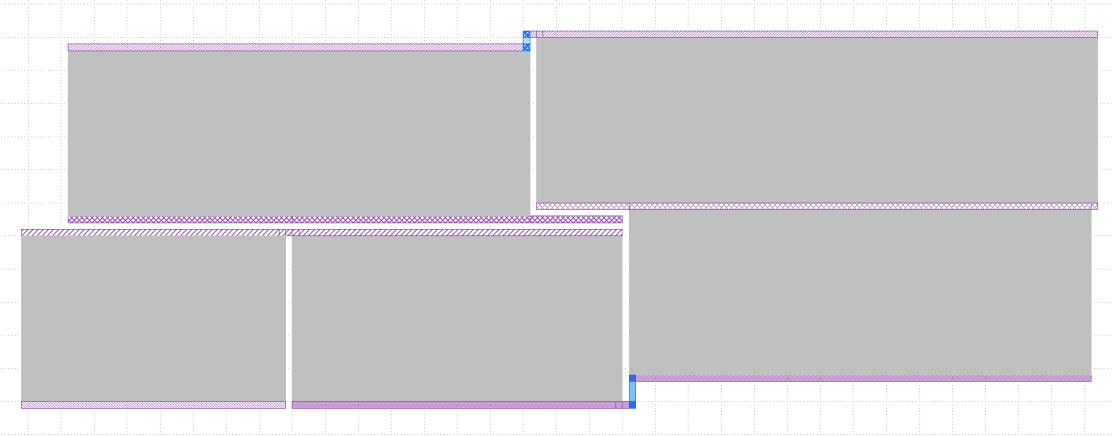


Figure 5.11: Optimal generated by the framework. Cost: 627, Length: 326, Area: 9291

To understand the influence of the annealing parameters the same experiment was executed for combinations of different temperature intervals and α. The temperature range was changed and some of the previous mentioned values for α where selected for that range, based on how much improvements were necessary. If $T \in [0.01, 1000]$ the corresponding values where low as the temperature interval is very spread out. The value of α was 0.95 or 0.98, if the temperature interval was close together.

The first observation is that almost each configuration reached at least one of the optimal configurations. Obviously by running a large amount of simulations, the chance increases of finding one. The main difference lies in the distribution of the cost.

The most obvious result is that for $\alpha = 0.98$ the layout generator performs the best. The average cost is the lowest and the difference between worse and best results is also the smallest. The downside is that these simulations take the most time: it takes 2 to 3 times longer to execute when compared to $\alpha = 0.95$. The reason this performs so optimally, is that the temperature stays similar a long time, which means that in the beginning a lot of uphill movement is allowed and at the end a lot of precise movement is required to finish the process. As this configuration requires a lot of improvements to finish, the temperature interval is quite small.

A large temperature interval with a lower alpha and the same T_{end} does obtains similar results, but with a higher upper bound. The temperature changes too fast for these configurations to have optimal uphill and precise movement. This is shown in figure 5.12. As the left configuration does require the most improvements (228), it performs the slowest and the best.

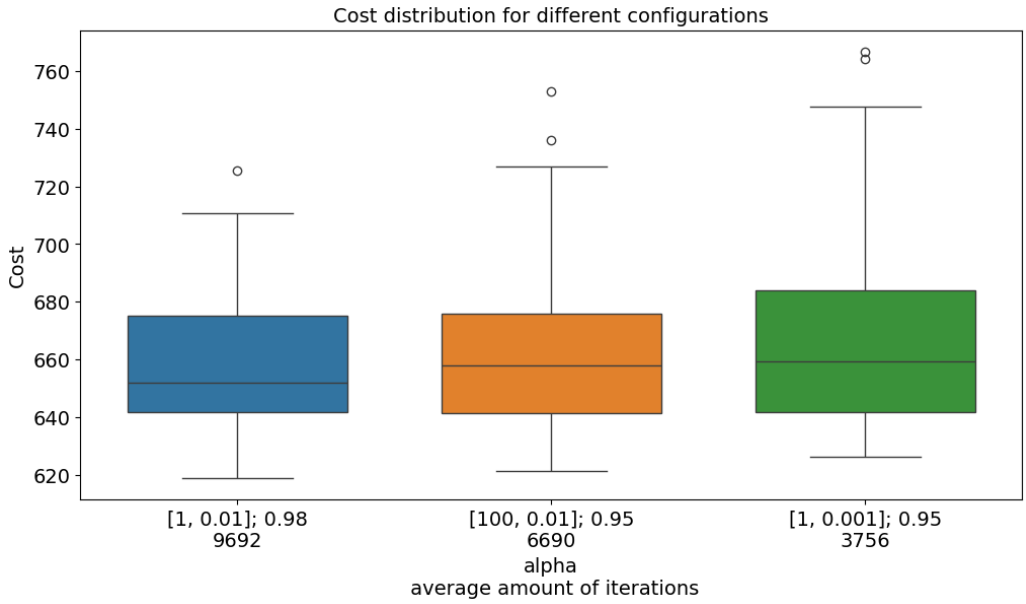


Figure 5.12: For similar amount of improvements, different configurations are displayed

If quick and rough results are desired, $T_{start} = 1000$ and $T_{end} >= 0.1$ for low α is advised. These results can be optimal, but are quite inconsistent for this very fast configuration. Choosing an even higher value for T_{start} is not useful as all uphill movement is allowed and once $T_i = 1000$ the current layout might be equal to the starting configuration of a different simulation with $T_{start} = 1000$. The other extreme example employs $T_{end} = 0.001$. These results are very spread out, as almost no uphill movement is allowed at the end, local minima are not avoided. The results can be very good, but can also get stuck very easily.

As stated earlier, the opamp circuit converged too slow. Hence, for $\alpha = 0.95$, smaller temperature intervals where used. The distribution for the cost is shown in figure 5.13. The first boxplot is for the default configuration and the second one increases both temperature bounds by 10, which leads to very similar results. The next three boxplots require less improvements, this leads to faster results. The optimal solutions can definitely be found but the upper bound is higher. Overall it is beneficial to use one of these intervals for the simplified operation amplifier. The last boxplot shows a very small temperature interval, the optimal solutions are not reached and it performs worse on average, because the algorithm stops too fast.

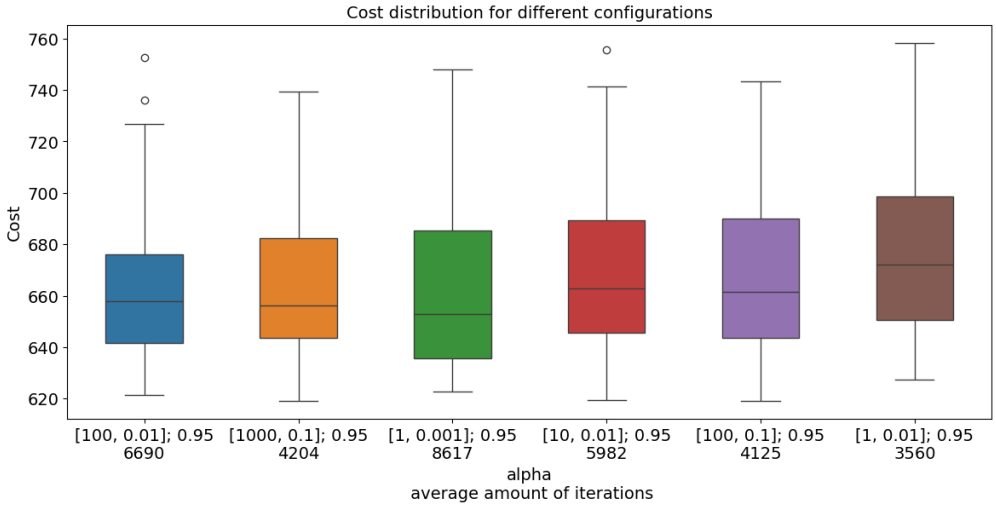


Figure 5.13: For $\alpha = 0.95$, the cost boxplot for different temperature intervals is displayed.

The final results is shown in figure 5.14 and displays a realistic DRC-clean operational opamp layout in Virtuoso as obtained by the layout generator.

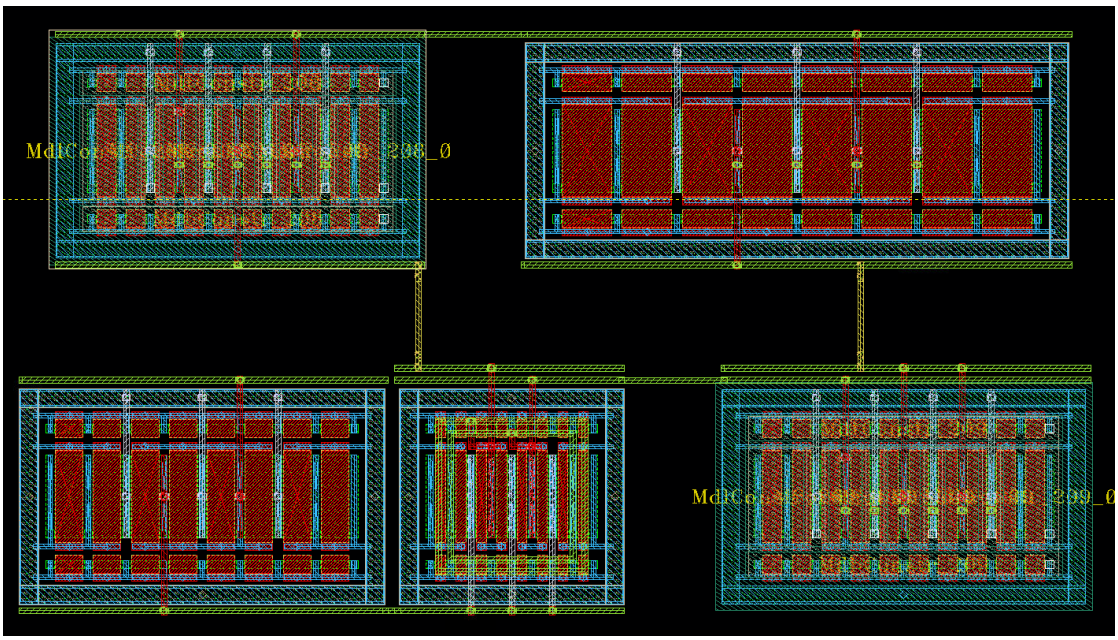


Figure 5.14: Layout in Virtuoso of an opamp. The n-well spacing is taken into account, as well as the constraints on the rails of the differential pair

6 CONCLUSION AND FUTURE WORKS

An effective layout generator is the result of this thesis. An optimal DRC-clean layout can be obtained. This layout generator employs an optimization process where the routing and placement happens simultaneously, while providing a lot of tunable parameters, such that its performance can be tweaked fitting. The optimization is done via simulated annealing which introduces random variations onto the layout. The routing is performed via an optimised and improved A* algorithm which allows 3D routing of the nets. However, the layout generator can still be improved a lot and has some shortcomings.

The layout generator does not take symmetry into account and, hence, is suboptimal for differential circuits. It can definitely route and place differential circuits, but due to matching problems the layout will not work properly. However, the fact that each primitive is symmetric already reduces the effect on matching significantly. Adding this would require detecting which cells require symmetry and then updating all actions to behave in a symmetric matter.

The router slows down the algorithm. This is why other layout generators opt for splitting the two main tasks. It is a dedicated choice to do the routing simultaneously as it allows for optimal results and more control during the process. To further increase the speed and performance, some potential improvements are provided.

One of the mentioned problem is that each explored cell requires the validation of all its neighbours, due to the required spacing between the traces. Nevertheless, this step can be skipped and `validMove` will only need to perform checks on the explored neighbours. By doing some preprocessing on the routing grid this can happen. If all the metal containing a different net includes a perimeter of the required spacing, these occupied cells will never be explored. This would speed up A* significantly and it will allow different spacings between different nets.

Furthermore, the concept of retrying previous interconnects can be explored. When the routing is complicated and only one cell is moved slightly the routing can happen almost instantly as only the nets of the moved cell require rerouting. Obviously this might inhibit the optimal solution, if certain traces are never redrawn. Therefore, traces can be reused for a maximum amount of, for example, 5 times after which they require rerouting. On top of that, certain actions could enforce a full redraw, specifically the actions that influence the router.

The limitation of the grid is avoided by not using metal layers with different minimal widths. If this would be the case, the grids would have different sizing and placing vias would be hard as it is difficult to know how the two grids would overlap and it would be likely that two traces above each other would not fully overlap.

The grid also allows making traces wider, but this is not yet added to the algorithm. One could retrieve the current density of certain nets and calculate the minimal required width of the traces. This would automatically mean that these nets get a higher priority as they contain more metal.

Another limitation is that adding larger components such as capacitors will immediately increase the gridsize a lot and slow down the routing. To solve this, the routing could be performed recursively. Larger cells could be routed on a more coarse grid to not require the large gridsize. The connection of the coarser grid to the smaller cells would then be done on a more precise grid, this would determine the exact position of the interconnect in the coarser grid. The idea of omitting the quantised grid could be explored ...

Now for the actions most of them perform quite good. However, one more action could improve on the placement. If cells are merged, moving them does not happen often, because the chance of having a higher cost is large, the only movement allowed is the movement along the rails. During the experiments many layouts where not optimal because of this, an example is shown in figure 6.1. Moving the group of cells that are merged would help this layout, therefore the new action would change the position of the cells that are merged together and then verifying whether no overlap happens. Perhaps merged cells could be interpreted as a single cell.

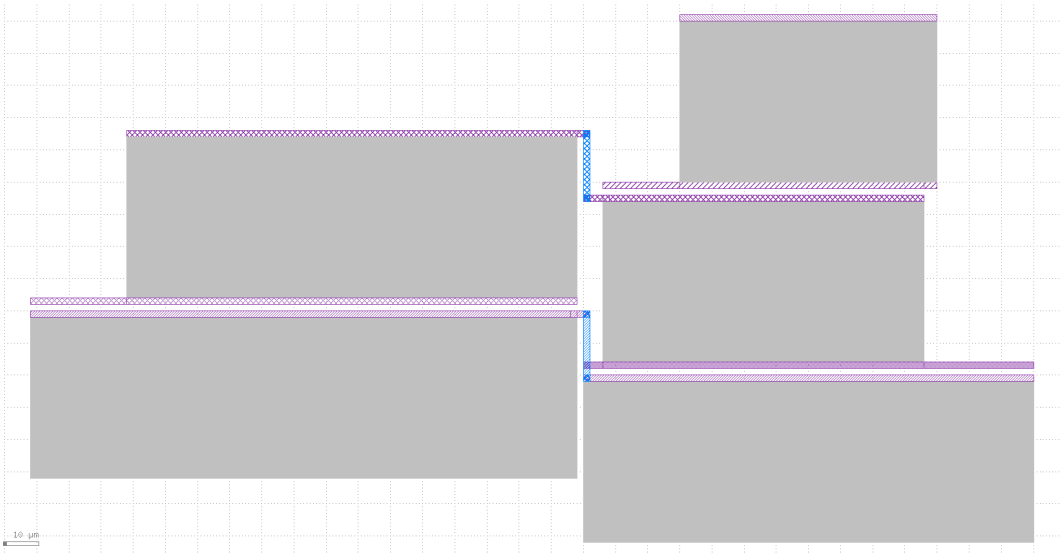


Figure 6.1: Less optimal layout, which requires additional action to improve

Another but less straightforward action would be to change the height of the rails. The reasoning for this can be seen in figure 6.2. Here the upper rail of the middle left cell could be risen to merge with the cell above. With the current actions and placement this merge cannot happen as it would mean shorting two cells. But if the rail could be placed higher, this could be resolved. This action would breach the black box assumption as now the connections from the terminal contacts to the rail interface need to change too, which would lead to an increase in resistance. This method imposes a trade-off between the decrease in resistance because of potential changes in interconnections and the added resistance due to the elongation of the connections inside the primitive.

This concept would also help for the cell in the upper right corner. The rail above that cell could be placed higher up, to align vertically with the adjacent rail, hence, omitting two vias and minimizing the interconnection length. This action is not easy to implement as it would also influence other actions. To retain the annealing characteristic of the action, it would be feasible to undo the change in height each time the specific cell is moved. Besides, it needs to be detected whether the change in height could allow rails to merge.

The way the actions are used could also be improved on. By using a function of the current temperature to determine the weight. This would mean that certain actions could be executed more frequently in certain parts of the algorithm. For example less merging in the beginning and less unprecise movement in the end of the process. Determining the specific function for this would be difficult and again reliant on the specific schematic, but it could provide a performance boost and more precise results.

Besides changing the actions based on the progress of the algorithm, it could also be beneficial to do, i.e.,

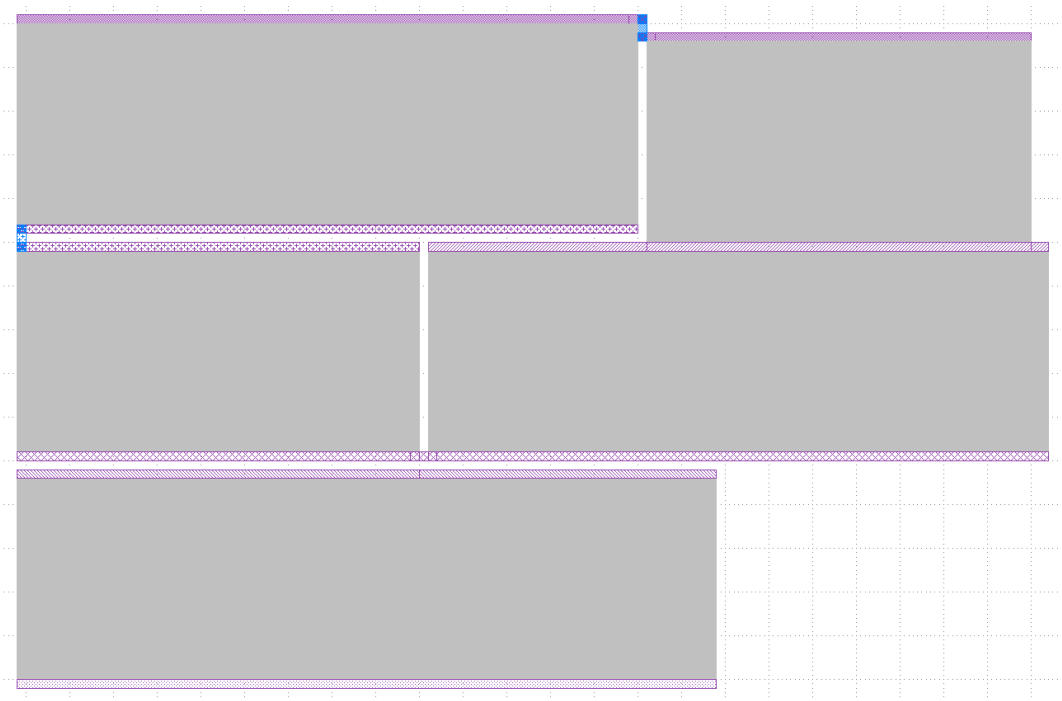


Figure 6.2: Almost optimal layout, which can benefit from a new action

the first quarter of the placement without the routing, which would lead to quick improvements on the area and placement. Then the routing is performed to further improve the layout, in a faster fashion, due to the cells already being merged or close together.

The layout generator was only tested for the opamp circuit, while it allowed much variation it was only a simple circuit without many different primitives. The specific algorithm that detects these primitives should be expanded further and how the detection behaves for other circuits with more complex primitives is unknown. Complex randomised structures where tested, but this is not a realistic way of gauging performance. For the moment the flexibility of the primitives is also not superb; unknown, predefined layouts can be added but the resulting layouts might not be optimal, because most actions cannot be performed on these predefined sub-blocks. In order to do that, the black box concept has to be lifted and more knowledge has to be obtained from these existing layouts.

Overall, this layout generator achieves precise routing and placement via simulated annealing and an improved A* algorithm. An operational amplifier was extensively studied to evaluate the performance of this tool, which extends the capabilities of IDcircuits, a python framework for automatic layout generation. By simplifying the available primitives, an optimization procedure was developed to place and route these primitives simultaneously. As the layout generator is meant to be used together with an IC designer, the framework allows a script and a configuration file to control the resulting layouts. The influence of the framework's parameters is clarified and a calibration method is provided to accurately gauge the layouts based on the enclosing area and parasitic resistance.

BIBLIOGRAPHY

[1] Noraziah Adzhar and Shaharuddin Salleh. "Simulated Annealing Technique for Routing in a Rectangular Mesh Network". In: Modelling and Simulation in Engineering 2014 (Dec. 2014). doi: 10.1155/2014/127359.

[2] Stefan Behnel et al. "Cython: The best of both worlds". In: Computing in Science & Engineering 13.2 (2011), pp. 31–39.

[3] Eric Chang et al. "BAG2: A process-portable framework for generator-based AMS circuit design". In: 2018 IEEE Custom Integrated Circuits Conference (CICC). 2018, pp. 1–8. doi: 10.1109/CICC.2018.8357061.

[4] Hao Chen et al. "MAGICAL: An Open-Source Fully Automated Analog IC Layout System from Netlist to GDSII". In: IEEE Design & Test 38.2 (2021), pp. 19–26. doi: 10.1109/MDAT.2020.3024153.

[5] Tonmoy Dhar et al. "ALIGN: A System for Automating Analog Layout". In: IEEE Design & Test 38.2 (2021), pp. 8–18. doi: 10.1109/MDAT.2020.3042177.

[6] Jaeduk Han et al. "LAYGO: A Template-and-Grid-Based Layout Generation Engine for Advanced CMOS Technologies". In: IEEE Transactions on Circuits and Systems I: Regular Papers 68.3 (2021), pp. 1012–1022. doi: 10.1109/TCSI.2020.3046524.

[7] Charles R. Harris et al. "Array programming with NumPy". In: Nature 585.7825 (Sept. 2020), pp. 357–362. doi: 10.1038/s41586-020-2649-2. url: https://doi.org/10.1038/s41586-020-2649-2.

[8] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: IEEE Transactions on Systems Science and Cybernetics 4.2 (1968), pp. 100–107. doi: 10.1109/TSSC.1968.300136.

[9] Paul G. A. Jespers and Boris Murmann. Systematic Design of Analog CMOS Circuits: Using Pre-Computed Lookup Tables. Cambridge University Press, 2017.

[10] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. "Optimization by Simulated Annealing". In: Science 220.4598 (1983), pp. 671–680. doi: 10.1126/science.220.4598.671. url: https://www.science.org/doi/abs/10.1126/science.220.4598.671.

[11] Ricardo Martins, Nuno Lourenço, and Nuno Horta. "LAYGEN II—Automatic Layout Generation of Analog Integrated Circuits". In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 32.11 (2013), pp. 1641–1654. doi: 10.1109/TCAD.2013.2269050.

[12] Taeho Shin et al. "LAYGO2: A Custom Layout Generation Engine Based on Dynamic Templates and Grids for Advanced CMOS Technologies". In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 42.12 (2023), pp. 4402–4412. doi: 10.1109/TCAD.2023.3294462.

[13] Guido Van Rossum and Fred L. Drake. Python 3 Reference Manual. Scotts Valley, CA: CreateSpace, 2009. isbn: 1441412697.