Optimal Layout for a Component Grid

Michael Ebert
Computer Science
California Polytechnic State University, San Luis Obispo

December 2017

Optimal Layout for a Component Grid

Michael Ebert

Abstract—Several puzzle games include a specific type of optimization problem: given components that produce and consume different resources and a grid of squares, find the optimal way to place the components to maximize output. I developed a method to evaluate potential solutions quickly and automated the solving of the problem using a genetic algorithm.

Index Terms—Combinatorial Mathematics, Genetic Algorithms, Simulation.

1 Introduction

Several strategy and puzzle games have a placement puzzle in them where you must place components on a grid in order to achieve some result. In two of these games, Reactor Idle [1] and IndustrialCraft^2 [2], the puzzle is power-themed. The goal of the puzzle in these two games is to generate electricity from a variety of components that can be put into four categories: Generators, which generate a fixed amount of heat; Amplifiers, which increase the effects of adjacent generators; Distributors, which distribute heat between adjacent components, and Boilers, which turn heat into electricity. If too much heat accumulates in a component, that component will break. The challenge, therefore, is to find a design of generators, amplifiers, distributors, and boilers that, in a limited amount of space and without exploding, will produce the most amount of power.

This is a NP problem; for C components and N tiles, there are $(C+1)^N$ possible layouts for a given problem. Taking a fairly tiny problem size of 4 different components and 20 tiles of space, this is approximately 9.5×10^13 combinations. If 1 million layouts were tested per second, this would take 3 years to brute force. Some of these puzzles have up to 10 components, and 100 tiles of space. Clearly, a better method is needed.

However, checking a layout is not fast. The way that the game runs is in discrete "ticks". Every tick, each component does its action once. Because of this, heat takes time to propagate across the design. An unstable design may not initially show any problems. A nearly stable design may slowly build up heat for several thousand ticks before being destroyed by excess heat. Additionally, components interact with each other, such as amplifiers and generators, or distributors and adjacent components. Because some of their actions depend on the heat levels of adjacent components, they remove the possibility of a simple O(1) confirmation that a layout is valid. A naïve simulation (and the ones used to check designs in games) simulates designs for hundreds or thousands of ticks to determine what happens to a layout. Simulating a design this way can take hundreds of microseconds. In order to efficiently find a good design, it is necessary to first construct an algorithm that can simulate a design several orders of magnitude faster.

M. Ebert, Computer Science, California Polytechnic State University, San Luis Obispo
E-mail: mebert@pacbell.net

I have been interested in these games for a while and find them fun. The problem also seems like a good candidate for a computer solution – discrete components and values, deterministic outcomes, and a clear grading algorithm in power produced.

1.1 Literature Overview

With regards to efficiently simulating a single layout, the most relevant area of research is flow maximization. There are several algorithms to do this, such as the Edmonds-Karp algorithm [3], which runs in $O(VE^2)$, and the push-relabel algorithm [4], which runs in $O(V^2E)$, and can run faster with optimizations. A modified push-relabel is needed to cope with the unique constraints of my problem, namely that 1) there are 'dumb' nodes that emit the same amount to all downstream nodes, which is not possible with the basic algorithms, and 2) the input is given in discrete steps, not a continuum. Max-flow algorithms assume that the flow is effectively continuous – that the total possible flow is an integer multiple of the potential input. With my puzzle, source components (generators) produce 100% output all the time, and the design is invalid if the component network cannot handle it.

With regards to finding an optimal layout of components, most placement optimization research is focused in two areas: very large integrated circuit (VLSI) placement and the facility layout problem (FLP). Unfortunately, both areas operate under a different set of assumptions. Usually, there is a known number of components to place in a larger area, and crucially, the components are separated from each other. This allows the efficiency of the design to be proportional to the distance between components on small scales. In the problem that I am trying to solve, components have very strict placement requirements, and there is a very complex relationship between component placement and efficiency.

Most placement optimization solutions rely on simulated annealing or genetic algorithms [5], [6]. Even though the details of the problems are not commutative, these algorithms are good places to start.

2 Implementation

As there are several different puzzles that vary only slightly from each other, I decided to create a solution that would

work for all of them with some tweaking. I wanted to avoid building my implementation around assumptions that were present in a single puzzle, but would cause my solver to be useless if used for an other, slightly different puzzle. As a result, I divided the puzzle-specific elements(number of components, what each component does, global properties of the environment, etc.) from the more general elements of the code (general setup and run drivers, solution generation algorithm, etc.), which would interact with the puzzle-specific elements through template instantiation and metaprogramming. Although separate, the puzzle specific elements are still defined at compile time for speed. Building this flexibility dynamically into my solver at runtime is possible, but would be inefficient, as constants, branches, and even the effects of entire functions would have to be evaluated at runtime instead of compile time. This could dramatically reduce the performance of the code (for example, the effects of the ”empty” component: If evaluated at compile time, it can compile to a no-op, whereas at runtime, the function still has to be called). Additionally, since so much more time is spent running the algorithm than compiling it (hours vs minutes), one of the major advantages of runtime configuration, being able to make changes quickly, is rendered meaningless.

2.1 Layout Evaluation

2.1.1 Optimizations

One of the two algorithms that I had to design was a way to evaluate the performance of a layout. In order to test a layout quickly, I allowed some inaccuracy in the simulation. All correct layouts are recognized as correct, but some incorrect designs are also marked as correct. These can be easily filtered out in a more rigorous simulation once a small group of candidate solutions are generated.

There were 4 techniques that I used to speed up the simulation that had a major impact on how the project was written.

The main speedup in the simulation comes from eliminating the multiple ticks of simulation needed in the naïve design, effectively simulating the design in a single step. The basic idea is that I simulate one tick propagating through the system and see what resources are left over. This requires some careful planning in order to avoid problems with loops and isolated parts of the design.

Secondly, I wrote the code with speed in mind. I used C++ with heavy use of templates and compile-time constants. This allows for elimination of unnecessary branching and function calls, resulting in a smaller and faster binary. Furthermore, most of the code is in header files, compiling in a single compilation unit. This allowes constant propagation, inlining, and other optimization techniques to happen throughout most of the code (This sped up the runtime by 20% when using GCC 6.1.0 -O3).

Thirdly, I wanted to enable concurrent and out-of-order execution wherever possible. To do this, I divided up each component’s functionality into two steps: component_setup() and component_action(). All the component_setup()’s can run in parallel, and same is true of the component_action()’s. After all the component have been run, usually a grading function is run in order to score the design.

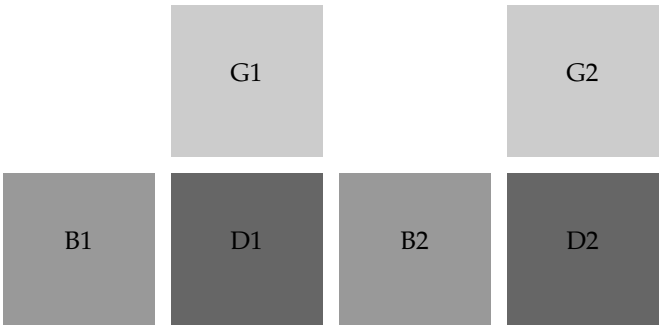

Fig. 1. Valid layout that may be marked as faulty

A fourth decision that I made was that instead of the traditional setup of a grid of contiguous component properties, I instead relied on a series of grids, one for each component property. For example, there is a grid storing the type of each component, a separate grid storing the heat of each component, and a third grid storing the energy production of each component. By storing properties by type instead of by component, memory access is optimized, as the simulation code tends to access one or two properties of all components rather than all the properties of a single component.

2.1.2 Major problems encountered and solved

One problem that I ran in to was that blindly traversing a layout is costly because of potential backtracking and loops. To solve this, I grouped the boilers into ”networks” of boilers connected by distributors. Since all boilers in a network are connected, the entire block acts as one large boiler. I did this by having each valid component (boiler or distributor) either connect to an adjacent component’s network or create a new network if one doesn’t exist. If two different networks are adjacent to a single valid component, then the two networks are merged. This makes the heat distribution work much better and have less error cases.

Another problem that I ran into was misallocation of resources. As an example, take the layout in figure 1. G1 and G2 generate 10 heat per tick, D1 and D2 are distributors (that distributes heat), and B1 and B2 are boilers that absorb 10 heat per tick. This layout would work in the actual simulation, but not in a single passthrough simulation like I am doing. This is because in a single pass, D1 will distribute 5 heat to B1 and B2, and then D2 will distribute 10 more heat to B2. This will result in 15 heat in B2 and only 5 in B1. As a result, B2 will not be able to remove all of the heat and the design will be marked as failed. In the actual sim, this would happen on the first tick, but on the second and subsequent ticks, the excess heat from B2 would be moved over to B1, allowing the design to succeed. It is not possible to simply sum up the total number of boilers in the design, as disconnected boilers would be included when they should not be. This problem prompted me to introduce a modified push-relabel maximum flow algorithm into the simulation. If the max flow can remove all the heat, then the

$num\_designs$ = number of designs in each generation
num_top = number of top designs to copy, unmodified, into the next generation
$num\_new\_designs$ = number of designs to generate new each generation $(num\_designs - num\_top)$
$num\_generations$ = number of generations to simulate
$mutate\_chance$ = chance of a spontaneous mutation occuring
end_ix =last index of a design (size of a design in places - 1)
for $num\_generations$ **do**
  score all designs
  copy top num_top designs into next generation
  for $num\_new\_designs$ **do**
    select parents A,B using exponential distribution
    select a uniformly random spot R in the middle of the design $0 \leq R < end\_ix$
    $N[0, R) = A[0, R)$
    $N[R, end\_ix) = B[R, end\_ix)$
    select random float S between 0 and 1
    if $S < mutate\_chance$ **then**
      select a random component in N and change it to a random component
    end if
  end for
end for

Fig. 2. Genetic algorithm pseudocode

design may work. If not, then the design is guaranteed to fail.

2.2 Layout Optimization

After I developed a faster evaluator, I used it along with a genetic algorithm to try to find the optimal layout.

I chose to use a genetic algorithm as it seemed best aligned with the design of the problem that I was trying to solve. Layouts already have a authorative score in the amount of electricity they produce, and this scoring algorithm is an essential part of the simulation itself. Furthermore, this problem does not lend itself to simulated annealing, as good, bad, and invalid layouts are extremely close to each other, and there isn't a linear path between them.

My genetic algorithm is fairly simple, and a high level overview is given in figure 2. I was able to parallelize the simulation of designs, making the algorithm scale nearly linearly with the number of cores.

3 Results

3.1 Layout Evaluation

The work on designing an algorithm that would evaluate designs quickly was a massive success. I achieved a 1000-fold increase in the evaluation speed with only a few subtly flawed invalid designs being misidentified. Additional speedup may be possible by simple checking of grids to see if they look invalid – for example, if the total heat emitted by reactors is greater than the total amount of heat that could be absorbed by every component on the board, regardless of connectivity, it can be thrown out immediately.

Additionally, my goal of allowing modification of the components and their behavior was a success. The code is very modular. Additional components can be added in a few minutes, and properties of components can be changed without having to rewrite significant parts of the code. Additionally, the grading algorithm can be replaced or modified with minimal effort (Integrating the maximum flow algorithm into the code took only a day or so). The same code framework, slightly modified, could be used to simulate tower defense designs, for example.

3.2 Layout Optimization

With regards to generating an optimal solution, there is much more work to be done. While I was able to generate valid designs, I was never able to produce a design that was significantly better than what a human could produce manually in the same amount of time. What would help would be pruning the space that the genetic algorithm explored, which I struggled with.

References

[1] Baldurans. (2015, Nov.) Reactor idle. [Online]. Available: http://reactoridle.com/
[2] Alblaka and IC2 Dev Team. (2011, Oct.) Industrialcraft 2. [Online]. Available: http://www.industrial-craft.net/
[3] J. Edmonds and R. M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," *J. ACM*, vol. 19, no. 2, pp. 248–264, Apr. 1972. [Online]. Available: http://doi.acm.org/10.1145/321694.321699
[4] A. V. Goldberg and R. E. Tarjan, "A new approach to the maximum-flow problem," *J. ACM*, vol. 35, no. 4, pp. 921–940, Oct. 1988. [Online]. Available: http://doi.acm.org/10.1145/48014.61051
[5] C. L. Valenzuela and P. Y. Wang, "Vlsi placement and area optimization using a genetic algorithm to breed normalized postfix expressions," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 4, pp. 390 – 401, Aug. 2002. [Online]. Available: http://doi.acm.org/10.1145/48014.61051
[6] T. D. Mavridou and P. M. Pardalos, "Simulated annealing and genetic algorithms for the facility layout problem: A survey," *Computational Optimization and Applications*, vol. 7, no. 1, pp. 111–126, Jan 1997. [Online]. Available: https://doi.org/10.1023/A:1008623913524