A Solution To The Developer Allocation Problem Using Simulated Annealing

Jorge Reyes-Spindola

September 20, 2018

1 Introduction

At large companies the project-developer allocation problem is encountered all the time. There are a number of projects available and requiring resources and developers wishing to work on them. The projects are managed by one or more project managers (PMs) and developers are required to rank a certain number of projects in order of preference. This process is the one-sided variant of the problem. The two-sided would be when the project (or PM) expresses preference for a programmer or programmers.

Usually, only one developer or pair of developers work on a given project. The problem arises when several developers choose a particular project as their first choice. Inevitably, some developers will be assigned their second or third choices which in turn causes secondary effects because the less preferred projects could be the first choice of another developer.

An additional issue is the PM's workload. Usually, a PM will propose more projects than they can feasibly oversee in order to provide a wider range of developer choices. Further complications arise when all of a PM's projects are popular because it guarantees some developers will not be assigned their first choice.

The developer-project problem is a specific case of the generalized assignment problem, a well-known optimization problem that consists of assigning sets of jobs to sets of agents while minimizing the cost associated with the assignment; this problem can be formulated as an integer linear problem. A wide range of methods have been devised for solving such problems including, a genetic algorithm, and a local-ratio technique for the knapsack problem.

In this paper we investigate how simulated annealing (SA), a method from statistical and computational physics, can be used to obtain a "good enough" solution to the developer-project problem.

2 Problem Specification

We focus on the assignment of development projects at an unnamed company. Each developer (or team of developers) is invited to submit a list of four project preferences, ranking them from 1 to 4 in order of preference (1 = most preferred). The PMs then assign developers to projects with the goal of maximizing developer satisfaction while meeting the following constraints:

1. Each developer is assigned to one of the projects on their preference list.
2. No project can be assigned to more than one developer (or team of developers).
3. PMs cannot be assigned more projects than they can feasibly oversee.

The mathematical framing of the problem is as follows:

Let N be the number of developers, M is the number of projects (with $M > N$) and S is the number of PMs. Define a $N \times M$ matrix C with elements

$$C_{ij} = \begin{cases} 1 & \text{if developer } i \text{ chose project } j \\ 0 & \text{otherwise,} \end{cases}. \tag{1}$$

and define an $N \times M$ allocation matrix X by

$$X_{ij} = \begin{cases} 1 & \text{if developer } i \text{ is assigned to project } j \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

Our goal is to maximize developer satisfaction. To do so, we need a definition of satisfaction. Clearly, developers will be happier if they receive their first choice and less happy if they receive their fourth choice. We define

$$\mathfrak{O} = \sum_{k=1}^{4} w_k n_k \tag{3}$$

where n_k is the number of developers assigned their kth choice in the allocation and w_k is the weighting assigned to the kth choice, with $w_k > w_{k+1}$. The sum in equation (3) is our measure of overall satisfaction. In the optimization literature such a quantity is referred to as the *objective function*.

The first two constraints can be written simply as

$$\sum_{j=1}^{M} C_{ij} X_{ij} = 1 \quad \forall i = 1, \ldots, N \tag{4}$$

$$\sum_{i=1}^{N} X_{ij} \leq 1 \quad \forall j = 1, \ldots, M. \tag{5}$$

To incorporate the third constraint, we let F_s denote the number of projects that supervisor s has in the assignment, and L_s denote the largest number of projects they can oversee. Then

$$F_s \leq L_s \quad \forall s = 1, \ldots, S. \tag{6}$$

Given a matrix of choices C, our goal is to find an allocation X_{ij} that maximizes the value of $\mathfrak{O}$ given the weights w_k and the constraints in equations (4) - (6). The following section will describe the SA algorithm that is used to solve this problem.

3 Simulated Annealing

Annealing is the physical process of heating up a solid until it melts, followed by cooling it down until crystallizes into a state with a perfect lattice. During this process, the free energy of the solid is minimized. Practice has shown that the cooling must be done carefully in order not to get trapped in locally optimal lattice structures with crystal imperfections.

In combinatorial optimization, we can define a similar process. This process can be formulated as the problem of finding -among a potentially very large number of solutions- a solution with minimal cost. Now, by establishing a correspondence between the cost of functions and the free energy, and between the solutions and the physical states, we can introduce a solution method in the field of combinatorial optimization based on a simulation of the physical annealing process. The resulting method is called *Simulated Annealing*.

Salient features of this method are its general applicability and its ability to obtain solutions arbitrarily close to an optimum. A major drawback however, is that finding high-quality solutions may require large computational efforts.

3.1 The Metropolis Algorithm

The physical annealing process can be modelled successfully by using computer simulation methods from condensed matter physics. As far back as 1953, Metropolis, *et.al.* introduced a simple algorithm for simulation the evolution of a solid in a heat bath to *thermal equilibrium*. The algorithm introduced is based on *Monte Carlo techniques* and generates a sequence of states of the solid in the following way. Given a current state i of the solid with energy E_i, then a subsequent state j is generated by applying a perturbation mechanism which transforms the current state into a next state by a small distortion, for instance by displacement of a particle. The energy of the next state is E_j. If *energy difference* $E_j - E_i$ is less than or equal to 0, the state j is accepted as the current state. If the energy difference is greater than 0, the state j is accepted with a certain probability given by

$$\exp\left(\frac{E_i - E_j}{k_B T}\right) \tag{7}$$

where T denotes the *temperature* of the heat bath and k_B a physical constant known as the *Boltzmann constant*. The acceptance rule described above is known as the *Metropolis criterion* and the algorithm that goes with it is known as the *Metropolis algorithm*.

3.2 The SA Algorithm

Returning to the algorithm, we can apply the Metropolis algorithm to generate a sequence of solutions of a combinatorial optimization problem. For this purpose we assume an analogy between a physical many-particle system and a combinatorial optimization problem based on the following equivalences.

- Solutions in a combinatorial optimization problem are equivalent to states of a physical system.
- The cost of a solution is equivalent to the energy of a state.

The simulated annealing algorithm can now be viewed as an iteration of Metropolis algorithms, evaluated at decreasing values of a control parameter which plays the role of the temperature. This parameter is called the *control parameter*. We now introduce the following definitions.

Definition. Let (S, f) denote an instance of a combinatorial optimization problem and i and j two solutions with cost $f(i)$ and $f(j)$ respectively. Then the *acceptance criterion* determines whether j is accepted from i by applying the following *acceptance probability*:

$$P_c(\text{accept } j) = \begin{cases} 1 & \text{if } f(j) \leq f(i) \\ \exp\left(\dfrac{f(i) - f(j)}{c}\right) & \text{if } f(j) > f(i), \end{cases} \tag{8}$$

where $c \in \mathbb{R}^+$ denotes the control parameter.

Definition. A *transition* is a combined action resulting in the transformation of a current solution into a subsequent one. The action consists of the following two steps: (*i*) application of the generation mechanism, (*ii*) application of the acceptance criterion. Therefore, the Metropolis algorithm operates as follows

1. Choose an initial arrangement **a**.
2. Choose a trial arrangement **a**′.
3. Calculate the energy difference.

4. Generate a random number $r \in (0, 1)$.
5. If $r < P(\mathbf{a} \rightarrow \mathbf{a}')$, move to state $\mathbf{a}'$
6. Generate a new trial arrangement and return to step 3.

4 Implementation

To find a good solution of the developer-project allocation problem using the SA algorithm, we implemented Python code in the following notebook. The notebook is available in Github (insert link here). While we won't discuss all the details of the implementation, we will cover some salient points.

4.1 Objective function

We strove to maximize developer happiness and so we defined the energy in terms of the objective function as

$$E = -\mathfrak{O} = -w_1 n_1 - w_2 n_2 - w_3 n_3 - w_4 n_4.$$

Recall that w_k is the weight of preference k and n_k is the number of developers assigned their kth preference. Naturally, the preference ranking (weights) should decrease with k. We initally chose to work with a set of linearly decreasing weights, that led to the objective function

$$E = -4n_1 - 3n_2 - 2n_3 - n_4.$$

However, after some work, it was observed that this objective function let to several degenerate (*i.e.* same energy) solutions. Because we wanted to discriminate among these and because a better way of gauging developer satisfaction was needed, a new objective function based on developer's opinions was developed. Developers were polled to determine their satisfaction outcome on the basis of their project assignation. Using this data, the new opinion-based weights were now determined to be $w_1 = 4.7$, $w_2 = 4.15$, $w_3 = 3.0$, and $w_4 = 2.35$, thus obtaining the new normalized objective function

$$E' = -\frac{100}{N}\left[n_1 + \frac{4.15}{4.7} n_2 + \frac{3.0}{4.7} n_3 + \frac{2.35}{4.7} n_4\right].$$

Using this objective function, the energy corresponding to the best possible allocation has a value of $E^* = -100$. The best possible allocation is where every developer gets their first choice project.

4.2 Sampling Strategy

The starting point for an SA optimization must be an initial allocation that satisfies the constraints in the equations above. To obtain a suitable initialization, we randomly assign each developer one of their four choices. In general, this assignment will lead to multiple constraint violations with different developers being assigned the same project and PMs getting overworked. To correct these violations, we perform a preliminary Monte Carlo procedure in which we choose a developer at random and randomly re-allocate them to one of their four project choices. If this proposed change results in a reduction in the number of violated constraints, we accept the move, otherwise we reject it. This process is iterated until we obtain an allocation with no violations, which we take as the initial configuration for the application of the SA algorithm.

An important aspect of the SA procedure is the manner in which trial allocations are proposed within the process. The method adopted should be reversible and ergodic -all allocations should be reachable within the sampling procedure. These conditions can be achieved by iterating the following procedure:

1. Choose a developer at random.

2. Choose a random integer from 1 to 4.
3. If the developer is currently assigned the project corresponding to that choice, repeat step 2. Otherwise, assign the developer the trial project corresponding to the random choice. This assignment is the trial allocation.
4. Check that for the trial allocation no project is assigned to multiple developers. If a conflict has arisen, retain the current allocation and go to step 1.
5. Check that the PM workload limit is not violated. If it is, then retain the current allocation and go to step 1.
6. Calculate the Metropolis acceptance probability $P(\mathbf{a} \to \mathbf{a}')$. Generate a random number $r \in (0,1)$. If $P < r$, retain the current allocation and go to step 1. If $P \geq r$, accept the trial allocation as the current allocation.

5 Results

A run of initial results is shown here below the notebook. The dataset used contained 19 developers looking to be allocated amongst 58 projects. The number of project managers available was 27. In Figure 1 we show a plot of the system's energy as a function of time instead of temperature. Since temperature is decreasing, we felt that a plot of energy vs. temperature wouldn't be very informative. It is possible to see how the energy peaks and dips along the process but always with a downward trend until it settles to a steady state.

While we are not certain that the optimal allocation reached is the absolute optimal, it is certainly one that is quite satisfactory. We see that out of the 19 developers (Figure 2), 10 were allocated their first choice, 6 their second choice and 3 their third.

In regards to computational efficiency, we regret that, for the time being, better use of Pandas and Numpy arrays was not achieved. We feel that it is possible to improve the calculation speed if the vectorizing capabilities of Pandas and Numpy were used better. Furthermore, it is possible that the number of moves required to achieve an energy state could be reduced but that is currently under research.

The SA algorithm could have many other applications. With more work it's reasonable to think that it could be applied to the two-sided allocation problem. This situation is one where the developers have a choice of their projects plus the projects also make choices amongst the developers.

6 Notebook

The following notebook was created using Jupyter Notebook version 5.6, Python 3.6, NumPy, Pandas, and the Plotly library.

```
In [1]: def createInitialConfiguration(choices, projNum, projPref, changes, pmConstraints):
            """
            Create an initial configuration. Start at random, and then accepty any randomly
            determined change that reduces the number of constraints being violated. We are
            done when no more constraints are violated.
            """
            for i in range(cols):
                pref = np.random.randint(1,5)
                for j in range(rows):
                    if choices.iat[j,i] == pref:
                        projNum[i] = j
                        projPref[i] = pref
```

```
            violationCount1 = countViolations(projNum, pmConstraints)
            while violationCount1 > 0:
                changeAllocationByPref( choices, projNum, projPref, changes )
                violationCount2 = countViolations( projNum, pmConstraints )
                if violationCount2 > violationCount1:
                    projNum[changes[0]] = changes[1]
                    projPref[changes[0]] = changes[2]
                else:
                    violationCount1 = violationCount2
```

```
In [2]: def countViolations(projNum, pmConstraints):
            """
            Returns a count of all violations.
            """
            count = 0
            count += projClashFullCount(projNum)
            for k in range(cols):
                count += countSupConstraintClashes(pmConstraints, projNum, projNum[k])

            return count
```

```
In [3]: def projClashFullCount(projNum):
            """
            Counts how many collisions are there in the allocation.
            Returns number of collisions.
            Returns 0 if no collisions.
            """
            count = 0
            for i in range(cols):
                for j in range(i,cols):
                    if i != j:
                        if projNum[i] == projNum[j]:
                            count+=1

            return count
```

```
In [4]: def countSupConstraintClashes(pmConstraints, projNum, proj):
            """
```

```
        Counts how many times the project manager constraint is violated.
        For each project assigned to a pair (or dev), how many times
        the PM constraint is violated.
        i is project/row
        j is project manager
        l is pair (or dev).
        """
        tsum = 0.0
        clash = 0
        # For the project proj we look across the row to see which supervisors it has.
        # Then we go down the supervisors column and we sum up the energy of the projects
        # allocated ONLY. If sum > 1, violation.
        for j in range(numPMs):
            tsum = 0.0
            if pmConstraints.iat[proj,j] != 0:
                for i in range(rows):
                    for l in range(cols):
                        if projNum[l] == i:
                            tsum += pmConstraints.iat[i,j]

                if tsum > 1:
                    clash += 1

        return clash
```

```
In [5]: def changeAllocationByPref(choices, projNum, projPref, changes):
        """
        This function changes the allocation. Based on picking a Developer,
        picking a project, and then making the change. Stores the change
        in the changes array.
        """
        go = True
        pair = np.random.randint(cols)

        while go:
            pref = np.random.randint(1,5)
            if projPref[pair] != pref:
                go = False

        changes[0] = pair
        changes[1] = projNum[pair]
        changes[2] = projPref[pair]

        # Make the change
```

```
            for j in range(rows):
                if choices.iat[j,pair] == pref:
                    projNum[pair] = j
                    projPref[pair] = pref
```

```
In [6]: def energy(projPref):
            """
            Calculates the energy of a given allocation.
            """
            energy = 0
            for i in range(cols):
                if projPref[i] == 1:
                    energy -= weight1
                elif projPref[i] == 2:
                    energy -= weight2
                elif projPref[i] == 3:
                    energy -= weight3
                elif projPref[i] == 4:
                    energy -= weight4

            return energy
```

```
In [17]: def cycleOfMoves(choices, projNum, projPref, changes, projmgrs):
             successfulMoves = 0
             moves = 0
             same = 0

             currentEnergy = energy(projPref)
             currentEnergyPlot.append(currentEnergy)
             #print('Temperature: {:3f} Current Energy: {:6f}'.format(temp, currentEnergy))

             while (moves < (1000*cols) and successfulMoves < (100*cols)) :
                 moves += 1
                 successfulMoves += 1
                 # Change the allocation here
                 changeAllocationByPref( choices, projNum, projPref, changes )
                 trialEnergy = energy(projPref) # Energy of the new allocation
                 changeEnergy = trialEnergy - currentEnergy

                 # projNum[changes[0]] != changes[1] at this point.
```

```
                # The former is current proj, the latter old proj
                pmClashes = countSupConstraintClashes( projmgrs, projNum, projNum[changes[0]] )

                if projClashFullCount(projNum) > 0: # Reject configuration due to collision
                    # - revert changes and reduce successful move counter
                    projNum[changes[0]] = changes[1]
                    projPref[changes[0]] = changes[2]
                    successfulMoves -= 1
                elif temp > 0 and np.random.random() > math.exp( -changeEnergy / temp ):
                    # Reject configuration due to energy - revert changes
                    projNum[changes[0]] = changes[1]
                    projPref[changes[0]] = changes[2]
                    successfulMoves -= 1
                elif temp == 0 and trialEnergy > currentEnergy:
                    # Reject due to energy in T=0 case
                    projNum[changes[0]] = changes[1]
                    projPref[changes[0]] = changes[2]
                    successfulMoves -= 1
                elif pmClashes > 0: # Reject due to PM constraint violation
                    projNum[changes[0]] = changes[1]
                    projPref[changes[0]] = changes[2]
                    successfulMoves -= 1

                if currentEnergy == trialEnergy:
                    # In theory, impossible but it serves as a bug alert
                    print("Same energy. Impossible state.")
                    same += 1
                    successfulMoves -= 1

                currentEnergy = energy(projPref)
```

```
In [8]: import numpy as np
        import pandas as pd
        import math

        cols = 19
        rows = 58
        numPMs = 27

        devsFilename = 'DevExampleNoA.csv'
        pmsFilename = 'ProjectManagerExample.csv'

        choices = pd.read_csv(devsFilename, header=None, dtype=int, na_values='')
        projmgrs = pd.read_csv(pmsFilename, header=None).fillna(0)
```

```
        np.random.seed(555)   # Seeded to allow replication.
        projNum = np.zeros(rows,dtype=int)
        projPref = np.zeros(rows, dtype=int)
        changes = np.zeros(3, dtype=int)
        currentEnergyPlot = []

        # Weightings
        # THIS IS VERSION WITH 4.7, 4.15, 3, 2.3 (out of 5)
        weight1 = 100/cols
        weight2 = 100/cols * 4.15/4.7
        weight3 = 100/cols * 3/4.7
        weight4 = 100/cols * 2.35/4.7

        temp = 5 # Starting temperature

        createInitialConfiguration( choices, projNum, projPref, changes, projmgrs )

        while temp >= 0:
            cycleOfMoves(choices, projNum, projPref, changes, projmgrs)
            temp -= 0.1
            temp = round(temp,2)
            print('Temp = {:2f}'.format(temp))

        print('Final energy is {:.4f}'.format(energy(projPref)))
        print('Final allocation: ')
        for i in range(cols):
            print('Developer {:2d}, Proj# {:2d}, Pref {:2d}'.format(i, projNum[i], projPref[i]))
```

```
Final energy is -90.5935
Final allocation:
Developer  0, Proj# 13, Pref  2
Developer  1, Proj# 29, Pref  3
Developer  2, Proj# 36, Pref  2
Developer  3, Proj#  4, Pref  1
Developer  4, Proj# 23, Pref  1
Developer  5, Proj# 51, Pref  1
Developer  6, Proj#  5, Pref  1
Developer  7, Proj# 26, Pref  1
Developer  8, Proj# 20, Pref  1
Developer  9, Proj#  3, Pref  1
Developer 10, Proj# 57, Pref  3
Developer 11, Proj# 46, Pref  2
Developer 12, Proj# 21, Pref  1
```

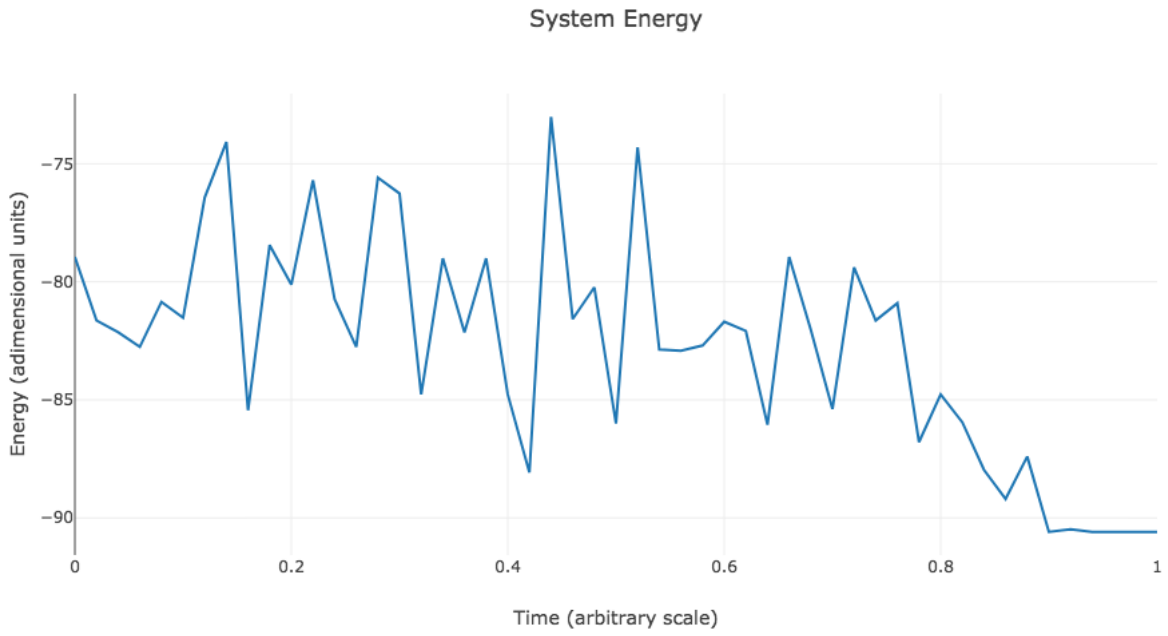
```
Developer 13, Proj# 56, Pref  1
Developer 14, Proj# 50, Pref  2
Developer 15, Proj#  1, Pref  3
Developer 16, Proj# 33, Pref  1
Developer 17, Proj# 14, Pref  2
Developer 18, Proj# 43, Pref  2
```

6.0.1 Plot of the system energy as a function of time

```
In [9]: import plotly
        import plotly.plotly as py
        import plotly.graph_objs as go

        N = len(currentEnergyPlot)
        random_x = np.linspace(0, 1, N)
        random_y = currentEnergyPlot

        # Create a trace
        trace = go.Scatter(
            x = random_x,
            y = random_y
        )

        data = [trace]

        layout = dict(title = 'System Energy',
                      xaxis = dict(title = 'Time (arbitrary scale)'),
                      yaxis = dict(title = 'Energy (adimensional units)'),
                      )

        fig = dict(data=data, layout=layout)
        py.iplot(fig, filename='basic-line')
```

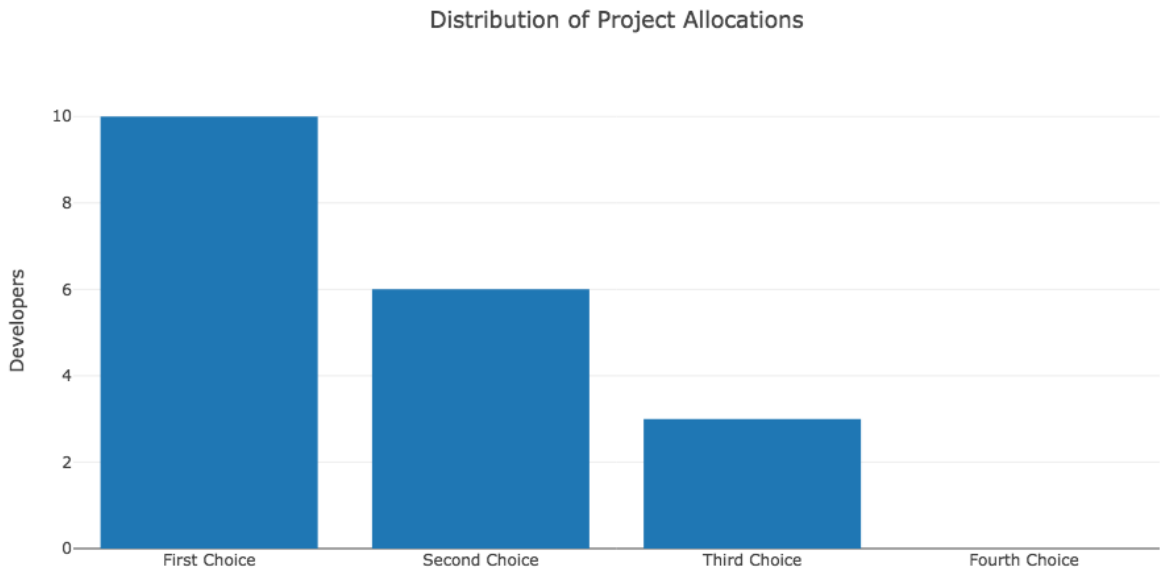
6.0.2 Distribution of developer's choices

```
In [10]: prefs = projPref.tolist()
         data = [go.Bar(
                     x=['First Choice', 'Second Choice', 'Third Choice', 'Fourth Choice'],
```



Plot of system energy vs. time

```
            y=[prefs.count(1), prefs.count(2), prefs.count(3), prefs.count(4)]
    )]
layout = dict(title= 'Distribution of Project Allocations',
              yaxis = dict(title='Developers'))
fig = dict(data=data, layout=layout)

py.iplot(fig, filename='basic-bar')
```



Allocation Histogram