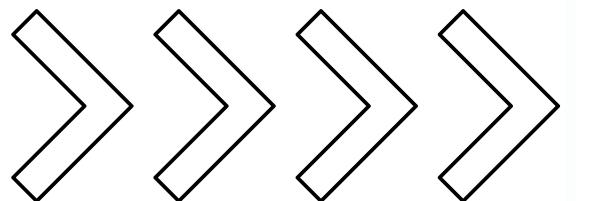
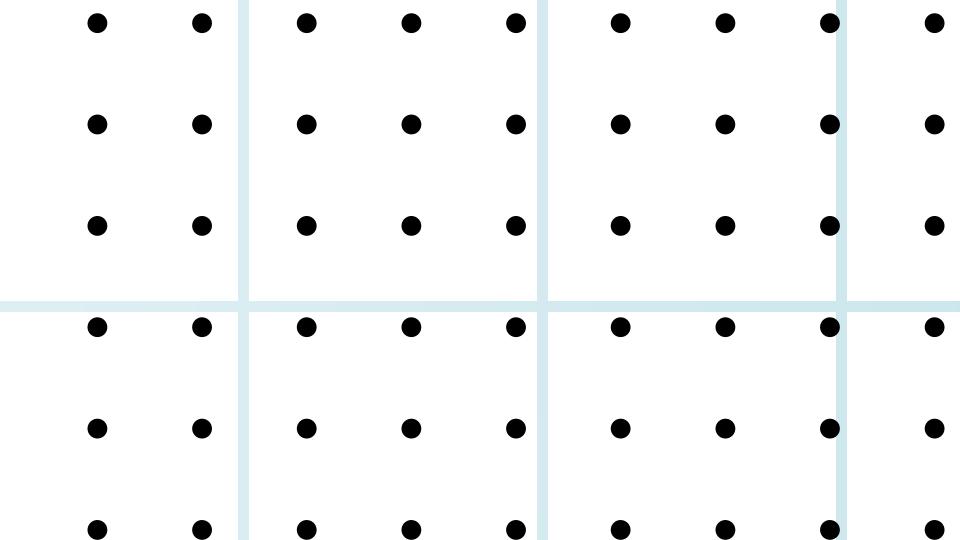


BALLS AND BINS

SIMULATED ANNEALING



João Vitor
Spotavore
Ignacio
Luis



Problema

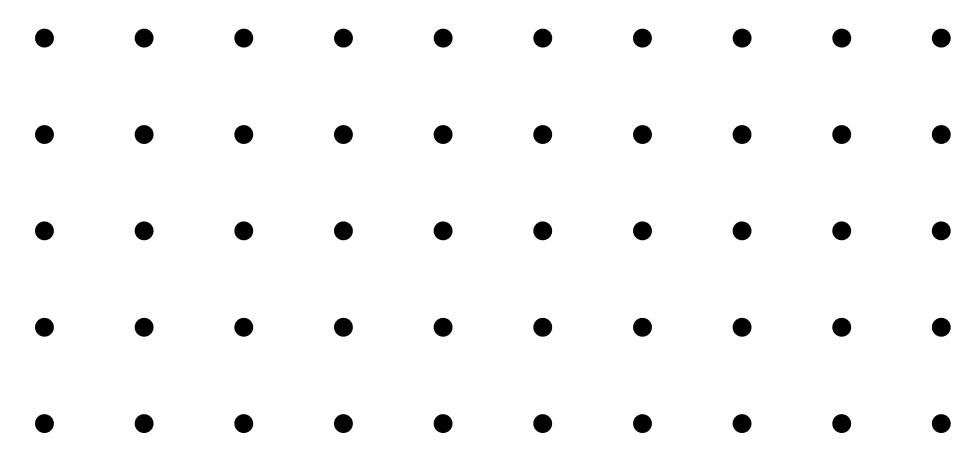
Balls and bins

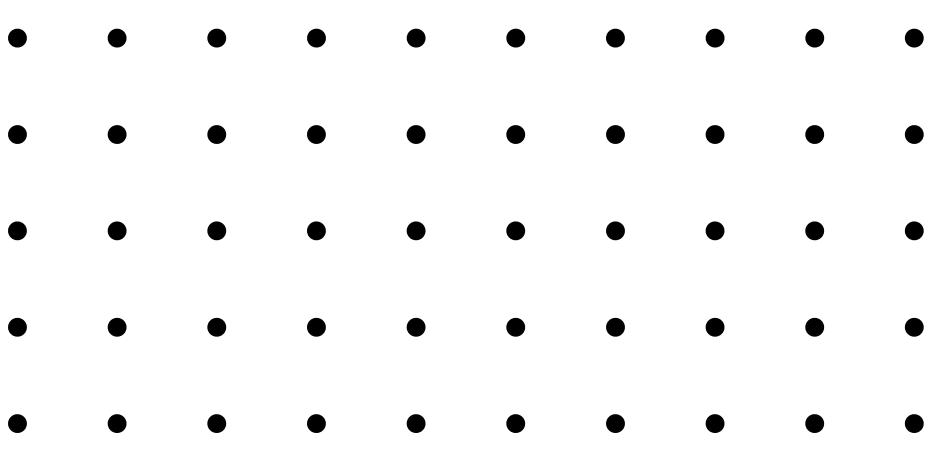
- * É um modelo probabilístico onde n bolas são distribuídas aleatoriamente em (m) caixas (bins).
- * Ele analisa questões como o número máximo de bolas em uma caixa, a probabilidade de uma caixa estar vazia ou quantas bolas cada caixa recebe.
- * Amplamente utilizado para estudar balanceamento de carga e alocação em sistemas computacionais. A distribuição das bolas é geralmente assumida como uniforme e independente.

Formulação inteira

01 Balls and Bins

Determinar quantas bolas serão depositadas em cada recipiente, de forma a maximizar o lucro total, enquanto respeitando as restrições de quantidade mínima e máxima de cada recipiente.

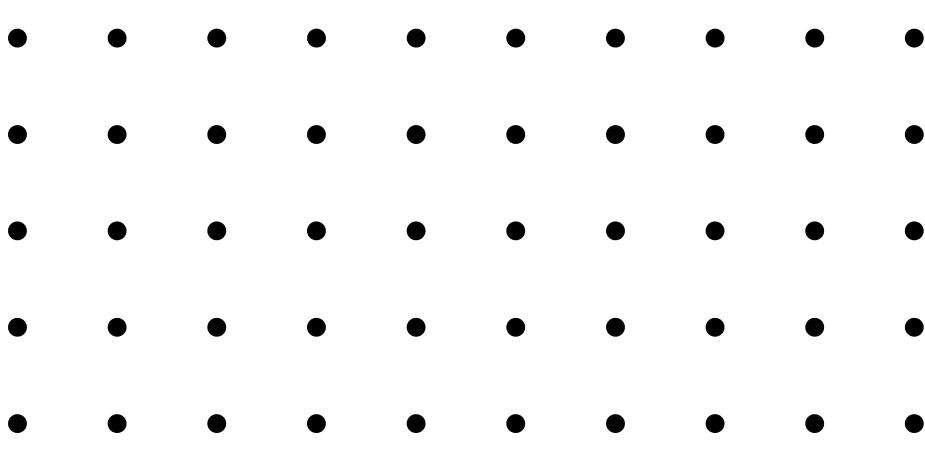




Balls and Bins

1.1 Variáveis

- l_i : Lower bound (limite inferior) do recipiente i .
- u_i : Upper bound (limite superior) do recipiente i .
- x_i : Número de bolas no recipiente i , $\forall i \in [n]$
- y_{ik} : Indica se o recipiente i contém k bolas, onde $l_i \leq k \leq u_i$, $\forall i \in [n]$
- m : Total de bolas disponíveis.
- $L_k = \frac{k(k+1)}{2}$: Lucro gerado por um recipiente com exatamente k bolas.



Balls and Bins

1.2 Função Objetivo

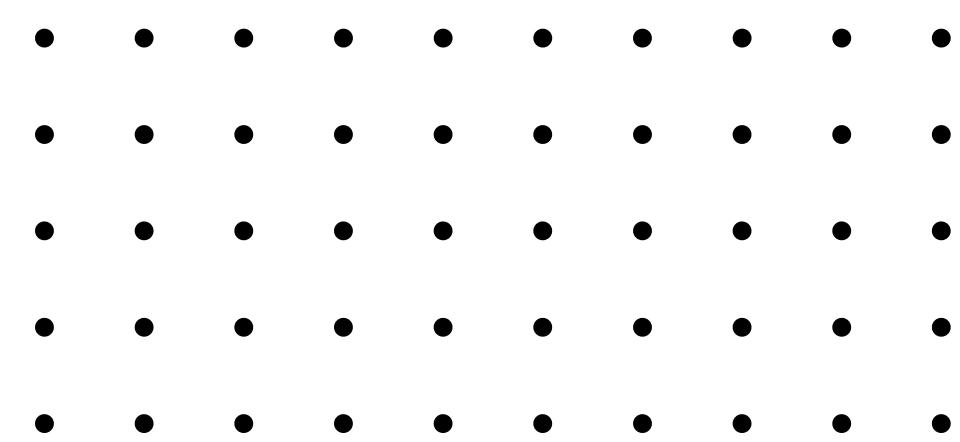
Queremos maximizar o lucro dos recipientes:

$$\max \sum_{i=1}^n \sum_{k=l_i}^{u_i} L_k \cdot y_{ik}$$

Onde,

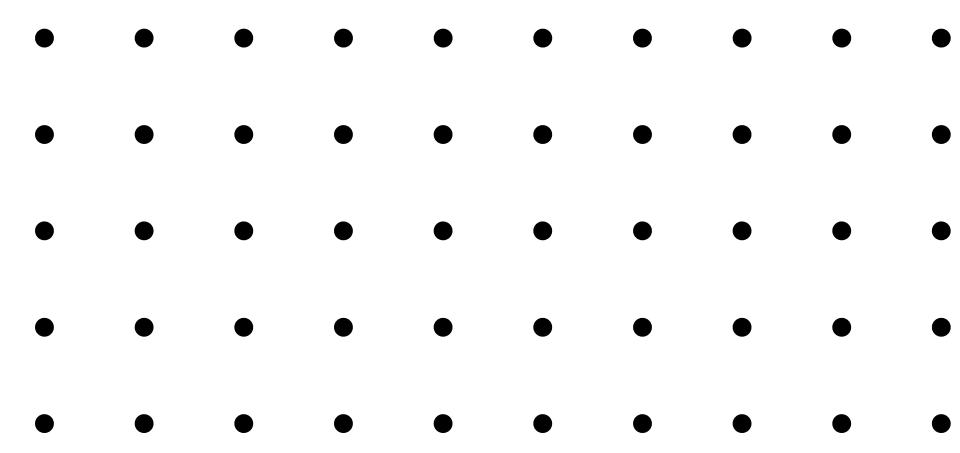
- $L_k = \frac{k(k+1)}{2}$: Lucro gerado por um recipiente com exatamente k bolas.
- y_{ik} : Indica se o recipiente i contém k bolas, onde $l_i \leq k \leq u_i$, $\forall i \in [n]$

Balls and Bins



Restrição para garantir que todas as bolas foram usadas

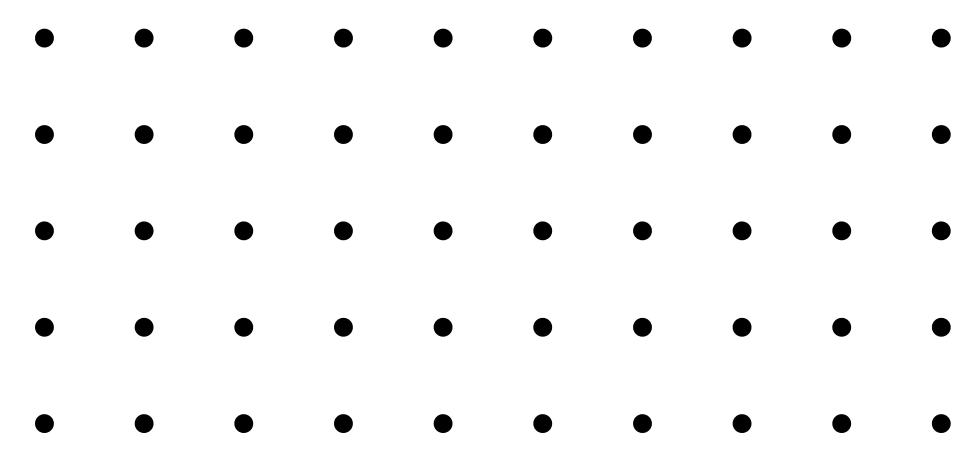
$$\sum_{i=1}^n x_i = m$$



Balls and Bins

Restrição de escolha única por recipiente: Cada recipiente deve conter exatamente uma quantidade k de bolas dentro de seu intervalo

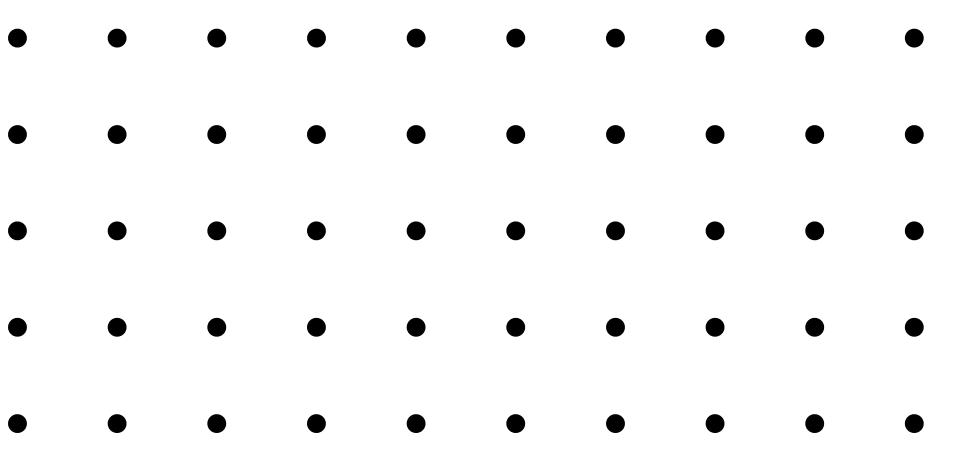
$$\sum_{k=l_i}^{u_i} y_{ik} = 1, \quad \forall i \in [n]$$



Balls and Bins

Definição de X_i dado Y_{ik} : Cada recipiente i deve conter um número válido de bolas

$$x_i = \sum_{k=l_i}^{u_i} k \cdot y_{ik}, \quad \forall i \in [n]$$



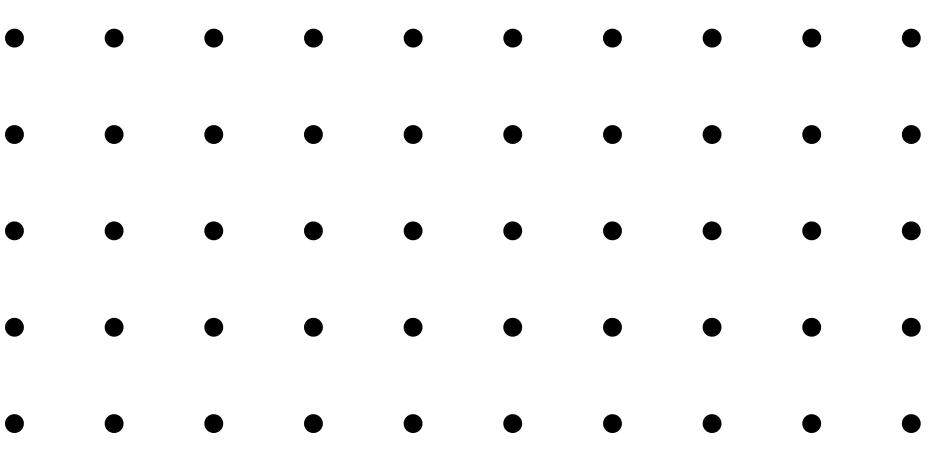
Balls and Bins

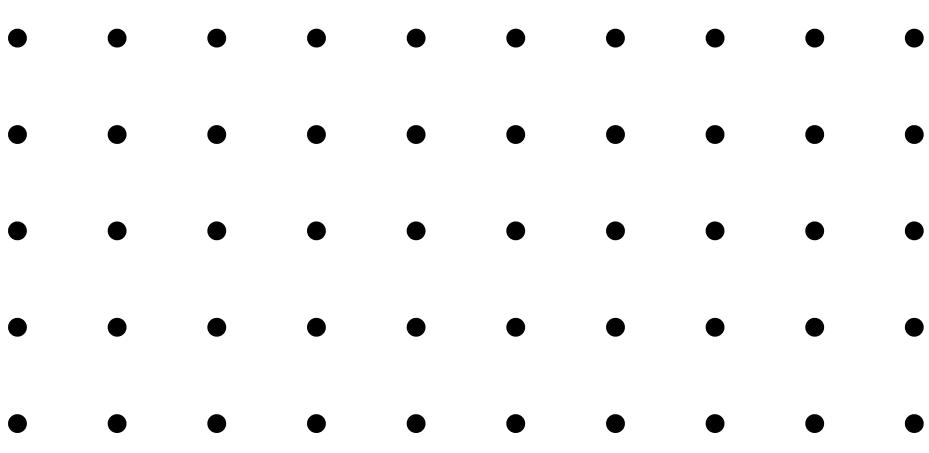
Domínio das variáveis

$$y_{ik} \in \mathbf{B}, \quad x_i \in \mathbb{Z}^+, \quad \forall i \in [n], \forall k \in \{l_i, \dots, u_i\}$$

Balls and Bins

Formulação em Julia





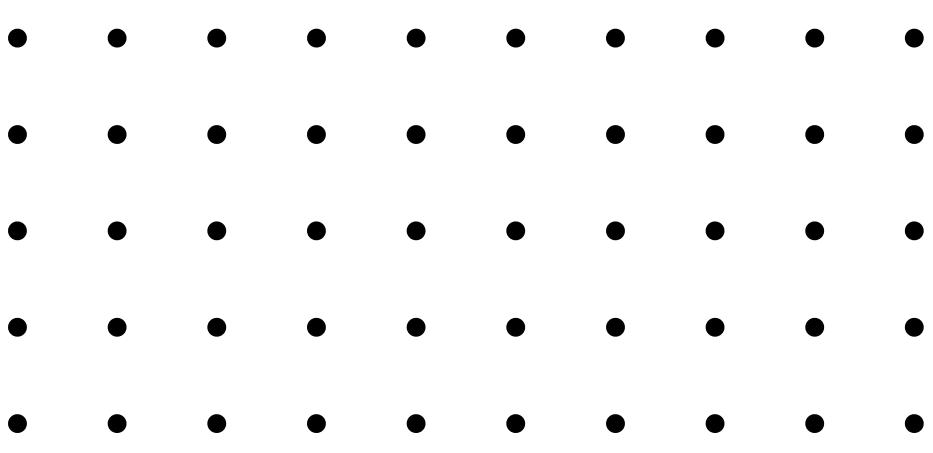
Balls and Bins

Variáveis

x_i : Número de bolas no recipiente i , $\forall i \in [n]$

y_{ik} : Indica se o recipiente i contém k bolas, onde $l_i \leq k \leq u_i$, $\forall i \in [n]$

```
# Xi e Yik
@variable(model, x[1:n], Int, lower_bound=0)
@variable(model, y[1:n, 1:maximum(limites_superiores)], Bin)
```

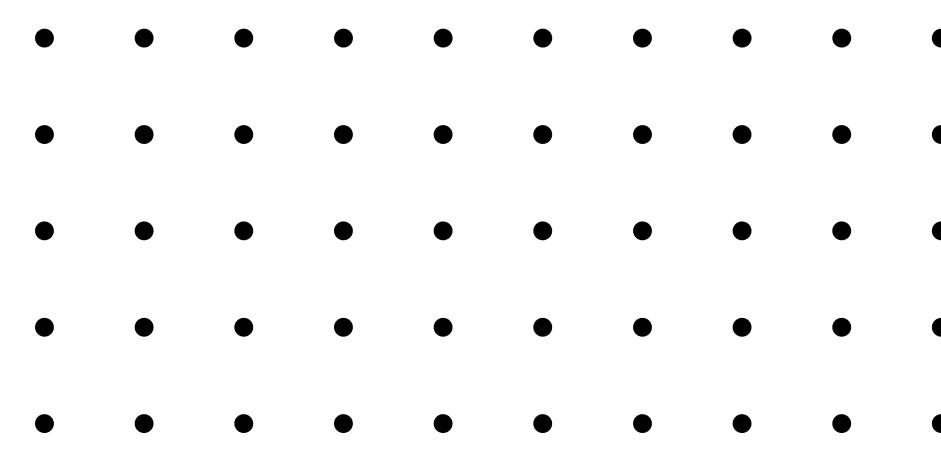


Balls and Bins

Função objetivo

$$\max \sum_{i=1}^n \sum_{k=l_i}^{u_i} L_k \cdot y_{ik}$$

```
# função objetivo
@objective(model, Max,
    sum([(k * (k + 1)) / 2] * y[i, k]
        for i in 1:n
            for k in limites_inferiores[i]:limites_superiores[i]))
```



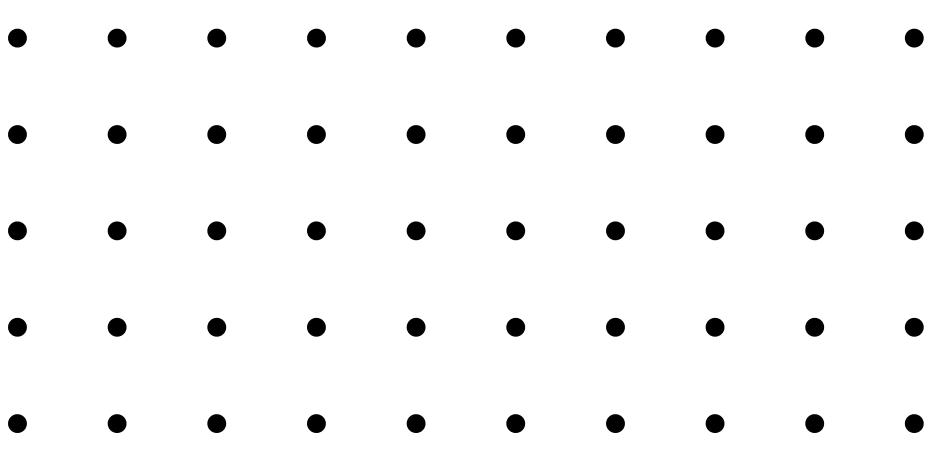
Balls and Bins

$$\sum_{i=1}^n x_i = m$$

```
# garantir o uso de todas as bolas  
@constraint(model, sum(x[i] for i in 1:n) == m)
```

$$\sum_{k=l_i}^{u_i} y_{ik} = 1, \quad \forall i \in [n]$$

```
# garantir que apenas uma variável Yik seja ativada para cada recipiente i  
for i in 1:n  
    @constraint(model, sum(y[i, k] for k in limites_inferiores[i]:limites_superiores[i]) == 1)  
end
```



Balls and Bins

Definição de X_i dado Y_{ik} : Cada recipiente i deve conter um número válido de bolas

$$x_i = \sum_{k=l_i}^{u_i} k \cdot y_{ik}, \quad \forall i \in [n]$$

```
# definição de Xi dado Yik
for i in 1:n
    @constraint(model, x[i] == sum(k * y[i, k] for k in limites_inferiores[i]:limites_superiores[i]))
end
```

Balls and Bins

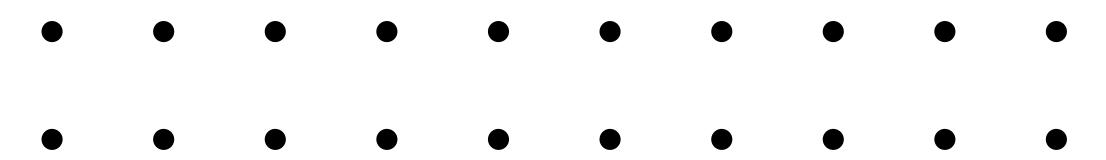
Alguns Resultados

05.txt					
Seed	Limite	Tempo (s)	Tempo Formulação	Valor Formulação	Limite Superior
3	5		4.13	8201961	
4	5		4.12	8201965	
5	5		4.1	8201915	
6	5		4.17	8201975	
7	5		TIME LIMIT	-inf	8202061
3	300		4.19	8201961	
4	300		4.13	8201965	
5	300		4.17	8201915	
6	300		4.17	8201975	
7	300		5.22	8201884	

Balls and Bins

Alguns Resultados

07.txt					
Seed	Limite Tempo (s)	Tempo Formulação	Valor Formulação	Limite Superior	
3	5	TIME_LIMIT	-inf	inf	
4	5	TIME_LIMIT	-inf	inf	
5	5	TIME_LIMIT	-inf	inf	
6	5	TIME_LIMIT	-inf	inf	
7	5	TIME_LIMIT	-inf	inf	
3	300	77.62	206913418		
4	300	73.77	206913418		
5	300	75.54	206913418		
6	300	75.05	206913418		
7	300	80.99	206913407		

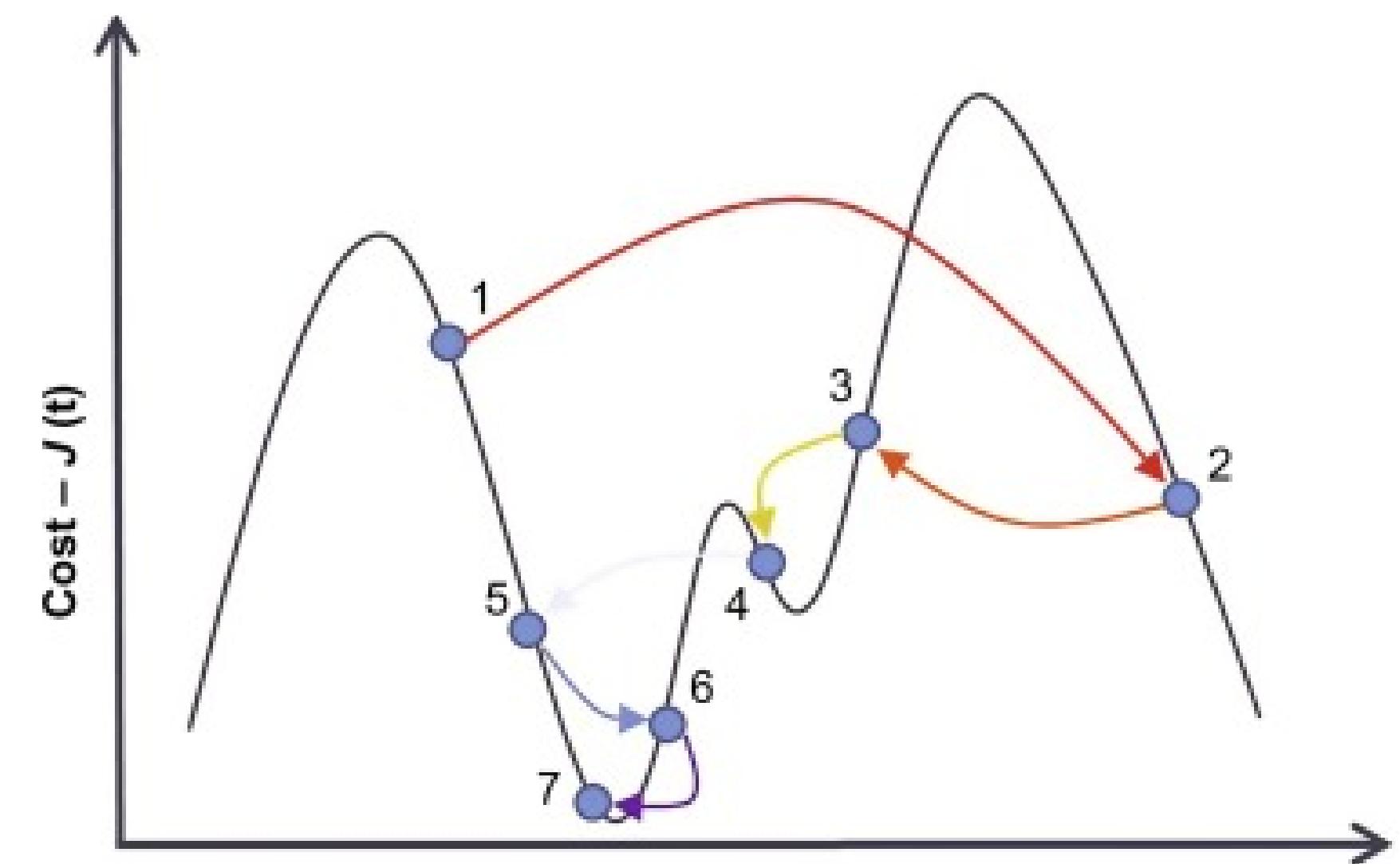


Meta heurística

Simulated annealing

É um algoritmo heurístico de otimização inspirado no processo de recocimento metalúrgico. Ele explora soluções candidatas de forma aleatória, aceitando ocasionalmente soluções piores para escapar de mínimos locais.

A probabilidade de aceitar piores soluções diminui ao longo do tempo, conforme a "temperatura" do sistema diminui. É usado em problemas de otimização combinatória e contínua, como roteamento e design de circuitos.



Meta-heurística

Linguagem

Implementada em Python, devido a maior facilidade e familiaridade dos integrantes com a mesma

Uso

Inserir da linha de comando:
python simulated_annealing.py <arquivo_entrada>
<seed> <max_iterações> <max_time>

Resposta

Após a execução do algoritmo para um dada instância, é retornado um log da seguinte maneira:

```
Dados da instância: .\instancias\09.txt
Número de bins: 400
Número de balls: 67676

Resultados:
Nome da instância: .\instancias\09.txt
Semente de aleatoriedade: 3
Valor da solução inicial: 8297000.0
Melhor valor da solução encontrada: 8297875.0
Tempo total de execução: 5.02s
Tempo limite superior: 5s
Temperatura final: 18.480456
Iterações restantes: 19832
```

Meta-heurística

* Instances é uma lista com tuplas referente a cada bin, onde a posição 0 é o LowerBound e a posição 1 é o UpperBound.

* Setamos a seed passada pela linha de comando.

* Geramos uma solução inicial válida e boa o suficiente para a instância do problema fornecida.

```
def simulated_annealing(bins, balls, instances, seed, max_iterations, max_time):  
    random.seed(seed)  
    start_time = time.time()  
  
    current_solution = create_initial_solution(bins, balls, instances)  
    best_solution = current_solution[:]  
    best_value = evaluate_solution(current_solution, instances)  
  
    temperature = 100.0  
    min_temperature = 0.000001  
    cooling_rate = 0.99  
  
    total_time = 0
```

create_initial_solution

* Se

* G
sufi

- Para cada bin adiciona primeiramente o até o **Lowerbound** dela. Da bin 0 até a n-ésima.
- Posteriormente, adiciona até o **Upperbound**, dela enquanto houver bolas disponíveis. Da bin 0 até a n-ésima.
- Caso o número de bolas disponíveis seja menor do que o **UpperBound** da bin atual, simplesmente adiciona o restante das bolas nessa bin.

```
def create_initial_solution(bins, balls, instances):  
    initial_solution = [0] * bins  
    for i in range(bins):  
        initial_solution[i] = instances[i][0]  
        balls -= instances[i][0]  
  
    index = 0  
    while balls > 0:  
        max_add = instances[index][1] - instances[index][0]  
        if(max_add > balls):  
            initial_solution[index] += balls  
            balls -= balls  
        else:  
            initial_solution[index] += max_add  
            balls -= max_add  
        index += 1  
    return initial_solution
```

Meta-heurística

* Instances é uma lista com tuplas referente a cada bin, onde a posição 0 é o LowerBound e a posição 1 é o UpperBound.

* Setamos a seed passada pela linha de comando.

* Geramos uma solução inicial válida e boa o suficiente para a instância do problema fornecida através da função create_initial_solution.

* Avaliamos a solução atual pegando o valor/custo dela através da função evaluate_solution

```
def simulated_annealing(bins, balls, instances, seed, max_iterations, max_time):  
    random.seed(seed)  
    start_time = time.time()  
  
    current_solution = create_initial_solution(bins, balls, instances)  
    best_solution = current_solution[:]  
    best_value = evaluate_solution(current_solution, instances)  
  
    temperature = 100.0  
    min_temperature = 0.000001  
    cooling_rate = 0.99  
  
    total_time = 0
```

evaluate_solution

- Para cada bin verifica se a quantidade de bolas presente nela respeita os limites do LowerBound e UpperBound
- Se alguma bin não satisfazer os limites retorna 0 e a solução não será considerada posteriormente
- Se os limites forem respeitados aplica a função $(n * (n + 1)) / 2$ para cada bin

```
def evaluate_solution(solution, instances):  
    best_value = 0  
  
    for index, ball in enumerate(solution):  
        if(ball > instances[index][1] or ball < instances[index][0]):  
            return 0  
        else:  
            best_value += (ball*(ball + 1))/ 2  
  
    return best_value
```

Meta-heurística

- * Instances é uma lista com tuplas referente a cada bin, onde a posição 0 é o LowerBound e a posição 1 é o UpperBound.
- * Setamos a seed passada pela linha de comando.
- * Geramos uma solução inicial válida e boa o suficiente para a instância do problema fornecida através da função create_initial_solution.
- * Avaliamos a solução atual pegando o valor/custo dela através da função evaluate_solution
- * Definimos a temperatura inicial como 100, o menor que ela pode chegar, nesse caso 0.000001 e o colling_rate de 0.99. Valores obtidos de maneira empírica.

```
def simulated_annealing(bins, balls, instances, seed, max_iterations, max_time):  
    random.seed(seed)  
    start_time = time.time()  
  
    current_solution = create_initial_solution(bins, balls, instances)  
    best_solution = current_solution[:]  
    best_value = evaluate_solution(current_solution, instances)  
  
    temperature = 100.0  
    min_temperature = 0.000001  
    cooling_rate = 0.99  
  
    total_time = 0
```

Meta-heurística

- * Será aplicado o simulated annealing enquanto a temperatura atual for maior que a temperatuda mínima definida e enquanto as retrições de tempo máximo e número de iterações forem respeitadas
- * Durante 200 iterações aplicamos o algoritmo de Metropolis. Número obtido empíricamente
- * Pegamos um vizinho da solução atual e se ele tiver um lucro/custo válido podemos pegar ele ou não.

```
while temperature > min_temperature and max_iterations != 0:  
    total_time = time.time() - start_time  
    if total_time >= max_time:  
        break  
    for _ in range(200):  
        neighbor = get_neighbor(current_solution[:], instances)  
        neighbor_value = evaluate_solution(neighbor, instances)  
        current_value = evaluate_solution(current_solution, instances)  
  
        delta = current_value - neighbor_value  
  
        if neighbor_value != 0:  
            if delta <= 0 or random.random() < math.exp(-delta / temperature):  
                current_solution = neighbor[:]  
        if current_value >= best_value:  
            best_solution = current_solution[:]  
            best_value = current_value  
  
        temperature *= cooling_rate  
        max_iterations -= 1
```

get_neighbor

*

Se

temp

míni

máx

;

- Iremos tirar 5 bolas de um bin

* Du
Met
aleatória e inserir essas 5 bolas em
outra bin também aleatória

* Pe
tiver
não.

- Fazemos um validação simples para
não tirar de uma bin que tenha balls =
lowerbound ou adicionar em uma bin
que tenha balls = upperbound.

- Não garante a geração de vizinhos
válidos

```
def get_neighbor(solution, instances):
    from_bin = random.randint(0, len(solution) - 1)
    to_bin = random.randint(0, len(solution) - 1)

    while solution[from_bin] == instances[from_bin][0]:
        from_bin = random.randint(0, len(solution) - 1)

    while solution[to_bin] == instances[to_bin][1]:
        to_bin = random.randint(0, len(solution) - 1)

    solution[from_bin] -= 5
    solution[to_bin] += 5

    return solution
```

Meta-heurística

- * Será aplicado o simulated annealing enquanto a temperatura atual for maior que a temperatuda mínima definida e enquanto as retrições de tempo máximo e número de iterações forem respeitadas
- * Durante 200 iterações aplicamos o algoritmo de Metropolis. Número obtido empíricamente
- * Pegamos um vizinho da solução atual e se ele tiver um lucro/custo válido podemos pegar ele ou não.
- * Atualiza-se a temperatura com base no cooling_rate

```
while temperature > min_temperature and max_iterations != 0:  
    total_time = time.time() - start_time  
    if total_time >= max_time:  
        break  
    for _ in range(200):  
        neighbor = get_neighbor(current_solution[:], instances)  
        neighbor_value = evaluate_solution(neighbor, instances)  
        current_value = evaluate_solution(current_solution, instances)  
  
        delta = current_value - neighbor_value  
  
        if neighbor_value != 0:  
            if delta <= 0 or random.random() < math.exp(-delta / temperature):  
                current_solution = neighbor[:]  
        if current_value >= best_value:  
            best_solution = current_solution[:]  
            best_value = current_value  
  
        temperature *= cooling_rate  
        max_iterations -= 1
```

Obrigado