Tópicos de Análise de Algoritmos

Algoritmos de clustering

Sec 4.5, 4,6 e 4.7 do KT

Clustering

U: conjunto de n itens $\{p_1, \ldots, p_n\}$.

$d(p_i, p_j)$: distância entre p_i e p_j.

- $d(p_i, p_i) = 0$ para $i = 1, \ldots, n$
- $d(p_i, p_j) = d(p_j, p_i) > 0$ para $i \neq j$

Clustering

U: conjunto de n itens $\{p_1, \ldots, p_n\}$.

$d(p_i, p_j)$: distância entre p_i e p_j.

- $d(p_i, p_i) = 0$ para $i = 1, \ldots, n$
- $d(p_i, p_j) = d(p_j, p_i) > 0$ para $i \neq j$

k: número de grupos.

k-clustering: partição de U em k partes.

Clustering

U: conjunto de n itens $\{p_1, \ldots, p_n\}$.

$d(p_i, p_j)$: distância entre p_i e p_j.

- $d(p_i, p_i) = 0$ para $i = 1, \ldots, n$
- $d(p_i, p_j) = d(p_j, p_i) > 0$ para $i \neq j$

k: número de grupos.

k-clustering: partição de U em k partes.

espaçamento de k-clustering: distância mínima
entre dois elementos de partes distintas do clustering.

Clustering

U: conjunto de n itens $\{p_1, \ldots, p_n\}$.

$d(p_i, p_j)$: distância entre p_i e p_j.

- $d(p_i, p_i) = 0$ para $i = 1, \ldots, n$
- $d(p_i, p_j) = d(p_j, p_i) > 0$ para $i \neq j$

k: número de grupos.

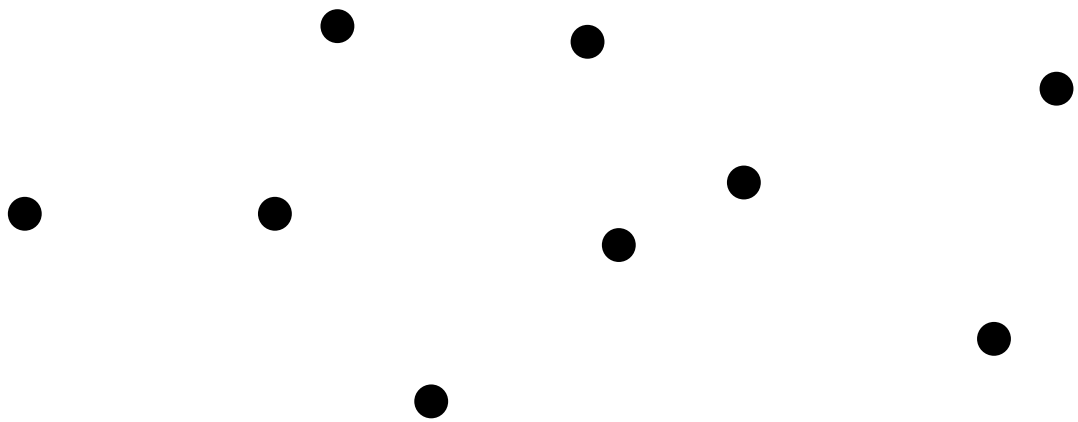
k-clustering: partição de U em k partes.

espaçamento de k-clustering: distância mínima
entre dois elementos de partes distintas do clustering.

Problema: Dados U, d e k,
encontrar um k-clustering com espaçamento máximo.

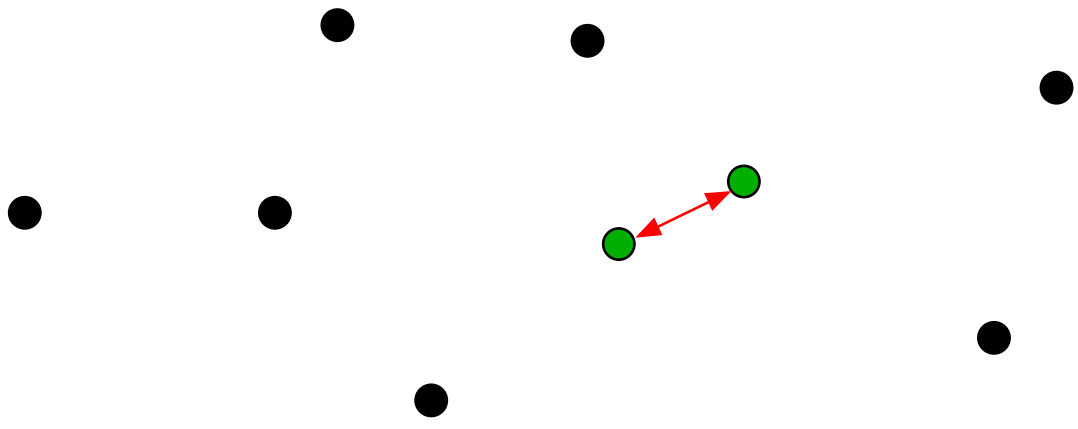
Clustering

Problema: Dados U, d e k,
encontrar um k-clustering com espaçamento máximo.



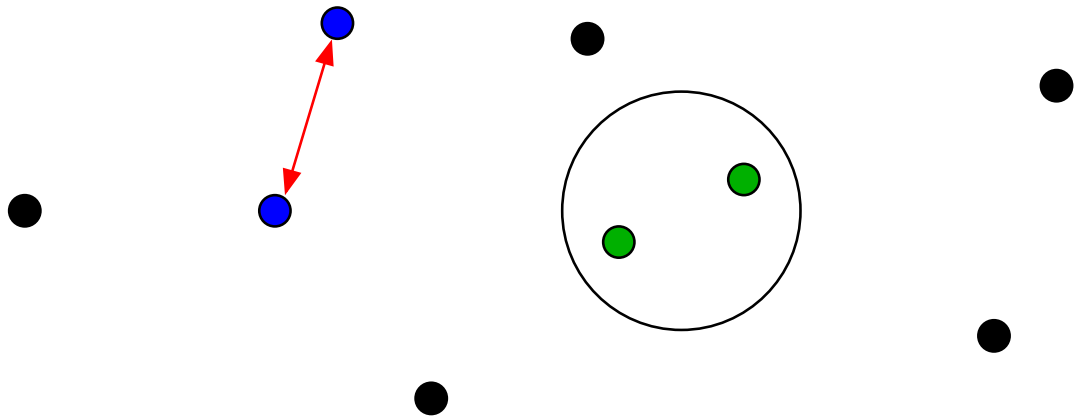
Clustering

Problema: Dados U, d e k,
encontrar um k-clustering com espaçamento máximo.



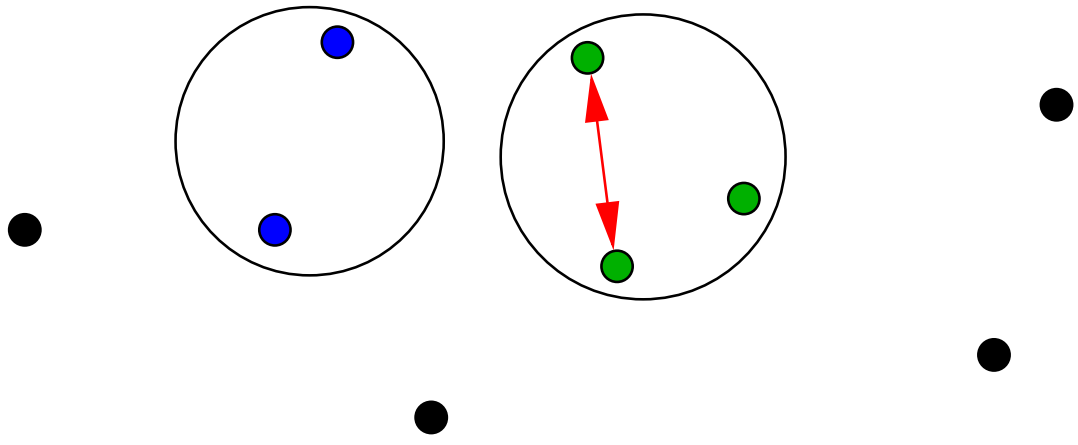
Clustering

Problema: Dados U, d e k,
encontrar um k-clustering com espaçamento máximo.



Clustering

Problema: Dados U, d e k,
encontrar um k-clustering com espaçamento máximo.



Algoritmo inspirado no Kruskal

Lembram do algoritmo de Kruskal?

Problema: Dado um grafo $G = (V, E)$ conexo,
e um custo $c_e > 0$ para cada aresta e em E,
encontrar uma árvore geradora mínima.

Árvore geradora mínima: árvore geradora de custo mínimo.

Algoritmo inspirado no Kruskal

Lembram do algoritmo de Kruskal?

Problema: Dado um grafo $G = (V, E)$ conexo,
e um custo $c_e > 0$ para cada aresta e em E,
encontrar uma árvore geradora mínima.

Árvore geradora mínima: árvore geradora de custo mínimo.

```
MST-KRUSKAL (V, E, c)
1   e_1, ..., e_m ← ORDENE(E, c)          ▷ pelo valor de c_e
2   T ← ∅        i ← 0
3   enquanto G' = (V, T) não é conexo faça
4         i ← i + 1
5         se T ∪ {e_i} não tem circuito
6               então T ← T ∪ {e_i}
7   devolva T
```

Algoritmo inspirado no Kruskal

Como adaptar MST-KRUSKAL para resolver o k-clustering?

Algoritmo inspirado no Kruskal

Como adaptar MST-KRUSKAL para resolver o k-clustering?

Acrescentamos arestas até termos k componentes.

Algoritmo inspirado no Kruskal

Como adaptar MST-KRUSKAL para resolver o k-clustering?

Acrescentamos arestas até termos k componentes.

CLUSTERING-KRUSKAL (V, E, c, k)

```
1  e_1, ..., e_m ← ORDENE(E, c)          ▷ pelo valor de c_e
2  F ← ∅        i ← 0
3  enquanto G' = (V, F) tem > k componentes faça
4        i ← i + 1
5        se F ∪ {e_i} não tem circuito
6              então F ← F ∪ {e_i}
7  devolva F
```

Algoritmo inspirado no Kruskal

Como adaptar MST-KRUSKAL para resolver o k-clustering?

Acrescentamos arestas até termos k componentes.

CLUSTERING-KRUSKAL (V, E, c, k)

```
1  e_1, ..., e_m ← ORDENE(E, c)          ▷ pelo valor de c_e
2  F ← ∅        i ← 0
3  enquanto G' = (V, F) tem > k componentes faça
4        i ← i + 1
5        se F ∪ {e_i} não tem circuito
6              então F ← F ∪ {e_i}
7  devolva F
```

Este algoritmo resolve o problema?

Análise da correção

CLUSTERING-KRUSKAL (V, E, c, k)

```
1   e_1, ..., e_m ← ORDENE(E, c)          ▷ pelo valor de c_e
2   F ← ∅        i ← 0
3   enquanto G' = (V, F) tem ≥ k componentes faça
4         i ← i + 1
5         se F ∪ {e_i} não tem circuito
6               então F ← F ∪ {e_i}
7   devolva as componentes de F
```

Qual é o espaçamento do clustering devolvido?

Análise da correção

CLUSTERING-KRUSKAL (V, E, c, k)

```
1   e_1, ..., e_m ← ORDENE(E, c)          ▷ pelo valor de c_e
2   F ← ∅        i ← 0
3   enquanto G' = (V, F) tem ≥ k componentes faça
4         i ← i + 1
5         se F ∪ {e_i} não tem circuito
6               então F ← F ∪ {e_i}
7   devolva as componentes de F
```

Qual é o espaçamento do clustering devolvido?

É o custo da próxima aresta que seria incluída em F.

Análise da correção

CLUSTERING-KRUSKAL (V, E, c, k)

```
1  e_1, ..., e_m ← ORDENE(E, c)          ▷ pelo valor de c_e
2  F ← ∅        i ← 0
3  enquanto G' = (V, F) tem ≥ k componentes faça
4        i ← i + 1
5        se F ∪ {e_i} não tem circuito
6              então F ← F ∪ {e_i}
7  devolva as componentes de F
```

Qual é o espaçamento do clustering devolvido?

É o custo da próxima aresta que seria incluída em F.

A próxima aresta que não forma circuito com F.

Análise da correção

Seja e a próxima aresta que seria incluída em F.

Análise da correção

Seja e a próxima aresta que seria incluída em F.

Seja $\mathcal{C} = \{C_1, \ldots, C_k\}$ o k-clustering devolvido,
e $\mathcal{C}^* = \{C_1^*, \ldots, C_k^*\}$ um k-clustering ótimo.

Análise da correção

Seja e a próxima aresta que seria incluída em F.

Seja $\mathcal{C} = \{C_1, \ldots, C_k\}$ o k-clustering devolvido,
e $\mathcal{C}^* = \{C_1^*, \ldots, C_k^*\}$ um k-clustering ótimo.

O espaçamento de $\mathcal{C}$ é c_e.

Basta mostrar que o espaçamento de $\mathcal{C}^*$ é no máximo c_e.

Análise da correção

Seja e a próxima aresta que seria incluída em F.

Seja $\mathcal{C} = \{C_1, \ldots, C_k\}$ o k-clustering devolvido,
e $\mathcal{C}^* = \{C_1^*, \ldots, C_k^*\}$ um k-clustering ótimo.

O espaçamento de $\mathcal{C}$ é c_e.

Basta mostrar que o espaçamento de $\mathcal{C}^*$ é no máximo c_e.

Ou $\mathcal{C} = \mathcal{C}^*$, ou algum C_i não está contido em nenhum C_j^*.

Análise da correção

Seja e a próxima aresta que seria incluída em F.

Seja $\mathcal{C} = \{C_1, \ldots, C_k\}$ o k-clustering devolvido,
e $\mathcal{C}^* = \{C_1^*, \ldots, C_k^*\}$ um k-clustering ótimo.

O espaçamento de $\mathcal{C}$ é c_e.

Basta mostrar que o espaçamento de $\mathcal{C}^*$ é no máximo c_e.

Ou $\mathcal{C} = \mathcal{C}^*$, ou algum C_i não está contido em nenhum C_j^*.

Seja j tal que $C_i \cap C_j^* \neq \emptyset$.

Análise da correção

Seja e a próxima aresta que seria incluída em F.

Seja $\mathcal{C} = \{C_1, \ldots, C_k\}$ o k-clustering devolvido,
e $\mathcal{C}^* = \{C_1^*, \ldots, C_k^*\}$ um k-clustering ótimo.

O espaçamento de $\mathcal{C}$ é c_e.

Basta mostrar que o espaçamento de $\mathcal{C}^*$ é no máximo c_e.

Ou $\mathcal{C} = \mathcal{C}^*$, ou algum C_i não está contido em nenhum C_j^*.

Seja j tal que $C_i \cap C_j^* \neq \emptyset$.

Seja f aresta de F em C_i com um único extremo em C_j^*.

Análise da correção

Seja e a próxima aresta que seria incluída em F.

Seja $\mathcal{C} = \{C_1, \ldots, C_k\}$ o k-clustering devolvido,
e $\mathcal{C}^* = \{C_1^*, \ldots, C_k^*\}$ um k-clustering ótimo.

O espaçamento de $\mathcal{C}$ é c_e.

Basta mostrar que o espaçamento de $\mathcal{C}^*$ é no máximo c_e.

Ou $\mathcal{C} = \mathcal{C}^*$, ou algum C_i não está contido em nenhum C_j^*.

Seja j tal que $C_i \cap C_j^* \neq \emptyset$.

Seja f aresta de F em C_i com um único extremo em C_j^*.

Claro que $c_f \leq c_e$ e espaçamento de $\mathcal{C}^*$ é no máximo c_f. □

Implementação

Como se implementa este algoritmo?

Implementação

Como se implementa este algoritmo?

Como se implementa o algoritmo de Kruskal?

Implementação

Como se implementa este algoritmo?

Como se implementa o algoritmo de Kruskal?

Estrutura de dados para conjuntos disjuntos...

Implementação

Como se implementa este algoritmo?

Como se implementa o algoritmo de Kruskal?

Estrutura de dados para conjuntos disjuntos...

makeset(x)
1. $pai[x] \leftarrow x$
2. $rank[x] \leftarrow 0$ ▷ heurística dos tamanhos

findset(x)
1. **se** $x \neq pai[x]$
2. **então** $pai[x] \leftarrow$findset($pai[x]$) ▷ compressão de caminhos
3. **devolva** $pai[x]$

Union-Find

makeset(x)
1. $pai[x] \leftarrow x$
2. $rank[x] \leftarrow 0$

union(x, y)
1. $x' \leftarrow$findset(x)
2. $y' \leftarrow$findset(y)
3. **se** $x' \neq y'$
4. **então** link(x', y')

findset(x)
1. **se** $x \neq pai[x]$
2. **então** $pai[x] \leftarrow$findset($pai[x]$)
3. **devolva** $pai[x]$

link(x, y)
1. **se** $rank[x] > rank[y]$
2. **então** $pai[y] \leftarrow x$
3. **senão** $pai[x] \leftarrow y$
4. **se** $rank[x] = rank[y]$
5. **então** $rank[y] \leftarrow rank[y] + 1$

Union-Find

makeset(x)
1. $pai[x] \leftarrow x$
2. $rank[x] \leftarrow 0$

findset(x)
1. **se** $x \neq pai[x]$
2. **então** $pai[x] \leftarrow$findset($pai[x]$)
3. **devolva** $pai[x]$

union(x, y)
1. $x' \leftarrow$findset(x)
2. $y' \leftarrow$findset(y)
3. **se** $x' \neq y'$
4. **então** link(x', y')

link(x, y)
1. **se** $rank[x] > rank[y]$
2. **então** $pai[y] \leftarrow x$
3. **senão** $pai[x] \leftarrow y$
4. **se** $rank[x] = rank[y]$
5. **então** $rank[y] \leftarrow rank[y] + 1$

Altura da árvore enraizada em x: comprimento de um caminho mais longo de x até um descendente de x na árvore.

Note que a altura da árvore enraizada em x é no máximo $rank[x]$.

Union-Find

makeset(x)
1. $pai[x] \leftarrow x$
2. $rank[x] \leftarrow 0$

findset(x)
1. **se** $x \neq pai[x]$
2. **então** $pai[x] \leftarrow$findset($pai[x]$)
3. **devolva** $pai[x]$

union(x, y)
1. $x' \leftarrow$findset(x)
2. $y' \leftarrow$findset(y)
3. **se** $x' \neq y'$
4. **então** link(x', y')

link(x, y)
1. **se** $rank[x] > rank[y]$
2. **então** $pai[y] \leftarrow x$
3. **senão** $pai[x] \leftarrow y$
4. **se** $rank[x] = rank[y]$
5. **então** $rank[y] \leftarrow rank[y] + 1$

Altura da árvore enraizada em x: comprimento de um caminho mais longo de x até um descendente de x na árvore.

Note que a altura da árvore enraizada em x é no máximo $rank[x]$.

Sempre que pendura uma subárvore em y, se for de mesma altura, aumenta o valor de $rank[y]$.

Union-Find

makeset(x)
1. $pai[x] \leftarrow x$
2. $rank[x] \leftarrow 0$

findset(x)
1. **se** $x \neq pai[x]$
2. **então** $pai[x] \leftarrow$findset($pai[x]$)
3. **devolva** $pai[x]$

union(x, y)
1. $x' \leftarrow$findset(x)
2. $y' \leftarrow$findset(y)
3. **se** $x' \neq y'$
4. **então** link(x', y')

link(x, y)
1. **se** $rank[x] > rank[y]$
2. **então** $pai[y] \leftarrow x$
3. **senão** $pai[x] \leftarrow y$
4. **se** $rank[x] = rank[y]$
5. **então** $rank[y] \leftarrow rank[y] + 1$

Altura da árvore enraizada em x: comprimento de um caminho mais longo de x até um descendente de x na árvore.

Note que a altura da árvore enraizada em x é no máximo $rank[x]$.

Exercício: Mostre que se x é uma raiz e $rank[x] = r$, então a árvore enraizada em x tem pelo menos 2^r elementos.

Union-Find: análise de pior caso

```
makeset(x)
1. pai[x] ← x
2. rank[x] ← 0
```

```
findset(x)
1. se x ≠ pai[x]
2.       então pai[x] ←findset(pai[x])
3. devolva pai[x]
```

```
union(x, y)
1. x′ ←findset(x)
2. y′ ←findset(y)
3. se x′ ≠ y′
4.       então link(x′, y′)
```

```
link(x, y)
1. se rank[x] > rank[y]
2.       então pai[y] ← x
3.       senão pai[x] ← y
4.             se rank[x] = rank[y]
5.                   então rank[y] ← rank[y] + 1
```

Exercício: Mostre que se x é uma raiz e $rank[x] = r$, então a árvore enraizada em x tem pelo menos 2^r elementos.

Conclua que, se n é o número de elementos nas florestas, então cada operação custa no pior caso $\mathrm{O}(\lg n)$.

Union-Find: análise amortizada

makeset(x)
1. $pai[x] \leftarrow x$
2. $rank[x] \leftarrow 0$

findset(x)
1. **se** $x \neq pai[x]$
2. **então** $pai[x] \leftarrow$findset($pai[x]$)
3. **devolva** $pai[x]$

union(x, y)
1. $x' \leftarrow$findset(x)
2. $y' \leftarrow$findset(y)
3. **se** $x' \neq y'$
4. **então** link(x', y')

link(x, y)
1. **se** $rank[x] > rank[y]$
2. **então** $pai[y] \leftarrow x$
3. **senão** $pai[x] \leftarrow y$
4. **se** $rank[x] = rank[y]$
5. **então** $rank[y] \leftarrow rank[y] + 1$

Próximas aulas: análise amortizada.

Union-Find: análise amortizada

```
makeset(x)
1. pai[x] ← x
2. rank[x] ← 0
```

```
findset(x)
1. se x ≠ pai[x]
2.       então pai[x] ←findset(pai[x])
3. devolva pai[x]
```

```
union(x, y)
1. x′ ←findset(x)
2. y′ ←findset(y)
3. se x′ ≠ y′
4.       então link(x′, y′)
```

```
link(x, y)
1. se rank[x] > rank[y]
2.       então pai[y] ← x
3.       senão pai[x] ← y
4.             se rank[x] = rank[y]
5.                   então rank[y] ← rank[y] + 1
```

Próximas aulas: análise amortizada.

Vamos mostrar que
cada operação acima tem custo amortizado de $\mathrm{O}(\lg^* n)$,
onde $\lg^* n$ é o número de vezes que temos que aplicar
o lg até atingir um número menor ou igual a 1.