Laboratório VISGRAF
Instituto de Matemática Pura e Aplicada

Um Framework para Escolha de Modelos em Grandes Bases de Dados

Leandro Cruz
Luiz Velho (supervisor)

Technical Report TR-17-04 Relatório Técnico

April - 2017 - Abril

The contents of this report are the sole responsibility of the authors.
O conteúdo do presente relatório é de única responsabilidade dos autores.

Um Framework para Escolha de Modelos em Grandes Bases de Dados

Leandro Cruz

January 6, 2017

Contents

Chapter 1

Introdução

Neste trabalho, apresentaremos um framework que pode ser utilizado no problema de escolher um conjunto de modelos em uma ampla base de dados. Existem muitas técnicas, em diversas áreas, que baseiam-se na combinação de informações provenientes de um conjunto de modelos previamente escolhidos. Tais modelos podem ser fotografias, músicas, modelos 3D, texturas, exemplares de terrenos, etc.

Mais especificamente, partimos de um amplo conjunto de um determinado tipo de modelos, o qual estamos chamando de base de dados e, entre todos esses elementos, escolheremos um pequeno subconjunto satisfazendo certas restrições. Apresentaremos diversos métodos que podem ser usados para esse propósito. Cada método satisfará um determinado conjunto de restrições. Essas restrições consistem em critérios de escolha satisfazendo propriedades relacionadas com as seguintes três medidas:

- funções de *scores*: uma função que determina uma medida para cada exemplar
- $\#\omega$: quantidade de elementos escolhidos
- $C(\omega)$: medida de correlação entre os elementos escolhidos

No Capítulo 2, apresentaremos os critérios de escolhas supracitados. Primeiramente, apresentaremos a abordagem mais simples: escolheremos os modelos que otimizem as funções scores. Dado um conjunto de funções scores, seus conjuntos de

argumentos podem conter mais de um elemento, e portanto, existem muitas soluções possíveis para o problema de escolha. Logo, em seguida, discutiremos como obter soluções com cardinalidade mínima. Em certos casos, pode ser mais interessante priorizarmos a redução da quantidade de modelos escolhidos, mesmo que para isso precisamos flexibilizar a otimalidade das funções scores. Nessa direção, apresentaremos um método que vai gradualmente aplicando essa flexibilização até atingir a meta da cardinalidade. Finalmente, também apresentaremos um método adequado aos problemas em possa ser necessário definir uma função que messa uma relação entre os modelos escolhidos.

Como previamente mencionado, a escolha de modelos é um passo preliminar em problemas de diversas áreas. No Capítulo 3 apresentaremos um estudo de caso mostrando como utilizar cada método apresentado para resolver o problema de criação de um mosaico de imagens. Esse problema é simples o suficiente para não confundir o leitor, mas suporta hipóteses extras que são necessárias para a utlização dos nossos métodos.

Além desse estudo de caso, no Capítulo 4, apresentaremos como podemos usar nossa abordagem para resolver de escolha de modelos como um passo preliminar em vários problemas distintos. Esses problemas surgem em áreas como processamento de imagens, modelagem geométrica, música computacional, entre outras.

Chapter 2

Critérios Para Escolha de Modelos

Ao longo desse capítulo mostraremos algumas técnicas para a escolha de modelos em uma ampla base de dados. Denotaremos por Ω o nosso conjunto de modelos disponíveis (a base de dados), e Ω^* o conjunto das partes desse conjunto, ou seja, todos os possíveis subconjuntos com os modelos disponíveis. Dessa forma, as abordagens descritas adiantes consistem em métodos para escolher $\omega \in \Omega^*$ satisfazendo um determinado conjunto de critérios baseados nas seguintes três medidas:

- Funções *Scores*: funções $f : \Omega \longrightarrow \mathbb{R}_+$ que quantificam um modelo dado
- $\#\omega$: quantidade de modelos escolhidos
- Medida de correlação: uma função $C : \Omega^* \longrightarrow \mathbb{R}_+$ que estabelece de correlação entre os elementos escolhidos

Um modelo pode ser pensado abstratamente como uma função $e : \mathbb{R}^m \longrightarrow \mathbb{R}^n$. Por exemplo, uma imagem digital é uma função $e : A \longrightarrow \{0, ..., 255\}^3$, onde $A \subset \mathbb{Z} \times \mathbb{Z}$ é um reticulado retangular finito. Analogamente, podemos representar um terreno como um Modelo de Elevação Digital (em inglês *Digital Elevation Model* - DEM) como $e : A \longrightarrow \mathbb{R}$ (é possível substituir o contradomínio por $\mathbb{Z}$ quando a unidade de medida é suficientemente pequena). Também podemos representar uma música como $e : I \subset \mathbb{R} \longrightarrow \mathbb{R}$. Nesses três casos, também podemos representar esses modelos como matrizes ou vetores.

Modelos 3D possuem podem possuir um conjunto de funções que o descrevem (um atlas de uma parametrização), ou ser representado como o zero de uma função (descrição implícita). Nesse caso, a construção da função score pode ser um pouco mais complicada, embora não contradiga nenhuma restrição relacionada com os nossos métodos.

No que diz respeito às funções scores, a única coisa que precisamos é sermos capazes de usá-las para quantificar cada modelo. Ou seja, elas serão funções capazes de ler um modelo dado e quantificá-lo de acordo com características específicas da aplicação. Por exemplo, podemos definir uma função score como a distância de uma imagem dada para uma outra imagem previamente fixada. Ou, o quanto um modelo se aproxima de uma dada estrutura de controle.

Descreveremos nas próximas seções diversos critérios para a escolha de um conjunto de modelos. Adotaremos uma abordagem construtiva, ou seja, partiremos do método mais simples, e iremos adicionar novas restrições, e mostraremos quais são as diferenças em relação ao que tinhamos anteriormente. Começaremos com .....

2.1 Otimizando as Funções Scores

O primeiro critério consiste em escolher os modelos que otimizem um conjunto de funções de score previamente definidas. Esse critério é muito básico, mas servirá como base para compreensão e solução dos demais critérios.

Problema 1: *Dadas* $f_j : \Omega \longrightarrow \mathbb{R}_+$, *com* $j = 1, ..., N$ *obter* $\omega = \{e_1, ..., e_K\} \in \Omega^*$, *tal que,* $\forall j \ \exists e \in \omega$ *onde* $e \in \underset{e\in\Omega}{\operatorname{argmax}} f_j(e)$.

Primeiramente, definiremos os conjuntos dos argumentos de cada função score da seguinte forma:

$$A^j = \underset{e\in\Omega}{\operatorname{argmax}} f_j(e) = \{\bar{e} \in \Omega; f_j(\bar{e}) \geq f_j(e), \forall e \in \Omega\}, \ \texttt{com} \ j = 1, ..., N \qquad (2.1)$$

Tais conjuntos são não vazios sempre que $\Omega \neq \emptyset$ (pois, como Ω é finito e f restrito a Ω é limitada, o máximo sempre será atingido por algum elemento). Além disso,

temos que $\#\omega \leq N$ (é possível que $\#\omega < N$, pois alguns modelos podem pertencer a mais de um conjunto solução A^j).

Vale destacar que o conjunto A^j pode conter mais de um elemento, e portanto existe mais de uma escolha possível para solução desse problema. Nas próximas seções, discutiremos sobre algumas estratégias para realizar tal escolha.

Além disso, em geral Ω é um conjunto sem uma estrutura particular que permita encontrar $e \in \underset{e \in \Omega}{\operatorname{argmax}} f_j(e)$ de maneira eficiente. Nesses casos, podemos adotar uma abordagem por força bruta, ou seja, para cada função score precisamos avaliar os modelos pertencentes a Ω.

A medida que $\#\Omega$ cresce, se torna muito custoso avaliar todos os modelos para calcular os conjuntos A^j. Dessa forma, é interessante pensar em estratégias que estimem os argumentos de uma função score sem avaliar todos os elementos do espaço. Contudo, deixaremos esse problema para um trabalho futuro, e portanto não entraremos em detalhes sobre esse tópico no momento.

2.2 Minimizando a Cardinalidade da Escolha entre os Argumentos

Nessa seção, apresentaremos uma abordagem para escolher os elementos de A^j minimizando a quantidade de modelos escolhidos. Podemos enunciar esse problema da seguinte forma:

Problema 2: *Dadas $f_j : \Omega \longrightarrow \mathbb{R}_+$, com $j = 1, ..., N$, obter $\omega = \{e_1, ..., e_K\} \in \Omega^*$ com cardinalidade mínima, tal que, $\forall j \ \exists e \in \omega$ onde $e \in \underset{e \in \Omega}{\operatorname{argmax}} f_j(e)$.*

Introduziremos a seguir uma solução para esse problema baseada na enumeração das soluções do *Problema de Soma de Subconjunto (Subset Sum Problem - SSP)*, descrito no Apêndice A. O enunciado de tal problema é:

SSP: *Dados um conjunto de números inteiros positivos $\tau = \{x_1, ..., x_n\}$ e um número inteiro $T > 0$, obter os subconjuntos de τ com cardinalidade mínima cuja soma de seus elementos é igual a T.*

Esse problema pode ser resolvido em duas etapas usando os Algoritmos 8 e 9, que

apresentam uma abordagem *dividir-para-conquistar* recursiva, usando Programação Dinâmica. Maiores detalhes podem ser encontrados no Apêndice A.

Para resolvermos o *Problema 2*, restringiremos nosso espaço aos modelos que são argumentos de alguma função score, ou seja, ao conjunto

$$A = \bigcup_{j=1}^{N} A^j \tag{2.2}$$

Para efeitos de algoritmo, consideraremos que o conjunto A é um array (e portanto nos referiremos ao i-ésimo elemento de A por $A[i]$). Também suporemos que não há elementos repetidos em A. A partir desse conjunto, definiremos o conjunto $\tau = \{x_1, ..., x_{\#A}\}$, cujos elementos $x_i = (x_{i,1}, ..., x_{i,N})$ são vetores de 0s e 1s, de modo que:

$$x_{i,j} = \begin{cases} 1, & A\left[i\right] \in A^j \\ 0, & \text{caso contrário} \end{cases} \tag{2.3}$$

Dessa forma, $x_{i,j}$ determina se o modelo $A\left[i\right]$ otimiza a função score f_j. Neste caso, nossa meta será obter um conjunto $\mu = \{\lambda_1, ..., \lambda_K\} \subset \{1, ..., \#A\}$, satisfazendo que, para cada $j = 1, ..., N$ existe $\lambda_i \in \mu$ o qual $x_{\lambda_i,j} = 1$. A partir desses elementos, podemos fazer uma analogia com o problema *SSP* para obter o conjunto ω. Finalmente, a solução do *Problema 2* será $\omega = \{A\left[\lambda_1\right], ..., A\left[\lambda_K\right]\}$.

A primeira diferença da abordagem que estamos apresentando para o *Problema 2* em relação ao que foi apresentado para o SSP é que agora, para alcançar o objetivo, substituiremos a soma de números por uma operação entre vetores (com dimensão N) de valores lógicos (*True* ou *False*) que são inicializados com *False* e cujo objetivo é preencher de *True* em todas as coordenadas. Dessa forma, em vez de soma de números, agora realizaremos uma operação lógica *OR* entre elementos desse tipo, a qual está definida como uma operação *OR* elemento a elemento. Ou seja, sejam $x = (x_1, ..., x_N)$ e $y = (y_1, ..., y_N)$, então $x \vee y = (x_1\ OR\ y_1, ..., x_N\ OR\ y_N)$.

Analogamente ao descrito no Apêndice A para o problema *SSP*, vale observar que a solução do problema de escolher elementos de $\{e_i, e_{i+1}, e_{i+2}, ..., e_{\#A}\} \subset A$ de modo a alcançar um objetivo T (problema que denotaremos por (T, i)) pode ser obtida a partir das soluções de dois problemas menores: (i) alcançar o objetivo $T \vee \tau\left[i\right]$ com

os elementos $\{e_{i+1}, e_{i+2}, ..., e_{\#A}\}$ (denotada por $(T \vee \tau[i], i+1)$), e (ii) alcançar T com $\{e_{i+1}, e_{i+2}, ..., e_{\#A}\}$ (denotada por $(T, i+1)$).

Observe que, se $\{e_{\alpha_1}, ..., e_{\alpha_m}\}$ é solução de $(T \vee \tau[i], i+1)$, então $\{e_i, e_{\alpha_1}, ..., e_{\alpha_m}\}$ é solução de (T, i). Além disso, se $\{e_{\beta_1}, ..., e_{\beta_m}\}$ é uma solução de $(T, i+1)$, então, também será uma solução de (T, i). Logo, as soluções de (T, i) é a união das soluções de $(T \vee \tau[i], i+1)$ acrescidas de e_i, com as soluções de $(T, i+1)$. Tal fato nos leva a uma abordagem dividir-para-conquistar recursiva, a qual denotaremos por $(T, i) = (T \vee \tau[i], i+1) \wedge (T, i+1)$, onde $\wedge$ significa a união das soluções de ambos problemas conforme descrito acima.

Essa abordagem previamente descrita nos permite pensar em um algoritmo recursivo no qual cada ramificação da execução testa uma possibilidade de um conjunto $\omega \subset \Omega$ ser uma solução do problema em questão (analogamente ao feito com a enumeração das soluções do SSP). Dado que nosso objetivo é encontrar apenas as soluções de cardinalidade mínima, sempre que encontrarmos uma solução precisamos comparar sua cardinalidade com a cardinalidade das soluções. Quando essa nova solução tiver cardinalidade maior então deve ser descartada. Quando igual, deve ser adicionada as soluções. E quando menor, as soluções previamente envontradas devem ser descartadas e em seguida essa nova solução é adicionada ao conjunto das soluções.

Uma implementação direta desse algoritmo, como adaptação do problema SSP é muito custosa. Mesmo uma abordagem com programação dinâmica, que armazena o resultado do processamento das recursões filhas de um par (T, i), temos que manter duas tabelas: uma que armazena a informação se haverá alguma solução a partir das ramificações das recursões (a qual chamaremos de *prev*), e outra que armazena os elementos que deverão ser adicionados a solução a partir do respectivo passo da recursão. Essa segunda tabela (a qual chamaremos de *tail*) consome muita memória, e mantê-la consistente é razoavelmente custoso.

De uma maneira mais detalhada, *prev* é uma tabela hash indexada por um par de inteiros, a qual cada elemento é um valor lógico (*True* ou *False*). Analogamente, *tail* também é uma tabela hash indexada por um par de inteiros, mas cada elemento é um conjunto de conjuntos de indices de modelos. Cada conjunto de indices do

Algorithm 1 Minimal Cardinality

```
1: procedure SOL-MIN(τ, T, i, ω, C, M)
2:     if (i >= #τ or #ω > M) then
3:         prev[key(T)][i] = 0
4:         return False
5:     else if (isKey(prev, key(T), i)) then
6:         if (prev[key(T)][i] == 1) then
7:             M = min{M, #ω + prev[key(T)][i]}
8:             return True
9:         end if
10:        return False
11:    else if (isFull(T)) then
12:        M = min{M, #ω}
13:        prev[key(T)][i] = 1
14:        return True
15:    end if
16:    isChanged = False
17:    if (sol-min(τ, T ∨ τ[i], i + 1, ω ∪ {i}, C, M) then
18:        prev[S][i] = prev[key(T ∨ τ[i])][i + 1] + 1
19:        isChanged = True
20:    end if
21:    if (sol-min(τ, T, i + 1, ω, C, M) then
22:        if (isChanged == True and prev[key(T)][i] > 0) then
23:            prev[key(T)][i] = min{prev[key(T)][i], prev[key(T)][i + 1]}
24:        else
25:            prev[key(T)][i] = prev[key(T)][i + 1]
26:        end if
27:        isChanged = True
28:    end if
29:    if (not isChanged) then
30:        prev[key(T)][i] = 0
31:    end if
32:    return isChanged
33: end procedure
```

elemento de *tail* associado a etapa (T, i) é composto pelos elementos adicionados a solução em uma ramificação da recursão que obteve uma solução válida.

O Algoritmo 1 introduz a solução do primeiro passo, enquanto o Algoritmo 2 determina as soluções de cardinalidade mínima. Ambos algoritmos recebem como parâmetros o conjunto de elementos disponíveis (τ), um vetor T com N elementos lógicos inicializado com *False*, o índice i do elemento de τ que será aprocessado, o conjunto $\omega \in \tau$ dos elementos previamente escolhidos (esse conjunto é inicializado vazio), o conjunto C das soluções e a cardinalidade mínima M (inicializada como $M = \infty$).

Vale destacar que muito processamento feito para manter a tabela *tail* consistente é feita com conjuntos que não tem cardinalidade mínima (enquanto o método ainda não sabe qual é essa cardinalidade mínima). Dessa forma, uma alternativa que evita tal processamento é abordar o problema em dois passos: (i) determinar qual é a cardinalidade mínima das soluções e (ii) enumerar as soluções de cardinalidade mínima.

Essa abordagem constrói a tabela *prev* no problema de determinar o tamanho mínimo, que é mais simples e mais rápido, e a reaproveita no problema de enumeração das soluções. Além disso, adiciona ao problema de enumeração a condição de descartar conjuntos com cardinalidade maior do que o mínimo, que é uma eficiente condição de parada, e evita adicionar ao conjunto solução, subconjuntos de τ que serão descartados a posteriori.

Quando $i >= \#\tau$ temos que essa ramificação das rercursões chegou ao fim e não encontrou nenhuma solução. Além disso, se $\#\omega > M$ significa que o conjunto dos elementos previamente escolhidos já tem cardinalidade maior do que a mínima (ou maior do que a cardinalidade da solução mínima encontrada até o momento no Algoritmo 1). Nesses casos, em que a recursão não chegou a uma solução, o elemento de *prev* associado a (T, i) é marcado com zero, para destacar que essa ramificação não leva a uma solução.

A função *isKey* determina se a tabela hash passada no primeiro parâmetro contém como chave o par de números passados nos dois seguintes parâmetros. Além disso, como, T é um vetor binário (em vez de um número, como no caso do *SSP*) precisamos convertê-lo em um número para utiliza-lo como um indexador das tabelas *prev* e *tail*. Essa conversão pode ser feita com seguinte função:

$$key(T) = \sum_{i=0}^{nm} 2^i T[i] \tag{2.4}$$

A função *isFull* determina se todos os elementos de um vetor são *True*. Quando essa função retorna verdadeito para T, isso significa que os elementos de ω na respectiva etapa é uma solução do problema. Uma vez que essa função retorna verdadeiro, interrompemos as ramificações da recursão para evitar soluções com redundância, pois se ω é uma solução, $\omega \cup \{x\}$, para qualquer $x \in \tau$, também é uma solução. No caso do Algoritmo 2, quando percebemos que *isFull(T)* é verdadeiro então adicionamos ω ao conjunto das soluções C e interrompemos a recursão.

O Algoritmo 1 retorna um valor booleano, associado se a recursão encontrou uma solução. Esse retorno é necessário para atualizar a tabela *prev* dos passos de recursão anteriores. Na execução do Algoritmo 2 podemos interromper rapidamente recursões que não chegarão a uma solução.

Vale destacar que podemos fazer algumas otimizações nesses algoritmos. A primeira, e óbvia, em cada passo de recursão, é calcular $keyT$ uma única vez e armazenar em uma variável e usar esse valor todas as vezes que forem necessárias. Mas existem outras otimizações menos óbvias.

A primeira é adicionarmos uma condição de parada equivalente a condição $S <$

Algorithm 2 Enumeração das soluções de cardinalidade mínima

```
1:  procedure MIN-CHOICE(τ, T, i, w, C, M)
2:      if (i >= #τ or #ω > M) then
3:          return
4:      else if (isKey(prev, key(T), i)) then
5:          return
6:      else if (isFull(T)) then
7:          C.append(ω)
8:          return
9:      end if
10:     min-choice(τ, T ∨ τ[i], i + 1, ω ∪ {i}, C, M)
11:     min-choice(τ, T, i + 1, ω, C, M)
12: end procedure
```

$\tau[i]$ usada nos algoritmos para resolver o SSP. Contudo existem duas diferenças grandes entre este problema e o SSP: (i) no caso do SSP a relação *ser menor do que* está bem definida pois estamos lidando com números, mas agora estamos lidando com vetores de booleanos; e (ii) essa condição só faz sentido quando os valores de τ estão ordenados (e para ordená-los precisamos primeiramente definir essa relação *ser menor do que*).

Dados dois vetores de booleanos A e B, de mesmo tamanho, dizemos que $A < B$ quando o elemento *True* de menor posição em ambos (digamos o i-ésimo elemento) ocorre em A (e $B[i] == False$). Caso contrário, $B < A$. Se a primeira ocorrência de um elemento *True* (na posição i) for tal que $A[i] == B[i] == True$, então $A < B$ se a quantidade de *True*s em A for menor do que em B (naturalmente, caso contrário, $B < A$). Se ambas quantidades forem iguais, o menor elemento será aquele que tiver um *True* na menor posição, sendo que o outro tenha um *False* na mesma posição. Com essa relação de desigualdade, podemos ordenar os modelos e construir τ, de modo que, na primeira linha esteja o menor modelo e siga crescentemente até o maior modelo na última linha.

A condição $S < \tau[i]$ significa no problema do SSP que não há outro elemento em τ a partir da i-ésima posição que ao ser somado com S alcançará o objetivo inicial (pois $S - \tau[j] < 0$, $\forall j \geq i$, enquanto uma solução só ocorre no passo seguinte ao caso em que $S - \tau[j] == 0$). No caso da escolha dos modelos, essa condição é um pouco mais complicada. Agora, precisamos determinar se os elementos de τ que possam ser usados para completar T (ou seja, os elementos nas posições de i até $\#\tau$) conseguirão preencher todos os elementos de T que são *False*. Para tal, nós contruimos uma tabela (nomeada *tauTail*), tal que, na posição i esteja armazenado um vetor (de mesmo tamanho de T) que seja a união dos elementos de τ da posição i até $\#\tau$. Dessa forma, se $T \vee tauTail[i]$ contém algum *False*, então não haverá elementos em τ que sejam suficientes para serem adicionados a ω de modo a obter uma solução.

Vale destacar que essa condição só passa a ser eficiente se os elementos de τ estiverem ordenados. Isso ocorre pois, os elementos de $tauTail$ tem a propriedade de serem todos *True* a partir de um certo índice. Se τ estiver ordenado, esse índice

assume o menor valor possível, enquanto se não estiver há uma grande chance de esse índice ser alto. Por outro lado, com a ordenação temos que T tem todos valores iguais a *True* até um certo índice, o que é maximizado com a ordenação (e não seria verdade sem a ordenação).

2.3 Escolhendo Conjuntos Mínimos entre Elementos Quasi-Ótimos

Existem alguns contextos em que podemos flexibilizar a otimalidade do *Problema 2*, ou seja, obter soluções quase ótimas para as funções scores, que nos permitam reduzir a cardinalidade do conjunto solução. Mais formalmente, podemos adaptar esse problema da seguinte forma:

Problem 3: *Dados* $\lambda_1, ..., \lambda_N \geq 0$ *e* $f_j : \Omega \longrightarrow \mathbb{R}_+$, *com* $i = 1, ..., N$, *obter* $\omega = \{e_1, ..., e_K\} \subset \Omega$ *com cardinalidade mínima, tal que,* $\forall j \; \exists e \in \omega$ *onde* $f_j(e) \geq \lambda_j$.

A solução deste problema é bastante parecida com a que foi apresentada para o *Problema 2*. Contudo, agora o conjunto $A = \bigcup_{j=1}^{N} A^j_{\lambda_j}$, onde $A^j_{\lambda_j} = \{e \in \omega; f_j(e) \geq \lambda_j\}$. Além disso, o conjunto $\tau = \{x_1, ..., x_{\#A}\}$ (construido analogamente ao previamente descrito) é tal que $x_{i,j}$ determina se $A[i] \in A^j_{\lambda_j}$.

Podemos assumir que $\lambda_j \leq \max_{e \in \Omega} f_j(e)$, pois caso contrário, $A^j_{\lambda_j}$ seria vazio. Contudo, a escolha dos valores de $\lambda_1, ..., \lambda_N$ é bastante sensível. Se eles forem grandes (muito próximos ao valor ótimo), será difícil reduzir a cardinalidade de ω. Se eles forem pequenos, conseguiremos baixar a cardinalidade da solução, mas teremos uma escolha pobre, no que tange as funções score.

Na próxima seção, apresentaremos uma abordagem para escolher os maiores valores para os λ_j tal que consigamos obter ω com cardinalidade pequena.

2.4 Condições de Parada para Escolha de Elementos

Nesta seção queremos resolver o seguinte problema:

Problem 4: *Dados* $f_j : \Omega \longrightarrow \mathbb{R}_+$ *com* $j = 1, ..., N$*, e uma função* $P : \Omega^* \longrightarrow \{0, 1\}$*, obter* $\lambda_1, ..., \lambda_N \geq 0$ *e* $\omega \subset \Omega$ *tais que* $\forall j \; \exists e \in \omega$ *onde* $f_j(e) \geq \lambda_j$*, e* $P(\omega) == 1$*.*

Para resolver esse problema inicializaremos $\lambda_j = \max_{e \in \Omega} f_j(e)$, e os decresceremos de maneira a satisfazer a condição dada por P. O Algoritmo 3 calcula as soluções para o *Problema 4*. Dessa forma, podemos escolher qualquer $\omega \in C$.

Algorithm 3 Minimal Choice

procedure MINIMAL(A)
  $B_j^0 = \phi, \forall j = 1, ..., N$
  $i = 0$
  do
    $\lambda_j^i = \max_{e \in \Omega \setminus B_j^i} f_j(e), \; \forall j = 1, ..., N$
    $A^i = \bigcup_{j=1}^{N} A^j_{\lambda_j^i}$
    $\tau = createTau(A^i)$
    $C^i = \text{omc}(A^i, \tau, [0, ..., 0], 1, \phi, C)$
    $B_j^{i+1} = B_j^i \bigcup A^j_{\lambda_j^i}, \; \forall j = 1, ..., N$
    $i = i + 1$
    $b = bestChoice(C)$
  while $P(C^i[b]) \neq 1$
  return C^{i-1}
end procedure

A função P determina um critério de parada para algoritmo. Essa função deve ser escolhida cuidadosamente para que a condição $P(C^i[b]) == 1$ seja satisfeita para algum i. A escolha de b depende da aplicação. Mais adiante, apresentaremos alguns critérios para determinar essa função.

A ideia desse algoritmo consiste em começar escolhendo os modelos que maximizam as funções scores. Em seguinda, para cada passo i, flexibilizaremos o critério de otimalidade dessas funções, o que amplia o conjunto A^i em relação a A^{i-1}, permitindo que os conjuntos soluções em C^i tenha cardinalidade não maior do que na iteração C^{i-1}.

Sejam w^i um elemento qualquer escolhido em C^i, temos que $N \geq \#\omega^0 \geq \#\omega^1 \geq$

$\#\omega^2 \geq ... \geq 1$. Contudo, mesmo que $\lambda_j^i < \lambda_j^{i-1}$ implica que $A^i \subsetneq A^{i-1}$, não podemos garantir que $\#\omega^i > \#\omega^{i+1}$, $\forall i$ (é possível construir exemplos em que $\#\omega^i = \#\omega^{i+1}$, para algum i).

Pela falta de estrutura de Ω não é possível dar boas garantias de convergência desse método. Mas, vale destacar que, pela finitude de Ω o método é finito. Mais do que isso, casos práticos testados ao longo dessa pesquisa, para quase todos i tinhamos que $\#\omega^i > \#\omega^{i+1}$ (fazendo com que o método parasse com um número de iterações não tão alto). Um trabalho futuro é aprimorarmos nossas conclusões sobre a complexidade desse método.

2.5 Escolher K Elementos

Nessa seção discutiremos sobre como escolher soluções com k elementos. Naturalmente, esperamos que todas as funções scores possuam entre os elementos da solução, algum com um valor satisfatoriamente alto. Contudo, a medida que se diminui k a tendência é que essa qualidade dos scores reduzam.

Formalmente, esse critério de parada é simples: vamos definir P tal que, $P(\omega) = 1$ se $\#\omega \geq K$ e $P(\omega) = 0$ caso contrário.

É importante destacar que no Algoritmo 3 assume que as funções scores tem distribuição semelhantes. Quando isso não for verdade, é necessário normalizá-las para evitar que exista alguma função score que seja muito prejudicada em relação as outras.

2.6 Escolher Elementos Correlacionados

Muitas aplicações podem ser significativamente beneficiadas de uma escolha de poucos elementos mas fortemente correlacionados. Nessa direção, poderíamos basear nosso problema na seguinte função de energia:

$$E(\omega) = -\alpha\#\omega + \beta C(\omega) \tag{2.5}$$

Poderíamos definir um algoritmo também baseado no SSP que aceitaria como solução apenas os conjuntos que estivessem maximizando E. A dificuldade dessa abordagem é que ela dependeria de avaliar todas as possíveis soluções para o problema, o que é proibitivo, dada a enorme quantidade de possíveis soluções (e portanto o alto custo em indentificá-las e processá-las). No método que propomos anteriormente, abordamos apenas as soluções de cardinalidade mínima (que são muito menos do que a quantidade total, e portanto manipuláveis).

Dessa forma, acreditamos que uma abordagem mais eficiente é, em vez de atacar o problema direto de maximizar $E(\omega)$, $\forall \omega \in \Omega^*$, utilizar uma técnica em dois passos: (i) primeiro determinar a cardinalidade e (ii) depois escolher a solução que maximiza a correlação. O primeiro passo pode ser feito utilizando o Algoritmo 2 com a função *bestChoice* segundo a Seção 2.5. Já o segundo, consiste em uma busca, entre as soluções encontradas no primeiro passo, pelo conjunto ω que maximiza uma certa função de correlação (C) entre os elementos.

Uma outra possibilidade é utilizar a função de energia E como a função *bestChoice* no Algoritmo 2. Dessa forma, caso a combinação entre cardinalidade e correlação não esteja satisfatória em um passo, flexibiliza-se a otimalidade das funções scores para obter mais opções de elementos e assim melhorar o resultado dessa função de energia. Contudo, vale destacar, que a função *bestChoice* pode ser chamada várias vezes ao longo da execução do Algoritmo 2, e portanto, escolhas muito complexas para correlação entre os elementos podem tornar o custo de execução proibitivo.

A escolha pela função de correlação ideal é arbitrária e depende da aplicação. Uma possibilidade é utilizar uma estrutura que estabeleça uma relação dois a dois entre todos os elementos de ω e definir a correlação como uma função dessas relações. Por exemplo, a soma entre todas as relações, a menor relação, o custo da *Maxima Spanning Tree* determinada nesse grafo, etc.

Chapter 3

Mosaico de Imagens: um estudo de caso

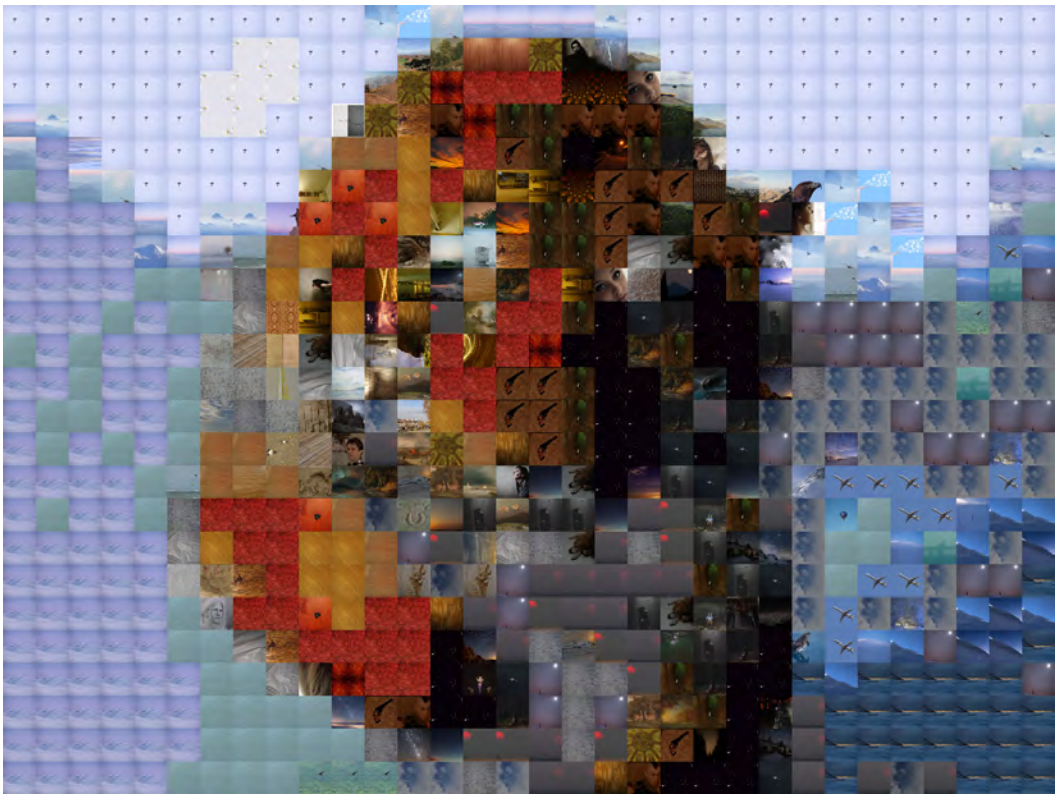
Nessa seção, apresentaremos um estudo de caso do uso de nosso framework para escolha de modelos: o mosaico de imagens. Escolhemos esse caso muito simples pois será fácil de entender a maneira como usamos cada método apresentado, e é possível adicionar hipóteses extras ao problema original o adaptando para os diferentes critérios de escolhas previamente apresentado.

O problema de mosaico de imagens consiste em dada uma imagem base, dividí-la em pequenas células (partes retangulares da imagem), para cada célula escolher uma imagem parecida, e finalmente, realizar uma colagem das imagens escolhidas de acordo com as respectivas células. A Figura 3.1 ilustra um exemplo em que as células de uma imagem base com resolução de 1600×1200 foi dividida 32×24 células de tamanho 50×50.

Ao longo desse capítulo vamos considerar que a imagem base I tem resolução $w \times h$ e foi dividida em $n \times m$ células de mesmo tamanho. Para cada par $(x, y) \in \{1, ..., n\} \times \{1, ..., m\}$ de células em I, definiremos a função score $f_{x,y} : \Omega \longrightarrow \mathbb{R}$ como a distância euclideana entre a célula (x, y) e um exemplar $s \in \Omega$, ou seja: $f_{x,y}(s) = -\sqrt{\sum_{i=0}^{n}\sum_{j=0}^{m}[I(xw+i, yh+j) - s(i,j)]^2}$. Observe que, como definimos nossa teoria

(a) Imagem base

(b) Células



(c) Mosaico

Figure 3.1: Exemplo de definição das células de uma imagem base

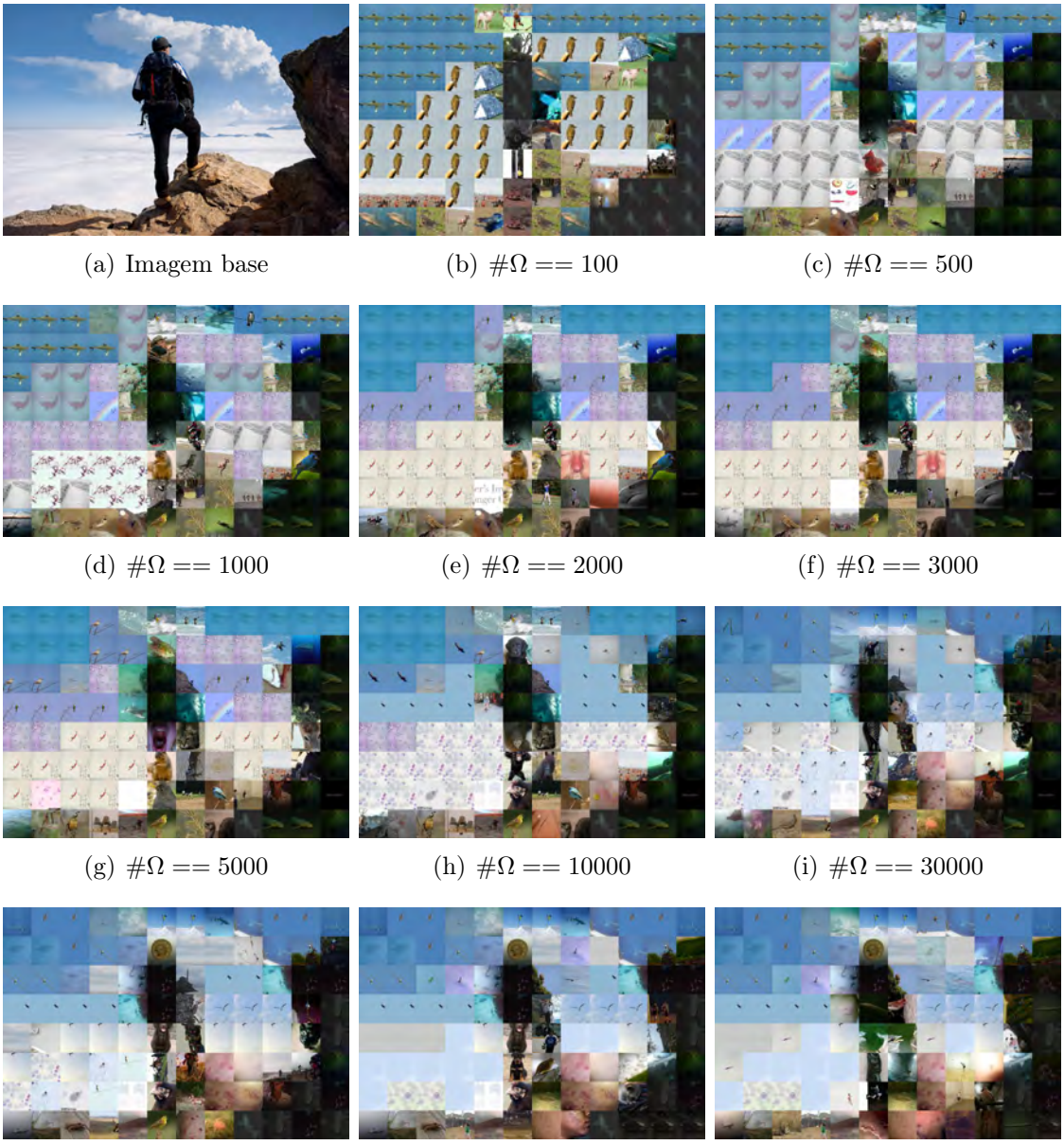
de escolha através de maximização das funções scores, e quanto menor a distância euclideana melhor será a escolha do modelo, então nossa função score é o oposto da distância.

No caso em que queremos otimizar as funções scores, criaremos os conjuntos argumentos para cada uma dessas funções (avaliando todo a base de dados e escolhendo os modelos que maximizam cada função).

Naturalmente, quanto maior o tamanho do espaço, maior a chance de encontrar uma imagem mais parecida com cada célula. Chamaremos de erro de colagem a soma dos erros entre a imagem escolhida para aquela célula e o valor original da célula. Naturalmente, quanto menor o erro, melhor o resultado. A Figura **??** ilustra onze exemplos de mosaicos criados a partir de conjuntos de imagens com tamanhos diferentes. A Tabela 3.2 mostra a relação entre o erro da colagem em relação ao tamanho do conjunto de modelos (e esse erro naturalmente cai a medida que o tamanho desse conjunto cresce).

Cardinality	Error
100	48820.2
500	38681
1000	37868.8
2000	35232.5
3000	34806.4
5000	34147.2
10000	32777.5
30000	30550.2
50000	29846
1e+05	29042.1
2e+05	28109.4

Figure 3.2: Avaliação do Erro de Colagem pelo tamanho do conjunto de modelos



(a) Imagem base (b) $\#\Omega == 100$ (c) $\#\Omega == 500$

(d) $\#\Omega == 1000$ (e) $\#\Omega == 2000$ (f) $\#\Omega == 3000$

(g) $\#\Omega == 5000$ (h) $\#\Omega == 10000$ (i) $\#\Omega == 30000$

(j) $\#\Omega == 50000$ (k) $\#\Omega == 100000$ (l) $\#\Omega == 200000$

Figure 3.3: Exemplo de definição das células de uma imagem base

A modificação da criação dos mosaicos para minimizar a quantidade de imagens distintas utilizadas não prejudica a qualidade do resultado dado que para cada célula, escolhe-se uma imagem tida como ótima ou quase-ótima (que estamos considerando que é bom o suficiente).

Ao limitar a quantidade de imagens utilizadas, se essa quantidade for muito pequena, para algumas células o modelo escolhido pode apresentar um erro perceptível. Essa restrição não é tão útil para o problema de criação de mosaicos de imagens, mas no Capítulo 4 mostraremos outros casos em que esse método é mais relevante.

Por fim, se quisermos fazer mosaicos com colagem de imagens sendo feita sem descontinuidade, podemos colocar as imagens escolhidas com uma certa sobreposição e fazer uma interpolação de cores nessa região. Neste caso, definiremos as funções de relação entre duas imagens como a soma das distâncias entre as possíveis sobreposições (dadas duas imagens A e B, temos quatro possíveis sobreposições: A a esquerda de B, B a esquerda de A, A acima de B e B acima de A).

Nesse caso, definiremos a correlação entre os conjuntos escolhidos como a média das distâncias entre as possíveis sobreposições. Dado que o custo desse exemplo de correlação é muito alto, potamos pela abordagem em dois passos descrita na Seção 2.6.

Chapter 4

Aplicações

Existem diversas aplicações em processamento de imagens, música computacional e modelagem geométrica que baseiam-se na combinação de informações provenientes de um conjunto de modelos previamente escolhidos (ω). Entre essas aplicações podemos citar: composição automática de música (CITATION), composição de imagens [1], super-resolução de imagens [2], síntese de textura [3, 4] e de terrenos [5, 6], e modelagem de objetos 3D [7].

Em muitos desses trabalhos, os modelos usados na composição do objetivo são chamados de exemplares. O termo exemplar surgiu no contexto de síntese de textura, mas hoje também é utilizado nos demais supracitados contextos de música computacional, processamento de imagens e modelagem geométrica. Nessas aplicações, tais modelos são combinados parcial ou integralmente para a construção de um modelo maior que contenha as mesmas características que os elementos de ω.

Apesar da variedade de métodos que combinam informações provenientes de um ou mais exemplares, em geral, pouco se discute sobre a metodologia adotada para a escolha desses modelos. Nesta direção, apresentaremos nessa seção, como podemos estabelecer estratégias de escolhas de modelos em uma base de dados de acordo com critérios adequados a tais aplicações.

Tais modelos (exemplares) podem ser imagens, eventualmente, com informações geométricas, tais como, imagens RGBD, que adicionam um canal com a profundidade, e imagens RGBN que utilizam um canal com a normal do modelo naquele

pixel; amostras de terrenos (a mais comum são os Modelos de Elevação Digital - DEM); objetos gráficos 3D, etc.

A principal motivação para essa pesquisa são os diversos trabalhos que criam imagens a partir da combinação de informações provenientes de uma outra imagem (ou de um conjunto dessas imagens). Nos casos em que a criação é feita a partir de uma única imagem, em geral, parece não haver muita necessidade de automatizar o processo. Por exemplo, para sintetizar uma textura a partir de um dado exemplar [3, 4]. Contudo, existem alguns casos em que a síntese é feita a partir de uma combinação de mútilplas imagens [8].

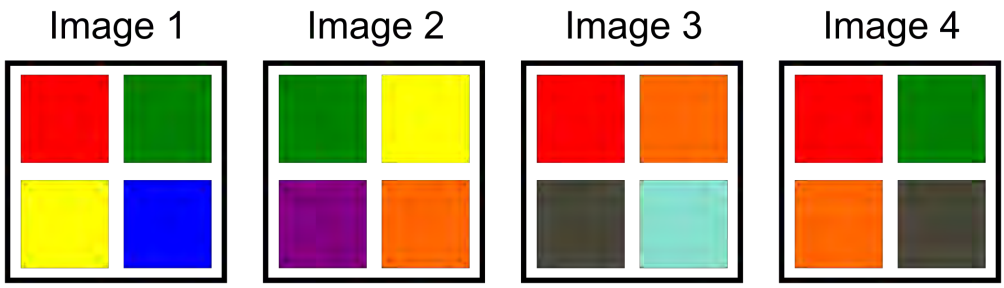
4.1 Síntese de Textura

Han et. al. [8] apresentaram uma técnica capaz de criar uma textura a partir de um grafo de exemplares. Essa estrutura além de poder conter várias imagens, também estabelece uma relação entre elas. No artigo [8], os autores não entraram em detalhes sobre como construir adequadamente tal estrutura (na prática, esse processo é feito ad hoc).

Nossos critérios de escolha se encaixam de maneira muito adequada a esse problema. A restrição da quantidade de modelos escolhidos é interessante, pois escolher muitos modelos aumenta muito o custo do processo de síntese. Além disso, a correlação entre os modelos é essencial para que a síntese funcione adequadamente.

Vamos adicionar uma hipótese (não levantada em tal trabalho) para o processo de construção do grafo: assumiremos que ele é n-partido. As imagens da primeira camada serão escolhidas arbitrariamente (dado que elas contém a macro estrutura do modelo que será sintetizado, ou seja, essa é uma suposição que oferece controle do resultado ao usuário). Para cada partição, deve se determinar quantas imagens serão escolhidas (digamos k), e qual é a diferença de escala entre cada partição. Em seguida, rodamos o método de escolha de k elementos correlacionados.

As funções scores serão definidas a partir de uma análise da partição anterior. Uma possibilidade é clusterizar as cores das imagens na partição anterior, categorizar os pixels, e em seguida, definir cada função score a partir de uma análise dessas



(a)

	red	green	yellow	dark blue	purple	orange	dark grey	light blue
Image 1	1	1	1	1	0	0	0	0
Image 2	0	1	1	0	1	1	0	0
Image 3	1	0	0	0	0	1	1	1
Image 4	1	1	0	0	0	1	1	0
Total	3	3	2	1	1	3	2	1

(b)

Figure 4.1: Exemplo de esquema de imagens escolhidas

categorias.

Um exemplo básico é assumir a cor média de cada categoria. Essa estratégia é simples, porém pode escolher modelos não tão adequados. Outra estratégia, é clusterizar cada uma dessas categorias, encontrando um conjunto de cores representativas para cada um delas. Em seguida, calcular o histograma de cada categoria (a partir das cores médias da nova clusterização) e definir a função score a partir da comparação do histograma de uma imagem da base de dados (de acordo com as respectivas cores) com a distribuição dessa categoria. Essa abordagem garante cores semelhantes, e uma distribuição da quantidade de pixels na nova imagem, semelhante a algo que existia na partição anterior (e portanto, no processo de síntese, ao sintetizar uma nova escala, as informações utilizadas será semelhante ao que tinha na escala anterior).

Além da coerência entre as escalas (dada pelas funções scores), também precisamos garantir uma coerência espacial (a cada escala sintetizada). Esse requisito é dado através da correlação entre as imagens. Nesse caso, podemos definir a correlação também utilizando a distribuição de cores previamente descrita para a construção das funções scores.

Nosso objetivo é diminuir a chance que a sintetizar de uma escala da textura produza artefatos na fronteira entre um pedaço criado por uma imagem com um pedaço criado por outra imagem. Para tal, vamos analisar o exemplo ilustrado na Figura 4.1.

Dadas quatro imagens, cada uma delas é composta por cores de algumas categorias diferentes (segundo o esquema de clusterização previamente descrito), vamos definir um descritor com N elementos (onde N é a quantidade de funções scores, ou seja, a quantidade de categorias), e descreveremos cada imagem colocando 1 quando a imagem contém uma quantidade suficiente de pixels associada a respectiva categoria e zero caso contrário. Em seguida criaremos um vetor com N números inteiros, colocando na i-ésima posição a soma da i-ésima componente de todas as imagens (linha do Total na tabela da Figura 4.1b). Nossa medida de correlação é a menor componente deste total. Dessa forma, queremos maximizar essa correlação. Ou seja, queremos maximizar a quantidade de imagens que podem ser usada para sintetizar partes relacionadas a cada categoria.

Com essa abordagem podemos construir um grafo de exemplares usado para síntese de texturas, mesmo usando como exemplares imagens que não são naturalmente utilizadas para tal efeito (e nesse caso, podemos destacar as regiões dessas imagens que poderão ser usadas).

4.2 Combinação de Imagens

Além dos algorimos de síntese de textura, outros algoritmos de síntese de imagens poderiam tirar vantagens do framework proposto nesse trabalho. Um trabalho muito importante que realiza uma substituição de parte de uma imagem com parte de outras imagens, através de um processo inteligente de composição, proposto por

Hays and Efros [1]. Nesse trabalho, dada uma imagem, os autores removiam parte dessa imagem, e substituia essa parte por um conteúdo vindo de uma outra imagem semantica e estruturalmente similar.

A escolha dessa outra imagem era feita através de uma busca no Flickr utilizando informações como grupos temáticos de imagens (por exemplo: imagens de paisagens, viagem, fotos de cidades, etc). Além dessa filtragem por metadados, os autores também utilizavam um descritor de cena [9].

Atualmente, os avanços de *Machine Learning* nos permitem utilizar enormes bases de dados de imagens e uma avaliação da probabilidade da ocorrência de um elemento (label) na imagem [10]. Dessa forma, o trabalho de associação semântica pode ser feito utilizando como funções scores a probabilidade de ocorrer nas imagens da base de dados os labels mais provável da imagem manipulada. Isso permitiria uma escolha muito mais eficiente das imagens do que a realizada usando descritores de imagens clássicos (como o GIST).

As demais partes do método, ou seja, o alinhamento das estruturas e a colagem seria feito como descrito pelos autores.

4.3 Super-Resolution

Freeman et. al. [2] apresentaram um método para realizar super-resolução de imagens a partir de exemplares previamente fornecidos. Segundo os autores: "The goal of this article is to estimate missing high-resolution detail that isn't present in the original image, and which we can't make visible by simple sharpening".

Para tal método funcionar adequadamente, é necessário estimar como magnificar a baixa resolução da imagem. Isso pode ser feito assumindo uma auto-correlação das escalas da imagem, ou seja, assumir que a alta resolução dos patches de em uma escala l será semelhante a alta resolução de patches de uma escala $l + 1$. Infelizmente, essa suposição é muito forte, e o resultado pode ser insatisfatório. Os autores obtiveram melhores resultados treinando uma rede neural com cerca de 100 mil imagens genéricas usadas para identificar a alta resolução dos patches da imagem da qual deseja-se aumentar a resolução.

Uma abordagem mais simples, mas ainda eficiente, seria escolher um conjunto de imagens relacionadas ao input, e que portanto poderia ser bem menor, contendo as informações necessárias para a superresolução. Ou seja, dada uma imagem I, para aumentar sua resolução n vezes, podemos criar uma imagem base B com a alta resolução de I. Para cada imagem da nossa base de dados (de imagens quaisquer), calcularemos reduziremos a escala n vezes, e calcularemos a alta resolução dessa nova imagem. Analogamente aos problema de Photo Collage, dividiremos o input em células b_μ, e para cada μ, dado um valor de tolerância ϵ, definiremos a função score aplicada a um modelo x da seguinte forma: $s_\mu(x)$ é a quantidade de células de x cuja distância a b_μ é menor do que ϵ. Em seguida, podemos escolher ω, analogamente ao que foi feito na Seção 5.1.2. e utilizar essas imagens para realizar a superresolução de I.

4.4 Síntese de Terrenos

Cruz et. al. [6] apresentou um método para síntese de terrenos inspirado no algoritmo *Image Quilting* [4]. Esse método contém uma etapa de especificação de estruturas de controle. A partir dessas estruturas sintetiza-se o terreno através de uma técnica patch-based (com particularidades relacionadas a geometria do modelo).

O processo de escolhas de exemplares usados para síntese pode ser feito de modo semelhante ao que foi apresentado para o problema de mosaicos, considerando uma combinação do guia e do mapa de categorias de maneira análoga a imagem base. Uma vez escolhidos os modelos, a síntese ocorre normalmente.

Chapter 5

Conclusões

Ao longo desse relatório, apresentamos um conjunto de técnicas de escolha de modelos em uma ampla base de dados. Também mostramos, como essas técnicas podem ser usadas para aprimorar etapas preliminares de várias pesquisas. Os resultados aqui apresentados são preliminares e serão aprimorados na continuidade dessa pesquisa.

Entre os principais trabalho futuros estão a estimativa do valor ótimo para uma função score sem avaliar toda a base de dados. Isso é relevante pois em muitos casos é caro avaliar um modelo pela função score e portanto uma estimativa do máximo pode permitir criar o conjunto de argumentos avaliando apenas parte da base de dados.

Outro trabalho futuro relevante é aprofundar o estudo sobre a correlação entre os elementos escolhidos. A correlação é uma importante ferramenta que pode agregar propriedades interessantes (relacionadas com a aplicação).

Finalmente, também pretendemos nos aprofundar na análise da aplicação desse método como ferramenta de outros métodos importantes conhecidos na literatura. No Capítulo 4 introduzimos como nosso framework pode ser aplicado em tais métodos, mas pretendemos apresentar futuramente testes mais consistentes. Além disso, pretendemos apresentar mais resultados em aplicações em contextos fora da manipulação de imagens.

Appendices

Appendix A

O Problema da Soma de Subconjuntos

Neste apêndice, abordaremos o *Problema de Soma de Subconjunto (Subset Sum Problem - SSP)*. Esse problema consiste em, dado um conjunto $\tau = \{x_1, ..., x_n\} \subset \mathbb{Z}$ e um objetivo T queremos saber quantos ou quais subconjuntos de τ são tais que a soma dos seus elementos vale T. Essa descrição para o problema SSP está diferente da que é enunciada por Cormen et. al. [11]. Lá, os autores estão apenas interessados em saber se existe algum subconjunto de τ cujos elementos somam T. Contudo, nós estamos mais interessados nas duas seguintes variações do problema: (i) contagem da quantidade de soluções do SSP, e (ii) enumeração das soluções.

Apresentaremos inicialmente o problema mais simples desse contexto: a contagem das soluções. Na Seção A.1, apresentaremos a abordagem *dividir-para-conquistar* que é central para a solução desse problema, e também, como podemos usar uma abordagem *top-down* de Programação Dinâmica para acelerar o processamento desse método. Na Seção A.2, apresentaremos como enumerar os subconjuntos de τ que são soluções para esse problema. Novamente, mostraremos como usar programação dinâmica, destacando a diferença entre esse caso e o problema de contagem. Finalmente, na Seção A.3, apresentaremos uma abordagem para enumerar as soluções desse problema que possuem cardinalidade mínima.

A partir de um estudo desse problema, obtivemos uma solução para o problema de Escolha de Modelos, descrita no Capítulo 2. Para essa aplicação, estamos interessados em enumerear as soluções do problema que possuam cardinalidade mínima. Dessa forma, adaptamos a solução do método de obtenção de soluções com cardinalidade mínima para realizar a escolha dos modelos satisfazendo um conjunto de critérios previamente especificados.

A.1 Contagem das soluções

Mais formalmente, o problema de contagem de soluções do SSP pode ser enunciado da seguinte forma:

SSP 1: *Dados um conjunto de números inteiros positivos* $\tau = \{x_1, ..., x_n\}$ *e um número inteiro* $T > 0$*, queremos saber quantos subconjuntos de* τ *satisfazem a propriedade que a soma de seus elementos é igual a* T*.*

Podemos assumir, sem perda de generalidade, que $\sum_{i=1}^{n} x_i \geq T$, caso contrário, o problema não teria solução. Também podemos supor que para todo i temos $x_i \leq T$, pois caso contrário, se $x_i > T$ então x_i não estará em nenhum subconjunto da solução. Outra suposição, é que os elementos de τ estão ordenados crescentemente (isso não afeta o resultado do problema, mas nos permite determinar métodos mais eficientes).

Além disso, também estamos supondo que os elementos de τ são todos diferentes. Embora essa seja uma suposição um pouco mais restritiva, isso facilita um pouco o nosso problema e está em pleno acordo com o que queremos resolver a partir da análise desse problema. Caso *tau* contenha elementos repetidos, os métodos apresentados a seguir não serão capazes de distinguir dois conjuntos que apesar de possuirem os mesmos elementos, tenham sido formados por elementos distintamente indexados em *tau*. Por exemplo, se os conjuntos $w_1 = \{x_3, x_4, x_7, x_1 1\}$ e $w_2 = \{x_3, x_4, x_8, x_1 1\}$, no caso em que $x_7 == x_8$, forem soluções do problema, apesar desses conjuntos serem iguais (formados pelos mesmos elementos) seriam contados e enumerados duas vezes.

Para resolver o problema SSP vamos partir da seguinte observação: os conjuntos solução do problema de alcançar o objetivo S a partir do i-ésimo elemento de τ

pode ser dividido em duas partes: (i) a dos conjuntos que inclui x_i e (ii) a dos conjuntos que não o inclui (é óbvio que essas duas partes são disjuntas e sua união é igual a totalidade do conjunto das soluções do problema original). Observe que o primeiro caso é análogo a determinar os subconjuntos de $\{x_{i+1}, ..., x_n\}$ que somam $S - x_i$, adicionando o elemento x_i a cada um dessas soluções. Da mesma forma, o segundo caso é análogo a determinar os subconjuntos de $\{x_{i+1}, ..., x_n\}$ que somam S. Nomearemos o caso original por (S, i), o primeiro subproblema de $(S - x_i, i + 1)$, e o segundo problema por $(S, i + 1)$.

Para efeito de algoritmo, consideraremos τ um vetor de números inteiros. Dessa forma, podemos acessar aleatoriamente seus elementos com complexidade $O(1)$. Além disso, nos referiremos ao i-ésimo elemento de τ como ora x_i ora como $\tau[i]$.

Essa abordagem *dividir-para-conquistar* nos leva a um método recursivo capaz de listar todos os $2^{\#\tau}$ subconjuntos de τ. Observe que esse problema é NP-completo e, naturalmente, a solução força bruta é inviável computacionalmente mesmo em casos que τ seja razoavelmente grande. Contudo, podemos otimizar esse processamento podando a árvore de recursão.

Um critério de parada óbvio é quando $i > \#\tau$. No caso em que τ está ordenado crescentemente, podemos parar quando $S < x_i$, pois todos os elementos $\{x_{i+1}, ..., x_n\}$ também serão maiores do que S e portanto não poderão ser adicionados a solução de modo a somar S. Logo, esses dois primeiros critérios de paradas referem-se a casos que não resultarão em uma solução para o *SSP*. Quando $S == x_i$ temos que o i-ésimo elemento junto com os demais adicionados nas recursões anteriores satisfazem a

Algorithm 4 SSP (Contagem das Soluções do SSP)

```
1: procedure SSP-COUNT(τ, S, i)
2:     if (i > #τ or S < τ[i]) then
3:         return 0
4:     else if (S == τ[i]) then
5:         return 1
6:     end if
7:     return ssp-count(τ, S − τ[i], i + 1) + ssp-count(τ, S, i + 1)
8: end procedure
```

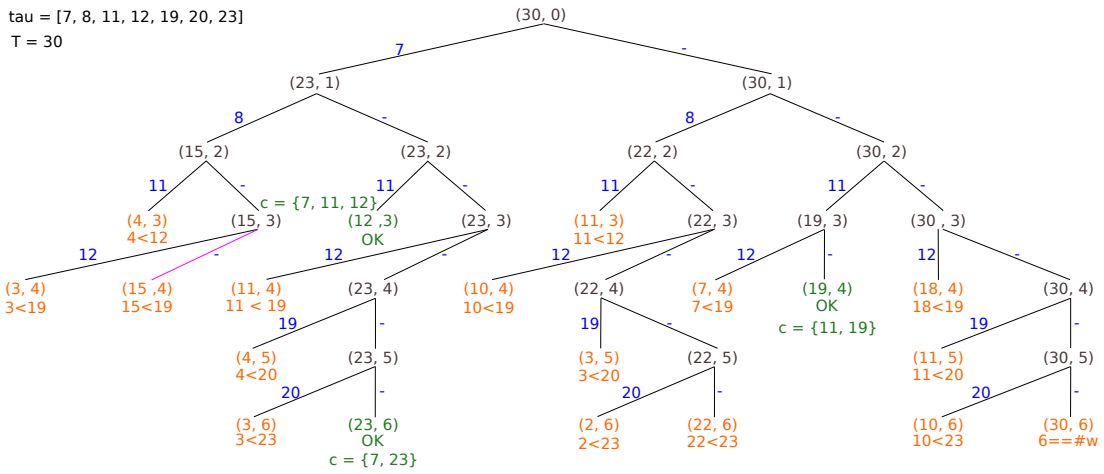


Figure A.1: Árvore de recursão do problema SSP no caso em que $\tau = \{7, 8, 11, 12, 19, 20, 23\}$ e $T = 30$.

condição de somar T. O Algoritmo 4 mostra um método para problema de contagem das soluções do *SSP*.

A Figura B.1 ilustra um exemplo de árvore de recursão para o problema SSP do conjunto $\tau = \{7, 8, 11, 12, 19, 20, 23\}$, para $T = 30$. Os desdobramentos para a esquerda referem-se a $(S - \tau[i], i + 1)$ e os à direita referem-se a $(S, i + 1)$. Nos desdobramentos à esquerda temos que $\tau[i]$ pertence ao conjunto candidato a solução, mas nos casos à direita não. Nesse esquema mostramos a árvore podada de acordo com os critérios descritos acima. Além disso, mostramos como foi o caminho de recursão para chegar às três soluções encontradas (folhas verde).

Além dos critérios de podagem da árvore de recursão previamente citados, podemos reduzir o custo de processamento evitando reprocessar ramificações iguais, ou seja, quando um par (S, i) ocorrer em dois nós distintos dá árvore (apesar de não aparecer no esquema da Figura B.1, a medida em que $\#\tau$ cresce, aumenta a quantidade de ocorrência dessas repetições). Para otimizar o processamento nesses casos, podemos utilizar uma abordagem top-down de programação dinâmica, salvando os resultados de cada par (S, i) calculado, e uma vez identificado a reincidência de um par, em vez de recalculá-lo, basta retornar o que estava armazenado.

Algorithm 5 SSP usando Programação Dinâmica

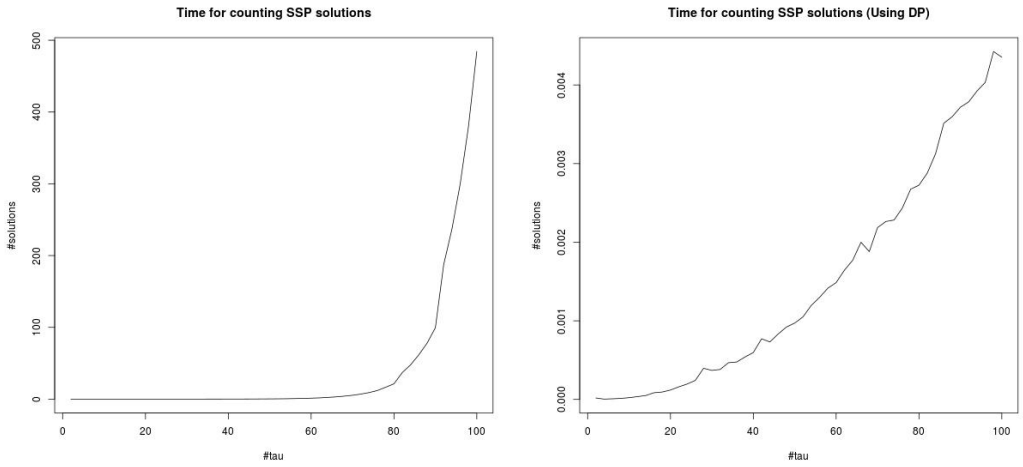
```
1: procedure SSP-COUNT-DP(τ, S, i)
2:     if (i >= #τ or S < τ[i]) then
3:         return 0
4:     else if (S == τ[i]) then
5:         return 1
6:     else if (exists(prev, S, i)) then
7:         return prev[S][i]
8:     end if
9:     prev[S][i] = ssp-count-dp(τ, S − τ[i], i + 1) + ssp-count-dp(τ, S, i + 1)
10:    return prev[S][i]
11: end procedure
```

O Algoritmo 5 mostra a uma solução usando Programação Dinâmica. Nele, usamos uma tabela hash (*prev*), indexada por dois parâmetros. O primeiro valor pode variar entre 0 e T e o segundo pode variar entre 1 e $\#\tau$. Cada campo dessa tabela contém os valores relativos ao passo referente (S, i). A função *exists* (usada no Algoritmo 5 e em outros mais adiante) testa se a tabela hash recebida no primeiro parâmetro contém como chave o par formado pelo segundo e terceiro parâmetro.

A Figura A.2 mostra uma tabela de execuções do Algoritmo 4 e do Algoritmo 5. A primeira coluna contém o tamanho de *tau*. A segunda e a terceira coluna ilustram o resultado dos Algoritmos 4 e 5, respectivamente (e portanto, são iguais). A quarta e quinta coluna mostram o tempo de execução de cada instância pelo respectivo algoritmo. A Figura A.3 mostra o tempo de execução desses algoritmos em função do tamanho da amostra. Observe que a escala de ambos gráficos são muito diferentes, ou seja, o uso de programação dinâmica reduz drasticamente o tempo de processamento. A sexta e sétima coluna ajuda a entender esse fenômeno. A sexta coluna mostra quantas vezes o Algoritmo 5 para sua execução na linha 7, evitando todas as recursões filhas. A sétima coluna mostra o tamanho da tabela *prev* (na maioria dos casos esse valor é maior do que a quantidade de processamentos evitados, mostrando que nem todos foram reaproveitados).

tau.size	count.sols	countdp.sols	count.time	countdp.time	avoided	prev.size
2	1	1	2e-06	1.8e-05	0	1
4	1	1	2e-06	4e-06	0	6
6	3	3	4e-06	8e-06	1	19
8	6	6	7e-06	1.3e-05	4	35
10	12	12	1.7e-05	2.2e-05	10	62
12	26	26	3.1e-05	3.5e-05	17	98
14	39	39	5.3e-05	4.9e-05	30	136
16	67	67	0.000101	8.5e-05	45	185
18	147	147	0.000223	9.4e-05	82	253
20	215	215	0.000383	0.000119	107	313
22	442	442	0.000712	0.00016	158	405
24	683	683	0.001127	0.000195	192	489
26	1667	1667	0.002777	0.000242	253	608
28	2774	2774	0.004761	0.000396	310	725
30	3852	3852	0.006998	0.000369	353	816
32	6128	6128	0.011168	0.000381	423	950
34	8689	8689	0.015863	0.000467	481	1080
36	11939	11939	0.023058	0.000476	542	1215
38	17429	17429	0.03394	0.00054	616	1393
40	26384	26384	0.049032	0.000596	715	1579
42	37073	37073	0.070708	0.000771	794	1768
44	49896	49896	0.098057	0.000731	872	1911
46	68275	68275	0.135936	0.000832	957	2096
48	104681	104681	0.212867	0.000921	1093	2336
50	139305	139305	0.282995	0.000971	1188	2508
52	185292	185292	0.379371	0.00105	1288	2712
54	244997	244997	0.516227	0.001196	1396	2924
56	324231	324231	0.763251	0.001298	1509	3159
58	444646	444646	0.956479	0.001415	1655	3457
60	589246	589246	1.29859	0.001486	1793	3742
62	770682	770682	1.70622	0.001644	1932	4027
64	1050514	1050514	2.34542	0.00177	2119	4349
66	1358070	1358070	3.08464	0.001999	2265	4581
68	1762572	1762572	4.09069	0.00188	2417	4856
70	2276828	2276828	5.30245	0.002186	2574	5173
72	2946918	2946918	6.98169	0.002263	2750	5511
74	3761543	3761543	9.10527	0.002282	2913	5761
76	4902964	4902964	11.9567	0.002436	3121	6167
78	6713395	6713395	16.5765	0.002675	3370	6530
80	8527730	8527730	21.317	0.002725	3551	6851
82	14921925	14921925	36.9139	0.002879	3862	7255
84	18946072	18946072	47.6451	0.003124	4054	7646
86	23965243	23965243	61.7357	0.003514	4250	8007
88	30294181	30294181	77.9609	0.003595	4454	8424
90	38079806	38079806	99.4787	0.003717	4661	8762
92	71800498	71800498	187.211	0.003785	4999	9264
94	90054828	90054828	236.944	0.003922	5214	9683
96	113736521	113736521	300.168	0.004033	5477	10131
98	142195519	142195519	380.266	0.004425	5700	10563
100	177223016	177223016	484.251	0.004353	5926	10976

Figure A.2: Avaliação do processamento da Contagem de soluções do SSP



(a) Sem DP (b) Com DP

Figure A.3: Tamanho de *tau* por tempo de execução

A.2 Enumeração das soluções

O problema de contagem descrito nos Algoritmos 5 e 5 responde a uma questão global. A seguir, descreveremos o problema de enumeração das soluções. Nesse caso, cada recursão que obtiver uma solução do problema agregará uma resposta específica (dependendo dos respectivos elementos de τ que foram selecionados). Adiante apresentaremos propriamente esse problema, mas antes vamos enunciá-lo formalmente:

SSP 2: *Dados um conjunto de números inteiros positivos $\tau = \{x_1, ..., x_n\}$ e um número inteiro $T > 0$, queremos saber quais são os subconjuntos de τ que satisfazem a propriedade de que a soma de seus elementos é igual a T.*

A solução mais simples para esse problema é feita inspirada na abordagem *dividir-para-conquistar* previamente explicada. Essa abordagem é apresentada no Algoritmo 6, onde $w \subset \tau$ é um candidato a solução e C o conjunto das soluções.

Vale destacar que nas linhas 3 e 5, utilizamos a notação $w \cup \{\tau[i]\}$ para denotar que adicionaremos o elemento $\tau[i]$ na lista w. Como este é um algoritmo de enumeração, a função *ssp-enum* não precisa retornar o valor total da quantidade de soluções. Em vez disso, cada solução encontrada é armazenada na lista de soluções C.

Algorithm 6 SSP: Enumeração das Soluções

```
1: procedure SSP-ENUM(τ, S, i, w, C)
2:     if (i >= #τ or S < τ[i]) then
3:         return
4:     else if (S == τ[i]) then
5:         C.append(w ∪ {τ[i]})
6:     else
7:         ssp-enum(τ, S − τ[i], i + 1, w ∪ {τ[i]}, C)
8:         ssp-enum(τ, S, i + 1, w, C)
9:     end if
10: end procedure
```

A solução apresentada no Algoritmo 6 não usa programação dinâmica. Assim como no Algoritmo 5, podemos adicionar uma tabela para reusar os resultados previamente calculados. Contudo, esse caso é um pouco mais complicado, pois a partir de um passo de (S, i) podemos obter mais de um resultado positivo. Nesse caso, precisamos armazenar os elementos que serão adicionados em cada um desses casos e uma vez que percebemos que o par (S, i) está sendo recalculado, adicionaremos a solução o w com cada uma dos conjuntos dos elementos adicionados a partir de (S, i).

Esses elementos, serão armazenados em uma tabela hash (indexada com dois parâmetros, semelhantemente ao que foi explicado para a tabela *prev*), a qual chamaremos de *tail*. Cada campos dessa tabela é uma lista de listas (ou seja, a reunião dos elementos que serão adicionados a partir de (S, i)). O Algoritmo 7 apresenta uma solução para o problema de enumeração de soluções do SSP usando uma abordagem top-down de programação dinâmica.

Algorithm 7 SSP: Enumeração usando Programação Dinâmica

```
1: procedure SSP-ENUM-DP(τ, S, i, w, C)
2:     if (i >= #τ or S < τ[i]) then
3:         prev[S][i] = 0
4:         return False
5:     end if
6:     if (exists(prev, S, i)) then
7:         if (prev[S][i] == 0) then
8:             return False
9:         else if (prev[S][i] ≥ 1) then
10:            foreach s in tail[S][i] do
11:                C.append(w ∪ s)
12:            end foreach
13:            return True
14:        end if
15:    end if
16:    if (τ[i] == S) then
17:        C.append(w ∪ {τ[i]})
18:        prev[S][i] = 1
19:        tail[S][i] = {{τ[i]}}
20:        return True
21:    end if
22:    isChanged = False
23:    if (ssp-enum-dp(τ, S − τ[i], i + 1, w ∪ {τ[i]}, C) then
24:        prev[S][i] = 1
25:        foreach s in tail[S − τ[i]][i + 1] do
26:            C.append(w ∪ {τ[i]} ∪ s)
27:        end foreach
28:        isChanged = True
29:    end if
30:    if (ssp-enum-dp(τ, S, i + 1, w, C) then
31:        prev[S][i] = 1
32:        foreach s in tail[S][i + 1] do
33:            C.append(w ∪ s)
34:        end foreach
35:        isChanged = True
36:    end if
37:    if (not isChanged) then
38:        prev[S][i] = 0
39:    end if
40:    return isChanged
41: end procedure
```

tau.size	enum.size	enumdp.size	countdp.size	enum.time	enumdp.time	countdp.time	avoided	prev.size
2	1	1	1	4e-06	1.7e-05	1e-06	0	1
4	1	1	1	4e-06	1.2e-05	3e-06	0	6
6	3	3	3	1.3e-05	3.4e-05	8e-06	2	19
8	6	6	6	2.6e-05	6.9e-05	1.3e-05	6	35
10	12	12	12	6e-05	0.000131	2.2e-05	17	62
12	26	26	26	0.000104	0.000208	3.5e-05	30	98
14	39	39	39	0.00018	0.000318	5e-05	52	136
16	67	67	67	0.000298	0.000474	8.6e-05	80	185
18	147	147	147	0.000681	0.000904	9.7e-05	143	253
20	215	215	215	0.001013	0.001547	0.00018	179	313
22	442	442	442	0.002679	0.001772	0.000155	267	405
24	683	683	683	0.003256	0.002514	0.000197	333	489
26	1667	1667	1667	0.008368	0.004914	0.000242	452	608
28	2774	2774	2774	0.01343	0.007973	0.000292	564	725
30	3852	3852	3852	0.018836	0.010239	0.000313	636	816
32	6128	6128	6128	0.03062	0.016305	0.000364	762	950
34	8689	8689	8689	0.045402	0.02233	0.000422	873	1080
36	11939	11939	11939	0.062006	0.029507	0.000465	973	1215
38	17429	17429	17429	0.09137	0.042529	0.000532	1116	1393
40	26384	26384	26384	0.138644	0.062598	0.000608	1301	1579
42	37073	37073	37073	0.195989	0.087758	0.000687	1463	1768
44	49896	49896	49896	0.285799	0.116536	0.000785	1608	1911
46	68275	68275	68275	0.386637	0.158293	0.000842	1779	2096
48	104681	104681	104681	0.594352	0.242231	0.000929	2029	2336
50	139305	139305	139305	0.801311	0.323343	0.00101	2197	2508
52	185292	185292	185292	1.06926	0.430418	0.001119	2385	2712
54	244997	244997	244997	1.45134	0.573032	0.001236	2582	2924
56	324231	324231	324231	2.00654	0.748354	0.001276	2793	3159
58	444646	444646	444646	2.7358	1.04356	0.001391	3075	3457
60	589246	589246	589246	3.69577	1.41053	0.00159	3331	3742
62	770682	770682	770682	5.063	1.79041	0.00178	3581	4027
64	1050514	1050514	1050514	6.5887	2.40195	0.001754	3913	4349
66	1358070	1358070	1358070	8.68419	3.12017	0.001858	4159	4581
68	1762572	1762572	1762572	11.4536	4.05303	0.001981	4450	4856
70	2276828	2276828	2276828	14.9369	5.25011	0.002127	4748	5173
72	2946918	2946918	2946918	19.6887	6.80021	0.002265	5074	5511
74	3761543	3761543	3761543	25.4017	8.72433	0.002398	5355	5761
76	4902964	4902964	4902964	34.4607	11.4432	0.002577	5760	6167
78	6713395	6713395	6713395	46.6188	15.9001	0.002799	6209	6530
80	8527730	8527730	8527730	60.7752	20.2409	0.002881	6521	6851

Figure A.4: Avaliação do processamento da Enumeração de soluções do SSP

A Figura A.4 ilustra o processamento da enumeração (semelhantemente ao que foi apresentado na Figura A.2). Da segunda a quarta coluna mostramos a quantidade de soluções obtidas pelos algoritmos 6, 7 e 5, respectivamente (e naturalmente, elas são iguais). Da quinta a sétima coluna, mostramos o tempo de processamento desses algoritmos.

O processo de enumeração usando programação dinâmica, embora bem mais rápido do que quando não se usa, é razoavelmente custoso (além de consumir muita memória também). Isso é de se esperar, dada a necessidade de armazenar uma quantidade muito grande de conjuntos na solução.

Vale observar que a quantidade de soluções do problema *SSP* cresce muito a medida que o tamanho do conjunto de entrada cresce. Devido a isso, o problema de enumeração dessas soluçõoes pode ser muito custoso (tanto do ponto de vista de processamento, quanto de consumo de memória). Observe que processo de enumeração usando programação dinâmica, embora bem mais rápido do que aquele que não usa, é razoavelmente custoso.

A.3 Soluções com Cardinalidade minima

No Capítulo 2, apresentamos uma abordagem inspirada no SSP, que utiliza apenas as soluções cuja cardinalidade é mínima. Nessa direção, existem algumas adaptações do problema que podemos fazer para otimizar o processamento para tratar esse caso particular (por exemplo, não precisamos lidar com as soluções cuja cardinalidade não seja mínima). Mais formalmente, agora queremos tratar o seguinte problema:

SSP 3: *Dados um conjunto de números inteiros positivos* $\tau = \{x_1, ..., x_n\}$ *e um número inteiro* $T > 0$*, obter os subconjuntos de* τ *com cardinalidade mínima cuja soma de seus elementos é igual a* T*.*

Nós adotaremos uma abordagem em duas etapas para resolver esse problema. No primeiro passo (Algoritmo 8), descobriremos qual é o tamanho mínimo dos conjuntos que resolvem o problema *SSP 3*. Na segunda etapa, enumeramos os conjuntos com tal cardinalidade (Algoritmo 9).

Algorithm 8 SSP 3 (Tamanho mínimo das Soluções do SSP)

```
1: procedure SSP-MIN(τ, S, i, w, M)
2:     if (i >= #τ or S < τ[i] or #w + 1 > M) then
3:         prev[S][i] = 0
4:         return False
5:     else if (S == τ[i]) then
6:         M = #w + 1
7:         prev[S][i] = 0
8:         return True
9:     else if (exists(prev, S, i)) then
10:        if ( thenprev[S][i] > 0)
11:            M = min(#w + prev[S][i], M)
12:            return True
13:        end if
14:        return False
15:    end if
16:    isChanged = False
17:    if ( ssp-min(τ, S − τ[i], i + 1, w ∪ {τ[i]} ) then
18:        prev[S][i] = prev[S − τ[i]][i + 1] + 1
19:        isChanged = True
20:    end if
21:    if ( ssp-min(τ, S, i + 1, w) ) then
22:        if ( isChanged and exists(prev, S, i) and prev[S][i] > 0 ) then
23:            prev[S][i] = min(prev[S][i], prev[S][i + 1])
24:        else
25:            prev[S][i] = prev[S][i + 1]
26:        end if
27:        isChanged = True
28:    end if
29:    if (not isChanged) then
30:        prev[S][i] = 0
31:    end if
32:    return isChanged
33: end procedure
```

Algorithm 9 SSP 3 (Enumeração das soluções de cardinalidade mínima)

```
1: procedure SSP-MIN-ENUM(τ, S, i, w, M, C)
2:     if (i >= #τ or S < τ[i] or #w + 1 > M) then
3:         return
4:     else if (exists(prev, S, i) and prev[S][i] + #w > M) then
5:         return
6:     else if (S == τ[i]) then
7:         C.append(w ∪ {τ[i]})
8:     end if
9:     ssp-min-enum(τ, S − τ[i], i + 1, w ∪ {τ[i]}, C)
10:    ssp-min-enum(τ, S, i + 1, w, C)
11: end procedure
```

Dessa forma, sempre que a recursão estiver processando um conjunto candidato com cardinalidade maior do que a mínima desejada, podemos interromper a recursão (sendo esse um critério de parada bastante eficiente). Além disso, para resolver o primeiro problema, utilizamos uma abordagem de programação dinâmica, construindo uma tabela que contabiliza para a recursão (S, i) se há uma solução possível e, no caso positivo, qual é a quantidade mínima de elementos que serão inseridos nas recursões filhas. Essa tabela, será reaproveitada na segunda recursão, pois saberemos a priori quais são as ramificações da recursão que resultarão em resultados para o problema.

Como a primeira etapa apenas calcula o tamanho mínimo do conjunto do solução, e não envolve enumeração, é um processamento bem rápido. Além disso, reaproveitar a tabela de programação dinâmica do primeiro passo, na etapa de enumeração, evita muito processamento. A Figura A.5 exemplifica esse ganho de processamento. A oitava coluna mostra o tempo total gasto para enumerar os conjuntos com cardinalidade mínima usando essa abordagem em dois passos, enquanto a nona coluna mostra o tempo gasto para enumerar todas as soluções.

Por fim, vale destacar que a quantidade de soluções para esse problema escala muito rapidamente. Mesmo a quantidade de soluções com cardinalidade mínima ultrapassa a ordem de 10^{15} muito rapidamente (conforme pode ser visto na quinta coluna da tabela exemplificada na Figura A.5). Dessa forma, enumerar todas as

tau.size	min	enumdp.min	min.size	enumdp.size	min.time	min.enum	min.total	enumdp.time
4	2	2	2	2	9e-06	9e-06	1.8e-05	4.5e-05
6	2	2	2	3	2.7e-05	1.1e-05	3.8e-05	6.6e-05
8	2	2	2	7	5.3e-05	1.1e-05	6.4e-05	7.9e-05
10	2	2	2	23	9.8e-05	1.4e-05	0.000112	0.000206
12	2	2	2	46	0.00015	2.2e-05	0.000172	0.000346
14	2	2	2	90	0.000211	2.3e-05	0.000234	0.000556
16	2	2	4	212	0.000308	1.9e-05	0.000327	0.001033
18	2	2	2	420	0.000421	2.3e-05	0.000444	0.001738
20	3	3	34	848	0.000528	0.000297	0.000825	0.003042
22	2	2	4	1373	0.000636	4.7e-05	0.000683	0.004707
24	3	3	52	2478	0.000835	0.000549	0.001384	0.008007
26	2	2	2	11925	0.000998	3.5e-05	0.001033	0.030562
28	2	2	2	17505	0.001129	4.4e-05	0.001173	0.043809
30	2	2	2	25820	0.001314	5.1e-05	0.001365	0.063834
32	3	3	70	47276	0.001544	0.000756	0.0023	0.116056
34	3	3	84	74261	0.001777	0.000956	0.002733	0.18156
36	2	2	2	108040	0.002061	6.7e-05	0.002128	0.266971
38	3	3	110	182052	0.002236	0.00171	0.003946	0.442772
40	2	2	2	361985	0.002678	8.5e-05	0.002763	0.877284
42	3	3	100	632821	0.003013	0.001977	0.00499	1.53197
44	3	3	130	895461	0.003311	0.002555	0.005866	2.18139
46	2	2	4	1336374	0.00365	9.5e-05	0.003745	3.25398
48	2	2	4	1909964	0.004007	1e-04	0.004107	4.67068
50	2	2	2	2659123	0.004364	0.000107	0.004471	6.57127
52	3	3	214	3760396	0.004805	0.003985	0.00879	9.30061
54	3	3	214	5461169	0.00526	0.004685	0.009945	13.5249
56	2	2	4	7754874	0.005774	0.00015	0.005924	19.2573
58	2	2	2	11917463	0.006249	0.000188	0.006437	29.6666
60	2	2	2	17148131	0.00685	0.000201	0.007051	41.8475
62	3	3	274	24037256	0.007371	0.006885	0.014256	59.2061
64	2	2	2	32932585	0.00788	0.000259	0.008139	81.8453
66	2	2	2	44643193	0.008403	0.000289	0.008692	111.899
68	2	2	2	60287442	0.040882	0.002815	0.043697	155.353

Figure A.5: Avaliação do processamento da Enumeração de soluções do SSP com cardinalidade mínima

Algorithm 10 SSP: Solução por partes

```
1: procedure SSP-PARTED-MIN-ENUM(τ, T, P)
2:     splitted = splitVector(τ, P)
3:     M = ∞
4:     for p in {1, ..., P} do
5:         w = Vector()
6:         prev.clear()
7:         ssp − min(splitted[p], T, 0, w, M)
8:     end for
9:     w = Vector()
10:    prev.clear()
11:    ssp − min(τ, T, 0, w, M)
12:    w = Vector()
13:    C = Vector < Vector < int >> ()
14:    ssp − min − enum(τ, T, 0, w, M, C)
15:    return C
16: end procedure
```

soluções é proibitivo (do ponto de vista de processamento e armazenamento). Sendo assim, podemos enumerar apenas uma quantidade X de subconjuntos de τ que são solução do problema *SSP 3*. Para tal, basta adicionar a condição de parada $\#E > X$ no Algoritmo 9 para interromper o processamento após alcançar o objetivo desejado. Essa restrição não afetará os propósitos de escolha de modelos, que é nosso maior objetivo.

Observe que a condição de parada $\#w + 1 > M$ é muito eficiente quando M é pequeno, evitando muito processamento desnecessário. Dessa forma, opções que reduzam M rapidamente otimizam o cálculo da cardinalidade mínima das soluções. Nessa direção, uma opção interessante é dividir o conjunto τ em pequenos subconjuntos, resolver o problema nesses subconjuntos, e finalmente rodar o problema completo, mas com um M pequeno. O Algoritmo 10 ilustra essa abordagem.

Vale destacar que a quantidade de possíveis subconjuntos de τ cresce exponencialmente a medida que seu tamanho cresce, e portanto rodar o processo P vezes para um conjunto de tamanho $\frac{1}{P}$ do tamanho original é significativamente mais rápido. Além disso, o valor de M é sempre menor do que ou igual ao valor da iteração ante-

p	min	time
0	3	5.44973
1	3	8.44255
2	3	10.677
3	3	12.93
4	3	15.1602
5	2	17.1513
6	2	17.7695
7	2	17.9993
8	2	18.2186
9	2	18.4402

Figure A.6: Tempo de processamento por iteração do Algoritmo 10

rior fazendo com que a medida em que p cresce, menor é o tempo de processamento de *ssp-min*. Também vale destacar que executar *ssp-min* em τ com M pequeno, é significativamente mais rápido, do que começar com $M = \infty$. Logo, o tempo total dessa execução dividida em partes é menor do que executar o problema original para $ssp - min(\tau, T, 0, w, M = \infty)$.

Por exemplo, o tempo gasto para processar diretamente um conjunto com 20000 números aleatórios entre 1 e 40000, para alcançar um alvo de 45159, com $P = 10$, é ilustrado na tabela da Figura A.6. O processamento das P partes demorou cerca de 18s, enquanto o cálculo final de ssp-min em todo τ para $M = 2$ foi de 160s (18s para as 10 primeiras etapas e mais 160 para a checagem final, totalizando 178s). Por outro lado, a execução do processo no conjunto todo, começando sem nenhuma informação prévia de M foi de 696s.

Appendix B

Criando Bases de Dados

Nesse apêndice, eu descreverei brevemente como criei a base de dados de imagens usada nessa pesquisa. O primeiro passo é a aquisição das imagens. Em seguida, criamos algumas listas com modelos respeitando certas características.

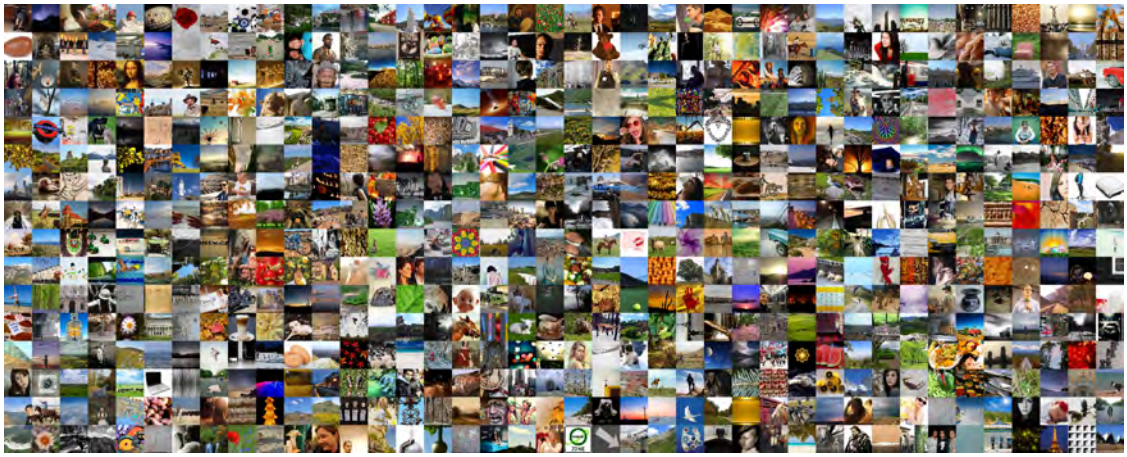


Figure B.1: Coleção de Imagens: um exemplo de base de dados

B.1 Aquisição das Imagens

Utilizei imagens de três fontes: ImageNet, Twitter e Flicker. Após a aquisição dessas imagens, elas foram agrupadas em diferentes coleções com tamanhos distintos.

As imagens do Twitter foram aproveitadas da pesquisa do Observatório Olímpico [12]. Nas seções a seguir, apresentarei como obtive as imagens do ImageNet e do Flickr.

B.1.1 ImageNet

As imagens do ImageNet foram baixadas do site: http://image-net.org/

Primeiramente obtivemos as URLs das imagens, da release *Fall 2011*. As URLs estão disponíveis em um arquivo texto, contendo o endereço de mais de 14 milhões de imagens.

Essas imagens são baixadas atraves do script *imagenet-downloader.py*. Dado a enorme quantidade de imagens, não baixei todas. Supondo as imagens indexadas pela respectiva linha no arquivo fonte, o script permite baixar um intervalo de imagens.

Infelizmente nem todas imagens são adequadas ao uso. Após o download dessas imagens, precisamos descartar as que nao sao adequadas. Isso é realizado pelo script *cleandb.py*. Esse script permite remover imagens com extensões não adequadas, imagens muito grande, e imagens que podem ter algum problema de codificação (imagens que a biblioteca Python Image Library não consiga decodificar e abri-las).

B.1.2 Flickr

As imagens do flickr foram baixadas utilizando uma versão adaptada da ferramenta FlickrDownloadr (*https://flickrdownloadr.com/*). A versão original baixa um album especificado. Na nossa versão, baixamos todos os albuns de um autor. O nome do arquivo de cada imagem é seu respectivo identificador do flickr (muitas imagens distintas tem títulos identicos).

As imagens são salvas com o identificador do flickr, em uma estrutura de diretórios satisfazendo $NOME\_DO\_AUTOR$ / $NOME\_DO\_ALBUM$.

Além da imagem, também salvamos um arquivo JSON com metadados como:

- descrição
- Id e nome do autor
- tags
- momento (dia e hora) em que foi tirada
- url
- licença e se a imagem pode ser compartilhada
- geolocalização

As imagens e os arquivos JSON com os metadados associados de um album *USERSET* de um usuário *USERNAME* estão salvas, respectivamente, nos seguintes diretórios:

- $FLICKR\_DIR/FILE/USERNAME - SETNAME/*.JPG$
- $FLICKR\_DIR/MEDIA/USERNAME - SETNAME/*.JSON$

Para usar esse script, precisamos de uma chave do flickr. Além disso, precisamos passar o nome do usuário. O código abaixo exemplifica uma chamada desse script para baixar todos os albuns do usuário *cookiesound*:
$ python flick_ download.py -k b5ba1be900afc0250f638531bf5901fb -s 40da7c4559b414a8 -u cookiesound

Dado que esse script baixa apenas as fotos de um único usuário, para baixar grandes quantidades de fotos, criamos um outro script (*exec-flickr-users.py*) que chama o *flick_ download.py* para cada usuário listado em um arquivo texto.

Para evitar problemas com manipulações de endereços, na hora de criar o diretório onde as imagens são armazenadas, os nomes dos usuários e de seus albuns são "limpos" removendo os characteres não alfanuméricos. Dessa forma, para não

perdermos os nomes originais registramos tais informações em arquivos armazenados na pasta *FLICKR_DIR+"/_CONFIG/"*. O programa registra os usuários baixados no arquivo *_USERS.txt"*. Além disso, para cada usuário (USERNAME), registra os albuns no arquivo *USERNAME.txt"*.

B.2 Criação de uma base de dados

No capítulo 2, apresentamos como escolher modelos de uma base de dados Ω. Nesse trabalho, representamos nossa base de dados como uma lista de modelos (sem nenhuma estrutura específica). O processo de criação de uma base de dados foi feito após a aquisição dos dados.

Uma vez baixado modelos de diversas fontes, criamos bases de dados com diferentes quantidades de elementos de uma mesma fonte, ou combinando fontes. O endereço desses elementos eram armazenados em um arquivo (dado que estavam espalhados em diferentes diretórios).

Para cada fonte, registramos todos os respectivos modelos (checando se não havia repetições). O processo de criação da base de dados consistia de uma amostragem desse registro (ou da união de registros de fontes diferentes).

Em muitos experimentos, as imagens precisavam ser pre-processadas (por exemplo, recortar u maior quadrado centralizado da imagem; e/ou escalar a imagem para um tamanho específico; etc). Para evitar o reprocessamento de uma imagem, armazenávamos o resultado como se fosse um novo conjunto de imagem vindo de uma fonte específica, e criamos as bases de dados utilizando esses elementos, em vez dos modelos originalmente baixados.

Bibliography

[1] J. Hays and A. A. Efros, "Scene completion using millions of photographs." SIGGRAPH, 2007.

[2] W. T. Freeman, T. R. Jones, and E. C. Pasztor, "Example-based super-resolution." Computer Graphics and Applications, 2002.

[3] L.-Y. Wei, S. Lefebvre, V. Kwatra, and G. Turk, "State of the art in example-based texture synthesis." Eurographics State of the Art Reports, 2009.

[4] A. A. Efros and W. T. Freeman, "Image quilting for texture synthesis and transfer." ACM SIGGRAPH, 2001.

[5] H. Zhou, J. Sun, G. Turk, and J. Rehg, "Terrain synthesis from digital elevation models." IEEE Transactions on Visualization and Computer Graphics, 2007.

[6] L. Cruz, L. Velho, E. Galin, A. Peytavie, and E. Guerin, "Patch-based terrain synthesis." GRAPP, 2015.

[7] P. Merrell, "Example-based model synthesis." Symposium on Interactive 3D Graphics and Games, 2007.

[8] C. Han, E. Risser, R. Ramamoorthi, and E. Grinspun, "Multiscale texture synthesis." ACM SIGGRAPH, 2008.

[9] A. Oliva and A. Torralba, "Building the gist of a scene: the role of global image features in recognition." Visual Perception, Progress in Brain Research, 2006.

[10] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "Imagenet large scale visual recognition challenge." International Journal of Computer Vision, 2015.

[11] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, 2009.

[12] J. Giannella and L. Velho, "Observator!o2016." Technical Report, 2016.