UNIVERSIDADE FEDERAL DE MINAS GERAIS

INSTITUTO DE CIÊNCIAS EXATAS

DEPARTAMENTO DE ESTATÍSTICA

PEDRO HENRIQUE MELO ALBUQUERQUE

CONGLOMERADOS ESPACIAIS:
UMA NOVA PROPOSTA.

Belo Horizonte

2008

PEDRO HENRIQUE MELO ALBUQUERQUE

CONGLOMERADOS ESPACIAIS:

UMA NOVA PROPOSTA.

Dissertação apresentada ao Curso de pós-graduação em Estatística, Departamento de Estatística, Instituto de Ciências Exatas, Universidade Federal de Minas Gerais, como requisito parcial para obtenção do grau de mestre em estatística.

Orientador: Luiz Henrique Duczmal

Belo Horizonte

2008

RESUMO

Este trabalho tem como proposta apresentar uma nova abordagem para geração de conglomerados espaciais, utilizando entropia não-paramétrica e algoritmos estocásticos, com objetivo de encontrar a melhor partição do mapa.

A abordagem univariada foi aqui implementada na linguagem C# e o caso multivariado foi sugerido em algumas das seções desta dissertação.

Palavras-chave: estatística espacial, conglomerados espaciais, núcleo estimador, entropia, otimização estocástica.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A busca por conglomerados espaciais tem ampla utilização em diversas áreas do conhecimento, sendo comumente utilizada em economia, sociologia, epidemiologia, pesquisa de mercado, entre outras.

Nosso objetivo é encontrar a partição ótima de uma região, em C conglomerados, de modo que os conglomerados sejam internamente conexos e minimizem uma determinada função objetivo.

Definição 1 *Dizemos que um conglomerado é internamente conexo, se todas as regiões de estudo, como por exemplo: municípios, setores censitários estão conectadas entre si de modo a criar um conjunto de polígonos não separáveis.*

Definição 2 *Um mapa pode ser particionado em C partições, onde cada uma dessas partições representam um conglomerado espacialmente conexo.O mapa formado por essas partições é também chamado de configuração espacial ou simplesmente configuração.*

Aqui, utilizaremos a definição de polígono como uma generalização da área a ser estuda, por exemplo, um município pode ser chamado de polígono já que é um contorno formado por segmentos de retas que não se cruzam.

Como critério de otimização utilizaremos a definição de entropia (capítulo 2), que como mostraremos, pode ser utilizada como ferramenta para a construção de conglomerados espaciais univariados.

Aqui apresentaremos uma nova proposta de busca e construção de *clusters* espacias; é necessário, no entanto, apresentar alguns resultados que serão pré-requisitos para a criação dos conglomerados espaciais.

Dividimos da seguinte maneira essa dissertação de mestrado:

1. Teoria dos grafos.

2. Teoria da informação.
3. Núcleo estimador.
4. Conglomerados espaciais.
5. Simulação.
6. Conclusão.

Cada um desses capítulos é importante para que se possa entender como funcionará o algoritmo de *clusterização*.

Na seção de teoria dos grafos apresentaremos como uma região pode ser representada através de grafos e introduziremos um conceito importante de *nós de articulação*.

Como critério de otimização utilizaremos a definição de *entropia* (capítulo 3) e uma estatística univariada que desejaremos minimizar.

A estimação da entropia será realizada de maneira *não-paramétrica* univariada (capítulo 4).

No capítulo (cap. 5) apresentaremos o algoritmo de *clusterização* espacial que é a proposta deste trabalho.

Finalmente nos dois últimos capítulos apresentaremos algumas simulações referentes ao algoritmo de *clusterização* espacial (cap. 6) e a conclusão (cap. 7).

Portanto, o objetivo desse trabalho é introduzir o leitor em uma nova proposta de *clusterização* espacial e aplicar essa proposta para o caso univariado contínuo e discreto, ficando a cargo dos interessados a expansão dessa metodologia para o caso multivariado.

2 TEORIA DOS GRAFOS.

Um grafo é uma estrutura na forma $G = (V, E)$ onde V é um conjunto discreto e E é uma família cujos elementos são definidos em função dos elementos de V.

Dizemos que um grafo é não direcional ou não-direcionado se pudermos ir de um nó u a outro um nó v em qualquer direção.

Os elementos em V são chamados vértices, nós ou pontos e o valor $n = |V|$ é a ordem do grafo, que determina a quantidade de nós que o grafo possui.

Já uma família E pode ser entendida como uma relação ou conjunto de relações de adjacências, cujos elementos são chamados, em geral, ligações. Particularmente nas estruturas não orientadas os $e \in E$ são conhecidas como arestas e nas estruturas não-orientadas são denominadas arcos.

O grafo pode ser representado por uma matriz de adjacência ou matriz de vizinhança $A(G)$. Trata-se de uma matriz de ordem n ($n = |V|$) na qual se associa cada linha e cada coluna a um vértice. Os valores nulos correspondem à ausência de ligação entre os nós e os valores não-nulos (habitualmente iguais a 1) representam a existência de ligação.

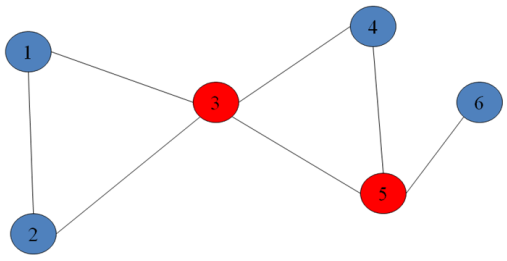

Figura 2.1: Exemplo de Grafo.

O grafo acima pode ser representado em forma matricial da seguinte maneira:

$$A(G) = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Onde o número do nó equivale a linha e a coluna da matriz acima, por exemplo, o nó 1 (um) possui ligação com os nós dois e três.

Estamos interessados em identificar em um grafo os nós de articulação, representados na figura acima pelos nós em vermelho; para isso utilizaremos o algoritmo DFS (Depth-First Search).

2.1 ALGORITMO DEPTH-FIRST SEARCH

O Algoritmo DFS (Depth-First Search) é um método de busca em grafos não orientados. Apesar de ser amplamente usado em diversas áreas do conhecimento (computação, topologia, psicologia,...etc), não é um algoritmo novo, pois é conhecido desde o século 19 como algoritmo de *Trémaux*.

Assumindo que nós temos um grafo finito $G(V,E)$ conexo, então começando em um dos vértices nós caminhamos ao longo das arestas de nó a nó visitando todos os nós.

Procuramos então um algoritmo que garantirá que todos os nós sejam visitados da maneira mais eficiente possível, ou seja, atravessar as arestas somente se necessário e no menor número de vezes.

Durante o processo marcamos o grafo, são apenas duas marcas necessárias: F e E. Marcaremos com F se for a primeira vez que entramos no vértice e E para qualquer outra passagem usada quando se deixa o nó. Nenhuma marca é apagada ou alterada durante o processo.

Essa marca é realizada computacionalmente, ou seja, guardaremos em um vetor o valor da marca de um determinado nó com os caracteres F e E.

Algoritmo 1 *Algoritmo de Trémaux*

1. *$v = s$ começa-se no nó s.*

2. *Se todas as passagens estão marcadas vá para 4.*

3. *Escolha uma passagem não marcada, marque com E e atravesse a aresta até o nó u.*

   Se u tem alguma passagem marcada (isto é , não é um vértice novo) marque esta passagem pelo qual u foi atingido por E, volte para v e vá para o passo 2.

   Caso contrário, se u é um novo nó, ou seja, não tem nenhuma marca de passagem, marque esta passagem pelo qual u foi atingido por F e faça $v = u$ *e vá para 2.*

4. *Se não há passagens marcadas com F pare, pois voltamos ao nó original.*

5. *Use a passagem marcada com F atravesse a aresta até o nó de destino u, faça* $v = u$ *e vá para 2.*

O algoritmo de *Trémaux* não permite que uma aresta seja atravessada duas vezes na mesma direção, o que garante a eficiência desse algoritmo.

Um grafo G(V, E), é dito ter um nó de separação (também chamado de nó de articulação) se existem os nós *a*, *v* e *b* tal que , $a \neq v$ e $b \neq v$ onde todos os caminhos que conectam *a* e *b* passam necessariamente por *v*; nesse caso *v* é chamado nó de separação (nó de articulação).

Outra forma de definir um nó de separação é dizer são aqueles vértices que se forem retirados do grafo produzem um grafo não-conexo.

Seja $K(v)$ o número do passo (iteração) em que o nó *v* foi descoberto, $L(v)$ o *lowpoint*, ou seja, é o menor $K(u)$ o qual atinge *v* por uma aresta *a* e $F(v)$ representa o *pai* do nó *v*, corresponde ao nó que precede o vértice *v* na busca dentro do grafo, utilizaremos então uma modificação do algoritmo de *Trémaux* para econtrar esses nós :

Algoritmo 2 *Algoritmo de Trémaux modificado*

1. *Inicialize todas as variáveis. Seja s o nó inicial. v = s.*

2. *i = i + 1; K(v) = i ; L(v) = i ;*

3. *Se v tem todas as arestas usadas vá para 5.*

4. *Escolha uma aresta não usada, marque a aresta: Se* $K(u) \neq 0$ *então* $L(v) = Min(L(v), K(u))$ *e vá para 3 Se* $K(u) = 0$ *então F(v) = v; v = u; Vá para 2*

5. *Se K(v) = 1 vá para 9. Então formou-se um grupo não separável.*

6. *Se* $L(v) < K(F(v))$ *então L(F(v)) = Min(L(F(v)), L(v)) vá para o passo 8*

7. *F(v) é um nó de separação.*
8. *v = F(v) vá para 3*
9. *Se s tem todas as arestas usadas PARE!*
10. *O nó s é um nó de separação. Faça v = s e vá para 4*

Esses nós de separação representarão um papel importante no algoritmo de *clusterização*, já que de todos os *vértices*, esses não poderão ser alterados, pois caso isso ocorra o grafo não será mais conexo.

Mais detalhes sobre o algoritmo *Depth-First Search* e a abordagem de teoria dos grafos pode ser encontrado Even (1979) e Netto (2006).

3 ENTROPIA.

3.1 DEFINIÇÃO DE ENTROPIA.

Entropia é uma medida de *incerteza* de uma variável aleatória. Utilizaremos essa medida na busca por conglomerados que possuam *miníma* entropia.

O conceito de entropia foi introduzido pela primeira vez na *termodinâmica*, e o seu objetivo é mensurar o *caos* de um sistema. Quanto mais organizado um sistema, menor então será o valor medido de entropia.(COVER; THOMAS, 2006)

É também muito utilizada em comunicação, para a busca e correção de ruídos em mensagens e também em codificação.

Definimos entropia da seguinte maneira:

Definição 3 *Seja então X uma variável aleatória definida no espaço de probabilidade* $(\Omega, \mathscr{F}, P)$ *então:*

$$H(X) = -\int_{-\infty}^{+\infty} log\,(f\,(x))\,dF\,(x) \tag{3.1}$$

Onde $f(x)$ é a função densidade de probabilidade, ou função massa de probabilidade da variável aleatória X.

Se o *log* estiver na base 2 (dois) dizemos que a entropia é expressa em *bits*, se a base do logaritmo for *b* temos a entropia expressa como $H_b(x)$ e se o logaritmo estiver na base *neperiano* dizemos que a entropia é expressa em *nats*.

Quando não indicado, dizemos que o log está na sua base natural, $e \approx 2.71828$, e portanto a entropia é expressa em *nats*.

A definição de entropia aqui apresentada está relacionada de certa maneira com a definição de entropia utilizada na termodinâmica.

No caso da variável aleatória X ser contínua e possuir função densidade de probabilidade, dizemos que a medida $H(.)$ é a entropia diferencial.

3.2 PROPRIEDADE ASSINTÓTICA DE EQUIPARTIÇÃO.

Em teoria da informação, o análogo a lei dos grandes números é a propriedade assintótica de equipartição (PAE).

Essa propriedade é uma conseqüência direta da lei fraca dos grandes números.

A lei dos grandes números nos diz que para variáveis aleatórias $(X_1,...,X_n)$ independentes e identicamente distribuídas e integravéis temos:

$$\frac{1}{n}\sum_{i=1}^{n} X_i \rightarrow^{P} E(X_1)$$

Em palavras equivale a dizer que a média aritmética da seqüência de variáveis aleatórias, $(X_1,...,X_n)$, converge em probabilidade para a sua esperança.

A PAE nos diz que $\frac{1}{n} log\left[\frac{1}{f(X_1,...,X_n)}\right]$ converge em probabilidade para a entropia H, onde $f(X_1,...,X_n)$ é a densidade de probabilidade conjunta ou função massa de probabilidade conjunta da seqüência de variáveis aleatórias.

Teorema 1 *Seja $(X_1,X_2,...)$ uma seqüência de variáveis aleatórias i.i.d , integravéis com função densidade de probabilidade (pdf) ou função massa de probabilidade (pmf) então:*

$$-\frac{1}{n} log\left(f(X_1,...,X_n)\right) = -\frac{1}{n}\sum_{i=1}^{n} log(f(X_i)) \rightarrow^{P} -E\left[log\left(f(X)\right)\right] = H(X) \qquad (3.2)$$

Nesse trabalho, estamos interessados somente no caso univariado, sendo que é possível a extensão para os casos multivariados discretos e multivariados contínuos; portanto para estimar a entropia precisamos especificar a $f(X)$ que será utilizada; trabalharemos com a estimação não-paramétrica dessa densidade de probabilidade ou da função massa de probabilidade, no caso discreto.

3.3 MEDIDA DE OTIMIZAÇÃO.

A função objetivo será aquela que é função da entropia sujeita as restrições espaciais que os conglomerados nos impõe.

Essas restrições foram comentadas brevemente no capítulo 1: Os conglomerados são conexos internamente. O que significa que um determinado conglomerado não poderá se partir em dois ou mais conglomerados do mesmo tipo e a outra restrição é que todos os conglomerados possuam um ou mais polígonos.

Como sugestão de medida a ser otimizada, utilizaremos a norma canônica do vetor $H(X_1,...,X_C)$ que é definida da seguinte maneira:

Definição 4 *Seja $H(X_1,...,X_C)$ a representação de um vetor em $\Re^C$ onde cada elemento nesse vetor representa a entropia estimada no c-ésimo conglomerado,isto é E_c, então as normas canônicas definidas nestes espaços são as chamadas normas l^k:*

$$E = ||H(X_1,...,X_C)||_k = \left(\sum_{c=1}^{C} |E_c|^k \right)^{\frac{1}{k}} \text{, com } 0 < k < \infty \tag{3.3}$$

No caso em que $k = 2$ temos a já conhecida norma euclidiana. Podemos então entender a medida $||H(X_1,...,X_C)||_k$ como uma distância entre os pontos do *hiperespaço*, nosso objetivo é minimizar essa quantidade, ou seja, encontrar os conglomerados espaciais que possuam a norma canônica miníma.

Nas aplicações aqui apresentadas, utilizamos $k = 1$ e não aplicaremos o valor absoluto, note então que podemos expressar a função objetivo, nessas condições como $\sum_{c=1}^{C} E_c$.

A idéia associada a esse objetivo é encontrar regiões que sejam mais *organizadas*, como mostraremos a seguir, a *entropia* está diretamente relacionada com o parâmetro de escala de uma densidade de probabilidade, e portanto relacionada com a variância e com a *curtose* da variável, obtendo assim, conglomerados que sejam homogêneos internamente e heterogêneos externamente.

Axioma 1 *Considere uma variável aleatória unidimensional X com função densidade de probabilidade $f(X)$. Seja X uma variável aleatória transformada em uma nova v.a Y através de uma função bijetora. Então a entropia associada a Y é dada por :*

$$H(Y) = -\int_{-\infty}^{+\infty} \left[f(x) \left| \frac{dx}{dy} \right| \right] log \left[f(x) \left| \frac{dx}{dy} \right| \right] dy$$

$$H(Y) = -\int_{-\infty}^{+\infty} f(x) log\,[f(x)]\,dx - \int_{-\infty}^{+\infty} f(x) log \left[\left| \frac{dx}{dy} \right| \right] dx$$

$$H(Y) = H(X) + \int_{-\infty}^{+\infty} f(x) log \left[\left| \frac{dx}{dy} \right| \right] dx$$

fazendo $Y = \sigma X + \mu$ temos $H(Y) = H(X) + log|\sigma|$.

Pelo axioma acima, temos que a entropia estimada em um determinado conglomerado, será função do parâmetro de escala da distribuição em estudo.

O parâmetro de locação não contribui para a comparação da entropia entre duas populações com iguais funções densidade de probabilidade, sendo que a população com menor entropia será aquela que possuir a menor variância, ou seja, a que for mais homogênea.

4 NÚCLEO ESTIMADOR.

Seja F a função distribuição de uma variável aleatória X com função massa de probabilidade $p(X)$ ou função densidade de probabilidade $f(X)$, seja também a amostra aleatória $(X_1,...,X_n)$ i.i.d de F.

O objetivo da estimação não paramétrica é estimar f, se a variável aleatória é contínua e p se tivermos uma variável aleatória discreta.

Denotaremos $\hat{f}_n$ e $\hat{p}_n$ o núcleo estimador da função densidade de probabilidade e da função massa de probabilidade, respectivamente.

O estimador irá depender do parâmetro de suavização h no caso contínuo e do parâmetro s no caso discreto, note que a escolha desse parâmetro é crucial para obtermos boas estimativas (SILVERMAN, 1986).

4.1 NÚCLEO ESTIMADOR PARA FUNÇÃO MASSA DE PROBABILIDADE.

Essa proposta é baseada no trabalho de Wang e Ryzin (1981) e o objetivo é encontrar uma classe de estimadores para a função massa de probabilidade $p_i = p(X)$ para $i = \pm 1, 2...$ que seja uma combinação linear ponderada das freqüências relativas.

Definição 5 *Uma função $W(s,i,j)$ definida em S x J x J é dita ser função de suavização (Kernel) se*

$$\sum_{j=-\infty}^{+\infty} W(s,i,j) = 1 \tag{4.1}$$

onde S é um intervalo na reta real contendo a origem, $J = (...,-1,0,1,...)$ é o conjunto de todos os inteiros, $W(s,i,j) \geq 0$ para todo $i, j \in J, s \in S$

Propriedades 1

$$W(0,i,j) = I(i,j) = \begin{cases} 0(i \neq j) \\ 1(i = j) \end{cases}$$

Aqui trataremos somente da função de suavização geométrica, Wang e Ryzin (1981) apresentam também a função uniforme.

A função de suavização geométrica é dada por :

$$W(s,i,j) = \begin{cases} \frac{1}{2}(1-s)s^{|i-j|} \text{ , se } |i-j| \geq 1 \\ 1-s \text{ , se } |i-j| = 0 \end{cases}$$

Finalmente definimos o estimador $p(X)$ como:

$$\hat{p}_i = \sum_{j=-\infty}^{+\infty} W(s_n,i,j)Y_n(j) \tag{4.2}$$

onde $Y_n(j) = \frac{1}{n}\sum_{k=1}^{n} I(X_k,j)$.

A escolha do parâmetro s_n é dado pela minimização do erro quadrático médio integrado $E\left[\sum_i (\hat{p}_i - p_i)^2\right]$.

Wang e Ryzin (1981) propuseram como estimador da janela global, aquela que minimiza o EQMI dado por:

$$s_n = \beta_1 \left[\frac{3}{2} + B_1 - B_2 + (n-1)\beta_0\right]^{-1} \tag{4.3}$$

onde $\beta_0 = \sum_i p_i^2 - B_1 + \frac{1}{4}B_0$, $\beta_1 = 1 - \sum_i p_i^2 + \frac{1}{2}B_1$, $B_0 = \sum_i (p_{i-1} + p_{i+1})^2$, $B_1 = \sum_i p_i(p_{i-1} + p_{i+1})$ e $B_2 = \sum_i p_i(p_{i-2} + p_{i+2})$.

Para os dados discretos que possuam suporte $X \in [0,...,n]$ podemos escrever B_0, B_1 e B_2 da seguinte maneira:

$$B_0 = p_1^2 + \sum_{i=1}^{n-1} (p_{i-1} + p_{i+1})^2 + p_{n-1}^2$$

$$B_1 = p_0 p_1 + \sum_{i=1}^{n-1} p_i (p_{i-1} + p_{i+1})^2 + p_n p_{n-1}$$

$$B_2 = p_0 p_2 + p_1 p_3 + \sum_{i=2}^{n-2} p_i (p_{i-1} + p_{i+1})^2 + p_{n-1} p_{n-3} + p_n p_{n-2}$$

4.2 NÚCLEO ESTIMADOR PARA FUNÇÃO DENSIDADE DE PROBABILIDADE.

No caso de uma amostra aleatória $(X_1,...,X_n)$ i.i.d proveniente de uma v.a contínua, definimos o núcleo estimador da função densidade de probabilidade como (SILVERMAN, 1986):

Definição 6

$$\hat{f}(x) = \frac{1}{nh}\sum_{i=1}^{n} k\left(\frac{x-X_i}{h}\right) \tag{4.4}$$

onde $k(.)$ é chamada função núcleo, geralmente uma f.d.p simétrica e o parâmetro h chamado janela, responsável pelo grau de suavização da estimativa.

Analogamente ao caso discreto, a escolha do parâmetro de suavização é de extrema importância para obtermos boas estimavas.

Como critério de escolha de *h* utilizaremos o erro quadrático médio integrado (EQMI) que é dado por $E\left[\int \left(\hat{f}(x) - f(x)\right)^2 dx\right]$.

Podemos reescrever o EQMI como:

$$\begin{aligned} EQMI = E\left[\int \left(\hat{f}(x) - f(x)\right)^2 dx\right] = \\ \int \left[E\left[\left(\hat{f}(x) - f(x)\right)\right]^2 dx\right] = \\ \int \left[E\left(\hat{f}(x) - f(x)\right)\right]^2 dx + \int Var\left[\hat{f}(x)\right] dx \end{aligned}$$

De acordo com Silverman (1986) podemos encontrar um valor aproximado para a janela *h* que minimiza o erro quadrático médio integrado. Esse valor é dado por :

$$h_{opt} = k_2^{-\frac{2}{5}} \left[\int k(t)^2 dt\right]^{\frac{1}{5}} \left[\int f''(x)^2 dx\right]^{-\frac{1}{5}} n^{-\frac{1}{5}} \tag{4.5}$$

Onde k_2 é a constante representada por $\int t^2 k(t)dt$ e $k(t)$ é uma função simétrica satisfazendo as seguintes condições:

- $\int k(t)dt = 1$
- $\int tk(t)dt = 0$
- $\int t^2 k(t)dt = k_2 \neq 0$

Nesse trabalho utilizaremos sempre quando não especificado, o *Kernel Gaussiano* representado por $k(t) = \frac{1}{\sqrt{2\pi}} e^{-\frac{t^2}{2}}$.

Usando o método proposto por Chiu (1991) estimaremos

$$G = \int \left[f''(x) \right]^2 dx = \frac{1}{2\pi} \int_0^{\infty} \lambda^4 |\psi(\lambda)|^2 d\lambda$$

por

$$\hat{G} = \frac{1}{\pi} \int_0^{\Lambda} \lambda^4 \left[|\hat{\psi}(\lambda)|^2 - \frac{1}{n} \right] d\lambda$$

onde $\psi(\lambda)$ é a função característica de $f(x)$, Λ é o primeiro valor de λ tal que $|\hat{\psi}(\lambda)|^2 < \frac{c}{n}$ onde aqui utilizaremos a proposta de Damasceno (2000) em que $c = 3$.

O desenvolvimento mais detalhado do cálculo dessa estimativa pode ser encontrado no trabalho de Bessegato (2006).

4.3 ESTIMAÇÃO NÃO-PARAMÉTRICA DA ENTROPIA.

No artigo de Wang e Ryzin (1981) e no trabalho Silverman (1986) encontramos as provas da convergência em probabilidade de $\hat{p}_n(i) \rightarrow p_i$ e $\hat{f}_n(x) \rightarrow f(x)$, respectivamente.

Com base nesses resultados, podemos derivar então a convergência do nosso núcleo estimador para a entropia.

Como mostrado na seção 3.2 temos

$$-\frac{1}{n} \sum_{i=1}^{n} log(f(X_i)) \rightarrow^P H(X)$$

imediatamente segue:

Teorema 2 *No caso de variáveis aleatórias discretas temos, quando* $\hat{s}_n \to 0$ *e* $n \to \infty$*, pelo teorema de Slutsky*

$$-\frac{1}{n}\sum_{i=1}^{n} log(\hat{p}(X_i)) \to^P -\frac{1}{n}\sum_{i=1}^{n} log(p(X_i)) \to^P H(X) \tag{4.6}$$

de maneira semelhante, temos para o caso contínuo:

Teorema 3 *Para variáveis aleatórias contínuas temos, quando* $h \to 0$ *,*$n \to \infty$ *e* $nh \to \infty$ *pelo teorema de Slutsky*

$$-\frac{1}{n}\sum_{i=1}^{n} log(\hat{f}(X_i)) \to^P -\frac{1}{n}\sum_{i=1}^{n} log(f(X_i)) \to^P H(X) \tag{4.7}$$

5 CONGLOMERADOS ESPACIAIS

Nesta seção, apresentaremos como podemos encontrar conglomerados espaciais, de modo que possuam mínima entropia internamente.

Acarretando assim, *clusters* em que os indivíduos que os compõe são muito semelhantes internamente ao conglomerado a que ele pertence.

5.1 O ALGORITMO

O Algoritmo será uma modificação dos métodos de otimização estocástico revisados por Collet e Rennard (2007), amplamente utilizados em situações em que o tratamento tradicional não pode ser utilizado.

O primeiro passo desse algoritmo será gerar populações iniciais de maneira aleatória. Esse passo compreende então, *sortear* de todas as possíveis configurações (partições) do mapa, uma amostra de tamanho N sem reposição.

Geramos as populações iniciais da seguinte maneira:

1. Sortear C polígonos no mapa. Esses polígonos darão início aos seus respectivos conglomerados.
2. Para cada conglomerado, sortear um número aleatório uniforme entre $[0, m]$ onde m é o número de polígonos ainda disponíveis.
3. Após o passo 3, há muitas áreas sem conglomerados, essas são classificadas de acordo com a distância ao conglomerado mais próximo, de modo que não haja quebra e nem conglomerados desconexos.

Essa metodologia de geração das populações iniciais, obviamente, não produz todos os tipos de configurações existentes, apenas um subconjunto de configurações, mas o suficiente para que possamos utilizá-las na busca da configuração ótima.

Após realizado a geração das populações iniciais, selecionaremos as *n* melhores configurações, ou seja, aquelas em que o valor de $\sum_{c \in C} \hat{E}_c$ seja o menor possível, onde $\hat{E} = \sum_{c \in C} \hat{E}_c$ é calculado através dos processos descritos no capítulo 4 utilizando núcleo estimador.

Então, para cada uma dessas *n* configurações do mapa, particionados em *C clusters*, procederemos com a *mutação* dessas configurações.

Esse passo, objetiva retirar polígonos que eventualmente não contribuem para a homogeneidade no conglomerado ou adicionar polígonos que contribuam com a função objetivo.

Para realizar então essas *mutações* é necessário listar todos os polígonos de fronteira.

Definição 7 *Polígono de fronteira será aquele polígono que é vizinho de um outro polígono, tal que ambos pertençam a conglomerados diferentes.*

Definição 8 *Dizemos que dois polígonos são vizinhos se existe pelo menos um ponto, que está contido no conjunto de pontos que formam cada um dos polígonos, tal que este ponto seja semelhante a ambos.*

Como exemplo, a figura abaixo:

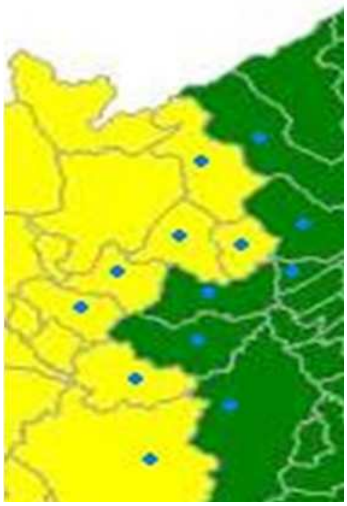

Figura 5.1: Exemplos de polígonos de fronteira.

Após ter elaborado a lista, escolhemos uma proporção desses polígonos de fronteira, que serão *mutados*, ou seja, pertencerão agora a um novo *cluster*.

No entanto, é preciso evitar que os polígonos críticos, sejam *mutados*, pois caso isso ocorra, o *cluster* não mais será conexo.

Definição 9 *Polígono crítico será aquele que for considerado um nó de articulação ou nó de separação (Cap. 2) dentro do grafo em que ele pertence, onde o grafo será composto por todos os polígonos que sejam do mesmo conglomerado.*

Realizado a mutação, avaliamos a função objetivo e se houver melhora no resultado mantemos a nova configuração, caso contrário manteremos a configuração anterior a mutação. Esse processo é realizado M vezes, e após esse período nossa melhor configuração será então o resultado final.

A cada passo do processo é necessário estimar novamente a entropia interna ao conglomerado, já que observações foram removidas ou adicionadas ao conglomerado, o que significa que as janelas estimadas, necessárias a utilização do núcleo estimador, são a cada iteração calculadas novamente e por isso dizemos que nesse contexto, a janela é estimada *localmente*.

5.1.1 ESTUDO DE CASO: CONGLOMERADOS ESPACIAIS PARA O CASO UNIVARIADO DISCRETO.

Primeiramente vamos procurar por conglomerados número de óbitos causados por neoplasias em São Paulo, a fonte desses dados é *Datasus* 2002.

Nosso objetivo é encontrar a partição do mapa, que forme a menor entropia internamente, ou seja, minimize $\hat{E} = \sum_{c \in C} \hat{E}_c$. Assim, esperamos que os conglomerados formados sejam homogêneos internamente.

Para localizar conglomerados homogêneos, utilizando a metodologia descrita anteriormente, é necessário calcular para cada conglomerado a entropia interna, tratando cada conglomerado como um estrato.

Simulamos 10000 configurações, mantivemos as 100 melhores e para 10% dos polígonos de fronteira, fizemos 1000 mutações.A distribuição de óbitos por neoplasias e a melhor configuração inicial são:

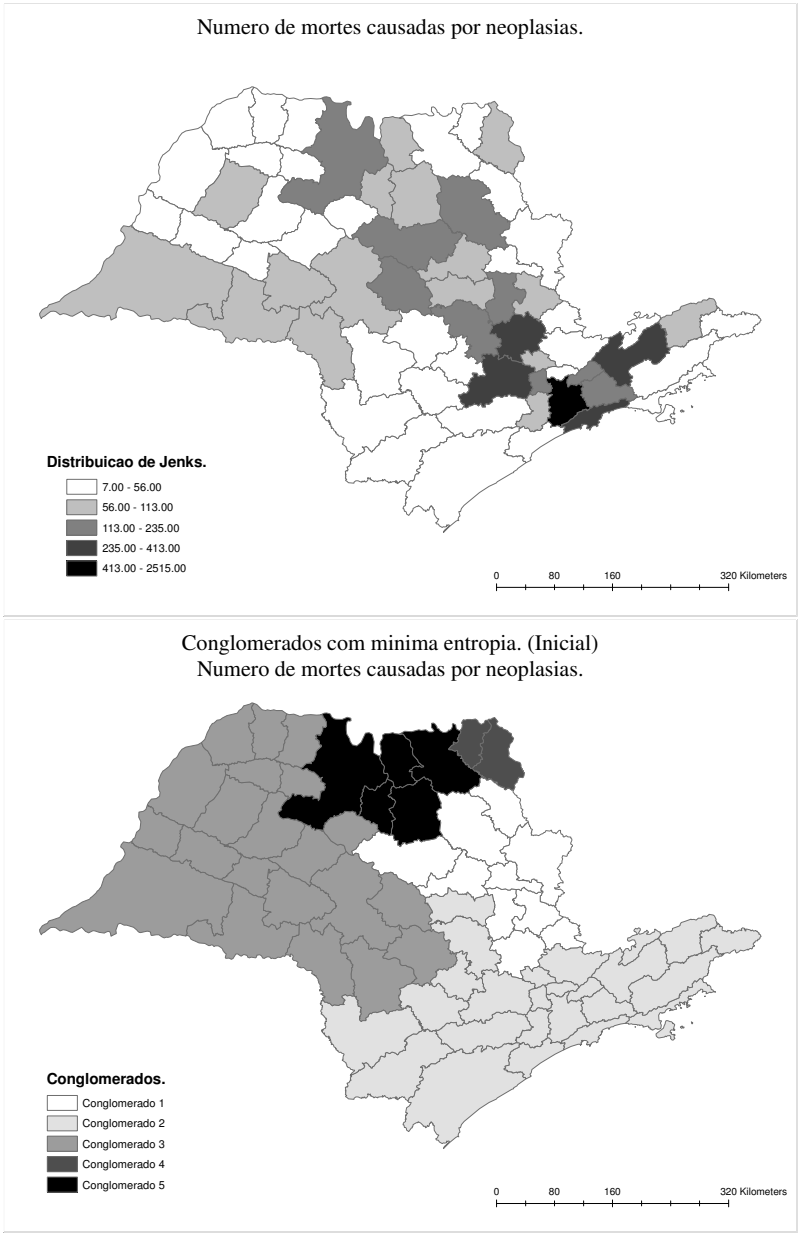

Figura 5.2: Distribuição do número de óbitos causados por neoplasias em São Paulo.

Após as mutações obtivemos:

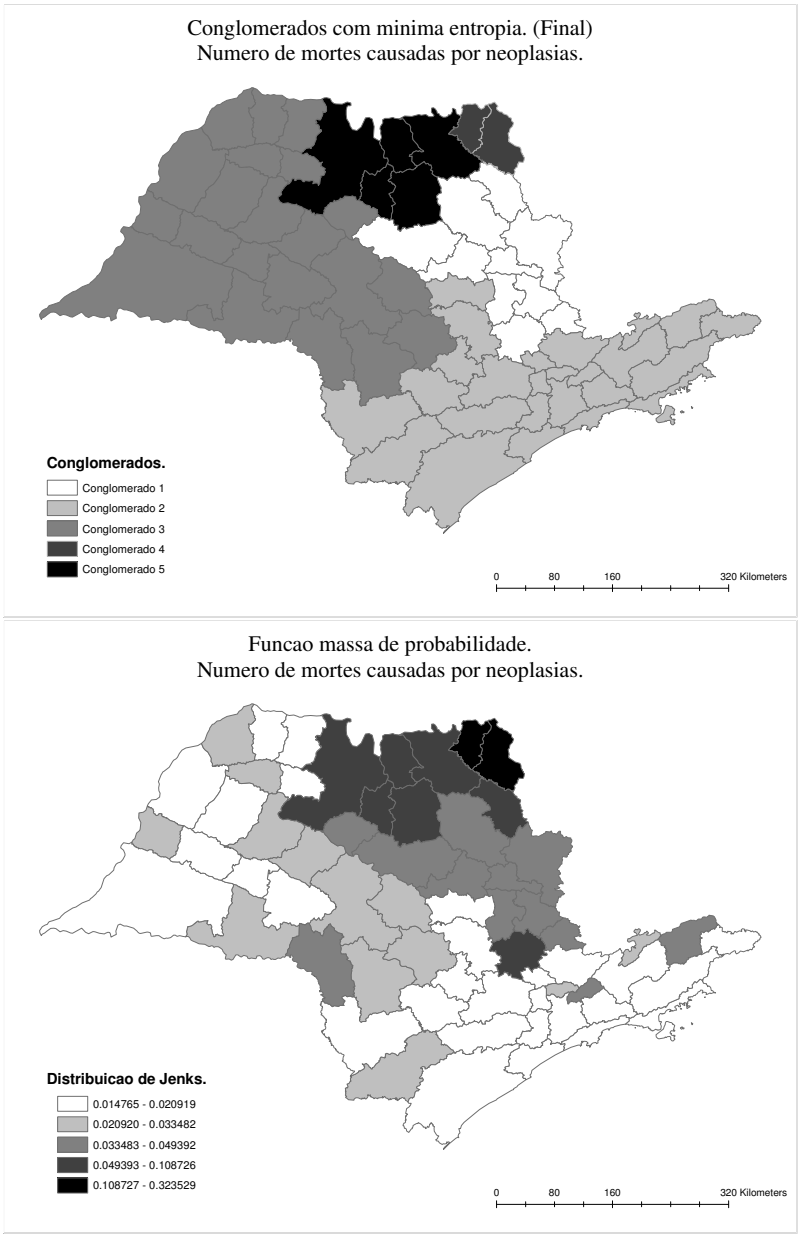

Figura 5.3: Número de óbitos causados por neoplasias em São Paulo.

Podemos perceber que há uma certa semelhança entre o mapa de probabilidade e a configuração ótima encontrada.

O método que criou as classes é denominado distribuição de *Jenks* ou quebra natural (*Natural Break*).

O método da quebra natural de Jenks (1967) tem como objetivo encontrar os intervalos de C classes de modo a minimizar a variância dentro das classes, portanto, através dessa metodologia, o mapa de probabilidade pode ser interpretado como um mapa divido em 5 grupos de modo que esses grupos possuem probabilidades estimadas semelhantes.

Os mapas de probabilidade são muito ilustrativos quanto a distribuição espacial de eventos. O caso paramétrico com variáveis aleatórias discretas foi proposto por Choynowski (1959) e também pode ser encontrado em Cressie (1993).

O escopo desse trabalho não é a análise de mapas de probabilidade, mas sim a busca por conglomerados espaciais, entretanto, deixamos registrado esse resultado: a aparente relação entre o mapa de probabilidades e a configuração final, pois caso essa relação seja confirmada em estudos posteriores, poderemos usar esse resultado de maneira a auxiliar a mutação.

No nosso estudo de caso para a variável número de óbitos causados por neoplasias, as mutações não auxiliaram na formação do conglomerado, já que após as 1000 simulações, a melhor configuração inicial se manteve inalterada.

Conceitualmente, esse algoritmo foi proposto para encontrar regiões homogêneas e não conglomerados que se destaquem da região, por isso, a população interna ao conglomerado não é levada em consideração o que acarreta, por exemplo, que se dois polígonos possuírem o mesmo número de óbitos causados por neoplasias, espera-se que eles sejam classificados no mesmo conglomerado ainda que a população de um dos polígonos seja muito superior a população do outro, entretanto essa abordagem poderia ser alterada utilizando a entropia conjunta entre a população e o número de casos, por exemplo.

Os resultados obtidos para esse estudo de caso foram:

Conglomerado	Janela estimada	Entropia (Nats)	Média	Variância
Conglomerado 1	0.5764	2.9814	118.10	13761.88
Conglomerado 2	0.6308	3.9712	191.40	243578.00
Conglomerado 3	0.6240	3.7222	50.04	1164.65
Conglomerado 4	0.3529	1.1284	46.50	684.50
Conglomerado 5	0.5783	2.3343	106.40	4345.30
Global	-	17.6488	121.30	100854.12

Tabela 5.1: Resultados: Caso univariado discreto - óbitos por neoplasias

O conglomerado número 4 representa os polígonos que possuem menor quantidade de casos por óbitos de neoplasias em quanto o conglomerado número 2 possui a maior média, em contra partida possui a maior variância.

O conglomerado mais homogêneo é o conglomerado número 4, pois é o que possui a menor entropia interna estimada (1.1284 nats).

Já o conglomerado mais heterogêneo é o conglomerado número 2, que deve-se ao fato da cidade de São Paulo ser um valor extremo dos casos de neoplasias.

Note que no conglomerado número 3 a entropia é alta, comparada com os outros conglomerados, mas a variância é relativamente baixa.

Abaixo o *box-plot* dos resultados do caso univarido discreto para o número de óbitos causados por neoplasias em 2002 segundo o *Datasus*.

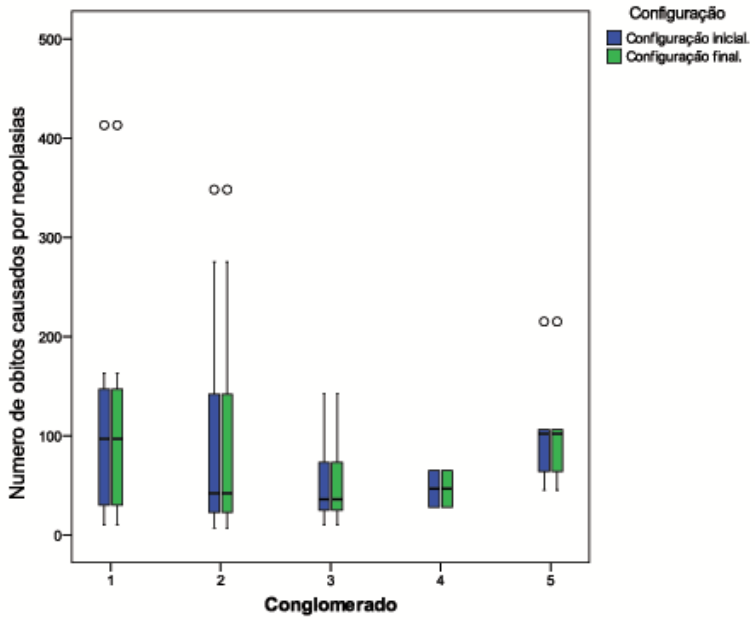

Figura 5.4: Box-plot - Conglomerados: Número de óbitos causados por neoplasias.

Fica evidente no entanto que o conglomerado 2 é fortemente influenciado pelos valores extremos (*outliers*) oriundos da cidade de São Paulo, cujo o valor é de 2515 casos de óbitos por neoplasias, que foi omitido propositalmente no *box-plot*, para que o gráfico ficasse mais explicativo.

O conglomerado 1 representa os polígonos com a maior mediana, seguido pelo conglomerado número 5 e com as menores quantidades medianas, temos os conglomerados 2 e 3 respectivamente.

O conglomerado número 2 é um caso especial por conter a cidade de São Paulo, esse fato contribui bastante para o aumento na variância interna do conglomerado e conseqüentemente um acréscimo na entropia estimada.

5.1.2 ESTUDO DE CASO: CONGLOMERADOS ESPACIAIS PARA O CASO UNIVARIADO CONTÍNUO.

Assim como no caso discreto univariado procuramos por conglomerados, nesse caso, a variável de interesse é o Índice de Desenvolvimento Humano em 2000, cuja a fonte é o Instituto

Brasileiro de Geografia e Estatística - IBGE.

Nosso objetivo aqui é encontrar a partição do mapa que forme a menor entropia internamente, o número de conglomerados que definimos também será igual a cinco.

A distribuição do índice de desenvolvimento humano e a melhor configuração inicial, usando os mesmos parâmetros do caso discreto, são:

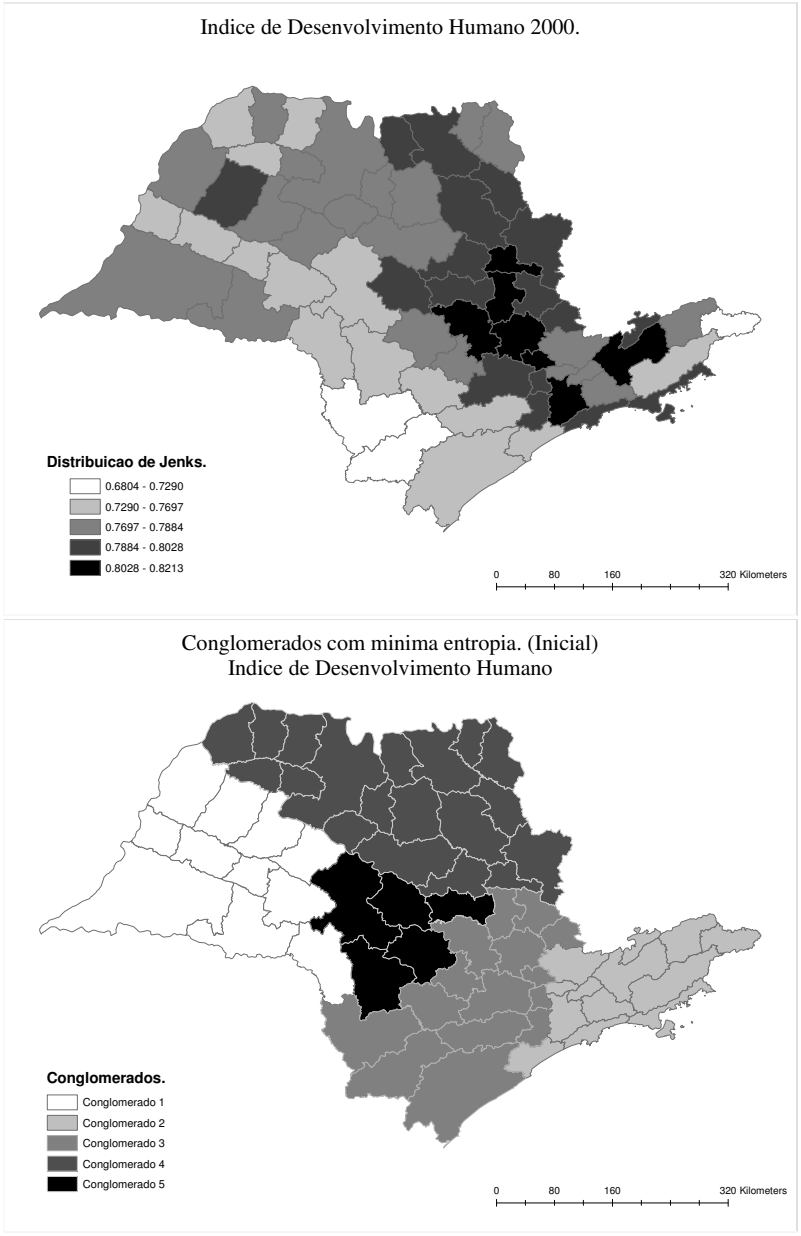

Figura 5.5: Distribuição do índice de desenvolvimento humano em São Paulo.

Após as 1000 mutações, obtivemos:

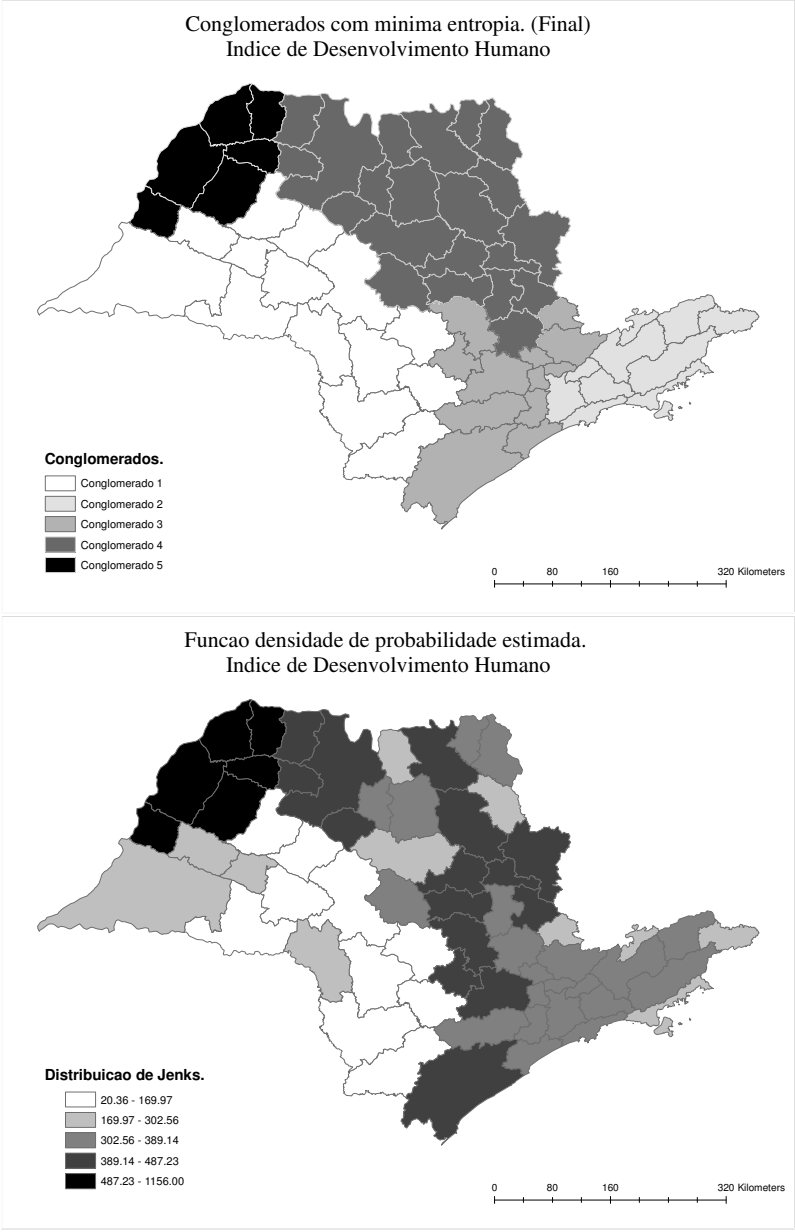

Figura 5.6: Índice de desenvolvimento humano em São Paulo.

Diferentemente do caso discreto univariado as mutações foram essenciais para a formação do conglomerado.

Os resultados para a formação dos conglomerados são apresentadas a seguir:

Conglomerado	Janela estimada	Entropia (Nats)	Média	Variância
Conglomerado 1	0.0113	-4.7619	0.7598	0.00082
Conglomerado 2	0.0156	-5.7816	0.7852	0.00080
Conglomerado 3	0.0169	-5.8977	0.7856	0.00037
Conglomerado 4	0.0163	-5.9549	0.7920	0.00016
Conglomerado 5	0.0254	-7.0313	0.7720	0.00017
Global	-	-35.1817	0.7806	0.00058

Tabela 5.2: Resultados: Caso univariado contínuo - IDH 2000

Os conglomerados número 2 e número 3 possuem índice de desenvolvimento humano muito semelhantes, entretanto a variância do segundo conglomerado é maior do que o dobro da variância do terceiro conglomerado.

O mapa de probabilidades, assim com no caso discreto, se assemelha bastante a formação de conglomerados espaciais que encontramos através das 10000 simulações de configurações inicias.

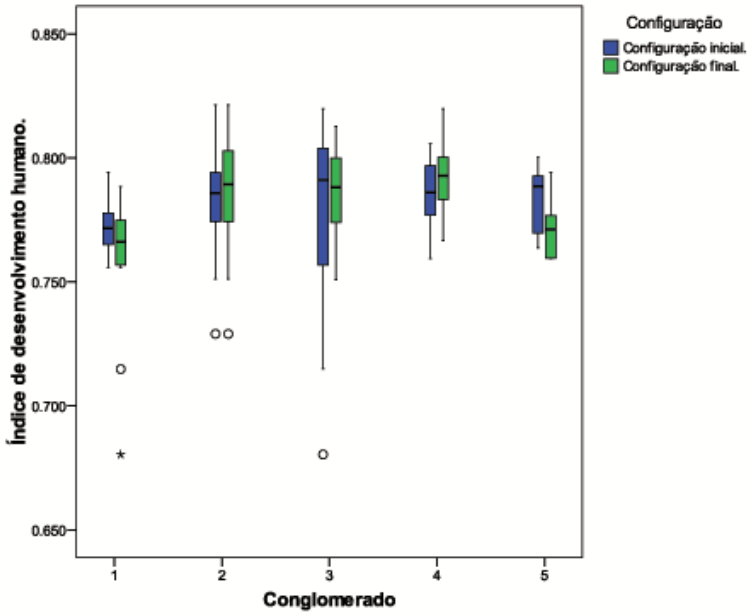

Figura 5.7: Box-plot - Conglomerados: IDH 2000.

Após as mutações, houve uma mudança significativa nas distribuições do índice de desenvolvimento humano, em geral, a variância interna ao *cluster* foi reduzida, o que nos indica maior similaridade internamente a esses conglomerados, comparado com a configuração inicial.

O conglomerado que possui menor entropia - conseqüentemente o mais *organizado* internamente - é o conglomerado número cinco; esse conglomerado também foi o mais modificado pela mutação, na segunda parte do algoritmo de *clusterização*, se movendo para oeste do mapa.

6 SIMULAÇÕES

Para testar o poder da metodologia de *clusterização* espacial, apresentada nesta dissertação, simulamos nesta seção, alguns casos:

Primeiramente quando a entropia total do sistema é conhecida e em seguida quando a configuração ótima é conhecida.

A primeira situação tem como objetivo, mensurar a diminuição do *caos* em relação a entropia existente no sistema.

No segundo caso, como a configuração ótima é conhecida, espera-se que após a metodologia, a configuração gerada seja o mais próxima possível da configuração ótima, já conhecida a *priori*.

6.1 SIMULAÇÃO DE UM SISTEMA ESPACIAL CAÓTICO.

Inicialmente, simulamos uma configuração espacial com 5 conglomerados, todos gerados através de uma distribuição Normal.

Definição 10 *A entropia de uma variável aleatória com distribuição de Normal(0,σ^2) é dada por*

$$\begin{aligned} H(X) = -\int_{-\infty}^{+\infty} log\left(f(x)\right) dF(x) = \\ -\int_{-\infty}^{+\infty} log\left(f(x)\right) f(x) dx = \\ -\int_{-\infty}^{+\infty} f(x)\left[-\frac{x^2}{\sigma^2} - log\left(\sqrt{2\pi\sigma^2}\right)\right] = \\ \frac{E(X^2)}{2\sigma^2} + \frac{1}{2} log(2\pi\sigma^2) = \\ \frac{1}{2} + \frac{1}{2} log(2\pi\sigma^2) = \end{aligned}$$

$$\frac{1}{2}log(e)+\frac{1}{2}log(2\pi\sigma^2)= \\ \frac{1}{2}log(2\pi\sigma^2 e) \tag{6.1}$$

Note que, como mostrado no capítulo 3, a entropia não depende do parâmetro de locação.

Para a seguinte configuração:

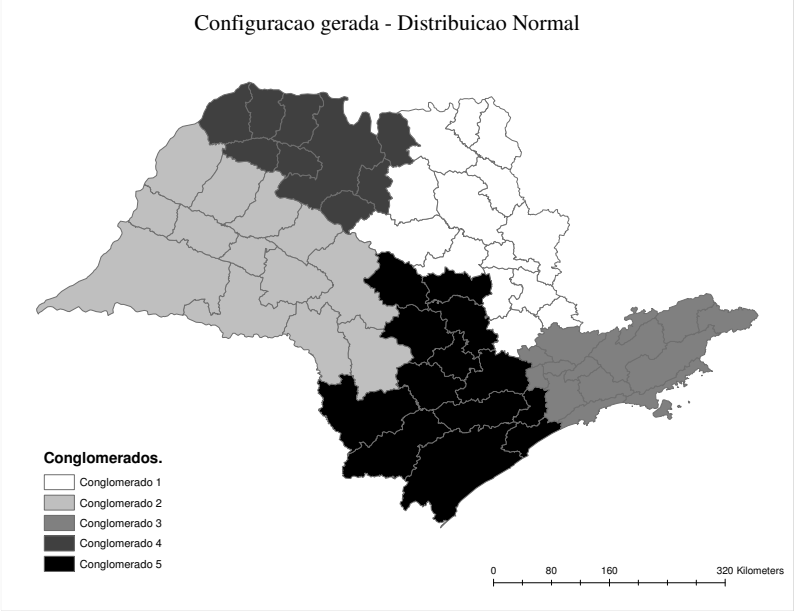

Figura 6.1: Configuração contínua - Simulada.

geramos as variáveis aleatórias com distribuição normal e parâmetros definidos como:

Conglomerado	Entropia (Nats)	Média	Variância
Conglomerado 1	0.6646	5.00	1.25
Conglomerado 2	0.8152	10.00	2.5
Conglomerado 3	1.0537	30.00	7.5
Conglomerado 4	1.1646	50.00	12.5
Conglomerado 5	1.3152	100.00	25.00
Global	5.0135	-	-

Tabela 6.1: Parâmetros: Caso univariado contínuo - Simulado.

Após as 10000 gerações de configurações iniciais e 1000 mutações em 10% dos polígonos de fronteira obtivemos a seguinte configuração final:

Conglomerado	Entropia (Nats)	Média	Variância	Janela
Conglomerado 1	-0.5322	6.9071	0.5790	1.3837
Conglomerado 2	-0.1418	11.9778	3.6924	1.4853
Conglomerado 3	-7.7379	33.7572	5.4084	0.8894
Conglomerado 4	0.4960	53.1808	10.0888	1.4392
Conglomerado 5	-0.5885	107.8889	18.2681	1.1015
Global	-8.5044	-	-	-

Tabela 6.2: Parâmetros: Caso univariado contínuo - Encontrado.

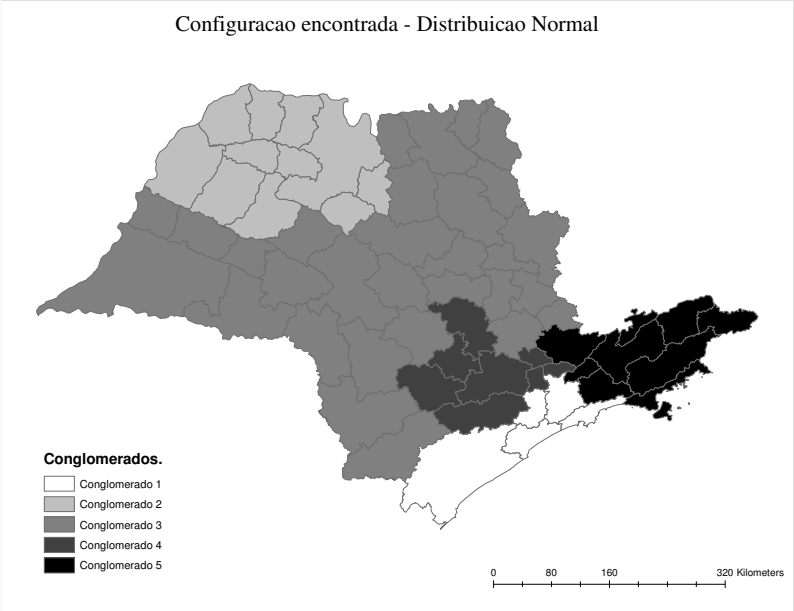

Figura 6.2: Configuração contínua (Final) - Simulada.

Globalmente, a entropia do sistema reduziu-se de 5.0135 *nats* à -8.5044 *nats*. Todos os conglomerados, exceto o *cluster* número 2 tiveram a sua variância reduzida, e em todos os casos a entropia interna aos conglomerados foi reduzida.

Procedendo de maneira semelhante, para a seguinte configuração:

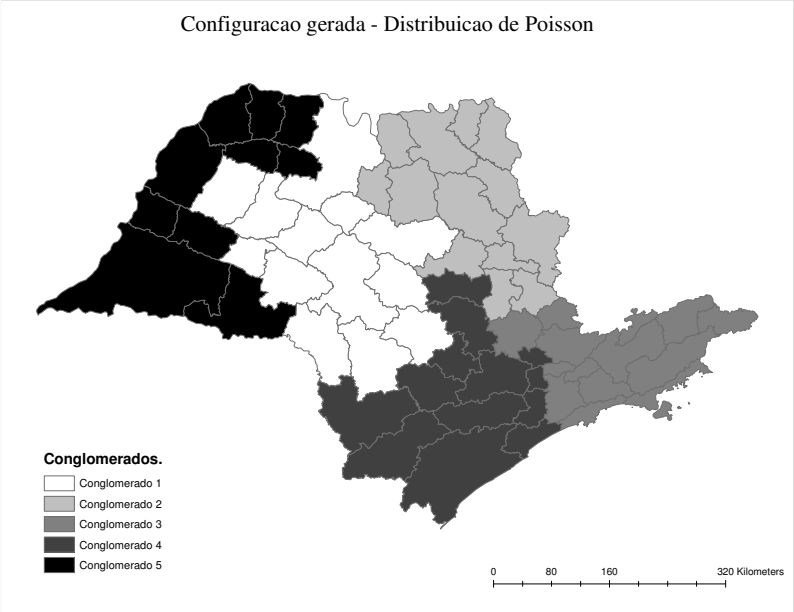

Figura 6.3: Configuração discreta - Simulada.

geramos números aleatórios com distribuição de Poisson.

Definição 11 *A entropia de uma variável aleatória com distribuição de Poisson(λ) é dada por Ronald (1988)*

$$H(X) = -\int_{-\infty}^{+\infty} log\left(f(x)\right) dF(x) =$$
$$-\sum_{i=0}^{+\infty} log\left[p(X_i)\right] p(X_i) =$$
$$-\sum_{i=0}^{+\infty} \left[-\lambda + X_i log(\lambda) - log(X_i!)\right] \frac{e^{-\lambda}\lambda^{X_i}}{X_i!} \approx$$
$$\frac{1}{2}\log(2\pi e\lambda) - \frac{1}{12\lambda} - \frac{1}{24\lambda^2} - \frac{19}{360\lambda^3} + O\left(\frac{1}{\lambda^4}\right) \tag{6.2}$$

Os parâmetros das distribuições estão apresentados na seguinte tabela:

Conglomerado	Entropia (Nats)	Média
Conglomerado 1	0.9469	5.00
Conglomerado 2	1.1074	10.00
Conglomerado 3	1.3519	30.00
Conglomerado 4	1.4640	50.00
Conglomerado 5	1.6153	100.00
Global	6.4858	-

Tabela 6.3: Parâmetros: Caso univariado discreto - Simulado.

Nesse caso, para o cálculo da entropia ignoramos o fator $O\left(\frac{1}{\lambda^4}\right)$.

Após o processo de *clusterização* espacial, já descrito anteriormente, obtivemos:

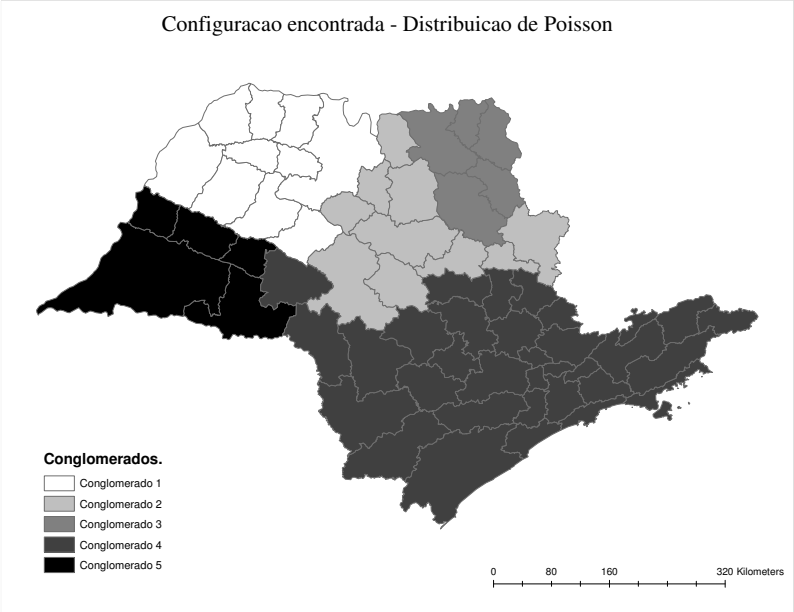

Figura 6.4: Configuração discreta (Final) - Simulada.

Conglomerado	Entropia (Nats)	Média	Variância	Janela
Conglomerado 1	0.6067	5.3846	4.4230	0.4236
Conglomerado 2	0.6832	9.8461	10.6410	0.3948
Conglomerado 3	0.7056	29.5000	38.7307	0.5783
Conglomerado 4	0.7756	49.3846	78.9230	0.5395
Conglomerado 5	0.5594	99.5000	46.2777	0.4356
Global	3.4005	-	-	-

Tabela 6.4: Parâmetros: Caso univariado discreto - Encontrado.

Todos os conglomerados tiveram a entropia reduzida, o que sugere a eficiência do algoritmo em criar conglomerados que possuam maior homogeneidade internamente.

6.2 SIMULAÇÃO DE UM SISTEMA ESPACIAL ORGANIZADO.

Usando o método de geração de população descrito na seção 5, simulamos uma configuração espacial, onde a variabilidade no conglomerado é zero, isto é, todos os valores são exatamente os mesmos para cada um dos conglomerados.

Conglomerado	Média	Variância
Conglomerado 1	1.00	0
Conglomerado 2	2.00	0
Conglomerado 3	3.00	0
Conglomerado 4	4.00	0
Conglomerado 5	5.00	0
Global	-	-

A configuração gerada é apresentada a seguir:

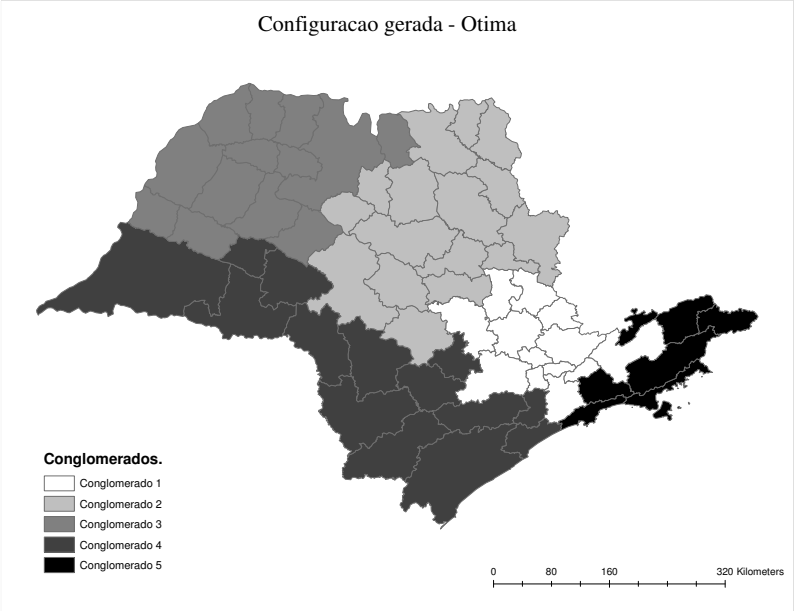

Figura 6.5: Configuração ótima gerada.

Após o precedimento de *clusterização* obtivemos a seguinte configuração

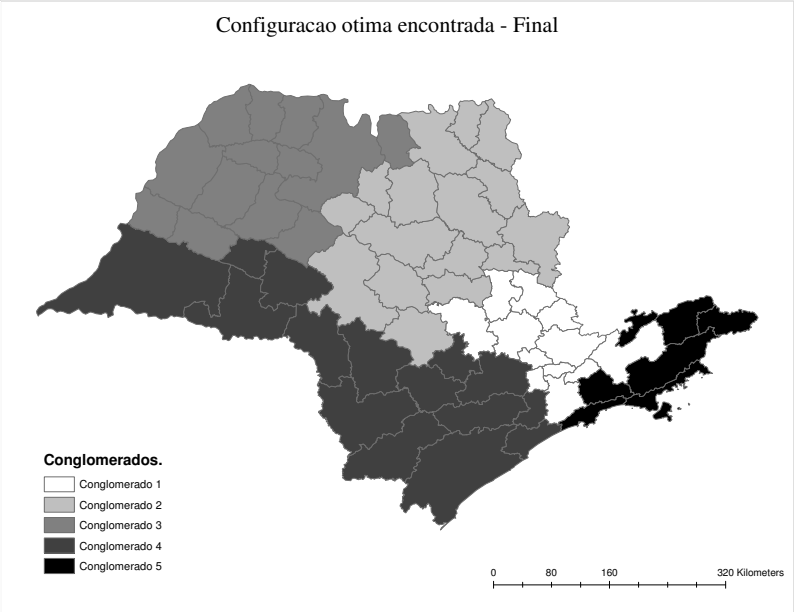

Figura 6.6: Configuração ótima encontrada.

Percebe-se que a configuração é praticamente a mesma, exceto por exatamente um polígono, o qual foi alocados em outro conglomerado a que não pertence.

7 CONCLUSÕES

Como uma nova proposta de formação de conglomerados espaciais, ainda há muitas possíveis modificações que essa proposta pode sofrer, a fim de melhorar o processo de *clusterização*.

A convergência das estimativas para a suas estatísticas pode ser prejudicada pela pequena quantidade de polígonos existentes em alguns conglomerados, isso pode ser suprido supondo uma distribuição paramétrica para o processo espacial ou trabalhando com regiões mais desagregadas, como municípios ou setores censitários.

É importante notar que ao trabalhar com a proposta intuitiva de utilizar a medida de variância dos dados no lugar da entropia, teríamos problema com a escala da função objetivo no caso multivariado, já com a entropia não teríamos esse problema, pois é medida em *nats* para todas as variáveis.

Por ser um algoritmo de otimização estocástica, para que o processo seja eficiente, é necessário milhares de iterações, o que pode ser inviável se o número de regiões em estudo for muito alto.

Esse método produz conglomerados espaciais irregulares, e todas as áreas são classificadas em algum tipo de conglomerado, diferentemente do que a maioria dos métodos atuais produzem.

Estes métodos usuais formam conglomerados circulares, o que é claro, pode subestimar ou superestimar a quantidade de polígonos que formam o *cluster*, além das propostas serem conceitualmente diferentes do método aqui apresentado.

A escolha do número de conglomerados que se deseja formar é arbitrário, cabendo ao pesquisador através de análises adicionais ou conveniência própria, escolher a quantidade desejada.

O valor de k escolhido na formula 3.3 foi igual a 1. Outras propostas, podem ser apresentadas afim de ponderar o valor da função objetivo, no caso multivariado.

7.1 TRABALHOS FUTUROS

Aos interessados em continuar pesquisando essa área, sugerimos adaptar o processo para conglomerados multivariados contínuos, multivariado discreto e o caso mais abrangente, mistura de variáveis.

A formação desses métodos pode ser implementada usando a teoria descrita nas seções anteriores, alterações na maneira como são geradas as populações iniciais e as mutações poderão também diminuir a quantidade de iterações necessárias, tornando o método mais viável em regiões com muitos polígonos.

Outra proposta interessante seria adaptar esse processo através da criação de conglomerados espaciais de maneira hierárquica, o que evitaria as milhares de iterações necessárias para que o processo se torne ótimo; um estudo de conglomerados espaciais hierárquicos foi realizado por Carvalho (2007), e poderia servir como base a essa abordagem do problema de *clusterização* espacial.

APÊNDICE A – FUNÇÕES EM C#

Para a geração dos conglomerados espaciais, utilizamos o compilador *Visual Studio Express 2005* que é gratuito e pode ser disponibilizado livremente para aqueles que programam nas linguagens *C# ,C++, J#* e *VB.*

O objeto que nos auxiliou a gerar a matriz de vizinhança dos polígonos foi o objeto *MapControl* disponibilizado pela empresa *Dundas.com*, esse objeto foi utilizado em sua versão demonstrativa e não pode ser comercializado, apenas utilizado em programas testes, pesquisas e trabalhos acadêmicos.

Esse objeto não é essencial aos métodos aqui explicitados, qualquer outro que gere uma matriz de vizinhança de uma arquivo *shapefile* pode substituir o objeto *mapControl* sem perda de generalidade.

Um método *open source* que permite trabalhar com mapas é o componente *Sharp Map*, mas até a finalização desse trabalho esse componente ainda não possuía a função que informasse se dois polígonos são ou não vizinhos, essa função é essencial para a criação das matrizes de contiguidade.

Parte destas funções tem como referência o livro de Vetterling e Flannery (2002).

A.1 FUNÇÕES: GERAIS.

```
public class Ran1
{
   private long IA = 16807;
   private long IM = 2147483647;
   private double AM = 1.0 / 2147483647.0;
   private long IQ = 127773;
   private long IR = 2836;
```

```
private int NTAB = 32;
private double NDIV = 1.0 + (2147483647.0 - 1.0) / 32.0;
private double RNMX = 1.0 - 1.2e-7;
private long iy = 0;
private long[] iv = new long[32];
private long idum = 1;
public long Idum
{
    get { return idum; }
    set { idum = value; }
}
public Ran1()
{
}
public Ran1(long a)
{
    idum = a;
}
public double ran1()
{
    int j;
    long k;
    double temp;
    if (idum <= 0 || iy == 0)
    {
        if (-idum < 1) idum = 1;
        else idum = -idum;
        for (j = NTAB + 7; j >= 0; j--)
        {
            k = idum / IQ;
            idum = IA * (idum - k * IQ) - IR * k;
            if (idum < 0) idum += IM;
            if (j < NTAB) iv[j] = idum;
        }
        iy = iv[0];
```

```
        }
        k = idum / IQ;
        idum = IA * (idum - k * IQ) - IR * k;
        if (idum < 0) idum += IM;
        j = Convert.ToInt32(iy / NDIV) % NTAB;
        iy = iv[j];
        iv[j] = idum;
        if ((temp = AM * iy) > RNMX) return RNMX;
        else return temp;
    }
}

private bool poligonosSaoVizinhos(Shape shape0, Shape shape1)
{
  for (int i = 0; i < shape0.ShapeData.Points.Length; i++)
  {
     for (int j = 0; j < shape1.ShapeData.Points.Length; j++)
     {
         double X0 = shape0.ShapeData.Points[i].X;
         double X1 = shape1.ShapeData.Points[j].X;
         double Y0 = shape0.ShapeData.Points[i].Y;
         double Y1 = shape1.ShapeData.Points[j].Y;

         string sX0 = X0.ToString("#00.000");
         string sX1 = X1.ToString("#00.000");
         string sY0 = Y0.ToString("#00.000");
         string sY1 = Y1.ToString("#00.000");

         if ((sX0 == sX1) && (sY0 == sY1))
         {
             return (true);
         }
     }
  }
  return (false);
```

```
}

private bool poligonosSaoVizinhos(MapControl mapControl, int shape0, int shape1)
{
 for (int i = 0; i < mapControl.Shapes[shape0].ShapeData.Points.Length; i++)
 {
     for (int j = 0; j < mapControl.Shapes[shape1].ShapeData.Points.Length; j++)
     {
         double X0 = mapControl.Shapes[shape0].ShapeData.Points[i].X;
         double X1 = mapControl.Shapes[shape1].ShapeData.Points[j].X;
         double Y0 = mapControl.Shapes[shape0].ShapeData.Points[i].Y;
         double Y1 = mapControl.Shapes[shape1].ShapeData.Points[j].Y;

         string sX0 = X0.ToString("#00.000");
         string sX1 = X1.ToString("#00.000");
         string sY0 = Y0.ToString("#00.000");
         string sY1 = Y1.ToString("#00.000");

         if ((sX0 == sX1) && (sY0 == sY1))
         {
             return (true);
         }
     }
 }
 return (false);
}



private bool poligonosSaoVizinhosKpontos(Shape shape0, Shape shape1, double k)
{
 double dummy = 0;
 for (int i = 0; i < shape0.ShapeData.Points.Length; i++)
 {
     for (int j = 0; j < shape1.ShapeData.Points.Length; j++)
```

```
    {
        if ((shape0.ShapeData.Points[i].X == shape1.ShapeData.Points[j].X) &&
      (shape0.ShapeData.Points[i].Y == shape1.ShapeData.Points[j].Y)) dummy += 1;
        if (dummy >= k)
        {
            return (true);
        }
    }
 }
 return (false);
}

private bool poligonosSaoVizinhosLado(Shape shape0, Shape shape1)
{
 for (int i = 0; i < shape0.ShapeData.Segments.Length; i++)
 {
    for (int j = 0; j < shape1.ShapeData.Segments.Length; j++)
    {
        if ((shape0.ShapeData.Segments[i].ToString() ==
            shape1.ShapeData.Segments[j].ToString()))
        {
            return (true);
        }
    }
 }
 return (false);
}

private bool existeIntersecao(int[,] intTodos, int[] vetor, int coluna)
{
 if (coluna > 0)
 {
    for (int i = 0; i < coluna; i++)
    {
        for (int j = 0; j < intTodos.GetLength(0); j++)
```

```
        {
            if (intTodos[j, i] == vetor[j])
            {
                return (true);
            }
            else if (intTodos[j, i] < 0 || vetor[j] < 0) break;
        }
    }
 }
 return (false);
}
private bool existeIntersecao(int[,] intTodos, int vetor, int coluna)
{
 if (coluna > 0)
 {
    for (int i = 0; i < coluna; i++)
    {
        for (int j = 0; j < intTodos.GetLength(0); j++)
        {
            if (intTodos[j, i] == vetor)
            {
                return (true);
            }
            else if (intTodos[j, i] < 0 || vetor < 0) break;
        }
    }
 }
 return (false);
}

private double distanciaEuclidiana(MapCoordinate x0, MapCoordinate
                          x1, MapCoordinate y0, MapCoordinate y1)
{
  return (Math.Sqrt(Math.Pow(x0.ToDouble() - x1.ToDouble(), 2)
          + Math.Pow(y0.ToDouble() - y1.ToDouble(), 2)));
```

```
}


private int maximo(int[] vetor)
{
  int maximo = vetor[0];
  for (int i = 1; i < vetor.Length; i++)
  {
      if (maximo < vetor[i]) maximo = vetor[i];
  }
  return (maximo);
}


private int minimo(int[] vetor)
{
 int minimo = vetor[0];
 for (int i = 1; i < vetor.Length; i++)
 {
     if (minimo > vetor[i]) minimo = vetor[i];
 }
 return (minimo);
}


private int minimoBusca(int[] vetor)
{
 int minimo = int.MaxValue;
 for (int i = 0; i < vetor.Length; i++)
 {
     if (vetor[i] != -1)
     {
         minimo = vetor[0];
         break;
     }
```

```
 }

 for (int i = 1; i < vetor.Length; i++)
 {
     if (vetor[i] != -1) if (minimo > vetor[i]) minimo = vetor[i];
 }
 return (minimo);
}
```

A.2 FUNÇÕES: GRAFOS.

```
private bool verificaArestas1(MapControl mapControl, bool[,]
matrizDeVizinhaca, bool[,] verdade, int linha, ref int intNaoUsada)
{
    for (int i = 0; i < verdade.GetLength(0); i++)
    {
        if (linha != i && matrizDeVizinhaca[linha, i] == true
         && verdade[linha, i] == true
           && mapControl.Shapes[linha]["Cluster"].ToString()
          == mapControl.Shapes[i]["Cluster"].ToString())
        {
            intNaoUsada = i;
            return (false);
        }
    }
    return (true);
}


private bool verificaArestas2(MapControl mapControl,
 bool[,] matrizDeVizinhaca, bool[,] verdade, int linha,
ref int intNaoUsada)
```

```
{
    //True= A aresta não está usada.
    //False= A aresta está usada
    for (int i = 0; i < verdade.GetLength(0); i++)
    {
        if (matrizDeVizinhaca[i, linha]== true  && linha != i)
        {
            if ((verdade[linha, i] == false) &&
           (mapControl.Shapes[linha]["Cluster"].ToString()
            == mapControl.Shapes[i]["Cluster"].ToString()))
            {
                intNaoUsada = i;
                return (false);
            }
        }
    }
    return (true);
}


private bool verificaArestas3(MapControl mapControl,
 bool[,] matrizDeVizinhanca, bool[,] verdade,  int linha)
{
    for (int i = 0; i < verdade.GetLength(0); i++)
    {
        if ((verdade[linha, i] == false &&
           matrizDeVizinhanca [linha,i]==true) && (linha != i)
         && (mapControl.Shapes[linha]["Cluster"].ToString()
         == mapControl.Shapes[i]["Cluster"].ToString()))
                {
                    return (false);
                }
            }
            return (true);
        }
```

```
private bool verificaArestas4(MapControl mapControl,
bool[,] matrizDeVizinhanca, bool[,] verdade, int linha,
string strCluster)
{
    for (int i = 0; i < verdade.GetLength(0); i++)
  {
        if ((matrizDeVizinhanca[linha, i] == true &&
        verdade[linha, i] == false) &&
       (mapControl.Shapes[linha]["Cluster"].ToString()
       == strCluster && mapControl.Shapes[i]["Cluster"].ToString()
        == strCluster))
         {
             return (false);
         }
     }
     return (true);
 }

private bool verificaArestas5(MapControl mapControl,
bool[,] matrizDeVizinhanca,bool[,]
 verdade, int linha, ref int intNaoUsada, string strCluster)
{
    for (int i = 0; i < verdade.GetLength(0); i++)
    {
        if ((matrizDeVizinhanca[linha, i] == true
       && verdade[linha, i] == false) &&
      (mapControl.Shapes[linha]["Cluster"].ToString()
       == strCluster &&
         mapControl.Shapes[i]["Cluster"].ToString()
          == strCluster))
        {
            intNaoUsada = i;
            return (false);
```

```
        }
    }
    return (true);
}

private bool valorExistenteNaMatriz(int[,] vetor,
int coluna, int valor)
{
    for (int i = 0; i < vetor.GetLength(0); i++)
    {
        if (vetor[i, coluna] == valor) return (true);
    }
    return (false);
}

private int escolheAresta(ref bool[,] verdade, int linha)
{
    for (int i = 0; i < verdade.GetLength(0); i++)
    {
        if (verdade[linha, i] == false)
        {
            verdade[linha, i] = true;
            return (i);
        }
    }
    return (-1);
}
private int escolheAresta(MapControl mapControl,
ref bool[,] verdade, int linha)
{
    for (int i = 0; i < verdade.GetLength(0); i++)
    {
        if (verdade[linha, i] == false &&
         mapControl.Shapes[linha]["Cluster"].ToString()
            == mapControl.Shapes[i]["Cluster"].ToString())
```

```
        {
            verdade[linha, i] = true;
            verdade[i, linha] = true;
            return (i);
        }
    }
    return (-1);
}

private bool vetorCompleto(int[] poligonosIniciais)
{
    for (int i = 0; i < poligonosIniciais.Length; i++)
    {
        if (poligonosIniciais[i] == -1) return (false);
    }
    return (true);
}


private bool shapeNoSeparavel(int shape, int[,] poligonosSeparaveis,
int intClsuter)
{
    for (int i = 0; i < poligonosSeparaveis.GetLength(0); i++)
    {
        if (poligonosSeparaveis[i, intClsuter] == -1) return (false);
        if (poligonosSeparaveis[i, intClsuter] == shape) return (true);
    }
    return (false);
}

private bool shapeNoSeparavel(int shape, int[] poligonosSeparaveis)
{
    for (int i = 0; i < poligonosSeparaveis.Length; i++)
    {
        if (poligonosSeparaveis[i] == -1) return (false);
```

```
        if (poligonosSeparaveis[i] == shape) return (true);
    }
    return (false);
}

private int MatrixVizinho(MapControl mapControl,
ref List<string> mMatrixShape1, ref List<string>
mMatrixShape2, Shape mShape, int mPoligonoNum,
string mFieldName, double mFieldValue, bool[,] MatrizDeVizinhanca)
{
    int mPoligonoFind = 0;

    //
    int b = mapControl.Shapes.GetIndex(mShape.Name);

    for (int f = 0; f < mMatrixShape1.Count; f++)
    {
        if (mPoligonoFind == mPoligonoNum)
            return mPoligonoFind;
        //
        int s = mapControl.Shapes.GetIndex(mMatrixShape1[f]);

        Shape mShapeTemp = mapControl.Shapes[s];

        //

        if (MatrizDeVizinhanca[b, s] == true)
        {
            mShapeTemp[mFieldName] = mFieldValue;

            //adiciona na matrix de segundo grau
            mMatrixShape2.Add(mMatrixShape1[f]);

            //remove da matriz atual
            mMatrixShape1.Remove(mMatrixShape1[f]);
```

```
            //
            f--;

            //Voltar
            mPoligonoFind++;

        }
    }
    return mPoligonoFind;
}

/// <summary>
/// Função para econtrar os nós de articulação.
///Fonte: Graph Algorithms Autor: Shimon Even
/// </summary>
/// <param name="mapControl">Mapa</param>
/// <param name="matrizDeVizinhaca">Matriz de vizinhança</param>
/// <param name="s">Vetor de poligonos iniciais</param>
/// <returns></returns>
private int[,] NosDeSeparacao(MapControl mapControl,
bool[,] matrizDeVizinhaca/*arestas*/, int[] s)
{
    //Nós de separação.
int[,] intNosDeSeparacao = new
int[matrizDeVizinhaca.GetLength(0), s.Length];
for (int y = 0; y < matrizDeVizinhaca.GetLength(0); y++)
for (int w = 0; w < s.Length; w++) intNosDeSeparacao[y, w] = -1;

    for (int cluster = 0; cluster < s.Length; cluster++)
    {
        bool[,] arestas = new bool[matrizDeVizinhaca.GetLength(0),
         matrizDeVizinhaca.GetLength(0)];

        string pilha = "";
```

```
        string nosDeSeparacao = "";
        //Vetor com o número em que o polígono foi descoberto.
        int[] k = new int[arestas.GetLength(0)];
        //Vetor com o polígono pai do polígono.
        int[] f = new int[arestas.GetLength(0)];
        //Lowpoint.
        int[] l = new int[arestas.GetLength(0)];
        //Inicializa os dados
        //for (int u = 0; u < k.Length; u++) f[u] = -1;

        int dummy = 0;
        int i = 0;
        int v = s[cluster];
    passo2:
        i++;
        k[v] = i;
        l[v] = i;
        pilha = v.ToString();
        string strCluster = mapControl.Shapes[v]["Cluster"].ToString();
        int strCluster0 = (cluster);

    passo3:
        int intNaoUsada = -1;
        if (verificaArestas5(mapControl, matrizDeVizinhaca,
arestas, v, ref intNaoUsada, strCluster0.ToString()) == true)
        {
            goto passo5;
        }
        //passo4:
        if (intNaoUsada != -1)
        {
            int u = intNaoUsada;
            arestas[v, u] = true;
            arestas[u, v] = true;
            if (k[u] != 0)
```

```
        {
            l[v] = Math.Min(l[v], k[u]);
            goto passo3;
        }
        else
        {
            f[u] = v;
            v = u;
            goto passo2;
        }
    }

passo5:
    if (k[f[v]] == 1)
    {
        goto passo9;
    }
    else
    {
        if (l[v] < k[f[v]])
        {
            l[f[v]] = Math.Min(l[f[v]], l[v]);
            goto passo8;
        }
        else
        {
            //Guarda o nó de separação problema no f[v]=1.
            intNosDeSeparacao[dummy, cluster] = f[v];
            nosDeSeparacao += f[v].ToString() + "\n";
            dummy++;
        }
    }

passo8:
    v = f[v];
```

```
            goto passo3;
        passo9:
            if (verificaArestas4(mapControl, matrizDeVizinhaca, arestas,
              s[cluster], strCluster0.ToString()) == true)
            {
                goto finalFor;
            }
            else
            {
                //Guarda o nó de separação
                intNosDeSeparacao[dummy, cluster] = s[cluster];
                nosDeSeparacao += s[cluster].ToString() + "\n";
                dummy++;
                v = s[cluster];
                goto passo3;
            }

        finalFor:
            int iDumy = 0;
        }

        return (intNosDeSeparacao);
    }

    /// <summary>
    /// Função para econtrar os nós de articulação.
    ///Fonte: Graph Algorithms Autor: Shimon Even
    /// </summary>
    /// <param name="mapControl">Mapa</param>
    /// <param name="matrizDeVizinhaca">Matriz de vizinhança</param>
    /// <param name="s">Poligono inicial</param>
    /// <returns></returns>
    private int[] NosDeSeparacao(MapControl mapControl,
     bool[,] matrizDeVizinhaca /*arestas*/, int s)
    {
```

```
bool[,] arestas = new bool[mapControl.Shapes.Count,
 mapControl.Shapes.Count];

    //Nós de separação.
    int[] intNosDeSeparacao = new int[arestas.GetLength(0)];
for (int y = 0; y < arestas.GetLength(0); y++)
intNosDeSeparacao[y] = -1;

    try
    {
        string pilha = "";
        string nosDeSeparacao = "";
        //Vetor com o número em que o polígono foi descoberto.
        int[] k = new int[arestas.GetLength(0)];
        //Vetor com o polígono pai do polígono.
        int[] f = new int[arestas.GetLength(0)];
        //Lowpoint.
        int[] l = new int[arestas.GetLength(0)];

        int dummy = 0;
        int i = 0;
        int v = s;
    passo2:
        i++;
        k[v] = i;
        l[v] = i;
        pilha = v.ToString();
    passo3:
        int intNaoUsada = -1;
        //TRUE = TODAS AS ARESTAS USADAS
        if (verificaArestas2(mapControl,
         matrizDeVizinhaca, arestas, v, ref intNaoUsada) == true)
        {
            goto passo5;
        }
```

```
    //passo4:
    if (intNaoUsada != -1)
    {
        int u = intNaoUsada;
        //FALSE = MARCA A ARESTA COMO USADA
        arestas[v, u] = true;
        arestas[u, v] = true;
        if (k[u] != 0)
        {
            l[v] = Math.Min(l[v], k[u]);
            goto passo3;
        }
        else
        {
            f[u] = v;
            v = u;
            goto passo2;
        }
    }

passo5:
    if (k[f[v]] == 1)
    {
        goto passo9;
    }
    else
    {
        if (l[v] < k[f[v]])
        {
            l[f[v]] = Math.Min(l[f[v]], l[v]);
            goto passo8;
        }
        else
        {
            //Guarda o nó de separação problema no f[v]=1.
```

```
                intNosDeSeparacao[dummy] = f[v];
                nosDeSeparacao += f[v].ToString() + "\n";
                dummy++;
            }
        }

    passo8:
        v = f[v];
        goto passo3;
    passo9:

        if (verificaArestas3(mapControl,
        matrizDeVizinhaca, arestas, s) == true)
        {
            goto finalFor;
        }
        else
        {
            //Guarda o nó de separação
            intNosDeSeparacao[dummy] = s;
            nosDeSeparacao += s.ToString() + "\n";
            dummy++;
            v = s;
            goto passo3;
        }
    finalFor:
        return (intNosDeSeparacao);
    }
    catch(Exception ex)
    {
        return (intNosDeSeparacao);
    }
}

private bool verificaNoDeSeparacao1Cluster(MapControl mapControl,
```

```
bool[,] matrizDeVizinhanca , int iPoligono)
{
    int[] vPoligonos = new int[mapControl.Shapes.Count];
vPoligonos=NosDeSeparacao(mapControl,
matrizDeVizinhanca, iPoligono);
    for (int i = 0; i < vPoligonos.Length; i++)
    {
        if (iPoligono == vPoligonos[i])
        {
            return (true);
        }
    }
    return (false);
}

/// <summary>
/// Função para verificar se é nó de articulação.
///Fonte: Graph Algorithms Autor: Shimon Even
/// </summary>
/// <param name="mapControl">Mapa</param>
/// <param name="matrizDeVizinhaca">Matriz de vizinhança</param>
/// <param name="s">Poligono inicial</param>
/// <returns></returns>
private bool VerificaNosDeSeparacao(MapControl mapControl,
 bool[,] matrizDeVizinhaca, int s)
{
bool[,] arestas = new bool[mapControl.Shapes.Count,
mapControl.Shapes.Count];

    //Nós de separação.
    int[] intNosDeSeparacao = new int[arestas.GetLength(0)];
    for (int y = 0; y < arestas.GetLength(0); y++)
    intNosDeSeparacao[y] = -1;

    for (int linha = 0; linha < mapControl.Shapes.Count; linha++)
```

```
    {
        for (int coluna = linha + 1; coluna
         < mapControl.Shapes.Count; coluna++)
        {
            arestas[linha, coluna] =
          matrizDeVizinhaca[linha, coluna];
            arestas[coluna, linha] =
           matrizDeVizinhaca[coluna, linha];
        }
    }
    //
    string pilha = "";
    string nosDeSeparacao = "";
    //Vetor com o número em que o polígono foi descoberto.
    int[] k = new int[arestas.GetLength(0)];
    //Vetor com o polígono pai do polígono.
    int[] f = new int[arestas.GetLength(0)];
    //Lowpoint.
    int[] l = new int[arestas.GetLength(0)];
    //Inicializa os dados
    for (int u = 0; u < k.Length; u++) f[u] = -1;

    int dummy = 0;
    int i = 0;
    int v = s;
passo2:
    i++;
    k[v] = i;
    l[v] = i;
    pilha = v.ToString();
passo3:
    int intNaoUsada = -1;
    if (verificaArestas2(mapControl, matrizDeVizinhaca,
    arestas, v, ref intNaoUsada) == true)
    {
```

```
            goto passo5;
        }
        //passo4:
        if (intNaoUsada != -1)
        {
            int u = intNaoUsada;
            arestas[v, u] = false;
            arestas[u, v] = false;
            if (k[u] != 0)
            {
                l[v] = Math.Min(l[v], k[u]);
                goto passo3;
            }
            else
            {
                f[u] = v;
                v = u;
                goto passo2;
            }
        }

    passo5:
        if (k[f[v]] == 1)
        {
            goto passo9;
        }
        else
        {
            if (l[v] < k[f[v]])
            {
                l[f[v]] = Math.Min(l[f[v]], l[v]);
                goto passo8;
            }
            else
            {
```

```
            intNosDeSeparacao[dummy] = f[v];
            nosDeSeparacao += f[v].ToString() + "\n";
            dummy++;
            bool blRepetido = false;
            for (int row = 0; row < intNosDeSeparacao.Length; row++)
            {
                if (intNosDeSeparacao[row] == f[v])
                {
                    blRepetido = true;
                    break;
                }
            }
            if (blRepetido == false)
            {
                //Guarda o nó de separação problema no f[v]=1.
                intNosDeSeparacao[dummy] = f[v];
                nosDeSeparacao += f[v].ToString() + "\n";
                dummy++;
            }
        }
    }

passo8:
    v = f[v];
    goto passo3;
passo9:

    if (verificaArestas3(mapControl,matrizDeVizinhaca
      ,arestas, s) == false)
    {
        goto finalFor;
    }
    else
    {
            bool blRepetido = false;
```

```
            for (int row = 0; row < intNosDeSeparacao.GetLength(0); row++)
            {
                if (intNosDeSeparacao[row] == f[v])
                {
                    blRepetido = true;
                    break;
                }
            }
            if (blRepetido == false)
            {
                //Guarda o nó de separação
                intNosDeSeparacao[dummy] = s;
                nosDeSeparacao += s.ToString() + "\n";
                dummy++;
                v = s;
                goto passo3;
            }
    }

finalFor:

    bool saida = false;
    for (int linha = 0; linha < intNosDeSeparacao.Length; linha++)
    {
        if (s ==intNosDeSeparacao[linha]) saida = true;
    }

    return (saida);
}


private bool verificaSeHaVizinhosCriticos(bool[,] matrizDeVizinhanca,
int intPoligono, int intClusterEntrada, int intClusterSaida,
 int[,] NosDeSeparacao)
{
```

```
    for (int i = 0; i < matrizDeVizinhanca.GetLength(0); i++)
    {
        if (matrizDeVizinhanca[i, intPoligono] == true)
        {
            for (int j = 0; j < NosDeSeparacao.GetLength(1); j++)
            {
                if (j != (intClusterEntrada - 1) && j != (intClusterSaida - 1))
                {
                    for (int k = 0; k < NosDeSeparacao.GetLength(0); k++)
                    {
                        if (i == NosDeSeparacao[k, j]) return (true);
                        if (NosDeSeparacao[k, j] == -1) break;
                    }
                }
            }
        }
    }
    return (false);
}

Random rnd = new Random(22211);

private int[] pontosIniciais(ref MapControl mapControl, int numCluster)
{
    int[,] intClusters = new int[mapControl.Shapes.Count, numCluster];
    for (int i = 0; i < mapControl.Shapes.Count; i++) for (int k = 0;
    k < numCluster; k++) intClusters[i, k] = -1;

    int[] intPontos = new int[numCluster];

    for (int i = 0; i < mapControl.Shapes.Count; i++)
    {
        int iCluster = Convert.ToInt32(mapControl.Shapes[i]["Cluster"]);
        intClusters[i, iCluster] = i;
```

```
    }

    bool blEcontrou = false;
    int iContador = 0;
    for (int i = 0; i < numCluster; i++)
    {
      iContador = 0;
      blEcontrou = false;
      do
      {
          if ((intClusters[iContador, i] != -1)
          && (intClusters[iContador, i] != 0))
          {
              intPontos[i] = intClusters[iContador, i];
              blEcontrou = true;
          }

          iContador++;
      } while (blEcontrou == false && iContador < intClusters.GetLength(0));
    }

    return (intPontos);
}


private void mapaCrossingOver(ref MapControl mapControl,
bool[,] matrizDeVizinhanca, double dblPercent, int numCluster)
{
    //Faz dblPercent% do mapa em mudanças
    int intPercent = Convert.ToInt32(mapControl.Shapes.Count * dblPercent);

    int cluster = -1;
    int clusterTroca = -1;
```

```
for (int linha = 0; linha < intPercent; linha++)
{
    testePintaMapa(ref mapControl, "Cluster", 5);
    //mapControl.SaveAsImage(" C:\\Programa
  //C#\\MAPA_cross_" + contCross.ToString() + ".jpeg", MapImageFormat.Jpeg);
    Application.DoEvents();
    concertaClustersPequenos(ref mapControl,
     matrizDeVizinhanca, numCluster, 5);

    mapControl.Refresh();
    mapControl.Update();


    //Sorteia um cluster
    cluster = rnd.Next(0, numCluster);
    do
    {
      clusterTroca = rnd.Next(0, numCluster);
    } while (clusterTroca == cluster);

    //Escolhe um poligono inicial desse cluster

    //Acha os poligonos criticos
    int[] s = new int[numCluster];
    s = pontosIniciais(ref mapControl, numCluster);

    //Econtra os nós de separação
    int[,] nosDeSeparacao = NosDeSeparacao(mapControl,
          matrizDeVizinhanca, s);

    //Cria a matriz de transição
    int numTransicao = 0;
    int[,] matrizDeTrasicao = new int[mapControl.Shapes.Count, 2];
    for (int w = 0; w < mapControl.Shapes.Count; w++) m
```

```
 atrizDeTrasicao[w, 0] = matrizDeTrasicao[w, 1] = -1;

for (int i = 0; i < mapControl.Shapes.Count; i++)
{
if (mapControl.Shapes[i]["Cluster"].ToString() ==
     cluster.ToString())
{
 Shape shape0 = mapControl.Shapes[i];
 for (int j = i + 1; j < mapControl.Shapes.Count; j++)
 {
     if (mapControl.Shapes[j]["Cluster"].ToString()
    == clusterTroca.ToString())
     {
         Shape shape1 = mapControl.Shapes[j];
         if ((shape0["Cluster"].ToString()
   != shape1["Cluster"].ToString()) && (matrizDeVizinhanca[i, j]
        == true) &&
     (shapeNoSeparavel(i, nosDeSeparacao, cluster)) == false)
         {
             matrizDeTrasicao[numTransicao, 0] = i;
             matrizDeTrasicao[numTransicao, 1] = j;
             numTransicao++;
         }
     }
 }
}
}

//Existe area de transição entre os conglomerados

if (numTransicao > 0)
{
int intTrocaLinha = 0;
intTrocaLinha = rnd.Next(0, numTransicao);
mapControl.Shapes[matrizDeTrasicao[intTrocaLinha, 0]]["Cluster"] =
```

```
            mapControl.Shapes[matrizDeTrasicao[intTrocaLinha, 1]]["Cluster"];

            testePintaMapa(ref mapControl, "Cluster", 5);
            Application.DoEvents();
        }

        mapControl.Update();
        mapControl.Refresh();

        }
    }

    Random mRnd = new Random(73737);
    private void CriaPopulacaoInicial(ref MapControl mapControl,
     int numCluster, int[] numPoligonos, bool[,] MatrizDeVizinhanca)
    {
        //fill
        foreach (Shape mShape in mapControl.Shapes)
        {
            mShape["Cluster"] = 0.0;
        }

        //Conta quantos poligonos existem
        int mTotalPoligonos = mapControl.Shapes.Count;

        //

        //Guarda os poligonos que existem
        List<string> mMatrixShape1 = new List<string>();
        for (int i = 0; i < mapControl.Shapes.Count; i++)
        {
            mMatrixShape1.Add(mapControl.Shapes[i].Name);
        }

        //Adiciona o poligono base - primeiro poligono do array
```

```
   string mFieldName = "Cluster"; //field a ser armazenado
   double mFieldValue = 0.0; //valor a ser armazenado

//Para cada cluster
for (int i = 0; i < numCluster; i++)
{
    //Para se não houver nenhum poligono
    if (mMatrixShape1.Count == 0) break;

    //Total de poligonos por cluster
    int mPoligonoNum = numPoligonos[i] - 1;

    //Cria o array - Segundo Grau
    List<string> mMatrixShape2 = new List<string>();

    //Sorteia o poligonio que começará o cluster
    int t = mRnd.Next(0, mMatrixShape1.Count);

    //Adiciona o poligono base - primeiro poligono do array
    mFieldName = "Cluster"; //field a ser armazenado
    mFieldValue = Convert.ToDouble(i); //valor a ser armazenado

    //
    mMatrixShape2.Add(mMatrixShape1[t]);

    //Remove o poligono base
    mMatrixShape1.Remove(mMatrixShape1[t]);


    //Executa funcao
    for (int f = 0; f < mMatrixShape2.Count; f++)
    {
        //Seleciona o poligono base
        Shape mShapeInicial = mapControl.Shapes[mMatrixShape2[0]];
        mShapeInicial[mFieldName] = mFieldValue;
```

```
        //Remove o poligo base do segundo array,
        ///pois ele eh ele mesmo - heheheh
        mMatrixShape2.Remove(mMatrixShape2[0]);

        //refaz o ponteiro
        f--;

        //retorna quantos poligonos encontrados
        int mPoligonoFind = MatrixVizinho(mapControl,
       ref mMatrixShape1, ref mMatrixShape2,
           mShapeInicial, mPoligonoNum, mFieldName, mFieldValue,
           MatrizDeVizinhanca);

        //Verifica o numero de poligonos
        mPoligonoNum -= mPoligonoFind;

        if (mPoligonoNum == 0) break;
    }

    //testePintaMapa(ref mapControl, "Cluster", i + 1);
    //mapControl.SaveAsImage(@"C:\Pedro\Duczmal\GeneticSpatial\"
    //+ i.ToString() + ".jpg", MapImageFormat.Jpeg);
}

while (mMatrixShape1.Count != 0)
{
    //
    int mPoligonoNum = mMatrixShape1.Count;


    //Adiciona o poligono base - primeiro poligono do array
    mFieldName = "Cluster"; //field a ser armazenado
    string mSelectedShape = "";
    //
```

```
    foreach (string m in mMatrixShape1)
    {
        int mP = mapControl.Shapes.GetIndex(m);

        for (int p = 0; p < mapControl.Shapes.Count; p++)
        {
            if (MatrizDeVizinhanca[mP, p] == true)
            {
                if (mapControl.Shapes[p]["Cluster"].ToString() != "0.0"
                && mapControl.Shapes[p]["Cluster"].ToString() != "0")
                {
                 mFieldValue = Convert.ToDouble(mapControl.Shapes[p]["Cluster"]);
                 mSelectedShape = m;
                 goto Continua_funcao;
                }
            }
        }
    }

Continua_funcao:

    //Cria o array - Segundo Grau
    List<string> mMatrixShape2 = new List<string>();

    if (mSelectedShape == "")
        break;

    //
    mMatrixShape2.Add(mSelectedShape);

    //Remove o poligono base
    mMatrixShape1.Remove(mSelectedShape);

    for (int f = 0; f < mMatrixShape2.Count; f++)
    {
```

```
          //Seleciona o poligono base
          Shape mShapeInicial = mapControl.Shapes[mMatrixShape2[0]];
          mShapeInicial[mFieldName] = mFieldValue;

          //Remove o poligo base do segundo array, pois
          //ele eh ele mesmo - heheheh
          mMatrixShape2.Remove(mMatrixShape2[0]);

          //refaz o ponteiro
          f--;

          //retorna quantos poligonos encontrados
          int mPoligonoFind = MatrixVizinho(mapControl,
         ref mMatrixShape1, ref mMatrixShape2,
               mShapeInicial, mPoligonoNum, mFieldName,
         mFieldValue, MatrizDeVizinhanca);

     }
 }

 //Atualiza do mapa
 mapControl.Refresh();
}
```

A.3 FUNÇÕES: KERNEL DISCRETO UNIVARIADO.

```
/// <summary>
/// Função geométrica, para núcleo estimador discreto.
/// </summary>
/// <param name="sn">Janela</param>
/// <param name="i">Parâmetro i</param>
/// <param name="j">Parâmetro j</param>
/// <returns></returns>
private double wKernel(double sn, int i, int j)
```

```
{
if (Math.Abs(i - j) >= 1) return (.5 *(1-sn) *
Math.Pow(sn, Math.Abs(i - j)));
    else return (1 - sn);
}

/// <summary>
/// Função Yn
/// </summary>
/// <param name="vetor">Vetor de dados inteiros.</param>
/// <param name="j">Parâmetro j</param>
/// <returns></returns>
private double funcYn(int[] vetor, int j)
{
    double soma = 0.0;
    double n = 0.0;
    for (int i = 0; i < vetor.Length; i++)
    {
        if (vetor[i] == j) soma++;
        n++;
    }
    return (soma / n);
}

/// <summary>
/// Função Yn
/// </summary>
/// <param name="vetor">Vetor de dados inteiros.</param>
/// <param name="j">Parâmetro j</param>
/// <returns></returns>
private double funcYnBusca(int[] vetor, int j)
{
    double soma = 0.0;
    double n = 0.0;
    for (int i = 0; i < vetor.Length; i++)
```

```
    {
        if (vetor[i] != -1)
        {
            if (vetor[i] == j) soma++;
            n++;
        }
    }
    return (soma / n);
}

/// <summary>
/// Função de suavização das probabilidades de um vetor
///aleatório discreto.
/// </summary>
/// <param name="vetor">Frequencia da observação i</param>
/// <param name="sn">Janela de suavização</param>
/// <param name="i">Parâmetro i</param>
/// <returns></returns>
public double kernelDiscreto(int[] vetor, double sn, int i)
{

    //Passo1: Econtrando os limtes de j (Aumentado em 20%)
    int limiteSup = Convert.ToInt32(maximo(vetor));
    int limiteInf = Convert.ToInt32(minimo(vetor));

    //Passo2:
    double probabilidade = 0;
    double Wk = 0;
    double Yn = 0;
    for (int j = limiteInf; j <= limiteSup; j++)
    {
        Yn = funcYn(vetor, j);
        Wk = wKernel(sn, i, j);
        probabilidade += (Wk * Yn);
    }
```

```
    return (probabilidade);
}

/// <summary>
/// Função de suavização das probabilidades de um vetor aleatório
///discreto. Somente para elementos != -1
/// </summary>
/// <param name="vetor">Frequencia da observação i</param>
/// <param name="sn">Janela de suavização</param>
/// <param name="i">Parâmetro i</param>
/// <returns></returns>
public double kernelDiscretoBusca(int[] vetor, double sn, int i)
{

    //Passo1: Econtrando os limtes de j (Aumentado em 20%)
    int limiteSup = Convert.ToInt32(maximo(vetor));
    int limiteInf = Convert.ToInt32(minimoBusca(vetor));

    //Passo2:
    double probabilidade = 0;
    double Wk = 0;
    double Yn = 0;
    for (int j = limiteInf; j <= limiteSup; j++)
    {
        Yn = funcYnBusca(vetor, j);
        Wk = wKernel(sn, i, j);
        probabilidade += (Wk * Yn);
    }

    return (probabilidade);
}


/// <summary>
```

```
/// Econtra a janela para o Kernel discreto.
/// </summary>
/// <param name="vetorCount">Vetor com as quantidades de cada tipo.</param>
/// <returns></returns>
public double janelaDiscreta(double[] vetorCount, double soma)
{
//Minimizando o erro quadrático médio Eq6 do texto "A class of
//smooth estimators for discrete distributions.pdf";
    double[] ps = new double[vetorCount.Length];
    for (int i = 0; i < vetorCount.Length; i++)
    {
        ps[i] = vetorCount[i] / soma;
    }


    double B0 = 0;
    double B1 = 0;
    double B2 = 0;
    double beta1 = 0;
    double beta10 = 0;

    double somaP2 = 0;
    for (int i = 0; i < ps.Length; i++) somaP2 += ps[i] * ps[i];
    for (int i = 1; i < ps.Length - 1; i++)
    {
        double tB0 = (ps[i - 1] + ps[i + 1]) * (ps[i - 1] + ps[i + 1]);
        B0 += (ps[i - 1] + ps[i + 1]) * (ps[i - 1] + ps[i + 1]);
        B1 += ps[i] * (ps[i - 1] + ps[i + 1]);
    }
for (int i = 2; i < ps.Length -2; i++) B2 += ps[i] *
(ps[i - 2] + ps[i + 2]);

    B0 += (ps[1] * ps[1]) + (ps[ps.Length - 2] * ps[ps.Length - 2]);
    B1 += ps[0] * ps[1] + ps[ps.Length - 1] * ps[ps.Length - 2];
B2 += ps[0] * ps[2] + ps[1] * ps[3] + ps[ps.Length - 2] *
```

```
 ps[ps.Length - 4] + ps[ps.Length - 1] * ps[ps.Length - 3];

    beta1 = 1 - somaP2 + 0.5 * B1;
    beta10 = somaP2 - B1 + 0.25 * B0;

    return (beta1 * (1 / (1.5 + B1 - B2 + (soma - 1) * beta10)));

}

private bool valorDiferente(ArrayList arList, int intValor)
{
    for (int i = 0; i < arList.Count; i++)
    {
        if (Convert.ToInt32(arList[i]) == intValor) return (false);
    }
    return (true);
}

/// <summary>
/// Tabela de Frequencias
/// </summary>
/// <param name="MicroDados">Vetor de dados.</param>
/// <returns></returns>
public double[] vetorPercentual(int[] dados)
{
    ArrayList arList = new ArrayList();
    double total = 0;

    for (int i = 0; i < dados.Length; i++)
    {
        if (arList.Count == 0 && dados[i] > -1)
        {
            arList.Add(dados[i]);
            total++;
        }
```

```
        else if (arList.Count > 0 && dados[i] > -1)
        {
            for (int j = 0; j < arList.Count; j++)
            {
                if (valorDiferente(arList, dados[i]) == true)
                {
                    arList.Add(dados[i]);
                    break;
                }
            }
            total++;
        }
    }
    double[] dblPercent = new double[arList.Count];

    for (int i = 0; i < dados.Length; i++)
    {
        if (dados[i] != -1)
        {
            dblPercent[arList.IndexOf(dados[i])] += 1;
        }
    }

    for (int i = 0; i < arList.Count; i++) dblPercent[i] /= total;

    return (dblPercent);
}

/// <summary>
/// Tabela de Frequencias
/// </summary>
/// <param name="MicroDados">Vetor de dados.</param>
/// <returns></returns>
public double[] vetorPercentual(int[] dados,ref double total)
{
```

```
ArrayList arList = new ArrayList();

for (int i = 0; i < dados.Length; i++)
{
    if (arList.Count == 0 && dados[i] > -1)
    {
        arList.Add(dados[i]);
        total++;
    }
    else if (arList.Count > 0 && dados[i] > -1)
    {
        for (int j = 0; j < arList.Count; j++)
        {
            if (valorDiferente(arList,dados[i])==true)
            {
                arList.Add(dados[i]);
                break;
            }
        }
        total++;
    }
}

double[] dblPercent = new double[arList.Count];

for (int i = 0; i < dados.Length; i++)
{
    if (dados[i] != -1)
    {
        dblPercent[arList.IndexOf(dados[i])] += 1;
    }
}

for (int i = 0; i < arList.Count; i++) dblPercent[i] /= total;
```

```
    return (dblPercent);
}


/// <summary>
/// Econtra a janela para o Kernel discreto.
/// </summary>
/// <param name="vetorCount">Vetor com as quantidades de cada tipo.</param>
/// <returns></returns>
public double janelaDiscretaBusca(int[] intVetor)
{
    Array.Sort(intVetor);
    double soma = 0;

    double[] ps = vetorPercentual(intVetor,ref soma);

    double B0 = 0;
    double B1 = 0;
    double B2 = 0;
    double beta1 = 0;
    double beta10 = 0;

    double somaP2 = 0;
for (int i = 0; i < ps.Length; i++) somaP2
+= ps[i] * ps[i];
    for (int i = 1; i < ps.Length - 1; i++)
    {
        double tB0 = (ps[i - 1] + ps[i + 1]) *
           (ps[i - 1] + ps[i + 1]);
        B0 += (ps[i - 1] + ps[i + 1]) * (ps[i - 1]
            + ps[i + 1]);
        B1 += ps[i] * (ps[i - 1] + ps[i + 1]);
    }
for (int i = 2; i < ps.Length - 2; i++) B2
+= ps[i] * (ps[i - 2] + ps[i + 2]);
```

```
B0 += (ps[1] * ps[1]) + (ps[ps.Length - 2]
 * ps[ps.Length - 2]);
B1 += ps[0] * ps[1] + ps[ps.Length - 1]
* ps[ps.Length - 2];

    double dblControle4 = 0;
    double dblControle3 = 0;
    double dblControle2 = 0;
    double dblControle1 = 0;

    double p0 = 0;
    double p1 = 0;
    double p2 = 0;
    double p3 = 0;


    if (ps.Length - 4 >= 0)
    {
        dblControle4 = ps[ps.Length - 4];
        p3 = ps[3];
    }
    if (ps.Length - 3 >= 0)
    {
        dblControle3 = ps[ps.Length - 3];
        p2 = ps[2];
    }
    if (ps.Length - 2 >= 0)
    {
        dblControle2 = ps[ps.Length - 2];
        p1 = ps[1];
    }
    if (ps.Length - 1 >= 0)
    {
        dblControle1 = ps[ps.Length - 1];
```

```
        p0 = ps[0];
    }

B2 += p0 * p2 + p1 * p3 + dblControle2 *
dblControle4 + dblControle1 * dblControle3;

    beta1 = 1 - somaP2 + 0.5 * B1;
    beta10 = somaP2 - B1 + 0.25 * B0;

    return (beta1 * (1 / (1.5 + B1 - B2 + (soma - 1) * beta10)));

}


/// <summary>
/// Econtra a entropia estimada para o vetor discreto univariado.
/// </summary>
/// <param name="intVetor">Vetor de dados</param>
/// <param name="vetorCount">Vetor das quantidades</param>
/// <param name="soma">Soma dos valores</param>
/// <returns></returns>
public double entropyUnivariadaDiscreta(int[] intVetor,
 double[] vetorCount, double soma)
{
    //Inicia as variáveis
    double dblEntropia = 0;

    //Econtra a janela ótima
    double dblJanela = janelaDiscreta(vetorCount, soma);

    for (int j = 0; j < intVetor.Length; j++)
    {
        dblEntropia += Math.Log(kernelDiscreto(intVetor ,
              dblJanela, intVetor[j]));
    }
```

```
    return (-(1 / intVetor.Length) * dblEntropia);

}

public double entropyUnivariadaDiscreta(MapControl mapControl,
 string strVariavel, double[] vetorCount, double soma)
{
    //Inicia as variáveis
    double dblEntropia = 0;

    //Econtra a janela ótima
    double dblJanela = janelaDiscreta(vetorCount, soma);

    //INICIO: Adaptação para o DUNDAS
    int[] intVetor = new int[mapControl.Shapes.Count];
    for (int i = 0; i < mapControl.Shapes.Count; i++)
    {
        if (string.IsNullOrEmpty(mapControl.Shapes[i][strVariavel].ToString())
 == false) intVetor[i] = Convert.ToInt32(mapControl.Shapes[i][strVariavel]);
        else intVetor[i] = 0;
    }
    //FIM: Adaptação para o DUNDAS

    for (int j = 0; j < intVetor.Length; j++)
    {
        dblEntropia += Math.Log(kernelDiscreto(intVetor,
          dblJanela, intVetor[j]));
    }

    return (-(1 / intVetor.Length) * dblEntropia);
}

public double entropiaAmostral(double cz,double dblObservacoes)
{
```

```
    return (cz / dblObservacoes);
}

public double[] vetorEntropia(MapControl mapControl,
string strVariavel, double dblJanela)
{
    int[] inVet = new int[mapControl.Shapes.Count];
    for (int i = 0; i < mapControl.Shapes.Count; i++)
    {
        if (string.IsNullOrEmpty(
     mapControl.Shapes[i][strVariavel].ToString()) == true)
        {
            inVet[i] = 0;
        }
        else if
   (string.IsNullOrEmpty(mapControl.Shapes[i][strVariavel].ToString()) != true)
        {
            inVet[i] = Convert.ToInt32(mapControl.Shapes[i][strVariavel]);
        }
    }

    //Econtra o vetor com os -log(pi)
    double[] dblEntropy = new double[mapControl.Shapes.Count];
    for (int i = 0; i < mapControl.Shapes.Count; i++)
    {
        dblEntropy[i] = -Math.Log(kernelDiscreto(inVet, dblJanela, inVet[i]));
    }

    return (dblEntropy);
}


private double[] vetorEntropiaCluster(int[] intBusca, double dblJanela)
{
```

```
    //Econtra o vetor com os -log(pi)
    double[] dblEntropy = new double[intBusca.Length];
    for (int i = 0; i <intBusca.Length ; i++)
    {
        if (intBusca[i] == -1) dblEntropy[i] = -1;
        else dblEntropy[i] = -Math.Log(kernelDiscretoBusca(intBusca,
 dblJanela, intBusca[i]));
    }
    return (dblEntropy);
}

private double EntropiaCluster(int[] intBusca, double dblJanela)
{

    //Econtra o vetor com os -log(pi)
    double dblEntropy = 0;
    double conta = 0;
    for (int i = 0; i < intBusca.Length; i++)
    {
        if (intBusca[i] != -1)
        {
            dblEntropy += -Math.Log(kernelDiscretoBusca(intBusca,
            dblJanela, intBusca[i]));
            conta++;
        }
    }
    return (dblEntropy / conta);
}

private void numPoligonosPorCluster(MapControl mapControl,
 ref int[] vetorNumCluster)
{
    int cluster;
    for (int j = 0; j < vetorNumCluster.Length; j++) vetorNumCluster[j] = 0;
    for (int i = 0; i < mapControl.Shapes.Count; i++)
```

```
    {
        cluster = Convert.ToInt32(mapControl.Shapes[i]["Cluster"]);
        vetorNumCluster[cluster] += 1;
    }
}

private void funcaoObjetivoEntropiaDiscretaUnivariada(MapControl mapControl,
int numCluster,
 string strVariaveis, ref double dblEntropia,
ref double[] vetorEntropia,ref double[] dblJanela)
{
    int[] intPoligonosCluster = new int[numCluster];
    numPoligonosPorCluster(mapControl, ref intPoligonosCluster);
    int soma=0;

    for (int iCluster = 0; iCluster < numCluster; iCluster++)
    {
        soma=0;
        int[] vetorDados = new int[intPoligonosCluster[iCluster]];
        for (int i = 0; i < mapControl.Shapes.Count; i++)
        {
            if (Convert.ToDouble(mapControl.Shapes[i]["Cluster"])
            == (double)iCluster)
            {
                vetorDados[soma] =
               Convert.ToInt32(mapControl.Shapes[i][strVariaveis]);
                soma++;
            }
        }
        dblJanela[iCluster] = janelaDiscretaBusca(vetorDados);
        vetorEntropia[iCluster] = EntropiaCluster(vetorDados, dblJanela[iCluster]);
        dblEntropia += vetorEntropia[iCluster];
    }
}
```

```
private void FazTrocaDePosicao(MapControl mapControl,
ref double[,] dblMatriz, int posicao)
{
    for (int i = 0; i < mapControl.Shapes.Count; i++)
{

        dblMatriz[i, posicao] =
 Convert.ToDouble(mapControl.Shapes[i]["Cluster"]);
    }
}

private void guardaValorNoHistorico(ref double[,] historico,
 double[,] dblMelhores, int colunaHistorico)
{
    for (int i = 0; i < historico.GetLength(0); i++)
    {
        historico[i, colunaHistorico] = dblMelhores[i, 0];
    }
}

private int intPosicaoMinimo(int[] vetor)
{
    int intPosicao=0;
    int intValor=vetor[0];
    for(int i=0;i<vetor.Length;i++)
    {
        if (vetor[i] < intPosicao)
        {
            intValor = vetor[i];
            intPosicao = i;
        }
    }
    return (intPosicao);
```

```
}

private void concertaClustersPequenos(ref MapControl mapControl,
bool[,] matrizDeVizinhaca,int numCluster,int intAdiciona)
{
    try
    {

        int[] vetorCluster = new int[numCluster];
        int intShape = -1;
        int j = 0;
        numPoligonosPorCluster(mapControl, ref vetorCluster);
        ArrayList arPoligonos = new ArrayList();
        for (int i = 0; i < numCluster; i++)
        {
            arPoligonos.Clear();
            if (vetorCluster[i] < intAdiciona)
            {
                int intAumenta = intAdiciona - vetorCluster[i];

                //Acha o conglomerado que iniciará o aumento
                for (j = 0; j < mapControl.Shapes.Count; j++)
                {
                    if (Convert.ToInt32(mapControl.Shapes[j]["Cluster"]) == i)
                    {
                        arPoligonos.Add(j);
                    }
                }

                //ko= indice do aumenta, lo=indice do poligono ip=poligono
                int lo = 0, ko = 0;
                bool entrou = false;
                int ip = 0;
                intShape = (int)arPoligonos[ip];
```

```
                //Faz enquanto ko<intAumenta e ip <
                 //quantidade de poligonos no cluster i
               do
               {

do

{
                       if ((matrizDeVizinhaca[intShape, lo] == true)
&& (mapControl.Shapes[intShape]["Cluster"].ToString()
!= mapControl.Shapes[lo]["Cluster"].ToString()))
            {
                if (verificaNoDeSeparacao1Cluster(mapControl,
matrizDeVizinhaca, lo) == false)
                {
                    mapControl.Shapes[lo]["Cluster"] = Convert.ToDouble(i);
                    entrou = true;
                    ko++;
                }
            }
            lo++;
        } while (lo < mapControl.Shapes.Count && entrou == false);

        if (lo == mapControl.Shapes.Count && ip < arPoligonos.Count)
        {
            intShape = (int)arPoligonos[ip];
            ip++;
            lo = 0;
        }
     } while (ko < intAumenta && ip < arPoligonos.Count);
 }
        }
    }
    catch (Exception ex)
    {
```

```
    }
}

private void probabilidadeDiscreta(ref MapControl mapControl,
 int numCluster, ref double[] janelas, string strVariavel)
{

    int[] intPoligonosCluster = new int[numCluster];
    numPoligonosPorCluster(mapControl, ref intPoligonosCluster);
    int soma = 0;
    ArrayList arPoligonos = new ArrayList();
    for (int iCluster = 0; iCluster < numCluster; iCluster++)
    {
        arPoligonos.Clear();
        soma = 0;
        int[] vetorDados = new int[intPoligonosCluster[iCluster]];
        for (int i = 0; i < mapControl.Shapes.Count; i++)
        {

            if (Convert.ToDouble(mapControl.Shapes[i]["Cluster"])
 == (double)iCluster)
            {
                vetorDados[soma] =
 Convert.ToInt32(mapControl.Shapes[i][strVariavel]);
                arPoligonos.Add(i);
                soma++;
            }
        }

        janelas[iCluster] = janelaDiscretaBusca(vetorDados);

        for (int i = 0; i < vetorDados.Length; i++)
        {
            if (vetorDados[i] != -1)
            {
```

```
                int iPoligono = Convert.ToInt32(arPoligonos[i]);
                mapControl.Shapes[iPoligono]["PROBABILIDADE"] =
 kernelDiscretoBusca(vetorDados, janelas[iCluster], vetorDados[i]);
            }
        }
    }
}

public void
algoritmoGeneticoEntropiaDiscretaUnivariado(
ref MapControl mapControl,string strVariaveis,
string strCluster, int numCluster,int numPopulacao,
int numMelhores, int numIteracoes, ref double[,]
historico, double propCrossing, double dblFatorNorma,
string strVariavel, ref ProgressBar pgBar)
{

    //Guarda os numMelhores
    double[,] dblMelhores = new double[mapControl.Shapes.Count, numMelhores];
    double[] dblMelhoresEntropia = new double[numMelhores];

    //Inicializa o vetor com o valor minimo.
for (int j = 0; j < numMelhores; j++) dblMelhoresEntropia[j]
 = double.MaxValue;

    //Vetor com o número de poligonos por cluster
    int[] vetCluster = new int[numCluster];

    /*********************************      Matriz de vizinhança**********/
bool[,] matrizDeVizinhaca = new bool[mapControl.Shapes.Count,
mapControl.Shapes.Count];
    for (int i = 0; i < mapControl.Shapes.Count; i++)
    {
        Shape m_shape0 = mapControl.Shapes[i];
        for (int j = i + 1; j < mapControl.Shapes.Count; j++)
```

```
        {
            Shape m_shape1 = mapControl.Shapes[j];
            if (poligonosSaoVizinhos(m_shape0, m_shape1) == true)
            {
                matrizDeVizinhaca[i, j] = true;
                matrizDeVizinhaca[j, i] = true;
            }
        }
    }


    int[] vetorDummy = new int[numMelhores];
double[,] dummyMelhores = new double[mapControl.Shapes.Count,
 numMelhores];
    Ran1 rnd = new Ran1();
    Ran1 rnd2 = new Ran1();

    Random rnd0 = new Random(43442);

    //Sementes
    rnd.Idum = 635325;
    rnd2.Idum = 162662;

//Dummy que impede que mais de uma posição
//seja alterada no vetor de melhores.
    double dummy_t = 0;

    pgBar.Minimum = 0;
    pgBar.Maximum = numPopulacao + (numIteracoes * numMelhores);

    /*********************************    Gera as populações iniciais***/
    for (int pop = 0; pop < numPopulacao; pop++)
    {
        pgBar.Increment(1);
        Application.DoEvents();
```

```
        int[] dblLimites = new int[numCluster];

        dblLimites[0] = dblLimites[1] = dblLimites[2]
= dblLimites[3] = 13;
        dblLimites[4] = 11;

        //Coloca os Clusters no mapa
        if (mapControl.ShapeFields.GetIndex("Cluster")
== -1)//Caso a variável não exista
        {
            Field Cluster = new Field();
            Cluster.Name = "Cluster";
            mapControl.ShapeFields.Add(Cluster);
            mapControl.ShapeFields["Cluster"].Type =
 typeof(System.Double);

        }

        /*Cria a arvore com a geração dos Clusters*/

        CriaPopulacaoInicial(ref mapControl, numCluster,
dblLimites, matrizDeVizinhaca);
        concertaClustersPequenos(ref mapControl,
 matrizDeVizinhaca, numCluster, 5);

        testePintaMapa(ref mapControl, "Cluster", 5);
        Application.DoEvents();

        mapControl.Update();
        mapControl.Refresh();

        /*Guarda o valor da Função Objetivo**/

        //(Minimizar)
```

```
        double dummyEntropia = 0;
        double[] vetorEntropia = new double[numCluster];
        double[] dblJanela = new double[numCluster];


        funcaoObjetivoEntropiaDiscretaUnivariada(mapControl, numCluster,
strVariaveis, ref dummyEntropia, ref vetorEntropia, ref dblJanela);

        //Guarda o individuo na posição dele, caso seja um dos numMelhores
        dummy_t = 0;
        for (int i = 0; i < numMelhores; i++)
        {
            if (pop <= numMelhores - 1)
            {
                FazTrocaDePosicao(mapControl, ref dummyMelhores, pop);
                dblMelhoresEntropia[pop] = dummyEntropia;
                vetorDummy[pop] = pop;
            }

            if (pop > numMelhores - 1)
            {
                if (pop == numMelhores && i == 0)
                {
                    if (dummyEntropia  < dblMelhoresEntropia[i])
                    {
                        FazTrocaDePosicao(mapControl, ref dblMelhores, i);
                        dblMelhoresEntropia[i] = dummyEntropia;
                    }
                }
                else if (pop > numMelhores)
                {
                    if ((dummyEntropia < dblMelhoresEntropia[i]) && dummy_t == 0)
                    {
                        //Impede que mais de uma posição seja alterada
                        dummy_t = 1;
```

```
                        FazTrocaDePosicao(mapControl, ref dblMelhores, i);
                        dblMelhoresEntropia[i] = dummyEntropia;
                    }
                }
            }
        }
    }

//Guarda o melhor valor gerado da população inicial
guardaValorNoHistorico(ref historico, dblMelhores, 0);
for (int k = 0; k < mapControl.Shapes.Count; k++)
mapControl.Shapes[k]["Cluster"] = dblMelhores[k, 0];
    testePintaMapa(ref mapControl, "Cluster", numCluster);
mapControl.SaveAsImage(" C:\\Programa C#\\MAPA_0.jpg",
 MapImageFormat.Jpeg);
    //Exportando mapa
TextWriter tw__ = new StreamWriter(" C:\\Programa C
#\\Univariado_Discreto_Mapa_Inicial_" + strVariavel + ".txt");
    tw__.WriteLine("MICROCOD" + "\t" + "Cluster");
    for (int i = 0; i < mapControl.Shapes.Count; i++)
    {
        tw__.WriteLine(mapControl.Shapes[i]["MICROCOD"].ToString() + "\t" +
mapControl.Shapes[i]["Cluster"].ToString());
    }
    tw__.Close();

    //Número de poligonos por Cluster
    numPoligonosPorCluster(mapControl, ref vetCluster);

    /*****  Começa o Crossing Over.  **/

    double dummyEntropia2 = 0;//(Minimizar)
    double[] vetorEntropia2 = new double[numCluster];
    double[] dblJanela2 = new double[numCluster];
```

```
double[,] vetorDeTransicao = new double[(int)((mapControl.Shapes.Count
 * mapControl.Shapes.Count) / 2), 2];

    for (int j = 0; j < numMelhores; j++)
    {
        for (int i = 0; i < numIteracoes; i++)
        {
            pgBar.Increment(1);
            Application.DoEvents();

            //Coloca o j-ésimo melhor no mapa
            for (int k = 0; k < mapControl.Shapes.Count; k++)
mapControl.Shapes[k]["Cluster"] = dblMelhores[k, 0];

            //Concerta Clusters pequenos
            concertaClustersPequenos(ref mapControl,
 matrizDeVizinhaca, numCluster, 5);

            //Faz a mutação no mapa
            mapaCrossingOver(ref mapControl, matrizDeVizinhaca,
 propCrossing, numCluster);

            //Verifica se houve melhora e guarda caso seja necessário
            funcaoObjetivoEntropiaDiscretaUnivariada(mapControl,
numCluster, strVariaveis, ref dummyEntropia2, ref vetorEntropia2,
ref dblJanela2);

            //Concerta clusters pequenos
            concertaClustersPequenos(ref mapControl,
 matrizDeVizinhaca, numCluster, 5);

            dummy_t = 0;

for (int l = 0; l < numMelhores; l++)
```

```
{
    //tw2.WriteLine(j.ToString() + "\t" + i.ToString() + "\t" + l.ToString());

    if (dummyEntropia2  < dblMelhoresEntropia[l]  && dummy_t == 0)
    {
        FazTrocaDePosicao(mapControl, ref dblMelhores, l);

        dblMelhoresEntropia[l] = dummyEntropia2;

        //if (l == 0) guardaValorNoHistorico(ref historico, dblMelhores, 0);

        guardaValorNoHistorico(ref historico, dblMelhores, l);

        //Impede que mais de uma posição seja alterada
        dummy_t = 1;
    }
}

            mapControl.Refresh();
            mapControl.Update();

            testePintaMapa(ref mapControl, "Cluster", 5);
            if (j == 0) mapControl.SaveAsImage(" C:\\Programa
 C#\\MAPA_" + i.ToString() + ".jpeg", MapImageFormat.Jpeg);
        }
    }

    //Guarda o melhor no mapa
for (int k = 0; k < mapControl.Shapes.Count; k++)
mapControl.Shapes[k]["Cluster"] = dblMelhores[k, 0];

    mapControl.Refresh();
    mapControl.Update();
```

```
    double[] janelas = new double[numCluster];
    probabilidadeDiscreta(ref mapControl, numCluster, ref janelas, strVariavel);

    testePintaMapa(ref mapControl, "Cluster", numCluster);
    //Exportando mapa
TextWriter tw_ = new StreamWriter(" C:\\Programa
C#\\Univariado_Discreto_Mapa_Cluster_" + strVariavel + ".txt");
    tw_.WriteLine("MICROCOD" + "\t" + "Cluster" + "\t" + "PROBABILIDADE");
    for (int i = 0; i < mapControl.Shapes.Count; i++)
    {
        tw_.WriteLine(mapControl.Shapes[i]["MICROCOD"].ToString() + "\t" +
mapControl.Shapes[i]["Cluster"].ToString() + "\t"
+ mapControl.Shapes[i]["PROBABILIDADE"].ToString());
    }
    tw_.Close();

    //Exportando mapa
TextWriter tw___ = new StreamWriter(" C:\\Programa
C#\\Univariado_Discreto_Mapa_Janelas_" + strVariavel + ".txt");
    tw___.WriteLine("Cluster" + "\t" + "Janela");
    for (int i = 0; i < numCluster; i++)
    {
        tw___.WriteLine(i.ToString() + "\t" + janelas[i].ToString());
    }
    tw___.Close();

    MessageBox.Show("Acabou!", "Final.");

}
```

A.4 FUNÇÕES: KERNEL CONTÍNUO UNIVARIADO.

```
//Econtrando o Lambda
 public double econtraLambdaContinuo(double[] dados)
```

```
 {
     //Copia o vetor de dados
     double Lambda=1-0.01;
     double tamanho = dados.Length;
     double valorLambda = 0;
     do
     {
         Lambda += 0.1;
         double somaCosseno = 0;
         double somaSeno = 0;

         for(int i=0;i<tamanho;i++)
         {
             somaCosseno += Math.Cos(dados[i] * Lambda);
             somaSeno += Math.Sin(dados[i] * Lambda);
         }
         valorLambda = (somaCosseno * somaCosseno
+ somaSeno * somaSeno) / (tamanho * tamanho);

     } while (Lambda < 10000 && valorLambda > (3 / tamanho));

     return (Lambda);
 }


 private double funcaoIntegral(double lambda,double[] dados)
 {
     double n = dados.Length;
     double cosseno = 0;
     double seno = 0;

     for (int i = 0; i < n; i++)
     {
         cosseno += Math.Cos(lambda*dados[i]);
         seno += Math.Sin(lambda*dados[i]);
```

```
    }

return(Math.Pow(lambda, 4) * ((((cosseno * cosseno)
/ (n * n)) + ((seno * seno) / (n * n))) - (1 / n)));
 }

 //Econtrando a janela ótima
 public double janelaOtimaContinuo(double[] dados, double Lambda)
 {
     //Qromb tr = new Qromb();
     Trapzd tr = new Trapzd();
     Delegates.FunctionDoubleToDoubleComVetorContinuo func =
new Delegates.FunctionDoubleToDoubleComVetorContinuo(funcaoIntegral);
     //double G=(1/(Math.PI))*tr.qromb(func, 0.0, Lambda,dados);
     double G = (1 / (Math.PI)) * tr.trapzd(func, 0.0, Lambda, 5 , dados);
     double tamanho = Convert.ToDouble(dados.Length);
     double parteG=G*tamanho;
     double janela = 1.000000029960001570802894122509 8 *
0.776388356423398286870038096481 69 * Math.Pow(parteG, -0.2);
     return (janela);
 }

 /// <summary>
 /// Função de suavização das probabilidades de um
vetor aleatório contínuo. Somente para elementos != -1
 /// </summary>
 /// <param name="vetor">Dados</param>
 /// <param name="sn">Janela de suavização</param>
 /// <param name="i">Parâmetro x</param>
 /// <returns></returns>
 public double kernelContinuoBusca(double[] vetor,
double h, double i)
 {
     double n = Convert.ToDouble(vetor.Length);
     double probabilidade = 1 / (n * h);
```

```
        double soma = 0;
        double x = 0;
        for (int j = 0; j < vetor.Length; j++)
        {
            x = (i - vetor[j]) / h;
            soma += (1/(Math.Sqrt(2*Math.PI)))*Math.Exp(-((x * x) / 2));
        }

        return (probabilidade*soma);
    }

    private double EntropiaClusterContinuo(double[]
intBusca, double dblJanela)
    {

        //Econtra o vetor com os -log(pi)
        double dblEntropy = 0;
        double conta = 0;
        for (int i = 0; i < intBusca.Length; i++)
        {
            if (intBusca[i] != -1)
            {
                double fx = kernelContinuoBusca(intBusca,
dblJanela, intBusca[i]);
                dblEntropy += -Math.Log(fx);
                conta++;
            }
        }
        return (dblEntropy / conta);
    }

private void funcaoObjetivoEntropiaContinuoUnivariada(MapControl
mapControl, int numCluster, string strVariaveis,
ref double dblEntropia, ref double[] vetorEntropia,
ref double[] dblJanela)
```

```
{
    int[] intPoligonosCluster = new int[numCluster];
numPoligonosPorCluster(mapControl,
 ref intPoligonosCluster);
int soma = 0;


    for (int iCluster = 0; iCluster < numCluster; iCluster++)
    {
        soma = 0;
        double[] vetorDados = new
double[intPoligonosCluster[iCluster]];
        for (int i = 0; i < mapControl.Shapes.Count; i++)
        {
            if (Convert.ToDouble(mapControl.Shapes[i]["Cluster"])
 == (double)iCluster)
            {
                vetorDados[soma] =
Convert.ToDouble(mapControl.Shapes[i][strVariaveis]);
                soma++;
            }
        }
        double lambda =
econtraLambdaContinuo(vetorDados);
        dblJanela[iCluster] =
janelaOtimaContinuo(vetorDados,lambda);
        vetorEntropia[iCluster] =
 EntropiaClusterContinuo(vetorDados, dblJanela[iCluster]);
        dblEntropia += vetorEntropia[iCluster];
    }
}

private void distanciasPoligono(MapControl mapControl, int iShape,
ref ArrayList arListaPoligonos, ref ArrayList arListaDistancias)
{
```

```
    for (int i = 0; i < mapControl.Shapes.Count; i++)
    {
        if (Convert.ToDouble(mapControl.Shapes[i]["Cluster"]) == -1.00)
        {
            double distancia =
Math.Sqrt(Math.Pow((mapControl.Shapes[i].CentralPoint.X -
 mapControl.Shapes[iShape].CentralPoint.X), 2) +
Math.Pow((mapControl.Shapes[i].CentralPoint.Y -
mapControl.Shapes[iShape].CentralPoint.Y), 2));
                    arListaDistancias.Add(distancia);
                    arListaPoligonos.Add(i);
                }
            }
        }

private void distanciasPoligonoGeral(MapControl mapControl, i
nt iShape, ref ArrayList arListaPoligonos, ref ArrayList arListaDistancias)
{
    for (int i = 0; i < mapControl.Shapes.Count; i++)
    {
        double distancia = Math.Sqrt(M
ath.Pow((mapControl.Shapes[i].CentralPoint.X
 - mapControl.Shapes[iShape].CentralPoint.X), 2) +
 Math.Pow((mapControl.Shapes[i].CentralPoint.Y
 - mapControl.Shapes[iShape].CentralPoint.Y), 2));
                arListaDistancias.Add(distancia);
                arListaPoligonos.Add(i);
            }
        }

private bool verificaVizinhoMesmoCluster(MapControl mapControl,
 int iPoligono, bool[,] matrizDeVizinhaca, double dblCluster)
{
    for (int i = 0; i < mapControl.Shapes.Count; i++)
    {
```

```
        if (matrizDeVizinhaca[iPoligono, i] == true &&
dblCluster.ToString() == mapControl.Shapes[i]["Cluster"].ToString())
        {
            return (true);
        }
    }
    return (false);
}

private void poligonosDosConglomerados(MapControl mapControl
 , ref ArrayList arLista, int iCluster)
{
    arLista.Clear();
    for (int i = 0; i < mapControl.Shapes.Count; i++)
    {
        if (mapControl.Shapes[i]["Cluster"].ToString()
 == iCluster.ToString())
        {
            arLista.Add(i);
        }
    }
}

private void probabilidadeContinua(ref MapControl mapControl,
 int numCluster,ref double[] janelas,string strVariavel)
{
    int[] intPo = new int[numCluster];
    numPoligonosPorCluster(mapControl,ref intPo);
    for (int i = 0; i < numCluster; i++)
    {
        double[] vetor = new double[intPo[i]];
        int iConta=0;
        for(int j=0;j<mapControl.Shapes.Count;j++)
        {
            if(mapControl.Shapes[j]["Cluster"].ToString()==i.ToString())
```

```
            {
                vetor[iConta]=Convert.ToDouble(mapControl.Shapes[j][strVariavel]);
                iConta++;
            }
        }
        double Lambda = econtraLambdaContinuo(vetor);
        janelas[i] = janelaOtimaContinuo(vetor, Lambda);
    }

    //Guarda as probabilidades
    double Soma=0, Xi=0, Xj=0;
    for (int c = 0; c < numCluster; c++)
    {
        for (int i = 0; i < mapControl.Shapes.Count; i++)
        {
            if (mapControl.Shapes[i]["Cluster"].ToString() == c.ToString())
            {
                Xi=Convert.ToDouble(mapControl.Shapes[i][strVariavel]);
                for (int j = 0; j < mapControl.Shapes.Count; j++)
                {
                   if (mapControl.Shapes[j]["Cluster"].ToString()
== c.ToString())
                   {
                   Xj=Convert.ToDouble(mapControl.Shapes[j][strVariavel]);
                  Soma+=(1/(Math.Sqrt(2*Math.PI)))*Math.Exp(-((Xi-Xj)/
janelas[c])*((Xi-Xj)/janelas[c])/2);
                   }
                }
                mapControl.Shapes[i]["PROBABILIDADE"] = (1 / (intPo[c]
* janelas[c])) * Soma;
            }
        }
    }
}
```

```
public void algoritmoGeneticoEntropiaContinuoUnivariado(ref MapControl mapControl,
 string strVariaveis, string strCluster, int numCluster, int numPopulacao,
 int numMelhores, int numIteracoes, ref double[,] historico,
double propCrossing, double dblFatorNorma, string strVariavel,
 ref ProgressBar pgBar)
{

//Guarda os numMelhores
double[,] dblMelhores = new double[mapControl.Shapes.Count, numMelhores];
double[] dblMelhoresEntropia = new double[numMelhores];

//Inicializa o vetor com o valor minimo.
for (int j = 0; j < numMelhores; j++) dblMelhoresEntropia[j]
= double.MaxValue;

//Vetor com o número de poligonos por cluster
int[] vetCluster = new int[numCluster];

/**********Matriz de vizinhança.             ******************/
bool[,] matrizDeVizinhaca = new bool[mapControl.Shapes.Count,
 mapControl.Shapes.Count];
    for (int i = 0; i < mapControl.Shapes.Count; i++)
    {
        Shape m_shape0 = mapControl.Shapes[i];
        for (int j = i + 1; j < mapControl.Shapes.Count; j++)
        {
            Shape m_shape1 = mapControl.Shapes[j];
            if (poligonosSaoVizinhos(m_shape0, m_shape1) == true)
            {
                matrizDeVizinhaca[i, j] = true;
                matrizDeVizinhaca[j, i] = true;
            }
        }
    }
```

```
    int[] vetorDummy = new int[numMelhores];
double[,] dummyMelhores = new double[mapControl.Shapes.Count,
 numMelhores];
    Ran1 rnd = new Ran1();
    Ran1 rnd2 = new Ran1();

    Random rnd0 = new Random(43442);

    //Sementes
    rnd.Idum = 635325;
    rnd2.Idum = 162662;

//Dummy que impede que mais de uma posição seja
//alterada no vetor de melhores.
    double dummy_t = 0;

    pgBar.Minimum = 0;
    pgBar.Maximum = numPopulacao + (numIteracoes * numMelhores);


    /*********************************    Gera as populações iniciais**********/
    for (int pop = 0; pop < numPopulacao; pop++)
    {
        pgBar.Increment(1);
        Application.DoEvents();

        int[] dblLimites = new int[numCluster];

        dblLimites[0] = dblLimites[1] = dblLimites[2] = dblLimites[3] = 13;
        dblLimites[4] = 11;

        //Coloca os Clusters no mapa
        if (mapControl.ShapeFields.GetIndex("Cluster")
== -1)//Caso a variável não exista
        {
```

```
            Field Cluster = new Field();
            Cluster.Name = "Cluster";
            mapControl.ShapeFields.Add(Cluster);
            mapControl.ShapeFields["Cluster"].Type =
typeof(System.Double);

        }

        /****Cria a arvore com a geração dos Clusters*****/
        CriaPopulacaoInicial(ref mapControl, numCluster, dblLimites,
 matrizDeVizinhaca);
        concertaClustersPequenos(ref mapControl, matrizDeVizinhaca,
 numCluster, 5);

        testePintaMapa(ref mapControl, "Cluster", 5);
        //mapControl.SaveAsImage(" C:\\Programa C#
\\POPULACAO" + pop.ToString() + ".Jpeg", MapImageFormat.Jpeg);
        Application.DoEvents();

        mapControl.Update();
        mapControl.Refresh();

        /****** Guarda o valor da Função Objetivo.  ****/

        //(Minimizar)
        double dummyEntropia = 0;
        double[] vetorEntropia = new double[numCluster];
        double[] dblJanela = new double[numCluster];


        funcaoObjetivoEntropiaContinuoUnivariada(mapControl, numCluster,
strVariaveis, ref dummyEntropia, ref vetorEntropia, ref dblJanela);

        //Guarda o individuo na posição dele, caso
seja um dos numMelhores
```

```
        dummy_t = 0;
        for (int i = 0; i < numMelhores; i++)
        {
            if (pop <= numMelhores - 1)
            {
                FazTrocaDePosicao(mapControl, ref dummyMelhores, pop);
                dblMelhoresEntropia[pop] = dummyEntropia;
                vetorDummy[pop] = pop;
            }

            if (pop > numMelhores - 1)
            {
                if (pop == numMelhores && i == 0)
                {
                    if (dummyEntropia < dblMelhoresEntropia[i])
                    {
                        FazTrocaDePosicao(mapControl, ref dblMelhores, i);
                        dblMelhoresEntropia[i] = dummyEntropia;
                    }
                }
                else if (pop > numMelhores)
                {
                    if ((dummyEntropia < dblMelhoresEntropia[i]) && dummy_t == 0)
                    {
                        //Impede que mais de uma posição seja alterada
                        dummy_t = 1;

                        FazTrocaDePosicao(mapControl, ref dblMelhores, i);
                        dblMelhoresEntropia[i] = dummyEntropia;
                    }
                }
            }
        }
    }
```

```
    //Guarda o melhor valor gerado da população inicial
    guardaValorNoHistorico(ref historico, dblMelhores, 0);
for (int k = 0; k < mapControl.Shapes.Count; k++)
mapControl.Shapes[k]["Cluster"] = dblMelhores[k, 0];
    testePintaMapa(ref mapControl, "Cluster", numCluster);
mapControl.SaveAsImage(" C:\\Programa C#\\MAPA_0.jpg",
 MapImageFormat.Jpeg);
    //Exportando mapa
TextWriter tw__ = new StreamWriter(" C:\\Programa C#
\\Univariado_Continuo_Mapa_Inicial_" + strVariavel + ".txt");
    tw__.WriteLine("MICROCOD" + "\t" + "Cluster");
    for (int i = 0; i < mapControl.Shapes.Count; i++)
    {
        tw__.WriteLine(mapControl.Shapes[i]["MICROCOD"].ToString() +
 "\t" + mapControl.Shapes[i]["Cluster"].ToString());
    }
    tw__.Close();

    //Número de poligonos por Cluster
    numPoligonosPorCluster(mapControl, ref vetCluster);

    /*****Começa o Crossing Over****/

    double dummyEntropia2 = 0;//(Minimizar)
    double[] vetorEntropia2 = new double[numCluster];
    double[] dblJanela2 = new double[numCluster];

double[,] vetorDeTransicao = new double[(int)((mapControl.Shapes.Count
 * mapControl.Shapes.Count) / 2), 2];

    for (int j = 0; j < numMelhores; j++)
    {
        for (int i = 0; i < numIteracoes; i++)
```

```
        {
            pgBar.Increment(1);
            Application.DoEvents();

            //Coloca o j-ésimo melhor no mapa
            for (int k = 0; k < mapControl.Shapes.Count; k++)
 mapControl.Shapes[k]["Cluster"] = dblMelhores[k, 0];

    //Concerta Clusters pequenos
    concertaClustersPequenos(ref mapControl, matrizDeVizinhaca, numCluster, 5);

    //Faz a mutação no mapa
    mapaCrossingOver(ref mapControl, matrizDeVizinhaca, propCrossing, numCluster);

     //Verifica se houve melhora e guarda caso seja necessário
     funcaoObjetivoEntropiaContinuoUnivariada(mapControl, numCluster,
strVariaveis, ref dummyEntropia2, ref vetorEntropia2, ref dblJanela2);

    //Concerta clusters pequenos
    concertaClustersPequenos(ref mapControl, matrizDeVizinhaca,
 numCluster, 5);

            dummy_t = 0;

            for (int l = 0; l < numMelhores; l++)
            {
                //tw2.WriteLine(j.ToString() + "\t"
+ i.ToString() + "\t" + l.ToString());

                if (dummyEntropia2 < dblMelhoresEntropia[l] && dummy_t == 0)
                {
                    FazTrocaDePosicao(mapControl, ref dblMelhores, l);

                    dblMelhoresEntropia[l] = dummyEntropia2;
```

```
                    guardaValorNoHistorico(ref historico, dblMelhores, l);

                    //Impede que mais de uma posição seja alterada
                    dummy_t = 1;
                }
            }

            mapControl.Refresh();
            mapControl.Update();

            testePintaMapa(ref mapControl, "Cluster", 5);
            //if (j == 0) mapControl.SaveAsImage(" C:\\Programa C#
\\MAPA_C_" + i.ToString() + ".jpeg", MapImageFormat.Jpeg);
        }
    }

    //Guarda o melhor no mapa
for (int k = 0; k < mapControl.Shapes.Count; k++)
mapControl.Shapes[k]["Cluster"] = dblMelhores[k, 0];
    double[] dblProbabilidade = new double[mapControl.Shapes.Count];

    mapControl.Refresh();
    mapControl.Update();

    double[] janelas = new double[numCluster];
    probabilidadeContinua(ref mapControl, numCluster, ref janelas, strVariavel);

    testePintaMapa(ref mapControl, "Cluster", numCluster);
    //Exportando mapa
TextWriter tw_ = new StreamWriter(" C:\\Programa
 C#\\Univariado_Continuo_Mapa_Cluster_" + strVariavel + ".txt");
    tw_.WriteLine("MICROCOD" + "\t" + "Cluster"+ "\t" + "PROBABILIDADE");
    for (int i = 0; i < mapControl.Shapes.Count; i++)
    {
        tw_.WriteLine(mapControl.Shapes[i]["MICROCOD"].ToString() + "\t" +
```

```
 mapControl.Shapes[i]["Cluster"].ToString() + "\t" +
mapControl.Shapes[i]["PROBABILIDADE"].ToString());
    }
    tw_.Close();

    //Exportando mapa
TextWriter tw___ = new StreamWriter(" C:\\Programa C#
\\Univariado_Continuo_Mapa_Janelas_" + strVariavel + ".txt");
    tw___.WriteLine("Cluster" + "\t" + "Janela");
    for (int i = 0; i < numCluster; i++)
    {
        tw___.WriteLine(i.ToString() + "\t" + janelas[i].ToString());
    }
    tw___.Close();

    MessageBox.Show("Acabou!", "Final.");

}
```

REFERÊNCIAS BIBLIOGRÁFICAS

EVEN, S. *Graph Algorithms*. [S.l.]: Computer Science Press, 1979.

NETTO, P. *Grafos Teoria, Modelos, Algoritmos*. [S.l.]: Edgard Blucher, 2006.

COVER, T. M.; THOMAS, J. A. *Elements of Information Theory 2nd Edition (Wiley Series in Telecommunications and Signal Processing)*. Wiley-Interscience, 2006. Hardcover. ISBN 0471241954. Disponível em: <http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0471241954>.

SILVERMAN, B. *Density Estimation for Statistics and Data Analysis*. [S.l.]: Chapman and Hall, London., 1986.

WANG, M.-C.; RYZIN, J. V. A class of smooth estimators for discrete distributions. *Biometrika*, v. 68, n. 1, p. 301–309, 1981. Disponível em: <http://biomet.oxfordjournals.org/cgi/content/abstract/68/1/301>.

CHIU, S. Bandwidth selection for kernel density estimation. *Annals of Statistics*, v. 33, p. 1883 - 1905, 1991.

DAMASCENO, E. *Escolha do parâmetro de suavidade em estimação funcional*. Tese (PhD em Estatística) — UFMG/Departamento de Estatística, 2000.

BESSEGATO, L. e. a. Rotinas em r para técnicas de suavização por núcleos estimadores. *Relatório técnico - UFMG. Departamento de estatística*, 2006.

COLLET; RENNARD, J.-P. *Stochastic Optimization Algorithms*. Apr 2007. Disponível em: <http://arxiv.org/abs/0704.3780>.

JENKS, G. The data model concept in statistical mapping. *International Cartographic Association ed. International Yearbook of Cartography 7*, p. 186 - 190, 1967.

CHOYNOWSKI, M. Maps based on probability. *Journal of American Statistical Association*, v. 54, p. 385 - 388, 1959.

CRESSIE, N. A. *Statistics for Spatial Data*. [S.l.]: Wiley Interscience, 1993.

RONALD, J. E. e. a. The entropy of a poisson distribution: Problem 87-6. *SIAM Review 30*, p. 314 - 317, 1988.

CARVALHO, A. e. a. *Clusterização dos municípios brasileiros. IPEA*. [S.l.]: Dinâmica dos Municípios., 2007. 181 208 p.

VETTERLING, W. T.; FLANNERY, B. P. *Numerical Recipes in C++: The Art of Scientific Computing*. Cambridge University Press, 2002. Hardcover. ISBN 0521750334. Disponível em: <http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0521750334>.