Structural Graph Extraction from Images

Antonio-Javier Gallego-Sánchez[1], Jorge Calera-Rubio[1], and Damián López[2]

Abstract We present three new algorithms to model images with graph primitives. Our main goal is to propose algorithms that could lead to a broader use of graphs, especially in pattern recognition tasks. The first method considers the q-tree representation and the neighbourhood of regions. We also propose a method which, given any region of a q-tree, finds its neighbour regions. The second algorithm reduces the image to a structural grid. This grid is postprocessed in order to obtain a directed acyclic graph. The last method takes into account the skeleton of an image to build the graph. It is a natural generalization of similar works on trees [8, 12]. Experiments show encouraging results and prove the usefulness of the proposed models in more advanced tasks, such as syntactic pattern recognition tasks.

1 Introduction

Many fields take advantage of graph representations, these fields include but are not limited to: biology, sociology, design of computer chips, and travel related problems. This is due to the fact that graph are suitable to represent any kind of relationships among data or their components. The expressive power of graph primitives has been specially considered in pattern recognition tasks, in order to represent objects [7] or bioinformatics [13]. Therefore, the development of algorithms to handle graphs is of major interest in computer science.

In this paper we present three new methods to model images using graph primitives. The main goal we aim to achieve is to propose algorithms that could lead to a broader use of graphs. In that way, the work of [4] shows that this kind of contribution is suitable and interesting.

[1] Departamento de Lenguajes y Sistemas Informáticos, University of Alicante, Spain, e-mail: {jgallego,calera}@dlsi.ua.es ·
[2] Departamento de Sistemas Informáticos y Computación, Technical University of Valencia, Spain, e-mail: dlopez@dsic.upv.es

The first method we propose (Section 2) takes into account the neighbour paths within a q-tree [2]. We present algorithms that extend other works for neighbourhood calculations in spatial partition trees [5, 11]. These works generally calculate only the 4-neighbours of each region, or use two steps to calculate the corner neighbours. The proposed method completes these works with an estimation of equal complexity for the 8-neighbours at any level of resolution (equal, higher or lower level). Moreover, neighbour calculation can help to improve other applications, speeding up the process to locate the next element, such as in: image representation, spatial indexing, or efficient collision detection.

The second method (Section 3) considers a structural grid of the image which is then postprocessed to obtain the graph model. This method maintains those desiderable features of q-tree representation, that is: the resolution is parametrizable and allows to reconstruct the image.

The third method (Section 4) extends a similar method used on trees [12, 8]. The initial image is preprocessed to obtain the skeleton. This skeleton is then traversed to build a graph that summarizes the core structure of the image.

2 Neighbourhood Graphs

Using the tree defined by a q-tree (see Fig. 1) and the proposed algorithm for neighbourhood calculation (see section 2.1), a directed acyclic graph (dag) can be extracted to represent the sample and to provide information such as: neighbourhood, structural traversal, resolution or thickness, in addition to allow its reconstruction.

The graph is created walking top-down the tree, and from left to right. For each nonempty node, its 8-neighbours are calculated. It is important that: 1) Do not add arcs to processed nodes. 2) If it has to create an arc from a non-pointed node to one that is already pointed: then the direction of the arc is changed. These criteria ensures that the graph is acyclic.

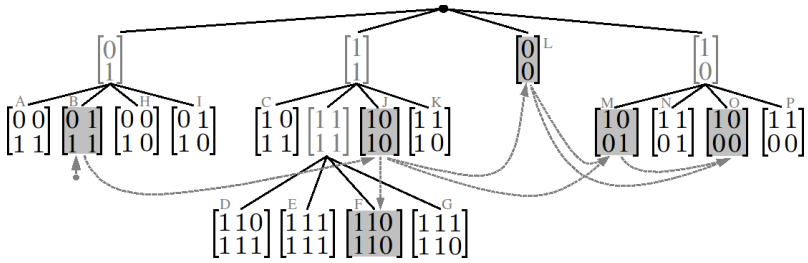

Fig. 1 Tree obtained from the q-tree of the Fig. 2(c), and its corresponding neighbour graph.

2.1 Calculating Neighbours

Binary encoding is used to encode the position of partitions whithin the image and their corresponding traversal path down the q-tree [5]. Each dimension is encoded with 1 bit coordinate, describing the direction as positive (1) or negative (0). A location array is defined using this encoding (Fig. 2(a)). In q-trees, this array has two rows: horizontal and vertical coordinates. Positive directions (up and right) are represented by a 1, and the negative directions (down and left) by 0 (Fig. 2(b)). In general, the location array associated with a given node is defined as the location array of his father, but adding on the right a new column with its own encoding. For each new partition, the origin of the reference system is placed in the center of the area where the subdivision is done. In Fig. 2(c), the second level node K is represented by the $\left[\begin{smallmatrix}1\\1\end{smallmatrix}\right]$ of his father, and then adding the $\left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]$ for its own level.

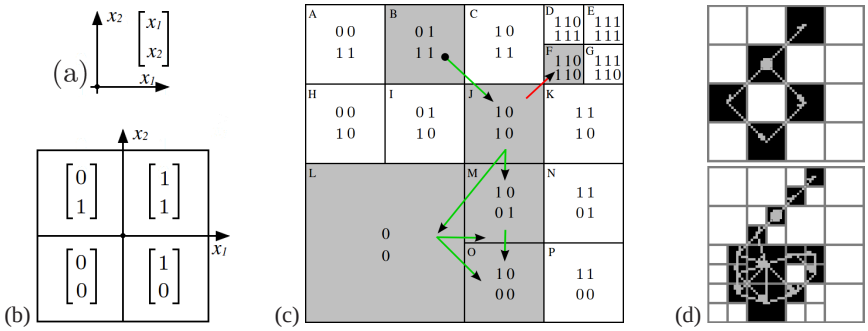

Fig. 2 (a) Reference system and location array, (b,c) Q-trees with the corresponding location arrays, (d) Graphs extracted using different truncate levels.

2.1.1 Algorithm

To calculate the neighbours of a given node, the location array described above is used. The proposed algorithm distinguishes two cases: face neighbours (two regions share more than one point, such as A with B in Fig. 2(c)), and corner neighbours (they share one point, A with I in Fig. 2(c)).

2.1.2 Face Neighbours

To calculate the face neighbour of a given node, the algorithm needs to know its location array (M), and the dimension dim and the direction dir in which the neighbour has to be calculated. The algorithm (see Sec. 2.1.3) visits each bit of the location array (M) in the given dimension from right to left, and negates the bits until the result is equal to the explored direction (1: positive, 0: negative).

As an example, the horizontal neighbour of I (Fig. 2(c)) in the positive direction (1) is calculated. The algorithm begins with the location array and negates each bit in the top row (dim 1) from right to left (indicated with a bar) until reaching the halt condition: the result is equal to the explored direction.

$$I = \begin{bmatrix} 0 & \overline{\mathbf{1}} \\ 1 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} \overline{\mathbf{0}} & 0 \\ 1 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} = J$$

2.1.3 Corner Neighbours

In the case of corner neighbours, the algorithm has to modify the two dimensions of the location array. As for the face neighbours, it begins with the location array M, but in this case it receives as input an array dir with both horizontal and vertical directions. Below is the complete algorithm to compute face and corner neighbours:

```
function GetNeighbour( M, dim, dir )
  level := |M[0]|    // Level is equal to the number of columns
  if( |dir| == 1 ) {      // Face neighbours
    do {
      M[dim][level] := ! M[dim][level]
      level := level − 1
    } while( level ≥ 0 ∧ M[dim][level] != dir )
  } else {      // |dir| == 2 → Corner neighbours
    flag1 := false ; flag2 := false
    do {
      if( !flag1 )
        M[0][level] := ! M[0][level]
        if( M[0][level] == dir[0] )   flag1 := true
      if( !flag2 )
        M[1][level] := ! M[1][level]
        if( M[1][level] == dir[1] )   flag2 := true
      level := level − 1
    } while( level ≥ 0 ∧ ( !flag1 ∨ !flag2))
  }
  return M
```

As example, the corner neighbours of N (Fig. 2(c)) are calculated:

$$N = \begin{bmatrix} 1 & \overline{\mathbf{1}} \\ 0 & \mathbf{1} \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 \\ \overline{\mathbf{0}} & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} = J \quad \Big\| \quad N = \begin{bmatrix} 1 & \overline{\mathbf{1}} \\ 0 & \mathbf{1} \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} = O$$

2.1.4 Type of Nodes

It is important to differentiate the type of a node within the structure (Fig. 1). The function *TypeOfNode(M)* returns this type, it can be: *leaf-node* (the node exists and has no children), *inner-node* (the node exists and has children) or *no-node* (the node does not exist). If the location array is a *leaf-node* the algorithm ends. If it is an *inner-node*, then the algorithm has to calculate the higher-level neighbours (Sec. 2.1.6). If the node does not exist, the algorithm has to calculate the lower-level neighbours (Sec. 2.1.5).

2.1.5 Lower Resolution Neighbours

In this case, the algorithm first calculates the same resolution neighbour using *M := GetNeighbour(M, dim, dir)*; and then eliminates the final column/s (on the right side) until it finds a *leaf-node*:

```
function GetLowerLevel( M )
  do {
    M_{2×n} ← M_{2×n−1}
  } while( TypeOfNode( M ) == no-node )
  return M
```

Example: region D in Fig. 2(c).

$$D=\begin{bmatrix}1&1&\overline{\mathbf{0}}\\1&1&1\end{bmatrix}\Rightarrow\begin{bmatrix}1&\overline{\mathbf{1}}&1\\1&1&1\end{bmatrix}\Rightarrow\begin{bmatrix}1&0&1\\1&1&1\end{bmatrix}\Rightarrow \textit{no-node}\Rightarrow\begin{bmatrix}1&0&\not{1}\\1&1&\not{1}\end{bmatrix}\Rightarrow\begin{bmatrix}1&0\\1&1\end{bmatrix}=C$$

2.1.6 Higher Resolution Neighbours

The algorithm first calculates the neighbour at the same level of resolution using $M := GetNeighbour(M,\ dim,\ dir)$. If M corresponds to an *inner-node*, then the higher-level neighbour/s is/are calculated. It is worth to note that, face neighbours may have two or more neighbours (e.g. L with H and I in Fig. 2(c)), whereas corner neighbours only have one neighbour. In this process, columns are added on the right side of the matrix M until reaching a *leaf-node*. These columns are created in the opposite direction of the dimension/s of interest. See the pseudocode below:

```
function GetHigherLevel( M, dim, dir )
  L := ⟨∅⟩      // Result set of higher-level neighbours
  if( |dir| == 1 ) {      // Face neighbours
    for( i = 1; i ≤ 2, ++i) {
      N_i[dim][0] :=!dir  ;  N_i[ ! dim][0] := i − 1
      M_i := M ⊕ N_i
      if( TypeOfNode( M_i ) == leaf-node )   L ← M_i
      else   L ← GetHigherLevel( M_i, dim, dir )
    }
  } else {      // |dir| == 2 → Corner neighbours
    do {
      N[0][0] :=!dir[0]  ;  N[1][0] :=!dir[1]
      M := M ⊕ N
    } while( TypeOfNode( M ) != leaf-node )
    L ← M
  }
  return L
```

The symbol $\oplus$ denotes the concatenation of a matrix M with a vector N:

$$M\oplus N=\begin{bmatrix}x_0^1&x_1^1\\x_0^2&x_1^2\end{bmatrix}\oplus\begin{bmatrix}y^1\\y^2\end{bmatrix}=\begin{bmatrix}x_0^1&x_1^1&y^1\\x_0^2&x_1^2&y^2\end{bmatrix}$$

Below is an example of higher-level neighbours calculation (see Fig. 2(c)):

$$C=\begin{bmatrix}1&\overline{\mathbf{0}}\\1&1\end{bmatrix}\Rightarrow\begin{bmatrix}1&1\\1&1\end{bmatrix}\Rightarrow \textit{inner-node}\Rightarrow\left\langle\begin{bmatrix}1&1&0\\1&1&1\end{bmatrix}=D;\quad\begin{bmatrix}1&1&0\\1&1&0\end{bmatrix}=F\right\rangle$$

2.1.7 Neighbourhood of Border Regions

When a region is on the border of the structure, do not present neighbours on that border side. In this case, the algorithm ends without reaching the halt condition. E.g. node A (horizontal-left):

$$A = \begin{bmatrix} 0 & \overline{\mathbf{0}} \\ 1 & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} \overline{\mathbf{0}} & 1 \\ 1 & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \Rightarrow border$$

3 Structural Grid Graph

We here propose an algorithm to model images using graph primitives. The main goal of this new proposal is to maintain the better properties of q-trees but introducing higher variability in the incoming and outgoing degrees. Thus, the properties to fulfill are: 1) The model should consider the resolution as a parameter, allowing higher and lower resolution representations. 2) The representation should contain enough information to reconstruct the original image.

Without loss of generality, we will consider directed acyclic graphs and two colour images: *background* and *foreground* colours. Starting from the top-left corner, the algorithm divides the image into squared regions of a given size k. Those regions with foreground colour will be considered the nodes of the graph. The edges are defined from a foreground region (node) to those adjacent-foreground regions to the east, south and southeast. Figure 3 shows an example.

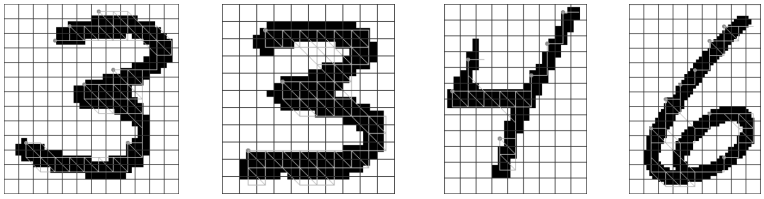

Fig. 3 Some examples of the grid graph modeling are shown. Those regions with at least 20% black pixels are considered foreground. Root nodes are marked in gray.

4 Skeleton Graphs

The last method models the graph using the skeleton that defines the shape of the figure. It is a natural extension of the algorithms proposed for trees in [8, 12]. First, a thinning process is applied to the image [1], and then the graph is generated as:

1. The top left pixel of the skeleton is chosen as the root of the graph.
2. Every node in the graph will have as many children as unmarked neighbour pixels has the current pixel. For each child, an arc is created following that direction until one of the following criteria is true:

a. The arc has the maximum fixed parameter size (window parameter).
b. The pixel has no unmarked neighbours (terminal node).
c. The pixel has more than one unmarked neighbour (intersection).
d. The pixel is marked (existing node).

3. A new node is assigned to every end of arc pixel obtained by the previous step (a,b,c), or it is joined with the corresponding node (d). If the node has unmarked neighbours, then go to step 2, otherwise it terminates.

Fig. 4 Skeleton and graph results.

5 Experimentation

This section aims to demonstrate the usefulness of the proposed models. For this reason, it uses simple classification methods, without stochastic layer and using only the structure of graphs (without labels). Results prove the correct separation of classes, in addition to be a baseline for future research.

To perform the experiments, 5 thousand images of digits (10 classes), with resolution of 64x64 pixels, from the NIST dataset are used. Graphs have been extracted using the three proposed methods, but varying the parameters (truncate level, size of cells, and window size) (see Fig. 5). Six methods are used to perform the classification: First, a feature vector is calculated using the graph degree distribution [10], and then classified using Naive Bayes, Random Forest, SVM and k-NN [6]. The 75% of the dataset is used for training and the rest for test. A preliminar error correcting distance is also defined to test the proposed methods. It is a simple algorithm with linear complexity. For this distance, the classification is performed using leaving-one-out. A semi-structural method of Flow Complexity is also used [3]. It analyzes the structure of the graph based on feature selection via spectral analysis and complexity.

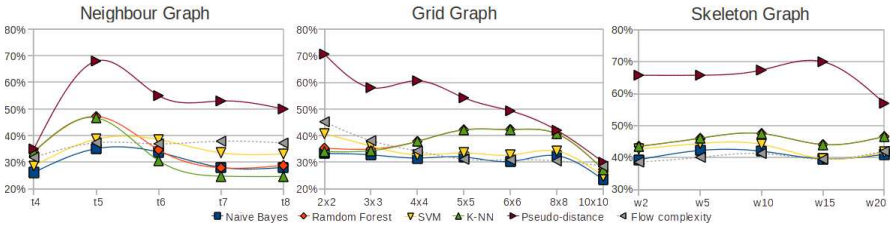

Fig. 5 Results using different classification methods.

6 Conclusions

This paper presents three methods for extracting graphs from images. The obtained graphs introduce a variability that should create new characteristic features in the modelling of the classes of a pattern recognition task, and thus, it should ease a syntactic approach. Experiments show encouraging results which prove the usefulness of the proposed models, the correct separation of classes, in addition to be a baseline for future work.

The first method creates neighbourhood paths or segmentations within the q-tree, providing more data and creating new possibilities of operations and applications. The second algorithm maintains the features of q-tree representation, that is: the resolution is parametrizable and allows to reconstruct the original image. The third method takes into account the skeleton of the shape to build the graph, summarizing the core structure of the image.

Acknowledgments

This work is partially supported by Spanish MICINN (contract TIN2011-28260-C03-01, contract TIN2009-14205-C04-C1) and CONSOLIDER-INGENIO 2010 (contract CSD2007-00018).

References

1. J. M. Cychosz. Thinning algorithm from the article: Efficient binary image thinning using neighbourhood maps. *Graphics Gems IV. Academic Press*, p. 465–473, 1994.
2. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational geometry*, pages 291–306. Springer-Verlag, 2000.
3. F.Escolano, D.Giorgi, E.R.Hancock, M.A.Lozano, and B.Falcidieno. Flow complexity: Fast polytopal graph complexity and 3d object clustering. *GbRPR*, 2009.
4. M.Flasinski, S.Myslinski. On the use of graph parsing for recognition of isolated hand postures of Polish Sign Language. *Pattern Recognition*, 43:2249–2264, 2010.
5. M. Goodchild. Quadtree algorithms and spatial indexes. *Technical Issues in GIS, NCGIA, Core Curriculum*, (37):5–6, 1990.
6. M.Hall, E.Frank, G.Holmes, B.Pfahringer, P.Reutemann, I.H.Witten. The WEKA Data Mining Software: An Update. *SIGKDD Explorations*, 11(1):10–18, 2009.
7. J. Liu, M. Li, Q. Liu, H. Lu, and S. Ma. Image annotation via graph learning. *Pattern Recognition*, 42:218–228, 2009.
8. D. López and I. Piñaga. Syntactic pattern recognition by error correcting analysis on tree automata. *Advances in Pattern Recognition*, LNCS 1876, p. 133–142, 2000.
9. R. G. Luque, J. ao L.D. Comba, and C. Freitas. Broad-phase collision detection using semi-adjusting bsp-trees. In *ACM i3D*, pages 179–186, 2005.
10. M.E.J.Newman. The structure and function of complex networks. *SIAM*, 45. 2003.
11. J. Poveda and M. Gould. Multidimensional binary indexing for neighbourhood calculations in spatial partition trees. *Comput. Geosci.*, 31(1):87–97, 2005.
12. J.R.Rico-Juan, L.Micó. Comparison of AESA and LAESA search algorithms using string and tree edit distances. *Pattern Recognition Letters*, 24:1427–1436, 2003.
13. H. Shin, K. Tsuda, and B. Schölkopf. Protein functional class prediction with a combined graph. *Expert Systems with Applications*, 36:3284–3292, 2009.