Chapter 4

Binary Partition Tree

4.1 Definition

The Binary Partition Tree (BPT) is a structured representation of the regions that can be obtained from an initial partition. In other words, it is a structured representation of a set of hierarchical partitions in which the finest level of detail is given by the initial partition. The leaves of the tree represent regions that belong to this initial partition. The remaining nodes of the tree are associated to regions that represent the union of two children regions. The root node usually represents the entire image support. This representation should be considered as a compromise between representation accuracy and processing efficiency. Indeed, all possible mergings of regions belonging to the initial partition (described by the RAG of the initial partition) are not represented in the tree. Only the most "likely" or "useful" merging steps are represented in the Binary Partition Tree. The connectivity encoded in the tree structure is binary in the sense that a region is explicitly connected to its sibling (since their union is a connected component represented by the parent), but the remaining connections between regions of the original partition are not represented in the tree. Therefore, the tree encodes only part of the neighborhood relationships between the regions of the initial partition.

The Binary Partition Tree should be created in such a way that the most "interesting" or "useful" regions are represented. This issue can be application dependent. However, a possible solution, suitable for a large number of cases, is to create the tree by keeping track of the merging steps performed by a segmentation algorithm based on region merging. In the following, this information is called the *merging sequence*. Starting from an initial partition which can be the partition of flat zones or any other pre-computed partition, the algorithm merges neighboring regions following a homogeneity criterion until a single region is obtained.

An example is shown in Fig. 4.1. The original partition involves four regions. The algorithm merges the four regions in three steps. In the first step, the pair of most similar regions, 1 and 2, are merged to create region 5. This is indicated in the Binary Partition

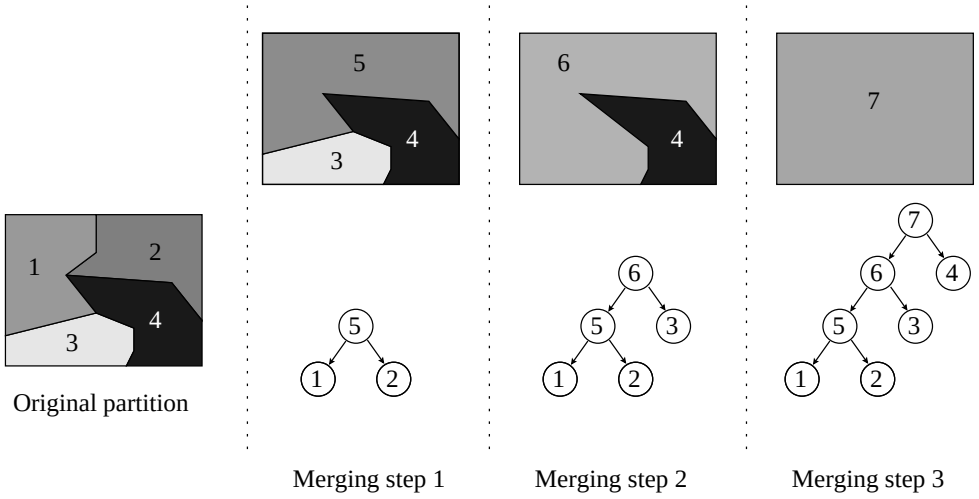


Figure 4.1: Example of Binary Partition Tree creation with a region merging algorithm.

Tree with a node whose label is 5 and that has to children nodes, 1 and 2. Then, region 5 is merged with region 3 to create region 6. Finally, region 6 is merged with region 4 and this crates region 7 corresponding to the region of support of the whole image. In this example, the merging sequence is: $(1,2)|(5,3)|(6,4)$. This merging sequence progressively defines the Binary Partition Tree as shown in Fig. 4.1. In this case the initial partition is made up of 4 regions and thus, the number of nodes of the tree is $4+(4-1)=7$.

In a more general case, we may start creating the tree from an initial partition $\mathcal{P}$ made of $N_{\mathcal{P}}$ regions. The number of mergings that are needed to obtain one region is $N_{\mathcal{P}}-1$. Therefore, the number of nodes of the Binary Partition Tree is thus $2N_{\mathcal{P}}-1$.

In practice, any region based merging algorithm may be used to create the tree. In our work we have developed the General Merging Algorithm for that purpose. Such algorithm is described in the next section.

4.2 General Merging Algorithm

4.2.1 Definition

The proposed strategy is based on an iterative merging algorithm on a Region Adjacency Graph. The approach taken in this work is similar to the one presented in [46, 97]. The novelty in this thesis is the dicussion of the elements that have to be defined to completely specify a merging algorithm, and the efficient implementation of the merging algorithm.

To completely specify a merging algorithm one has to define three notions:

1. The *merging order*: it defines the scanning order of the links, that is the order in which the links are studied to know whether the corresponding linked nodes should be merged or not. This order $O(R_1, R_2)$ is a floating point value and is a function of each pair of neighboring regions R_1, R_2.

2. The *merging critertion*: each time a link is processed, the merging criterion decides if the merging has actually to be done or not. It also is a function $C(R_1, R_2)$ of two neighboring regions R_1 and R_2, but it can only take two values ("merge" and "do-not-merge").

3. The *region model*: it defines how to represent each of the regions of the RAG. When two regions are merged, the model defines how to model its union and what are the main characteristics that should be kept in order to continue the process (that is, to compute future merging criterion values and possibly the future merging order). Let us denote the model of region R with M_R, and let $M_R(p)$ denote the value of the model at pixel p.

The definition of a complete algorithm requires the specification of the merging order, merging criterion and region model. To illustrate this approach, let us intuitively analyze the meaning of each of these notions.

- The merging order defines the homogeneity notion and is closely related to the notion of objects. It can be seen as a measure of the likelihood that two neighboring regions belong to the same object. A similarity measure based on gray-level difference assumes objects to be homogeneous in gray-level, whereas a similarity measure based on squared difference between two affine motion models assumes that objects are homogeneous in motion.

- The merging criterion allows to decide which of the mergings proposed by the merging order should be really done. This means that the merging criterion acts as a sieve among the set of objects defined by the merging order. An example of a simple merging criterion is one that states that all merging should be done until a certain termination criterion is reached (for example, a determined number of regions or a quality criterion). Such criterion is usually used in segmentation algorithms. More complex merging criteria may be used to control more precisely the way regions are merged. For example, if a set of mask of the objects included in the image is available, such information may be used by the merging algorithm to deny mergings between two regions that do not belong to the same object.

- The region model is closely related to the merging order. If we want to merge regions according to a color homogeneity criterion, a region model based on the mean color of

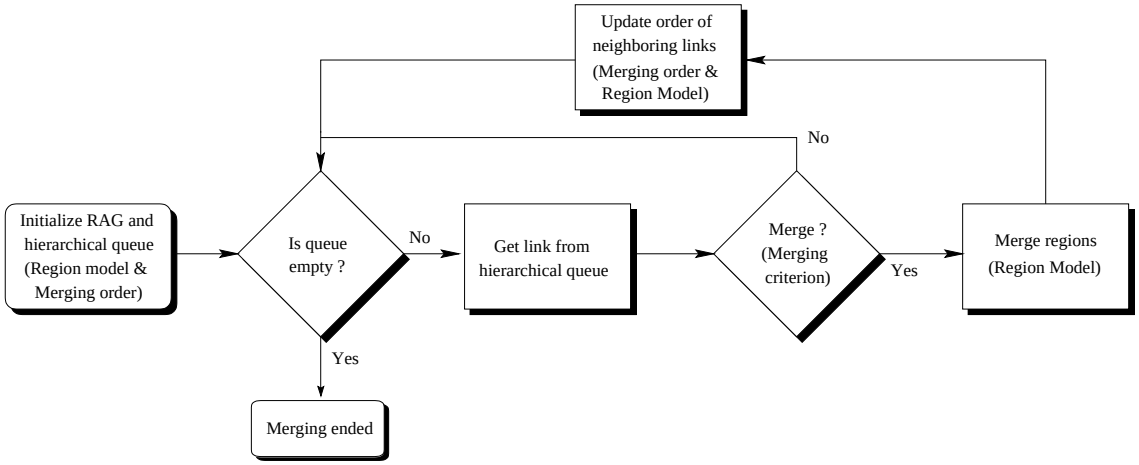


Figure 4.2: Block diagram for the General Merging Algorithm.

each region may be used. On the other hand, if a motion homogeneity criterion is to be used, a model based on an affine motion model may be used. Thus, the merging order and the region model are related to each other.

The next section describes precisely how these three notions are used to implement a region merging algorithm. The details associated to the efficient implementation of the algorithm are discussed in Sec. 4.5.

4.2.2 Merging Algorithm

The General Merging Algorithm works using a Region Adjacency Graph. In Fig. 4.2, the general scheme of the merging process is represented. In each block, the merging notion involved (merging order, merging criterion and region model) is indicated in parentheses. The merging algorithm can be divided into two stages. The first one ("initialization" block) is devoted to the initialization of the structures needed for the merging, i.e. the RAG structure and the hierarchical queue (see Sec. 4.5 for a detailed discussion of such structures). The hierarchical queue is used to index and process the links according to its merging order. Each region is initialized by computing its model. The set of initial regions can be either the set of pixels (each pixel is one region), the set of flat zones or any other pre-segmentation. The next step consists in calculating a homogeneity criterion (that is, the merging order) for each pair of neighboring regions using the model of the associated regions. This information is then introduced into the hierarchical queue. The homogeneity value (represented by a floating point number) defines the insertion point of each link (representing a pair of neighboring regions) into the hierarchical queue.

Once the first stage is completed, the merging process begins. Observe that the merging

algorithm is an iterative process that ends once the hierarchical queue is empty. The first step in this iterative process is to extract the link with highest priority of the queue. The next step consists in deciding whether the link has to be merged or not. As discussed in the previous section, this decision is defined by the merging criterion. If the merging criterion decides not to merge the regions, the algorithm returns to the first block of the merging algorithm. Note that this decision is final in the sense that the link is removed from the queue and the corresponding pair of regions will never be merged. If the merging criterion decides to merge the information of both regions associated to the link is merged together. As a result, the region model has to be updated. Once the regions have been merged, the values of the merging order of the neighboring links are updated. This implies the extraction of the corresponding links from the queue, the computation of the new priority (using the models of neighboring regions) and the insertion of the links into the queue with their new priority. At this point the merged RAG structure has been computed and updated: the iterative process starts again by checking if the queue is empty. The merging process ends when the hierarchical queue is empty.

4.3 BPT construction with the General Merging Algorithm

As it has been explained at the beginning of Sec. 4.1, the tree is created by keeping track of the merging steps performed by a region based merging algorithm. Thus, the General Merging Algorithm may be used to create the Binary Partition Tree.

For that purpose, the three notions have to be defined. The merging order $O(R_1, R_2)$, the merging criterion $C(R_1, R_2)$, and the region model M_R have to be defined. Note that since the merging algorithm keeps on merging regions until a single region is obtained, the tree construction only makes use of the merging order and the region model whereas the merging criterion is trivial: allow the merging of regions until all regions have been merged.

4.3.1 Gray-level homogeneity

Merging order

The first choice that has to be made is how each region is going to be modeled. In this section we assume that the model is constant within the region ($\forall p \in R, M_R(p) = const$). Two simple self-dual models of a gray-level distribution are the mean and the median. In the case of the mean model, one has to compute the average of the pixel gray-level values over the support of the region:

$$M_R = \frac{\sum_{p \in R} f(p)}{A_R} \tag{4.1}$$

where A_R denotes the number of pixels of region R. In order to allow a fast implementation of the merging process, the model of the region $R = R_1 \cup R_2$ should be computed recursively

from the models of the two merged regions R_1 and R_2 (see Sec. 4.5).

$$M_R = \frac{A_{R_1} \times M_{R_1} + A_{R_2} \times M_{R_2}}{A_{R_1} + A_{R_2}}$$

In the case of the median, the approach we have taken is to simply to select the model of the largest region[1]:

$$\begin{aligned} \text{if } A_{R_1} < A_{R_2} &\Rightarrow M_R = M_{R_2} \\ \text{if } A_{R_1} > A_{R_2} &\Rightarrow M_R = M_{R_1} \\ \text{if } A_{R_1} = A_{R_2} &\Rightarrow M_R = (M_{R_1} + M_{R_2})/2 \end{aligned} \tag{4.2}$$

From our practical experience, the use of the median model rapidly defines areas which are homogeneous. These areas can be seen as the core of the final regions. Then, small regions are progressively merged to the core without modifying the model. This property does not hold in the case of the mean, since the merging of a small region with a large one modifies the model of the large region. However, in practice similar results are obtained in both cases. In the sequel, we assume that the mean is used in all cases (except if stated differently). Finally, note that the median and the mean models can be considered as simple zero order models. In Sec. 4.3.3 the model is extended to a first order model in order to deal with motion homogeneity.

Merging order: $O(R_1, R_2)$

The merging order defines the notion of object. It can be seen as a measure of the likelihood that two neighboring regions belong to the same object. In the case of a gray-level image, a natural measure is the squared error between the original image and the model associated to the region of support defined by $R = R_1 \cup R_2$:

$$O(R_1, R_2) = \sum_{p \in R_1 \cup R_2} (f(p) - M_{R_1 \cup R_2}(p))^2 \tag{4.3}$$

where $f(p)$ denotes the gray-level value of the original image at position p. This criterion actually allows the extraction of meaningful objects that are homogeneous in gray-level, however it does not define precisely the contours. This drawback is related to the size dependence of the criterion. Indeed, small regions have a tendency to merge together since the squared error contribution of the union of two small regions is small compared to the contribution resulting from the merging with a large region. An alternative solution is to use the Mean Squared Error (MSE): $O(R_1, R_2) = \sum_{p \in R_1 \cup R_2} (f(p) - M_{R_1 \cup R_2}(p))^2 / (A_{R_1} + A_{R_2})$, where A_{R_1} and A_{R_2} denote the numbers of pixels of regions R_1 and R_2, or the squared difference between the models

[1] Note that the resulting model is not exactly the median of the original pixels since in this case the median is computed in an iterative way.

of the two neighboring regions when a zero order model is used: $O(R_1, R_2) = (M_{R_1} - M_{R_2})^2$. These criteria are independent of the size and provide a very good definition of the contours. However, they do not define in a robust way the regions themselves. In practice, they produce a few large regions surrounded by a very large number of tiny regions. To obtain a compromise between the two previous orders, we propose the following merging order:

$$\begin{aligned} O(R_1, R_2) &= A_{R_1}(M_{R_1} - M_{R_1 \cup R_2})^2 \\ &+ A_{R_2}(M_{R_2} - M_{R_1 \cup R_2})^2 \end{aligned} \tag{4.4}$$

Note that in the case of the median model, if R_1 is smaller than R_2, the model after merging is M_{R_2} and the order reduces to $O(R_1, R_2) = A_{R_1}(M_{R_1} - M_{R_2})^2$. It is the squared difference between the models weighted by the size of the smallest region.

4.3.2 Color homogeneity

The merging order defined in Sec. 4.3.1 can be easily extended to deal with color or multichannel images. In this case, M_R is a multidimensional model and each component is modeled independently. In the case of a zero order model, a constant value is used to model each of the color components the image is made of. The merging order is defined to be a linear combination of the order values defined for each component:

$$O_{color}(R_1, R_2) = \sum_i \omega_i O_i(R_1, R_2) \tag{4.5}$$

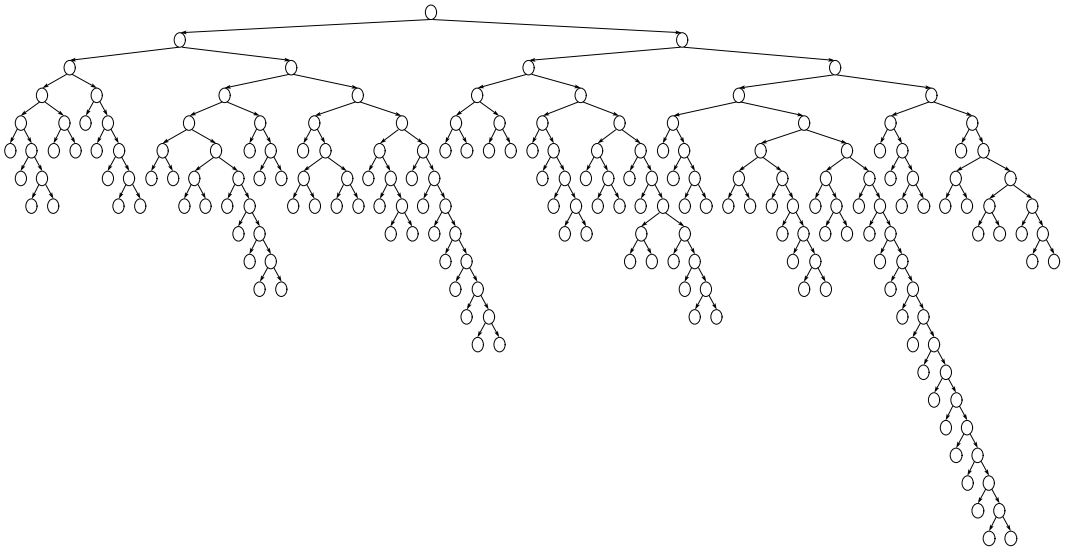
where $O_i(R_1, R_2)$ is the merging order for the i'th component of the image, as discussed in Sec. 4.3.1, and ω_i is the weight associated to the i'th component. The values of ω_i should be set according to the importance of the visual content of each component.

For color images, the color homogeneity criterion should be based on a space such as the YUV or HLS, since those color spaces model more precisely the way the human eye perceives objects than the classical RGB space. In our work we have mainly used the YUV space. Moreover, the luminance (Y) component is known to contain visually more information than the chrominance (U,V) components. Thus, on may state that a higher weight ω_i should be associated to the luminance than to the chrominance components.

In Fig. 4.3 an example of a Binary Partition Tree created using color homogeneity criterion is shown. The initial segmentation, depicted in Fig. 4.3b, has been obtained using the general merging algorithm using a color based region model and order with a merging criterion that allows merging of regions until the termination criterion (in our case 200 regions) has been reached.

4.3.3 Motion homogeneity

It should be noticed that the homogeneity criterion has not to be restricted to color. For example, if the image for which we create the Binary Partition Tree belongs to a sequence of



a) b) c)

Figure 4.3: Example of Binary Partition Tree (top). a) Original image. b) Initial partition with 200 regions. c) Regions of the partition represented by their mean value.

images, motion information may also be used to generate the tree. Assume that we start from and initial segmentation resulting from a spatial segmentation (for instance, a segmentation such as the presented in Fig. 4.3b). The motion can be estimated on each region of the initial partition. The motion estimation, based on differential methods [18, 78], assigns to each region a polynomial model describing the apparent motion in the horizontal and vertical directions. From this motion model, two dense motion fields can be created by assigning to each pixel the values of its horizontal and vertical displacements. These two dense motion fields are now used as input to the merging algorithm. Previous approaches on motion based segmentation may be found in [7, 8, 19, 25, 45, 53, 84, 90].

In our work, a first order region model, $M_R = (M_R^H, M_R^V)$, is used, where $M_R^{\{V,H\}}(p) = \alpha p_x + \beta p_y + \gamma$ with $p = (p_x, p_y)$ denoting the spatial coordinates of the pixel and M_R^H (resp. M_R^V) denoting horizontal (resp. vertical) motion model. Thus, $M_R(p)$ represents the displacement vector at position p. The strategy that has been adopted to obtain the merging order $O(R_1, R_2)$ in our work is based on checking if the motion model of R_1 is able to represent correctly the motion $R = R_1 \cup R_2$, and inversely, if the motion model associated to R_2 is able to model the motion of $R = R_1 \cup R_2$. For that purpose the Displaced Frame Difference (DFD) is used.

$$DFD(R, M_K) = \sum_{p \in R} |f_T(p) - f_{T-1}(p - M_K(p))|$$

where f_T and f_{T-1} are the frames at time instants $t = T$ and $t = T - 1$ respectively, M_K represents the affine motion model of a region K, and $M_K(p)$ is the motion vector evaluated at position p of the image. The merging order is then computed as:

$$O(R_1, R_2) = \min_{K \in \{R_1, R_2\}} \left(\frac{DFD(R, M_K) - (DFD(R_1, M_{R_1}) + DFD(R_2, M_{R_2}))}{\left(\sum_{p \in R} |\nabla f_T(p)|\right) / A_R} \right) \tag{4.6}$$

where $R = R_1 \cup R_2$ and A_R is the number of pixels of R. The DFD is normalized by the gradient of the image, ∇f_T, in order to reduce the effect of possible inaccuracy in the motion estimation. Note that it is possible for the merging order to become negative. Such case may happen if, for instance, the motion model associated to R_2, M_{R_2} models the motion of R_1 better than the model it has associated, M_{R_1}. When two regions merge, the model of the union is the model of the region, R_1 or R_2, that minimizes Eq. 4.6.

In Fig. 4.4, the Binary Partition Tree has been constructed exclusively with the color homogeneity criterion described in the previous section. In this case, it is not possible to concentrate the information about the foreground object (head and shoulders of the foreman object) within a single sub-tree. For example, the face mainly appears in the sub-tree hanging from region A, whereas the helmet regions are located below region D. In practice, the nodes

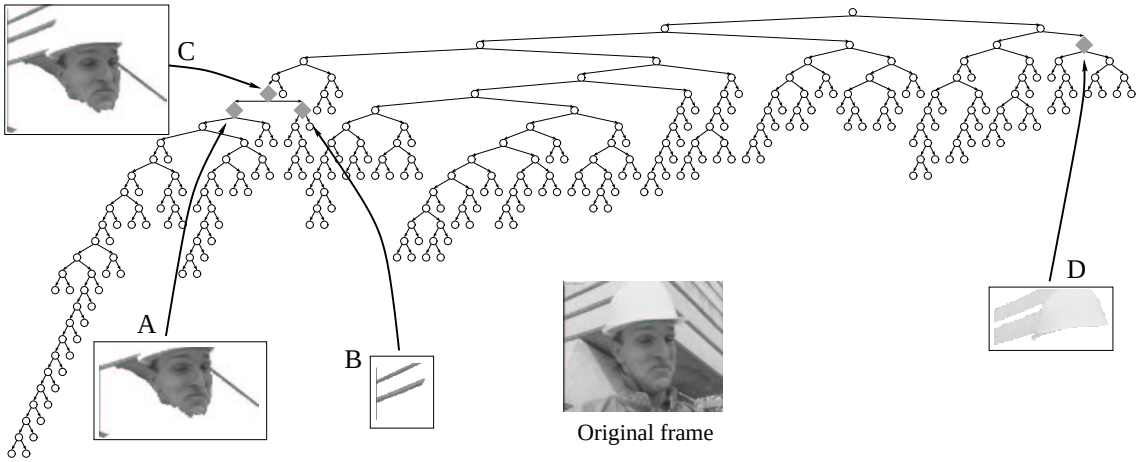


Figure 4.4: Example of Binary Partition Tree created using a color homogeneity criterion.

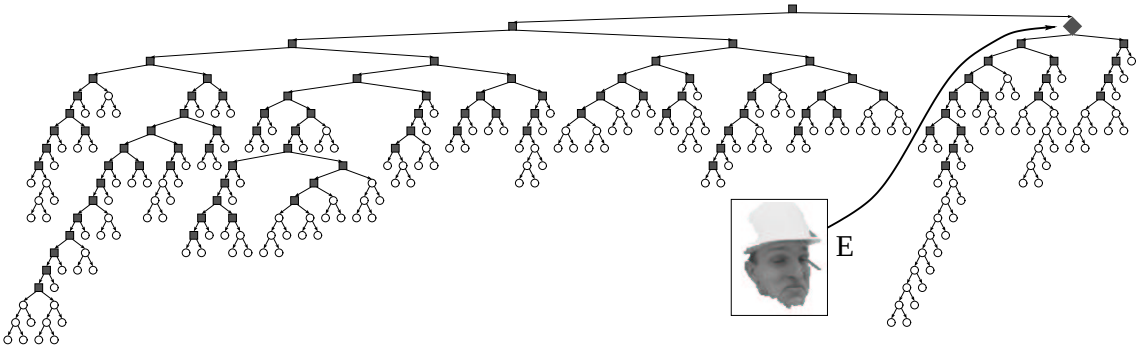


Figure 4.5: Example of Binary Partition Tree created using color and motion homogeneity criteria.

that are close to the root node have no clear meaning because they are not homogeneous in color.

Fig. 4.5 presents and example of Binary Partition Tree created with color and motion criteria. The nodes appearing at the lower part of the tree as white circles correspond to the color criterion, whereas the dark squares correspond to a motion criterion. As can be seen, the process starts with the color criterion as in Fig. 4.4 and then, when a given Peak Signal to Noise Ratio (PSNR) is reached, the associated partition is used to compute the motion on each region and merging algorithm continues merging regions using a motion criterion. Using motion information, the face and the helmet now appear as a single region E.

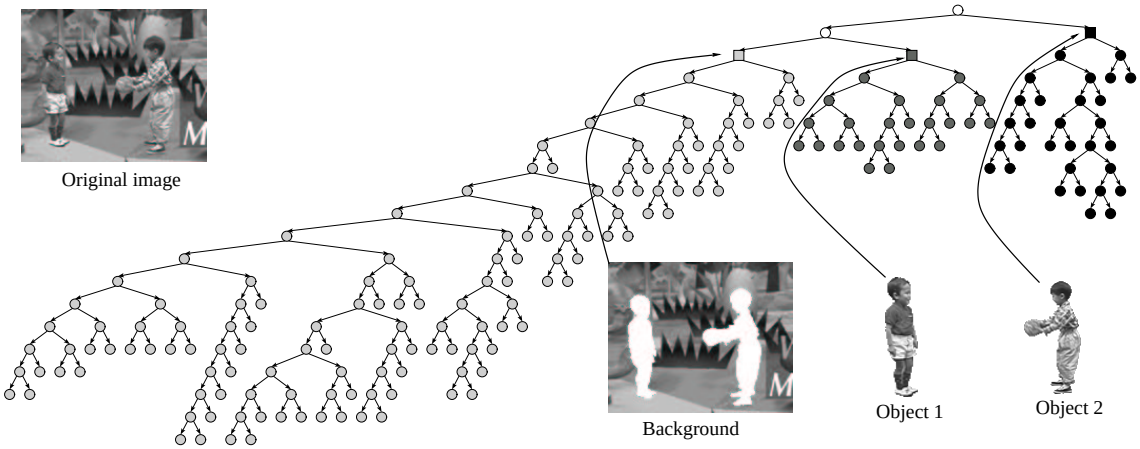


Figure 4.6: Example of Binary Partition Tree creation with restriction imposed by object masks.

4.3.4 Forcing support of nodes

Additional information of previous processing or detection algorithms can also be used to generate the tree in a more robust way. For instance, a mask of a set of objects included in the image can be used to impose constraints on the merging algorithm in such a way that the object itself is represented with only one node in the tree. Typical examples are face, skin, character or foreground object detection.

In this case, the merging is constrained to merge regions within each object mask before dealing with the remaining regions. The merging order is defined as follows: if R_1 and R_2 do not belong to the same object mask its associated distance is infinity, $O(R_1, R_2) = \infty$, if region R_1 (resp. R_2) does not have a neighboring region that belongs to the same object mask than R_1 (resp. R_2). Otherwise, the merging order $O(R_1, R_2)$ is defined to be the distance according to the homogeneity measure that one wants to use.

An example is shown in Fig. 4.6. Assume for example that the original image sequence has been analyzed so that the masks of the two foreground objects (children) are available. In order to create the tree, first only regions inside each of the objects are allowed to merge. When the object is represented as a single node in the graph it is allowed to merge with neighboring regions. The resulting tree is shown in Fig. 4.6. The nodes corresponding to the background and the two foreground objects are represented by squares. The three sub-trees further decompose each object into elementary regions.

An object mask may also be used to define the support of the root node to other than the whole image support. In this particular case, the strategy has been taken in this work is to construct the RAG (over which the merging algorithm is applied) only over the region of support of the object mask. Fig. 4.7 shows an example in which the region of support of the

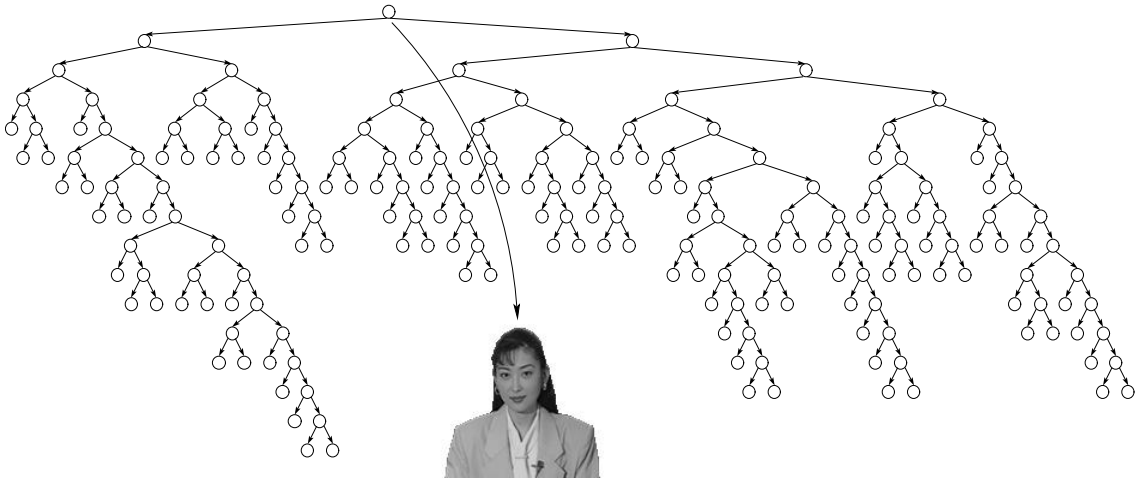


Figure 4.7: Example of Binary Partition Tree creation imposing support of root node.

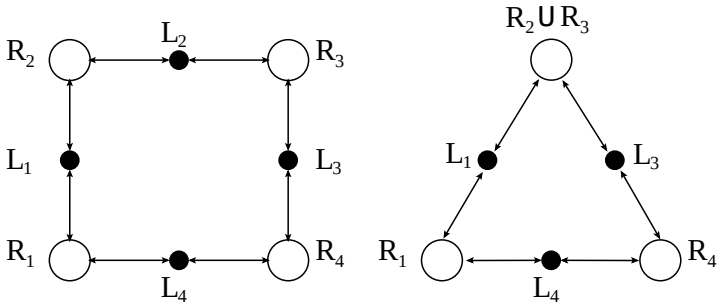
root node is an object rather than the whole image.

4.4 Discussion

The Binary Partition Tree, as the Max and Min-Tree, are scale-space representations. Connected components whose size is small appear near the leaf nodes, whereas large size connected components appear near the root node.

As can be seen, the construction of a Binary Partition Tree is fairly more complex than the creation of a Max-Tree or Min-Tree. However, Binary Partition Trees offer more flexibility because one can choose the homogeneity criterion through the proper selection of the region model and the merging order. Furthermore, if the functions defining the region model and the merging order are self-dual, the tree itself is self-dual. The same Binary Partition Tree can be used to represent f and $-f$.

We want to mention that it seems possible to construct the Max-Tree (and thus, the Min-Tree) using the Binary Partition Tree construction algorithm. Starting from the partition of the flat zones, the approach that should be taken in this case is to use a zero order model and a merging order based on the absolute gray-level value of the flat zones. The merging algorithm merges then regions according to their gray-level values: regions with higher gray-level value should be merged first. The resulting tree is binary and it seems feasible to obtain the Max-Tree after reorganization of the nodes of the tree. The proposed strategy is, however, out of the scope of this work since the merging algorithm is computationally much more expensive than the Max-Tree construction algorithm.



a) Original RAG

b) RAG after merging of regions R_1 and R_2

Figure 4.8: Region Adjacency Graph structure used for the General Merging Algorithm.

4.5 Efficient implementation

The merging algorithm has been implemented following the scheme depicted in Fig. 4.2. We describe here the different techniques that have been used to allow fast access and manipulation of data during the merging process.

4.5.1 Graph structure

The Region Adjacency Graph structure that has been used to implement the merging algorithm is depicted in Fig. 4.8. The graph structure used here is similar to the classical RAG but contains some more information. The white nodes in Fig. 4.8a, called "region-nodes", represent the regions of the image, whereas the black nodes, called "link-nodes", represent the links between regions. Observe that each link-node points to the two region-nodes it is linking and that each region-node points to the neighboring link-nodes. By using this representation, the following advantages arise:

- The access to the set of link-nodes that should be updated after the merging of two region-nodes can be done efficiently. Suppose, for example, that R_2 and R_3 are merged (see Fig. 4.8b). The link to remove is L_2, and the link-nodes whose merging order has to be updated are L_1 and L_3.

- Managing the elements of the hierarchical queue is easier. The queue has to index the order in which the links have to be processed, and this can be performed by inserting (a pointer to) the link-node into the queue instead of the associated neighboring regions.

- Each link-node stores the model of the region resulting from from merging its two associated regions. This model is obtained when the merging algorithm notifies the link-node that it should update itself and recalculate its merging order. Note that the

model of the union is usually needed to obtain the merging order. For instance, when the link L_3 in Fig. 4.8b is notified to perform an update, the model and the merging order associated to the union of R_3 and R_4 is computed and stored on L_3. By using this approach, we do not need to recompute again the model if those regions are merged afterwards.

One clear disadvantage of using the proposed graph structure is the higher memory cost. Each region-node allocates the model for its associated region and each link-node stores the model associated to the merging of its associated region-nodes. See Sec. 4.5.4 for a dicussion of the memory consumption of the algorithm.

4.5.2 Recursivity

Recursivity is an important issue if one wants to avoid redundant computations. "Recursive" here means that, for instance, the model of the union of the two regions should be computed from the models of the two initial regions. The advantage of using recursive algorithms is the higher performance of the algorithm in terms of time. The drawback of this strategy is the need of some overhead information for each model that allows implementing such recursivity. The memory allocation need for each region model is higher and, therefore the memory cost is higher.

Assume that a zero order model is used. The following data can be computed, for instance, in a recursive way

- Region Model: assume that a zero order model based on the mean gray-level value is used. For that purpose, in each node the number of pixels, A_R, associated to the region, and the volume under $f(p)$, $V_R = \sum_{p \in R} f(p)$, needs to be stored. Thus, $M_R = V_R / A_R$. Both values can be updated as follows when regions R_1 and R_2 are merged together: $A_{R_1 \cup R_2} = A_{R_1} + A_{R_2}$ and $V_{R_1 \cup R_2} = V_{R_1} + V_{R_2}$.

- Squared Error: a usual way of setting a termination criterion for a segmentation algorithm is to use a threshold based on the Mean Squared Error. This parameter can also be easily updated in a recursive way when two regions are merged together. For that purpose, in addition to A_R and V_R, $E_R = \sum_{p \in R} f^2(p)$ has to be stored in each of the nodes of the graph. In fact, the squared error is computed as:

$$SE(f, \{R_i\}) = \sum_i \sum_{p \in R_i} (f(p) - M_{R_i})^2$$

  where $f(p)$ represents the gray-level value of the original image, and M_{R_i} represents the zero order model of region R_i. The index i of the previous expression should run through all regions R_i the graph is made of. Expanding the previous expression taking

into account that a zero order model is used it follows:

$$\begin{aligned} SE(f,\{R_i\}) &= \sum_i \sum_{p\in R_i} \left(f^2(p) - 2f(p)M_{R_i} + M_{R_i}^2\right) \\ &= \sum_i \left(\sum_{p\in R_i} f^2(p) - 2M_{R_i} \sum_{p\in R_i} f(p) + A_{R_i} M_{R_i}^2 \right) \end{aligned}$$

Taking into account the expressions of V_R and E_R we obtain:

$$SE(f,\{R_i\}) = \sum_i \left(E_{R_i} - 2M_{R_i}V_{R_i} + A_{R_i}M_{R_i}^2\right) = \sum_i SE(f,R_i)$$

Let us denote with $SE(f,R_i)$ the expression $SE(f,R_i) = E_{R_i} - 2M_{R_i}V_{R_i} + A_{R_i}M_{R_i}^2$.

Assume now that $SE(f,\{R_i\})$ is known for the step j of the merging process and that the partition associated to the graph is $\mathcal{P}^j = \{R_i\}$. If regions R_1 and R_2 are merged together to form the partition $\mathcal{P}^{j+1} = \{R_1 \cup R_2, R_{i\neq 1,2}\}$, we find that:

$$SE(f,\mathcal{P}^{j+1}) - SE(f,\mathcal{P}^j) = SE(f,R_1\cup R_2) - (SE(f,R_1) + SE(f,R_2))$$

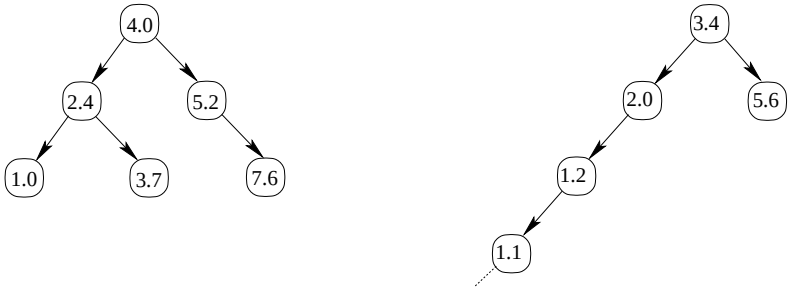
As can be seen, the (mean) squared error may be updated in a straightforward manner using the previous relation. For that purpose, A_R, V_R and E_R have to be stored in each node. Note that the latter measures can be updated easily when two regions merge.

In order to enable an efficient (recursive) computation of the model and squared error (for a zero order based segmentation algorithm) each node of the graph needs to store A_R, V_R and E_R in each of the nodes of the graph. As can be seen, the drawback of this strategy is the higher memory cost, with respect a non-recursive algorithm, due to the overhead to be stored in each node.

4.5.3 Hierarchical Queues and Binary Search Trees

The implementation of the merging algorithm described in Sec. 4.2.2 has strong similarities with efficient implementations of connected operators involving the so-called reconstruction process and of the watershed algorithm [95, 94]. In all cases, the key element of the algorithm is the hierarchical queue. In the context of our general merging algorithm, the main features of the queue should be: first, fast access, insertion and deletion of an element and, second, no constraints on the dynamic range of the priority (floating point ordering).

Hierarchical queue structures have been extensively used for fast implementation of connected operators or for the watershed algorithm [95, 94]. The main difference with the implementation proposed here is that the merging order is a priori defined in the case of classical connected operators or of the watershed, whereas the merging order has to be constantly updated in our case. Therefore, the hierarchical queue has to be updated and re-organized on



a) Binary search tree b) Unbalanced tree

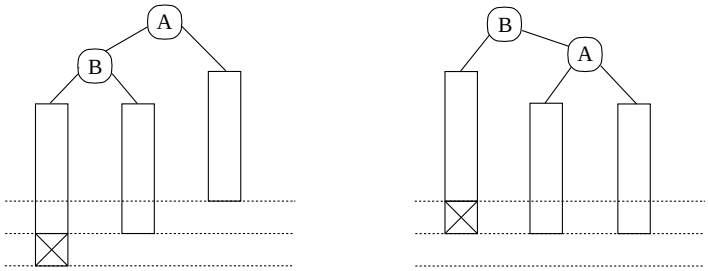
Figure 4.9: Examples of binary search tree, in each node its associated priority is indicated.

line. It can be seen as a dynamic hierarchical queue by contrast to the more classical "static" hierarchical queue. Another drawback of the classical hierarchical queue is that it is generally limited in the range of priority it can deal with and does not allow a floating point priority.

The solution proposed here is based on a binary search tree [29]. Binary search trees are extensively used to implement fast searching, insertion and deletion algorithms and have no constraints on the dynamic range of the data allocated in it. The basic idea behind the implementation of hierarchical queues with binary trees is depicted in Fig. 4.9a. Each node of the tree represents a given priority. A left child node is characterized by having a priority lower than its father, while a right child node has a priority greater than its father. At each node, the link-nodes having the same priority are stored in a FIFO structure.

In order to extract the node (and therefore the list of link-nodes) with highest priority, one begins with the root node and walks down the tree using the right branches until a node with no right branch is found.

Search, insertion and deletion of nodes in the queue can be done in $O(log_2 N)$ steps where N is the number of nodes in the tree [29]. One of the drawbacks of using binary trees to implement hierarchical queues is that the tree may degenerate. This problem happens when the input data (in the merging algorithm the input data is the merging order) is not random. Suppose, for example, that the ordering depends on the area of the region: for example, the higher the area, the lower the priority. As a result, the priority of the links will decrease as the regions grow in area. This results in a tree that "leans" too much to the left (see Fig. 4.9b). In this case, the $O(log_2 N)$ efficiency does not hold anymore, because the tree is becoming a simple FIFO queue. Fortunately, there is a solution to this problem called "balanced trees". The method [29, 99] is based on keeping track of the balance factor of the tree each time an element is inserted or deleted. A rough idea is presented in Fig. 4.10: at each node, the height of the left tree minus the height of the right one is computed (the height at a given node is defined as the length of the longest path from the node to a leaf node). A binary tree is called balanced if the balance factor of every node of the tree never differs by more than ± 1.



a) Unbalancing due to insertion b) Restoring the balance

Figure 4.10: Example of binary hierarchical tree balancing.

Size 352 × 288 Size 720 × 576

Figure 4.11: Images used to test the General Merging Algorithm performance.

If unbalancing occurs because of insertion or deletion, the balancing is reestablished properly by "rotating" some nodes of the tree.

The algorithm involved in the balancing of the tree is very efficient if one has to manage thousands of nodes. Efficient implementations of insertion and deletion using balancing techniques can be found in [99].

It should be noted that in the particular case of our work, the priority of the links is associated to the merging order. As it has been seen in the previous sections (see for instance Eqs. 4.3, 4.4 and 4.6) the merging order has a higher priority as its associated value gets lower. Thus, in the implementation of the hierarchical queue we extract first those items having a lower floating point order.

4.5.4 Performance

Fig. 4.11 shows the images that have been used to test the merging algorithm. The approach taken to perform construct subimages form the test images is described in Sec. 3.4. To test the performance we have started the merging algorithm constructing a graph for each of the previous subimages where each pixel is represented with node. The algorithm merges regions based on a gray-level homogeneity (see Eq. 4.4) until one region is obtained. The tests have been performed on a Pentium II 400Mhz Linux based system with 256Mb of RAM.

CPU time

Fig. 4.12 shows the time performance of the algorithm. The number of seconds (with a resolution of 0.01 seconds) needed to perform the previous described merging process is shown. The number of mergings needed to obtain one region is $2N_{\mathcal{P}} - 1$ where $N_{\mathcal{P}}$ is the number of nodes of the initial partition, in this case made up of individual pixels. As can be seen, the performance of the algorithm is rather high but much less efficient than the Max-Tree construction algorithm since in this case the merging order of the links is updated dynamically during the merging process.

Memory consumption

Fig. 4.13 shows the plot of the memory consumption for the merging algorithm taking several subimages from the image shown on the left of Fig. 4.11. As can be seen, the memory consumption is rather high in comparison with the one of the Max-Tree construction (see Fig. 3.10 on page 49). From our experimental results, we have observed that the memory consumption is quite constant independently of the contents of the image. In fact, we may say that it depends on a high degree on the number of nodes (region-nodes) and links (link-nodes) of the initial graph. Thus, the memory consumption of the merging algorithm can be reduced if the initial number of nodes is reduced. For intance, performing the merging algorithm for the image shown on the left of Fig. 4.11 starting from the pixel level takes about 108Mb, whereas 12Mb are needed if the merging algorithm is started from a partition made up of 500 regions.

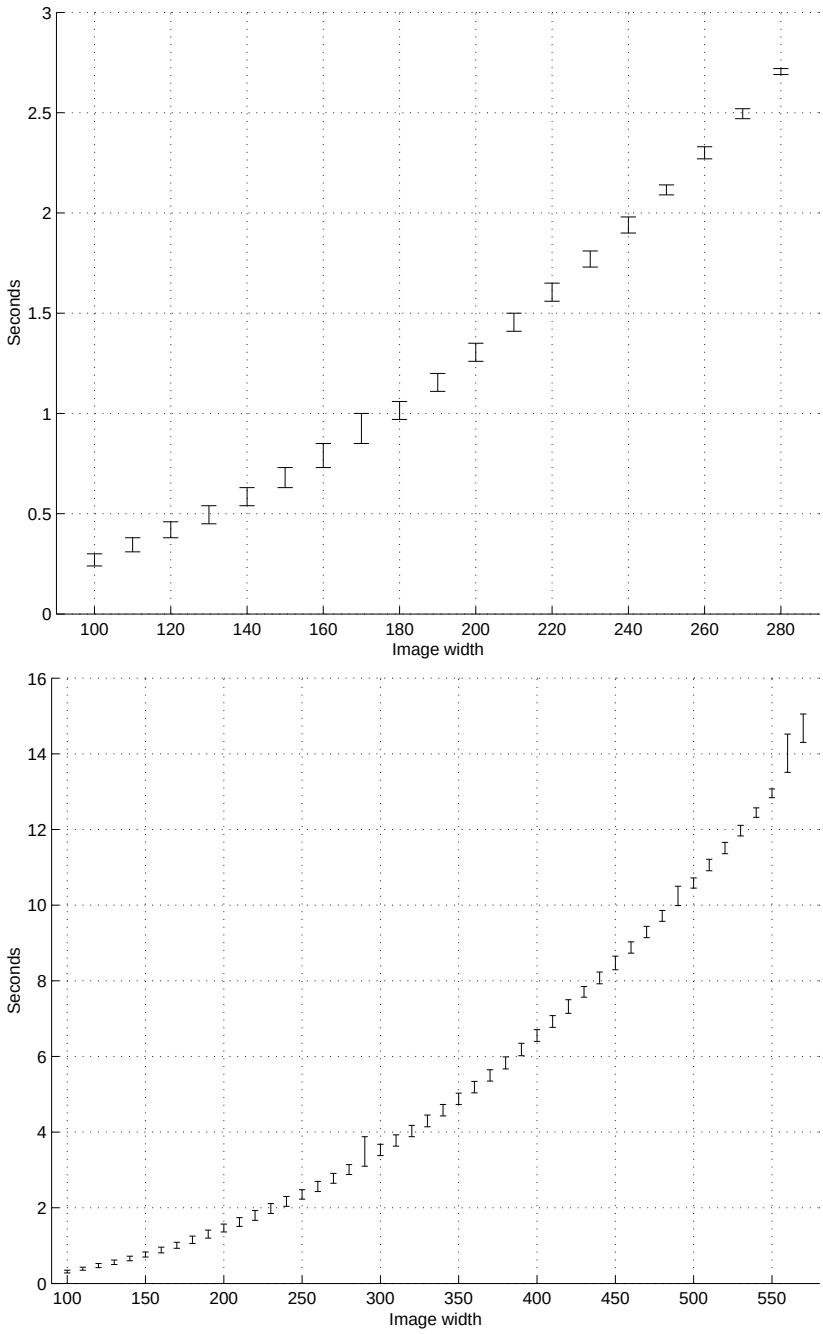


Figure 4.12: Performance of the General Merging Algorithm for several squared subimages. The maximum and minimum measured values is depicted. Top: Plot for the image shown on the left of Fig. 4.11. Bottom: Plot for the image shown on the right of Fig. 4.11.

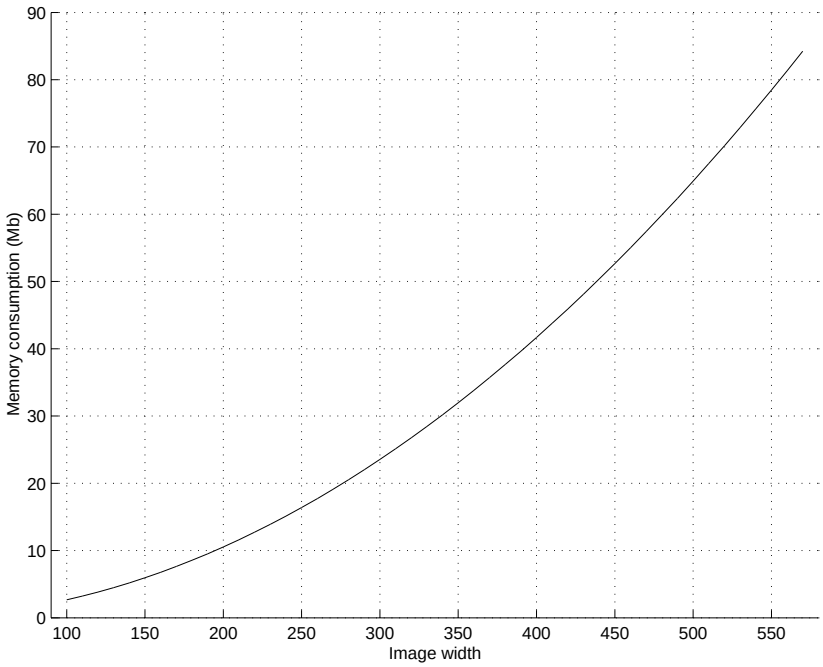


Figure 4.13: Memory consumption, in Megabytes (Mb), for the tree construction using as input several squared subimages of the image shown on the right of Fig. 4.11.