Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia
COPPE UFRJ

USING WEIGHTED VERTEX COLLECTIONS FOR COMPUTING MAP OVERLAY OPERATIONS

José Augusto Sapienza Ramos

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Claudio Esperança

Rio de Janeiro
Junho de 2014

USING WEIGHTED VERTEX COLLECTIONS FOR COMPUTING MAP OVERLAY OPERATIONS

José Augusto Sapienza Ramos

RIO DE JANEIRO, RJ – BRASIL
JUNHO DE 2014

Sapienza Ramos, José Augusto

Using Weighted Vertex Collections for Computing Map Overlay Operations/José Augusto Sapienza Ramos. – Rio de Janeiro: UFRJ/COPPE, 2014.

XII, 68 p.: il.; 29, 7cm.

Orientador: Claudio Esperança

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2014.

Bibliography: p. 65 – 68.

1. Map Overlay. 2. Spatial Database. 3. Spatial Queries. I. Esperança, Claudio. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Dedico à ciência e ao amor, sem os quais minha vida teria menos valor.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

USING WEIGHTED VERTEX COLLECTIONS FOR COMPUTING MAP OVERLAY OPERATIONS

José Augusto Sapienza Ramos

Junho/2014

Orientador: Claudio Esperança

Programa: Engenharia de Sistemas e Computação

Essa dissertação investiga a nova proposta chamada Coleção de Vértices Ponderados ou Weighted Vertex Collection (WVC) para executar sobre dados espaciais de duas dimensões operações de Map Overlay (sobreposição de mapas), que é um tipo de junção espacial. São descritas estruturas de dados e algoritmos baseados no conceito de campo escalar e no paradigma de plano de varredura. Uma implementação é apresentada como prova de conceito, onde os resultados demonstram um tempo de processamento competitivo frente a principais softwares de GIS (Sistemas de Informações Geográficas) do mercado: ArcGIS for Desktop e QGIS. É esperado que com algum avanço, o WVC se apresente uma interessante proposta para processar Map Overlay sobre Spatial Big Data, dados não indexados ou em outros cenários onde novos estudos indiquem potencialidades.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

USING WEIGHTED VERTEX COLLECTIONS FOR COMPUTING MAP OVERLAY OPERATIONS

José Augusto Sapienza Ramos

June/2014

Advisor: Claudio Esperança

Department: Systems Engineering and Computer Science

This dissertation investigates the new proposal called Weighted Vertex Collection (WVC) to perform Map Overlay operations - a type of spatial join - on two-dimensional spatial data, there are described data structures and algorithms based scalar field concept and plane sweep paradigm. An implementation as proof of concept is presented, where the results demonstrate a competitive processing time compared with mainstream GIS (Geographic Information System) software: ArcGIS for Desktop and QGIS. It is expected that with some advance the WVC becomes an interesting alternative to processing of Map Overlay with Big Spatial Data, unindexed data or other scenarios where new studies indicate potentialities.

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgments

First, I wish to thank God who guides us on the path of knowledge. I also want to thank my parents Iracema Sapienza and Antonio Ribeiro Ramos who have created the person I am, likewise I thank my orientator and master Claudio Esperança for your help, patience and for being the role model of a researcher I seek to be. With great affection, I also thank my fiancee Camila Moraes Ribeiro, my uncle Paulo Ramos, mu cousin Eduardo Melo and my whole family, who always supported my studies. I would like to give acknowledgment to every teacher, every professional and every person who gave me a piece of knowledge to compose the professional, researcher and citizen that I am. Lastly, I thank the friends, who made many moments in my life sublime.

Chapter 1

Introduction

In recent years, technologies that work with digital maps have grown in importance. It is noticeable the rapid rate in which new tools, applications and techniques have been developped in recent years. In particular, we may point out the ever-increasing amount of data acquired by remote sensing – both airborne and with the help of artificial satellites including the recent use of unmanned aerial vehicles (UAV) – and by topographic surveys using laser scanners, the rise of new proprietary software and free Geographic Information Systems (GIS) software. In Brazil, it is worth mentioning the renewed efforts dedicated to the systematic cartographic mapping, the beginning of the Spatial Data Infrastructure (SDI) initiative (Infraestrutura Nacional de Dados Espaciais in Portuguese). Also important is the popularization of Global Navigation Satellite Systems (GNSS) such as the American NAVSTAR/GPS and the Russian GLONASS, as well as the advent of interactive maps in websites (GISWeb), among many other examples.

As a result, we are witnessing an enormous increase in the volume of geographic data. Geographic data can be divided into three components [1, 2], namely: *where*, *what* e *when*. Another notation applied to these components is <X;A;T>. The component *where* or X refers to the geographic location of an object or phenomenon, together with a scaled cartographic representation within a geographic or projected coordinate system. The component *what* or A is a non-empty set of attributes that describes a georeferenced object or phenomenon, where usually these attributes are structured in tabular form. Lastly, the component *when* or T is the time reference where the other two components were measured.

According to [1], in the past two decades the database became the core component of information systems from the point of view of project and also of operation. Therefore, when an organization collects a set of spatial data and needs to manage it in a computationally efficient manner, it is necessary to use geographic databases. Apart from the geographic database storing the components <X;A;T>, the Spatial Database Management System (Spatial DBMS) must be able to retrieve and process these components [2]. For example, a Spatial DBMS must be able to answer spatial queries such as: "What are the hospitals contained in the district of Vila Isabel?" or "What are the environmental areas that are within 2 km of the pipelines?". These queries return geometries and their attributes.

Spatial database technologies bring new challenges, which become targets for research in the academic field of computing. Casanova et al. [1] compile proposals to store the component X of geographical data in a database within the relational paradigm. Orenstein [3] describes several techniques for using spatial indexes to perform a filter step in spatial queries, while other works [4, 5] discuss efficient structures for progressive transmission of vector spatial data in the Web. As other examples of important works in the field, we may mention the discussion of redundancy checking in spatial queries [6], cache structures [7] and studies and evaluations for efficient spatial join operations discussed in [8] and compiled in [9].

Increasingly, geographically based solutions generate datasets with a size, complexity and update rate that exceeds the capacity of today's computing technologies [10]. Concepts, proposals and implementations involving large geographic database structures are explored as Spatial Data Infrastructures (see a discussion in [11]) and Spatial Big Data, where the challenges were listed in [10, 12]. Aside from that, papers such as [13] examine GIS requirements in mobile devices with hardware constraints.

In spatial databases two essential classes of operations are required: spatial selection queries and spatial joins. According to [14], the spatial selection query returns a subset of relation tuples using a spatial predicate, such as a window operation, for instance, that aim to determine what geographic features must be shown at a zoom into map. On the other hand, the spatial join is a spatial query which performs a comparison between two database relations - a database cartesian product - according to relationships between their geographic objects, and a predicate, returning attributes and geometries of the relations or even new geometric results obtained, say by operations like intersection, union, buffer, among others. Queries like "What are the hospitals contained in the district of Vila Isabel?" or "Intersect oil pipelines and municipalities boundaries." are examples of spatial joins.

One kind of spatial join operation is the *Map Overlay*. A Map Overlay combines two or more maps of different themes into a single new map [15], where the objective is process spatial relationships between the input maps, usually involving complex conditions. Commonly this operation type is performed on spatial data represented by polygons. Many works such as [12, 15–17] have focused on proposals that can lead to efficient processing of Map Overlay operations.

The objective of this work is to investigate a new proposal to perform Map Overlay operations on two-dimensional spatial data. In particular, it describes data

structures and algorithms based on scalar fields and the plane sweep paradigm. Lastly, an implementation of this proposal has been built and used as a proof of concept, i.e., experiments were conducted where this implementation is compared with mainstream GIS software in the computation of several map overlay operations.

Chapter 2

Spatial Joins

Geographic information systems (GIS) commonly allow queries to be posed by users and analysts. In addition to run-of-the-mill queries supported by other kinds of information systems, a GIS must support spatial queries. For example, in a desktop or web system, interactive navigation of maps require display operations like zoom or pan, which generate a spatial query to determine which features are visible in a given window. See a simple example in Figure 2.1.

Camara [18] lists and defines, within the field of conceptual modeling of geographic databases, what are the possible spatial relationships between classes of georeferenced information. Figure 2.2 illustrates some spatial predicates that a GIS should support in spatial join operations using polygonal geometries. Algorithms to evaluate predicates and other spatial operations on polygons have been intensely investigated within the field of Computational Geometry.

An example of a spatial join query is shown in Figure 2.3, namely, what are the points of relation A that are contained in polygons of relation B? According to [1], to compute a spatial join is more complex than the computation of a non-spatial join - the inherent characteristics of spatial predicates affect traditional strategies to compute joins, thus making the execution inefficient. To compute such operations efficiently it is necessary to consider mainly: (1) the query plan that defines the order for performing joins, selections and others query operations [8]; (2) the processing effort of geometric algorithms - uses of filter steps, for instance; and (3) the computational constraints in scenarios such as Spatial Big Data, mobile devices or Web.

A major component of all management system database is the query processor [1, 19], that receives a query Q with a set of operations as joins, selects and projects and prepares a execution plan $P(Q)$ - also called *query plan*. This plan is a ordered set of steps to compute the result of Q, where is built by an optimizer that incorporates heuristics to limit the search space mainly at the most costly operation - like cartesian product in spatial join. For this, the query processor computes

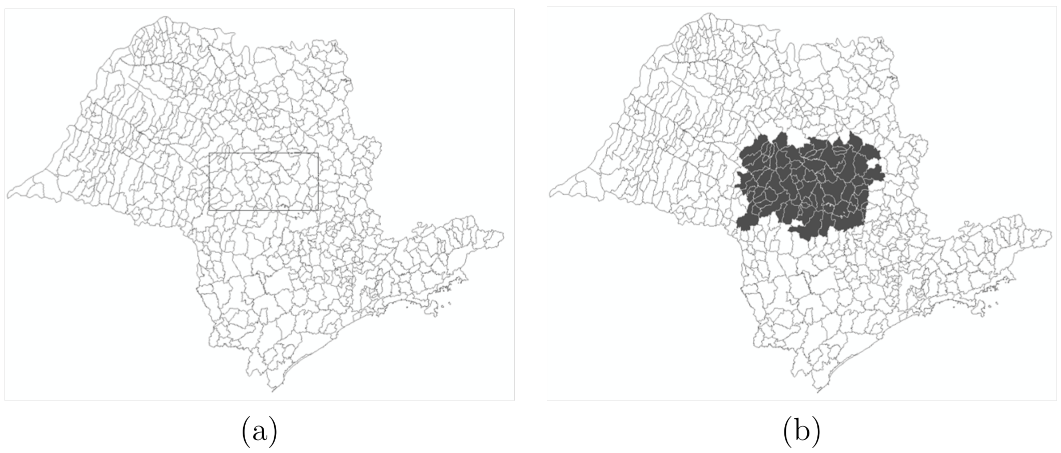


Figure 2.1: Example of a simple spatial query: (a) performing rectangular zoom on a plane partition; and (b) selecting the subset of polygons - filled with dark gray - that must be shown in the window.

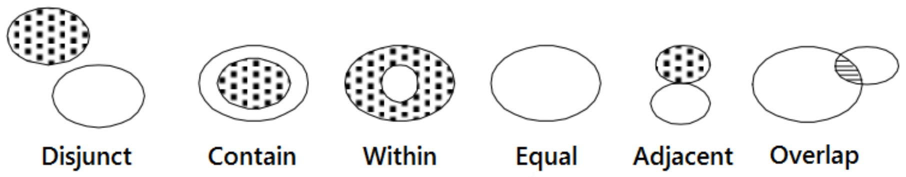


Figure 2.2: Some spatial relationships between polygons. Adapted from [18].

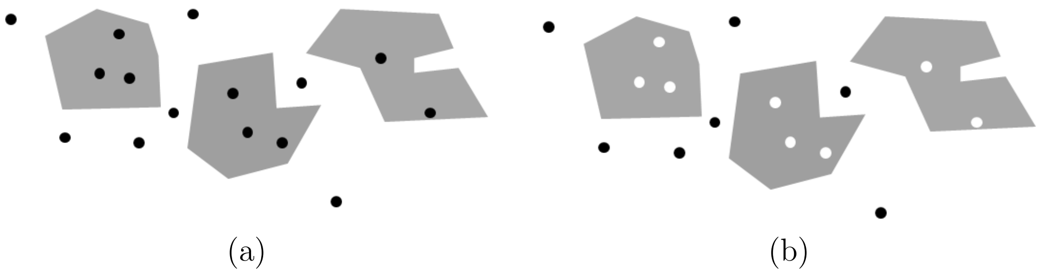


Figure 2.3: Perfoming a spatial join operation with predicate “contains” between two relations, where the query return is marked with white dots in (b).

informations as metadata of relations and statistics of previous queries performed.

In order to exemplify, let us consider the following query Q_1: *select streams that cross road BR-101.* Let S and R be the database relations containing streams and roads, respectively; $A \bowtie_{\times} B$ be the spatial join that returns the join of attributes of A and B when geometries of A cross geometries of B; $\sigma_{br101}(A)$ be the selection that returns tuples of A where NAME='BR-101', and $||A||$ be the number of tuples in A. Then, the query optimizer, could build two plans: $P_1(Q_1) = \sigma_{br101}(R \bowtie_{\times} S)$ and $P_2(Q_1) = S \bowtie_{\times} \sigma_{br101}(R)$. If $||R|| \gg ||\sigma_{br101}(R)||$ or, in other words, the selectivity of $\sigma_{br101}(R)$ is relatively high, then $P_2(Q_1)$ is more efficient that $P_1(Q_1)$, since the most costly operation $\bowtie_{\times}$ is performed with fewer pairs of tuples. Otherwise the efficiency of $P_1(Q_1)$ and $P_2(Q_1)$ are equivalent.

In addition to a good query plan, to process the query operations it is necessary to use appropriate data structures and algorithms to perform the spatial join operation itself in an efficient way.

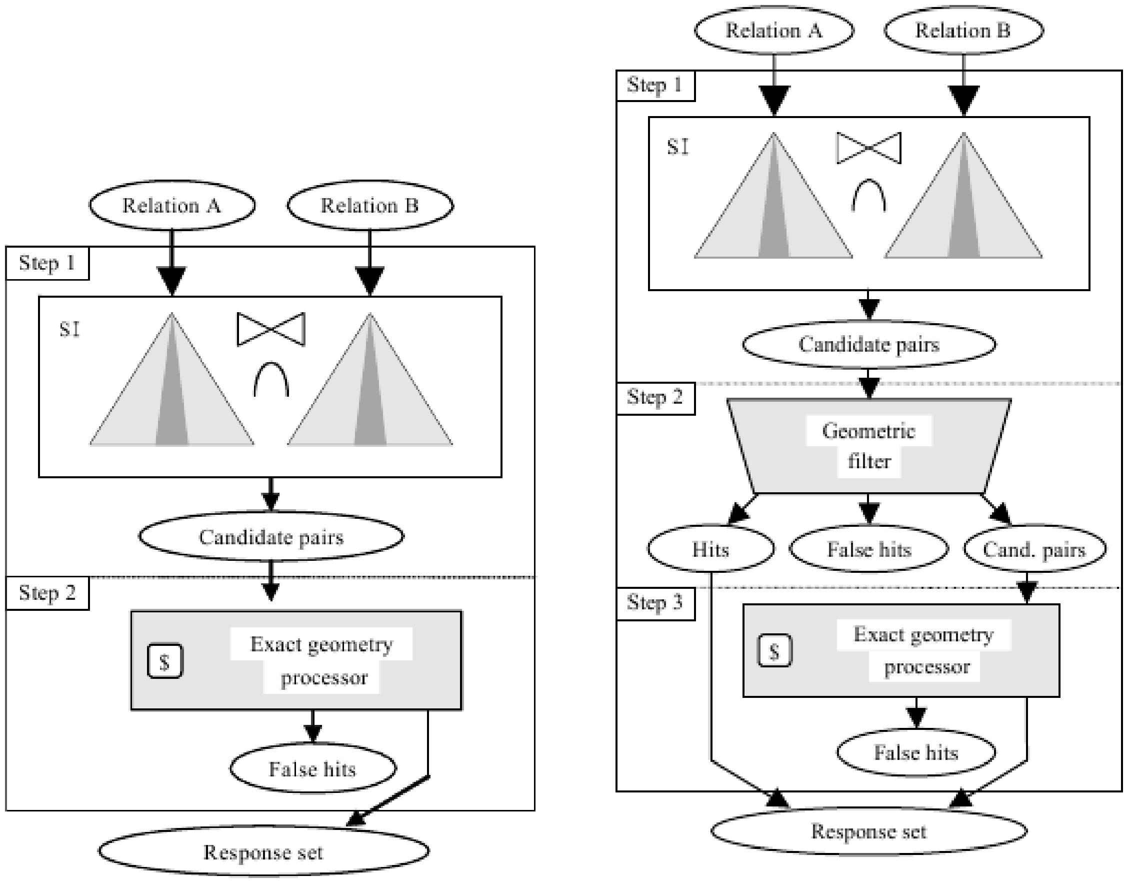
2.1 Spatial join processing strategies

The processing of spatial joins is similar in nature with that of other join types. The major difference lies in that operations on spatial attributes can require an inordinate amount of effort, depending on their complexity. For this reason, special processing strategies have been proposed for spatial data [3, 8, 9, 17, 20].

The most widely adopted strategy for dealing with spatial joins is known as the *filter-and-refine* strategy. It encompasses at least two steps: the filtering step quickly discards some pairs of the cartesian product that do not satisfy the spatial predicate, while the remaining candidate pairs are processed by the refinement step to construct the final response set - see Figure 2.4.a.

The filtering step is usually implemented by a spatial index such as a Quadtree, an R-Tree and their variants - see [21] for a survey of these structures. There are proposals with more than two steps, where the filtering step is subdivided to identify more ocurrencies of false hits prior to processing the exact geometry, including comparisons between simplified geometries such as Minimum Bounding Boxes (MBB). For instance, Kriegel, Brinkhoff and others [16, 19] use spatial indices and MBBs in the filtering step - see Figure 2.4.b - while [20] alternatively uses rasterization techniques. In [1] some strategies for processing spatial joins using two or three steps are exemplified in detail.

The refinement step performs the predicate evaluation on exact geometries. Of particular interest are the algorithms for processing regions, typically represented by polygons. Most algorithms assume polygons encoded as circulations of vertices [8, 12, 22] while others use other encodings that can be handled using the plane



(a) (b)

Figure 2.4: (a) Example of a two-step architecture to efficiently process spatial queries: filtering with spatial index (SI) and refinement with exact geometry; (b) the architeture in three steps adds simplified geometries as second filter and to compose part of the response set. The second architeture is more usual in mainstream databases. Adapted from [16].

sweep paradigm [8, 9, 15, 22]. This second group is more used when there is no filtering stage.

The filtering step requires the use of spatial indexes, simplified geometries or other specific data structures. In some cases - like when building temporary query results with new geometries, or when the filtering step identifies few false hits - the building or maintainance of these structures become more costly than working directly with the exact geometry in the refinement step. Thus, the query processor must consider whether or not to use filters when building spatial join query plans.

As an illustration, let us examine an example query Q_2 - *select streams that cross the stretch of BR-101 road that intersects the Rio de Janeiro district.* Let S, R and D be the database relations containing stream, road and district data, respectively; $A \bowtie_{\times} B$ be the spatial join that returns the join of attributes of relations A and B when geometries of A cross geometries of B; $A \bowtie_{\cap} B$ be the spatial join that returns non-empty geometries $A \cap B$ with the intersection of attributes of A and B; and $\sigma_{br101}(A)$ and $\sigma_{rj}(A)$ respectively as the selections that return tuples of A where NAME='BR-101' and where NAME='Rio de Janeiro'. Then, the query processor may build a plan $P(Q_2) = S \bowtie_{\times} (\sigma_{br101}(R) \bowtie_{\cap} \sigma_{rj}(D))$ to compute Q. Assuming that: (a) the query processor applies a filtering step using R*-Tree [23] as a spatial index (which is common in mainstream spatial databases); (b) MBBs are applied as a second and geometric filter; and (c) the data structures of the R*-Tree and MBBs have already been built for the database relations, then performing $P(Q_2)$ may require that a new R*-Tree and MBB set be built for the new geometries generated in $\bowtie_{\cap}$. Besides, if the selectivity of $\sigma_{br101}(R)$ is high, then the R*-Tree for R may be updated to the result of $\sigma_{br101}(R)$ operation, lest the efficiency of the R*-Tree data structure be compromised.

Spatial joins also play an important role in spatial database modeling. For instance, it is common to store spatial relationships as foreign keys, say, when one wants to link cities to the county they belong to. Thus, a database trigger may be defined for changes in the county borders, so as to update the foreign key link from cities to counties by means of a spatial join.

2.2 Map Overlay

In Geographic Information Systems, the Map Overlay is a crucial operation, since it is responsible for answering questions such as (a) “Where have the vegetation classes changed in the last two years?” (b) “What are the areas of human occupation within the boundaries of protected areas?”; (c) “Overlay slope, land use, vegetation, geology, soil and other layers to generate a map of erosion susceptibility using Fuzzy Logic”. Operations such as intersection, union, clip, dissolve, buffer and other simple geometric operations are usually called “Geoprocessing Tools” in mainstream GIS software. However, other advanced GIS operations such as raster calculations - also called raster overlay - are also considered Map Overlay operations.

The most commonly applied algorithms in Map Overlay process polygonal regions represented as circulations of vertices. These algorithms typically evaluate each pair of polygons in order to find boundary crossings. Therefore, the complexity of these algorithms is related to the amount of vertices. When performing a Map Overlay between many sets of regions, such as the fuzzy logic example mentioned above, we may have to deal with a lot of overlap between the regions. In other words, a set of regions resulting from $R_1 \cap R_2 \cap R_3 \cap ... \cap R_n$ may be produced where the amount of crossing tests may not be significantly reduced by the filtering step.

For these reasons, it is not uncommon a conversion of polygons to a matricial representation in order to increase processing performance. Once the data is stored in matrices, the operation is performed on the overlapping cells. An alternative in this scenario is to work with the plane sweep paradigm, where a two-dimensional problem is solved by breaking it up into several one-dimensional subproblems.

Chapter 3

Weighted Vertex Collections

In this chapter we discuss a data structure named *Weighted Vertex Collection* which will be used to solve the *Map Overlay* problem. The following sections describe the main aspects of the structure, basic operations and algorithms.

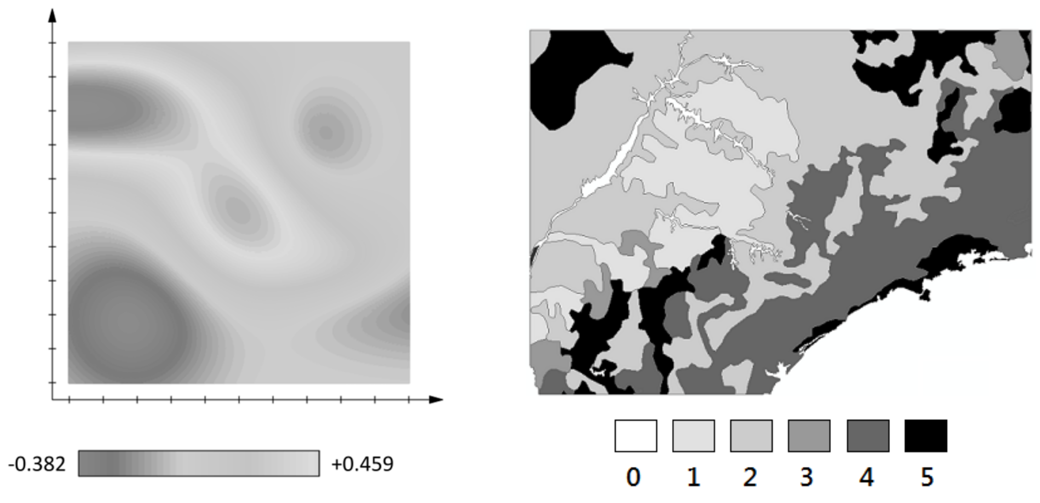
3.1 Weighted Vertex

The proposed data structure structure represents bidimensional discrete scalar fields bounded by polygonal lines, i.e., functions that map $\mathbb{R}^2$ onto $\mathbb{Z}$ where regions that are mapped to the same value are delimited by straight line segments (see Figure 3.1). In essence, the structure is a collection of elements called *weighted vertices.* Each weighted vertex is defined by a position, i.e., a point with coordinates (x, y), a weight w and an angle θ. A weighted vertex adds w to the scalar field for points inside its area of influence – also termed *cone* – which is an infinite sector spanned by two rays intersecting at the vertex position: the first is a vertical ray and the second makes an angle of θ with the x axis (see Figure 3.2).

In order to illustrate the concept, let us model a scalar field called S by means of a collection of weighted vertices C. Initially, the scalar field is null, i.e., all points of the domain are mapped to zero. When a weighted vertex v_1 is added to C, the region corresponding to cone c_1 defined by angle θ_1 is altered by adding $w(v_1)$ to the scalar field, as shown in Figure 3.3.a. Note that S is still null outside c_1, that is, the weight is added to the field at point p only if $p \in c_1$.

By adding a second vertex v_2 to collection C, another cone c_2 is generated, changing S in its area of influence by $w(v_2)$ (see Figure3.3.b). Notice that the value of the scalar field at a given point p is computed by sum of the weights of cones that intersect p. Adding to C a group of vertices with weight $+w$ and another group with weight $-w$, it is poossible to define a limited region mapped to w in S (see Figure 3.3).

The *Weighted Vertex Collection* is closely related to two previously proposed



(a) (b)

Figure 3.1: The 2D points are mapped in (a) continuous and (b) discrete scalar values, where the discrete values in (b) represent a planar subdivision.

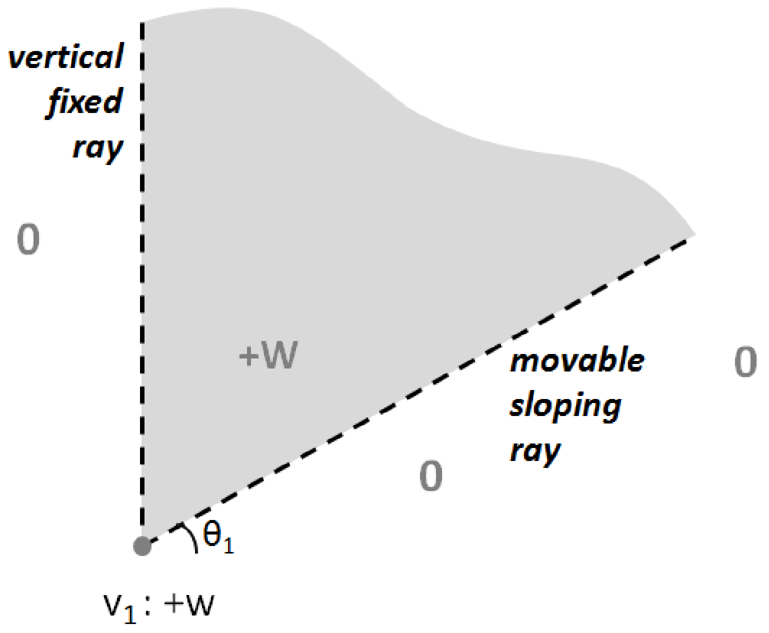


Figure 3.2: The weighted vertex v_1 with $w(v_1) = +w$ and $\theta(v_1) = \theta_1$ generates a cone that perturbs the scalar field with weight $+w$.

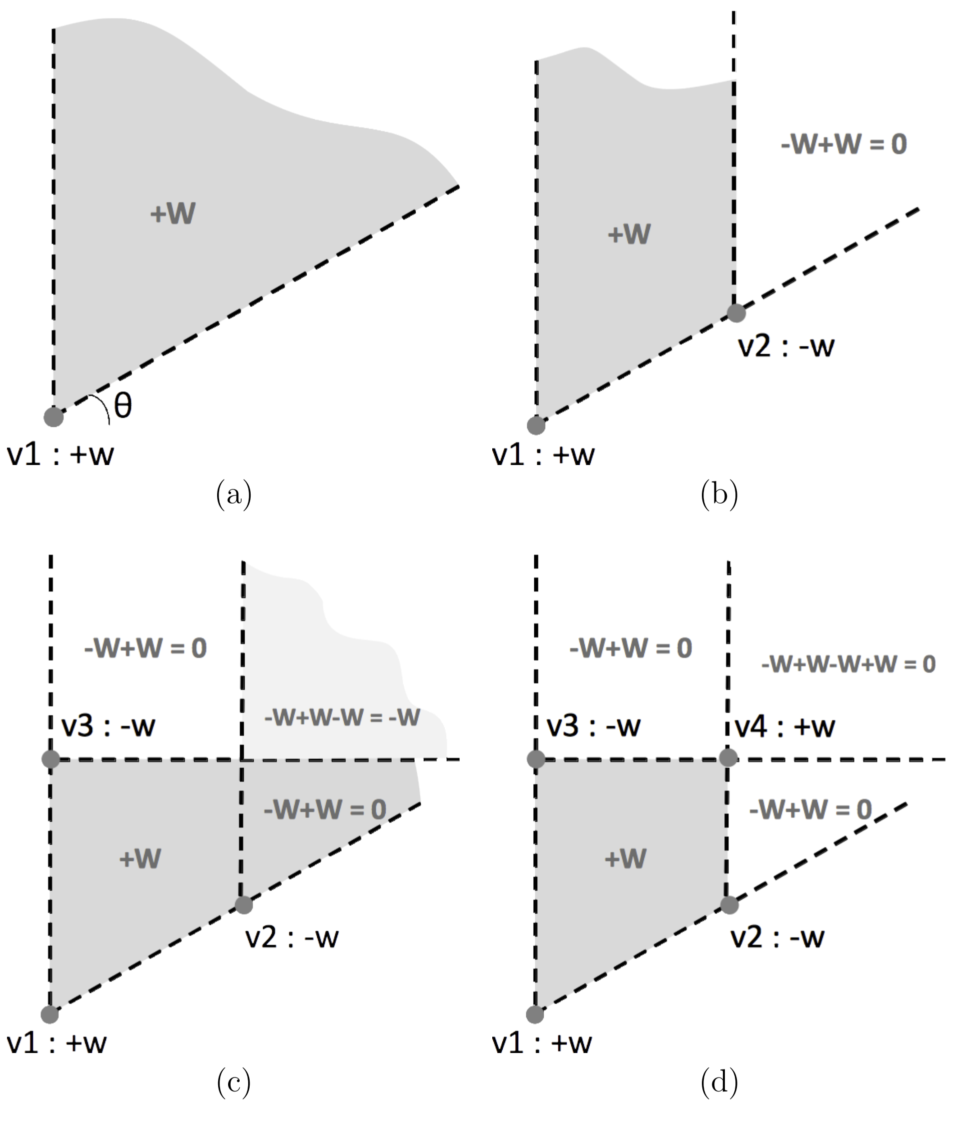


Figure 3.3: (a) The weighted vertex v_1 generates a cone $+w$ with slope θ; (b) another vertex v_2 with the same slope generates a new cone $-w$ which limits the region mapped to w; (c) (d) v_3 and v_4 are added with $\theta = 0$ to determine all sides of the region.

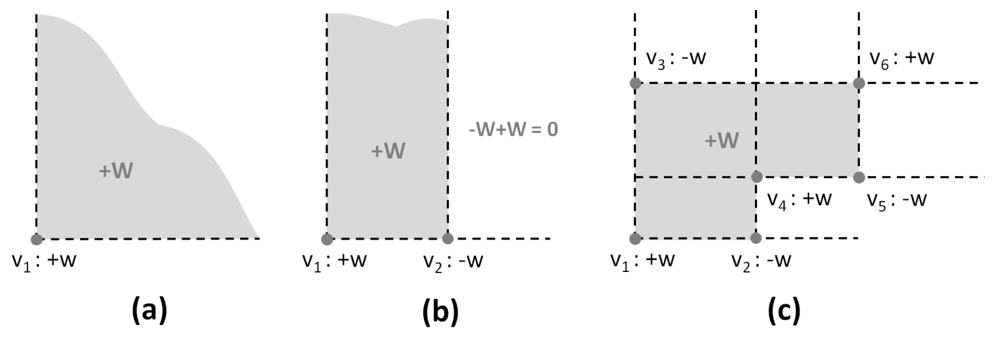


Figure 3.4: *Vertex Representation* example: (a) vertex v_1 is added generating a cone; (b) v_2 is added and limits the first cone; (c) the orthogonal object with weight w is defined by six vertices.

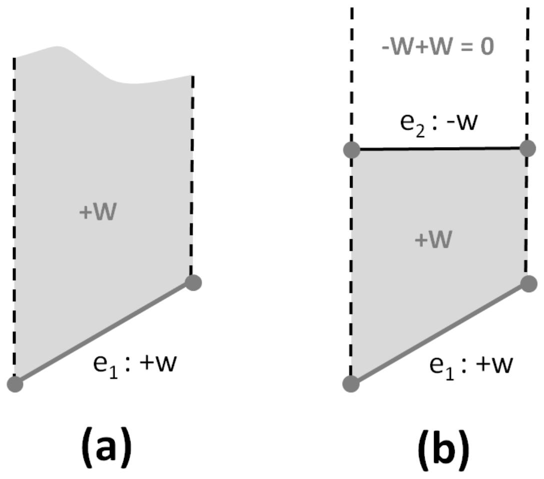


Figure 3.5: *Weighted Edge Collection*, (a) edge e_1 is added generating an infinite trapezoid; (b) e_2 is added defining the top edge of the trapezoid.

approaches for representing discrete scalar fields, namely, the *Vertex Representation* [24] and the *Weighted Edge Collection* [25]. In particular, the *Vertex Representation* is a simplified version of the present scheme where all vertices have $\theta = 0$ and thus are able to model only fields with borders perpendicular to the coordinate axes. On the other hand, in the *Weighted Edge Collection* each weighted edge can be seen as a conjugation of two vertices having the same θ but with symmetric weights, and thus model trapezoidal regions such as the one shown in Figure 3.5.a. Clearly, the *Weighted Vertex Collection* can generate the same trapezoidal regions (compare with Figure 3.3.b). The difference between these proposals is the way they define the primitive elements that change the scalar field, but the *Weighted Vertex Collection* and the *Weighted Edge Collection* have equivalent modeling capability. One of the aims of the present work is to simplify the algorithms and data structures by replacing edges with vertices.

In a canonical representation of a *Weighted Vertex Collection*, the following constraints are imposed: (1) the weight of a vertex can not be zero; (2) two vertices v_1 and v_2 with the same position may be defined only if the $\theta_1 \neq \theta_2$. Otherwise, v_1 and v_2 are replaced by v_3 at the same position, where $w(v_3) = w(v_1) + w(v_2)$ and $\theta_3 = \theta_1 = \theta_2$; and (3) $\theta \neq \pi/2$ and $\theta \neq 3\pi/2$ lest this vertex create a degenerate cone with null area.

The *Vertex Representation*, *Weighted Edge Collection* and *Weighted Vertex Collection* can represent digital images, understood as scalar fields where cells of a regular grid are mapped onto colors. They can also serve as representations of planar subdivisions, where each partition is mapped onto a label or a set of attributes. In the *Object Modeling Technique for Geographic Application (OMT-G)* presented in [26] and discussed in [1], the conceptualization of a representation of the real world through geographic data can be divided two classes of entities: geo-fields and geo-objects. The first represents the group of continuous phenomena in space such as temperature, soil, relief, geology and rainfall. The geo-field class is further divided into planar subdivisions, triangular irregular networks (TIN), tesselations, sampling and isolines. On the other hand, Geo-objects are the group of objects with boundaries defined and individualized as buildings, streets, rivers and crimes. This group is divided into objects "with geometry" such as points, lines and polygons, and those "with geometry and topology": nodes, unidirectional lines and bidirectional lines. Analyzing the classification of the OMT-G, one concludes that the *Weighted Vertex Collection* can be used to represent planar subdivisions (geo-field), tesselations (geo-field) with discrete values, and polygons (geo-object).

3.2 Properties and operations on Weighted Vertex Collections

Let α and β be scalar values, S be a scalar field, C be a weighted vertex collection that represents S, v_1 and v_2 be vertices in C, S_v be the field-induced by vertex v and $p(v)$, $w(v)$ and $\theta(v)$ be the position, weight and θ of v respectively. Then we may recognize the following properties of *Weighted Vertex Collections* (WVCs):

Scalar Multiplication the multiplication of field S by a scalar α, denoted αS, means keeping the v in the same position, and multiplying $w(v)$ by α, $\forall v \in C$.

Vertex Addition the sum of vertices v_1 and v_2 is equivalent to put them in the same representation. The intersection of the areas of influence produced by $v_1 + v_2$ is equal to the sum of their weights: $S_{v_1+v_2} = S_{v_1} + S_{v_2}$. See Figure 3.3.

Uniqueness of representation if a scalar field S can be represented by a canonical WVC C, then no other WVC can represent S.

Boundary property weighted vertices of C are found only on points where scalar field S changes.

In order to process WVCs, we define the following set of operations:

Add The sum of two scalar fields, denoted $S = S_1 + S_2$, is performed by placing the vertices of both collections in the same vertex collection, i.e., $C = C_1 \cup C_2$.

Value at Let P be a set of points p_i and $S(p_i)$ denote the value of S at p_i. Then, the *Value at* operation determines $S(p_i), \forall p_i \in P$.

Draw Creates a picture that represents the values of a scalar field using different colors.

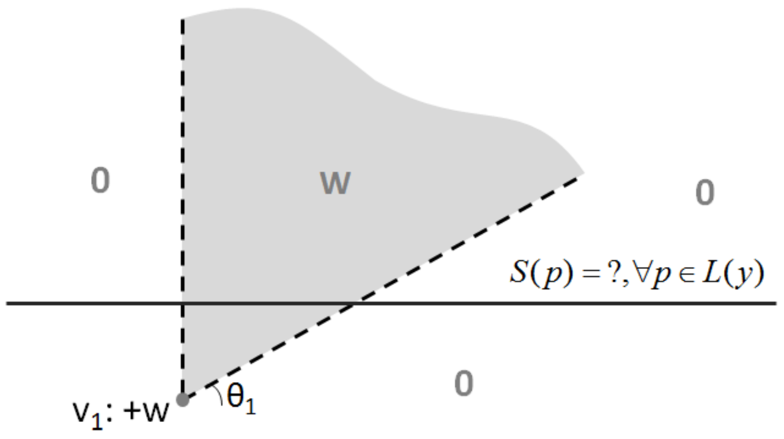
Scalar Transformation Computes a scalar field $S' = f(S)$, where f is a scalar function $\Re \rightarrow \Re$ such that $f(0) = 0$. In other words, this function computes a new scalar field where, if $S(p) = x$, then $S'(p) = f(x)$ for all points p of the plane. Therefore, a new vertex collection is created to represent the transformed scalar field. This operation can be used to compute typical *Map Overlay* operations such as union, intersection and difference.

Convert Converts a collection of circulations of vertices (polygons) into a *Weighted Vertex Collection* and vice-versa.

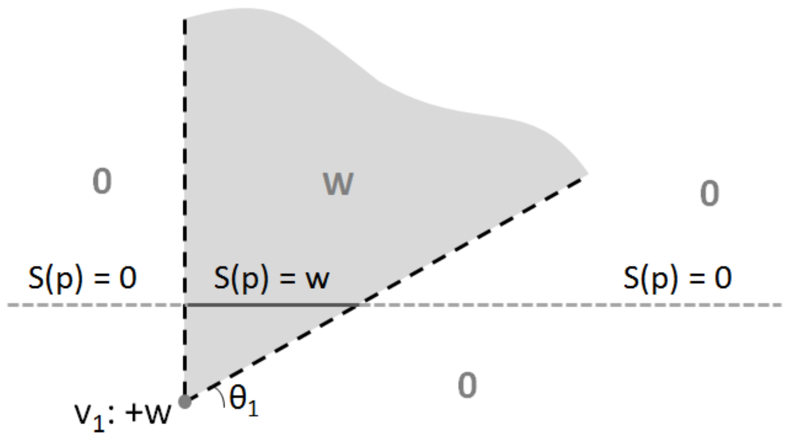
3.3 Scan

Since the WVC represents the changes in a scalar field – but not the field directly – all operations that must evaluate the field are computed with the aid of a so-called *Scan* procedure. A Scan is based on the *plane sweep* [22] or, more specifically, the *line sweep* paradigm, since the problems at hand are two-dimensional. In a nutshell, the line sweep paradigm can be viewed as a form of divide-and-conquer approach, where a two-dimensional problem is solved by breaking it up into several one-dimensional subproblems. This subdivision is accomplished by analyzing points in two-space as a horizontal (or vertical) line is swept along the y (or x) axis.

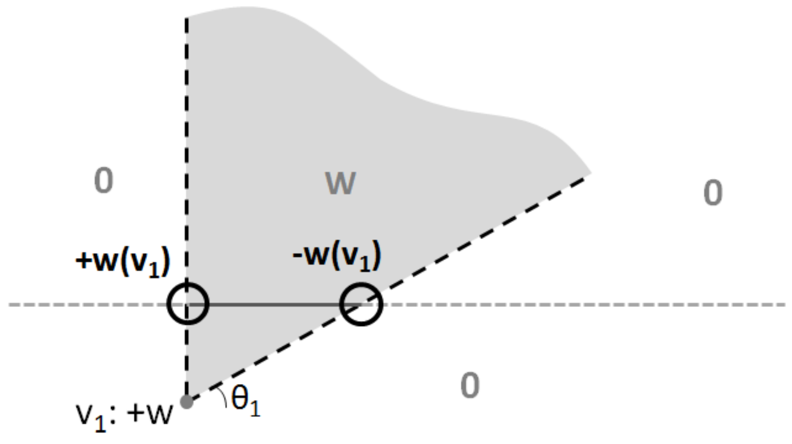
In order to process WVCs, a scan line is swept upwards along the y axis. Notice that the vertices generate areas of influence, or cones, spanning upwards as a result of the constraint $0 \leq \theta < \pi$ and $\theta \neq \pi/2$. To achieve an efficient scan of a scalar field, it is necessary to implement a data structure $L(y)$ that describes the variation of the field along a horizontal line $y = c$, for a constant c (see Figure 3.6). In other words, $L(y)$ records which x coordinates correspond to changes of the scalar field. To accomplish this, the scan line needs to be updated at so-called *stop events*. An



(a)



(b)



(c)

Figure 3.6: Illustrating the relationship between scanlines and cones: (a) vertex v_1 with weight $w(v_1) = +w$ generates a cone crossed by a scanline $L(y)$; (b) to determine the scalar values $S(p_i)$ at points $p_i, \forall p_i \in L(y)$, we need to record the changes above $L(y)$; and (c) changes occur when $L(y)$ crosses the cone rays – these are called *stop events* in the plane sweep paradigm. The ray that crosses $L(y)$ at a smaller x coordinate generates a change of $+w(v_1)$ along the scanline, while the other ray introduces a change of $-w(v_1)$.

event is associated to one or more rays, and can occur (a) when a vertex enters the scan line, thus generating two new *insert* events ev_{in} one for the sloping ray and another for the vertical ray; and (b) when the rays – boundaries of the cones – intersect themselves, generating an *intersect* event ev_{inter}.

3.3.1 Scan order

The Scan procedure requires that vertices are sorted in the order in which they will be encountered by the sweep. The same order is also imposed on the events generated by the vertices. In fact, the events are usually stored in a priority queue E where the priority attached to the event is given by the scan order.

If the vertices of the collection are already in scan order, then the insert events e_{in} are also created in the correct order. The first vertex in the WVC has the either the lowest y coordinate or, in case of equal abscissas, the lowest x coordinate. When two vertices occur at the same position, the highest geometric angle θ is used as a tie breaker.

While events e_{in} are created at the beginning of the procedure, events ev_{inter} are detected during the Scan, as described in the classic *plane sweep* problem. Just as with the vertices, the first scanned event has the either the lowest y coordinate or, in case of equal abscissas, the lowest x coordinate. When two insert events occur at the same position, the highest geometric angle θ of the associated ray is used as a tie breaker. Aside from that, if two intersect events or two events of different types occur at same point, then they may be processed in any order.

3.3.2 Event processing

At the beginning of the Scan, all events of type ev_{in} are added to E and other pertinent structures are created - see Algorithm 3.1. The processing loop starts by popping the next event from E: $ev_{current} = pop(E)$. If E becomes empty, then the processing stops.

The scan strategy employed requires all events at a certain point p to be processed in batch. In other words, to maintain a consistent state of scanline $L(y_n)$ before and after each p, it is not updated at every event, but only when all events occurring at the same coordinate p are known. See Algorithm 3.2.

As shown in the example of Figure 3.6, changes in the scalar field happen when the scan line crosses the rays of a given cone c. It is essential to record the change these rays impose on $L(y)$, since they represent the knowledge at current y of changes on the scalar field at future (higher) y values. These rays are placed in $L(y)$ according to the x coordinate of their intersection with the scan line, termed $cross(r, L(y))$, thus respecting the *scan order* rule. Note, however, that it is not necessary to

Algorithm 3.1 *prepareScan* - creates the initial structures to Scan process

```
Input: $C$ - a WVC collection
Output: $L(y), E$ - returns the scanline structure and event queue
  $E \leftarrow new\ EventList$
  $L(y) \leftarrow new\ ScanLine$
  for each $v \in C$ do
      $r_v, r_s \leftarrow rays(v)$                                   ▷ Getting rays of $v$
      if $\theta(v) < \pi/2$ then                           ▷ Which ray has the higher angle?
          $r_1, r_2 \leftarrow r_v, r_s$              ▷ Insert $e_{in}(r_v)$ first to maintain $E$ in scan order
      else
          $r_1, r_2 \leftarrow r_s, r_v$                              ▷ Else, insert $e_{in}(r_s)$ first
      end if
      $e_{in}(r_1) \leftarrow new\ Insert\ Event(r_1, +w(v))$  ▷ Create event with $ds(r_1) = +w(v)$
      $push(E, e_{in}(r_1))$                                    ▷ Insert new event in $E$
      $e_{in}(r_2) \leftarrow new\ Insert\ Event(r_2, -w(v))$  ▷ Create event with $ds(r_2) = -w(v)$
      $push(E, e_{in}(r_2))$
  end for
  return $L(y), E$
```

Algorithm 3.2 *scan* - scan next point p

```
Input: $L(y), E$
Output: $L(y), E$
  Creates empty lists $Lst_{in}$ and $Lst_{inter}$
  $p_{curr} \leftarrow point(top(E))$          ▷ Gets cordinate value of top element in $E$
  repeat
      $e \leftarrow pop(E)$                                      ▷ Get the next event
      if $e$ is an insert event then
          $push(Lst_{in}, e)$                             ▷ To put event in insert list
      else if $e$ is an intersect event then
          $push(Lst_{inter}, e)$                       ▷ To put event in intersect list
      end if
  until $p_{curr} \neq point(top(E)) \vee empty(E)$ ▷ Gets events until to change point or $E$
  becomes empty.
  $processEvents(L(y), Lst_{in}, Lst_{inter}, E)$      ▷ Update scanline - see Algorithm 3.3
```

explicitly record the coordinate $cross(r, L(y))$ because this point changes as $L(y)$ moves upwards. According to the *plane sweep* paradigm, any need to update $L(y)$ and its list of rays originates from the processing of an event ev_{in} or ev_{inter}.

Let v denote a weighted vertex, $w(v)$ its weight, $c(v)$ the cone representing the region affected by v, $r(v)$ the rays of v – $r_v(v)$ the vertical and $r_s(v)$ the sloping ray, E the priority queue of events, $ev_{in}(r)$ the event for inserting r in the scan line, $ev_{inter}(r_{cr1}, r_{cr2})$ an intersect event between rays $r_{cr1}(v)$ and $r_{cr2}(v)$, $L(y)$ the current status of scan line at y, and $ds(r)$ the value added to the scalar field on the right of $r(v)$ as it crosses $L(y)$. Then, the event processing occurs in the following way:

Processing of insert event $ev_{in}(r)$: Since the scan line has just reached the tip of cone $c(v)$, $r_v(v)$ and $r_s(v)$ are inserted into $L(y)$. The scalar increments $ds(r)$ were set to $+w(v)$ for the first ray and $-w(v)$ for the second ray with respect to the *scan order* rule - see Figure 3.6 and Algorithm 3.1. Procedures *Insert ray in scanline* and *Check new intersections* are performed on inserted rays. See the first loop of Algorithm 3.3.

Processing of intersection event $ev_{inter}(r_{cr1}, r_{cr2})$: Since $r_{cr1}(v)$ and $r_{cr2}(v)$ are already in $L(y)$, their order must be switched - this is done by removing and reinserting the rays. The new neighbors of the two rays in $L(y)$ need to be examined by executing *Check overlapping rays*. See the last loop of Algorithm 3.3.

Check overlapping rays: Before a ray $r(v)$ is added or relocated to $L(y)$, it is checked if $r(v)$ coincides with another ray (r_{ovlap}) already in $L(y)$. If so, $r(v)$ is not inserted or is removed from $L(y)$ and r_{ovlap} has $ds(r_{ovlap})$ updated to $ds(r_{ovlap}) + ds(r)$. If the resulting $ds(r_{ovlap})$ is zero, then r_{ovlap} is removed too. See the Algorithm 3.4.

Check new intersections When a ray $r(v)$ is inserted, relocated or removed from $L(y)$, it is necessary to check if new neighbors in $L(y)$ cross themselves later in the scan process. Let i be the position (i.e., index) of $r(v)$ in the list of rays of $L(y)$, then the two neighboring rays are in positions $i - 1$ and $i + 1$ when $r(v)$ is inserted or relocated. When $r(v)$ is removed from $L(y)$, then its two former neighbors are checked for intercection with each other. If the intersection occurs, a new event ev_{inter} on the intersection point is added to E. See Algorithm 3.5.

The Algorithms 3.2, 3.3, 3.4 and 3.5 are a straightforward pseudocode to highlight the main points of Scan process. Figures 3.7, 3.8, 3.9, 3.10 and 3.11 show an example of the scan process for an isosceles triangle. Some important details of the process are listed below:

1. The events are processed in batch for each point p.

2. In some steps, such as depicted in Figure 3.7.f and Figure 3.8.f, the rays cross themselves, however an intersection event is added in E only when the cross point occurs later in scan process or, in other words, when the cross point has a higher y than the current y as in Figure 3.8.e.

3. In Figure 3.10.e an intersection event is processed, thus the positions of $r_s(v_2)$ and $r_v(r_1)$ were switched in $L(y)$.

4. Events can ocur at the same point like, for instance, in Figure 3.8.c, where four insert events are processed following the scan order.

5. Only the events of highest priority in E are shown at each moment in the figures. The complete list E at the beginning of the scan is: $ev_{in}(r_v(v_1)), ev_{in}(r_s(v_1)), ev_{in}(r_s(v_2)), ev_{in}(r_v(v_2)), ev_{in}(r_v(v_3)), ev_{in}(r_s(v_3)), ev_{in}(r_s(v_4))$ and $ev_{in}(r_v(v_4))$.

6. As discussed previously, the intersection events are discovered during the scan process. For example, in Figure 3.8.e event $e_{inter}(r_v(v_1), r_s(v_2))$ is detected and added to E.

7. The algorithm explanations presented here do not consider horizontal rays, that is, the sloping rays generated by weighted vertices where $\theta = 0$. Horizontal rays are processed in one update of $L(y)$, therefore they are never inserted or removed from the scanline. The algorithms can be adjusted to include a new event type $ev_{hline}(r)$, where the sweep must detect and process the intersection of a horizontal ray with other rays.

Algorithm 3.3 *processEvents* - process events to update $L(y)$

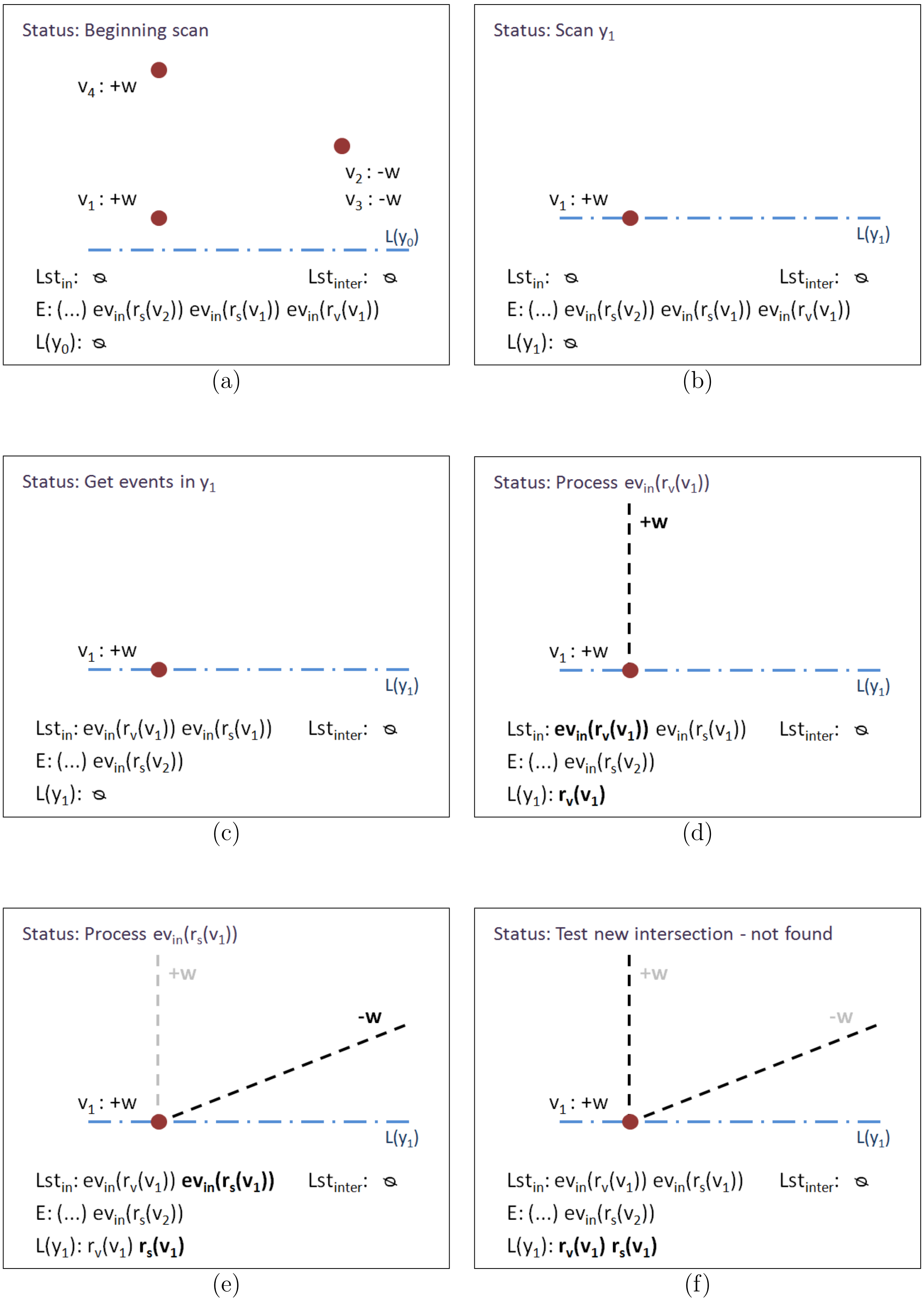
```
Input: $L(y), Lst_{in}, Lst_{inter}, E$
Output: none
  for each $e_{in} \in Lst_{in}$ do
      $r \leftarrow ray(e_{in})$                          ▷ Ray associated to event
      $insertRay(L(y), r, E)$              ▷ Insert ray in scanline - see Algorithm 3.4
  end for
  for each $e_{inter} \in Lst_{inter}$ do
      $r_{cr1}, r_{cr2} \leftarrow rays(e_{inter})$                ▷ Rays associated to event
      $deleteRay(L(y), r_{cr1})$
      $deleteRay(L(y), r_{cr2})$                    ▷ Remove rays of scanline ...
      $insertRay(L(y), r_{cr1}, E)$                   ▷ ... and reinsert to reorder
      $insertRay(L(y), r_{cr2}, E)$                          ▷ See Algorithm 3.4
  end for
```

Algorithm 3.4 *insertRay* - insert ray in scanline

```
Input: $L(y), r, E$
Output: $none$
  if $overlap(L(y), r)$ then                          ▷ Does $r$ overlap another ray in the scanline?
      $i, r_{ovlap} \leftarrow getOverlap(L(y), r)$            ▷ Obtain overlapped ray and its index
      $ds(r_{ovlap}) \leftarrow ds(r_{ovlap}) + ds(r)$           ▷ Update value added to scalar field
      if $ds(r_{ovlap}) = 0$ then                                   ▷ Zero field increment?
          $checkInter(L(y), i-1, i+1)$            ▷ Test new neighbors - see Algorithm 3.5
          $delete(L(y), r_{ovlap})$                            ▷ Remove ray from scanline
      end if
  else                                         ▷ Rays do not overlap: insert $r$ in scanline
      $i \leftarrow insert(L(y), r)$                 ▷ Insert ray, obtaining the insertion index $i$
      $checkInter(L(y), i, i+1, E)$               ▷ Test new neighbors - see Algorithm 3.5
      $checkInter(L(y), i, i-1, E)$
  end if
```

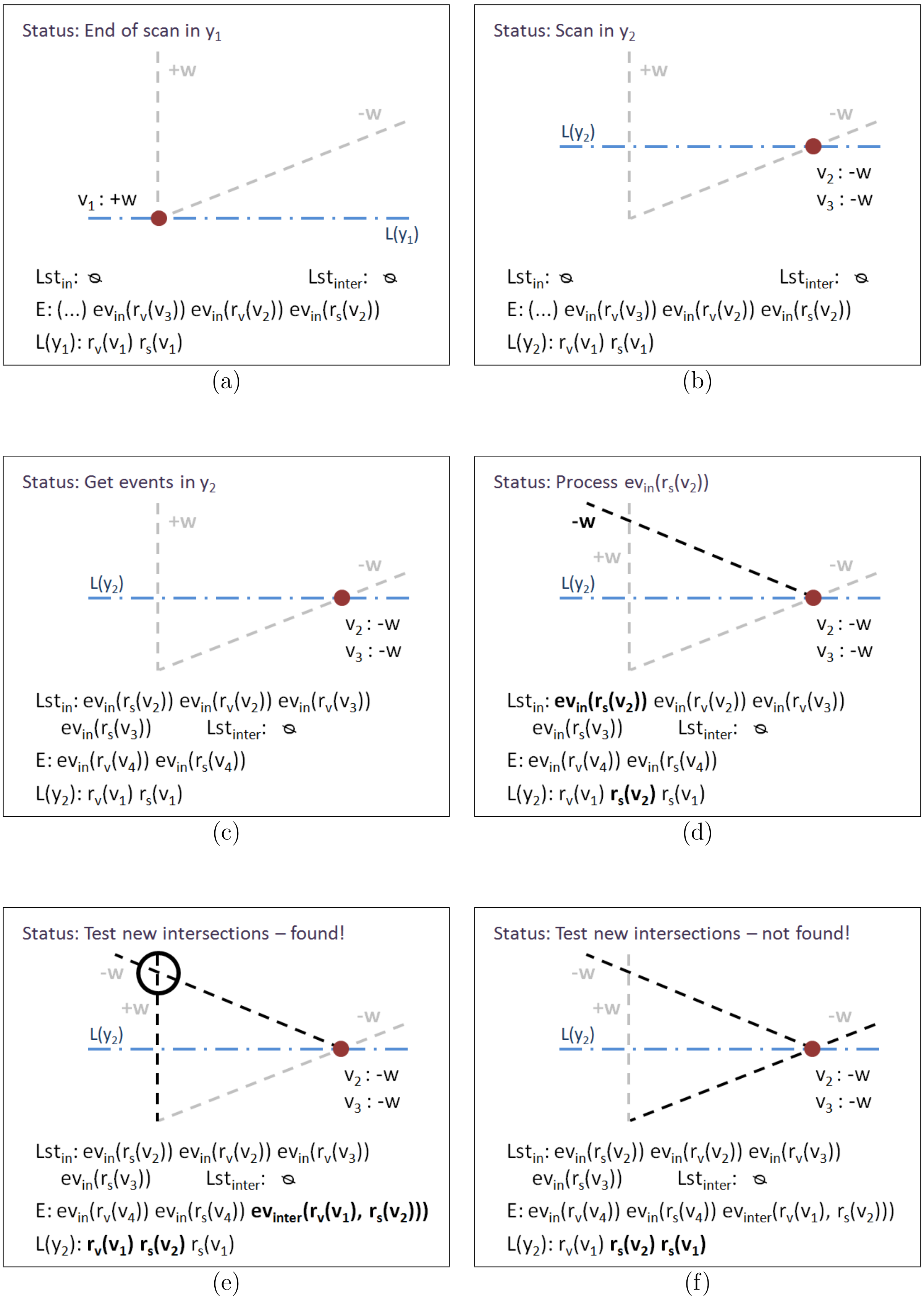
Algorithm 3.5 *testInter* - test scanline rays for intersection

```
Input: $L(y), i_1, i_2, E$
Output: none
  $r_1 \leftarrow getAt(L(y), i_1)$                            ▷ Get rays at the given indices
  $r_2 \leftarrow getAt(L(y), i_2)$
  if $cross(r_1, r_2)$ then                                           ▷ Crossing rays?
      $p_{cross} \leftarrow crossPoint(r_1, r_2)$                ▷ Compute the intersection point
      $e_{inter} \leftarrow new\ Intersect\ Eevent(r_1, r_2, p_{cross})$        ▷ Create intersection event
      $insert(E, e_{inter})$                                  ▷ Add event to event queue
  end if
```



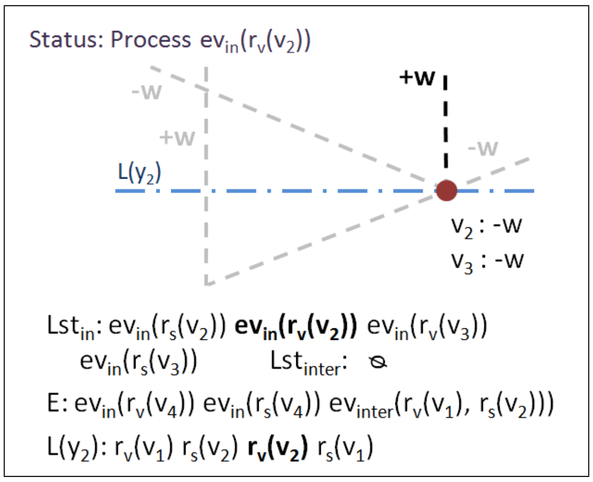
Status: Beginning scan
v_4 : +w
v_2 : -w
v_3 : -w
v_1 : +w
$L(y_0)$
Lst_{in}: ∅
Lst_{inter}: ∅
E: (...) $ev_{in}(r_s(v_2))$ $ev_{in}(r_s(v_1))$ $ev_{in}(r_v(v_1))$
$L(y_0)$: ∅
(a)
Status: Scan y_1
v_1 : +w
$L(y_1)$
Lst_{in}: ∅
Lst_{inter}: ∅
E: (...) $ev_{in}(r_s(v_2))$ $ev_{in}(r_s(v_1))$ $ev_{in}(r_v(v_1))$
$L(y_1)$: ∅
(b)
Status: Get events in y_1
v_1 : +w
$L(y_1)$
Lst_{in}: $ev_{in}(r_v(v_1))$ $ev_{in}(r_s(v_1))$
Lst_{inter}: ∅
E: (...) $ev_{in}(r_s(v_2))$
$L(y_1)$: ∅
(c)
Status: Process $ev_{in}(r_v(v_1))$
+w
v_1 : +w
$L(y_1)$
Lst_{in}: $\mathbf{ev_{in}(r_v(v_1))}$ $ev_{in}(r_s(v_1))$
Lst_{inter}: ∅
E: (...) $ev_{in}(r_s(v_2))$
$L(y_1)$: $\mathbf{r_v(v_1)}$
(d)
Status: Process $ev_{in}(r_s(v_1))$
+w
-w
v_1 : +w
$L(y_1)$
Lst_{in}: $ev_{in}(r_v(v_1))$ $\mathbf{ev_{in}(r_s(v_1))}$
Lst_{inter}: ∅
E: (...) $ev_{in}(r_s(v_2))$
$L(y_1)$: $r_v(v_1)$ $\mathbf{r_s(v_1)}$
(e)
Status: Test new intersection - not found
+w
-w
v_1 : +w
$L(y_1)$
Lst_{in}: $ev_{in}(r_v(v_1))$ $ev_{in}(r_s(v_1))$
Lst_{inter}: ∅
E: (...) $ev_{in}(r_s(v_2))$
$L(y_1)$: $\mathbf{r_v(v_1)}$ $\mathbf{r_s(v_1)}$
(f)

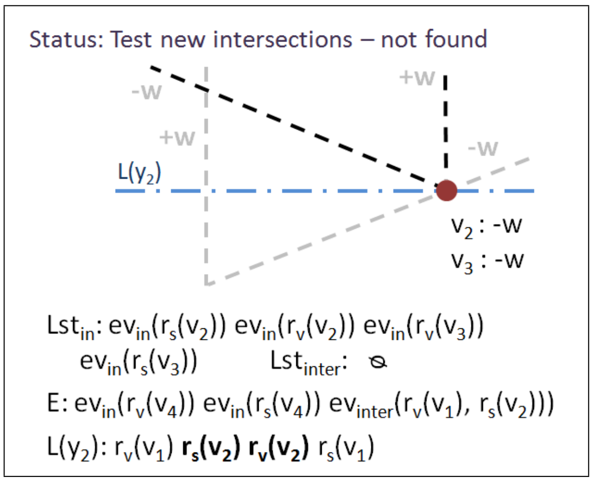
Figure 3.7: Step by step scan of an isosceles triangle - part 1 of 5.

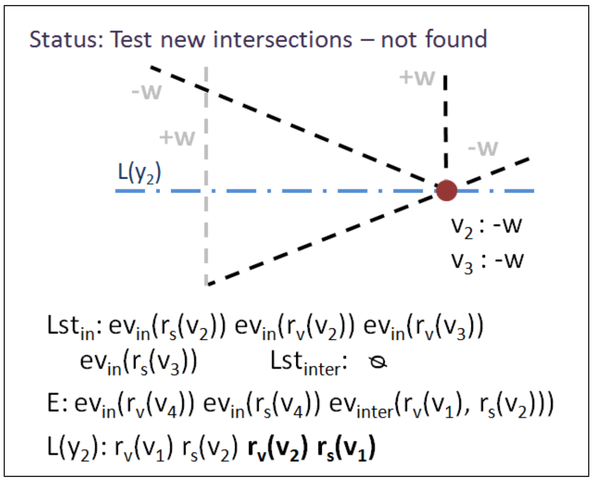


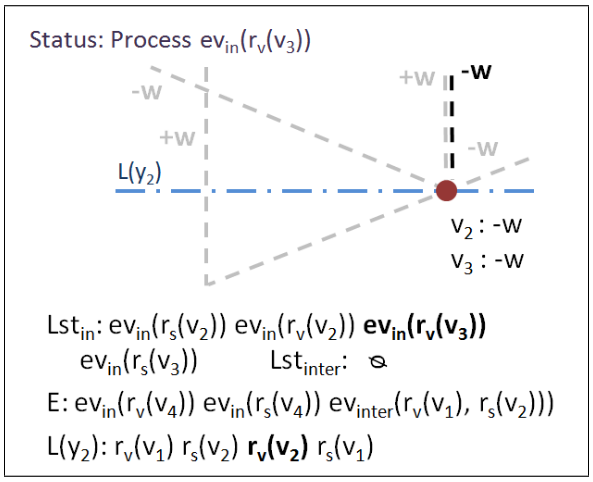
Status: End of scan in y_1
+w
-w
v_1 : +w
$L(y_1)$
Lst_{in}: ∅
Lst_{inter}: ∅
E: (...) $ev_{in}(r_v(v_3))$ $ev_{in}(r_v(v_2))$ $ev_{in}(r_s(v_2))$
$L(y_1)$: $r_v(v_1)$ $r_s(v_1)$
(a)
Status: Scan in y_2
+w
-w
$L(y_2)$
v_2 : -w
v_3 : -w
Lst_{in}: ∅
Lst_{inter}: ∅
E: (...) $ev_{in}(r_v(v_3))$ $ev_{in}(r_v(v_2))$ $ev_{in}(r_s(v_2))$
$L(y_2)$: $r_v(v_1)$ $r_s(v_1)$
(b)
Status: Get events in y_2
+w
-w
$L(y_2)$
v_2 : -w
v_3 : -w
Lst_{in}: $ev_{in}(r_s(v_2))$ $ev_{in}(r_v(v_2))$ $ev_{in}(r_v(v_3))$ $ev_{in}(r_s(v_3))$
Lst_{inter}: ∅
E: $ev_{in}(r_v(v_4))$ $ev_{in}(r_s(v_4))$
$L(y_2)$: $r_v(v_1)$ $r_s(v_1)$
(c)
Status: Process $ev_{in}(r_s(v_2))$
-w
+w
-w
$L(y_2)$
v_2 : -w
v_3 : -w
Lst_{in}: $\mathbf{ev_{in}(r_s(v_2))}$ $ev_{in}(r_v(v_2))$ $ev_{in}(r_v(v_3))$ $ev_{in}(r_s(v_3))$
Lst_{inter}: ∅
E: $ev_{in}(r_v(v_4))$ $ev_{in}(r_s(v_4))$
$L(y_2)$: $r_v(v_1)$ $\mathbf{r_s(v_2)}$ $r_s(v_1)$
(d)
Status: Test new intersections – found!
-w
+w
-w
$L(y_2)$
v_2 : -w
v_3 : -w
Lst_{in}: $ev_{in}(r_s(v_2))$ $ev_{in}(r_v(v_2))$ $ev_{in}(r_v(v_3))$ $ev_{in}(r_s(v_3))$
Lst_{inter}: ∅
E: $ev_{in}(r_v(v_4))$ $ev_{in}(r_s(v_4))$ $\mathbf{ev_{inter}(r_v(v_1), r_s(v_2))}$
$L(y_2)$: $\mathbf{r_v(v_1)}$ $\mathbf{r_s(v_2)}$ $r_s(v_1)$
(e)
Status: Test new intersections – not found!
-w
+w
-w
$L(y_2)$
v_2 : -w
v_3 : -w
Lst_{in}: $ev_{in}(r_s(v_2))$ $ev_{in}(r_v(v_2))$ $ev_{in}(r_v(v_3))$ $ev_{in}(r_s(v_3))$
Lst_{inter}: ∅
E: $ev_{in}(r_v(v_4))$ $ev_{in}(r_s(v_4))$ $ev_{inter}(r_v(v_1), r_s(v_2))$
$L(y_2)$: $r_v(v_1)$ $\mathbf{r_s(v_2)}$ $\mathbf{r_s(v_1)}$
(f)

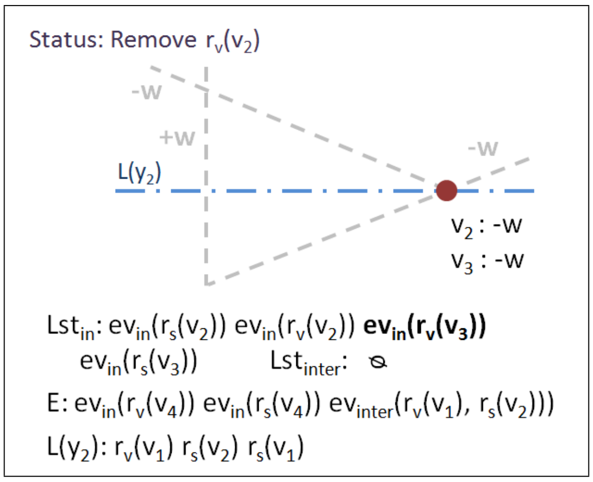
Figure 3.8: Step by step scan of an isosceles triangle - part 2 of 5.


Status: Process $ev_{in}(r_v(v_2))$
-w
+w
+w
-w
$L(y_2)$
v_2 : -w
v_3 : -w
Lst_{in}: $ev_{in}(r_s(v_2))$ $\mathbf{ev_{in}(r_v(v_2))}$ $ev_{in}(r_v(v_3))$ $ev_{in}(r_s(v_3))$
Lst_{inter}: ∅
E: $ev_{in}(r_v(v_4))$ $ev_{in}(r_s(v_4))$ $ev_{inter}(r_v(v_1), r_s(v_2)))$
$L(y_2)$: $r_v(v_1)$ $r_s(v_2)$ $\mathbf{r_v(v_2)}$ $r_s(v_1)$
(a)


Status: Test new intersections – not found
-w
+w
+w
-w
$L(y_2)$
v_2 : -w
v_3 : -w
Lst_{in}: $ev_{in}(r_s(v_2))$ $ev_{in}(r_v(v_2))$ $ev_{in}(r_v(v_3))$ $ev_{in}(r_s(v_3))$
Lst_{inter}: ∅
E: $ev_{in}(r_v(v_4))$ $ev_{in}(r_s(v_4))$ $ev_{inter}(r_v(v_1), r_s(v_2)))$
$L(y_2)$: $r_v(v_1)$ $\mathbf{r_s(v_2)}$ $\mathbf{r_v(v_2)}$ $r_s(v_1)$
(b)


Status: Test new intersections – not found
-w
+w
+w
-w
$L(y_2)$
v_2 : -w
v_3 : -w
Lst_{in}: $ev_{in}(r_s(v_2))$ $ev_{in}(r_v(v_2))$ $ev_{in}(r_v(v_3))$ $ev_{in}(r_s(v_3))$
Lst_{inter}: ∅
E: $ev_{in}(r_v(v_4))$ $ev_{in}(r_s(v_4))$ $ev_{inter}(r_v(v_1), r_s(v_2)))$
$L(y_2)$: $r_v(v_1)$ $r_s(v_2)$ $\mathbf{r_v(v_2)}$ $\mathbf{r_s(v_1)}$
(c)


Status: Process $ev_{in}(r_v(v_3))$
-w
+w
+w
-w
-w
$L(y_2)$
v_2 : -w
v_3 : -w
Lst_{in}: $ev_{in}(r_s(v_2))$ $ev_{in}(r_v(v_2))$ $\mathbf{ev_{in}(r_v(v_3))}$ $ev_{in}(r_s(v_3))$
Lst_{inter}: ∅
E: $ev_{in}(r_v(v_4))$ $ev_{in}(r_s(v_4))$ $ev_{inter}(r_v(v_1), r_s(v_2)))$
$L(y_2)$: $r_v(v_1)$ $r_s(v_2)$ $\mathbf{r_v(v_2)}$ $r_s(v_1)$
(d)


Status: Remove $r_v(v_2)$
-w
+w
-w
$L(y_2)$
v_2 : -w
v_3 : -w
Lst_{in}: $ev_{in}(r_s(v_2))$ $ev_{in}(r_v(v_2))$ $\mathbf{ev_{in}(r_v(v_3))}$ $ev_{in}(r_s(v_3))$
Lst_{inter}: ∅
E: $ev_{in}(r_v(v_4))$ $ev_{in}(r_s(v_4))$ $ev_{inter}(r_v(v_1), r_s(v_2)))$
$L(y_2)$: $r_v(v_1)$ $r_s(v_2)$ $r_s(v_1)$
(e)

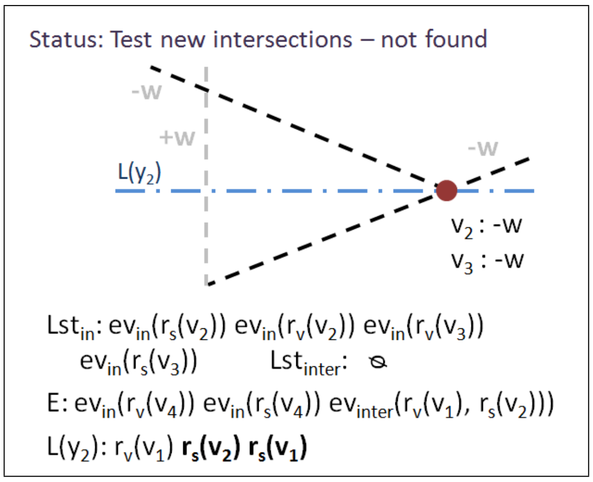

Status: Test new intersections – not found
-w
+w
-w
$L(y_2)$
v_2 : -w
v_3 : -w
Lst_{in}: $ev_{in}(r_s(v_2))$ $ev_{in}(r_v(v_2))$ $ev_{in}(r_v(v_3))$ $ev_{in}(r_s(v_3))$
Lst_{inter}: ∅
E: $ev_{in}(r_v(v_4))$ $ev_{in}(r_s(v_4))$ $ev_{inter}(r_v(v_1), r_s(v_2)))$
$L(y_2)$: $r_v(v_1)$ $\mathbf{r_s(v_2)}$ $\mathbf{r_s(v_1)}$
(f)

Figure 3.9: Step by step scan of an isosceles triangle - part 3 of 5.

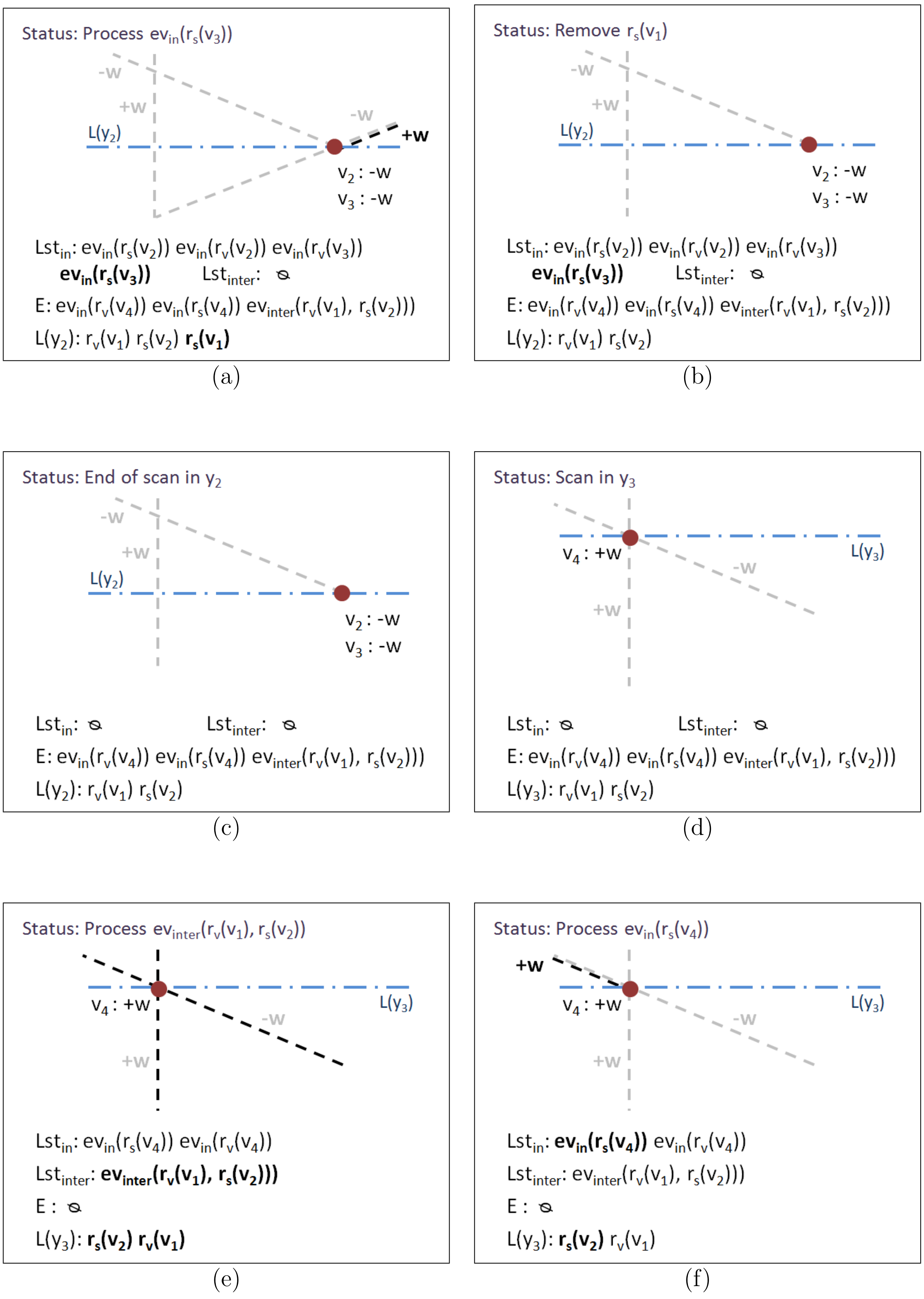

Status: Process $ev_{in}(r_s(v_3))$
-w
+w
$L(y_2)$
-w
+w
v_2 : -w
v_3 : -w
Lst_{in}: $ev_{in}(r_s(v_2))$ $ev_{in}(r_v(v_2))$ $ev_{in}(r_v(v_3))$ $\mathbf{ev_{in}(r_s(v_3))}$ Lst_{inter}: $\emptyset$
E: $ev_{in}(r_v(v_4))$ $ev_{in}(r_s(v_4))$ $ev_{inter}(r_v(v_1), r_s(v_2)))$
$L(y_2)$: $r_v(v_1)$ $r_s(v_2)$ $\mathbf{r_s(v_1)}$
(a)
Status: Remove $r_s(v_1)$
-w
+w
$L(y_2)$
v_2 : -w
v_3 : -w
Lst_{in}: $ev_{in}(r_s(v_2))$ $ev_{in}(r_v(v_2))$ $ev_{in}(r_v(v_3))$ $\mathbf{ev_{in}(r_s(v_3))}$ Lst_{inter}: $\emptyset$
E: $ev_{in}(r_v(v_4))$ $ev_{in}(r_s(v_4))$ $ev_{inter}(r_v(v_1), r_s(v_2)))$
$L(y_2)$: $r_v(v_1)$ $r_s(v_2)$
(b)
Status: End of scan in y_2
-w
+w
$L(y_2)$
v_2 : -w
v_3 : -w
Lst_{in}: $\emptyset$ Lst_{inter}: $\emptyset$
E: $ev_{in}(r_v(v_4))$ $ev_{in}(r_s(v_4))$ $ev_{inter}(r_v(v_1), r_s(v_2)))$
$L(y_2)$: $r_v(v_1)$ $r_s(v_2)$
(c)
Status: Scan in y_3
v_4 : +w
$L(y_3)$
-w
+w
Lst_{in}: $\emptyset$ Lst_{inter}: $\emptyset$
E: $ev_{in}(r_v(v_4))$ $ev_{in}(r_s(v_4))$ $ev_{inter}(r_v(v_1), r_s(v_2)))$
$L(y_3)$: $r_v(v_1)$ $r_s(v_2)$
(d)
Status: Process $ev_{inter}(r_v(v_1), r_s(v_2))$
v_4 : +w
$L(y_3)$
-w
+w
Lst_{in}: $ev_{in}(r_s(v_4))$ $ev_{in}(r_v(v_4))$
Lst_{inter}: $\mathbf{ev_{inter}(r_v(v_1), r_s(v_2)))}$
E : $\emptyset$
$L(y_3)$: $\mathbf{r_s(v_2)}$ $\mathbf{r_v(v_1)}$
(e)
Status: Process $ev_{in}(r_s(v_4))$
+w
v_4 : +w
$L(y_3)$
-w
+w
Lst_{in}: $\mathbf{ev_{in}(r_s(v_4))}$ $ev_{in}(r_v(v_4))$
Lst_{inter}: $ev_{inter}(r_v(v_1), r_s(v_2)))$
E : $\emptyset$
$L(y_3)$: $\mathbf{r_s(v_2)}$ $r_v(v_1)$
(f)

Figure 3.10: Step by step scan of an isosceles triangle - part 4 of 5.

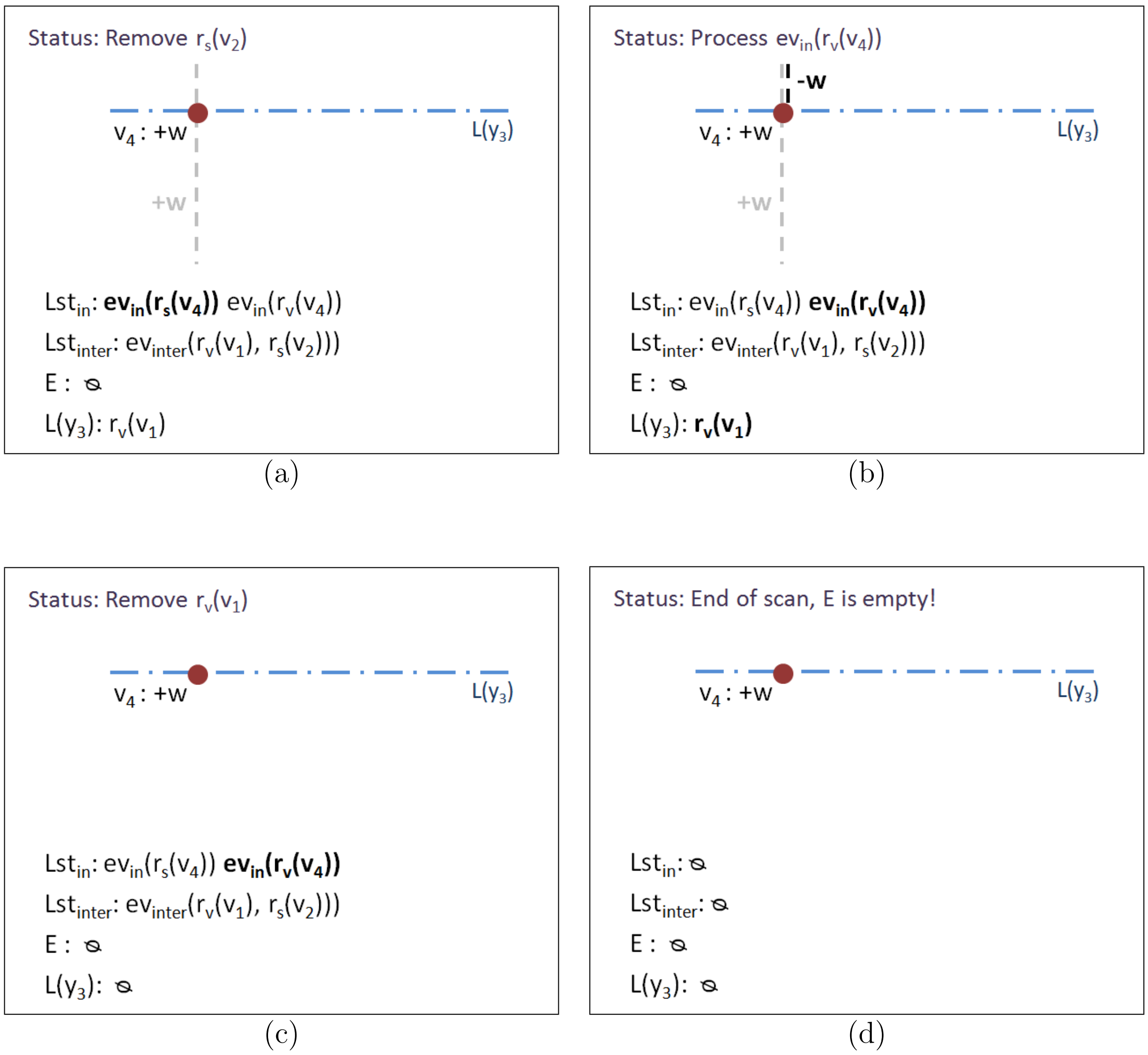

Status: Remove $r_s(v_2)$
v_4 : +w
$L(y_3)$
+w
Lst$_{in}$: **ev$_{in}$(r$_s$(v$_4$))** ev$_{in}$(r$_v$(v$_4$))
Lst$_{inter}$: ev$_{inter}$(r$_v$(v$_1$), r$_s$(v$_2$)))
E : ∅
$L(y_3)$: $r_v(v_1)$
(a)
Status: Process ev$_{in}$(r$_v$(v$_4$))
-w
v_4 : +w
$L(y_3)$
+w
Lst$_{in}$: ev$_{in}$(r$_s$(v$_4$)) **ev$_{in}$(r$_v$(v$_4$))**
Lst$_{inter}$: ev$_{inter}$(r$_v$(v$_1$), r$_s$(v$_2$)))
E : ∅
$L(y_3)$: **$r_v(v_1)$**
(b)
Status: Remove $r_v(v_1)$
v_4 : +w
$L(y_3)$
Lst$_{in}$: ev$_{in}$(r$_s$(v$_4$)) **ev$_{in}$(r$_v$(v$_4$))**
Lst$_{inter}$: ev$_{inter}$(r$_v$(v$_1$), r$_s$(v$_2$)))
E : ∅
$L(y_3)$: ∅
(c)
Status: End of scan, E is empty!
v_4 : +w
$L(y_3)$
Lst$_{in}$: ∅
Lst$_{inter}$: ∅
E : ∅
$L(y_3)$: ∅
(d)

Figure 3.11: Step by step scan of an isosceles triangle - part 5 of 5.

3.4 Operations

3.4.1 Sum

Consider two weighted vertex collections C_1 and C_2 defining scalar fields S_1 and S_2 respectively. To create a new field $S = S_1 + S_2$ simply perform $C = C_1 \cup C_2$. As C_1 and C_2 are sorted in scan order, this operation can be implemented by a merge (see Algorithm 3.6). Note that when computing the union of the collections, coinciding vertices with the same θ may have to be merged to maintain the canonical representation as described in Section 3.1.

The *Sum* operation together with the *Scalar Transformation* can be used to implement *Map Overlay* operations. Therefore, if the resulting collection needs not be stored in disk, i.e., it is to be used as input to a Scalar Transformation and then discarded, then we can build C with memory pointers to elements of C_1 and C_2.

Algorithm 3.6 *sum* – add two scalar fields represented by two WVC

```
Input: $C_1, C_2$
Output: New collection $C$
  $i_1 \leftarrow 0$
  $i_2 \leftarrow 0$
  $C \leftarrow new\ WVC$
  while $i_1 < |C_1| \wedge i_2 < |C_2|$ do                ▷ $|C|$ returns the size of collection
      if $getAt(C_1, i_1) < getAt(C_2, i_2)$ then               ▷ Next vertex in scan order?
          $push(C, getAt(C_1, i_1))$                       ▷ Insert the vertex at end of $C$
          $i_1 \leftarrow i_1 + 1$
      else
          $push(C, getAt(C_2, i_2))$
          $i_2 \leftarrow i_2 + 1$
      end if
  end while
  while $i_1 < |C_1|$ do
      $push(C, getAt(C_1, i_1))$
      $i_1 \leftarrow i_1 + 1$
  end while
  while $i_2 < |C_2|$ do
      $push(C, getAt(C_2, i_2))$
      $i_2 \leftarrow i_2 + 1$
  end while
  return $C$
```

3.4.2 Value At

Let S be a scalar field, P be a set of points in scan order and $S(p)$ denote the value of S at p. Then, to perfom *Value At*, the algorithm obtains the next point

$p_{valueAt} = pop(P)$ and executes the *Scan* procedure until reaching $p_{scan} = p_{valueAt}$, where p_{scan} is the last coordinate updated in $L(y)$. Thus, $S(p)$ is equal to the sum of field changes introduced by rays to the left of p_x. The process is repeated until P is empty.

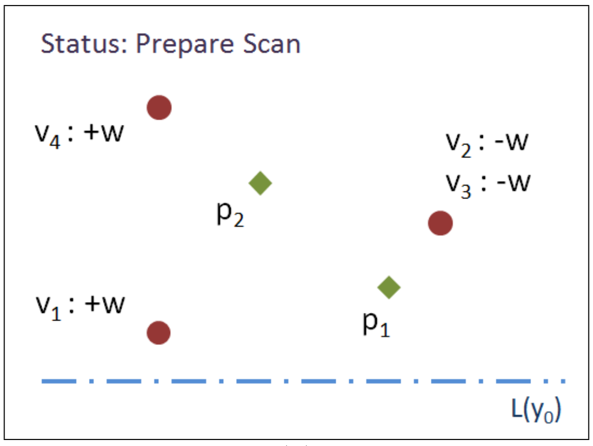
Algorithms 3.7 and 3.8 describe the process in detail. Procedure *prepareScan*, described in Algorithm 3.1, must be modified so as to include special evaluation events $ev_{eval}(p)$ in E. Notice, however these events do not update $L(y)$, serving only to ensure that scanline will stop at p to evaluate it. In addition, the modified *prepareScan* simply creates events for the merge of two sorted lists in scan order: the WVC and P. Figure 3.12 illustrates the process to evaluate two points.

Algorithm 3.7 *valueAt* – calculates the scalar values at a set of points

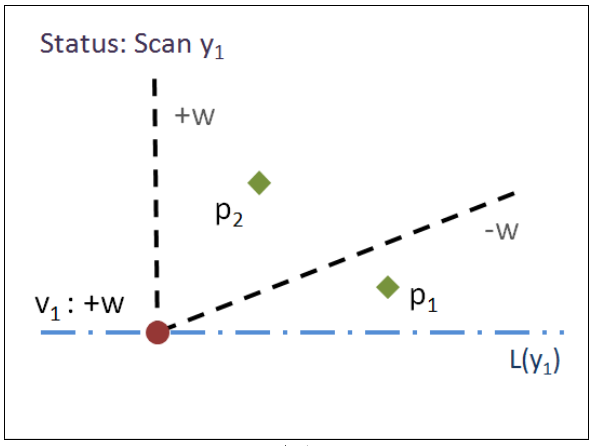
```
Input: $C, P$ – a WVC and a point list
Output: $V_p$ – a list of scalar values   ▷ The calculated values are returned in a list
  $L(y), E \leftarrow prepareScan(C, P)$   ▷ A variant of Algorithm 3.1
  while $|P| > 0 \lor |E| > 0$ do
      $p \leftarrow pop(P)$   ▷ Get next point
      while $point(top(E)) > p$ do   ▷ Scan until the next event in $E$ stays after $p$ in scan order
          $scan(L(y), E)$   ▷ Scan next point $p_{scan}$
      end while
      $s \leftarrow computeValueAt(L(y), p)$   ▷ See Algorithm 3.8
      $push(V_p, s)$   ▷ Add the scalar value to the result
  end while
  return $V_p$
```

Algorithm 3.8 *computeValueAt* – calculates the scalar value at one point on a scanline

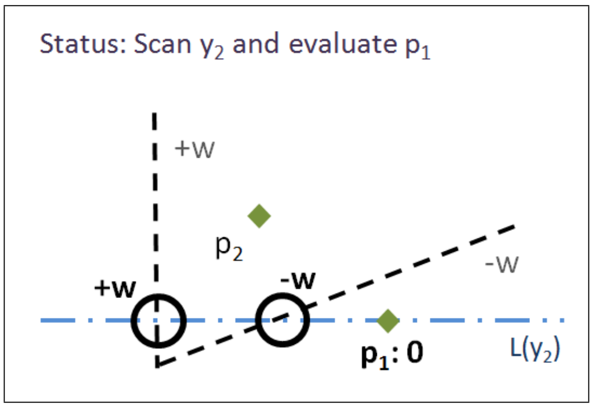
```
Input: $L(y), p$
Output: $s$ – the value of scalar field at
  $s \leftarrow 0$
  for each $r \in L(y)$ do
      $p_{cross} \leftarrow crossPoint(r, L(y))$   ▷ Compute the point where ray crosses $L(y)$
      if $x(p_{cross}) < x(p)$ then   ▷ Crossing point is at a lower $x$ than $p$?
          $s \leftarrow s + ds(r)$   ▷ Add ray change to scalar field
      else
          return $s$   ▷ Stop the loop and return the value
      end if
  end for
  return $s$   ▷ $p$ has bigger abscissa than all rays
```



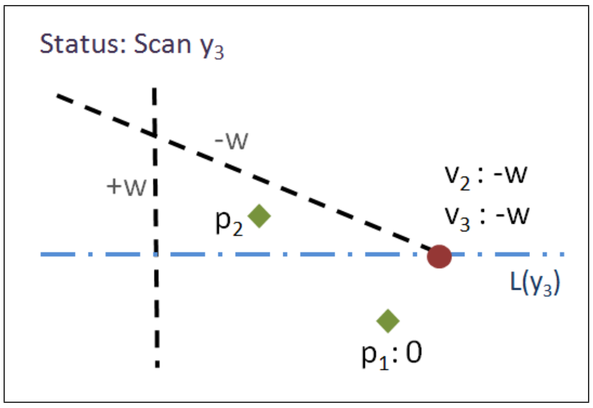
(a)



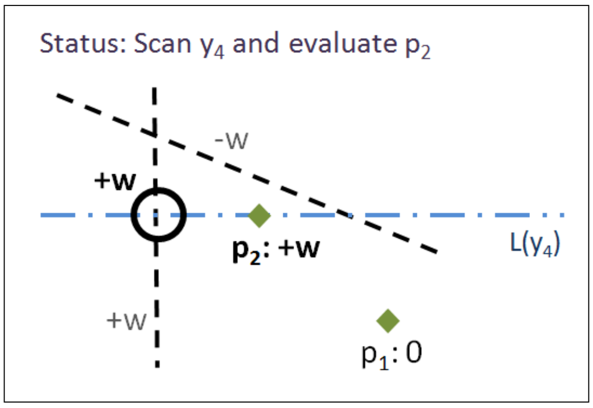
(b)



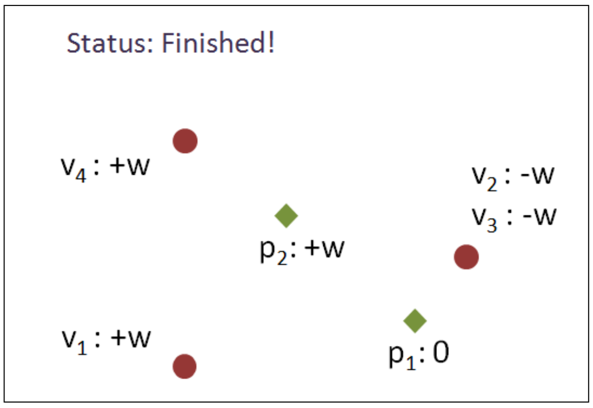
(c)



(d)



(e)



(f)

Figure 3.12: Example of *Value At* operation with two points. Details about the scan update at each y can be seen in Figures 3.7 through 3.11

.

3.4.3 Draw

The purpose of the *Draw* operation is to generate a picture of the regions where each class of the map symbology is painted in its corresponding color. Each color is associated to a value w of the scalar field, thus it may be necessary to display a scalar field represented by WVC as, for instace, a result of *Scalar Transformation.*

A convenient way to paint a polygonal region represented by a WVC is to generate a trapezoidal decomposition of the region, and rendering each trapezoid with the appropriate color. The Computational Geometry literature (e.g., [22]) describes a classical algorithm for tiling the plane with trapezoids with sides adjacent to a given collection of line segments.

A trapezoid t must be created whenever a $ev_{in}(r)$ or $ev_{inter}(r_{cr1}, r_{cr2})$ are processed in Scan operation, where this new t is limited by two rays laterally, in top by the $L(y)$ and in bottom by the above trapezoid already created or by $L(y_0)$, ie, the horizontal line in beginning of scan. To compute the weight, namely $w(t)$, it is necessary performs the *Value At* operation. The following procedures describe the creation of trapezoids given the events:

1. If a $ev_{in}(r)$ inserts r in $L(y)$ at position i, then one trapezoid must be created between rays $i - 1$ and $i + 1$.

2. When a $ev_{in}(r)$ is processed and r overlaps another ray r_{ovlap} in scanline at position i, thus one trapezoid limited laterally by rays i and $i + 1$ must be created. Besides from that, if $ds(r_{ovlap})$ is updated to zero, thus will be removed from $L(y)$, then other trapezoid is generated between rays at $i - 1$ and i.

3. Whenever that one or more $ev_{inter}(r_{cr1}, r_{cr2})$ updated $L(y)$ onto point p_{cross}, thus the trapezoids need be generated for all rays r passing in p_{cross}. Each trapezoid is limited laterally by ray r and its left ray.

It is noteworthy that two vertical rays are added in Scan process to support the trapezoidal decomposition, in order that each sentinel ray be added rightmost and leftmost than the events and that this sentinels stay in scanline during the entire scan. These sentinel rays not change $L(y)$. Figure 3.13 shows a trapezoidal decomposition of Figure 3.7.

3.4.4 Convert

Since GIS software typically represent regions by polygons, it is essential to devise a procedure to convert such representations to *Weighted Vertex Collections.* An operation to convert a *Weighted Vertex Collection* into vertex circulations is also important.

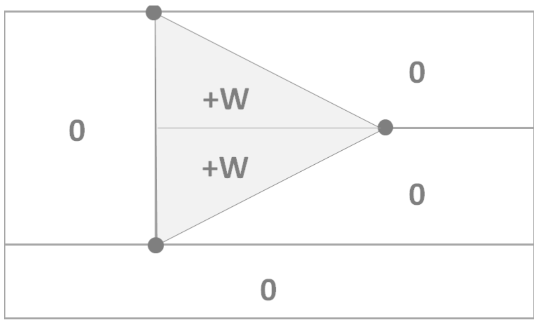


Figure 3.13: Example of trapezoidal decomposition, where the white trapezoids have $w = 0$ and gray trapezoids have $w = +w$.

The conversion to *Weighted Vertex Collection* is relatively trivial. First, it is created one collection C for each polygon P of the map. Let S be a scalar field, C a weighted vertex collection, P a polygon with counterclockwise vertex circulation, e an edge of P, p_1 the first endpoint of e in circulation order and p_2 as the last, x_1 and x_2 the abscissas of p_1 and p_2, w the weight to be assigned in S to the region defined by P and $w(v)$ the weight of weighted vertex v. Each pair of points p_1 and p_2 generates weighted vertices v_1 and v_2 in C. The θ of v_1 and v_2 are equal to slope of e. If $x_1 < x_2$, then $w(v_1) = +w$ and $w(v_2) = -w$. Otherwise, if $x_1 > x_2$, then $w(v_1) = -w$ and $w(v_2) = +w$. If $x_1 = x_2$, then e is a vertical edge, and need not be represented in C. After generating the WVC for each polygon, a sum operation is used to generate one collection with the vertices of all input WVCs. The weight values may be, for instance, an incremental number given to each class of map or to each feature.

The conversion of the collection into vertex circulations is performed by a *plane sweep* procedure. The scan searches the regions boundaries with the same value in the scalar field. These boundaries are given by the cone rays generated by the weighted vertices.

3.4.5 Scalar Transformation

The scalar transformation is defined as a function $f(x) = y$; $\Re \rightarrow \Re$ that changes the values of the scalar field S to new values in a transformed field S_t, where a new vertex collection C_t must be created to represent S_t.

Typical *Map Overlay* operations such as union, intersection and difference can be performed with the aid of a scalar transformation. Consider two collections C_1 and C_2, representing polygons P_1 and P_2 respectively. The region defined by P_1 is mapped to 1 and the P_2 region to 2 - see Figure 3.4.5. For instance, by applying the *Sum* operation $C_3 = C_1 \cup C_2$, it is possible to create the obtain different results by using an appropriate function:

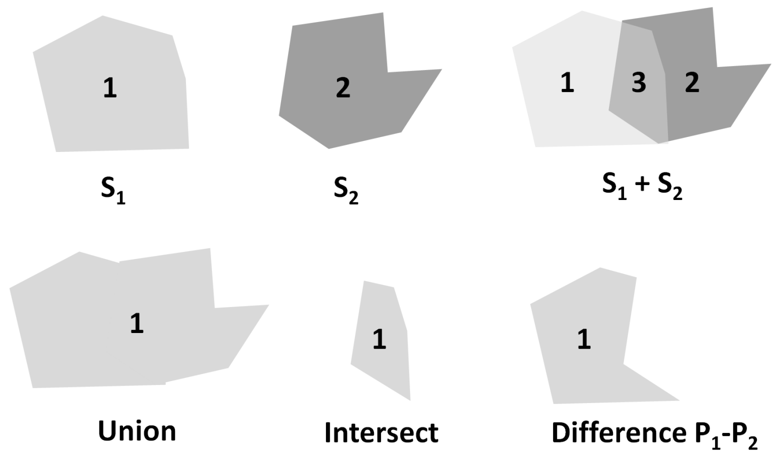


Figure 3.14: Some basic functions to *Scalar Transformation*.

$$Union : f_{\cup}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$Intersect : f_{\cap}(x) = \begin{cases} 1 & \text{if } x > 1 \\ 0 & \text{otherwise} \end{cases}$$

$$Difference\ P_1 - P_2 : f_{-}(x) = \begin{cases} 1 & \text{if } x = 1 \\ 0 & \text{otherwise} \end{cases}$$

At first, the transformed collection C_t is empty and a scan process is executed together on C and C_t using respectively the scanlines $L(y)$ and $L_t(y)$. In initial understanding, the condition $S_t(p) = f(S(p)), \forall p \in S$ must be satisfied. However, it is noteworthy by plane sweep paradigm that this check is only necessary in the stop events e of scan process. Given this consideration, the check may be performed for each point with a event e processed in $L(y)$ and it may be rewritten to $S_t(e) = f(S(e)), \forall e \in L(y)$. The $S(e)$ is summed similarly to point in *Value At* operation, besides all rays of $L(y)$ and $L_t(y)$ that crosses e must be checked. For aim this, each ray r is examined sum up to $S(e)$ the pertubation of rays that cross e and have bigger angles than r, that is, summed pertubations of rays in $L(y)$ that are lower than r in scan order using the operation $cross(r, L(y))$, because the interest is to examine the status of the scalar field above $L(y)$ and on the right of r. Ultimately, a convinient way to perform and understand the *Scalar Transformation* is processing all events of the next point p in scan order and checking the condition $S_t(r) = f(S(r)), \forall r$ in $L(y)$ or $L_t(y)$ and that cross p, where $S(r, L(y))$ or simply $S(r)$ is the value of scalar field on right of r and above from $L(y)$ until other change occurs in S.

If this condition is not met, it is forced adding a new weighted vertex v_{new} in C_t

and adding the rays $r_s(v_{new})$ and $r_v(v_{new})$ in $L_t(y)$, where $w(v_{new}) = S_t(r) - f(S(r))$ and $\theta(v_{new}) = \theta(r)$, where $\theta(r)$ is the ray angle. If the failure of condition is detected when a vertical ray is checked, then the new vertex is added in the next check with a sloping ray - and this necessarily occurs. Otherwise, a vertex with $\theta = \pi/2$ will be created.

The Algorithm 3.9 represents a pseudocode of *Scalar Transformation* with the above descriptions. The procedure *insertRay* showed in Algorithm 3.4 must be ajusted in Algorithm 3.9, because in this case it is not necessary detect intersect events in $L_t(y)$ - they are already detected in $L(y)$. Thus the procedures *checkInter* need not be called.

It is necessary performs $S(r)$ many times for same coordinate p, where at each computing all rays lower than r in scan order must have their perturbations summed again. Then seeking a efficient way, the Algorithms 3.9, 3.10 and 3.11 propose to evaluate the scalar field at left of first ray that crosses p, updating it incrementally when the procedure *getNextRay* is performed.

Consider the Figure 3.15 showing a scalar field S with two triangles and the Figures 3.16 through 3.33 ilustrating an partial example of intersection operation using the function $f_\cap$ above, where the Figure 3.34 presents the transformation result. It is highlighted in these figures:

1. As the figures are illustrating, it may be defined two distinct moments in *Scalar Transformation*: scan next p and check the last p scanned.

2. The C_t received the first vertex in Figure 3.24.c, but the condition fails in 3.24.a when is checked the vertical ray $r_v(v_1)$. Thus the deficit is computed (see variable d in Algorithm 3.9) to create a new vertex in the next check over $r_s(v_2)$. A similary case occurs in Figure 3.27.a, while in Figure 3.26.b shows a vertex creation immediately when the condition is not met.

3. The rays of $L(y)$ and also $L_t(y)$ must be checked, as shown in Figure 3.27.a.

4. About the scan process, two points have events in $L(y_1)$ and they are updated separately.

3.5 The WVC data structure

Following are recommendations of convenient data structures to a WVC implementation.

The vertices, events and rays can overlap and thus share a point in space, thus to optimize storage space these points are organized in a hash table. Considering

Algorithm 3.9 *scalarTransf* - perform the operation of scalar transformation

Input: C, f - a WVC and the transformation function.

Output: C_t - the transformed WVC

```
  $L(y), E \leftarrow prepareScan(C)$                                ▷ See Algorithm 3.1
  $C_t \leftarrow new\ WVC$                                          ▷ Create a empty WVC
  while $|E| > 0$ do                                                 ▷ Until the scan finished
      $p \leftarrow point(top(E))$   ▷ Get the next coordinate of event that will be processed
      $scan(L(y), E)$                                                ▷ See Algorithm 3.2
      $R \leftarrow getRays(L(y), p)$          ▷ Get rays that crosses $p$ sorted by higher angle
      $R_t \leftarrow getRays(L_t(y), p)$
      $S(r) \leftarrow valueAtRay(L(y), top(R))$                     ▷ See Algorithm 3.11
      $S_t(r) \leftarrow valueAtRay(L_t(y), top(R_t))$
      $d \leftarrow 0$                          ▷ Stored the deficit in scalar field at last check
      while $|R| > 0 \vee |R_t| > 0$ do                    ▷ While there are rays to test...
          $r \leftarrow getNextRay(R, R_t, S(r), S_t(r))$              ▷ See Algorithm 3.10
          if $S_t(r) \neq f(S(r)) \vee d \neq 0$ then                 ▷ Is the condition met?
              $w \leftarrow S_t(r) - f(S(r)) + d$   ▷ Compute the weight - see Algorithm 3.11
              if $\theta(r) = \pi/2$ then                             ▷ Is a vertical ray?
                  $d = w$                               ▷ Stored the deficit to next loop...
              else
                  $d = 0$                                       ▷ Nulled the deficit, if exist
                  $v_{new} \leftarrow new\ Vertex(p, \theta(r), w)$   ▷ Creating the necessary vertex
                  $push(C_t, v_{new})$                          ▷ Insert the new vertex in $C_t$
                  if $\theta(r) > \pi/2$ then     ▷ To insert rays in $L_t(y)$ obeying scan order
                      $insertRay(L_t(y), r_s(v_{new}))$           ▷ A variant of Algorithm 3.4
                      $insertRay(L_t(y), r_v(v_{new}))$
                  else
                      $insertRay(L_t(y), r_v(v_{new}))$
                      $insertRay(L_t(y), r_s(v_{new}))$
                  end if
              end if
          end if
      end while
  end while
  return $C_t$                                                       ▷ Return the result
```

Algorithm 3.10 *getNextRay* - get the next ray to be checked

Input: R_1, R_2, s_1, s_2 - two lists of rays sorted by higher angle and the scalar value of S_1 and S_2 to the right of the top ray of each list.

Output: r - the next ray

```
$r_1 \leftarrow top(R_1)$        ▷ Look next ray of each list, if $|R| = 0$ thus $\theta(top(R)) = -\pi$...
$r_2 \leftarrow top(R_2)$        ▷ ...and this ray never is returned in this function.
if $\theta(r_1) > \theta(r_2)$ then
    $r \leftarrow pop(R_1)$        ▷ The next ray is from $R_1$
    $s_1 \leftarrow s_1 + ds(r)$        ▷ Updating the scalar value so that $s_1 = S_1(r)$
else if $\theta(r_1) < \theta(r_2)$ then
    $r \leftarrow pop(R_2)$        ▷ The next ray is from $R_2$
    $s_2 \leftarrow s_2 + ds(r)$        ▷ Updating the scalar value so that $s_2 = S_2(r)$
else        ▷ $\theta(r_1) = \theta(r_2)$
    $r \leftarrow pop(R_2)$        ▷ Withdraw, so gets next in both lists
    $s_2 \leftarrow s_2 + ds(r)$
    $r \leftarrow pop(R_1)$
    $s_1 \leftarrow s_1 + ds(r)$
end if
return $r$        ▷ Return the result
```

Algorithm 3.11 *valueAtRay* - get the value of scalar field given a ray

Input: $L(y), r_1$ - a scanline and a ray.

Output: s - scalar value

```
$p_{1cr} \leftarrow cross(L(y), r_1)$        ▷ Compute the cross point between ray and scanline
$s \leftarrow 0$
for each $r_2 \in L(y)$ do
    $p_{2cr} \leftarrow cross(L(y), r_2)$
    if $x(p_{cr1}) > x(p_{cr2}) \vee \theta(r_1) < \theta(r_2)$ then        ▷ Comparing by scan order
        $s \leftarrow s + ds(r_2)$        ▷ Updating the scalar value
    else
        return s        ▷ Stop the loop and return the scalar value
    end if
end for
return s        ▷ $r_1$ is rightmost than all rays of $L(y)$
```

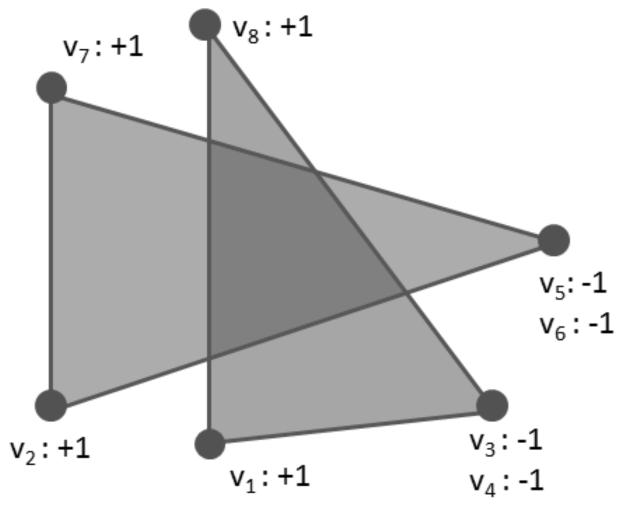


Figure 3.15: A scalar field S to be transformed. The white area values 0, darkest area values +2 and otherwise +1.

that points are ajusted a fine grid to robustness of floating-point operations, the hash function may to receive the coordinate and to give a grid position, where it is important choose a function that avoids collisions of two different coordinates as, for instance, a function that uses the first integers digits of coordinates. Otherwise, the complexity $O(1)$ of hash table will be compromised [21].

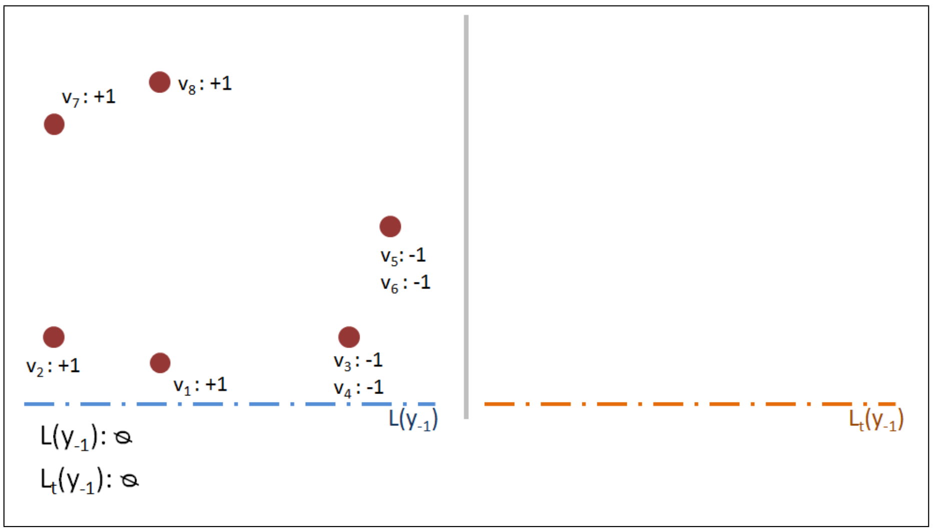
The structures of event queue E and scanline $L(y)$ must have its elements ordered in scan order, where ordered lists or balanced trees can be used. As long as is applied a sorting algorithm suitable, then the complexity is $O(\log n)$ for search or update and $O(n \log n)$ for sorting the entire structure.

Analysing the complexity of operations described in Section 3.4:

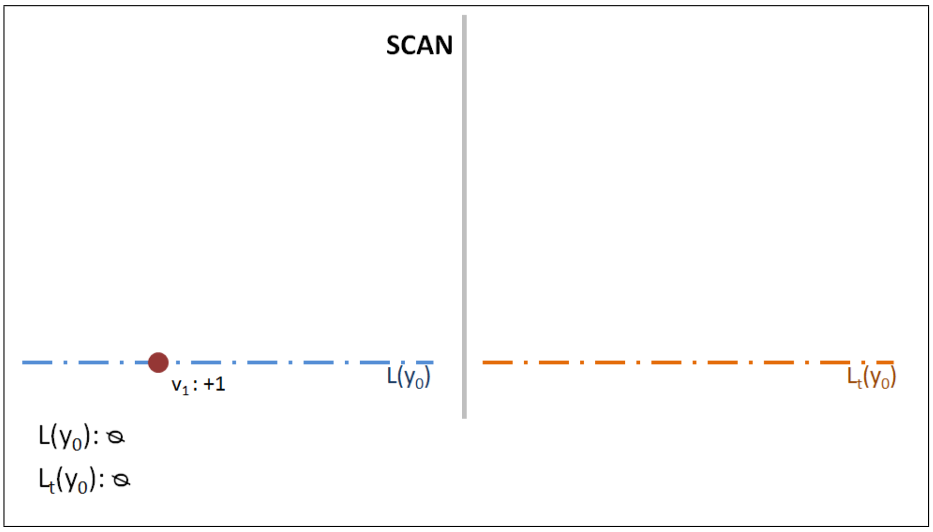
Sum: the complexity is $O(n)$ to merge the sorted lists of vertices, where n is the size of two summed collections.

Value At: given n as the size of $L(y)$, p as number of points to be evaluated, and I as the number of ray intersections, then the scan process provides a complexity of $O((I + n + p) \log n)$ [22]. However, to compute the scalar value at a point in $L(y)$ as described by procedure *computeValueAt* in Algorithm 3.8, it is necessary $O(n)$ time because the sum is computed by a sequential examining of $L(y)$. Thus the total complexity in the worst case is $O(n(I+n+p) \log n) = O(n^2)$. Alternatively, a data structure such as a *skip list* [27] can be applied to grant $O((I + n + p) \log n)$.

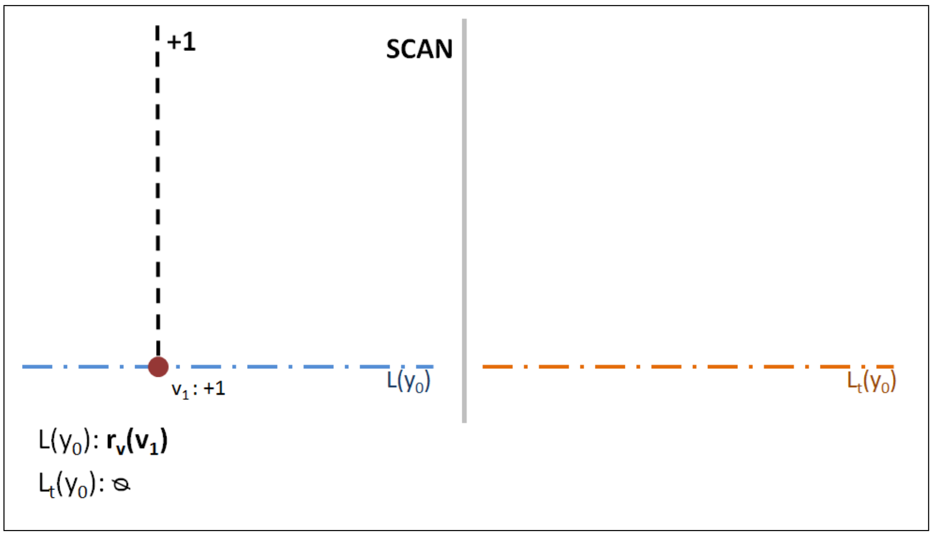
Convert: to convert vertex circulations in WVC is complexity $O(n \log n)$ to build the ordened collections of weighted vertices, given n as the number of vertices in all circulations. On the other hand, the conversion of WVC to a vertex



(a)

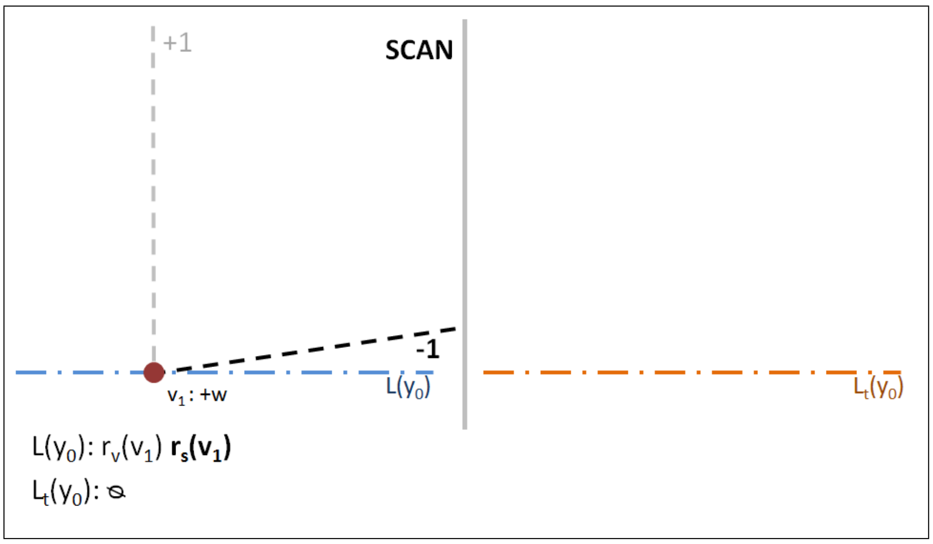


(b)

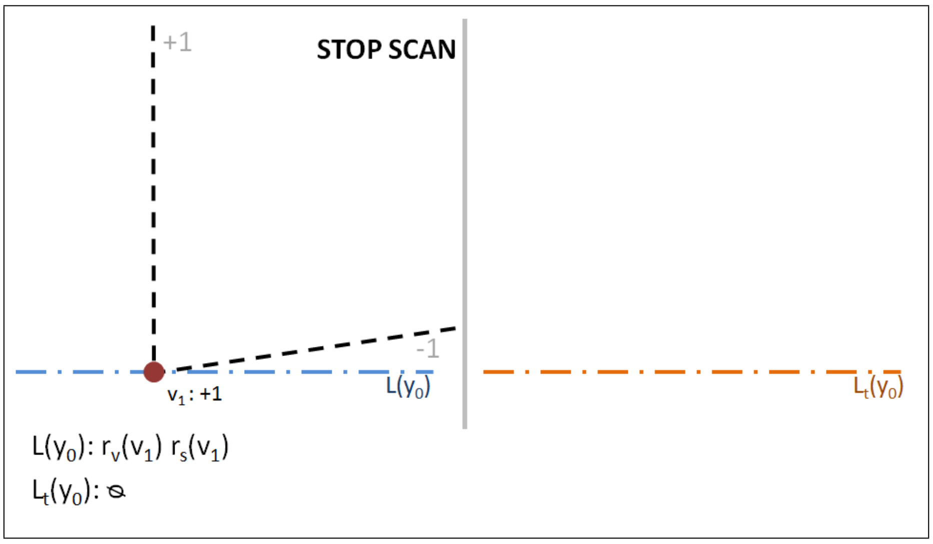


(c)

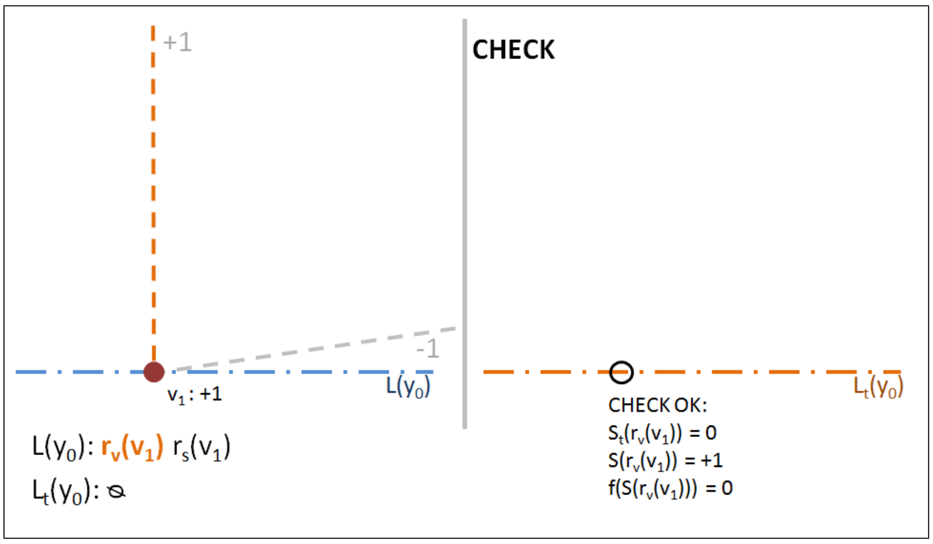
Figure 3.16: Example in step by step of *Scalar Transformation* - part 1 of 18.



(a)

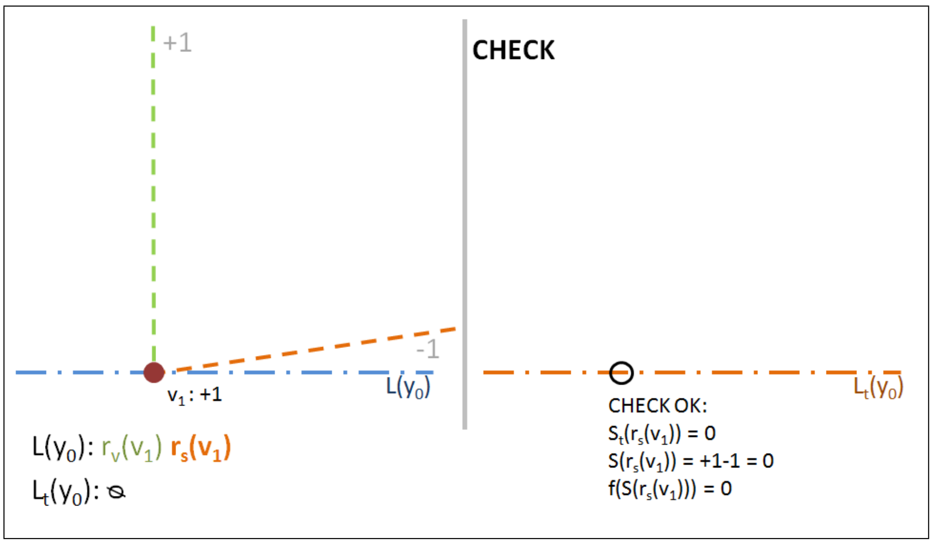


(b)

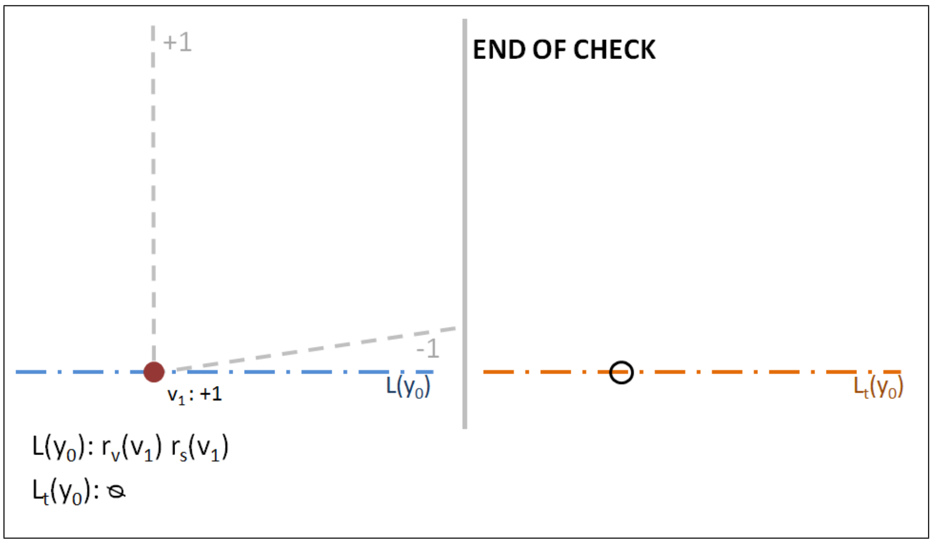


(c)

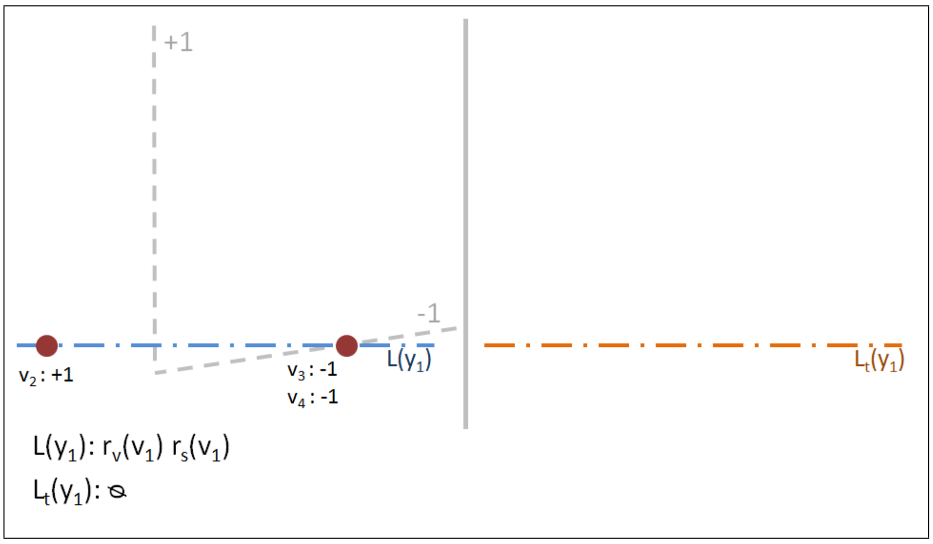
Figure 3.17: Example in step by step of *Scalar Transformation* - part 2 of 18.



(a)

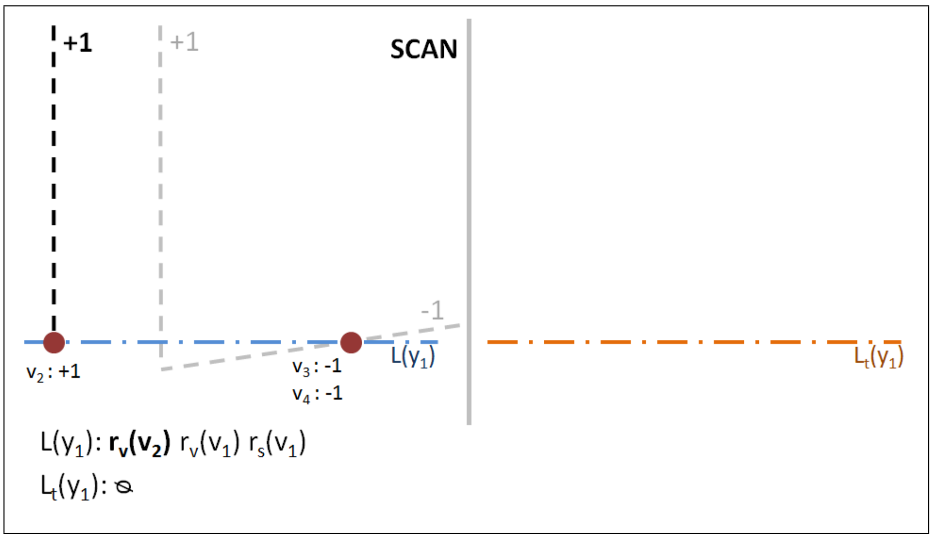


(b)

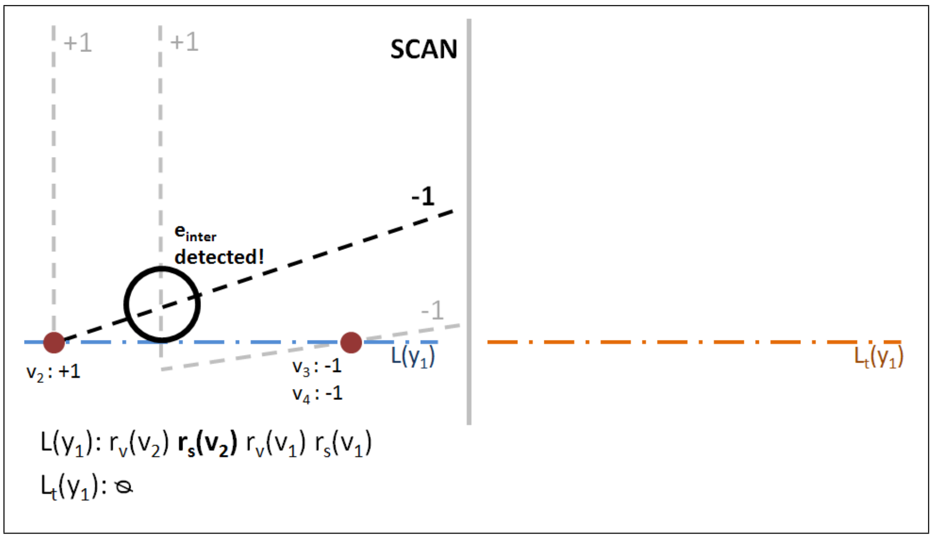


(c)

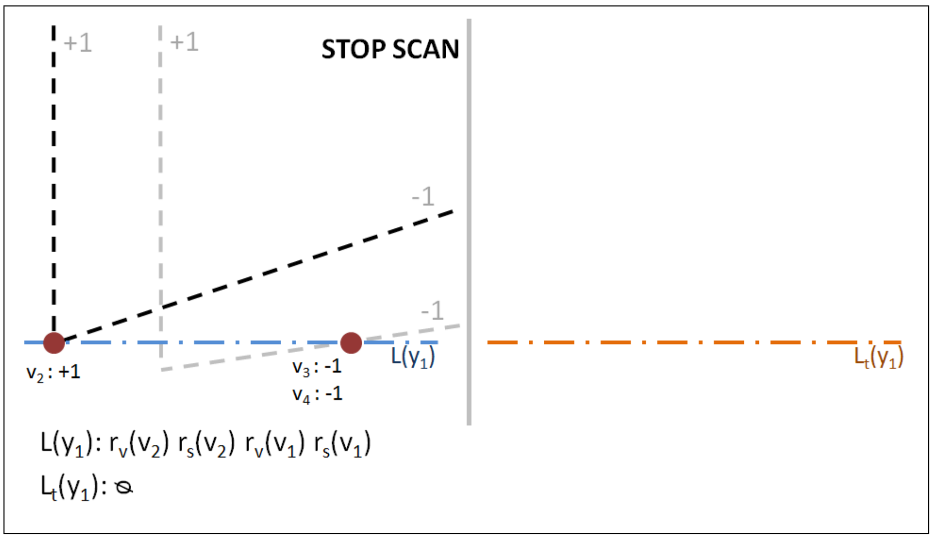
Figure 3.18: Example in step by step of *Scalar Transformation* - part 3 of 18.



(a)

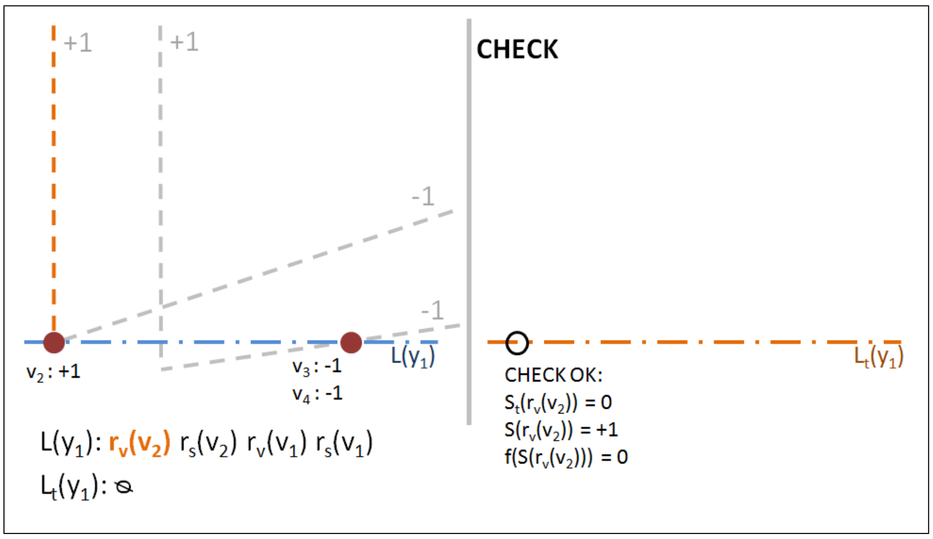


(b)

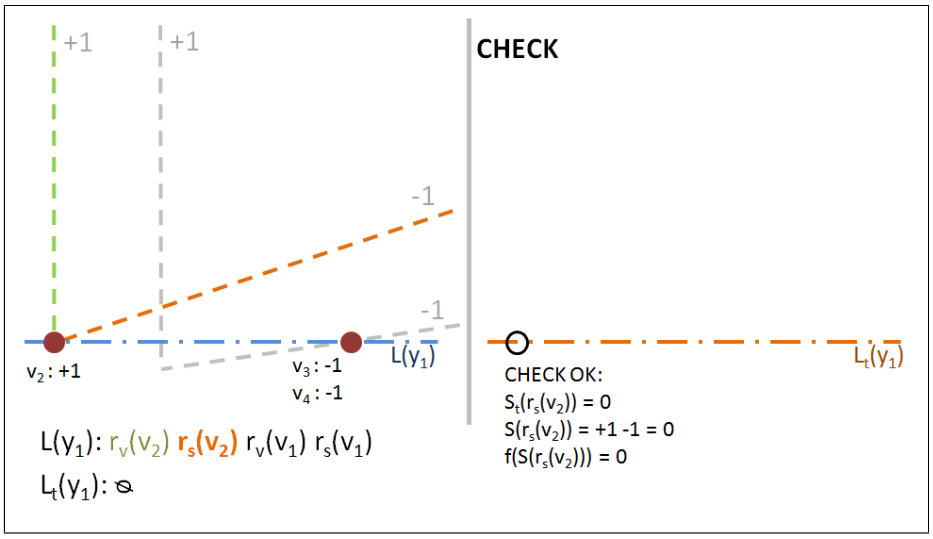


(c)

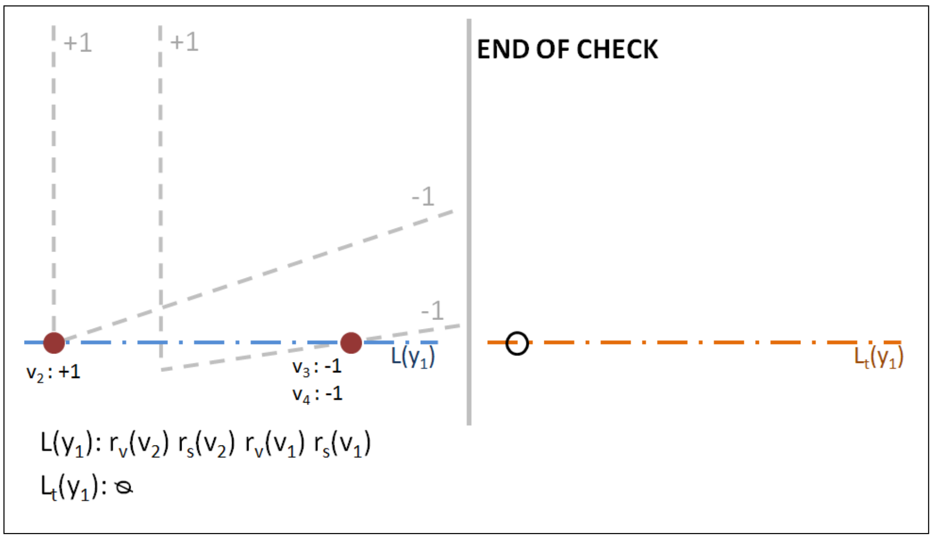
Figure 3.19: Example in step by step of *Scalar Transformation* - part 4 of 18.



(a)

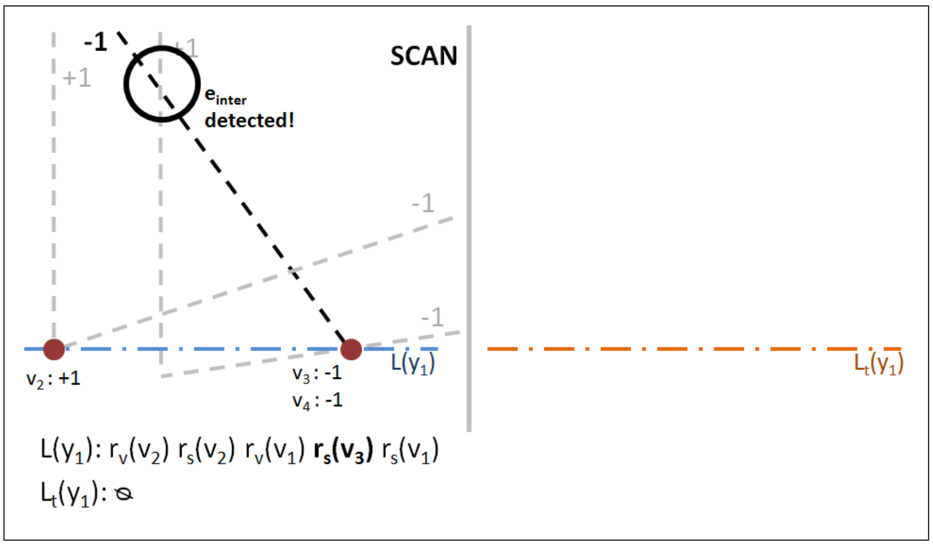


(b)

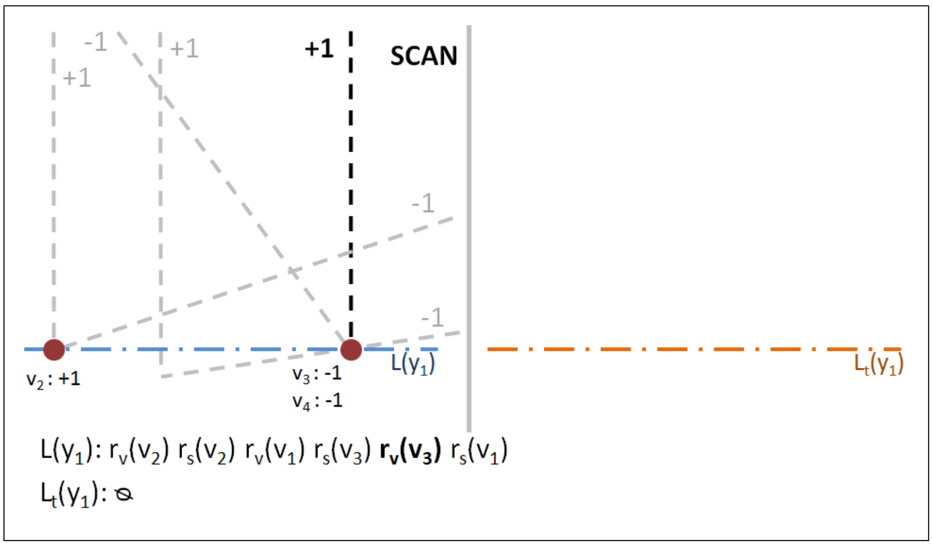


(c)

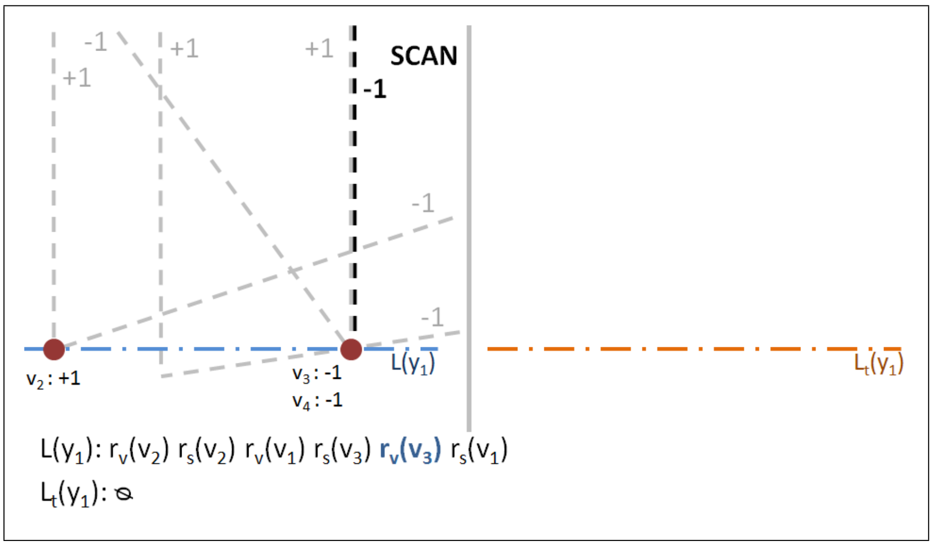
Figure 3.20: Example in step by step of *Scalar Transformation* - part 5 of 18.



(a)

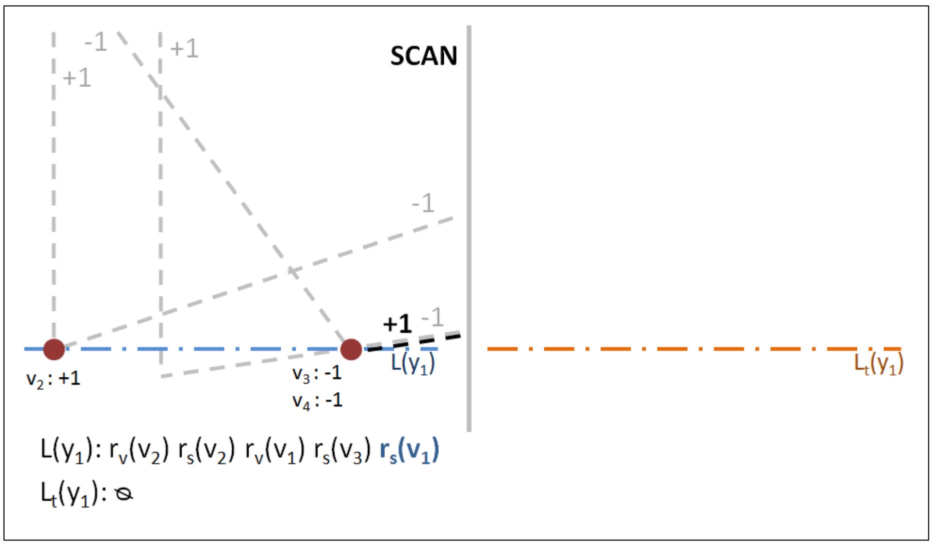

-1
+1
+1
+1
SCAN
-1
-1
v_2 : +1
v_3 : -1
v_4 : -1
$L(y_1)$
$L_t(y_1)$
$L(y_1)$: $r_v(v_2)$ $r_s(v_2)$ $r_v(v_1)$ $r_s(v_3)$ $\mathbf{r_v(v_3)}$ $r_s(v_1)$
$L_t(y_1)$: ∅

(b)

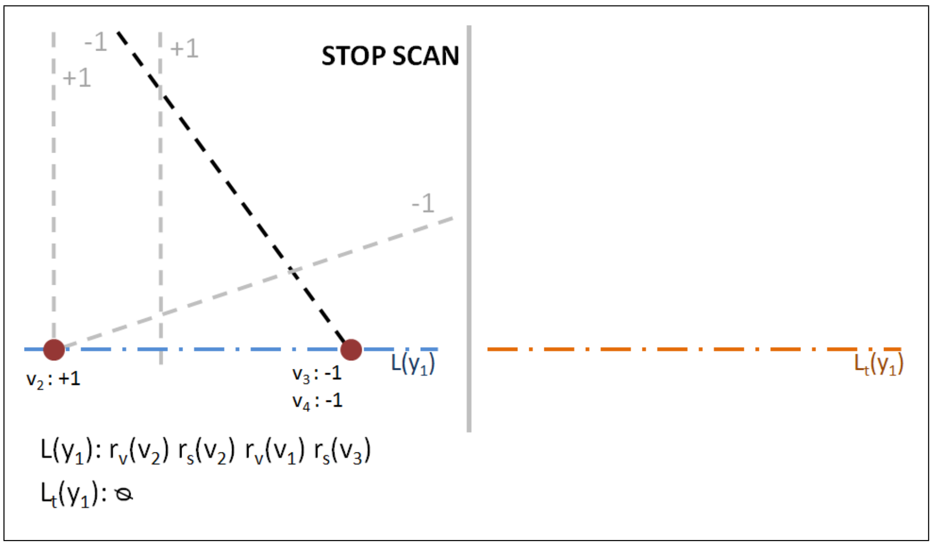

-1
+1
+1
+1
-1
SCAN
-1
-1
v_2 : +1
v_3 : -1
v_4 : -1
$L(y_1)$
$L_t(y_1)$
$L(y_1)$: $r_v(v_2)$ $r_s(v_2)$ $r_v(v_1)$ $r_s(v_3)$ $\mathbf{r_v(v_3)}$ $r_s(v_1)$
$L_t(y_1)$: ∅

(c)

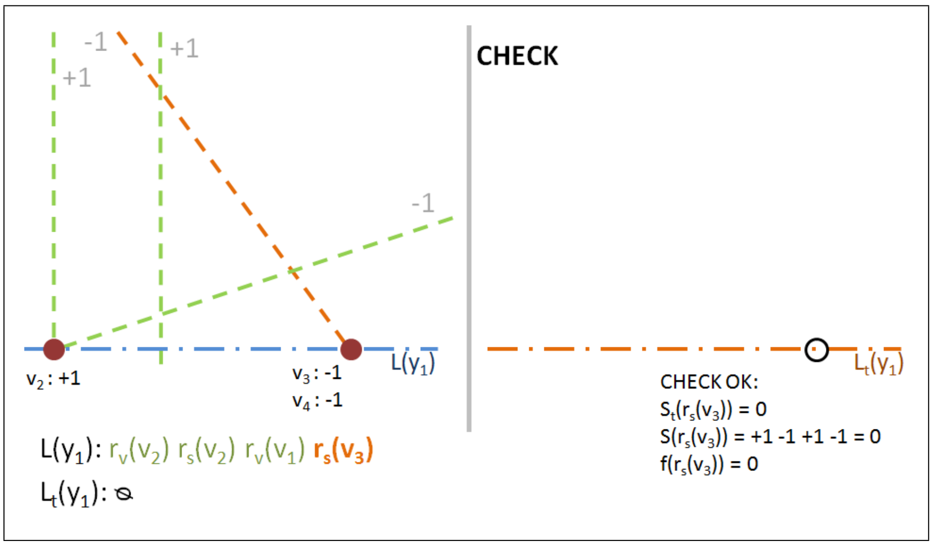
Figure 3.21: Example in step by step of *Scalar Transformation* - part 6 of 18.



(a)

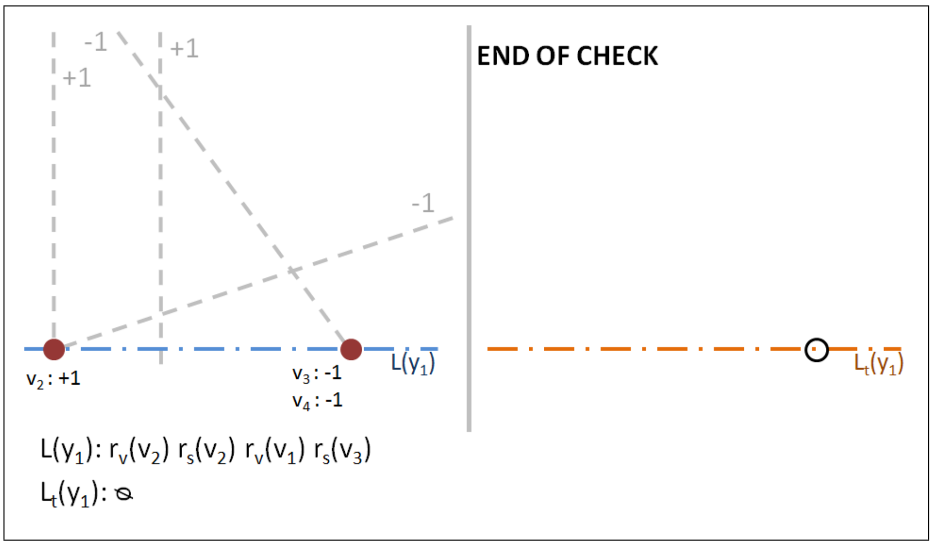


(b)

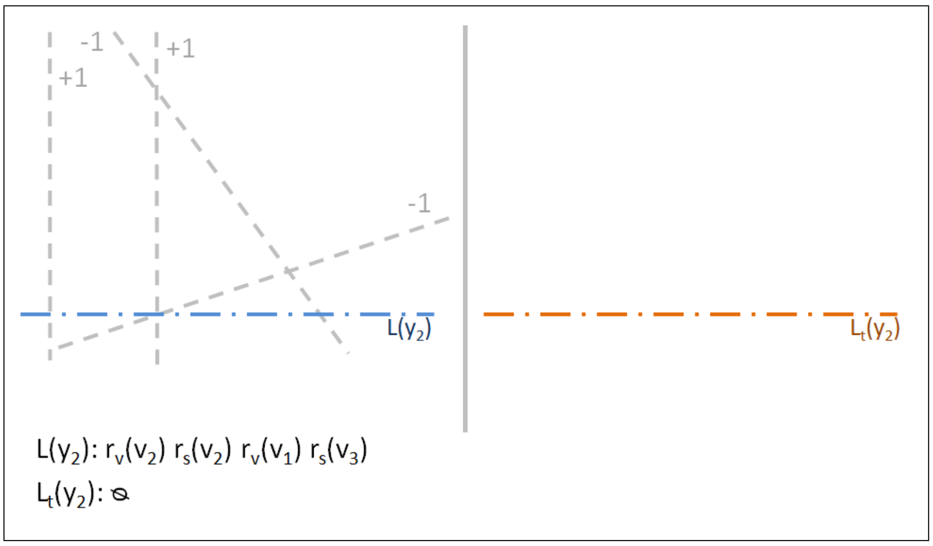


(c)

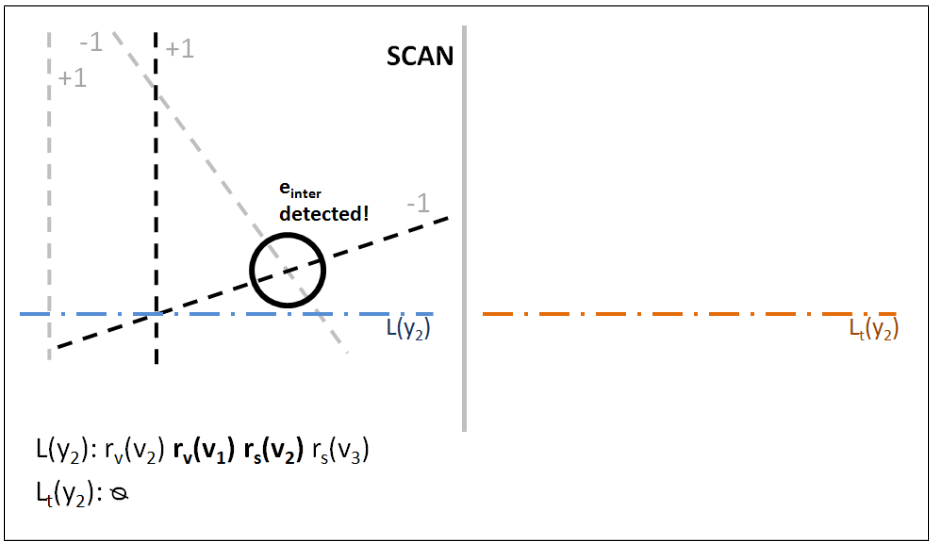
Figure 3.22: Example in step by step of *Scalar Transformation* - part 7 of 18.



(a)

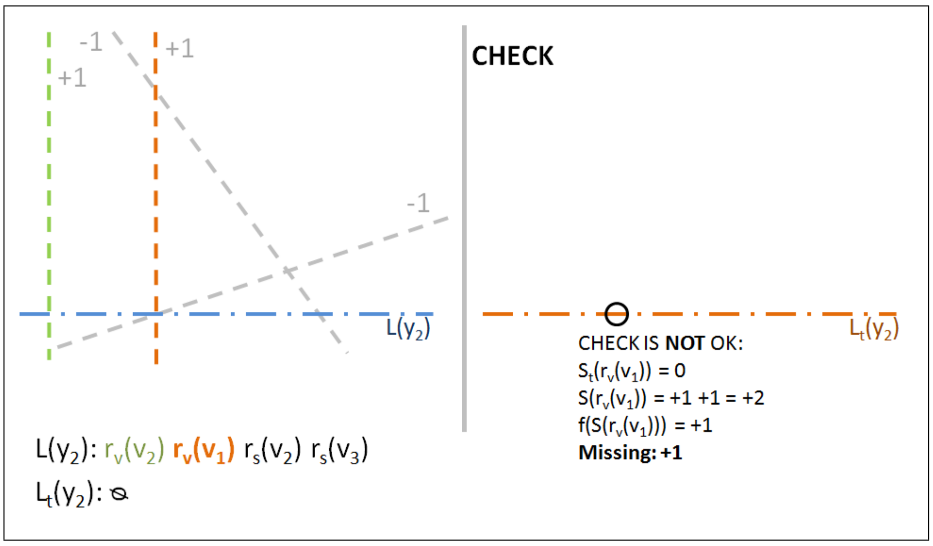


(b)

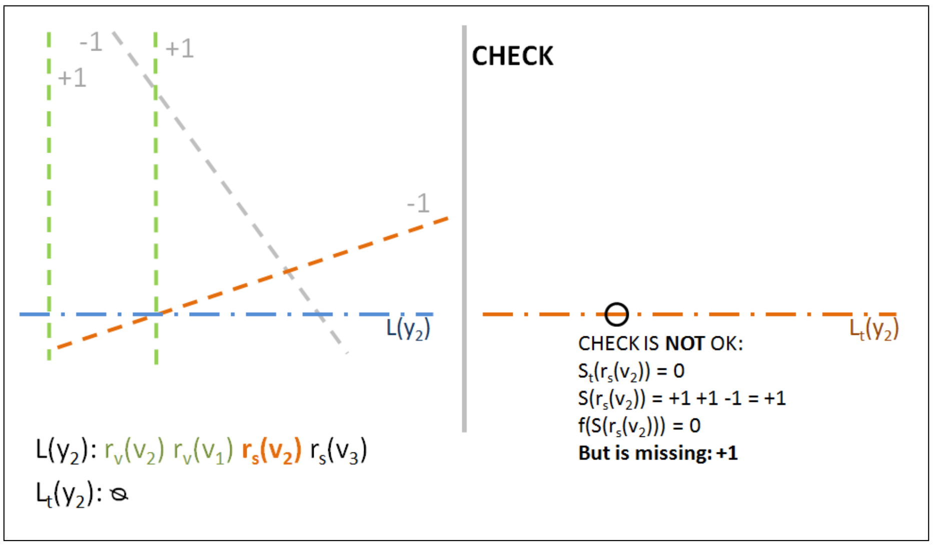


(c)

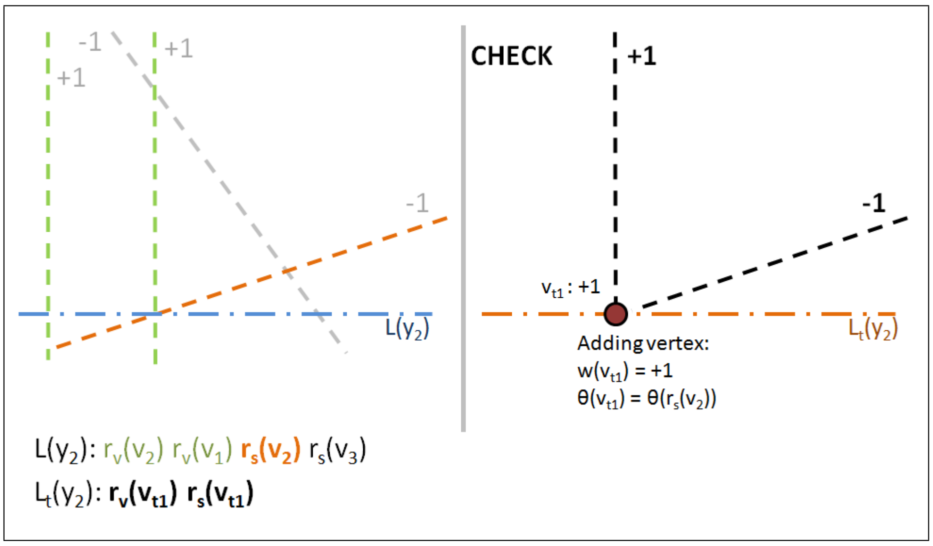
Figure 3.23: Example in step by step of *Scalar Transformation* - part 8 of 18.



(a)

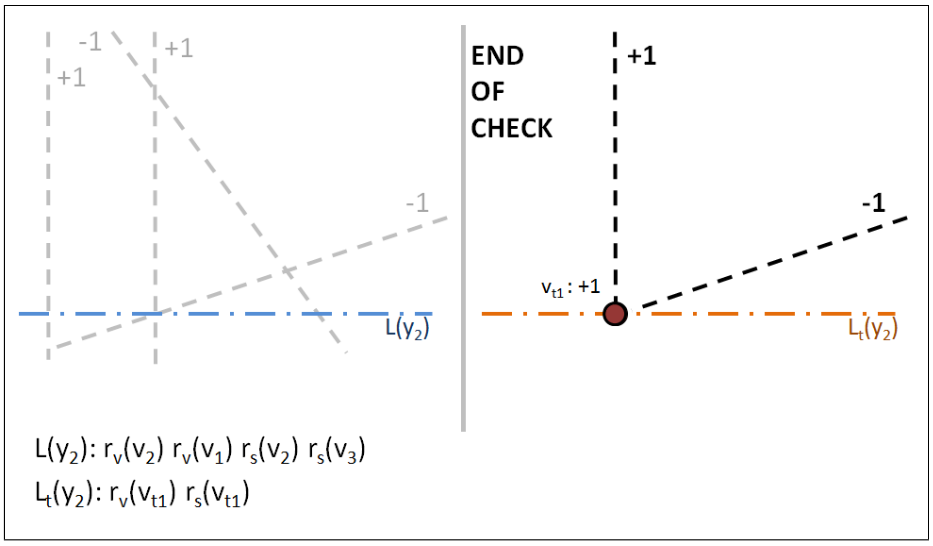


(b)

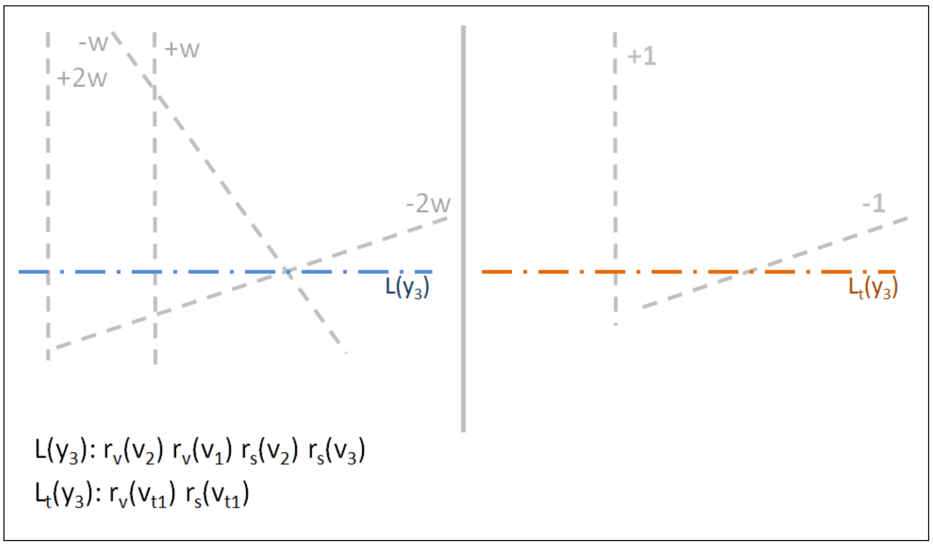


(c)

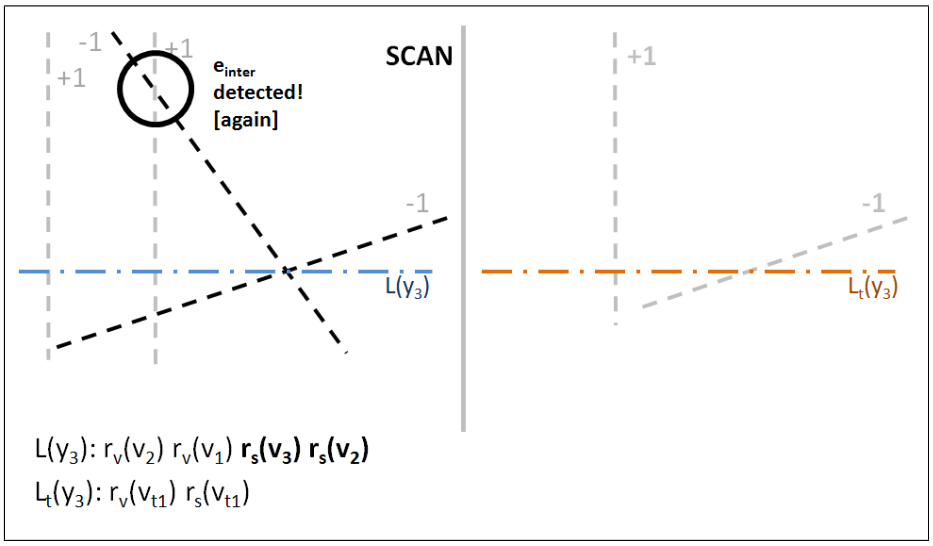
Figure 3.24: Example in step by step of *Scalar Transformation* - part 9 of 18.


-1
+1
+1
-1
L(y_2)
END OF CHECK
+1
-1
v_{t1} : +1
$L_t(y_2)$
$L(y_2)$: $r_v(v_2)$ $r_v(v_1)$ $r_s(v_2)$ $r_s(v_3)$
$L_t(y_2)$: $r_v(v_{t1})$ $r_s(v_{t1})$

(a)

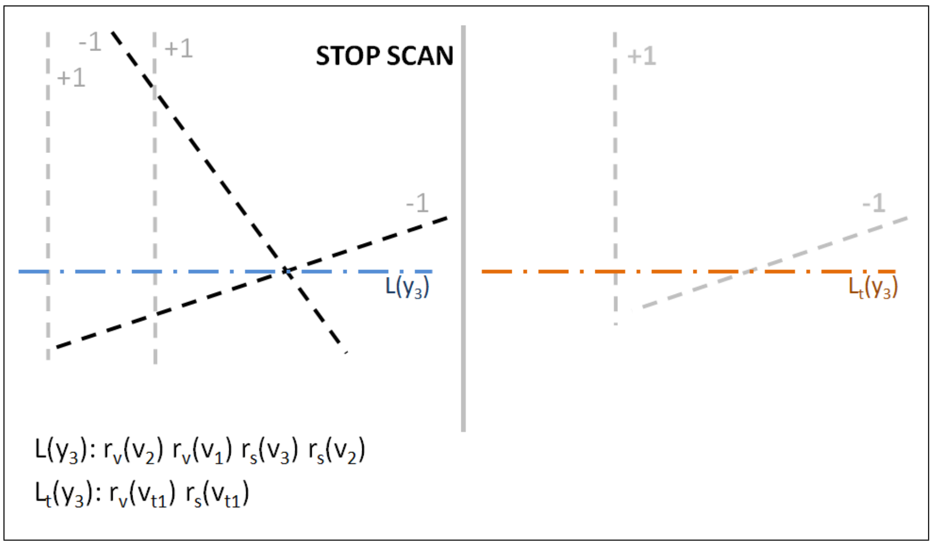

-w
+2w
+w
-2w
L(y_3)
+1
-1
$L_t(y_3)$
$L(y_3)$: $r_v(v_2)$ $r_v(v_1)$ $r_s(v_2)$ $r_s(v_3)$
$L_t(y_3)$: $r_v(v_{t1})$ $r_s(v_{t1})$

(b)

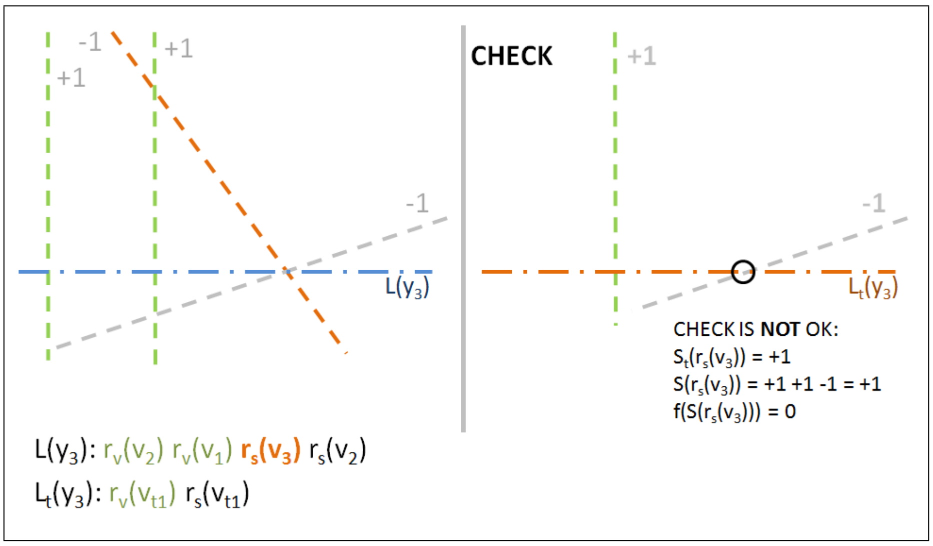

-1
+1
+1
e_{inter} detected! [again]
SCAN
-1
L(y_3)
+1
-1
$L_t(y_3)$
$L(y_3)$: $r_v(v_2)$ $r_v(v_1)$ **$r_s(v_3)$ $r_s(v_2)$**
$L_t(y_3)$: $r_v(v_{t1})$ $r_s(v_{t1})$

(c)

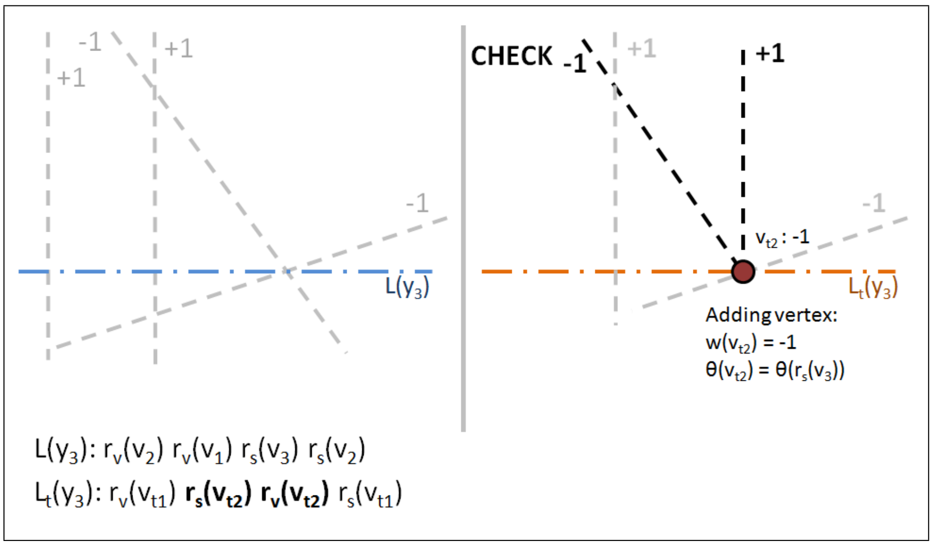
Figure 3.25: Example in step by step of *Scalar Transformation* - part 10 of 18.


-1
+1
+1
STOP SCAN
+1
-1
-1
$L(y_3)$
$L_t(y_3)$
$L(y_3)$: $r_v(v_2)$ $r_v(v_1)$ $r_s(v_3)$ $r_s(v_2)$
$L_t(y_3)$: $r_v(v_{t1})$ $r_s(v_{t1})$

(a)

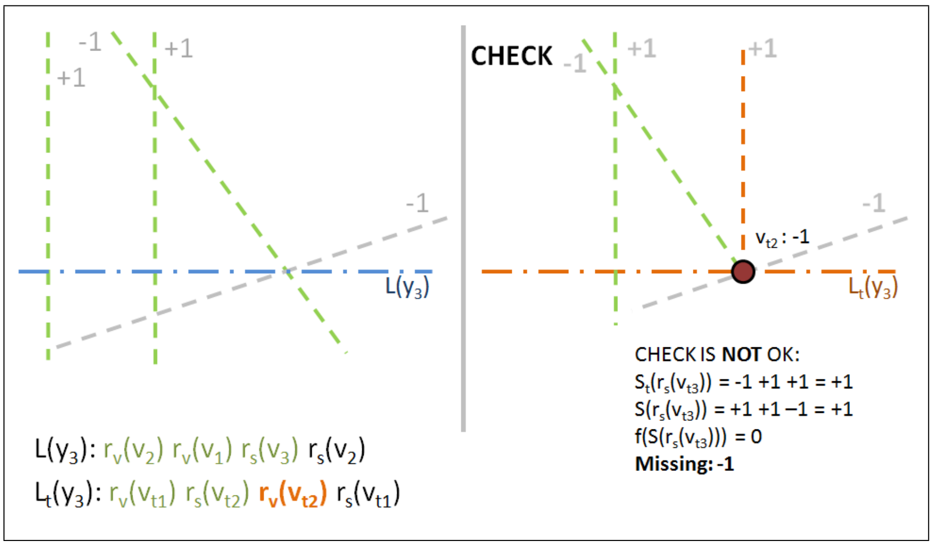

-1
+1
+1
CHECK
+1
-1
-1
$L(y_3)$
$L_t(y_3)$
CHECK IS **NOT** OK:
$S_t(r_s(v_3)) = +1$
$S(r_s(v_3)) = +1 +1 -1 = +1$
$f(S(r_s(v_3))) = 0$
$L(y_3)$: $r_v(v_2)$ $r_v(v_1)$ $r_s(v_3)$ $r_s(v_2)$
$L_t(y_3)$: $r_v(v_{t1})$ $r_s(v_{t1})$

(b)


-1
+1
+1
CHECK
-1
+1
+1
-1
-1
v_{t2} : -1
$L(y_3)$
$L_t(y_3)$
Adding vertex:
$w(v_{t2}) = -1$
$\theta(v_{t2}) = \theta(r_s(v_3))$
$L(y_3)$: $r_v(v_2)$ $r_v(v_1)$ $r_s(v_3)$ $r_s(v_2)$
$L_t(y_3)$: $r_v(v_{t1})$ $\mathbf{r_s(v_{t2})}$ $\mathbf{r_v(v_{t2})}$ $r_s(v_{t1})$

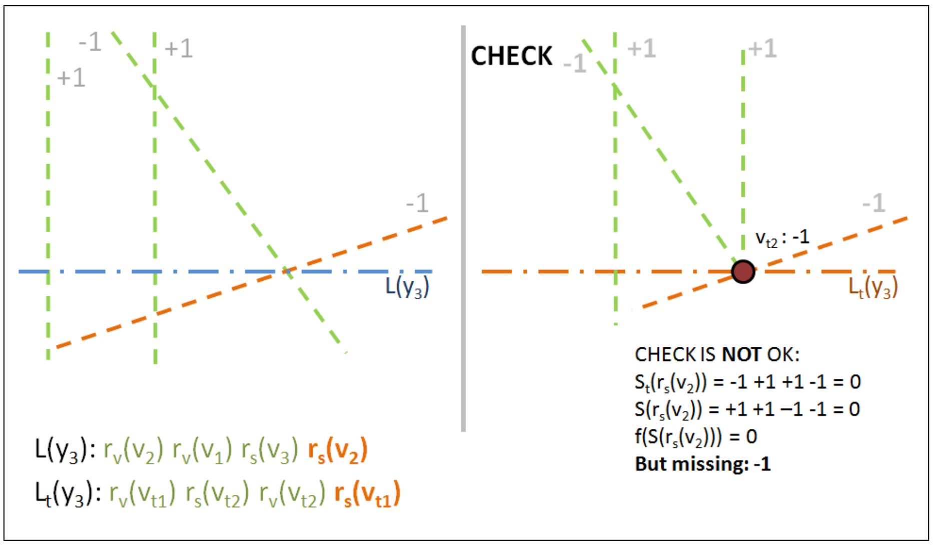
(c)

Figure 3.26: Example in step by step of *Scalar Transformation* - part 11 of 18.



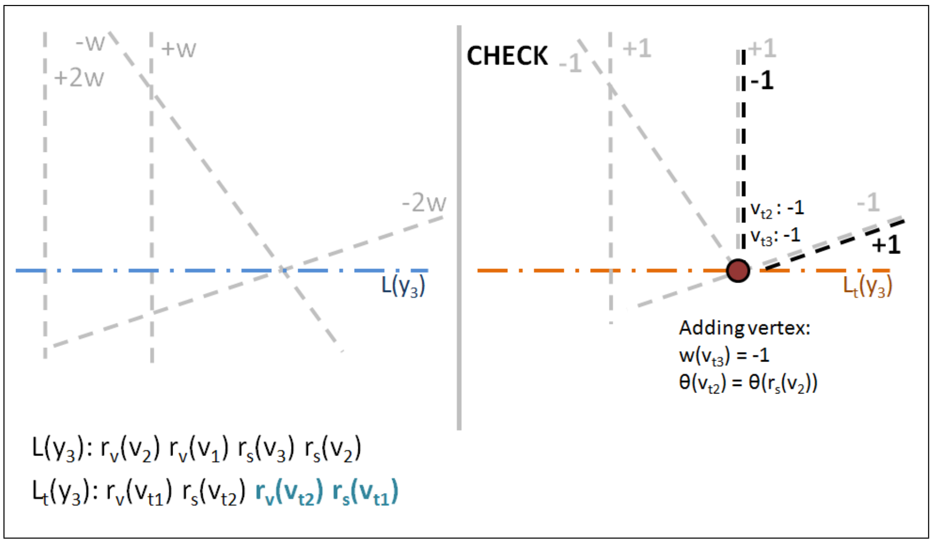
-1
+1
+1
-1
L(y3)
CHECK
-1
+1
+1
-1
vt2 : -1
Lt(y3)
CHECK IS NOT OK:
St(rs(vt3)) = -1 +1 +1 = +1
S(rs(vt3)) = +1 +1 –1 = +1
f(S(rs(vt3))) = 0
Missing: -1
L(y3): rv(v2) rv(v1) rs(v3) rs(v2)
Lt(y3): rv(vt1) rs(vt2) rv(vt2) rs(vt1)

(a)



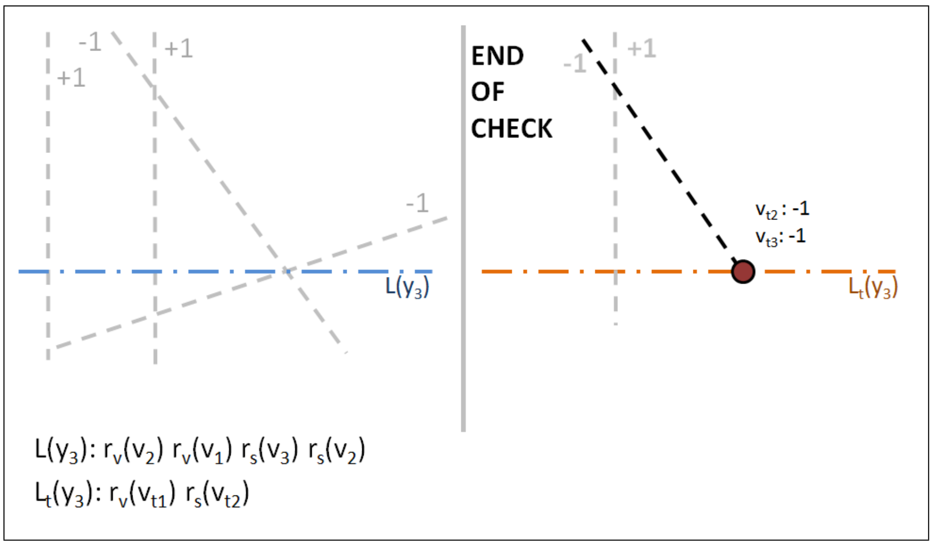
-1
+1
+1
-1
L(y3)
CHECK
-1
+1
+1
-1
vt2 : -1
Lt(y3)
CHECK IS NOT OK:
St(rs(v2)) = -1 +1 +1 -1 = 0
S(rs(v2)) = +1 +1 –1 -1 = 0
f(S(rs(v2))) = 0
But missing: -1
L(y3): rv(v2) rv(v1) rs(v3) rs(v2)
Lt(y3): rv(vt1) rs(vt2) rv(vt2) rs(vt1)

(b)

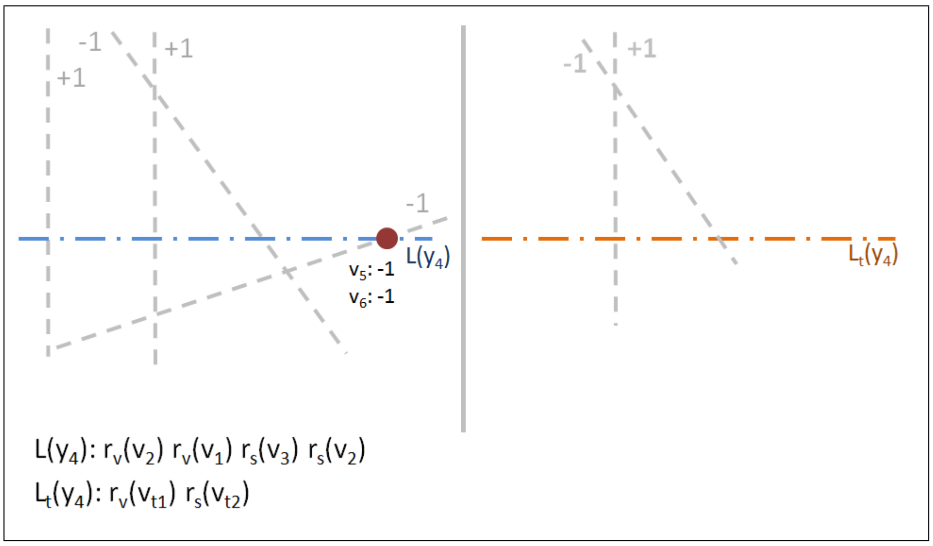


(c)

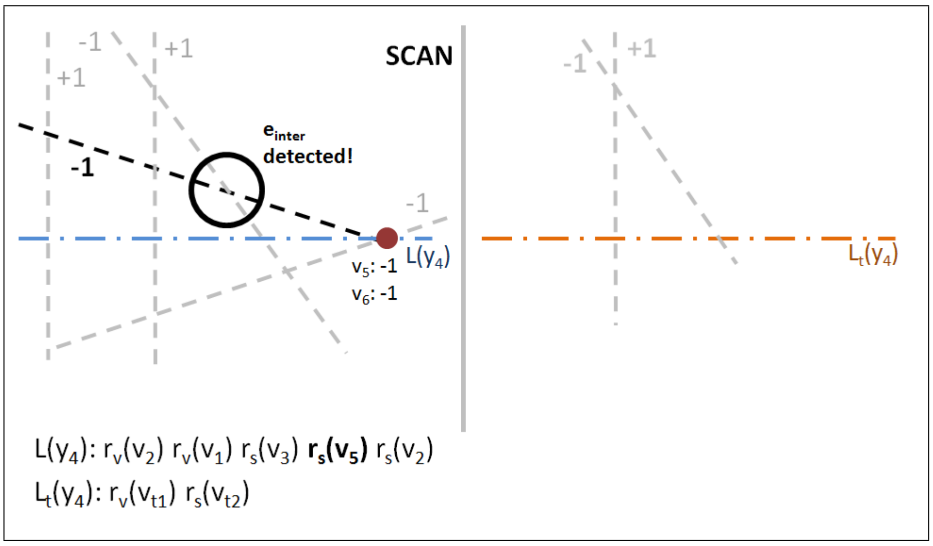
Figure 3.27: Example in step by step of *Scalar Transformation* - part 12 of 18.



(a)

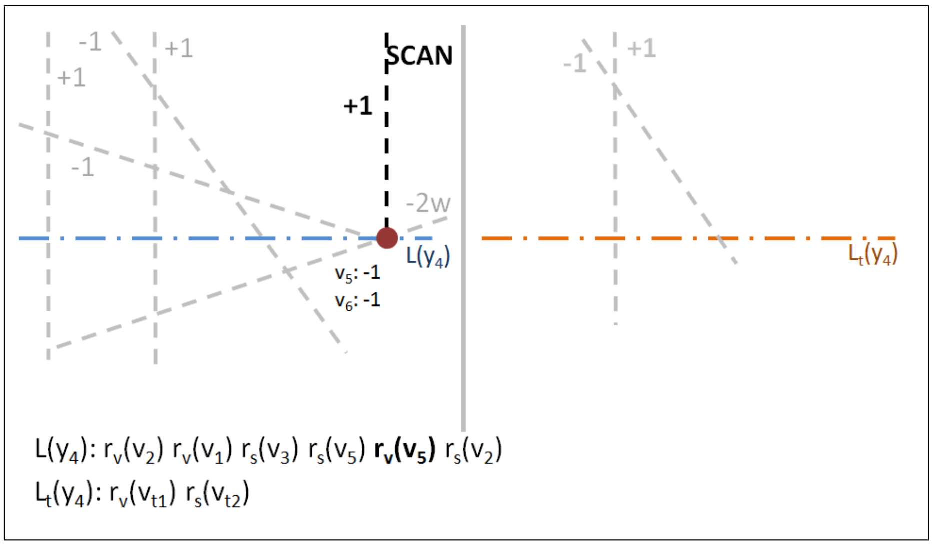


(b)

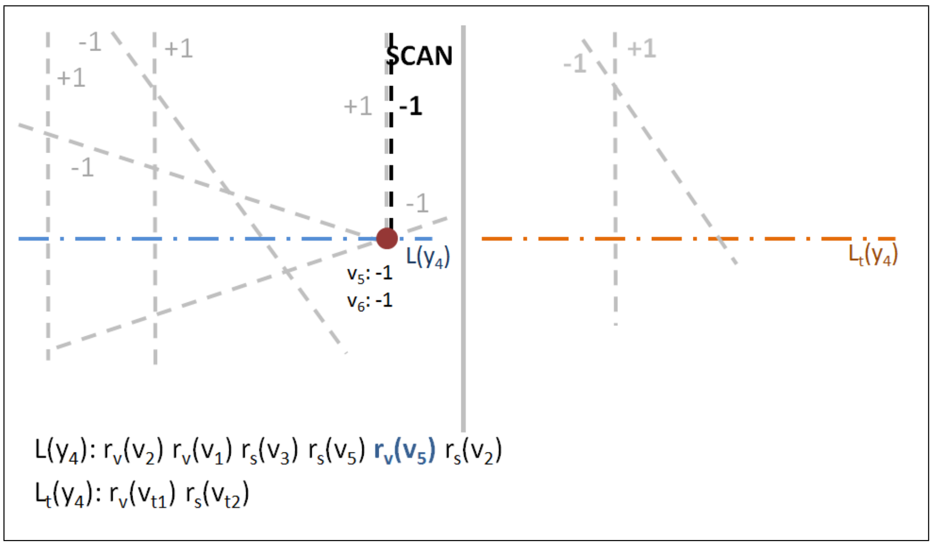


(c)

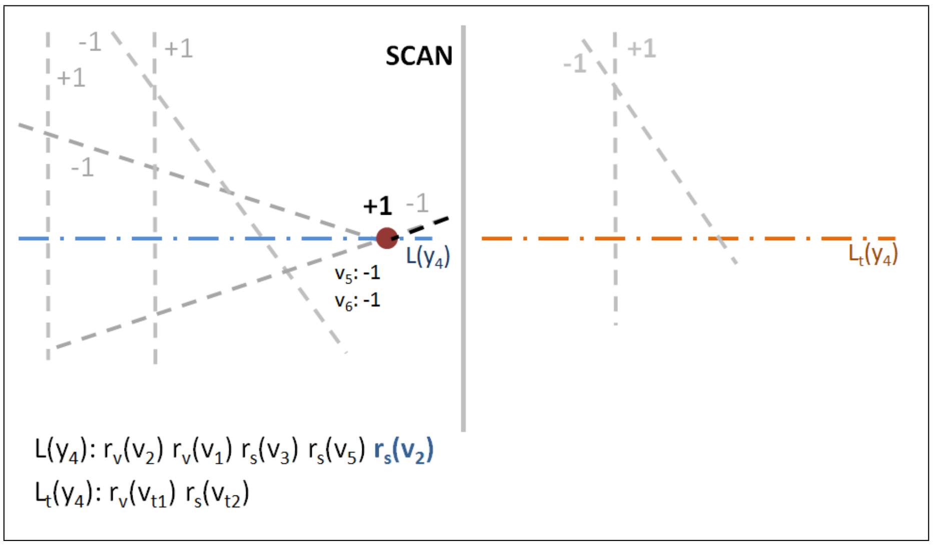
Figure 3.28: Example in step by step of *Scalar Transformation* - part 13 of 18.



(a)

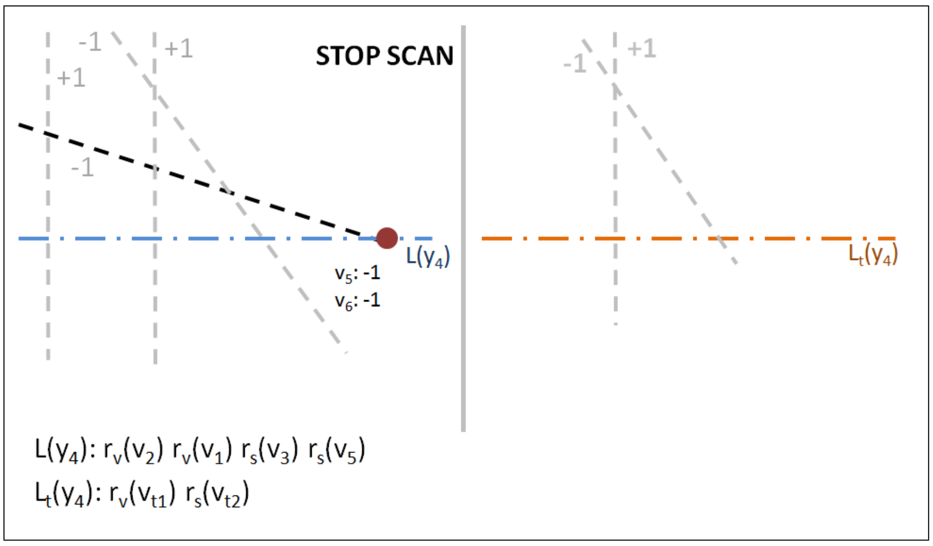


(b)

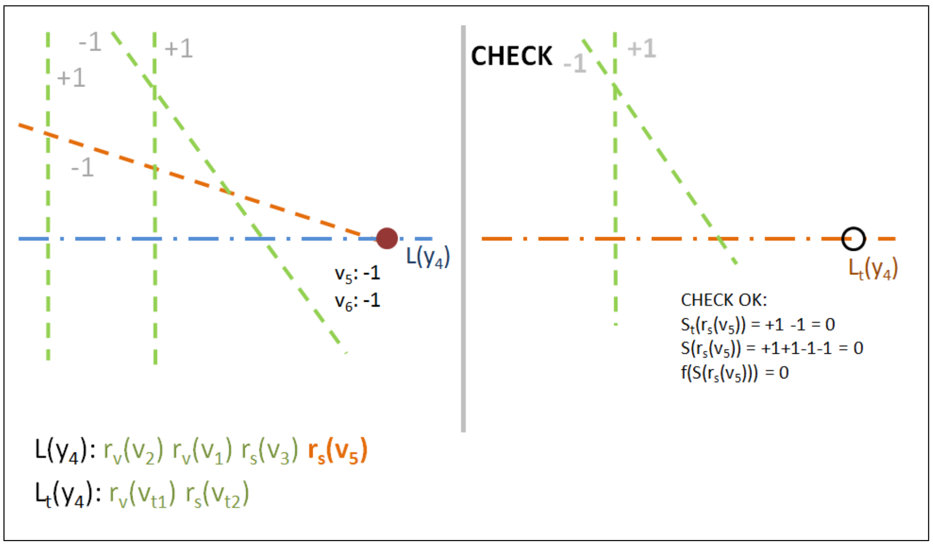


(c)

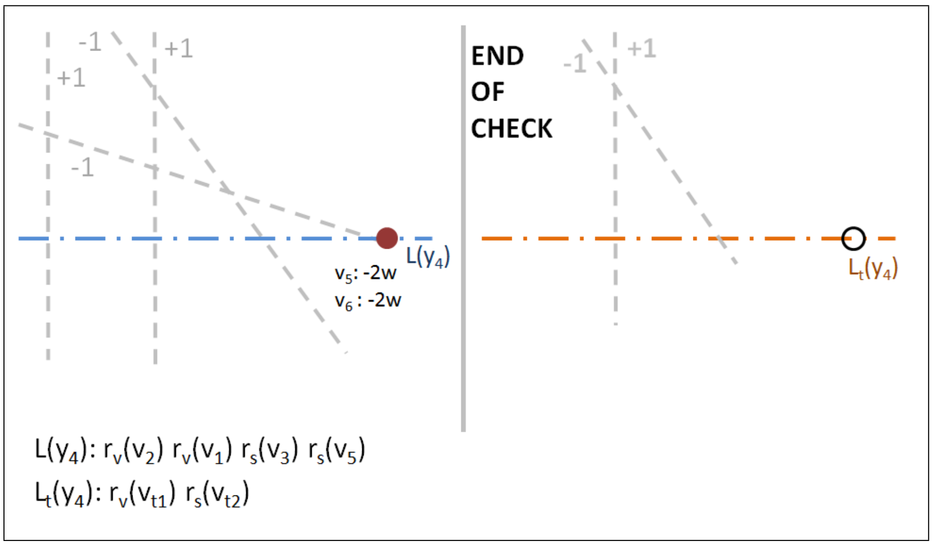
Figure 3.29: Example in step by step of *Scalar Transformation* - part 14 of 18.



(a)

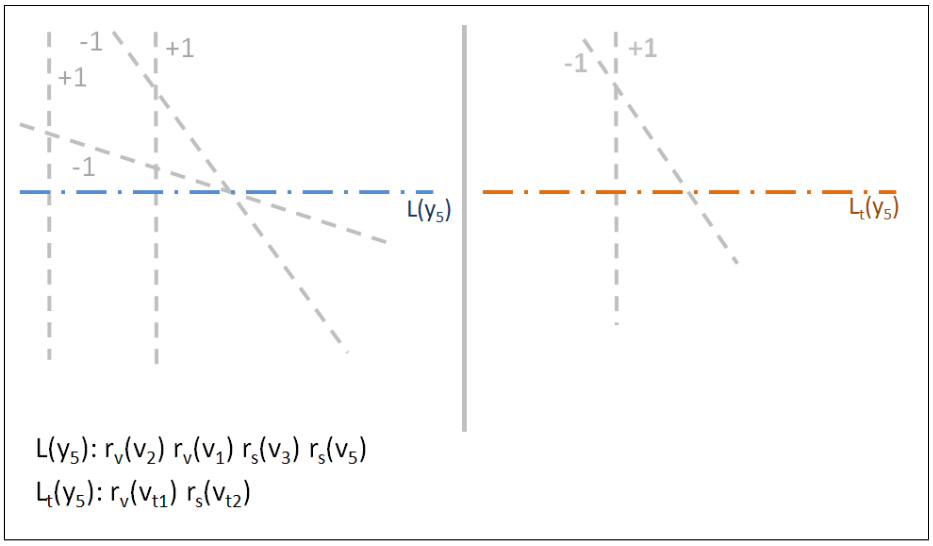


(b)

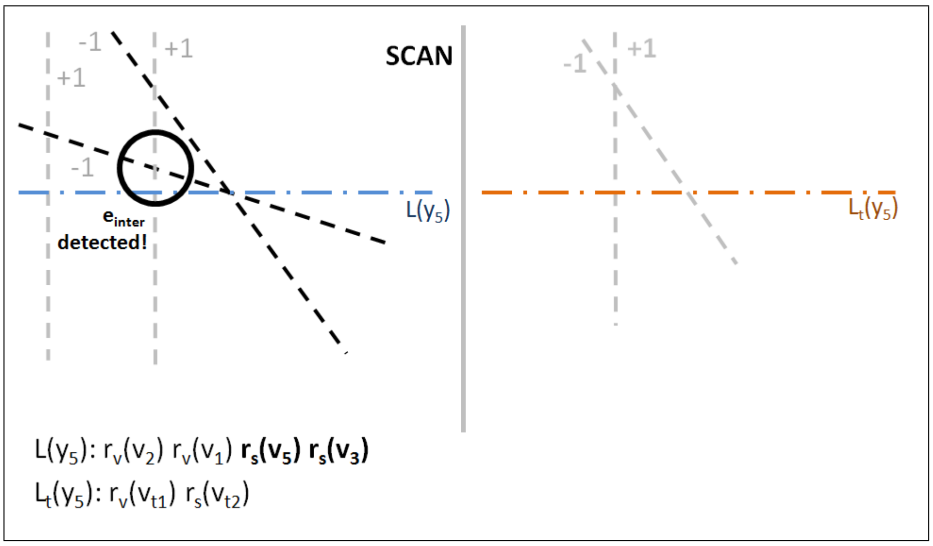


(c)

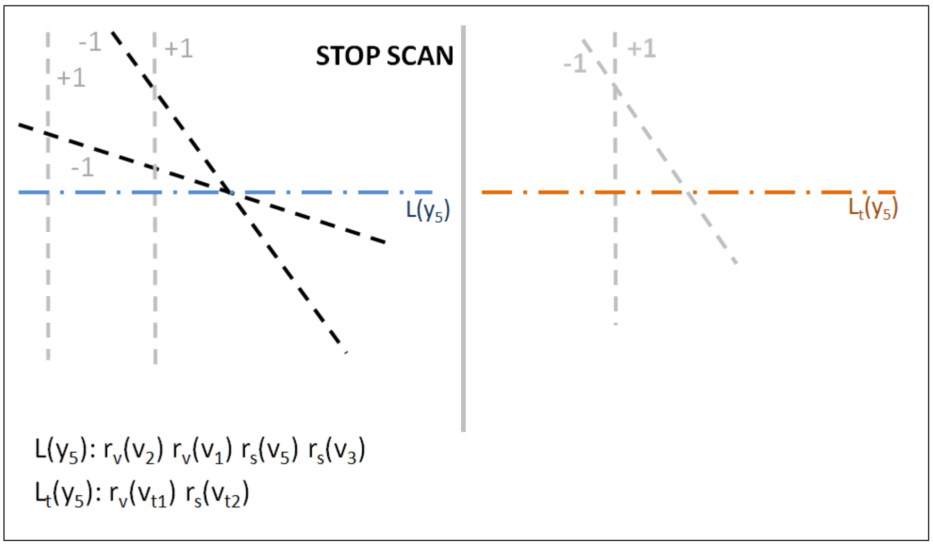
Figure 3.30: Example in step by step of *Scalar Transformation* - part 15 of 18.


-1
+1
+1
-1
$L(y_5)$
-1
+1
$L_t(y_5)$
$L(y_5)$: $r_v(v_2)$ $r_v(v_1)$ $r_s(v_3)$ $r_s(v_5)$
$L_t(y_5)$: $r_v(v_{t1})$ $r_s(v_{t2})$

(a)

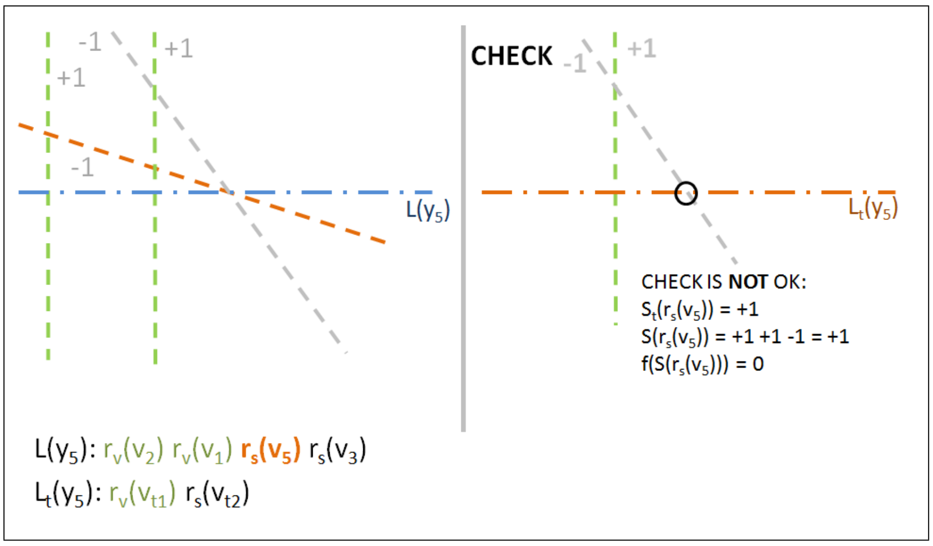

-1
+1
+1
SCAN
-1
e_{inter} detected!
$L(y_5)$
-1
+1
$L_t(y_5)$
$L(y_5)$: $r_v(v_2)$ $r_v(v_1)$ $\mathbf{r_s(v_5)}$ $\mathbf{r_s(v_3)}$
$L_t(y_5)$: $r_v(v_{t1})$ $r_s(v_{t2})$

(b)


-1
+1
+1
STOP SCAN
-1
$L(y_5)$
-1
+1
$L_t(y_5)$
$L(y_5)$: $r_v(v_2)$ $r_v(v_1)$ $r_s(v_5)$ $r_s(v_3)$
$L_t(y_5)$: $r_v(v_{t1})$ $r_s(v_{t2})$

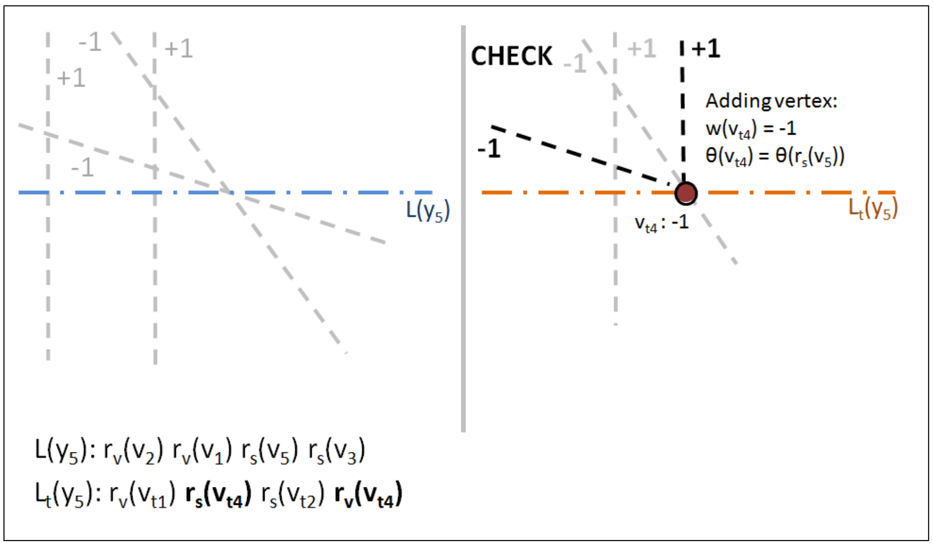
(c)

Figure 3.31: Example in step by step of *Scalar Transformation* - part 16 of 18.



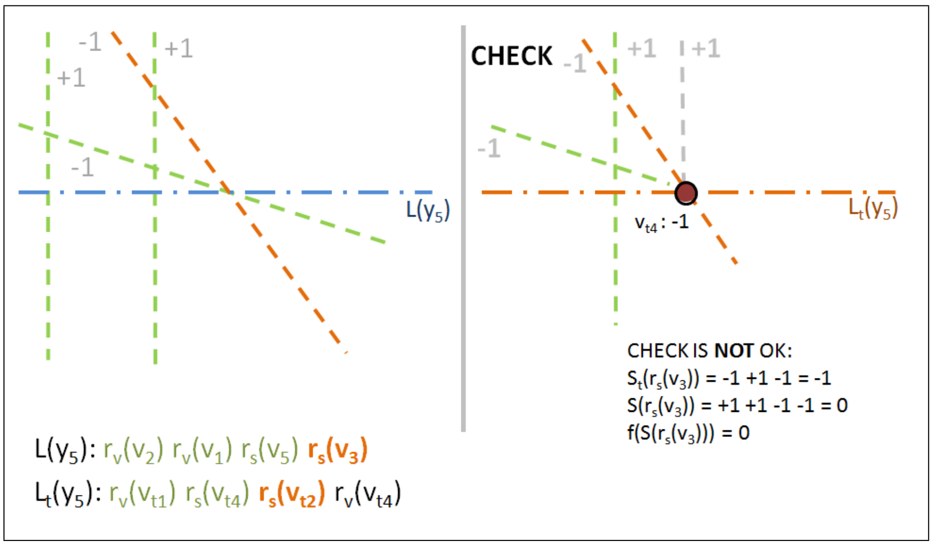
-1
+1
+1
-1
$L(y_5)$
CHECK
-1
+1
$L_t(y_5)$
CHECK IS **NOT** OK:
$S_t(r_s(v_5)) = +1$
$S(r_s(v_5)) = +1 +1 -1 = +1$
$f(S(r_s(v_5))) = 0$
$L(y_5)$: $r_v(v_2)$ $r_v(v_1)$ $\mathbf{r_s(v_5)}$ $r_s(v_3)$
$L_t(y_5)$: $r_v(v_{t1})$ $r_s(v_{t2})$

(a)



-1
+1
+1
-1
$L(y_5)$
CHECK
-1
+1
+1
-1
Adding vertex:
$w(v_{t4}) = -1$
$\theta(v_{t4}) = \theta(r_s(v_5))$
v_{t4} : -1
$L_t(y_5)$
$L(y_5)$: $r_v(v_2)$ $r_v(v_1)$ $r_s(v_5)$ $r_s(v_3)$
$L_t(y_5)$: $r_v(v_{t1})$ $\mathbf{r_s(v_{t4})}$ $r_s(v_{t2})$ $\mathbf{r_v(v_{t4})}$

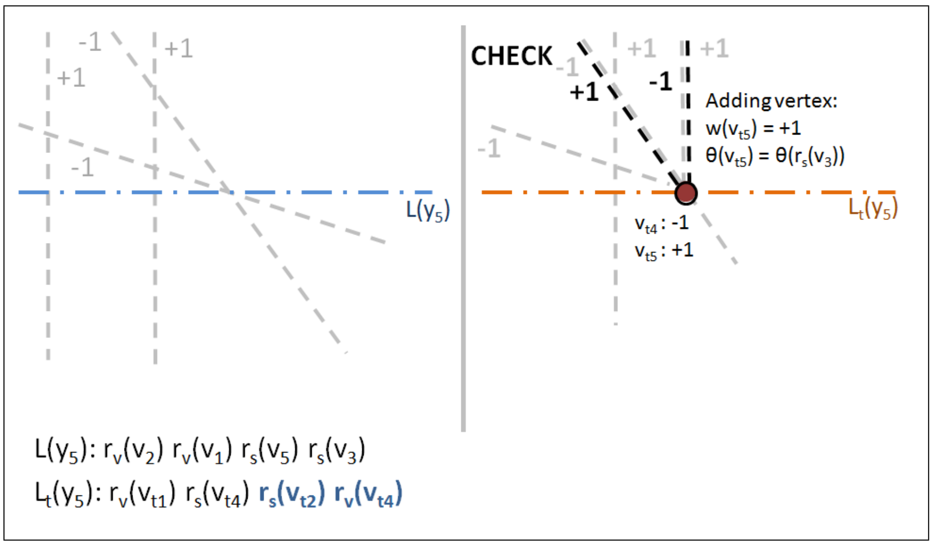
(b)



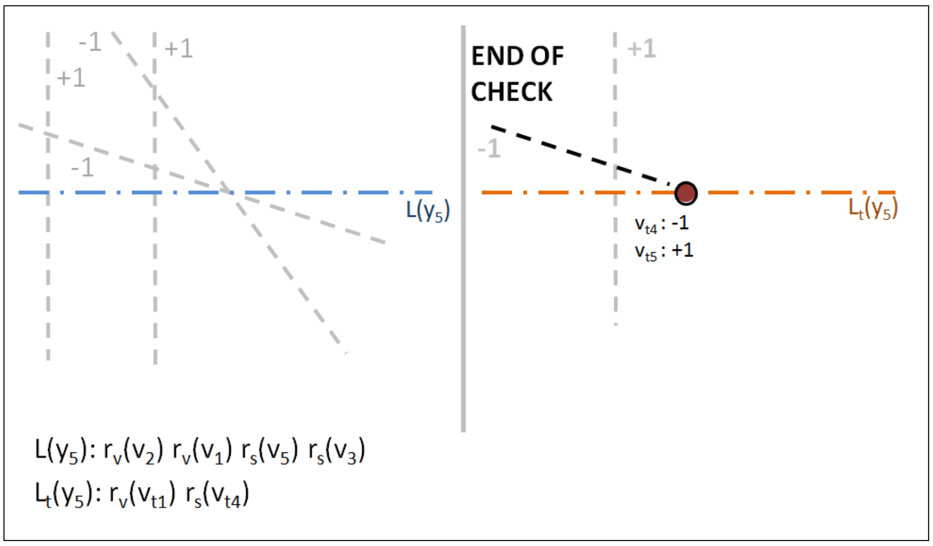
-1
+1
+1
-1
$L(y_5)$
CHECK
-1
+1
+1
-1
v_{t4} : -1
$L_t(y_5)$
CHECK IS **NOT** OK:
$S_t(r_s(v_3)) = -1 +1 -1 = -1$
$S(r_s(v_3)) = +1 +1 -1 -1 = 0$
$f(S(r_s(v_3))) = 0$
$L(y_5)$: $r_v(v_2)$ $r_v(v_1)$ $r_s(v_5)$ $\mathbf{r_s(v_3)}$
$L_t(y_5)$: $r_v(v_{t1})$ $r_s(v_{t4})$ $\mathbf{r_s(v_{t2})}$ $r_v(v_{t4})$

(c)

Figure 3.32: Example in step by step of *Scalar Transformation* - part 17 of 18.



(a)



(b)

Figure 3.33: Example in step by step of *Scalar Transformation* - part 18 of 18.

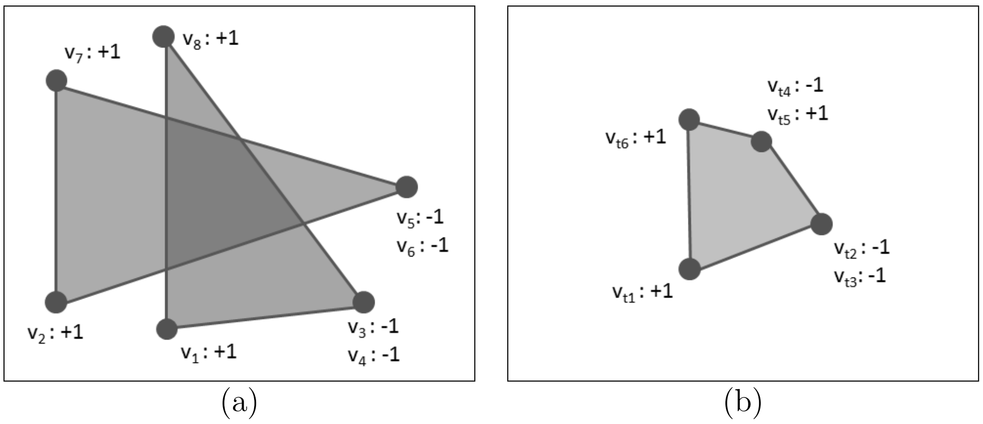


Figure 3.34: The result of *Scalar Transformation* that perform a intersect operation applying the function $f_\cap$ described above.

circulations is performed by scan and demands $O((I + n) \log n + n)$ to build the output with size n using *skip list* and $O(n^2 + n)$ without.

Draw: similarity to *Value At* complexity, it has $O((I + n) \log n + m)$ using *skip list* and $O(n^2 + m)$ otherwise, where n is the size of WVC and m the number of created trapezoids.

Scalar Transformation: also has its complexity given by scan operations and the size of output: $O((I + n) \log n + m \log m)$ or $O(n^2 + m \log m)$ if applying or not the *skip list*, consider n the size of WVC and m the size of transformed WVC.

The space complexity using all these structures is $O(n)$.

An import decision in a implemetation is the use of *skip list* structure, in spite of seem reasonable to assume that the cost of maintaining this or similar structure is not worthwhile if $||L(y)||$ stays small. Second Pugh [27], the skip lists are a data structure that can be used in place of balanced trees, where are used probabilistic balancing rather than strictly enforced balancing and as a result the algorithms for insertion and deletion in skip lists are much simpler and significantly faster than equivalent algorithms for balanced trees. The skip list guarantees that pertubations over $L(y)$ can be summed visiting $O(n \log n)$ rays.

Certainly the critical operation is the *Scalar Transformation*, because it perfoms scan and is necessary to execute Map Overlay. A classical pipeline of WVC is perfomed with (1) *Convert* two spatial data to WVC, (2) *Sum* this two WVC, (3) *Scalar Transformation* for Map Overlay and maybe (4) *Convert* the result to circulations of vertices. Thus we have using skip list: $O(n \log n + n + (n + I) \log n + m \log m + (n + I) \log n + n)$ or simply $O((n + I) \log n + m \log m)$. If it is assumed that $n \equiv m$, then $O((n + I) \log n)$ summarizes the time complexity. Without the use of the skip list, the complexities are $O(n^2 + m \log m)$ or $O(n^2)$.

Chapter 4

Implementation, Tests and Results

A straightforward implementation of the *Weighted Vertex Collection* data structure was written in the *C++* language and compiled using the Gnu Compiler Collection 4.2.6 (g++). It is capable of computing the basic operations, properties and the *line sweep* process described in Chapter 3, namely, the *Scan*, *Add*, *Value At*, *Draw*, *Scalar Transformation* and *Convert* operations. Main geometric entities were implemented using the Computational Geometry Algorithms Library (CGAL) 4.2 [28], whereas the ShapeLib 1.2.9 [29] library provided support for reading and writing cartographic data in the ubiquitous *Shapefile* format.

The objectives of this implementation are (1) to demonstrate that the *WVC* can be applied to the processing of *Map Overlay* operations, and (2) that the processing time of *WVC* can be competitive with mainstream GIS software.

The implementation presented in this work, processes the Value at operation or similar sequentially, since it seems reasonable to assume that the cost of maintaining a *skip list* or similar structure is not worthwhile if $||L(y_i)||$ stays small. See a discussion in section 3.5.

In this implementation the weighted vertices are ajusted a grid with 2^n positions to robustness of floating-point operations, where the n is large enough to the positional accuracy of spatial data. The angle θ is replaced by a direction vector to facilitate geometric operations as ray intersect and ray orientation.

The experiments described below aim to compare the efficiency of the a *WVC* implementation against those of two mainstream GIS systems, namely ArcGIS for Desktop 10.2.1 and QGIS 2.2 Valmiera. Timings were obtained for the processing of *Map Overlay* operations using four real spatial datasets in *Shapefile* format. Some actions were taken to ensure a fair comparison of processing times: (a) the first run is discarded and the ten subsequent ones are averaged; (b) the *Map Overlay* operations are called by scripts in Python packages: *arcpy* and *processing*, respectively, for the ArcGIS and QGIS systems; (c) timings for the *WVC* implementations comprise only the execution of *Scalar Transformation* operations; (d) QGIS performs

ID	Data	Poly	Vertices	I_{poly}	I_{point}	I_{line}
1	District	3	838	4	4	0
	Forest	9	3,831			
2	Municipals	645	83,782	942	1,485	0
	Soil Potential	45	4,758			
3	Land Use	1,394	134,682	1,913	16,890	13,311
	Vegetation	1,473	139,054			
4	Municipals	5,564	1,195,910	10,007	19,327	0
	Soil Potential	409	49,073			

Table 4.1: Characteristics of datasets used in the experiments. *Poly* is the number of polygons, *Vertices* is the total number of polygon vertices, I_{poly}, I_{point} and I_{line} are the number of 2D, 1D and 0D geometries returned in a intersection operation between data.

operations without access to files, ie it reads and writes only in memory; (e) since it was not possible to measure the execution time of the ArcGIS for Desktop (t'_{arc}) without including the time to access input files, the processing time required by a test program written in C++ using ShapeLib to load files in memory was measured (t_{shplib}) subtracted from the total time, i.e., $t_{arc} = t'_{arc} - 2t_{shplib}$.

Details about the spatial datasets used in the experiments are shown in Table 4. The first dataset is the forest cover of Parque Nacional da Tijuca and boundaries of three districts in Tijuca administrative region, that dataset was produced by Instituto Pereira Passos (an agency of Rio de Janeiro's municipal government) at 1:100,000 scale. The second and fourth datasets were produced by Instituto Brasileiro de Geografia e Estatística (IBGE), where municipal boundaries of 2007 are at 1:250,000 scale and soil potential for agriculture at 1:1,000,000. The second dataset is a subset of fourth. The third dataset is a mapping at 1:50,000 scale of vegetation cover and land use in the Teresópolis municipality. This dataset of 1996 was produced by Sistema Labgis of Universidade do Estado do Rio de Janeiro. Figure 4.1 shows an overview of the dataset geometries.

Map Overlay operations were performed using the three softwares and execution times are shown in Table 4. Column *Diff %* shows the percentage difference of processing time between *WVC* and the faster GIS software applying equation below. All times were measured on a laptop computer equipped with a Pentium Dual-Core 2.2 GHz 64-bits processor and 4 GB of RAM, running 64-bit Windows 7. All softwares were compiled for 32-bit architectures.

$$Dif\% = \frac{t_{wvc} * 100}{min(t_{arc}, t_{qgis})}$$

Dataset	Test	Operation	t_{wvc}	t_{arc}	t_{qgis}	Dif %
1	1	UNION	0.074	0.149	0.122	60.66%
	2	INTER	0.071	0.200	0.120	59.1%
	3	XOR	0.090	0.226	0.116	77.6 %
2	4	UNION	1.206	2.094	21.085	57.6%
	5	INTER	1.128	1.851	16.562	60.9%
	6	XOR	0.915	1.186	**	77.2%
	7	CLIP	0.876	0.865	2.66	101.3%
3	8	UNION	1.948	2.421	4.813	80.5%
	9	SELECT	1.591	1.092	3.875	145.7%
	10	DISSOLVE	1.561	1.003	59.115	155.6%
4	11	UNION	17.707	23.476	>600	75.43%

Table 4.2: Executions time in seconds with *WVC*, ArcGIS for Desktop and QGIS.

For these tests, the *INTER* operation computes a geometric intersection of two polygon collections A and B, recording in the output $A \cap B$ geometries, i.e., the intersection of each polygon of one collection with each polygon of the other. The *XOR* operation - also called of symmetric difference - is similar and computes $(A - B) \cup (B - A)$. The *UNION* operation produces a similar output, i.e., it computes $A - B$, $B - A$ and $A \cap B$, but keeps the geometries separate. The *CLIP* operation cuts the geometries of A using B polygons as a cutter shape, while *DISSOLVE* aggregates polygons of a collection based on equality of specified attributes. And last, *SELECT* operation performs an *INTER* operation on a subset of polygons also based on attributes. The *XOR* operation failed in test 6 using QGIS, because the software crashes when executed.

4.1 Comments about processing the Map Overlay operations with WVCs

As a complement to the examples shown in Section 3.2, this section discusses some details of the implementation steps used to perform the map overlay operations with *WVCs*.

The first step is to perform a *Convert* operation on the datasets used in the experiments. When converting the collection of circulations of vertices to *WVC* representation, the spatial data of vegetation, soil potential and land use were associated with a different w value for each map class. The other spatial data were processed so as to associate each polygon with a different w value. The *Convert* operation reads data in Shapefile format, creates weighted vertices, sorts the collection in scan order and creates a canonical representation. It processed the datasets 1, 2, 3 and 4 respectively in 0.04, 1.26, 5.44 and 19.22 seconds. Each Shapefile generates a *WVC*, i.e., two collections C_1 and C_2 for each *Map Overlay* operation.

The next two steps consist of performing a *Scalar Multiplication* operation using a constant α, followed by an *Add* operation. In other words, a WVC representing $C_1 + \alpha C_2$. Constant α must be large enough to allow bitwise operations to identify the regions of interest in the resulting scalar field. For instance, the Shapefile representing the land use dataset has 25 classes and was converted to C_1. On the other hand, the vegetation dataset has 15 classes and was converted to C_2. Therefore, by computing $C_3 = C_1 + 100C_2$ allows us to conclude that, say, a scalar value 1025 in C_3 corresponds to a region where the vegetation class is 10 and the land use class is 25. Similarly, a scalar value 900 identifies regions where vegetation class is 9 and that do not lie in any land use class.

With the new collection C_3 built, a *Scalar Transformation* is computed, where the choice of the function depends on the *Map Overlay* operation to be executed. This transformation generates a new collection, which can then be converted back to a circulation of vertices if so desired. Below are the functions used on the experiments described in this chapter.

$$Union : f_{\cup}(x) = x$$

$$Intersection : f_{\cap}(x) = \begin{cases} x & \text{if } (x \geq \alpha) \wedge (x \bmod \alpha \neq 0) \\ 0 & \text{otherwise} \end{cases}$$

$$XOR : f_{XOR}(x) = \begin{cases} x & \text{if } (x < \alpha) \vee (x \bmod \alpha = 0) \\ 0 & \text{otherwise} \end{cases}$$

$$Clip : f_{Clip}(x) = \begin{cases} x \bmod \alpha & \text{if } (x \geq \alpha) \\ 0 & \text{otherwise} \end{cases}$$

$$Dissolve : f_{Diss}(x) = \begin{cases} 1 & \text{if } (x \in V) \\ 0 & \text{otherwise} \end{cases}$$

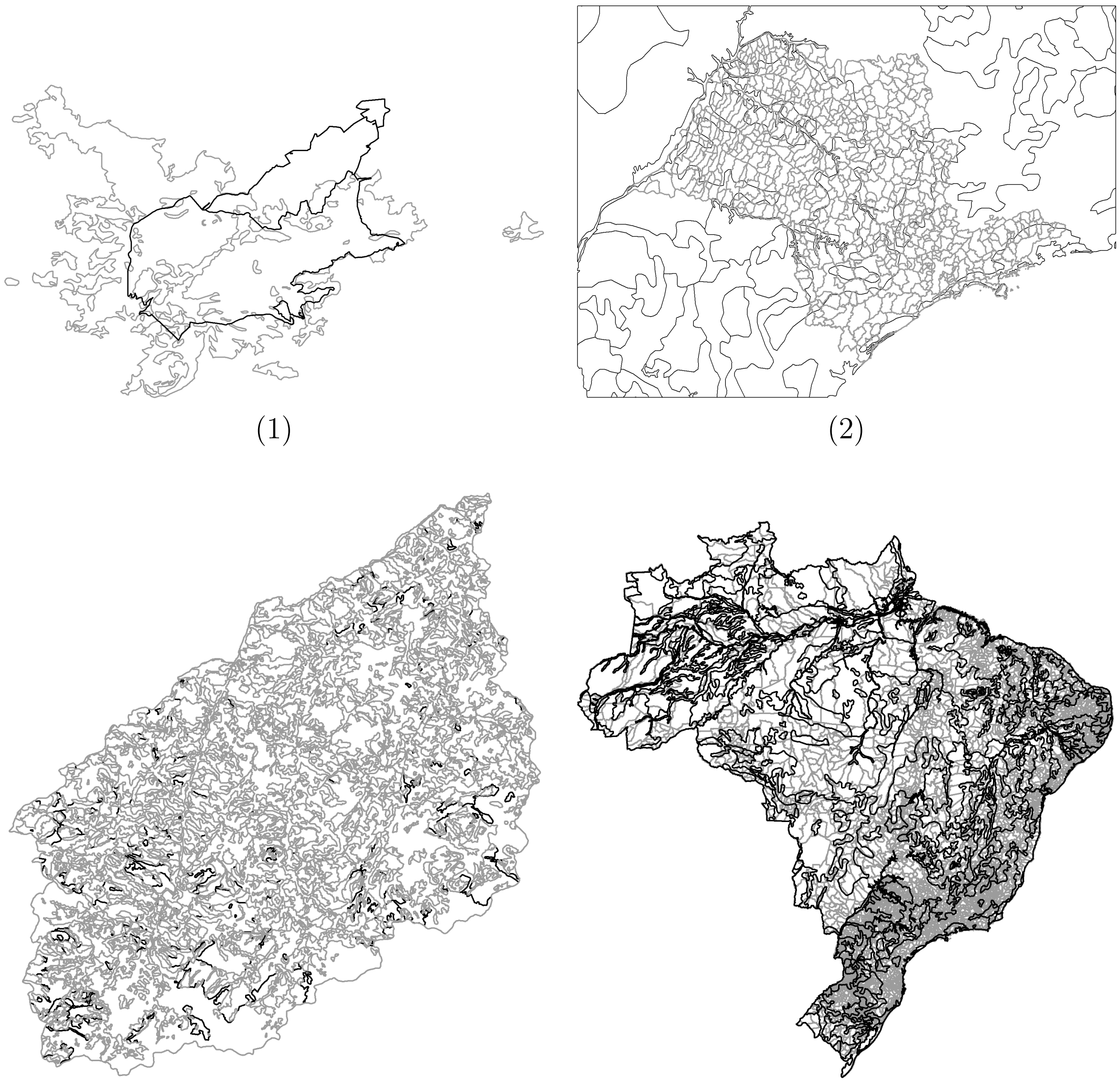


Figure 4.1: Overview of dataset geometries with black and gray outlines.

Test	Operation	$\|\|WVC_{in}\|\|$	$\|\|WVC_{out}\|\|$	$\|\|L(y)\|\|$	$\|\|SHP_{in}\|\|$	$\|\|SHP_{out}\|\|$
1	UNION	8,748	8,748	22.42	4,669	6,656
2	INTER	8,748	2,912	6.12	4,669	1,836
3	XOR	8,748	9,021	22.06	4,669	4,820
4	UNION	97,709	97,709	35.70	88,540	96,212
5	INTER	97,709	131,731	71.24	88,540	81,221
6	XOR	97,709	22,495	28.31	88,540	14,991
7	CLIP	97,709	10,432	11,98	88,540	6,292
8	UNION	141,985	141,985	80.56	273,054	145,243
9	SELECT	141,985	72,846	53.36	75,195	37,134
10	DISSOLVE	136,283	82,824	29,20	44,439	42,191
11	UNION	2,045,487	2,198,936	91.45	1,244,983	1,319,852

Table 4.3: The sizes of converted (WVC_{in}) and transformed (WVC_{out}) *Weighted Vertex Collections*; $||L(y)||$ average size scanline onto stop events; and numbers of vertices in Shapefiles before ($||SHP_{in}||$) and after ($||SHP_{out}||$) *Map Overlay* operations in GIS software.

4.2 Discussion of results

The *WVC* implementation had a better processing time in 8 of 11 tests. As discussed in Chapter 2, the mainstream GIS softwares perform *Map Overlay* processing circulations of vertices at least two steps [20], namely: filter and refinement. In the filter step are used structures as MBB (minimum bounding box) and spatial indexes, already the refinement step computes the resultant geometries with tradicionals algorithms of computacional geometry, see a compilation in [22]. Even without has a filter step, the *WVC* shows a competitive processing time applying a dimension-reduction by *plane sweep* paradigm.

It is noteworthy that one edge of circulation of vertices adds two weighted vertices in a collection, while each weighted vertex generates two rays. In Shapefile format the first and last vertices of circulations are coincidents, so we have $n_v = n_c + n_e$, where n_v, n_c and n_e are respectively number of vertices, circulations and edges. Observing the worst case simplistic, size of a weighted vertex collection (n_{wv}) and the number of rays (n_r) into a scan process may be $n_{vc} = 2n_e$ and $n_r = 4n_e$. However the Table 4.1 shows a smaller relationship between $||WVC_{in}||$, $||L(y)||$ and $||SHP_{in}||$ because to two reasons. The first is the canonical form of *WVC*, that remove coincidents vertices with same angle θ. See the Figure 4.1 and Table 4.1 and 4, a lowest ratio ${}^{||WVC_{in}||}/_{||SHP_{in}||}$ is showed in spatial data with more adjacent polygons - municipalities as example - and the dataset 3 with thousands of lines intersection.

The second reason is the behavior of $L(y)$ during the scan process, due to overlapping of rays. When comparing the size of $L(y)$ at a certain y_i ($||L(y_i)||$) to the number of circulations of vertices that crosses at same y_i ($||CV_{cross}(y_i)||$), then $||L(y_i)|| \leq ||CV_{cross}(y_i)||$ or also $||L(y_i)|| = ||CV_{cross}(y_i)|| - D(y_i)$, where $D(y_i)$ is a number of circulations that crosses y_i at same point.

The small value of $||L(y)||$ shown in Table 4.1 corroborates the choice of not using a data structure $O(nlogn)$ to determine the value of the scalar field at a point onto $L(y)$, as discussed earlier in this chapter.

The *WVC* implementation had a worse time processing than ArcGIS for Desktop on tests 8 and 9, where there was selections by attributes in GIS softwares. For instance, to perform the intersection on test 8 were filtered the classes "casual livestock" and "meadow" of land use and vegetation data. Obviously the selection reduced the number of input polygons in *Map Overlay* operation, then what reduced the processing time - see Table 4.1 and 4. The times to GIS software performs these selections by attributes were summed in your total processing time, nevertheless the *WVC* implementation had a worse time processing because the weighted vertex collection had no similar input reduction. Within this *WVC* proposal, a vertex weight w does not linked a feature, polygon or map class, it reflects a change in the scalar field. Thus, it is necessary performs a *Scalar Transformation* to reconstruct and filter w values, and it was done in tests. Was considered that the execution time tie in test 7.

Chapter 5

Conclusion

The purpose of this work comes to join with the *Weighted Edge Collection* [25] and the *Vertex Representation* [24] to perform *Map Overlay* operations using *plane sweep* paradigm and scalar field concept. Properties, data structures and operations of *Weighted Vertex Collection* were defined and a straightforward implementation using C++ language was presented, where the results demonstrate a competitive execution time when compared to mainstream GIS software: ArcGIS for Desktop and QGIS.

As future works more tests and evaluations need to be performed on the WVC implementation to analyze its behavior on other spatial data and operations, in addition the proposal of WVC may be increased by optimizing structures, filter steps or changes in the algorithm. Another interesting way is to extend the WVC to include the possibility of filtering by attribute, to process geometries of lines or points, and adoption of parallel algorithms for plane sweep - some papers as [30, 31] show proposals.

Aside from works as [8, 9, 15, 22] that apply plane sweep to processing Map Overlay, the present paper uses scalar field to simplify the algorithms and to introduce new way to approach. Definitely the WVC is consistent proposal, while is expected that with some advance, it becomes an interesting alternative to processing of Map Overlay with Big Spatial Data, unindexed data or other scenarios where new studies indicate potentialities.

Bibliography

[1] CASANOVA, M., CÂMARA, G., DAVIS, C., et al. *Bancos de Dados Geográficos*. São José dos Campos, MundoGEO, 2005.

[2] LAURINI, R., THOMPSON, D. *Fundamentals of spatial information systems*. A.P.I.C. studies in data processing. Academic Press, 1992.

[3] ORENSTEIN, J. A. "Spatial query processing in an object-oriented database system". In: *Proceedings of the 1986 ACM SIGMOD international conference on Management of data*, SIGMOD '86, New York, NY, USA, 1986. ACM.

[4] RAMOS, J. A. S., ESPERANÇA, C., CLUA, E. W. G. "A progressive vector map browser for the web", *Journal of the Brazilian Computer Society*, v. 15, pp. 35 – 48, 06 2009. ISSN: 0104-6500. Disponível em: <`http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0104-65002009000200004&nrm=iso`>.

[5] BERTOLOTTO, M., EGENHOFER, M. J. "Progressive Transmission of Vector Map Data over the World Wide Web", *Geoinformatica*, v. 5, n. 4, pp. 345–373, dez. 2001. ISSN: 1384-6175. doi: 10.1023/A:1012745819426. Disponível em: <`http://dx.doi.org/10.1023/A:1012745819426`>.

[6] SIQUEIRA, T. L. A.-S. L., CIFERRI, C. D. D. A., TIMES, V. A. C. A., et al. "The impact of spatial data redundancy on SOLAP query performance", *Journal of the Brazilian Computer Society*, v. 15, pp. 19 – 34, 06 2009. ISSN: 0104-6500. Disponível em: <`http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0104-65002009000200003&nrm=iso`>.

[7] MEIJERS, M. "Cache-friendly progressive data streaming with variable-scale data structures", 2011.

[8] BRINKHOFF, T., KRIEGEL, H.-P., SCHNEIDER, R., et al. "Multi-Step Processing of Spatial Joins". In: *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1994.

[9] JACOX, E. H., SAMET, H. “Spatial join techniques”, *ACM Trans. Database Syst.*, 2007.

[10] SHEKHAR, S., GUNTURI, V., EVANS, M. R., et al. “Spatial big-data challenges intersecting mobility and cloud computing”. In: *Proceedings of the Eleventh ACM International Workshop on Data Engineering for Wireless and Mobile Access*, MobiDE ’12, pp. 1–6, New York, NY, USA, 2012. ACM. ISBN: 978-1-4503-1442-8. doi: 10.1145/2258056.2258058. Disponível em: <`http://doi.acm.org/10.1145/2258056.2258058`>.

[11] CROMPVOETS, J., RAJABIFARD, A., VAN LOENEN, B., et al. *A Multi-View Framework to Assess Spatial Data Infrastructures.* Printed by Digital Print Centre, The University of Melbourne, Australia, 2008.

[12] BECKER, L., GIESEN, A., HINRICHS, K. H., et al. “Algorithms for Performing Polygonal Map Overlay and Spatial Join on Massive Data Sets”. In: *Advances in Spatial Databases-6th International Symposium, SSD ’99, Hong Kong.* Springer-Verlag, 1999.

[13] BIUK-AGHAI, R. “A mobile GIS application to heavily resource-constrained devices”, *Geo-spatial Information Science*, v. 7, n. 1, pp. 50–57, 2004. ISSN: 1009-5020. doi: 10.1007/BF02826676. Disponível em: <`http://dx.doi.org/10.1007/BF02826676`>.

[14] GÜTING, R. H. “An Introduction to Spatial Database Systems”, *VLDB J.*, v. 3, n. 4, 1994.

[15] KRIEGEL, H.-P., BRINKHOFF, T., SCHNEIDER, R. “An Efficient Map Overlay Algorithm Based on Spatial Access Methods and Computational Geometry”. In: *Proceedings of the International Workshop on DBMS’s for Geographic Applications.* Springer Verlag, 1991.

[16] KRIEGEL, H., BRINKHOOF, T., SCHNEIDER, R. “Efficient Spatial Query Processing in Geographic Database System”, *Data Enginnering Bulletin*, v. 16, pp. 10 – 15, 1993.

[17] ZHU, H., SU, J., IBARRA, O. H. “Toward Spatial Joins for Polygons”. In: *Proceedings of the 12th International Conference on Scientific and Statistical Database Management*, SSDBM ’00, Washington, DC, USA, 2000. IEEE Computer Society.

[18] CÂMARA, G., DAVIS, CLODOVEU E MONTEIRO, M. V. M. *Introdução à Ciência da Geoinformação.* INPE, São José dos Campos, SP, 2001.

[19] BRINKHOFF, T., HORN, H., KRIEGEL, H.-P., et al. "A storage and access architecture for efficient query processing in spatial database systems". In: *Advances in Spatial Databases*, pp. 357–376. Springer, 1993.

[20] AZEVEDO, L. G., GÜTING, R. H., RODRIGUES, R. B., et al. "Filtering with raster signatures". In: *Proceedings of the 14th annual ACM international symposium on Advances in geographic information systems*, GIS '06, New York, NY, USA, 2006. ACM.

[21] SAMET, H. *Foundations of Multidimensional and Metric Data Structures.* The Morgan Kaufmann Series in Computer Graphics, 2005.

[22] DE BERG, M., VAN KREVELD, M., OVERMARS, M., et al. *Computational geometry: algorithms and applications.* Springer, 2008.

[23] BECKMANN, N., KRIEGEL, H.-P., SCHNEIDER, R., et al. "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles", *SIGMOD Rec.*, v. 19, n. 2, pp. 322–331, maio 1990. ISSN: 0163-5808. doi: 10.1145/93605.98741. Disponível em: <`http://doi.acm.org/10.1145/93605.98741`>.

[24] ESPERANÇA, C., SAMET, H. "Vertex representations and their applications in computer graphics." *The Visual Computer*, 1998.

[25] XAVIER, A. P. T. T. *Junção Espacial de Regiões Poligonais Usando Campos Escalares.* Tese de Mestrado, Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa em Engenharia da Universidade do Estado do Rio de Janeiro, Rio de Janeiro, Brasil, 2013.

[26] BORGES, K. A., DAVIS, C. A., LAENDER, A. H. "OMT-G: An Object-Oriented Data Model for Geographic Applications", *GeoInformatica*, v. 5, n. 3, pp. 221–260, 2001. ISSN: 1384-6175. doi: 10.1023/A:1011482030093. Disponível em: <`http://dx.doi.org/10.1023/A%3A1011482030093`>.

[27] PUGH, W. "Skip Lists: A Probabilistic Alternative to Balanced Trees", *Commun. ACM*, v. 33, n. 6, pp. 668–676, jun. 1990. ISSN: 0001-0782. doi: 10.1145/78973.78977. Disponível em: <`http://doi.acm.org/10.1145/78973.78977`>.

[28] CGAL EDITORIAL BOARD. "CGAL - Computational Geometry Algorithms Library". 2014 June. Disponível em: <`https://www.cgal.org/`>.

[29] FRANK WARMERDAM. "Shapefile C Library". 2014 June. Disponível em: <`http://shapelib.maptools.org/`>.

[30] ATALLAH, M. J., GOODRICH, M. T. “Efficient Plane Sweeping in Parallel”. In: *Proceedings of the Second Annual Symposium on Computational Geometry*, SCG ’86, pp. 216–225, New York, NY, USA, 1986. ACM. ISBN: 0-89791-194-6. doi: 10.1145/10515.10539. Disponível em: <`http://doi.acm.org/10.1145/10515.10539`>.

[31] GOODRICH, M., GHOUSE, M., BRIGHT, J. “Generalized Sweep Methods for Parallel Computational Geometry”. In: *Proceedings of the Second Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA ’90, pp. 280–289, New York, NY, USA, 1990. ACM. ISBN: 0-89791-370-1. doi: 10.1145/97444.97695. Disponível em: <`http://doi.acm.org/10.1145/97444.97695`>.