

Walking Tree Method for 3D Vision

Tai Hsu
Department of Computer Engineering
Northwestern Polytechnic University
Fremont, California
hsut5@yahoo.com

Paul Cull
Department of Computer Science
Oregon State University
Corvallis, Oregon
pc@cs.orst.edu

Abstract

One of the major problems of visual perception is that the world we see is 3-dimensional, but we see it through a 2-dimensional retina. We all know part of the trick, binocular vision converts the 2-D image into a 3-D image. Here we want to describe a simple mechanism which makes this transformation computationally feasible.

keywords: stereo vision, 3D vision, binocular fusion, walking tree method, recombination, rearrangement, genome alignment.

Introduction

Stereo vision is to add an extra piece of information to every pixel of a 2D image: distance. Namely, every pixel contains two pieces of information: a color and the distance from the observer to the pixel. Stereo vision usually requires the comparison of two or more pictures, if using several cameras simultaneously. Sequences of pixels information have to be aligned between pictures in order to use the distance "shift" of objects to determine distances, e.g., the closer the object, the bigger its shift. We found that the Walking Tree Method [1, 2, 3, 4, 5, 6, 7] is excellent in doing such computations. Unlike the edit-distance model [9, 11] which assumes that changes between strings occur locally, the Walking Tree Method is mathematically modeled for sequence recombination in which "local alignment" and "global alignment" are just our method's two special cases. It's based on evidence showing that large scale changes are possible [8], e.g., meiotic cells' DNA replication is usually followed by homologous chromosomes' DNA recombination in which large pieces of DNA can be moved from one location to another (translocations), or replaced by their reversed complements (inversions). We have demonstrated its use in phylogeny, gene discovery, and gene verification [1, 2, 3, 4, 5, 6, 7]. Since Walking Tree Method is a powerful tool in sequence recombination, this paper will show that it's also a reasonable tool to compute the 3D image.

Walking Tree Methods

• Why Are Walking Tree Methods Needed?

Dynamic programming [10] is a way to implement the stereo vision by comparing two pictures. For two segments that both exist in two pictures, each picture taken by one of the two cameras (Figure 1),

dynamic programming cannot detect the two segments when both segments don't have the same sequential ordering on both pictures. But, the Walking Tree Method can detect the them even when the two segments' locations are swapped in one of the two pictures (Figure 1). Apparently, including the segment swapping case we mentioned, we need a method that can handle insertions, deletions, substitutions, and translocations altogether. The Walking Tree heuristic [1, 2, 3, 4, 5, 6, 7] can handle translocations and tends to highlight features of objects (Figure 2). Since pictures don't have segments of inversion, we have kept all functions of the Walking Tree Method, except its inversion computation. This allows it to run twice as fast since inversion accounts for 50% of computation.

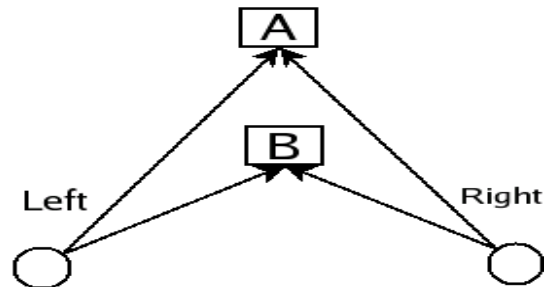
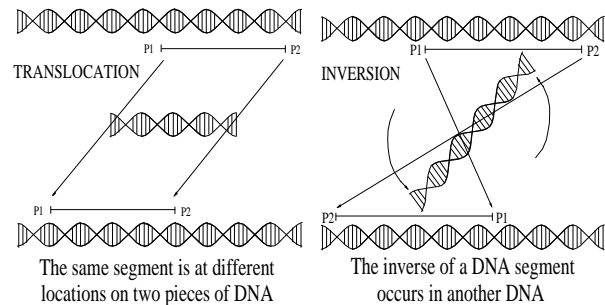


Figure 1: The segments' ordering observed by the left camera is "AB", while it will be "BA" by the right camera. A standard dynamic programming method can't detect such a translocation. But, the Walking Tree Method can.



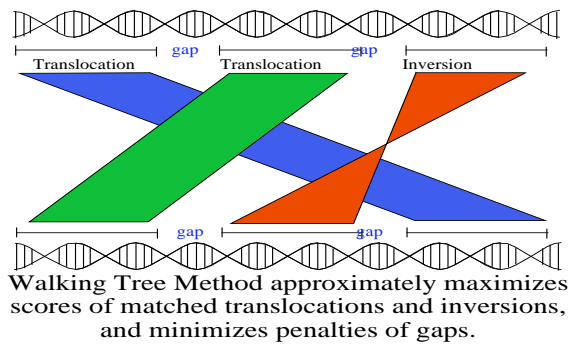


Figure 2: These three pictures illustrate translocation, inversion, and what Walking Tree Method computes, respectively. DNA strings can be replaced by strings of pixels for image comparison.

• **The Basic Method**

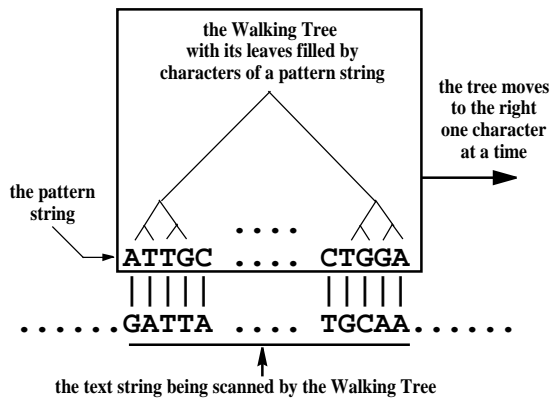


Figure 1: This picture shows the walking tree's structure, a binary tree. Leaves of the tree contain the characters of the pattern string P. After comparing each leaf with a corresponding character of the text string, the walking tree updates its nodes with new scores, then moves to the next position by moving each of its leaves one character to the right. Then it repeats the leaf comparison, and updates its node scores until it reaches the end of the text string.

The problem is to find an approximate alignment between two strings of pixels, one called pattern P, and the other called text T. Our metaphor is to consider the data structure as a walking tree [12] with |P| leaves, one for each pixel in the pattern. When the walking tree is considering position $l + 1$, the internal nodes remember some of the information for the best alignment within the first l pixels for the text (see Figure 3). On the basis of this remembered information and the comparisons of the leaves with the text pixels under them, the leaves update their information and pass this information to their parents. The data will percolate up to the root where a new best score is calculated. The tree can then walk to the next position by moving each of its leaves one pixel to the right. The whole text has been processed when the leftmost leaf of the walking tree has processed the rightmost pixel of the text.

We use a function that gives a positive contribution based on the similarity between aligned pixels, and a negative contribution that is related to the number and length of gaps, translocations, and inversions. A gap in an alignment occurs when adjacent pixels in the pattern are aligned with non-adjacent pixels in the text. The length of the gap is the number of pixels between the non-adjacent characters in the text.

The computation at each leaf node makes use of two functions, MATCH and GAP. MATCH looks at the current text pixel and compares it with the pattern pixel represented by the leaf. In the simplest case we use

$$\begin{aligned} \text{MATCH}(P_i, T_j) &= c && \text{if } P_i = T_j \\ \text{MATCH}(P_i, T_j) &= 0 && \text{if } P_i \neq T_j \end{aligned}$$

For many of our examples we use $c = 2$. If we were matching similar pixels then MATCH could return a value depending on how similar the two compared pixels are.

If we only used the MATCH function, the leaves would simply remember if they had ever seen a matching pixel in the text. We use the GAP function to penalize the match for being far from the current position of the tree. So the leaf needs to remember both if it found a match and the position of the walking tree when it found a match. For example, a simple GAP function could be:

$$\text{GAP}(\text{currentpos}, \text{pos}) = \log |\text{currentpos} - \text{pos}|,$$

where currentpos is the current position of the walking tree, and pos is the position at which the walking tree found a match. Then the leaf could compute

$$\text{SCORE} = \max[\text{MATCH}(P_i, T_j), \text{SCORE} - \text{GAP}(\text{currentpos}, \text{pos})]$$

and update pos to currentpos if MATCH is maximal. This means that a leaf will forget an actual match if it occurred far from the current position.

An internal node will only look at what it is remembering and at what its children have computed. Like a leaf node, the internal node computes

$$\text{SCORE} - \text{GAP}(\text{currentpos}, \text{pos})$$

which depends on what the node is remembering. From its two children, the node computes

$$\text{SCORE.left} + \text{SCORE.right} - \text{GAP}(\text{pos.left}, \text{pos.right})$$

This will penalize the sum of the children's scores because the position for the two scores may be different. But, the internal node also has to penalize this

score because the left or right position may be far from the current position, so it also subtracts

$$\min[\begin{array}{l} \text{GAP}(\text{currentpos}, \text{pos.left}), \\ \text{GAP}(\text{currentpos}, \text{pos.right}) \end{array}].$$

The internal node will keep the better of the score from its remembered data and the score computed from its children.

The walking tree will be computing a score for the current position of the tree. It is possible that it could forget a better score that was far from the present position. To avoid this problem, we add a special root node which simply keeps track of the best score seen so far.

In short, the walking tree finds an alignment f so that

$$f: [1, \dots, |P|] \rightarrow [1, \dots, |T|]$$

The alignment found approximately maximizes

$$\sum_{i=1}^{|P|} \text{MATCH}(P_i, T_{f(i)}) - \sum_{i=1}^{|P|-1} \text{GAP}(f(i), f(i+1)).$$

The actual functional being maximized is best described by the method itself and has no simple formula. Among other reasons, the above formula is only an approximation because the method tries to break strings into substrings whose lengths are powers of 2, even when using other lengths would increase the value of this formula.

• **Adjusting Gaps**

The basic method places gaps close to their proper positions. If we use the method to align the string “ABCDEF” in the string “ABCXXDEF” the gap may be placed between ‘B’ and ‘C’, rather than between ‘C’ and ‘D’. This is a result of the halving behavior of the basic method. By searching in the vicinity of the position that the basic method places a gap we can find any increase in score that can be obtained by sliding the gap to the left and right. The cost of finding better placements of the gaps is a factor of $\log|P|$ increase in runtime since at each node we have to search a region of the text of length proportional to the size of the substring represented by the node.

Experiment

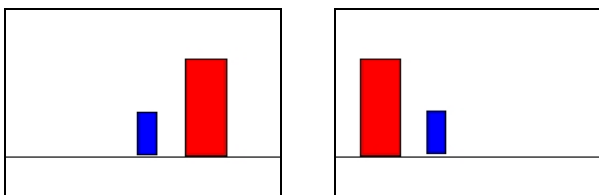


Figure 4: Analogous to Figure 1, the bigger block is A, and the smaller block is B. The left picture is what the left camera sees, and the right picture is what the right camera sees. Do you know which object is closer? The big block is closer to the cameras, while the small block is further. Why? You may try to look at them using only one eye per picture. Namely, use a piece of paper to separate the pictures and eyes so that the left eye can see only the left picture, and the right eye can see only the right picture. When both eyes focus on the big block, your eyes are twisted inward, meaning the big block is closer. When the small block is focused, both eyes relaxed, meaning it's further.

We simulated two pictures (see Figure 4) in which two objects are placed in a way similar to Figure 1. Since both cameras' lines of sight are parallel, the objects taken by the left camera position a little to the right, while the objects taken by the right camera position a little to the left. Then, the Walking Tree Method processes the pixels of the two pictures, line by line, horizontally, to determine the depth of each pixel. The more "shift" a pixel does, the closer it is to the observer.

Why do we want to highlight this special simulated condition in which the two objects' locations are swapped in the two cameras? Because such a condition is the most difficult one for classic dynamic programming methods to solve.

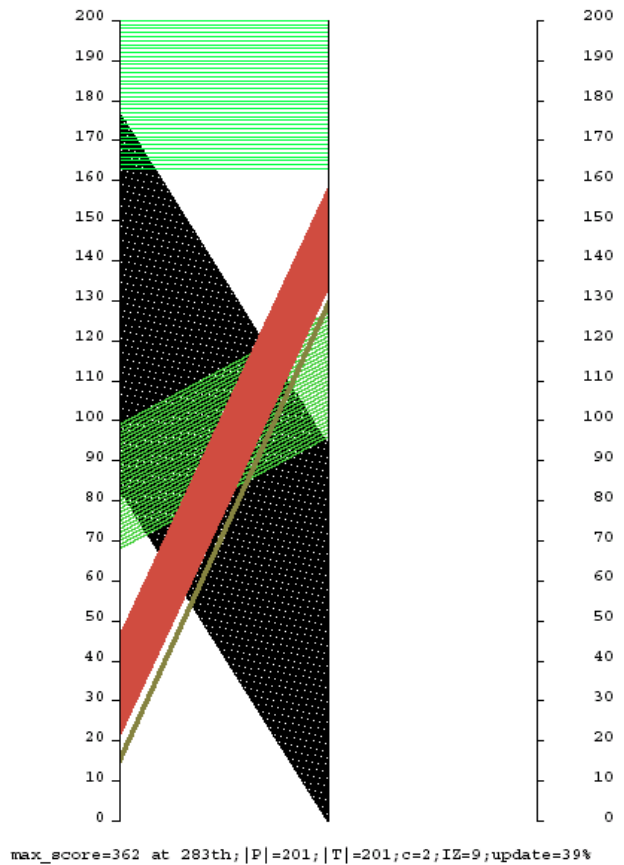


Figure 5: This is the alignment computed by the Walking Tree Method using the vertical lines passing through both objects in pictures of Figure 4. The middle axis represents the pixel string of Figure 4's left picture, and the left axis represents the pixel string of Figure 4's right picture. Because it's impossible to have a positive "shift" from the left picture to the right picture, this figure's two widest bands will be ignored, while only the bands of negative shift will be considered. And, the bigger shift of the red segment does tell us that it's closer to us, while the blue segment is further from us.

Figure 5 shows the alignment computed by the Walking Tree Method to demonstrate its ability to see swapped objects. We define a "positive" shift to be an object shifting its position, from the left location of a picture, to the right location of another picture. And, all other cases to be the "negative" shift. Because it's impossible to have a "positive" or "non-negative" shift from the left picture to the right picture, figure 5's two widest bands will be ignored, while only the bands of negative shift will be considered. So, the alignment tells us that the Walking Tree Method does see the two segments, even when their locations are swapped. And, the bigger shift of the big block does tell us that it's closer to us, while the small block's smaller shift tells us that it's further from us. The only problem left is that the area between both segments seem to be part of one of the two segments. This problem can be easily corrected by measuring the vertical shift using another pair of cameras.

Conclusion

From the result, we found that Walking Tree Method is also a powerful tool for stereo vision. The technique works by finding a "best" alignment between two strings of pixels. In addition to what can be done by a standard dynamic programming approach, the Walking Tree Method can find segments of swaps (translocations), which are required to accurately identify objects at various distances. What we did in the experiments is just a small step toward a much bigger goal of ours: to handle extremely complicated pictures.

Reference

- [1] P. Cull and J. Holloway, "Divide and Conquer Approximate String Matching: When Dynamic Programming is not Powerful Enough", Technical Report 92-20-06, Computer Science Department, Oregon State University, 1992.
- [2] P. Cull and J. Holloway, "Aligning genomes with inversion and swaps", Second International Conference on Intelligent Systems for Molecular Biology. Proceedings of ISBM '94, 195-202. AAAI Press, Menlo Park CA 1994.
- [3] P. Cull, J. Holloway, J. Cavener, "Walking Tree Heuristics for Biological String Alignment, Gene Location, and Phylogenies", CASYS '98, Computing Anticipatory Systems (D. M. Dubois, editor), American Institute of Physics, Woodbury, New York, pp201-215,1999.
- [4] Paul Cull, Tai Hsu, "Improved Parallel and Sequential Walking Tree Algorithms for Biological String Alignments", Supercomputing Conference, 1999.
- [5] Paul Cull, Tai Hsu, "Gene Verification and Discovery by Walking Tree Method", Pacific Symposium on Biocomputing, pp287-298, 2001.
- [6] Paul Cull, Tai Hsu, "Fast Walking Tree Method via Recurrent Reduction for Biological String Alignment", the 7th World Multi-Conference on Systemics, Cybernetics and Informatics, p.228-233, vol. XIV, 2003.
- [7] Paul Cull and Tai Hsu, "Parallel Walking Method for Sequence Recombination", PDCS 2004
- [8] K. M. Devos, M. D. Atkinson, C. N. Chinoy, H. A. Francis, R. L. Harcourt, R. M. D. Koebner, C. J. Liu, P. Masojc, D. X. Xie, M. D. Gale, "Chromosomal rearrangements in the rye genome relative to that of wheat", Theoretical and Applied Genetics 85:673-680, 1993
- [9] D. Gusfield, "Algorithms on strings, trees, and sequences: computer science and computational biology", Cambridge University Press, New York, NY, USA, 1997.
- [10] D. Scharstein and R. Szeliski, "A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms", International Journal of Computer Vision, 47(1/2/3):7-42, April-June 2002.
- [11] J. Setubal and J. Meidanis, "Introduction to Computational Molecular Biology", PWS Publishing, Boston, MA, 1996.
- [12] M. Python, "Just the Words." Methuen, London, 1989