E7 Final Project

Alexandre Dang, Dante Gao, Daniel Gribble

April 29, 2016

Abstract

This algorithm (attempts to) solve the marching band problem. That is, given the initial and final positions of the marchers, and a time limit, the algorithm will attempt to find a way where the band members can move towards their assigned target position without colliding with each other. This algorithm should always assign a marcher to each target position, but may or may not determine an instruction set that does not involve collisions.

1 Introduction

To run, the algorithm requires the functions `munkres.m` and `simulator.m` to be in the same directory as the main `calband_transition.m` function.

Finding a solution to this problem involves two main aspects: assigning each marcher to a unique target position, and addressing the collisions that may arise during the transitions.

2 Overview

We will only discuss the most pertinent elements of the algorithm; notes regarding other sections may be found as comments within the source code.

2.1 Preliminary Target Assignment

In order to assign a target position to each marcher, the algorithm calculates the L_1 distance, or taxicab distance, between each marcher and every possible target position, and stores it in the array `dist`. One must compute the taxicab distance as opposed to Euclidean distance as the marchers may only travel in the four cardinal directions. If the distance is too great for a marcher to travel within the maximum number of beats, the corresponding value is considered to be infinity.

`dist` can be considered the cost matrix of a linear assignment problem. As such, we decided to implement the Hungarian algorithm to determine the optimal pairing of marcher and target position, as one can significantly reduce the difficulty of resolving collisions by utilizing optimal target assignemnts. Our initial implementation of the Hungarian algorithm in MATLAB was poorly optimized and as a result, we decided to utilize Dr. Yi Cao's implementation of the Munkres assignment algorithm, `munkres.m`, which is a modified version of the Hungarian algorithm. Our implementation ran in factorial time, and as such, our implementation was unable to solve larger test cases within a reasonable time frame, while Dr. Cao's algorithm runs in polynomial time.

Broadly speaking, the Hungarian algorithm (and by extension the Munkres assignment algorithm) works by subtracting the smallest possible distance a marcher must travel from every other potential distance, and similarly, subtracting the smallest distance between a particular final position and every marcher. Through multiple iterations of this process, the total distance collectively traveled by every band member is minimized.

The results of the Munkres assignment algorithm are then stored in the `instructions` struct.

2.2 Collision Detection and Correction

After the initial assignment, the `instructions` is passed to `simulator.m`, which preforms direction assignment and collision detection. `simulator.m` returns a "processed" `instructions` struct as well as an overview of the collisions that would occur with that particular `instructions` struct.

An `instructions` struct is "processed" when the `wait` field contains a valid entry, and the value of the `direction` field will allow the marcher to reach their target. This is accomplished by comparing each marcher's initial position with their intended final position, and assigning the corresponding value.

`simulator.m` also determines the collisions that will occur within the "processed" `instructions` struct. This is accomplished by creating a 3D array containing the positions of each marcher at each frame during the transition, and detecting duplicate entries. Data regarding the frame number, row index, column index, and marchers involved is returned in an array named `collisions`.

Collision correction is handled within `calband_transition.m`. This segment runs within a `while` loop, which is set to terminate upon 100 iterations of the loop or after 90 seconds have elapsed, whichever condition is satisfied first. As such, in worst-case scenarios, this section of code will only run for 90 seconds. The algorithm first makes a copy of the `instructions` struct, named `save`. It then looks for certain criteria involving colliding band members, and makes changes to `save` accordingly.

The first criterion for adjustment involves band members who are to move in purely horizontal or vertical paths. If marchers satisfying this condition collide, their assigned targets are swapped, and the number of collisions within this instruction set is recorded. If fewer collisions occur than in the instruction set stored in `save`, the previous iteration of `save` is overwritten by this instruction set.

The next criterion determines, for applicable marchers, whether it is more optimal to head north or south first, or east or west first. The default state, as assigned in `simulator.m`, is to move north or south first, and east or west second. The process used to determine optimality is very similar to above. For each marcher involved in a collision, the value of the `direction` field is inverted, and the resulting number of collisions is computed. If there is an improvement, the previous iteration is discarded.

We planned to utilize the `wait` field to further augment the current collision correction algorithm, but ultimately, the algorithm not make use of the `wait` field. However, we would design an algorithm which would first determine the maximum number of beats each marcher may wait, depending on their distance from their target position and the maximum number of beats for the transition in question. Afterwards, much like as previously described, the algorithm would increment the number of beats to wait and determine the number of collisions associated with that instruction set, discarding those who do not improve the number of collisions.

3 Acknowledgments

The authors would like to thank the E7 instructional team for their support and dedication this semester and Dr. Yi Cao of Cranfield University for his implementation of the Munkres assignemnt algorithm.

`munkres.m` was adapted from `http://www.mathworks.com/matlabcentral/fileexchange/20652`. Its license is included within the download for our algorithm.