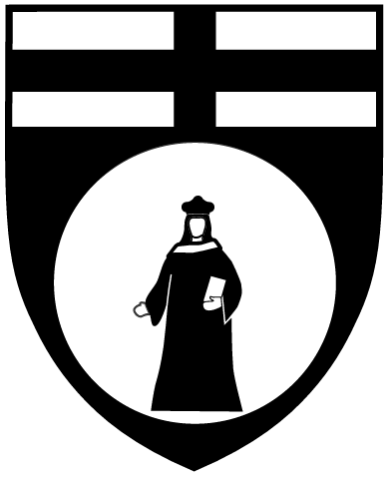

Dibris Dipartimento di Informatica, Bioingegneria,
Robotica e Ingegneria dei Sistemi

**TODO ask for dibrisunige-thesis instead of this report format**

---

**Thesis**

# Origin of Movement

---

Advisors Gualtiero Volpe
Luca Oneto

UNIVERSITY OF GENOA, AUGUST 2023

# Contents

# 1 Introduction

ighjghbygbgt

# 2 Motivation

Movement is one of the first complex actions that we learn when we are born. It defines how we interact with the world around us. When moving our bodies,objective we are not simply performing an action aiming to achieve a direct and explicit result, but most of the time we are also communicating with others. Furthermore, we strive to constantly improve human performances, prevent injuries, promote physical activity and health and inform rehabilitation strategies. For this reason, the research in human movement has branches in various fields of study such as psychology, biomechanics, physiology and sociology.

# 3 Aim

This thesis aims to provide an automated way to recognize the origin of movement. The optimal result would be to obtain results with stronger confidence scores than previous works on this topic made with different methods.

# 4 State of Art

## 4.1 Origin of Movement

# 5 Methodology

We chose to create a Python script to model a Recurrent Neural Network.

# 6 Results

Here the main achieved or potential results. Use in case tables, figures, etc.

# 7 Temporal stabilization of clusters

Dopo la clusterizzazione dobbiamo fare un riassegnamento di colori ai clusters per stabilizzarli nel tempo. Come si è già detto, l'ordine con cui i clusters vengono decisi ad ogni istante è randomico e l'ordine di scelta dei clusters viene rappresentato da un colore.
Quindi, dall'iniziale assegnazione di colori randomica, vogliamo passare ad un'assegnazione di colori dove, due clusters dello stesso colore, da un istante all'altro, abbiano in comune più nodi, ossia più joints, possibili.
Questo perchè l'assegnazione randomica porta dei problemi a livello di visualizzazione del grafo, in quanto è possibile che da un istante all'altro, due cluster con molte joints in comune abbiano colore diverso e quindi la visualizzazione nel tempo potrebbe risultare come un continuo switch di colori facendo perdere un'idea di continuità.

Per stabilizzare i cluster nel tempo, e quindi far si che le coppie dei vari clusters colorati da un istante all'altro mantengano il maggior numero di joint in comune, ci riconduciamo al Maximum Perfect Weight Matching.
Abbiamo implementato due algoritmi: uno di Brute Force, in cui calcoliamo i pesi totali di tutte le permutazioni degli assegnamenti possibili di colori e scegliamo l'assegnamento con il peso totale massimo, e l'Hungarian Algorithm, un metodo di ottimizzazione combinatoria che risolve in tempo polinomiale il problema dell'assegnamento. Dato che nel nostro caso il numero di cluster è solitamente piccolo, in alcuni casi risulta essere più efficiente l'algoritmo di Brute Force.

## 7.1 Maximum Perfect Weight Matching for bipartite graphs

Il Maximum Perfect Weight Matching è un problema di ottimizzazione riguardante i grafi.
I grafi in questione sono bipartiti, ossia i vertici possono essere suddivisi in due insiemi distinti in modo tale che tutti gli archi del grafo connettano vertici di insiemi diversi, ma non ci siano archi che connettano vertici all'interno dello stesso insieme. Inoltre, questi grafi sono non diretti, ossia la relazione tra due nodi è bidirezionale, in altre parole, l'arco che collega il nodo A al nodo B implica automaticamente l'esistenza dell'arco che collega il nodo B al nodo A.
Nel problema del Maximum Perfect Weight Matching i vertici rappresentano gli elementi da accoppiare e gli archi rappresentano le possibili connessioni tra questi elementi. Ogni arco ha associato un peso che rappresenta il valore o l'importanza dell'accoppiamento tra i vertici connessi.
L'obiettivo è trovare un accoppiamento perfetto, ovvero un insieme di archi in cui ogni nodo è connesso esattamente a un altro nodo, senza sovrapposizioni o nodi isolati, massimizzando il peso totale degli archi nell'accoppiamento. Il peso totale è la somma dei pesi degli archi inclusi nell'accoppiamento.
I pesi degli archi vengono raccolti in una matrice che prende il nome di matrice di utilità.
La matrice di utilità sarà una matrice quadrata, dato che il numero di nodi facenti parte dei due insiemi distinti è lo stesso. Le righe potrebbero rappresentare i nodi facenti parte del primo insieme, mentre le colonne potrebbero rappresentare i nodi facenti parte del secondo insieme, oppure viceversa. L'elemento (i, j) della matrice rappresenta il peso dell'arco che collega il vertice "i" al vertice "j".

### 7.1.1 Maximum Perfect Weight Matching: temporal stabilization

In questo esempio, anzichè avere nodi collegati da archi, avremo una situazione leggermente diversa.
Al posto dei nodi introdotti precedentemente, avremo dei clusters, ossia degli insiemi di nodi, che per noi rappresentano le joints dello scheletro, collegati fra loro da archi.

Consideriamo i clusters all'istante t e i clusters all'istante t+1 come due insiemi distinti, perciò non esisteranno archi che collegano clusters facenti parte dello stesso insieme.
Ogni cluster di un insieme deve essere collegato tramite un arco ad uno ed un solo cluster dell'altro insieme, con l'obbiettivo di massimizzare il peso totale degli archi. Il peso di un arco è pari al numero di nodi che hanno in comune i due clusters che l'arco in questione sta collegando.
Come già detto, a questo punto della pipeline, conosciamo già ad ogni istante le joints che fanno parte dei vari clusters e i colori dei vari clusters, ma vogliamo fare un riassegnamento di colori per stabilizzare la visualizzazione nel tempo. La clusterizzazione precedente raggruppa le varie joints in clusters e gli assegna un colore random, mentre in questa fase della pipeline ottimizziamo l'assegnazione dei colori.
Partiremo perciò analizzando i colori dei clusters all'istante iniziale e sceglieremo l'assegnamento migliore di colori per l'istante successivo. Ripeteremo questo procedimento iterativamente fino all'ultimo istante.

### 7.1.2 Maximum Perfect Weight Matching: mock example

Consideriamo un caso semplificato per vedere in termini di visualizzazione come funziona il Maximum Perfect Weight Matching e quindi supponiamo di avere all'istante t 9 nodi rappresentati dalle lettere dell'alfabeto dalla A alla I e 3 clusters, mentre all'istante t+1 gli stessi 9 nodi precedenti e 3 clusters differenti da quelli dell'istante precedente.
I colori possibili sono 3, supponiamo che siano per esempio rosso, blu e verde, conosciamo il colore dei clusters all'istante t e vogliamo assegnare i colori ai clusters all'istante t+1.
Ci sono quindi 3! possibili assegnamenti di colore.
Come già detto, il peso di un arco è equivalente al numero di nodi che hanno in comune quei due clusters che vengono collegati dall'arco in questione.
Da questo esempio si può notare che l'assegnamento con peso totale massimo per l'istante t+1 è quello visivamente più coerente con quello dell'istante t.

¡¡ disegno ¿¿

## 7.2 Brute Force Algorithm

### 7.2.1 Brute Force Algorithm summarized

Nel caso del Brute Force, quindi, si costruisce inizialmente la matrice di utilità e si calcola il peso totale per ciascuna permutazione di assegnamento ed infine si sceglie l'assegnamento con peso massimo.

### 7.2.2 Brute Force Algorithm application

## 7.3 Hungarian Algorithm

The Hungarian matching algorithm, also called the Kuhn-Munkres algorithm, is a O grande V alla 3 algorithm that can be used to find maximum-weight matchings in bipartite graphs, which is sometimes called the assignment problem. A bipartite graph can easily be represented by an utility matrix, where the weights of edges are the entries. A maximum-weight matching problem using an utility matrix can be transformed into a minimum-weight matching problem. This can be achieved by introducing an auxiliary matrix, referred to as the "cost matrix." The cost matrix is essentially a duplicate of the utility matrix, but with all its elements negated.

### 7.3.1 Hungarian Algorithm summarized

The operation of the Hungarian algorithm for minimum-weigth matching problems can be summerized in these 5 steps:

1. Subtract the smallest entry in each row from all the other entries in the row. This will make the smallest entry in the row now equal to 0.
2. Subtract the smallest entry in each column from all the other entries in the column. This will make the smallest entry in the column now equal to 0.
3. Draw lines through the row and columns that have the 0 entries such that the fewest lines possible are drawn.
4. If there are n lines drawn, an optimal assignment of zeros is possible and the algorithm is finished. If the number of lines is less than n, then the optimal number of zeroes is not yet reached. Go to the next step.
5. Find the smallest entry not covered by any line. Subtract this entry from each row that isn't crossed out, and then add it to each column that is crossed out. Then, go back to Step 3.

### 7.3.2 Hungarian Algorithm application

Let's take for example the following cost matrix and try to apply the Hungarian algorithm for minimum-weigth matching problems:

| | | |
|---|---|---|
| 108 | 125 | 150 |
| 150 | 135 | 175 |
| 122 | 148 | 250 |

Subtract the smallest value in each row from the other values in the row:

| | | |
|---|---|---|
| 0 | 17 | 42 |
| 15 | 0 | 40 |
| 0 | 26 | 128 |

Now, subtract the smallest value in each column from all other values in the column:

| | | |
|---|---|---|
| 0 | 17 | 2 |
| 15 | 0 | 0 |
| 0 | 26 | 88 |

There are 2 lines drawn, and 2 is less than 3, so there is not yet the optimal number of zeroes.

| | | |
|---|---|---|
| 0 | 17 | 2 |
| 15 | 0 | 0 |
| 0 | 26 | 88 |

Find the smallest entry not covered by any line. Subtract this entry from each row that isn't crossed out, and then add it to each column that is crossed out. Then, go back to Step 3. 2 is the smallest entry. First, subtract from the uncovered rows:

| | | |
|---|---|---|
| -2 | 15 | 0 |
| 15 | 0 | 0 |
| -2 | 24 | 86 |

Now add to the covered columns:

| | | |
|---|---|---|
| 0 | 15 | 0 |
| 17 | 0 | 0 |
| 0 | 24 | 86 |

Now go back to step 3, drawing lines through the rows and columns that have 0 entries:

| | | |
|---|---|---|
| 0 | 15 | 0 |
| 17 | 0 | 0 |
| 0 | 24 | 86 |

There are 3 lines (which is nn), so we are done. The assignment will be where the 0's are in the matrix such that only one 0 per row and column is part of the assignment.

| | | |
|---|---|---|
| 0 | 15 | 0 |
| 17 | 0 | 0 |
| 0 | 24 | 86 |

Replace the original values:

| | | |
|---|---|---|
| 108 | 125 | 150 |
| 150 | 135 | 175 |
| 122 | 148 | 250 |

# 8 Conclusions

Here recall the overall project done, and in case of findings some suggestions for future works.

# References

[1] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[2] Donald E. Knuth. *The TEX Book*. Addison-Wesley Professional, 1986.

[3] Leslie Lamport. *LATEX: a Document Preparation System*. Addison Wesley, Massachusetts, 2 edition, 1994.

[4] Michael Lesk and Brian Kernighan. Computer typesetting of technical journals on UNIX. In *Proceedings of American Federation of Information Processing Societies: 1977 National Computer Conference*, pages 879–888, Dallas, Texas, 1977.

[5] Frank Mittelbach, Michel Gossens, Johannes Braams, David Carlisle, and Chris Rowley. *The LATEX Companion*. Addison-Wesley Professional, 2 edition, 2004.