Índex

[Algorismes i Heurístiques Implementades 1](#_Toc104198239)

[Comparació d’Algorismes 1](#_Toc104198240)

[Temporal 1](#_Toc104198241)

[Solucions Trobades 1](#_Toc104198242)

[Justificació del Llenguatge escollit per a programar els algorismes 1](#_Toc104198243)

[Resultats Obtinguts 1](#_Toc104198244)

[Bilbiografia 1](#_Toc104198245)

# Algorismes i Heurístiques Implementades

Per aquesta explicació la hem dividit en dos apartats, l’explicació del algorisme A\* i la del algorisme CSP. Se’ns va demanar que realitzéssim la implementació d’aquests dos algorismes per fer la cerca del camí més curt entre dues poblacions. Aquestes poblacions es troben dins d’un json proveeït per l’assignatura.

## A\*

L’A\* és, avui en dia, l’algorisme per excel·lència en el camp del pathfinding (ja sigui en grafs, laberints, mapes de jocs...): òptim i ens assegura sempre trobar la millor solució.

És l’equivalent al Djikstra però en una versió informada: en l’A\*, la heurística ens serveix per determinar el camí a explorar (quin dels nodes que es poden explorar a continuació té millor pinta perquè sigui el següent a ser explorat). La heurística és el que fa que aquest algorisme sigui molt més òptim que el Djikstra, ja que no ens cal explorar totes les possibilitats, sinó que sempre agafarem la que creiem que es millor en un moment donat.

Per suposat en l’espai que estiguem explorant s’ha de poder definir una heurística, que en el cas de l’A\* és una mesura per saber de forma aproximada com de llunyà ens queda el node objectiu. En el cas que no tinguem manera de determinar algun tipus de distància fins al node objectiu, haurem d’implementar un altre algorisme d’exploració, com per exemple el ja anteriorment mencionat Djikstra.

L’A\* assigna a cada node un cost f(n) = g(n) + h(n), on n és el node, g és la funció que determina la distància del node inicial fins a n, i h és la funció heurística que estima el cost entre el node n i el node objectiu. Tal i com hem dit, el pròxim node a explorar en tot moment serà el que tingui la f mes baixa, fins a trobar el node objectiu.

És molt important mencionar que aquesta heurística sempre ha d’estimar un cost igual o més baix al cost real. Del contrari, l’algorisme no ens assegura trobar la solució més òptima del node inicial al node final.

**Implementació**

Per la implementació del A\* hem fet ús de pseudocodis que hem trobat cercant a internet informant-nos del funcionament del algorisme. Per fer-ho hem de primer llegir el json i crear la estructura de dades que creguéssim més convenient, en el nostre cas un graph dirigit.

Per fer-ho disposem de dues classes, la primera, la classe Node que té el seguit de atributs de una ciutat del json. A més d’aquests atributs té també un array de veïns, aquests veïns son instàncies de la classe Edge. Aquests veïns son els nodes amb els que esta connectat el actual i també té un camp parent que ens permetrà recórrer la solució un cop la trobem.

La classe Edge per altra banda és una classe més senzilla que ens permet travessar el graph que té el pes de passar per aquella aresta, que en el nostre cas és, per exemple, la distància que les separa, i rep també el node destí d’aquest edge.

Un cop plantejada la estructura de dades, la omplim recorrent el json i definint aquestes relacions del graph, i finalment és quan cridem a la nostre funció aStar.

La funció aStar rep dos arguments, node inici i node final, e intentarà retornar-nos el camí a recórrer, aquest camí estarà representat per un node, aquest hauria de ser el node final, el node destí, i tindrà en el seu parent el node a través del qual s’ha arribat, i si seguim pujant pels parents acabaríem arribant al node inici.

Per fer-ho fem ús de dues llistes, i afegim a una el node inici, e iterarem fins que aquesta estigui buida. Idealment aquesta llista conté tots els nodes travessats. En cas que el primer node de la llista sigui el node destí, voldrà dir que ja hem arribat i per tant marxem de la funció. En cas contrari, iterem pels vehins cercant aquell que no hagi estat visitat i el visitem, i en cas que sí que hagi estat visitat mirem si té menys pes que el actual i fem que el actual sigui aquest.

Essencialment

## CSP

# Comparació d’Algorismes

## Temporal

## Solucions Trobades

# Justificació del Llenguatge escollit per a programar els algorismes

Pel que fa al llenguatge triat, hem emprat Python. Vam considerar un parell de llenguatges a part de python: Java, Go, Dart i C. Però cada un d’ells tenia costats negatius de cara a fer la implementació dels algorismes triats.

En el cas de Java, tant Python com Java compilen a bytecode per executar-ho amb màquines virtuals. Això aïlla el codi de tenir diferències en múltiples sistemes operatius, el que fa que siguin cross platform. Dit això, hi ha una diferència crítica: Python es compila en runtime mentre que java es compila per adelantat i distribueix el bytecode.

# Resultats Obtinguts

# Bibliografia

1. <https://www.youtube.com/watch?v=eSOJ3ARN5FM>
2. <https://www.geeksforgeeks.org/a-search-algorithm/>
3. <https://en.wikipedia.org/wiki/Constraint_satisfaction_problem>
4. <https://www.baeldung.com/cs/csp>