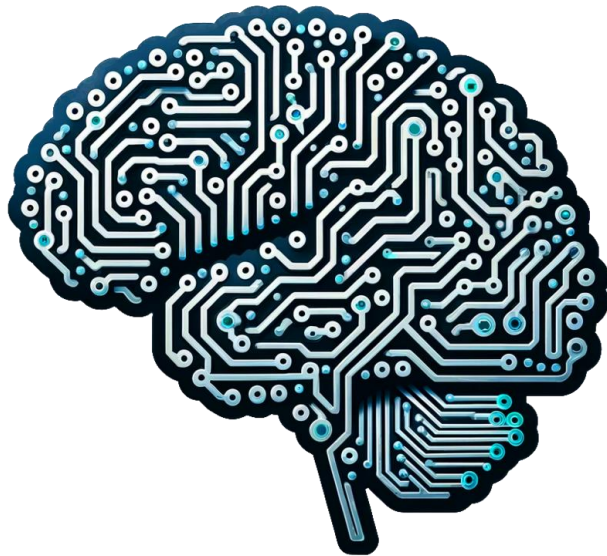


# Intel·ligència Artificial

## Pràctica 1: Cerca informada



Alumnes: Lyubomyr Grygoriv & Satxa Fortuny

Laboratori: L2

Data: 16/03/2025









## Índex

1 – Especificacions .....	3
2 – Objectius .....	3
3 - Formalització del problema .....	4
4 - Estructura general del projecte .....	5
PathAlgorithm.java .....	5
Heuristica.java .....	5
State.java.....	5
Node.java .....	6
5 – Algoritmes de cerca.....	7
5.1 - Algoritme Best-first.....	7
5.2 - Algoritme A* .....	9
6 – Heurístiques .....	11
6.1 - Heurística 1 .....	11
6.2 - Heurística 2 .....	12
6.3 - Heurística 3 .....	13
6.4 - Resum Heurístiques .....	13
7 – Proves realitzades .....	14
7.1 - Mapa proporcionat per l'enunciat.....	14
7.1.1 - Heurística 1 – Tipus de Pitàgores.....	15
7.1.2 - Heurística 2 – Tipus de Pitàgores*importància .....	17
7.1.3 - Heurística 3 – Distància de Manhattan.....	18
7.2 -Mapa propi .....	20
7.2.1 - Heurística 1 – Tipus de Pitàgores.....	21
7.2.2 - Heurística 2 – Tipus de Pitàgores*importància .....	22
7.2.3 - Heurística 3 – Distància de Manhattan.....	23
7.3 - Hill Climbing .....	24
7.3.1 - Mapa proporcionat per l'enunciat.....	24
7.3.1 - Mapa proporcionat per l'enunciat.....	27

## 1 – Especificacions

En aquesta pràctica volem estudiar formes de trobar camins per moure'ns ràpidament per un mapa amb diferents alçades.

El mapa estarà representat per una matriu de  $X \times Y$  caselles. Cada casella té associat un valor d'alçada; algunes caselles podran tenir un precipici insalvable. Cada problema estarà definit per una configuració del mapa, una casella inicial ( $x_i, y_i$ ) i una casella final ( $x_f, y_f$ ). L'aspecte que pot tenir un problema concret és el següent:

0	1	2	3	3		0	1	2	2
2	1	2	4	3	2	1	3	3	3
2	2		4	5	3	2		4	4
3	3		4	6	4	2	3	3	3
2	2	3	3	5	3	3	2	3	3
2	1	1	3	4		2	2	3	4
2	0	0	1	2	1	1	1	1	
	1	0	1	2	0	2	3	2	3
2	2	2	1	1	0	2	3	3	4
4	3	2	2		1	1	2	4	5

Podem desplaçar-nos en horitzontal i en vertical, però no en diagonal. El temps que triguem en moure'ns serà:

- $1 + (\text{diferència d'alçades entre la casella destí i la d'origen})$ , si la diferència és positiva o 0.
- $-1/2$  unitat, si la diferència és negativa.
- No podem desplaçar-nos a una casella amb precipici. Es pot considerar que la distribució d'alçades és geogràficament consistent (muntanyes, valls, pla), però els precipicis són aleatoris.

**\*Important:** en aquesta pràctica hem suposat que tots els mapes seran de  $10 \times 10$ .

## 2 – Objectius

L'objectiu d'aquesta pràctica és aprendre a formalitzar un problema, crear 3 heurístiques diferents (una d'aquestes ha de ser admissible respecte al temps), implementar en Java els algoritmes Best-first i A\* i per últim analitzar els diferents resultats obtinguts per cada heurística i algoritme.

### 3 - Formalització del problema

El problema es basa principalment en un mapa de 10x10 caselles, on en cada casella disposem d'un valor el qual fa referència a l'alçada. S'ha de trobar un camí des d'una coordenada inicial a una final.

Davant d'aquestes especificacions, vam veure que una manera de representar els estats, seria representar les coordenades x i y. En cas de l'alçada de cada casella, no vam veure necessari guardar-la a l'estat, ja que a partir de les coordenades i el mapa, la podem obtenir de forma dinàmica.

0	1	2	3	3		0	1	2	2
2	1	2	4	3	2	1	3	3	3
2	2		4	5	3	2		4	4
3	3		4	6	4	2	3	3	3
2	2	3	3	5	3	3	2	3	3
2	1	1	3	4		2	2	3	4
2	0	0	1	2	1	1	1	1	
	1	0	1	2	0	2	3	2	3
2	2	2	1	1	0	2	3	3	4
4	3	2	2		1	1	2	4	

Abstracte 1. Un possible mapa del problema.

La coordenada de la casella de color groc, fa referència a les coordenades (0,0). En cas de la casella de color verd, les coordenades són (9,9). (coordenades invertides).

Per cada estat (coordenada x i y) disposem de 4 operadors diferents:

- "Right" -> on realitzem un increment a la coordenada x ( $x+1$ ).
- "Down" -> on realitzem un increment de la coordenada y ( $y+1$ ).
- "Left" -> on realitzem un decrement a la coordenada x ( $x-1$ ).
- "Up" -> on realitzem un decrement de la coordenada y ( $y-1$ ).

Per cada operador disposem d'una certa aplicabilitat (no sortir del mapa):

- "Right" -> si coordenada x està al límit dret ( $x=9$ ).
- "Down" -> si coordenada y està al límit inferior ( $y=0$ ).
- "Left" -> si coordenada x està al límit esquerre ( $x=0$ ).
- "Up" -> si coordenada y està al límit superior ( $y=9$ ).

**\*PER TOTS ELS OPERADORS. No podem aplicar-los si cap a on ens dirigim és un precipici.**

*IMPORTANT -> en el programa, llegim el mapa d'un fitxer, llavors surt al revés. Davant d'aquesta situació, els operadors up i down estan invertits, down ( $y-1$ ) i up ( $y+1$ ).*

## 4 - Estructura general del projecte

Hem decidit estructurar aquest projecte de manera que sigui fàcilment expansible i hi hagi el màxim de reutilització de codi possible, així doncs explicarem les principals classes que ens han permès aconseguir-ho.

### PathAlgorithm.java

L'objectiu d'aquesta classe és definir una sèrie de fills de tipus d'algorismes de cerca. Com que realment l'única cosa que canvia d'una implementació a una altra és la cerca en si mateixa, hem decidit que ajuntar tots els constructors en un és una bona opció. Per aquesta raó la classe és abstracta, ja que si fos una interfície no podrem unificar la implementació d'aquest. Per una altra banda, si fos una classe estàndard, a l'hora de cridar a l'algorisme de cerca podríem fer servir la mateixa classe, quan aquesta realment no hi té cap implementació.

El constructor bàsicament passa el mapa donat per un fitxer a una llista de Java. Hem decidit que aquesta implementació és millor que la d'una matriu.

A part, també passa les coordenades inicials i finals a estats, d'aquesta manera segueix l'estructura donada dels algorismes, on es treballa amb estats, no amb coordenades. També, disposem de diverses funcions que usarem en els fills, com podria ser *seeMap* (per mostrar el mapa) i *calculeTime* (per calcular el temps d'un estat a un altre), entre altres.

### Heuristica.java

En aquesta classe el que hem fet ha sigut unificar tots els tipus d'heurístiques en una família. A diferència dels algorismes de cerca, les heurístiques no necessiten constructor ni cap mena de codi comú, per la qual cosa hem definit la classe com a una interfície. D'aquesta manera evitem que es pugui fer servir la interfície en sí encara podrem unificar totes les classes filles per a poder reutilitzar codi a la part d'algorismes i no haver de fer un mètode per cada heurística.

### State.java

Aquesta classe existeix com a part de la formalització del problema. Per no haver de trencar la implementació donada a teoria i per poder utilitzar el mateix algorisme de cerca per a diferents objectius, haurem d'universalitzar els problemes. D'aquesta manera doncs hem creat la classe State que junta les coordenades en una sola instància.

## Node.java

Amb aquesta classe és amb la que realment treballarem a nivell d'algorisme, ja que serà la que guardarà tota la informació. En aquesta classe doncs guardem:

- **Estat.** El que vindrien a ser les coordenades.
- **Camí.** Per a poder saber el recorregut que hem fet al final de l'algorisme haurem de guardar el procediment d'alguna manera, aquí ho farem a través d'un String on hi guardarem les direccions que ha pres.
- **Heurística.** El valor que ens ha proporcionat la funció heurística en aquesta coordenada.
- **Time.** Aquesta variable ens permet anar guardant el valor dels pendents per les quals hem passat, així al final de l'algorisme podrem saber el temps total.

## 5 – Algoritmes de cerca

### 5.1 - Algoritme Best-first

```
Cerca (Ei, Ef, estats)
pends := ([Ei, θ, h(Ei)]);
tracts := θ;
trobat := Fals;
mentre no(trobat) i (pends ≠ θ) fer
    [N, camí, valor] := Primer(pends);
    Eliminar_primer(pends);
    si (N=Ef)
        llavors trobat := cert; solució = camí;
    sino per tot successor X de N fer
        si X ∉ tracts i X ∉ pends
            llavors pends := Afegir_orden(pends, [X, camí+op, h(X)]);
        fsi
    fper
        tracts := tracts + {N}
    fsi
fmentre
si trobat llavors retorna(solució) sino retorna("no existeix el camí");
```

Abstracte 2. Algoritme Best-first.

Aquest algoritme es caracteritza per escollir en funció del valor de l'heurística. Bàsicament, el que fa és, d'un node agafar els seus successors i classificar-los en dues llistes, en la de pendents i en la de tractats. De normal el que farà bàsicament serà posar tots els nodes veïns a la llista de pendents mentre no estiguin a la de tractats. A partir d'aquí al següent cicle escollirà el primer node de la llista, ja que aquesta estarà ordenada, d'aquí el nom Best First, ja que sempre escull el millor.

Procedirem llavors a explicar el codi per parts

#### Inicialització

```
this.pendents = new LinkedList<>();
this.tractats = new LinkedList<>();
boolean trobat = false;
Solution solution = new Solution(path:"", time:0, it:0);
int nIteration = 0;
if (correctInitialValues()) {
    pendents.add(new Node(this.current, camí:"", resultat:0));
}
else{
    solution.setPath(path:"Initial values are incorrect");
    return solution;
}
```

Abstracte 3. Inicialització codi Best-first.

En aquesta part definim dues llistes de Java on hi guardarem els nodes pendents i tractats. Posteriorment, tenim el booleà del bucle i creem la classe Solution on hi guardarem tota la informació rellevant de manera encapsulada. A partir d'allí passarem unes petites comprovacions per saber que tota la informació és acceptable. En cas que estigui tot correcte posarem el node inicial a la llista de pendents per a començar la cerca.

## Principi bucle

```
while ((!trobat) && (!pendents.isEmpty())){
    Node node = pendents.get(index:0);
    pendents.remove(index:0);
    if (node.getEstat().isEqual(this.end)){
        trobat = true;
        solution.setPath(node.getCami());
        solution.setTime(node.getTime());
    }
}
```

*Abstracte 4. Inici bucle codi Best-first*

En aquesta part del bucle escollirem el millor node de la llista i comprovarem si aquest és el final, en cas afirmatiu ho emmagatzemem tot a la solució i finalitzem amb el bucle.

## Successors

```
for (Node successor:succ){
    if (!tractats.stream().anyMatch(x -> x.isEqual(successor.getEstat()))
        && !pendents.stream().anyMatch(x -> x.getEstat().isEqual(successor.getEstat()))){
        // Calcular nou heuristic
        successor.setHeuristica(h.heuristica(this.map, node.getEstat(), successor.getEstat(), this.end));
        successor.setTime(node.getTime()+this.calculaTime(node.getEstat(),successor.getEstat()));
        pendents.add(successor);
        pendents = pendents.stream().sorted(Comparator.comparing(Node::getHeuristica))
            .collect(Collectors.toList());
        nIteration++;
    }
}
tractats.add(node.getEstat());
```

*Abstracte 5. Tractament successors codi Best-first.*

Arribats en aquest punt i seguint l'esquema del Best First, haurem de visitar tots els veïns i classificar-los per llistes. Això ho aconseguirem filtrant amb streams per saber que no estem tractant un estat que ja hem tractat i filtrat també la de pendents per saber que no estem duplicant nodes. Cal diferenciar el fet que la llista de tractats és d'estats, ja que no haurem de recuperar cap informació respecte a aquests estats, mentre que els nodes de la de pendents, poden arribar a ser part de la solució. Un cop filtrats aquests veïns verges, els hi calcularem una heurística i un temps, per després afegir a la llista de pendents i ordenar-la per la següent iteració.

Al final de tot afegirem el node que acabem de tractar a la llista de tractats.

## Final

```
solution.setIteration(nIteration);
if (trobat) return solution;
else{
    solution.setPath(path:"No solution");
    return solution;
}
```

*Abstracte 6. Finalització codi Best-first.*



Un cop acabat l'algoritme mirem si hem trobat la solució o si no n'hi ha.

## 5.2 - Algoritme A\*

```

Cerca ( $E_i, E_f$ , estats)
  pends := ( $E_i$ ,  $\theta$ ,  $h(E_i)$ );
  tracts :=  $\theta$ ;
  trobat := Fals;
  mentre no(trobat) i (pends  $\neq \theta$ ) fer
    [N, camí, valor] := Primer(pends);
    Eliminar_primer(pends);
    si (N= $E_i$ )
      llavors trobat := cert; solució = camí;
    sino per tot successor X de N fer
      si (X  $\notin$  tracts) llavors
        si (X  $\notin$  pends) llavors pends := Afegir_orden(pends, [X, camí+op,  $h(X)$ ]);
        Si millora sino si ( $\text{cost}(\text{camí}+op) < \text{cost}(\text{obtenirCamí}(X, \text{pends}))$ ) llavors
          camí a X pends := Sobreescure_orden(pends, [X, camí+op,  $h(X)$ ]); fsi
        fsi
      fter
      tracts := tracts + {N}
    fsi
  imentre
  si trobat llavors retorna (solució); sino retorna ("no existeix el camí");

```

Abstracte 7. Algoritme A\*.

Aquest algoritme segueix en gran part l'esquema del best first, l'única diferència és que en aquest cas, els costos dels nodes pendents es van actualitzant i ordenem la llista de pendents pel cost juntament amb l'heurística, d'aquesta manera sabem que la solució serà òptima. Procedirem llavors a explicar els canvis amb aquesta nova implementació.

### Successor nou

```

List<Node> succ = node.successors();
succ = this.filterStates(succ);
for (Node successor:succ){
  if (!tracts.stream().anyMatch(x -> x.isEquals(successor.getEstat()))){
    if (!pendents.stream().anyMatch(x -> x.getEstat().isEquals(successor.getEstat()))){
      successor.setHeuristica(h.heuristica(this.map, node.getEstat(), successor.getEstat(), this.end));
      successor.setTime(node.getTime()+this.calculTime(node.getEstat(),successor.getEstat()));
      pendants.add(successor);
      // Order by time+heuristic
      pendants = pendants.stream().sorted(Comparator.comparing(x -> (x.getTime()+x.getHeuristica())))
        .collect(Collectors.toList());
    }
  }
}

```

Abstracte 8. Tractament d'un nou successor.

El que farem en aquesta part serà assegurar-nos que l'estat no ha sigut ni tractat ni pendent, és a dir, un node verge. Bàsicament, farem el mateix que al best first.

## Successor pendent

```
else{ //If new path is less expensive
    double newTime = node.getTime()+this.calculTime(node.getEstat(),successor.getEstat());
    if (newTime < successor.getTime()){
        successor.setHeuristica(h.heuristica(this.map, node.getEstat(), successor.getEstat(), this.end));
        successor.setTime(newTime);
        pendants = pendants.stream().sorted(Comparator.comparing(x -> (x.getTime()+x.getHeuristica())))
            .collect(Collectors.toList());
    }
}
nIteration++;
```

*Abstracte 9. Tractament d'un successor que ja es treoba a pendants.*

En aquesta part ja sabem que el node està a la llista de tractats. El que haurem de fer en aquest cas és mirar si és el temps és millor per aquest camí, d'aquesta manera tots els nodes pendants tindran el millor camí possible cap a ells sempre.

## 6 – Heurístiques

Per poder provar els algorismes que se'ns demanava (Best-fit i A\*), hem realitzat 3 heurístiques diferents. Per construir les heurístiques disposem d'una interfície, on els hi passem els següents paràmetres:

```
public abstract double heuristica(List<Integer> map, State actual, State successor, State end);
```

Map -> el mapa amb tots els valors de les alçades.

Actual -> fa referència a l'estat en el qual estem situats actualment

Successor -> fa referència a l'estat que estem tractant (al que tenim pensat moure'ns a partir d'un operador).

End -> fa referència a l'estat final

**\*No totes les heurístiques usen tots els paràmetres.**

Un cop vist amb quins paràmetres treballem, podem explicar les diferents heurístiques.

### 6.1 - Heurística 1

```
public double heuristica(List<Integer> map, State actual, State successor, State end) {  
    int x, y;  
    x = end.getPosX() - successor.getPosX();  
    y = end.getPosY() - successor.getPosY();  
    x = x*x; y = y*y;  
    return x+y;  
}
```

*Abstracte 10. Heurística 1.*

Realitzem un tipus de Pitàgores a partir de les coordenades del successor i el final, per obtenir la distància entre aquests dos. Anomenem tipus, perquè no realitzem l'arrel quadrada, només la suma dels quadrats. Aquesta decisió s'ha pres per tal que l'heurística sigui més eficient.

Degut a aquest factor, aquesta heurística **no és admissible** respecte al temps. En cas de realitzar Pitàgores com toca (amb l'arrel quadrada), sí que seria admissible respecte al temps, perquè suposaríem el cost que ens costa arribar al final < al que realment és.

## 6.2 - Heurística 2

```
public double heuristica(List<Integer> map, State actual, State successor, State end) {
    int x, y;
    x = end.getPosX() - successor.getPosX();
    y = end.getPosY() - successor.getPosY();
    x = x * x; y = y * y;
    int actualAltitude = map.get(actual.getPosY()*Size.SIZE + actual.getPosX());
    int nextAltitude = map.get(successor.getPosY()*Size.SIZE + successor.getPosX());
    if (actualAltitude <= nextAltitude){
        return (x+y)*(1+(nextAltitude-actualAltitude));
    }
    else {
        return (x+y)*(1+(actualAltitude-nextAltitude)/(double)2);
    }
}
```

Abtracte 11. Heurística 2.

En aquest cas, realitzem la mateixa operació que en l'heurística anterior (Pitàgores sense arrel quadrada) i la multipliquem per una certa "importància".

Aquesta importància es calcula a partir del cost que ens suposa moure'ns de la casella actual (a la que estem situats) a la següent casella (la que consultem). Aquest cost es basa en el que es diu a l'enunciat:

- $1 + (\text{diferència d'alçades entre la casella destí i la d'origen})$ , si la diferència és positiva o 0.
- $1/2$  unitat, si la diferència és negativa.

En aquest cas, l'heurística **no és admissible** respecte al temps, supera per molt el cost que realment ens costaria arribar a l'estat final, ja que realitzem una operació entre valors elevats.

### 6.3 - Heurística 3

```
public double heuristica(List<Integer> map, State actual, State successor, State end) {
    int movX = Math.abs((end.getPosX()-successor.getPosX()));
    int movY = Math.abs((end.getPosY()-successor.getPosY()));
    int j = obtainPenalization(map,successor,end);
    return movX+movY+4*j;
}

public int obtainPenalization(List<Integer> map, State successor, State end){
    int j = 0;
    if ((successor.getPosY() == end.getPosY()) && (successor.getPosX() < (Size.SIZE-1)) &&
        (map.get((successor.getPosX()+1)+successor.getPosY()*Size.SIZE) == -1)){
        j++;
    }
    if ((successor.getPosX() == end.getPosX()) && (successor.getPosY() < (Size.SIZE-1)) &&
        (map.get(successor.getPosX()+(successor.getPosY()+1)*Size.SIZE) == -1)){
        j++;
    }
    return j;
}
```

Abtracte 12. Heurística 3.

Aquesta és l'heurística més simple, el que fem és calcular el número de moviment de l'estat que estem tractant a l'estat final (distància de Manhattan). Amb això el que fem, és calcular el cost total considerant que l'alçada de cada estat és el mateix (no varia). També es té en compte el cas en què hi ha un precipici entremig del camí, llavors s'hauria de rodejar (4 moviments). Això només passa si la coordenada x o la y de l'estat que estem tractant coincideix amb la del final.

Podem deduir, que és **admissible**, ja que el càlcul d'aquesta és inferior o igual al cost real. Això és així, ja que, com hem dit anteriorment, aquesta heurística calcula el cost tenint en compte que totes les alçades són iguals, llavors si ho comparem amb el cost real, serà < en cas que el camí tingui pendents distintes i = en cas de no haver-hi diferència d'alçades.









### 6.4 - Resum Heurístiques

- Heuristica 1 -> tipus de pitàgores. NO admissible.
- Heuristica 2 -> tipus de pitàgores\*importància (cost que ens suposa moure'ns a aquell estat). NO admissible.
- Heuristica 3 -> distància Manhattan. Admissible.

## 7 – Proves realitzades

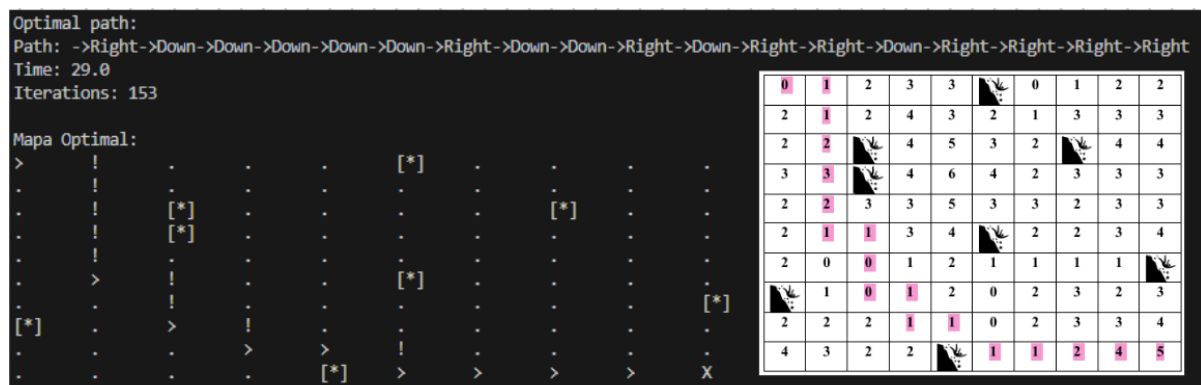
### 7.1 - Mapa proporcionat per l'enunciat

Disposem del següent mapa per realitzar les primeres proves:

0	1	2	3	3		0	1	2	2
2	1	2	4	3	2	1	3	3	3
2	2		4	5	3	2		4	4
3	3		4	6	4	2	3	3	3
2	2	3	3	5	3	3	2	3	3
2	1	1	3	4		2	2	3	4
2	0	0	1	2	1	1	1	1	
	1	0	1	2	0	2	3	2	3
2	2	2	1	1	0	2	3	3	4
4	3	2	2		1	1	2	4	5

Abstracte 13. Mapa proporcionat per l'enunciat.

Per saber quin és el cost del camí òptim, el que hem realitzat és realitzar una heurística que sempre retorni el mateix resultat, llavors recorrerà tots els estats possibles fins a trobar el millor cost, aquest és el resultat:



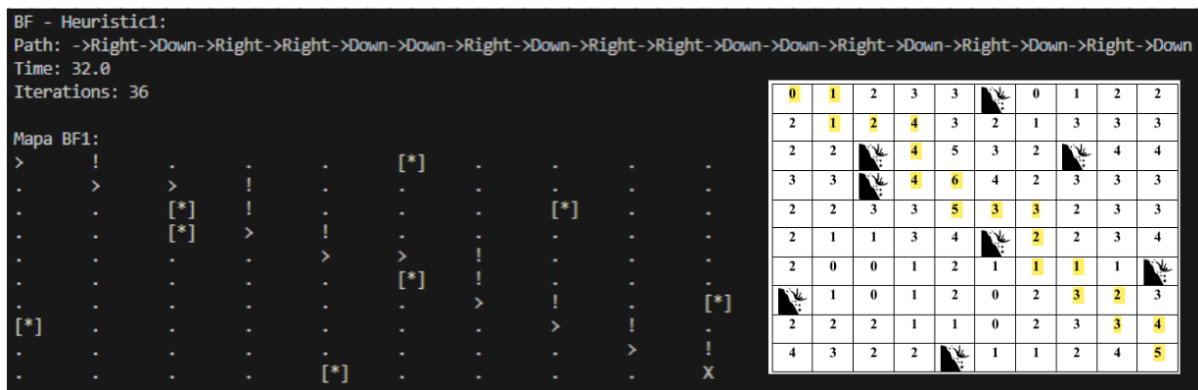
Abstracte 14. Solució òptima mapa 1.

Observem que el cost òptim és de 29, passant per 153 estats.

### 7.1.1 - Heurística 1 – Tipus de Pitàgores

#### Algoritme Best-first:

Un cop realitzada l'execució ens dona el següent resultat:



Abstracte 14. Solució Best-first amb heurística 1.

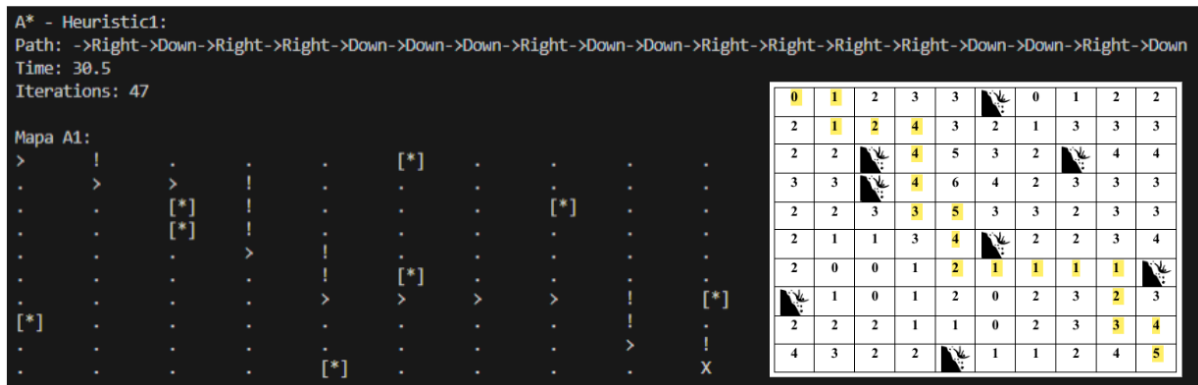
El resultat que ens dona és un **temps de 32** i amb **36 estats tractats**. Podem observa que **no és una solució òptima** respecte al temps, però encara així ens dona un resultat bastant bo i amb poques iteracions.

Sabem que el best-firts ordena la llista de pendents segons l'heurística. També tenim que l'heurística 1 obté el valor a partir de la "distància" entre l'estat que estem tractant i el final, llavors podem deduir varies coses:

- La solució és una línia en diagonal des de l'estat inicial al final, independentment de l'alçada dels estats.
- El número d'estats tractats no és molt elevat, ja que sempre ens anirem apropant a l'estat final.
- El cost que ens suposa moure'ns a l'estat final no és òptim, ja que per on passa la solució hi ha bastant diferència entre altures (pujem, baixem i tornem a pujar).

## Algoritme A\*:

Un cop realitzada l'execució ens dona el següent resultat:



Abstracte 15. Solució A\* amb heurística 1.

El resultat que ens dona és un **temps de 30.5** i amb **47 estats tractats**. Podem observa que **no és una solució òptima** respecte al temps, però encara així ens dona un resultat millor que amb el best-first i amb una mica més d'iteracions.

Podem deduir que aquests valors són correctes, ja que l'A\* ordena la llista de pendents pel cost que ens suposa arribar a aquell estat + l'heurística i perquè ens permet recalculer el cost d'un estat que està a aquesta llista. Això suposa més iteracions però un millor resultat.

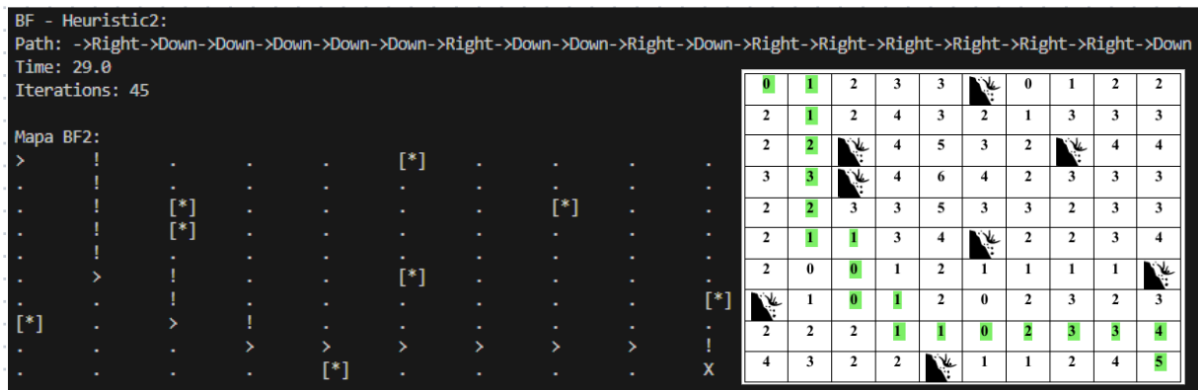
Com ja sabem, aquesta heurística no és admissible, llavors el seu resultat no és òptim.



### 7.1.2 - Heurística 2 – Tipus de Pitàgores\*importància

### Algorithme Best-first:

Un cop realitzada l'execució ens dona el següent resultat:

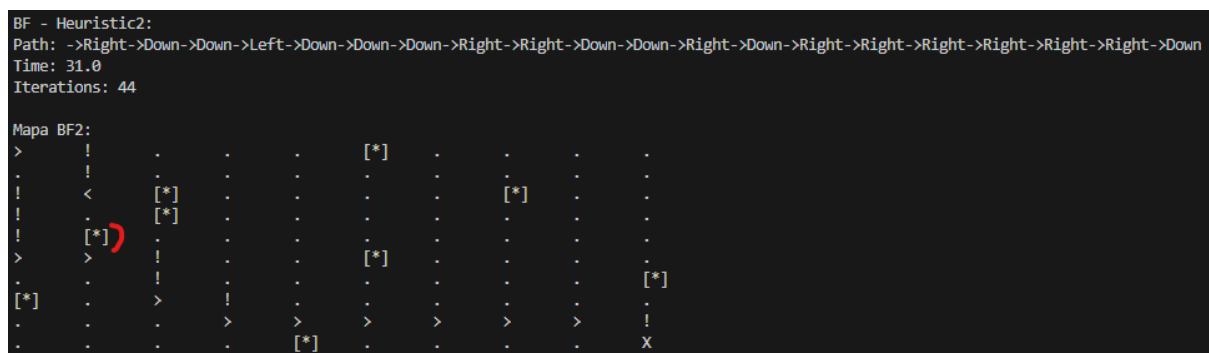


*Abstracte 16. Solució Best-first amb heurística 2.*

El resultat que ens dona és un **temps de 29** i amb **45 estats tractats**. Podem observa que **és una solució òptima** respecte al temps.

En l'anterior heurística el que fem és triar un camí el qual s'apropa, llavors sempre anirà en diagonal. En aquest cas, ho multipliquem per la importància, fent que sigui una mica més intel·ligent i triï un camí menys costos.

Observant el resultat, podem deduir que l'heurística 2 és bona en aquest mapa, ja que amb poques iteracions podem obtenir el resultat òptim. Això es deu a que en el camí resultant, no ens obliguen a retrocedir respecte a l'estat final (pujar o anar a l'esquerra). Això ho podem observar introduint un precipici en mig del camí, d'aquesta manera:

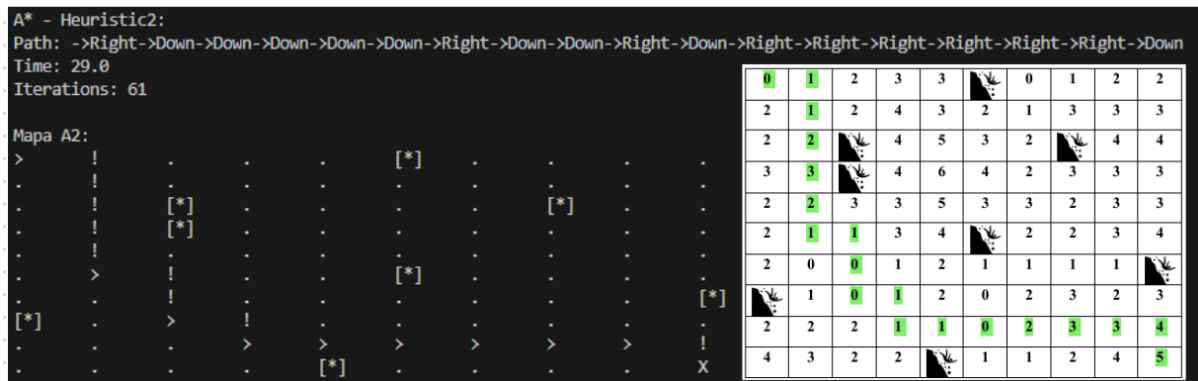


*Abstracte 17. Prova modificant mapa 1.*

Realitzant aquest petit canvi, es veu com el cost resultat ja no és òptim.

### Algoritme A\*:

Un cop realitzada l'execució ens dona el següent resultat:



Abstracte 18. Solució A\* amb heurística 2.

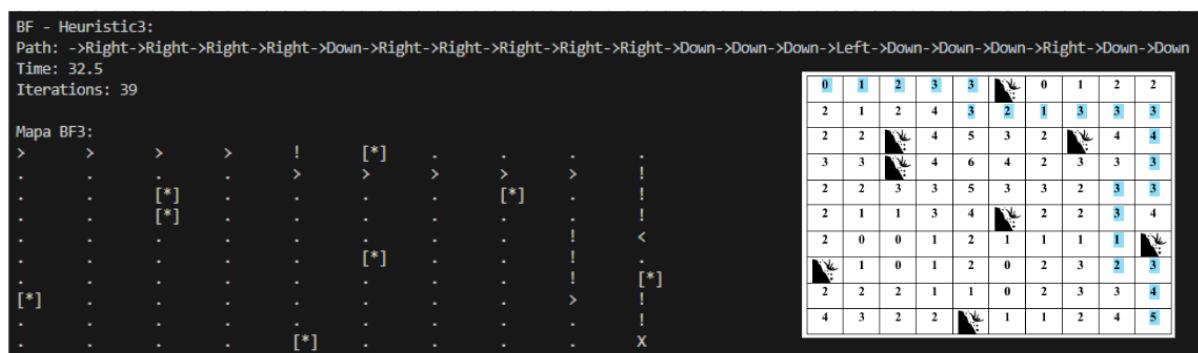
El resultat que ens dona és un **temps de 29** i amb **61 estats tractats**. Podem observa que **és una solució òptima** respecte al temps.

Per l'algoritme A\* seguim la mateixa lògica que amb el Best-first, tenim una solució òptima però és a causa de l'estructura del mapa i el camí resultant. En aquest cas, el temps no es pot millorar, però el número d'iteracions sí que augmenta, ja que comprovem més estats, a causa d'ordenar la llista pel cost+heurística.

### 7.1.3 - Heurística 3 – Distància de Manhattan

#### Algoritme Best-first:

Un cop realitzada l'execució ens dona el següent resultat:



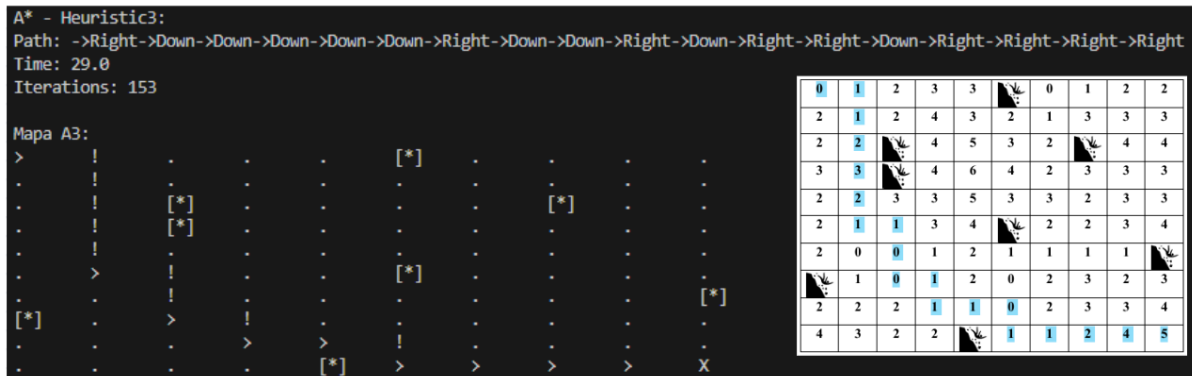
Abstracte 19. Solució Best-first amb heurística 3.

El resultat que ens dona és un **temps de 32.5** i amb **39 estats tractats**. Podem observa que **no és una solució òptima** respecte al temps.

Per aquest algoritme i aquesta heurística, sempre triarà el camí que s'apropa més a l'estat final, dreta o baix (tenint en compte els pendents), llavors el número d'iteracions és baix. En aquest mapa, això no suposa que el camí triat serà el millor, llavors la solució no és òptima.

### Algoritme A\*:

Un cop realitzada l'execució ens dona el següent resultat:



Abstracte 20. Solució A\* amb heurística 3.

El resultat que ens dona és un **temps de 29** i amb **153 estats tractats**. Podem observa que és una **solució òptima** respecte al temps.

En aquest cas, l'heurística és admissible respecte al temps, llavors ens dona una solució òptima. Respecte al nombre d'iteracions, podem veure que realment el que fa és comprovar tots els possibles estats. Podem deduir que això és així perquè la deducció del cost de l'estat que estem tractant és molt baixa, respecte al cost que realment és. Llavors, com que A\* ordena pel cost+heurística, els estats anteriors tindran un valor inferior, i es tractaran abans que els que s'apropen més a l'estat final. Aquí un petit exemple: (valor en vermell el cost per arribar a aquell estat i en blau el de l'heurística)

$$\text{Coordenada (0,0)} \rightarrow \text{cost} = 0 + 9 + 9 = 18$$

$$\text{Coordenada (1,1)} \rightarrow \text{cost} = 2 + 9 + 8 = 19$$

$$\text{Coordenada (2,1)} \rightarrow \text{cost} = 2 + 2 + 1 + 7 + 8 = 20$$

$$\text{Coordenada (1,4)} \rightarrow \text{cost} = 2 + 1 + 2 + 2 + 1,5 + 8 + 5 = 21,5$$

Amb aquest exemple, podem concloure que aquesta heurística és bona per quant la diferència entre alçades es baixa o mínima (mapa pla). En aquest mapa no és del tot bona, encara que ens treu el resultat òptim realitza un recorregut de tot el mapa.

## 7.2 -Mapa propi

1	3	3	2	1	2	2	1	1	2
2	P	2	2	3	2	P	P	P	2
2	6	6	4	2	4	P	5	5	P
P	6	6	1	4	5	P	4	5	4
5	5	4	2	4	P	1	2	4	5
4	3	1	3	P	1	1	2	2	4
5	4	2	2	3	2	5	4	2	4
6	5	4	1	2	3	4	P	1	1
6	6	5	4	3	4	5	5	P	2
7	7	7	6	5	5	6	7	P	2

Abstracte 21. Mapa propi.

Per saber quin és el cost del camí òptim, el que hem realitzat és realitzar una heurística que sempre retorni el mateix resultat, llavors recorrerà tots els estats possibles fins a trobar el millor cost, aquest és el resultat: **Time: 34.0 Iterations: 136**

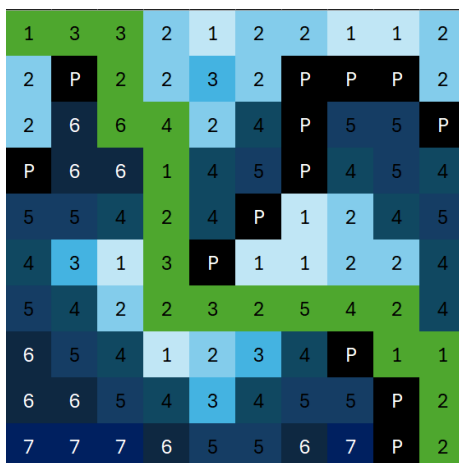
1	3	3	2	1	2	2	1	1	2
2	P	2	2	3	2	P	P	P	2
2	6	6	4	2	4	P	5	5	P
P	6	6	1	4	5	P	4	5	4
5	5	4	2	4	P	1	2	4	5
4	3	1	3	P	1	1	2	2	4
5	4	2	2	3	2	5	4	2	4
6	5	4	1	2	3	4	P	1	1
6	6	5	4	3	4	5	5	P	2
7	7	7	6	5	5	6	7	P	2

Abstracte 22. Solució òptima mapa propi.

### 7.2.1 - Heurística 1 – Tipus de Pitàgores

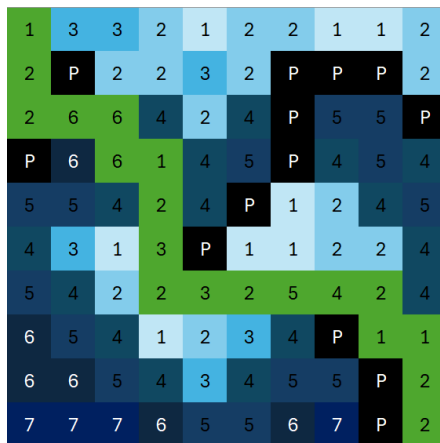
En aquesta heurística, veiem que l'algoritme va el més diagonal possible. L'única diferència és que l'A\* escull el camí tenint en compte el pendent també, això es deu a la naturalesa de l'algoritme en si, referint-nos que a l'hora d'ordenar els nodes pendents no només es fixa en l'heurística sinó que també en el temps, que es causat pel pendent.

**Algoritme Best-first:** Time: 37.0 Iterations: 40



Abstracte 23. Solució Best-first amb heurística 1.

**Algoritme A\*:** Time: 35.5 Iterations: 41



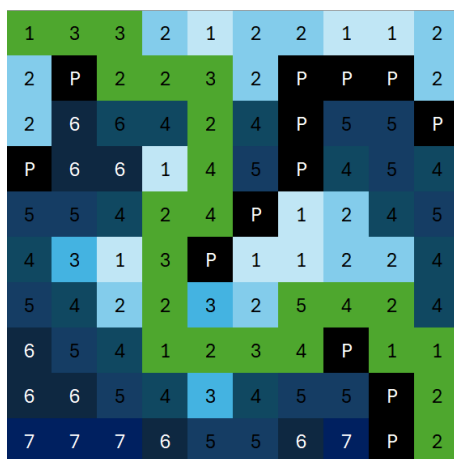
Abstracte 24. Solució A\* amb heurística 1.

### 7.2.2 - Heurística 2 – Tipus de Pitàgores\*importància

Amb aquesta heurística veiem que no va el més diagonal possible, sinó que també té en consideració els pendents, això s'implementa a l'hora de multiplicar la distància amb la importància, és a dir, el pendent. Veiem, però que encara així no té en compte totalment el temps, ja que hi ha escenaris en els quals no agafa la menor pendent.

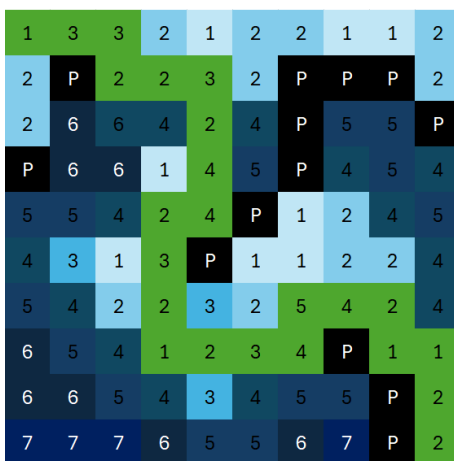
Podem observar, que en comparació al mapa anterior, aquesta heurística no ens proporciona el resultat òptim.

**Algoritme Best-first:** Time: 38.0 Iterations: 55



Abstracte 25. Solució Best-first amb heurística 2.

**Algoritme A\*:** Time: 38.0 Iterations: 74

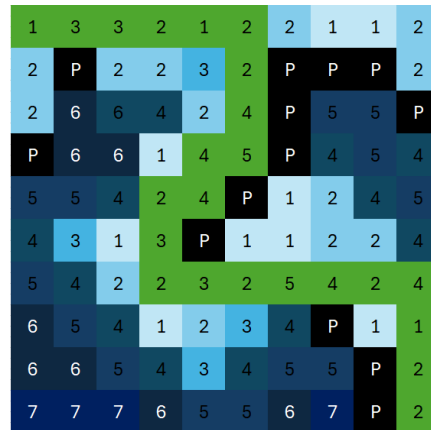


Abstracte 26. Solució A\* amb heurística 2.

### 7.2.3 - Heurística 3 – Distància de Manhattan

En aquest cas passa el mateix que amb el mapa anterior, anem a veure-ho amb més detall:

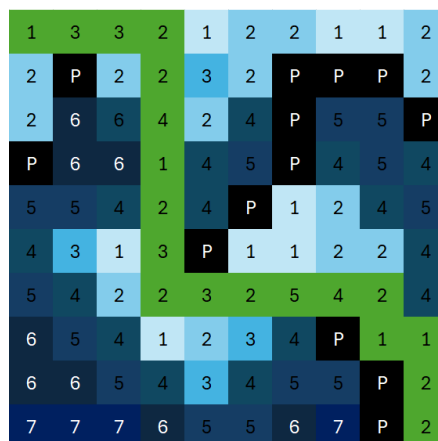
**Algoritme Best-first:** Time: 42.5 Iterations: 46



Abstracte 27. Solució Best-first amb heurística 3.

Per aquest algoritme i aquesta heurística, podem observar com sempre agafa el camí de la dreta, això fa que a causa dels diferents precipicis hàgim de tornar enrere, fent que la solució no sigui òptima.

**Algoritme A\*:** Time: 34.0 Iterations: 128



Abstracte 28. Solució A\* amb heurística 3.

Per l'A\*, podem observar com tenim la solució òptima, ja que l'heurística és admissible respecte al temps, però que realitza el màxim d'iteracions possibles. Això és així, pel que es comentava en l'anterior mapa, tenim que la nostra estimació és molt baixa en comparació al temps real, fent que els anterior nodes tinguin un valor d'heurístic menor i que es tractin abans que l'estat amb el camí correcte.

## 7.3 - Hill Climbing









Sabem que aquest algoritme és similar al Best-first, però en cap moment guarda algun estat a pendents, llavors únicament es pot realitzar amb problemes i heurístiques les quals sabem que per cada operador que realitzem ens apropem a l'estat final (el valor de l'heurística cada vegada és menor).

En aquest apartat raonarem si les 3 heurístiques funcionarien per cada un dels mapes amb aquest algoritme.

### 7.3.1 - Mapa proporcionat per l'enunciat

**Heurística 1** – Tipus de Pitàgores -> **DEPEN**

Per aquest cas, l'heurística únicament usa la distància entre l'estat que estem tractant i el final per fer el càlcul (no usa ni les alçades i els pendents). Davant d'aquesta situació, sabem que sempre hi haurà 2 estats que estaran a la mateixa distància de l'estat final. Davant d'aquesta situació, depèn de la política d'elecció. Observem un exemple:

0	1	2	3	3		0	1	2	2
2	1	2	4	3	2	1	3	3	3
2	2		4	5	3	2		4	4
3	3		4	6	4	2	3	3	3
2	2	3	3	5	3	3	2	3	3
2	1	1	3	4		2	2	<del>3</del>	4
2	0	0	1	2	1	1	1	1	
	1	0	1	2	0	2	3	2	3
2	2	2	<del>1</del>	1	0	2	3	3	4
4	3	2	2		1	1	2	4	5

Abstracte 29. Camins que no ens permetrien usar hill climbing amb l'heurística1.

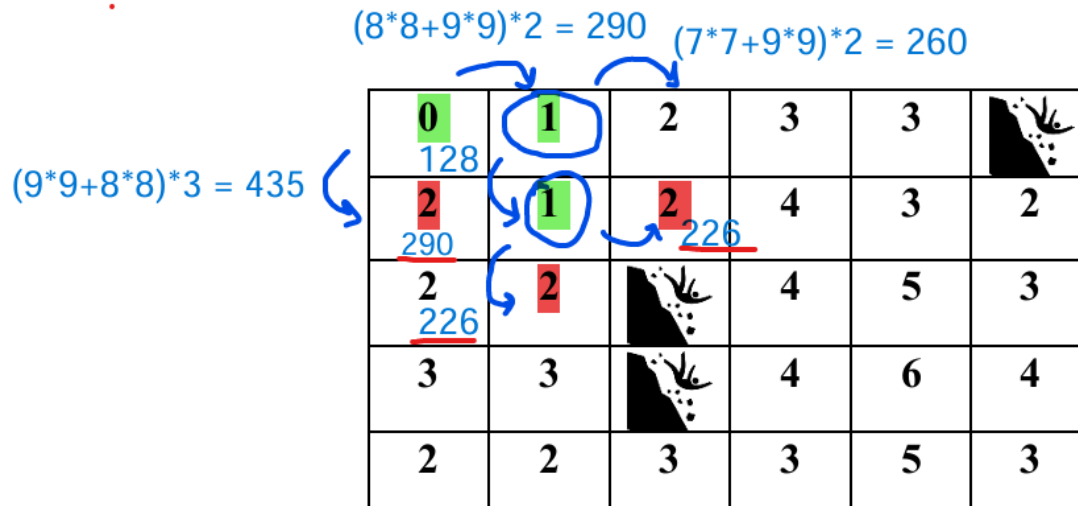
En l'exemple veiem que triem una política de tot a la dreta o tot a cap a baix, llavors arribem a un punt en el qual hem de retrocedir a causa d'un pendent, fent que l'heurística del nou estat que estem consultant sigui superior a l'actual.

En canvi, si realitzem el camí que ens dona la solució del Best-first, podríem arribar a la solució sense cap problema.



## Heurística 2 – Tipus de Pitàgores -> NO

Per aquesta heurística, sí que usem el pendent entre els estats per calcular el seu valor, llavors analitzem el seu camí.



Abstracte 30. Càlcul heurística 2 mapa 1.

Observem que ens hem pogut moure a 2 estats, però després, tots els següents estats tenen un valor d'heurística superior al de l'estat actual ( $128 < 226$  i  $290$ ). Llavors podem concloure que aquesta heurística, no és vàlida per aquest algoritme i per aquest mapa. També podem deduir, que una de les condicions perquè un mapa sigui vàlid és que un cop tenen una mateixa alçada repetida 2 vegades en el camí, l'alçada es manté en aquell valor, inclòs el final.

## Heurística 3 - Distància de Manhattan -> DEPEN







Per aquest cas, passa el mateix que amb la primera heurística. Observem que passa si seguim la solució que ens dona el Best-fit amb aquesta heurística:

0	1	2	3	3	Muntanya	0	1	2	2
2	1	2	4	3	2	1	3	3	3
2	2	Muntanya	4	5	3	2	Muntanya	4	4
3	3	Muntanya	4	6	4	2	3	3	3
2	2	3	3	5	3	3	2	3	3
2	1	1	3	4	Muntanya	2	2	3	4
2	0	0	1	2	1	1	1	1	Muntanya
Muntanya	1	0	1	2	0	2	3	2	3

*Abstracte 31. Anàlisi solució mapa 1 heurística 3.*

Observem que agafant aquest camí, arribem a un punt on un precipici ens afegeix més cost i la millor opció es anar a l'esquerra, però encara així el valor és superior.

Deduïm que aquesta heurística és vàlida per aquest algoritme segons el seu criteri de selecció. Els 2 criteris més simples serien agafar sempre el camí cap a la dreta o cap a baixa, el primer és el que hem vist anteriorment, llavors anem a veure el segon (sempre cap a baix):

0	1	2	3	3	
2	1	2	4	3	2
2	2		4	5	3
3	3		4	6	4
2	2	3	3	5	3
2	1	1	3	4	
2	0	0	1	2	1
	1	0	1	2	0
2	2	2	1	1	0
4	3	2	2		1

+2  
+4

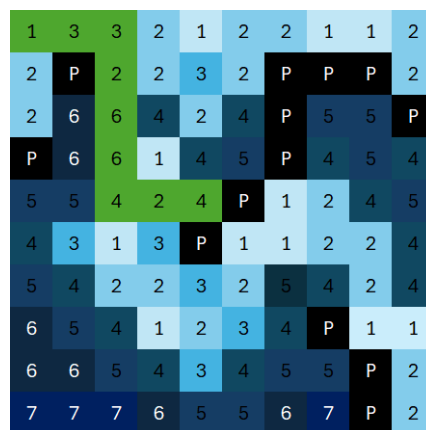
*Abstracte 32. Solució invàlida mapa 1 heurística 3.*

Observem que en els dos casos, tenim un precipici que ens obliga a donar la volta (cost +4) o allunyar-nos de l'estat final (+2). Llavors podem deduir que aquesta heurística no és vàlida per aquests algoritme i aquest mapa. En cas de voler que sigui vàlida, s'ha de fer un criteri de selecció més complex.

### 7.3.1 - Mapa proporcionat per l'enunciat

#### Heurística 1 – Tipus de Pitàgores -> **DEPEN**

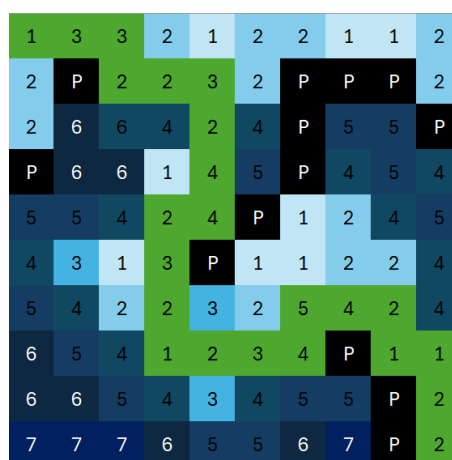
En aquesta heurística, en l'únic que es fixa és en la distància. El tema és que realment sense a implementar l'algoritme no podem saber quin camí prendrà, ja que hi ha casos ambigus. El que sí que sabem és que si fa un recorregut similar al de la imatge, no trobarà solució, això es deu a l fet que hi haurà un mur de nodes tractats pel qual no podrà passar. Cal dir, però que si prenem de referència les solucions donades pels altres dos algoritmes podem concloure que sí que trobarà la solució.



Abstracte 33. Camí encallat.

#### Heurística 2 – Tipus de Pitàgores -> **NO**

En aquesta heurística, prenent per referència la solució del best first podem deduir que al principi ha intentat moure's per baix, però el 6 l'ha fet canviar de camí. Això amb el hill climbing no passaria, és a dir, es quedaria encallat.



Abstracte 25. Solució Best-first amb heurística 2.

### Heurística 3 - Distància de Manhattan -> **DEPEN**

En aquesta heurística si segueix una estratègia similar a la del best first, hauria de trobar la solució. Però el mapa es va fer amb la intenció de crear camins sense sortida, en aquest cas veiem que dalt a la dreta no hi ha sortida, per la qual cosa probablement intentarà entrar per aquí i no podrà tornar.

1	3	3	2	1	2	2	1	1	2
2	P	2	2	3	2	P	P	P	2
2	6	6	4	2	4	P	5	5	P
P	6	6	1	4	5	P	4	5	4
5	5	4	2	4	P	1	2	4	5
4	3	1	3	P	1	1	2	2	4
5	4	2	2	3	2	5	4	2	4
6	5	4	1	2	3	4	P	1	1
6	6	5	4	3	4	5	5	P	2
7	7	7	6	5	5	6	7	P	2

*Abstracte 27. Solució Best-first amb heurística 3.*