

PONTIFICIA UNIVERSIDAD CATÓLICA DE VALPARAÍSO  
FACULTAD DE INGENIERÍA  
ESCUELA DE INGENIERÍA INFORMÁTICA

## **Estructura de datos Avanzada y Algoritmos**

**GONZALO TELLO VALENZUELA**

Docente:  
Ignacio Araya Zamorano  
Alen Figueroa  
Javier Peña

2do Semestre, 2020

# Índice general

<b>Lista de Figuras</b>	<b>v</b>
<b>Lista de Tablas</b>	<b>vi</b>
<b>1 Grafos y resolución de problemas</b>	<b>1</b>
1.1 Problema de la mochila . . . . .	1
1.1.1 ¿Cómo resolverlo? . . . . .	1
1.1.2 Aplicaciones del problema . . . . .	2
1.2 Problema de entregas (TSP) . . . . .	2
1.2.1 ¿Cómo resolver? . . . . .	2
1.2.2 Aplicaciones del problema . . . . .	3
1.3 ¿En qué se parecen estos problemas? . . . . .	3
1.4 Diagrama de estados (grafo) . . . . .	3
1.4.1 ¿Qué es un grafo? . . . . .	4
1.4.2 ¿Cómo resolver usando un grafo? . . . . .	4
1.5 Problema de 8 reinas . . . . .	4
1.6 ¿Cómo modelar el problema en un PC? . . . . .	6
1.6.1 Funciones para construir el grafo . . . . .	7
1.6.2 Ejemplo de 8 reinas . . . . .	7
1.7 Resolución de problemas usando grafos . . . . .	8
1.8 Algoritmos de búsqueda . . . . .	9
1.8.1 Búsqueda en profundidad . . . . .	9
1.8.2 Búsqueda en anchura . . . . .	10
1.8.3 Observaciones . . . . .	11
1.8.4 Best-first . . . . .	12
1.8.5 ¿Cómo diseño una heurística? . . . . .	12
1.8.5.1 ¿Cómo mejorar la heurística? . . . . .	14
1.8.6 ¿Cómo crear heurísticas para evaluar estados? . . . . .	15
1.8.7 Heurísticas admisibles . . . . .	15

1.8.7.1	Ejemplos . . . . .	15
1.8.8	Algoritmo de búsqueda A* . . . . .	15
1.9	Estrategias incompletas de búsqueda . . . . .	16
1.9.1	Greedy . . . . .	16
1.9.2	GRASP - Greedy Randomized Search Procedure . . .	16
1.9.3	Beam Search . . . . .	16
1.9.4	Ant Colony Optimization . . . . .	17
1.9.5	¿Cómo evaluar los estados? . . . . .	17
1.10	Métodos de búsqueda local . . . . .	17
1.10.1	Hill Climbing . . . . .	18
1.11	Algoritmos evolutivos . . . . .	19
<b>2</b>	<b>Búsqueda adversaria</b>	<b>20</b>
2.1	Game Tree Search . . . . .	20
2.1.1	Introducción . . . . .	20
2.1.2	Minimax . . . . .	22
2.1.2.1	Rekursivo . . . . .	22
2.1.2.2	Iterativo . . . . .	22
2.1.2.3	Observaciones . . . . .	24
2.1.3	AlphaBeta - Prunning . . . . .	24
2.1.3.1	Observaciones . . . . .	24
2.1.3.2	¿Cómo implemento el algoritmo? . . . . .	25
2.1.4	Mejoras/variantes . . . . .	25
2.1.5	Extensión de minimax . . . . .	25
2.2	Montecarlo Tree Search . . . . .	26
2.2.1	¿Qué es Montecarlo Tree Search? . . . . .	26
2.2.2	¿Cómo funciona? . . . . .	26
2.2.3	Resumen . . . . .	26
2.2.4	Simulación . . . . .	27
2.2.5	Ventajas y desventajas . . . . .	27
2.2.6	Pasos de Montecarlo Tree Search . . . . .	27
2.2.7	Algoritmo MonteCarlo Tree Search . . . . .	28
2.2.8	Política de selección (UCT) . . . . .	28
2.2.9	¿Qué acción retornar (BestChild)? . . . . .	29
2.2.10	Pseudocódigo . . . . .	29
2.2.11	Mejores/variantes (selección) . . . . .	30
2.3	Aprendizaje reforzado . . . . .	31
2.3.1	Definiciones . . . . .	31
2.3.1.1	¿Qué es el aprendizaje automatizado? . . . .	31
2.3.1.2	¿Qué es el aprendizaje por refuerzo? . . . .	31

2.3.2	¿Cómo funciona?	31
2.3.3	Política de gradiente ascendiente	32
2.3.4	Maneras de aprender	32
<b>3</b>	<b>Indexación de datos espaciales</b>	<b>35</b>
3.1	Modelando problemas usando estructuras espaciales	35
3.1.1	Problema de turismo	35
3.2	Estructuras espaciales	37
3.2.1	Tipos de estructuras espaciales	37
3.2.2	AABB Tree (Axis-Aligned Bounding Box Tree)	38
3.2.2.1	Estructura básica	38
3.2.2.2	Inserción	39
3.2.3	Segmentation Tree	41
3.2.3.1	Estructura básica	41
3.2.3.2	Insertion	41
3.2.3.3	Búsqueda	42
3.2.4	KD - Tree	43
3.2.4.1	Estructura básica	43
3.2.5	QuadTree	44
3.2.6	OctTree	44
3.2.7	Local-sensitive Hashing	45
3.2.7.1	Función Hash	45
3.2.7.2	Estructura básica	46
3.2.7.3	Hash	46
3.3	Algoritmos espaciales	47
3.3.1	Vecino(s) más cercano(s)	47
3.3.2	Detección de colisiones	50
3.3.3	¿QuadTree y OctTree?	51
3.3.4	Filtros de imágenes	51
<b>4</b>	<b>Perceptron multicapa</b>	<b>52</b>
4.1	Introducción a Redes neuronales	52
4.1.1	Problema 1	52
4.1.1.1	Aplicaciones del problema	53
4.1.2	Problema 2	53
4.1.2.1	Aplicaciones del problema	54
4.1.3	¿En qué se parecen estos problemas?	54
4.1.4	Cambio de paradigma	54
4.1.5	Red Neuronal	55
4.1.6	La Neurona	56

4.1.7	¿Cómo funciona una red neuronal? . . . . .	57
4.1.8	¿Cómo aprende una red neuronal? . . . . .	59
4.2	Backpropagation . . . . .	59
4.2.1	Recapitulación . . . . .	59
4.2.2	Entrenamiento . . . . .	60
4.2.3	Optimizadores . . . . .	60
4.2.4	Descenso del gradiente . . . . .	61
4.2.4.1	Descenso del gradiente estocástico . . . . .	61
4.2.4.2	Algoritmo de entrenamiento . . . . .	61
4.2.4.3	Algoritmo . . . . .	62
4.2.4.4	Regla Delta . . . . .	62
4.3	Otros tipos de redes neuronales . . . . .	65
4.3.1	Recapitulación . . . . .	65
4.3.2	Problemas con las redes neuronales . . . . .	66
4.3.3	Problemas con el perceptrón . . . . .	66
4.3.4	Redes Neuronales Convolucionales . . . . .	67
4.3.5	Neurona convolucional . . . . .	67
4.3.6	Max Pooling . . . . .	69
4.3.7	Red Neuronal recurrente . . . . .	70
4.3.8	Redes generativas adversarias . . . . .	71
4.3.9	Red Deconvolucional . . . . .	72
4.3.9.1	Entrenamiento . . . . .	72
4.3.10	Redes generativas adversarias . . . . .	73
<b>5</b>	<b>Indexación de Motores de búsqueda</b>	<b>74</b>
5.1	Motores de búsqueda . . . . .	74
5.2	Predictor léxico . . . . .	75
5.2.1	Trie . . . . .	76
5.3	Analizador Léxico . . . . .	79
5.3.1	Optimizador temporal . . . . .	80
5.3.2	Contexto del documento . . . . .	80
5.3.3	Hot-encoding . . . . .	80
5.3.4	Word2Vec . . . . .	81
5.3.5	Inverted Index . . . . .	82
5.3.6	Document-term Matrix . . . . .	82

# Lista de Figuras

1.1	Tablero de 8 reinas . . . . .	5
1.2	Diagrama de estados para 4 reinas . . . . .	5
1.3	Ejemplo en Valor actual en heurística . . . . .	13
1.4	Ejemplo Cubo Rubik . . . . .	13
1.5	Ejemplo Suma de conflictos . . . . .	14
1.6	Ejemplo de distancia recorrida . . . . .	14
1.7	Representación greedy . . . . .	16
2.1	Política de estados . . . . .	20
2.2	Game Tree Search . . . . .	21
2.3	Minimax Recursivo . . . . .	22
2.4	Alpha Beta Prunning . . . . .	24
4.1	Sigmoide con $b = 0$ . . . . .	59
4.2	Sigmoide con $b = -2$ . . . . .	59
4.3	Visualización de optimizadores . . . . .	61
5.1	Predictor léxico - Google . . . . .	75
5.2	Trie . . . . .	76
5.3	Ejemplo Trie . . . . .	77
5.4	Analizador léxico . . . . .	79

# Lista de Tablas

# Capítulo 1

## Grafos y resolución de problemas

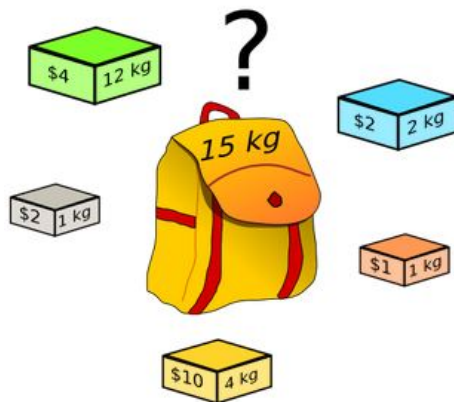
### 1.1. Problema de la mochila

Supongamos que somos parte de una banda de ladrones y queremos robar un museo. Llenamos una mochila para guardar los ítems que vamos sacando y esta mochila soporta un peso máximo.

Cada ítem tiene un valor y un peso asociados

Nuestro objetivo es cololar ítems en la mochila tratando de maximizar el valor total.

#### 1.1.1. ¿Cómo resolverlo?



Podríamos

- Comenzar con la mochila vacía
- Ir agregando ítems de a uno.
- Cuando no podamos agregar más elementos nos detenemos y guardamos el valor alcanzado.
- Repetimos para ver si tenemos más suerte.

### 1.1.2. Aplicaciones del problema

- **Estudiar para una prueba:** tengo un tiempo limitado (mochila) para estudiar los capítulos del libro (ítems). ¿Qué capítulos debería estudiar para maximizar mi nota?
- En una tienda, ¿Qué productos me conviene mantener en la vitrina?
- Me voy de viaje, ¿Qué cosas debería llevar en la maleta?
- Visitarás París por un par de días, ¿Qué monumentos/museos/lugares deberías visitar?

## 1.2. Problema de entregas (TSP)

Una compañía de entregas diariamente cuenta con un vehículo y una lista de productos que debe despachar a distintas direcciones.

El objetivo del problema es encontrar una ruta que minimice el coste del viaje (por ejemplo, la ruta con la mínima distancia).

### 1.2.1. ¿Cómo resolver?

- Comenzamos en el punto de partida.
- Agregamos entregas una a una (pueden ser las más cercanas de la última entrega).
- Al terminar con todas las entregas volvemos al punto de partida.
- Evaluamos la ruta y volvemos a empezar.

### 1.2.2. Aplicaciones del problema

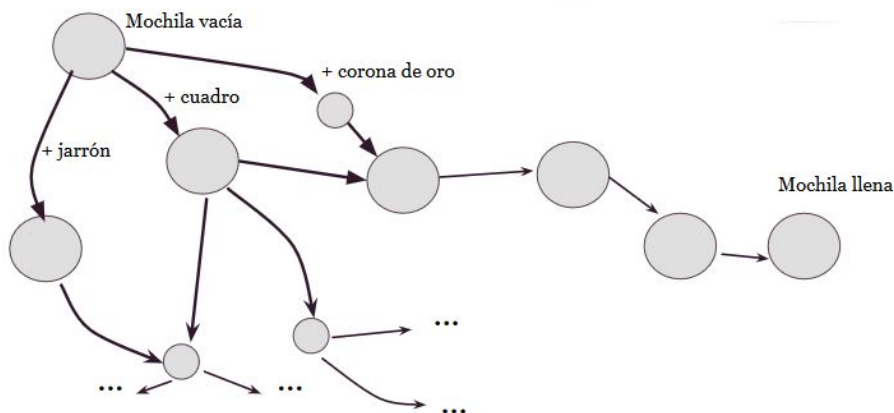
- Servicios de entrega que cuentan con varios vehículos intentan minimizar el costo de las rutas pero también la cantidad de vehículos. Además, se pueden agregar restricciones de tiempo para las entregas.
- En el proceso de ensamblado de microchips, se intenta minimizar el tiempo generando una ruta óptima para el brazo robótico que coloca los componentes.
- Estarás un fin de semana en Roma, ¿en qué orden deberías visitar los monumentos/lugares para perder el menor tiempo posible?

### 1.3. ¿En qué se parecen estos problemas?

La resolución de estos problemas se pueden representar usando:

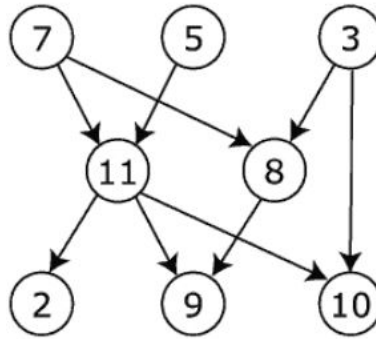
- Un estado inicial
- Un conjunto de reglas o acciones para pasar de un estado a otro.
- Un conjunto de estados posibles
- Un estado final

### 1.4. Diagrama de estados (grafo)



#### 1.4.1. ¿Qué es un grafo?

- Un conjunto de nodos/estados.
- Cada nodo puede estar conectado a uno o más nodos adyacentes.
- Conexiones se llaman arcos.



#### 1.4.2. ¿Cómo resolver usando un grafo?

Existen distintas técnicas para analizar el grafo y resolver problemas. Por ejemplo, podemos:

- Buscar una o todas las soluciones 'alcanzables' desde un estado inicial.
- Buscar la mejora solución alcanzable desde un estado inicial (e.g, la que minimiza la distancia o el costo).
- Guardar el camino a esta solución.
- La acción que maximiza la 'probabilidad de alcanzar un estado final 'favorable'.

### 1.5. Problema de 8 reinas

El problema de las 8-reinas consiste en colocar 8 reinas en un tablero sin que se amenacen entre ellas.

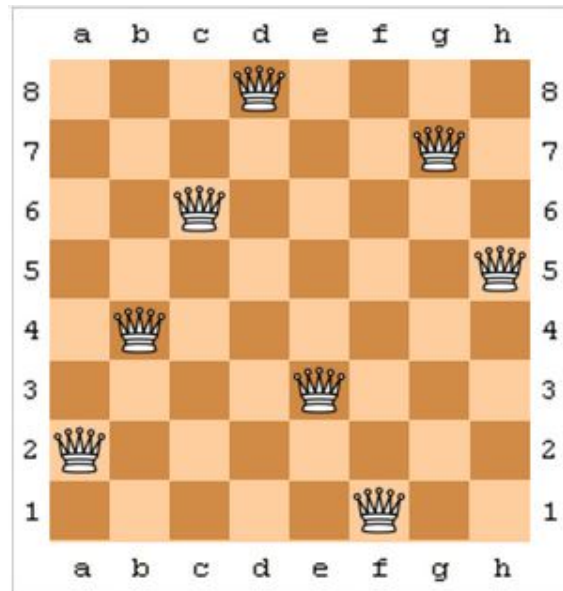


Figura 1.1: Tablero de 8 reinas

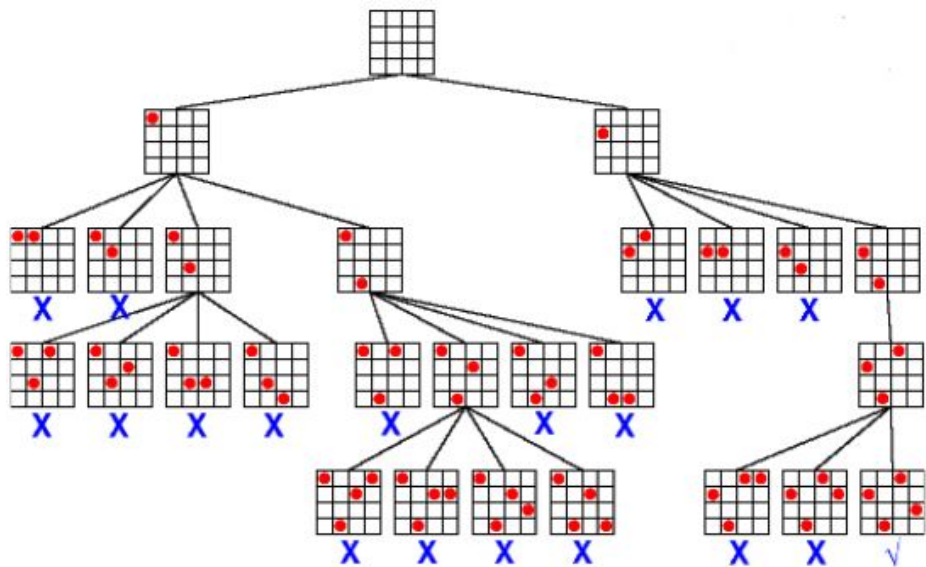


Figura 1.2: Diagrama de estados para 4 reinas

## 1.6. ¿Cómo modelar el problema en un PC?

Conociendo el modelo, podemos implementar un grafo implícito. Es decir, un grafo que se vaya construyendo automáticamente a medida que se va explorando.

La idea es que luego podamos buscar soluciones con algoritmos de búsqueda en grafos.

Primero definimos la estructura de los nodos o estados, ejemplo (en c++):

```
class State{ //8puzzle
public:
    int puzzle[3][3];
    int i,j; //espacio vacio
};

class State{ //mochila
public:
    list<int> items;
    int carga_actual;
};
```

También es recomendable definir la estructura de las acciones (o arcos del grafo). Por ejemplo:

```
class Action{ //8puzzle
public:
    int move; //arriba(1), abajo(2)
};

class Action{ //n-reinas
public:
    int i,j;
    //posicion en donde se agregara la reina
};

class Action{ //carga container
public:
    int x,y,z;
    Box caja;
};
```

### 1.6.1. Funciones para construir el grafo

Para poder construir el grafo necesitamos implementar un par de funciones

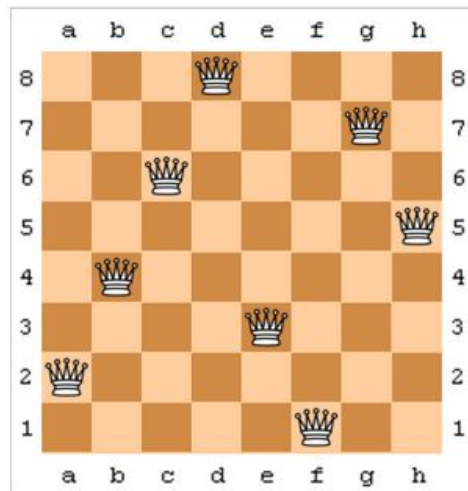
```
list<Action> get_actions(State& state)
```

Retorna todas las acciones que se pueden realizar a partir del estado.

```
State transition(State& state, Action& action)
```

Retorna el estado que resulta de aplicar la acción.

### 1.6.2. Ejemplo de 8 reinas



#### ■ Estado.

Arreglo de 8 elementos. El i-ésimo elemento indica la fila de la i-ésima reina (izq->der). En la imagen se representa el estado 2468175.

```
class State{  
public:  
    array<int ,N> cols  
};
```

#### ■ Acción.

Agregar una reina en la siguiente columna.

```

class Action{
public:
    //columna y fila en la que se ubica
    //la siguiente reina
    int col, row;
};

```

#### ■ Otros métodos

```

State transition(State& s, Action& a){
    State new_state = s;
    new_state.cols[a.col] = a.row;
    return new_state;
}

list<Action> get_actions(state& s){
    int i, j;
    list<Action> actions;
    for(i = 0; i < N; i++){
        if(s.col[i] == 0){
            for(j = 1; j < N+1; j++){
                s.cols[i] = j;
                if(is_valid(s)){
                    Action a;
                    a.col=i;
                    a.row=j;
                    actions.push_back(a);
                }
            }
            s.cols[i]=0;
            return actions;
        }
    }
    return actions;
}

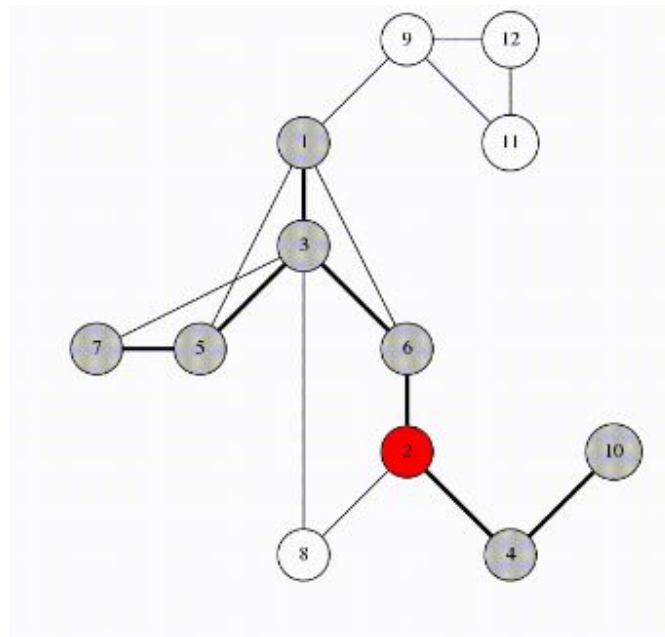
```

## 1.7. Resolución de problemas usando grafos

Si tenemos un problema cuyo proceso de resolución se puede modelar con:

- Un estado inicial
- Un conjunto de acciones para pasar de un estado a otro.
- Un conjunto de estados posibles y un estado final

Entonces lo podemos modelar con un grafo y resolver usando algoritmos de búsqueda en grafos.



## 1.8. Algoritmos de búsqueda

Recorren el grafo "visitando" todos sus nodos. Son útiles si nuestro objetivo es encontrar una o todas las soluciones factibles para un problema (**Problemas de satisfacción**). También es posible usarlos si queremos encontrar la mejor solución de un problema (**Problemas de optimización**).

### 1.8.1. Búsqueda en profundidad

1. Parte del nodo inicial
2. Avanza hasta el final de un camino

3. Si no encuentra una solución, se devuelve hasta encontrar un camino alternativo y vuelve a 2.

```
void DFS(State& initial){
    stack<State> S;
    S.push(initial);
    while(!S.empty()){
        State s= S.top();
        if(visited(s)) continue;
        visit(s);
        list<Action> actions = get_actions(s);
        for( Action a : actions){
            State ss=transition(s,a);
            if(!visited(ss)) S.push(ss);
        }
    }
}
```

El hecho de devolverse al camino anterior, se conoce como **Backtracking**. Los nodos visitados se marcan para evitar pasar nuevamente por ellos.

### 1.8.2. Búsqueda en anchura

- Parte del estado inicial
- Visita los nodos adyacentes
- Visita a los nodos adyacentes de estos nodos.
- Y así sucesivamente hasta encontrar una solución (o todas).

```
void BFS(State& initial){
    queue<State> S;
    S.push(initial);
    while(!S.empty()){
        State s= S.top();
        if(visited(s)) continue;
        visit(s);
        list<Action> actions = get_actions(s);
        for( Action a : actions){
            State ss=transition(s,a);
        }
    }
}
```

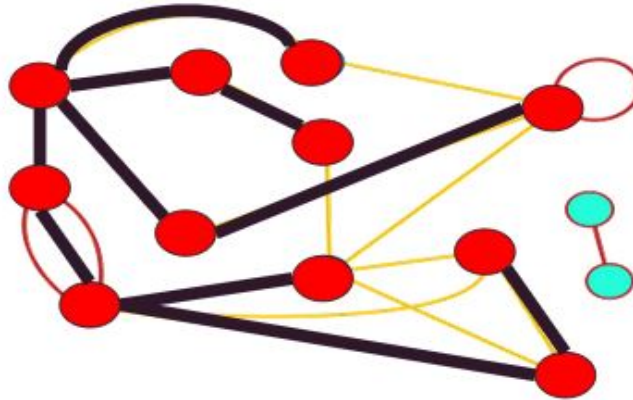
```

        if (!visited(ss)) S.push(ss);
    }
}

```

### 1.8.3. Observaciones

- Al realizar una búsqueda en un grafo, visitamos todos los nodos 'alcanzables' desde un nodo inicial.
- Al evitar pasar por nodos repetidos, la búsqueda en el grafo tiene el aspecto de un árbol por lo que se conoce como árbol de búsqueda.



- Generalmente los árboles de búsqueda para problemas complejos son gigantes. Ejemplo:
  - n-reinas:  $n!$
  - Cubo rubik 3x3:  $10^{19}$
  - Ajedrez:  $10^{120}$
- La búsqueda en profundidad ocupa menos recursos de memoria: en la pila sólo se almacenan los nodos de una rama del árbol, i.e, complejidad espacial  $O(\log(n))$ .
- La búsqueda en anchura ocupa más memoria ( $O(n)$ ). Sin embargo, es más eficiente en tiempo cuando queremos encontrar la solución más cercana al estado inicial.

- En problemas complejos grandes es imposible pasar por todos los nodos del grafo. Generalmente, sólo se visitan los más prometedores.

#### 1.8.4. Best-first

Algoritmo de búsqueda que visita primero los estados **más** prometedores (de acuerdo a una función o heurística). Se necesita una función para evaluar o rankear los estados del grafo.

```
void best_first(State& initial){
    priority_queue<State> S;
    S.push(initial);
    while(!S.empty()){
        State s = S.top();
        if(visited(s)) continue;
        visit(s)
        list<Action> actions = get_actions(s);
        for(Action a: actions){
            State ss = transition(s,a);
            if(!visited(ss)) S.push(ss);
        }
    }
}
```

#### 1.8.5. ¿Cómo diseño una heurística?

En problemas de optimización, puede ser el costo o valor de estado actual.

- Valor en la mochila
- Distancia recorrida
- Cantidad de movimientos (cubo rubik)

En problemas de satisfacción, pueden ser las restricciones no cumplidas aún.

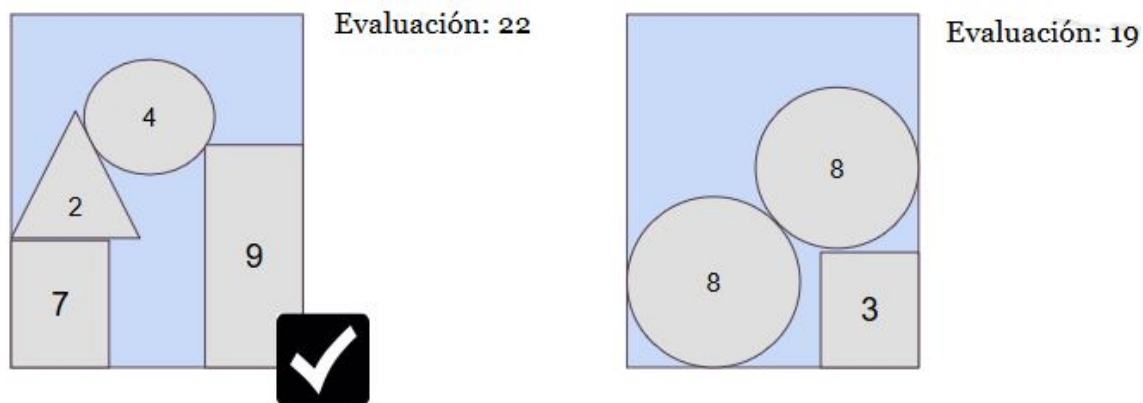


Figura 1.3: Ejemplo en Valor actual en heurística

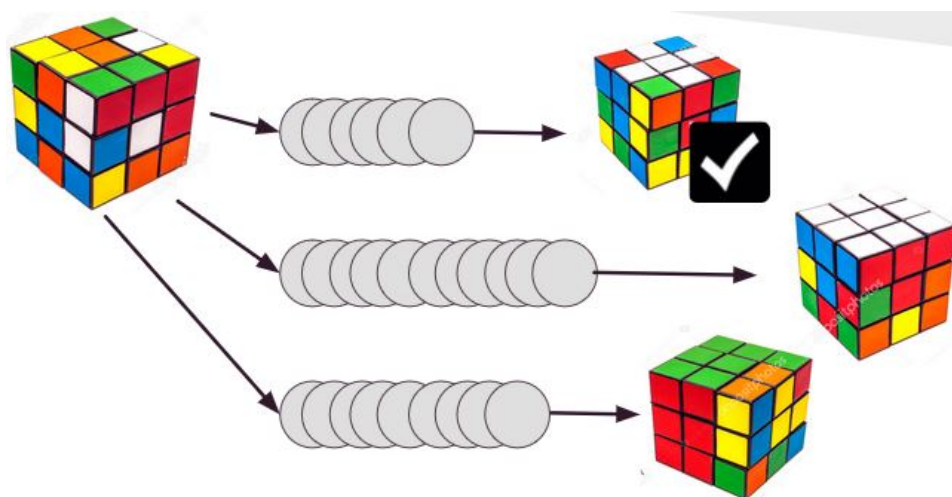


Figura 1.4: Ejemplo Cubo Rubik

X	X		
		X	
			X

Evaluación:  $3 + 1 + 2 + 2 = 8$

	X		
X			
		X	
			X

Evaluación:  $1 + 1 + 1 + 1 = 4$



Figura 1.5: Ejemplo Suma de conflictos

	Birmingham						
177		Bradford					
32	193		Coventry				
67	117	66		Derby			
146	55	149	80		Doncaster		
172	16	186	111	419		Leeds	
129	57	160	93	442	67		Manchester
119	59	126	57	376	54	63	Sheffield

Estados:

1. Birmingham -> Coventry -> Derby:  $32+66$
2. Birmingham -> Leeds -> Manchester:  $172 + 67$
3. Birmingham -> Doncaster -> Leeds:  $146 + 419$



Figura 1.6: Ejemplo de distancia recorrida

#### 1.8.5.1. ¿Cómo mejorar la heurística?

Podemos mejorar una heurística agregando información específica del problema a la función. No existe un método estándar, pues depende del problema que estemos resolviendo.

### 1.8.6. ¿Cómo crear heurísticas para evaluar estados?

Use su intuición

1. Tome dos estados del problema
2. Compárelos y decida, intuitivamente, cuál es más prometedor
  - a) Llegará a un mejor resultado
  - b) Llegará al estado final
3. **¿Por qué cree que es mejor?** Identifique qué tomó en cuenta para decidir.
4. Describa su razonamiento como una regla lógica o función matemática (heurística).
5. Repita el procedimiento y ajuste la heurística. Vea qué ocurre en caso de que se produzca un 'empate' de la heurística.

### 1.8.7. Heurísticas admisibles

En algunos problemas de optimización es posible calcular un mínimo costo (o máximo valor) alcanzable a partir de un estado. Una heurística de esa forma se conoce **heurística admisible**.

#### 1.8.7.1. Ejemplos.

**Mochila - Simulación(relajación del problema)**

- Se colocan los items maximizando valor/peso
- Si el item no cabe entero se coloca una parte y se asume que su valor también se fracciona
- El valor obtenido en esta 'relajación' siempre es mayor o igual al máximo valor posible del problema original.

### 1.8.8. Algoritmo de búsqueda A\*

El algoritmo A\* no es más que el best-first usado junto a una heurística admisible. A\* tiene la siguiente propiedad:

*El primer estado final que se visita es una solución óptima*

## 1.9. Estrategias incompletas de búsqueda

### 1.9.1. Greedy

Algoritmo de búsqueda que avanza por el 'mejor nodo' en cada iteración descartando el resto.

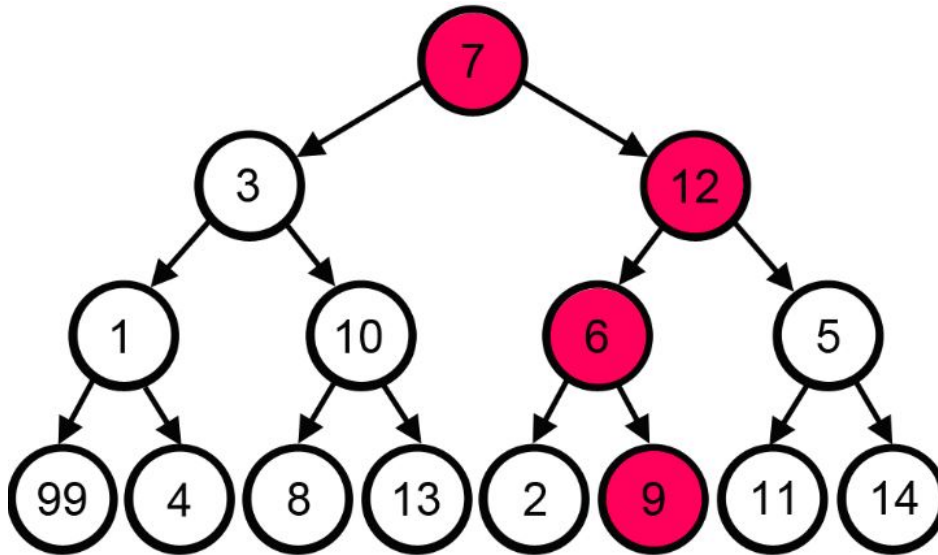


Figura 1.7: Representación greedy

### 1.9.2. GRASP - Greedy Randomized Search Procedure

Greedy aleatorio que en cada iteración selecciona '**uno de los mejores estados**' descartando el resto.

A diferencia de un greedy, se puede ejecutar varias veces obteniendo diferentes resultados. El mejor de ellos es generalmente mejor que el obtenido por un greedy.

### 1.9.3. Beam Search

Búsqueda en anchura que mantiene un conjunto de los  $w$  mejores nodos en cada nivel del árbol, descartando el resto.

Un greedy se considera como un beam search con  $w = 1$ , por otro lado, un BFS o búsqueda en anchura se considera como un beam search con  $w$  infinito.

Cuando se usa un greedy para evaluar los estados, denominamos a esta técnica como **Beam Search Greedy**

#### 1.9.4. Ant Colony Optimization

Aplicar greedies aleatorios (GRASP) en el árbol de búsqueda, a partir de esto, los estados en los mejores caminos se recompensan (se puede recompensar la acción o alguna característica compartida por varios estados).

Bajo la idea del parrafo anterior, se vuelve aplicar el GRASP, pero ahora tomando en cuenta las recompensas, que denominaremos como *feromonas*.

#### 1.9.5. ¿Cómo evaluar los estados?

- Función heurística
- Mejor solución posible (lower bound en problemas de minimización)
- Solución factible a partir del nodo (greedy)
- Analizar algunos niveles a partir del nodo y valorar de acuerdo al mejor estado encontrado (look ahead)
- Las acciones también se pueden evaluar para descartar estados antes de generarlos.

### 1.10. Métodos de búsqueda local

Lo que si podemos hacer es:

- Modelar nuestro problema con un conjunto de estados y acciones para pensar de un estado al otro.
- Asociar a cada estado un valor y función objetivo.

Y queremos encontrar el estado que maximice (o minimice) su valor, entonces podemos usar métodos de búsqueda local.

Para los métodos de búsqueda local todos los estados con 'finales', por lo que el objetivo es encontrar un buen estado final, cabe recordar que durante el inicio del capítulo definimos una acción como un movimiento, y ahora denominaremos como *vecindario* a todos los nodos adyacentes al estado.

Debemos definir la representación de los estados, sus movimientos (vecindario) y una función para evaluar estados.

Por ejemplo:

### 1. N-reinas

- Estados: Tablero con n reinas colocadas (una por columna)
- Función Objetivo: Cantidad de conflictos
- Movimiento: Mover una reina hacia una casilla hacia arriba o hacia abajo.

### 2. Problema de la Mochila

- Estados: Mochila con items
- Función Objetivo: Valor total de los items en la mochila
- Movimiento: Agregar o cambiar un item de la mochila.

### 3. Traveling Salesman Problem (TSP)

- Estados: Secuencia de ciudades
- Función Objetivo: Distancia de la ruta (siguiendo secuencia)
- Movimiento: Intercambiar la posición de 2 ciudades en la ruta.

#### 1.10.1. Hill Climbing

Es un algoritmo similar al greedy, en donde partiendo de un estado inicial, en cada iteración se escoge un estado vecino que mejore la evaluación del estado actual.

- Mejor vecino
- Primer vecino que mejore.

Puede caer en óptimos locales. Por otro lado, también existen variantes que permiten escapar de óptimos locales como:

- Simulated Annealing
- Tabu search

### 1.11. Algoritmos evolutivos

Los algoritmos evolutivos trabajan con varios estados a la vez, realizando modificaciones aleatorias a las que denomina como *mutación*. Estos algoritmos aplican técnicas de búsqueda local, de forma que combinan los estados con el objetivo de mejorarlos. Este último proceso mencionado se le conoce como *cruzamiento*.

Sin embargo, existe otro proceso que selecciona los mejores estados y realizan copias de el, bajo esto, el proceso es denominado *elitismo*.

Ejemplos de algoritmos evolutivos son:

- Algoritmos genéticos
- Particle Swarm Optimization

## Capítulo 2

# Búsqueda adversaria

## 2.1. Game Tree Search

### 2.1.1. Introducción

Un **juego** es un conjunto de reglas que permite interactuar a uno o varios jugadores produciendo recompensa/castigo para cada uno de ellos. Este se puede modelar usando grafos, con sus respectivos estados y acciones.

Cada **estado** (final) está asociado a un vector de puntaje para los jugadores. Por otro lado, la estrategia de un jugador (política) es la función que aplica para seleccionar una acción dado un estado.

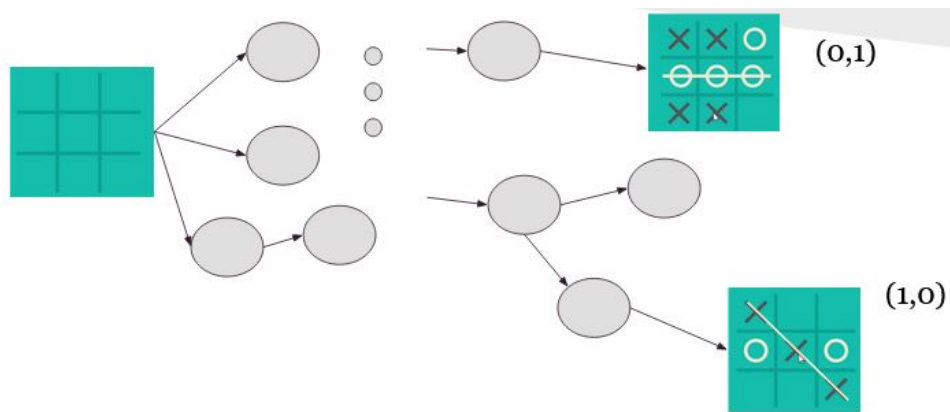


Figura 2.1: Política de estados

Visto del punto de vista de cada jugador el objetivo es escoger la acción que maximice su recompensa esperada, sin embargo, los adversarios intro-

ducen incertidumbre en el proceso de búsqueda.

En ambientes no deterministas, un adversario puede ser referido simplemente al impredecible comportamiento de la naturaleza.

La teoría de juegos presenta multiples problemas, empezando porque un juego consiste de 2 jugadores, por lo tanto, se hace referencia el resultado final se aplica el concepto de **suma cero**, lo que implica que mientras un jugador gana, otro debe perder. Otro problema es la **información perfecta**, debido a que ambos jugadores conocen el 100 % de la información referente al juego, pero a fin de cuentas, todo conlleva a realizar la mejor jugada para un estado dado.

Algunos ejemplos que se aplican al parrafo anterior son:

- Gato
- Ajedrez
- Damas

Ahora, es importante detallar cada elemento del grafo que representará el juego. En efecto:

- Raíz: Estado actual de juego
- Arcos: Acciones/movidas de los jugadores
- Nodos: Estados del juego.
  - MAX Nodes: Le toca jugar al **jugador**
  - MIN Nodes: Le toca jugar al **Oponente**

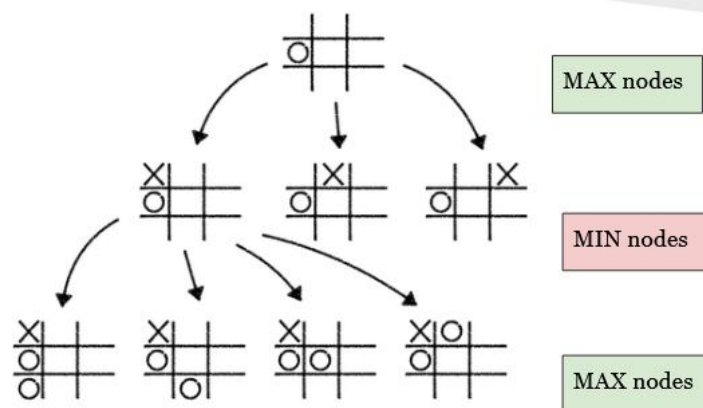


Figura 2.2: Game Tree Search

## 2.1.2. Minimax

### 2.1.2.1. Recursivo

Retorna la acción que genera el estado con un máximo valor. Aplica el siguiente criterio para calcular el valor de un estado:

- Si le toca al jugador (MAX), el valor será igual al **máximo valor** de los estados hijos.
- Si le toca al oponente (MIN), el valor será igual al **mínimo valor** de los estados hijos.

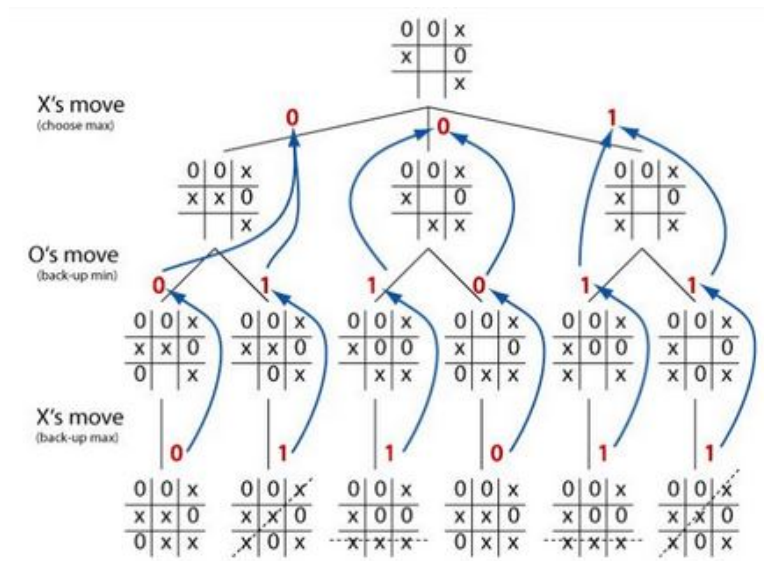
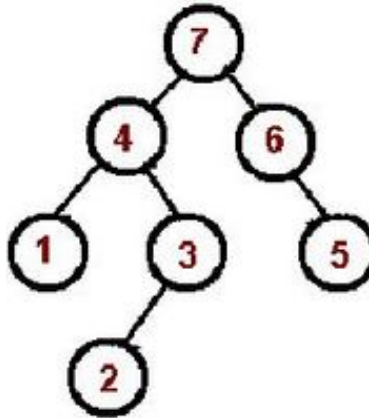


Figura 2.3: Minimax Recursivo

### 2.1.2.2. Iterativo

Es una **búsqueda en profundidad** con límite de niveles de estado inicial (post-order). Por lo tanto, hay que calcular el valor de cada nodo con el valor de sus hijos (max o min).



```

Action minimax(state):
    Stack S;
    S.push(state)
    while S is not empty:
        s = S.top()
        if s.first_visit:
            s.first_visit=False
            actions = s.get_actions()
            for a in actions:
                s_child=s.transition(a)
                s.add_child(s_child)
                S.append(s_child)
        else: #second visit
            s.value = s.eval()
            S.pop()
    return max_eval_action(state)
  
```

- Si es nodo **terminal** la evaluación corresponde a la recompensa obtenida (o se usa función de evaluación).
- Si **no es terminal**, la evaluación corresponderá al máximo (o mínimo) valor de los hijos.
- Retorna la acción que resulta en el estado con mayor value

### 2.1.2.3. Observaciones

- Generalmente sólo se analiza un número limitado de niveles del árbol (puede ir incrementando hasta que se acabe el tiempo).
- Estados terminales son evaluados y su 'valor' es propagado hacia arriba.
- La acción que maximiza el valor 'esperado' es la seleccionada.
- Las funciones de evaluación para **estados terminales** son esenciales.

### 2.1.3. AlphaBeta - Pruning

Es una mejora de minimax el cual evita buscar en regiones innecesarias, donde cada nodo almacena adicionalmente:

- $\alpha$ : Mejor valor actual para el jugador
- $\beta$ : Mejor valor para el oponente

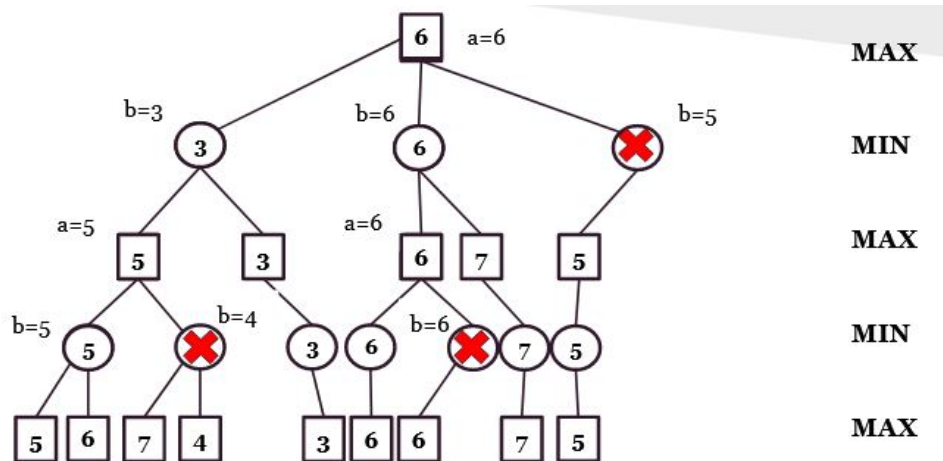


Figura 2.4: Alpha Beta Pruning

#### 2.1.3.1. Observaciones

- El desempeño mejora si se visitan primero los hijos más prometedores.
- En el caso ideal,  $\alpha\beta$ -pruning puede revisar el doble de movimientos que un minimax en el mismo tiempo.

- En el ajedrez, los movimientos se pueden ordenar así:
  - Captura de pieza
  - Amenazas
  - Movimientos hacia adelante
  - Movimientos hacia atrás

#### 2.1.3.2. ¿Cómo implemento el algoritmo?

- Cada nodo almacena valores  $a=-\text{inf}$  y  $b=+\text{inf}$
- Búsqueda en profundidad post-order con límite de altura.
- Cada nodo que sale de la pila copia  $a$  y  $b$  de su padre.
- Si  $b \leq a$ , se 'salta' el nodo y continua con el siguiente
- Al calcular valor de nodo se actualiza  $a$  o  $b$  de su padre ( $a$  si es padre es MAX o  $b$  si es MIN).

#### 2.1.4. Mejoras/variantes

- **Quiescence search:** Árbol se extiende hasta llegar a estado estable (quiescente).
- **Singular extensión:** Jugadas especialmente buenas se extienden un nivel más.
- **Conspiracy search:** Mezcla las dos ideas.
- **Monte Carlo Tree Search**

#### 2.1.5. Extensión de minimax

- **Expectimax:**  
Generalización de minimax para juegos estocásticos (e.g, backgammon). Valor de *chance nodes* se calculan ponderando las probabilidades de cada nodo hijo.
- **Miximax**  
Generalización del anterior para juegos estocásticos con información imperfecta (e.g, poker). Usa estrategias rivales predefinidas y las trata de manera estocástica.

## 2.2. Montecarlo Tree Search

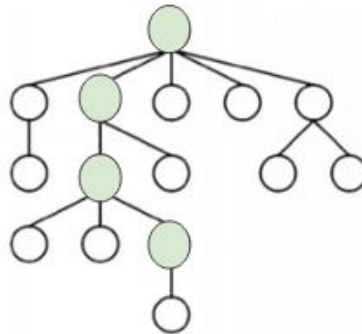
### 2.2.1. ¿Qué es Montecarlo Tree Search?

- Es un algoritmo para explorar grafos que utilizan simulaciones para obtener información de los estados.
- Es aplicado en:
  - Juegos deterministas con informacion perfecta (go, ajedrez)
  - Juegos con aleatoriedad e información imperfecta (backgammon, scrabble).
  - Problemas de tomas de decisiones.

### 2.2.2. ¿Cómo funciona?

- Árbol de búsqueda es construido de manera iterativa.
- En cada iteración comienza de la raíz y usa una *política* para descender y así hasta llegar al nodo **más urgente**.
- La política ofrece un balance entre exploración (áreas no explorada) y explotación (áreas prometedoras).

### 2.2.3. Resumen



- El nodo más urgente es simulado.
- Se agrega un nuevo hijo correspondiente a la acción realizada.
- El resultado es propagado a los ancestrales (estadística).

#### 2.2.4. Simulación

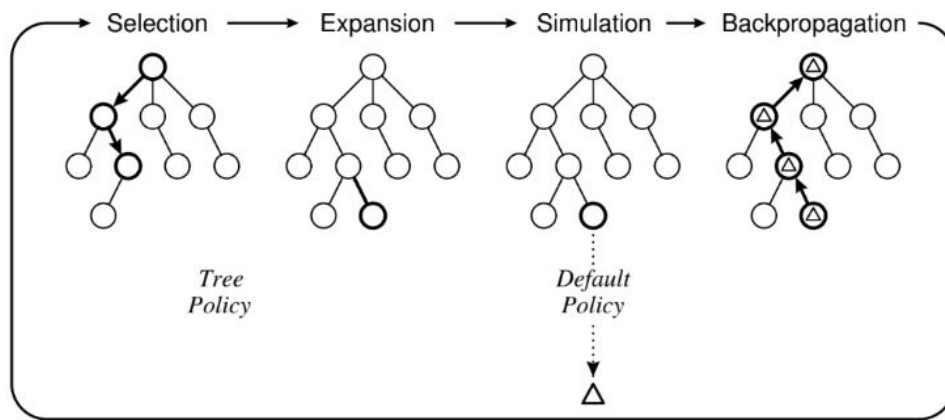
Aplica política por defecto.

- Movimientos aleatorios (light playouts)
- Movimientos influenciados por regla heurísticas (heavy playouts)

#### 2.2.5. Ventajas y desventajas

- No requiere función heurística para evaluar estados (como minimax o alfa-beta).
- El algoritmo puede detenerse en cualquier momento retornando el movimiento más prometedor encontrado.
- La desventaja es que estas simulaciones podrían pasar por alto alguna jugada rival que lo lleva a perder.

#### 2.2.6. Pasos de Montecarlo Tree Search



### 2.2.7. Algoritmo MonteCarlo Tree Search

---

#### Algorithm 1 General MCTS approach.

---

```

function MCTSSEARCH( $s_0$ )
  create root node  $v_0$  with state  $s_0$ 
  while within computational budget do
     $v_l \leftarrow \text{TREEPOLICY}(v_0)$ 
     $\Delta \leftarrow \text{DEFAULTPOLICY}(s(v_l))$ 
     $\text{BACKUP}(v_l, \Delta)$ 
  return  $a(\text{BESTCHILD}(v_0))$ 

```

---

### 2.2.8. Política de selección (UCT)

- Upper Bound Confidence for Trees
- Basada en algoritmo UCV (problema del bandido multibrazos).
- UCT estima el máximo valor que podría obtenerse al simular el nodo (optimista)
- Se selecciona el nojo  $j$  que maximiza UCT

$$UCT = \bar{X}_j + 2C_p \sqrt{\frac{2 \ln n}{n_j}}$$

$\bar{X}_j$  = promedio de simulaciones  
 $n$  = simulaciones padre de  $j$   
 $n_j$  = simulaciones en nodo  $j$   
 $C_p$  = nivel de "optimismo"

- Primer término prioriza buenos resultados en simulaciones (explotación).
- Segundo término prioriza nodos con pocas simulaciones (exploración).
  - Hijos no visitados son priorizados ( $n_j = 0$  implica  $UCT = +\infty$ )
- $c_p$  se puede ajustar para controlar la exploración o la explotación.
- $c_p = \frac{1}{\sqrt{2}}$  es un buen valor si  $X$  se mueve entre 0 y 1.

### 2.2.9. ¿Qué acción retornar (BestChild)?

- Nodo con mejor promedio de resultados.
- Nodo más visitado.
- Nodo que maximiza UCT
- Seguir iterando hasta que nodo más visitado = nodo con mejor promedio.

### 2.2.10. Pseudocódigo

```
function TREEPOLICY( $v$ )  
  while  $v$  is nonterminal do  
    if  $v$  not fully expanded then  
      return EXPAND( $v$ )  
    else  
       $v \leftarrow$  BESTCHILD( $v, Cp$ )  
  return  $v$ 
```

```
function BESTCHILD( $v, c$ )  
  return  $\arg \max_{v' \in \text{children of } v} \frac{Q(v')}{N(v')} + c \sqrt{\frac{2 \ln N(v)}{N(v')}}$ 
```

```

function EXPAND( $v$ )
    choose  $a \in$  untried actions from  $A(s(v))$ 
    add a new child  $v'$  to  $v$ 
        with  $s(v') = f(s(v), a)$ 
        and  $a(v') = a$ 
    return  $v'$ 

```

```

function DEFAULTPOLICY( $s$ )
    while  $s$  is non-terminal do
        choose  $a \in A(s)$  uniformly at random
         $s \leftarrow f(s, a)$ 
    return reward for state  $s$ 

```

```

function BACKUP( $v, \Delta$ )
    while  $v$  is not null do
         $N(v) \leftarrow N(v) + 1$ 
         $Q(v) \leftarrow Q(v) + \Delta(v, p)$ 
         $v \leftarrow$  parent of  $v$ 

```

#### 2.2.11. Mejores/variantes (selección)

- **First Play urgency:** fija un  $score \rightarrow +\infty$  para nodos no visitados (incrementa explotación).
- Agrupar acciones similares generando decisiones adicionales (qué grupo seleccionar, luego que acción tomar).

- Uso de **tablas de transposición** para mantener información de estados que se repiten (hashing).
- **Progressive bias:** Usar información heurística + política UCT.
- **Opening book:** Usar diccionario (mapa) con jugadas de libro (mientras sea posible).
- Simulación basada en reglas (evaluación de acciones usando función).
- Ponderar recompensas en simulación.
  - Premiar simulaciones cortas o tardías
  - Diferenciar entre partidas ganadas por mucho y por poquito.

## 2.3. Aprendizaje reforzado

### 2.3.1. Definiciones

#### 2.3.1.1. ¿Qué es el aprendizaje automatizado?

Es el estudio científico de algoritmos y modelo estadístico donde sistemas computacionales usan para realizar tareas específicas sin utilizar instrucciones específicas.

#### 2.3.1.2. ¿Qué es el aprendizaje por refuerzo?

Es el área de aprendizaje automatizado que se preocupa de qué acción debe tomar un agente en un entorno, para maximizar la noción de recompensa acumulativa.

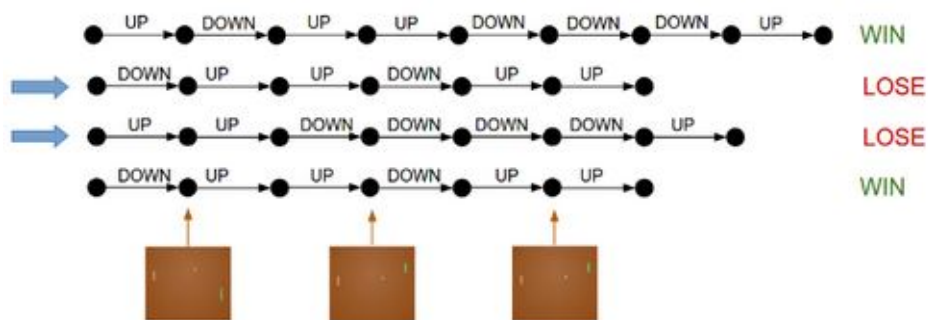
### 2.3.2. ¿Cómo funciona?

Basándose en el proceso de decisiones de Markov.

- Funciona como el condicionamiento (ej. adiestrar animales).
- Cada acción significará una reacción, la cuál obtendrá un feedback en forma de recompensa.
- Priorizar acciones buenas

### 2.3.3. Política de gradiente ascendente

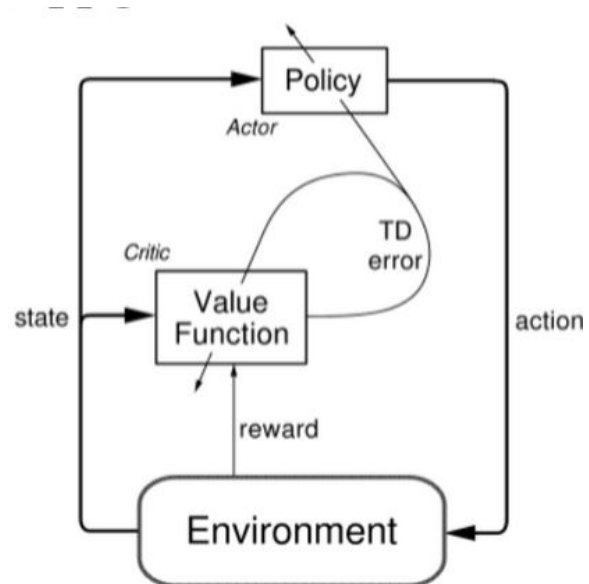
Correr la política  $n$  veces. Ver las acciones que dieron buenos resultados, incrementar su  $p(x)$ .



### 2.3.4. Maneras de aprender

#### On Policy

- Aprende en base a una política, la cuál estimará la mejor acción para un estado  $s$ .
- La política se actualizará dependiendo de la recompensa que obtenga al realizar acciones.
- PPO



### Off Policy

- La estimación será realizada a partir de una política (ej. política greedy), sin embargo, la política no será la utilizada para realizar la acción.
- Se actualizará otra política denominada política de comportamiento.
- Deep Q-Network

Initialized							
Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	327	0	0	0	0	0	0
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
States	499	0	0	0	0	0	0
	.	.	.	.	.	.	.

Training

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	328	-2.30108105	-1.97092096	-2.30357004	-2.20591839	-10.3607344	-8.5583017
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
States	499	9.96984239	4.02706992	12.96022777	29	3.32877873	3.38230603
	.	.	.	.	.	.	.

## Capítulo 3

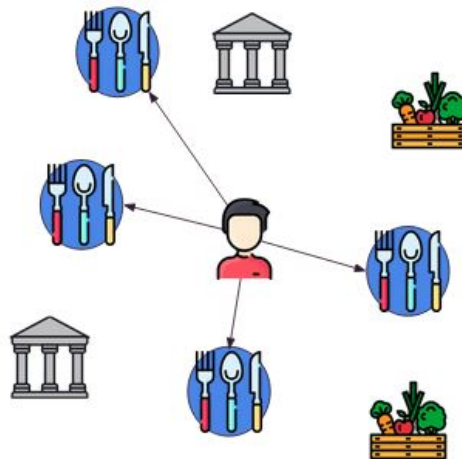
# Indexación de datos espaciales

### 3.1. Modelando problemas usando estructuras espaciales

#### 3.1.1. Problema de turismo

Antes se trabajó el cómo encontrar el camino más corto de un punto A a un punto B más cercano, pero ¿Qué sucede si el punto B no es el que quiere el usuario?, ¿y si el usuario quiere encontrar otros puntos similares al punto B y que estén cerca?

Supongamos que está de viaje y desea encontrar los restauranets más cercanos a su ubicación.



### ¿Cómo lo resolvemos?

Podríamos:

- Buscar todos los restaurantes que tengamos a una distancia máxima.
- Ordenarlos por distancia
- Mostrarlos ordenados al usuario.

### Aplicaciones del problema.

- **Buscar un concepto desconocido.**

Necesito saber que significa una palabra o frase, pero no sale directo en el diccionario, ¿busco la primera palabra que encuentre que tenga sentido en el contexto?, ¿O buscar todas las que tengan un parecido en la situación?

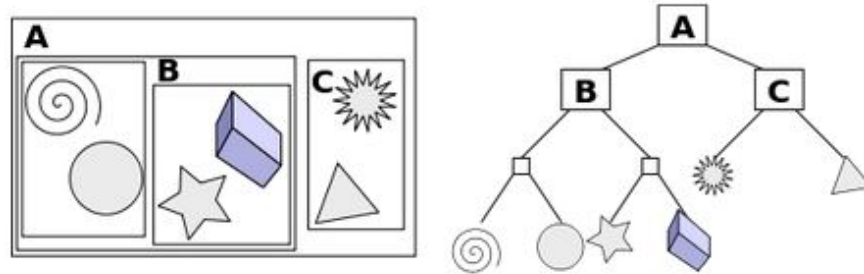
- Acabo de ver una película que me gustó mucho, ¿Existirán otras películas similares?.

### ¿En qué se parecen estos problemas?

Todos estos problemas se pueden representar usando:

- Un conjunto de puntos o vectores
- Una distancia entre los distintos puntos.

## 3.2. Estructuras espaciales

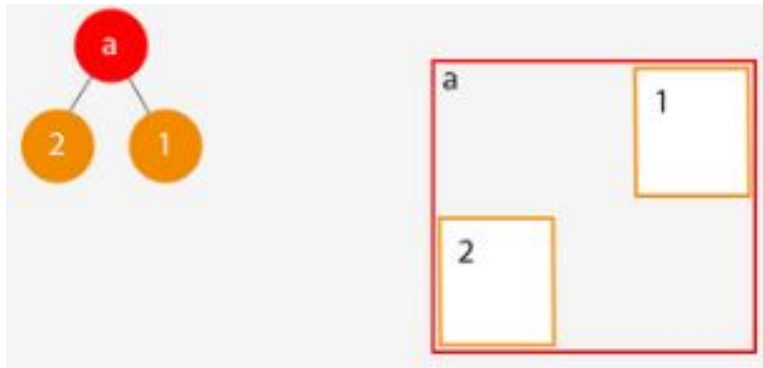


### 3.2.1. Tipos de estructuras espaciales

Existen varias estructuras que se enfocan en solucionar estos problemas, como por ejemplo:

- AABB Tree (Axis-Aligned Bounding Box Tree)
- Segmentation Tree
- KD-Tree
- QuadTree
- OctTree
- Local-sensitive Hashing

### 3.2.2. AABB Tree (Axis-Aligned Bounding Box Tree)



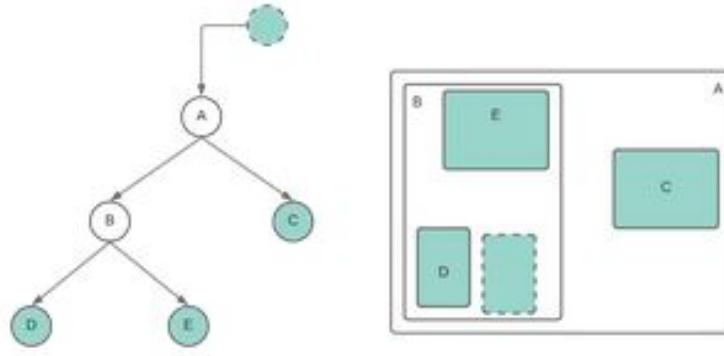
- Un **conjunto de puntos** representado como cajas.
- Cada caja se ubica al interior de una caja superior
- Los datos se ubican en las hojas

#### 3.2.2.1. Estructura básica

```
public Class Node{
    private Node left;
    private Node right;
    /*private Node Parent;*/
    private int box[][];
    private Object data;
    private int dim;
}

public Class AABBTre{
    private Node root;
}
```

### 3.2.2.2. Inserción



```

void InsertLeaf(Tree tree , int ObjectIndex , AABB box)
{
    int leafIndex = AllocateLeafNode (tree ,objectIndex ,box);
    if(tree.nodeCount == 0){
        tree.rootIndex = leafIndex;
        return;
    }
    //Stage 1: find the best sibling for the new leaf
    int bestSibling = 0;
    for(int i = 0; i < m_nodeCount; i++){
        bestSibling = PickBest(bestSibling ,i);
    }
    //Stage 2: Create a new parent
    int oldParent = tree.nodes[sibling].parentIndex;
    int newParent = AllocateInternalNode(tree);
    tree.nodes[newParent].parentIndex = oldParent;
    tree.nodes[newParent].box = Union(box, tree.nodes[sibling].box);
    if (oldParent != nullIndex){
        //The sibling was not the root
        if (tree.nodes[oldParent].child1 == sibling){
            tree.nodes[oldParent].child1 = newParent;
        } else{
            tree.nodes[oldParent].child2 = newParent;
        }
    }
}

```

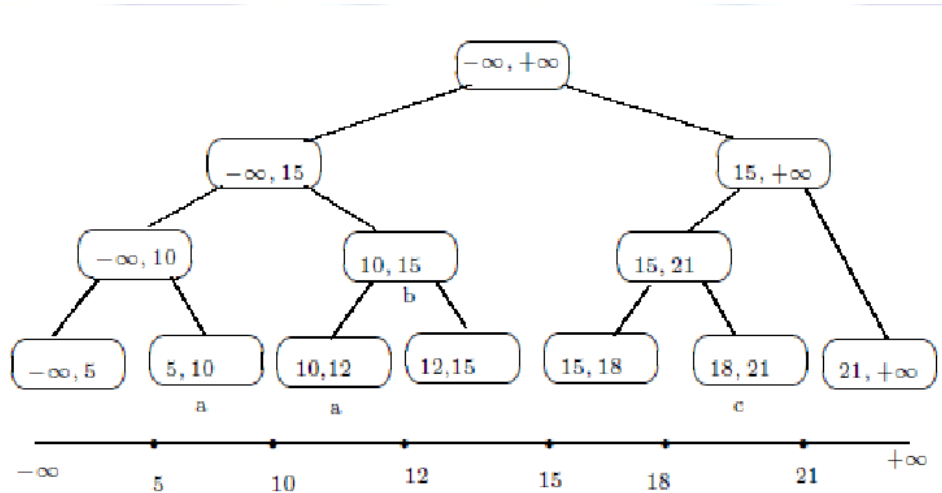
```

        tree.nodes[newParent].child1 = sibling;
        tree.nodes[newParent].child2 = leafIndex;
        tree.nodes[sibling].parentIndex = newParent;
        tree.nodes[leafIndex].parentIndex = newParent;
    } else {
        //The sibling was the root
        tree.nodes[newParent].child1 = sibling;
        tree.nodes[newParent].child2 = leafIndex;
        tree.nodes[sibling].parentIndex = newParent;
        tree.nodes[leafIndex].parentIndex = newParent;
    }
    //Stage 3: Walk back up the tree refitting AAB's
    int index = tree.nodes[leafIndex].parentIndex;
    while(index != nullIndex){
        int child1 = tree.nodes[index].child1;
        int child2 = tree.nodes[index].child2;
        int boxChild1 = tree.nodes[child1].box;
        int boxChild2 = tree.nodes[child2].box;

        tree.nodes[index].box = Union(boxChild1, boxChild2);
        index = tree.nodes[index].parentIndex;
    }
}

```

### 3.2.3. Segmentation Tree



- Un **Conjunto de vectores** almacenados en un segmento.
- Cada segmento solo indica los bordes.
- Cada nivel del árbol indica los bordes de segmentos mayores.

#### 3.2.3.1. Estructura básica

```
public Class Node{
    private Node left;
    private Node right;
    /*private Node Parent; */
    private int vector[];
    private Object data;
}
```

```
public Class SegmentTree{
    private Node root;
}
```

#### 3.2.3.2. Insertion

```
bool searchRec(Node* root, int point[], unsigned depth)
{
```

```

//Base cases
if(root == NULL)
    return false;
if(arePointsSame(root->point , point))
    return true;
//Current dimension is computed using current depth and total
//dimension(k)
unsigned cd = depth%k;

//Compare point with root with respect to cd (Current dimension)
if( point[cd] < root->point[cd])
    return searchRec(root->left , point , depth+1);

return searchRec(root->right , point , depth+1);
}

```

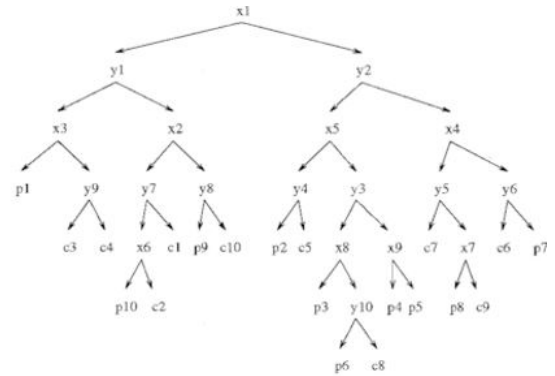
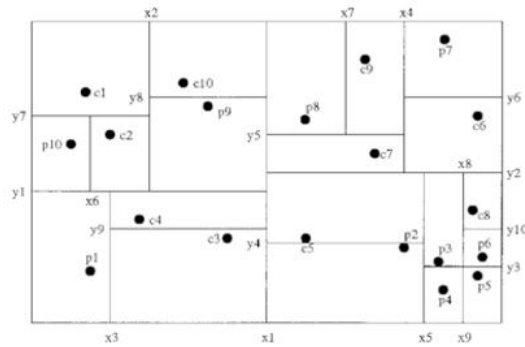
### 3.2.3.3. Búsqueda

```

public Object Buscar(int [] data){
    Node actual = root;
    while(actual != null){
        if(compare(actual,data) == 0)
            return actual.getData();
        else if( compare(actual,data) == -1 && actual.getLeft() != null )
            actual = actual.getLeft();
        else if( compare(actual,data) == -1 && actual.getRight() != null )
            actual = actual.getRight();
        else
            return null;
    }
}

```

### 3.2.4. KD - Tree



- Un conjunto de vectores o puntos .
- El más similar a un arbol binario
- Cada nivel separa por una dimensión distinta.

#### 3.2.4.1. Estructura básica

```

public class Node{
    private Node left;
    private Node right;
    /*private Node Parent;*/
    private int vector[];
    private Object data;
    private int dim;
}

public Class KDTree{
    private Node root;
}

public object Buscar(int[] data){
    Node actual = root;
    while(actual != null){
        if(compare(actual,data) == 0)
            return actual.getData();
        else if( compare(actual,data) == -1 && actual.getLeft() != null )

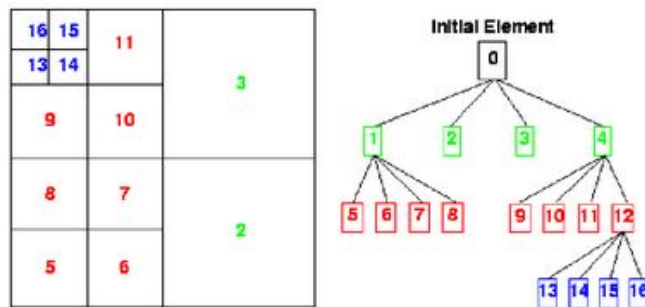
```

```

        actual = actual.getLeft();
    else if( compare(actual,data) == -1 && actual.getRight() != null )
        actual = actual.getRight();
    else
        return null;
    }
}

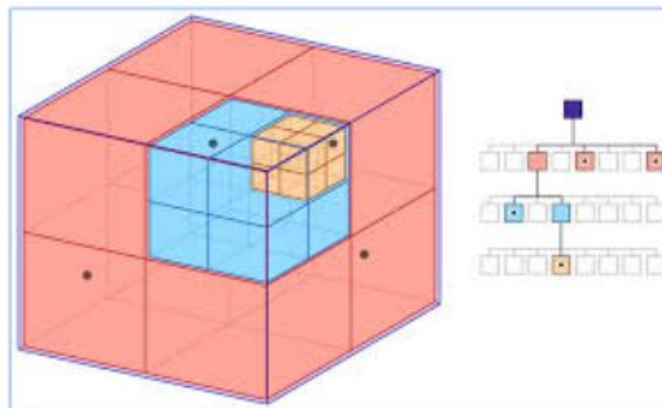
```

### 3.2.5. QuadTree



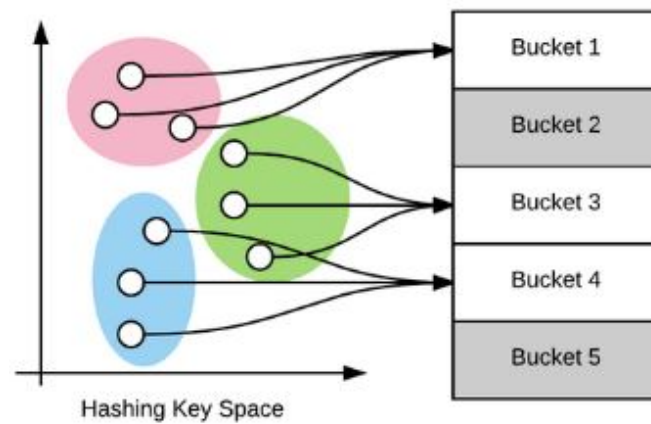
- Un conjunto de puntos o cajas
- Cada nivel indica que tan particionado está el área 2D
- Cada nodo tiene 0 o 4 hijos, con al meno 1 no vacío.

### 3.2.6. OctTree



- Un conjunto de puntos o cajas
- A diferencia del QuadTree este trabaja con 3 dimensiones al mismo tiempo
- Todo el funcionamiento restante es igual.

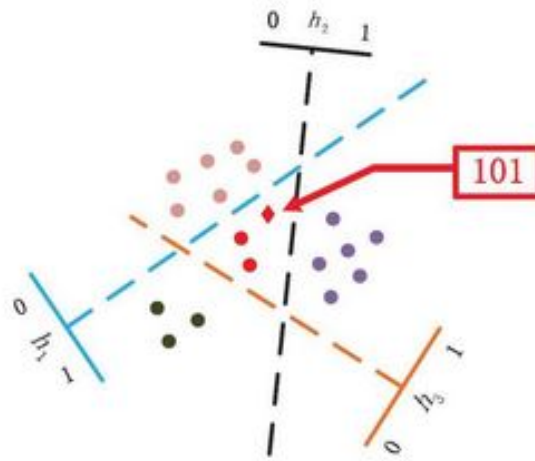
### 3.2.7. Local-sensitive Hashing



- Un **conjunto de puntos**
- Cada punto se guarda en un balde o bucket.
- Cada bucket puede poseer más de 1 punto.

#### 3.2.7.1. Función Hash

La función hash se puede realizar a fuerza brutas forzando que cada dato entre en al menos 1 bucket o en base de proyecciones.



### 3.2.7.2. Estructura básica

```

public class Bucket{
    private int center [];
    private ArrayList data;
    private int canData;
}

public class LSH{
    private Bucket Buckets [];
    private int distMax;
    private int cantBuckets;
}

```

### 3.2.7.3. Hash

```

public int hash(int [] data){
    for(int i = 0; i < cantBuckets; i++){
        if(dist(Buckets[i],data) < distMax){
            return i;
        }
    }
    for(int i = 0; i < cantBuckets ; i++){
        if(Bucket[i].getCantData() == 0)
            return i;
    }
}

```

}  
}

### 3.3. Algoritmos espaciales

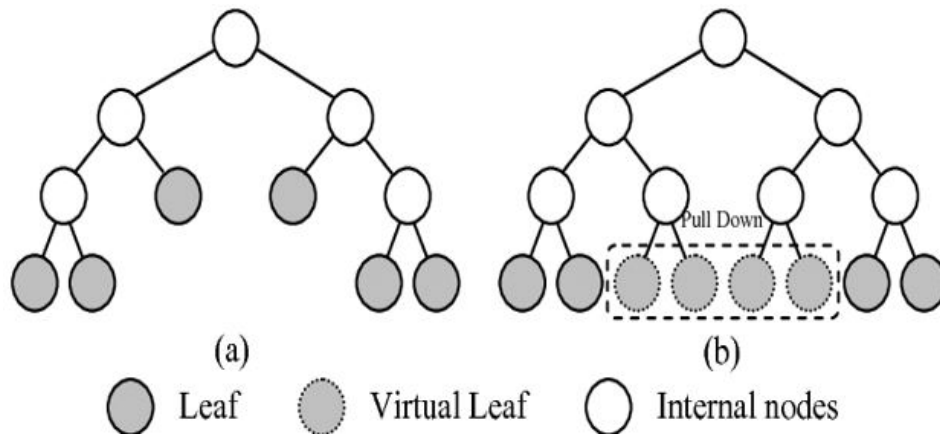
En la subsección anterior, se presentaron las estructuras que pueden manejar datos espaciales, como AABB, Segment Tree y KD-Tree, pero **no es suficiente sólo manejar los datos, sino que hace falta poder realizar acciones con ellos.**

La estructura pueden realizar diversas acciones con los datos, pero en el caso de algoritmos espaciales, los más importantes son la búsqueda de los vecinos más cercanos y la detección de colisiones.

#### 3.3.1. Vecino(s) más cercano(s)

La cantidad siempre se elige en ejecución, pero la lógica siempre es la misma, buscar los n vecinos más cercanos a un dato específico. El paso inicial siempre puede ser el mismo, intentar colar el dato en la posición que le corresponde en la estructura, para tener un punto inicial más cercano con el que partir.

##### ■ AABB: KNN



- Como los datos se encuentran en las hojas, **solo comparamos las hojas.**
- Luego de colocar la caja en el árbol, se compara por cercanía en el árbol, hasta que la distancia del punto al nodo interno sea más grande.

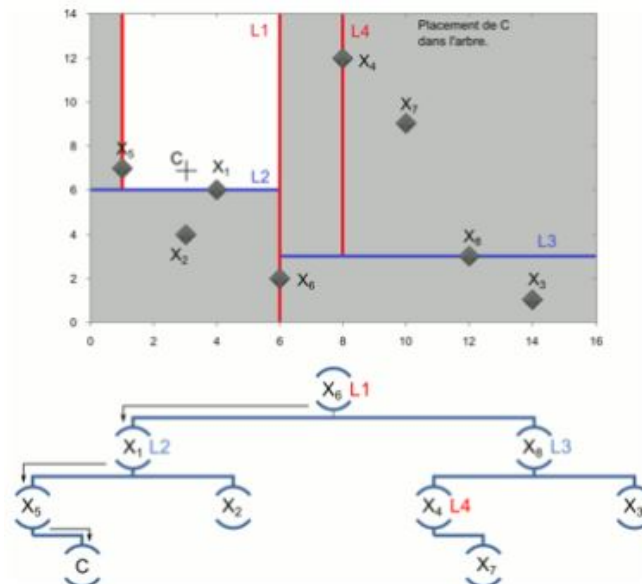
```

VecinosMasCercanos(int n, box b){
    vecinos <- conjuntoOrdenado
    nodos_visitados <- Conjunto
    realizar "insercion" de la caja b
    S <- pila
    insertar a S el/los nodos hoja mas cercanos
    Mientras S no este vacio:
        pop(S)
        analizar si el dato es mas cercano
        Si posee padre y no se recorrio:
            push(S,padre)
        Si posee rama izquierda, no se recorrio y
        esta posee algun dato potencialmente cercano:
            push(S,rama izquierda)

        Si posee rama derecha, no se recorrio y
        esta posee algun dato potencialmente cercano:
            push(S,rama_derecha)
    }

```

#### ■ KD Tree: KNN



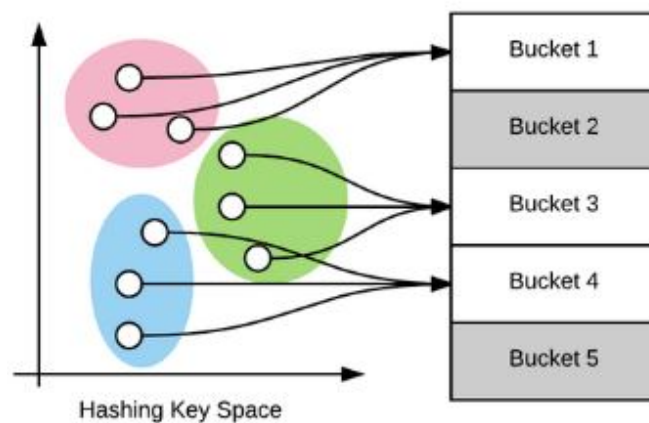
- Lógica similar al árbol AABB, solo que todos los nodos poseen datos.
- Esto hace que la búsqueda sea más precisa pero en más tiempo.

```

VecinosMasCercanos(int n, point p){
    vecinos<-conjunto ordenado
    Nodos_visitados <-conjunto
    realizar "insercion" del punto p
    S <- pila
    insertar a S los nodos visitados
    Mientras S no este vacio:
        pop(S)
        analizar si el dato es mas cercano
        si posee padre y no se recorrio:
            push(S,padre)
        si posee rama izquierda, no se recorrio y esta posee
        algun dato potencialmente cercano:
            push(S, rama izquierda)
        si posee rama derecha, no se recorrio y esta posee
        algun dato potencialmente cercano:
            push(S,rama derecha)
    }

```

#### ■ Local-Sensitive Hashing: KNN



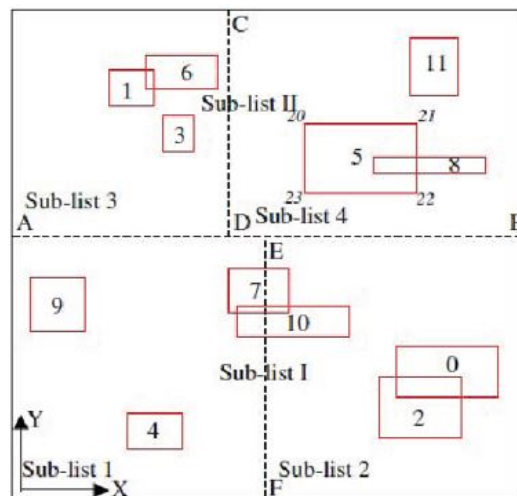
- Para obtener los vecinos más cercanos, solo basta con 'insertar' el punto y obtener el bucket.
- Esto puede fallar si el bucket se encuentra vacío.

### 3.3.2. Detección de colisiones

Ese algoritmo se utiliza cuando se poseen datos físicos de objetos en movimiento, o en el análisis del diseño arquitectónico de una ubicación.

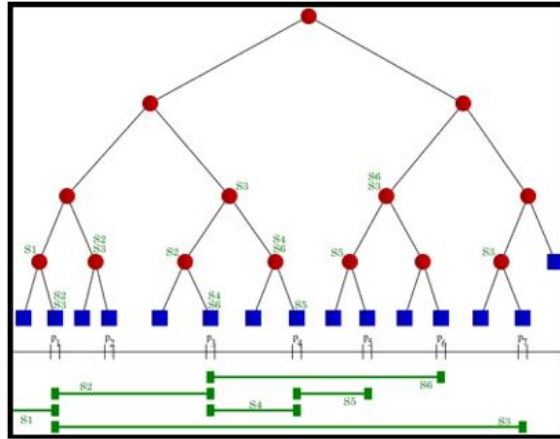
Para efectos de este documento, sólo se estudiará la detección de colisiones de datos estáticos.

#### ■ AABB: CD



- La detección de colisiones en AABB es revisar cada dato y analizar si este está superpuesto a otro dato.
- Normalmente, los objetos que colisionan se encuentran en nodos hojas contiguas.

#### ■ Segment Tree: CD



Por la composición de Segment tree, este puede detectar la colisión al ingresar el dato, pero puede requerirse un análisis posterior para futuras operaciones.

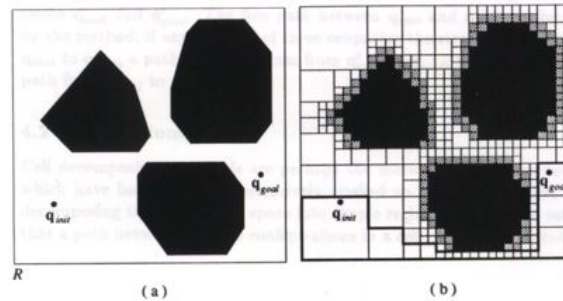
### 3.3.3. ¿QuadTree y OctTree?

Estas estructuras, a pesar de ser estructuras espaciales, no se enfocan en la obtención de los vecinos más cercanos o en la detección de colisiones.

Estas estructuras se enfocan en la aplicación de cambios vectoriales o en el valor de sus nodos.

### 3.3.4. Filtros de imágenes

Al aplicar un cambio en una imagen, esta estructura automáticamente genera un cambio en los nodos cercanos, proporcional en menor escala con el nodo principal cambiado.



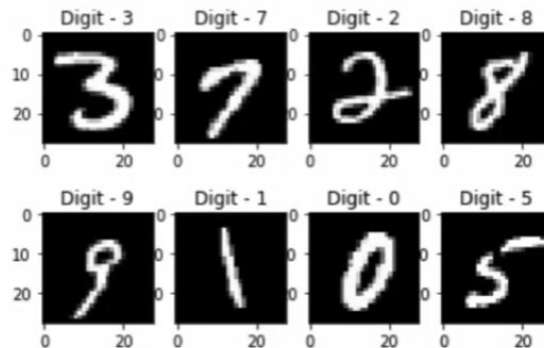
## Capítulo 4

# Perceptron multicapa

### 4.1. Introducción a Redes neuronales

#### 4.1.1. Problema 1

Supongamos que queremos traducir dígitos escritos a mano a su representación digital



Podríamos

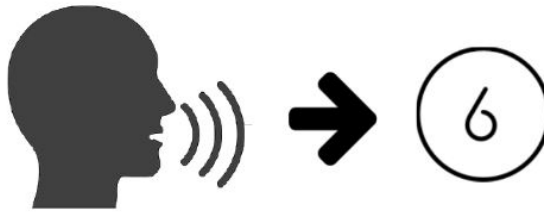
- Podríamos para el 0, buscar un círculo
- Podríamos para el 9, buscar un círculo con un 'palito' vertical.
- Para un 1 buscar un 'palito' vertical
- Para el 8 dos círculos
- Y así con el resto

#### 4.1.1.1. Aplicaciones del problema

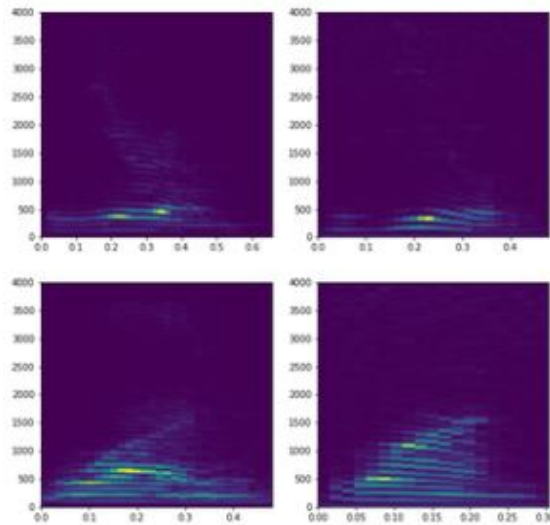
- **Digitalizar texto:** Extender el algoritmo para reconocer otros caracteres.
- Detección y traducción de texto.

#### 4.1.2. Problema 2

A partir de un audio, reconocer qué letra/número se está diciendo



¿Cómo lo resolvemos?



- Buscar en el espectrograma patrones para cada fonema.
- Ejemplo:

'b' tiene 2 fonemas: /b/ y /e/

- Luego, buscar la secuencia de fonemas que se ajuste al audio.

#### 4.1.2.1. Aplicaciones del problema

- Utilización en asistentes inteligentes.
- Speech-to-text

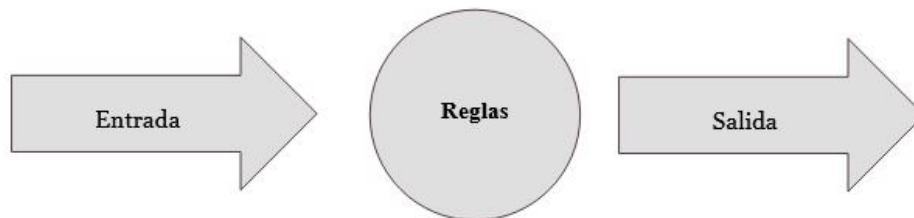
#### 4.1.3. ¿En qué se parecen estos problemas?

Todos estos problemas se resuelven reconociendo patrones:

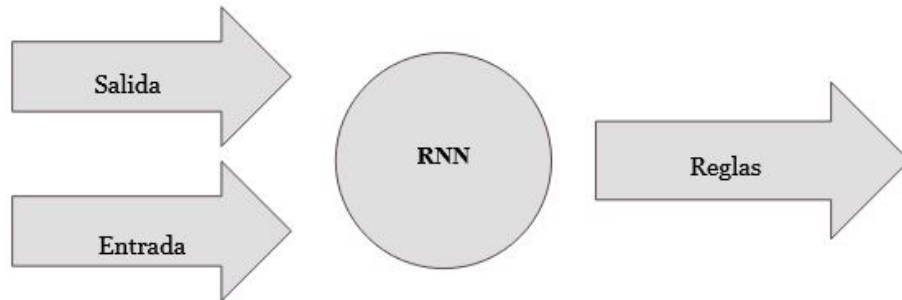
- Se pueden analizar datos
- Se busca dividir cada resultado posible en distintos segmentos más reconocibles, para luego reconocer relaciones entre ellos
- Al intentar llevar esas reglas, hasta un programa es demasiado complejo

#### 4.1.4. Cambio de paradigma

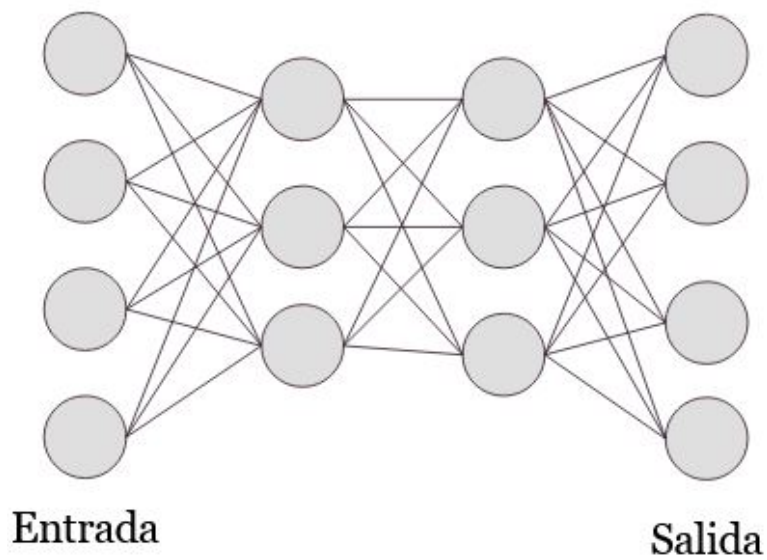
Normalmente, se programan reglas para que, a partir de una entrada se genera una salida.



Ahora en redes neuronales, usamos un algoritmo genérico que a partir de la entrada y la salida esperada, generemos las reglas o patrones a buscar para lograr dicha salida.



#### 4.1.5. Red Neuronal

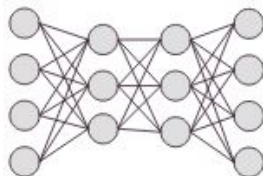


De entrada, tenemos los datos iniciales (todos datos numéricos, de preferencia normalizados), cada 'columna' de neuronas es una capa:

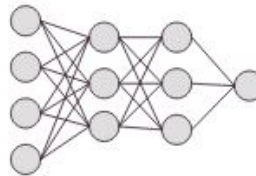
- 1 capa de entrada
- n capas ocultas que procesan los datos
- 1 capa de salida, que representa el resultado.

Cada neurona tiene de entrada la salida de todas las neuronas anteriores. Finalmente, la 'probabilidad' de que la entrada corresponda a esa característica.

Sin embargo, lo último no siempre es así, dependerá de el trabajo que están realizando con la red.



**Clasificación**  
(Probabilidad)



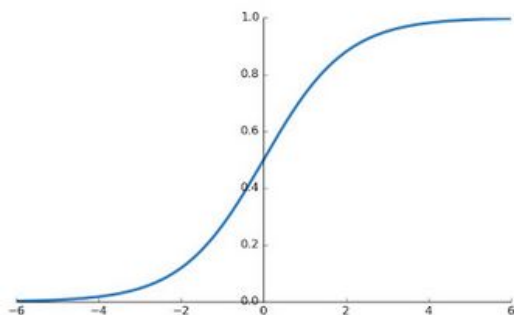
**Predicción de valores**  
(valores directos)

#### 4.1.6. La Neurona

La estructura que cobija a una neurona es análoga a un nodo, la cual guarda un dato valorizado entre 0 y 1, y referencia hacia los nodos de entrada. El dato permite generar 2 valores útiles:

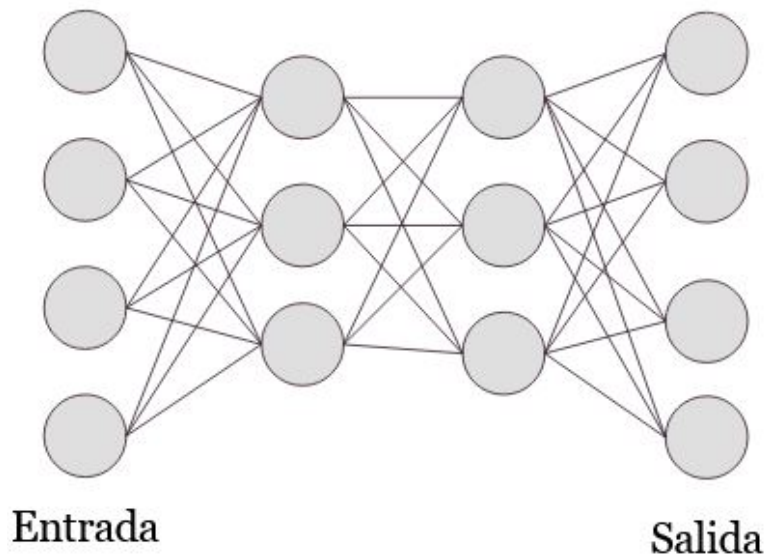
- Valor activado (resultado de una función de activación)
- Valor derivado

La **función de activación** cambia el valor de una neurona por otro, por lo que permite trabajar datos lineales como no lineales. Para eso, existe una función recomendada llamada **el sigmoide**

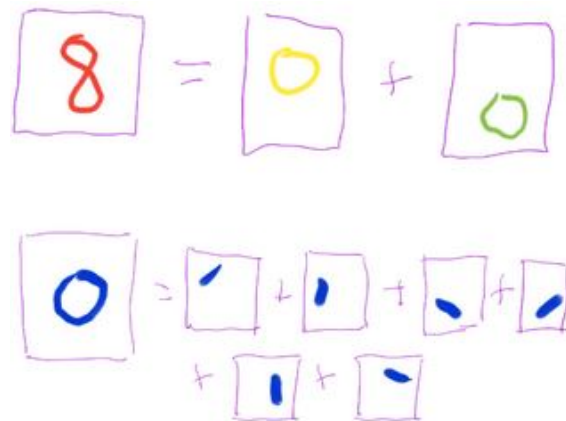


$$f(x) = \frac{1}{1+e^{-x}}$$

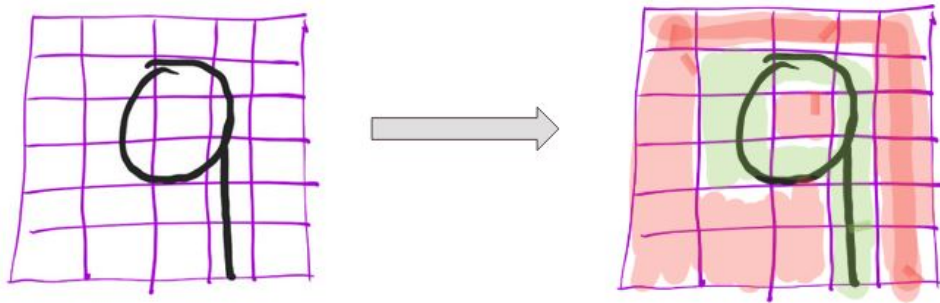
#### 4.1.7. ¿Cómo funciona una red neuronal?



Pensemos en el primer problema, se podría complejizar la búsqueda de patrones en cada capa.



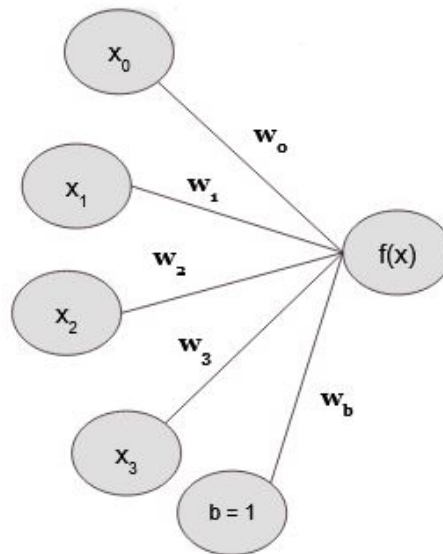
Considerando esto, debe haber una forma de dar prioridad a ciertas partes de la imagen, esto se podría conseguir dando una ponderación a cada neurona de la entrada (pixel).



De esta manera, se consigue un valor más positivo si se tienen sólo líneas en los píxeles correctos, más negativo si no. Matemáticamente hablando se tiene:

$$f(x) = \sum_i x_i * w_i$$

Donde  $x_i$  es el valor del pixel y  $w_i$  es el peso o ponderación que se le da.



Si generalizamos la función anterior, tenemos que:

$$f(x) = \left( \sum_i x_i * w_i \right) + b$$

Donde  $x_i$  es el **valor activado** de la neurona anterior,  $w_i$  es el peso que se le da a la conexión, y  $b$  es el sesgo que ajusta la función de activación para una mejor predicción.

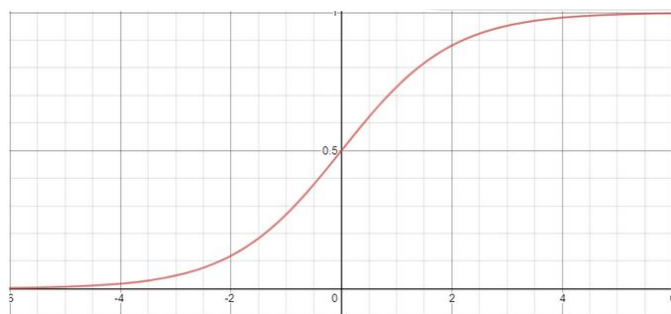


Figura 4.1: Sigmoide con  $b = 0$

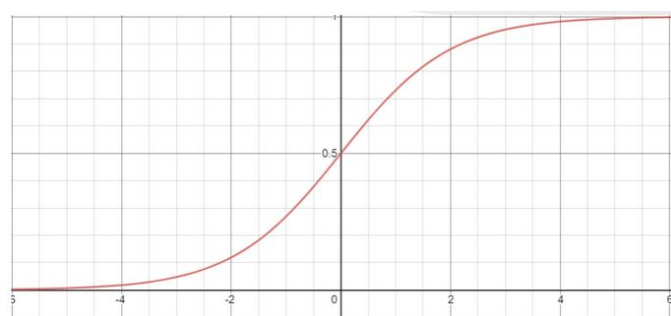


Figura 4.2: Sigmoide con  $b = -2$

#### 4.1.8. ¿Cómo aprende una red neuronal?

Se necesita un proceso para ajustar los pesos y sesgos. Como se dijo antes, se tiene la entrada y salida esperada, por lo que podemos intentar retroalimentar a la red, comparando su salida con los valores esperados.

## 4.2. Backpropagation

### 4.2.1. Recapitulación

- La red neuronal es un red de  $n$  capas, con 1 de entrada, y 1 de salida.

- La cantidad de neuronas de entrada depende de la representación que se le dé a los datos.
- La cantidad de neuronas de salida y su interpretación depende de la tarea que se espera que realice la red.
- Cada neurona de la red está conectada con todas las neuronas de la capa anterior, otorgándole una ponderación o peso a cada una, y agregando un sesgo.
- La suma ponderada +sesgo se pasan por una función de activación, que transforma ese valor a un valor real entre 0 y 1.

#### 4.2.2. Entrenamiento

- Es el proceso de aprendizaje de las redes neuronales
- Consiste en corregir los pesos y sesgos de la neurona mediante la comparación del valor predicho con el esperado.
- Para saber qué tanto hay que modificar los pesos, se utiliza un **optimizador**.
- Luego de repetir el proceso suficientes veces, se espera **que la red sea capaz de generalizar**.

#### 4.2.3. Optimizadores

- Se puede interpretar la red neuronal como una función, donde los valores de entrada son las variables.
- Los pesos y sesgos son los que hay que cambiar para obtener el resultado esperado.
- Finalmente, el objetivo es reducir el **error de predicción**. Este se puede calcular de distintas maneras, pero nos centraremos en usar el **error cuadrático medio**.

$$ECM = \frac{1}{n} \sum_{i=1}^n (y_{obtenido} - y_{esperado})^2$$

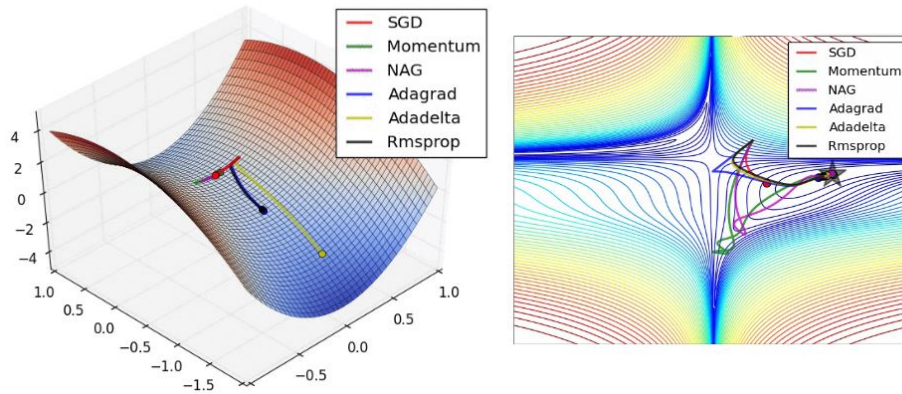


Figura 4.3: Visualización de optimizadores

#### 4.2.4. Descenso del gradiente

Es el optimizador más simple, el cual utiliza el promedio de los errores para ajustar los pesos. Cambia los pesos una vez que se han pasado por todos los ejemplos que se le proporcionan para aprender. Debido a lo anterior, el proceso de optimización es muy lento, ya que requiere que se pase por todos los datos múltiples veces.

##### 4.2.4.1. Descenso del gradiente estocástico

Es una modificación del descenso del gradiente. Este optimiza en pasos más pequeños ya que realiza el **backpropagation** con cada ejemplo. Es mucho más rápido pero realiza más pasos, además datos más diversos hacen que se pueda alejar del óptimo global

##### 4.2.4.2. Algoritmo de entrenamiento

1. Se inicializa la red con peso y sesgos aleatorios.
2. Se intenta predecir un valor con la red.
3. Se compara el valor esperado con el valor obtenido
4. Con el gradiente, se sabe qué tanto afecta cada neurona de la capa anterior, así que modifican sus pesos con respecto a esto (backpropagation).

5. Se repite hasta terminar con el conjunto de entrenamiento.

#### 4.2.4.3. Algoritmo

1. Se mezclan los datos de entrenamiento
2. Se predice y guarda el error de cada predicción
3. Se calcula el promedio de errores
4. Se propaga la corrección del error:
  - a) Se parte en la capa de salida
  - b) Se determina cuánto afecta cada peso para obtener el valor final (mediante el gradiente).
5. Una vez calculados todos los pesos, se procede a actualizarlos.

#### 4.2.4.4. Regla Delta

Es una fórmula o regla que nos permite determinar el vector de pesos sobre el que hay que modificar los pesos de una red. Asumiendo una medida del error de la red como el error cuadrático:

$$E = \sum_{i=1}^n (y_{esperado} - y_{obtenido})^2$$

Si aplicamos regla de la cadena varias veces llegaremos a lo siguiente:

$$\Delta w_{ij} = \alpha (y_{esperado} - y_{obtenido}) y'_{obtenido} * x_{entrada}$$

Esste cálculo luego se puede generalizar a un perceptrón multicapa mediante el siguiente algoritmo:

1. Con el error calculado, si la neurona  $j$  es de salida, el delta está dado por:

$$\delta_j = (f(x_i) - y_{esperado}) * f'(x_i)$$

2. Para las otras capas, el error de una neurona  $j$  se obtiene:

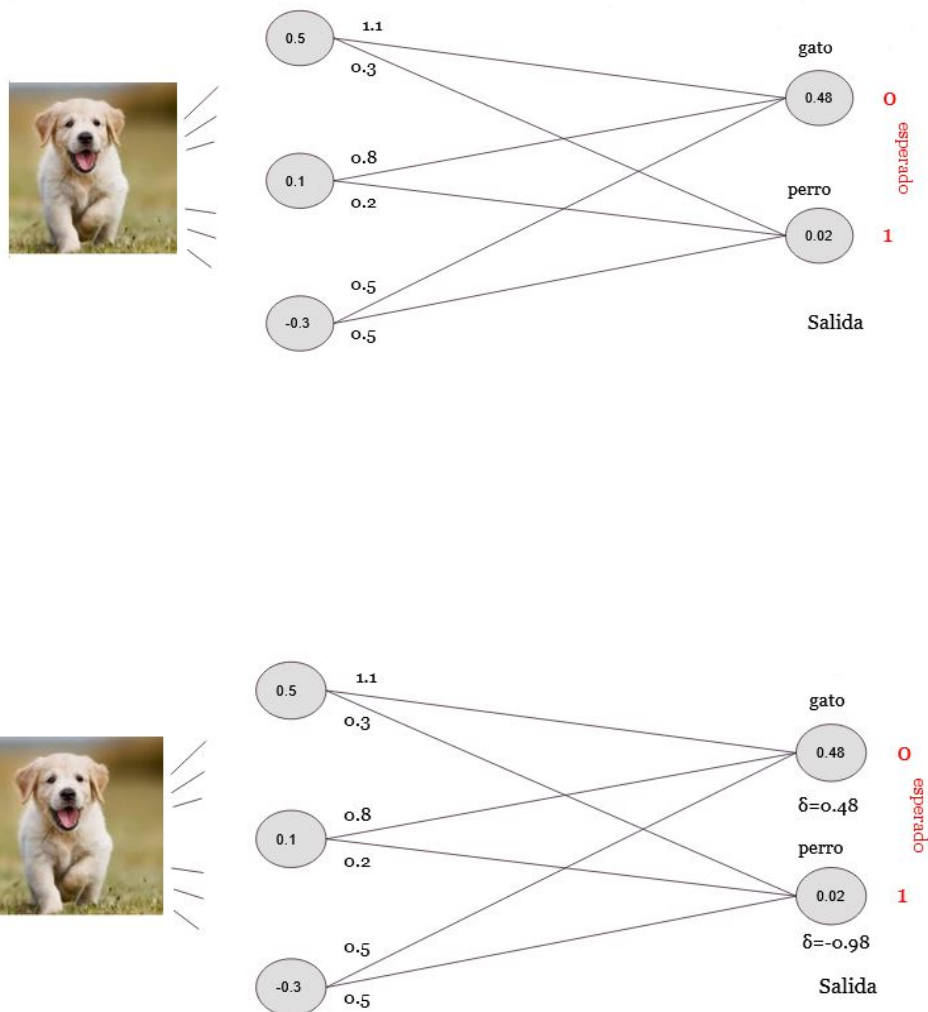
$$\delta_j = \left( \sum_{i \text{ in } I} \delta_i - w_{ji} \right) * f'(x_j)$$

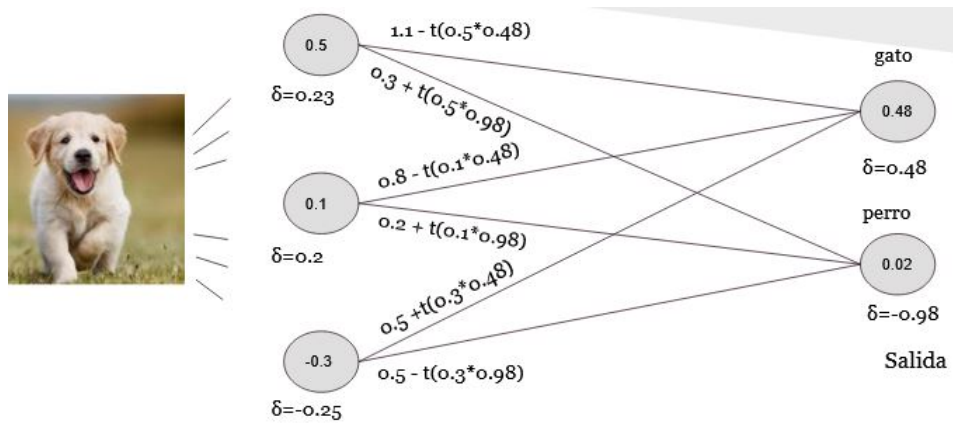
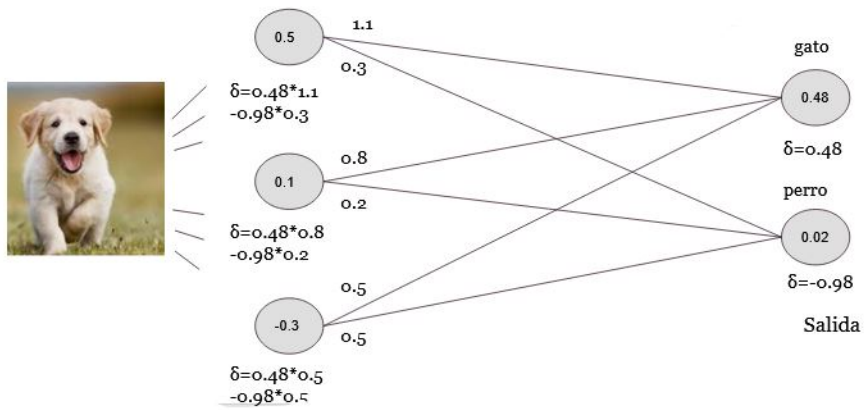
Con las  $I$  neuronas de la capa siguiente de la neurona  $j$

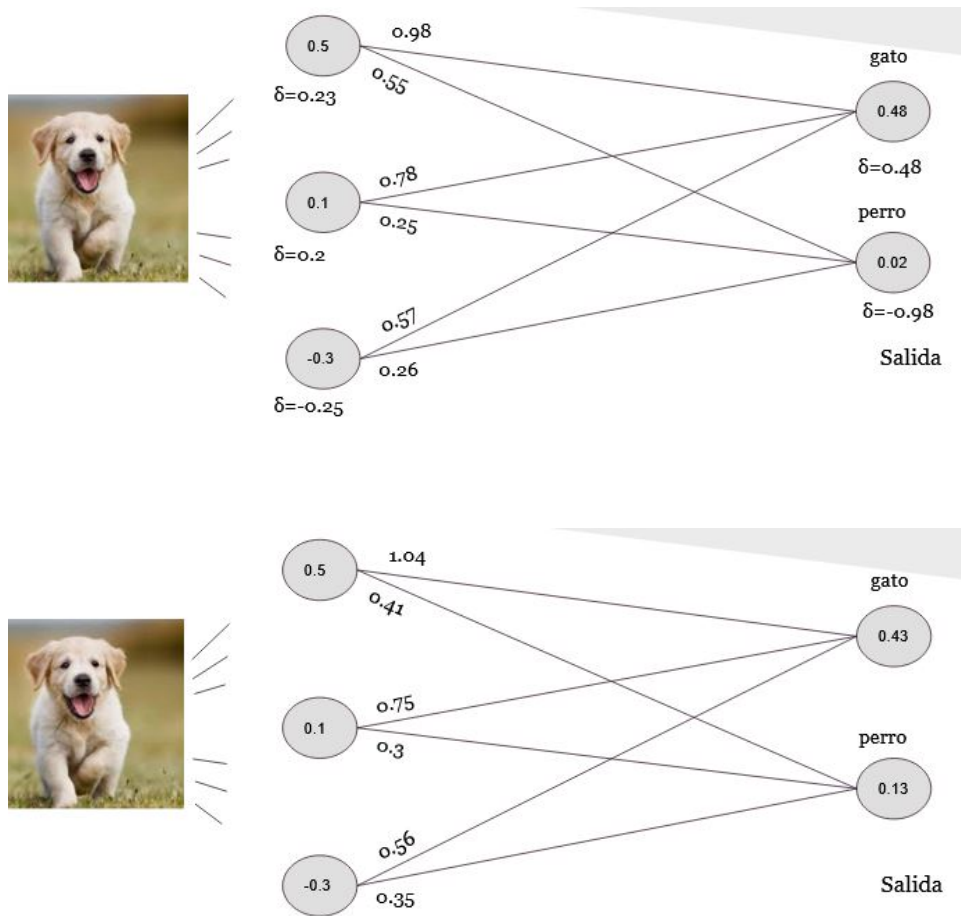
3. Luego, la modificación del peso entre una neurona  $j$  y una neurona  $i$  (de la siguiente capa) viene dada por:

$$\Delta w_{ij} = -T_{learn} * \delta_i * f'(x_j)$$

donde  $T_{learn}$  es la tasa de aprendizaje, es decir, cuánto queremos avanzar en la dirección del gradiente  $f'(x_i)$  corresponde al valor activo de la neurona  $j$ .







### 4.3. Otros tipos de redes neuronales

#### 4.3.1. Recapitulación

- Una red neuronal es un grafo dividido por n capas.
- Sus utilidades van por predicción de valores continuos hasta clasificación a partir de un número de parámetros numéricos.
- Aprende de manera supervisada, es decir, se le deben proporcionar tanto los datos de entrada como la salida esperada para que sea capaz de mejorar en su tarea

- El proceso de retroalimentación se llama **Backpropagation**, el cual, a través de un optimizador, va ajustando los parámetros con el objetivo de reducir la función de coste o error.

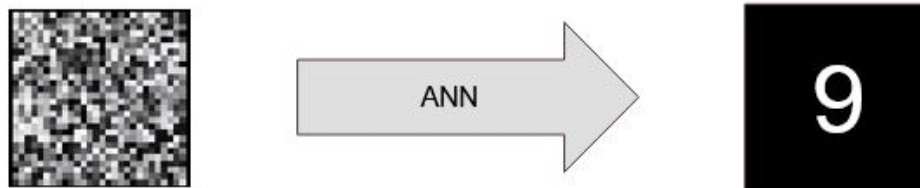
#### 4.3.2. Problemas con las redes neuronales

Si bien las redes neuronales son herramientas muy útiles, pero pueden generarse problemas si no se tiene en consideración que son **herramientas estadísticas**, y como tal, son muy dependientes de la muestra que se usa. Y es por eso que los datos de entrenamiento deben ser lo más representativos posibles.

Si bien esperábamos que la red reconociera patrones como nosotros lo interpretamos, en la práctica no podemos entender de la misma manera que ellas lo hacen.



Es por esto último, lo que podría darse el caso de datos basura sean interpretados como algo específico.



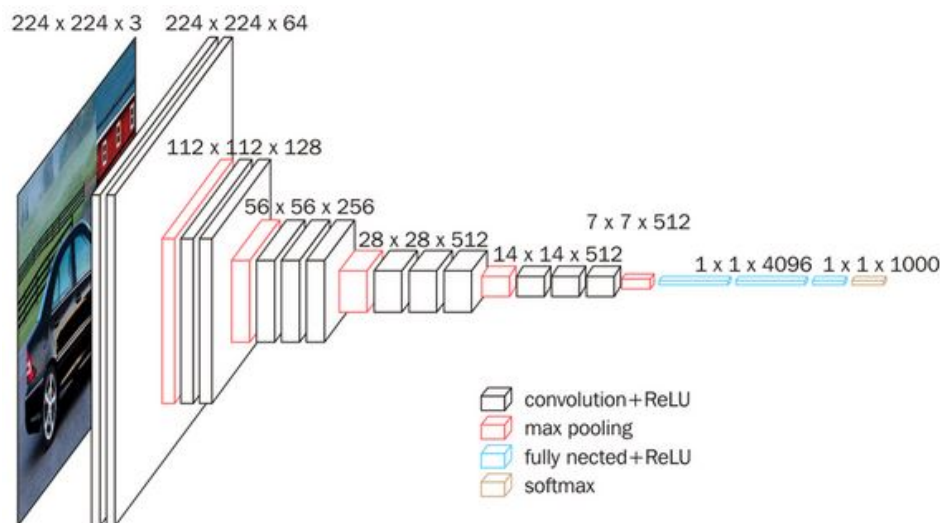
#### 4.3.3. Problemas con el perceptrón

- Un problema importante que tiene el perceptrón es que es una tecnología bastante limitada.

- Si queremos clasificar imágenes más complejas, el perceptrón no podría lograrlo con mucha eficiencia.
- Para esto y muchos otros problemas, se han creado variantes que puedan solucionarlos.

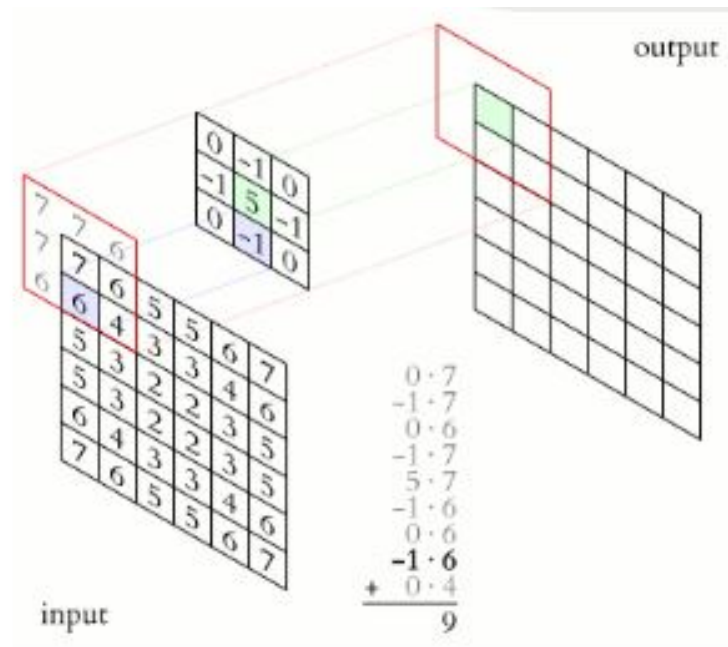
#### 4.3.4. Redes Neuronales Convolucionales

- Una variante del perceptrón que permite encontrar patrones, principalmente usado en análisis de imágenes.
- Mejora el proceso de detección de patrones haciendo el uso de filtros convolucionales.
- Se tiene la capa de entrada, capas ocultas que podrían ser convolucionales, normales (fully connected) o de max pooling, y la capa de salida.



#### 4.3.5. Neurona convolucional

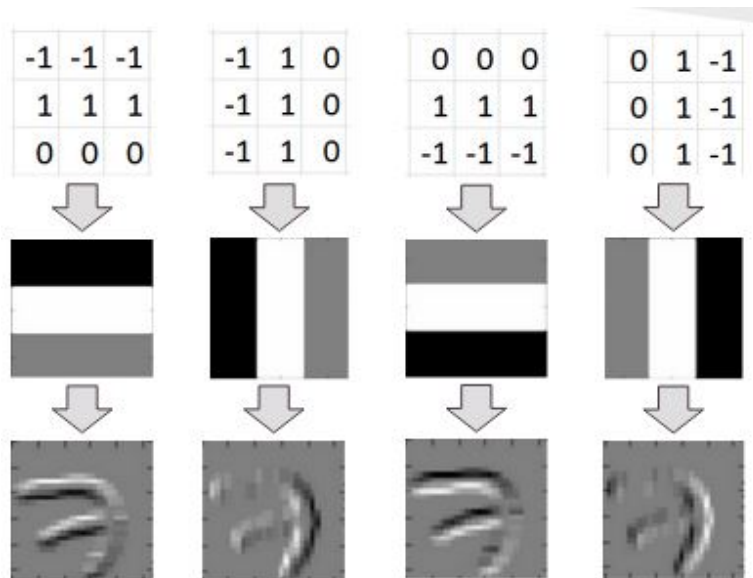
Es una neurona que aplica un 'filtro' a una imagen. Este filtro es una matriz cuadrada, que realiza el producto punto con una subsección de la imagen con las mismas dimensiones.



Teniendo un valor de entrada del tipo

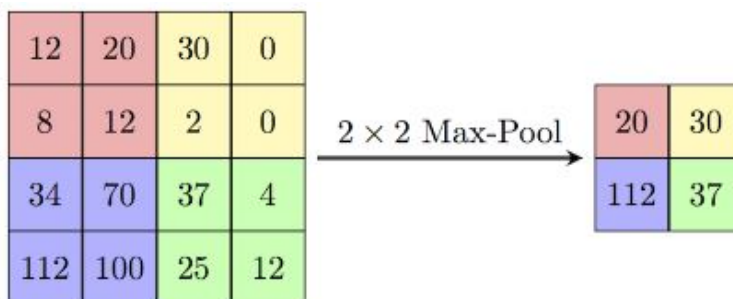


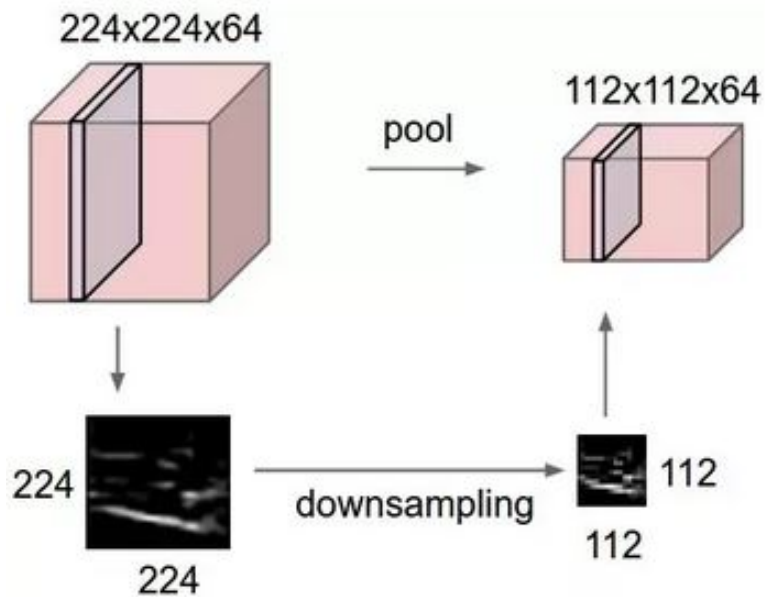
Como se puede ver, el filtro permite centrarse en patrones específicos. El primero busca bordes superiores, el tercero bordes inferiores y así sucesivamente.



#### 4.3.6. Max Pooling

Es una capa cuyo trabajo es reducir la dimensionalidad de la entrada, su utilidad es reducir la dificultad de las capas siguientes. Cada píxel de la salida es el valor máximo de una zona de la imagen de entrada.

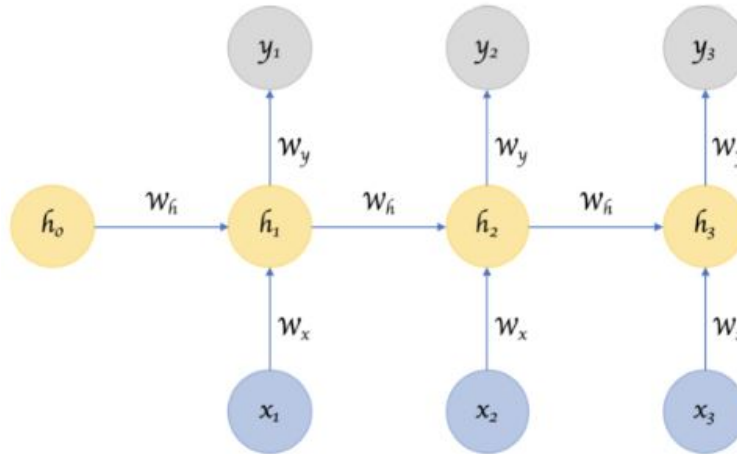




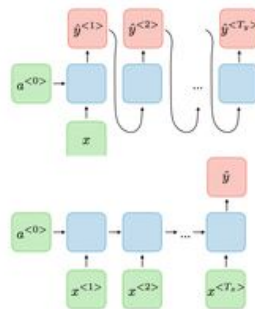
#### 4.3.7. Red Neuronal recurrente

Nacida para resolver problemas que impliquen series de entradas, cuyo valor siguiente dependa del anterior. Un ejemplo serían los fotogramas de un video o datos metereológicos.

Consiste en una capa oculta que guarda el último estado generado, haciendo uso de este cálculo del valor activado. Para esto, se interpreta el valor de salida anterior como una neurona, aplicando su peso correspondiente.



Existen muchas variantes de este diseño básico. Depende de la cantidad de salidas esperadas.



Generación a partir de un valor inicial (ej: música o texto)

Clasificación respecto al contexto (ej: sentimientos)

#### 4.3.8. Redes generativas adversarias

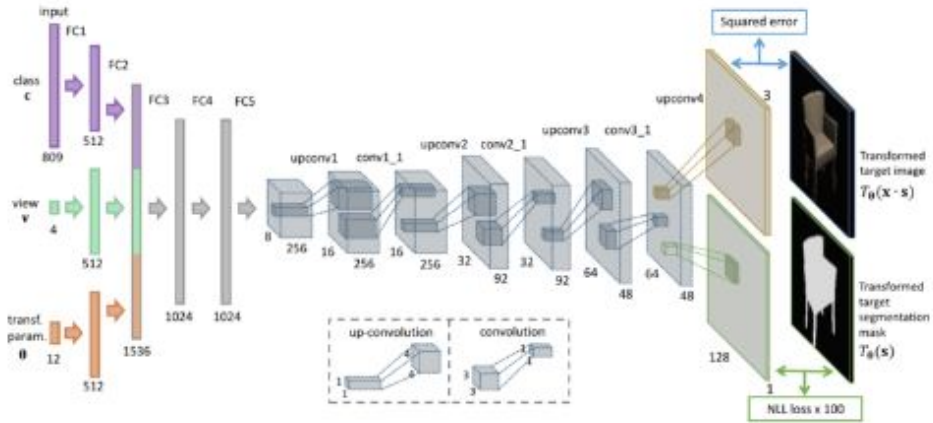
Es una arquitectura útil para el **aprendizaje no supervisado**. Consiste en 2 redes que compiten entre sí en 2 tareas diferentes:

- Una genera cosas
- La otra detecta si lo que recibe es generado o real

La red generadora es una red deconvolucional, y la discriminadora una red convolucional. Debido a que cada una utiliza la retroalimentación de la otra, ambas irán mejorando sus tareas, obteniendo muchos mejores resultados con el tiempo.

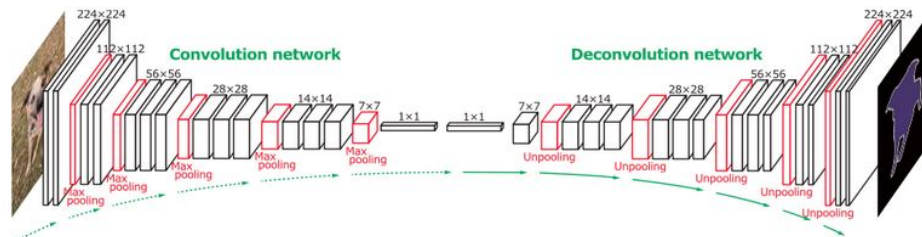
### 4.3.9. Red Deconvolucional

Parecido a una red convolucional, pero al revés. Va aumentando la dimensionalidad, generando finalmente el producto con el que se le fué entregado.

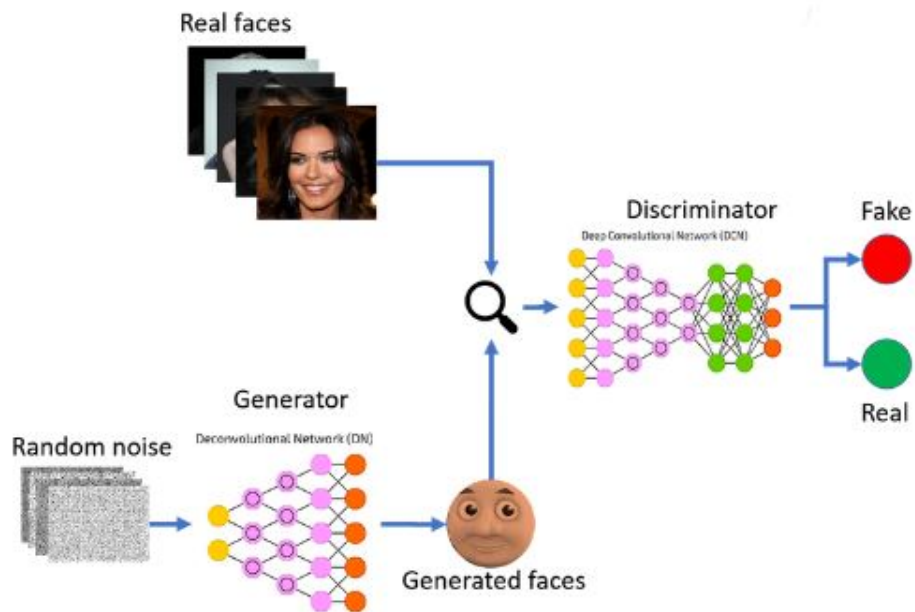


#### 4.3.9.1. Entrenamiento

Se usa la misma arquitectura en una red convolucional, para luego volver a aumentar la dimensionalidad.



#### 4.3.10. Redes generativas adversarias



- Sus aplicaciones son variadas, entre ellas:
  - Creación de imágenes
  - Autocompletado de texto
  - Edición de video
  - Descripción de imágenes a texto
- Variantes de GAN

## Capítulo 5

# Indexación de Motores de búsqueda

### 5.1. Motores de búsqueda

Suponga que se tiene una biblioteca personal, y desea buscar un tema específico entre todos los libros disponibles. Si esta librería es pequeña, es fácil buscar libro por libro. Si esta librería es pequeña, es fácil buscar libro por libro, sin embargo, ¿Qué sucede cuando la librería es demasiado grande, como una biblioteca escolar o una municipal? ¿Cómo se maneja la búsqueda las páginas/aplicaciones web?

Para la búsqueda de cualquier tema, palabra o frase (incluso imágenes) se utiliza la ayuda de **motores de búsqueda** que realizan un barrido rápido entre toda la información disponible y muestre al usuario lo que desea encontrar.

Los motores de búsqueda se componen de 2 elementos:

- Predicción
- Analizador

## 5.2. Predictor léxico

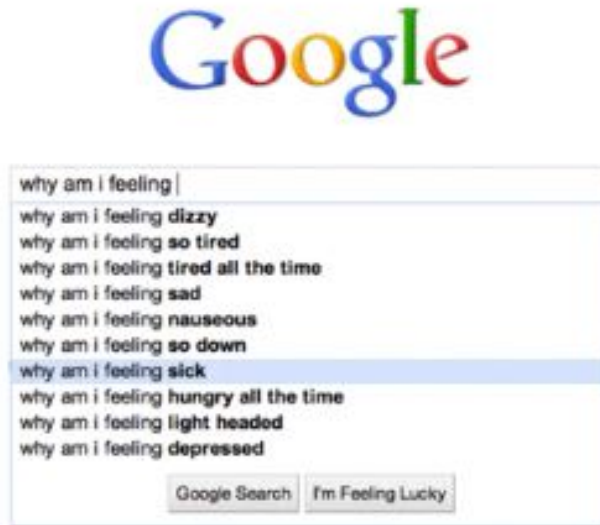


Figura 5.1: Predictor léxico - Google

Antes de que el usuario realice una búsqueda, los predictores léxicos están analizando cada cambio en el cuadro de texto para mostrar posibles búsquedas. Los predictores léxicos más avanzados utilizando modelos de aprendizaje automático para detectar los patrones de búsqueda por idioma.

Los motores más básicos, en cambio, utilizan estructuras de datos específicos para el manejo del predictor: *Trie* y *Suffix-Tree*.

### 5.2.1. Trie

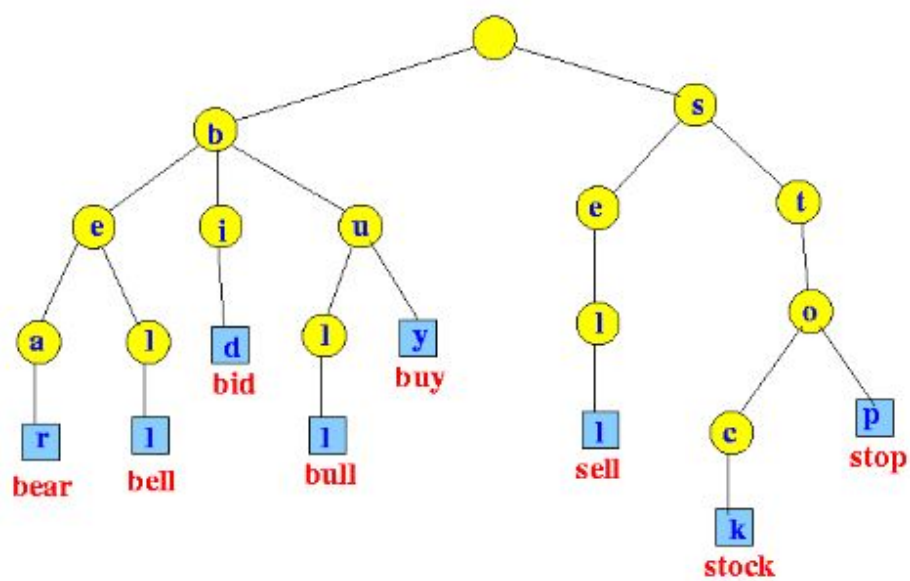


Figura 5.2: Trie

Es un árbol especializado en guardar palabras. Separa la frase en letras o palabras y las ramas indican la probabilidad de que el usuario continúe por ese camino.

Consideremos la siguiente estructura:

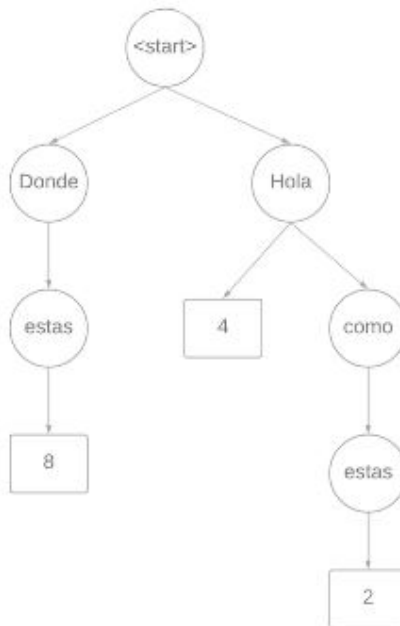


Figura 5.3: Ejemplo Trie

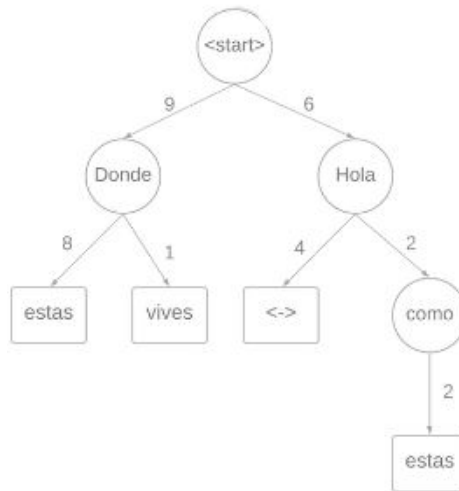
Queremos insertar la pregunta al usuario '¿Dónde vives?'

- Primero, podemos retirar los signos de pregunta
- Luego, vamos comparando nivel a nivel las diferencias y semejanzas que posean los nodos con las palabras.
- A la primera diferencia se inserta el/los nodos necesarios y se inicializa el contador en 1.
- En otro caso, se aumenta el contador en 1.

El entrenamiento de la estructura se realiza en base a los datos iniciales y el uso de los mismos usuarios.

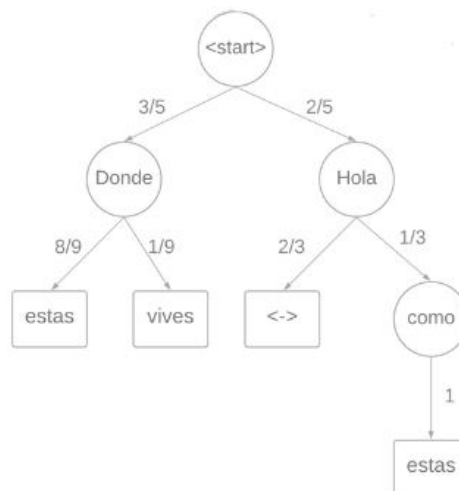
Existen motores de búsqueda especializados (Google, Facebook, entre otros) que poseen un híbrido entre distintas versiones de la estructura. Actualmente, la lógica de la estructura es ampliamente utilizado.

La búsqueda se realiza en base a los contadores finales. El proceso es muy lento mientras mayor sea la cantidad de datos, ¿Cómo lo mejoramos?



Eliminamos el nodo contador y asignamos los nodos terminales a la última palabra de la frase. Las conexiones poseen la cantidad de veces que se continúo por ese camino.

Se puede normalizar los valores para obtener una probabilidad de que se escoja la frase. Estas probabilidades se escogen entre 0 y 1 para aplicar propiedades de probabilidades.



La probabilidad de que el usuario quiera escribir '*Hola como estas*' desde

el inicio es de:

$$\frac{2}{5} * \frac{1}{3} * 1 = \frac{2}{15}$$

En cambio, la probabilidad de escribir lo mismo desde el nodo '*Hola*' es

$$\frac{1}{3} * 1 = \frac{1}{3}$$

El cálculo de la probabilidad se realiza cada vez que el usuario completa una búsqueda, ya que siempre existe la posibilidad de eliminar palabras ya escritas. Para una correcta estimación, normalmente se eligen solo las frases que tengan una probabilidad mayor a 10 %.

### 5.3. Analizador Léxico

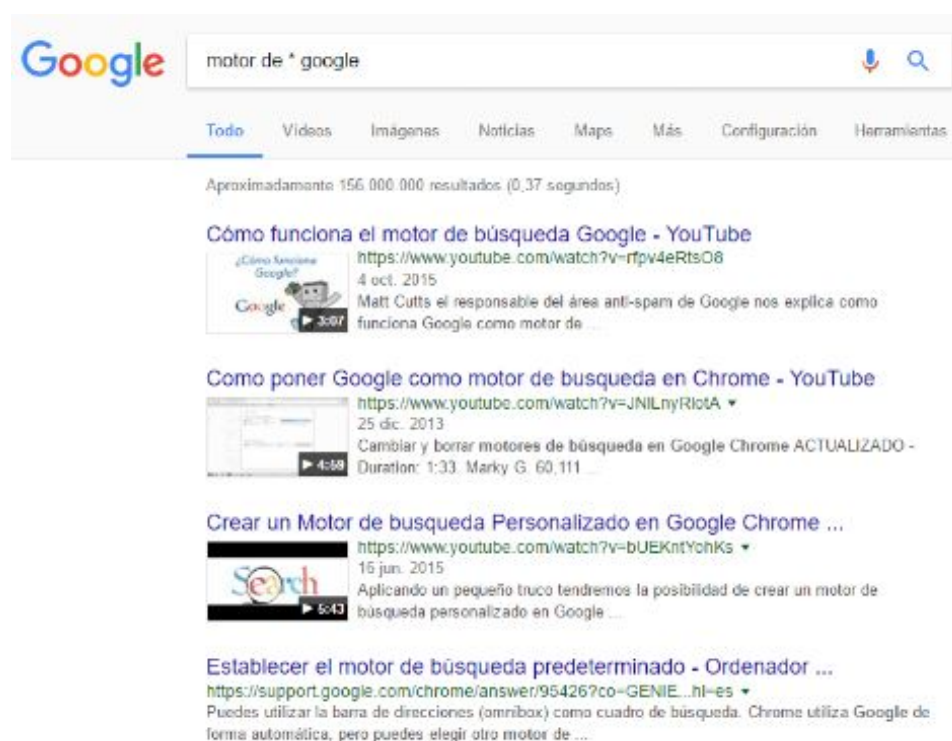


Figura 5.4: Analizador léxico

Después de que el usuario ingresa la frase, los analizadores léxicos buscan la frase en el contexto de cada archivo posible y los ordena por la probabilidad

de éxito.

En esta sección se consideran varios factores en la frase:

*Se eliminan predicados innecesarios, se busca por contexto y luego por palabra en sí.*

Los analizadores léxicos se componen de estructuras principales que almacenan la información y algoritmos de apoyo que reducen el tiempo de cómputo.

Para realizar un análisis léxico correcto, es necesario indexar las palabras de los documentos de tal manera que sea fácil de detectar la búsqueda.

**Hot-Encoding** permite identificar la existencia de las palabras en un texto, pero no poseen información del contexto.

**Word 2 Vec** soluciona el problema anterior, identificando la existencia de la palabra en el contexto necesario.

La búsqueda se compone de 2 secciones: un optimizador de tiempo encargado en entregar el resultado más rápido posible y una base de datos que posee toda la información preprocesada.

### 5.3.1. Optimizador temporal

Podemos realizar una búsqueda documento a documento, pero sería demasiado costoso para la cantidad de datos a analizar.

Si realizamos un filtro de los documentos (Contexto del documento x Contexto del usuario) podemos reducir el tiempo de cómputo considerablemente).

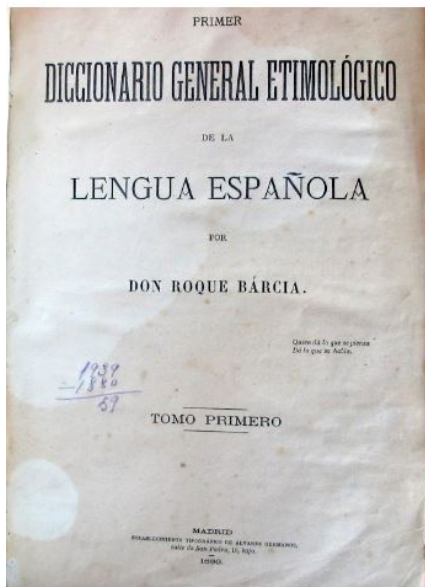
### 5.3.2. Contexto del documento

El contexto del documento parte con información básica (año de creación, última modificación, lenguaje detectado) para completarse con etiquetas de la información que posee (noticias, entretenimiento, RRSS, etc).

Estas etiquetas se obtienen al analizar los contenidos de los documentos.

### 5.3.3. Hot-encoding

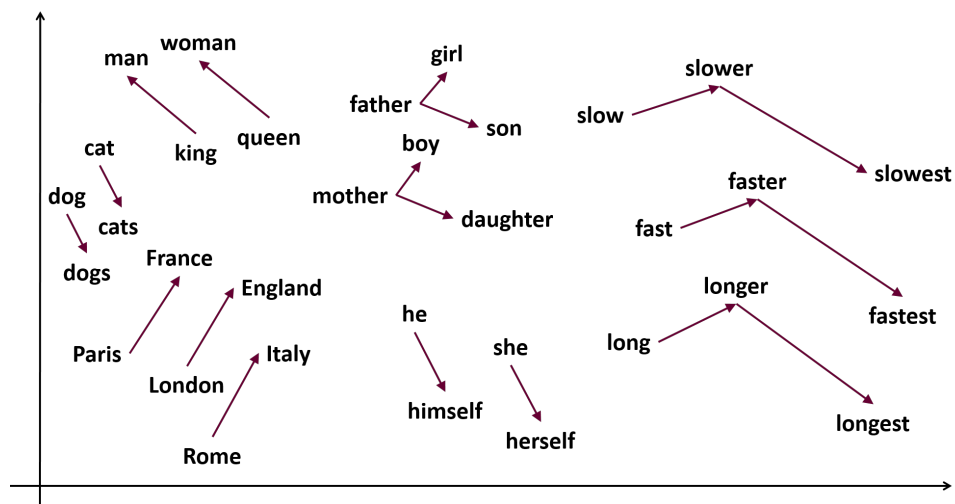
Se pretende transformar un documento en un vector con la cantidad de veces que existe cada palabra del diccionario en el documento. Si solo considera existencia de las palabras es considerado one hot encoding.



⇒ [1,4,65,23,12,32,4...]

#### 5.3.4. Word2Vec

Esta técnica crea un vector similar al anterior, sólo que genera relaciones entre palabras similares.



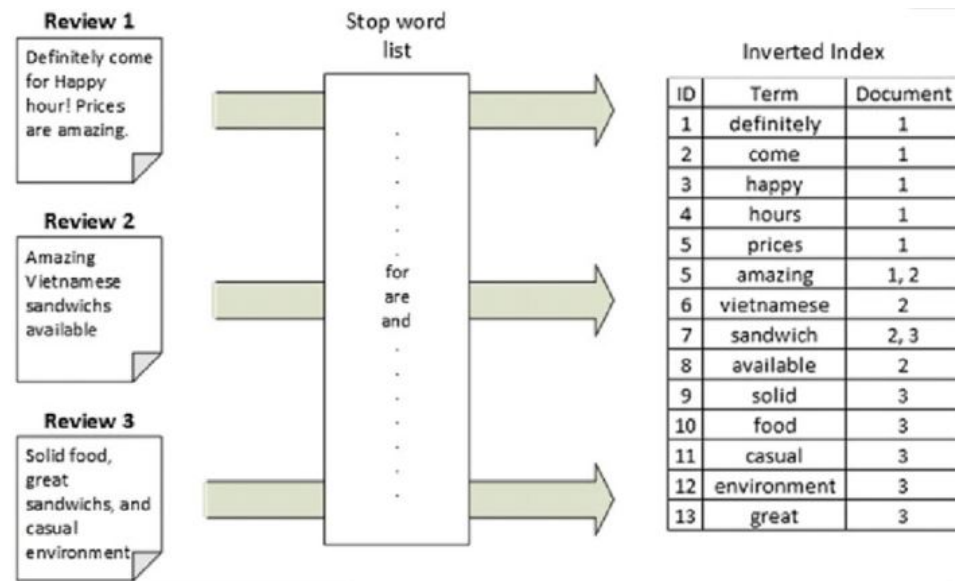
Esta relación se obtiene en base a la creación de vectores del tipo (entrada, salida) para entrenar modelos predictivos y que estos simplifiquen la búsqueda en los documentos. Ejemplo:

*'El Zorro marrón rápido saltó sobre el perro perezoso'*

([el, marrón] rápido), ([zorro, rápido] marrón), ([marrón, saltó] sobre),...

### 5.3.5. Inverted Index

Al contrario de las técnicas anteriores esta técnica almacena en una base de datos la existencia de la palabra en un documento, con prioridad las palabras.



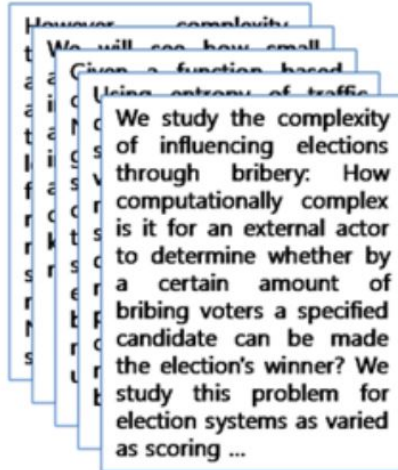
### 5.3.6. Document-term Matrix

Versión más compleja del anterior, almacena la cantidad de veces que cada palabra se repite en cada documento.

Documents



Vector-space  
representation



	D1	D2	D3	D4	D5
complexity	2		3	2	3
algorithm	3			4	4
entropy	1			2	
traffic		2	3		
network		1	4		

Term-document matrix