UTEC
UNIVERSIDAD DE INGENIERÍA
Y TECNOLOGÍA

Welcome to Algorithms and Data Structures! - CS2100

Búsqueda en grafos

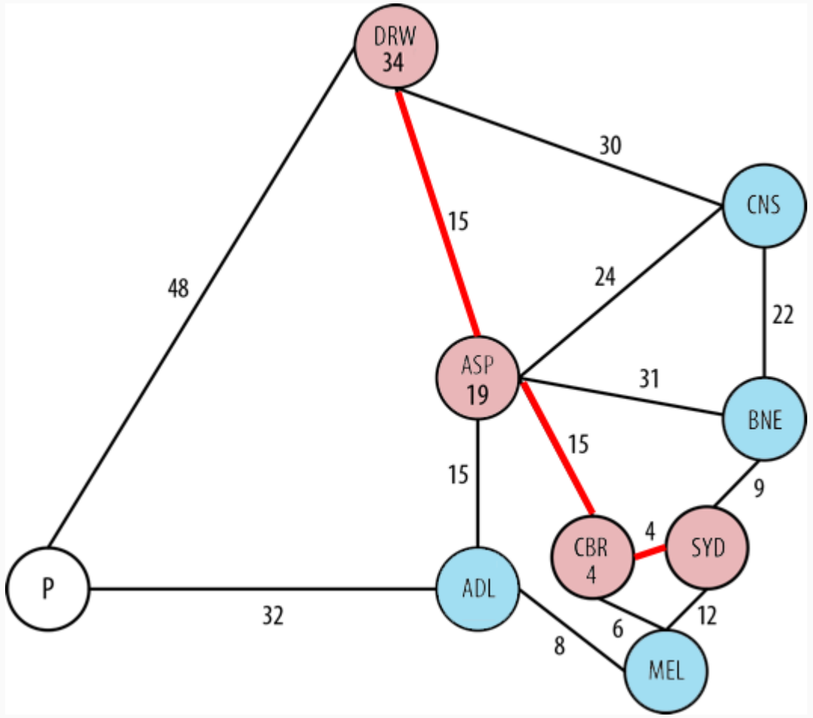
Se refiere a la exploración en un grafo.

Cada manera de explorar un grafo puede utilizar una estructura de apoyo diferente

El peor de los casos es $O(|V|^2)$

Para cada caso se utilizará memoria como ayuda. Recordando los vértices que ya han sido visitados

Se tiene varias aplicaciones: encontrar los vértices conexos, encontrar ciclos, caminos más cortos, etc

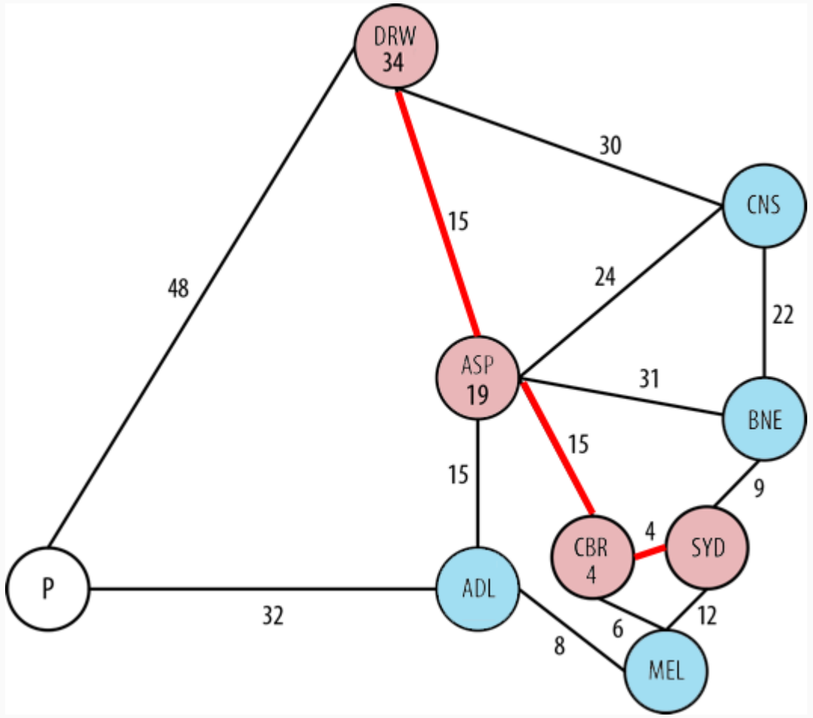


Búsqueda en grafos

La idea básica es ir expandiendo un árbol de estados al generar sucesores (estados) de un vértice ya visitado.

Cada estado es evaluado para saber si ya cumplimos con nuestro objetivo.

Se debe usar alguna heurística para generar sucesores, como el siguiente vértice al que se llega con menor peso, o uno aleatorio.

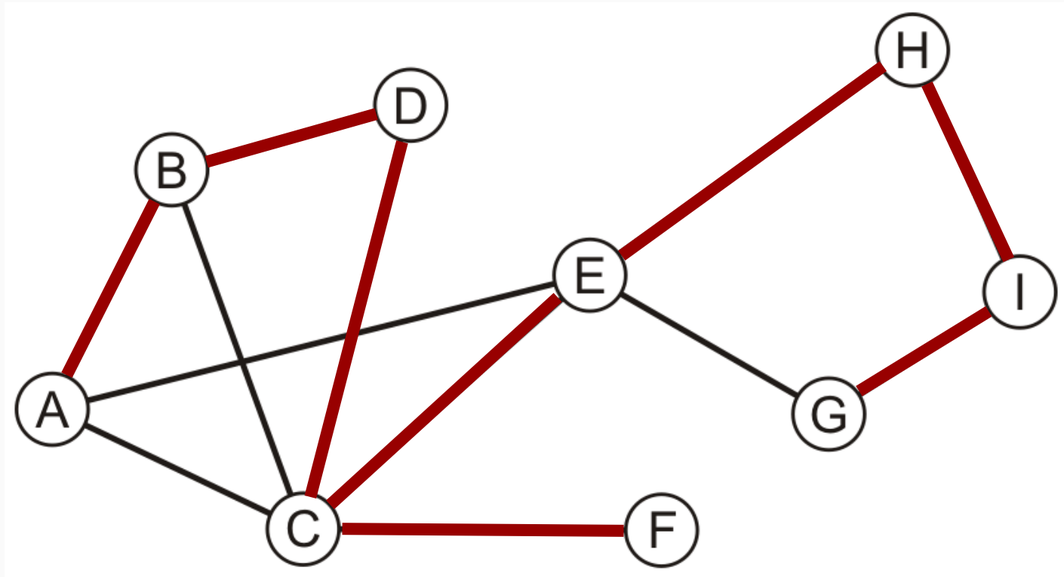


Búsqueda en profundidad (DFS)

Para explorar un grafo de esta manera vamos a necesitar una pila (stack)

DFS puede ser implementado de manera recursiva o iterativa

1. Elige cualquier vértice u, y agrégalo al stack marcándolo como visitado
   a. Desde u, si hay algún vértice v que no haya sido visitado agrégalo al stack y continúa la búsqueda desde v
   b. De otra forma, remueve el vértice u del stack y continúa desde el siguiente en la pila
2. Continúa el proceso hasta que no hayan más vértices que visitar

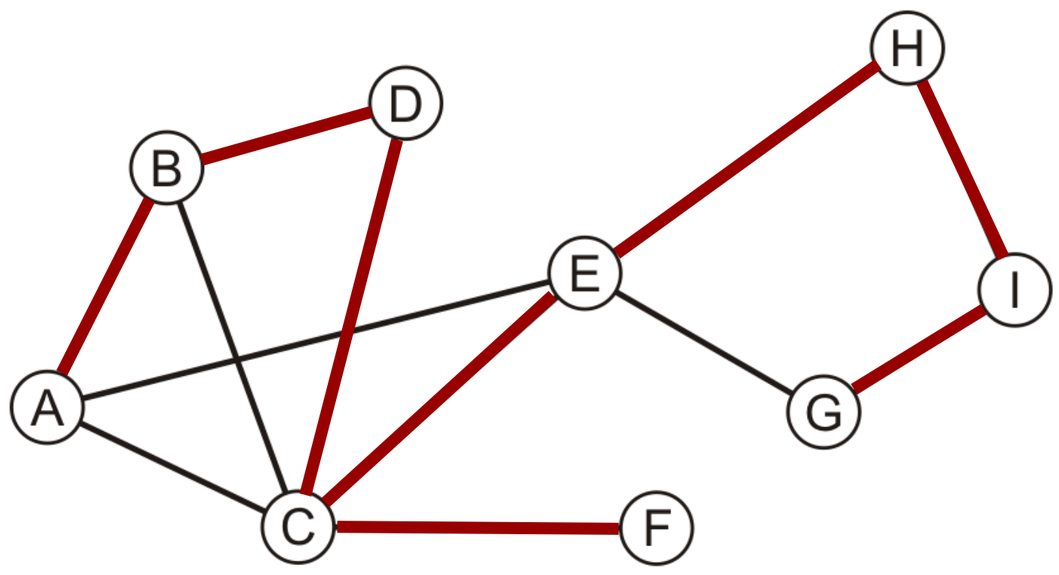


Búsqueda en profundidad (DFS)

```
function DFS(Graph G(V, E))
{
    for (i = 0; i < V.length; i++)
        visited[i] = false;

    for (i = 0; i < V.length; i++)
        if (!visited[i])
            DFS(G, i);
}

function DFS(Graph G(V, E), int i)
{
    visited[i] = true; //add to solution
    foreach(v[j] adjacent to v[i])
        if (!visited[j])
            DFS(G, j);
}
```



O(|V|+|E|)

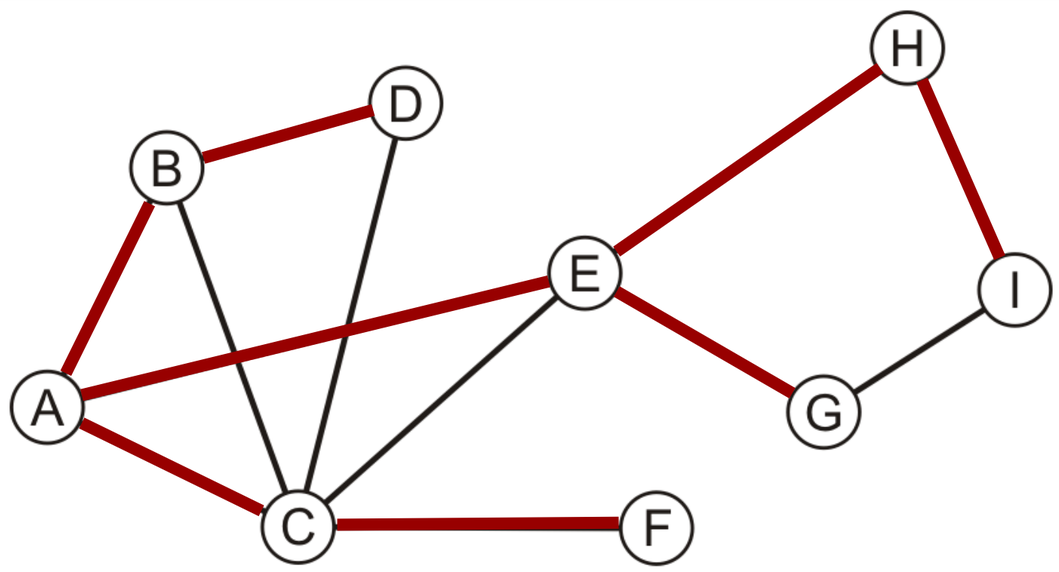
https://www.cs.usfca.edu/~galles/visualization/DFS.html
https://visualgo.net/es/dfsbfs

Búsqueda en amplitud (BFS)

Para explorar un grafo de esta manera vamos a necesitar una cola (queue)

El tamaño de la queue puede llegar a ser O(|V|)

1. Elige cualquier vértice u, y agrégalo a la queue marcándolo como visitado
   a. Pop el vértice v en la cima de la queue
   b. Agrega todos los vértices adyacentes a v que no hayan sido visitados
   c. Marca como visitados todos los vértices agregados a la queue
2. Continúa el proceso hasta que no hayan más vértices que visitar

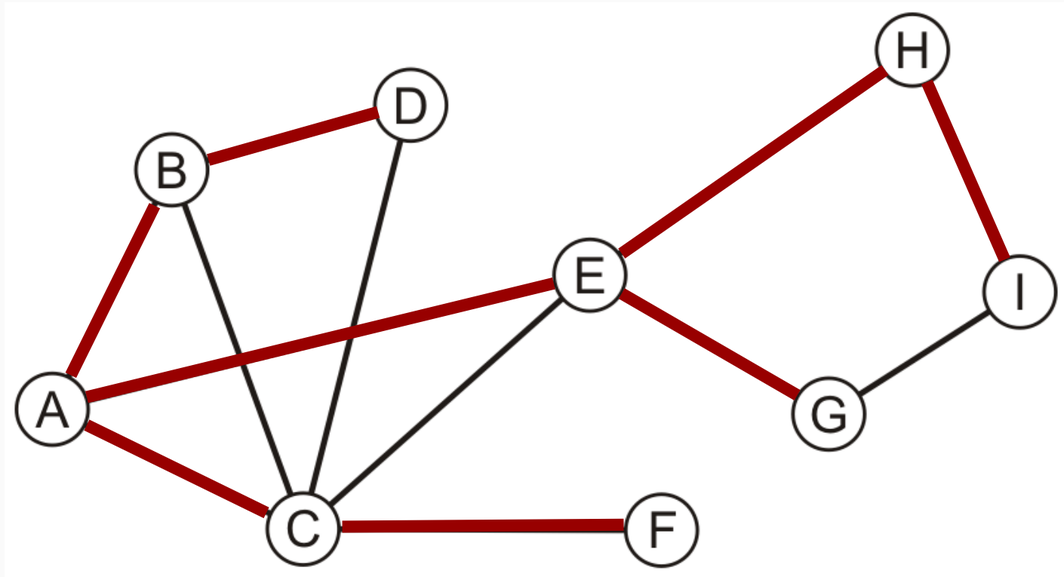


Búsqueda en amplitud (BFS)

```
function BFS(Graph G(V, E))
{
    for (i = 0; i < V.length; i++)
        visited[i] = false;
    for (i = 0; i < V.length; i++)
        if (!visited[i])
            BFS(G, i);
}

function (Grafo G(V, E), int i)
{
    Queue Q;
    visited[i] = true; // add to solution
    Q.add(i);
    while (!Q.empty()) {
        x = Q.extract();
        foreach(v[j] adjacent to v[x])
            if (!visited[j]) {
                visited[j] = true;  // add to solution
                Q.add(i);
            }

    }
}
```



O(|V|+|E|)

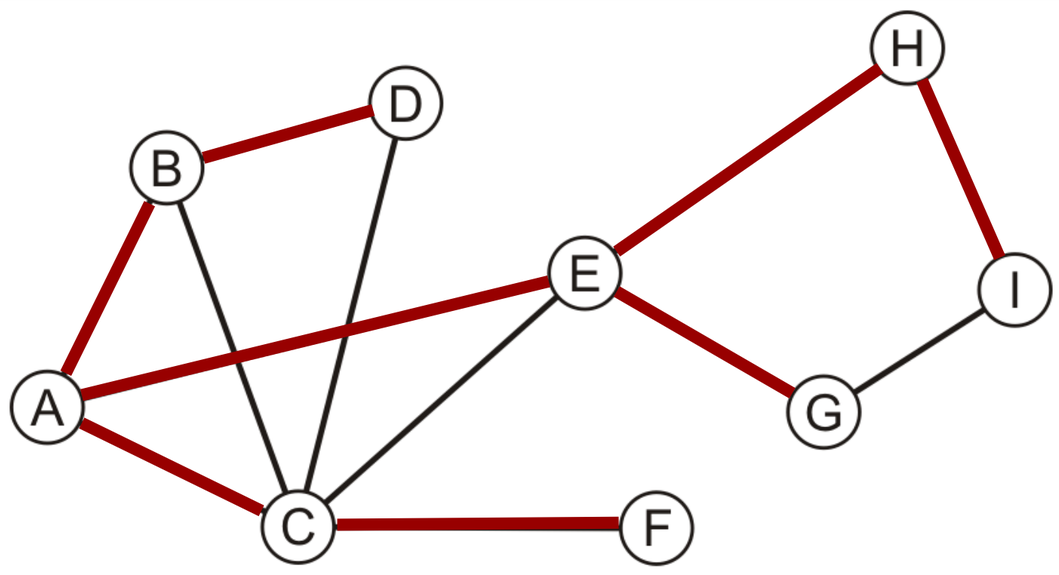
https://www.cs.usfca.edu/~galles/visualization/BFS.html
https://visualgo.net/es/dfsbfs

Búsqueda en amplitud (BFS)

¿Cómo harían para obtener el camino de un vértice inicial al final?

Se tendría que guardar (quizás en un map) el padre del nodo actual para poder iterar hacia atrás. Por ejemplo:

P(B) = A, P(E) = A, P(C) = A, P(D) = B, P(H) = E, P(G) = E, P(I) = H, y P(C) = F



Comparación

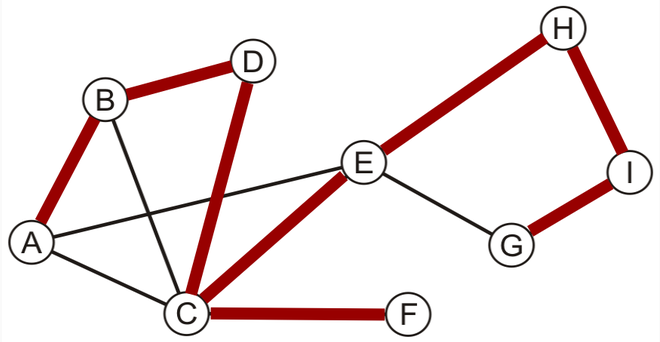
DFS es un algoritmo basado en vértices, mientras que BFS es basado en aristas. Utilizan estructuras de apoyo diferentes.

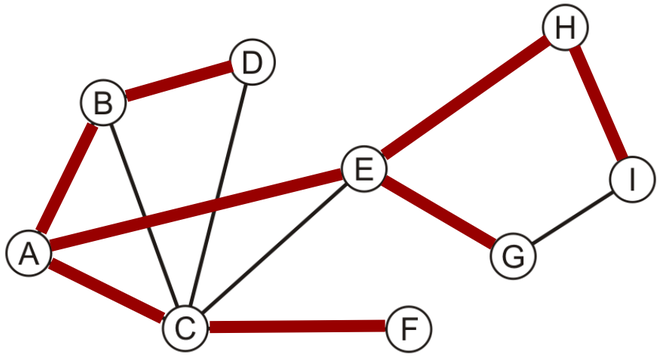
El uso de memoria en BFS es ineficiente

BFS puede encontrar el camino más corto en grafos con aristas del mismo peso, grafos bipartitos, etc

DFS se utiliza para obtener el ordenamiento topológico, saber si un grafo es fuertemente conexo

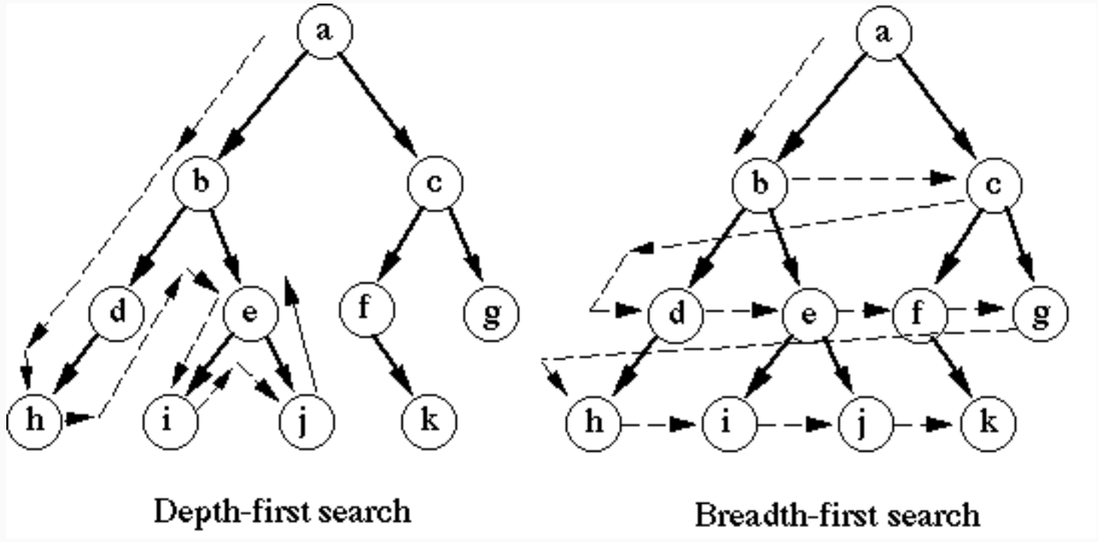
Cómo serán los árboles que generan?
DSF son árboles profundos y estrechos mientras que BFS produce árboles pequeños y amplios





Aplicaciones

1. Determinar la conectividad de un grafo.
2. Encontrar el camino de un vértice a todos los demás
3. Probar si un grafo es bipartito
4. Análisis de una red y sus relaciones
5. Detección de ciclos
6. Ordenamiento topológico
7. Detección de árboles

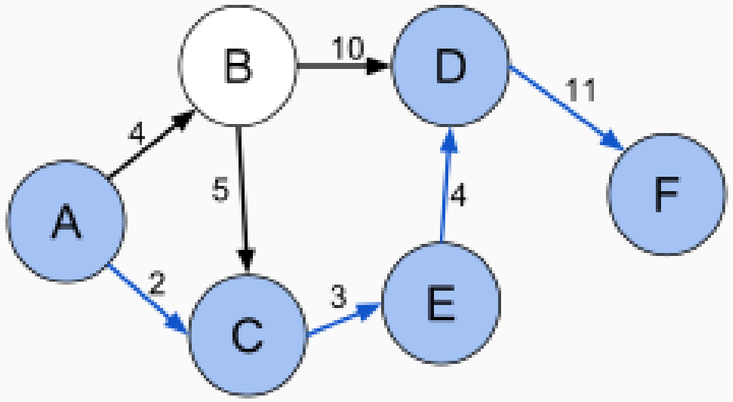


Camino más corto

Estos algoritmos buscan encontrar el camino entre dos vértices, tal que la suma de los pesos de las aristas sea el mínimo posible

BFS puede resolver este problema en un grafo no ponderado

El problema del camino más corto puede existir en grafos dirigidos o no dirigidos



Dijkstra

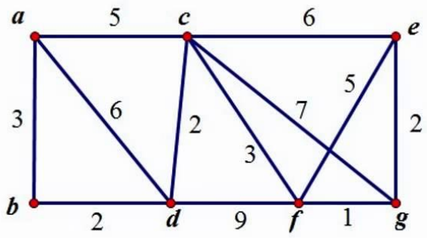
Es un algoritmo para obtener el camino mínimo en un grafo de un vértice a otro.

Este algoritmo solo funciona con aristas de valores positivos, ya que asume que agregar una arista sólo incrementará el peso total del camino

Básicamente construye un árbol de los caminos más cortos de un vértice fuente a todos los demás nodos

El algoritmo termina cuando se visitan todos los vértices

Se utiliza una tabla para mantener las distancias, las cuales serán inicialmente infinito excepto para el vértice fuente



v	a	b	c	d	e	f	g
a	0_a	3_a	5_a	6_a	∞_a	∞_a	∞_a
b	0_a	3_a	5_a	5_b	∞_a	∞_a	∞_a

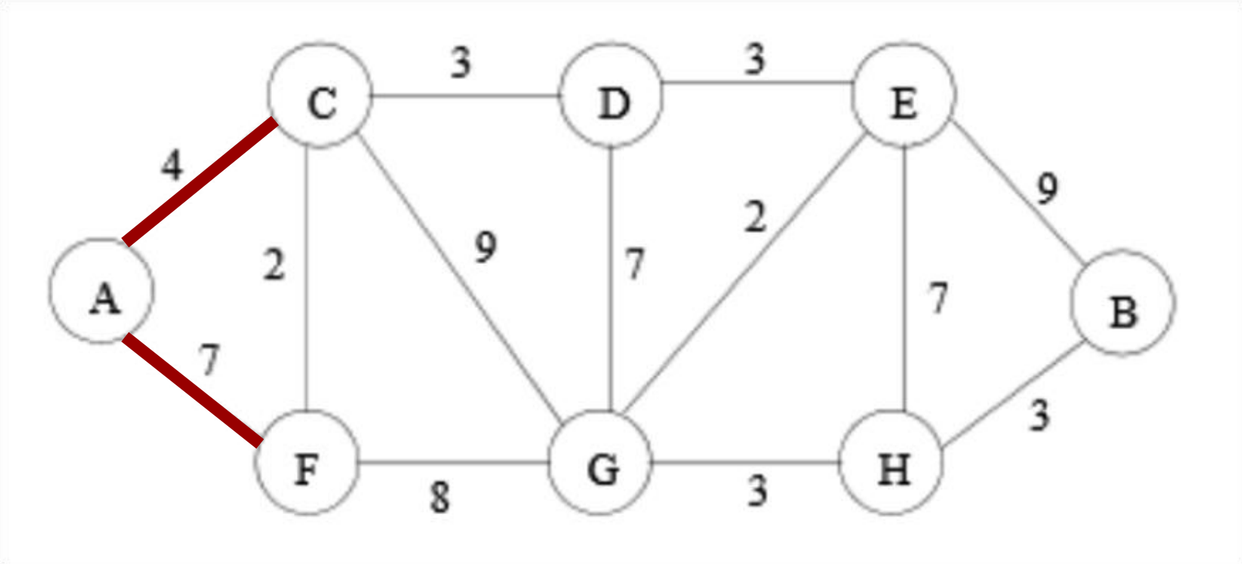
Dijkstra

Con un grafo conexo dirigido o no dirigido ponderado, se debe seleccionar un vértice inicial *u*. Se usará un arreglo de vértices visitados, y un arreglo para las distancias de un vértice *x* a uno *y*.

1. Todas las distancias entre todos los vértices empiezan con infinito, excepto el vértice *u* que iniciará con 0
2. Tomamos como *actual,* a nuestro vértice inicial *u*
3. Se recorren todos los vértices adyacentes a *actual,* excepto por los ya visitados y se calcula la distancia tentativa. Si la distancia tentativa es menor que la distancia almacenada, esta se actualiza.
4. Una vez se hayan completado todos los vértices adyacentes, *actual* se marca como visitado (un vértice visitado jamás será visto de nuevo).
5. Se toma como próximo *actual* el de menor distancia (se puede usar una cola de prioridad o un vector) y se regresa al paso 3, mientras existan vértices no visitados.

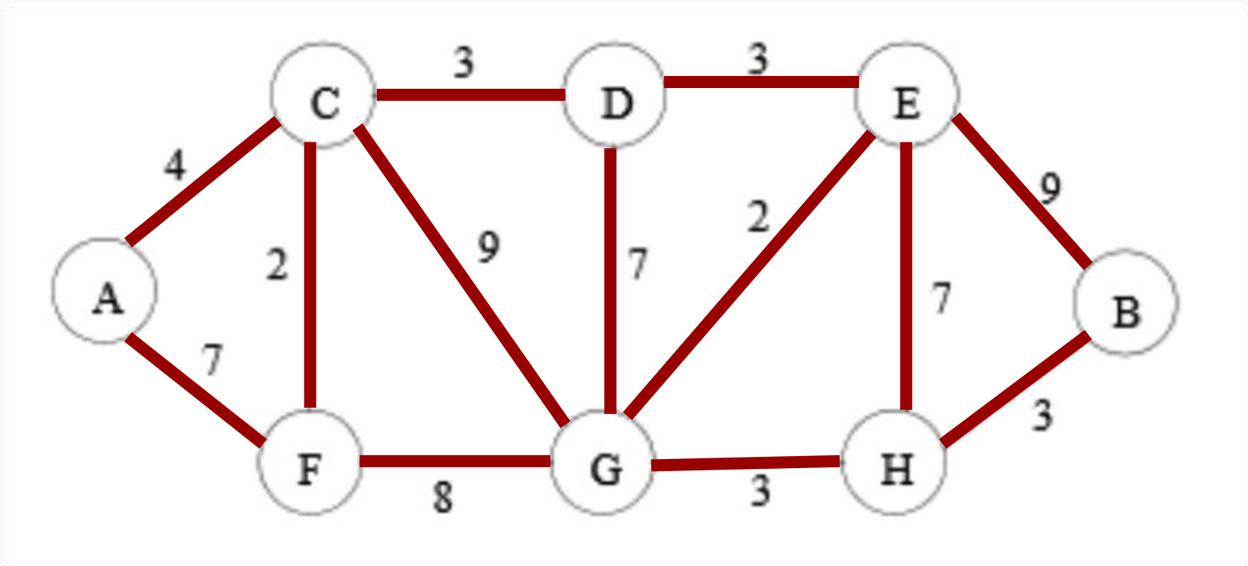
Una vez terminado al algoritmo, tendremos todas las distancias más cortas desde el vértice inicial *u*, a todos los demás vértices.

Dijkstra (A)



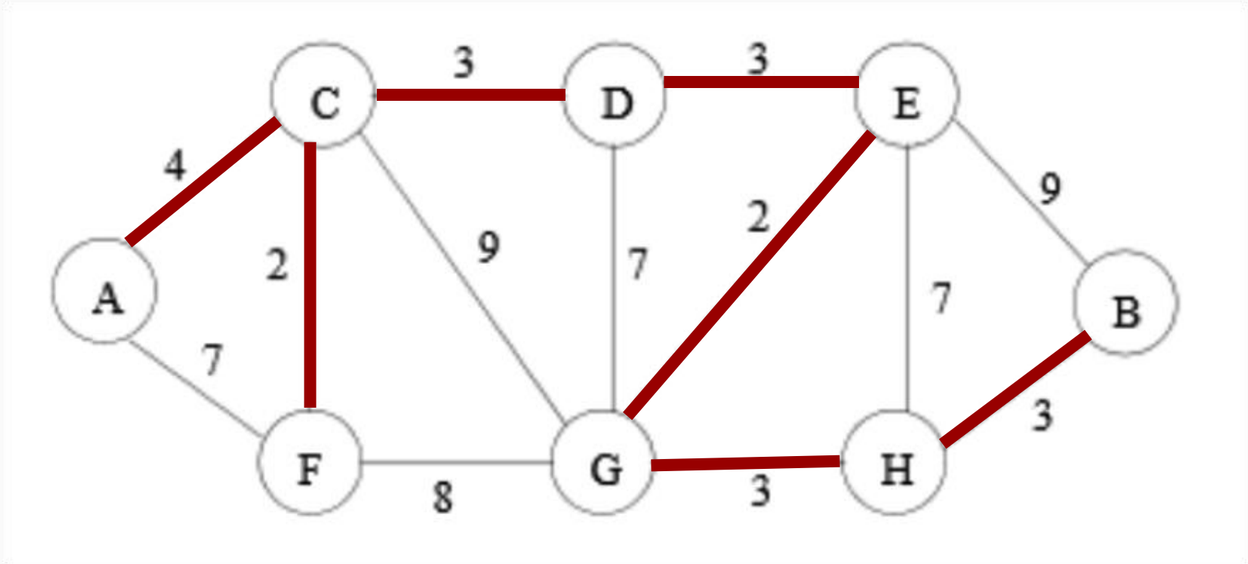
Actual	Distancia desde A							
	A	B	C	D	E	F	G	H
A	0	INF	INF	INF	INF	INF	INF	INF

Dijkstra (A)



Actual	Distancia desde A							
	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**
A	0	INF	4	INF	INF	7	INF	INF
C	0	INF	4	7	INF	6	13	INF
F	0	INF	4	7	INF	6	13	INF
D	0	INF	4	7	10	6	13	INF
E	0	19	4	7	10	6	12	17
G	0	19	4	7	10	6	12	15
H	0	18	4	7	10	6	12	15
B	0	18	4	7	10	6	12	15

Dijkstra (A)



Actual	Distancia desde A							
	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**
A	0	INF	4	INF	INF	7	INF	INF
C	0	INF	4	7	INF	6	13	INF
F	0	INF	4	7	INF	6	13	INF
D	0	INF	4	7	10	6	13	INF
E	0	19	4	7	10	6	12	17
G	0	19	4	7	10	6	12	15
H	0	18	4	7	10	6	12	15
B	0	18	4	7	10	6	12	15

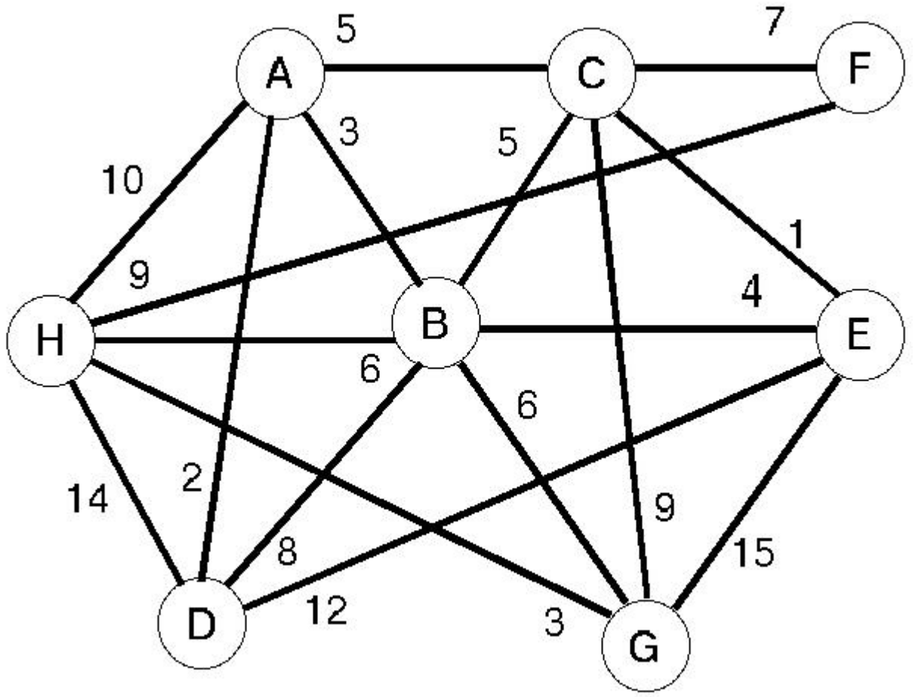
Dijkstra (Algoritmo)

```
function Dijkstra(Graph G(V, E), u)
{
    for (i = 0; i < V.length; i++){
        distances[i] = INF;
        visited[i] = false;
    }
    distances[u] = 0;
    while(anyFalse(visited))
    {
        V[i] = minDistance(distances, visited);
        visited[i] = true;
        foreach(V[j] adjacent to V[i])
            if(distances[i] + G.weight(i, j) < distances[j])
                distances[j] = distances[i] + G.weight(i, j);
    }
}
```

Dijkstra (Algoritmo, guardando padre)

```
function Dijkstra(Graph G(V, E), u)
{
    for (i = 0; i < V.length; i++) {
        distances[i] = INF;
        visited[i] = false;
    }
    distances[u] = 0;
    while(anyFalse(visited))
    {
        V[i] =  minDistance(distances, visited);
        visited[i] = true;
        foreach(V[j] adjacent to V[i])
            if(distances[i] + G.weight(i, j) < distances[j]){
                distances[j] = distances[i] + G.weight(i, j);
                parent[j] = i;
             }
    }
}
```

Dijkstra (Ejercicio, H)



Visitado

Distancia desde H

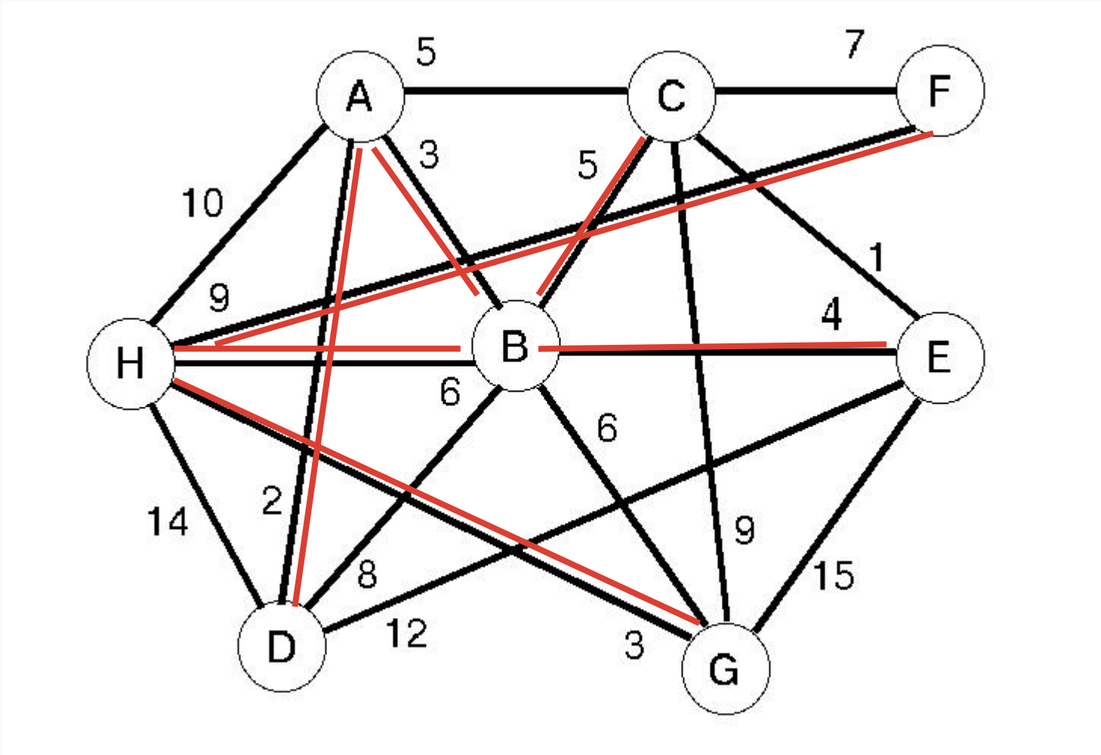
	A	B	C	D	E	F	G	H

Visitados:

P(A)=
P(D)=
P(B)=
P(C)=

P(F)=
P(E)=
P(G)=

Dijkstra (Ejercicio, H)



Actual

Distancia desde H

	A	B	C	D	E	F	G	H
D	9	6	11	11	10	9	3	0

Visitados:

1	1	1	1	1	1	1	1

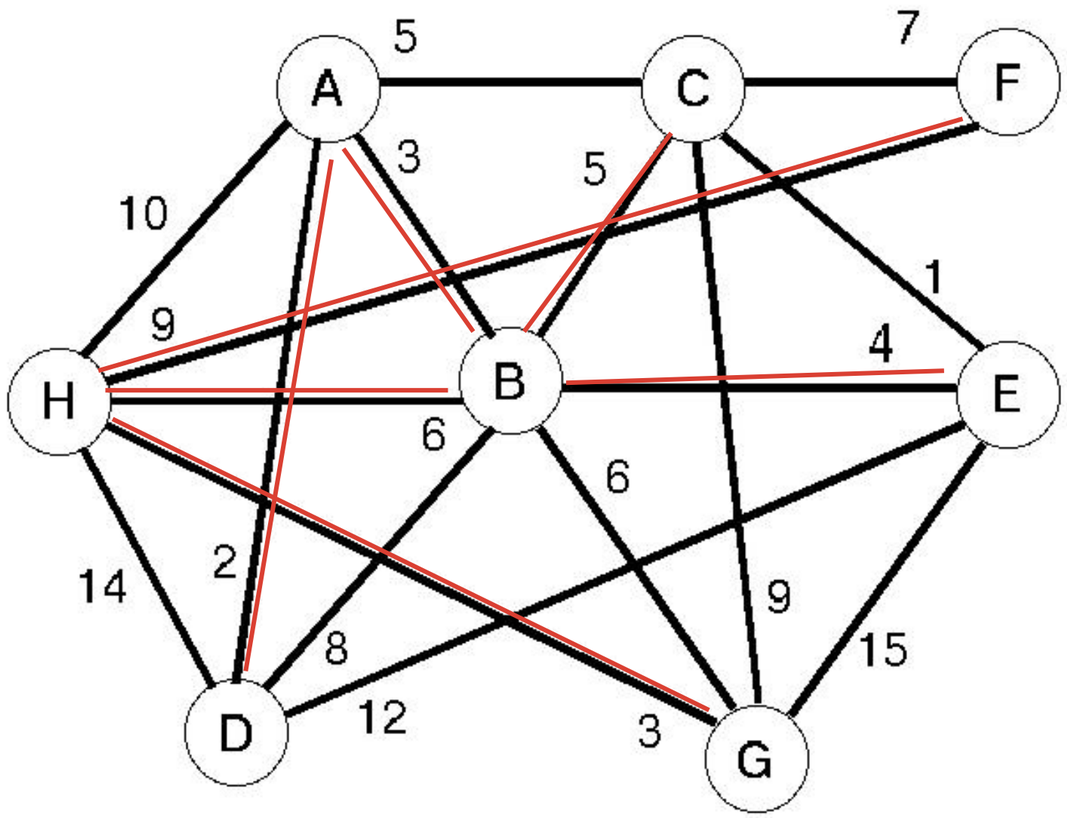
P(A)=B
P(D)=A
P(B)=H
P(C)=B

P(F)=H
P(E)=B
P(G)=H

Dijkstra (Ejercicio, H)

P(A)= B
P(D)= A
P(B)= H
P(C)= B

P(F)= H
P(E)= B
P(G)= H



Actual	Distancia desde H							
	A	B	C	D	E	F	G	H
C	9	6	11	11	10	9	3	0

Visitados: H, G, B, A, F, E, C, D

Búsqueda codiciosa (Greedy BFS)

Dijkstra nos puede servir para encontrar el camino más corto entre dos vértices, el problema está en que desperdicia tiempo explorando vértices que pueden no interesarnos

Best First Search (BFS) son considerados los algoritmos de búsqueda que se expandan a los estados más prometedores

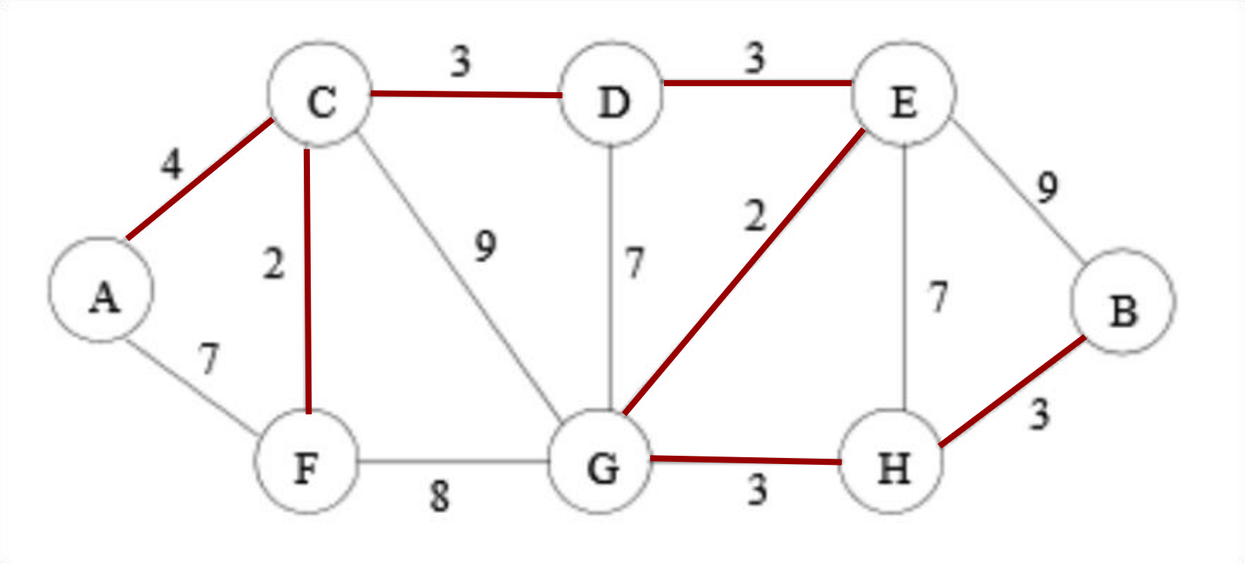
Estos estados prometedores se suelen escoger a través de una regla (heurística).

Típicamente se implementa utilizando una cola de prioridad.

https://www.youtube.com/watch?v=A8pmud1Uh0Q

Búsqueda codiciosa (Greedy BFS)

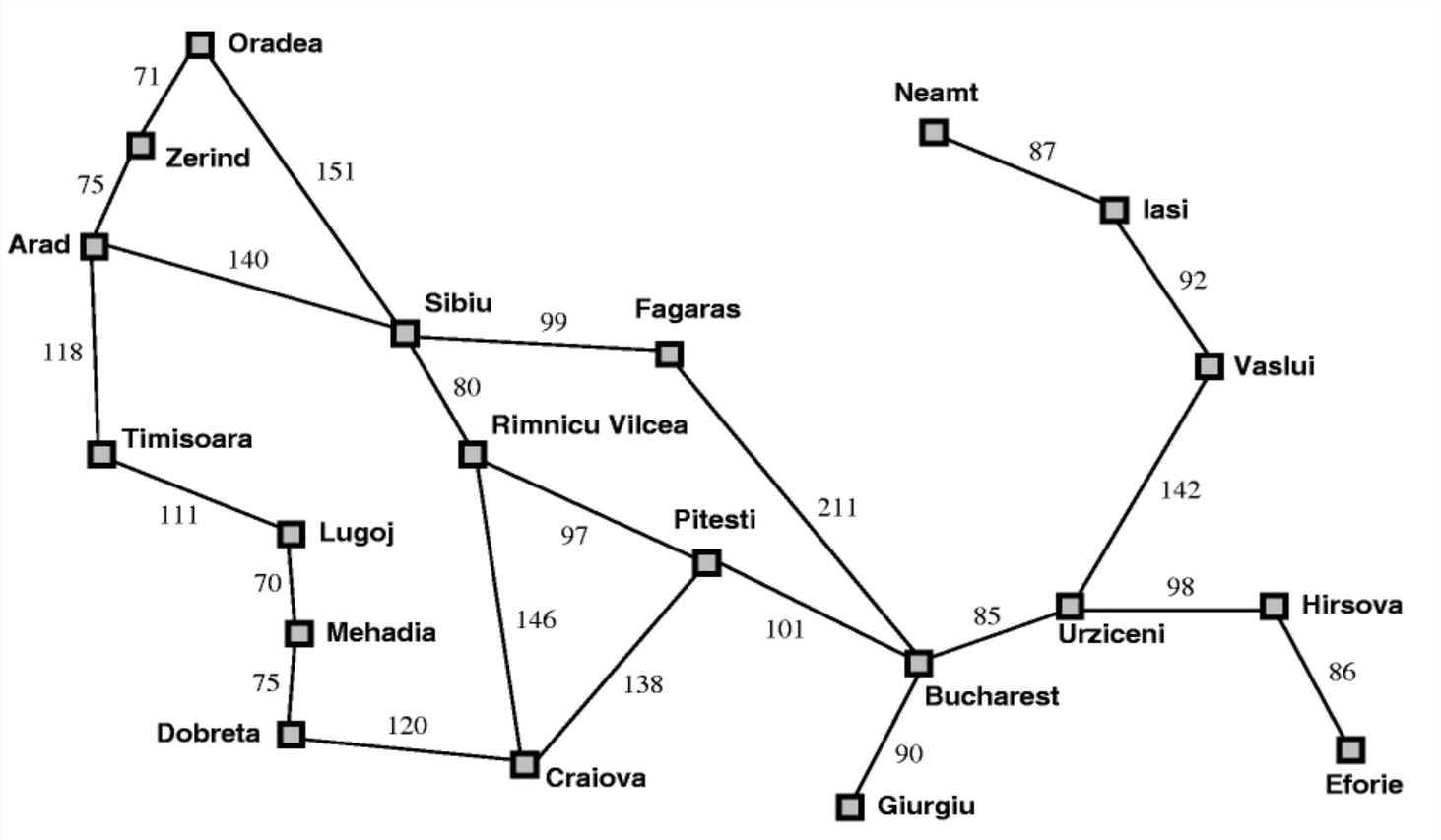
Veamos el siguiente ejemplo de A a B usando los pesos de las aristas:



Greedy BFS realiza búsquedas en caminos prometedores, pero puede ser que no nos retorne el camino más corto.

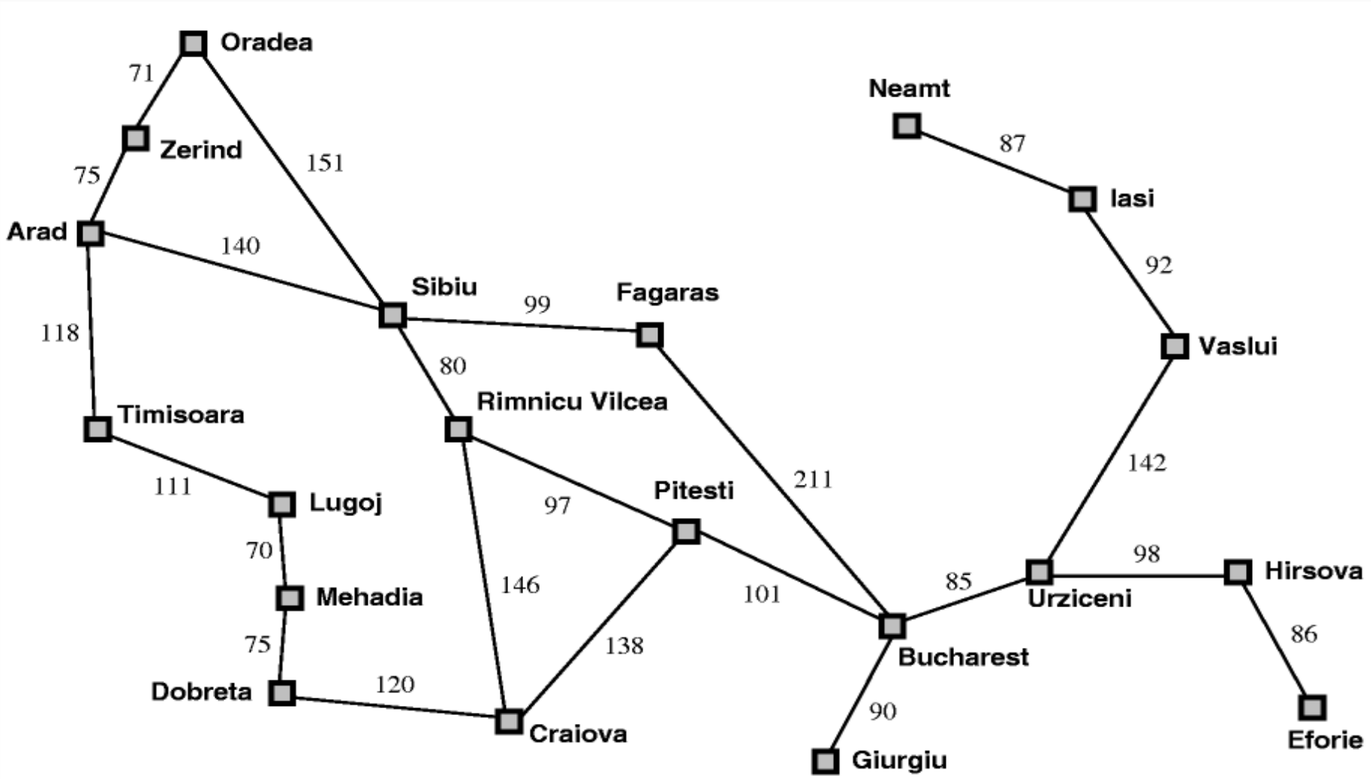
Búsqueda codiciosa (Greedy BFS)

Veamos el siguiente ejemplo para hallar el mejor camino entre Arad y Bucharest:



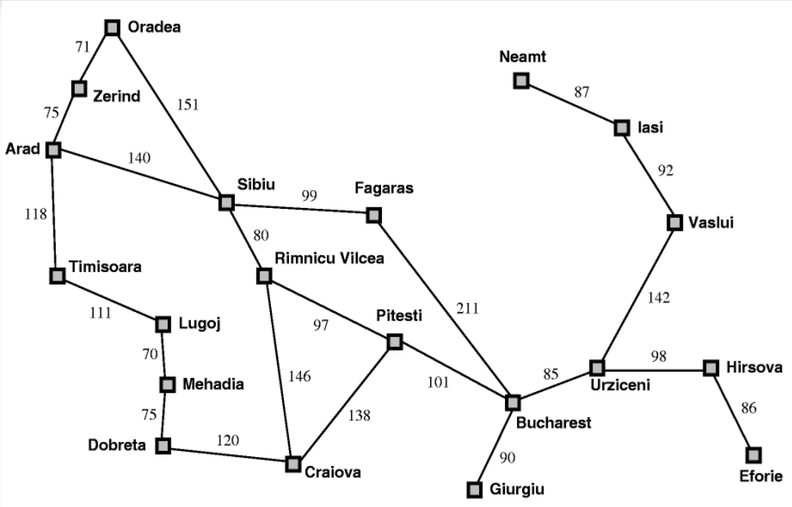
Búsqueda codiciosa (Greedy BFS)

Veamos el siguiente ejemplo para hallar el mejor camino entre Arad y Bucharest **usando una regla heurística:**



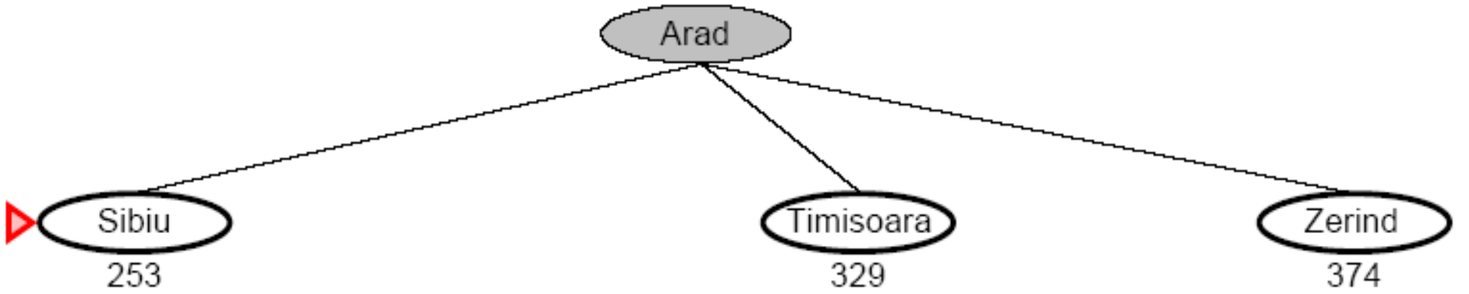
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Búsqueda codiciosa (Greedy BFS)

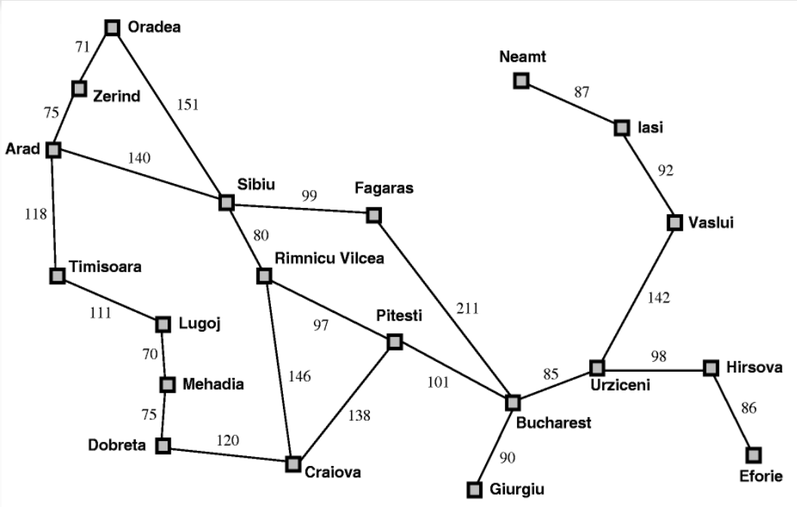


Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Desde Arad a Bucharest **usando regla heurística:**

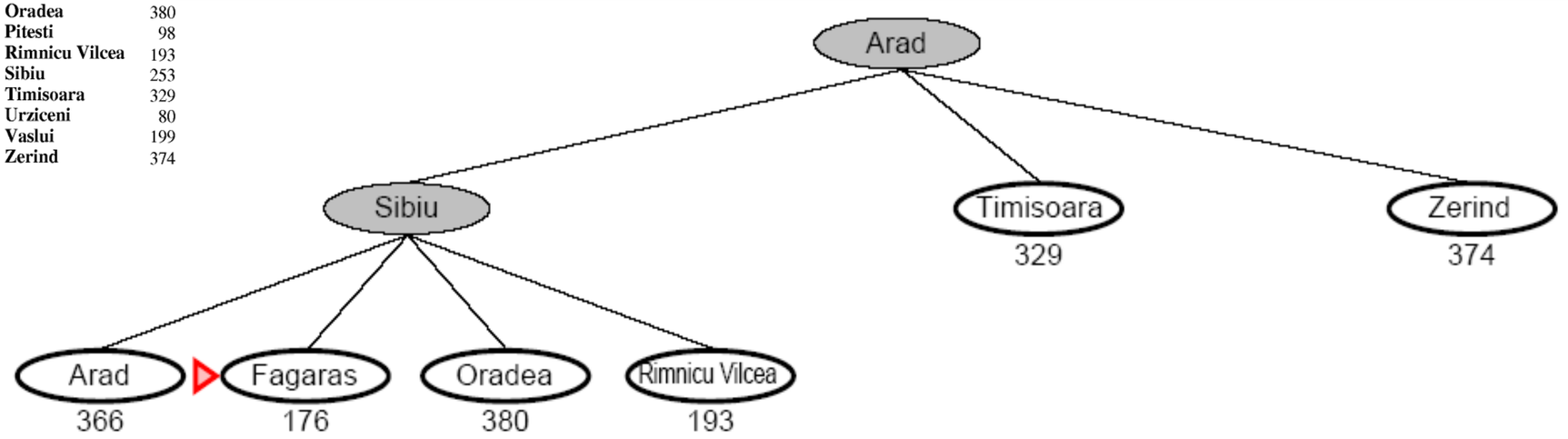


Búsqueda codiciosa (Greedy BFS)

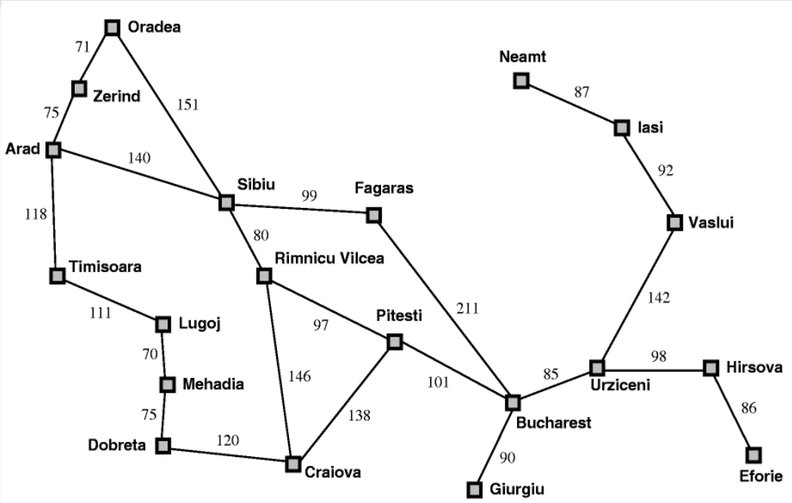


Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Desde Arad a Bucharest **usando regla heurística:**

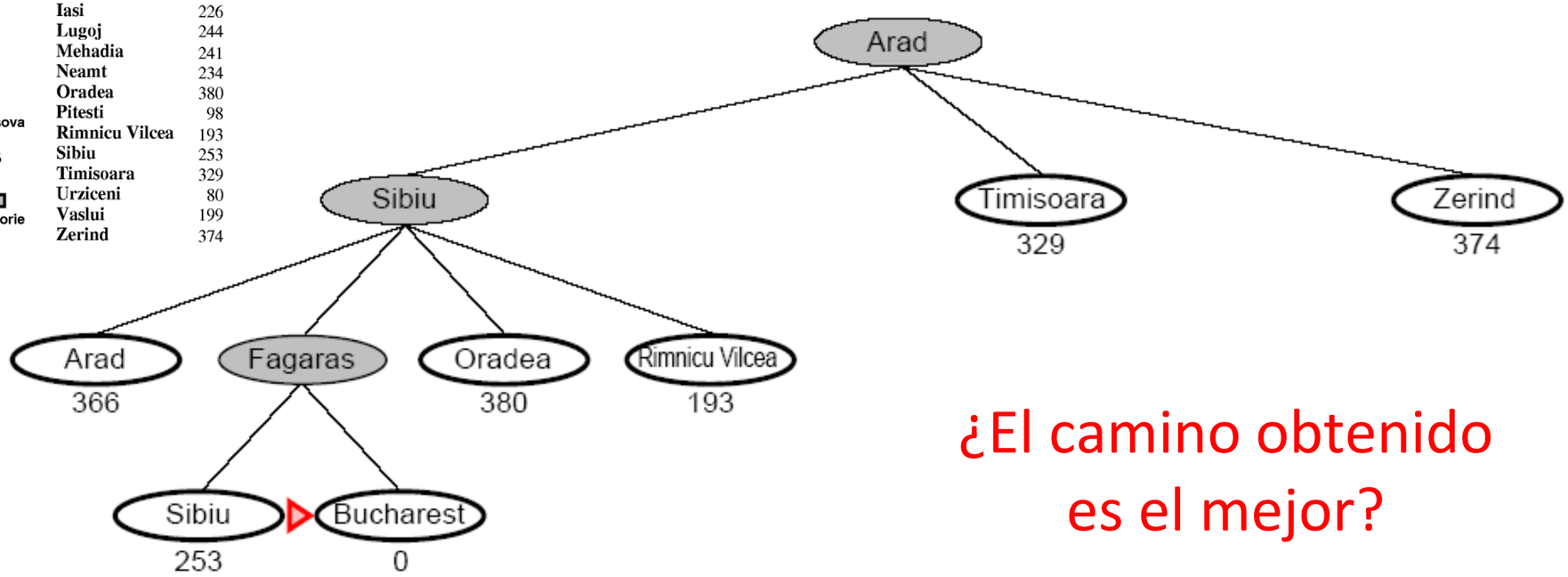


Búsqueda codiciosa (Greedy BFS)



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Desde Arad a Bucharest **usando regla heurística:**



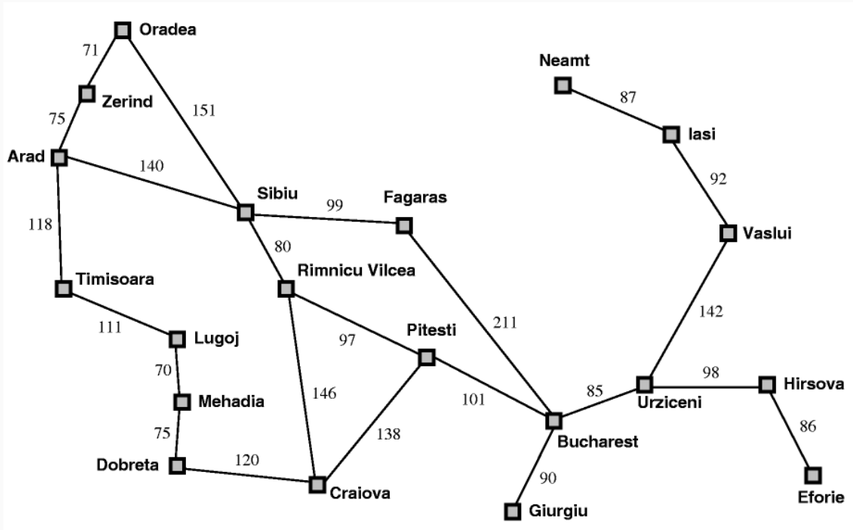
¿El camino obtenido es el mejor?

Algoritmo A*

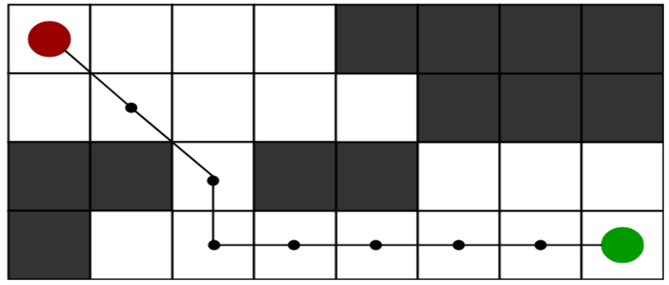
A* utiliza la distancia actual y una distancia estimada (heurística).

En caso de que exista una solución, este algoritmo la encontrará. Su complejidad va a depender de su heurística.

Es uno de los algoritmos más utilizados en distintas aplicaciones como videojuegos.



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

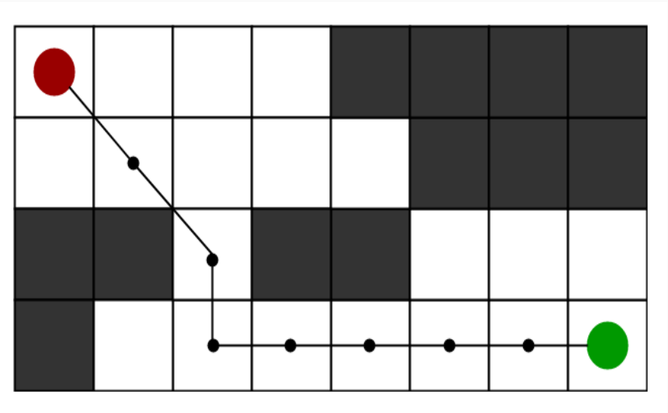


Algoritmo A*

Es un algoritmo que se extiende dependiendo del costo, ese costo es calculado utilizando una heurística

f(n)= g(n) + h(n)

g(n) es la distancia desde el origen al vértice *n*, y h(n) es la distancia estimada hacia el destino (heurística)

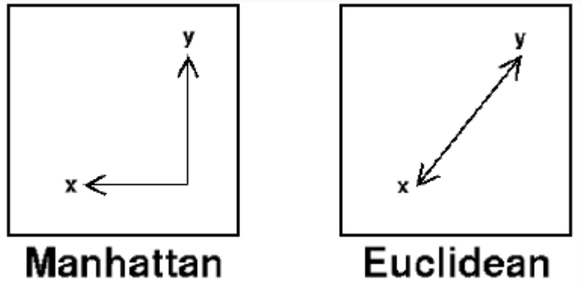


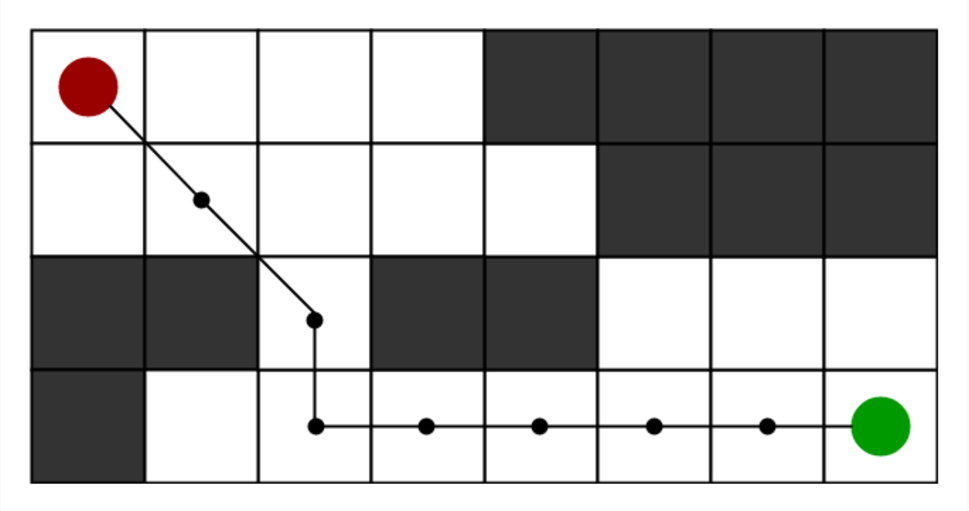
Algoritmo A*

Es un Dijkstra modificado que solo busca llegar a un destino

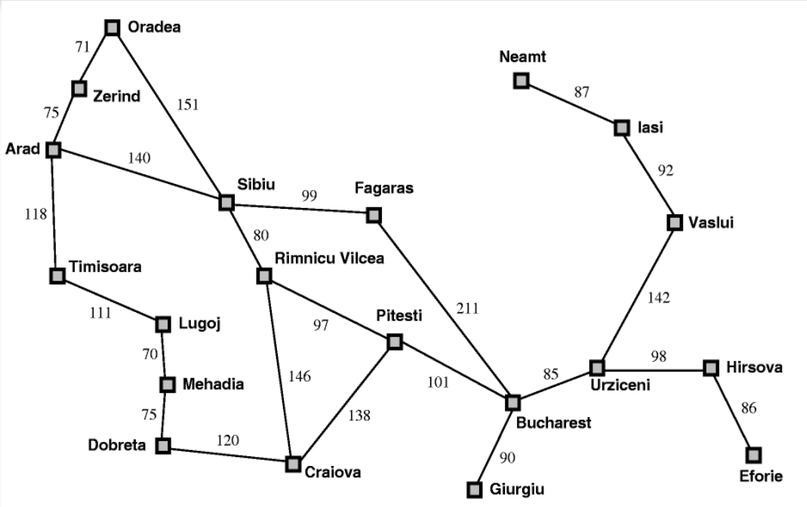
Siempre se busca evitar expandirse a vértices (estados) que suponemos son caros

La heurística podría ser por ejemplo la distancia de manhattan o la euclidiana



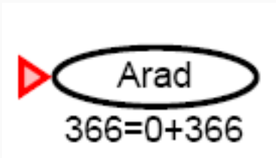


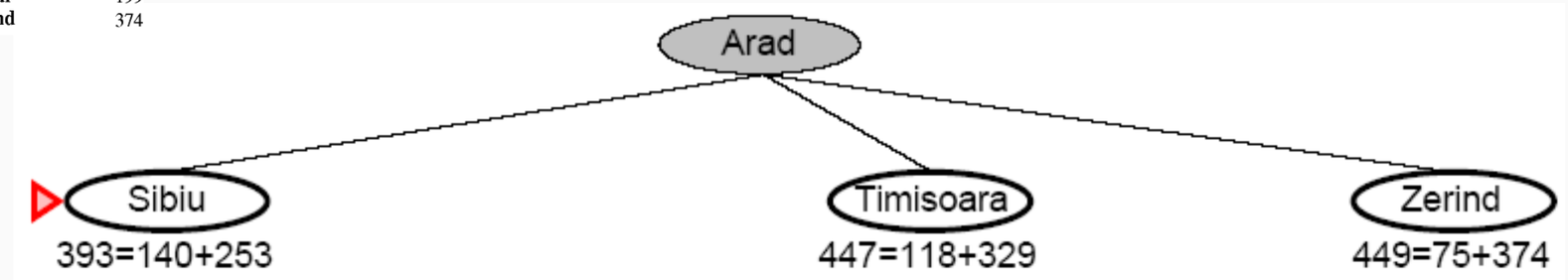
Algoritmo A* (Ejemplo 1)



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

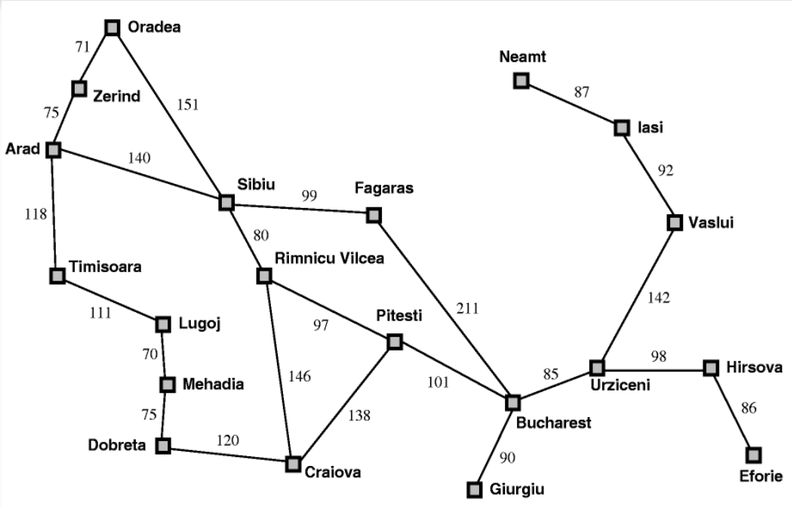
Desde Arad a Bucharest **usando regla heurística:**





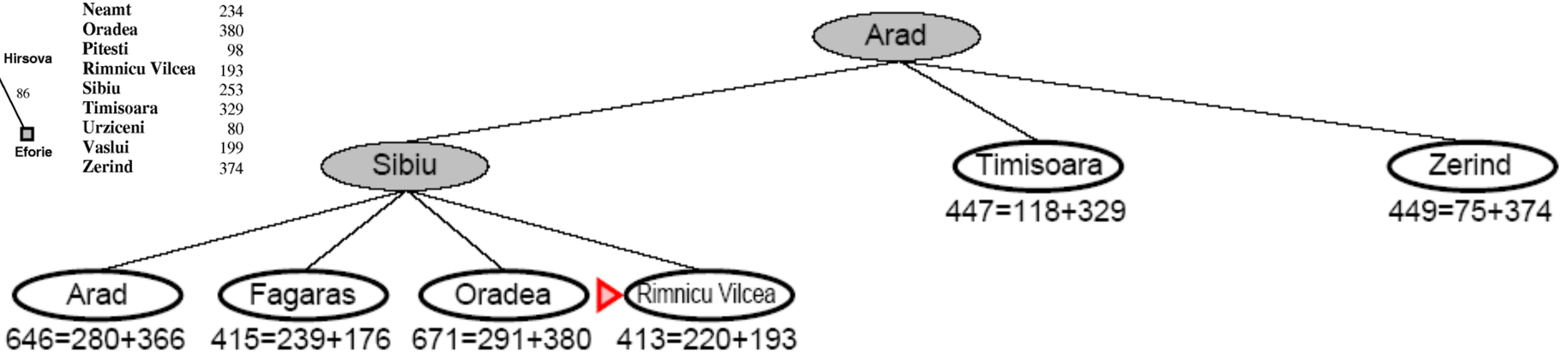
$$f(n) = g(n) + h(n)$$

Algoritmo A* (Ejemplo 1)



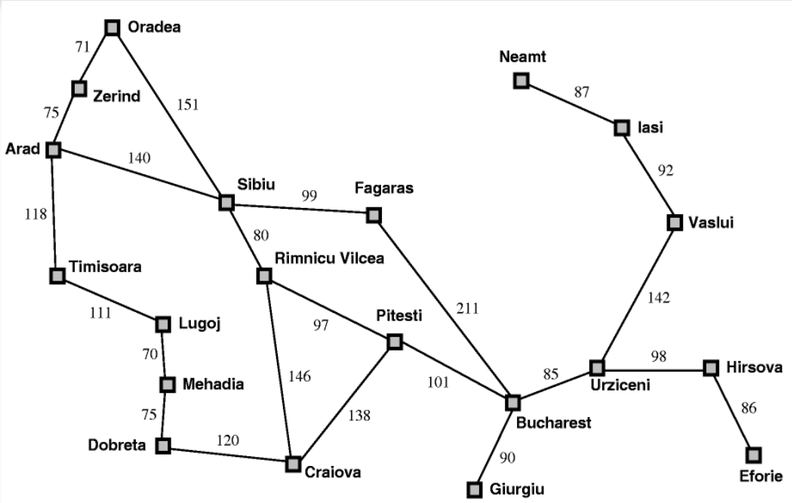
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Desde Arad a Bucharest **usando regla heurística:**



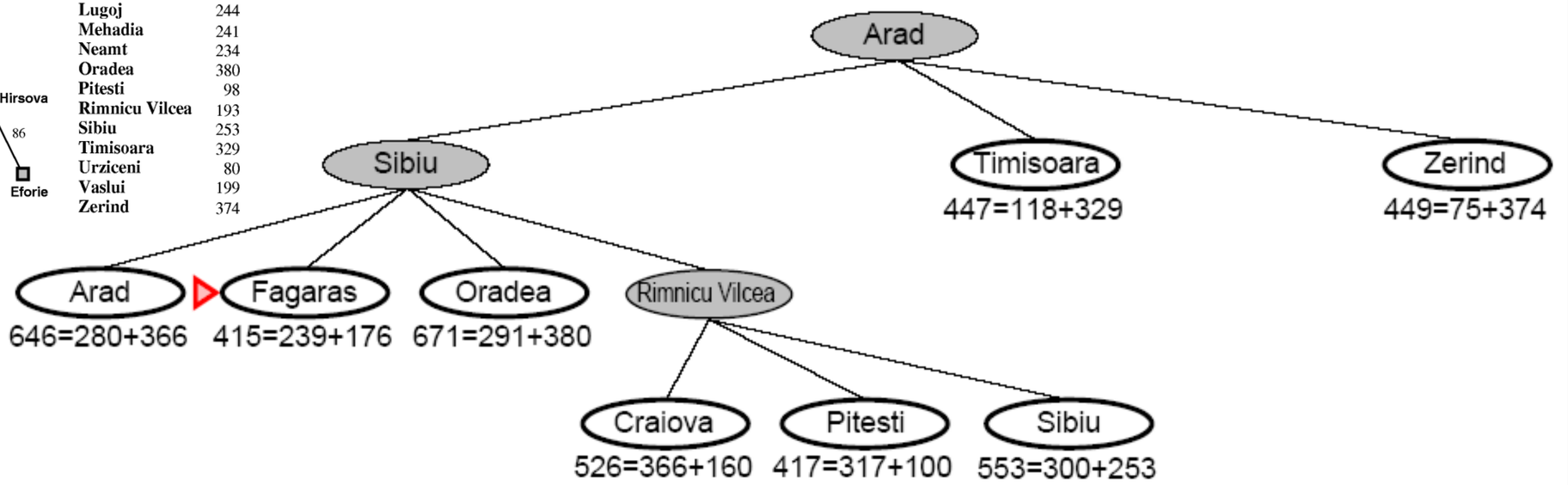
$$f(n) = g(n) + h(n)$$

Algoritmo A* (Ejemplo 1)



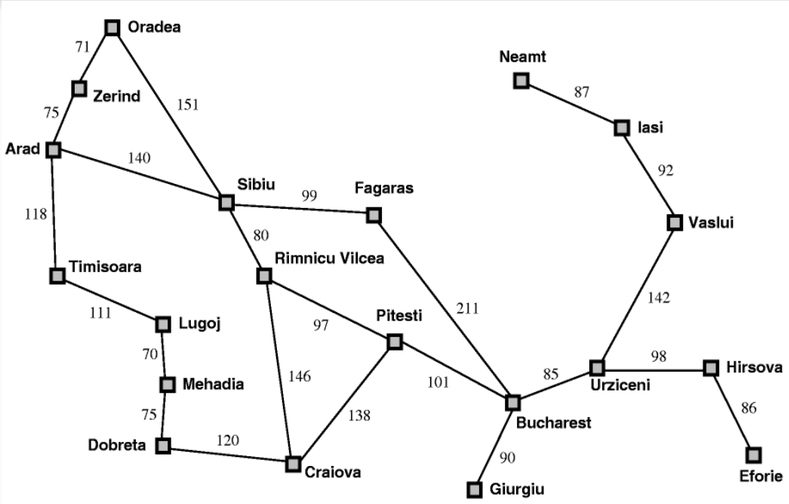
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Desde Arad a Bucharest **usando regla heurística:**



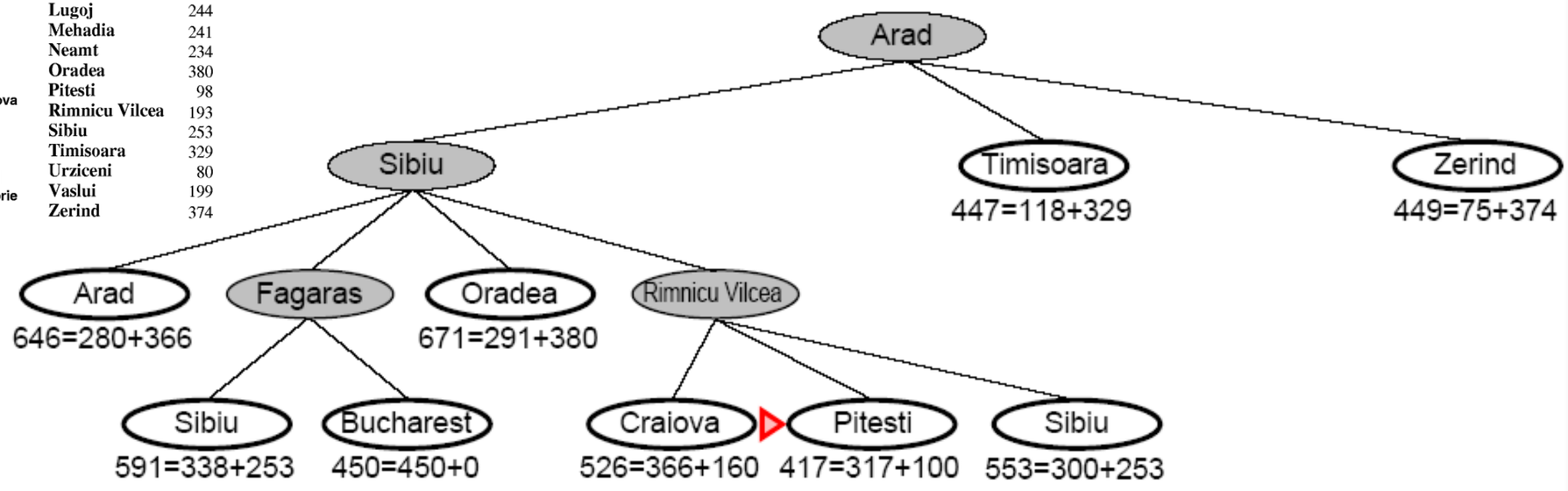
$$f(n) = g(n) + h(n)$$

Algoritmo A* (Ejemplo 1)



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

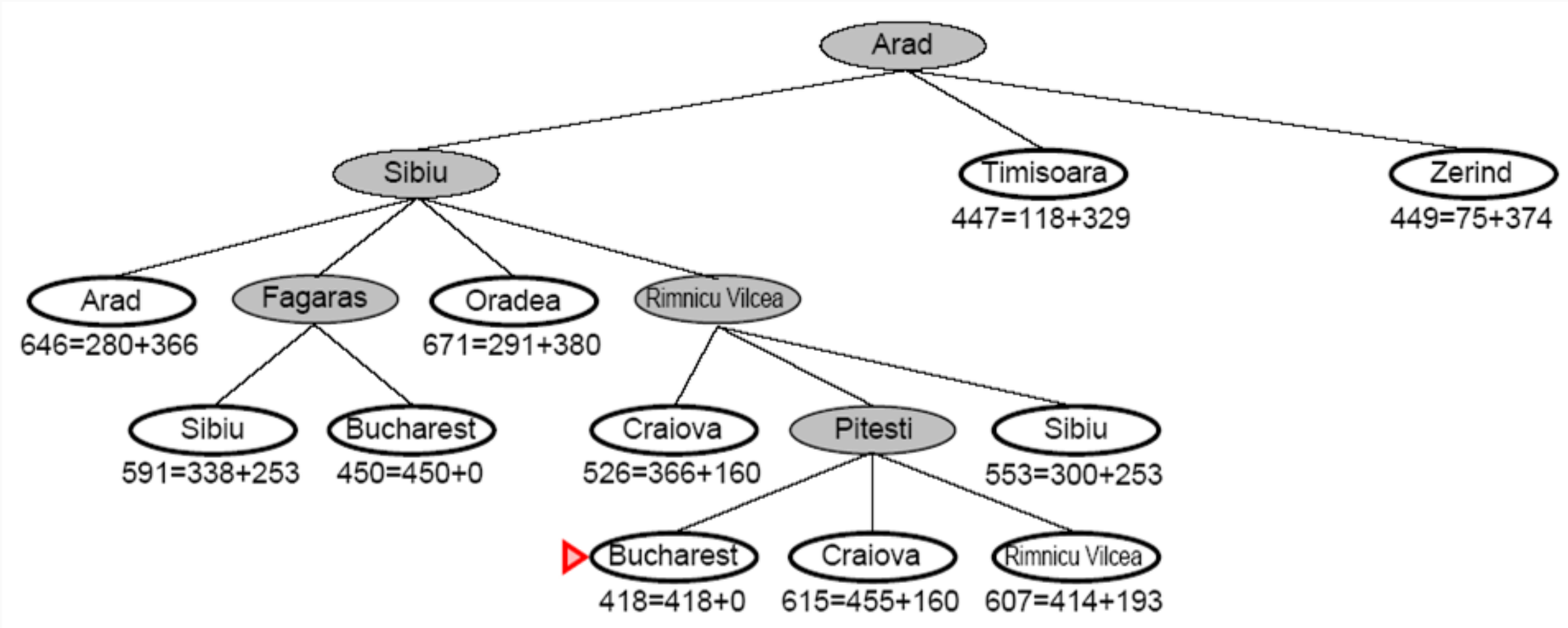
Desde Arad a Bucharest **usando regla heurística:**



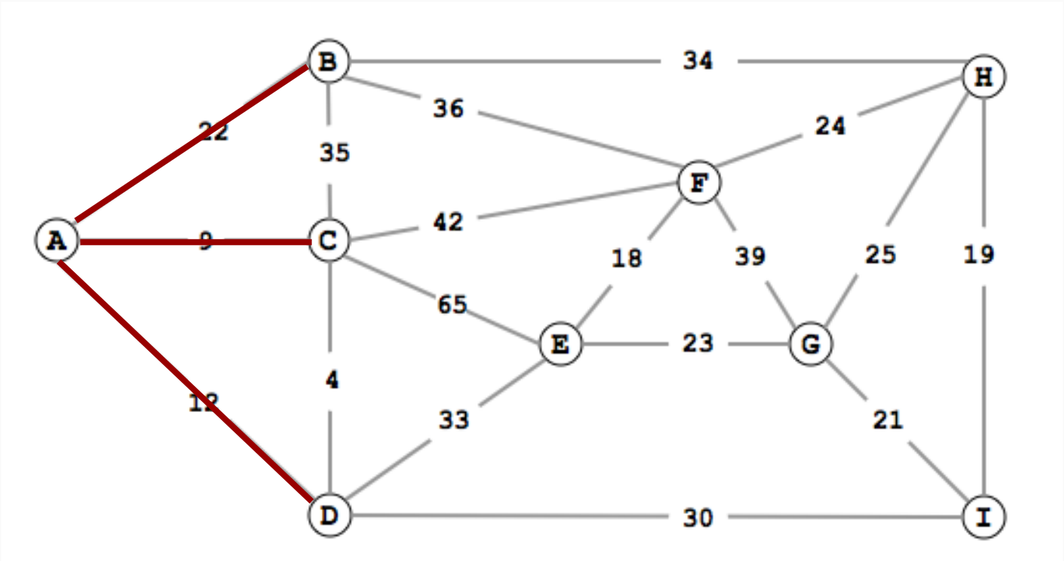
$$f(n) = g(n) + h(n)$$

Algoritmo A* (Ejemplo 1)

Desde Arad a Bucharest **usando regla heurística:**



Algoritmo A* (Ejemplo 2, de A a I)

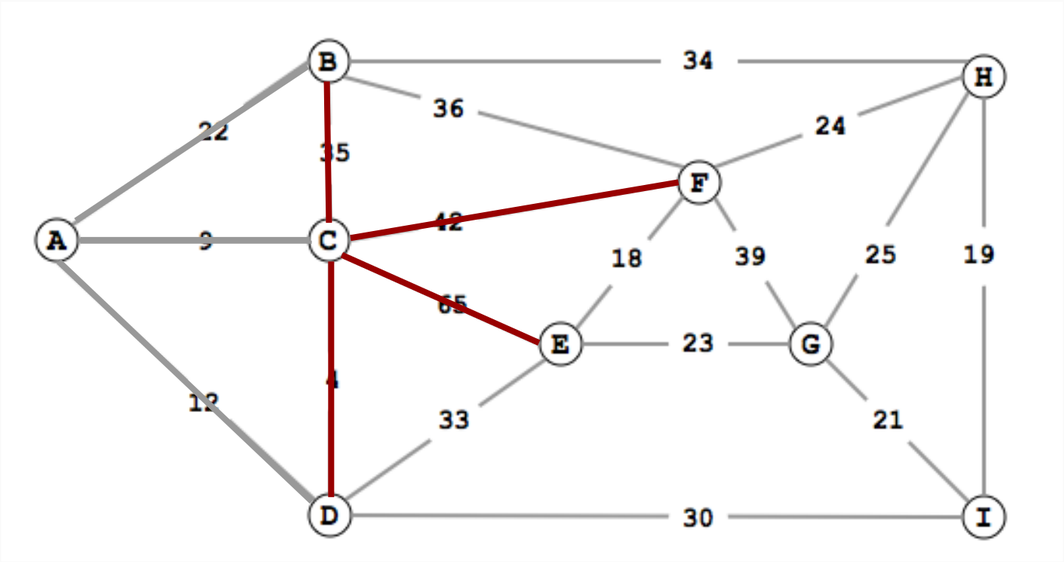


	A	B	C	D	E	F	G	H	I
H(n)	36	39	31	30	34	32	21	19	0

	G(n)	F(n)	Parent
A	0	36	
B	22	61	A
C	9	40	A
D	12	42	A

Heurística: Distancia euclideana

Algoritmo A* (Ejemplo 2, de A a I)

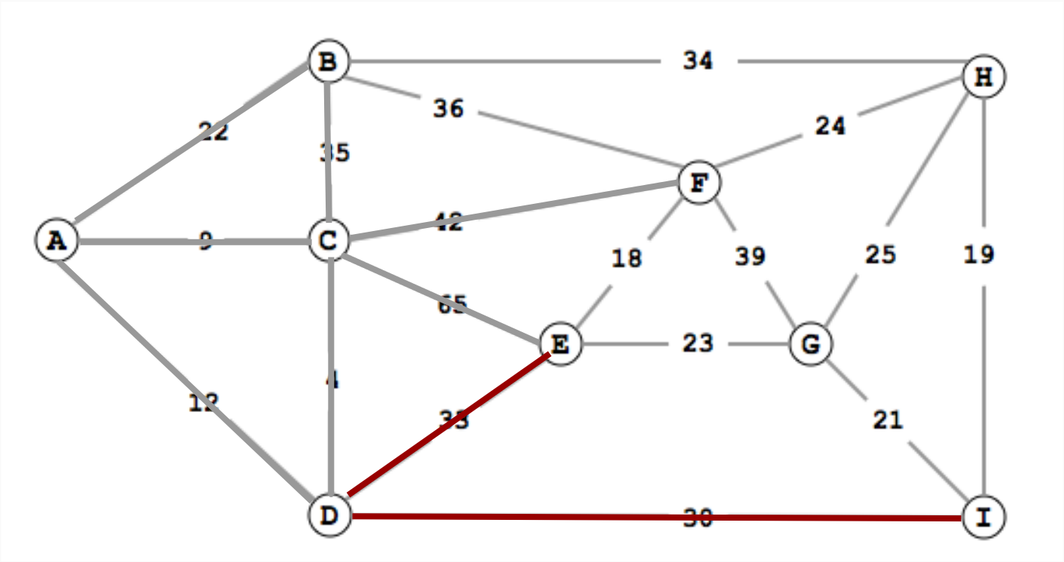


	A	B	C	D	E	F	G	H	I
H(n)	36	39	31	30	34	32	21	19	0

	G(n)	F(n)	Parent
A	0	36	
B	22	61	A
C	9	40	A
D	12	42	A
E	74	108	C
F	51	83	C

Heurística: Distancia euclideana

Algoritmo A* (Ejemplo 2, de A a I)

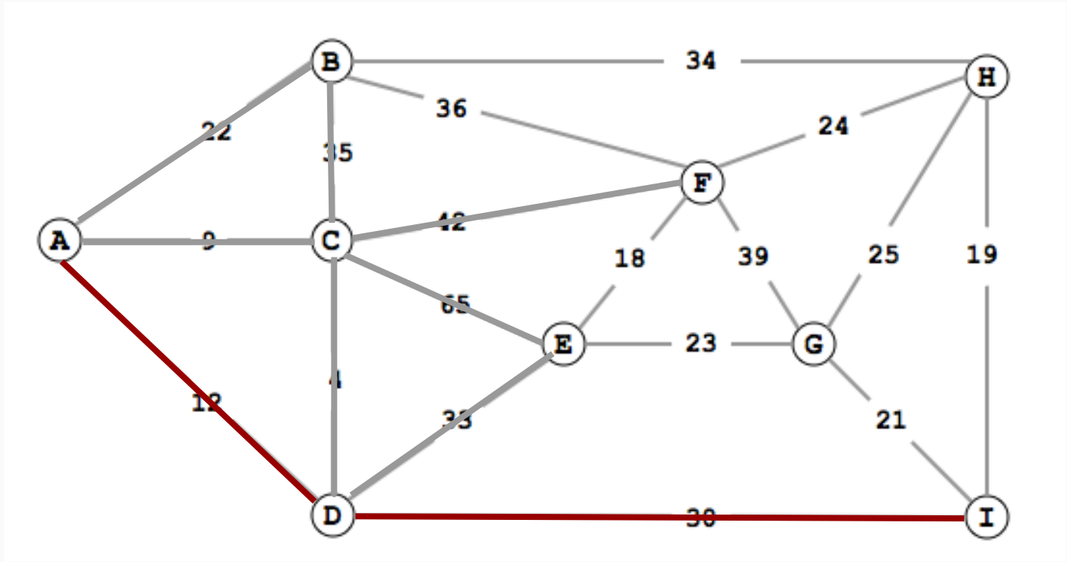


	A	B	C	D	E	F	G	H	I
H(n)	36	39	31	30	34	32	21	19	0

	G(n)	F(n)	Parent
A	0	36	
B	22	61	A
C	9	40	A
D	12	42	A
E	45	79	D
F	51	83	C
I	42	42	D

Heurística: Distancia euclideana

Algoritmo A* (Ejemplo 2, de A a I)



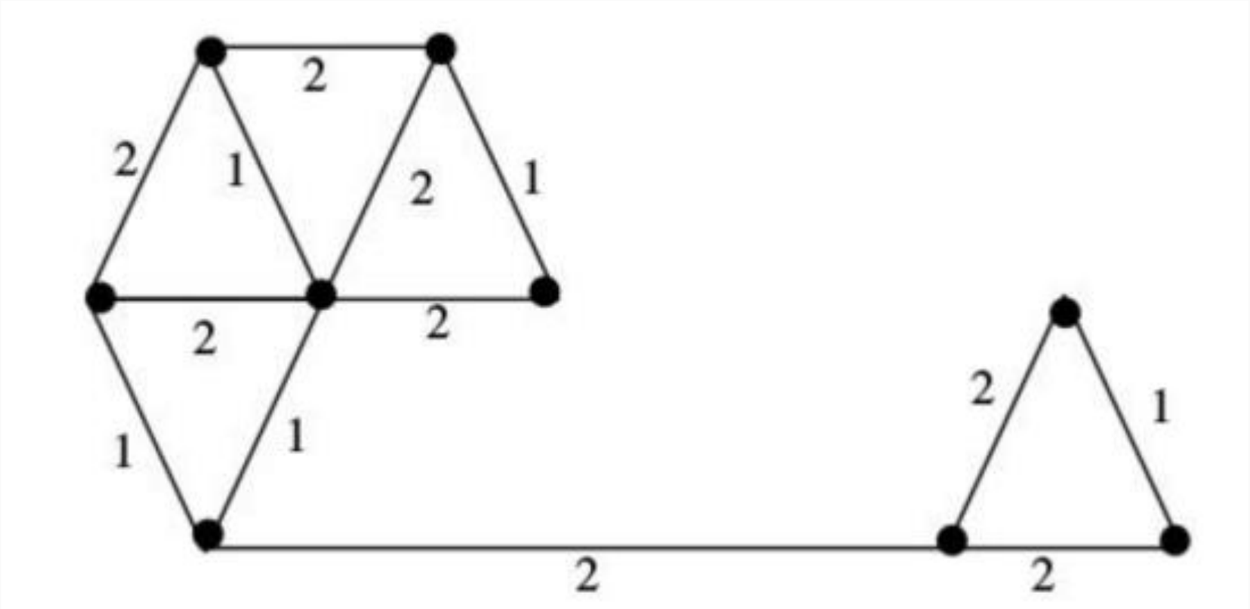
	A	B	C	D	E	F	G	H	I
H(n)	36	39	31	30	34	32	21	19	0

	G(n)	F(n)	Parent
A	0	36	
B	22	61	A
C	9	40	A
D	12	42	A
E	45	79	D
F	51	83	C
I	42	42	D

Heurística: Distancia euclideana

Ejercicios

¿Cual es el numero de arboles de expansión mínima distintos para el siguiente grafo?



Ejercicios

Dibujen y apliquen los algoritmos a los siguientes grafos:

BFS(A)

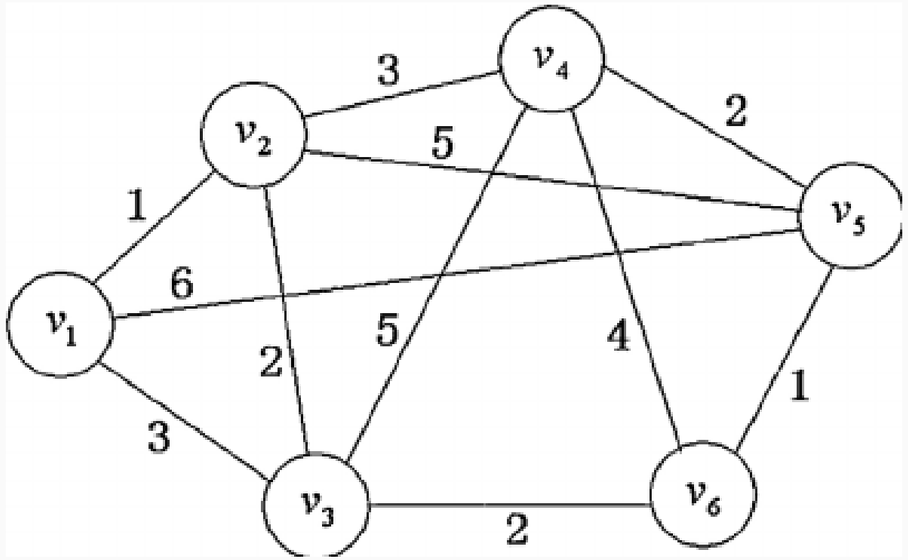
- V = {A, B, C, D, E, F}
- E = {(A, B), (A, E), (B, F), (C, A), (C, E), (E, B), (E, D), (F, E)}

DFS(F)

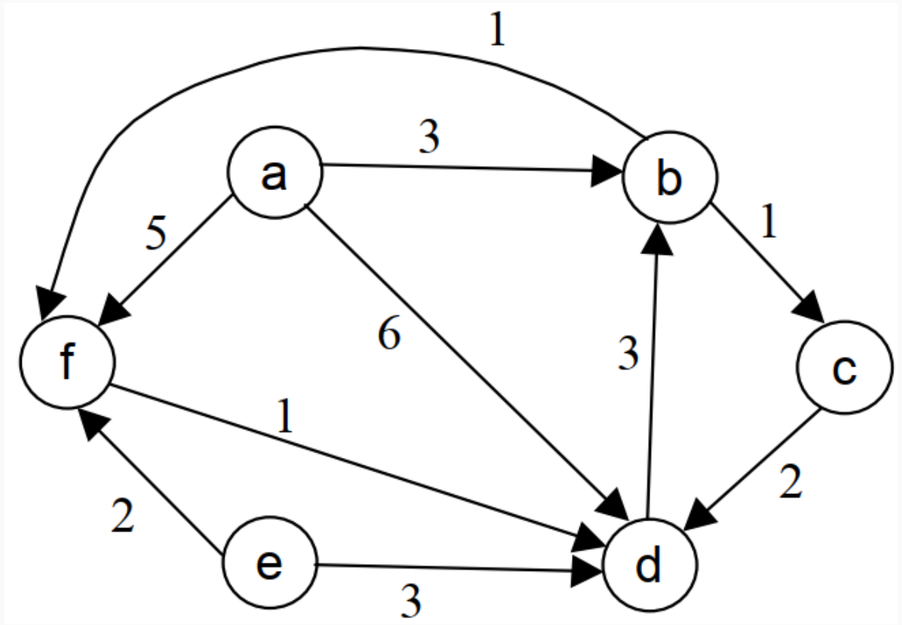
- V = {A, B, C, D, E, F}
- E = {{A, B}, {A, C}, {A, F}, {B, D}, {B, C}, {B, F}, {C, E}, {C, D}, {D, F}, {E, F}}

Ejercicios

Dijkstra (v1):

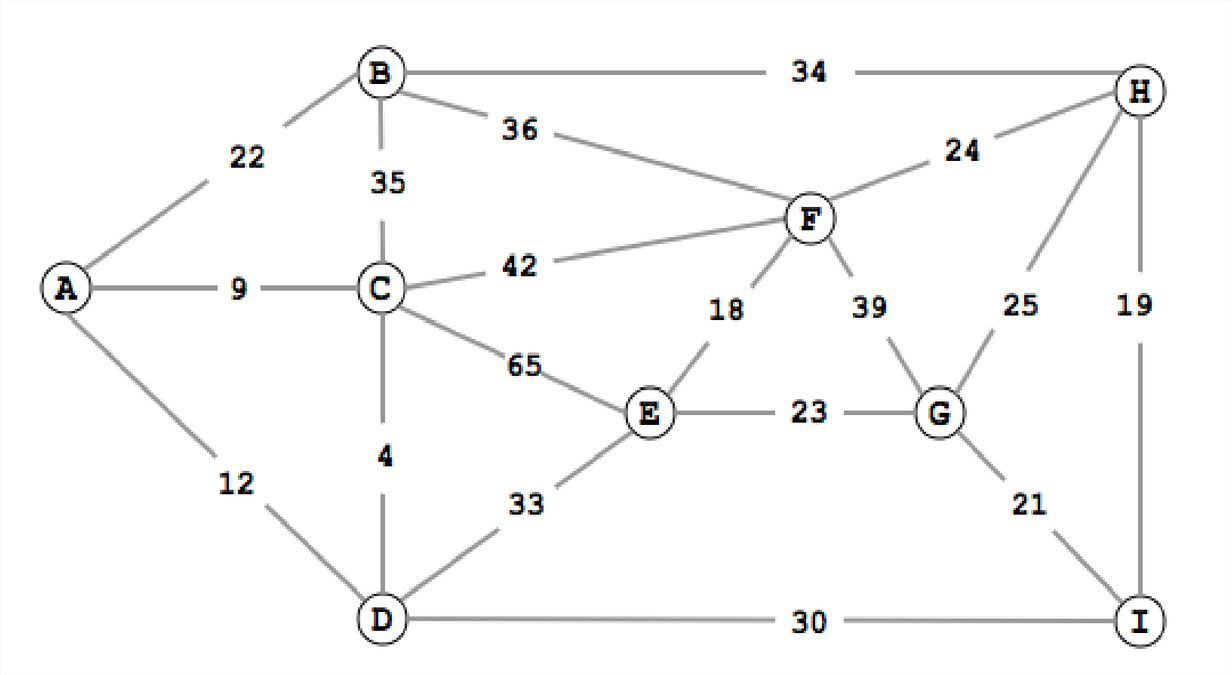


Dijkstra (a):



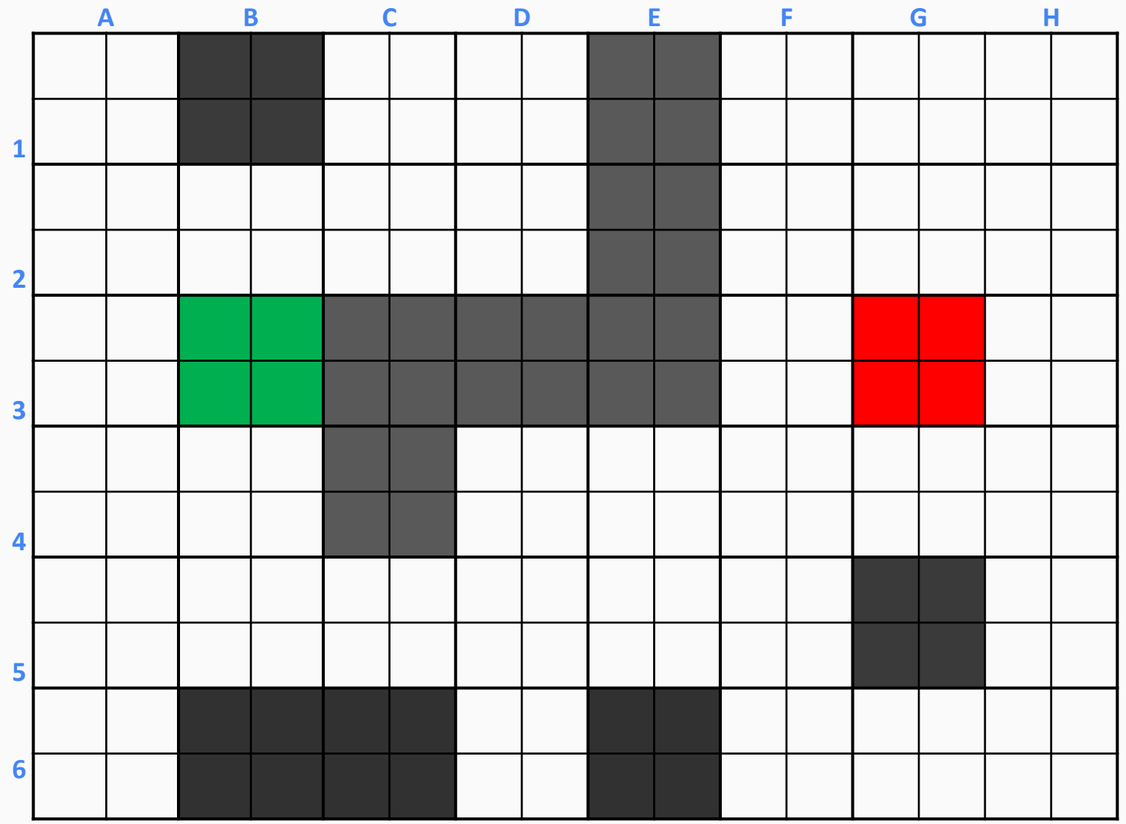
Ejercicios

A* (B a I):



Ejercicios

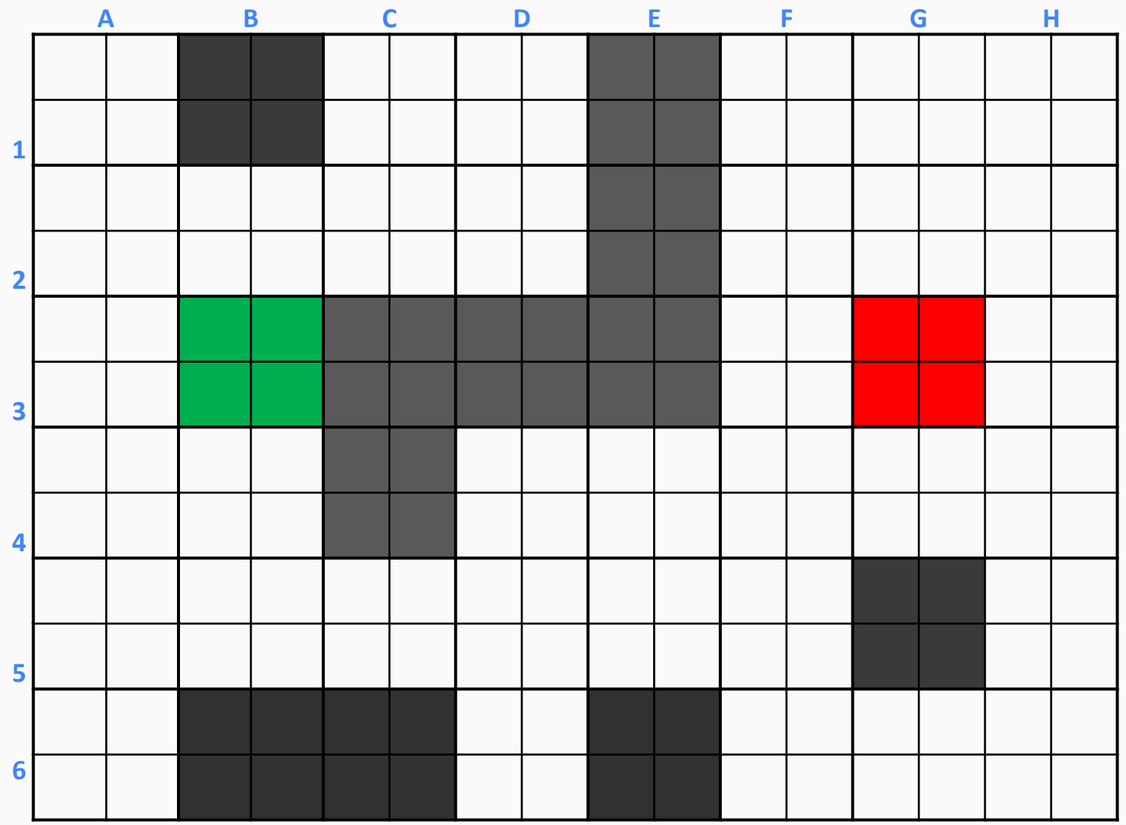
Greedy BFS (del verde al rojo):



usando como heurística la distancia Manhattan
Y luego la distancia Euclediana

Ejercicios

A* (del verde al rojo):



usando como heurística la distancia Manhattan
Y luego la distancia Euclediana

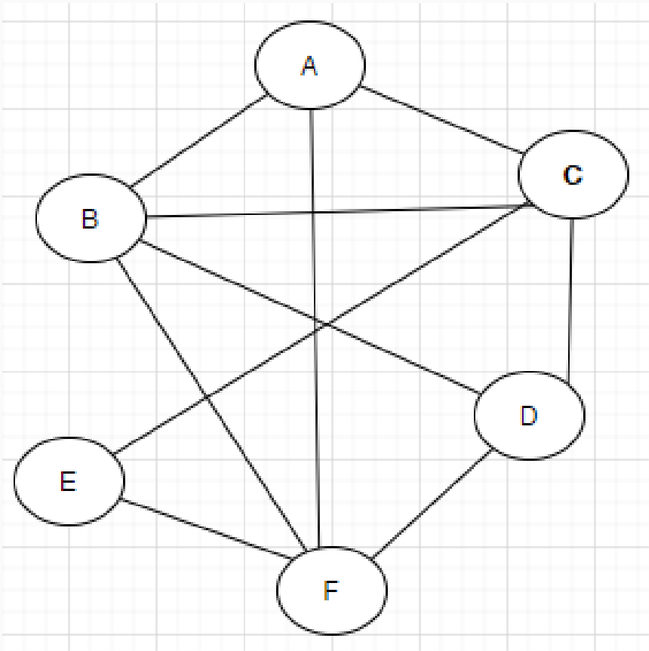
Solución

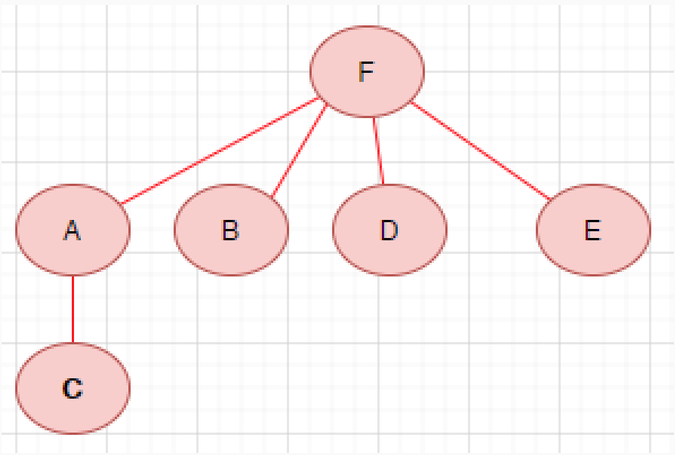
Solución

Dibujen y apliquen los algoritmos a los siguientes grafos:

BFS(F)

- V = {A, B, C, D, E, F}
- E = {{A, B}, {A, C}, {A, F}, {B, D}, {B, C}, {B, F}, {C, E}, {C, D}, {D, F}, {E, F}}



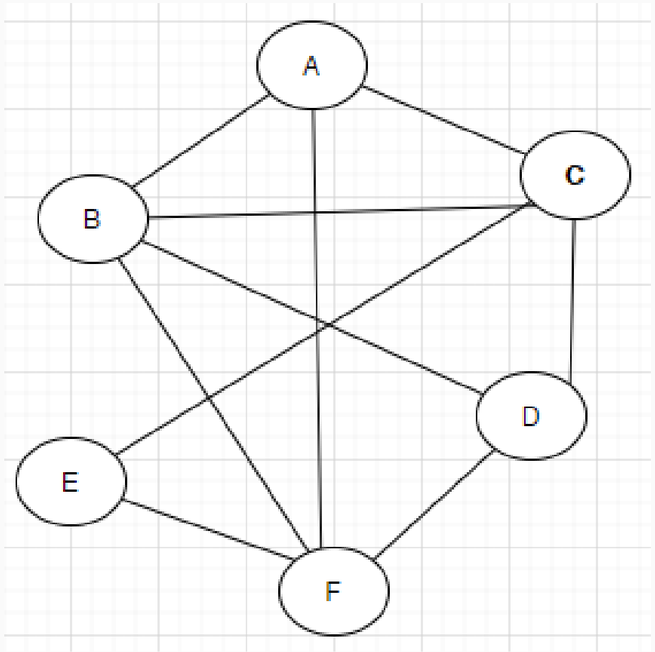


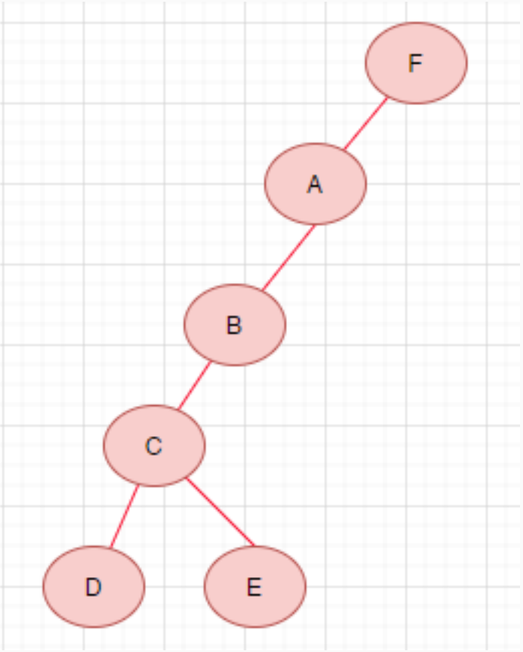
Solución

Dibujen y apliquen los algoritmos a los siguientes grafos:

DFS(F)

- V = {A, B, C, D, E, F}
- E = {{A, B}, {A, C}, {A, F}, {B, D}, {B, C}, {B, F}, {C, E}, {C, D}, {D, F}, {E, F}}



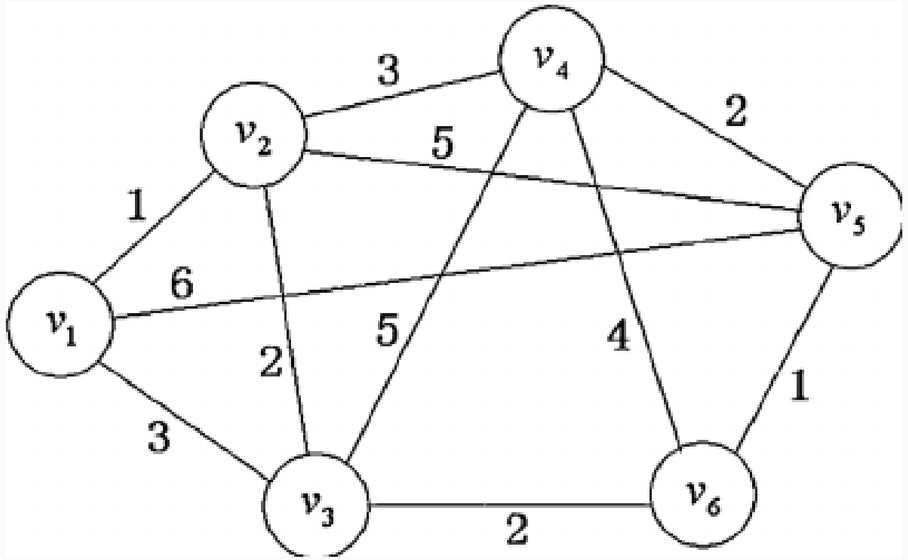


Solución

Padres:

P(v2)=v1
P(v5)=v1
P(v3)=v1
P(v4)=v2
P(v6)=v3

Dijkstra (v1):



Visitado | **Distancia desde v1**

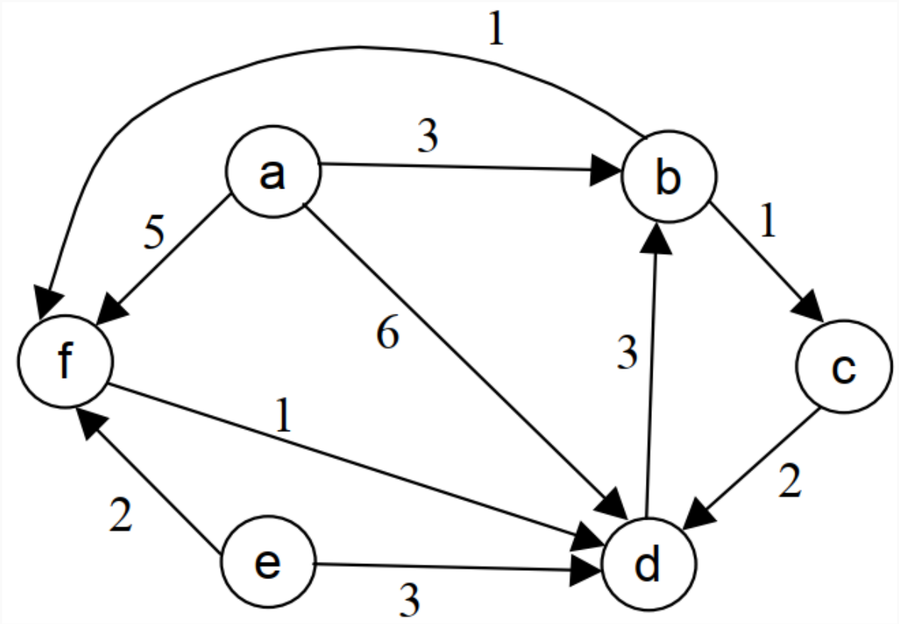
	V1	V2	V3	V4	V5	V6
v1	0	INF	INF	INF	INF	INF
v2	0	1	3	INF	6	INF
v3	0	1	3	4	6	INF
v4	0	1	3	4	6	5
v6	0	1	3	4	6	5

Solución

Padres:

P(B)=A
P(C)=B
P(F)=B
P(D)=F

Dijkstra (a):



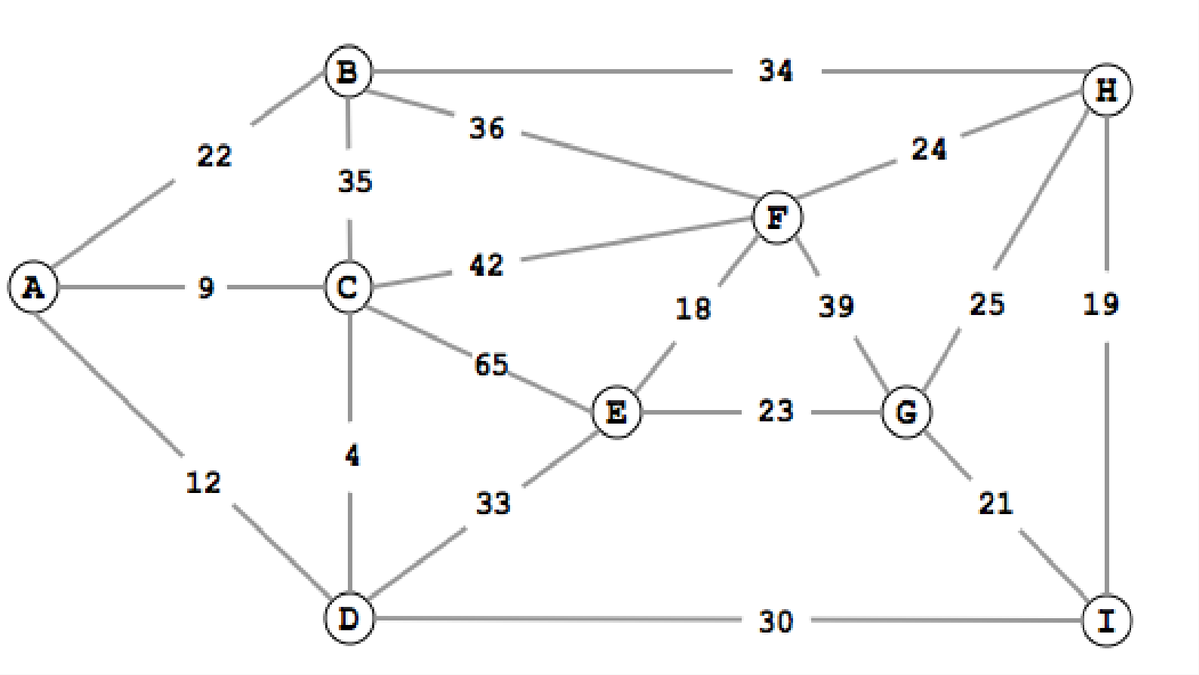
Visitado — **Distancia desde a**

	A	B	C	D	E	F
A	0	INF	INF	INF	INF	INF
B	0	3	INF	6	INF	5
C	0	3	4	6	INF	4
F	0	3	4	6	INF	4
D	0	3	4	5	INF	4
E	0	3	4	5	INF	4

Solución

$f(n) = g(n) + h(n)$

A* (B a I):

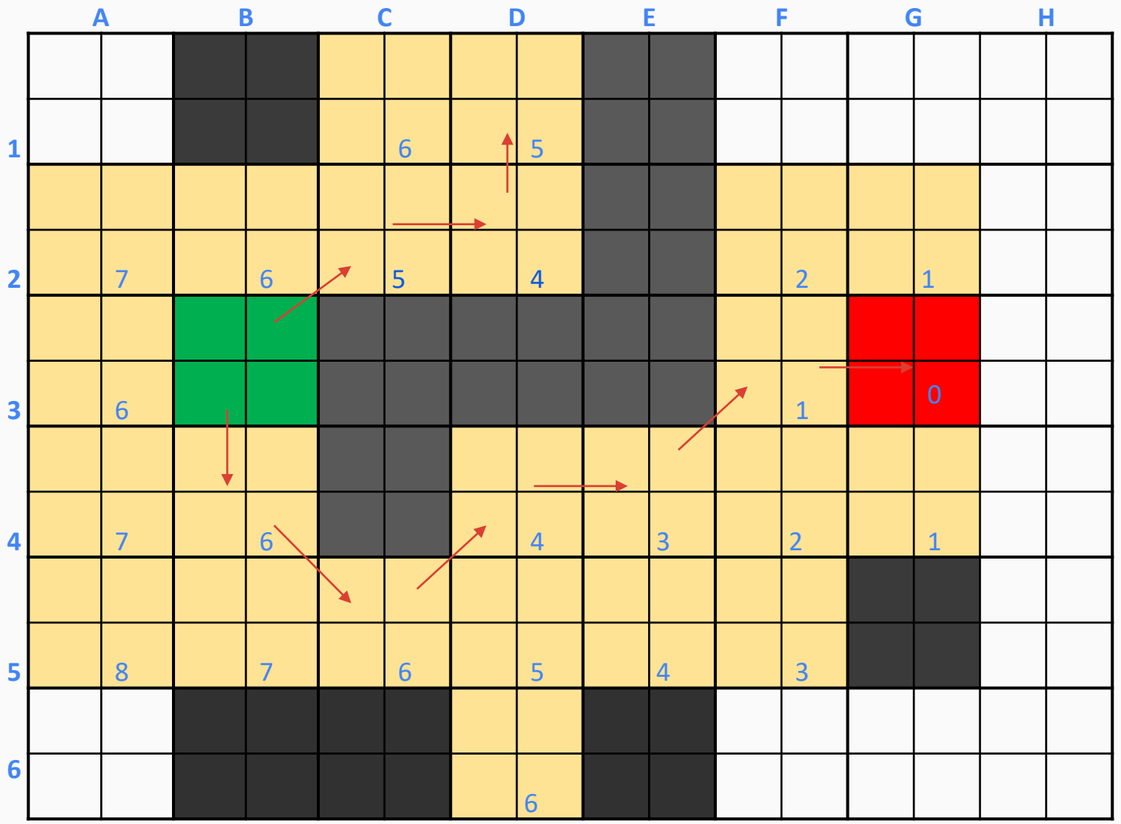


	A	B	C	D	E	F	G	H	I
H(n)	36	39	31	30	34	32	21	19	0

	G(n)	F(n)	Parent
B	0	39	
A	22	58	B
C	35	66	B
F	36	68	B
H	34	53	B
G	59	80	H
I	53	53	H

Solución

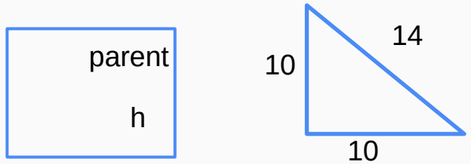
Greedy BFS (del verde al rojo):



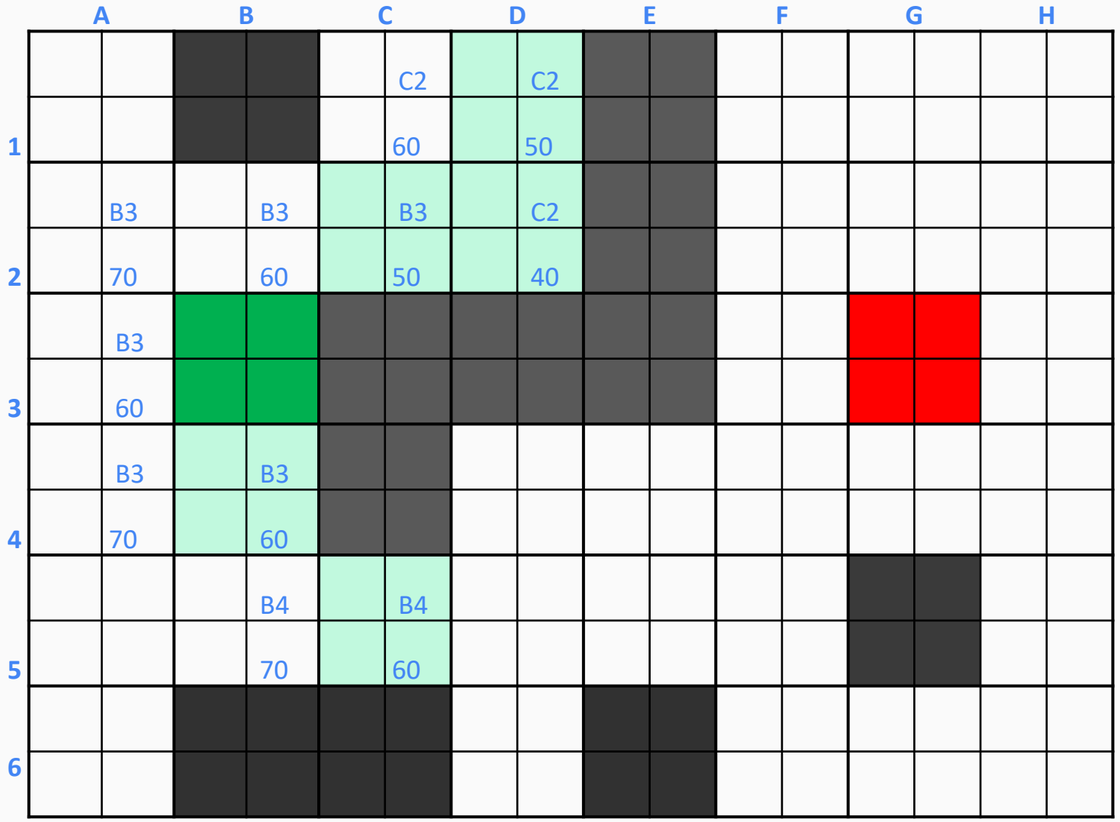
usando como heurística la distancia Manhattan
Y luego la distancia Euclediana

Solución

Greedy BFS (del verde al rojo):



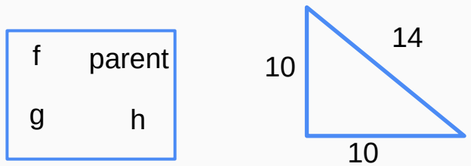
Heurística
Distancia Manhattan



Abiertos	B2, A2, A3, A4,D1, C1
Visitados	B3, C2, D2, D1, B4

Solución

A* (del verde al rojo):



Heurística
Distancia Manhattan

	A		B		C		D		E		F		G		H	
					84	C2	78	C2								
1					24	60	28	50								
	84	B3	70	B3	54	B3	64	C2								
2	14	70	10	60	14	50	24	40								
	70	B3									70	E4	72	F3		
3	10	60									62	10	72	0		
	84	B3	70	B3			78	C5	78	D4	78	E4	86	F3		
4	14	70	10	60			38	40	48	30	58	20	76	10		
			90	B4	84	B4	84	C5	92	D4	92	E4				
5			20	70	24	60	34	50	52	40	62	30				
							98	C5								
6							38	60								

Abiertos	A2, A4, C1, A5, B5, D5, D6, E5, F5, F4, F3, G3
Visitados	B3, C2, D2, B4, A3, B2, D1, C5, D4, E4, F3

G3 <- F3 <- E4 <- D4 <- C5 <- B4 <- B3

Más algoritmos →

- **Camino mínimo**
  - **Floyd Warshall**
  - Johnson
- Ordenamiento topológico
- Componentes fuertemente conexos (Kosaraju)
- Flujos de redes (Ford-Fulkerson)

UTEC
UNIVERSIDAD DE INGENIERÍA
Y TECNOLOGÍA

Welcome to Algorithms and Data Structures! - CS2100