

The background features a complex network diagram with numerous white circular nodes connected by thin gray lines. A specific cluster of nodes in the center is highlighted with a light blue shaded area and a dashed blue border. One node within this highlighted cluster is colored a solid blue.

# **Busqueda en espais d'estats**

**Models d'intel·ligència artificial**

# Busqueda

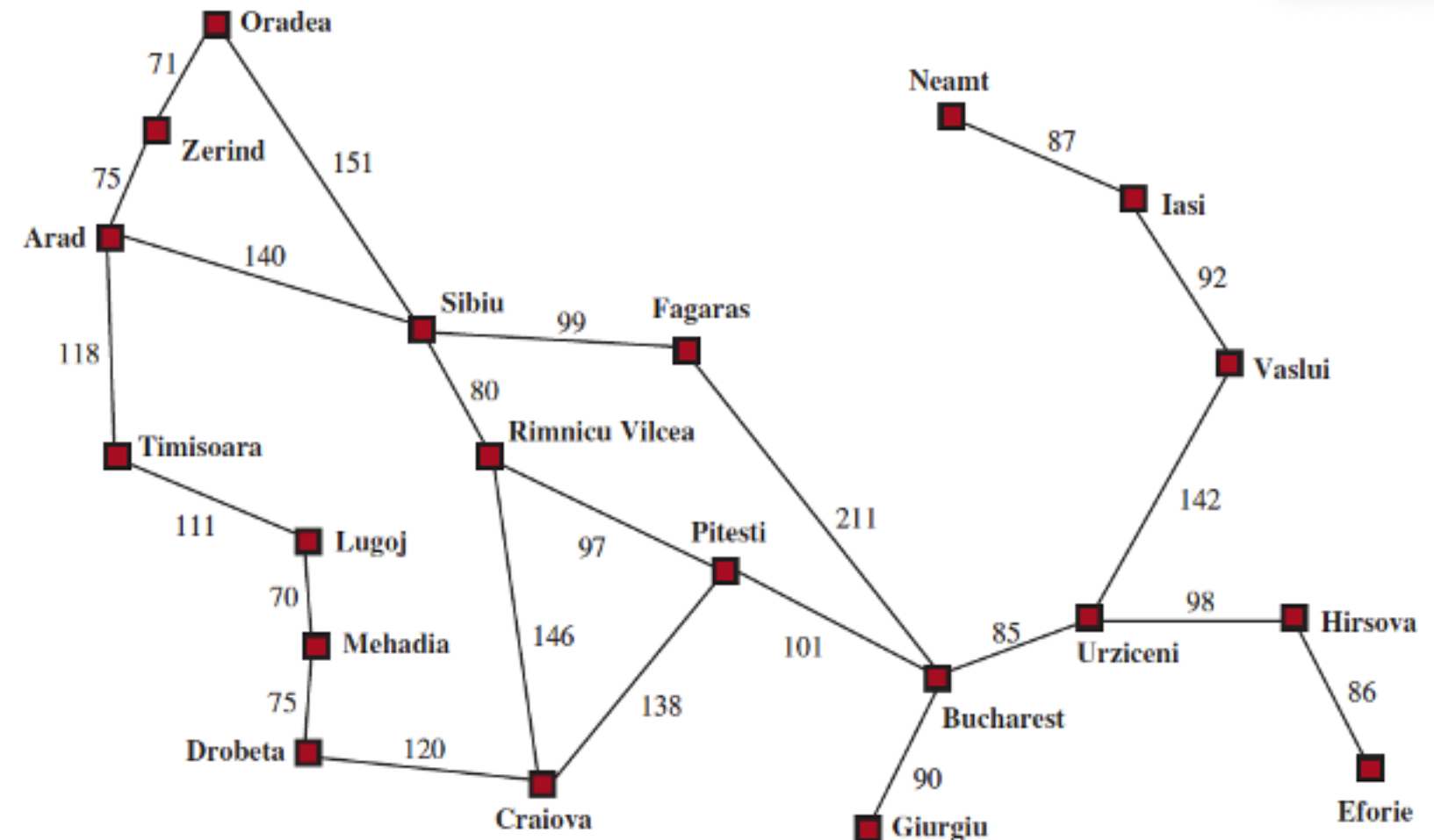
- Fonamental en molts dominis
  - Jocs, Planificació...
- Funciona molt bé en altres tipus de problemes
  - Diagnòstic, Control, Aprenentatge, ...
- És una tècnica molt general
  - Podem aplicar-la a problemes sense solució específica
  - Molt útil per aproximar
- Les tècniques de cerca són la base de molts sistemes intel·ligents

# Busqueda en espais d'estats

- Els **problemes de búsqüeda** tindran
  - Un estat inicial
  - Una funció de sucesió
    - Defineix els estats successors d'un estat i el cost per arribar a aquests estats
  - Un estat final
- Una **solució** és una seqüència d'estats (un plà) que ens porten de l'estat inicial a l'estat final

# Exemple: Viatjar per Romania

- **Espai d'estats:** Ciutats
- **Funcio de successió:** Carreteres.  
Cost: Distància
- **Estat inicial:** Arad
- **Comprovar si un estat és final:**  
Estat = Bucharest
- **Solució:** Seqüència de ciutats que ens porten d'Arad a Bucharest



# Exemple: Botelles d'aigua (I)

*Tenim dues botelles d'aigua, una de 4 litres i una altra de 3 litres. Volem obtenir 2 litres d'aigua.*

*Podem omplir les botelles, buidar-les o trasvasar l'aigua d'una a l'altra.*

- **Espai d'estats:** Estat de les botelles
- **Funcio de succesió:** Operacions de buidar, omplir i trasvasar
- **Estat inicial:**  $(0,0)$
- **Comprovar si un estat és final:** Estat =  $(2,0)$
- **Solució:** Seqüència d'operacions que ens porten de  $(0,0)$  a  $(2,0)$

# Exemple: Botelles d'aigua (II)

## Observacions

- Tal com està formulat el problema, no poden haver-hi estats no enters.
- Alguns estats són impossibles d'aconseguir. Ex: (1, 2)
- Algunes accions no produeixen canvis.
  - Ex:  $(0,0) + \text{buidar}(4) = (0,0)$

# Exemple: Puzzle 8 (I)

*Tenim un tauler de 3x3 amb 8 peces numerades del 1 al 8 i un espai buit. Volem moure les peces per aconseguir l'estat final.*

- **Espai d'estats:** -
- **Funcio de succesió:** -
- **Estat inicial:** -
- **Comprovar si un estat és final:** -
- **Solució:** -

7	2	4
5		6
8	3	1

**Estat inicial**

	1	2
3	4	5
6	7	8

**Estat final**

## Exemple: Puzzle 8 (II)

- **Espai d'estats:** Les diferents posicions de les peces. *Quantes?*
- **Funcio de succesió:** Podem moure la peça buida en les 4 direccions. *Podem sempre fer els 4?*
- **Estat inicial:** P.e. el de la figura anterior
- **Comprovar si un estat és final:**  
Verificar que les peces estan a la posició correcta.
- **Solució:** Seqüència de moviments que ens porten a l'estat final

7	2	4
5		6
8	3	1

**Estat inicial**

	1	2
3	4	5
6	7	8

**Estat final**



# Exemple: Puzzle 8 (III)

## Observacions

- El nombre d'estats és molt gran:  $9! = 362.880$  estats
- Solament quan la peça buida està al centre del tauler podem fer els 4 moviments
- No tots els estats tenen solució
- Hi ha moltes solucions
  - Quina és la millor?

# Situacions més complicades

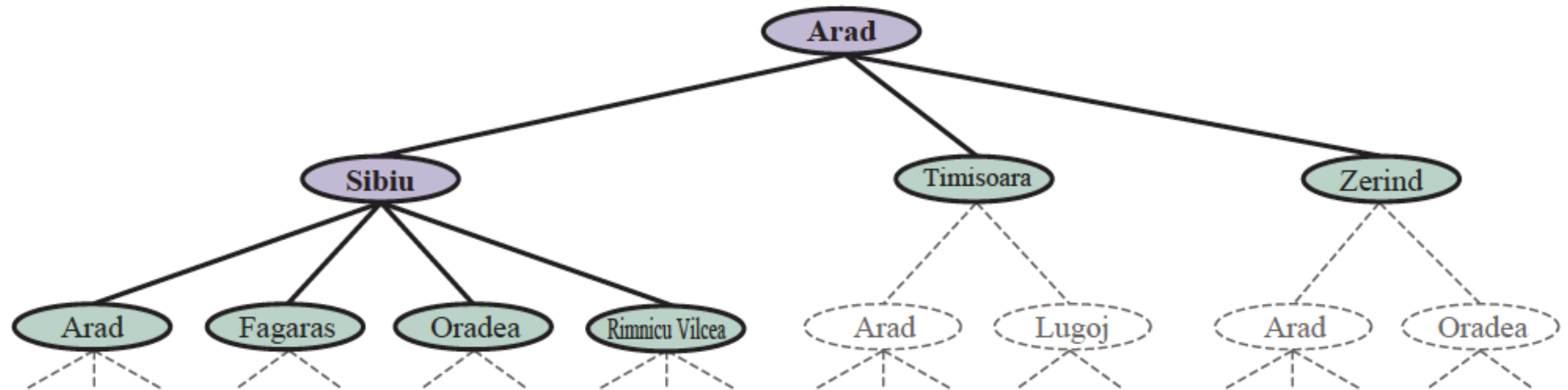
- En els exemples anteriors, de cada estat coneixíem
  - Els estats successors
  - El cost de cada estat successor
- En altres problemes el resultat de cada acció és incert
- Veurem tècniques per tractar algunes d'aquestes situacions
  - *Mètodes probabilístics*: assignar probabilitats als estats successors
  - *Mètodes de cerca adversarial*: els estats successors són determinats per un oponent

# **Representació dels problemes de cerca**

## **Arbres de cerca**

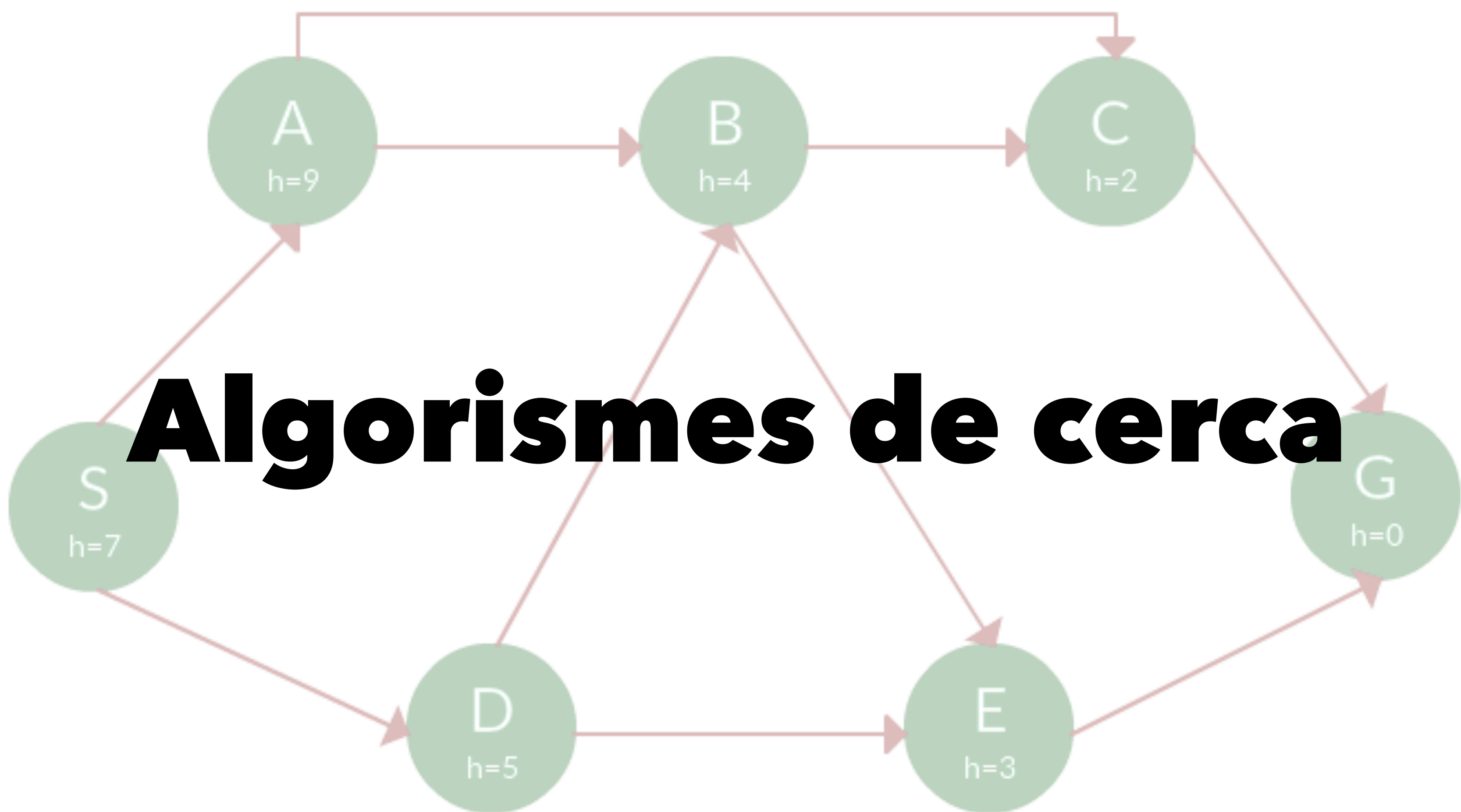
- Els problemes de cerca es poden representar com un arbre
- Els nodes són els estats
- Les arestes són les accions
- Els costos són els pesos de les arestes
- Podem aplicar algorismes de cerca de camins mínims
- L'espai de cerca ha de ser finit

# Exemple: Viatjar per Romania



- Els nodes ja visitats es mostren en gris
- Els oberts en blanc i els encara per visitar en linees discontinúes.

# Algoritmes de cerca



# Definició

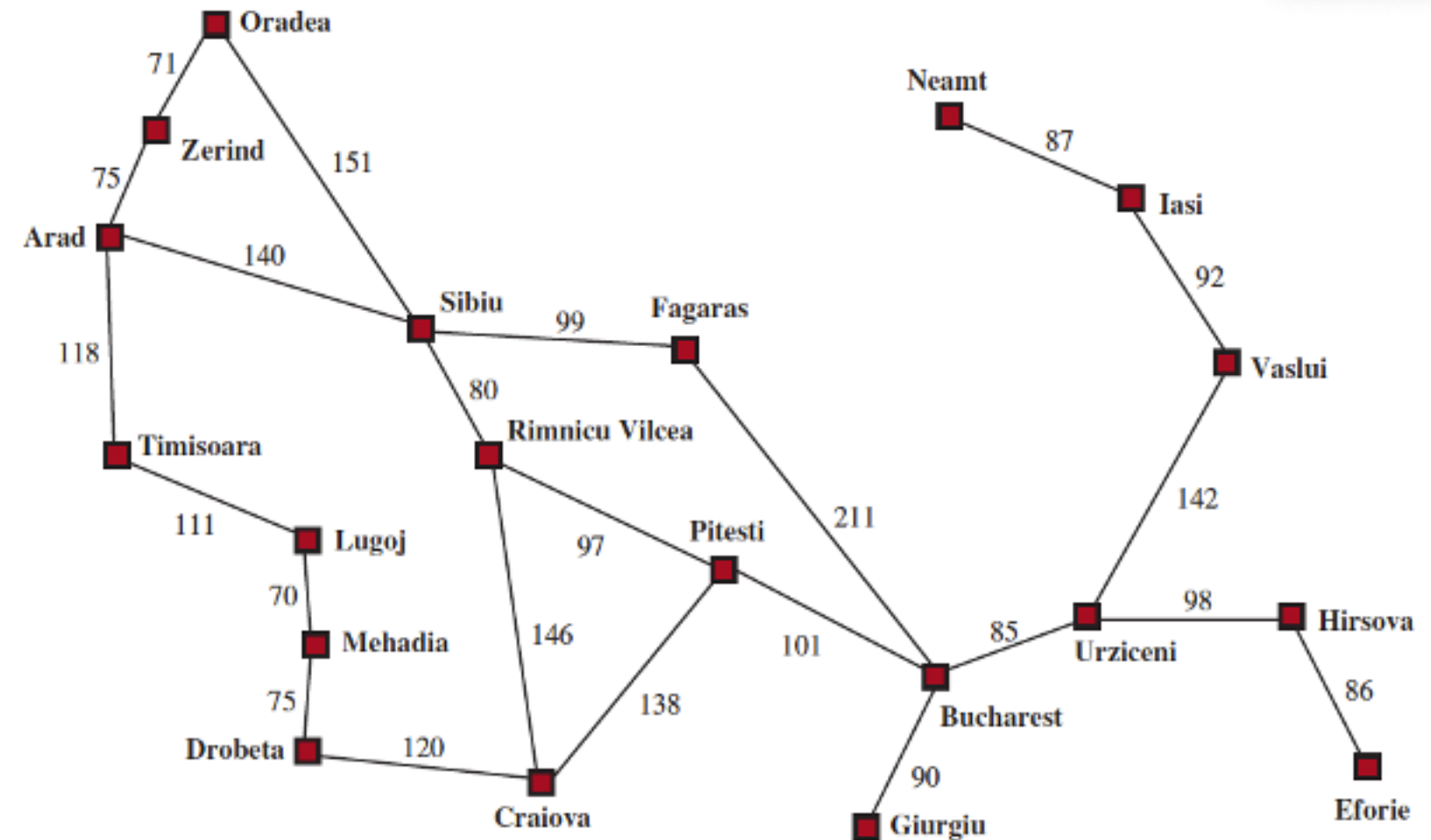
- Els algorismes de cerca són algorismes de propòsit general
  - Poden ser aplicats a qualsevol problema de cerca
  - Els problemes de cerca són un cas particular dels problemes de camins mínims
- Sortida:
  - Una serie d'accions que ens porten de l'estat inicial a l'estat final
  - El resultat pot ser una solució òptima en cost, òptima en temps o no tindre cap tipus de garantia d'optimalitat.

# Funcionament general

- Tindrem una llista d'estats coneguts pero per visitar anomenada **frontera**.
- Inicialment la frontera conté l'estat inicial.
- En cada iteració, agafarem un estat de la frontera, aplicarem la funció de successió i afegirem els nous estats a la frontera.
  - Si l'estat és final, hem acabat.
  - Si la frontera està buida, no hi ha solució.
  - Per cada estat anotarem el seu pare, per poder reconstruir la solució.
- En cada iteració, podem aplicar una estratègia per decidir quin estat de la frontera agafem.

## Exemple de funcionament (Alternativa I)

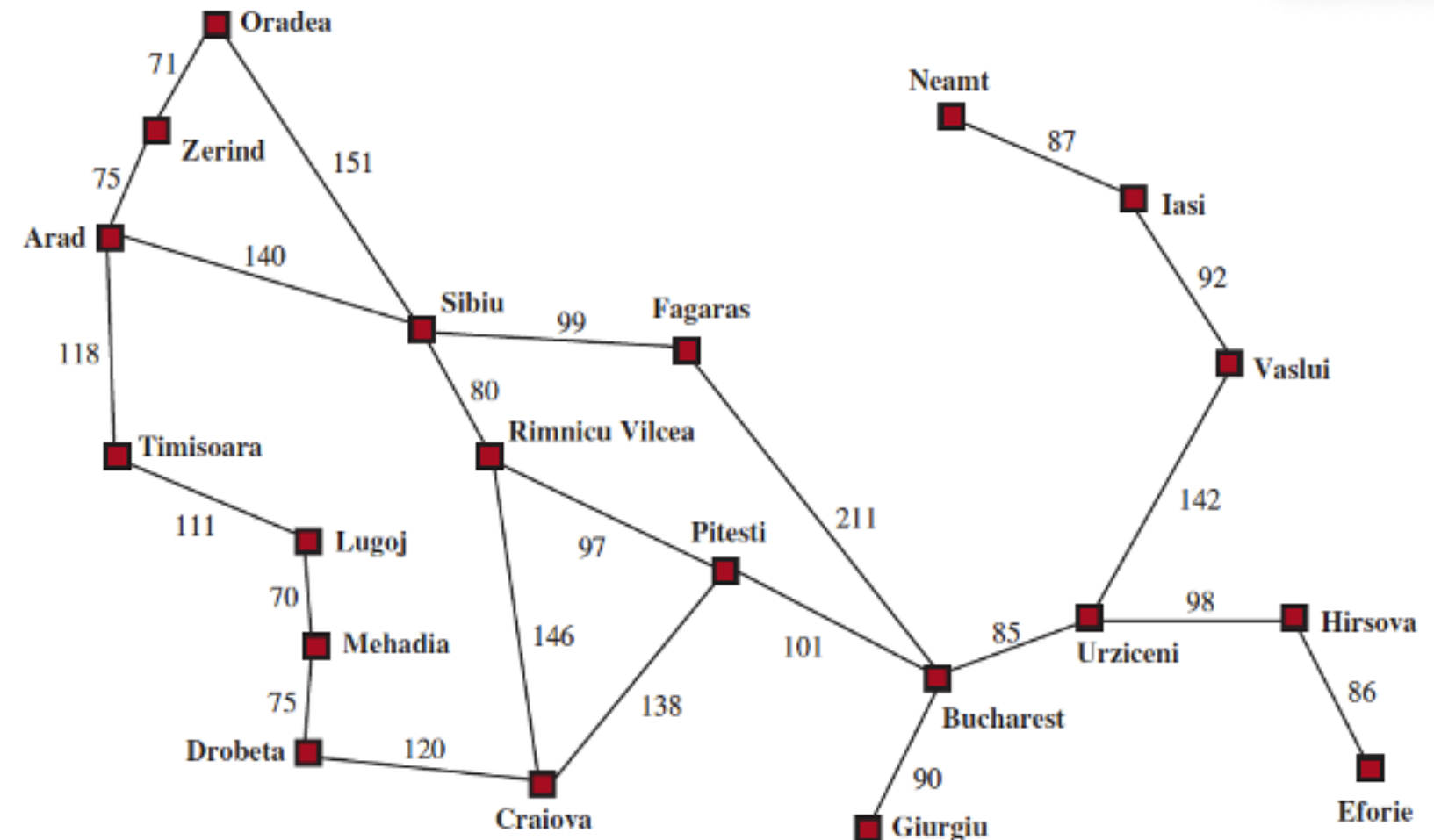
- Frontera: {**Arad**}. Objectiu: **Bucharest**
1. Obrim **Arad**: {Z<A>, T<A>, **S<A>**},
  2. Obrim **Sibiu**: {Z<A>, T<A>, A<S,A>, O<S,A>, **F<S,A>**, R<S,A>}
  3. Obrim **Fagaras**: {Z<A>, T<A>, A<S,A>, O<S,A>, R<S,A>, S<F,S,A>, **B<F,S,A>**}
  4. Tenim la solució en la frontera. Cost:  
140+99+211 = **450**





## Exemple de funcionament (Alternativa II)

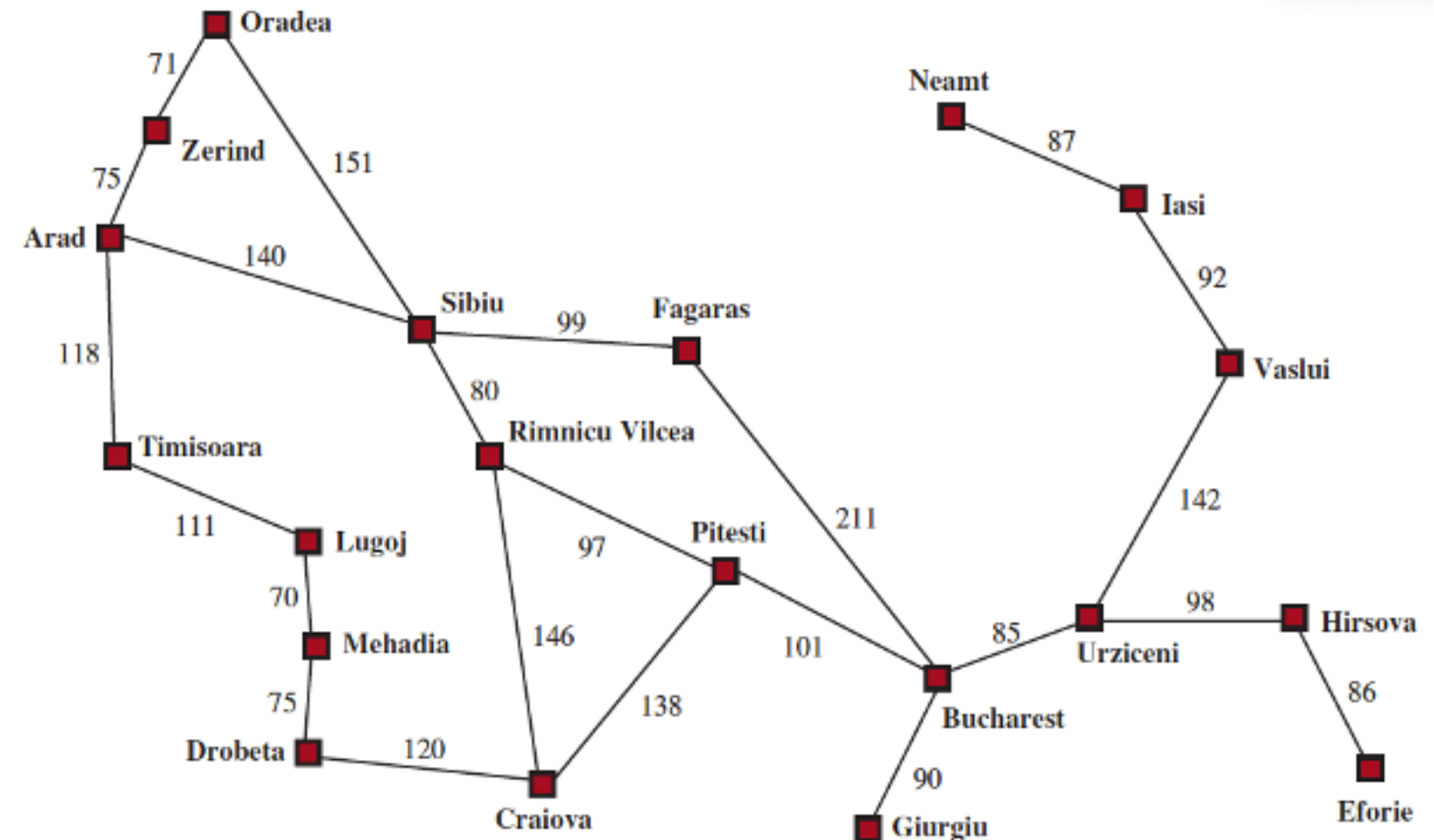
- Frontera: **{Arad}**. Objectiu: **Bucharest**
1. Obrim **Arad**: {Z<A>, T<A>, **S<A>**},
  2. Obrim **Sibiu**: {Z<A>, T<A>, A<S,A>, O<S,A>, **F<S,A>**, R<S,A>}
  3. Obrim **R.V**: {Z<A>, T<A>, A<S,A>, O<S,A>, R<S,A>, S<R,S,A>, **P<R,S,A>**, C<R,S,A>}
  4. Obrim **Pitesti**: {Z<A>, T<A>, A<S,A>, O<S,A>, R<S,A>, S<R,S,A>, P<R,S,A>, C<R,S,A>, R<P,R,S,A>, C<P,R,S,A>, **B<P,R,S,A>**}
  5. Tenim la solució en la frontera. Cost:  
**140+80+97+101 = 418**



# Exemple de funcionament

## Observacions

- En aquest exemple, els estats són les ciutats
- Problemes:
  - Poden apareixer estats repetits en la frontera
  - També es poden produir cicles
- L'ordre de les ciutats en la frontera determinarà:
  - Si trobem o no la solució
  - El cost de la solució
  - El temps d'execució i l'espai de memòria necessari



# Propietats dels algorismes de cerca

- **Criteris** per comparar algorismes de cerca
  - **Completitud**: Garantia de trobar una solució si existeix
  - **Optimalitat**: Garantia de trobar la solució òptima
  - **Complexitat temporal**: Temps d'execució
  - **Complexitat espacial**: Memòria necessària

7	2	4
5		6
8	3	1

# Búsqueda no informada

L

7	2	4
	5	6
8	3	1

R

7	2	4
5	6	
8	3	1

U

7		4
5	2	6
8	3	1

D

7	2	4
5	3	6
8		1

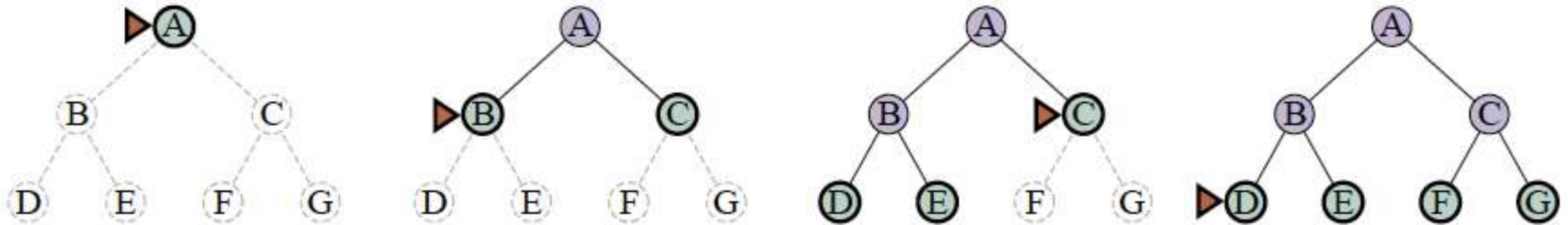
# Búsqueda no informada

## Característiques

- No utilitza cap informació sobre el problema
- Aplica una estratègia de cerca fins que troba la solució
- Aquesta estratègia determina l'ordre en què s'exploraran els estats
- L'estrategia serà fixa, no pot canviar en funció del problema
- Alguns algorismes de cerca no informada:
  - **Amplitud, Profunditat, Cost uniforme, Profunditat limitada, Profunditat iterativa**

# Búsqueda en amplitud

- Estrategia utilitzable quan totes les accions tenen el mateix cost
- Explora tots els estats a una profunditat  $p$  abans d'explorar els estats a profunditat  $p + 1$
- Garanteix trobar la solució òptima
- Definim la **frontera** com una **cua** (FIFO)
- Els estats ja visitats es guarden en una llista o conjunt (per evitar cicles)



# Búsqueda en amplitud

## Implementació

```
def cerca_amplada(estat_inicial):  
    """Cerca en amplada en un problema."""  
    frontera = collections.deque([estat_inicial])  
    visitats = set()  
  
    while frontera:  
        estat = frontera.popleft()  
        visitats.add(estat)  
  
        if es_solucio(estat):  
            return estat  
  
        for sucesor in successors(estat):  
            if sucesor not in visitats:  
                frontera.append(sucesor)
```

# Búsqueda en amplitud

## Exemple: Botelles d'aigua (I)

- **Estat inicial:**  $(0, 0)$  - **Estat final:**  $(2, *)$  o  $(*, 2)$
- **Funcio de succesió:** Operacions de buidar, omplir i trasvasar

1. Frontera =  $\{ \langle \mathbf{(0,0)} \rangle \}$

2. Frontera =  $\{ \langle \mathbf{(0,0)}, (3,0) \rangle, \langle \mathbf{(0,0)}, (0,4) \rangle \}$

3. Frontera =  $\{ \langle (0,0), (0,4) \rangle, \langle \mathbf{(0,0)}, (3,0), (0,0) \rangle, \langle \mathbf{(0,0)}, (3,0), (3,4) \rangle, \langle \mathbf{(0,0)}, (3,0), (0,3) \rangle \}$

4. Frontera =  $\{ \langle (0,0), (3,0), (0,0) \rangle, \langle (0,0), (3,0), (3,4) \rangle, \langle (0,0), (3,0), (0,3) \rangle, \langle \mathbf{(0,0)}, (0,4), (0,0) \rangle, \langle \mathbf{(0,0)}, (0,4), (3,4) \rangle, \langle \mathbf{(0,0)}, (0,4), (3,1) \rangle \}$

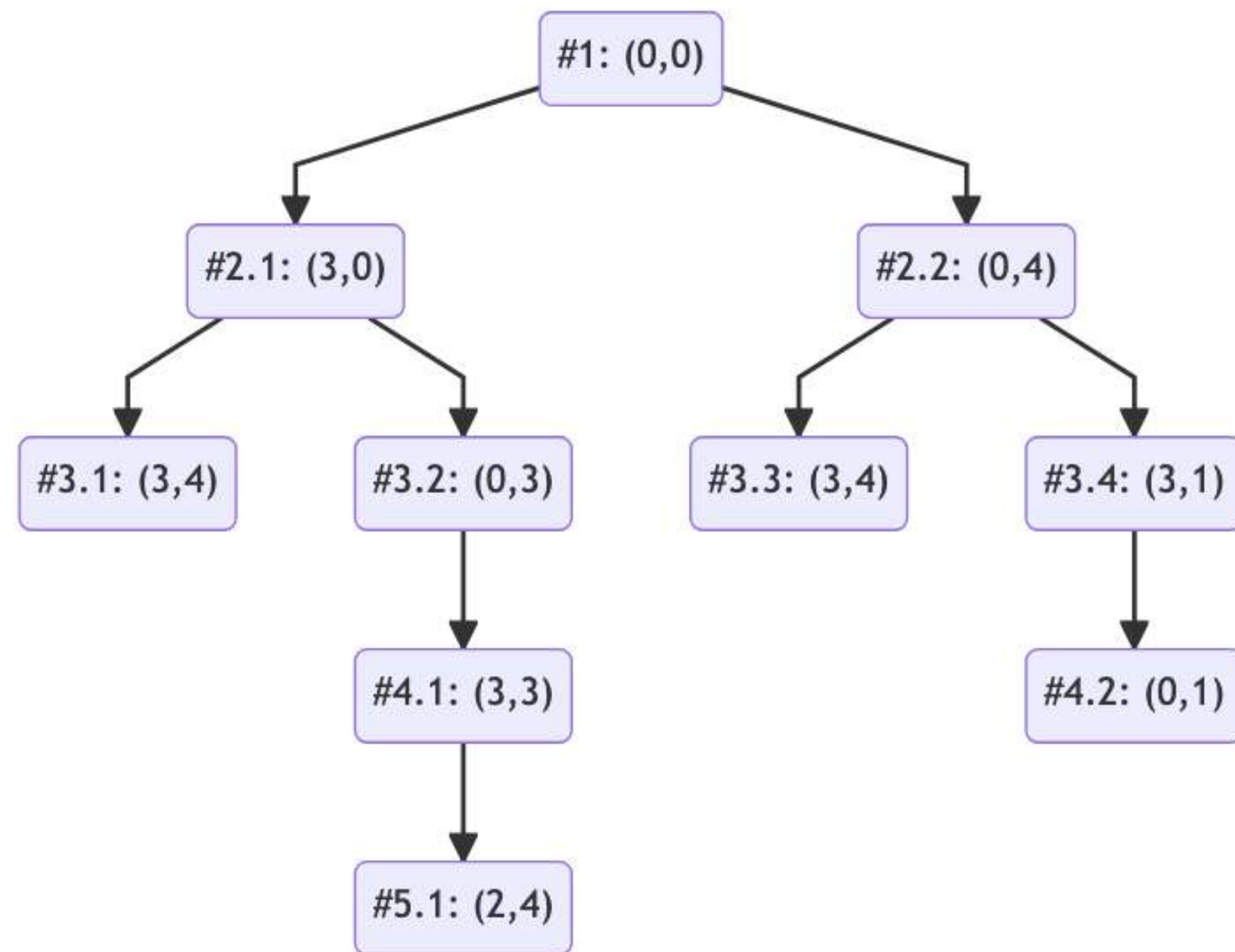
5. ...



# Búsqueda en amplitud

## Exemple: Botelles d'aigua (II)

- Representació de l'arbre de cerca
  - Cada node és un parell de valors (a,b) que representen l'estat de les botelles
  - La busca en amplitud explora l'arbre per nivells
  - Podem observar que solament s'explora un nombre molt reduït de tots els possibles estats



# Búsqueda en amplitud

## Propietats

- **Completitud:** Sí
  - Si l'espai de cerca és finit, la solució es trobarà en algun moment
- **Optimalitat:** Sí
  - Si totes les accions tenen el mateix cost, la primera solució trobada serà òptima
- **Complexitat temporal:**  $O(b^d)$ 
  - On  $b$  és el factor de ramificació i  $d$  és la profunditat de la solució
- **Complexitat espacial:**  $O(b^d)$ 
  - On  $b$  és el factor de ramificació i  $d$  és la profunditat de la solució

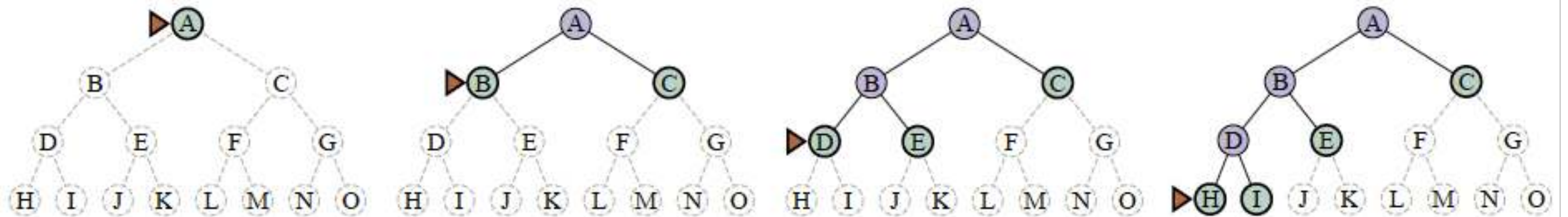
# Búsqueda en amplitud

## Problemes

- La complexitat espacial és un problema real.
  - Per exemple, suposem que cada estat ocupa **1KB** i que el **factor de ramificació és 10**.
  - Si la solució es troba a una profunditat de **10**, necessitarem **10GB** de memòria.
  - Si la solució es troba a una profunditat de **100**, necessitarem **10TB** de memòria.
  - Si la solució es troba a una profunditat de **1000**, necessitarem **10PB** de memòria.
- Típicament, *ens quedarem sense espai abans de quedar-nos sense temps.*

# Búsqueda en profundidad

- L'estratègia de cerca en profunditat és similar a la de cerca en amplitud
- Utilitza una **pila** (LIFO) en lloc d'una cua
- Aquesta estratègia **no** garanteix trobar la solució òptima
- L'algorisme arriba fins a una profunditat màxima *m* i després retrocedeix fins a trobar un camí alternatiu



# Búsqueda en profunditat

## Implementació (I)

```
def cerca_profunditat(estat_inicial):  
    """Cerca en profunditat en un problema."""  
    frontera = collections.deque([estat_inicial])  
  
    while frontera:  
        estat = frontera.pop()  
  
        if es_solucio(estat):  
            return estat  
  
        for sucesor in sucesors(estat):  
            if not cicle(problema, sucesor):  
                frontera.append(sucesor)
```

# Búsqueda en profundidad

## Exemple: Botelles d'aigua (I)

- **Estat inicial:**  $(0, 0)$  - **Estat final:**  $(2, *) \text{ o } (*, 2)$
- **Funcio de succesió:** Operacions de buidar, omplir i trasvasar

1. Frontera =  $\{ \langle (0, 0) \rangle \}$

2. Frontera =  $\{ \langle (0, 0), (3, 0) \rangle, \langle (0, 0), (0, 4) \rangle \}$

3. Frontera =  $\{ \langle (0, 0), (3, 0), (0, 0) \rangle, \langle (0, 0), (3, 0), (3, 4) \rangle, \langle (0, 0), (3, 0), (0, 3) \rangle, \langle (0, 0), (0, 4) \rangle \}$

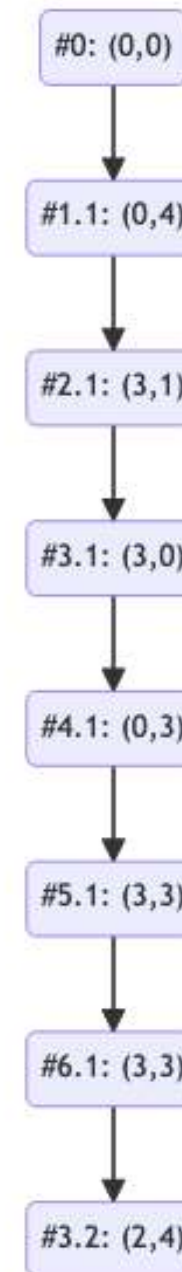
4. Frontera =  $\{ \langle (0, 0), (3, 0), (0, 0), (3, 0) \rangle, \langle (0, 0), (3, 0), (0, 0), (0, 4) \rangle, \langle (0, 0), (3, 0), (3, 4) \rangle, \langle (0, 0), (3, 0), (0, 3) \rangle, \langle (0, 0), (0, 4) \rangle \}$

5. ...

# Búsqueda en profundidad

## Exemple: Botelles d'aigua (II)

- Representació de l'arbre de cerca
  - Cada node és un parell de valors (a,b) que representen l'estat de les botelles
  - La busca en profunditat explora l'arbre fins a trobar una solució
  - Si no troba una solució, torna enrere fins a trobar un camí alternatiu
  - Si les solucions son infinites, l'algorisme pot no acabar mai



# Búsqueda en profundidad

## Propietats

- **Compleitud:** No
  - Si l'espai de cerca és finit, la solució es trobarà en algun moment
- **Optimalitat:** No
  - La primera solució trobada no té perquè ser òptima
- **Complexitat temporal:**  $O(b^m)$ 
  - On  $b$  és el factor de ramificació i  $m$  és la profunditat màxima de l'arbre
  - Si  $m$  és molt gran, la complexitat temporal pot ser molt alta
- **Complexitat espacial:**  $O(bm)$ 
  - On  $b$  és el factor de ramificació i  $m$  és la profunditat màxima de l'arbre
  - La complexitat espacial és molt millor que la de la cerca en amplitud si no hi ha cicles
  - Si hi ha cicles, la complexitat espacial es la mateixa que la de la cerca en amplitud



# Búsqueda en profunditat

## Quan utilitzar-la?

- En la pràctica, la cerca en profunditat és molt més ràpida que la cerca en amplitud
- La cerca en profunditat no necessita tant espai com la cerca en amplitud
- La cerca en profunditat és molt útil quan:
  - El factor de ramificació és molt gran
  - La solució es troba a una profunditat molt baixa
  - No ens importa trobar la solució òptima
  - Verifiquem que no es creen cicles

# Búsqueda en profundidad limitada

- La cerca en profunditat limitada és una variant de la cerca en profunditat
- En aquest cas, la cerca s'atura quan s'arriba a una profunditat màxima  $l$
- Si la solució es troba a una profunditat  $d > l$ , no es trobarà
- La cerca en profunditat limitada és completa si  $l$  és suficientment gran
- Ens permet evitar el problema de la cerca en profunditat quan les solucions son infinites

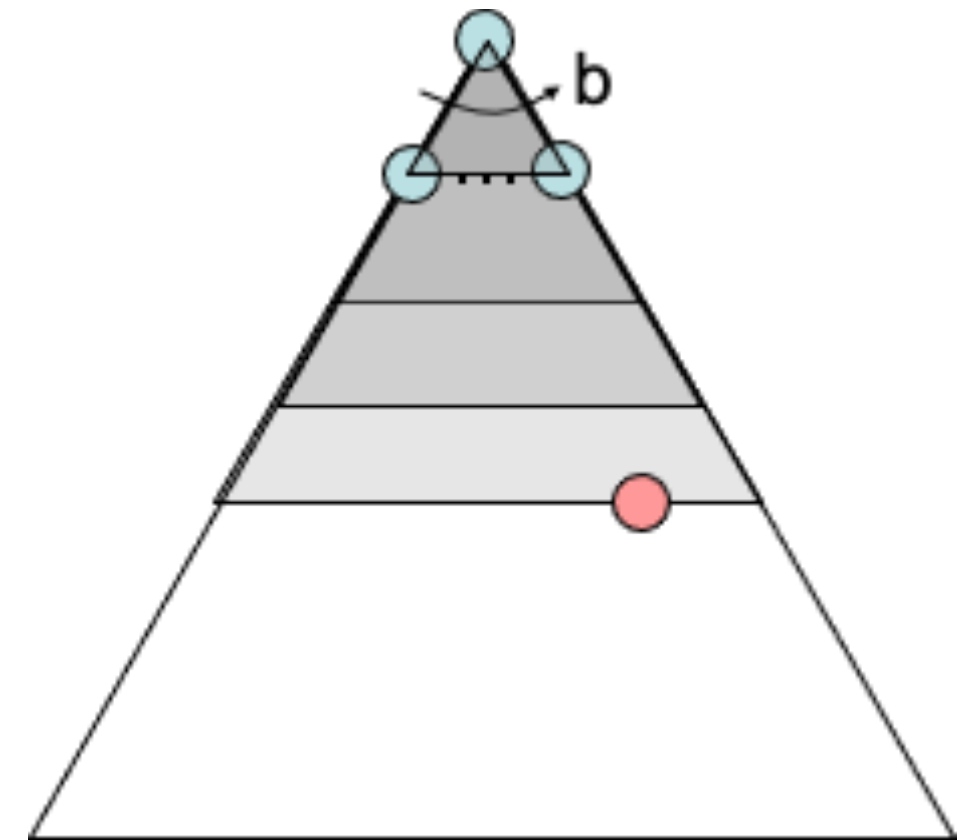
# Búsqueda en profundidad limitada

## Implementació

```
def cerca_profunditat_limitada(estat_inicial, l):  
    """Cerca en profunditat limitada en un problema."""  
    frontera = collections.deque([estat_inicial])  
  
    while frontera:  
        estat = frontera.pop()  
  
        if es_solucio(estat):  
            return estat  
  
        for sucesor in sucesors(estat):  
            if not cicle(problema, sucesor) and profunditat(sucesor) < l:  
                frontera.append(sucesor)
```

# Búsqueda en profundidad iterativa

- Solució al problema de la cerca en amplitud y la cerca en profunditat utilitzant una única estratègia
- La cerca en profunditat iterativa és una cerca en profunditat limitada amb  $l$  creixent
- Comença amb  $l = 0$  i va incrementant  $l$  fins a trobar la solució



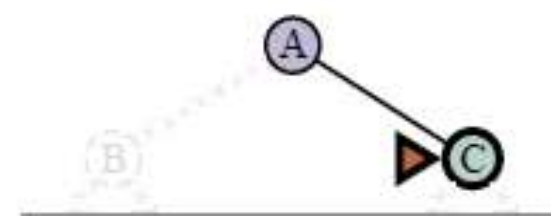
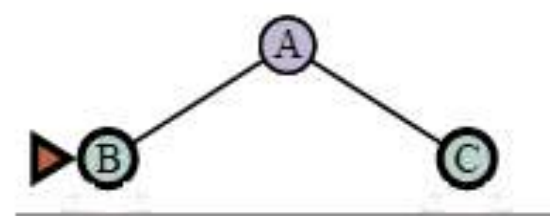
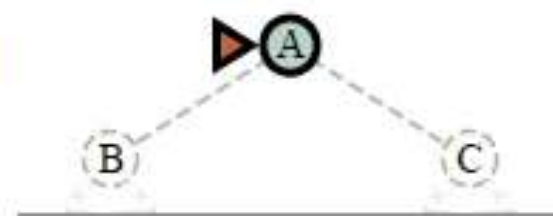
# Búsqueda en profundidad iterativa

## Traçat de l'algorisme (I)

limit: 0



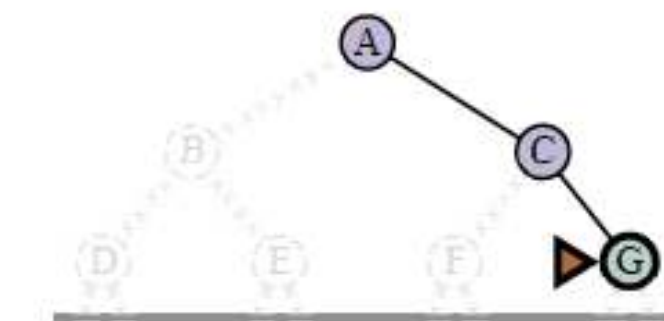
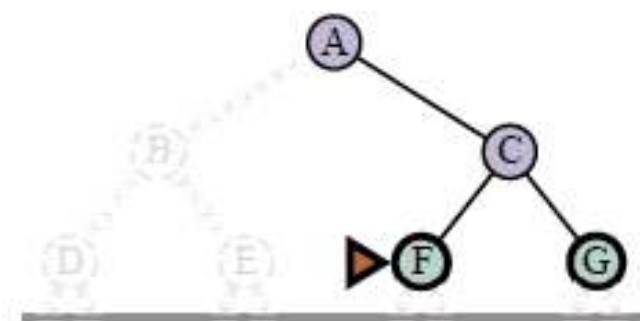
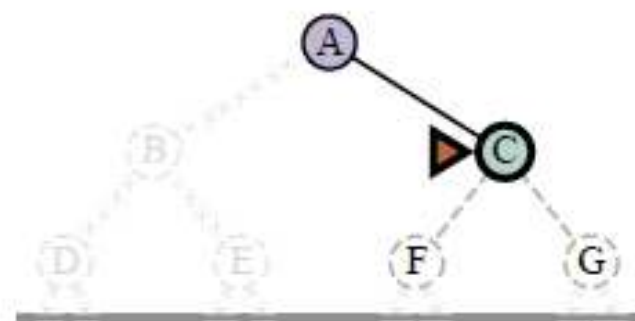
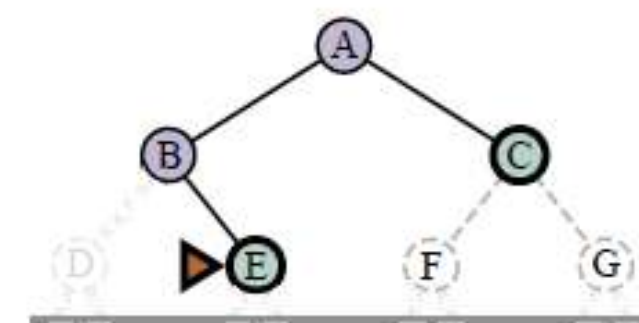
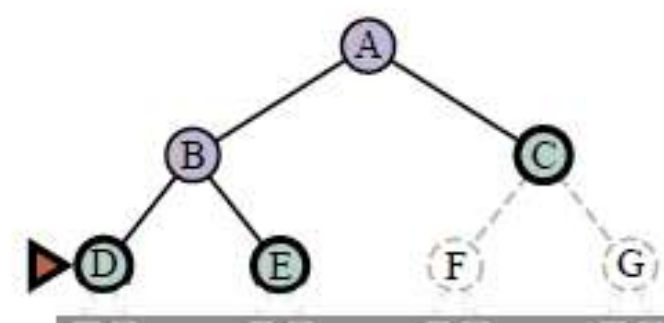
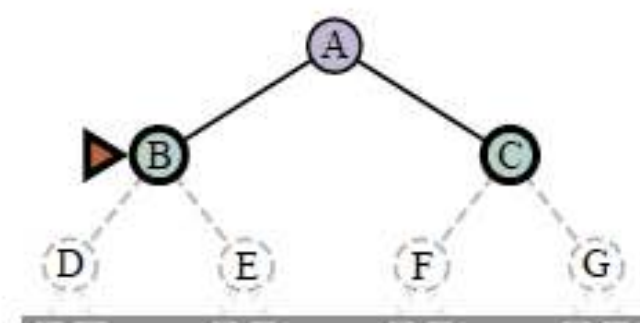
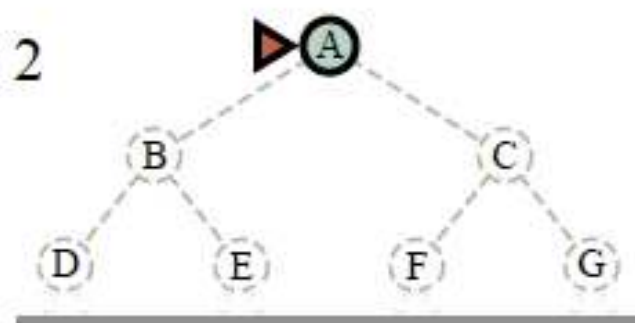
limit: 1



# Búsqueda en profundidad iterativa

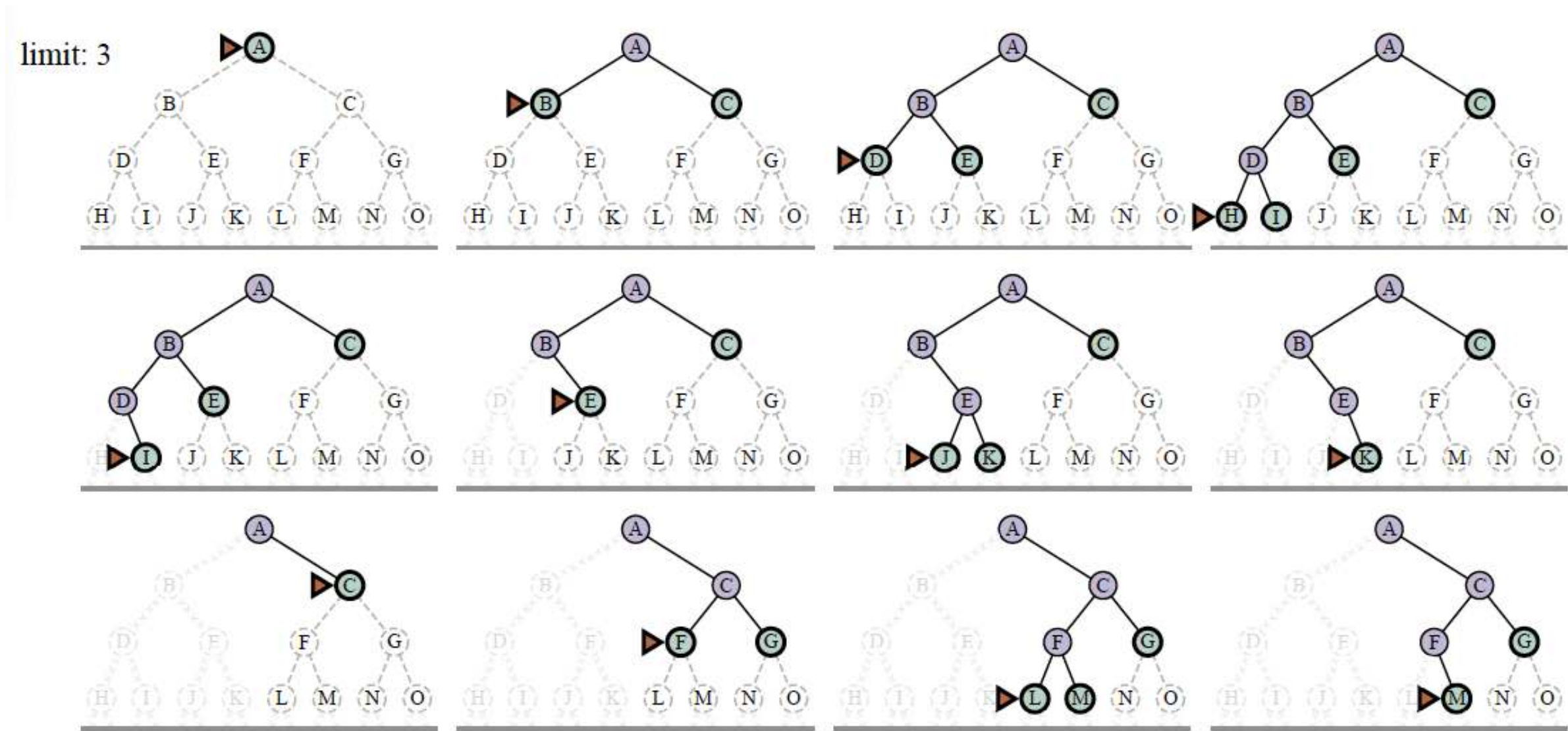
## Traçat de l'algorisme (II)

limit: 2



# Búsqueda en profundidad iterativa

## Traçat de l'algorisme (III)



# Búsqueda en profundidad iterativa

## Implementació

```
def cerca_profunditat_iterativa(estat_inicial):  
    """Cerca en profunditat iterativa en un problema."""  
    l = 0  
    while True:  
        solucio = cerca_profunditat_limitada(estat_inicial, l)  
        if solucio is not None:  
            return solucio  
        l += 1
```



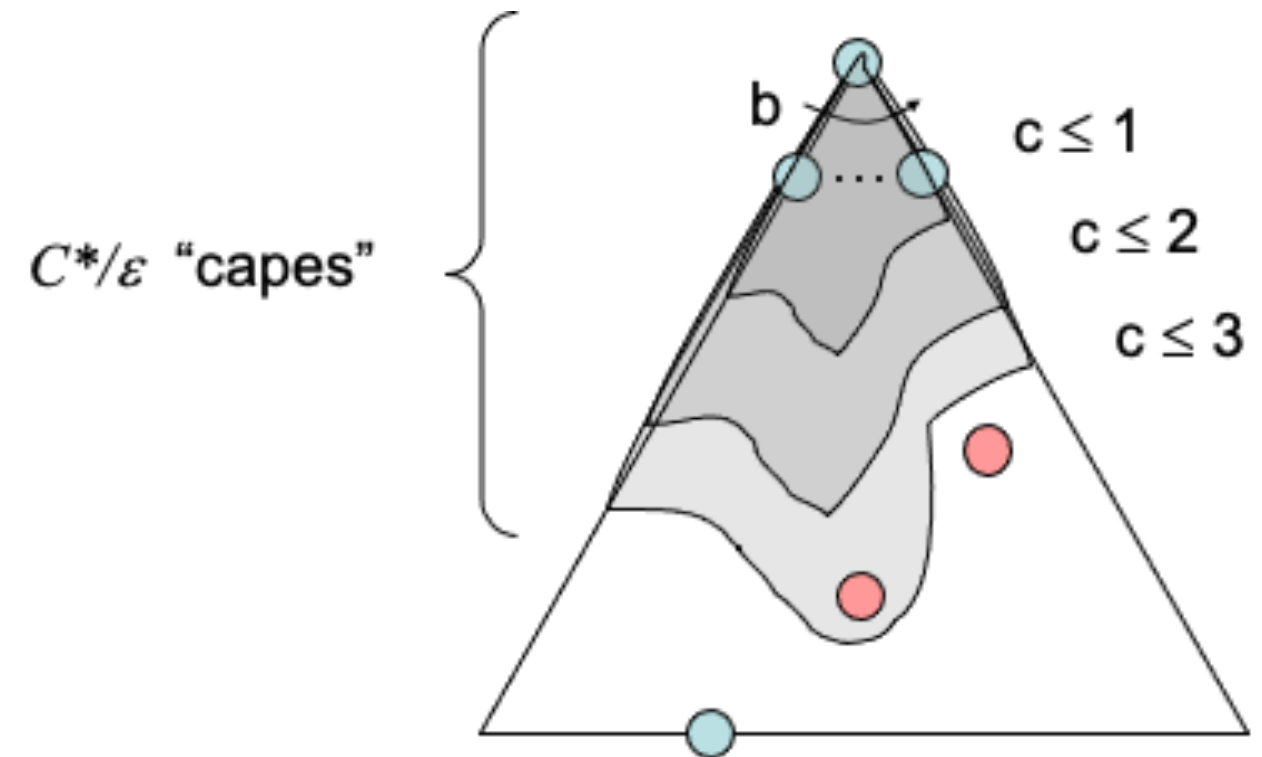
# Búsqueda en profundidad iterativa

## Propietats

- **Completitud:** Sí
  - Si l'espai de cerca és finit, la solució es trobarà en algun moment
- **Optimalitat:** Sí
  - La primera solució trobada serà òptima
- **Complexitat temporal i espacial:** Com la de la cerca en profunditat (com a màxim)

## Búsqueda de cost uniforme

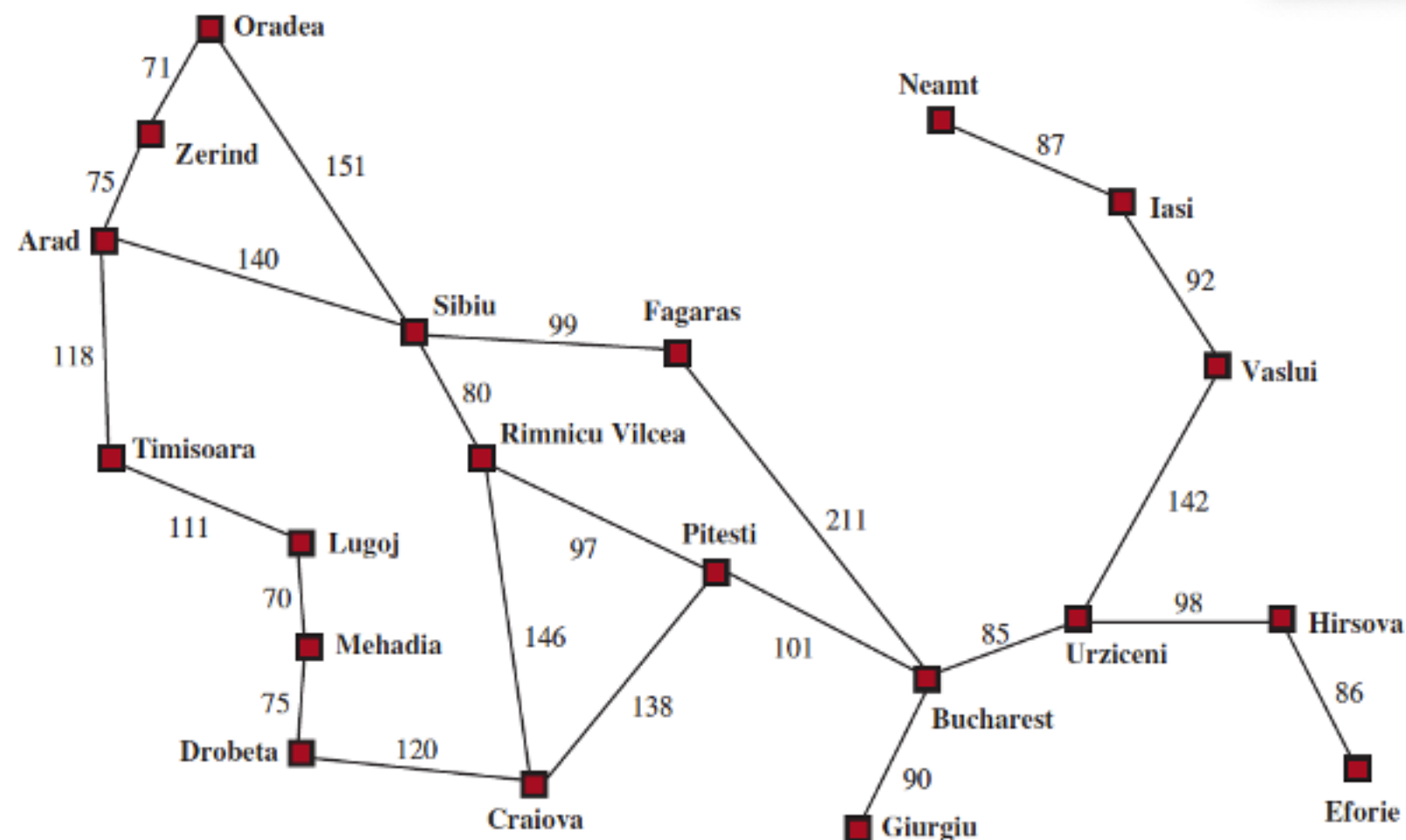
- La cerca de cost uniforme és una variant de la cerca en amplitud
- En aquest cas, la frontera s'ordena segons el cost del camí a cada estat (**cua de prioritat**)
- Estats visitats: de manera iterativa, es van visitant tots els que tenen un cost menor que l'actual
- Sí totes les accions tenen el mateix cost, la cerca de cost uniforme és equivalent a la cerca en amplitud



# Búsqueda de cost uniforme

## Exemple: Viatjar per Romania (I)

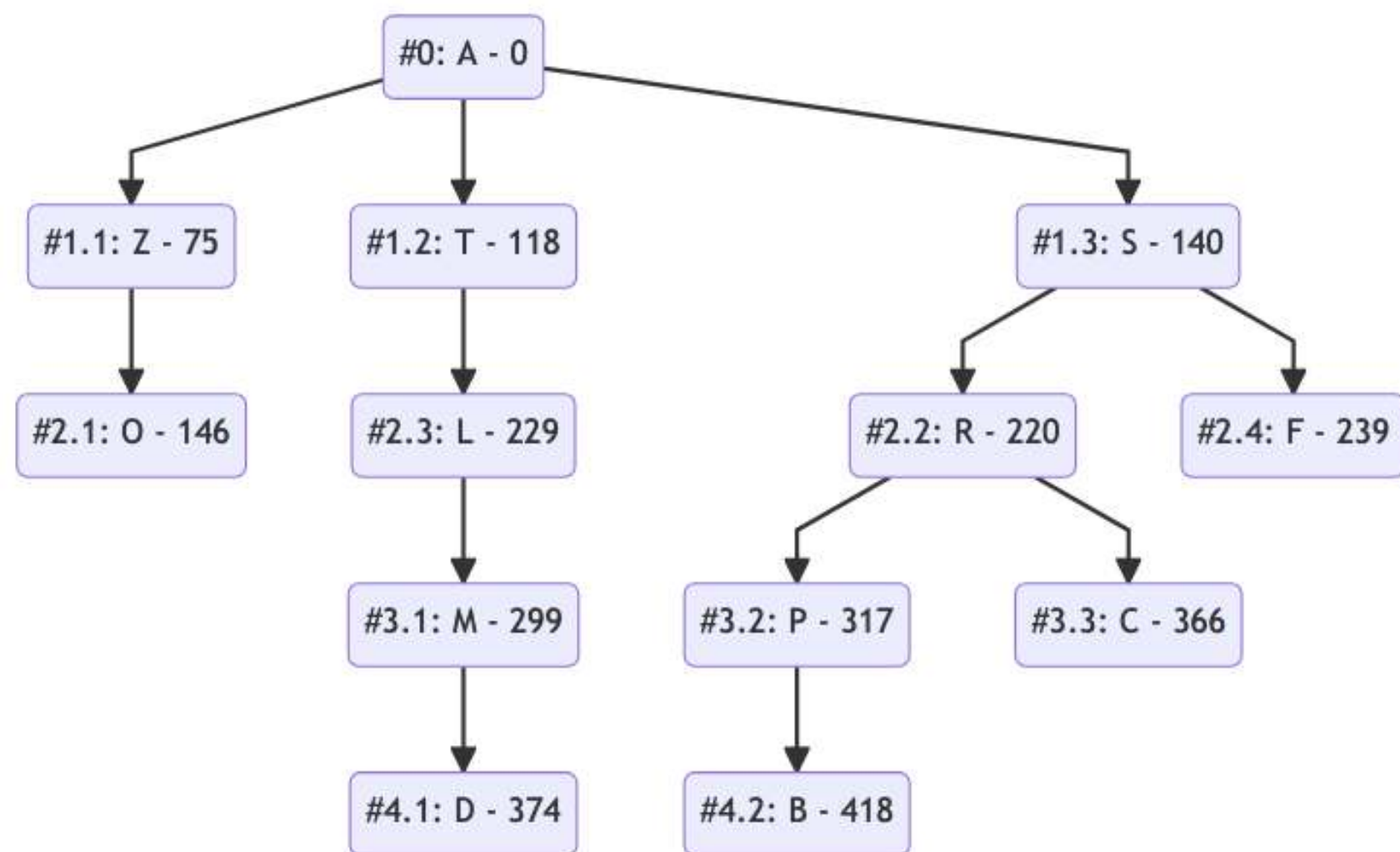
- **Estat inicial:** Arad
- **Funcio de successió:** Carreteres.
- **Cost:** Distància entre ciutats (en Km)
- **Comprovar si un estat és final:**  
Estat = Bucharest
- **Solució:** Seqüència de ciutats que ens porten d'Arad a Bucharest



# Búsqueda de cost uniforme

## Exemple: Viatjar per Romania (II)

- Representació de l'arbre de cerca
  - Cada node és un parell de valors (a,b) que representen l'estat de les botelles
  - La busca en amplitud explora l'arbre per nivells
  - Podem observar que solament s'explora un nombre molt reduït de tots els possibles estats



# Búsqueda de cost uniforme

## Implementació

```
def cerca_cost_uniforme(estat_inicial):  
    """Cerca de cost uniforme en un problema."""  
    frontera = priority_queue([(0, estat_inicial)])  
    visitats = set()  
  
    while frontera:  
        cost_actual, estat = frontera.pop()  
        visitats.add(estat)  
  
        if es_solucio(estat):  
            return estat  
  
        for cost, sucesor in sucesors(estat):  
            if sucesor not in visitats:  
                frontera.append(cost + cost_actual, sucesor)
```

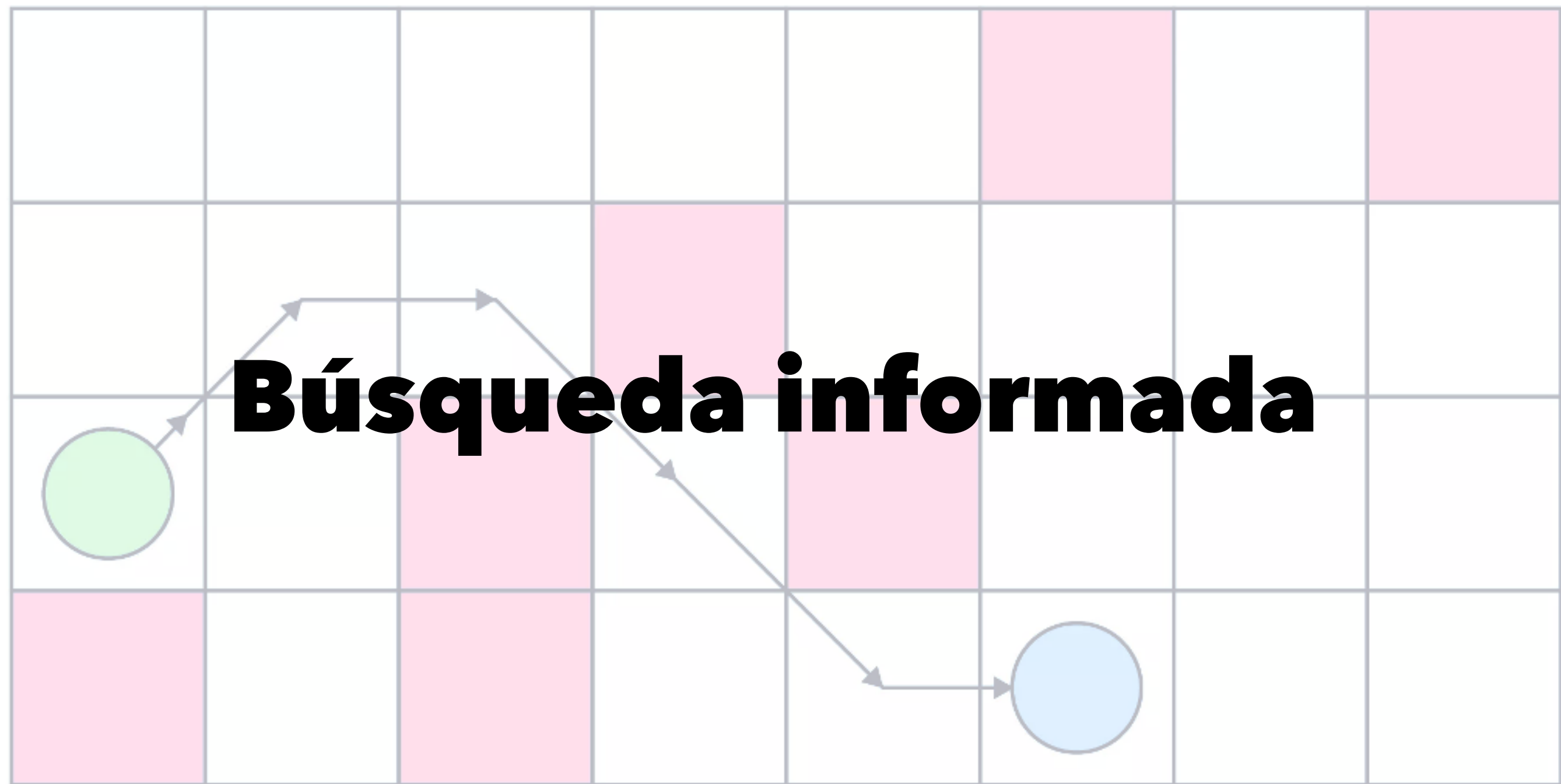
# Búsqueda de cost uniforme

## Propietats

- **Completitud:** Sí
  - Si l'espai de cerca és finit, la solució es trobarà en algun moment
- **Optimalitat:** Sí
  - La primera solució trobada serà òptima
- **Complexitat temporal i espacial:**  $O(b^C * \frac{1}{\epsilon} + 1)$ 
  - On  $b$  és el factor de ramificació i  $C^*$  és el cost de la solució òptima

# Gestió de fronteres

- La gestió de **fronteres** és un problema important en els algorismes de cerca
- Els algorismes que hem vist son tots molt semblants, la diferència està en com gestionen la frontera
  - Conceptualment sempre es tracta d'una **cua amb prioritat**
  - En la pràctica, per a les busquedes en profunditat i amplada **podem utilitzar una cua o una pila**
    - Per estalviar-nos el **sobrecost** de  $O(\log n)$  de la cua de prioritat
  - Podriem, fins i tot, programar una implementació on pugam **variar l'objecte frontera**.



**Búsqueda informada**



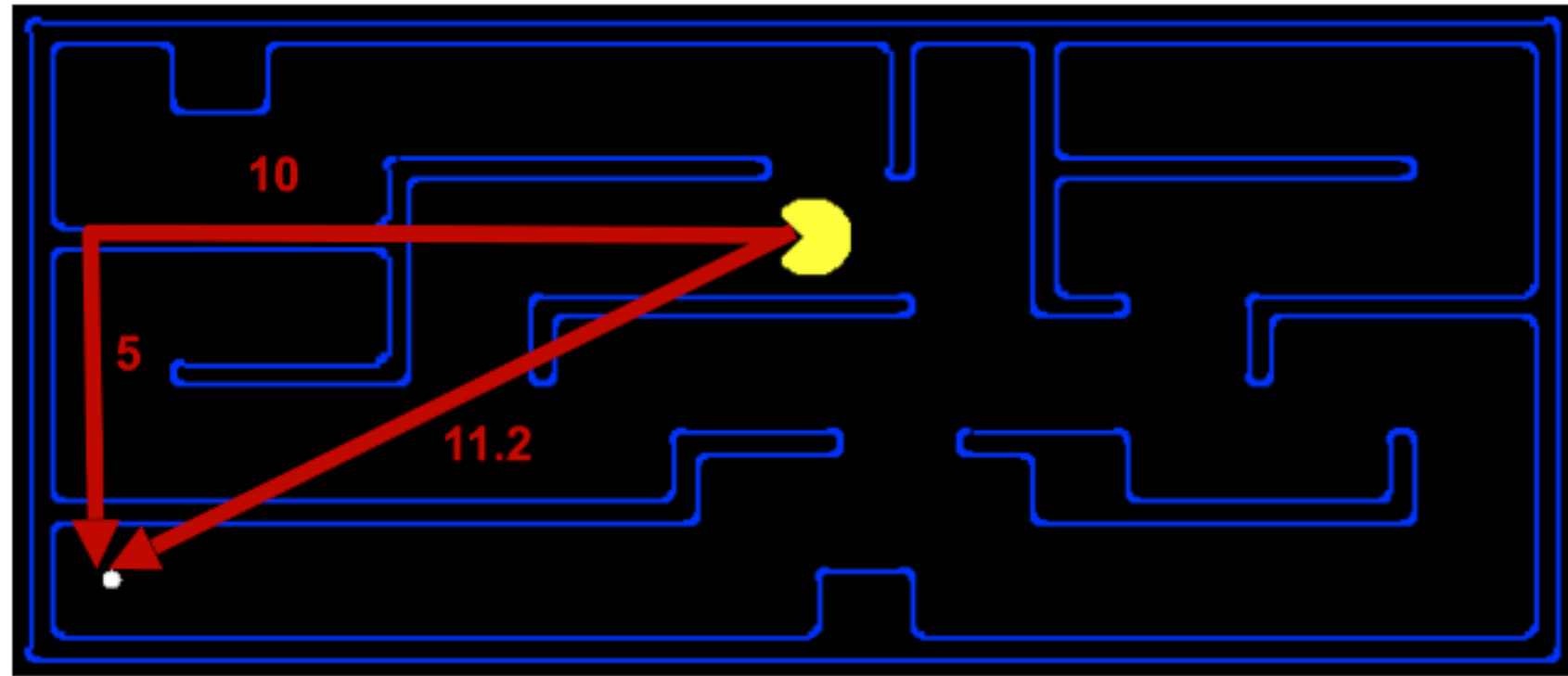
# Búsqueda informada

## Definició

- L'algorisme de búsqueda de cost uniforme és un algoritme molt eficient, té, però alguns problemes
  - Busca en totes les direccions, sense tenir en compte la direcció cap a la solució
  - Per tant, analitza més estats dels que seria estrictament necessari
- En aquesta part de la unitat veurem tècniques per solucionar aquestos problemes

# Heurístiques

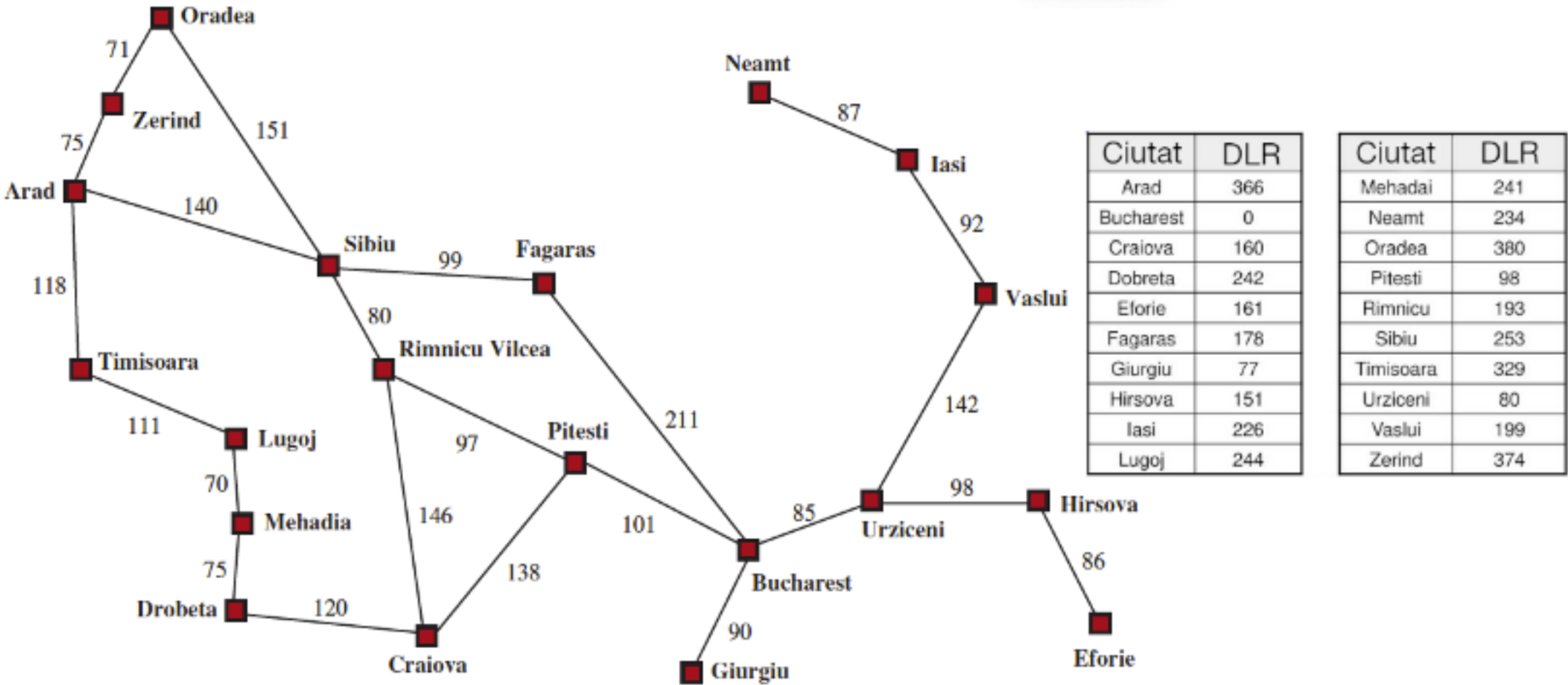
- Una **heurística** és:
  - Una funció que ens permet *estimar* el cost d'arribar a la solució des d'un estat
  - Dissenyada per un problema concret
- Heurístiques per rutes:
  - Distància en línia recta (euclidiana)
  - Distància manhattan



# Heurístiques

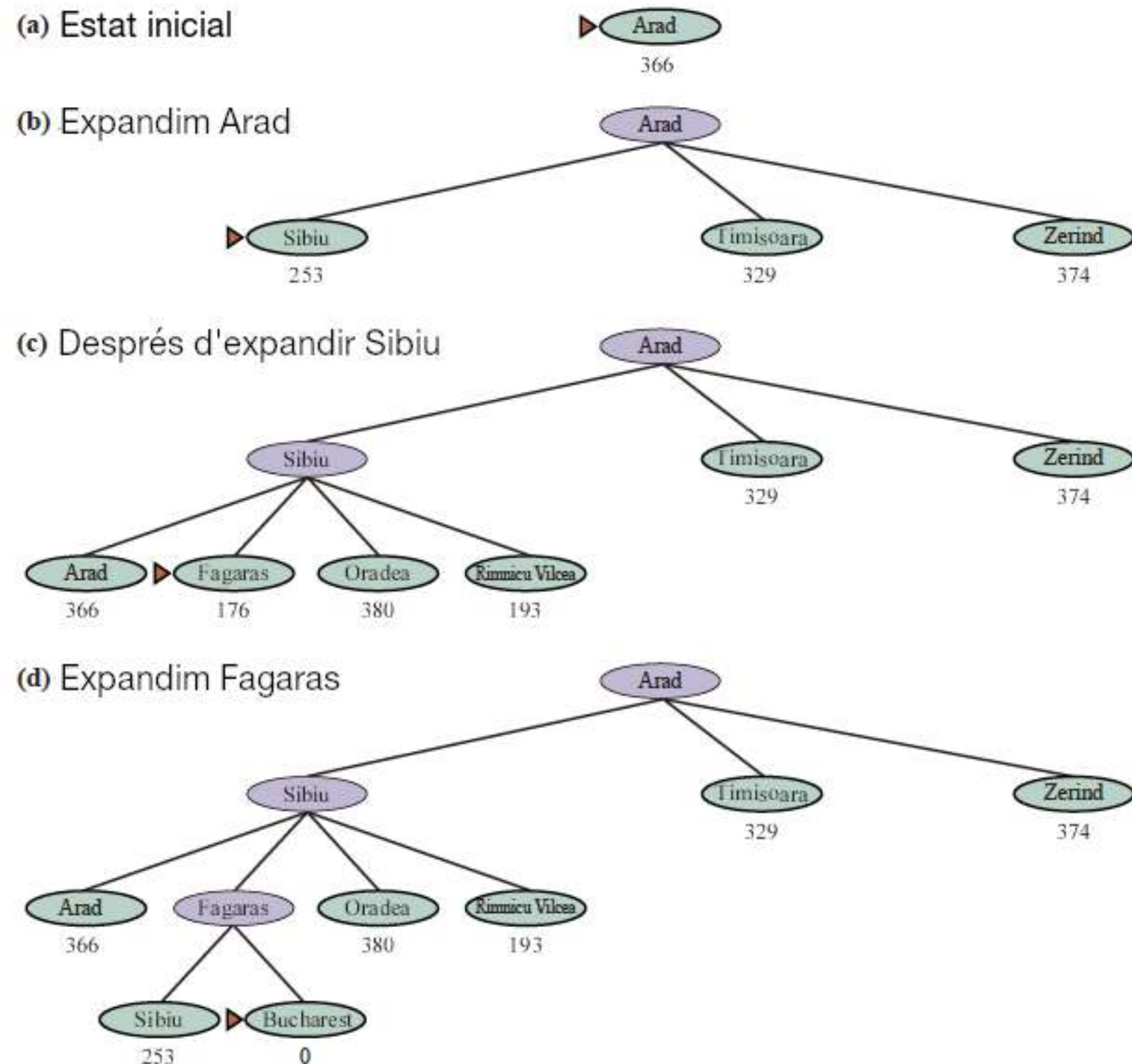
## Exemple: Viatjar per Romania

- **Heurística:** Distància en línia recta (euclidiana)



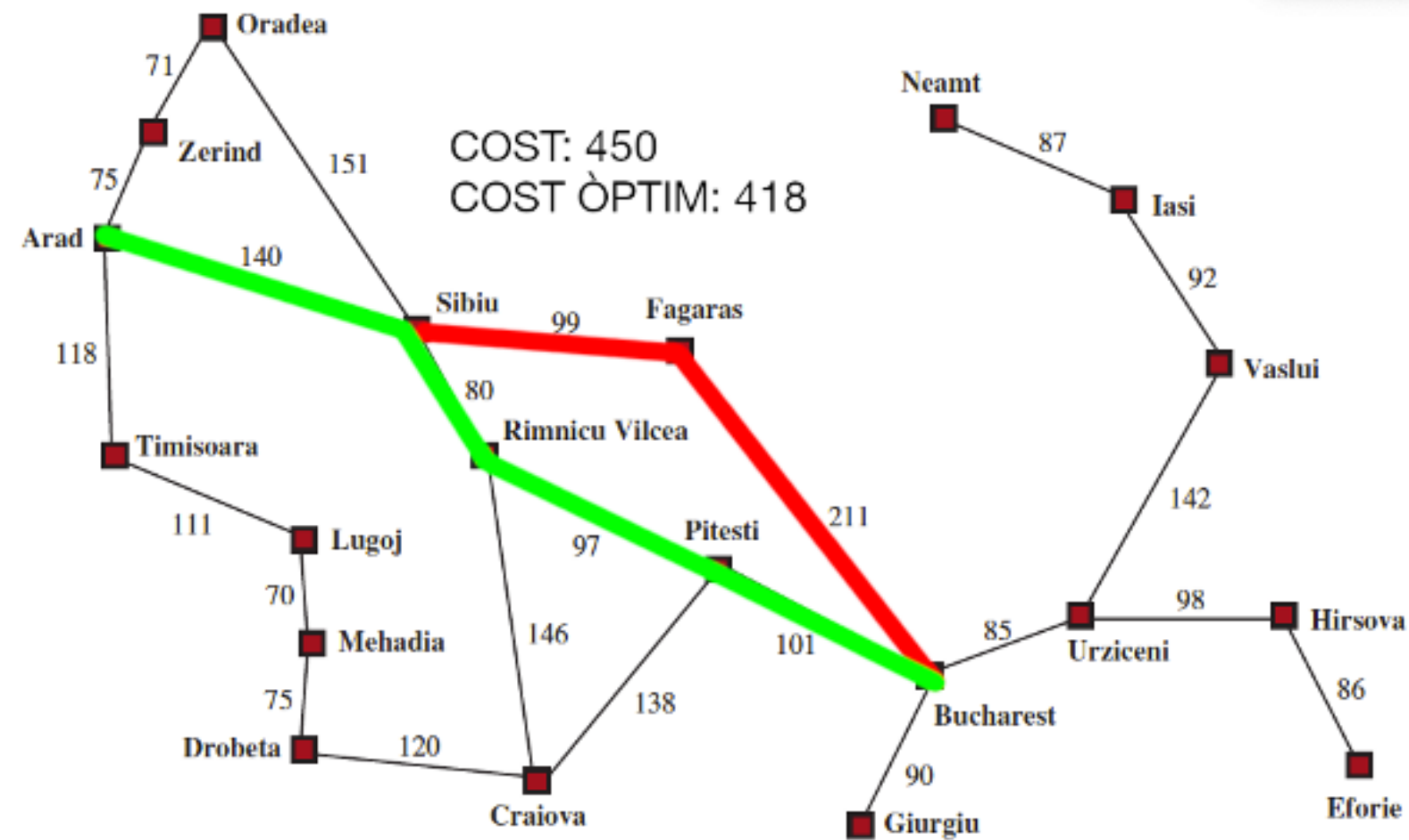
# Búsqueda voraç

- Si solament utilitzem la heurística per decidir quin estat de la frontera seguim:
- **Búsqueda voraç**
- Més eficient que la búsqueda de cost uniforme
- No garanteix trobar la solució òptima



# Búsqueda voraç

- En verd la ruta correcta i en roig la nostra
- Que podem fer perquè el nostre algorisme trobi la solució correcta?



# Búsqueda voraç

## Implementació

```
def cerca_voraç(estat_inicial):  
    """Cerca voraç en un problema."""  
    frontera = priority_queue([(0, estat_inicial)])  
    visitats = set()  
  
    while frontera:  
        cost_actual, estat = frontera.pop()  
        visitats.add(estat)  
  
        if es_solucio(estat):  
            return estat  
  
        for cost, sucesor in sucesors(estat):  
            if sucesor not in visitats:  
                frontera.append(heuristica(sucesor), sucesor)
```

# Búsqueda voraç

## Propietats

- **Completitud:** Sí
  - Si l'espai de cerca és finit, trobarà una solució en algun moment
- **Optimalitat:** No
  - La primera solució trobada no té perquè ser òptima
- **Complexitat temporal i espacial:**  $O(bm)$ 
  - On  $b$  és el factor de ramificació i  $m$  és la profunditat màxima de l'arbre

# **A\***

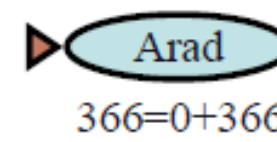
- L'algorisme **A\*** és una combinació de la búsqueda de cost uniforme i la búsqueda voraç
  - La búsqueda de cost uniforme ordena pel cost del camí o cost *cap enrere*: **g(n)**
  - La búsqueda voraç ordena pel cost de la heurística o cost *endavant*: **h(n)**
  - L'algorisme **A\*** ordena per la suma dels dos: **f(n) = g(n) + h(n)**
- **Garanteix trobar la solució òptima (si  $h(n)$  és admissible)**



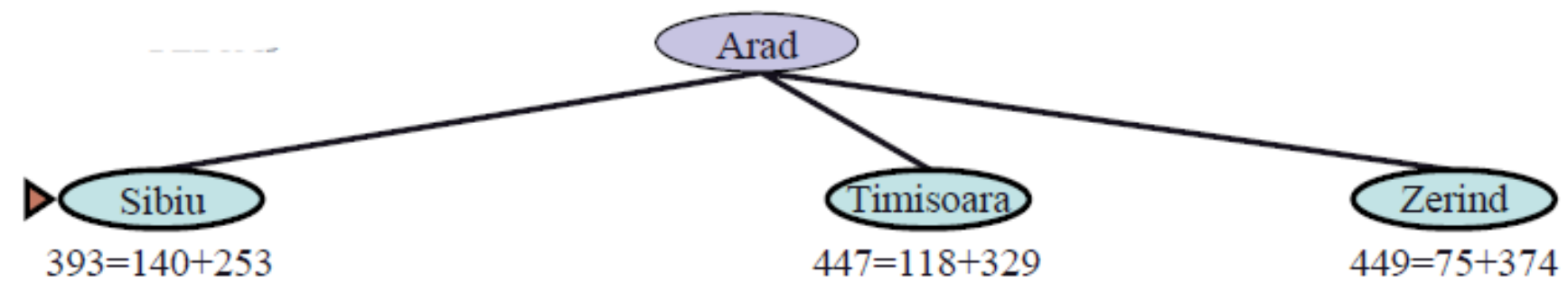
# A\*

## Exemple: Viatjar per Romania (I)

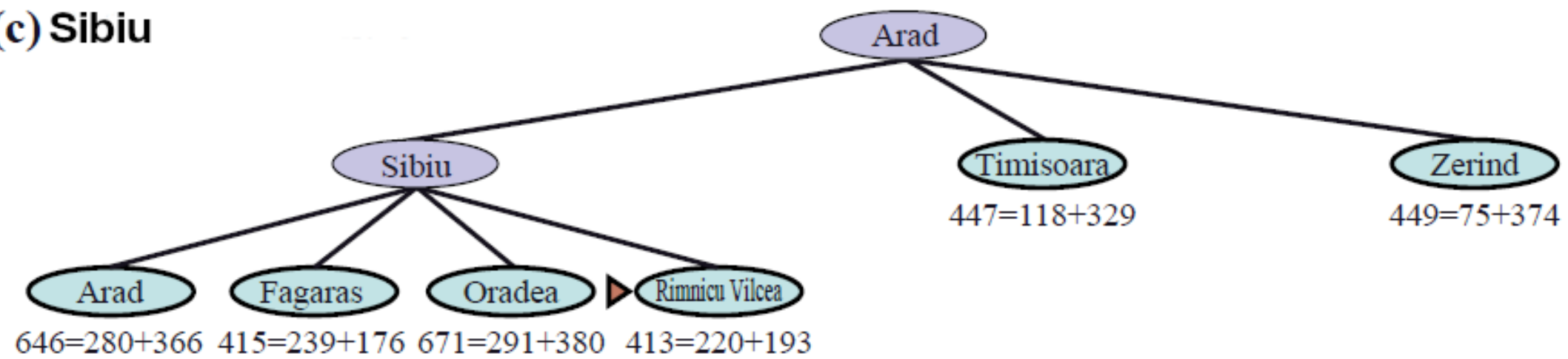
(a) Inicial



(b) Arad



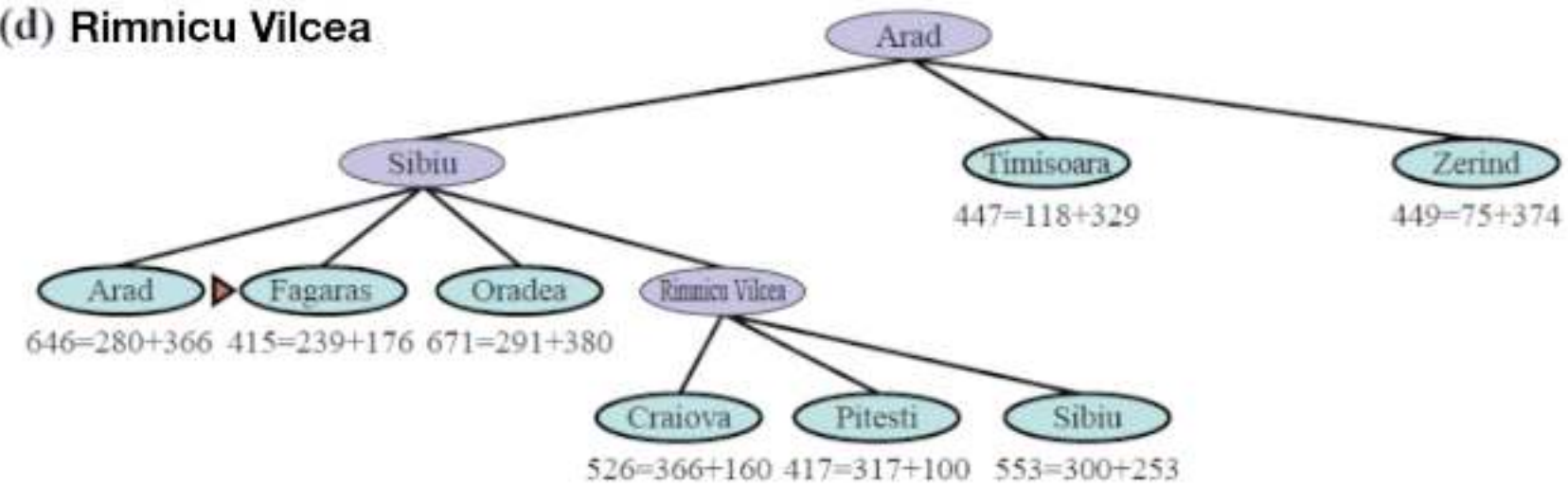
(c) Sibiu



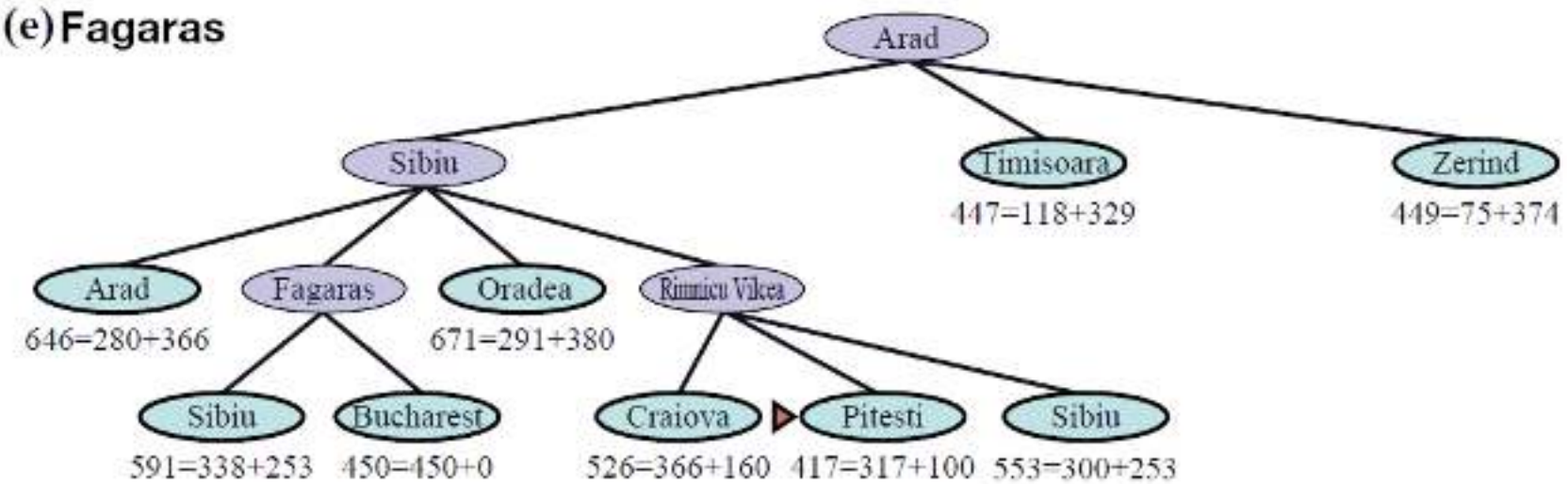
# A\*

## Exemple: Viatjar per Romania (II)

(d) Rimnicu Vilcea



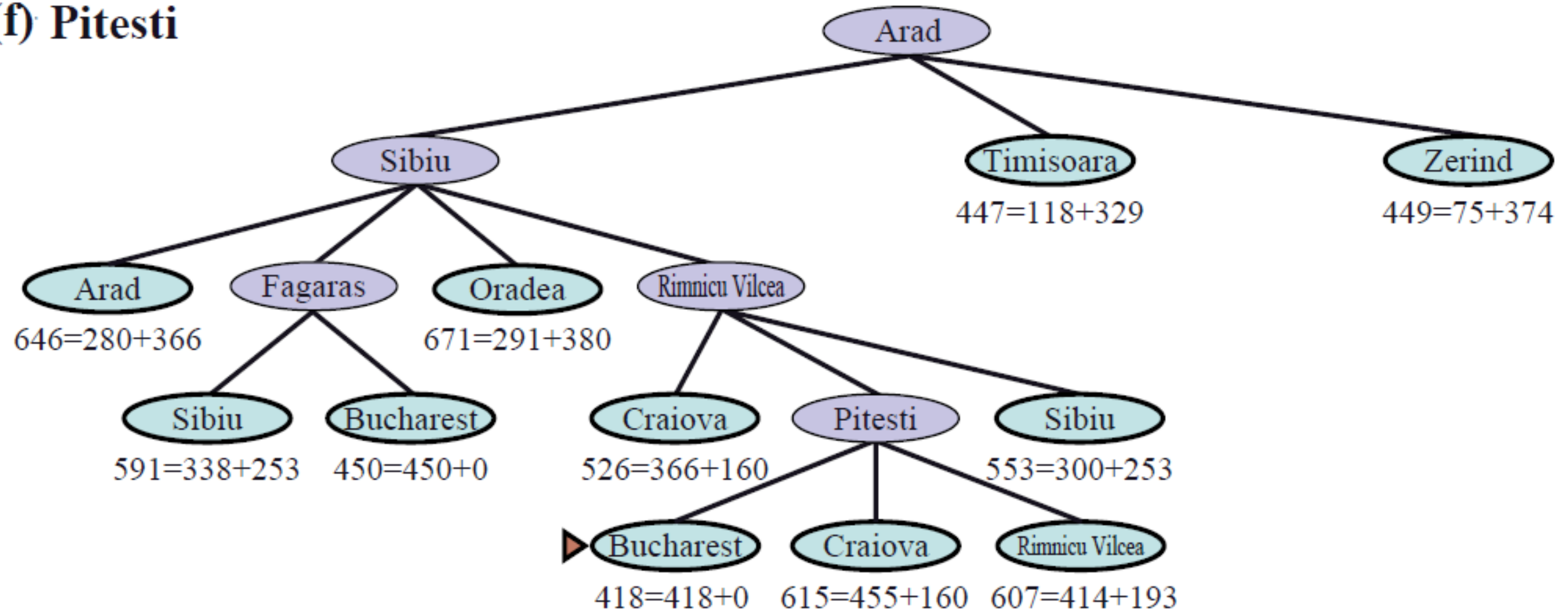
(e) Fagaras



# A\*

## Exemple: Viatjar per Romania (III)

(f) Pitesti



# A\*

## Implementació

```
def cerca_a_estrella(estat_inicial):  
    """Cerca A* en un problema."""  
    frontera = priority_queue([(0, estat_inicial)])  
    visitats = set()  
  
    while frontera:  
        cost_actual, estat = frontera.pop()  
        visitats.add(estat)  
  
        if es_solucio(estat):  
            return estat  
  
        for cost, sucesor in sucesors(estat):  
            if sucesor not in visitats:  
                frontera.append(cost + cost_actual, sucesor)
```

# A\*

## Propietats

- **Completitud:** Sí
- **Optimalitat:** Sí
- **Complexitat temporal i espacial:**  $O(b^d)$ 
  - On  $b$  és el factor de ramificació i  $d$  és la profunditat de la solució
- **Condició:**
  - Aquestes propietats es compleixen si la heurística és ***admissible***

# Heurístiques admissibles (I)

- Una heurística és **admissible** si:
  - No sobreestima el cost de la solució
  - És a dir, si el cost real de la solució és  $C$ , la heurística és admissible si  $h(n) \leq C$
- Si la heurística no és admissible:
  - L'algorisme **A\*** és equivalent a la búsqueda voraz
- Trobar una heurística admissible és un problema difícil.

# Construcció de heurístiques admissibles

## Exemple: Puzzle 8 (I)

- Técnica útil reduir el problema a un problema més senzill
- **Relaxació de les regles del joc**
  1. Permetre que les peces s'intercanviïn entre elles
  2. Permetre que les peces es moguin a qualsevol posició, si està buida
  3. Permetre que les peces es moguin a qualsevol posició, sense restriccions (1+2)

7	2	4
5		6
8	3	1

**Estat inicial**

	1	2
3	4	5
6	7	8

**Estat final**

# Construcció de heurístiques admissibles

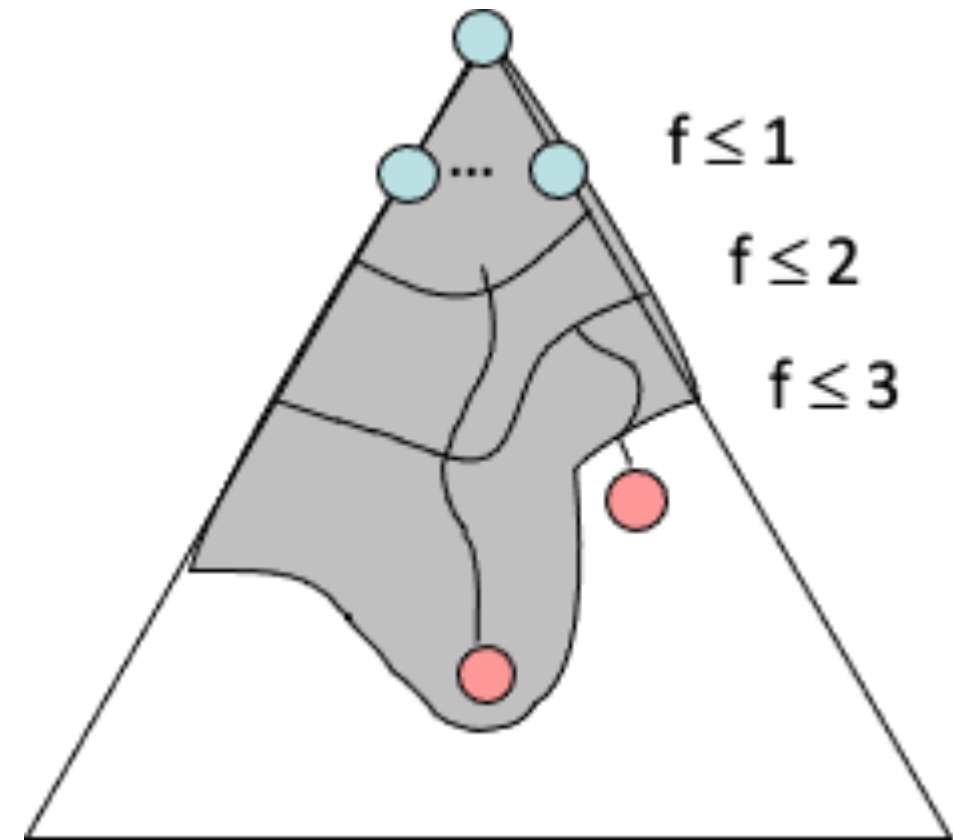
## Exemple: Puzzle 8 (II)

- La primera opció ens porta la heurística **distància manhattan**
  - Equival a un problema on hem de lliscar les peces fins a la seva posició.
  - Suma de les distàncies horitzontals i verticals de cada peça a la seva posició final
  - És admissible perquè no sobreestima el cost de la solució
- La tercera opció ens porta la heurística **nombre de peces fora de lloc**
  - Equival a un problema on hem de deixar directament en la seva posició.
  - Suma de les peces que no estan a la seva posició final
  - És admissible perquè no sobreestima el cost de la solució



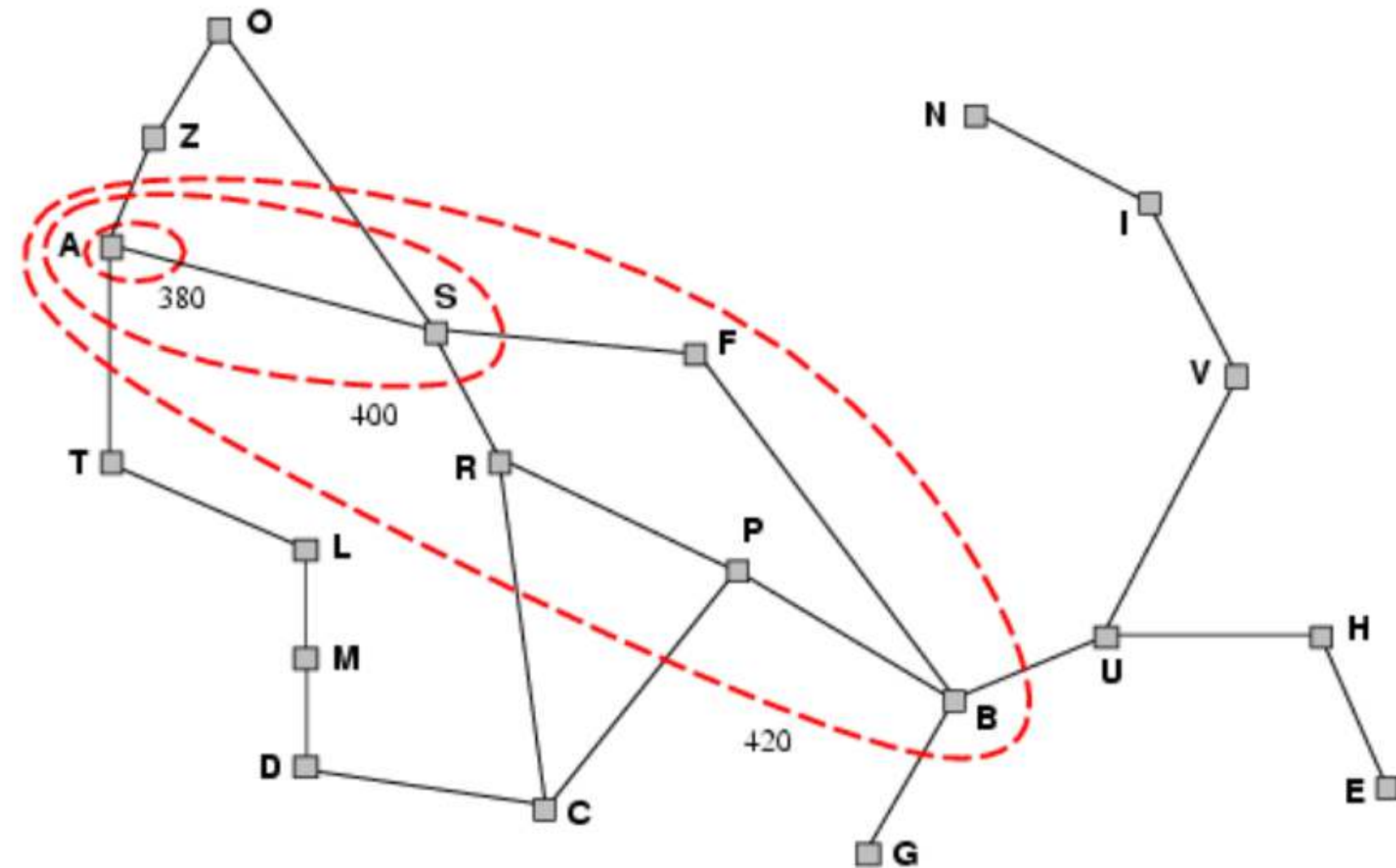
# Propietat Óptima de les heurístiques admissibles (I)

- Si tenim una ruta òptima fins a  $n_d$  amb cost  $g(n_d)$ .
  - $n'_g$  serà una ruta subòptima fins a  $n_d$  amb cost  $g(n'_d)$ , sent  $g(n'_d) > g(n_d)$ .
  - $n''$  serà una subpart de la ruta òptima desde la frontera
- Es possible que agafem  $n'_g$  abans de  $n_d$ ?.
  - **No**, perquè  $f(n'_g) > f(n_d)$
  - També,  $f(n'_g) > f(n'')$ , perquè la nostra heurística és admissible
  - Així,  $f(n'_g) > f(n'') > f(n_d)$
- Les subbrutes en la ruta òptima sempre seran més barates la ruta subòptima



## Propietat Óptima de les heurístiques admissibles (II)

- A\* explora els nodes en ordre creixent de  $f(n)$
- Va agregant, de forma gradual, corbes de nivell de grau  $f$
- Cada corba de nivell representa un conjunt de nodes amb un valor d' $f(n)$  inferior a un valor concret



## Propietat Óptima de les heurístiques admissibles (III)

- Si tenim dues heurístiques admissibles  $h_1$  i  $h_2$ , amb  $h_1(n) \leq h_2(n)$  per a tots els estats  $n$
- Llavors,  $h_2$  és més informativa que  $h_1$
- Per tant,  $h_2$  serà més eficient que  $h_1$
- Es per això que, preferirem l'heurística Manhattan a l'heurística de peces fora de lloc

7	2	4
5		6
8	3	1

**Estat inicial**

	1	2
3	4	5
6	7	8

**Estat final**

# Limitacions de l'algorisme $A^*$

- L'algorisme  $A^*$  és òptim i una millora respecte a la búsqueda de cost uniforme
- Però, l'algorisme  $A^*$  té dues limitacions:
  - **Espai de memòria:** L'espai de memòria necessari pot ser molt gran
  - **Temps d'execució:** El temps d'execució pot ser molt gran
- Per això, s'han desenvolupat variants de l'algorisme  $A^*$  que intenten millorar aquestes limitacions
- En aquesta unitat veurem dues:
  - $A^*$  de profunditat iterativa
  - $A^*$  ponderat

# **A\* de profunditat iterativa**

- L'algorisme **A\*** de profunditat iterativa és una variant de l'algorisme **A\***
- Molt semblant a l'algorisme de profunditat iterativa
  - Utilitza la funció  $f(n)$  per tallar, en compte de la profunditat
- Ens permet reduir l'espai de memòria necessari
  - A costa de tindre que visitar alguns nodes més d'una vegada

# A\* de profunditat iterativa

## Implementació (I)

```
def cerca_a_limitada(estat_inicial, l):  
    """Cerca A* limitada en un problema."""  
    frontera = priority_queue([(0, estat_inicial)])  
    visitats = set()  
  
    while frontera:  
        cost_actual, estat = frontera.pop()  
        visitats.add(estat)  
  
        if es_solucio(estat):  
            return estat  
  
        for cost, sucesor in successors(estat):  
            if sucesor not in visitats and cost_actual + cost < l:  
                frontera.append(cost_actual + cost, sucesor)
```

# A\* de profunditat iterativa

## Implementació (II)

```
def cerca_a_iterativa(estat_inicial):  
    """Cerca A* iterativa en un problema."""  
    l = 0  
    while True:  
        solucio = cerca_a_limitada(estat_inicial, l)  
        if solucio is not None:  
            return solucio  
        l += 1
```

# A\* Ponderat

## Definició

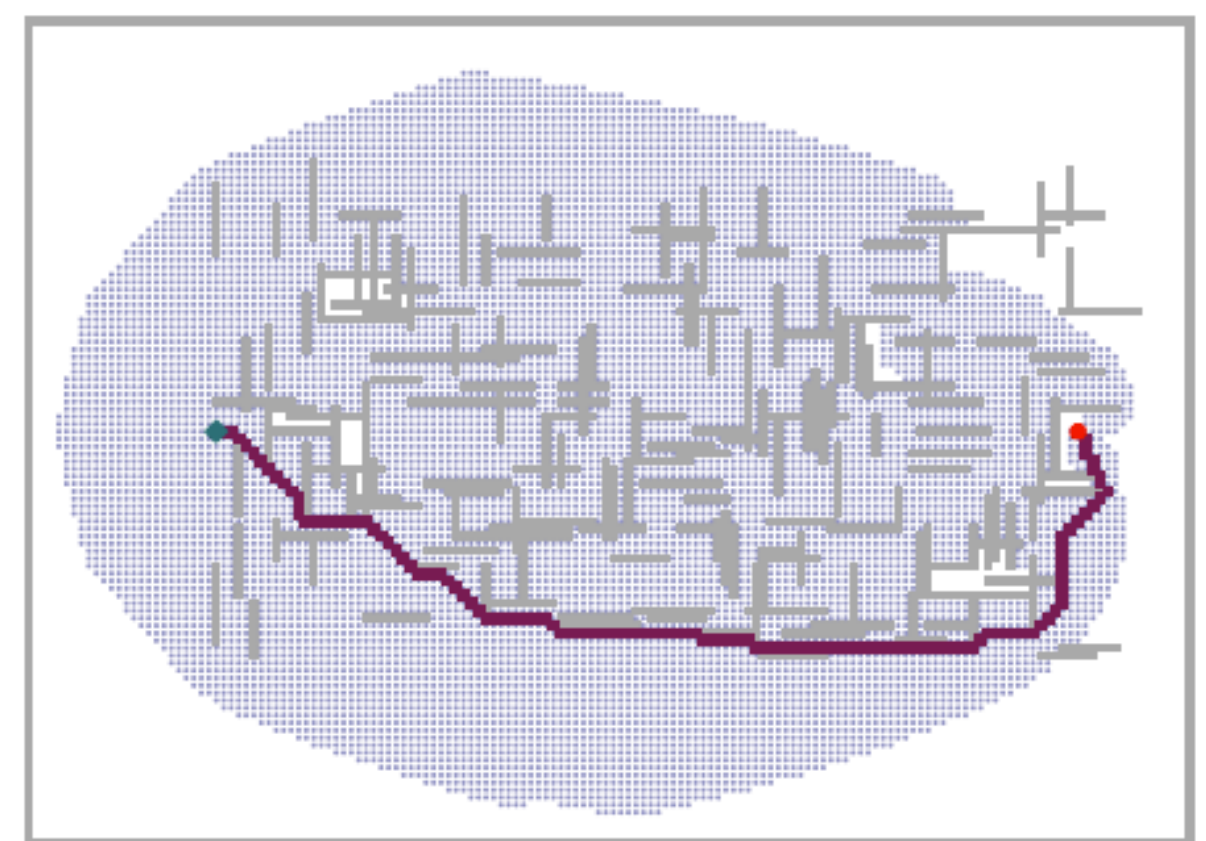
- L'algorisme **A\*** ponderat és una variant de l'algorisme **A\***
- Es defineix un factor de ponderació  $\epsilon$  que determina el pes de la heurística
- L'algorisme **A\*** ponderat ordena per  $f(n) = g(n) + \epsilon h(n)$
- Si  $\epsilon = 1$ , l'algorisme **A\*** ponderat és equivalent a l'algorisme **A\***
- Si  $\epsilon > 1$ , l'algorisme **A\*** ponderat és s'apropa a la búsqueda voraç



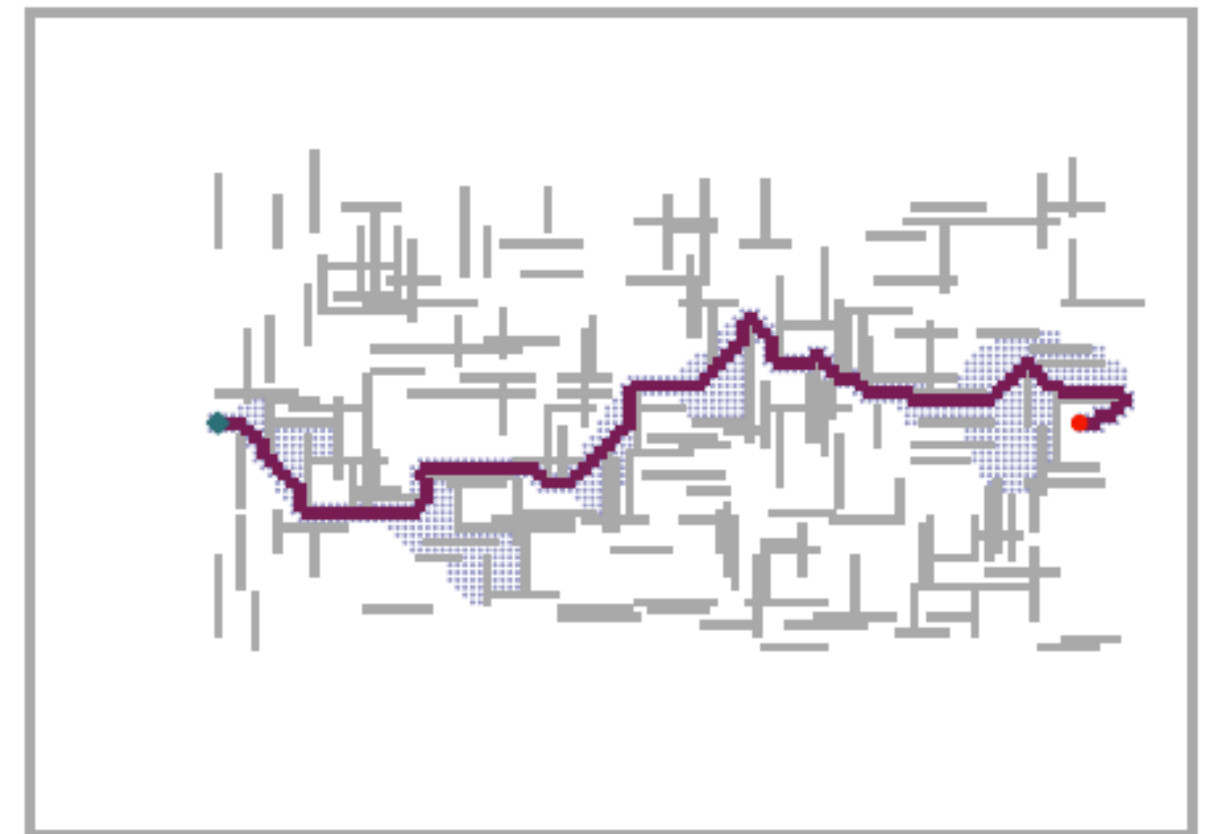
# A\* Ponderat

## Utilitat

- L'algorisme **A\*** ponderat és útil per:
  - Reduir el cost de l'espai de memòria
  - Reduir el cost de l'espai de temps
  - A costa d'una solució no tan òptima
- En l'exemple de la dreta en una  $W = 2$  (la b)
  - S'estudien 7 vegades menys estats
  - Per una solució un 5% menys eficient



(a)



(b)

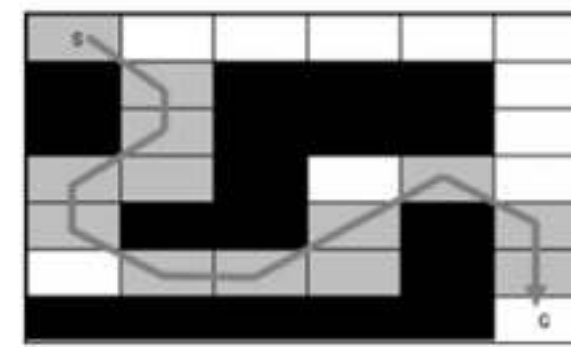
# A\* Ponderat

## Implementació

```
def cerca_a_ponderat(estat_inicial, epsilon):  
    """Cerca A* ponderat en un problema."""  
    frontera = priority_queue([(0, estat_inicial)])  
    visitats = set()  
  
    while frontera:  
        cost_actual, estat = frontera.pop()  
        visitats.add(estat)  
  
        if es_solucio(estat):  
            return estat  
  
        for cost, sucesor in sucesors(estat):  
            if sucesor not in visitats:  
                frontera.append(cost_actual + epsilon * cost, sucesor)
```

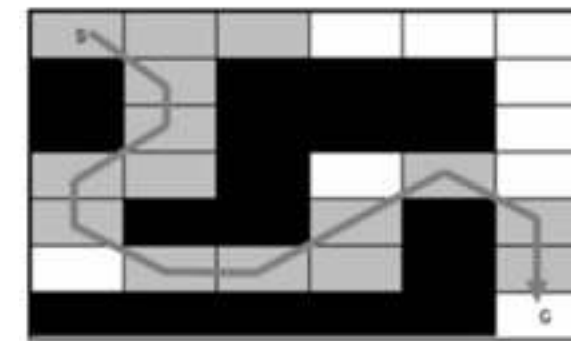
# Anytime A\*

- Podem aprofitar l'algorisme **A\*** ponderat per construir un algorisme **Anytime A\***
- Busquem el camí òptim amb un  $\epsilon$  gran
- Anem reduint  $\epsilon$  fins a que  $\epsilon = 1$
- Així, obtenim una bona solució en un temps raonable
- Si tenim temps, podem seguir buscant una solució millor, fins arribar a la solució òptima



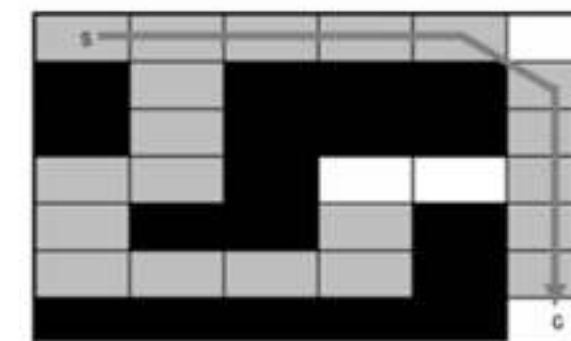
$\epsilon = 2$

13 node expansions  
Solution length: 12



$\epsilon = 1.5$

15 node expansions  
Solution length: 12



$\epsilon = 1$

20 node expansions  
Solution length: 10

# Anytime A\*

## Implementació

```
def cerca_anytime_a(estat_inicial):  
    """Cerca Anytime A* en un problema."""  
    epsilon = 100  
    while epsilon > 1:  
        solucio = cerca_a_ponderat(estat_inicial, epsilon)  
        yield solucio  
        epsilon /= 2
```

# Demostració de búsqüedes

## Pac-Man

<https://www.youtube.com/watch?v=2XjzjAfGWzY>



Voraç



Cost Uniforme



Cost Uniforme