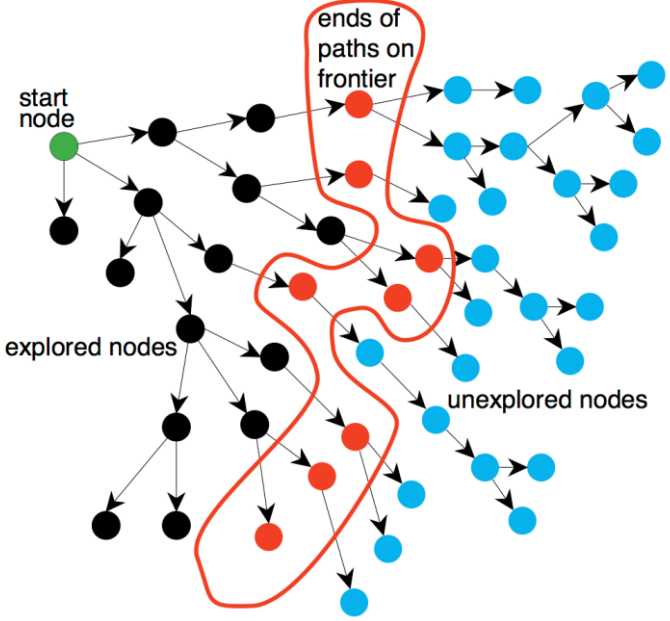
ΤΕΧΝΗΤΗ ΝΟΗΜΟΣΥΝΗ

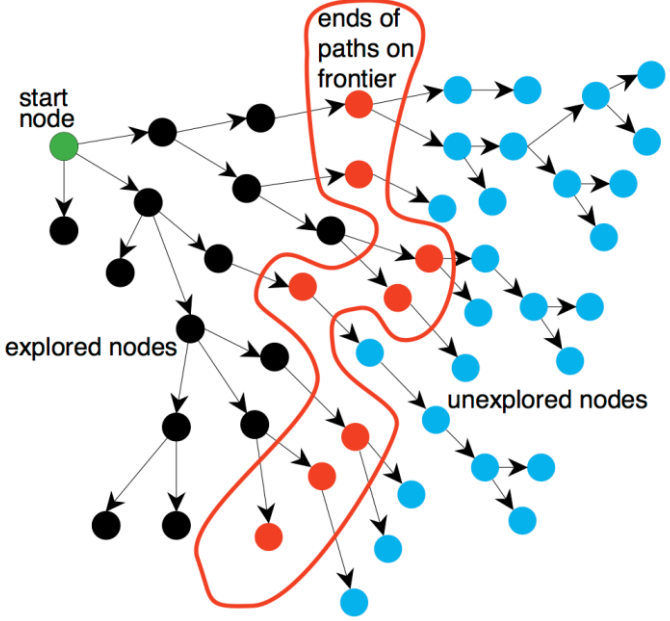
ΥΛΟΠΟΙΗΣΗ ΑΛΓΟΡΙΘΜΩΝ ΑΝΑΖΗΤΗΣΗΣ

Γενικός Αλγόριθμος Αναζήτησης



Deterministic-Search(Σ, s_0, g)
  Frontier ← {(⟨⟩, s_0)}
  Expanded ← ∅
  while *Frontier* ≠ ∅ do
    select a node ν = (π, s) ∈ *Frontier* (*i*)
    remove ν from *Frontier*
    add ν to *Expanded*
    if *s* satisfies *g* then return *π*
    Children ← {(*π.a*, *γ*(*s*,*a*)) | *s* satisfies pre(*a*)}
    prune 0 or more nodes from
      Children, *Frontier*, *Expanded* (*ii*)
    Frontier ← *Frontier* ∪ *Children*
  return failure

Γενικός Αλγόριθμος Αναζήτησης



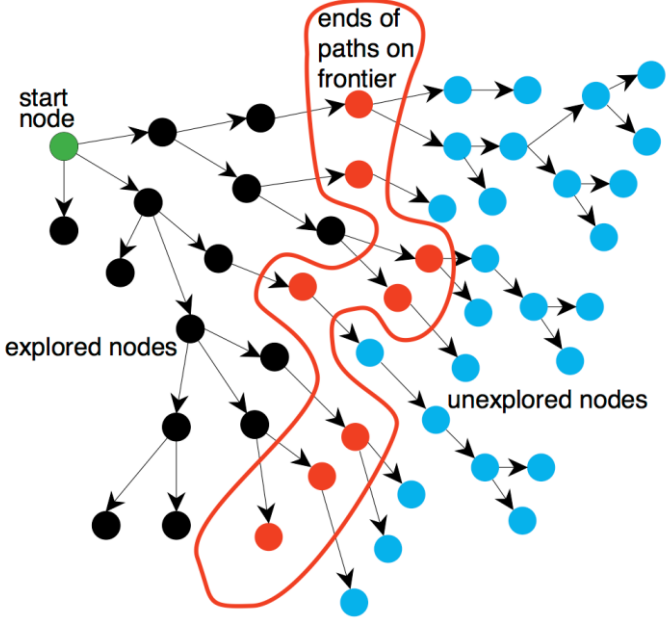
Αρχική Κατάσταση

Τελική Κατάσταση ?

Κόμβος Αναζήτησης
<μονοπάτι, κατάσταση>

Deterministic-Search(Σ, s_0, *g*)
  Frontier ← {(⟨⟩, s_0)}
  Expanded ← ∅
  while *Frontier* ≠ ∅ do
    select a node v = (π, s) ∈ *Frontier* (*i*)
    remove v from *Frontier*
    add v to *Expanded*
    if *s* satisfies *g* then return *π*
    Children ← {(*π.a*, *γ*(*s*,*a*)) | *s* satisfies pre(*a*)}
    prune 0 or more nodes from
      Children, *Frontier*, *Expanded* (*ii*)
    Frontier ← *Frontier* ∪ *Children*
  return failure

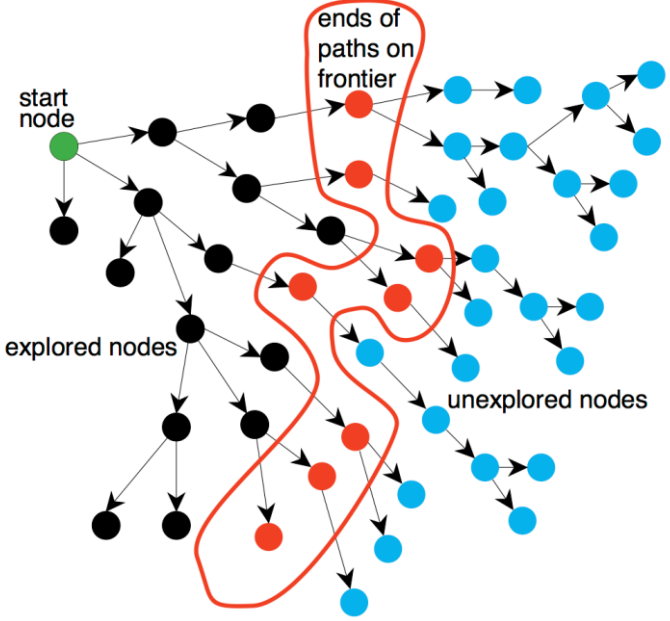
Γενικός Αλγόριθμος Αναζήτησης



Deterministic-Search(Σ, s_0, g)
  Frontier ← {(⟨⟩, s_0)}
  Expanded ← ∅
  while *Frontier* ≠ ∅ do
    select a node ν = (π, s) ∈ *Frontier* (*i*)
    remove ν from *Frontier*
    add ν to *Expanded*
    if *s* satisfies *g* then return *π*
    Children ← {(*π*.*a*, *γ*(*s*,*a*)) | *s* satisfies pre(*a*)}
    prune 0 or more nodes from
      Children, *Frontier*, *Expanded* (*ii*)
    Frontier ← *Frontier* ∪ *Children*
  return failure

Κατάσταση

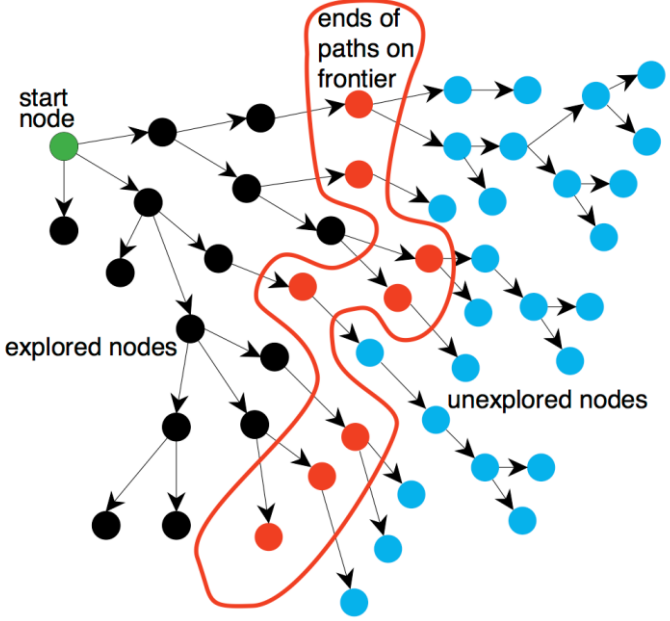
Γενικός Αλγόριθμος Αναζήτησης



Deterministic-Search(Σ, s_0, g)
  Frontier ← {(⟨⟩, s_0)}
  Expanded ← ∅
  while *Frontier* ≠ ∅ do
    select a node ν = (π, s) ∈ *Frontier* (*i*)
    remove ν from *Frontier*
    add ν to *Expanded*
    if *s* satisfies *g* then return *π*
    Children ← {(*π*.*a*, *γ*(*s*,*a*)) | *s* satisfies pre(*a*)}
    prune 0 or more nodes from
      Children, *Frontier*, *Expanded* (*ii*)
    Frontier ← *Frontier* ∪ *Children*
  return failure

Κατάσταση
αναπαράσταση

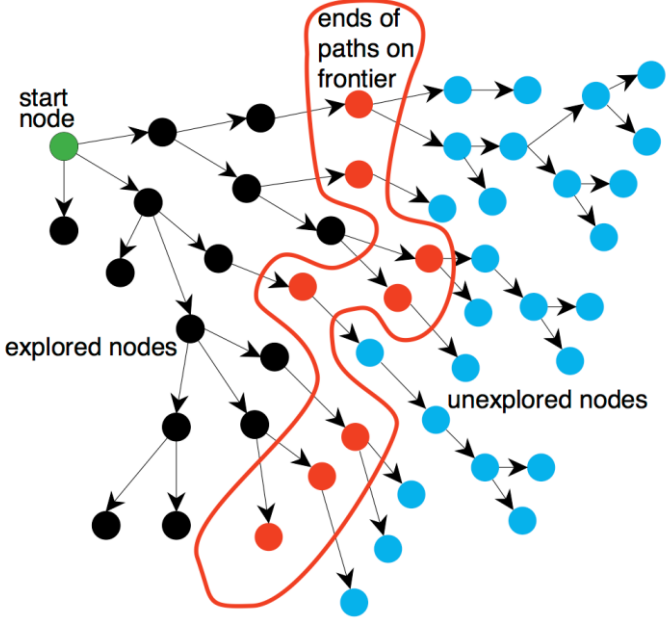
Γενικός Αλγόριθμος Αναζήτησης



Deterministic-Search(Σ, s_0, g)
  Frontier ← {(⟨⟩, s_0)}
  Expanded ← ∅
  while *Frontier* ≠ ∅ do
    select a node ν = (π, s) ∈ *Frontier* (*i*)
    remove ν from *Frontier*
    add ν to *Expanded*
    if *s* satisfies *g* then return *π*
    Children ← {(*π*.*a*, *γ*(*s*,*a*)) | *s* satisfies pre(*a*)}
    prune 0 or more nodes from
      Children, *Frontier*, *Expanded* (*ii*)
    Frontier ← *Frontier* ∪ *Children*
  return failure

Κατάσταση
αναπαράσταση
μονοπάτι από ρίζα

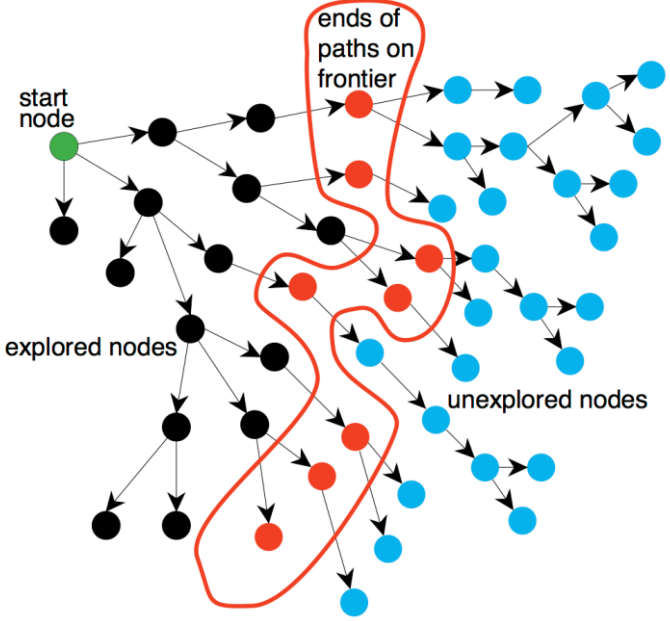
Γενικός Αλγόριθμος Αναζήτησης



Deterministic-Search(Σ, s_0, g)
  Frontier ← {(⟨⟩, s_0)}
  Expanded ← ∅
  while *Frontier* ≠ ∅ do
    select a node v = (π, s) ∈ *Frontier* (*i*)
    remove v from *Frontier*
    add v to *Expanded*
    if *s* satisfies *g* then return *π*
    Children ← {(*π.a*, *γ*(*s*,*a*)) | *s* satisfies pre(*a*)}
    prune 0 or more nodes from
      Children, *Frontier*, *Expanded* (*ii*)
    Frontier ← *Frontier* ∪ *Children*
  return failure

Κατάσταση
αναπαράσταση
μονοπάτι από ρίζα
συνάρτηση == (σύγκριση καταστάσεων)

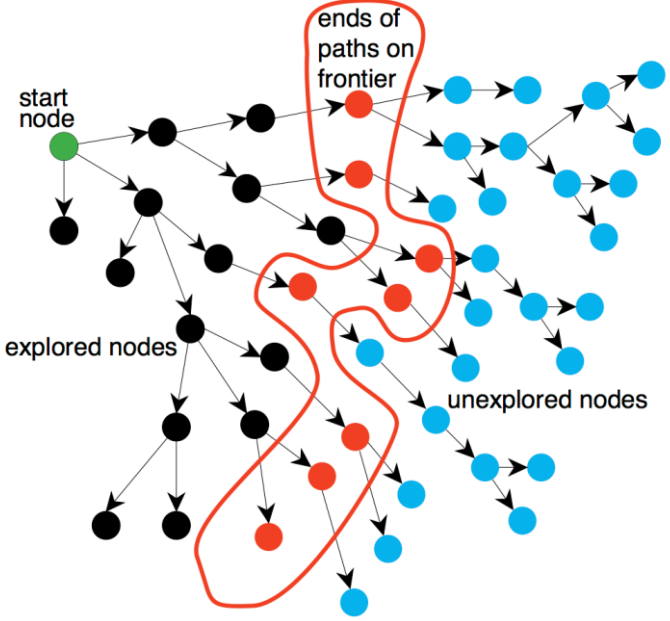
Γενικός Αλγόριθμος Αναζήτησης



Deterministic-Search(Σ, s_0, g)
  Frontier ← {(⟨⟩, s_0)}
  Expanded ← ∅
  while *Frontier* ≠ ∅ do
    select a node ν = (π, s) ∈ *Frontier* (*i*)
    remove ν from *Frontier*
    add ν to *Expanded*
    if *s* satisfies *g* then return *π*
    Children ← {(*π.a*, *γ*(*s*,*a*)) | *s* satisfies pre(*a*)}
    prune 0 or more nodes from
      Children, *Frontier*, *Expanded* (*ii*)
    Frontier ← *Frontier* ∪ *Children*
  return failure

Κατάσταση
- αναπαράσταση
- μονοπάτι από ρίζα
- συνάρτηση == (σύγκριση καταστάσεων)
- τελεστές μετάβασης (κινήσεις)

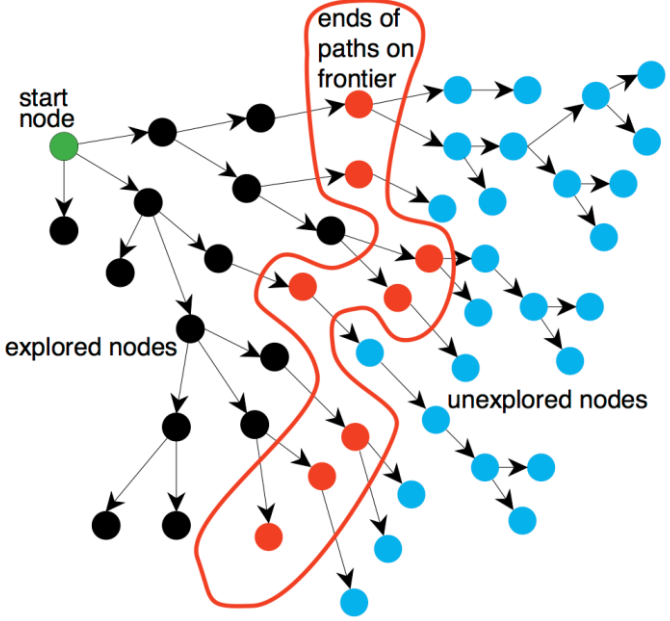
Γενικός Αλγόριθμος Αναζήτησης



Deterministic-Search(Σ, s_0, g)
  Frontier ← {(⟨⟩, s_0)}
  Expanded ← ∅
  while *Frontier* ≠ ∅ do
    select a node ν = (π, s) ∈ *Frontier* (*i*)
    remove ν from *Frontier*
    add ν to *Expanded*
    if *s* satisfies *g* then return *π*
    Children ← {(*π*.*a*, *γ*(*s*,*a*)) | *s* satisfies pre(*a*)}
    prune 0 or more nodes from
      Children, *Frontier*, *Expanded* (*ii*)
    Frontier ← *Frontier* ∪ *Children*
  return failure

Κατάσταση
- αναπαράσταση
- μονοπάτι από ρίζα
- συνάρτηση == (σύγκριση καταστάσεων)
- τελεστές μετάβασης (κινήσεις)
- έλεγχος εφαρμοσιμότητας τελεστή

Γενικός Αλγόριθμος Αναζήτησης

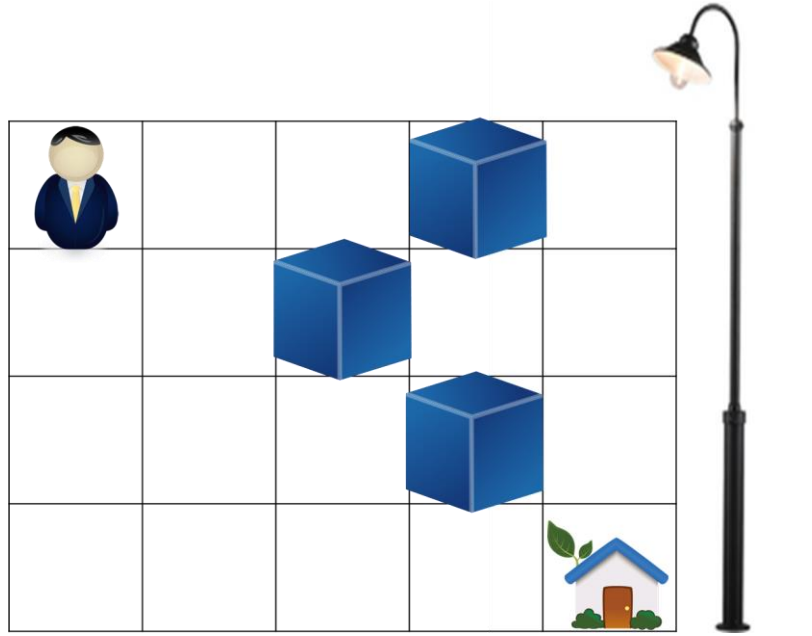


Deterministic-Search(Σ, s_0, g)
  Frontier ← {(⟨⟩, s_0)}
  Expanded ← ∅
  while *Frontier* ≠ ∅ do
    select a node v = (π, s) ∈ *Frontier* (*i*)
    remove v from *Frontier*
    add v to *Expanded*
    if *s* satisfies *g* then return *π*
    Children ← {(*π.a*, *γ*(*s*,*a*)) | *s* satisfies pre(*a*)}
    prune 0 or more nodes from
      Children, *Frontier*, *Expanded* (*ii*)
    Frontier ← *Frontier* ∪ *Children*
  return failure

Κατάσταση
- αναπαράσταση
- μονοπάτι από ρίζα
- συνάρτηση == (σύγκριση καταστάσεων)
- τελεστές μετάβασης
- έλεγχος εφαρμοσιμότητας τελεστή
- επέκταση κατάστασης

Παράδειγμα 1

- Έστω ένα πρόβλημα λαβυρίνθου
  - Ο κόσμος αποτελείται από ένα πλέγμα N*M
  - Υπάρχουν εμπόδια σε κάποια κελιά
  - Στο περιβάλλον υπάρχει και ένα φως το οποίο ανάβει/σβήνει

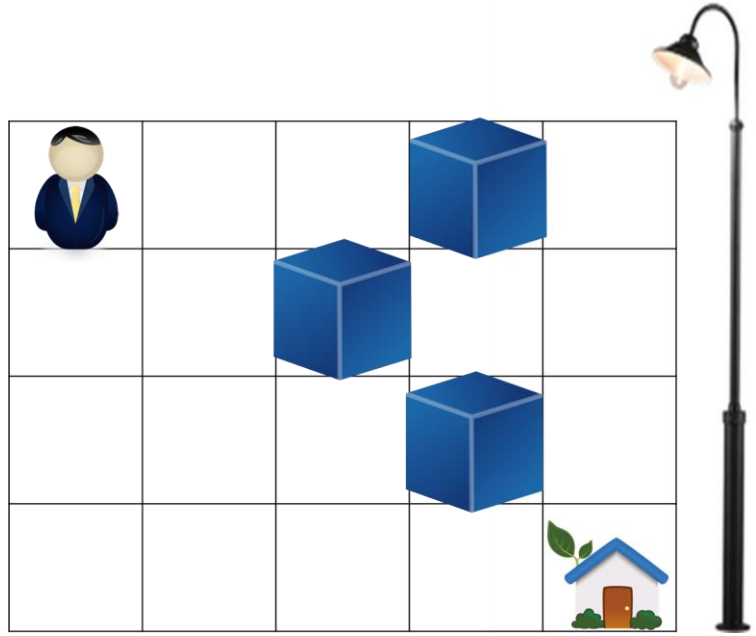


Κατάσταση

αναπαράσταση
μονοπάτι από ρίζα
συνάρτηση == (σύγκριση καταστάσεων)
τελεστές μετάβασης
έλεγχος εφαρμοσιμότητας
επέκταση κατάστασης

Παράδειγμα 1

```
class Maze
{
    public:
        Maze();
        Maze(int X, int Y, bool lights);
        void setFree(int i, int j, bool f);
        int getY();
        int getX();
        bool isFree(int x,int y);
        void setX(int x);
        void setY(int y);
        string toString () const;
        string getPath();
        …
    private:
        int robX,robY;
        bool lights;
        bool free[WIDTH][HEIGHT];
        string actionName;
        Maze *prev;
};
```



Κατάσταση

αναπαράσταση

μονοπάτι από ρίζα

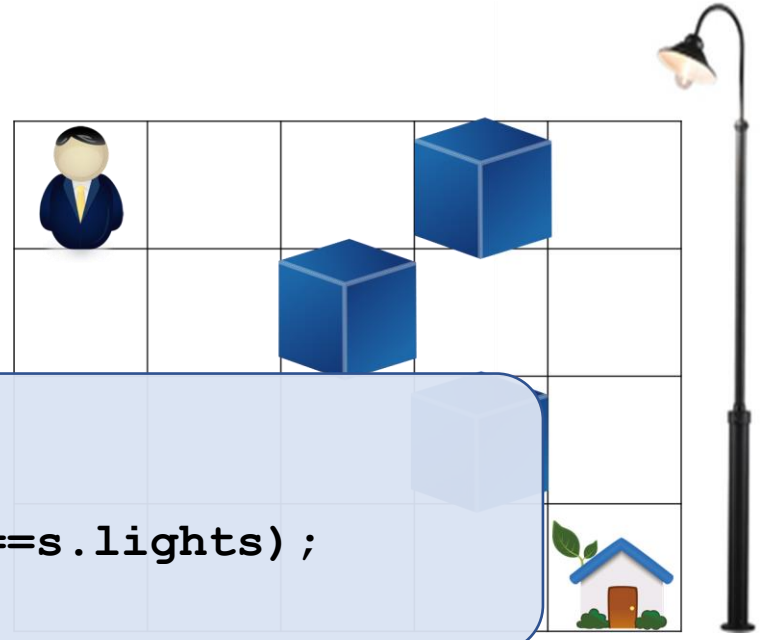
συνάρτηση == (σύγκριση καταστάσεων)

τελεστές μετάβασης

έλεγχος εφαρμοσιμότητας

επέκταση κατάστασης

Παράδειγμα 1

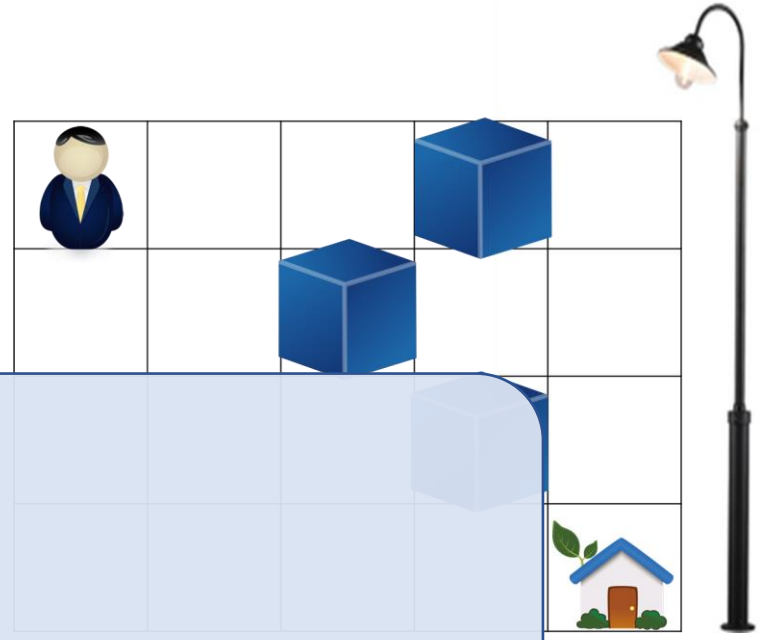


```
class Maze
{
bool Maze::operator==(const Maze& s) const
{
    return (robX==s.robX && robY==s.robY && lights==s.lights);
}
        int getX();
        bool isFree(int x,int y);
        void setX(int x);
        void setY(int y);
        string toString () const;
        string getPath();
        …
    private:
        int robX,robY;
        bool free[WIDTH][HEIGHT];
        bool lights;
        vector <string> path;
};
```

Κατάσταση

- αναπαράσταση
- μονοπάτι από ρίζα
- συνάρτηση == (σύγκριση καταστάσεων)
- τελεστές μετάβασης
- έλεγχος εφαρμοσιμότητας
- επέκταση κατάστασης

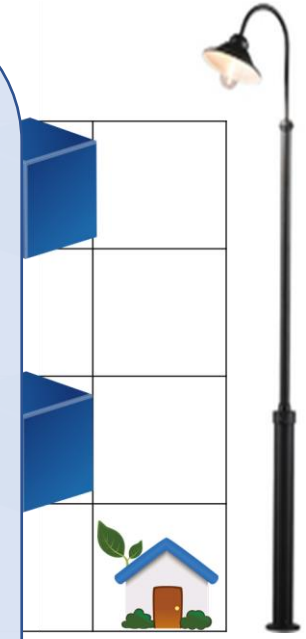
Παράδειγμα 1



```
class Maze
{
    bool turnOn(Maze &n);
    bool turnOff (Maze &n);
    bool goUp(Maze &n);
    bool goDown(Maze &n);
    bool goLeft(Maze &n);
    bool goRight(Maze &n);
    void setX(int x);
    void setY(int y);
    string toString () const;
    string getPath();
    …
  private:
    int robX,robY;
    bool free[WIDTH][HEIGHT];
    bool lights;
    vector <string> path;
};
```

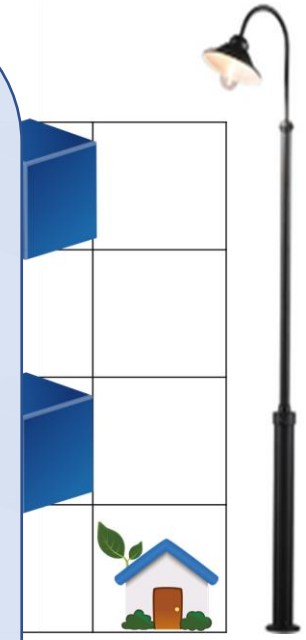
Κατάσταση
αναπαράσταση
μονοπάτι από ρίζα
συνάρτηση == (σύγκριση καταστάσεων)
τελεστές μετάβασης
έλεγχος εφαρμοσιμότητας
επέκταση κατάστασης γ(s,α)

```
…
bool goUp(Maze &n)
{
 if (getY()>0 && isFree(getX(),getY()-1))
    {
        n=*this;
        n.setY(n.getY()-1);
        n.setActionName("Up");
        n.setPrevious(this);
        return true;
    }
    return false;
}
…
        void setY(int y);
        string toString () const;
        string getPath();
        …
    private:
        int robX,robY;
        bool free[WIDTH][HEIGHT];
        bool lights;
        vector <string> path;
};
```



συνάρτηση == (σύγκριση καταστάσεων)
τελεστές μετάβασης
έλεγχος εφαρμοσιμότητας
επέκταση κατάστασης

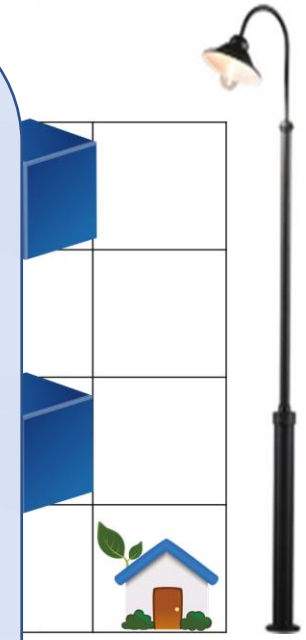
```
…
bool Maze::goLeft(Maze &n)
{
 if (getX()>0 && isFree(getX()-1,getY()))
    {
        n=*this;
        n.setX(n.getX()-1);
        n.setActionName("Left");
        n.setPrevious(this);
        return true;
    }
    return false;
}
…
```



```
        void setY(int y);
        string toString () const;
        string getPath();
        …
    private:
        int robX,robY;
        bool free[WIDTH][HEIGHT];
        bool lights;
        vector <string> path;
};
```

συνάρτηση == (σύγκριση καταστάσεων)
τελεστές μετάβασης
έλεγχος εφαρμοσιμότητας
επέκταση κατάστασης

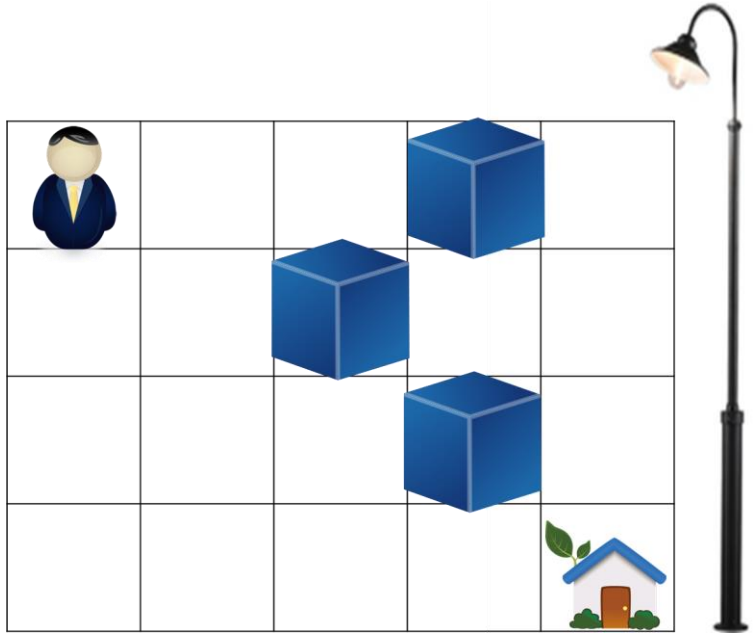
```
…
bool Maze::turnOn(Maze &n)
{
 if (!lights)
    {
        n=*this;
        n.lights=true;
        n.setActionName("SwitchOn");
        n.setPrevious(this);
        return true;
    }
    return false;
}
…
```



συνάρτηση == (σύγκριση καταστάσεων)
τελεστές μετάβασης
έλεγχος εφαρμοσιμότητας
επέκταση κατάστασης

```
        void setY(int y);
        string toString () const;
        string getPath();
        …
    private:
        int robX,robY;
        bool free[WIDTH][HEIGHT];
        bool lights;
        vector <string> path;
};
```

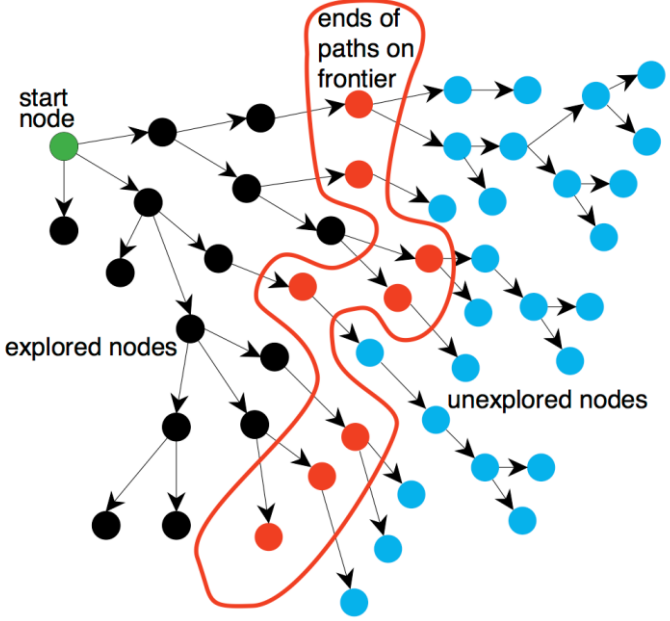
```
vector <Maze *> Maze::expand()
{
    vector <Maze *> children;
    Maze *child;
    child = new Maze(*this);
    if (goUp(*child))
        children.push_back(child);
    else
        delete child;

    child = new Maze(*this);
    if (goDown(*child))
        children.push_back(child);
    else
        delete child;
    …

    child = new Maze(*this);
    if (turnOff(*child))
        children.push_back(child);
    else
        delete child;

    return children;
}
```



Κατάσταση

αναπαράσταση
μονοπάτι από ρίζα
συνάρτηση == (σύγκριση καταστάσεων)
τελεστές μετάβασης
έλεγχος εφαρμοσιμότητας
επέκταση κατάστασης

Γενικός Αλγόριθμος Αναζήτησης



Deterministic-Search(Σ, s_0, g)
  Frontier ← {(⟨⟩, s_0)}
  Expanded ← ∅
  while *Frontier* ≠ ∅ do
    select a node ν = (π, s) ∈ *Frontier* (*i*)
    remove ν from *Frontier*
    add ν to *Expanded*
    if *s* satisfies *g* then return *π*
    Children ← {(*π.a*, *γ*(*s*,*a*)) | *s* satisfies pre(*a*)}
    prune 0 or more nodes from
      Children, *Frontier*, *Expanded* (*ii*)
    Frontier ← *Frontier* ∪ *Children*
  return failure

BFS

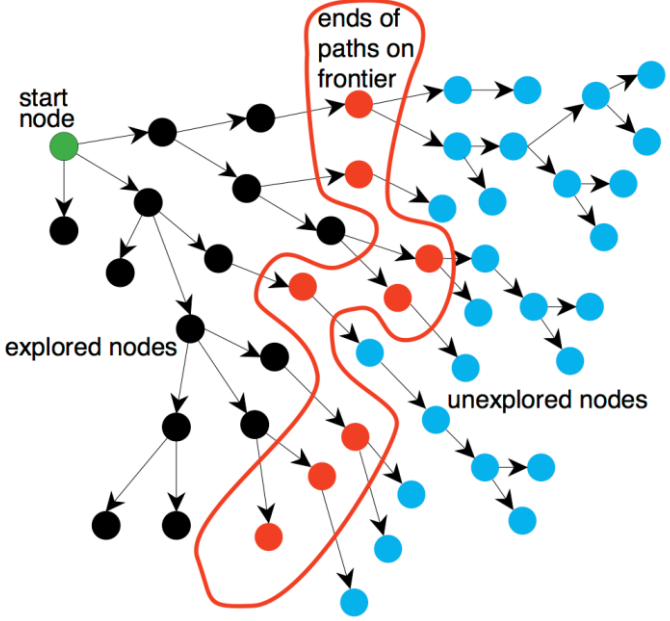
(*i*): select (*π*,*s*) ∈ *Frontier*
with smallest length(*π*)

- tie-breaking rule: select oldest

(*ii*): remove every
(*π*,*s*) ∈ *Children* ∪ *Frontier*
such that *s* is in *Expanded*

- Thus expand states at most once

BFS



Deterministic-Search(Σ, s_0, g)
  Frontier ← {($\langle\rangle$, s_0)}
  Expanded ← ∅
  while *Frontier* ≠ ∅ do
    select a node v = (π, s) ∈ *Frontier* (*i*)
    remove v from *Frontier*
    add v to *Expanded*
    if *s* satisfies *g* then return π
    Children ← {(π.*a*, γ(*s*,*a*)) | *s* satisfies pre(*a*)}
    prune 0 or more nodes from
      Children, *Frontier*, *Expanded* (*ii*)
    Frontier ← *Frontier* ∪ *Children*
  return failure

queue

vector, unordered_map

(*i*): select (π,*s*) ∈ *Frontier*
with smallest length(π)

- tie-breaking rule: select oldest

(*ii*): remove every
(π,*s*) ∈ *Children* ∪ *Frontier*
such that *s* is in *Expanded*

- Thus expand states at most once

Λύση με vector<Maze>

```
Maze *BFS(Maze *initial,Maze *goal)
{
    queue<Maze *> agenda;
    vector <Maze> closed;
    agenda.push(initial);
    while (agenda.size()>0)
    {
        Maze *s = agenda.front();
        agenda.pop();
        if (find(closed.begin(), closed.end(), *s)==closed.end())
        {
            if (*s==*goal)
                return s;
            closed.push_back(*s);
            vector<Maze *> children =s->expand();
            for (unsigned int i=0;i<children.size();i++)
                if (find(closed.begin(), closed.end(), *children[i])==closed.end())
                    agenda.push(children[i]);
        }
    }
    return nullptr;
}
```

Δημιουργώ το Μέτωπο

Δημιουργώ το Κλειστό Σύνολο

Τοποθετώ την αρχική κατάσταση στο Μέτωπο

Όσο το Μέτωπο δεν είναι άδειο

Παίρνω τη πρώτη κατάσταση από το Μέτωπο

Ελέγχω αν η τρέχουσα κατάσταση ανήκει στο Κλειστό Σύνολο

Αν η τρέχουσα κατάσταση είναι η τελική, την επιστρέφω – Λύση Προβλήματος

Προσθέτω τη τρέχουσα κατάσταση στο Κλειστό Σύνολο

Βρίσκω τα παιδιά της τρέχουσας κατάστασης

Ελέγχω αν τα παιδιά της τρέχουσας κατάστασης ανήκουν στο Κλειστό Σύνολο

Αν δεν ανήκουν τότε τα προσθέτω στο μέτωπο

Αν αδειάσει το μέτωπο επιστρέφω αποτυχία

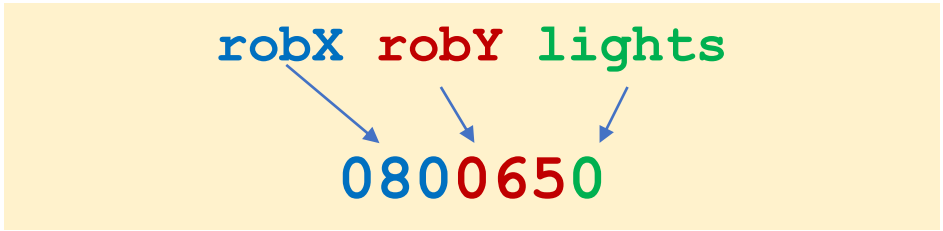
unordered_map

Unordered maps are associative containers that store elements formed by the combination of a *key value* and a *mapped value*, and which allows for **fast retrieval** of individual elements based on their keys.

In an unordered_map, the *key value* is generally used to uniquely identify the element, while the *mapped value* is an object with the content associated to this *key*.

```
pair<unsigned long, Maze *> k (s->getKey(),s);
```

```
unsigned long getKey()
{
    unsigned long k = robX*10000+robY*10;
    if (lights) k+=1;
       return k;
}
```



Λύση με unordered_map<unsigned long,Maze>

```
Maze *BFS(Maze *initial,Maze *goal)
{
    queue<Maze *> agenda;
    unordered_map <unsigned long,Maze *> closed;
    agenda.push(initial);
    while (agenda.size()>0)
    {
        Maze *s = agenda.front();
        agenda.pop();
        if (closed.count(s->getKey())==0)
        {
            if (*s==*goal)
                return s;
            pair<unsigned long,Maze *> k (s->getKey(),s);
            closed.insert(k);
            vector<Maze *> children =s->expand();
            for (unsigned int i=0;i<children.size();i++)
                if (closed.count(children[i]->getKey())==0)
                    agenda.push(children[i]);
        }
    }
    return nullptr;
}
```

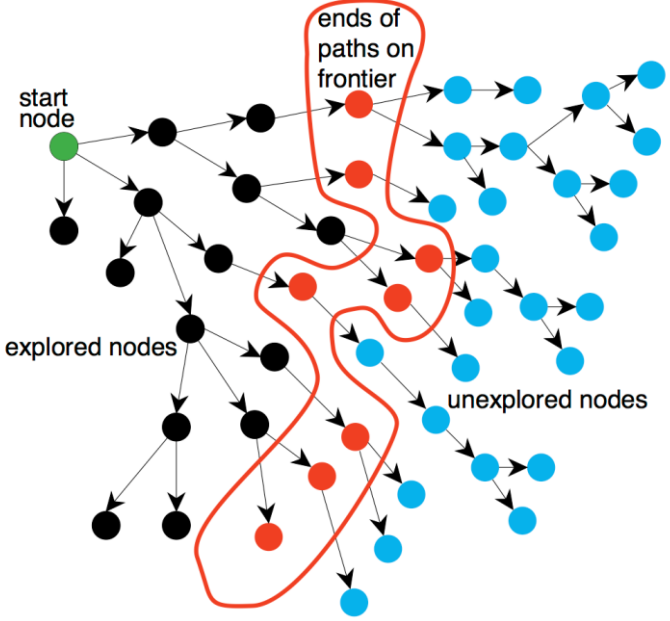
Δημιουργώ το Κλειστό Σύνολο

Ελέγχω αν η τρέχουσα κατάσταση ανήκει στο Κλειστό Σύνολο

Προσθέτω τη τρέχουσα κατάσταση στο Κλειστό Σύνολο

Ελέγχω αν τα παιδιά της τρέχουσας κατάστασης ανήκουν στο Κλειστό Σύνολο

Γενικός Αλγόριθμος Αναζήτησης



Deterministic-Search(Σ, s_0, g)
  Frontier ← {(⟨⟩, s_0)}
  Expanded ← ∅
  while *Frontier* ≠ ∅ do
    select a node ν = (π, s) ∈ *Frontier* (*i*)
    remove ν from *Frontier*
    add ν to *Expanded*
    if *s* satisfies *g* then return *π*
    Children ← {(*π*.*a*, *γ*(*s*,*a*)) | *s* satisfies pre(*a*)}
    prune 0 or more nodes from
      Children, *Frontier*, *Expanded* (*ii*)
    Frontier ← *Frontier* ∪ *Children*
  return failure

DFS

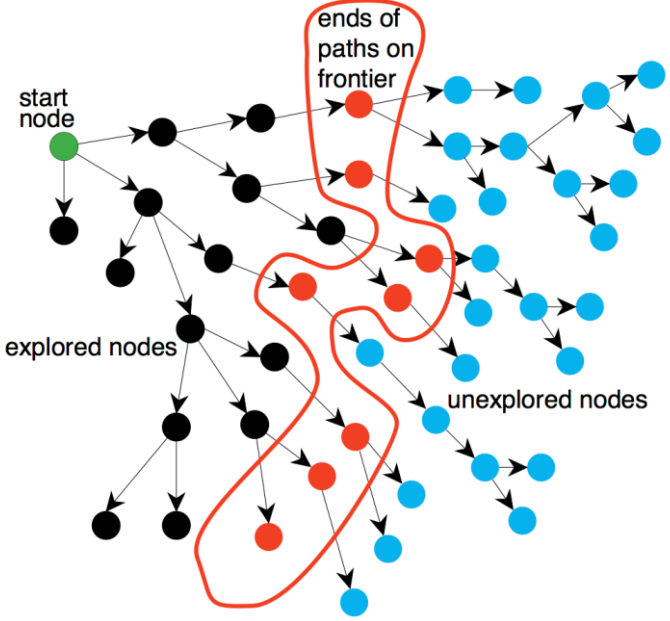
(*i*): Select (*π*,*s*) ∈ *Children* that has largest length(*π*)

- Possible tie-breaking rules: left-to-right, smallest *h*(*s*)

(*ii*): remove every (*π*,*s*) ∈ *Children* ∪ *Frontier* such that *s* is in *Expanded*

- Thus expand states at most once

DFS



Deterministic-Search(Σ, s_0, g)
  Frontier ← {($\langle\rangle$, s_0)}
  Expanded ← ∅
  while *Frontier* ≠ ∅ do
    select a node v = (π, s) ∈ *Frontier* (*i*)
    remove v from *Frontier*
    add v to *Expanded*
    if *s* satisfies *g* then return π
    Children ← {(π.*a*, γ(*s*,*a*)) | *s* satisfies pre(*a*)}
    prune 0 or more nodes from
      Children, *Frontier*, *Expanded* (*ii*)
    Frontier ← *Frontier* ∪ *Children*
  return failure

stack

vector, unordered_map

DFS

(*i*): Select (π,*s*) ∈ *Children* that has largest length(π)

- Possible tie-breaking rules: left-to-right, smallest *h*(*s*)

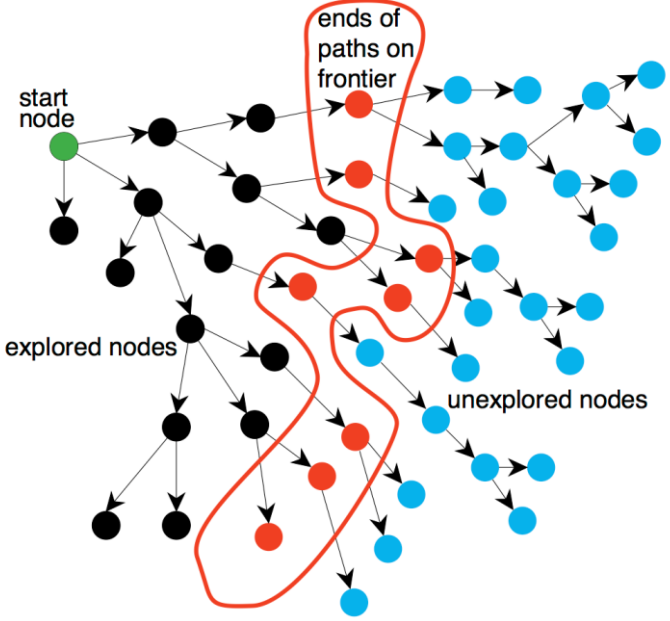
(*ii*): remove every (π,*s*) ∈ *Children* ∪ *Frontier* such that *s* is in *Expanded*

- Thus expand states at most once

Λύση με unordered_map<unsigned long,Maze>

```
Maze *DFS(Maze *initial,Maze *goal)
{
    stack<Maze *> agenda; // stack αντί για queue (BFS)
    unordered_map <unsigned long,Maze *> closed;
    agenda.push(initial);
    while (agenda.size()>0)
    {
        Maze *s = agenda.top(); // top αντί για front (BFS)
        agenda.pop();
        if (closed.count(s->getKey())==0)
        {
            if (*s==*goal)
                return s;
            pair<unsigned long,Maze *> k (s->getKey(),s);
            closed.insert(k);
            vector<Maze *> children =s->expand();
            for (unsigned int i=0;i<children.size();i++)
                if (closed.count(children[i]->getKey())==0)
                    agenda.push(children[i]);
        }
    }
    return nullptr;
}
```

Γενικός Αλγόριθμος Αναζήτησης



Deterministic-Search(Σ, s_0, g)
  Frontier ← {(⟨⟩, s_0)}
  Expanded ← ∅
  while *Frontier* ≠ ∅ do
    select a node ν = (π, s) ∈ *Frontier* (*i*)
    remove ν from *Frontier*
    add ν to *Expanded*
    if *s* satisfies *g* then return *π*
    Children ← {(*π.a*, *γ*(*s*,*a*)) | *s* satisfies pre(*a*)}
    prune 0 or more nodes from
      Children, *Frontier*, *Expanded* (*ii*)
    Frontier ← *Frontier* ∪ *Children*
  return failure

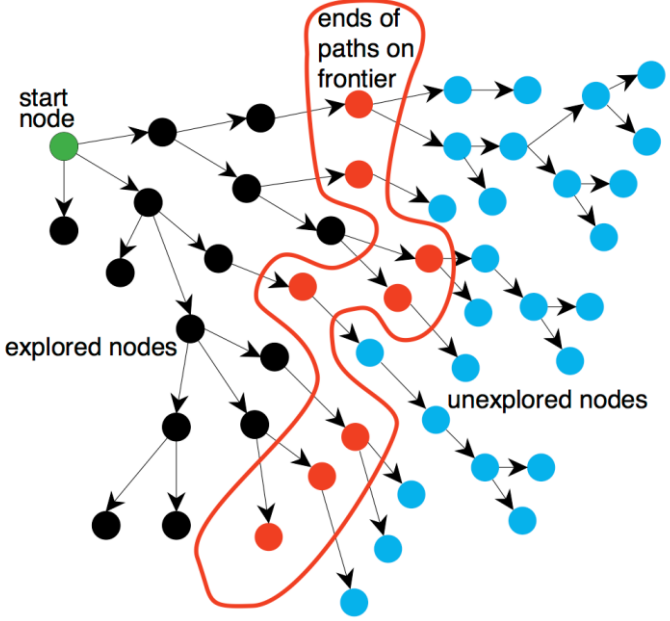
BestFS

(*i*): Select a node (π,s) ∈ Frontier that has smallest h(s)

(*ii*): remove every
(*π,s*) ∈ *Children* ∪ *Frontier*
such that *s* is in *Expanded*

- Thus expand states at most once

BestFS



Deterministic-Search(Σ, s_0, g)
  Frontier ← {(⟨⟩, s_0)}
  Expanded ← ∅
  while *Frontier* ≠ ∅ do
    select a node ν = (π, s) ∈ *Frontier* (*i*)
    remove ν from *Frontier*
    add ν to *Expanded*
    if *s* satisfies *g* then return *π*
    Children ← {(*π.a*, *γ*(*s*,*a*)) | *s* satisfies pre(*a*)}
    prune 0 or more nodes from
      Children, *Frontier*, *Expanded* (*ii*)
    Frontier ← *Frontier* ∪ *Children*
  return failure

vector (manual), priority_queue

vector, unordered_map

BestFS

(*i*): Select a node (π,s) ∈ Frontier that has smallest h(s)

(*ii*): remove every (*π,s*) ∈ *Children* ∪ *Frontier* such that *s* is in *Expanded*

- Thus expand states at most once

priority_queue

Priority queues are a type of container adaptors, specifically designed such that its first element is always the greatest of the elements it contains, according to some strict weak ordering criterion.

```
priority_queue <Maze*, vector <Maze *>, myComparator > agenda;
```

```
class myComparator
{
  public:
    int operator() (Maze *p1 , Maze *p2) const
    {
        return p1->getHvalue()>p2->getHvalue();
    }
};
```

```
Maze *BestFS2(Maze *initial, Maze *goal)
{
    priority_queue <Maze*, vector<Maze *>, myComparator > agenda;
    unordered_map <unsigned long,Maze*> closed;
    agenda.push(initial);
    while (agenda.size()>0)
    {
        Maze *s = agenda.top();
        agenda.pop();
        if (closed.count(s->getKey())==0)
        {
            if (*s==*goal)
                return s;
            pair<unsigned long,Maze*> k (s->getKey(),s);
            closed.insert(k);
            vector<Maze *> children = s->expand();
            for (unsigned int i=0;i<children.size();i++)
                if (closed.count(children[i]->getKey())==0)
                {
                    children.at(i)->setHvalue(children.at(i)->heuristic(goal));
                    agenda.push(children.at(i));
                }
         }
    }
    return nullptr;
}
```

```
Maze *BestFS2(Maze *initial, Maze *goal)
{
    priority_queue <Maze*, vector<Maze *>, myComparator > agenda;
    unordered_map <unsigned long,Maze*> closed;
    agenda.push(initial);
    while (agenda.size()>0)
    {
        Maze *s = agenda.top();
        agenda.pop();
        if (closed.count(s->getKey())==0)
        {
            if (*s==*g
                return
            pair<unsig
            closed.ins
            vector<Maz
            for (unsig
                if (cl
                {
                    children.at(i)->setHvalue(children.at(i)->heuristic(goal));
                    agenda.push(children.at(i));
                }
         }
    }
    return nullptr;
}
```

```
int Maze::heuristic (Maze *goal)
{
    int dist = abs(goal->robX-robX) + abs(goal->robY-robY);
    if (lights!=goal->lights)
        dist++;
    return dist;
};
```