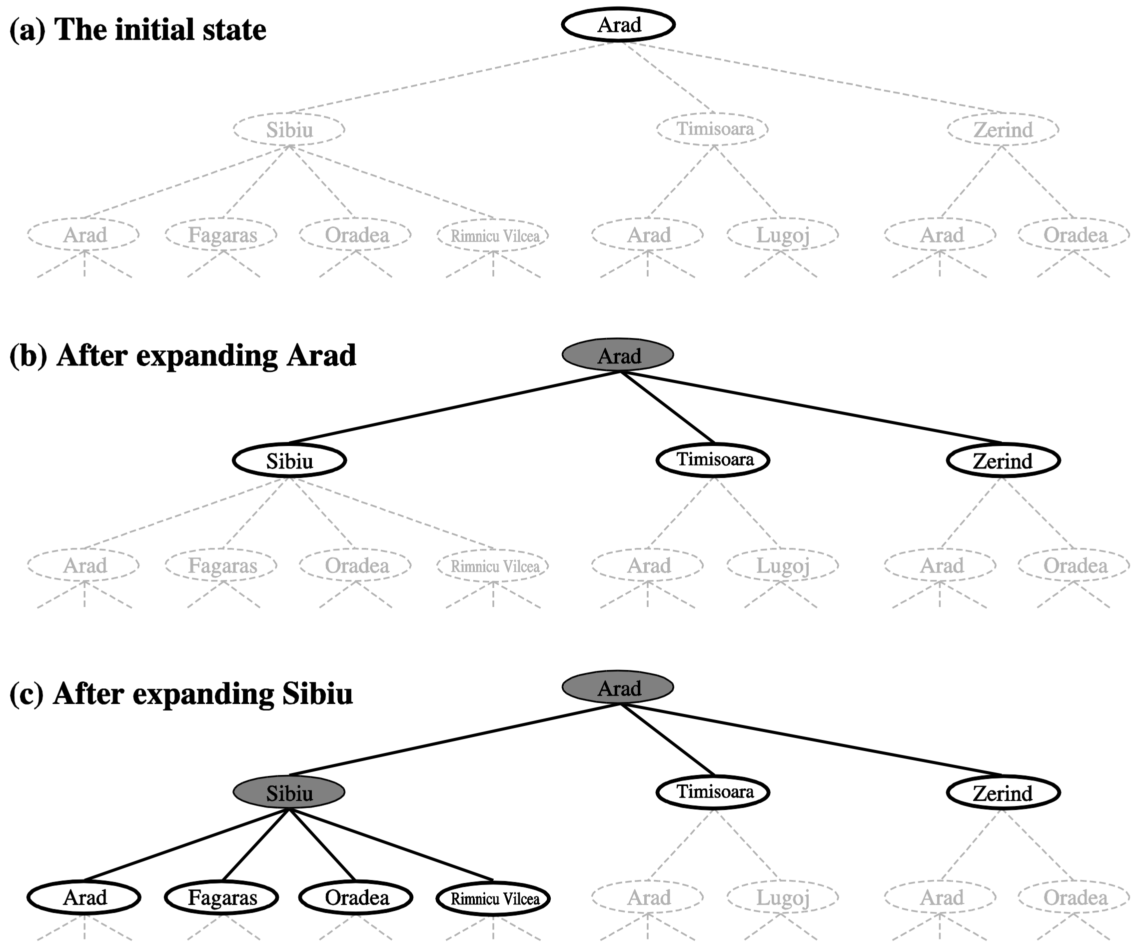
Αναζήτηση σε Γράφους

Μανόλης Κουμπαράκης

Πρόλογος

- Μια ενδιαφέρουσα περίπτωση προβλημάτων αναζήτησης είναι αυτά στα οποία ο χώρος καταστάσεων είναι **γράφος.**

Παράδειγμα


(a) The initial state
Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea
(b) After expanding Arad
Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea
(c) After expanding Sibiu
Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea

Επαναλαμβανόμενες Καταστάσεις

- Βλέπουμε ότι η κατάσταση *In(Arad)* είναι **επαναλαμβανόμενη** και παράγεται από ένα μονοπάτι που περιέχει **κύκλο**.
- Άρα το δένδρο αναζήτησης για το πρόβλημα μας είναι **άπειρο** αν και ο χώρος καταστάσεων έχει μόνο **20 καταστάσεις**!
- Ευτυχώς **δεν χρειάζεται** οι αλγόριθμοι αναζήτησης να εξετάζουν επαναλαμβανόμενες καταστάσεις.
- Ένα μονοπάτι που περιέχει κύκλο έχει πάντα μεγαλύτερο κόστος από το ίδιο μονοπάτι αν αφαιρέσουμε τον κύκλο (με την υπόθεση ότι το **κόστος βήματος είναι μη αρνητικό**).

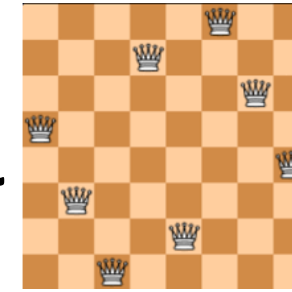
Πλεονάζοντα Μονοπάτια

- Τα μονοπάτια που έχουν κύκλο είναι μια ειδική περίπτωση της πιο γενικής έννοιας των **πλεοναζόντων μονοπατιών**.
- Έχουμε πλεονάζοντα μονοπάτια όταν υπάρχουν περισσότεροι από ένας τρόποι για να πάμε από μια κατάσταση σε μια άλλη.

- **Παράδειγμα**: Τα μονοπάτια Arad-Sibiu (140 χιλιόμετρα) και Arad-Zerind-Oradea-Sibiu (297 χιλιόμετρα). Προφανώς το 2ο μονοπάτι είναι πλεονάζον.

- Αν θέλουμε να φτάσουμε σε μια κατάσταση στόχου, δεν υπάρχει λόγος να λαμβάνουμε υπόψη μας πλεονάζοντα μονοπάτια.

Πλεονάζοντα Μονοπάτια

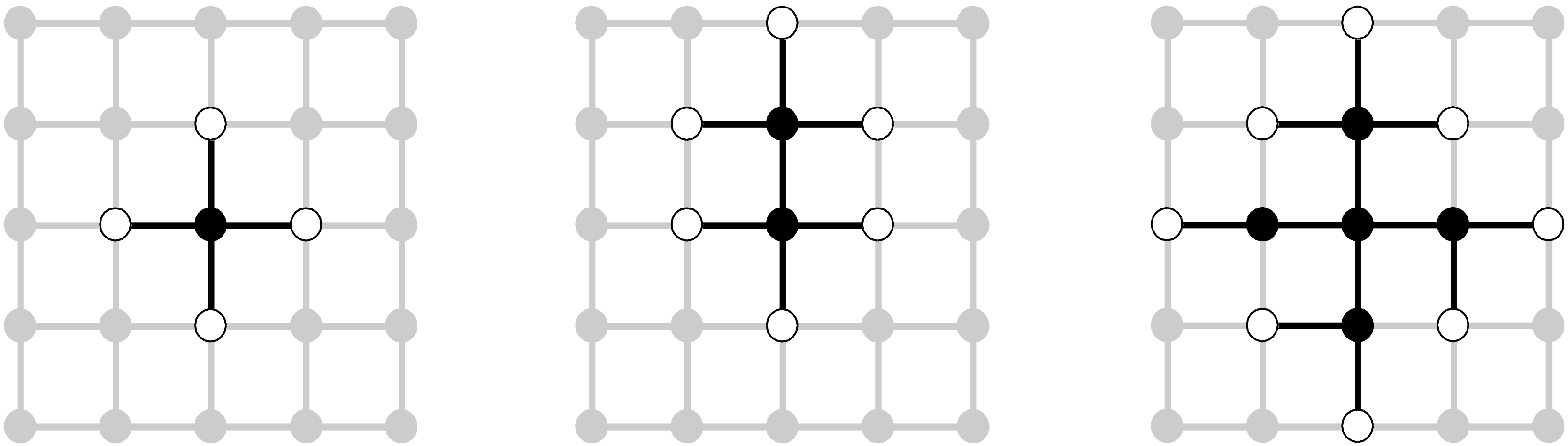


- Μερικές φορές μπορούμε να ορίσουμε ένα πρόβλημα αναζήτησης με τέτοιο τρόπο ώστε να μην δημιουργούνται πλεονάζοντα μονοπάτια.
- **Παράδειγμα**: το πρόβλημα των 8 βασιλισσών.
- Αν ορίσουμε το πρόβλημα ώστε μια βασίλισσα να καταλαμβάνει οποιοδήποτε τετραγωνάκι, τότε έχουμε πλεονάζοντα μονοπάτια.
- Αν ορίσουμε το πρόβλημα ώστε μια βασίλισσα καταλαμβάνει ένα τετραγωνάκι στην αριστερότερη ελεύθερη στήλη, τότε δεν έχουμε πλεονάζοντα μονοπάτια.

Πλεονάζοντα Μονοπάτια

- Υπάρχουν περιπτώσεις που η ύπαρξη πλεοναζόντων μονοπατιών δεν μπορεί να αποφευχθεί.
- **Παράδειγμα**: όλα τα προβλήματα για τα οποία οι ενέργειες είναι αντιστρέψιμες (π.χ., προβλήματα εύρεσης διαδρομής ή προβλήματα ολισθαινόντων πλακιδίων).

Παράδειγμα



(a) (b) (c)

- Η εύρεση διαδρομής σε **ορθογώνια πλέγματα** όπως το παραπάνω είναι σημαντική σε παιχνίδια υπολογιστή π.χ., Pacman.

Παράδειγμα

- Σε ένα τέτοιο πλέγμα κάθε κατάσταση έχει 4 διάδοχες καταστάσεις, οπότε ένα δένδρο αναζήτησης με βάθος d που περιέχει επαναλαμβανόμενες καταστάσεις έχει 4^d φύλλα.
- Όμως υπάρχουν μόνο περίπου $2d^2$ διαφορετικές καταστάσεις σε απόσταση d βημάτων από οποιαδήποτε κατάσταση (άσκηση!).
- Για $d = 20$, αυτό σημαίνει **ένα τρισεκατομμύριο** κόμβους αλλά μόνο **800** διαφορετικές καταστάσεις.

Το Εξερευνημένο Σύνολο

- Για να αποφύγουμε τη δημιουργία πλεοναζόντων μονοπατιών, χρησιμοποιούμε μια δομή δεδομένων που λέγεται **εξερευνημένο σύνολο (explored set)** ή **κλειστή λίστα (closed list)**.
- Η δομή αυτή **«θυμάται» την κατάσταση** κάθε εξερευνημένου κόμβου.

Ο Αλγόριθμος GRAPH-SEARCH

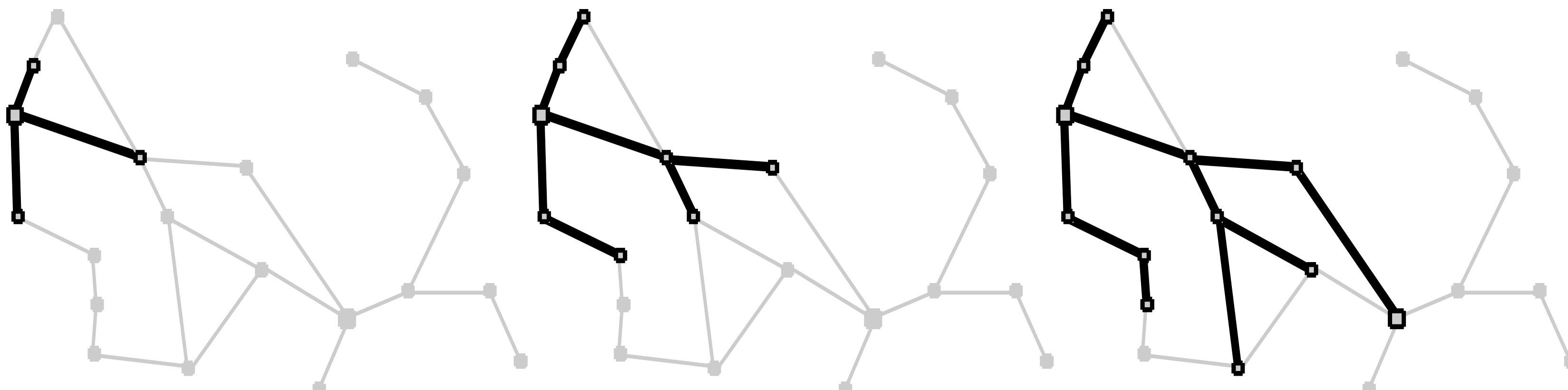
function GRAPH-SEARCH(*problem*) **returns** a solution or failure

initialize the frontier using the initial state of *problem*.
initialize the explored set to be empty
loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    add the state of the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier **only if their state is not in the frontier or the explored set**

Ο Αλγόριθμος GRAPH-SEARCH

- Το δένδρο αναζήτησης που κατασκευάζεται από τον αλγόριθμο GRAPH-SEARCH έχει **το πολύ ένα** αντίγραφο κάθε κατάστασης.
- Άρα μπορούμε να το απεικονίσουμε να δημιουργείται **πάνω στον γράφο του χώρου καταστάσεων** όπως φαίνεται στο παρακάτω παράδειγμα.

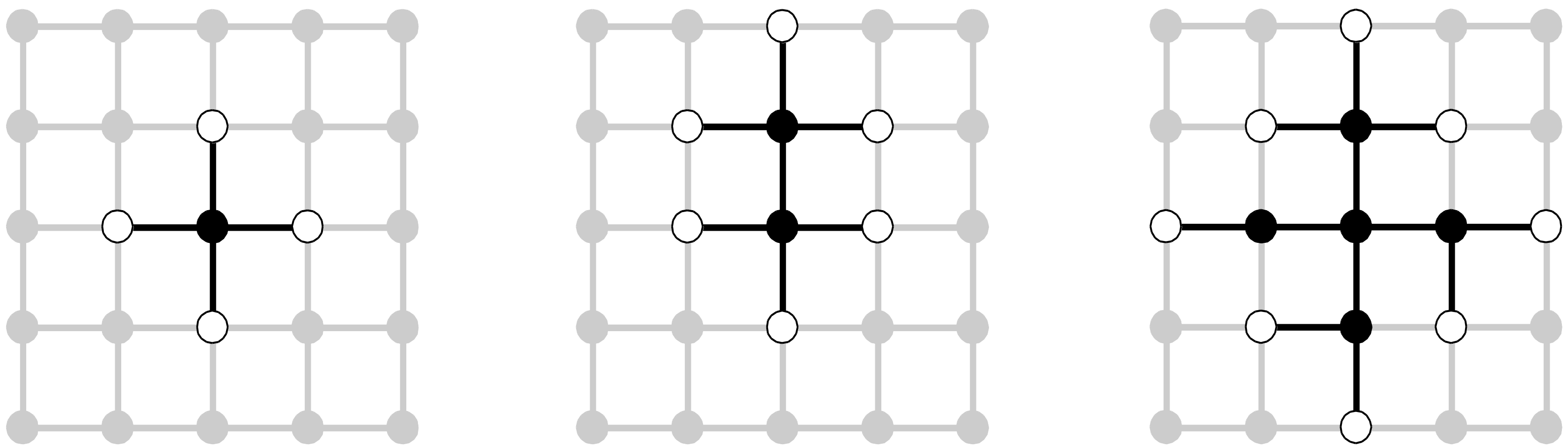
Παράδειγμα



Ο Αλγόριθμος Graph-Search

- Ο αλγόριθμος Graph-Search έχει επίσης την παρακάτω ιδιότητα:
  - Το σύνορο **διαχωρίζει τον γράφο του χώρου καταστάσεων** σε μια εξερευνημένη περιοχή και σε μια ανεξερεύνητη περιοχή.
  - Κάθε μονοπάτι από την αρχική κατάσταση σε μια ανεξερεύνητη κατάσταση **περνάει υποχρεωτικά** από ένα κόμβο του συνόρου.

Παράδειγμα

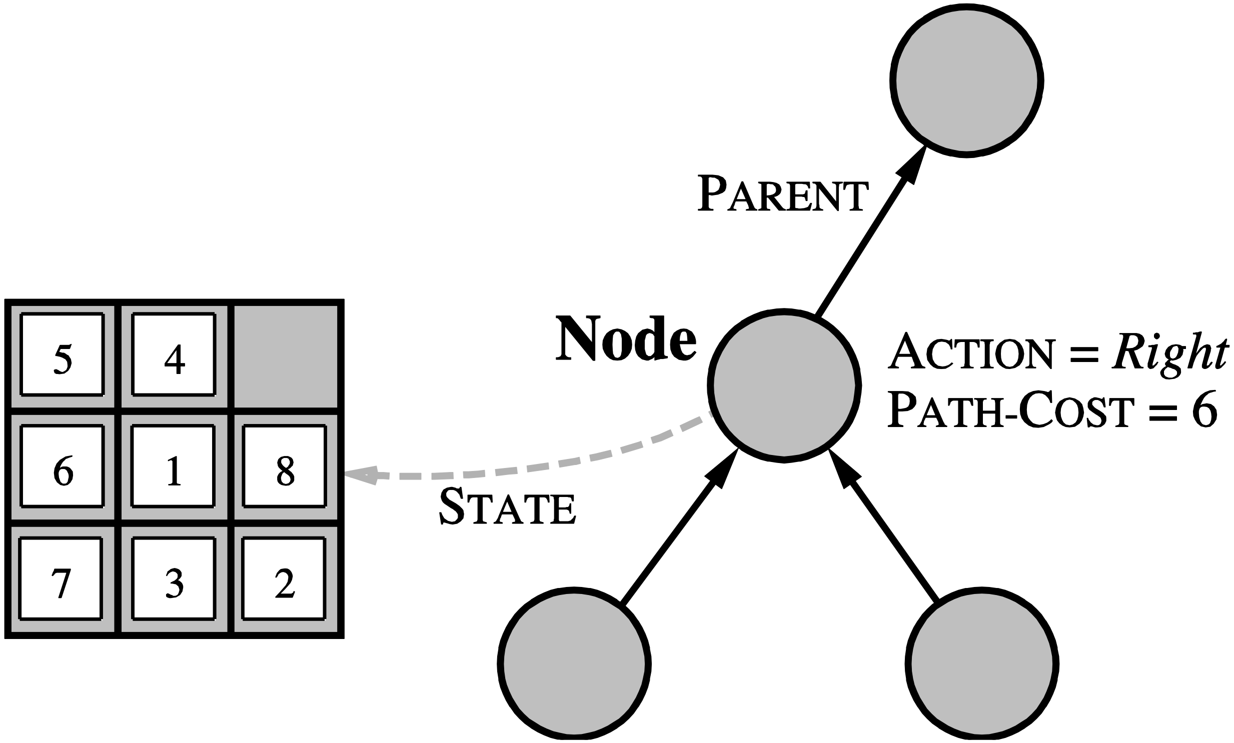


(a) (b) (c)

Υλοποίηση του GRAPH-SEARCH

- Για κάθε **κόμβο** n του δένδρου αναζήτησης, θα έχουμε μια δομή που περιέχει τα ακόλουθα πεδία:
  - n.STATE: η κατάσταση του κόμβου.
  - n.PARENT: ο κόμβος του δένδρου αναζήτησης που παρήγαγε τον n.
  - n.ACTION: η ενέργεια που εφαρμόστηκε στον πατέρα του κόμβου για να δημιουργηθεί ο n.
  - n.PATH-COST: το κόστος του μονοπατιού από την αρχική κατάσταση μέχρι τον n. Το μονοπάτι ορίζεται από τους δείκτες στον πατέρα κάθε κόμβου.

Η Δομή Δεδομένων για Ένα Κόμβο



Σχόλια

- Παρατηρήστε ότι οι δείκτες PARENT σχηματίζουν τη **δομή του δένδρου**.
- Οι ίδιοι δείκτες μας επιτρέπουν να **κατασκευάσουμε τη λύση** όταν βρούμε ένα κόμβο στόχου.
- Χρησιμοποιούμε τη συνάρτηση SOLUTION η οποία επιστρέφει μια ακολουθία ενεργειών ακολουθώντας δείκτες PARENT πίσω στην ρίζα.

Υλοποίηση του GRAPH-SEARCH

- Η παρακάτω συνάρτηση παίρνει ένα κόμβο-πατέρα και μια ενέργεια και δημιουργεί τον αντίστοιχο κόμβο-παιδί:

function CHILD-NODE(*problem, parent, action*) **returns** a node
return a node with
    STATE = *problem*.RESULT(parent.STATE, *action*)
    PARENT = *parent*
    ACTION = *action*
    PATH-COST = *parent*.PATH-COST +
        problem.STEP-COST(*parent*.STATE, *action*)

Η Δομή Δεδομένων για την Ουρά

- Όπως και στην περίπτωση του TREE-SEARCH, θα έχουμε μια **ουρά (queue)** η οποία υλοποιεί το σύνορο.
- Οι λειτουργίες της ουράς είναι οι παρακάτω:
  - EMPTY?(*queue*): ελέγχει αν η ουρά *queue* είναι άδεια.
  - POP(*queue*): αφαιρεί το πρώτο στοιχείο από την ουρά *queue* και το επιστρέφει.
  - INSERT(*element, queue*): εισάγει το στοιχείο *element* στην ουρά *queue* και επιστρέφει την ουρά που προκύπτει.

Τύποι Ουράς

- Ουρά **FIFO** (first-in first-out). Εξάγει το στοιχείο της ουράς που εισήχθηκε πρώτο (**ουρά αναμονής**).
- Ουρά **LIFO** (last-in first-out). Εξάγει το στοιχείο της ουράς που εισήχθηκε τελευταίο. Λέγεται επίσης **στοίβα** (**stack**).
- **Ουρά προτεραιότητας (priority queue)**. Εξάγει το στοιχείο της ουράς που έχει τη **μεγαλύτερη προτεραιότητα** με βάση μια συνάρτηση διάταξης των στοιχείων.

Ερώτηση

- Πως υλοποιούμε μια **ουρά αναμονής, μια στοίβα** ή μια **ουρά προτεραιότητας** σε Python;

Απάντηση

- Εύκολα! (Εργασία 0 και συνέχεια).

Ερώτηση

- Πως υλοποιούμε το εξερευνημένο σύνολο στην Python;

Απάντηση

- Το εξερευνημένο σύνολο μπορεί να υλοποιηθεί με τη δομή **σύνολο (set)** της Python.
- Τα σύνολα υλοποιούνται εσωτερικά με **πίνακες κατακερματισμού (hash tables)** οπότε το να βρούμε αν μια κατάσταση είναι στοιχείο ενός συνόλου γίνεται πολύ αποδοτικά (χρονική πολυπλοκότητα $\boldsymbol{O}(\mathbf{1})$).

Αλγόριθμοι

- Ας δούμε τώρα πως μπορούμε να υλοποιήσουμε τους παρακάτω αλγόριθμους στην περίπτωση που ο χώρος καταστάσεων είναι γράφος:
  - Αναζήτηση πρώτα κατά πλάτος (breadth-first search)
  - Αναζήτηση ομοιόμορφου κόστους (uniform cost search)
  - Αναζήτηση πρώτα κατά βάθος (depth-first search)
- Οι υπόλοιποι αλγόριθμοι που μελετήσαμε για την περίπτωση της αναζήτησης σε δένδρο υλοποιούνται με παρόμοιο τρόπο.

Αναζήτηση Πρώτα κατά Πλάτος

function BREADTH-FIRST-SEARCH(*problem*) **returns** a solution or failure

node ← a node with STATE=*problem*.INITIAL-STATE, PATH-COST=0
if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)

frontier ← a FIFO queue with *node* as the only element
explored ← an empty set
loop do
    if EMPTY?(*frontier*) **then return** failure
    node ← POP(*frontier*)
    add *node*.STATE to *explored*
    for each *action* in *problem*.ACTIONS(*node*.STATE) **do**
        child ← CHILD-NODE(*problem*, *node*, *action*)
        if *child*.STATE is not in *explored* or *frontier* **then**
            if *problem*.GOAL-TEST(*child*.STATE) **then return** SOLUTION(*child*)
            frontier ← INSERT(*child*, *frontier*)

Αναζήτηση Πρώτα κατά Πλάτος

- Ο προηγούμενος αλγόριθμος διαφέρει από τους γενικούς αλγόριθμους TREE-SEARCH και GRAPH-SEARCH επειδή ελέγχει αν ένας κόμβος είναι κόμβος στόχου **όταν ο κόμβος παράγεται** και όχι όταν ο κόμβος αφαιρείται από το σύνορο.
- Όπως και ο GRAPH-SEARCH, ο αλγόριθμος απορρίπτει κάθε μονοπάτι που οδηγεί σε μία κατάσταση που είναι ήδη στο εξερευνημένο σύνολο ή στο σύνορο.
- Κάθε τέτοιο μονοπάτι έχει κόστος τουλάχιστον όσο κάποιο μονοπάτι που έχουμε βρει ήδη, άρα μπορούμε να το απορρίψουμε.

Ερώτηση

- Ποια είναι η **χρονική πολυπλοκότητα** του αλγόριθμου BREADTH-FIRST-SEARCH τώρα;

Χρονική Πολυπλοκότητα

- Υποθέστε ότι κάνουμε αναζήτηση σε ένα ομοιόμορφο δένδρο όπου κάθε κατάσταση έχει $\boldsymbol{b}$ **διάδοχες καταστάσεις** ($\boldsymbol{b}$ είναι ο παράγοντας διακλάδωσης).
- Η ρίζα του δένδρου παράγει b κόμβους στο πρώτο επίπεδο, καθένας από τους οποίους παράγει b κόμβους στο δεύτερο επίπεδο. Άρα στο δεύτερο επίπεδο έχουμε συνολικά b^2 κόμβους.
- Με όμοιο τρόπο βλέπουμε ότι στο 3ο επίπεδο έχουμε συνολικά b^3 κόμβους κ.ο.κ.
- Τώρα υποθέστε ότι η λύση βρίσκεται σε **βάθος** $\boldsymbol{d}$. Στην χειρότερη περίπτωση, **ο τελευταίος κόμβος** που παράγεται θα είναι σε αυτό το επίπεδο.
- Άρα ο **συνολικός αριθμός κόμβων** που παράγεται στη χειρότερη περίπτωση είναι

$$1 + b + b^2 + b^3 + \cdots + b^d = O(b^d)$$

- Άρα η πολυπλοκότητα χρόνου στην χειρότερη περίπτωση είναι $\boldsymbol{O(b^d)}$.

Χρονική Πολυπλοκότητα

- Έχουμε δει ήδη ότι, αν ο αλγόριθμος ελέγχει ότι ένας κόμβος είναι κόμβος στόχου όταν ο κόμβος αυτός βγαίνει από το σύνορο και όχι όταν δημιουργείται, τότε η πολυπλοκότητα χρόνου είναι $\boldsymbol{O}\left(\boldsymbol{b}^{\boldsymbol{d}+\mathbf{1}}\right)$.
- Άρα ο προηγούμενος αλγόριθμος είναι πιο αποδοτικός.

Χωρική Πολυπλοκότητα

- Με όμοιο τρόπο μπορούμε να δούμε ότι το εξερευνημένο σύνολο και το σύνορο θα έχουν $\boldsymbol{O(b^d)}$ κόμβους.
- Άρα η πολυπλοκότητα χώρου είναι επίσης $\boldsymbol{O\left(b^d\right)}$.

Χρονική και Χωρική Πολυπλοκότητα

- Η επόμενη διαφάνεια μας δείχνει τις απαιτήσεις του BFS σε χρόνο και χώρο.
- Υποθέτουμε ότι $b = 10$, ότι μπορούμε να επεξεργαζόμαστε 10^6 κόμβους το δευτερόλεπτο και ότι κάθε κόμβος χρειάζεται 1KB μνήμη.

Μετρήσεις

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Άλλες Ιδιότητες

- Ο αλγόριθμος αναζήτησης πρώτα κατά πλάτος στην περίπτωση που ο χώρος αναζήτησης είναι γράφος, έχει τις ίδιες καλές ιδιότητες που είδαμε στην περίπτωση του Tree-Search:
  - Είναι **πλήρης** (με την υπόθεση ότι ο παράγοντας διακλάδωσης b είναι πεπερασμένος).
  - Είναι **βέλτιστος** (με την υπόθεση ότι όλες οι ενέργειες έχουν το ίδιο μη αρνητικό κόστος).

Αναζήτηση Ομοιόμορφου Κόστους

- Όταν όλα τα κόστη βήματος είναι σταθερά, ο αλγόριθμος αναζήτησης πρώτα κατά πλάτος είναι **βέλτιστος** και βρίσκει την **πιο αβαθή λύση.**
- Μπορούμε εύκολα να επεκτείνουμε αυτό τον αλγόριθμο στην περίπτωση που έχουμε οποιαδήποτε συνάρτηση κόστους βήματος.
- Αντί να επεκτείνει τον πιο αβαθή κόμβο, ο αλγόριθμος **αναζήτησης ομοιόμορφου κόστους** επεκτείνει τον κόμβο n με το **μικρότερο κόστος μονοπατιού** $g(n)$.
- Στην περίπτωση αυτή το σύνορο υλοποιείται σαν μια **ουρά προτεραιότητας** με συνάρτηση g.

Αναζήτηση Ομοιόμορφου Κόστους

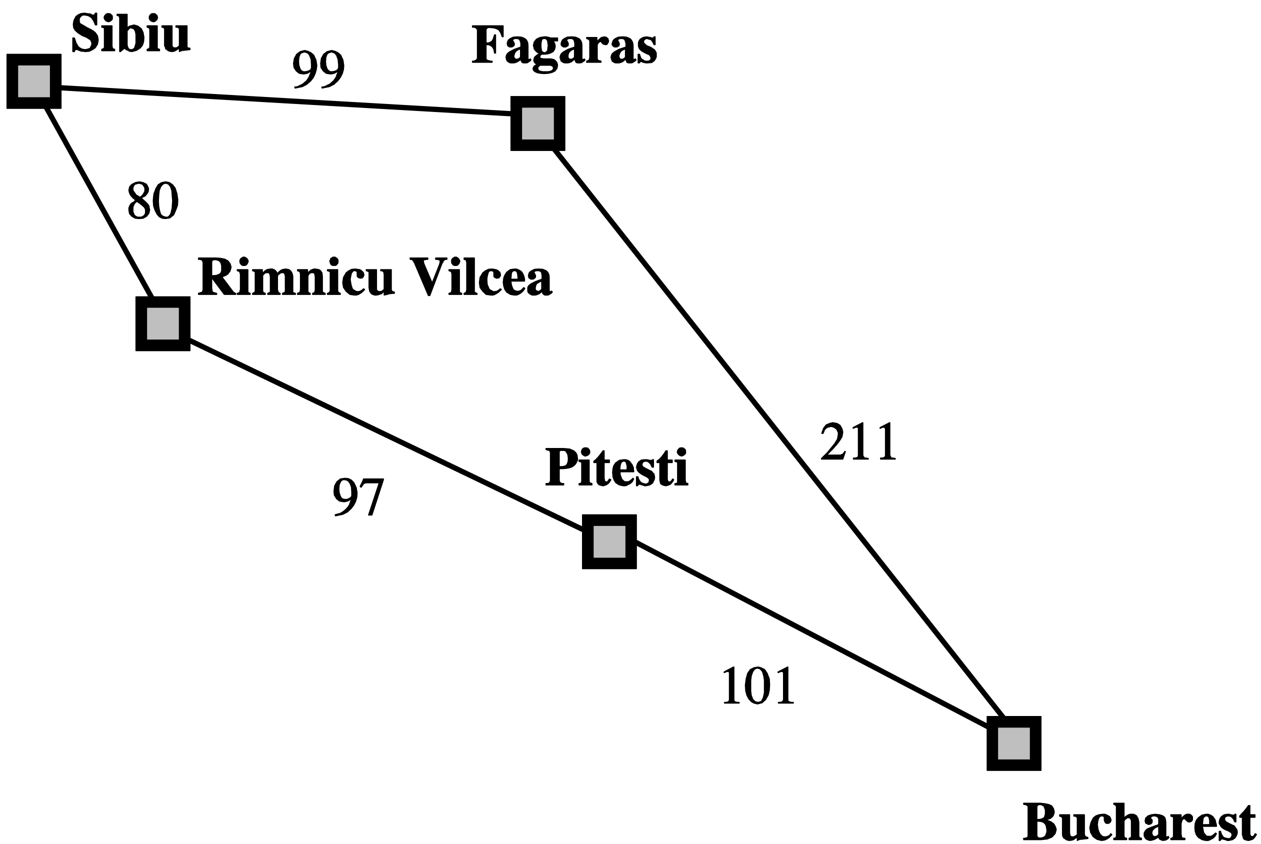
```
function UNIFORM-COST-SEARCH(problem) returns a solution or failure

node ← a node with STATE=problem.INITIAL-STATE, PATH-COST=0
frontier ← a priority queue ordered by PATH-COST, with node as the only element
explored ← an empty set
loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
            frontier ← INSERT(child, frontier)
        else if child.STATE is in frontier with higher PATH-COST then
            replace that frontier node with child
```

Αναζήτηση Ομοιόμορφου Κόστους

- Παρατηρήστε τις **διαφορές** με τον αλγόριθμο αναζήτησης πρώτα κατά πλάτος:
  - Χρησιμοποιείται ουρά προτεραιότητας με συνάρτηση g αντί για ουρά αναμονής.
  - Ο έλεγχος στόχου εφαρμόζεται όταν ένας κόμβος επιλέγεται από το σύνορο για να επεκταθεί και όχι όταν δημιουργείται (όπως και στον γενικό αλγόριθμο GRAPH-SEARCH).
  - Ελέγχουμε αν έχουμε βρει ένα καλύτερο μονοπάτι προς ένα κόμβο που ήδη βρίσκεται στο σύνορο.
- Όλες αυτές οι διαφορές είναι απαραίτητες όπως φαίνεται στο παρακάτω παράδειγμα.

Παράδειγμα


Sibiu
99
Fagaras
80
Rimnicu Vilcea
211
Pitesti
97
101
Bucharest

Συζήτηση

- Αν ο έλεγχος στόχου ήταν όπως στον αλγόριθμο αναζήτησης πρώτα κατά πλάτος, θα είχαμε επιλέξει ένα μονοπάτι με **μη βέλτιστο κόστος:** το μονοπάτι Sibiu-Fagaras-Bucharest με κόστος 310 αντί του βέλτιστου Sibiu-Rimnicu Vilcea-Pitesti-Bucharest με κόστος 278.

Ιδιότητες

- Ο αλγόριθμος αναζήτησης ομοιόμορφου κόστους που παρουσιάσαμε έχει τις ίδιες ιδιότητες με την περίπτωση του TREE-SEARCH (**πληρότητα, βέλτιστη συμπεριφορά, υπολογιστική πολυπλοκότητα χώρου και χρόνου**).

Ερώτηση

- Πως υλοποιούμε αποδοτικά τη δομή για το **σύνορο** σε Python για την περίπτωση της αναζήτησης ομοιόμορφου κόστους;

Υλοποίηση

- Η δομή δεδομένων για το σύνορο πρέπει να υποστηρίζει αποδοτικά τις λειτουργίες της **ουράς προτεραιότητας** και την **λειτουργία του ελέγχου αν ένα στοιχείο ανήκει στο σύνορο**.
- Μια τέτοια δομή είναι η υλοποίηση της ουράς προτεραιότητας με **σωρό (heap)**. Για την δομή αυτή έχουμε τις εξής πολυπλοκότητες (n είναι ο αριθμός των στοιχείων της ουράς):
  - Εισαγωγή στοιχείου: $O(\log n)$
  - Εξαγωγή στοιχείου με τη μεγαλύτερη προτεραιότητα: $O(\log n)$
  - Έλεγχος αν ένα στοιχείο ανήκει στην ουρά: $O(n)$. Αυτή η πολυπλοκότητα μπορεί να γίνει $O(1)$ αν κρατάμε παράλληλα ένα **πίνακα κατακερματισμού** που χρησιμοποιείται μόνο για αυτό τον έλεγχο. Έτσι όμως ξοδεύουμε διπλό χώρο.

Υλοποίηση σε Python

- Δοκιμάστε το!

Αναζήτηση Πρώτα Κατά Βάθος

- Ο αλγόριθμος αναζήτησης πρώτα κατά βάθος μπορεί να υλοποιηθεί όπως ο γενικός αλγόριθμος GRAPH-SEARCH χρησιμοποιώντας μια **στοίβα** για την υλοποίηση του συνόρου.
- Μπορεί επίσης να υλοποιηθεί και **αναδρομικά** με μία συνάρτηση που καλεί τον εαυτό της σε κάθε ένα από τα παιδιά της.

Πληρότητα

- Όταν ο αλγόριθμος αναζήτησης πρώτα κατά βάθος υλοποιείται ώστε να ελέγχουμε ότι κάθε νέα κατάσταση δεν περιέχεται στο εξερευνημένο σύνολο, τότε, για **πεπερασμένους χώρους καταστάσεων**, είναι **πλήρης** σε αντίθεση με την υλοποίηση με TREE-SEARCH.
- Για **άπειρους χώρους καταστάσεων** και οι δύο εκδόσεις είναι **μη πλήρεις** (όταν συναντήσουν ένα άπειρο μονοπάτι που δεν περιέχει κατάσταση στόχου).

Παράδειγμα

- Το παρακάτω πρόβλημα αναζήτησης έχει άπειρο χώρο καταστάσεων:
  - **Καταστάσεις**: Θετικοί και αρνητικοί ακέραιοι.
  - **Αρχική κατάσταση**: Ο αριθμός 0.
  - **Ενέργειες**: Προσθέστε 1 ή αφαιρέστε 1 από τον ακέραιο στο οποίο βρισκόμαστε.
  - **Κατάσταση στόχου**: Ένας δοσμένος ακέραιος.

Παράδειγμα

- Ο ακέραιος 5 προκύπτει ως εξής:
$$(((((0+1)+1)+1)+1)+1)$$

- **Άπειρο μονοπάτι** προκύπτει αν για την παραπάνω κατάσταση στόχου επιλέγουμε να χρησιμοποιούμε πάντα πρώτη την πράξη της αφαίρεσης.

Παράδειγμα

- Το παρακάτω πρόβλημα αναζήτησης που προτάθηκε από τον Knuth έχει άπειρο χώρο καταστάσεων:
  - **Καταστάσεις**: Θετικοί ακέραιοι.
  - **Αρχική κατάσταση**: Ο αριθμός 4.
  - **Ενέργειες**: Εφαρμόστε τις πράξεις παραγοντικό (για ακεραίους μόνο), τετραγωνική ρίζα ή κατώφλι.
  - **Κατάσταση στόχου**: Ένας δοσμένος ακέραιος.

Παράδειγμα

- Μπορούμε να φτάσουμε στην κατάσταση στόχου 5 ως εξής:

$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \right\rfloor = 5$$

- **Άπειρο μονοπάτι** προκύπτει αν για την παραπάνω κατάσταση στόχου επιλέγουμε να χρησιμοποιούμε πάντα πρώτη την πράξη του παραγοντικού.

Άπειροι Χώροι Καταστάσεων

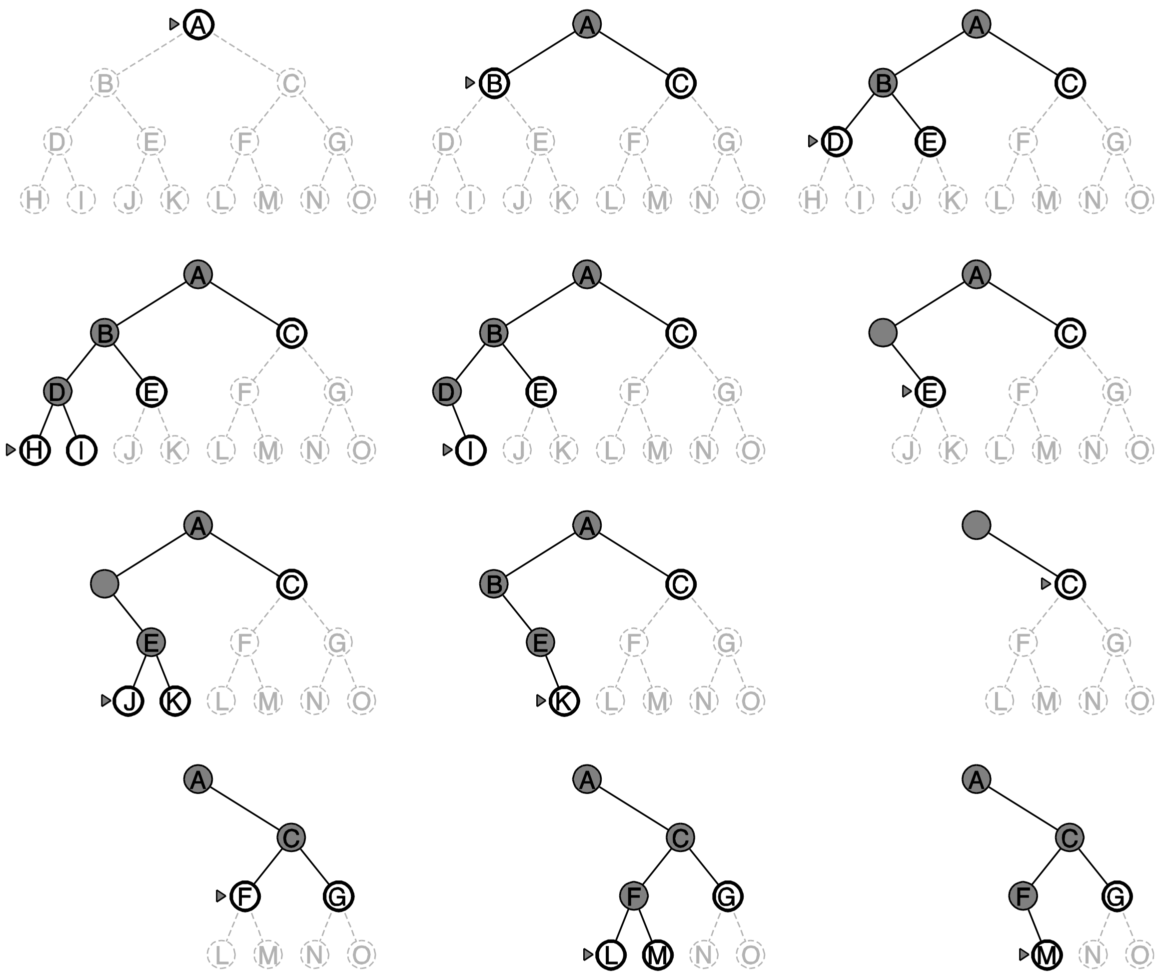
- Άπειροι χώροι καταστάσεων συναντιούνται συχνά σε προβλήματα που αφορούν μαθηματικές εκφράσεις, αποδείξεις, κυκλώματα, προγράμματα και άλλα αντικείμενα που ορίζονται αναδρομικά.

Βέλτιστη Συμπεριφορά

- Και στις δύο περιπτώσεις (πεπερασμένος ή άπειρος χώρος καταστάσεων), η αναζήτηση πρώτα κατά βάθος **δεν είναι βέλτιστη**.

- **Παράδειγμα**: στην επόμενη διαφάνεια αν οι καταστάσεις J και C είναι καταστάσεις στόχου. Ο αλγόριθμος θα επιστρέψει το μονοπάτι προς τη μη βέλτιστη κατάσταση J.

Παράδειγμα



Πολυπλοκότητα

- Η **χρονική πολυπλοκότητα** του αλγόριθμου αναζήτησης πρώτα κατά βάθος στην περίπτωση της αναζήτησης σε γράφο είναι $O(b^m)$ όπως αποδείξαμε για την περίπτωση του TREE-SEARCH όπου m είναι το βάθος οποιουδήποτε κόμβου.
- Το m μπορεί να είναι πολύ μεγαλύτερο του d που είναι το βάθος της ποιο ρηχής λύσης.

- Η χρήση του εξερευνημένου συνόλου όμως **αλλάζει την γραμμική πολυπλοκότητα χώρου** του αλγόριθμου που τώρα γίνεται **ίδια με αυτή της αναζήτησης πρώτα κατά πλάτος.**

Χωρική Πολυπλοκότητα

- Με τις ίδιες υποθέσεις που κάναμε νωρίτερα για τον BFS, ο DFS σε δένδρα χρειάζεται χώρο **156KB αντί για 10 exabytes** στο βάθος $d = 16$.

Αναζήτηση με Υπαναχώρηση

- Μια παραλλαγή της αναζήτησης πρώτα κατά βάθος είναι **η αναζήτηση με υπαναχώρηση (backtracking search).**
- Ο αλγόριθμος αυτός δημιουργεί μόνο ένα παιδί ενός κόμβου κάθε φορά αντί για όλα τα παιδιά. Κάθε κόμβος όμως πρέπει να θυμάται ποιό παιδί να δημιουργήσει την επόμενη φορά.
- Έτσι η πολυπλοκότητα χώρου γίνεται $O(m)$ αντί για $O(bm)$.
- Θα μελετήσουμε λεπτομερώς παραλλαγές αυτού του αλγόριθμου στην ενότητα «Προβλήματα Ικανοποίησης Περιορισμών».

Ο DFS Είναι Δημοφιλής

- Η καλή πολυπλοκότητα χώρου της αναζήτησης πρώτα κατά βάθος στην περίπτωση του Tree-Search είναι υπεύθυνη για τη **χρήση της σε πολλές περιοχές της Τεχνητής Νοημοσύνης** π.χ., προβλήματα ικανοποίησης περιορισμών, λογικός προγραμματισμός, ικανοποιησιμότητα στη προτασιακή λογική κλπ.

Υλοποίηση

- Οι αλγόριθμοι που παρουσιάσαμε σε αυτές τις διαφάνειες είναι κάποιοι από αυτούς που θα υλοποιήσετε στο Pacman project P1.

Μελέτη

- Βιβλίο AIMA, 4η έκδοση (μπορείτε να το πάρετε από τον Εύδοξο).
  - Κεφάλαιο 3 (3.1-3.4)