ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
TECHNICAL UNIVERSITY OF CRETE

Τεχνητή Νοημοσύνη

ΠΛΗ 311

Project 1

Ρεβύθης Κωνσταντίνος
AM: 2012030136
krevythis@isc.tuc.gr

Σκευάκης Βασίλειος
AM: 2012030033
vskevakis@isc.tuc.gr

Τμημα Ηλεκτρολογων Μηχανικων
& Μηχανικων Υπολογιστων

28 Μαρτίου 2021

Περιεχόμενα

Εισαγωγή

Στην παρούσα εργασία καλούμαστε να λύσουμε το πρόβλημα εύρεσης της βέλτιστης διαδρομής που αντιμετωπίζει ένας εργαζόμενος ο οποίος θέλει να μεταβεί από το σπίτι του στον χώρο εργασίας του, σε ένα διάστημα 80 ημερών. Σαν δεδομένα, μας παρέχεται ένα αρχείο το οποίο περιέχει τους δρόμους της πόλης στην οποία ζεί, με πληροφορίες για τα Nodes τα οποία ενώνει ο κάθε δρόμος, το μήκος του κάθε δρόμου, καθώς και ο βαθμός της κίνησης που επικρατεί στον κάθε δρόμο.

Βοηθητικές Συναρτήσεις

0.1 Εξαγωγή δεδομένων

Τα δεδομένα εισόδου, δίνονται σε ένα αρχείο το οποίο έχει μια μορφή που μοιάζει με XML αρχείο. Το διαβάζουμε με έναν XML Parser και ξεχωρίζουμε τα source και destination σε δυο μεταβλητες με την συνάρτηση `parse_sourcedest(sample)`. Έπειτα, με την συνάρτηση `parse_roads(sample)`, διαβάζουμε τους δρόμους και τα nodes που ενώνει ο κάθε δρόμος και δημιουργούμε για τον κάθε δρόμο και το κάθε ξεχωριστό node που συναντάμε, αντικείμενα "Road" και "Node" αντίστοιχα και στη συνέχεια τα περνάμε στις αντίστοιχες λίστες με αντικείμενα. Παρακάτω βλέπουμε τι περιέχει κάθε ένα από τα αντικείμενα που δημιουργήσαμε:

```
class Road:
    def __init__(self, name, node_a, node_b, weight):
        self.name = name # Road Name
        self.node_a = node_a # First node of the road
        self.node_b = node_b # Second node of the road
        self.weight = int(weight) # Road length (distance)


class Node:
    def __init__(self, name, neighbours):
        self.name = name # Node name
        self.neighbours = neighbours # A list containing names of node neighbours
        self.weight = None # Used to store road weight for our final paths
        self.previous = None # Used to store parent node for itterations
        self.road_name = None # Used to store road name for our final paths
        self.heuristic = 0 # Used to store calculated node heuristics inside the algorithms
```

0.2 Υπολογισμός κόστους

Δημιουργήσαμε μια συνάρτηση υπολογισμού κόστους, η οποία λαμβάνει ως δεδομένα, τη λίστα με τους δρόμους, δυο κόμβους και τα δεδομένα για την κίνηση στους δρόμους και επιστρέφει τον δρόμο που ενώνει τους δύο κόμβους και έχει το λιγότερο βάρος (σσ. Το βάρος υπολογίζεται από την απόσταση σε συνδιασμό με την κίνηση).

Μέρος Α

Στο πρώτο μέρος της άσκησης, καλούμαστε να υλοποιήσουμε δύο διαφορετικούς αλγόριθμους "offline" αναζήτησης της βέλτιστης διαδρομής, που θα "τρέχουν" κάθε βράδυ και θα επιλέγουν την διαδρομή που θα ακολουθίσει ο εργαζόμενος το επόμενο πρωί.

Α.2 Αλγόριθμος Dijkstra

Για το πρώτο μερος της άσκησης, μας ζητήθηκε να επιλέξουμε έναν αλγόριθμο απληροφόρητης αναζήτησης της επιλογής μας. Επιλέξαμε τον αλγόριθμο Dijkstra καθώς είναι ένας από τους βασικούς αλγόριθμους εύρεσης βέλτιστης διαδρομής. Είναι ένας άπληστος αλγόριθμος που αναζητεί τοπικά ελάχιστα σε κάθε προσπέλαση και οδηγείται έτσι σε ολικό ελάχιστο.

Ο αλόριθμος Dijkstra καλείται κάθε "βραδυ" και υπολογίζει τη διαδρομή που πρέπει να ακουλουθίσει ο εργαζόμενος, το επόμενο πρωί. Λαμβάνει ως δεδομένα τις λίστες με τους δρόμους και τους κόμβους, την κίνηση (για τον υπολογισμό του βάρους) καθώς και τον αρχικό και τον τελικό κόμβο. Ακολουθίσαμε την απλή υλοποίηση του αλγορίθμους, όπως φαίνεται παρακάτω.

```
1  function Dijkstra(Graph, source):
2
3      create vertex set Q
4
5      for each vertex v in Graph:
6          dist[v] <- INFINITY
7          prev[v] <- UNDEFINED
8          add v to Q
9      dist[source] <- 0
10
11     while Q is not empty:
12         u <- vertex in Q with min dist[u]
13
14         remove u from Q
15
16         for each neighbor v of u:  // only v that are still in Q
17             alt <- dist[u] + length(u, v)
18             if alt < dist[v]:
19                 dist[v] <- alt
20                 prev[v] <- u
```

```
21
22      return dist[], prev[]
```

Αλγόριθμος 1: Dijkstra's algorithm - Wikipedia

A.3 Αλγόριθμος IDA*

Στη συνέχεια, μας ζητήθηκε να υλοποιήσουμε έναν αλγόριθμο Iterative Deepening A*. Ο αλγόριθμος αυτός, σε κάθε προσπέλαση, ελέγχει το "heuristic" του επόμενου κόμβου και με τον τρόπο αυτό, επιτυγχάνει πολύ μεγάλη μείωση στους συνολικούς κόμβους που επισκέπτεται. Ως heuristic, ορίζουμε την απόσταση του κόμβου αυτού, από τον τελικό κόμβο μας και πρέπει πάντα να παίρνει τιμή ίση ή μικρότερη της ελάχιστης πραγματικής τιμής της απόστασης. Σε περίπτωση που τα heuristics πάρουν μεγαλύτερη από αυτή την τιμή, τότε ενδέχεται ο αλγόριθμος να αποκλείσει κάποιο βέλτιστο μονοπάτι λόγω του threshold. Για να έχουμε αρκετά ακριβή heuristics στο πρόγραμα μας, χρησιμοποιήσαμε τον αλγοριθμο Dijkstra που υλοποιήσαμε στο προηγούμενο κομμάτι με αφετηρία τον κάθε κόμβο και στόχο τον τελικό κόμβο.

Threshlold ορίζουμε το ανώτατο όριο heuristic που ο αλγόριθμος θεωρεί αποδεκτό, για την τρέχουσα προσπέλαση. Σε κάθε προσπέλαση που δεν βρεί estimate (weight + heuristic) μικρότερο του threshold, ορίζει ως καινούργιο threshold το ελάχιστο από τα estimate που θα συναντήσει. Στο πρόγραμμα μας, όταν ενημερώνουμε το threshold, προσθέτουμε ένα επιπρόσθετο μικρό ποσοστό του estimate, ώστε να μειωσουμε τις προσπελάσεις. Το παραπάνω, όμως, ενδέχεται να κοστίσει σε ακρίβεια αποτελέσματος.

```
1  path              current search path (acts like a stack)
2  node              current node (last node in current path)
3  g                 the cost to reach current node
4  f                 estimated cost of the cheapest path (root..node..goal)
5  h(node)           estimated cost of the cheapest path (node..goal)
6  cost(node, succ)  step cost function
7  is_goal(node)     goal test
8  successors(node)  node expanding function, expand nodes ordered by g + h
9  ida_star(root)    return either NOT_FOUND or a pair with the best path
       and its cost
10
11 procedure ida_star(root)
12     bound := h(root)
13     path := [root]
14     loop
15         t := search(path, 0, bound)
16         if t = FOUND then return (path, bound)
17         if t = inf then return NOT_FOUND
18         bound := t
19     end loop
20 end procedure
21
22 function search(path, g, bound)
23     node := path.last
```

```
    f := g + h(node)
    if f > bound then return f
    if is_goal(node) then return FOUND
    min := inf
    for succ in successors(node) do
        if succ not in path then
            path.push(succ)
            t := search(path, g + cost(node, succ), bound)
            if t = FOUND then return FOUND
            if t < min then min := t
            path.pop()
        end if
    end for
    return min
end function
```

Αλγόριθμος 2: Iterative deepening A* - Wikipedia

Μέρος Β

Για το δεύτερο μέρος της εργασίας, ζητήθηκε να υλοποιήσουμε έναν online αλγόριθμο αναζήτησης Learning Real-Time A* (LRTA*).

Β.1 Αλγόριθμος LRTA*

Ο αλγόριθμος LRTA*, όπως και ο IDA*, είναι παραλαγγή του A* αλγόριθμου και συνεπώς είναι αρκετά παρόμοιοι. Η κύρια διαφοροποίηση τους, είναι πως ο LRTA*, αρχικά, έχει μηδενικά heuristics, τα οποία όμως ενημερώνονται, μετά από κάθε προσπέλαση.

Πιο συγκεκριμένα, ανανεώνουμε το heuristic του τωρινού κόμβου (σε κάθε προσπέλαση) με την τιμή του ελάχιστου estimate, το οποίο είναι μόνιμα μικρότερο του πραγματικού κόστους, καθώς χρησιμοποιούμε low traffic. Παρακάτω παραθέτεται ο αλγόριθμος που μας δώθηκε στο μάθημα και χρησιμοποιήσαμε για την υλοποίηση μας.

```
function LRTA*-AGENT(s') returns an action
    inputs: s'          ,a percept that identifies the current state
    persistent: result  ,a table, indexed by the state and action,
   initially empty
             H          ,a table of cost estimates indexed by state,
   initially empty
             s, a       ,the previous state and action, initially null

    if GOAL-TEST(s') then return stop
    if s' is a new state(not in H) then H[s'] <- h(s')
    if s is not null
        result[s,a] <- s'
        H[s] <- min LRTA*-COST(s,b,result[s,b],H)
```

```
12      a <- an action b in ACTIONS(s') that minimizes LRTA*-COST(s',b,
        result[s',b], H)
13      return a
14
15      function LRTA*-COST(s,a,s',H) returns a cost estimate
16          if s' is undefined then return h(s)
17          else return c(s,a,s') + H[s']
```

Αλγόριθμος 3: Learning Real-Time A* - Lectures

Παρατηρήσεις

Με βάση τη θεωρία, γνωρίζουμε πως ο Dijkstra θα βρίσκει πάντα το βέλτιστο μονοπάτ, οπότε θα έχει και πάντα τη σωστή τιμή απόστασης για το ανάλογο predicted traffic. Παρατηρούμε πως ο IDA*, βρίσκεται πολύ κοντά στις τιμές του Dijkstra για την μέση απόσταση, χρησιμοποιώντας λιγότερους κόμβους για να το βρεί (Δίνοντας του όμως πολύ καλά heuristics). Η διαφορά αυτή, φαίνεται ακόμα καλύτερα στον χρόνο εκτέλεσης, όπου ο IDA*, εκτελείται περίπου 7 φορές πιο γρήγορα.

Για τον LRTA* αλγόριθμο βλέπουμε πως έχουμε μια απόκλιση στο κόστος του μονοπατιού που επιλέγει, που δείχνει το learning curve του αλγορίθμου. Ήδη μετά τα πρώτα iteration, έχει διορθώσει αρκετά τα heuristics, ώστε να μας δίνει βέλτιστα, ή κοντά σε αυτά, μονοπάτια. Όσον αφορά τους επισκεπτόμενους κόμβους αλλά και τον χρόνο εκτέλεσης, βρίσκεται πολύ κοντά στον IDA*.

Average Distance	Dijkstra	IDA*	LRTA*
sampleGraph1	111.05	111.545	162.13
sampleGraph2	184.75	186.44	241.64
sampleGraph3	97.32	97.83	122.04

Average Visited	Dijkstra	IDA*	LRTA*
sampleGraph1	39	13.4	6.18
sampleGraph2	31.85	24.35	7.05
sampleGraph3	32.56	17.13	5.7