Τεχνητή Νοημοσύνη

Αναφορά 1ου project

Παντελιδάκης Μηνάς 2011030110

Στόχος του project ήταν ο πειραματισμός με διαφορετικούς path finding αλγορίθμους.

Συγκεκριμένα έπρεπε να υλοποιηθούν τρεις αλγόριθμοι :

- Ένας απληροφόρητος αλγόριθμος αναζήτησης
- Ο αλγόριθμος πληροφορημένης αναζήτησης IDA*
- Ο αλγόριθμος online πληροφορημένης τοπικής αναζήτησης LRTA*

Το project αναπτύχθηκε σε JAVA 8

Η **υλοποίηση** που ακολουθήθηκε είναι η ακόλουθη:

Διάβασμα του αρχείου εισόδου . Parsing με regular expressions για τη δημιουργία μιας λίστας με αντικείμενα τύπου Road. (Κάτι κακό στο θέμα μνήμης αλλά, κάνει τη διαχείριση των δρόμων πολύ πιο εύκολη).

Μέσα από αυτή τη λίστα , δημιουργείται ένας γράφος που περιέχει κόμβους, κάθε ένας εκ των οποίων περιέχει μια λίστα με Edges (όλοι οι δρόμοι που συνδέονται με αυτόν).Κάθε edge περιέχει πληροφορία για τον κόμβο στον οποίο καταλήγει, το όνομα του δρόμου, το φυσικό, το προβλεπόμενο και το πραγματικό κόστος διάνυσης του. Το προβλεπόμενο και το πραγματικό κόστος μεταβάλλονται από μέρα σε μέρα, ενώ το φυσικό κόστος παραμένει αμετάβλητο.

Η δημιουργία ολόκληρου γράφου είναι κακή υλοποίηση από μεριάς μνήμης, αλλά ξεκίνησα έτσι και δεν πρόλαβα να υλοποιήσω κάτι πιο έξυπνο. Μελλοντική βελτίωση του κώδικα είναι να μην φτιάχνεται γράφος σύμφωνα με τα δεδομένα εισόδου , αλλά κάθε node που χρειάζεται κατά την εκτέλεση των αλγορίθμων να δημιουργείται εκείνη τη στιγμή από τα parsed δεδομένα. Σίγουρα θα χρειαστούν πιο πολλές αναζητήσεις μέσα στα δεδομένα κατά την εκτέλεση των αλγορίθμων, αλλά όλη αυτή η μνήμη που δεσμεύεται για τη δημιουργία ολόκληρου του γράφου θα είναι πλέον ελεύθερη.

Τελικά μέσα από αυτή τη διαδικασία προκύπτει ένας μη κατευθυνόμενος κυκλικός γράφος

Απληροφόρητη offline Αναζήτηση :

Για την απληροφόρητη αναζήτηση επιλέχθηκε ο αλγόριθμος

Iterative Deepening Depth-First search a.k.a IDDFS καθώς συνδυάζει τα καλύτερα στοιχεία των depth-first και breadth-first search. Όπως γράφεται στο wiki : IDDFS combines depth-first search's space-efficiency and breadth-first search's completeness .

Ο IDDFS είναι depth-first search με όριο εκβάθυνσης. Το όριο εκβάθυνσης για κάθε iteration αυξάνεται έως ότου βρεθεί ο κόμβος στόχου. Πρακτικά υλοποιεί breadth-first search αλλά

χρησιμοποιεί πολύ λιγότερη μνήμη. Σε κάθε iteration επισκέπτεται τους κόμβους με τη σειρά που θα τους επισκεπτόταν ένα depth-first search, αλλά συνολικά η σειρά με την οποία επισκέπτεται τους κόμβους είναι breadth-first search.

Επίσης από το wiki : Since iterative deepening visits states multiple times, it may seem wasteful, but it turns out to be not so costly, *since in a tree most of the nodes are in the bottom level, so it does not matter much if the upper levels are visited multiple times*.

Το *παραπάνω* δεν ισχύει σε γράφο, οπότε έφτιαξα ένα είδος μνήμης για να αποφύγω μερικά από τα περιττά expands. Τα αποτελέσματά του ήταν σχετικά καλά. Λόγω του περιοριστικού παράγονται επιτρεπόμενου βάθους , αποφεύγει τα πολύ βαθιά μονοπάτια τα οποία θα εξερευνούσε ο απλός depth-first search αλγόριθμος.

Ο ψευδοκώδικας που υλοποιήθηκε :

```
function IDDFS(root)
for depth from 0 to ∞
found ← DLS(root, depth)
if found ≠ null
return found

function DLS(node, depth)
if depth = 0 and node is a goal
return node
if depth > 0
foreach child of node
found ← DLS(child, depth−1)
if found ≠ null
return found
return null
```

Σε αυτόν έκανα μια ελαφριά τροποποίηση για να αποφύγω μερικά αχρείαστα expands. Δυστυχώς όμως δεν κατάφερα να τα εξαλείψω όλα. Δείτε το @brief του αρχείου IDDFS.

Αξίζει να σημειωθεί πως η σειρά με την οποία γίνονται expand οι κόμβοι , δεν είναι προκαθορισμένη, σε αντίθεση με συνηθισμένες υλοποιήσεις που κάνουν expand από αριστερά προς τα δεξιά ή αντίστροφα. Λόγω αποθήκευσης των κόμβων προς επέκταση σε ένα hashmap (στο οποίο η σειρά προσπέλασης δεν είναι η σειρά εισαγωγής), υπάρχει ένα randomizer στον κόμβο που επιλέγεται να γίνει expand κάθε φορά, κάτι το οποίο βοηθάει στην απόδοση του αλγορίθμου στη γενική περίπτωση.

Πληροφορημένη offline αναζήτηση με IDA*

Ο IDA* είναι στην ουσία iterative deepening depth-first search που δανείζεται την ιδέα της χρήσης ευρετικής συνάρτησης h για την εκτίμηση της απόστασης του εκάστοτε κόμβου από το στόχο, από τον A*. Συγκεκριμένα αντί να χρησιμοποιεί ένα επιτρεπτό βάθος σε κάθε iteration του depth-limited search , χρησιμοποιεί σαν όριο το minimum f = g+h από τα f των κόμβων που ξεπέρασαν το εκάστοτε όριο.

Ο ψευδοκώδικας που υλοποιήθηκε :

```
node                current node
g                   the cost to reach current node
f                           estimated cost of the cheapest path
(root..node..goal)
h(node)             estimated cost of the cheapest path (node..goal)
cost(node, succ)    step cost function
is_goal(node)       goal test
successors(node)    node expanding function, expand nodes ordered by
g + h(node)

procedure ida_star(root)
bound := h(root)
loop
t := search(root, 0, bound)
if t = FOUND then return bound
if t = ∞ then return NOT_FOUND
bound := t
end loop
end procedure

function search(node, g, bound)
f := g + h(node)
if f > bound then return f
if is_goal(node) then return FOUND
min := ∞
for succ in successors(node) do
t := search(succ, g + cost(node, succ), bound)
if t = FOUND then return FOUND
if t < min then min := t
end for
return min
end function
```

Όπως γράφεται στο wiki : Since it is a depth-first search algorithm, its memory usage is lower than in A*, but unlike ordinary iterative deepening search, it concentrates on exploring the most promising nodes and thus does not go to the same depth everywhere in the search tree. Unlike A*, IDA* does not utilize dynamic programming and therefore often ends up exploring the same nodes many times.

IDA* is beneficial when the problem is memory constrained. A* search keeps a large queue of unexplored nodes that can quickly fill up memory. By contrast, because IDA* does not remember any node except the ones on the current path, it requires an amount of memory that is only linear in the length of the solution that it constructs.

Προφανώς για τη σωστή λειτουργία του αλγορίθμου χρειάζεται μια ευρετική συνάρτηση η οποία είναι συνεπής (για αναζήτηση σε γράφο) , δηλαδή τηρεί την τριγωνική ανισότητα. Οπότε πρέπει να ισχύει το ακόλουθο :

$h(n) \leq \text{cost}(n, n') + h(n')$ όπου n' successor του n.

Μια συνεπής ευρετική συνάρτηση είναι εύκολο να αποδειχθεί πως είναι και παραδεκτή, ότι δηλαδή δεν υπερεκτιμά το κόστος προσπέλασης του στόχου. Η απόδειξη υπάρχει στο wiki.

Δημιουργία κατάλληλης ευρετικής :

Πειραματίστηκα με διάφορες ευρετικές συναρτήσεις. Κατέληξα στο ότι η εφαρμογή της ίδιας φιλοσοφίας καθώς αλλάζει το πρόβλημα, δεν είναι καλή λύση.

Π.χ. ευρετικές που δούλευαν καλά για το SampleGraph1 , δε δούλευαν καλά στο SampleGraph2 κ.ο.κ.

Για να υπολογίσω την ευρετική χρησιμοποίησα κάτι σαν IDDFS . Αντί για depth , χρησιμοποίησα μια μεταβλητή levelOfConnection. Αυτή η μεταβλητή, όπως υποδηλώνει και το όνομά της, αποθηκεύει το επίπεδο σύνδεσης του εκάστοτε κόμβου από το στόχο.

Π.χ. αν ένας κόμβος είναι γείτονας του κόμβου στόχου, έχει levelOfConnection=1 , αν είναι γείτονας του γείτονα του κόμβου στόχου έχει levelOfConnection =2. Για κάθε κόμβο, η ανάθεση της ευρετικής γίνεται για το χαμηλότερο levelOfConnection, δηλαδή εάν ο κόμβος είναι γείτονας του στόχου και ταυτόχρονα γείτονας του γείτονα του στόχου , το levelOfConnection του είναι 1 και όχι 2.

Ξεκινώντας λοιπόν από το στόχο , του οποίου το h=0 είναι το μόνο που γνωρίζουμε, βρίσκω για κάθε κόμβο το επίπεδο σύνδεσης. Αν το επίπεδο σύνδεσης του εκάστοτε κόμβου με τον κόμβο στόχου είναι μεγαλύτερο του 50 , το εκάστοτε instance της αναδρομής τερματίζει.(Χρειάζομαι μια συνθήκη τερματισμού γιατι αλλιώς θα κάνω κύκλους το γράφο.)

Χρησιμοποιώ ένα hashmap, το οποίο περιέχει για κάθε κόμβο το levelOfConnection. Αν λοιπόν φτάσω στον κόμβο αυτό και το hashmap για το κλειδί αυτό δεν έχει πάρει τιμή , αποθηκεύω τον κόμβο (κλειδί) μαζί με το levelOfConnection (value) στο hashmap. Έπειτα δίνω τιμή στο h του κόμβου. Αν οποιαδήποτε στιγμή ξαναφτάσω στον κόμβο αυτό με χαμηλότερο levelOfConnection , ο κόμβος αυτός παίρνει εκ νέου τιμή στην ευρετική του.

Η ευρετική συνάρτηση είναι η εξής :

```
tempHeur=nodeData.getH()+ 0.25*tempEntry.getValue().getPredictionCost();
```

Πρακτικά, το Η του κόμβου που με έστειλε εκεί + 0.25*προβλεπόμενο κόστος μονοπατιού.

Ο δρόμος που ακολουθείται για να πάω από έναν κόμβο σε έναν άλλο, αν υπάρχουν πάνω από ένας δρόμοι που συνδέουν τους 2 κόμβους, είναι αυτός με το καλύτερο προβλεπόμενο κόστος για κάθε μέρα. (Η ευρετική υπολογίζεται καθημερινά).

Πληροφορημένη online τοπική αναζήτηση με LRTA* :

Ένας πράκτορας που πραγματοποιεί online αναζήτηση διαφέρει αρκετά από έναν πράκτορα offline αναζήτησης.

Στην online αναζήτηση, ο πράκτορας χρησιμοποιεί την αντίληψή του για την κατάσταση στην οποία βρίσκεται για να ενισχύσει το χάρτη του περιβάλλοντός του. Ο πράκτορας μπορεί να επεκτείνει μόνο κόμβους στους οποίους βρίσκεται, οπότε βγάζει νόημα οι κινήσεις να είναι τοπικές.

Περιγραφή αλγορίθμου LRTA* :

Η βασική ιδέα είναι να αποθηκεύουμε μία προσωρινά καλύτερη εκτίμηση H(s) του κόστους για να φτάσουμε στο στόχο από κάθε κατάσταση s που έχουμε επισκεφθεί. Το H(s) αρχικά παίρνει την τιμή της ευρετικής h(s) και γίνεται update καθώς ο πράκτορας αποκτά εμπειρία στο χώρο καταστάσεων. Αυτή η στρατηγική έχει σαν αποτέλεσμα να μην παγιδεύεται ο πράκτορας σε πιθανά τοπικά ελάχιστα.

Ο ψευδοκώδικας που υλοποιήθηκε :

function LRTA*-AGENT(s') **returns an** action
  inputs: s', a percept that identifies the current state
  static: *result*, a table, indexed by action and state, 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** $H[s'] \leftarrow h(s')$
  unless s is null
    $result[a, s] \leftarrow s'$
    $H[s] \leftarrow \min_{b \in \text{ACTIONS}(s)}$ LRTA*-COST($s, b, result[b, s], H$)
  a $\leftarrow$ an action b in ACTIONS(s') that minimizes LRTA*-COST($s', b, result[b, s'], H$)
  $s \leftarrow s'$
  return a

function LRTA*-COST(s, **a**, s', H) **returns** a cost estimate
  if s' is undefined **then return** $h(s)$
  else return $c(s, \mathbf{a}, s') + H[s']$

Το path που επιστρέφει ο LRTA* δεν είναι το βέλτιστο μονοπάτι προς το στόχο, αλλά το μονοπάτι που ακολούθησε ο πράκτορας μέχρι να φτάσει στο στόχο. Το βέλτιστο μονοπάτι θα προκύψει μέσα από τον πίνακα result, ακολουθώντας τους δρόμους που μας πηγαίνουν στις καταστάσεις με το καλύτερο H κάθε φορά.

Σχετικά με την πιθανοτική κατανομή των predictions :

Στην παρούσα υλοποίηση τα predictions για τους δρόμους αντιμετωπίζονται το καθένα ανεξάρτητα από το άλλο. Κάθε μέρα, αποδίδεται 0.6 πιθανότητα στο να είναι το actual cost αυτό που δόθηκε σαν predicted cost και 0.4 να είναι ένα από τα άλλα 2.

Το σωστό θα ήταν να γίνει πιθανοτική ανάλυση.

Υπήρξε μια σκέψη για υλοποίηση χρησιμοποιώντας το θεώρημα του bayes.

Ενδεικτικά :

high=P(High|predictionHigh)*P(High)/(P(High|predictionHigh)*P(High)+P(High|predictionNotHigh)*P(predictionNotHigh))

Τελικά PredictedRoadCost=high*0,6*1.25*NaturalCost

Αντίστοιχα και τα normal και low.

Δυστυχώς δεν υπήρξε χρόνος για αυτήν την υλοποίηση.

Καταλήγοντας : Ο κώδικας έχει πολύ περιθώριο για βελτίωση. Η υλοποίησή μου χρησιμοποιεί αρκετή extra μνήμη και είναι ακατάλληλη για αρχεία εισόδου με πολύ μεγάλο πλήθος κόμβων.

Σχετικά με τις απαιτήσεις μνήμης των αλγορίθμων, οι συναρτήσεις την κλάσης runtime της java δεν προσέφεραν κάτι κατάλληλο, καθώς το μόνο που μπορούσα να πάρω από αυτές ήταν η συνολική μνήμη του pc, και η μνήμη που χρησιμοποιούταν εκείνη τη στιγμή από όλες τις διεργασίες του συστήματος , και όχι αποκλειστικά από τον κώδικα.

Η ευρετική συνάρτηση που υλοποίησα αποδίδει σχετικά καλά, αλλά δε λαμβάνει υπόψιν αρκετά metrics. Οι αλγόριθμοι τρέχουν και για τα 3 αρχεία που μας δόθηκαν, αλλά αυτοί που χρησιμοποιούν ευρετική συνάρτηση ίσως μπουν σε ατέρμονα loops για άλλα αρχεία εισόδου.

Υπάρχουν αρκετά σχόλια στον κώδικα για την αποσαφήνιση της σκέψης μου.

Έτοιμοι κώδικες δε χρησιμοποιήθηκαν.