Αντωνία Τσίλη 1115201400209
Μερκούρης Παπαμιχαήλ 1115201400148

Στα παραδοτέο αρχείο περιλαμβάνονται τα εξής:

Κατάσταση (State)

- data State = State Grid PointColoring InverseLineSegmentColoring
    - data Grid = Grid (Int, Int)
    - data PointColoring = PointColoring (Map ((Int, Int), (Int, Int)) char)
    - data InverseLineSegmentColoring (Map Char ((Int, Int), (Int, Int))

Ο τύπος δεδομένων State περιγράφει μια κατάσταση ως μια τριάδα, ενός πλέγματος (Grid), ενός χρωματισμού των σημείων του πλαισίου (PointColoring) και ενός αντίστροφου χρωματισμού των ευθύγραμμων τμημάτων (αμαξιδίων) του πλέγματος.

Το πλέγμα (Grid), περιγράφεται μόνο από την πάνω δεξιά άκρη του (edge).
$Grid = \{(x, y) \subseteq N^2 | 1 \leq x \leq edge.x, 1 \leq y \leq edge.y\}$
Έχει γίνει η παραδοχή ότι είναι ορθογώνιο παραλληλόγραμμο, συνεπώς αρκεί για να γνωρίζουμε εάν έχουμε βγει εκτός από αυτό το πάνω δεξιά άκρο του. Το κάτω αριστερά άκρο του είναι το (1,1). (Το (0,0) θεωρείται ότι δεν ανήκει στο πλέγμα.)

Ο χρωματισμός σημείων του πλέγματος (PointColoring) αναπαριστά μια συνάρτηση:
$c : Grid \rightarrow Char \setminus \{'\backslash n'\}$,
όπου Char το σύνολο των χαρακτήρων που υποστηρίζει η Haskell. Ο χαρακτήρας '.' δεσμεύεται για να υποδεικνύει το αχρωμάτιστο σημείο (δεν έχει κάποιο αμαξίδιο πάνω του), ενώ ο '=' για το "κόκκινο" αμαξίδιο, το οποίο πρέπει να απεγκλωβιστεί.

Ο αντίστροφος χρωματισμός ευθύγραμμων τμημάτων (InverseLineSegmentColoring) αναπαριστά μια συνάρτηση
$i : LineSegment \rightarrow Char$,
όπου LineSegment είναι το σύνολο των ευθύγραμμων τμημάτων (αμαξιδίων) που βρίσκονται πάνω στο πλέγμα. Τα ευθύγραμμα τμήματα, σαν υποπερίπτωση των ορθογώνιων παραλληλογράμμων, εφόσον έχουν μόνο δύο προσανατολισμούς (παράλληλα στον άξονα x, ή στον άξονα y), μπορούν να αναπαρασταθούν με την χρήση μόνο δύο σημείων, έστω το ευθύγραμμο τμήμα l, τότε, αν s το ένα άκρο του l και e άλλο, τότε:
$l = \{(x, y) \subseteq N^2 | s.x \leq x \leq e.x, s.y \leq y \leq e.y\}$.

- Βασικές συναρτήσεις
    - readState
    - writeState

Η readState, δουλεύει όπως περιγράφεται στην εκφώνηση. Είναι σχετικά απλή η διαδικασία που ακολουθείται ώστε να δημιουργηθούν οι δομές Grid και PointColoring. Κάποιο ενδιαφέρον παρουσιάζει η InverseLineSegmmentColoring, όπου δημιουργείται με την βοήθεια της δομής InversePointColoring και της συνάρτησης grid_traverse, η οποία "διαβάζει" το πλέγμα από αριστερά προς τα δεξιά. Εφόσον πολλά σημεία στο πλέγμα έχουν το ίδιο χρώμα δεν είναι η PointColoring ισομορφισμός, ώστε να μπορούμε να βρούμε την αντίστροφη συνάρτηση. Με χρήση των συναρτήσεων \new old -> old, \new old -> new, μπορούμε καθώς απομακρυνόμαστε από το (1, 1), να κρατάμε είτε το πρώτο σημείο κάθε χρώματος, είτε το τελευταίο. 'Ετσι δημιουργούμε δύο αντίστροφους χρωματισμούς σημείων

(InversePointColoring), από τους οποίους προκύπτει ο αντίστροφος χρωματισμός ευθύγραμμων τμημάτων (InverseLineSegmmentColoring).

Η writeState είναι η αντίστροφή από την readState και ουσιαστικά αποτυπώνει τον χρωματισμό σημείων (PointColoring) της κατάστασης (State) πίσω σε συμβολοσειρά

Κίνηση (Move)

- data Move = Move Char Int

Ο τύπος δεδομένων Move περιγράφεται από το χρώμα και το πλήθος των βημάτων (σημείων μετακίνησης της άκρης) του αμαξιδίου. Ο αριθμός μπορεί να είναι θετικός ή και αρνητικός ανάλογα με τη φορά της κίνησής του.

- Βασικές συναρτήσεις:
    - successorMoves
    - makeMove

Η πρώτη από αυτές καλεί τη legalmoves, η οποία δίνει μία λίστα από νόμιμες κινήσεις για το κάθε αμαξίδιο. Αυτές υπολογίζονται σύμφωνα με την κατεύθυνση (data Direction) του αμαξιδίου και τον ελεύθερο χώρο (data Space) που υπάρχει μεταξύ των γειτονικών του αμαξιδίων (margin_x και margin_y ορίζουν κάθε φορά τα όρια μεταξύ των οποίων μπορεί το αμάξι να κινηθεί στον άξονά του, οριζόντιο ή κατακόρυφο αντίστοιχα). Οι κινήσεις χωρίζονται σε αριστερά/δεξιά και πάνω/κάτω ανάλογα με την κατεύθυνση του αμαξιού. Επιπλέον, έχουν ορισθεί και χρησιμοποιηθεί συναρτήσεις με όνομα get_* <args> για τη χρήση συγκεκριμένου μέρους της παραμέτρου <args>.
Η συνάρτηση makeMove μετατρέπει τα mappings των σημείων και των χρωμάτων σε λίστες και αλλάζει το χρώμα των σημείων (x+steps,y) ή (x,y+steps) ώστε να έχουν εκείνο του αμαξιού που κινήθηκε και τα σημεία της θέσης που βρισκόταν το αμάξι να γίνουν empty (color == '.').

Επίλυση (Solve)

- data PriorityQueue t = PriorityQueue (Map Int [t])

Η δομή ουράς προτεραιότητας (PriorityQueue) υλοποιεί έναν δυαδικό σωρό με την βοήθεια της δομής Map της Haskell. Η ουρά προτεραιότητας αντιστοιχεί μια προτεραιότητα Int με μια fifo ουρά από στοιχεία τύπου t που έχουν την ίδια προτεραιότητα· σε περίπτωση ίσης προτεραιότητας επιλέγει στοιχεία fifo.

- Βασικές συναρτήσεις
    - heuristic
    - solve
    - solve_astar

Η υλοποίηση της συνάρτησης heuristic βασίστηκε στην ιδέα που περιγράφεται στο https://github.com/atheed/UnblockMe-Solver/blob/master/unblockme.py. Πρώτα βρίσκεται μία λίστα με όλα τα σημεία μεταξύ του αμαξιού "==" και του δεξιού άκρου του πλαισίου Grid τα οποία έχουν αμάξια που μπλοκάρουν την έξοδό του. Το τελικό μικρότερο κόστος υπολογίζεται ως το άθροισμα των κινήσεων των αμαξιδίων αυτών που μπλοκάρουν (οι μικρότερες δυνατές κινήσεις υποθέτοντας ότι τα ίδια δεν μπλοκάρονται) και τα βήματα που χρειάζεται το "==" για να φτάσει στη δεξιά άκρη.

Τόσο η solve, όσο και η solve_astar είναι κλήσεις στην solve_BestFirst, με ευριστική συνάρτηση την \x -> 0 και την heuristic αντίστοιχα.

Σημειώσεις:

1. Εάν δοθεί ως είσοδος στην solve ή την solve_astar ένα πλέγμα για το οποίο δεν υπάρχει λύση το πρόγραμμα θα πέσει σε ατέρμον βρόγχο.
2. Έχουν δοκιμαστεί και δουλεύουν σωστά όλα τα παραδείγματα που δίνονται στην εκφώνηση.
3. Το πρόγραμμα εκτελείται με την εντολή > ghci ./rush_hour.hs στον φάκελο της εφαρμογής