ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΕΙΟ ΑΘΗΝΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΜΑΘΗΜΑ: **ΤΕΧΝΗΤΗ ΝΟΗΜΟΣΥΝΗ**

ΤΙΤΛΟΣ ΕΡΓΑΣΙΑΣ

AI SOKOBAN SOLVER USING A* ALGORITHM

ΟΝΟΜ/ΜΟ:	ΜΑΡΑΚΗΣ ΜΙΧΑΗΛ	ΜΑΚΑΡΟΥΝΑΣ ΠΡΟΔΡΟΜΟΣ-ΑΡΗΣ	ΜΠΑΛΛΑΣ ΑΡΙΣΤΕΙΔΗΣ
ΕΤΟΣ:	3ο	3ο	3ο
ΑΡ. ΜΗΤΡΩΟΥ:	p3230267	p3230111	p3230130

Πίνακας περιεχομένων

Class Node 2

Class Main 2

Αλγόριθμος A* 2

Βασικές Μέθοδοι 3

Βοηθητικές Μέθοδοι 5

Υπολογισμός Ευρετικής Συνάρτησης 7

Σύγκριση κατανάλωσης μνήμης BFS VS IDS 8

Class Node

Κάθε αντικείμενο της κλάσης Node αναπαριστά μια κατάσταση του παιχνιδιού. Δηλαδή έχει ως ιδιότητες τις συντεταγμένες του παίχτη, την προσομοίωση της πίστας με την αλλαγή που υπέβαλλε το Node, τον γονιό του, το κόστος h , g και f = g + h. Έχει υλοποιηθεί ο κατασκευαστής και οι μέθοδοι **equals(…)** και **hashcode(…)** για την σωστή λειτουργία και βελτιστοποίηση του hashSet. Επίσης υπάρχει και μια μέθοδος **print()**

Class Main

Περιέχεται όλος ο κώδικας του project εκτός από τον αντίστοιχο της class Node. Η μέθοδος **public static void main(String[] args)** απλά καλεί την μέθοδο **AstarAlgorithm()**. Επίσης η εκτέλεση της μεθόδου **AstarAlgorithm()** καλύπτεται από try-catch block για το exception case : **OutOfMemory Error** καθώς πολλές δύσκολες και μεγάλες πίστες μπορεί να απαιτούν περισσότερη μνήμη από ότι υπάρχει διαθέσιμη.

Αλγόριθμος A*

Ο αλγόριθμος έχει υλοποιηθεί στην μέθοδο **AstarAlgorithm().** Αρχικά ζητείται από τον χρήστη να επιλέξει το επίπεδο πίστας που επιθυμεί το AI Sokoban Solver να εξετάσει αν υπάρχει λύση. Τα επίπεδα 2-4 είναι απο την δοσμένη σελίδα github που υπάρχουν αρχικές καταστάσεις Sokoban απο την εκφώνηση της άσκησης. Αφού ο χρήστης επιλέξει το επίπεδο δυσκολίας τότε τρέχει ο αλγόριθμος τρέχει μέχρι να βρει λύση ή να καταλήξει ότι δεν μπορεί να λυθεί. Αν υπάρχει λύση στο πρόβλημα τότε θα εκτυπωθεί κάθε βήμα που έκανε ο παίχτης στο μονοπάτι για να καταλήξει στην τελική κατάσταση. Παράλληλα εκτυπώνεται και το κόστος f και h που υπάρχει σε κάθε κατάσταση. Είναι σημαντικό να σημειωθεί πως όταν επιλεγεί η πίστα από τον χρήστη, τότε αυτόματα δημιουργείται ένα περίβλημα από τοίχους(#) και η πίστα βρίσκεται πλέον σε ένα ορθογώνιο παραλληλόγραμμο **ΧΩΡΙΣ** να έχει επηρεαστεί η δομή της πίστας. Σε αυτό βοηθάει η μέθοδος **makeRectangularWithBorder(char[][] grid).**

Αρχικά γίνεται έλεγχος εγκυρότητας της πίστας ελέγχοντας αν υπάρχει ίδιο πλήθος από κουτιά(0) και διαθέσιμους στόχους($) στον χάρτη. Αν δεν υπάρχει, τερματίζει το πρόγραμμα ενώ έχει εμφανιστεί αντίστοιχο μήνυμα. Επίσης γίνεται έλεγχος και throw exception **ArrayIndexOutOfBounds** για την περίπτωση που δεν υπάρχει ακριβώς ένας παίχτης στην πίστα. Στην συνέχεια με την βοήθεια της βοηθητικής μεθόδου **findplayer(level)** εντοπίζονται οι συντεταγμένες του παίχτη και δημιουργείται ο πρώτος κόμβος Node. Αυτός δέχεται σαν ορίσματα τις αρχικές συντεταγμένες του παίχτη, ένα αντίγραφο της πίστας το οποίο προέρχεται από την βοηθητική συνάρτηση **copyGrid(level)**, null στο πεδίο του γονιού, g = 0 καθώς δεν έχει γίνει κανένα βήμα μέχρι τώρα και τον αρχικό υπολογισμό από την ευρετική συνάρτηση h. Για την προσομοίωση του μετώπου έχει χρησιμοποιηθεί PriorityQueue η οποία δίνει προτεραιότητα στον κόμβο με το μικρότερο f, ενώ για το κλειστό σύνολο έχει χρησιμοποιηθεί hashset καθώς εδώ γίνεται έλεγχος αν υπάρχει ακριβώς ένα συγκεκριμένο Node μέσα σε αυτό σε O(1) χρόνο. Το κύριο μέρος του αλγορίθμου τρέχει μέχρι να αδειάσει το μέτωπο. Αρχικά με την μέθοδο .**poll()** παίρνουμε τον επόμενο κόμβο από το μέτωπο και ελέγχουμε αν είναι τελική κατάσταση, ελέγχοντας αν ισχύει h == 0 και ότι δεν βρίσκονται πλέον άλλα ελεύθερα κουτιά και στόχοι στον χάρτη. Ο δεύτερος έλεγχος γίνεται προληπτικά καθώς βασική συνθήκη τελικής

κατάστασης του A* είναι αν h == 0. Αν δεν είναι τελική κατάσταση τότε για κάθε ένα από τα 4 παιδιά του κόμβου(Κίνηση πάνω, Κίνηση δεξιά, Κίνηση κάτω, Κίνηση αριστερά) ελέγχουμε αν είναι έγκυρες κινήσεις και αν προκαλούν deadlock. Αν περάσουν αυτούς τους ελέγχους τότε δημιουργούνται τα αντικείμενα Node με όρισμα τις νέες συντεταγμένες του παίχτη σύμφωνα με την κατεύθυνση που αντιπροσωπεύει το κάθε Node, την ανανεωμένη πίστα , πατέρα ως τον κόμβο current, το κόστος g = current.g + 1 καθώς κάθε κίνηση αυξάνει το κόστος κατά 1 και το κόστος της ευρετικής για αυτήν τη κατάσταση. Τέλος ελέγχεται αν αυτό το συγκεκριμένο Node έχει ξανά- προκύψει, με την βοήθεια του κλειστού συνόλου και αποφασίζουμε αν το κρατήσουμε ή όχι. Στο πέρας του αλγορίθμου, ανεξαρτήτως αν υπάρχει λύση ή όχι, τυπώνεται ο χρόνος σε milliseconds για το πόση ώρα έτρεξε ο αλγόριθμος.

Βασικές Μέθοδοι

- **isValidMove(int row, int col, char[][] board, int direction):** Η γενική ιδέα αυτής της μεθόδου είναι να ελέγξει αν μια συγκεκριμένη κίνηση, που αναπαριστάται από ένα νέο Node, είναι επιτρεπτή από τους κανόνες και τους περιορισμούς του παιχνιδιού. Πιο συγκεκριμένα ο παίχτης δεν μπορεί να μετακινηθεί σε ένα μπλοκ όπου βρίσκεται τοίχος(#). Επίσης ο παίχτης δεν έχει την δυνατότητα να μετακινήσει 2 ή παραπάνω κουτιά (0) τα οποία είναι κολλημένα μεταξύ τους προς την συγκεκριμένη κατεύθυνση η οποία καθορίζεται από το όρισμα direction**.** Επίσης δεν μπορεί να μετακινήσει ένα κουτί προς την κατεύθυνση που υπάρχει τοίχος στο επόμενο κελί. Η μέθοδος δέχεται σαν όρισμα τις συντεταγμένες του παίχτη λαμβάνοντας υπόψη την κίνηση του, την τρέχουσα κατάσταση της πίστας και την κατεύθυνση στην οποία έγινε η υποψήφια κίνηση.

  Αρχικά γίνεται έλεγχος αν στην νέα θέση του παίχτη υπάρχει τοίχος. Εφόσον αυτός ο περιορισμός δεν κόβει την κίνηση, τότε βλέπουμε αν υπάρχει κουτί ή κουτί σε στόχο(*). Αυτό σημαίνει πως ο παίχτης θέλει να μετακινήσει το κουτί. Οπότε λαμβάνοντας υπόψη την κατεύθυνση του παίχτη, ελέγχουμε αν υπάρχει κουτί ή τοίχος στην επόμενη θέση προς εκείνη την κατεύθυνση. Αν και αυτός ο έλεγχος δεν απορρίψει την κίνηση τότε η κίνηση είναι νόμιμη με βάση τους κανόνες και είναι υποψήφια για να γίνει. Επομένως για το συγκεκριμένο Node επιστρέφεται true στον αλγόριθμο A* και στην συνέχεια θα ελεγχθεί αν αυτή η κίνηση προκαλεί deadlock στην μέθοδο **isDeadlock(..)**. Αν η κίνηση δεν είναι έγκυρη και απορριφτεί σε έναν από τους παραπάνω ελέγχους, τότε επιστρέφεται false και το συγκεκριμένο Node σίγουρα δεν θα ληφθεί υπόψη από τον παίχτη. Και στα 2 κελιά που γίνονται έλεγχοι, εξετάζεται ταυτόχρονα και το ενδεχόμενο αν βρίσκονται εκτός ορίων της πίστας.

- **updateGrid(int newRow, int newCol, char[][] grid, int direction):** Βασικός στόχος αυτής της μεθόδου είναι η σωστή αναπαράσταση των πιθανών κινήσεων του παίχτη και η σωστή προσομοίωση των αλλαγών πάνω στην πίστα. Όπως και στην **isValidMove(…)**, έτσι και τώρα, γίνονται κατάλληλοι έλεγχοι για να καλύπτονται οι περιπτώσεις εκτός ορίων πίστας. Η μέθοδος δέχεται σαν όρισμα τις πιθανές ανανεωμένες συντεταγμένες του παίχτη, τον δισδιάστατο πίνακα char[][] level που έχει την αναπαράσταση της πίστας πριν την πιθανή νέα κίνηση και την κατεύθυνση στην οποία επιθυμεί να γίνει η κίνηση. Αρχικά είναι σημαντικό να κρατήσουμε την τρέχουσα θέση του παίχτη. Επομένως σε μεταβλητές κρατάμε τις συντεταγμένες του παίχτη, και το περιεχόμενο του κελιού που ήταν(αν ήταν «1» ή «+»). Στην περίπτωση που ο παίχτης αφήνει πίσω

«1» τότε μετά την κίνηση το κελί θα έχει ως περιεχόμενο πλέον κενό (‘ ‘). Αλλιώς αν αφήνει πίσω «+», αυτό σημαίνει ότι ο παίχτης βρισκόταν πάνω σε στόχο και έτσι μετά την κίνηση το παλιό κελί θα αντικατασταθεί πλέον με «$». Από την άλλη πλευρά είναι εξίσου σημαντικό να αποθηκεύσουμε σε μια μεταβλητή και το περιεχόμενο του κελιού που θα πάει ο παίχτης με την κίνηση από το Node. Σε περίπτωση που η κίνηση τον κατευθύνει σε κενό (‘ ‘) τότε απλά θα αντικατασταθεί το κενό με το σύμβολο του παίχτη (1). Αντίστοιχα, αν ο παίχτης πέσει πάνω σε goal ($) τότε θα αντικατασταθεί με το σύμβολο (+) το οποίο δείχνει πως ο παίχτης βρίσκεται πάνω σε στόχο στο συγκεκριμένο κελί της πίστας. Για τις περιπτώσεις που ο παίχτης πάει να σπρώξει κουτί, δηλαδή το κελί που πρόκειται να πάει ο παίχτης είναι (0) ή (*) τότε αρχικά αποθηκεύονται οι συντεταγμένες και του επόμενου κελιού από αυτό που πρόκειται να καταλήξει ο παίχτης για να δούμε που θα καταλήξει και το κουτί μετά το σπρώξιμο. Αν υπάρχει κενό μετά το κουτί τότε η μετακίνηση γίνεται κανονικά κατά μία θέση προς την κατεύθυνση που σπρώχνει ο παίχτης, ενώ αν υπάρχει στόχος μετά το κουτί τότε γίνεται κανονικά η μετακίνηση αλλά θα αναπαρασταθεί με (*) καθώς βρίσκεται πάνω σε αυτό τον στόχο. Δεν υπάρχει περίπτωση να βρίσκεται κουτί ή τοίχος στο επόμενο κελί από το κουτί καθώς τέτοιες κινήσεις έχουν απορριφτεί από την μέθοδο **isValidMove(…).** Στην περίπτωση που ο παίχτης μετακινεί κουτί από τον στόχο του, τότε προφανώς θα γίνουν αντίστοιχοι έλεγχοι για την επόμενη θέση του κουτιού, αλλά στο σημείο που θα βρίσκεται ο παίχτης, θα βρίσκεται ταυτόχρονα και στόχος, επομένως η αναπαράσταση θα γίνει με το σύμβολο (+). Η μέθοδος επιστρέφει την πίστα με την μετακίνηση που ζητάει το παιδί Node, και γίνεται μέρος του αντικειμένου Node.

- **isDeadlock(int row, int col, char[][] board, int counter)**: Η συγκεκριμένη μέθοδος έχει ίδια ορίσματα με την **isValidMove(…)** αλλά εξυπηρετεί έναν εντελώς διαφορετικό και ιδιαίτερα κρίσιμο σκοπό. Ο ρόλος της είναι να εντοπίζει αν μια κατάσταση στο παιχνίδι οδηγεί σε deadlock**,** δηλαδή σε σημείο όπου ένα ή περισσότερα κιβώτια έχουν τοποθετηθεί με τέτοιο τρόπο ώστε να μην μπορούν πλέον να μετακινηθούν αυτά ή ο παίχτης προς κάποια έγκυρη κατεύθυνση και συνεπώς καθίσταται αδύνατη η ολοκλήρωση του επιπέδου. Αν διαπιστωθεί κάτι τέτοιο, η μέθοδος επιστρέφει true, σηματοδοτώντας ότι το συγκεκριμένο μονοπάτι αναζήτησης δεν έχει νόημα να συνεχιστεί, καθώς δεν πρόκειται ποτέ να οδηγήσει σε τελική λύση. Η λειτουργία της μεθόδου βασίζεται στον εντοπισμό βασικών τύπων αδιεξόδων που εμφανίζονται στο παιχνίδι. Αρχικά, η μέθοδος σαρώνει ολόκληρο τον πίνακα board και εντοπίζει όλα τα κελιά που περιέχουν κιβώτια ('0'). Για κάθε τέτοιο κελί εφαρμόζονται διαφορετικοί έλεγχοι ώστε να διαπιστωθεί αν υπάρχει κάποιος από τους τύπους αδιεξόδων:

- **Deadlock γωνίας (corner deadlock):** Ελέγχεται αν το κιβώτιο βρίσκεται σε γωνία, δηλαδή σε σημείο όπου δύο τοίχοι συναντώνται κάθετα, χωρίς να υπάρχει στόχος ('*') στη θέση αυτή. Ένα κιβώτιο σε τέτοια θέση δεν μπορεί να μετακινηθεί ποτέ, αφού και οι δύο δυνατές κατευθύνσεις του είναι μπλοκαρισμένες από τοίχους.

- **Deadlock σε τούνελ (corridor deadlock):** Μέσω της βοηθητικής μεθόδου **isCorridorDeadlock(…)**, ελέγχεται αν το κιβώτιο βρίσκεται μέσα σε στενό διάδρομο (το κουτί ακουμπάει έναν τοίχο από δύο παράλληλες πλευρές του) όπου η μόνη διαθέσιμη κίνηση είναι μέσα στο τούνελ. Σε αυτήν την περίπτωση αν δεν υπάρχει διαθέσιμος στόχος στο τούνελ, και το τούνελ δεν εμφανίζει κάποιο σημείο από το οποίο μπορεί να «δραπετεύσει» ο παίχτης τότε υπάρχει deadlock.

- **Deadlock σε ευθεία (1 wall-side deadlock):** Η μέθοδος εξετάζει αν το κιβώτιο βρίσκεται σε σειρά ή στήλη παράλληλη με τοίχο, με τον τοίχο να εφάπτει με μία πλευρά του κουτιού, χωρίς να υπάρχει στόχος στην ευθεία αυτή. Σε τέτοιες περιπτώσεις, το κιβώτιο έχει παγιδευτεί στην

συγκεκριμένη γραμμή χωρίς να μπορεί να φύγει από αυτήν την γραμμή. Αν υπάρχει στόχος τότε ελέγχεται αν υπάρχει κάποιος τοίχος ενδιάμεσα του παίχτη και του στόχου. Αν υπάρχει τότε υπάρχει deadlock, διαφορετικά όχι. Η μέθοδος επιστρέφει true αν υπάρχει deadlock και false αν δεν υπάρχει.

Βοηθητικές Μέθοδοι

- **isCorridorDeadlock(int I, int j, char[][] board)** : Η μέθοδος αυτή εξετάζει αν στον δεδομένο πίνακα υπάρχει deadlock μέσα από "τούνελ" είτε κάθετα είτε οριζόντια, δηλαδή να υπάρχει κατάσταση στην οποία ο παίχτης βρίσκεται μέσα σε έναν διάδρομο από τοίχους και δεν υπάρχει στόχος , ούτε κουτί μέσα σε αυτόν τον διάδρομο. Αν ισχύουν αυτά τότε υπάρχει deadlock και επιστρέφει true. Ο κανόνας για το τούνελ είναι να μην υπάρχουν αδιέξοδοι για τον παίχτη. Αν υπάρχει έστω και μια τρύπα από την οποία μπορεί να φύγει ο παίχτης τότε δεν υπάρχει deadlock γιατί έχει την δυνατότητα να ξεφύγει, άρα επιστρέφει false. Η περίπτωση που υπάρχει τοίχος στον δρόμο του παίχτη προς την τρύπα είναι deadlock και καλύπτυεται από την μέθοδο **isDeadlock(…)** στην περίσταση όπου ο παίχτης ακουμπάει παραπάνω από 2 τοίχους. Επίσης η περίπτωση να υπάρχουν 2 συνεχόμενα κουτιά μέχρι την τρύπα καλύπτεται από την **IsValidMove(…)**

- **isItCorner(char[][] board, int i, int j):** Η συγκεκριμένη μέθοδος ελέγχει αν οι συγκεκριμένες συντεταγμένες βρίσκονται σε γωνία. Δηλαδή ελέγχεται αν υπάρχει τοίχος στις εξής 4 περιπτώσεις: «Πάνω-Δεξιά, Πάνω-Αριστερά, Κάτω-Δεξιά, Κάτω Αριστερά. Αν ο αυτή η θέση είναι όντως γωνία που δεν υπάρχει στόχος για το κουτί, τότε αν το κουτί καταλήξει εκεί θα έχουμε deadlock, επομένως πρέπει να αποφευχθεί. Χρησιμοποιείται στην μέθοδο **IsDeadlock(…)** για να αποφεύγονται τα deadlocks σε περίπτωση που οι συντεταγμένες έχουν σε 2 ακριβώς πλευρές τοίχο.

- **checkFourDirections(int row, int col, char[][] board):** Αυτή η μέθοδος ελέγχει τις τέσσερις κατευθύνσεις(πάνω, κάτω, αριστερά, δεξιά) γύρω από ένα συγκεκριμένο κελί του πίνακα που υπάρχει κουτί, ελέγχει αν υπάρχουν τοίχοι "#" στα πλευρά του. Για κάθε τοίχο που εντοπίζεται, αποθηκεύεται και ένας αναγνωριστικός αριθμός ο οποίος δείχνει αν ο τοίχος βρίσκεται σε στήλη ή σε γραμμή. Αν ο αριθμός είναι το 1 τότε αυτό σημαίνει πως βρίσκεται σε γραμμή, ενώ αν είναι 0, θα βρίσκεται σε στήλη. Δημιουργούμε μία άδεια λίστα με πίνακα από int σε κάθε κελί, wallDirections, και αν στο πάνω ή κάτω κελί υπάρχει τοίχος τότε προσθέτει στην λίστα τον αναγνωριστικό αριθμό = 1 . Το ίδιο ισχύει για όλες τις κατευθύνσεις. Η μέθοδος επιστρέφει την λίστα αυτή με τις κατευθύνσεις στους δύο άξονες(τοίχος πάνω ή κάτω , τοίχος αριστερά ή δεξία). Η μέθοδος καλείται από την **isDeadlock(…)** με σκοπό να την βοηθήσει στην αναγνώριση της κατάστασης του κουτιού αν πρόκειται για deadlock. Με βάση το περιεχόμενο και το μέγεθος αυτής της λίστας, είναι οργανωμένη η μέθοδος **isDeadlock(…)**

- **noMoneyOrBox(char[][] grid):** Εδώ εξετάζουμε αν στον πίνακα της τρέχουσας κατάστασης που δίνεται υπάρχει έστω και ένα κουτί ή στόχος. Χρησιμοποιείται σαν έξτρα έλεγχος για να εξεταστεί αν μια κατάσταση είναι τελική. Πιο συγκεκριμένα στον Αλγόριθμο A* αν μια κατάσταση έχει h == 0, τότε γίνεται αυτός ο έξτρα έλεγχος προληπτικά. Αν όντως δεν υπάρχουν ούτε κουτιά, ούτε στόχοι τότε επιστρέφει True. Σε κάθε άλλη περίπτωση επιστρέφει false.

- **findPlayer(char[][] grid):** Η μέθοδος διατρέχει όλο τον πίνακα (grid) και ψάχνει κελί-κελί για τον παίχτη ,δηλαδή το 1 ή το + (Ο παίχτης βρίσκεται πάνω από ένα goal). Μόλις τον βρει τότε επιστρέφει την θέση του στον πίνακα. Αν δεν τον βρει τότε επιστρέφει τις τιμές {-1,1} όπου καταλαβαίνουμε ότι έχει δοθεί λανθασμένος πίνακας. Η μέθοδος αυτή καλείται και στην **AstarAlgorithm()** και στην **IDSPlayertobox(…)** για να αναζητήσει τον παίχτη και να ξεκινήσει με τη σειρά της η κάθε μέθοδος τις λειτουργίες της.

- **copyGrid(char[][] grid):** Στην συγκεκριμένη μέθοδο αντιγράφεται ο πίνακα σε έναν άλλο καινούργιο με ίδια δεδομένα αλλά ανεξάρτητη μνήμη. Η μέθοδος καλείται μέσα στην AstarAlgorithm() έτσι ώστε οι αναζητήσεις να μην επηρεάσουν καθόλου τον αρχικό πίνακα.

- **printSolutionPath(Node goal):** Η μέθοδος αρχίζει από τον κόμβο της τελικής κατάστασης και μέσω της σχέσης child-parent εντοπίζει όλους τους κόμβους του συγκεκριμένου μονοπατιού μέχρι και τον αρχικό που αντιστοιχεί στην αρχική κατάσταση. Στην συνέχεια εκτυπώνεται ο πίνακας που κατέχει κάθε βήμα-κατάσταση του μονοπατιού. Έτσι ο χρήστης βλέπει step-by-step την λύση του προβλήματος. Χρονική πολυπλοκότητα : O(d), όπου d = βάθος της τελικής κατάστασης

- **makeRectangularWithBorder(char[][] grid):** Η συγκεκριμένη μέθοδος έχει ως σκοπό να μετατρέψει οποιοδήποτε επίπεδο (πίστα) σε ορθογώνιο πίνακα και να το περικλείσει με ένα εξωτερικό περίγραμμα από τοίχους (‘#’), χωρίς να αλλοιώνει τη δομή του εσωτερικού χώρου. Με αυτόν τον τρόπο διασφαλίζεται ότι ο αλγόριθμος δεν θα επιχειρήσει πρόσβαση εκτός ορίων (out of bounds), ακόμα κι αν οι γραμμές του αρχικού πίνακα έχουν διαφορετικό μήκος.
  Η μέθοδος ελέγχει αρχικά το μέγιστο μήκος γραμμής του πίνακα ώστε να δημιουργήσει έναν νέο πίνακα με +2 επιπλέον γραμμές και στήλες (για την εξωτερική περίμετρο). Στη συνέχεια, αντιγράφει τα αρχικά κελιά μέσα στον νέο πίνακα και τοποθετεί τοίχους μόνο στα εξωτερικά όρια (επάνω, κάτω, αριστερά, δεξιά). Σε περίπτωση που κάποια γραμμή του αρχικού πίνακα είναι μικρότερη από τη μεγαλύτερη, τα κενά κελιά συμπληρώνονται επίσης με τοίχους ‘#’, ώστε ο νέος πίνακας να είναι πλήρως ορθογώνιος.
  Με αυτόν τον τρόπο επιτυγχάνεται προληπτικά η ασφαλής ορθογωνιοποίηση της πίστας, χωρίς να αλλοιωθεί η αρχική γεωμετρία της, προστατεύοντας παράλληλα τον παίκτη και τα κουτιά από ενδεχόμενες εσφαλμένες αναφορές εκτός ορίων κατά την εκτέλεση του αλγορίθμου.

- **isGoal(Node current):** Η μέθοδος αυτή επιστρέφει true όταν η ευρετική τιμή του κόμβου-κατάσταση είναι μηδέν. Όταν όλα τα κουτιά έχουν τοποθετηθεί πάνω σε στόχους, τότε ο πρώτος όρος της ευρετικής γίνεται 0 (κάθε κουτί συμπίπτει με έναν στόχο → συνολικό άθροισμα αποστάσεων = 0). Επιπλέον, επειδή δεν υπάρχουν ελεύθερα κουτιά '0' στον πίανακα, η **IDSPlayertobox(…)** δεν βρίσκει κουτί για να υπολογιστεί η απόσταση από τον παίχτη και τελικά επιστρέφει 0, άρα και ο δεύτερος όρος της ευρετικής γίνεται 0. Συνεπώς h = 0 και η μέθοδος σηματοδοτεί ότι βρισκόμαστε σε τελική κατάσταση. Μόνο όταν ισχύει ότι h = 0 και δεν υπάρχει κανένα “0” και “$” στο ταμπλό ο αλγόριθμος τυπώνει τη λύση και τερματίζει. Η **isGoal(…)** έχει σταθερή πολυπλοκότητα O(1).

Υπολογισμός Ευρετικής Συνάρτησης

- **IDSPlayertobox(char[][] level)** : Η μέθοδος IDSPlayertobox(…) υπολογίζει την απόσταση του παίκτη από το πλησιέστερο κουτί, χρησιμοποιώντας τον αλγόριθμο IDS(Iterative Deepening Search). Ο λόγος που χρησιμοποιούμε IDS αντί για BFS είναι ότι το BFS απαιτεί να κρατά στη μνήμη όλα τα επίπεδα του γράφου, άρα η χωρική πολυπλοκότητα είναι O(b^d), όπου b είναι ο branching factor και d το βάθος. Σε αντίθεση ο IDS συνδυάζει την μικρή μνήμη που απαιτεί ο DFS, επειδή δεν χρειάζεται να αποθηκεύει όλους τους κόμβους του επιπέδου, με την ορθότητα του BFS – εκτελεί διαδοχικά περιορισμένες αναζητήσεις με βάθος 0,1,2,… μέχρι να βρεθεί κάποιο κουτί. Έτσι η πολυπλοκότητα χώρου γίνεται O(bd), πολύ μικρότερη και κατάλληλη για Sokoban, που μπορεί να προκύψει τεράστιος χώρος καταστάσεων .Η πολυπλοκότητα χρόνου είναι ίδια με τον bfs, δηλαδή O(b^d) . Παρόλο που επαναλαμβάνει κόμβους, ο συνολικός χρόνος παραμένει πρακτικός, γιατί τα περισσότερα πιθανά μονοπάτια κόβονται γρήγορα επειδή δεν είναι έγκυρα ή προκαλούν deadlock. Η τιμή που επιστρέφει η IDS είναι μια εκτίμηση του κόστους της απόστασης του παίκτη προς το κοντινότερο κουτί σε αυτόν κάθε φορά και χρησιμοποιείται στον A* ως μέρος του συνολικού h. Είναι πλήρες διότι βρίσκει πάντα μια λύση όπως ο bfs και εννοείται θα είναι βέλτιστη γιατί ακολουθεί το στυλ του bfs. Ο IDSPlayertobox(…) αυξάνει το βάθος αναζήτησης μόνο όταν όλες οι προηγούμενες προσπάθειες απέτυχαν. Αυτό εξασφαλίζει ότι βρίσκουμε την μικρότερη πιθανή απόσταση του παίχτη προς το πλησιέστερο κουτί σε αυτόν χωρίς να κρατάμε ολόκληρο το χώρο καταστάσεων στη μνήμη, όπως κάνει το BFS . Γι’ αυτόν ο IDS είναι ιδανικός όταν η διαθέσιμη μνήμη είναι περιορισμένη αλλά η βέλτιστη λύση πρέπει να βρεθεί. Και στην περίπτωση μας βρίσκει όντως την μικρότερη απόσταση παίχτη -> πλησιέστερο κουτί σε αυτόν.

- **DLS(char[][] level, int row, int col, int limit, boolean[][] visited)** : Η μέθοδος DLS(Depth Limited Search) αποτελεί βοηθητική μέθοδο της IDSPlayertobox(…) για να κάνει αναζήτηση βάθους με συγκεκριμένο όριο. Η DLS ξεκινά από τη θέση του παίκτη και εξερευνά τις γειτονικές θέσεις μέχρι να βρει κάποιο κουτί ή να εξαντληθεί τα επιτρεπόμενο βάθος .Εάν το όριο φτάσει στο 0 χωρίς να βρεθεί λύση, σταματά και επιστρέφει αποτυχία ,ώστε η IDS να αυξήσει το βάθος και να ξανατρέξει. Με αυτόν τον τρόπο αποφεύγεται η άσκοπη εξερεύνηση πολύ βαθιών ή άπειρων διαδρομών, ενώ παράλληλα η μνήμη παραμένει γραμμική ως προς το βάθος, δηλαδή O(d). Επιπλέον, η DLS(…) χρησιμοποιεί πίνακα επισκέψεων (visited[row][col])για να αποτρέψει επαναληπτικές διελεύσεις από τα ίδια τετράγωνα, κάτι που μειώνει την πολυπλοκότητα και αποφεύγει ατέρμονους κύκλους. Αν η DLS βρει κουτί μέσα στο επιτρεπτό βάθος, σημαίνει ότι υπάρχει προσβάσιμο κουτί σε αυτή την απόσταση, και έτσι επιστρέφει το συγκεκριμένο κόστος. Η DLS είναι μια αναδρομική αναζήτηση κατά βάθος. Για παράδειγμα έστω limit = όριο βάθους. Εάν το limit<0,αυτό σημαίνει ότι εξαντλήθηκε το όριο κινήσεων πριν φτάσει στον στόχο, οπότε επιστρέφει false. Εάν το τρέχον κελί (level[row][col]) είναι κουτί '0', τότε ο στόχος βρέθηκε, και επιστρέφει true. Η μέθοδος δοκιμάζει αναδρομικά και τις 4 δυνατές κινήσεις(πάνω, κάτω, αριστερά, δεξιά). Για κάθε έγκυρη κίνηση(isValid(…)),καλείται ξανά η DLS με το όριο μειωμένο κατά ένα (limit - 1). Εάν οποιαδήποτε από αυτές τις αναδρομικές κλήσεις επιστρέψει true, τότε το κιβώτιο βρέθηκε, και η μέθοδος επιστρέφει αμέσως true, διακόπτοντας την αναζήτηση σε αυτό.

- **isValid(char[][] level, boolean[][] visited, int row, int col)**: Πρόκειται για μια βοηθητική μέθοδο που επιστρέφει true αν η επόμενη κίνηση είναι εντός ορίων, το κελί δεν πρόκειται για τοίχο (#) και δεν έχει ξανά-επισκεφτεί για την αναζήτηση IDS. Ουσιαστικά παραβλέπει όλα τα υπόλοιπα(0,$,*) για να βρει μια υποεκτίμηση της απόστασης.

- **Heuristic(char[][] grid):** Η ευρετική συνάρτηση υπολογίζει μια εκτίμηση του κόστους που απαιτείται σε κάθε τρέχουσα κατάσταση για να λυθεί το πρόβλημα, δηλαδή την h στον A*. Η h εδώ αποτελείται από δύο μέρη: h1 το οποίο απαρτίζεται από το άθροισμα των ελάχιστων Μανχάταν αποστάσεων κάθε κουτιού από το κοντινότερο στόχο, και h2, δηλαδή την απόσταση του παίκτη από το κοντινότερο κουτί (η τιμή που επιστρέφει η μέθοδος **IDSPlayertobox(…)**). Η Μανχάταν απόσταση ορίζεται ως |x1−x2| + |y1−y2| και είναι συμβατή με την πραγματική κίνηση στο ταμπλό (πάνω–κάτω–δεξιά–αριστερά). Έτσι η ευρετική δεν υπερεκτιμά ποτέ το πραγματικό κόστος, άρα είναι και αποδεκτή. Επιπλέον είναι συνεπής, αφού για κάθε διαδοχική κατάσταση n και n’ με πραγματικό κόστος μετάβασης c(n,n’) = 1 ισχύει ότι h(n) ≤ 1 + h(n’) . Αυτό ισχύει διότι η Μανχάταν απόσταση (h1) αλλάζει το πολύ κατά 1 σε κάθε μετακίνηση, ενώ και η h2 (απόσταση παίκτη–κουτιού) προκύπτει από IDS το οποίο υπολογίζει πάντα την ελάχιστη πραγματική απόσταση του παίκτη, άρα και αυτό μειώνεται ή αυξάνεται το πολύ κατά 1 ανά βήμα. Επομένως τόσο το h1 όσο και το h2 είναι συνεπείς ευρετικές, και το άθροισμά τους παραμένει συνεπές. Πρέπει να σημειωθεί ότι το πραγματικό κόστος κάθε κίνησης παίχτη από ένα κελί στο επόμενο είναι g = 1. Επομένως, όταν ο παίκτης πλησιάζει προς ένα κουτί η τιμή της ευρετικής μειώνεται κατά 1 (h = h − 1), ενώ όταν ένα κουτί μετακινείται προς τον στόχο η Μανχάταν απόσταση για αυτό το κουτί επίσης μειώνεται κατά 1. Η συνολική h μπορεί να μειωθεί κατά 2 μόνο όταν σε μία κίνηση μειώνεται ταυτόχρονα η απόσταση κουτιού–στόχου *και* η απόσταση παίκτη–κουτιού. Επίσης υπάρχουν περιπτώσεις που η τιμή της h θα αυξάνεται το οποίο γίνεται για παράδειγμα όταν ο παίχτης βάλει κουτί σε στόχο και αρχίζει να πηγαίνει προς ένα άλλο κουτί το οποίο θα είναι πιο μακριά προφανώς απο αυτό που έβαλε στον στόχο μόλις. Επομένως ποτέ δεν υπερεκτιμά το πραγματικό κόστος για την μετάβαση ενώ σε πολλές περιπτώσεις την υποεκτιμά, ειδικότερα όταν βάζει ένα κουτί σε στόχο αλλά και λόγω της **isValid(…)** η οποία αγνοεί τα πάντα εκτός από τοίχους στην αναζήτηση απόστασης με σκοπό να προσκομίσουμε μια υποεκτίμηση για την ευρετική μας. Επίσης ένας άλλος βασικός λόγος που δεν γίνεται ποτέ υπερεκτίμηση είναι πως η ευρετική δεν λαμβάνει υπόψιν τις κινήσεις που χρειάζεται ο παίχτης για να ευθυγραμμιστεί με το κουτί και τον στόχο ώστε να αρχίζει να σπρώχνει προς την σωστή κατεύθυνση. Επομένως σχεδόν πάντα θα υποεκτιμάει και σε λίγες περιπτώσεις όπως η πρώτη επιλογή κατάστασης θα την εκτιμάει ακριβώς. Συνολικά, η **Heuristic(…)** υπολογίζει πόσο μακριά βρίσκονται τα κουτιά από τους στόχους + πόσο απέχει ο παίκτης από το κοντινότερο σε αυτόν κουτί την τρέχουσα κατάσταση, δηλαδή είναι γραμμικός συνδιασμός 2 άλλων συναρτήσεων: για κάθε κατάσταση n: h(n)= h1(n)+h2(n). Με αυτόν τον τρόπο ο A* βάζει σε προτεραιότητα καταστάσεις που «φαίνονται» πιο κοντά στη λύση.

Σύγκριση κατανάλωσης μνήμης BFS VS IDS

Παράδειγμα 4ο case στις επιλογές του χρήστη.

Όπως προαναφέρθηκε η προτίμηση του αλγορίθμου IDS υπέρ του BFS για τον υπολογισμό της απόστασης μεταξύ παίχτη και κοντινότερου κουτιού έχει γίνει σκόπιμα για την μείωση της κατανάλωσης μνήμης. Αυτό μας επιτρέπει να εφαρμόζουμε τον αλγόριθμο μας σε μεγαλύτερες και δυσκολότερες πίστες οι οποίες απαιτούν μεγαλύτερο όγκο μνήμης για να φτάσουν σε τελική κατάσταση και να διαπιστωθεί αν υπάρχει λύση ή όχι. Πιο συγκεκριμένα, στην $4^{η}$ πίστα που δίνεται για επιλογή στον παίχτη, η τελική κατάσταση βρίσκεται στο $389^{ο}$ βήμα. Αυτό σημαίνει πως το βάθος d του δέντρου που έχει δημιουργηθεί είναι d = 389. Συνεπώς, καθώς b = 4 ο BFS θα απαιτούσε O(b^d) = O(4^389) μνήμη. Αυτό έχει ως αποτέλεσμα να έχουμε πιθανόν exception error: **OutOfMemory error** αν δεν υπάρχει πάρα πολύ μνήμη. Αντιθέτως με την υλοποίηση του IDS η απαιτούμενη μνήμη είναι O(b*d) = O(4*389). Παρατηρούμε δραματική διαφορά στο μέγεθος της απαιτούμενης μνήμης και συνεπώς είναι εφικτό να βρεθεί λύση σε αυτό και πολλά άλλα προβλήματα που απαιτούνται τόσοι πολλοί υπολογισμοί καταστάσεων.