# Τεχνητή Νοημοσύνη
# Εαρινό Εξάμηνο 2023
# Εργαστηριακή Άσκηση 1 (Τοποθέτηση κύβων σε επιθυμητό σχηματισμό)


## Ανδρεόπουλος Στάθης ΑΜ: 4630
## Κοντάκης Σπύρος ΑΜ: 4702


***<u>Περιγραφή του κώδικα για την άσκηση 1 i).</u>***

Γραμμές 1-5. Συμπεριλαμβανόμενες βιβλιοθήκες: stdio.h: Παρέχει συναρτήσεις εισόδου/εξόδου. stdlib.h: Παρέχει συναρτήσεις γενικής χρήσης, όπως εκχώρηση μνήμης και δημιουργία τυχαίων αριθμών. time.h: Παρέχει λειτουργίες για τον χειρισμό του χρόνου. math.h: Παρέχει μαθηματικές συναρτήσεις. string.h: Παρέχει συναρτήσεις για χειρισμό συμβολοσειράς.

Γραμμή 7. Σταθερές: Max: Αντιπροσωπεύει τη μέγιστη τιμή για μια σταθερά. Ορίζεται ως 1000.

Γραμμές 11-19. Ορισμός δομής: status: Αντιπροσωπεύει μια κατάσταση στο χώρο αναζήτησης. A: Ένας πίνακας 2D για την αποθήκευση των συντεταγμένων (x, y) κάθε μπλοκ (1, ..., N). running_cost: Αντιπροσωπεύει το κόστος επίτευξης αυτής της κατάστασης από την αρχική κατάσταση. extension, childs, path, closed_set: δείκτες σε άλλες δομές κατάστασης.

Γραμμή 21- 26. Καθολικές μεταβλητές: head and tail: Δείκτες προς το πρώτο και το τελευταίο στοιχείο μιας συνδεδεμένης λίστας δομών κατάστασης. total_neighbor_states: Μια μεταβλητή μετρητή για την παρακολούθηση του αριθμού των γειτονικών κρατών που δημιουργούνται. total_extension: Μια μεταβλητή μετρητή για να παρακολουθείτε τον αριθμό των επεκτάσεων που εκτελούνται (καταστάσεις κλαδεύονται από το MA). closed_h και closed_t: Δείχνουν την αρχή και το τέλος του κλειστού συνόλου, το οποίο είναι μια συνδεδεμένη λίστα εξερευνημένων καταστάσεων.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <string.h>

#define Max 1000

int K, N, L ;

struct status
{
    int A[Max][2];          // Array to store the coordinates (x, y) of each block (1, ..., N)
    double running_cost;    // The cost g(n) [theory] = cost of reaching this status from the initial status
    struct status *extension;
    struct status *childs;
    struct status *path;
    struct status *closed_set;
};

struct status *head = NULL;
struct status *tail = NULL;
long int total_neighbor_states = 1;   // Number of generated neighbor states
long int total_extension = 0;         // Number of extensions performed (number of states pruned from the MA)
struct status *closed_h = NULL;
struct status *closed_t = NULL;       // Start and end of the closed set (list of explored states)
```

Γραμμές 29-48. valid_one (const int B[Max][2]): Αυτή η συνάρτηση ελέγχει την εγκυρότητα μιας διαμόρφωσης μπλοκ που αντιπροσωπεύεται από τον 2D πίνακα B. Επαναλαμβάνεται σε κάθε μπλοκ στη διαμόρφωση και ελέγχει εάν υπάρχει ένα έγκυρο μπλοκ κάτω από αυτό. Η συνάρτηση χρησιμοποιεί έναν ένθετο βρόχο για να συγκρίνει τις συντεταγμένες κάθε μπλοκ με τις συντεταγμένες άλλων μπλοκ. Εάν ένα μπλοκ δεν βρίσκεται στην πρώτη σειρά και δεν υπάρχει κανένα μπλοκ κάτω από αυτό στη διαμόρφωση, η συνάρτηση επιστρέφει 0, υποδεικνύοντας ότι η διαμόρφωση δεν είναι έγκυρη. Εάν όλα τα μπλοκ έχουν ένα έγκυρο μπλοκ κάτω από αυτά, η συνάρτηση επιστρέφει 1, υποδεικνύοντας ότι η διαμόρφωση είναι έγκυρη.

Γραμμές 51-57. empty_one(const int A[Max][2], int x, int y): Αυτή η συνάρτηση ελέγχει εάν μια καθορισμένη θέση (x, y) στον πίνακα 2D A είναι κενή (δεν υπάρχει μπλοκ σε αυτήν τη θέση). Επαναλαμβάνει πάνω από κάθε μπλοκ στον πίνακα και συγκρίνει τις συντεταγμένες του με την καθορισμένη θέση. Εάν υπάρχει ένα μπλοκ στην καθορισμένη θέση, η συνάρτηση επιστρέφει 0, υποδεικνύοντας ότι δεν είναι κενό. Εάν δεν υπάρχει μπλοκ στην καθορισμένη θέση, η συνάρτηση επιστρέφει 1, υποδεικνύοντας ότι είναι κενή.

```
29  int valid_one(const int B[Max][2]) {
30      for (int i = 0; i < N; i++) {
31          if (B[i][1] >= 1) { // Check if the block is not in the first row
32              int found = 0; // Flag to track if a valid block below is found
33
34              // Iterate through the blocks to find a valid block below the current block
35              for (int j = 0; j < N; j++) {
36                  if (j != i && B[j][0] == B[i][0] && B[j][1] == B[i][1] - 1) {
37                      found = 1; // A valid block below is found
38                      break;
39                  }
40              }
41
42              if (!found)
43                  return 0; // If no valid block below is found, the configuration is invalid
44          }
45      }
46
47      return 1; // All blocks have a valid block below them, so the configuration is valid
48  }
49
50
51  int empty_one(const int A[Max][2], int x, int y) {
52      for (int i = 0; i < N; i++) {
53          if (A[i][0] == x && A[i][1] == y)
54              return 0; // A block exists at the specified position, so it is not empty
55      }
56      return 1; // No block exists at the specified position, so it is empty
57  }
```

Γραμμές 59-73. free_one(int A[Max][2], int nx, int ny): Αυτή η συνάρτηση ελέγχει εάν μια καθορισμένη θέση (nx, ny) στον πίνακα 2D A είναι ελεύθερη ή μη. Η συνάρτηση ελέγχει πρώτα αν η καθορισμένη θέση βρίσκεται στην επάνω σειρά (ny == 2). Εάν βρίσκεται στην επάνω σειρά, η συνάρτηση επιστρέφει αμέσως 1, υποδεικνύοντας ότι η θέση είναι ελεύθερη. Εάν η καθορισμένη θέση δεν βρίσκεται στην επάνω σειρά, η συνάρτηση στη συνέχεια ελέγχει εάν η παραπάνω θέση (nx, ny + 1) είναι κενή χρησιμοποιώντας τη συνάρτηση empty_one. Η συνάρτηση empty_one καλείται με τις παραμέτρους A, nx και ny + 1 για να ελεγχθεί αν η παραπάνω θέση είναι κενή. Εάν η παραπάνω θέση είναι κενή, η συνάρτηση επιστρέφει 1, υποδεικνύοντας ότι η καθορισμένη θέση είναι ελεύθερη. Εάν καμία από τις παραπάνω συνθήκες δεν πληρούνται, η συνάρτηση επιστρέφει 0, υποδεικνύοντας ότι η καθορισμένη θέση δεν είναι ελεύθερη.

```
59  int free_one(int A[Max][2], int nx, int ny) {
60      int isTopRow = (ny == 2); // Check if it is in the top row
61
62      if (isTopRow) {
63          return 1; // It is free
64      }
65
66      int isAboveEmpty = empty_one(A, nx, ny + 1); // Check if the position above is empty
67
68      if (isAboveEmpty) {
69          return 1; // It is free
70      }
71
72      return 0; // It is not free
73  }
```

Γραμμές 76-109. initialization(void): Αυτή η συνάρτηση είναι υπεύθυνη για την προετοιμασία του αλγόριθμου αναζήτησης λαμβάνοντας μια αρχική διαμόρφωση μπλοκ από τον χρήστη. Χρησιμοποιεί έναν βρόχο με μια συνθήκη που βασίζεται στη σημαία validStatus για να επαναλάβει τη διαδικασία μέχρι να εισαχθεί μια έγκυρη αρχική διαμόρφωση. Εντός του βρόχου, εκτελεί τα ακόλουθα βήματα: Εκχωρεί μνήμη για μια νέα δομή κατάστασης χρησιμοποιώντας malloc. Ελέγχει εάν η κατανομή ήταν επιτυχής. Εάν όχι, εκτυπώνει ένα μήνυμα σφάλματος και βγαίνει από το πρόγραμμα. Ορίζει το τρέχον κόστος της κατάστασης σε 0 και αρχικοποιεί τους δείκτες επέκτασης, παιδιά, closed_set και διαδρομή σε NULL. Προτρέπει τον χρήστη να εισαγάγει τις συντεταγμένες (x, y) για κάθε κύβο/μπλοκ. Αποθηκεύει τις εισαγόμενες συντεταγμένες στον πίνακα A της δομής κατάστασης. Ελέγχει την εγκυρότητα της αρχικής διαμόρφωσης χρησιμοποιώντας τη συνάρτηση valid_one. Εάν η διαμόρφωση είναι έγκυρη (το valid_one επιστρέφει 1), ορίζει τη σημαία validStatus σε 1. Εάν η διαμόρφωση δεν είναι έγκυρη, εκτυπώνει ένα μήνυμα σφάλματος και ελευθερώνει τη μνήμη που έχει εκχωρηθεί για τη μη έγκυρη κατάσταση. Μόλις επιτευχθεί μια έγκυρη αρχική διαμόρφωση, θέτει τους δείκτες κεφαλής και ουράς στη δομή κατάστασης.

```c
void initialization(void) {
    struct status *state = NULL; // Pointer to the newly created status
    int i, x, y;
    int validStatus = 0; // Flag to track the validity of the initial configuration

    do {
        state = (struct status *)malloc(sizeof(struct status)); // Create a new status (struct status)
        if (state == NULL) {
            printf("No Memory.\n");
            exit(-1);
        }
        state->running_cost = 0.0; // Set the running cost of the status to 0
        state->extension = NULL; // Initialize the extension pointer to NULL
        state->childs = NULL; // Initialize the childs pointer to NULL
        state->closed_set = NULL; // Initialize the closed_set pointer to NULL
        state->path = NULL; // Initialize the path pointer to NULL

        for (i = 0; i < N; i++) {
            printf("Give me the coordinates (x,y) of %d cube: \n", i + 1);
            scanf("%d %d", &x, &y);
            state->A[i][0] = x; // Store the x-coordinate of the cube in the A array
            state->A[i][1] = y; // Store the y-coordinate of the cube in the A array
        }

        if (valid_one(state->A) == 1)
            validStatus = 1; // If the initial configuration is valid, set the validStatus flag to 1
        else {
            printf("Something is wrong!!!! \n");
            free(state); // Free the memory allocated for the invalid status
        }
    } while (!validStatus); // Repeat the loop until a valid initial configuration is entered

    head = tail = state; // Set the head and tail pointers to the newly created status
}
```

Γραμμές 113-120. is_solution(void): Αυτή η συνάρτηση ελέγχει εάν έχει βρεθεί μια καλή λύση εξετάζοντας την τιμή του δείκτη κεφαλής. Εκτελεί τα ακόλουθα βήματα: Ελέγχει εάν ο δείκτης κεφαλής είναι NULL, κάτι που υποδεικνύει ότι δεν υπάρχει καλή λύση. Εάν ο δείκτης κεφαλής είναι NULL, εκτυπώνει ένα μήνυμα που υποδεικνύει την απουσία καλής λύσης και επιστρέφει 0 για να υποδείξει την αποτυχία. Εάν ο δείκτης κεφαλής δεν είναι NULL, σημαίνει ότι έχει βρεθεί μια καλή λύση και η συνάρτηση επιστρέφει 1 για να υποδείξει την επιτυχία.

```c
int is_solution(void) {
    if (head == NULL) {
        printf("No good solution.\n");
        return 0; // No good solution found, return 0 to indicate failure
    }

    return 1; // Good solution found, return 1 to indicate success
}
```

Γραμμές 123-158 find_the_lower(void): Αυτή η λειτουργία χρησιμοποιείται για την εύρεση και κατάργηση της κατάστασης με το χαμηλότερο κόστος από μια συνδεδεμένη λίστα καταστάσεων. Εκτελεί τα ακόλουθα βήματα: Ελέγχει εάν ο δείκτης κεφαλής είναι NULL, κάτι που υποδεικνύει ότι η λίστα είναι κενή. Εάν ο δείκτης κεφαλής είναι NULL, εκτυπώνει ένα μήνυμα που υποδεικνύει μια κενή λίστα και επιστρέφει NULL. Αρχικοποιεί τους απαραίτητους δείκτες και μεταβλητές: previous_one: Δείκτης στην προηγούμενη κατάσταση στη λίστα. current_one: Δείκτης στην τρέχουσα κατάσταση στη λίστα. best_one: Δείκτης στην κατάσταση με το χαμηλότερο κόστος (αρχικοποιήθηκε ως η πρώτη κατάσταση στη λίστα). low_cost: Αρχικοποιεί το low_cost με το κόστος της πρώτης κατάστασης. Διασχίζει τη συνδεδεμένη λίστα έως ότου επιτευχθεί η τελευταία κατάσταση: Ελέγχει εάν το κόστος της επόμενης κατάστασης (current_one->childs) είναι χαμηλότερο από το τρέχον χαμηλότερο κόστος. Εάν το κόστος της επόμενης κατάστασης είναι χαμηλότερο, ενημερώνει το low_cost με το κόστος της επόμενης κατάστασης και ενημερώνει τον δείκτη best_one με την επόμενη κατάσταση. Ενημερώνει επίσης τον δείκτη previous_one με την τρέχουσα κατάσταση. Μετακινείται στην επόμενη κατάσταση στη λίστα ενημερώνοντας τον δείκτη current_one. Καταργεί την κατάσταση best_one από τη λίστα: Εάν η κατάσταση best_one δεν είναι η κορυφή της λίστας, ενημερώνει τον επόμενο δείκτη της προηγούμενης κατάστασης (previous_one->childs) και ενημερώνει τον ουραίο δείκτη εάν η κατάσταση best_one είναι η τελευταία κατάσταση. Εάν η κατάσταση best_one είναι η κεφαλή της λίστας, ενημερώνει τον δείκτη κεφαλής στην επόμενη κατάσταση και ενημερώνει τον ουραίο δείκτη εάν η κατάσταση best_one είναι η μόνη κατάσταση. Τέλος, ορίζει τον επόμενο δείκτη της κατάστασης best_one σε NULL. Επιστρέφει την κατάσταση best_one, η οποία είναι η κατάσταση με το χαμηλότερο κόστος.

```
123 struct status *find_the_lower(void) {
124     if (head == NULL) {
125         printf("You give an empty list.\n");
126         return NULL; // Return NULL when the list is empty
127     }
128
129     struct status *previous_one = NULL; // Pointer to the previous status
130     struct status *current_one; // Pointer to the current status
131     struct status *best_one = head; // Pointer to the status with the lowest cost
132     double lower_cost = best_one->running_cost; // Initialize the lower_cost with the cost of the first status
133
134     current_one = head; // Set the current_one pointer to the head of the list
135     while (current_one->childs != NULL) { // Traverse the list until the last status is reached
136         if (current_one->childs->running_cost < lower_cost) { // If the cost of the next status is lower than the current lowest cost
137             lower_cost = current_one->childs->running_cost; // Update the lower_cost with the cost of the next status
138             best_one = current_one->childs; // Update the best_one pointer with the next status
139             previous_one = current_one; // Update the previous_one pointer with the current status
140         }
141         current_one = current_one->childs; // Move to the next status in the list
142     }
143
144     // Remove the best_one status from the list
145     if (previous_one != NULL) { // If the best_one status is not the head of the list
146         previous_one->childs = best_one->childs; // Update the next pointer of the previous status
147         if (best_one->childs == NULL)
148             tail = previous_one; // Update the tail pointer if the best_one status is the last status
149         best_one->childs = NULL; // Set the next pointer of the best_one status to NULL
150     } else { // If the best_one status is the head of the list
151         head = best_one->childs; // Update the head pointer to the next status
152         if (best_one->childs == NULL)
153             tail = NULL; // Update the tail pointer if the best_one status is the only status
154         best_one->childs = NULL; // Set the next pointer of the best_one status to NULL
155     }
156
157     return best_one; // Return the status with the lowest cost
158 }
```

Γραμμές 161-184. is_final_state(struct status *state): Αυτή η συνάρτηση ελέγχει εάν η δεδομένη κατάσταση αντιπροσωπεύει την τελική κατάσταση με βάση τις συντεταγμένες των κύβων. Εκτελεί τα ακόλουθα βήματα: Ελέγχει τις συντεταγμένες του πρώτου συνόλου κύβων (state->A[0] to state->A[K-1]): Επαναλαμβάνει στην περιοχή από 0 έως K-1 και ελέγχει εάν η συντεταγμένη x (κατάσταση->A[i][0]) είναι ίση με i και η συντεταγμένη y (κατάσταση->A[i][1]) είναι ίσο με 0. Εάν οι συντεταγμένες οποιουδήποτε κύβου δεν ταιριάζουν με τις αναμενόμενες τιμές, επιστρέφει 0 για να δείξει ότι η δεδομένη κατάσταση δεν είναι η τελική κατάσταση. Ελέγχει τις συντεταγμένες του δεύτερου συνόλου κύβων (state->A[K] to state->A[2*K-1]): Επαναλαμβάνει στην περιοχή από K έως 2*K-1 και ελέγχει εάν η συντεταγμένη x (κατάσταση->A[i][0]) είναι ίση με i - K και η συντεταγμένη y (κατάσταση->A[i] [1]) ισούται με 1. Εάν οι συντεταγμένες οποιουδήποτε κύβου δεν ταιριάζουν με τις αναμενόμενες τιμές, επιστρέφει 0 για να δείξει ότι η δεδομένη κατάσταση δεν είναι η τελική κατάσταση. Ελέγχει τις συντεταγμένες του τρίτου συνόλου κύβων (state->A[2*K] to state->A[3*K-1]): Επαναλαμβάνει στην περιοχή από 2*K έως 3*K-1 και ελέγχει εάν η συντεταγμένη x (κατάσταση->A[i][0]) είναι ίση με i - 2*K και η συντεταγμένη y (κατάσταση-> Το A[i][1]) ισούται με 2. Εάν οι συντεταγμένες οποιουδήποτε κύβου δεν ταιριάζουν με τις αναμενόμενες τιμές, επιστρέφει 0 για να δείξει ότι η δεδομένη κατάσταση δεν είναι η τελική κατάσταση. Εάν όλες οι συντεταγμένες του κύβου ταιριάζουν με τις αναμενόμενες τιμές για τα αντίστοιχα σύνολά τους, υποδηλώνει ότι η δεδομένη κατάσταση είναι η τελική κατάσταση και επιστρέφει 1.

```
161 int is_final_state(struct status *state) {
162     // Check the coordinates of the first set of cubes
163     for (int i = 0; i < K; ++i) {
164         if (state->A[i][0] != i || state->A[i][1] != 0) {
165             return 0; // Not the final state, return 0
166         }
167     }
168
169     // Check the coordinates of the second set of cubes
170     for (int i = K; i < 2 * K; ++i) {
171         if (state->A[i][0] != (i - K) || state->A[i][1] != 1) {
172             return 0; // Not the final state, return 0
173         }
174     }
175
176     // Check the coordinates of the third set of cubes
177     for (int i = 2 * K; i < 3 * K; ++i) {
178         if (state->A[i][0] != (i - 2 * K) || state->A[i][1] != 2) {
179             return 0; // Not the final state, return 0
180         }
181     }
182
183     return 1; // It is the final state, return 1
184 }
```

Γραμμές 186-220. best_sequence(struct status *st): Αυτή η συνάρτηση εκτυπώνει την ακολουθία των καταστάσεων από την αρχική έως την τελική κατάσταση, μαζί με το αντίστοιχο κόστος και τις συντεταγμένες κύβου. Εκτελεί τα ακόλουθα βήματα: Αρχικοποιεί τις μεταβλητές state, st1, i και j. κατάσταση: Δείκτης για την τρέχουσα κατάσταση υπό επεξεργασία. st1: Δείκτης στην αρχική κατάσταση. i: Μεταβλητή για την παρακολούθηση του αριθμού κατάστασης. j: Μεταβλητή που χρησιμοποιείται για επανάληψη. Διασχίζει τη διαδρομή από την τελευταία κατάσταση στην αρχική κατάσταση: Ξεκινά από το δεδομένο st (τελική κατάσταση) και μετακινείται μέσω των δεικτών επέκτασης για να φτάσει στην αρχική κατάσταση. Ορίζει τον δείκτη διαδρομής κάθε κατάστασης στην προηγούμενη κατάσταση. Εκτυπώνει την ακολουθία των καταστάσεων από την αρχική έως την τελική κατάσταση: Χρησιμοποιεί έναν βρόχο που συνεχίζεται έως ότου η κατάσταση φτάσει στο st1 (αρχική κατάσταση). Εκτυπώνει τον αριθμό κατάστασης (i), το κόστος της κατάστασης (state->running_cost) και τις συντεταγμένες κάθε κύβου στην κατάσταση. Αυξάνει τον αριθμό κατάστασης (i) και μετακινεί την κατάσταση στην προηγούμενη κατάστασή της (state-> path). Εκτυπώνει την τελική κατάσταση: Εκτυπώνει τον αριθμό κατάστασης (i), το κόστος της κατάστασης (state->running_cost) και τις συντεταγμένες κάθε κύβου στην τελική κατάσταση. Σημείωση: Σε αυτό το σημείο, η κατάσταση είναι η αρχική κατάσταση (st1).

```
186 void best_sequence(struct status *st) {
187     struct status *state = st;
188     struct status *st1 = st;
189     int i = 1;
190     int j;
191
192     // Traverse the path from the last state to the initial state
193     while (state->extension != NULL) {
194         state->extension->path = state;
195         state = state->extension;
196     }
197
198     // Print the sequence of states from the initial state to the final state
199     while (state != st1) {
200         printf("The %d state with cost: %f is: ", i, state->running_cost);
201
202         // Print the coordinates of each cube in the state
203         for (j = 0; j < N; ++j) {
204             printf("%d:(%d,%d) ", j, state->A[j][0], state->A[j][1]);
205         }
206         printf("\n");
207
208         state = state->path;
209         ++i;
210     }
211
212     // Print the final state
213     printf("The %d final state with cost: %f is: ", i, state->running_cost);
214
215     // Print the coordinates of each cube in the final state
216     for (j = 0; j < N; ++j) {
217         printf("%d:(%d,%d) ", j, state->A[j][0], state->A[j][1]);
218     }
219     printf("\n");
220 }
```

Γραμμές 223-247. searching_algorithm(int B[Max][2]): Αυτή η συνάρτηση εκτελεί μια αναζήτηση στο κλειστό σύνολο καταστάσεων για να ελέγξει εάν υπάρχει κατάσταση με συντεταγμένες κύβου που ταιριάζουν με τον δεδομένο πίνακα B. Επιστρέφει μια ακέραια τιμή my_count που υποδεικνύει εάν βρέθηκε ή όχι μια αντιστοίχιση. Εκτελεί τα ακόλουθα βήματα: Αρχικοποιεί τις μεταβλητές my_count στο 0 και st για να δείχνει την κεφαλή του κλειστού συνόλου. Διασχίζει το κλειστό σύνολο καταστάσεων: Εισάγει έναν βρόχο που συνεχίζεται έως ότου το st φτάσει στο τέλος του κλειστού συνόλου (NULL). Αρχικοποιεί τη βοήθεια στο 1 και το i στο 0 για κάθε κατάσταση στο κλειστό σύνολο. Συγκρίνει τις συντεταγμένες του κύβου της κατάστασης B με κάθε κατάσταση st στο κλειστό σύνολο: Εισάγει έναν άλλο βρόχο που επαναλαμβάνεται πάνω από τις συντεταγμένες του κύβου από 0 έως N-1. Συγκρίνει B[i][0] (συντεταγμένη x) με st->A[i][0] (συντεταγμένη x) και B[i][1] (συντεταγμένη y) με st->A[i] [1] (υ-συντεταγμένη). Εάν και οι δύο συντεταγμένες κύβου ταιριάζουν, αυξάνει το i και συνεχίζει στην επόμενη επανάληψη του βρόχου. Εάν κάποιες συντεταγμένες κύβου δεν ταιριάζουν, θέτει τη βοήθεια στο 0 και σπάει τον βρόχο. Εάν όλες οι συντεταγμένες του κύβου ταιριάζουν (δηλαδή, το i ισούται με N), ορίζει το my_count σε 1 (που υποδεικνύει μια αντιστοίχιση) και σπάει τον εξωτερικό βρόχο. Μετακινεί το st στην επόμενη κατάσταση στο κλειστό σύνολο (st->closed_set). Επιστρέφει το my_count (1 αν βρεθεί αντιστοιχία, 0 διαφορετικά). Ο σκοπός της συνάρτησης searching_algorithm είναι να αναζητήσει το κλειστό σύνολο καταστάσεων και να καθορίσει εάν υπάρχει κατάσταση με συντεταγμένες κύβου που ταιριάζουν με τον δεδομένο πίνακα B. Συγκρίνει τις συντεταγμένες κύβου κάθε κατάστασης στο κλειστό σύνολο με τον δεδομένο πίνακα και επιστρέφει 1 εάν βρέθηκε ένα ταίριασμα ή 0 εάν δεν βρεθεί αντιστοιχία.

```
223 int searching_algorithm(int B[Max][2]) {
224     int my_count = 0;
225     struct status *st = closed_h;
226
227     // Traverse the closed set of states
228     while (st != NULL) {
229         int help = 1;
230         int i = 0;
231
232         // Compare the cube coordinates of state B with each state in the closed set
233         while (i < N && (B[i][0] == st->A[i][0] && B[i][1] == st->A[i][1])) {
234             i++;
235         }
236
237         // If all cube coordinates match, set my_count to 1 (indicating a match) and break the loop
238         if (i == N) {
239             my_count = 1;
240             break;
241         }
242
243         st = st->closed_set; // Move to the next state in the closed set
244     }
245
246     return my_count; // Return my_count (1 if a match is found, 0 otherwise)
247 }
```

Γραμμές 250-417. Αρχικοποίηση μεταβλητών: Δηλώστε μια τοπική μεταβλητή κατάσταση τύπου struct για να αναπαραστήσετε τις desiredcoord καταστάσεις. Δηλώστε τις ακέραιες μεταβλητές i, a, επιθυμητός κώδικας και y για επανάληψη και υπολογισμούς. Δηλώστε έναν 2D ακέραιο πίνακα B για να αποθηκεύσετε ένα αντίγραφο των συντεταγμένων του κύβου της τρέχουσας κατάστασης. Επαναλάβετε μέσα από κάθε κύβο στην τρέχουσα κατάσταση st: Ελέγξτε εάν ο τρέχων κύβος μπορεί να μετακινηθεί σε νέα θέση (free_one(st->A, st->A[i][0], st->A[i][1]) == 1). Ελέγξτε τις διαθέσιμες κενές θέσεις στη συντεταγμένη L: Επαναλάβετε τις θέσεις a από 0 έως L-1. Ελέγξτε αν η θέση (a, 0) είναι κενή (empty_one(st->A, a, 0) == 1). Δημιουργήστε ένα αντίγραφο της τρέχουσας κατάστασης st στον πίνακα B. Ενημερώστε τις συντεταγμένες του κύβου του τρέχοντος κύβου (B[i][0] = a; B[i][1] = 0). Ελέγξτε εάν η ενημερωμένη κατάσταση B είναι έγκυρη (valid_one(B) == 1). Δημιουργήστε μια νέα κατάσταση κατάστασης και εκχωρήστε μνήμη για αυτήν. Ενημερώστε τις συντεταγμένες του κύβου της κατάστασης και υπολογίστε το τρέχον κόστος του. Ορίστε τους δείκτες της κατάστασης της επέκτασης, των παιδιών και του κλειστού συνόλου. Προσθέστε τη νέα κατάσταση κατάστασης στη λίστα των καταστάσεων (κεφαλή και ουρά). Σπάστε τον βρόχο εάν η επιθυμητή συντεταγμένη (α) υπερβαίνει το K. Ελέγξτε τις διαθέσιμες κενές θέσεις στη συντεταγμένη K: Επαναλάβετε τα ίδια βήματα όπως στο βήμα 3 αλλά για τη συντεταγμένη K. Ελέγξτε τις διαθέσιμες κενές θέσεις στις συντεταγμένες 2K: Επαναλάβετε τα ίδια βήματα όπως στο βήμα 3 αλλά για τη συντεταγμένη 2K. Προσθέστε την τρέχουσα κατάσταση st στο κλειστό σύνολο καταστάσεων: Εάν το κλειστό σετ είναι κενό (closed_h == NULL), ρυθμίστε την κεφαλή και την ουρά στο st. Διαφορετικά, ενημερώστε τον δείκτη κλειστού συνόλου της τελευταίας κατάστασης στο κλειστό σύνολο σε st και ορίστε την ουρά σε st.

```
250 void extend(struct status *st) {
251     struct status *state;
252     int i, a, desiredcoord, y;
253     int B[Max][2];
254 
255     // Iterate through each cube in state st
256     for (i = 0; i < N; i++) {
257         if (free_one(st->A, st->A[i][0], st->A[i][1]) == 1) {
258 
259             // Check available empty positions in the L coordinate
260             for (a = 0; a < L; a++) {
261                 if (empty_one(st->A, a, 0) == 1) {
262 
263                     // Create a copy of the current state st into B
264                     memcpy(B, st->A, sizeof(int) * N * 2);
265                     B[i][0] = a;
266                     B[i][1] = 0;
267 
268                     // Check if the updated state B is valid
269                     if (valid_one(B) == 1) {
270                         state = (struct status *)malloc(sizeof(struct status));
271                         if (state == NULL) {
272                             printf("There's no memory.\n");
273                             exit(-1);
274                         }
275                         total_neighbor_states++;
276                         memcpy(state->A, st->A, sizeof(int) * N * 2);
277                         state->A[i][0] = a;
278                         state->A[i][1] = 0;
279 
280                         desiredcoord = 0;
281                         y = st->A[i][1];
282 
283                         // Calculate the running cost for the new state
284                         if (desiredcoord > y)
285                             state->running_cost = st->running_cost + desiredcoord - y;
286                         else if (desiredcoord < y)
287                             state->running_cost = st->running_cost + 0.5 * (y - desiredcoord);
288                         else
289                             state->running_cost = st->running_cost + 0.75;
```

```
                        state->extension = st;
                        state->childs = NULL;
                        state->closed_set = NULL;

                        // Add the new state to the list of states
                        if (head == NULL) {
                            head = state;
                            tail = state;
                        } else {
                            tail->childs = state;
                            tail = state;
                        }

                        // Break the loop if the desired coordinate exceeds K
                        if (a >= K)
                            break;
                    }
                }
            }

            // Check available empty positions in the K coordinate
            for (a = 0; a < K; a++) {
                if (empty_one(st->A, a, 1) == 1) {

                    // Create a copy of the current state st into B
                    memcpy(B, st->A, sizeof(int) * N * 2);
                    B[i][0] = a;
                    B[i][1] = 1;

                    // Check if the updated state B is valid
                    if (valid_one(B) == 1) {
                        state = (struct status *)malloc(sizeof(struct status));
                        if (state == NULL) {
                            printf("There's no memory.\n");
                            exit(-1);
                        }
                        total_neighbor_states++;
                        memcpy(state->A, st->A, sizeof(int) * N * 2);
                        state->A[i][0] = a;
                        state->A[i][1] = 1;

                        desiredcoord = 1;
                        y = st->A[i][1];

                        // Calculate the running cost for the new state
                        if (desiredcoord > y)
                            state->running_cost = st->running_cost + desiredcoord - y;
                        else if (desiredcoord < y)
                            state->running_cost = st->running_cost + 0.5 * (y - desiredcoord);
                        else
                            state->running_cost = st->running_cost + 0.75;

                        state->extension = st;
                        state->childs = NULL;
                        state->closed_set = NULL;

                        // Add the new state to the list of states
                        if (head == NULL) {
                            head = state;
                            tail = state;
                        } else {
                            tail->childs = state;
                            tail = state;
                        }
                    }
                }
            }
        }
```

```
            // Check available empty positions in the 2K coordinate
            for (a = 0; a < K; a++) {
                if (empty_one(st->A, a, 2) == 1) {

                    // Create a copy of the current state st into B
                    memcpy(B, st->A, sizeof(int) * N * 2);
                    B[i][0] = a;
                    B[i][1] = 2;

                    // Check if the updated state B is valid
                    if (valid_one(B) == 1) {
                        state = (struct status *)malloc(sizeof(struct status));
                        if (state == NULL) {
                            printf("There's no memory.\n");
                            exit(-1);
                        }
                        total_neighbor_states++;
                        memcpy(state->A, st->A, sizeof(int) * N * 2);
                        state->A[i][0] = a;
                        state->A[i][1] = 2;

                        desiredcoord = 2;
                        y = st->A[i][1];

                        // Calculate the running cost for the new state
                        if (desiredcoord > y)
                            state->running_cost = st->running_cost + desiredcoord - y;
                        else if (desiredcoord < y)
                            state->running_cost = st->running_cost + 0.5 * (y - desiredcoord);
                        else
                            state->running_cost = st->running_cost + 0.75;

                        state->extension = st;
                        state->childs = NULL;
                        state->closed_set = NULL;

                        // Add the new state to the list of states
                        if (head == NULL) {
                            head = state;
                            tail = state;
                        } else {
                            tail->childs = state;
                            tail = state;
                        }
                    }
                }
            }
        }
    }

    // Add the current state st to the closed set of states
    if (closed_h == NULL) {
        closed_h = st;
        closed_t = st;
    } else {
        closed_t->closed_set = st;
        closed_t = st;
    }
}
```

Γραμμές 420-455. Δήλωση μεταβλητών: Δηλώστε έναν δείκτη κατάστασης δομής currentState για να αναπαραστήσει την τρέχουσα κατάσταση που εξερευνάται. Δηλώστε μια ακέραια μεταβλητή isFound για να παρακολουθήσετε εάν βρέθηκε η κατάσταση στόχου. Δηλώστε μια ακέραια μεταβλητή userInput για να αποθηκεύσετε την είσοδο χρήστη. Δηλώστε τις ακέραιες μεταβλητές i, isSafe και isFinalState για επανάληψη και έλεγχο των συνθηκών κατάστασης. Καλέστε τη συνάρτηση αρχικοποίησης για να αρχικοποιήσετε την αρχική κατάσταση (Αλγόριθμος Βήμα 1). Εμφανίστε τις συντεταγμένες AK της αρχικής κατάστασης. Ζητήστε από το χρήστη να εισαγάγει μια τιμή. Εισαγάγετε τον κύριο βρόχο αναζήτησης: Καλέστε τη συνάρτηση is_solution για να ελέγξετε εάν η τρέχουσα κατάσταση είναι λύση. Καλέστε τη συνάρτηση find_the_lower για να βρείτε την κατάσταση με το χαμηλότερο κόστος. Καλέστε τη συνάρτηση searching_algorithm για να ελέγξετε εάν η τρέχουσα κατάσταση είναι ασφαλής. Εάν η τρέχουσα κατάσταση δεν είναι ασφαλής (!isSafe): Ελέγξτε εάν η τρέχουσα κατάσταση είναι η κατάσταση τελικού στόχου καλώντας τη συνάρτηση is_final_state. Εάν είναι η τελική κατάσταση στόχου (isFinalState), ορίστε το isFound σε 1 για να υποδείξετε ότι βρέθηκε η κατάσταση στόχου. Καλέστε τη συνάρτηση best_sequence για να δημιουργήσετε και να εμφανίσετε την καλύτερη ακολουθία καταστάσεων. Εάν δεν είναι η κατάσταση του τελικού στόχου, συνεχίστε με την αναζήτηση: Αυξήστε τον μετρητή total_extension. Εάν ο αριθμός των επεκτάσεων είναι πολλαπλάσιο του 3000, εκτυπώστε τον αριθμό των γονέων και το τρέχον κόστος. Καλέστε τη συνάρτηση επέκτασης για να δημιουργήσετε και να προσθέσετε νέες καταστάσεις στον χώρο αναζήτησης. Ο βρόχος αναζήτησης συνεχίζεται μέχρι να βρεθεί η κατάσταση στόχου (το isFound είναι 1).

```
420  void ucs_algorithm(void)
421  {
422      struct status *currentState;
423      int isFound = 0, userInput, i, isSafe, isFinalState;
424
425      initialization(); // Algorithm Step 1: Initialize the initial state
426
427      // Display the AK
428      for (i = 0; i < N; i++) {
429          printf("%d: (%d, %d) ", i, head->A[i][0], head->A[i][1]);
430      }
431      printf("\n");
432
433      printf("Enter a value: ");
434      scanf("%d", &userInput);
435
436      while (!isFound) {
437          is_solution(); // Check if the current state is a solution
438          currentState = find_the_lower(); // Find the state with the lowest cost
439
440          isSafe = searching_algorithm(currentState->A); // Check if the current state is safe
441          if (!isSafe) {
442              isFinalState = is_final_state(currentState); // Check if the current state is the final goal state
443              if (isFinalState) {
444                  isFound = 1; // Goal state is found
445                  best_sequence(currentState); // Generate and display the best sequence of states
446              } else {
447                  total_extension++;
448                  if (total_extension % 3000 == 0) {
449                      printf("Number of parents: %ld, Current cost: %f\n", total_extension, currentState->running_cost);
450                  }
451                  extend(currentState); // Generate and add new states to the search space
452              }
453          }
454      }
455  }
```

Γραμμές 458-471. Ζητήστε από το χρήστη να εισαγάγει την τιμή του K (τον αριθμό των κύβων σε κάθε σετ). Διαβάστε την τιμή του K από τον χρήστη χρησιμοποιώντας scanf. Ορίστε τις τιμές N (συνολικός αριθμός κύβων) και L (αριθμός διαθέσιμων θέσεων) με βάση το K. Καλέστε τη συνάρτηση ucs_algorithm για να εκτελέσετε τον αλγόριθμο Uniform Cost Search (UCS). Αφού ολοκληρωθεί ο αλγόριθμος, εκτυπώστε την τιμή total_neighbor_states, η οποία αντιπροσωπεύει τον συνολικό αριθμό των γειτονικών καταστάσεων που δημιουργήθηκαν κατά την αναζήτηση. Εκτυπώστε την τιμή total_extension, η οποία αντιπροσωπεύει τον συνολικό αριθμό των επεκτάσεων κατάστασης που εκτελέστηκαν κατά την αναζήτηση. Επιστρέψτε το 0 για να υποδείξετε την επιτυχή εκτέλεση του προγράμματος.

```
458  int main()
459  {
460      printf("Enter K: ");
461      scanf("%d", &K);
462      N = 3 * K;
463      L = 4 * K;
464
465      ucs_algorithm(); // Run the UCS algorithm
466
467      printf("The value of total_neighbor_states is %ld.\n", total_neighbor_states);
468      printf("The value of total_extension is %ld.\n", total_extension);
469
470      return 0;
471  }
```

***<u>Παράδειγμα για το πως τρέχει το πρόγραμμα μας.</u>***

Με bold φαίνεται τι πρέπει να γραφτεί από το πληκτρολόγιο

Enter K: **2**

Give me the coordinates (x,y) of 1 cube:

**5 0**

Give me the coordinates (x,y) of 2 cube:

**1 0**

Give me the coordinates (x,y) of 3 cube:

**0 0**

Give me the coordinates (x,y) of 4 cube:

**1 1**

Give me the coordinates (x,y) of 5 cube:

**0 1**

Give me the coordinates (x,y) of 6 cube:

**1 2**

0: (5, 0) 1: (1, 0) 2: (0, 0) 3: (1, 1) 4: (0, 1) 5: (1, 2)

Enter a value: **1**

Number of parents: 3000, Current cost: 4.250000

Number of parents: 6000, Current cost: 4.500000

The 1 state with cost: 0.000000 is: 0:(5,0) 1:(1,0) 2:(0,0) 3:(1,1) 4:(0,1) 5:(1,2)

The 2 state with cost: 0.500000 is: 0:(5,0) 1:(1,0) 2:(0,0) 3:(1,1) 4:(2,0) 5:(1,2)

The 3 state with cost: 1.250000 is: 0:(5,0) 1:(1,0) 2:(3,0) 3:(1,1) 4:(2,0) 5:(1,2)

The 4 state with cost: 2.000000 is: 0:(0,0) 1:(1,0) 2:(3,0) 3:(1,1) 4:(2,0) 5:(1,2)

The 5 state with cost: 3.000000 is: 0:(0,0) 1:(1,0) 2:(0,1) 3:(1,1) 4:(2,0) 5:(1,2)

The 6 final state with cost: 5.000000 is: 0:(0,0) 1:(1,0) 2:(0,1) 3:(1,1) 4:(0,2) 5:(1,2)

The value of total_neighbor_states is 94420.

The value of total_extension is 7042.