ΚΟΤΟΦΩΛΗ ΧΡΙΣΤΙΝΑ
ΤΣΑΠΙΚΟΥΝΗ ΓΕΩΡΓΙΑ

ΤΕΧΝΗΤΗ ΝΟΗΜΟΣΥΝΗ
1Η ΕΡΓΑΣΤΗΡΙΑΚΗ ΑΣΚΗΣΗ

Στην δοθείσα άσκηση καλούμαστε να τοποθετήσουμε Ν κύβους σε έναν επιθυμητό σχηματισμό. Ουσιαστικά τοποθετούμε κάθε κύβο πάνω στους άξονες x,y, όπου πρώτη θέση θεωρούμε από τα δεδομένα της άσκησης την (x,y)=(1,1). Η άσκηση έχει θεωρήσει πως οι κύβοι βρίσκονται πάνω σε ένα τραπέζι, επομένως για y=1 θεωρούμε την επιφάνεια του τραπεζιού, για y=2 την μεσαία σειρά που στηρίζεται στην πρώτη, και τέλος για y=3 την πάνω σειρά που στηρίζεται στην μεσαία. Το πρόβλημα περιμένει από τον χρήστη να του εισάγει την αρχική κατάσταση των κύβων όπου εάν είναι έγκυρη συνεχίζει για να φτάσει στην τελική κατάσταση υπολογίζοντας και το κόστος. Τελική κατάσταση θεωρείται όταν για y=1 οι κύβοι είναι τοποθετημένοι σε σειρά από τον 1 έως τον Κ, για y=2 οι κύβοι είναι τοποθετημένοι σε σειρά από τον Κ+1 έως τον 2Κ, και για y=3 οι κύβοι είναι τοποθετημένοι σε σειρά από τον 2Κ+1 έως τον 3Κ. Η άσκηση έχει δύο ζητούμενα, το πρώτο είναι να κάνω αναζήτηση ομοιόμορφου κόστους δηλαδή μεσώ του αλγορίθμου UCS, και το δεύτερο να κάνω αναζήτησης Α* χρησιμοποιώντας μια αποδεκτή ευρετική συνάρτηση μέσω του αλγόριθμου Α*.

Αυτό που γνωρίζουμε και από την θεωρεία είναι πως το ελάχιστο κόστος στον αλγόριθμό UCS ορίζεται ως **min[g(n)]** , ενώ στον αλγόριθμο Α* ορίζεται ως **min[g(n)+h(n)]**. Η διαφορά ανάμεσα στους δύο αλγορίθμους σημειώνεται στο τι εξετάζει ο καθένας. Πιο συγκεκριμένα ο UCS εξερευνά όλες τις διαδρομές και επιλέγει αυτή με το ελάχιστο κόστος, επεκτείνει τους κόμβους με βάση αποκλειστικά το αθροιστικό κόστος διαδρομής του. Ενώ ο Α* δεν εξερευνά όλες τις διαδρομές αλλά μόνο τις «απαραίτητες», χρησιμοποιεί τόσο το **πραγματικό κόστος g(n)** όσο και το **εκτιμώμενο κόστος (ευρετικό) h(n)** για να δώσει προτεραιότητα στην επέκταση του κόμβου.

Παρακάτω θα παραθέσω δύο εικόνες από την εκτέλεση των δυο αλγορίθμων για Κ=2 δηλαδή 6 αριθμημένους κύβους.

```
Give the coordinates (x,y) of cube number 1:
5 1
Give the coordinates (x,y) of cube number 2:
2 1
Give the coordinates (x,y) of cube number 3:
1 1
Give the coordinates (x,y) of cube number 4:
2 2
Give the coordinates (x,y) of cube number 5:
1 2
Give the coordinates (x,y) of cube number 6:
2 3
AK is:1:(5,1) 2:(2,1) 3:(1,1) 4:(2,2) 5:(1,2) 6:(2,3)
Enter:
1
P 3000 C 4.500000
--------------------------------------------------------------------------------
The 1 state with cost 0.000000 is:1:(5,1) 2:(2,1) 3:(1,1) 4:(2,2) 5:(1,2) 6:(2,3)
The 2 state with cost 0.500000 is:1:(5,1) 2:(2,1) 3:(1,1) 4:(2,2) 5:(3,1) 6:(2,3)
The 3 state with cost 1.250000 is:1:(5,1) 2:(2,1) 3:(4,1) 4:(2,2) 5:(3,1) 6:(2,3)
The 4 state with cost 2.000000 is:1:(1,1) 2:(2,1) 3:(4,1) 4:(2,2) 5:(3,1) 6:(2,3)
The 5 state with cost 3.000000 is:1:(1,1) 2:(2,1) 3:(1,2) 4:(2,2) 5:(3,1) 6:(2,3)
The 6 final state with cost 5.000000 is:1:(1,1) 2:(2,1) 3:(1,2) 4:(2,2) 5:(1,3) 6:(2,3)
--------------------------------------------------------------------------------
Were 4158 expanded!

--------------------------------
Process exited after 60.04 seconds with return value 20
Press any key to continue . . .
```

Εικόνα 1:UCS

```
Give the coordinates (x,y) of cube number 1:
5 1
Give the coordinates (x,y) of cube number 2:
2 1
Give the coordinates (x,y) of cube number 3:
1 1
Give the coordinates (x,y) of cube number 4:
2 2
Give the coordinates (x,y) of cube number 5:
1 2
Give the coordinates (x,y) of cube number 6:
2 3
AK is:1:(5,1) 2:(2,1) 3:(1,1) 4:(2,2) 5:(1,2) 6:(2,3)
Enter:
1
--------------------------------------------------------------------------------
The 1 state with cost 0.000000 is:1:(5,1) 2:(2,1) 3:(1,1) 4:(2,2) 5:(1,2) 6:(2,3)
The 2 state with cost 0.500000 is:1:(5,1) 2:(2,1) 3:(1,1) 4:(2,2) 5:(3,1) 6:(2,3)
The 3 state with cost 1.250000 is:1:(5,1) 2:(2,1) 3:(4,1) 4:(2,2) 5:(3,1) 6:(2,3)
The 4 state with cost 2.000000 is:1:(1,1) 2:(2,1) 3:(4,1) 4:(2,2) 5:(3,1) 6:(2,3)
The 5 state with cost 3.000000 is:1:(1,1) 2:(2,1) 3:(1,2) 4:(2,2) 5:(3,1) 6:(2,3)
The 6 final state with cost 5.000000 is:1:(1,1) 2:(2,1) 3:(1,2) 4:(2,2) 5:(1,3) 6:(2,3)
--------------------------------------------------------------------------------
Were 528 expanded!

--------------------------------
Process exited after 27.08 seconds with return value 19
Press any key to continue . . .
```

*Εικόνα 2:A**

Έτσι λοιπόν εκτελώντας και τους δύο αλγορίθμους παρατηρούμε πως ο A* είναι πιο αποτελεσματικός όσον αφορά τον χρόνο καθώς χρειάστηκε τον μισό σχεδόν συγκριτικά με τον UCS , αλλά και χώρο καθώς βλέπουμε πως ο αριθμός των επεκτάσεων είναι πολύ πολύ μικρότερος από αυτόν του UCS. Βλέπουμε πως η διαδρομή που ακολουθεί από την αρχική κατάσταση στη τελική είναι ιδιά, με την διαφορά πως ο Α* δεν εξερευνά με λεπτομέρεια κάθε μονοπάτι.

Πάμε να δούμε πως εκτελέσαμε αυτούς τους δύο βασικούς αλγορίθμους

UCS:

```
void UCS(void)
{
    struct status *st1;
    int flag=0;
    int w,i;
    int secur,check;


    initial_state_on_search();

    printf("AK is:");
    for(i=0;i<N;i++){
        printf("%d:(%d,%d) ",i+1,initial->pos[i][0],initial->pos[i][1]);
    }
    printf("\n");
    printf("Enter:\n");
    scanf("%d",&w);

    while(flag==0){

        st1=smaller_state();
        secur=searching(st1->pos);

        if (secur==0){
            check=final_state(st1);
            if (check==1) {
                flag=1;
                print_top_route(st1);
            }
            else{
                number_of_extensions++;
                if (number_of_extensions % 3000 == 0){
                    printf("P %d C %f\n",number_of_extensions,st1->cost);
                }
                new_situations(st1);

            }
        }

    }
}
```

Αρχικά καλούμε την συνάρτηση **initial_state_on_search()** η οποία δημιούργει την αρχική κατάσταση και στην συνέχεια ζητάω να τυπωθεί στην οθόνη μου. Έπειτα μέσω της εντολής **while(flag==0)** εξετάζω επαναληπτικά τις καταστάσεις μέχρι να βρω την τελική. Καλώντας την **smaller_state()** τσεκάρω ποια κατάσταση έχει το μικρότερο κόστος, έπειτα καλώ την **searching()** με την οποία τσεκάρω ποια κατάσταση είναι μέρος του κλειστού συνόλου. Αν είναι καλώ την συνάρτηση **final_state()** με την οποία ελέγχω αν κάποια κατάσταση είναι τελική, αν είναι απλά τυπώνω στην οθόνη την βέλτιστη διαδρομή, αν δεν έχω φτάσει ακόμα στην τελική συνεχίζω και καλώ την συνάρτηση **new_situations()** η οποία ελέγχει τις επεκτάσεις που γίνονται.

A*:

```c
void ASTAR(void)
{
    struct status *st1;
    int flag=0;
    int w,i;
    int secur,check;


    initial_state_on_search();

    printf("AK is:");
    for(i=0;i<N;i++){
        printf("%d:(%d,%d) ",i+1,initial->pos[i][0],initial->pos[i][1]);
    }
    printf("\n");
    printf("Enter:\n");
    scanf("%d",&w);

    while(flag==0){

        st1=smaller_state();
        secur=searching(st1->pos);

        if (secur==0){
            check=final_state(st1);
            if (check==1) {
                flag=1;
                print_top_route(st1);
            }
            else{
                number_of_extensions++;
                if (number_of_extensions % 3000 == 0){
                    printf("P %d C %f\n",number_of_extensions,st1->cost);
                }
                new_situations(st1);

            }
        }

    }
}
```

Βλέπουμε πως τα βήματα και οι συναρτήσεις που καλούνται είναι ακριβώς ίδια. Η διαφορά βρίσκεται στην συνάρτηση smaller_state() μέσα στην οποία έχουμε κάνει ενημέρωση το κόστος από g(n) σε g(n)+h(n)

```c
struct status *smaller_state(void)
{
    struct status *st,*before,*cur,*best_comb;
    double smaller_cost;

    before=NULL;

    st=initial;
    smaller_cost=st->cost + accepted_heuristic_function(st->pos);
    best_comb=st;
    while(st->next_parent!=NULL){
        cur=st;
        st=st->next_parent;
        if (st->cost + accepted_heuristic_function(st->pos) < smaller_cost){
            smaller_cost=st->cost + accepted_heuristic_function(st->pos);
            best_comb=st;
            before=cur;
        }
    }

    if (before!=NULL){
        before->next_parent=best_comb->next_parent;
        if (best_comb->next_parent==NULL){
            final=before;
        }
        best_comb->next_parent=NULL;
    }
    else{
        initial=best_comb->next_parent;
        if (best_comb->next_parent==NULL){
            final=NULL;
        }
        best_comb->next_parent=NULL;
    }
    return(best_comb);
}
```

Η ενημέρωση του ελάχιστου κόστους γίνεται στην γραμμή 172 με την εντολή **smaller_cost=st->cost + accepted_heuristic_function(st->pos);.** Η **accepted_heurist_function** είναι η ευρετική συνάρτηση που χρησιμοποίησα

```c
int accepted_heuristic_function(int pos2[N][2])
{
    double cost_hn = 0.0;
    int i;

    for(i=0;i<K;i++) {
        if (pos2[i][0] != (i+1) || pos2[i][1] != 1){
            cost_hn = cost_hn + 0.4;
        }
    }

    for(i=K;i<2*K;i++){
        if (pos2[i][0] != (i-K+1) || pos2[i][1] != 2){
            cost_hn = cost_hn + 0.4;
        }
    }

    for(i=2*K;i<3*K;i++){
        if (pos2[i][0] != (i-2*K+1) || pos2[i][1] != 3){
            cost_hn = cost_hn + 0.4;
         }
    }

    return cost_hn;
}
```

Έχω βάλει ως **h(n)=0.4** και η συνάρτηση αυτή είναι αποδεκτή καθώς δεν υπερεκτιμά το πραγματικό κόστος g(n) για την επίτευξη του στόχου δηλαδή την βέλτιστη διαδρομή. Με βάση τα δεδομένα που μου δίνει η άσκηση το πραγματικό κόστος μπορεί να πάρει ως μικρότερη τιμή την 0.5 **(g(n)>=0.5),** με το δεδομένο αυτό έβαλα ως εκτιμώμενο κόστος το ο.4 **(h(n)=0.4).**

Οι δύο αλγόριθμοί κάνουν compile με τον ίδιο τρόπο, αρχικά ο χρήστης δίνει την αρχική κατάσταση, μετέπειτα εμφανίζεται στην οθόνη ολοκληρωμένη αυτή η κατάσταση, στην συνέχεια δίνει ο χρήστης έναν τυχαίο ακέραιο αριθμό για να συνεχίσει την διαδρομή του, τυπώνει στην οθόνη την διαδρομή που έχει κάνει καθώς και το νούμερο των επεκτάσεων που έγιναν.