Yapay Zekaya Giriş Ödevi Grup 2 - Hanoi Kuleleri Problemi ve Arama Algoritmaları Raporu

Eyüp Can Erol 19052083
Merve Deniz ÖZ 19052012
Enes Bilik 19052087
Bora Erçelik 20052020
İbrahim Bayburtlu 19052017

Mayıs 2024

1. Problemin Tanımı

Hanoi Kuleleri, n diskten oluşan üç çubuk ve bu diskleri sıralama kriterleriyle yerleştirmeyi hedefleyen bir bulmacadır. Diskler çapraz olarak birbirlerinin üzerine yerleştirilemez ve herhangi bir çubuğun üstündeki disklerin çaprazların- dakilerden daha küçük olmalıdır. Problemin amacı, başlangıç çubuğunda verilen n diskli kuleyi bitiş çubuğuna taşımaktır. Her adımda sadece bir disk taşınabilir ve herhangi bir çubukta küçük bir disk, büyük bir diskin üzerine yerleştirilemez.

2. Sezgisel Fonksiyonlar

Hanoi Kuleleri problemi için kullanılacak sezgisel fonksiyonlar, her adımda en uygun disk taşıma seçeneğini belirlemek için kullanılır. Bu fonksiyonlar, her bir durum için en az hamle ile hedefe ulaşmayı sağlamak amacıyla tasarlanır. Örneğin, bir disk taşıma durumu değerlendirilirken, hedef çubuğa en yakın olan disklerin tercih edilmesini sağlayacak sezgisel bilgi kullanılabilir.

2.1. Genişlik Öncelikli Arama (Breast First Search)

2.1.1 Temel Adımlar:

BFS algoritması, genellikle graf tabanlı problemleri çözmek için kullanılan bir arama algoritmasıdır. Bu algoritma, bir başlangıç düğümünden başlayarak tüm komşu düğümleri arar ve ardından bu komşu düğümler üzerinde aynı işlemi tekrar eder. Yani, öncelikle başlangıç düğümünün tüm komşularını keşfeder, sonra bu komşuların komşularını keşfeder ve bu işlemi hedef düğüme ulaşılana kadar tekrar eder. BFS, genellikle en kısa yol problemlerini çözmek için kullanılır çünkü en kısa yolu garanti eder.

2.1.2 Hanoi Kuleleri Problemine Uyarlanması:

Başlangıç durumu: Tüm disklerin ilk çubukta, sıralı olarak büyükten küçüğe doğru yerleştirildiği durum.

Hedef durum: Tüm disklerin son çubukta, sıralı olarak büyükten küçüğe doğru yerleştirildiği durum.

Adımlar:

Başlangıç durumunu bir düğüm olarak tanımlayın ve kuyruğa ekleyin. Kuyruk boşalana kadar:

a. Kuyruktan bir durum alın.

b. Bu durumdan erişilebilecek tüm olası durumları hesaplayın (yani, mevcut durumda hangi diskleri nereye taşıyabileceğinizi hesaplayın).

c. Her bir olası durumu kuyruğa ekleyin (daha önce ziyaret edilmediyse).

d. Eğer hedef duruma ulaşıldıysa, işlemi sonlandırın ve çözümü gösterin. Hedef duruma ulaşılamadıysa, çözüm bulunamamıştır.

2.2. Düzgün Maliyetli Arama (Uniform Cost Search)

2.2.1 Temel Adımlar:

Düzgün Maliyetli Arama, her adımda minimum maliyetli eylemi seçerek ilerler. Hanoi kuleleri gibi bir problemde, bu algoritma genellikle optimum çözümü bulmak için gereksiz işlem yapar çünkü minimum maliyetli yol genellikle en kısa yol değildir. Bu nedenle, düzgün maliyetli arama Hanoi kuleleri problemi için tercih edilen bir seçenek değildir.

Dijkstra tarafından 1959 yılında yayınlanan "A note on two problems in connexion with graphs" adlı makalede tanıtılan Dijkstra Algoritması, aslında Düzgün Maliyetli Arama'nın temelini oluşturur. Dijkstra Algoritması, graf tabanlı bir veri yapısı üzerinde belirli bir köşeden diğer tüm köşelere olan en kısa yol uzunluklarını hesaplar.

Düzgün Maliyet Arama'sı, Dijkstra Algoritmasından farklı olarak hem graf tabanlı hem de tree tabanlı bir veri yapısı üzerinde çalışabilir. Başlangıç düğümünden hedef düğüme olan minimum maliyetli yolu bulmayı çalışır. Bunu yaparken hedefe ulaşana kadarki bütün minimum yolları dener. Bu maliyeti hesaplarken gerçek maliyeti (g(n)) kullanır.

Maliyet şu denklemle bulunur:

f(n) = g(n)

- $f(n)$: Hesaplanan toplam yol fonksiyonu
- $g(n)$: İlk düğüm noktasıyla mevcut düğüm noktası arasındaki maliyet

Örneğin, aşağıdaki şekildeki bir örneği ele alalım: Örnek güzergah : Başlangıç noktası S'den G noktasına gitmek istiyoruz. S noktasından A noktasına gidilmesi durumunda yeni bir düğüm noktasına geçildiği için yol maliyetini ekleyerek yeni değerini alır.

A noktasının $(f(n))$ değeri ise g(n)'e eşit olarak bulunur. Bu yöntemi kullanarak bütün noktaların $(f(n))$ değerlerini bulmak istersek:

- S — A: $(g(n) = 5 = 5)$
- A — G: $(g(n) = 5 + 1 = 6)$
- S — B: $(g(n) = 1 = 1)$
- B — C: $(g(n) = 3 + 1 = 4)$
- C — G: $(g(n) = 5 + 4 = 9)$

Yukarıdaki basit örnekte görüldüğü üzere en kısa yol S-A-G düğümleri ile gidilen yoldur. Bu yolun maliyeti 6 birim olurken alternatif S-B-C-G yolunun maliyeti 9 birimdir

2.2.2 Hanoi Kuleleri Problemine Uyarlanması:

Hanoi Kuleleri problemi Uniform Cost algoritması terminolojisiyle şu şekilde tanımlanabilir:

- **Durum:** Her bir durum, üç çubuğun (A, B, C) her birinde bulunan disklerin düzenlenmiş bir konfigürasyonunu temsil eder. Her disk, boyutuna göre farklı bir değere sahiptir.
- **Başlangıç Durumu:** Başlangıç durumu, tüm disklerin sadece bir çubukta, genellikle soldaki çubukta (A), küçükten büyüğe doğru sıralandığı durumu ifade eder.
- **Hedef Durum:** Hedef durumu, tüm disklerin sadece bir çubukta, genellikle sağdaki çubukta (C), küçükten büyüğe doğru sıralandığı durumu ifade eder.
- **Operatörler:** Her adımda, bir disk yukarıdan alınıp, boş bir çubuğa veya üzerinde daha büyük bir disk bulunmayan bir çubuğa yerleştirilebilir.
- **Maliyet:** Her bir adımın maliyeti, hareket ettirilen disklerin boyutuna bağlı olarak belirlenir. Örneğin, daha büyük bir diskin üzerine daha küçük bir disk yerleştirmek, daha küçük bir maliyete sahiptir.
- **H**edef durumu elde etmek, tüm disklerin sadece bir çubukta, sağdaki çubukta (C), küçükten büyüğe doğru sıralandığı durumu kontrol etmek anlamına gelir.

2.3. Derinlik Öncelikli Arama (Depth First Search)

DFS, bir graf veya ağaç yapısında dolaşmak için kullanılan bir algoritmadır. Algoritma, bir başlangıç noktasından başlayarak en derine iner, ardından geri dönüp bir alt düğüme geçer. Bu adım, hiç keşfedilmemiş yollar kalmayana kadar devam eder.

2.3.1 Temel Adımlar:

1. **Başlangıç Noktası Belirleme:** Başlangıç düğümünü belirleyerek işe başlanır.
2. **Yığın (Stack) Oluşturma:** DFS algoritması, ilerledikçe keşfedilen düğümleri saklamak için bir yığın veri yapısı kullanır.
3. **Düğümleri İşaretleme ve Keşfetme:** Her düğümü işaretleyerek, henüz keşfedilmemiş komşu düğümleri arar ve yığına ekler.
4. **Yığından Düğüm Çıkarma ve İlerleme:** Yığından düğümler çıkarılarak keşif devam ettirilir. Bu işlem, ilgili düğüme ulaşılana kadar devam eder.

Bu adımları takip ederek, DFS algoritması bir grafı veya ağacı dolaşır ve istenen sonuca ulaşır.

2.3.2 Hanoi Kuleleri Problemine Uyarlanması:

Hanoi Kuleleri problemi, üç çubuktan oluşan ve farklı boyutlarda disklerin yer aldığı bir bulmacadır. Amaç, başlangıç çubuğundaki tüm diskleri bitiş çubuğuna taşımaktır. Ancak, her adımda büyük bir diski daha küçük bir diskin üzerine koyamazsınız.

- **Durumları Düğümlere Dönüştürme:** Her durum, disklerin çubuklara yerleştirilme şekli olarak düşünülür ve bir düğüm olarak temsil edilir.
- **Başlangıç Durumu Belirleme:** İlk durum, başlangıç düğümü olarak belirlenir.
- **Mümkün Hareketleri Bulma:** Her adımda mevcut durumda mümkün olan tüm hareketler bulunur ve yeni durumlar oluşturulur.
- **Yığında Yeni Durumları Saklama:** Yeni durumlar bir yığında saklanır ve DFS algoritmasıyla işlenir.
- **Bitiş Durumunu Kontrol Etme:** Bitiş durumu elde edildiğinde, çözüm bulunmuş olur.

2.4. İteratif Derinlik Artırma Araması (Iterative Deepening Search)

İteratif Derinlik Artırma Araması (IDA*), derinlik sınırlı arama ile benzerdir ancak her seferinde derinlik sınırını artırarak çalışır. Bu algoritma, herhangi bir durum için en iyi maliyetli yolun derinliğini bulmayı amaçlar. Aşağıda IDA*'nın temel adımları ve Hanoi Kuleleri problemine nasıl uyarlanabileceği açıklanmaktadır.

2.4.1 Temel Adımlar:

1. **Başlangıç Noktası Belirleme:** Başlangıç düğümü belirlenir.
2. **Derinlik Sınırını Belirleme:** Başlangıçta derinlik sınırı 0 olarak belirlenir.
3. **Arama ve Genişletme:** Her iterasyonda, derinlik sınırı bir artırılarak derinlik sınırlı arama yapılır.
4. **Optimum Çözümü Bulma:** Her iterasyonda, hedef duruma ulaşılmışsa optimum çözüm bulunmuş olur.

2.4.2 Hanoi Kuleleri Problemine Uyarlanması:

Hanoi Kuleleri problemi için IDA* uygulanırken, her adımda mevcut durumdaki tüm olası hamleler incelenir ve her bir hamle için derinlik sınırı artırılarak arama yapılır. Bu sayede, her seferinde daha derin dallara inilir ve böylece optimum çözüme daha yakın olunur.

- **Başlangıç Durumu Belirleme:** Başlangıç durumu olarak, başlangıç çubuğundaki tüm disklerin sıralı olduğu bir durum seçilir.
- **Derinlik Sınırını Belirleme:** Başlangıçta derinlik sınırı 0 olarak belirlenir.
- **Arama ve Genişletme:** Her iterasyonda, derinlik sınırı bir artırılarak derinlik sınırlı arama yapılır. Her adımda mevcut durumdaki tüm olası hamleler incelenir ve derinlik sınırı artırılarak arama devam ettirilir.
- **Optimum Çözümü Bulma:** Hedef duruma ulaşıldığında, optimum çözüm bulunmuş olur.

2.5. Açgözlü Arama Algoritması (Greedy Search Algorithm)

Açgözlü arama algoritması, her adımda en iyi yerel seçimi yaparak çözüme ulaşmayı hedefler. Bu algoritma, her adımda en az maliyetli veya en çok kazanç sağlayacak hamleyi seçer. Aşağıda, Greedy algoritmasının temel adımları ve Hanoi Kuleleri problemine nasıl uygulanabileceği açıklanmaktadır.

2.5.1 Temel Adımlar:

1. **Başlangıç Noktası Belirleme:** Başlangıç durumu olarak, tüm disklerin büyükten küçüğe doğru sıralandığı bir durum seçilir.
2. **En İyi Yerel Seçimi Yapma:** Her adımda, diskleri hedef çubuğa taşıyacak en iyi hamle seçilir.
3. **Arama ve Genişletme:** Seçilen hamle uygulanır ve yeni durum değerlendirilir.
4. **Çözüme Ulaşma:** Tüm diskler hedef çubuğa taşındığında çözüme ulaşılmış olur.

2.5.2 Hanoi Kuleleri Problemine Uyarlanması:

Hanoi Kuleleri problemi için Greedy algoritması uygulanırken, her adımda en az hamle ile en üstteki diski hedef çubuğa taşıyacak hamle tercih edilir. Bu, genellikle en küçük diski taşımak anlamına gelir.

- **Başlangıç Durumu Belirleme:** Başlangıç durumu olarak, tüm disklerin büyükten küçüğe doğru sıralandığı bir durum seçilir.
- **En İyi Yerel Seçimi Yapma:** Her adımda, en üstteki diski hedef çubuğa taşıyacak en az hamle sayısını gerektiren hamle seçilir.
- **Arama ve Genişletme:** Seçilen hamle uygulanır ve yeni durum değerlendirilir. Eğer hedef duruma ulaşılmazsa, yeni durumdan itibaren en iyi yerel seçim tekrar yapılır.
- **Çözüme Ulaşma:** Tüm diskler hedef çubuğa taşındığında, en az hamle sayısı ile çözüme ulaşılmış olur.

2.6. A* Araması (A* Search)

2.6.1 Temel Adımlar:

A* algoritması, ağırlıklı graf terimleriyle formüle edilmiş bilgilendirilmiş bir arama algoritmasıdır. Belirli bir başlangıç düğümünden başlayarak, verilen hedef düğüme en küçük maliyetli yolunu bulmayı amaçlar. Bu algoritma, iki nokta arasındaki en kısa yolu bulmak için kullanılan etkili bir yol bulma algoritmasıdır. İlk olarak 1968 yılında Peter Hart, Nils Nilsson ve Bertram Raphael tarafından yayınlanmıştır.

A* algoritması, temelde başlangıç düğümüyle bitiş düğümü arasındaki bütün diğer düğümlerin konumlarına göre hesaplama yaparak optimum sonuca ulaşır. Bu algoritma, iki önemli bileşeni birleştirir:

- Gerçek maliyet ($g(n)$): Başlangıç düğümü ile mevcut düğüm arasındaki gerçek maliyeti ifade eder. Yani başlangıç düğümünden mevcut düğüme ulaşmak için harcanan maliyet.
- Sezgisel tahmin ($h(n)$): Mevcut düğüm ile hedef düğüm arasındaki sezgisel tahmini maliyeti ifade eder. Bu, kuş uçuşu mesafesi gibi bir sezgisel değer olabilir.

Bu iki bileşen kullanılarak düğümlerin toplam maliyeti hesaplanır:

$$f(n) = g(n) + h(n)$$

Yukarıdaki denklemde:

- $f(n)$: Hesaplanan toplam yol fonksiyonu
- $g(n)$: İlk düğüm noktasıyla mevcut düğüm noktası arasındaki maliyet
- $h(n)$: Sezgisel fonksiyon

Örneğin, aşağıdaki şekildeki bir örneği ele alalım: Örnek güzergah : Başlangıç noktası S'den G noktasına gitmek istiyoruz. S noktasından A noktasına gidilmesi durumunda yeni bir düğüm noktasına geçildiği için $(g(n) = 5)$ (yol maliyeti) değerini alır. Sezgisel fonksiyon $(h(n) = 1)$ olarak belirlenir.

A noktasının $(f(n))$ değeri ise $(5 + 1 = 6)$ olarak bulunur. Bu yöntemi kullanarak bütün noktaların $(f(n))$ değerlerini bulmak istersek:

- S — A: $(g(n) + f(n) = 5 + 1 = 6)$
- A — G: $(g(n) + f(n) = 6 + 0 = 6)$
- S — B: $(g(n) + f(n) = 1 + 4 = 5)$
- B — C: $(g(n) + f(n) = 3 + 2 = 5)$
- C — G: $(g(n) + f(n) = 5 + 0 = 5)$

Yukarıdaki basit örnekte görüldüğü üzere en kısa yol S-B-C-G düğümleri ile gidilen yoldur. Bu yolun maliyeti 5 birim olurken alternatif S-A-G yolunun maliyeti 6 birimdir.

2.6.2 Hanoi Kuleleri Problemine Uyarlanması:

Hanoi Kuleleri problemi, birbirine geçirilmiş üç kuleden oluşur ve bu kulelerde farklı boyutlarda diskler bulunur. A* algoritması, bu problemi çözmek için kullanılabilir.

Öncelikle, Hanoi Kuleleri problemini A* algoritmasının terminolojisiyle tanımlayalım:

- **Başlangıç durumu:** Disklerin başlangıç konumunu ifade eder.
- **Bitiş durumu:** Disklerin hedef konumunu ifade eder (genellikle tüm disklerin farklı bir kuleye taşınmış olduğu durum).
- **Operatörler:** Diskleri bir kuleden diğerine taşımak için yapılan hamlelerdir. Her hamle bir geçişin maliyetini temsil eder.
- **Gerçek maliyet ($g(n)$):** Başlangıç durumundan mevcut duruma ulaşmak için yapılan hamlelerin toplam maliyetini ifade eder.
- **Sezgisel tahmin ($h(n)$):** Mevcut durumdan bitiş durumuna ulaşmanın en az maliyetli tahmini maliyeti ifade eder.

A* algoritması, başlangıç durumundan bitiş durumuna kadar olan toplam maliyeti minimize ederek en kısa yol bulmayı amaçlar.

Örneğin, 3 diskli bir Hanoi Kuleleri problemi için şöyle bir durum düşünelim:

- **Başlangıç durumu:** Tüm diskler A kulesinde, diğer iki kule boş.
- **Bitiş durumu:** Tüm diskler C kulesinde, diğer iki kule boş.

Bu durumu A* algoritmasına uyarlamak için her adımın maliyetini ve sezgisel tahminini belirlememiz gerekecek. Örneğin, her disk hareketi bir birim maliyetle gerçekleşebilir, ve sezgisel tahmin olarak da her disk için bitiş durumuna olan mesafe kullanılabilir (örneğin, hedef kuleden başlangıç kulesine taşınması gereken disk sayısı).

Sonuç olarak, A* algoritması bu sezgisel tahmin ve maliyet hesaplamalarını kullanarak her adımda en uygun hamleyi seçerek Hanoi Kuleleri problemini çözebilir.

2.7. Genelleştirilmiş A* Araması (Generalized A* Search))

2.7.1 Temel Adımlar:

Genelleştirilmiş A Araması (Generalized A Search)*, A algoritmasının genelleştirilmiş bir versiyonudur. A* algoritması, başlangıç düğümünden hedef düğüme en kısa yolu bulmak için kullanılan etkili bir arama algoritmasıdır. Genelleştirilmiş A* ise bu temel algoritmayı daha geniş bir bağlamda uygulamak için tasarlanmıştır.

İşte Genelleştirilmiş A* Aramasının temel özellikleri:

- Ağırlıklı Graf Terimleriyle Formülasyon: Genelleştirilmiş A*, ağırlıklı graf terimleriyle formüle edilmiş bilgilendirilmiş bir arama algoritmasıdır. Bu, düğümler arasındaki maliyetleri ve sezgisel tahminleri içerir.
- Sezgisel Fonksiyonlar: Genelleştirilmiş A*, sezgisel fonksiyonları kullanarak düğümler arasındaki tahmini maliyetleri hesaplar. Bu, yolun daha etkili bir şekilde aranmasına yardımcı olur.
- Optimalite ve Bellek Kullanımı: Genelleştirilmiş A*, tamlığı, optimalliği ve optimal bellek kullanımını hedefler. Ancak, tüm düğümleri bellekte tuttuğundan alan karmaşıklığı dezavantajı vardır.

2.7.2 Hanoi Kuleleri Problemine Uyarlanması:

1. **Başlangıç Durumu:** Tüm diskler A kulesinde, diğer iki kule boş.
2. **Bitiş Durumu:** Tüm diskler C kulesinde, diğer iki kule boş.
3. **Durumların Temsili:** Her durum, disklerin her birinin hangi kulede olduğunu belirten bir durum vektörü olarak temsil edilir. Örneğin, $[3, 2, 1]$ vektörü, 3 diskli bir Hanoi Kulesi problemi için tüm disklerin A kulesinde olduğunu belirtir.
4. **Operatörlerin Belirlenmesi:** Bir durumdan diğerine geçmek için yapılacak işlemler belirlenir. Hanoi Kuleleri problemi için operatörler, bir diskin bir kuleden diğerine taşınmasıdır. Ancak, her adımda yalnızca en üstteki diski taşıyabiliriz.
5. **Hareket Maliyeti ve Sezgisel Tahminler:** Her operasyonun maliyeti belirlenir. Her disk taşıma işlemi için 1 birimlik bir maliyetle gerçekleşebilir. Sezgisel tahmin olarak, her disk için hedef kuleden başlangıç kulesine taşınması gereken disk sayısı kullanılabilir.
6. **Genelleştirilmiş A* Araması Uygulanması:** Her adımda, mevcut durumun maliyeti ve sezgisel tahmini hesaplanır ve en uygun durum seçilerek ilerleme sağlanır. Bu, minimum maliyetli yolun bulunmasını sağlar.
7. **Hedef Duruma Ulaşılıncaya Kadar Adımların Tekrarlanması:** En uygun durum seçilerek adımların tekrarlanmasıyla hedef duruma ulaşılır.

Bu şekilde, Genelleştirilmiş A* Araması Hanoi Kuleleri problemi için uygulanabilir. Bu yöntem, karmaşık problem alanlarında da kullanılabilir ve optimize edilmiş bir çözüm sunabilir.

3. Simülasyon Sonuçları

Hanoi kuleleri problemi için çeşitli arama algoritmaları kullanılabilir. Ancak, her bir algoritmanın avantajları ve dezavantajları vardır ve problemle başa çıkmak için en uygun olanını seçmek önemlidir. Genellikle, A* araması gibi bilgi bazlı algoritmalar, Hanoi kuleleri gibi karmaşık problemlerde daha iyi performans gösterir. Ancak, problem ve gereksinimlere bağlı olarak diğer algoritmalar da tercih edilebilir.

4. HashTable.c

```
#include "data_types.h"
#include "hash_table.h"
#include "graph_search.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

// =========  YOUR COMPULSORY (BUT SPECIFIC TO THE PROBLEM) FUNCTIONS =======

% Alttaki kod bloğu Hanoi Kulesinin o anki durumuna göre benzersiz
bir anahtar üretir.
Anahtar, çubuklardaki disklerin konumunu kodlayarak oluşturulur.

% Kod, çubukları tarar ve her disk için bir karakter çifti oluşturur:
çubuk adı (A, B veya C) ve disk numarası.

% Anahtarın boyutu, tanımlı maksimum boyutu aşarsa bir hata mesajı
verilir ve program sonlandırılır. Son olarak, anahtarın sonuna
NULL karakteri eklenir,
böylece anahtar bir C-string olarak kullanılabilir.

% Bu kod, oyun durumunu bir hash tablosunda depolamak veya benzeri
işlemler için anahtar üretmek üzere tasarlanmıştır.

//____________________ Create unique char key for each state_________________________
void Generate_HashTable_Key(const State *const state, unsigned char *key)
{
    int index = 0;

    // 1. çubuktaki diskleri anahtara eklemek için
    for (int i = 0; i < 3; i++) {
        if (state->rodA[i] != 0) {
            key[index++] = 'A';
            key[index++] = state->rodA[i] + '0';
        }
    }

    for (int i = 0; i < 3; i++) {
        if (state->rodB[i] != 0) {
            key[index++] = 'B';
            key[index++] = state->rodB[i] + '0';
        }
    }

    for (int i = 0; i < 3; i++) {
        if (state->rodC[i] != 0) {
            key[index++] = 'C';
            key[index++] = state->rodC[i] + '0';
        }
    }

    key[index] = '\0';
```

```
    //printf("Anahtar: %s\n", key);

    if (index > MAX_KEY_SIZE)
    {
        printf("ERROR: MAX_KEY_SIZE is exceeded in Generate_HashTable_Key. \n");
        exit(-1);
    }
}

// ======= YOU DO NOT NEED TO CHANGE THIS COMPULSORY DECLARATIONS ==============

//____________________ Determine whether x is prime or not_____________________
static int is_prime(const unsigned int x)
{
    int i;

    if (x < 2)
        return FALSE;

    for (i = 2; i <= x / 2; i++)
    {
        if (x % i == 0)
            return FALSE;
    }

    return TRUE;
}

//________ Return the next prime after x, or x if x is prime_________________
static unsigned next_prime(unsigned int x)
{
    while (is_prime(x) == FALSE)
    {
        x++;
    }
    return x;
}

//____________________ Hash Function _____________________________________
static unsigned int hash_func(const char *key, const int size)
{
    unsigned int hash = 0, i;
    // a should be a prime number larger than the size of the alphabet
    const int a = 151;
    const int length_key = strlen(key);

    for (i = 0; i < length_key; i++)
    {
        hash += (unsigned int)pow(a, length_key - (i + 1)) * key[i];
        hash = hash % size;
    }

    return hash;
}

//____________________ Create new Hash Table_________________________
```

```
Hash_Table *New_Hash_Table(const int size)
{
    Hash_Table *ht = (Hash_Table *)malloc(sizeof(Hash_Table));
    if (ht == NULL)
        Warning_Memory_Allocation();

    ht->size = next_prime(size);

    ht->count = 0;

    ht->State_Key = (unsigned char **)calloc(ht->size, sizeof(unsigned char *));
    if (ht->State_Key == NULL)
        Warning_Memory_Allocation();

    return ht;
}

//____________________ Insert ___________________________________
void ht_insert(Hash_Table *ht, const State *const state)
{
    char key[MAX_KEY_SIZE];

    Generate_HashTable_Key(state, key);
    ht_insert_key(ht, key);
}

void ht_insert_key(Hash_Table *ht, const char *key)
{
    unsigned int index;
    unsigned const int load = ht->count * 100 / ht->size;
    unsigned int new_size;

    if (load > HASH_TABLE_INCREASING_RATE)
    {
        new_size = ht->size * 2;
        Resize_Hash_Table(ht, new_size);
    }

    if (ht->size == ht->count)
    {
        printf("ERROR: Hash table is full.\n");
        exit(-1);
    }

    index = hash_func(key, ht->size);
    while (ht->State_Key[index] != NULL)
    {
        if (index == ht->size - 1)
            index = 0;
        else
            index++;
    }

    ht->State_Key[index] = (unsigned char *)malloc(MAX_KEY_SIZE * sizeof(unsigned char *));
    if (ht->State_Key[index] == NULL)
        Warning_Memory_Allocation();
```

```
    strcpy(ht->State_Key[index], key);
    ht->count++;
}

//____________________ Search ___________________________________
int ht_search(Hash_Table *ht, const State *const state)
{
    char key[MAX_KEY_SIZE];
    unsigned int first_index, index;

    Generate_HashTable_Key(state, key);
    index = hash_func(key, ht->size);
    first_index = index;

    printf("key = %s,  index = %u\n", key, index);

    while (ht->State_Key[index] != NULL)
    {
        if (strcmp(ht->State_Key[index], key) == 0)
            return TRUE;

        if (index == ht->size - 1)
            index = 0;
        else
            index++;

        if (index == first_index)
            return FALSE;
    }

    return FALSE;
}

//________ Resize Hash Table _____________________________________
void Resize_Hash_Table(Hash_Table *ht, const int size)
{
    int i;
    unsigned int temp_size, temp_count;
    unsigned char **temp_key;
    Hash_Table *new_ht = New_Hash_Table(size);

    // create new larger hash table
    for (i = 0; i < ht->size; i++)
    {
        if (ht->State_Key[i] != NULL)
        {
            ht_insert_key(new_ht, ht->State_Key[i]);
        }
    }

    // swap size
    temp_size = ht->size;
    ht->size = new_ht->size;
    new_ht->size = temp_size;

    // swap count
```

```
    temp_count = ht->count;
    ht->count = new_ht->count;
    new_ht->count = temp_count;

    // swap keys
    temp_key = ht->State_Key;
    ht->State_Key = new_ht->State_Key;
    new_ht->State_Key = temp_key;

    Delete_Hash_Table(new_ht);
}

//________ Delete Hash Table ____________________________________
void Delete_Hash_Table(Hash_Table *ht)
{
    int i;

    for (i = 0; i < ht->size; i++)
    {
        if (ht->State_Key[i] != NULL)
        {
            free(ht->State_Key[i]);
        }
    }
    free(ht->State_Key);
    free(ht);
}

//____________________ Show hash table ___________________________________
void Show_Hash_Table(Hash_Table *ht)
{
    unsigned int i;

    printf("\nHASH TABLE IS (Size = %u, Count = %u ): \n", ht->size, ht->count);
    for (i = 0; i < ht->size; i++)
        if (ht->State_Key[i] != NULL)
            printf("[%u] --> %s\n", i, ht->State_Key[i]);
}
```

5. graph$_search.h$

```
#ifndef GRAPH_SEARCH_H
#define GRAPH_SEARCH_H


#include "data_types.h"

// =================== WRITE YOUR OPTIONAL COMMANDS ==========================
#define TOWER_NUMBER 3
#define DISK_NUMBER 3

// ========  YOUR COMPULSORY (BUT SPECIFIC TO THE PROBLEM) COMMANDS ============
#define PREDETERMINED_GOAL_STATE 1  // User will initially determine the goal state if it is true (1)
#define ACTIONS_NUMBER 6          // The number of all possible actions
#define MAX_SEARCHED_NODE 1000000000// exit from the search process if it is exceeded

// ================== YOU DO NOT NEED TO CHANGE THIS PART =====================
#define NO_ACTION 0
#define TRUE 1
#define FALSE 0
#define FAILURE NULL

// =========  YOUR COMPULSORY (BUT SPECIFIC TO THE PROBLEM) DECLARATIONS =======
State *Create_State();
void Print_State(const State *const state);
void Print_Action(const enum ACTIONS action);
int Result(const State *const parent_state, const enum ACTIONS action, Transition_Model *const trans_m
float Compute_Heuristic_Function(const State *const state, const State *const goal);
int Goal_Test(const State *const state, const State *const goal_state);

// ======= YOU DO NOT NEED TO CHANGE THIS COMPULSORY DECLARATIONS EXCEPT INSERTION OF THE GENERALIZED
Node *First_GoalTest_Search_TREE(const enum METHODS method, Node *const root, State *const goal_state)
Node *First_InsertFrontier_Search_TREE(const enum METHODS method, Node *const root, State *const goal_
Node *DepthType_Search_TREE(const enum METHODS method, Node *const root, State *const goal_state, cons
Node *Child_Node(Node *const parent, const enum ACTIONS action);
Queue *Start_Frontier(Node *const root);
int Empty(const Queue *const frontier);
Node *Pop(Queue **frontier);
void Insert_FIFO(Node *const child, Queue **frontier);
void Insert_LIFO(Node *const child, Queue **frontier);
void Insert_Priority_Queue_UniformSearch(Node *const child, Queue **frontier);
void Insert_Priority_Queue_GreedySearch(Node *const child, Queue **frontier);
void Insert_Priority_Queue_A_Star(Node *const child, Queue **frontier);
void Insert_Priority_Queue_GENERALIZED_A_Star(Node *const child, Queue **frontier, float alpha); // Up
void Print_Frontier(Queue *const frontier);
void Show_Solution_Path(Node *const goal);
void Print_Node(const Node *const node);
int Level_of_Node(Node *const node);
void Clear_All_Branch(Node *node, int *Number_Allocated_Nodes);
void Clear_Single_Branch(Node *node, int *Number_Allocated_Nodes);
void Warning_Memory_Allocation();
int Compare_States(const State *const state1, const State *const state2);
Node *Frontier_search(Queue *const frontier, const State *const state);
void Remove_Node_From_Frontier(Node *const old_child, Queue **const frontier);
```

```
#endif //GRAPH_SEARCH_H
```

6. hash_table.h

```
#ifndef HASH_TABLE_H
#define HASH_TABLE_H


#include "data_types.h"

// ========  YOUR COMPULSORY (BUT SPECIFIC TO THE PROBLEM) COMMANDS ============
#define HASH_TABLE_BASED_SIZE 25
#define HASH_TABLE_INCREASING_RATE 70
#define MAX_KEY_SIZE 20

// ================== YOU DO NOT NEED TO CHANGE THIS PART =====================
typedef struct
{
    unsigned int size;
    unsigned int count;
    unsigned char **State_Key;
} Hash_Table;

// =========  YOUR COMPULSORY (BUT SPECIFIC TO THE PROBLEM) DECLARATIONS =======
void Generate_HashTable_Key(const State *const state, unsigned char *key);

// ======= YOU DO NOT NEED TO CHANGE THIS COMPULSORY DECLARATIONS ==============
Hash_Table *New_Hash_Table(const int size);
void Resize_Hash_Table(Hash_Table *ht, const int size);
void Delete_Hash_Table(Hash_Table *ht);
void ht_insert(Hash_Table *ht, const State *const state);
void ht_insert_key(Hash_Table *ht, const char *key);
int ht_search(Hash_Table *ht, const State *const state);
void Show_Hash_Table(Hash_Table *ht);

#endif //HASH_TABLE_H
```

7. Standart_Search.c

[language=C] void Insert$_{P}riority_{Q}ueue_{G}ENERALIZED_{A}Star(Node*constchild, Queue**frontier, floatalpha)//UPD_{A}$
return;

```
/*
These functions are standard for graph search algorithms and you do not need to
change them for different search problems.

However, if you insert the generalized A* algorithm,
you should update First_InsertFrontier_Search_TREE()
  and Insert_Priority_Queue_GENERALIZED_A_Star(),  which is similar to Insert_Priority_Queue_A_Star()
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "graph_search.h"
```

```
#include "data_types.h"
#include "hash_table.h"

//_____________________________________________________________________________
Node *First_InsertFrontier_Search_TREE(const enum METHODS method, Node *const root, State *const goal_
{
int Number_Searched_Nodes = 0; // The number of nodes passing goal test
int Number_Generated_Nodes = 1; // The number of generated nodes (The first one is the root)
int Number_Allocated_Nodes = 1; // The number of nodes in memory (The first one is the root)
enum ACTIONS action;
Node *node, *child, *temp_node;
Queue *frontier;
Hash_Table *explorer_set;

// a priority queue ordered by PATH-COST(or evaluation function f), with node as the only element
frontier = Start_Frontier(root);
Print_Frontier(frontier);

explorer_set = New_Hash_Table(HASH_TABLE_BASED_SIZE);
Show_Hash_Table(explorer_set);

while (Number_Searched_Nodes < MAX_SEARCHED_NODE)
{
if (Empty(frontier))
return FAILURE;

node = Pop(&frontier);

// GOAL-TEST
Number_Searched_Nodes++;
if (Goal_Test(&(node->state), goal_state))
{
printf("\nThe number of searched nodes is : %d\n", Number_Searched_Nodes);
printf("\nThe number of generated nodes is : %d\n", Number_Generated_Nodes);
printf("\nThe number of generated nodes in memory is : %d\n", Number_Allocated_Nodes);
Delete_Hash_Table(explorer_set);
return node;
}

ht_insert(explorer_set, &(node->state));
Show_Hash_Table(explorer_set);

for (action = 0; action < ACTIONS_NUMBER; action++)
{
child = Child_Node(node, action);

if (child != NULL)
{
Number_Generated_Nodes++;
Number_Allocated_Nodes++;

if (ht_search(explorer_set, &(child->state)))
continue; // child.STATE is in explored set

temp_node = Frontier_search(frontier, &(child->state));
```

```
switch (method)
{
case UniformCostSearch:
if (temp_node != NULL)
{
if (child->path_cost < temp_node->path_cost) // child.STATE has been in frontier with higher cost
Remove_Node_From_Frontier(temp_node, &frontier);
else // child.STATE has already been in frontier with lower cost
continue;
}
Insert_Priority_Queue_UniformSearch(child, &frontier);
break;
case AStarSearch:
if (temp_node != NULL)
{
if (child->path_cost + child->state.h_n < temp_node->path_cost + temp_node->state.h_n) // child.STATE
Remove_Node_From_Frontier(temp_node, &frontier);
else // child.STATE has already been in frontier with lower cost
continue;
}
child->state.h_n = Compute_Heuristic_Function(&(child->state), goal_state);
Insert_Priority_Queue_A_Star(child, &frontier);
break;
case GeneralizedAStarSearch:
if (temp_node != NULL) {
float child_eval = alpha * (child->path_cost + child->state.h_n) + (1 - alpha) * child->path_cost;
float temp_node_eval = alpha * (temp_node->path_cost + temp_node->state.h_n) + (1 - alpha) * temp_node

if (child_eval < temp_node_eval) // child.STATE has been in frontier with higher cost
Remove_Node_From_Frontier(temp_node, &frontier);
else // child.STATE has already been in frontier with lower cost
continue;
}
child->state.h_n = Compute_Heuristic_Function(&(child->state), goal_state);
Insert_Priority_Queue_A_Star(child, &frontier);
break;

temp_node değişkeni NULL değilse, yani geçerli bir düğüm varsa
bir karşılaştırma yapılır.
Bu karşılaştırmada, child düğümünün önceden frontier (öncelikli kuyruk) yapısında
var olup olmadığı kontrol edilir ve varsa, bu düğümün
değerlendirme fonksiyonu (child_eval) ve temp_node ile karşılaştırılır.
Bu değerlendirme,  (alpha) değeri kullanılarak hesaplanır
Çocuk düğümün maliyeti ile heuristik fonksiyonun değeri
arasında bir ağırlıklandırma yapılır.

Eğer child_eval, temp_node_eval'den küçükse, yani çocuk düğümün değerlendirme maliyeti
daha düşükse, bu durumda temp_node önceden frontier yapısında var olmuş bir düğümdür
Daha yüksek bir maliyetle var olmuştur. Bu durumda,
bu düğüm frontier yapısından kaldırılır
Remove_Node_From_Frontier fonksiyonu çağrılır

Eğer çocuk düğümün değerlendirme maliyeti, önceden frontier yapısında
var olan bir düğümün  değerlendirme maliyetinden daha yüksekse veya temp_node
değişkeni NULL ise, bu durumda çocuk düğüm frontier yapısına eklenir.
Öncelikli kuyruğa ekleme işlemi Insert_Priority_Queue_A_Star fonksiyonu
```

```
aracılığıyla gerçekleştirilir. Döngüden çıkılır ve sonraki adıma geçer.

default:
printf("ERROR: Unknown method in First_InsertFrontier_Search_TREE.\n");
Delete_Hash_Table(explorer_set);
exit(-1);
}
Print_Frontier(frontier);
}
}
}

printf("Maximum number of searched nodes is exceeded. %d nodes are searched, but the goal could not fo
Delete_Hash_Table(explorer_set);
return FAILURE;
}

//_____________________________________________________________________________
Node *First_GoalTest_Search_TREE(const enum METHODS method, Node *const root, State *const goal_state)
{
int Number_Searched_Nodes = 0; // The number of nodes passing goal test
int Number_Generated_Nodes = 1; // The number of generated nodes (The first one is the root)
int Number_Allocated_Nodes = 1; // The number of nodes in memory (The first one is the root)
enum ACTIONS action;
Node *node, *child;
Queue *frontier;
Hash_Table *explorer_set;

// GOAL-TEST
Number_Searched_Nodes++;
if (Goal_Test(&(root->state), goal_state))
{
printf("\nThe number of searched nodes is : %d\n", Number_Searched_Nodes);
printf("\nThe number of generated nodes is : %d\n", Number_Generated_Nodes);
printf("\nThe number of generated nodes in memory is : %d\n", Number_Allocated_Nodes);
return root;
}

frontier = Start_Frontier(root);
Print_Frontier(frontier);

explorer_set = New_Hash_Table(HASH_TABLE_BASED_SIZE);
Show_Hash_Table(explorer_set);

while (Number_Searched_Nodes < MAX_SEARCHED_NODE)
{
if (Empty(frontier))
return FAILURE;

node = Pop(&frontier);

ht_insert(explorer_set, &(node->state));
Show_Hash_Table(explorer_set);

for (action = 0; action < ACTIONS_NUMBER; action++)
{
```

```
child = Child_Node(node, action);

if (child != NULL)
{
Number_Generated_Nodes++;
Number_Allocated_Nodes++;

if (ht_search(explorer_set, &(child->state)) || Frontier_search(frontier, &(child->state)) != NULL)
continue; // child.STATE is in explored set or frontier

// GOAL-TEST
Number_Searched_Nodes++;
if (Goal_Test(&(child->state), goal_state))
{
printf("\nThe number of searched nodes is : %d\n", Number_Searched_Nodes);
printf("\nThe number of generated nodes is : %d\n", Number_Generated_Nodes);
printf("\nThe number of generated nodes in memory is : %d\n", Number_Allocated_Nodes);
Delete_Hash_Table(explorer_set);
if (method == GreedySearch && !PREDETERMINED_GOAL_STATE)
child->state.h_n = Compute_Heuristic_Function(&(child->state), goal_state);
return child;
}

switch (method)
{
case BreastFirstSearch:
Insert_FIFO(child, &frontier);
break;
case GreedySearch:
child->state.h_n = Compute_Heuristic_Function(&(child->state), goal_state);
Insert_Priority_Queue_GreedySearch(child, &frontier);
break;
default:
printf("ERROR: Unknown method in First_GoalTest_Search_TREE.\n");
exit(-1);
}
Print_Frontier(frontier);
}
}
}

printf("Maximum number of searched nodes is exceeded. %d nodes are searched, but the goal could not fo
Delete_Hash_Table(explorer_set);
return FAILURE;
}

//_____________________________________________________________________________
Node *DepthType_Search_TREE(const enum METHODS method, Node *const root, State *const goal_state, cons
{
static int Number_Searched_Nodes = 0;  // The number of nodes passing goal test
static int Number_Generated_Nodes = 1; // The number of generated nodes (The first one is the root)
static int Number_Allocated_Nodes = 1; // The number of nodes in memory (The first one is the root)
enum ACTIONS action;
Node *node, *child;
Queue *frontier;
Hash_Table *explorer_set;
```

```
// GOAL-TEST
Number_Searched_Nodes++;
if (Goal_Test(&(root->state), goal_state))
{
printf("\nThe number of searched nodes is : %d\n", Number_Searched_Nodes);
printf("\nThe number of generated nodes is : %d\n", Number_Generated_Nodes);
printf("\nThe number of generated nodes in memory is : %d\n", Number_Allocated_Nodes);
return root;
}

frontier = Start_Frontier(root);
Print_Frontier(frontier);

explorer_set = New_Hash_Table(HASH_TABLE_BASED_SIZE);
Show_Hash_Table(explorer_set);

while (Number_Searched_Nodes < MAX_SEARCHED_NODE)
{
if (Empty(frontier))
return FAILURE;

node = Pop(&frontier);

ht_insert(explorer_set, &(node->state));
Show_Hash_Table(explorer_set);

if (method == DepthLimitedSearch || method == IterativeDeepeningSearch)
if (Level_of_Node(node) == Max_Level)
{
Clear_All_Branch(node, &Number_Allocated_Nodes);
continue;
}

for (action = 0; action < ACTIONS_NUMBER; action++)
{
child = Child_Node(node, action);

if (child != NULL)
{
Number_Generated_Nodes++;
Number_Allocated_Nodes++;

if (ht_search(explorer_set, &(child->state)) || Frontier_search(frontier, &(child->state)) != NULL)
{
Clear_Single_Branch(child, &Number_Allocated_Nodes); // if child.STATE is in explored set or frontier
}
else
{
// GOAL-TEST
Number_Searched_Nodes++;
if (Goal_Test(&(child->state), goal_state))
{
printf("\nThe number of searched nodes is : %d\n", Number_Searched_Nodes);
printf("\nThe number of generated nodes is : %d\n", Number_Generated_Nodes);
printf("\nThe number of generated nodes in memory is : %d\n", Number_Allocated_Nodes);
```

```
Delete_Hash_Table(explorer_set);
return child;
}

Insert_LIFO(child, &frontier);
Print_Frontier(frontier);
}
}

if (action == ACTIONS_NUMBER - 1 && node->Number_of_Child == 0) // If node has not child, clear it
Clear_All_Branch(node, &Number_Allocated_Nodes);
}
}

printf("%d nodes are searched, but the goal could not found.\n", Number_Searched_Nodes);
Delete_Hash_Table(explorer_set);
return FAILURE;
}

//_____________________________________________________________________________
Node *Child_Node(Node *const parent, const enum ACTIONS action)
{
Node *child = NULL;
Transition_Model trans_model;
if (Result(&(parent->state), action, &trans_model))
{
child = (Node *)malloc(sizeof(Node));
if (child == NULL)
Warning_Memory_Allocation();

child->state = trans_model.new_state;
child->path_cost = parent->path_cost + trans_model.step_cost;
child->parent = parent;
child->action = action;
child->Number_of_Child = 0;
child->parent->Number_of_Child++;
}
return child;
}

//_____________________________________________________________________________
Queue *Start_Frontier(Node *const root)
{
Queue *frontier = (Queue *)malloc(sizeof(Queue));

if (frontier == NULL)
Warning_Memory_Allocation();

frontier->node = root;
frontier->next = NULL;

return frontier;
}

//_____________________________________________________________________________
int Empty(const Queue *const frontier)
```

```
{
return (frontier == NULL);
}

//_____________________________________________________________________________
Node *Pop(Queue **frontier)
{
Node *node = NULL;
Queue *temp_queue;

if (!Empty(*frontier))
{
node = (*frontier)->node;
temp_queue = *frontier;
*frontier = (*frontier)->next;
free(temp_queue);
}

printf("\nPOP: ");
Print_Node(node);
printf("\n");

return node;
}

//_____________________________________________________________________________
void Insert_FIFO(Node *const child, Queue **frontier)
{
Queue *temp_queue;
Queue *new_queue = (Queue *)malloc(sizeof(Queue));
if (new_queue == NULL)
Warning_Memory_Allocation();

new_queue->node = child;
new_queue->next = NULL;

if (Empty(*frontier))
*frontier = new_queue;
else
{ // If frontier is not empty, find the last element of the queue.
for (temp_queue = *frontier; temp_queue->next != NULL; temp_queue = temp_queue->next)
;
temp_queue->next = new_queue;
}
}

//_____________________________________________________________________________
void Insert_LIFO(Node *const child, Queue **frontier)
{
Queue *new_queue = (Queue *)malloc(sizeof(Queue));
if (new_queue == NULL)
Warning_Memory_Allocation();

new_queue->node = child;
new_queue->next = *frontier;
*frontier = new_queue;
```

```
}

//_____________________________________________________________________________
void Insert_Priority_Queue_UniformSearch(Node *const child, Queue **frontier)
{
Queue *temp_queue;
Queue *new_queue = (Queue *)malloc(sizeof(Queue));
if (new_queue == NULL)
Warning_Memory_Allocation();

new_queue->node = child;

if (Empty(*frontier))
{
new_queue->next = NULL;
*frontier = new_queue;
}
else
{ // If frontier is not empty, find appropriate element according to ordered cost.
if (child->path_cost < (*frontier)->node->path_cost)
{ // Child has lowest cost
new_queue->next = *frontier;
*frontier = new_queue;
}
else
{
for (temp_queue = *frontier; temp_queue->next != NULL; temp_queue = temp_queue->next)
{
if (child->path_cost < temp_queue->next->node->path_cost)
{
new_queue->next = temp_queue->next;
temp_queue->next = new_queue;
return;
}
} // If child has highest cost
temp_queue->next = new_queue;
new_queue->next = NULL;
}
}
}

//_____________________________________________________________________________
void Insert_Priority_Queue_GreedySearch(Node *const child, Queue **frontier)
{
Queue *temp_queue;
Queue *new_queue = (Queue *)malloc(sizeof(Queue));
if (new_queue == NULL)
Warning_Memory_Allocation();

new_queue->node = child;

if (Empty(*frontier))
{
new_queue->next = NULL;
*frontier = new_queue;
}
```

```
else
{ // If frontier is not empty, find appropriate element according to ordered cost.
if (child->state.h_n < (*frontier)->node->state.h_n)
{ // Child has lowest cost
new_queue->next = *frontier;
*frontier = new_queue;
}
else
{
for (temp_queue = *frontier; temp_queue->next != NULL; temp_queue = temp_queue->next)
{
if (child->state.h_n < temp_queue->next->node->state.h_n)
{
new_queue->next = temp_queue->next;
temp_queue->next = new_queue;
return;
}
} // If child has highest cost
temp_queue->next = new_queue;
new_queue->next = NULL;
}
}
}

//_____________________________________________________________________________
void Insert_Priority_Queue_A_Star(Node *const child, Queue **frontier)
{
Queue *temp_queue;
Queue *new_queue = (Queue *)malloc(sizeof(Queue));
if (new_queue == NULL)
Warning_Memory_Allocation();

new_queue->node = child;

if (Empty(*frontier))
{
new_queue->next = NULL;
*frontier = new_queue;
}
else
{ // If frontier is not empty, find appropriate element according to ordered evaluation function value
if (child->path_cost + child->state.h_n < (*frontier)->node->path_cost + (*frontier)->node->state.h_n)
{ // Child has the lowest cost evaluation function value
new_queue->next = *frontier;
*frontier = new_queue;
}
else
{
for (temp_queue = *frontier; temp_queue->next != NULL; temp_queue = temp_queue->next)
{
if (child->path_cost + child->state.h_n < temp_queue->next->node->path_cost + temp_queue->next->node->
{
new_queue->next = temp_queue->next;
temp_queue->next = new_queue;
return;
}
```

```
} // If child has the highest evaluation function value
temp_queue->next = new_queue;
new_queue->next = NULL;
}
}
}

//_____________________________________________________________________________
void Insert_Priority_Queue_GENERALIZED_A_Star(Node *const child, Queue **frontier, float alpha)
{
// UPDATE THIS FUNCTION FOR HE GENERALIZED A* ALGORITHM

return;
}
//_____________________________________________________________________________
void Print_Frontier(Queue *const frontier)
{
Queue *temp_queue;

printf("\nQUEUE: [ ");
for (temp_queue = frontier; temp_queue != NULL; temp_queue = temp_queue->next)
{
Print_Node(temp_queue->node);
if (temp_queue->next != NULL)
printf(" ,");
}
printf(" ]\n");
}

//_____________ Remove the node old_child from the frontier__________
void Remove_Node_From_Frontier(Node *const old_child, Queue **const frontier)
{
Queue *curr_queue, *prev_queue;

for (curr_queue = *frontier; curr_queue != NULL; curr_queue = curr_queue->next)
{
if (curr_queue->node == old_child)
{
// Remove the old child
if (curr_queue == *frontier) // for the first node
*frontier = curr_queue->next;
else
prev_queue->next = curr_queue->next;
}
prev_queue = curr_queue;
}
}

//_____________________________________________________________________________
Node *Frontier_search(Queue *const frontier, const State *const state)
{
Queue *temp_queue;

for (temp_queue = frontier; temp_queue != NULL; temp_queue = temp_queue->next)
{
if (Compare_States(&(temp_queue->node->state), state))
```

```
return temp_queue->node;
}
return NULL;
}

//_____________________________________________________________________________
int Frontier_update(Queue *const frontier, const State *const state)
{
Queue *temp_queue;

for (temp_queue = frontier; temp_queue != NULL; temp_queue = temp_queue->next)
{
if (Compare_States(&(temp_queue->node->state), state))
return TRUE;
}
return FALSE;
}

//_____________________________________________________________________________
void Print_Node(const Node *const node)
{
if (node != NULL)
{
printf("NODE(");
Print_State(&(node->state));
if (node->parent)
{
printf(", parent:");
Print_State(&(node->parent->state));
printf(", action:");
Print_Action(node->action);
printf(", path_cost: %.1f )", node->path_cost);
}
else
printf(":root)");
}
else
printf("NODE:NULL");
}

//_____________________________________________________________________________
void Show_Solution_Path(Node *const goal)
{
Node *temp;
if (goal == FAILURE)
printf("THE SOLUTION CAN NOT BE FOUND.\n");
else
{
printf("\nTHE COST PATH IS %.2f.\n", goal->path_cost);
printf("\nTHE SOLUTION PATH IS:\n");
for (temp = goal; temp != NULL; temp = temp->parent)
{
Print_State(&(temp->state));
if (temp->parent != NULL)
{
printf("\n\taction(");
```

```
Print_Action(temp->action);
printf(")\n");
}
}
}
}

//_____________________________________________________________________________
int Level_of_Node(Node *const node)
{
int counter = 0;
Node *temp = node;
while (temp->parent != NULL)
{
temp = temp->parent;
counter++;
}
return counter;
}

//_____________________________________________________________________________
void Clear_All_Branch(Node *node, int *Number_Allocated_Nodes)
{
Node *parent = node->parent;
if (Level_of_Node(node) == 0)
return;

Clear_Single_Branch(node, Number_Allocated_Nodes);

if (parent->Number_of_Child == 0) // Clear nodes having no child.
Clear_All_Branch(parent, Number_Allocated_Nodes);
}

void Clear_Single_Branch(Node *node, int *Number_Allocated_Nodes)
{
if (Level_of_Node(node) == 0)
return;

printf("\nCLEARING: ");
Print_Node(node);
printf("\n");
node->parent->Number_of_Child--;
free(node);
(*Number_Allocated_Nodes)--;
}

//_____________________________________________________________________________
void Warning_Memory_Allocation()
{
printf("The memory Error in alloacation process! Press a key to exit.\n");
exit(-1);
}

//_____________________________________________________________________________
int Compare_States(const State *const state1, const State *const state2)
{
```

```
unsigned char key1[MAX_KEY_SIZE], key2[MAX_KEY_SIZE];
Generate_HashTable_Key(state1, key1);
Generate_HashTable_Key(state2, key2);
return !strcmp(key1, key2);
}
```

8. SpesificToProblem.c

```
/*
    These functions are compulsory for search algorithms but they are specific
    to problems. More clearly, you must must update their blocks but do not change
    their input and output parameters.

    Also, you can add new functions at the end of file by declaring them in GRAPH_SEARCH.h
*/

#include "graph_search.h"
#include "data_types.h"
#include <stdio.h>
#include <stdlib.h>

int FindTargetRodIndex(int rod[]);
int FindDiskToMoveIndex(int rod[]);
int IsValidMove(const State *const state, const enum ACTIONS action, int disk, int target_rod_index);
void MoveDisk(State *state, const enum ACTIONS action, int disk, int target_rod_index);


//_____________________________________________________________________________
State *Create_State()
{
    State *state = (State *)malloc(sizeof(State));
    if (state == NULL)
        Warning_Memory_Allocation();

    for (int i = 0; i < DISK_NUMBER; i++) {
        state->rodA[i] = DISK_NUMBER - i;
        state->rodB[i] = 0;
        state->rodC[i] = 0;
    }

    return state;
}

void Print_State(const State *const state)
{
    printf("A: ");
    for (int i = 0; i < DISK_NUMBER; i++) {
        if (state->rodA[i] != 0) {
            printf("%d ", state->rodA[i]);
        } else {
            printf("- ");
        }
    }
    printf("\n");
```

```
    printf("B: ");
    for (int i = 0; i < DISK_NUMBER; i++) {
        if (state->rodB[i] != 0) {
            printf("%d ", state->rodB[i]);
        } else {
            printf("- ");
        }
    }
    printf("\n");

    printf("C: ");
    for (int i = 0; i < DISK_NUMBER; i++) {
        if (state->rodC[i] != 0) {
            printf("%d ", state->rodC[i]);
        } else {
            printf("- ");
        }
    }
    printf("\n");
}


void Print_Action(const enum ACTIONS action)
{
    switch (action)
    {
        case Move_A_to_B:
            printf("Move disk from A to B\n");
        break;
        case Move_A_to_C:
            printf("Move disk from A to C\n");
        break;
        case Move_B_to_A:
            printf("Move disk from B to A\n");
        break;
        case Move_B_to_C:
            printf("Move disk from B to C\n");
        break;
        case Move_C_to_A:
            printf("Move disk from C to A\n");
        break;
        case Move_C_to_B:
            printf("Move disk from C to B\n");
        break;
    }
}
int Result(const State *const parent_state, const enum ACTIONS action, Transition_Model *const trans_m
    State new_state = *parent_state;
    int disk_to_move_index = -1;
    int target_rod_index = -1;

    switch (action) {
        case Move_A_to_B:
            disk_to_move_index = FindDiskToMoveIndex(new_state.rodA);
            target_rod_index = FindTargetRodIndex(new_state.rodB);
            break;
```

```
        case Move_A_to_C:
            disk_to_move_index = FindDiskToMoveIndex(new_state.rodA);

            target_rod_index = FindTargetRodIndex(new_state.rodC);
            break;
        case Move_B_to_A:
            disk_to_move_index = FindDiskToMoveIndex(new_state.rodB);

            target_rod_index = FindTargetRodIndex(new_state.rodA);
            break;
        case Move_B_to_C:
            disk_to_move_index = FindDiskToMoveIndex(new_state.rodB);

            target_rod_index = FindTargetRodIndex(new_state.rodC);

            break;
        case Move_C_to_A:
            disk_to_move_index = FindDiskToMoveIndex(new_state.rodC);

            target_rod_index = FindTargetRodIndex(new_state.rodA);
            break;
        case Move_C_to_B:
            disk_to_move_index = FindDiskToMoveIndex(new_state.rodC);

            target_rod_index = FindTargetRodIndex(new_state.rodB);
            break;
    }

    // Hareket geçerli mi kontrol et
    int isValidMove = IsValidMove(parent_state, action, disk_to_move_index, target_rod_index);

    if (!isValidMove) {
        return FALSE;
    }

    // Diski taşı
    MoveDisk(&new_state, action, disk_to_move_index, target_rod_index);

    trans_model->new_state = new_state;
    trans_model->step_cost = 1;

    return TRUE;
}

% FindTargetRodIndex fonksiyonu, verilen çubuğun indekslerini tarar ve
ilk boş hedef çubuğunun indeksini bulur. Eğer hiç boş hedef çubuğu yoksa,
-1 döndürür.

FindDiskToMoveIndex fonksiyonu, verilen çubuğun indekslerini ters sırayla
tarar ve en alttaki disk indeksini bulur. Eğer çubukta hiç disk yoksa,
-1 döndürür.

int FindTargetRodIndex(int rod[]) {
    for (int i =0; i < DISK_NUMBER; i++) {
        if (rod[i] == 0) {
            return i;
```

```
        }
    }
    return -1;
}
int FindDiskToMoveIndex(int rod[]) {
    for (int i = DISK_NUMBER - 1; i >=0 ; i--) {
        if (rod[i] != 0) {
            return i;
        }
    }
    return -1;
}
// Hareketin geçerli olup olmadığını kontrol et
int IsValidMove(const State *const state, const enum ACTIONS action, int disk_to_move_index, int targe
    if (disk_to_move_index == -1 || target_rod_index == -1)
        return 0;

    if (target_rod_index == 0) {
        return 1;
    }

    switch (action) {
        case Move_A_to_B:


            if (state->rodB[target_rod_index - 1] < state->rodA[disk_to_move_index]) {
                return 0; // Geçersiz hareket, büyük disk üzerine küçük disk
            }
            break;
        case Move_A_to_C:

            if (state->rodC[target_rod_index - 1] < state->rodA[disk_to_move_index]) {
                return 0;
            }
            break;
        case Move_B_to_A:

            if (state->rodA[target_rod_index  - 1 ] < state->rodB[disk_to_move_index]) {
                return 0;
            }
            break;
        case Move_B_to_C:
            if (state->rodC[target_rod_index - 1] < state->rodB[disk_to_move_index]) {
                return 0;
            }
            break;
        case Move_C_to_A:
            if (state->rodA[target_rod_index - 1] < state->rodC[disk_to_move_index]) {
                return 0;
            }
            break;
        case Move_C_to_B:
            if (state->rodB[target_rod_index - 1] < state->rodC[disk_to_move_index]) {
                return 0;
            }
            break;
```

```
    }
    return 1; // Geçerli hareket
}


IsValidMove metodu, belirli bir oyun durumu ve hareketle (eylem),
taşınacak diskin indeksi ve hedef çubuğun indeksi verildiğinde,
hareketin geçerli olup olmadığını kontrol eder.
Eğer hedef çubuk dizini veya taşınacak diskin dizini $-1$ ise,
bu geçersiz bir hamle olduğunu gösterir.
Ayrıca, hedef çubuğun $0$ (ilk çubuk) olması durumunda, bu her zaman

geçerli bir hamledir.
Son olarak, belirli bir eylemin (hareketin) geçerliliği kontrol edilir.
Örneğin, Move_A_to_B eylemi için, hedef çubuktaki en üstteki disk,
taşınacak diskin altına yerleştirilebilirse bu geçerli bir harekettir.
Diğer eylemler için de benzer kontrol mekanizmaları bulunur:
Hangi çubuktan hangi çubuğa taşınacağına bağlı olarak,
hedef çubuktaki en üstteki diskin boyutu kontrol edilir ve
bu, taşınacak diskten daha küçük olmalıdır.
Bu metodun temel amacı,
belirli bir hamlenin oyun kurallarına uygun olup olmadığını belirlemektir.


/// Diski taşı
void MoveDisk(State *state, const enum ACTIONS action, int disk_to_move_index, int target_rod_index) {
    switch (action) {
        case Move_A_to_B:
            state->rodB[target_rod_index] = state->rodA[disk_to_move_index];
            state->rodA[disk_to_move_index] = 0;
        break;

        case Move_A_to_C:
            state->rodC[target_rod_index] = state->rodA[disk_to_move_index];
            state->rodA[disk_to_move_index] = 0;
        break;

        case Move_B_to_A:
            state->rodA[target_rod_index] = state->rodB[disk_to_move_index];
            state->rodB[disk_to_move_index] = 0;
        break;

        case Move_B_to_C:
            state->rodC[target_rod_index] = state->rodB[disk_to_move_index];
            state->rodB[disk_to_move_index] = 0;
        break;

        case Move_C_to_A:
            state->rodA[target_rod_index] = state->rodC[disk_to_move_index];
            state->rodC[disk_to_move_index] = 0;
        break;

        case Move_C_to_B:
            state->rodB[target_rod_index] = state->rodC[disk_to_move_index];
            state->rodC[disk_to_move_index] = 0;
        break;
```

```
    }
}

Bu MoveDisk fonksiyonu, Hanoi Kuleleri problemi için diskleri taşımak için kullanılır.
Fonksiyon, mevcut durumu temsil eden bir State yapısı alır, ardından bir eylem,
diskin dizindeki indeksi ve hedef çubuğun dizindeki indeksi alır.

Fonksiyon, verilen eyleme göre diskleri taşır.
Örneğin, Move_A_to_B eylemi, A çubuğundaki bir diski B çubuğuna taşır.

Diğer eylemler de benzer şekilde, diskleri farklı çubuklara taşır.

Işlev, geçerli durumu değiştirerek, diskleri taşıdığı çubuklarda uygun yerlere
koyar. Örneğin, state->rodB[target_rod_index] = state->rodA[disk_to_move_index];
kod satırı, A çubuğundan B çubuğuna bir diski taşırken gerçekleşir.

Bu fonksiyon, Hanoi Kuleleri probleminde mevcut durumu değiştirmek için kullanılır
ve belirtilen eyleme göre diskleri farklı çubuklara taşır.

//_____________________________________________________________________________
float Compute_Heuristic_Function(const State *const state, const State *const goal)
{
    // 2^n - 1 adım gerekir, n disk sayısı.
    // En fazla adım sayısını hesapladık.

    return (float)((1 << DISK_NUMBER) - 1);
}

//_____________ Update if your goal state is not determined initially _______________________________
int Goal_Test(const State *const state, const State *const goal_state)
{
    if (PREDETERMINED_GOAL_STATE)
        return Compare_States(state, goal_state);
    else
        return 1;
}
```

9. main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "graph_search.h"
#include "data_types.h"

State *Create_Goal_State();

int main()
{
    Node root, *goal;
    State *goal_state = NULL;
    enum METHODS method;
    int Max_Level, level;
    float alpha;

    // This part must be updated if a new algorithm is added.
    printf("1 --> Breast-First Search\n");
```

```
printf("2 --> Uniform-Cost Search\n");
printf("3 --> Depth-First Search\n");
printf("4 --> Depth-Limited Search\n");
printf("5 --> Iterative Deepening Search\n");
printf("6 --> Greedy Search\n");
printf("7 --> A* Search\n");
printf("8 --> Generalized A* Search\n");
printf("Select a method to solve the problem: ");
scanf("%d", &method);
if (method == DepthLimitedSearch)
{
    printf("Enter maximum level for depth-limited search : ");
    scanf("%d", &Max_Level);
}
if (method == GeneralizedAStarSearch)
{
    printf("Enter value of alpha for Generalized A* Search : ");
    scanf("%f", &alpha);
}

// Creating the root node ...
root.parent = NULL;
root.path_cost = 0;
root.action = NO_ACTION; // The program will not use this part. (NO_ACTION-->0)
root.Number_of_Child = 0;

printf("======== SELECTION OF INITIAL STATE =============== \n");
root.state = *(Create_State());

if (PREDETERMINED_GOAL_STATE) // User will determine the goal state if it is true
{
    printf("======== SELECTION OF GOAL STATE =============== \n");
    goal_state = Create_Goal_State();
}

if (method == GreedySearch || method == AStarSearch || method == GeneralizedAStarSearch)
{
    root.state.h_n = Compute_Heuristic_Function(&(root.state), goal_state);
    if (PREDETERMINED_GOAL_STATE)
        goal_state->h_n = 0;
}

switch (method)
{
case BreastFirstSearch:
case GreedySearch:
    goal = First_GoalTest_Search_TREE(method, &root, goal_state);
    break;
case DepthFirstSearch:
case DepthLimitedSearch:
    goal = DepthType_Search_TREE(method, &root, goal_state, Max_Level);
    break;
case IterativeDeepeningSearch:
    for (level = 0; TRUE; level++)
    {
        goal = DepthType_Search_TREE(method, &root, goal_state, level);
```

```
            if (goal != FAILURE)
            {
                printf("The goal is found in level %d.\n", level);
                break;
            }
        }
        break;
    case UniformCostSearch:
    case AStarSearch:
    case GeneralizedAStarSearch:
        goal = First_InsertFrontier_Search_TREE(method, &root, goal_state, alpha);
        break;

    default:
        printf("ERROR: Unknown method.\n");
        exit(-1);
    }

    Show_Solution_Path(goal);

    return 0;


}

State *Create_Goal_State() {
    State *state = (State *)malloc(sizeof(State));
    if (state == NULL)
        Warning_Memory_Allocation();

    for (int i = 0; i < DISK_NUMBER; i++) {
        state->rodA[i] = 0; // Büyük diskler altta, küçük diskler üstte
        state->rodB[i] = 0;
        state->rodC[i] = DISK_NUMBER - i;
    }

    return state;
}
```

Create_Goal_State fonksiyonu çağrıldığında, bellekten bir State yapısı için yeterli alan ayrır ve bu alan bir state ile işaretlenir. Bu işlem malloc fonksiyonu kullanılarak gerçekleştirilir. Eğer bellek ayrılması başarısız olursa, malloc fonksiyonu NULL işaretçisi döndürecektir ve bunun kontrolünü yapar.

Oluşturulan State yapısının her çubuğu (rodA, rodB, rodC) başlangıçta disklerle dolu olmayacak şekilde ayarlar. Bu işlem bir döngü kullanılarak gerçekleştirilir. Hanoi Kuleleri problemi genellikle büyük disklerin altta ve küçük disklerin üstte olması gerektiğinden, rodC çubuğu ters sıralı olarak ayarlanır. Yani, en büyük disk en alta, en küçük disk en üste gelecek şekilde diskler sıralanır.

Oluşturulan hedef durumu yapısı state ile birlikte döner.

10. data$_types.h$

```
#ifndef DATA_TYPES_H
#define DATA_TYPES_H


Bu enum, Hanoi Kuleleri problemi için altı farklı hareketi temsil eder.

Her bir hareket, bir diskin bir çubuktan diğerine taşınmasını ifade eder.

Örneğin, Move_A_to_B, A çubuğundaki bir diskin B çubuğuna taşınması anlamına gelir.

Bu enum, kodun belirli kısımlarında, örneğin Result fonksiyonunda ve

Print_Action fonksiyonunda kullanılır. Bu fonksiyonlar, mevcut duruma bağlı olarak
hangi aksiyonların geçerli olduğunu belirlemek ve bu aksiyonları gerçekleştirmek
için enum ACTIONS'ı kullanır.


enum ACTIONS // All possible actions
{
     Move_A_to_B,
     Move_A_to_C,
     Move_B_to_A,
     Move_B_to_C,
     Move_C_to_A,
     Move_C_to_B
};

typedef struct State
{
    int rodA[3]; //disk sayısı
     int rodB[3];
    int rodC[3];
    float h_n;

} State;

// ================= YOU DO NOT NEED TO CHANGE THIS PART ====================

enum METHODS
{
    BreastFirstSearch = 1,
    UniformCostSearch = 2,
    DepthFirstSearch = 3,
    DepthLimitedSearch = 4,
    IterativeDeepeningSearch = 5,
    GreedySearch = 6,
    AStarSearch = 7,
    GeneralizedAStarSearch = 8
};

// This struct is used to determine a new state and action in transition model
typedef struct Transition_Model
{
    State new_state;
```

```
    float step_cost;
} Transition_Model;

typedef struct Node
{
    State state;
    float path_cost;
    enum ACTIONS action; // The action applied to the parent to generate this node
    struct Node *parent;
    int Number_of_Child; // required for depth-first search algorithms
} Node;

typedef struct Queue // Used for frontier
{
    Node *node;
    struct Queue *next;
} Queue;



#endif //DATA_TYPES_H
```