DATA STRUCTURES

PROJE – 4A

SOSYAL AĞ PROJESİ

TOLGAHAN AKGÜN – 05130000248

ÇAĞRI YILDIZ – 05130000203

MERVE GÜLCAN TECİK – 05120000232

Teslim Tarihi : 31.12.2014

PROGRAMCI KATALOĞU

* 1. Program yazılırken Visual Studio 2010 ,C#.NET 4.5 kullanılmıştır.
  2. Bir sosyal ağ içindeki insanlar arasındaki ilişkileri, onların arkadaşlık durumlarının bulunması istenmektedir. Burada kişiler arasındaki uzaklıklar birbirlerinin çağrılarına ne kadar süre sonra cevap verdiklerini ifade etmektedir. Buna göre Dijkstra, Depth First Traverse, Breadth First Traverse, Floyd-Warshall, Minimum Spanning Tree ile aralarındaki ilişkilere dair bilgiler gösterilmektedir.
  3. Kullanılan Sınıflar:

Graph: Çizgenin oluşturulması ve yolların bulunmasıyla ilgili metodların yer aldığı sınıf.

Edge: Çizgenin kenarlarıyla ilgili sınıf.

Vertex: Çizgenin köşeleriyle ilgili sınıf.

PriorityQ: Kenarların ve köşelerin tutulduğu sınıf.

Kullanılan Metodlar:

getAdjMat, getMaxVerts, addVertex, addEdge, displayVertex, mst, putInPQ, minDistance, dijkstra, DFS, BFS, floydWarshall, getName, getID

* 1. Kullanılan Veri Yapıları:

Queue, LinkedList, List

* 1. Programda dosya kullanılmamaktadır.
  2. Yazılım Geliştirme İçin Harcanan Süreler:

Tolgahan AKGÜN – 48 Saat

Çağrı YILDIZ – 48 Saat

Merve Gülcan TECİK – 48 Saat

--- Bu örnek için Short Path(SP) : İki kişi arasındaki en kısa yolu bulunmasında kullanılır. Aralara farklı kişiler girse dahi en etkin şekilde gideceği yolu bulur.

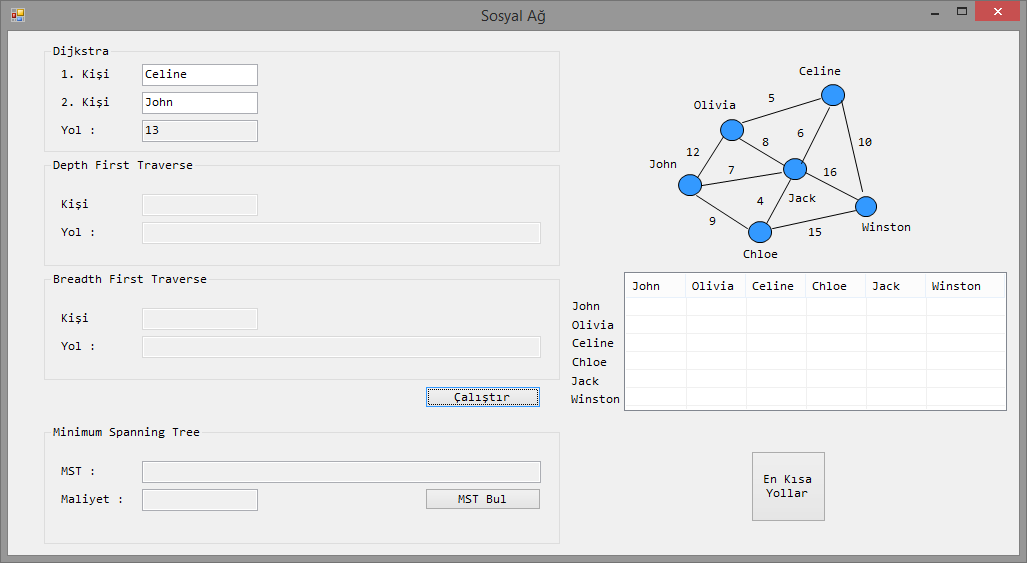
--- Bu örnek için BFS(Breadth First Tree) : Bir çizgenin bipartite olup olmadığını bulmaya, Cuthill-McKee göz(mesh) numaralandırmasına, bir ağdaki maksimum veri akışını hesaplamaya yarar.

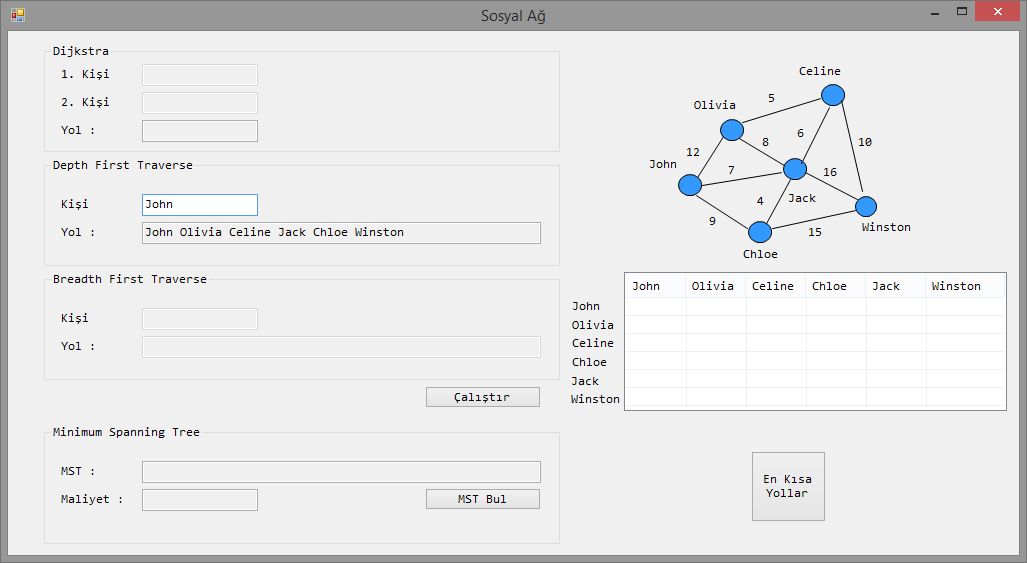
--- Bu örnek için MST(Minumum Spanning Tree) : Bir yerleşme yerindeki bilgisayarlar ağa katılacakları zaman kabloların en etkin şekilde bağlanmasını hesaplar. Elektronik devrelerde minumum alana maksimum parça entegre edilmesini ve bunların birbiriyle olan bağlarının optimize edilmesini sağlar.

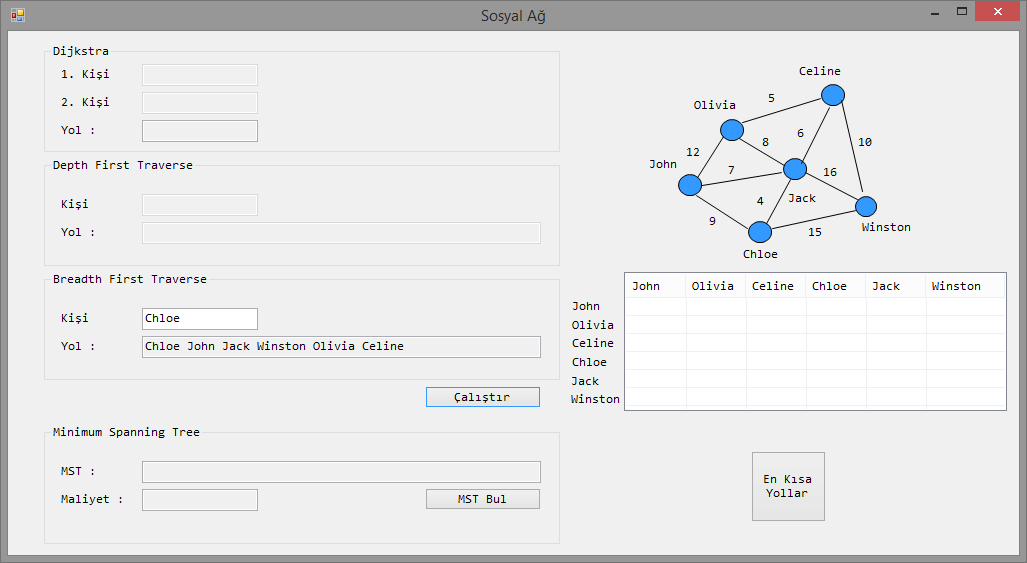
--- Bu tür bir çizge verildiğinde; kişileri masalara yerleştirirken Minimum Spanning Tree oluşturulurken elemanlar bir kuyruğa atılır. Masalara kişiler yerleştirilirken bu kuyruktan çıkma sırası esas alınır.

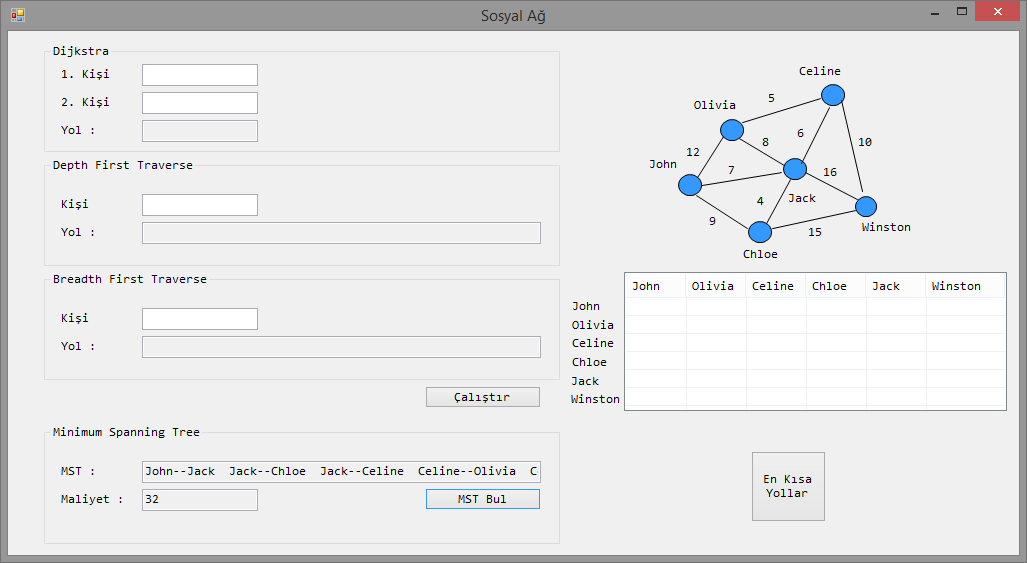
KULLANICI KATALOĞU

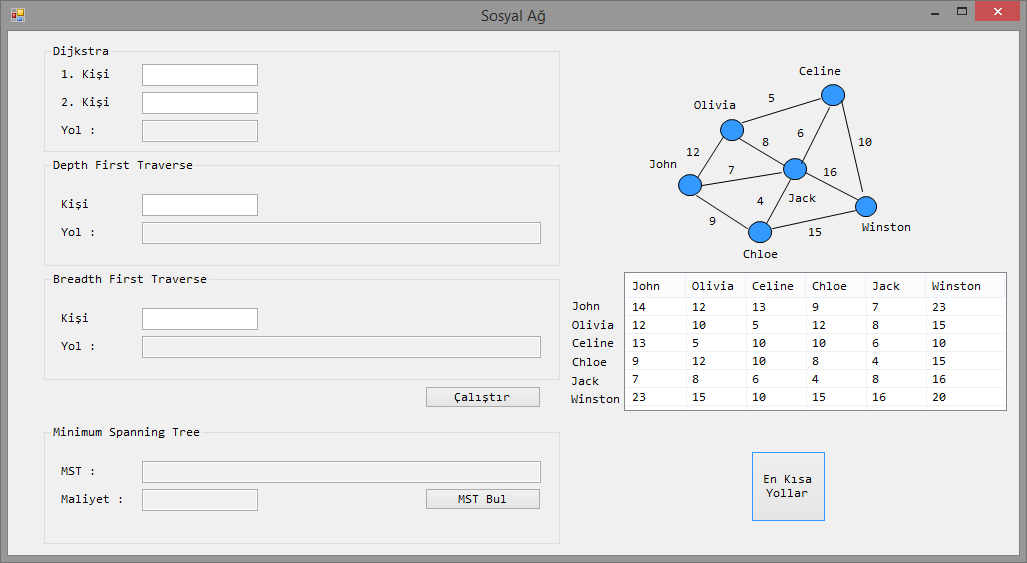
2.1 Ekran Görüntüleri









2.2 Kullanıcı Kılavuzu

Kullanıcı bulmak istediği seçeneğin verilerini girip ilgili

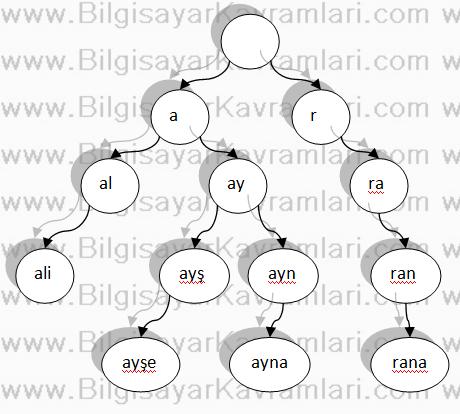
butona bastığında sonuçlar gösterilecektir.

2.3 Çizgenin güncellenmesi mümkün değildir, çizge sabittir.

Araştırma

# METİN AĞAÇLARI(trie/prefix tree)

Metin ağaçları(trie), her düğümün kendisinden sonra gelen harfi işaret ettiği multi-way [ağaçlardır](http://bilgisayarkavramlari.com/2008/05/07/agaclar-tree/). Ayrıca bilgisayar bilimlerinde digital,[radix tree](http://en.wikipedia.org/wiki/Radix_tree),prefix tree isimleriyle de anılmaktadır.Basitçe ağacın üzerine bir metin kodlanabilir ve bu metni veren ağacın üzerinde tek bir yol izlenebilir (deterministic). Veri yapılarıdaki klasik ikili arama ağaçlarından farklı olarak(binary search tree) metin ağacındaki düğümler kendilerine has bir anahtar değere sahip değillerdir. Ağaçtaki bütün çocuk düğümler, yapraklar geldikleri ebeveyn düğümdeki eki kendilerinde önek olarak bulundururlar. Bütün yapraklar ve çocuk düğümler ortak kökten yani boşluk karakterinden gelmektedirler. Ağaçtaki yaprak haricindeki çocuk düğümlerinin aranan kelimeye anahtar teşkil etmesi ve yol göstermesi haricinde bir anlamı yoktur. Yani kökten çıkarak bir yaprağa varılmadığı sürece anlamlı bir string elde edilemez. Durum aşağıdaki örnek üzerinde daha rahat anlaşılabilir:



http://www.bilgisayarkavramlari.com/wp-content/uploads/trieagaci.jpg

Yukarıdaki ağaçta dikkat edilirse kök düğüm her zaman boş metni (string) ifade etmektedir. Bu boş metin hangi harf ile devam edilirse ilgili kolu takip eder ve gitmiş olduğu düğüm o ana kadar geçmiş olduğu kollardaki harflerin birleştirilmiş halidir. Bir düğümden bir harf taşıyan sadece bir kol çıkabilir.

Yukarıdaki ağaçta dikkat edilirse kök düğüm her zaman boş metni (string) ifade etmektedir. Kökten başlayarak kelimenin içindeki harflerin sırasını teker teker takip ederek ağaçta kökten yapraklara doğru ilerlenir. Eğer sıradaki harf ile bulunulan düğümden bir alt dala gidilemez ise kelime bulunamamış demektir. Sorunsuz bir şekilde kökten başlayarak kelimedeki harflerin sırasıyla herhangi bir yaprağa ulaşabiliyorsak aradığımız kelime ağaçta yer alıyor demektir.

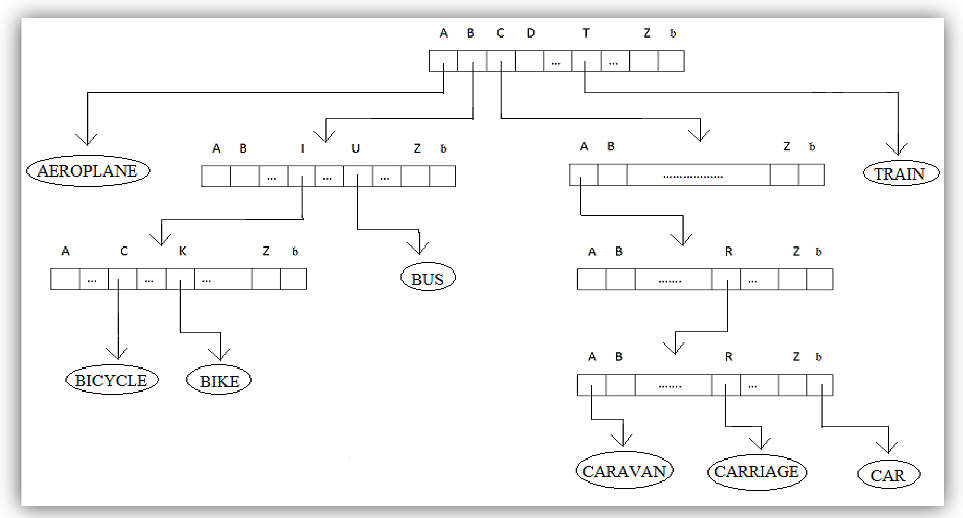
Trie ağacının ismi re**trie**val kelimesinin ortasındaki 4 harften gelmektedir.

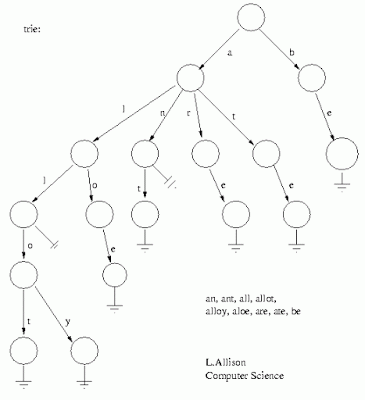
Metin ağaçlarının (trie), [ikili arama ağaçlarına](http://bilgisayarkavramlari.com/2008/05/07/ikili-arama-agaci-binary-search-tree/) göre en önemli avantajları bir metni aramanın, metin boyutu kadar işlem gerektirmesidir. [İkili arama ağaçlarında](http://bilgisayarkavramlari.com/2008/05/07/ikili-arama-agaci-binary-search-tree/) ise bu süre log n kadar varkit almaktadır. Buradaki n, ağaçtaki düğüm sayısıdır dolayısıyla [ikili arama ağaçları](http://bilgisayarkavramlari.com/2008/05/07/ikili-arama-agaci-binary-search-tree/), ağaçtaki bilgiye göre hızlı veya yavaş çalışırken, metin ağaçları, ağaçta ne kadar bilgi bulunduğundan bağımsız olarak çalışırlar.

Metin ağaçları [hafızayı](http://www.bilgisayarkavramlari.com/2008/11/07/rastgele-erisilebilir-bellek-random-access-memory-ram/) da verimli kullanırlar çünkü bir metin ağacının en derin noktası, ağaç üzerindeki en uzun metin kadardır. [İkili ağaçlar](http://bilgisayarkavramlari.com/2008/05/07/ikili-arama-agaci-binary-search-tree/)da ise bu derinlik eklenen düğüm sayısına göre en kötü ihtimalle düğüm sayısı kadar olabilmektedir.

Ayrıca metin ağaçları [en uzun önek eşlemesi (longest prefix matching)](http://bilgisayarkavramlari.com/2007/12/22/en-uzun-onek-eslesmesi-longest-prefix-matching/) gibi problemlerin çözümünde de avantaj sağlar.

İkili arama ağaçlarındaki aynı anahtar değere sahip düğümlerin eklenmesinde sorunlar çıkarken metin ağaçlarında böyle bir sorun yaşanmamaktadır.





Metin Ağacına Kelime Ekleme Algoritması

\*\*Procedure Insert(trie, keyWord)

begin

lenght <- length(keyWord)

next <- trie;

//

if trie = NULL then

trie = create empty internal node

new\_leaf = create leaf with keyWord

trie->pChildren[keyWord[0]-'a'] <- new\_leaf

exit

else

index <- next

end if

inWordIndex <- 0//

while inWordIndex < lenght and index->NotLeaf = true and

index->pChildren[keyWord[inWordIndex]-'a'] <> NULL)) do

parent <- next;

next <- index->pChildren[keyWord[inWordIndex]-'a'];

index <- next;

inWordIndex <- inWordIndex + 1

end while

if inWordIndex < lenght and index->pChildren[keyWord[inWordIndex]-'a'] = NULL and

index->NotLeaf = true then

new\_index <- NewLeaf(keyWord)

index->pChildren[keyWord[inWordIndex]-'a'] <- new\_index

exit

else

data <- next

if data->word = keyWord then

print "Word already exists in trie !!!"

else

oldChildren <- parent->pChildren[keyWord[inWordIndex-1]-'a']

newWord <- NewLeaf(keyWord)

prefixLenght <- lenght(keyWord)

if data->word[0] <> '\0' then

if lenght(data->word) < prefixLenght then

prefixLenght = lenght(data->word)

createIntern <- false

while inWordIndex <= prefixLenght and (data->word[0] <> '\0' and

(data->word[inWordIndex-1] = keyWord[inWordIndex-1]) or (data->word[0] == '\0' ) do

intern <-NewIntern()

parent->pChildren[keyWord[inWordIndex-1]-'a'] <- intern

parent->NotLeaf <- true

parent <- intern

inWordIndex <- inWordIndex +1

createIntern = true

end while

if createIntern then

inWordIndex <- inWordIndex -1

if inWordIndex <> prefixLenght or (inWordIndex = prefixLenght and

length(keyWord) = length(data->word)) then

parent->pChildren[data->word[inWordIndex]-'a'] <- oldChildren

parent->pChildren[keyWord[inWordIndex]-'a'] <- newWord

else

if data->word[0] <> '\0' then

if lenght(data->word) <= prefixLenght then

parent->pChildren[26] = oldChildren

parent->pChildren[keyWord[prefixLenght]-'a'] = newWord

else

parent->pChildren[26] = newWord

parent->pChildren[data->word[prefixLenght]-'a'] = oldChildren

end if

else

for (int count = 0 ; count < 27;count++)

parent->pChildren[count] = oldChildren->pChildren[count]

parent->pChildren[26] = newWord

end if

end if

exit

end

Metin Ağacında Kelime Arama Algoritması

\*\*Procedure FIND(trie, string)

begin

if trie = NULL then

return FALSE

else

nex <- index <- trie

count <- 0

while index->NotLeaf and count < lenght(keyWord) and

index->pChildren[keyWord[count]-'a'] <> NULL do

next <- index->pChildren[keyWord[count]-'a']

index <- next

count <- count +1

end while

if next = NULL then

return TRUE

else

data <- next

if data->word = keyWord then

return TRUE

else

if data->pChildren[26]->word = keyWord then

return TRUE

else

return NULL

end

Metin Ağaçlarının Kullanım Alanları

* Metin ağaçlarındaki arama en kötü ihtimalle O(n) kadar bir karmaşıklığa sahip olduğundan ve arama hızı ağacın büyüklüğüyle değişmediğinden dolayı sözlük yapımında kullanılırlar.
* Arama motorlarındaki indexler sıkıştırılmış merin ağacında tutulurlar. Metnin her bir yaprağı aranan bir kelimeyle ilgili veriyi tutar.
* Routerlar paket yönlendirme algoritmaları içerisinde metin ağaçlarını kullanırlar. Burada ağ alt geçini önek bir ağaçta tutar ve buradaki yollara göre yönlendirme yapar.

4 Tam puan bekliyoruz. Ekstra seçim olarak 2 tane seçim yaptık.

Floyd-Warshall ve Breadth First Tree yi’seçtik.