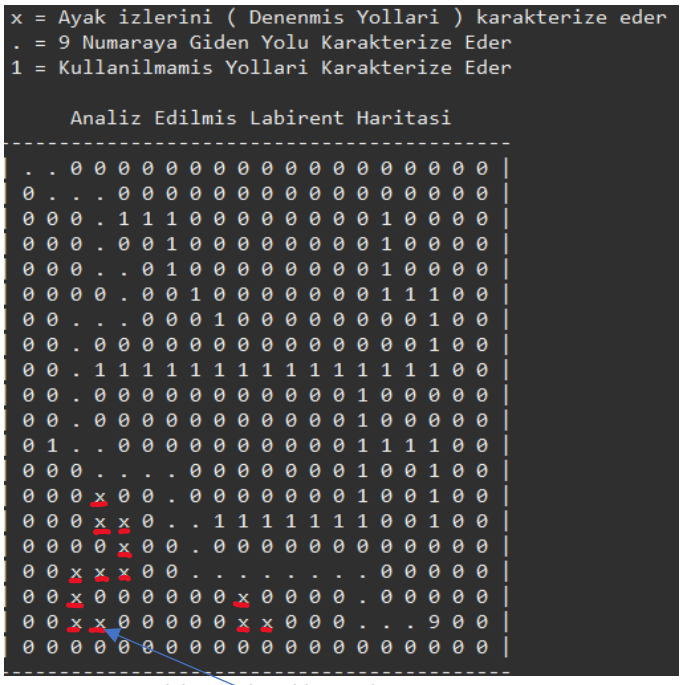
Labirent Algoritması Detaylı Dokümantasyon ve Fonksiyonlarla İlgili Bilgiler

Bazı Önemli Notlar:

Algoritma ‘da değişken isimleri programın sağlıklı çalışması amacıyla İngilizce olarak belirlenmiş, Türkçe karakterlere yer verilmemiştir, değişken isimleri İngilizce olarak mantıksal bir anlama sahiptir, rastgele olarak seçilmemiştir.



Şekil 1 Analiz Edilmiş Labirent Haritası

labirent.txt dosyasındaki 1 ve 0’lar “Integer” sınıfından bir diziye değil de “char” sınıfından bir diziye aktarılmıştır, bunun nedeni algoritmanın 9 numaraya giden nihai yolu bulana kadar, denediği farklı (9 numaraya ulaşmayan) yolları da ekranda ‘x’ işareti ile daha okunaklı gösterme çabasıdır, diğer yandan analiz edilmiş labirent haritasında daha mantıksal işaretlerle, bilgileri karakterize etme fırsatı yakalanmıştır.
(Dilediğiniz taktirde, aynı algoritmayı “Integer” dizisiyle yapıp tarafınıza tekrar iletebilirim, teknik fizibilite bakımından “char” dizisi ve “Integer” dizisi kullanmak arasında kayda değer bir fark yoktur, “char” dizisini kullanma amacım altı çizili cümlelerden anlaşıldığı üzere estetik kaygı ve anlaşılabilir bir çıktı üretme çabasıdır.)

Algoritmayı Detaylı Olarak İnceleyelim:

```
static char[][] matrix = new char[20][20];
static Scanner scan;
static char [][] roadOutLine = new char[20][20];
```

"labirent.txt" dosyasındaki 20 satırdan oluşan ve her satırında 20 karakter olan metni tutması amacıyla "matrix" (matris) adında, 2 boyutlu bir dizi oluşturulmuş ve ardından
"labirent.txt" dosyasını taratmak için "Scanner" sınıfından bir nesne (scan) oluşturulmuştur.
Bu dizilerin sınıf altında ve “static” olarak tanımlanasının sebebi bütün fonksiyonlardan rahatça bu değişkenlere erişilmesine olanak sağlamaktır.

```
File file = new File("C:\\development\\eclipse-java-2022-09-R-win32-
x86_64\\eclipse\\workspace\\labyrinth\\src\\metin.txt");
		scan = new Scanner(file);
		String lineHolder = "";

		for (int i = 0; i < 20; i++) {

			scan.hasNextLine();
			lineHolder = scan.next();

			for (int j = 0; j < 20; j++) {

				matrix[i][j] = lineHolder.charAt(j);
			}
		}
```

"File" sınıfından bir nesne (Constructor'ı ile) oluşturulmuş, parametre olarak "labirent.txt" dosyasının yolu girilmiştir, böylelikle "file" nesnesine "labirent.txt" dosyası atanmıştır.
"Scanner" sınıfından oluşturduğumuz "scan" nesnesi, "file" nesnesine atadığımız "labirent.txt" dosyasını satır satır tarar. "lineHolder" (Satır tutucu) bir bayrak (temporary variable) değişkenidir, "labirent.txt" dosyasındaki her satırı geçici olarak tutar, tuttuğu her satırdaki karakterlerin tamamı "matrix" dizisine teker teker atanır, ardından bir sonraki satır için aynı işlem yapılır, işlem 20 kez tekrarlandığında "labirent.txt" dosyasındaki 400 karakterin hepsi düzgün bir şekilde "matrix" 2 boyutlu dizisine aktarılmış olur.

```
for (int i = 0; i < 20; i++) {
			for (int j = 0; j < 20; j++) {

				roadOutLine[i][j] = ' ';

			}

		}
```

"roadOutLine" (Yol Taslağı) dizisinin her değeri boş (" ") olarak atanmıştır, sonrasında analiz edilmiş haritamızdan “9” numaraya giden yol indisleri "1" değeriyle override edilip yol taslağımız konsola yazdırılacaktır, analiz edilmiş harita çıktımızda 9 numaraya giden yol açıkça (noktalar ile) belirtilecek olmasına rağmen, böyle bir çıktı ödev gereksinimlerindeki çıktıyı elde etmek için oluşturulmuştur.

```
if (pathValidator(matrix, 0, 0)) {

			System.out.printf("%30s\n\n", "Labirent Cozuldu ! ");
} else {

			System.out.printf("%30s\n\n", "Labirent Cozumsuz ! ");
			System.exit(0);
```

Temel olarak algoritmamız yol bulmaya bu karar yapısından itibaren başar, "pathValidator" fonksiyonu

“true” değerini döndürürse labirentin çözüldüğü, “false” değerini döndürürse 9 numaraya giden herhangi bir yol olmadığı anlamına gelir "pathValidator" fonksiyonun nasıl çalıştığı ayrı olarak ilerleyen kısımlarda ele alınmıştır.

```
private static void appearanceShaper(char[][] matrix) {

		System.out.println("----------------------------------------");
		for (int x = 0; x < 20; x++) {
			System.out.print("| ");
			for (int y = 0; y < 20; y++) {
				System.out.print(matrix[x][y] + " ");
			}

			System.out.println("|");
		}
		System.out.println("----------------------------------------");

	}
```

"labirent.txt" dosyasından “matri”x dizisine aktarılan elemanların konsoldan daha okunaklı görünmesini sağlamak amacıyla oluşturulmuş appearanceShaper() (Görünüm Şekillendirici) fonksiyonu, parametre olarak okunaklı yapılacak diziyi alır ardından her dizi elamanın arasına bir boşluk bırakır; sağ ve sola "|" karakteri, aşağı ve yukarı "-" karakteri ile çevreler.

```
public static boolean doorValidator(char[][] matrix, int row, int column) {

		if (row >= 0 && row < 20 && column >= 0 && column < 20 &&
matrix[row][column] == '9') {
			roadOutLine[row][column] = '9';
			return true;

		}
		if (row >= 0 && row < 20 && column >= 0 && column < 20 &&
matrix[row][column] == '1')
			return true;

		return false;

	}
```

doorValidator (Kapı Geçerleyicisi) parametre olarak “matrix”, “row” ve “column” değişkenleri alıp bir boolean değeri geri döndürür, ilk karar yapısı row (satır) ve column(sütun) değerlerinin 0-19 (dahil) arasında olduğunda **ve** matrix[row][column] karakterinin "9" a eşit olması durumunda true değerini döndürür çünkü 9 değeri çıkış kapısıdır ve aynı zamanda potansiyel bir geçerli kapıdır. Bu karar yapısındaki durumlardan (condition) herhangi biri sağlanmadığı taktirde ikinci karar yapısı çalışır. İkinci karar yapısı da birinci karar yapısı gibi satır sütün değerlerinin uygunluğunu kontrol eder ve matrix[row][column] karakterinin "1" e eşit olması durumunda true değerini döndürür, çünkü "1" değeri geçerli bir kapı değeridir. Bu iki karar yapısı sağlanmadığı taktirde ilgili satır ve sütun değerleri geçerli bir kapıya refere etmiyordur bu sebeple “false” değeri geri döndürülür.

pathValidator() Fonksiyonu (EN ÖNEMLİ FONKSİYON)

Eğer ki bu fonksiyonun detaylı raporunu okuyorsanız, şu an tam olarak en can alıcı bölüme geldiniz demektir, algoritmamızın en temel fonksiyonu **pathValidator()** fonksiyonudur**,** fonksiyonumuzun kaynak kodlarını ayrı bir sekmede açıp birlikte satır satır inceleyelim ;

Parametre olarak “matrix” dizisini “row” ve “column” değerlerini alan fonksiyonumuz, ilk karar yapısında kendisine gelen ilgili dizi için, satır ve sütun değerlerinin geçerli bir kapıya refere edip etmediğini “doorValidator” fonksiyonunu kullanarak doğrular. Karar yapısı içerisinde ilgili parametreler ile çağrılan “doorValidator” fonksiyonu geçerli kapı olması durumunda “true”, olmaması durumunda ise “false” değerlerini geri döndürür.
Algoritmamızda bu fonksiyon ilk olarak matrix dizimiz, 0.satır ve 0.sütun için çağrılmıştır, **yani fonksiyon çağrılır çağrılmaz başlangıçta (0.satır, 0.sütun değerlerinde) kapı olup olmadığının sonucunu kesin olarak belirler**, başlangıçta kapı olmadığı taktirde direkt olarak “false” değerini geri döndürür ve labirent çözümsüz yazısı ekrana bastırılır. İlk kapı değerimizin geçerli bir kapıya refere ettiğini varsayarak anlatmaya devam edelim, eğerki 0,0 indisleri geçerli bir kapıya refere ediyorsa, bu karar yapımız içerisindeki kapsam bölgesi çalıştırılır ardından ikinci karar yapımız bu kapının 9 numara olup olmadığı denetlenir eğer ki 9 numara ise direkt olarak “true” değeri döndürülür ve labirent çözülür, ilgili indis değerleri geçerli bir kapıya refere ediyor (“1” değeri) fakat 9 numaraya eşit değilse, haritamızda ilgili indise “x” sembolü ile ayak izi bırakılır, sonrasında geçerli indis değerlerinin bir altında geçerli kapı olup olmadığı denetlenir, eğerki geçerli bir kapı değeri ilgili indis değerinin altında varsa aynı işlemler özyineli olarak tekrar gerçekleşir,(fonksiyon kendi kendisini çağırır)

```
boolean recursiveReturnValue = pathValidator(matrix, row + 1, column);
```

→Alt kapı kontrol

Eğerki altta geçerli bir kapı değeri yok ise aşağıdaki karar yapısı kontrol edilecektir.

```
if (recursiveReturnValue != true) {recursiveReturnValue = pathValidator(matrix, row, column +1);}
```

→Sağ Kapı Kontrol

Karar yapısı ile ilgili indisin sağında geçerli kapı değerinin olup olmadığı kontrol edilecektir eğerki geçerli bir kapı var ise sağdaki kapının indis değerlerini parametre alan yeni bir iterasyon başlayacaktır. Burada dikkat edilmesi gereken nokta; bir alt indiste geçerli bir kapı **yok ise**, sağ indiste kapı araştırmasının yapılacağıdır. Sağ indiste geçerli kapı yok ise aşağıdaki karar yapısı kontrol edilecektir.

```
if (recursiveReturnValue != true) {recursiveReturnValue = pathValidator(matrix, row - 1, column);}
```

→Yukarıda Kapı Kontrol

Bu karar yapısı kapsamındaki ifade ise ilgili indis değerlerinin yukarısında geçerli bir kapı olup olmadığını özyineli olarak kontrol eder eğerki geçerli bir kapı değeri alt indiste mevcut ise ilgili indis değerlerini parametre alan yeni bir alt iterasyon (subiteration) başlayacaktır, bu karar yapısı kapsamındaki ifade ilgili indis değerlerinin **altında ve sağında geçerli bir kapı olmadığı** durumda icra edilecektir. Eğerki yukarıda da geçerli bir kapı yok ise aşağıdaki karar yapısı kontrol edilecektir.

```
if (recursiveReturnValue != true) {recursiveReturnValue = pathValidator(matrix, row, column - 1);}
```

→Sol Kapı Kontrol

Bu karar yapısında da son olarak solda kapı olup olmadığı kontrol edilir, eğerki varsa yeni bir alt iterasyon ilgili indis değerleri için başlatılır, fakat bu karar yapısından sonra da “**recursiveReturnValue”** değişkeninin değeri hala “false” ise demek ki; aşağıda, sağda, yukarıda ve solda geçerli bir kapı yoktur yani labirent çözümsüzdür, dolayısıyla **pathValidator()** fonksiyonu “false” değerini döndürür.

10’larca kez kendi kendini çağıran özyineli **pathValidator()** fonksiyonu herhangi bir alt iterasyon sonucu 9 numaraya ulaştığında ise ilgili alt iterasyon için dönen true sonucu algoritmanın başında ilk kez çağrılan fonksiyona kadar geri döndürülür, dönerken de 9 numaraya giden indisler analiz edilmiş haritada “.” , yol taslağında ise “1” ile işaretlenir.

```
if (recursiveReturnValue) {
```

```
matrix[row][column] = '.';
roadOutLine[row][column] = '1';
}
```