Bir Mantık Programlama Dili

PROLOG

(Bölüm V)

Selçuk TARAL*

Daha önceki bölümlerde Prolog programlama dilinin temel yapısını kısa örnek programlarla anlatmaya çalıştık. Programların okunuşlarını gördünüz. Pür mantık programlamasında tümcelerin ve tümce içindeki yüklemlerin yazılış sıranın önemini daha önce belirtmiştik. Bu anlamda Prolog bir *pür mantık programlama dili* değildir. Prolog'u anlatırken, **Program = Mantık + Kontrol** formülünün şimdiye değin 'Mantık' tarafına ağırlık verdik. Gelecek bölümde 'Kontrol' tarafına da bakacağız. Ama bu bölümde tanınmış bir bulmacanın Prolog'la çözümünü vermek istiyoruz.

Bilgisayar biliminde birçok problem, sonlu veya sonsuz sayıda *durum* ve bunların oluşturduğu *uzay* (state-space) üzerinde tanımlanmış çiftli (binary) bir *geçiş ilişkisi* (transition relation) şeklinde formüle edilebilir. Problem, bu uzayda verilmiş bir başlangıç noktasından hedef noktalarından birine geçiş kuralları aracılığıyla varılıp varılamıyacağının belirlenmesidir. Varmak istediğimiz noktayı *sonuç durumu* (final state), yola çıktığımız noktayı ise *başlangıç durumu* (initial state) olarak adlandıracağız. Bu tür problemlere tipik bir örnek olarak, hemen herkesin bildiği *çiftçi, kurt, keçi ve lahana bulmacasını* verebiliriz; *Kuzuyu kurda, lahanayı keçiye yedirmeden* iki kişilik bir kayıkla hepsini bir derenin karşı yakasına nasıl geçirirsiniz? Başlangıç durumunda, dördü de derenin batı yakasında, bulmacanın çözümü olan sonuç durumunda ise hepsi, sağ salim, derenin doğu yakasında bulunurlar. Bulmacada bazı durumlar *emniyetli*, diğerleri ise *emniyetsizdir*. Örneğin, kurt ile lahananın batı yakasında, çiftçi ile keçinin doğu yakasında bulunması emniyetli, dolayısıyla kabul edilebilir bir durumu gösterir. Buna karşılık, kurt ile keçinin bir yakada, çitfçi ile lahananın karşı yakada bulunduğu durum ise emniyetsizdir. Emniyetsiz durumlar problemin *kısıtlamalarını* oluştururlar.

Bu tür problemleri bir ağ (graph) yapısında gösterebiliriz: Ağın *düğüm* noktaları (node) problemin durumlarını gösterir ve aralarında kurallara uygun bir geçiş (transition) olan noktalar birer *kenar* (edge) ile birleştirilir. Kural gereği, çiftçi karşı tarafa bir kerede kurt, keçi ve lahanadan ancak birini geçirebilir. Problemin çözümü, başlangıç durumundan hedeflediğimiz duruma giden bir yolun (path) bulunması anlamına gelir. Bu yolun üzerindeki tüm noktaların (ara durumlar) problemin kısıtlamalarını sağlaması şarttır. Bu nedenle, bir durumdan diğerine geçiş ancak problemin kısıtlamaları izin verirse yapılabilir. Örneğin, bulmacadaki başlangıç durumu ile, keçiyle lahananın batı, çitfçiyle kurdun ise doğu yakasında bulunduğu durum arasında bir kenar olmasına karşın, keçi lahanayı yiyeceğinden, çözüme bu nokta üzerinden gitmeye olanak yoktur.

Şimdi bulmacanın Prolog ile çözümüne geçelim. Problemin yüklemlerini tanımlayan kuralları yazmadan önce, bu kuralların işleyeceği veri yapılarına karar vermemiz gerekir. Kuralların yazılışı *veri yapılarına* göre değişir. Bulmacayı kapalı bir sistem olarak algılarsak, sistemin herhangi bir andaki durumunu çiftçi, kurt, keçi ve lahanadan her birinin pozisyonu belirler. Kayığın pozisyonu ise, her zaman çiftçiyle aynı tarafta olduğu için bir önem taşımaz. Bu pozisyonları Prolog'ta iki ayrı veri yapısı aracılığıyla gösterebiliriz:

1. Dört değişkenli bir karmaşık terim: **yer (F,W,G,C). F**(armer) değişkeni çiftçinin, **W**(olf) kurdun, **G**(oat) keçinin ve **C**(abbage) ise lahanın yerlerini gösterir. (Kurt ile kuzu sözcükleri aynı harfle başladıklarından burada ingilizce karşılıklarını kullandık). Örneğin başlangıç durumunu bu yapıda **yer(b,b,b,b)** ile gösterebiliriz. Batıyı 'b', doğuyu ise 'd' sabiti ile gösterdik. Açıklık amacıyla her değişkenin önüne neye ait olduğunu gösteren bir isim de koyabilirdik: **yer(farmer(F), wolf(W), goat(G), cabbage(C)).** Biz daha kolay olduğu için birinci yazılış şeklini kullanacağız.

2. Derenin iki tarafındakileri içeren listelerin listesi: Örneğin, **[[çiftçi, kurt, keçi, lahana], [ ]]** listesi başlangıç durumunu, **[[kurt, lahana], [çiftçi, keçi]]** bir ara durumu ve **[[ ], [çiftçi, kurt, keçi, lahana]]** ise sonuç durumunu gösterir.

Şimdi yapılacak iş bu veri yapılarından birini kullanarak, mevcut durumdan yeni bir duruma geçişi

* *TÜBİTAK, Bilgi İşlem Daire Başkanı.*

belirleyen bir yüklem tanımlamaktır. Bu yükleme **geçiş** adını verelim ve içinde bulunulan durumu **Şimdiki**, geçiş sonucu varılan durumu ise, **Sonraki** değişkeni ile gösterelim. Bu yüklemi tanımlayan birinci kural, çiftçinin kendi başına bir taraftan ötekine geçebileceğini ifade eder. Tabiî, yeni durumun emniyetli olması koşuluyla!

```
geçiş(yer(F,W,G,C),yer(F2,W,G,C)) :—
        karşıt(F,F2), emin(yer(F2,W,G,C)).
```

Kuralın sağındaki **karşıt** ve **emin** ilişkilerini az sonra tanımlayacağız. Diğer geçiş kuralları ise **Şekil 6**'da verilmiştir. Bunlar sırasıyla çiftçinin kurdu, keçiyi ve lahanayı karşı tarafa geçirmesini verir. Örneğin, ilk kuralda çiftçi ile kurdun pozisyonları aynı değişken ismiyle verilerek, derenin bir yakasından karşı yakasına geçmeleri gösterilmiştir. Hatırlarsanız, taşıma işinin kuralın başında yapılmasını daha önce *ekle* yükleminin tanımında görmüştük.

```
geçiş(yer(F,F,G,C),yer(F2,F2,G,C)) <-
        karşıt(F,F2), emin(yer(F2,F2,G,C)).

geçiş(yer(F,W,F,C),yer(F2,W,F2,C)) <-
        karşıt(F,F2), emin(yer(F2,W,F2,C)).

geçiş(yer(F,W,G,F),yer(F2,W,G,F2)) <-
        karşıt(F,F2), emin(yer(F2,W,G,F2)).
```

Şekil 6:

Kurallarda *geçiş* yüklemini tanımlamak için iki yeni yüklem kullandık: **karşıt** ve **emin**. Bu yüklemlerin tanımlarını Şekil 7'de görebilirsiniz. Bir pozisyonun *emin* olabilmesi için, ya çiftçiyle keçinin aynı tarafta olması veya keçinin bir tarafta diğerlerinin karşı tarafta olması gerekir.

```
karşıt(d,b).
karşıt(b,d).

emin(yer(F,W,F,C)).
emin(yer(F,F,G,F)) :- karşıt(F,G).
```

Şekil 7:

Bu kurallar *çiftçi problemini* tamamen tanımlar. Artık geriye kalan arama işlemini gerçekleştirecek genel bir yordamın (procedure) yazılmasıdır. Bu yordam problemden bağımsız kurallarla tanımlanır. Diğer bir deyişle, aynı arama yordamını başka bir *durum-uzayı problemi* için de kullanabilirsiniz. En basit arama yordamını, Prolog'un içindeki geriye doğru iz sürme (backtracking) mekanizmasını kullanarak yazabiliriz. Şimdi bulmacayı çözecek yüklemi tanımlayalım. Bu yükleme, bu tarz arama için kullanılan ingilizce **'depth_first'** deyiminin Türkçe karşılığı 'önce derinlemesine'yi kısaltarak **ö_derin** adını verdik:

ö_derin(Hedef,Geçmiş,Çözüm).

Özyinelemeli bu yüklemin birinci argümanı (Hedef) hedeflenen durumu gösterir. İkinci argüman (Geçmiş) ise daha önce üzerinden geçilmiş noktaların (durumların) listesidir. Böyle bir listenin tutulmasının önemini tahmin etmişsinizdir. Son argüman (Çözüm) ise, başlangıç durumundan sonuca giden kabul edilebilir durumların bir listesidir: Yani problemin çözümü.

```
ö_derin(Hedef,[Hedef|_],[Hedef]).
ö_derin(Hedef,[Şimdiki|Öncekiler],[Şimdiki|Gelecek]) :-
        geçiş(Şimdiki,Sonraki),
        not üye(Sonraki,Öncekiler),
        ö_derin(Hedef,[Sonraki,Şimdiki|Öncekiler],Gelecek).
```

Şekil 8:

Arama işlemini gerçekleştiren **ö_derin** yüklemi-nin tanımını **Şekil 8**'de verdik. Birinci tümce şu anlama gelir: Üzerinde bulunduğumuz durum Şimdiki ile Hedef aynı iseler, problem çözülmüş demektir. Bu durumda, **Çözüm** değişkeni **[Hedef]** listesine bağlanır. Bu son işlem size garip gelebilir; Çözüm değişkeninin, hesaplamanın sonunda, başlangıçtan sonuca kadar tüm durumları içeren bir liste olacağını söylemiştik. Çelişki olmadığı gerçeğin kuralla birlikte yorumlanması sonucu ortaya çıkar.

Kuralın yaptığı ise, önce *geçiş* yüklemiyle yeni bir duruma (Sonraki) geçip sonra bu noktadan daha önce geçilmediğine emin olmak ve son olarak (yeni durumu geçmişi tutan listenin başına atıp yeni durumlar bulmak için) kendini özyinelemeli olarak çağırmaktır. *Daha önce üzerinden geçilen bir duruma yeniden gelinmesi halinde, kısır döngüye girmemiz tehlikesi vardır.* Bu nedenle, 'değil' anlamındaki 'not' kelimesi kullanılarak, şimdiki durumun daha önce geçilen durumların listesinin bir elemanı olmadığı kontrol edilir. Bir elemanın verilen bir listenin içinde *olmadığını* göstermek için başına bir 'not' koymanız yeterlidir.

Arama işleminin başlatılması, programa aşağıdaki hedefin verilmesiyle olur:

ö_derin(yer(d,d,d,d),[yer(b,b,b,b)],Çözüm).

Hesaplamanın sonunda **Çözüm** değişkeni aşağıda verilen listeye bağlanır:

[yer(b,b,b,b),yer(d,b,d,b),yer(b,b,d,b), yer(d,d,d,b),yer(b,d,b,b),yer(d,d,b,d),yer(b,d,b,d), yer(d,d,d,d)]

Çözümün doğruluğunun kontrolünü okuyucuya bırakıyoruz. Geçen bölümün sonunda verdiğimiz **son** yüklemi verilen elemanın listenin son elemanı olup olmadığını kontrol eder. Birinci argüman yerine bir değişken kullanılırsa, listenin son elemanını bulup bu değişkene bağlar. Örneğin, **?-son(X,[a,b,c,d])** sorusunun yanıtı **X = d** olur.

(Devam edecek.)