Durum Temelli Modeller El Kitabı VIP

Afshine Amidi ve Shervine Amidi

September 14, 2019

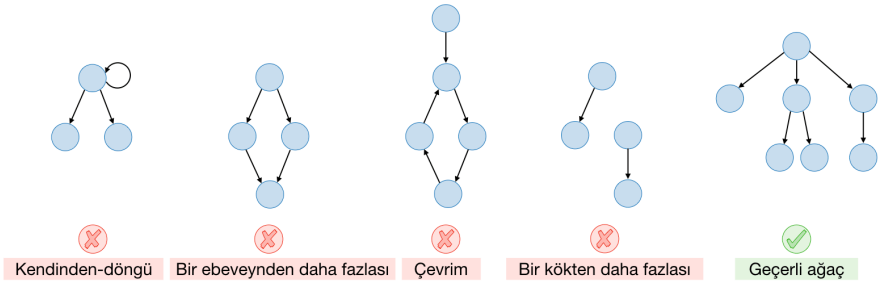
Cemal Gurpinar ve Başak Buluz tarafından çevrilmiştir

Arama optimizasyonu

Bu bölümde, s durumunda a eylemini gerçekleştirdiğimizde, Succ(s,a) durumuna varacağımızı varsayıyoruz. Burada amaç, başlangıç durumundan başlayıp bitiş durumuna götüren bir eylem dizisi (a_1,a_2,a_3,a_4,...) belirlenmesidir. Bu tür bir problemi çözmek için, amacımız durum-temelli modelleri kullanarak asgari maliyet yolunu bulmak olacaktır.

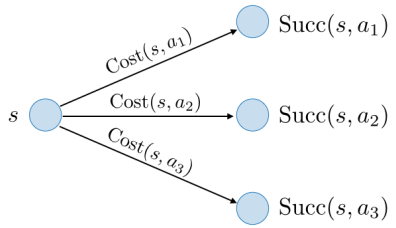
Ağaç arama

Bu durum-temelli algoritmalar, olası bütün durum ve eylemleri araştırırlar. Oldukça bellek verimli ve büyük durum uzayları için uygundurlar ancak çalışma zamanı en kötü durumlarda üstel olabilir.



❒ **Arama problemi** – Bir arama problemi aşağıdaki şekilde tanımlanmaktadır:

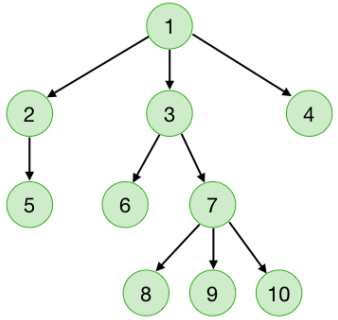
- bir başlangıç durumu s_{start}
- s durumunda gerçekleşebilecek olası eylemler Actions(s)
- s durumunda gerçekleşen a eyleminin eylem maliyeti Cost(s,a)
- a eyleminden sonraki varılacak durum Succ(s,a)
- son duruma ulaşılıp ulaşılamadığı IsEnd(s)



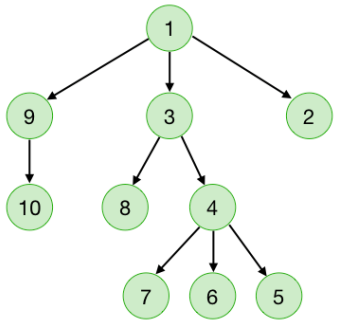
Amaç, maliyeti en aza indiren bir yol bulmaktır.

❒ **Geri izleme araması** – Geri izleme araması (backtracking search), asgari maliyet yolunu bulmak için tüm olasılıkları deneyen saf (naive) bir özyinelemeli algoritmadır. Burada, eylem maliyetleri pozitif ya da negatif olabilir.

❒ **Genişlik öncelikli arama (BFS)** – Genişlik öncelikli arama (BFS, breadth-first search), seviye seviye arama yapan bir çizge arama algoritmasıdır. Gelecekte her adımda ziyaret edilecek düğümleri tutan bir kuyruk yardımıyla yinelemeli olarak gerçekleyebiliriz. Bu algoritma için, eylem maliyetlerinin belirli bir sabite $c \geqslant 0$ eşit olduğunu kabul edebiliriz.



❒ **Derinlik öncelikli arama (DFS)** – Derinlik öncelikli arama (DFS, depth-first search), her bir yolu olabildiğince derin bir şekilde takip ederek çizgeyi dolaşan bir arama algoritmasıdır. Bu algoritmayı, ziyaret edilecek gelecek düğümleri her adımda bir yığın yardımıyla saklayarak, yinelemeli (recursively) ya da tekrarlı (iteratively) olarak uygulayabiliriz. Bu algoritma için eylem maliyetlerinin 0 olduğu varsayılmaktadır.



❒ **Tekrarlı derinleşme** – Tekrarlı derinleşme (iterative deepening) hilesi, derinlik-ilk arama algoritmasının değiştirilmiş bir halidir, böylece belirli bir derinliğe ulaştıktan sonra durur, bu da tüm işlem maliyetleri eşit olduğunda en iyiliği (optimal) garanti eder. Burada, işlem maliyetlerinin $c \geqslant 0$ gibi sabit bir değere eşit olduğunu varsayıyoruz.

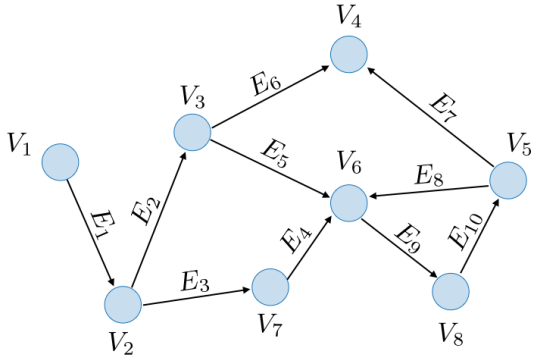
❒ **Ağaç arama algoritmaları özeti** – b durum başına eylem sayısını, d çözüm derinliğini ve D en yüksek (maksimum) derinliği ifade ederse, o zaman:

Algoritma	Eylem maliyetleri	Arama uzayı	Zaman
Geri izleme araması	herhangi bir şey	$\mathcal{O}(D)$	$\mathcal{O}(b^D)$
Genişlik öncelikli arama	$c \geqslant 0$	$\mathcal{O}(b^d)$	$\mathcal{O}(b^d)$
Derinlik öncelikli arama	0	$\mathcal{O}(D)$	$\mathcal{O}(b^D)$
DFS-tekrarlı derinleşme	$c \geqslant 0$	$\mathcal{O}(d)$	$\mathcal{O}(b^d)$

Çizge arama

Bu durum-temelli algoritmalar kategorisi, üssel tasarruf sağlayan en iyi (optimal) yolları oluşturmayı amaçlar. Bu bölümde, dinamik programlama ve tek tip maliyet araştırması üzerinde duracağız.

❒ **Çizge** – Bir çizge, V köşeler (düğüm olarak da adlandırılır) kümesi ile E kenarlar (bağlantı olarak da adlandırılır) kümesinden oluşur.

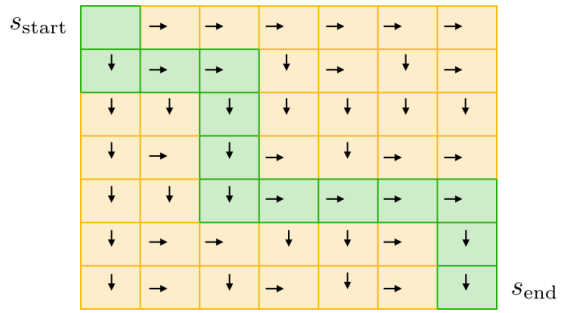


Not: çevrim olmadığında, bir çizgenin asiklik (çevrimsiz) olduğu söylenir.

❒ **Durum** – Bir durum gelecekteki eylemleri en iyi (optimal) şekilde seçmek için, yeterli tüm geçmiş eylemlerin özetidir.

❒ **Dinamik programlama** – Dinamik programlama (DP, dynamic programming), amacı s durumundan bitiş durumu olan s_{end}'e kadar asgari maliyet yolunu bulmak olan hatırlamalı (memoization) (başka bir deyişle kısmi sonuçlar kaydedilir) bir geri izleme (backtracking) arama algoritmasıdır. Geleneksel çizge arama algoritmalarına kıyasla üstel olarak tasarruf sağlayabilir ve yalnızca asiklik (çevrimsiz) çizgeler ile çalışma özelliğine sahiptir. Herhangi bir durum için gelecekteki maliyet aşağıdaki gibi hesaplanır:

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{eğer IsEnd}(s) \\ \min\limits_{a \in \text{Actions}(s)} \big[\text{Cost}(s,a) + \text{FutureCost}(\text{Succ}(s,a))\big] & \text{aksi taktirde} \end{cases}$$

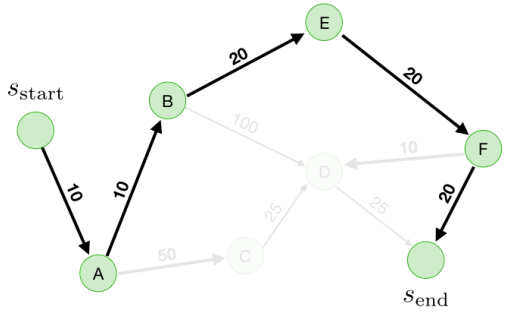


Not: yukarıdaki şekil, aşağıdan yukarıya bir yaklaşımı sergilerken, formül ise yukarıdan aşağıya bir önsezi ile problem çözümü sağlar.

❒ **Durum türleri** – Tek tip maliyet araştırması bağlamındaki durumlara ilişkin terminoloji aşağıdaki tabloda sunulmaktadır:

Durum	Açıklama
Keşfedilmiş $\mathcal{E}$	En iyi (optimal) yolun daha önce bulunduğu durumlar
Sırada $\mathcal{F}$	Görülen ancak hala en ucuza nasıl gidileceği hesaplanmaya çalışılan durumlar
Keşfedilmemiş $\mathcal{U}$	Daha önce görülmeyen durumlar

❒ **Tek tip maliyet araması** – Tek tip maliyet araması (UCS, uniform cost search) bir başlangıç durumu olan s_{start}, ile bir bitiş durumu olan s_{end} arasındaki en kısa yolu bulmayı amaçlayan bir arama algoritmasıdır. Bu algoritma s durumlarını artan geçmiş maliyetleri olan PastCost(s)'a göre araştırır ve eylem maliyetlerinin negatif olmayacağı kuralına dayanır.



Not 1: UCS algoritması mantıksal olarak Dijkstra algoritması ile aynıdır.
Not 2: algoritma, negatif eylem maliyetleriyle ilgili bir problem için çalışmaz ve negatif olmayan bir hale getirmek için pozitif bir sabit eklemek problemi çözmez, çünkü problem farklı bir problem haline gelmiş olur.

❒ **Doğruluk teoremi** – s durumu sıradaki (frontier) $\mathcal{F}$'den çıkarılır ve daha önceden keşfedilmiş olan $\mathcal{E}$ kümesine taşınırsa, önceliği başlangıç durumu olan s_{start}'dan, s durumuna kadar asgari maliyet yolu olan PastCost(s)'e eşittir.

❒ **Çizge arama algoritmaları özeti** – N toplam durumların sayısı, n-bitiş durumu s_{end}'ndan önce keşfedilen durum sayısı ise:

Algoritma	Asiklik	Maliyetler	Zaman/arama uzayı
Dinamik programlama	evet	herhangi bir şey	$\mathcal{O}(N)$
Tek tip maliyet araması	değil	$c \geqslant 0$	$\mathcal{O}(n\log(n))$

Not: karmaşıklık geri sayımı, her durum için olası eylemlerin sayısını sabit olarak kabul eder.

Öğrenme maliyetleri

Diyelim ki, Cost(s,a) değerleri verilmedi ve biz bu değerleri maliyet yolu eylem dizisini, $(a_1, a_2, ..., a_k)$, en aza indiren bir eğitim kümesinden tahmin etmek istiyoruz.

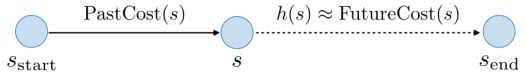
❒ **Yapılandırılmış algılayıcı** – Yapılandırılmış algılayıcı, her bir durum-eylem çiftinin maliyetini tekrarlı (iteratively) olarak öğrenmeyi amaçlayan bir algoritmadır. Her bir adımda, algılayıcı:

- eğitim verilerinden elde edilen gerçek asgari y yolunun her bir durum-eylem çiftinin tahmini (estimated) maliyetini azaltır,
- öğrenilen ağırlıklardan elde edilen şimdiki tahmini(predicted) y' yolununun durum-eylem çiftlerinin tahmini maliyetini artırır.

Not: algoritmanın birkaç sürümü vardır, bunlardan biri problemi sadece her bir a eyleminin maliyetini öğrenmeye indirger, bir diğeri ise öğrenilebilir ağırlık öznitelik vektörünü, Cost(s,a)'nın parametresi haline getirir.

A* arama

❒ **Sezgisel işlev** – Sezgisel (heuristic), s durumu üzerinde işlem yapan bir h fonksiyonudur, burada her bir $h(s)$, s ile s_{end} arasındaki yol maliyeti olan FutureCost(s)'yi tahmin etmeyi amaçlar.



❒ **Algoritma** – A^* s durumu ile s_{end} bitiş durumu arasındaki en kısa yolu bulmayı amaçlayan bir arama algoritmasıdır. Bahse konu algoritma PastCost(s) + $h(s)$'yi artan sıra ile araştırır. Aşağıda verilenler ışığında kenar maliyetlerini de içeren tek tip maliyet aramasına eşittir:

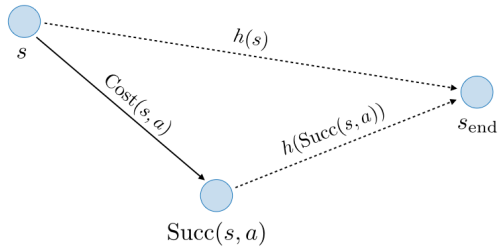
$$\boxed{\text{Cost}'(s,a) = \text{Cost}(s,a) + h(\text{Succ}(s,a)) - h(s)}$$

Not: bu algoritma, son duruma yakın olduğu tahmin edilen durumları araştıran tek tip maliyet aramasının taraflı bir sürümü olarak görülebilir.

❒ **Tutarlılık** – Bir sezgisel h, aşağıdaki iki özelliği sağlaması durumunda tutarlıdır denilebilir:

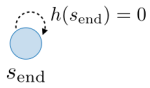
- Bütün s durumları ve a eylemleri için,

$$\boxed{h(s) \leqslant \text{Cost}(s,a) + h(\text{Succ}(s,a))}$$



- Bitiş durumu aşağıdakileri doğrular:

$$\boxed{h(s_{\text{end}}) = 0}$$



❒ **Doğruluk** – Eğer h tutarlı ise o zaman A^* algoritması asgari maliyet yolunu döndürür.

❒ **Kabul edilebilirlik** – Bir sezgisel h kabul edilebilirdir eğer:

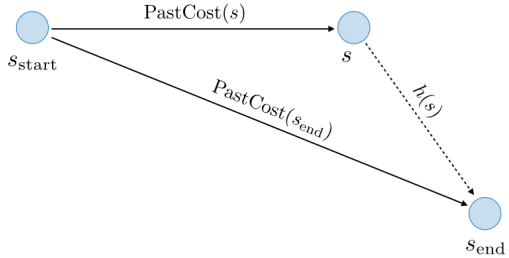
$$\boxed{h(s) \leqslant \text{FutureCost}(s)}$$

❒ **Teorem** – $h(s)$ sezgisel olsun ve:

$$\boxed{h(s) \text{ tutarlı} \implies h(s) \text{ kabul edilebilir}}$$

❒ **Verimlilik** – A^* algoritması aşağıdaki eşitliği sağlayan bütün s durumlarını araştırır:

$$\boxed{\text{PastCost}(s) \leqslant \text{PastCost}(s_{\text{end}}) - h(s)}$$



Not: $h(s)$'nin yüksek değerleri, bu eşitliğin araştırılacak olan s durum kümesini kısıtlayacak olması nedeniyle daha iyidir.

Rahatlama

Bu tutarlı sezgisel için bir altyapıdır. Buradaki fikir, kısıtlamaları kaldırarak kapalı şekilli (closed-form) düşük maliyetler bulmak ve bunları sezgisel olarak kullanmaktır.

❒ **Rahat arama problemi** – Cost maliyetli bir arama probleminin rahatlaması, Cost_{rel} maliyetli P_{rel} ile ifade edilir ve kimliği karşılar:

$$\boxed{\text{Cost}_{\text{rel}}(s,a) \leqslant \text{Cost}(s,a)}$$

❒ **Rahat sezgisel** – Bir P_{rel} rahat arama problemi verildiğinde, $h(s) = \text{FutureCost}_{\text{rel}}(s)$ rahat sezgisel (relaxed heuristic) eşitliğini $\text{Cost}_{\text{rel}}(s,a)$ maliyet çizgesindeki s durumu ile bir bitiş durumu arasındaki asgari maliyet yolu olarak tanımlarız.

❒ **Rahat sezgisel tutarlılığı** – P_{rel} bir rahat problem olarak verilmiş olsun. Teoreme göre:

$$\boxed{h(s) = \text{FutureCost}_{\text{rel}}(s) \implies h(s) \text{ tutarlı}}$$

❒ **Sezgisel seçiminde ödünleşim** – Sezgisel seçiminde iki yönü dengelemeliyiz:

- Hesaplamalı verimlilik: $h(s) = \text{FutureCost}_{\text{rel}}(s)$ eşitliği kolay hesaplanabilir olmalıdır. Kapalı bir şekil, daha kolay arama ve bağımsız alt problemler üretmesi gerekir.
- Yeterince iyi yaklaşım: sezgisel $h(s)$, $\text{FutureCost}(s)$ işlevine yakın olmalı ve bu nedenle çok fazla kısıtlamayı ortadan kaldırmamalıyız.

❒ **En yüksek sezgisel** – $h_1(s)$ ve $h_2(s)$ aşağıdaki özelliklere sahip iki adet sezgisel olsun:

$$\boxed{h_1(s),\, h_2(s) \text{ tutarlı} \implies h(s) = \max\{h_1(s),\, h_2(s)\} \text{ tutarlı}}$$

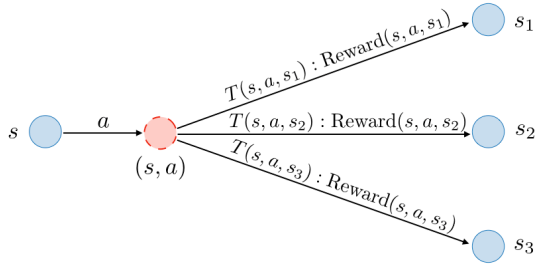
Markov karar süreçleri

Bu bölümde, s durumunda a eyleminin gerçekleştirilmesinin olasılıksal olarak birden fazla durum s'_1,s'_2,..., ile sonuçlanacağını kabul ediyoruz. Başlangıç durumu ile bitiş durumu arasındaki yolu bulmak için amacımız, rastgelelilik ve belirsizlik ile başa çıkabilmek için yardımcı olan Markov karar süreçlerini kullanarak en yüksek değer politikasını bulmak olacaktır.

Gösterimler

❒ **Tanım** – Markov karar sürecinin (MDP, Markov decision process) amacı ödülleri en yüksek seviyeye çıkarmaktır. Markov karar süreci aşağıdaki bileşenlerden oluşmaktadır:

- başlangıç durumu s_{start}
- s durumunda gerçekleştirilebilecek olası eylemler $\text{Actions}(s)$
- s durumunda a eyleminin gerçekleştirilmesi ile s' durumuna geçiş olasılıkları $T(s,a,s')$
- s durumunda a eyleminin gerçekleştirilmesi ile elde edilen ödüller $\text{Reward}(s,a,s')$
- bitiş durumuna ulaşılıp ulaşılamadığı $\text{IsEnd}(s)$
- indirim faktörü $0 \leqslant \gamma \leqslant 1$



❒ **Geçiş olasılıkları** – Geçiş olasılığı $T(s,a,s')$ s durumundayken gerçekleştirilen a eylemi neticesinde s' durumuna gitme olasılığını belirtir. Her bir $s' \mapsto T(s,a,s')$ aşağıda belirtildiği gibi bir olasılık dağılımıdır:

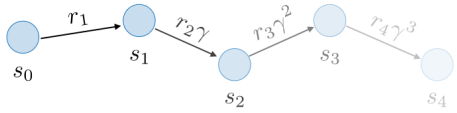
$$\forall s,a, \quad \boxed{\sum_{s' \in \text{ States}} T(s,a,s') = 1}$$

❒ **Politika** – Bir π politikası her s durumunu bir a eylemi ile ilişkilendiren bir işlevdir:

$$\boxed{\pi : s \mapsto a}$$

❒ **Fayda** – Bir $(s_0, ..., s_k)$ yolunun faydası, o yol üzerindeki ödüllerin indirimli toplamıdır. Diğer bir deyişle,

$$\boxed{u(s_0,...,s_k) = \sum_{i=1}^{k} r_i \gamma^{i-1}}$$



Yukarıdaki şekil $k = 4$ durumunun bir gösterimidir.

❒ **Q-değeri** – s durumunda gerçekleştirilen bir a eylemi için π politikasının Q-değeri (Q-value), $Q_\pi(s,a)$ olarak da gösterilir, a eylemini gerçekleştirip ve sonrasında π politikasını takiben s durumundan beklenen faydadır. Q-değeri aşağıdaki şekilde tanımlanmaktadır:

$$\boxed{Q_\pi(s,a) = \sum_{s' \in \text{ States}} T(s,a,s') \left[\text{Reward}(s,a,s') + \gamma V_\pi(s')\right]}$$

❒ **Bir politikanın değeri** – s durumundaki π politikasının değeri, $V_\pi(s)$ olarak da gösterilir, rastgele yollar üzerinde s durumundaki π politikasını izleyerek elde edilen beklenen faydadır. s durumundaki π politikasının değeri aşağıdaki gibi tanımlanır:

$$\boxed{V_\pi(s) = Q_\pi(s,\pi(s))}$$

Not: eğer s bitiş durumu ise $V_\pi(s)$ sıfıra eşittir.

Uygulamalar

❒ **Politika değerlendirme** – Bir π politikası verildiğinde, politika değerlendirmesini (policy evaluation), V_π, tahmin etmeyi amaçlayan bir tekrarlı (iterative) algoritmadır. Politika değerlendirme aşağıdaki gibi yapılmaktadır:

- İlklendirme: bütün s durumları için

$$\boxed{V_\pi^{(0)}(s) \longleftarrow 0}$$

- Tekrar: 1'den T_{PE}'ye kadar her t için, ile

$$\forall s, \quad \boxed{V_\pi^{(t)}(s) \longleftarrow Q_\pi^{(t-1)}(s,\pi(s))}$$

ile

$$\boxed{Q_\pi^{(t-1)}(s,\pi(s)) = \sum_{s' \in \text{ States}} T(s,\pi(s),s')\Big[\text{Reward}(s,\pi(s),s') + \gamma V_\pi^{(t-1)}(s')\Big]}$$

Not: S durum sayısını, A her bir durum için eylem sayısını, S' ardılların (successors) sayısını ve T yineleme sayısını gösterdiğinde, zaman karmaşıklığı $\mathcal{O}(T_{PE}SS')$ olur.

❒ **En iyi Q-değeri** – s durumunda a eylemi gerçekleştirildiğinde bu durumun en iyi Q-değeri, $Q_{\text{opt}}(s,a)$, herhangi bir politika başlangıcında elde edilen en yüksek Q-değeri olarak tanımlanmaktadır. En iyi Q-değeri aşağıdaki gibi hesaplanmaktadır:

$$\boxed{Q_{\text{opt}}(s,a) = \sum_{s' \in \text{ States}} T(s,a,s')\left[\text{Reward}(s,a,s') + \gamma V_{\text{opt}}(s')\right]}$$

❒ **En iyi değer** – s durumunun en iyi değeri olan $V_{\text{opt}}(s)$, herhangi bir politika ile elde edilen en yüksek değer olarak tanımlanmaktadır. En iyi değer aşağıdaki gibi hesaplanmaktadır:

$$\boxed{V_{\text{opt}}(s) = \max_{a \in \text{ Actions}(s)} Q_{\text{opt}}(s,a)}$$

❒ **En iyi politika** – En iyi politika olan π_{opt}, en iyi değerlere götüren politika olarak tanımlanmaktadır. En iyi politika aşağıdaki gibi tanımlanmaktadır:

$$\forall s, \quad \boxed{\pi_{\text{opt}}(s) = \operatorname*{argmax}_{a \in \text{ Actions}(s)} Q_{\text{opt}}(s,a)}$$

❒ **Değer tekrarı** – Değer tekrarı (value iteration) en iyi politika olan π_{opt}, yanında en iyi değeri V_{opt}'ı, bulan bir algoritmadır. Değer tekrarı aşağıdaki gibi yapılmaktadır:

- İlklendirme: bütün s durumları için

$$\boxed{V_{\text{opt}}^{(0)}(s) \longleftarrow 0}$$

- Tekrar: 1'den T_{VI}'ya kadar her bir t için:

$$\forall s, \quad \boxed{V_{\text{opt}}^{(t)}(s) \longleftarrow \max_{a \in \text{ Actions}(s)} Q_{\text{opt}}^{(t-1)}(s,a)}$$

with

$$\boxed{Q_{\text{opt}}^{(t-1)}(s,a) = \sum_{s' \in \text{ States}} T(s,a,s')\Big[\text{Reward}(s,a,s') + \gamma V_{\text{opt}}^{(t-1)}(s')\Big]}$$

Not: eğer $\gamma < 1$ ya da Markov karar süreci asiklik olursa, o zaman değer tekrarı algoritmasının doğru cevaba yakınsayacağı garanti edilir.

Bilinmeyen geçişler ve ödüller

Şimdi, geçiş olasılıklarının ve ödüllerin bilinmediğini varsayalım.

❒ **Model-temelli Monte Carlo** – Model-temelli Monte Carlo yöntemi, $T(s,a,s')$ ve $\text{Reward}(s,a,s')$ işlevlerini Monte Carlo benzetimi kullanarak aşağıdaki formüllere uygun bir şekilde tahmin etmeyi amaçlar:

$$\boxed{\widehat{T}(s,a,s') = \frac{\#\text{ kere } (s,a,s') \text{ gerçekleşme sayısı}}{\#\text{ kere } (s,a) \text{ gerçekleşme sayısı}}}$$

ve

$$\boxed{\widehat{\text{Reward}}(s,a,s') = r \text{ içinde } (s,a,r,s')}$$

Bu tahminler daha sonra Q_π ve Q_{opt}'yi içeren Q-değerleri çıkarımı için kullanılacaktır.

Not: model-tabanlı Monte Carlo'nun politika dışı olduğu söyleniyor, çünkü tahmin kesin politikaya bağlı değildir.

❒ **Model içermeyen Monte Carlo** – Model içermeyen Monte Carlo yöntemi aşağıdaki şekilde doğrudan Q_π'yi tahmin etmeyi amaçlar:

$$\boxed{\widehat{Q}_\pi(s,a) = \text{ortalama } u_t,\ s_{t-1} = s \text{ ve } a_t = a \text{ olduğunda}}$$

u_t belirli bir bölümün t anında başlayan faydayı ifade etmektedir.

Not: model içermeyen Monte Carlo'nun politikaya dahil olduğu söyleniyor, çünkü tahmini değer veriyi üretmek için kullanılan π politikasına bağlıdır.

❒ **Eşdeğer formülasyon** – Sabit tanımı $\eta = \frac{1}{1+(\#\text{güncelleme sayısı } (s,a))}$ ve eğitim kümesinin her bir (s,a,u) üçlemesi için, model içermeyen Monte Carlo'nun güncelleme kuralı dışbükey bir kombinasyon formülasyonuna sahiptir:

$$\boxed{\widehat{Q}_\pi(s,a) \leftarrow (1-\eta)\widehat{Q}_\pi(s,a) + \eta u}$$

olasılıksal bayır formülasyonu yanında:

$$\boxed{\widehat{Q}_\pi(s,a) \leftarrow \widehat{Q}_\pi(s,a) - \eta(\widehat{Q}_\pi(s,a) - u)}$$

❒ **SARSA** – Durum-eylem-ödül-durum-eylem (SARSA, State-Action-Reward-State-Action), hem ham verileri hem de güncelleme kuralının bir parçası olarak tahminleri kullanarak Q_π'yi tahmin eden bir destekleme yöntemidir. Her bir (s,a,r,s',a') için:

$$\boxed{\widehat{Q}_\pi(s,a) \longleftarrow (1-\eta)\widehat{Q}_\pi(s,a) + \eta\Big[r + \gamma\widehat{Q}_\pi(s',a')\Big]}$$

Not: the SARSA tahmini, tahminin yalnızca bölüm sonunda güncellenebildiği model içermeyen Monte Carlo yönteminin aksine anında güncellenir.

❒ **Q-öğrenme** – Q-öğrenme (Q-learning), Q_{opt} için tahmin üreten politikaya dahil olmayan bir algoritmadır. Her bir (s,a,r,s',a') için:

$$\boxed{\widehat{Q}_{\text{opt}}(s,a) \leftarrow (1-\eta)\widehat{Q}_{\text{opt}}(s,a) + \eta\Big[r + \gamma \max_{a' \in \text{ Actions}(s')} \widehat{Q}_{\text{opt}}(s',a')\Big]}$$

❒ **Epsilon-açgözlü** – Epsilon-açgözlü politika, ϵ olasılıkla araştırmayı ve $1-\epsilon$ olasılıkla sömürüyü dengeleyen bir algoritmadır. Her bir s durumu için, π_{act} politikası aşağıdaki şekilde hesaplanır:

$$\boxed{\pi_{\text{act}}(s) = \begin{cases} \underset{a \in \text{ Actions}}{\text{argmax}} \ \widehat{Q}_{\text{opt}}(s,a) & \text{olasılıkla } 1-\epsilon \\ \text{Actions}(s) \text{ eylem kümesi içinden rastgele} & \text{olasılıkla } \epsilon \end{cases}}$$

Oyun oynama

Oyunlarda (örneğin satranç, tavla, Go), başka oyuncular vardır ve politikamızı oluştururken göz önünde bulundurulması gerekir.

❒ **Oyun ağacı** – Oyun ağacı, bir oyunun olasılıklarını tarif eden bir ağaçtır. Özellikle, her bir düğüm, oyuncu için bir karar noktasıdır ve her bir kökten (root) yaprağa (leaf) giden yol oyunun olası bir sonucudur.

❒ **İki oyunculu sıfır toplamlı oyun** – Her durumun tamamen gözlendiği ve oyuncuların sırayla oynadığı bir oyundur. Aşağıdaki gibi tanımlanır:

- bir başlangıç durumu s_{start}
- s durumunda gerçekleştirilebilecek olası eylemler $\text{Actions}(s)$
- durumunda a eylemi gerçekleştirildiğindeki ardıllar $\text{Succ}(s,a)$
- bir bitiş durumuna ulaşılıp ulaşılmadığı $\text{IsEnd}(s)$
- s bitiş durumunda etmenin elde ettiği fayda $\text{Utility}(s)$
- s durumunu kontrol eden oyuncu $\text{Player}(s)$

Not: oyuncu faydasının işaretinin, rakibinin faydasının tersi olacağını varsayacağız.

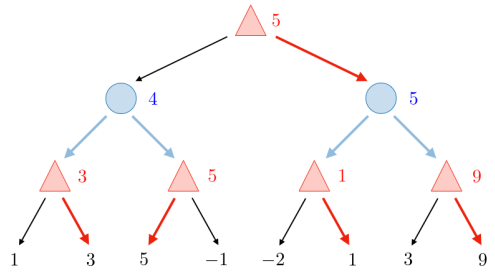
❒ **Politika türleri** – İki tane politika türü vardır:

- $\pi_p(s)$ olarak gösterilen belirlenimci politikalar, p oyuncusunun s durumunda gerçekleştirdiği eylemler.
- $\pi_p(s,a) \in [0,1]$ olarak gösterilen olasılıksal politikalar, p oyuncusunun s durumunda a eylemini gerçekleştirme olasılıkları.

❒ **Expectimax** – Belirli bir s durumu için, en yüksek beklenen değer olan $V_{\text{exptmax}}(s)$, sabit ve bilinen bir rakip politikası olan π_{opp}'a göre oynarken, bir oyuncu politikasının en yüksek beklenen faydasıdır. En yüksek beklenen değer (expectimax) aşağıdaki gibi hesaplanmaktadır:

$$\boxed{V_{\text{exptmax}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \max\limits_{a \in \text{Actions}(s)} V_{\text{exptmax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{agent} \\ \sum\limits_{a \in \text{Actions}(s)} \pi_{\text{opp}}(s,a) V_{\text{exptmax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{opp} \end{cases}}$$

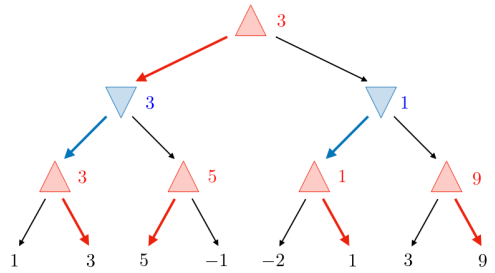
Not: en yüksek beklenen değer, MDP'ler için değer yinelemenin analog halidir.



❒ **Minimax** – En küçük-enbüyük (minimax) politikaların amacı en kötü durumu kabul ederek, diğer bir deyişle; rakip, oyuncunun faydasını en aza indirmek için her şeyi yaparken, rakibe karşı en iyi politikayı bulmaktır. En küçük-en büyük aşağıdaki şekilde yapılır:

$$\boxed{V_{\text{minimax}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \max\limits_{a \in \text{Actions}(s)} V_{\text{minimax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{agent} \\ \min\limits_{a \in \text{Actions}(s)} V_{\text{minimax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{opp} \end{cases}}$$

Not: π_{max} ve π_{min} değerleri, en küçük-en büyük olan $V_{minimax}$'dan elde edilebilir.



❒ **Minimax özellikleri** – V değer fonksiyonunu ifade ederse, en küçük-en büyük ile ilgili aklımızda bulundurmamız gereken 3 özellik vardır:

- *Özellik 1*: oyuncu politikasını herhangi bir π_{agent} ile değiştirecek olsaydı, o zaman oyuncu daha iyi olmazdı.

$$\boxed{\forall \pi_{\text{agent}}, \quad V(\pi_{\max}, \pi_{\min}) \geqslant V(\pi_{\text{agent}}, \pi_{\min})}$$

- *Özellik 2*: eğer rakip oyuncu politikasını π_{min}'den π_{opp}'a değiştirecek olsaydı, o zaman rakip oyuncu daha iyi olamazdı.

$$\boxed{\forall \pi_{\text{opp}}, \quad V(\pi_{\text{max}},\pi_{\text{min}}) \leqslant V(\pi_{\text{max}},\pi_{\text{opp}})}$$

- *Özellik 3*: eğer rakip oyuncunun muhalif (adversarial) politikayı oynamadığı biliniyorsa, o zaman en küçük-en büyük politika oyuncu için ey iyi (optimal) olmayabilir.

$$\boxed{\forall \pi, \quad V(\pi_{\text{max}},\pi) \leqslant V(\pi_{\text{exptmax}},\pi)}$$

Sonunda, aşağıda belirtildiği gibi bir ilişkiye sahip oluruz:

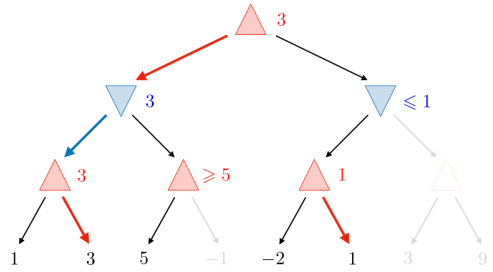
$$\boxed{V(\pi_{\text{exptmax}},\pi_{\text{min}}) \leqslant V(\pi_{\text{max}},\pi_{\text{min}}) \leqslant V(\pi_{\text{max}},\pi) \leqslant V(\pi_{\text{exptmax}},\pi)}$$

Minimax hızlandırma

❒ **Değerlendirme işlevi** – Değerlendirme işlevi, alana özgü ve $V_{\text{minimax}}(s)$ değerinin yaklaşık bir tahminidir. Eval(s) olarak ifade edilmektedir.

Not: FutureCost(s) arama problemleri için bir benzetmedir.

❒ **Alpha-beta budama** – Alpha-beta budama (alpha-beta pruning), oyun ağacının parçalarının gereksiz yere keşfedilmesini önleyerek en küçük-en büyük algoritmasını en iyileyen (optimize eden) alana-özgü olmayan genel bir yöntemdir. Bunu yapmak için, her oyuncu ümit edebileceği en iyi değeri takip eder (maksimize eden oyuncu için α'da ve minimize eden oyuncu için β'de saklanır). Belirli bir adımda, $\beta < \alpha$ koşulu, önceki oyuncunun emrinde daha iyi bir seçeneğe sahip olması nedeniyle en iyi (optimal) yolun mevcut dalda olamayacağı anlamına gelir.



❒ **TD öğrenme** – Geçici fark (TD, temporal difference) öğrenmesi, geçiş/ödülleri bilmediğimiz zaman kullanılır. Değer, keşif politikasına dayanır. Bunu kullanabilmek için, oyununun kurallarını, Succ(s,a), bilmemiz gerekir. Her bir (s,a,r,s') için, güncelleme aşağıdaki şekilde yapılır:

$$\boxed{w \longleftarrow w - \eta\Big[V(s,w) - (r + \gamma V(s',w))\Big]\nabla_w V(s,w)}$$

Eşzamanlı oyunlar

Bu, oyuncunun hamlelerinin sıralı olmadığı sıra temelli oyunların tam tersidir.

❒ **Tek-hamleli eşzamanlı oyun** – Olası hareketlere sahip A ve B iki oyuncu olsun. $V(a,b)$, A'nın a eylemini ve B'nin de b eylemini seçtiği A'nın faydasını ifade eder. V, getiri dizeyi (payoff matrix) olarak adlandırılır.

❒ **Stratejiler** – İki tane ana strateji türü vardır:

- Saf strateji, tek bir eylemdir:

$$\boxed{a \in \text{Actions}}$$

- Karışık strateji, eylemler üzerindeki bir olasılık dağılımıdır:

$$\forall a \in \text{Actions}, \quad \boxed{0 \leqslant \pi(a) \leqslant 1}$$

❒ **Oyun değerlendirme** – Oyuncu A π_A'yı ve oyuncu B de π_B'yi izlediğinde, oyun değeri $V(\pi_A,\pi_B)$:

$$\boxed{V(\pi_A,\pi_B) = \sum_{a,b} \pi_A(a)\pi_B(b)V(a,b)}$$

❒ **Minimax teoremi** – π_A, π_B'nin karma stratejilere göre değiştiğini belirterek, sonlu sayıda eylem ile eşzamanlı her iki oyunculu sıfır toplamlı oyun için:

$$\boxed{\max_{\pi_A}\min_{\pi_B} V(\pi_A,\pi_B) = \min_{\pi_B}\max_{\pi_A} V(\pi_A,\pi_B)}$$

Sıfır toplamı olmayan oyunlar

❒ **Getiri matrisi** – $V_p(\pi_A,\pi_B)$'yi oyuncu p'nin faydası olarak tanımlıyoruz.

❒ **Nash dengesi** – Nash dengesi (π_A^*,π_B^*) öyle birşey ki hiçbir oyuncuyu, stratejisini değiştirmeye teşvik etmiyor:

$$\boxed{\forall \pi_A, V_A(\pi_A^*,\pi_B^*) \geqslant V_A(\pi_A,\pi_B^*)} \quad \text{and} \quad \boxed{\forall \pi_B, V_B(\pi_A^*,\pi_B^*) \geqslant V_B(\pi_A^*,\pi_B)}$$

Not: sonlu sayıda eylem olan herhangi bir sonlu oyunculu oyunda, en azından bir tane Nash denegesi mevcuttur.