8.6 Trajectory Sampling

이 section 에서는 updates를 distribute 하는 두 방법을 비교한다.
classical approach 는 dynamic programming에서 나왔고?
전체 state (or state-action) space 를 sweeps 하는 거다.
매 sweep 마다 각 state(or state-action pair) 를 update 하고.
이건 large tasks 에선 문제가 된다.
한 sweep 조차도 완료할 시간이 없을 수 있거든.
많은 tasks 에서 대다수의 states 는 irrelevant 하다.
걔들은 very poor policies 에서만 방문되거나 매우 낮은 확률로 방문되기 때문이다.
exhaustive sweeps 는 암묵적으로 state space의 모든 부분에 같은 시간을 할애한다.
필요한 부분에 초점을 맞추면 좋은데 말이지.
챕터 4에서 얘기했듯, exhaustive sweeps 와 그들이 내포하는 all states 에 같은 treatment 는 dynamic programming 의 필요한 특성이 아니다.
이론적으로, updates 는 마음대로 distribute 할 수 있다.(convergence를 보장하고, 모든 state or state-action pairs 가 무한개의 time 동안 방문되도록만 하면? 예외는 8.7에서 논한다.)
하지만 실제로는 exhaustive sweeps 이 자주 쓰인다.

두 번째 approach 는 어떤 분포에 따라 state 나 state-action space 에서 sample 하는거다.
Dyna-Q agent 에서처럼 uniformly sample 할 수도 있지만, 그러면 exhaustive sweeps 와 같은 문제를 겪는다.
더 좋은 방법은, on-policy distribution 에 따라 updates 를 distribute 하는거다.
그러니까, 현재 policy 를 따를 때 관찰한 distribution 에 따라 말이지.
이 분포의 장점은 생성하기 쉽다는 거다.
현재 policy 를 따르며 model 과 interact 하기만 하면 된다.
episodic task 에선, start state 에서 시작한다.(또는 starting-state distribution 에 따라)
그리고 terminal state 까지 simulate 한다.
continuing task 에선, 아무곳에서나 시작해서 그냥 계속 simulate 한다.
두 case 모두, sample state transitions 과 rewards 는 model 에 의해 주어지고
sample actions 은 현재 policy 에 의해 주어진다.
그러니까, explicit individual trajectories 를 simulate 하고 그 과정에서 만나는 state 나 state-action pairs 에서 update를 수행한다.
이런 방식으로 experience 와 updates 를 생성하는 걸
trajectory sampling 이라고 한다.

trajectory sampling 말고는 on-policy distribution 을 따라 updates 를 효율적으로 distribute 하는 방법을 생각해내기 어렵다.
on-policy distribution 의 explicit representation 이 있으면,
모든 states 를 sweep 할 수 있을거다. on-policy distribution 을 따라 각 state 의 update 를 weighting 하면서.
하지만 그러면 exhaustive sweeps 처럼 계산 비용이 크다.
아마 그 distribution 에서 개별 state-action pairs를 sample 하고 update 할 수 있을거야.
하지만 이게 효율적으로 된다해도, 이게 simulating trajectories 하는 것보다 뭐가 더 좋을까?
on-policy distribution 을 explicit form 으로 아는 것조차 불가능하다.
distribution 은 policy 가 바뀔때마다 변하고,
distribution 을 계산하려면 전체 policy evaluation 에 필적하는 계산이 필요하다.
그런 다른 가능성들을 고려하면 trajectory sampling 이 efficient 하면서 elegant 해 보인다.

updates 의 on-policy distribution 이 좋은걸까?
직관적으로 좋아보이긴 한다. 최소한 uniform distribution 보단 일단 좋아보여.
예를 들어, 체스를 학습한다면, 실제 게임에서 일어날만한 위치를 학습하는 게 낫겠지. 그냥 랜덤한 위치보다.
후자는 valid states 일 수 있지만(유효할 수도 있지만),
걔들을 정확히 value 하기 위해선 실제 게임에서의 위치를 evaluate 하는 다른 스킬이 필요하다.
part 2 에서 on-policy distribution 이 function approximation 이 쓰였을 때 아주 큰 이점이 있다는 걸 볼거야.
function approximation 이 쓰이든 안 쓰이든, on-policy focusing 이 planning 속도를 크게 개선시켜줄거라 기대할거야.

on-policy distribution 에 초점을 맞추는 건 beneficial 일 수 있어.
왜냐면 space 의 넓은 uninteresting parts 를 무시하거든.
하지만 detrimental 할 수도 있어.
space의 같은 old parts 가 계속 반복해서 update 되게 하거든.
그 효과를 empirically(경험적으로) 평가하기 위해 작은 실험을 수행했어.
update distribution 의 효과를 isolate 하기 위해, 8.1 에서 정의한 것과 같은 one-step expected tabular methods 를 전체적으로 썼다.

uniform case에선, 모든 state-action pairs 를 반복했다. 각각 제 자리에 update 하면서.
on-policy case에선, episodes 를 simulate 했다. 모두 같은 state 에서 시작해, 현재의 ε-greedy policy(ε = 0.1) 하에서 발생한 각 state-action pair 를 update 했다.

tasks 는 undiscounted episodic tasks 였고 다음과 같이 랜덤하게 생성됐다.
각 $|\mathcal{S}|$ states 에서,
두 가지 actions 이 가능했고, 각각 b next states 중 하나로 갔다.
all equally likely(확률이 같았고),
각 state-action pair 에 대해 b states 의 다른 random selection 을 했다?
branching factor b는 모든 state-action pairs 에 대해 같았다.
모든 transitions 에서 0.1 확률로 terminal state 로 갔다.
현재 policy 의 quality 의 명확한 measure를 얻기 위해 episodic tasks 를 썼다.
planning process의 어떤 point 에서든 멈추고 $v_\pi(s_0)$를 exhaustively 계산할 수 있다.
$v_\pi(s_0)$는 action-value function Q 이 주어졌을 때 greedy policy π~ 하에서 start state 의 true value 인가봐.
그걸 agent 가 greedy하게 움직이는 새로운 episode 에서 얼마나 잘할지의 지표로 쓴다?
(내내 model 이 정확하다는 가정을 하고..)

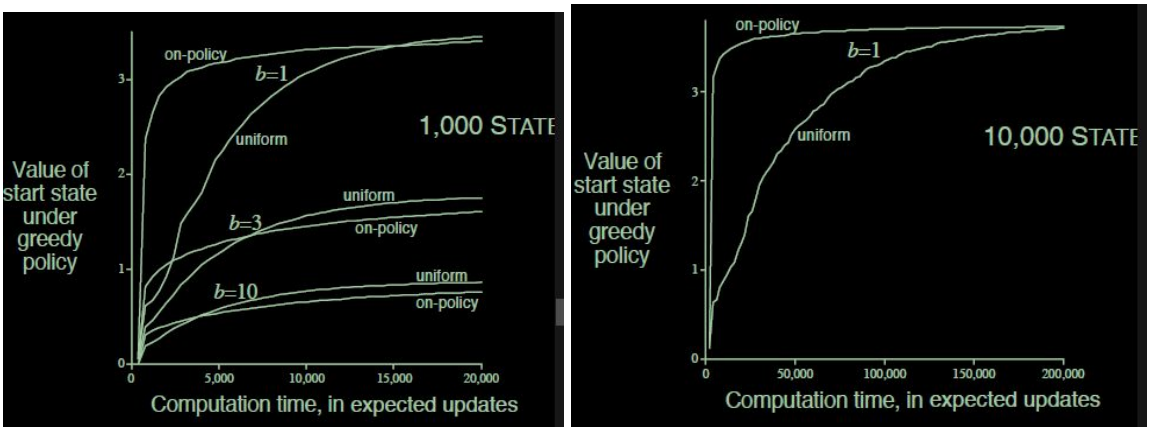


Figure 8.9: Relative efficiency of updates distributed uniformly across the state space versus focused on simulated on-policy trajectories, each starting in the same state. Results are for randomly generated tasks of two sizes and various branching factors, b.

Fig 8.9 의 왼쪽은 200 sample tasks 를 평균한 결과다.
1000 states 와 branching factor 가 1, 3, 10이다.
찾아진 policies 의 quality 가 완료된 expected updates 의 수의 함수로 그려져 있다.
모든 cases 에서 on-policy distribution 에 따른 sampling 이
처음엔 planning 을 빠르게 하지만 long run 에선 느리다.
effect 가 강할수록, 더 빠른 planning 의 초반 기간이 길다. b가 더 작을때.

다른 실험에서는, 이런 효과들이 states 가 커지면 더 강하게 나타나는 걸 알 수 있다.
예를 들어 오른쪽 그림에서 b가 1이고 states 가 10,000 일 때
on-policy focusing 의 장점이 크게 오래 간다.

이 모든 결과는 말이 된다.
short term 에선, on-policy distribution 에 따른 sampling 이 도움이 된다.
start state 의 후손에 가까운 states 에 집중하니까.
states 가 많고 branching factor 가 작으면
이 효과가 크고 오래간다.
long run 에선, on-policy distribution 에 집중하는건 안 좋을 수 있다.
흔히 발생하는 states 들은 이미 correct values 를 갖고 있기 때문이다.
그들을 sampling 하는 건 필요없고, 다른 states 를 sampling 하는 게 좋겠지.
이게 아마 exhaustive, unfocused approach 가 long run 에서 좋은 이유일거야.
최소한 작은 문제에서 말이야.
하지만 큰 문제에선 on-policy distribution 에 따른 sampling 이 좋겠지.
특히 on-policy distribution 하에서 state-action space 의 작은 subset 만 방문되는 문제에서.

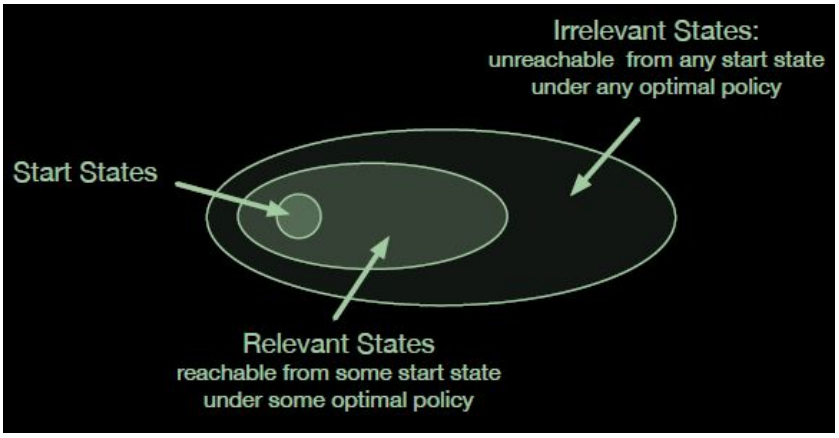
8.7 Real-time Dynamic Programming (RTDP)

real-time dynamic programming, or RTDP 는
DP의 value iteration 알고리즘의 on-policy trajectory-sampling 버전이다.
전통적인 sweep-based policy iteration 과 깊이 연관돼 있기 때문에
RTDP는 on-policy trajectory sampling 이 줄 수 있는 이점을 명확히 그려준다.

RTDP 는 (4.10) 에서 정의한 것처럼
expected tabular value-iteration updates로
actual 이나 simulated trajectories 에서 방문된 states 의 value 를 update 한다.
기본적으로 Fig 8.9 에서 본 on-policy 결과를 만든 알고리즘이다.

RTDP 와 conventional DP 사이의 close connection 은 존재하는 이론을 적용해 이론적 결과를 이끌어내는 걸 가능하게 한다. 이게 뭔 말..
RTDP는 asynchronous DP 알고리즘의 예시다. section 4.5에서 봤었지?
asynchronous DP 알고리즘은 state set 의 systematic sweeps 의 관점에서 organize 되지 않았다.
걔들은 순서가 어떻게 되든 상관없이 state를 udpate 한다. 다른 가능한 states의 values 아무거나 이용해서?
RTDP 에선, update 순서가 real 이나 simulated trajectories 에서 방문된 states 의 순서에 따라 정해진다.

trajectories 가 start states 의 정해진 set 에서만 시작할 수 있으면,
그리고 주어진 policy에 대해 prediction problem 에 관심이 있다면,
그럼 on-policy trajectory sampling 이, 알고리즘이 주어진 policy 에서 어떤 start states 에서도 갈 수 있는 states 는 완전히 skip 하게 해준다.
unreachable states 는 prediction problem 에 상관없는 애들이다.
주어진 policy 에서 evaluate 하는 게 아니라 optimal policy 를 찾는 게 목적인 control problem 에선, 어떤 optimal policy 하에서든 start states 에서 도달할 수 없는 states 가 있을 수 있고,
이런 상관없는 states 에 대한 optimal actions 을 찾을 필요가 없다.
필요한 건 optimal partial policy 이다.
이게 뭐냐면, policy인데, relevant states 에 대해선 optimal 이지만
irrelevant states 에 대해선 임의의 actions 을 specify 하거나 undefine 할 수 있는 policy 다.



하지만 Sarsa(6.3)와 같은 그런
on-policy trajectory-sampling control method 를 쓰는
optimal partial policy 를 찾는 건,
일반적으로 모든 state-action pairs 를 무한 번 방문하는 걸 필요로 한다. irrelevant 로 판명된 state-action pairs 라 할지라도.
이건 예를 들어 exploring starts 를 사용해 수행할 수 있다.(5.3)
이건 RTDP 에서도 마찬가지다.
exploring starts 를 쓰는 episodic tasks 에서

RTDP는 discounted finite MDPs 에 대한 optimal policies로 수렴하는 asynchronous value-iteration 알고리즘이다.(어떤 조건 하에서는 undiscounted case 에 대해서)
prediction problem 에서의 상황과 다르게,
optimal policy 로의 convergence가 중요하다면,
일반적으로 state 나 state-action pair 를 update 하는 걸 멈추는 건 불가능하다.

RTDP 의 가장 흥미로운 결과는
일정 조건을 만족하는 특정 타입의 문제에 대해
RTDP는 모든 state 를 무한 번 방문하지 않고 relevant states 에 대한 optimal policy를 찾을 수 있음이 보장된다는 것이다.
어떤 states 는 아예 방문 안해도 말이지.
실제로, 어떤 문제에선, states 의 조그만 부분만 방문해도 된다.
한 번의 sweep 도 어러운 large state sets 땐 아주 좋겠지?

이 결과가 유효한 tasks 는 0의 rewards를 생성하는 absorbing goal states 가 있는 MDPs 의 undiscounted episodic tasks 이다. section 3.4 에 있는거.
real or simulated trajectory 의 모든 step 에서,
RTDP 는 greedy action 을 고른다. (tie 는 randomly break)
그리고 현재 state 에 expected value iteration update operation 을 적용한다.
각 step 에서 다른 states 의 임의의 collection 의 values 를 update 할 수도 있다.
예를 들어, 현재 state 에서 limited-horizon look-ahead search? 로 방문한 states 의 values 를 update 할 수 있다.

start states 의 set에서 랜덤하게 골라 각 episode를 시작해 goal state 로 끝나는 이런 문제들에서, RTDP 는 1의 확률로 모든 relevant states 에 대해 optimal 인 policy 로 수렴한다.
다음 조건들을 만족한다면 말이지.

1) 모든 goal state 의 initial value 는 0이다.
2) 어느 start state 에서도 반드시 goal state 에 도달하는 걸 보장하는 oplicy 가 최소한 하나는 있다.
3) non-goal staets 에서의 transitions 에 대한 모든 rewards 는 strictly negative.
4) 모든 initial values 는 그들의 optimal values 보다 크거나 같다.(다 0으로 세팅해서 만족시킬 수 있음)

이런 특성들을 가진 tasks 는 stochastic optimal path problems 의 예시들이다.
주로 reward maximization 대신 cost minimization 의 관점으로 보는 애들이다.
우리가 여기서 하는 것처럼.
우리 버전에서 negative returns 을 maximize 하는건 start state 에서 goal state로 가는 길의 costs 를 minimize 하는 것과 equivalent 하다.
이런 종류의 task 의 예는 minimum-time control tasks 이다.
goal 로 향하는 각 time step 에서 -1 의 reward 를 내거나, section 3.5 의 golf 예시처럼 objective 가 가장 적은 strokes 로 hole 에 넣는 문제들을 말해.

(Example 8.6): RTDP on the Racetrack
Exercise 5.7(page 90) 의 racetrack 문제는 stochastic optimal path problem 이다.
이 문제에서 RTDP와 conventional DP value iteration algorithm 을 비교해보면
on-policy trajectory sampling 의 장점을 볼 수 있다.

Fig 5.5 의 그림이다.
exercise 에서, agent 가 turn 하는 방법을 학습해야했고, finish line 에 최대한 빨리 도착해야했었다.
start states 는 starting line 의 zero-speed states다.
goal states 는 track 안에서 한 스텝으로 finish line 에 도달할 수 있는 모든 states다.
exercise 5.7 과 다르게, 여기선 speed 에 제한이 없다.
그래서 state set이 잠재적으로 무한이다.
하지만 start states 의 set 에서 any policy로 도달할 수 있는 states 의 set은 finite 이고 problem 의 state set 으로 간주될 수 있다.
각 episode 는 랜덤하게 선택된 start state 에서 시작하고 finish line 을 지나면 끝난다.
finish line 을 지나기 전까지 각 step 에서 rewards는 -1이다.
track 의 경계를 넘으면 random start state 로 돌아가고 episode 는 계속된다.

왼쪽의 작은 racetrack 은 any policy 에 의해 start state 에서 도달가능한 states 가 9,115개 있고 그 중 599 개만이 relevant 하다. 그게 무슨 뜻이냐면, 어떤 start state 에서 어떤 optimal policy 를 통해 도달가능하다는 뜻.
relevant states 의 수는 10^7 개의 episodes 에 대해 optimal actions 을 수행하는 동안 방문한 states를 세서 estimate 한 것.

아래 표는 이 문제를 conventional DP 와 RTDP 로 푼 걸 비교한 거다.
이 결과들은 25 runs 를 평균하고 각기 다른 random number seed 로 시작했다.
이 case에서 conventional DP 는 state set 의 exhaustive sweeps 을 사용하는 value iteration이다. 한 번에 한 state 씩 value 를 update 한다.
각 state에 대한 update 가 다른 states 의 가장 최근 values 를 사용한다는 걸 뜻한다.
이게 Gauss-Seidel version 의 value iteration 이다. Jacobi 보다 두 배 더 빠른. section 4.8 을 보도록.
updates 의 순서에는 주의를 기울이지 않는다.
다른 orderings 이 더 빠른 convergence를 만들수도.
두 방법 다 initial values 는 각 run 에서 전부 0.
DP 는 한 sweep 에서 state value 의 maximum change 가 10^{-4} 보다 작으면 수렴으로 판정.
RTDP 는 finish line 을 넘는 20번의 episodes 동안의 average time 이 특정 수의 steps 으로 안정적으로 점근?해가면 수렴으로 판정.
이 버전의 RTDP는 각 step 에서 현재 state의 value 만 update 했다.

	DP	RTDP
Average computation to convergence	28 sweeps	4000 episodes
Average number of updates to convergence	252,784	127,600
Average number of updates per episode	—	31.9
% of states updated $\leq$ 100 times	—	98.45
% of states updated $\leq$ 10 times	—	80.51
% of states updated 0 times	—	3.18

두 방법 모두 평균적으로 14-15 steps 사이에 finish line 을 통과하는 policies 를 만들었다.
하지만 RTDP는 DP 가 한 updates 의 거의 절반만 필요했을 뿐이다.
이게 RTDP의 on-policy trajectory sampling의 결과다.
DP의 각 sweep 에서 모든 state 의 value가 update 된 반면,
RTDP는 더 적은 states 에 초점을 맞췄다.
average run 에서, RTDP는 98.45% 의 states 의 values를 100 번 미만으로 update 했고
80.51% 의 states 가 10번 미만이다.
290개 states 의 values 는 average run 에서 아예 update 안 됐다.

RTDP의 다른 장점은, value function 이 optimal value function v_* 로 다가감에 따라,
agent 가 trajectories 를 생성하기 위해 사용한 policy 가 optimal policy 로 다가간다.
왜냐면 이건 항상 현재 value function 에 대해 greedy 거든.
이건 conventional value iteration 에서의 상황과 대조적이다.
실제에선, value iteration 은 value function 이 한 sweep 에서 조금만 변할 때 끝난다.
위 table 에서 이렇게 했었어.
이 부분에서, value function 은 v_*에 가깝게 근사하고,
greedy policy 는 optimal policy 에 가깝다.
하지만, 가장 최근의 value function 에 대해 greedy 한 policies 가 optimal 일 수 있다.
아니면 거의 그렇거나. value iteration 이 종료되기 한참 전부터.
챕터 4에서 optimal policies 가 v_* 뿐 아니라 여러 다른 value functions 에 대해 greedy 할 수 있는 걸 봤었지. 기억나? 안나여..ㅜㅠ
value iteration 이 수렴하기 전에 optimal policy 가 발생하는 걸 체크하는 건
conventional DP 알고리즘의 부분이 아니야. 추가적인 계산이 필요하지.

racetrack example 에서, 각 DP sweep 이후 test episodes 를 많이 돌려보자.
해당 sweep 의 결과에 따라 actions 을 greedy하게 선택하면서.
DP 계산에서 충분히 좋을 정도로 근사된 optimal evaluation function 을 일찍 찾을 수 있다.
대응하는 greedy policy 가 거의 optimal 인 애 말이지.
이 racetrack에 대해, close-to-optimal policy 는 value iteration 을 15번 sweep 해서 나왔어.
아니면 136,725 value-iteration updates 이후에 말이지.
이건 DP 가 v_*에 수렴하기 위해 필요했던 252,784보다 훨씬 적지?
하지만 여전히 RTDP 보다 커.

이런 simulations 이 절대적인 비교는 아니었어.
하지만 on-policy trajectory sampling 의 장점들을 엿볼 수 있었지.
conventional value iteration 이 모든 states의 value 를 계속 update 하는 반면
RTDP는 문제의 목적에 관련된 states 에만 초점을 맞췄지.
이 초점이 학습이 진행될수록 아주 narrow 해 져.
RTDP에 대한 수렴 이론을 simulations 에 적용하기 때문에 RTDP가 결국 relevant states 에만 집중하게 될 거란 걸 알아.
optimal paths 를 만드는 states 들 말이지.
RTDP는 sweep-based value iteration 보다 50% 정도의 계산으로 optimal control 을 달성했어.

8.8 Planning at Decision Time

planning 은 최소 두 가지 방식으로 쓰일 수 있다.

하나는 이번 챕터에서 한 건데, dynamic programming 과 Dyna 로 정형화된다.
planning 을 policy 나 value function 을 gradually improve 하는 데 쓰는 방법이다.
model 로부터 얻은 simulated experience 에 기반해서.(sample 이나 distribution model 아무거나)
그러면 actions을 선택하는 건,
tabular case 에선 table 에서 얻은 현재 state 의 action values 비교하거나
아니면 아래의 part 2 에서 다룰 approximate methods 에선 mathematical expression 을 evaluate 하는 문제가 된다.
any current state S_t 에 대해 action 이 선택되기 한참 전에,
planning 이 S_t를 포함해 많은 states 에서의 action 을 선택하는 데 필요한 table entries 나 mathematical expression을 개선하는 역할을 한다.
이런 식으로 사용했을 때, planning 은 current state에 초점을 맞추는 게 아니다.
이런 걸 *background planning* 이라고 한다.

planning 을 이용하는 다른 방법은,
각 새로운 state S_t를 만난 후에 시작하고 완료하는거다.
single action A_t 를 고르는 computation을 하는거다.
다음 step에서 planning 은 S_{t+1}로 시작해 A_{t+1}을 만든다.
이렇게 planning 을 사용하는 예 중 가장 간단하고 거의 퇴화한? 예는,
state values 만이 available 하고, action 이 각 action에 대해 model-predicted next states 의 values 를 비교해 선택되는 경우다. (또는 tic-tac-toe 처럼 afterstates 의 values 를 비교해서 선택)
더 일반적으로, 이렇게 사용되는 planning 은 one-step-ahead 보다 훨씬 더 깊이 보고
다른 많은 predicted state 와 reward trajectories 로 이끄는 action choices 를 evaluate 할 수 있다.
planning 을 처음 사용한 것과는 달리, 여기선 planning 이 특정 state 에 초점을 맞춘다.
이걸 *decision-time planning* 이라고 부른다.

planning 에 대한 이 두 가지 사고방식,
그러니까 simulated experience를 써서 policy 나 value function 을 gradually improve 하는 거,
또는 simulated experience를 써서 현재 state 에 대한 action 을 고르는 것,
들이 합쳐질 수 있다. 근데 분리돼서 연구되는 경향이 있고 이해하기에 좋은 방법이다.
이제 decision-time planning 을 구체적으로 보자.

planning 이 decision time 에만 수행되는 경우에도, 우린 그걸 볼 수 있다. section 8.1 에서 한 것처럼.
simulated experience 에서 updates 와 values, 그리고 궁극적으로 policy 에 이르기까지.
이제 values 와 policy 가 현재 state와 거기서 선택가능한 action 이 specific 하니,
planning process에서 생성된 values 와 policy 는 일반적으로 현재 action 을 선택하는 데 쓰이고 버려진다.
많은 applications 에서 이건 큰 loss 가 아냐. 왜냐면 매우 많은 states 가 있고 우린 오랜 시간동안 같은 state로 돌아갈 가능성이 없기 때문이야.
일반적으로, 둘을 섞길 원할거야.
현재 state 에 planning 을 집중하고,
planning 결과를 저장해 나중에 같은 state 로 돌아올 경우 훨씬 더 멀어지도록 하는거지.
decision-time planning 은 빠른 반응이 필요하지 않은 applications 에서 아주 유용하다.
예를 들어, 체스에선 각 move 에 수 초에서 수 분까지 허용된다.

잘 짜여진 프로그램은 이 시간동안 여러 moves 를 plan 할 수 있겠지.
반면, low latency action selection 이 필요할 땐,
background 에서 planning 해서 새로 발생한 각 state 에 빠르게 적용될 수 있는 policy 를 계산하는 게 좋다.

8.9 Heuristic Search

인공지능에서 classical 한 state-space methods 는
decision-time planning methods 이다.
heuristic search 라고 알려져 있지.
heuristic search 에선, 만난 각 state에 대해,
가능한 continuations 의 큰 트리를 생각한다.
approximate value function 이 leaf nodes 에 적용되고
root 에서 현재 state 로 backed up 된다.
search tree 내에서의 backing up 은 v_*, q_*에 대해 max 로 expected updates 할 때 했던 것과 같다.
backing up 은 현재 state 에 대한 state-action nodes 에서 멈춘다.
이 노드들의 backed-up values가 계산되면,
그 중 best 가 현재 action 으로 선택되고,
모든 backed-up values 는 버린다.

conventional heuristic search 에서는 approximate value function 을 바꿔서 backed-up values 를 저장하려는 노력을 하지 않았다.
사실, value function 은 일반적으로 사람에 의해 디자인되고 search 의 결과로서 바뀌지 않았다.
하지만, 시간에 따라 value function 이 향상되는 게 자연스럽다.
heuristic search 동안 계산된 backed-up values 를 사용하든
이 책에서 소개한 다른 어떤 방법을 사용하든 말이다.
어떤 의미에서 우리는 이 approach 를 줄곧 취해왔다.
greedy, ε-greedy, UCB action-selection methods 는 heuristic search 와 다르지 않다.
비록 작은 scale 에 대한 것이긴 해도.
예를 들어, 주어진 model 과 state-value function 에서 greedy action 을 계산하기 위해서,
각 가능한 next state 로 가는 각 가능한 action 을 미리 내다봐야한다.
rewards 와 estimated values 를 고려하면서.
그리고는 best action 을 고르지.
conventional heuristic search 에서처럼,
이 프로세스는 가능한 actions 의 backed-up values 를 계산한다.
하지만 저장할 생각은 않지.
그래서, heuristic search 는 single step 을 넘어 greedy policy idea 의 확장으로 볼 수 있다.

one step 보다 더 깊이 search 하는 것의 포인트는,
더 좋은 action selections 을 얻는다는 것이다.
만약 완벽한 모델이 있고 불완전한 action-value function 이 있을 때,
deeper search 는 주로 더 나은 policies 를 만든다.
확실히, search 가 episode 의 끝까지 이어진다면,
불완전한 value function 의 영향은 없어질 거다.
그리고 이렇게 결정된 action 은 반드시 optimal 일 거고.

search 가 γ^k 가 아주 작을 정도로 충분한 depth k 라면,
actions 은 그에 따라 optimal 에 가까울 것이다.
반면 search 가 깊어질수록, 더 많은 연산이 필요하다.
그래서 response time 이 더 느려진다.
좋은 예는 section 16.1 의 TD-Gammon 이다.
이 system 은 TD learning 을 이용해 self-play 로 afterstate value function 을 학습했다.
움직이기 위해 heuristic search 의 폼을 썼다.
모델로서, TD-Gammon 은 주사위 확률의 prior knowledge 를 사용했다.
그리고 상대가 항상 TD-Gammon 이 best 로 평가한 actions 을 선택했다는 가정을 했다.
heuristic search 가 깊어질수록 TD-Gammon 에 의한 움직임은 더 좋아졌지만 시간은 더 길어졌다.
backgammon 은 branching factor 가 큰데 moves 는 몇 초반에 해야한다.
선택적으로 몇 steps 앞만 search 했지만 search 는 상당히 더 좋은 action selections 을 했다.

heuristic search 가 updates 에 초점을 맞추는 가장 명백한 방법을 간과해선 안 된다.
현재 state 에 한단 말이지.
heuristic search 의 유효성의 대부분은 current state 에 바로 뒤따르는 states 와 actions 에 바짝 초점을 맞추는 search tree 덕분이다.
체스둘 때 가능성이 높은 다음 moves 와 이어지는 positions 을 생각하겠지.
내가 어떻게 actions 을 선택하든 updates 의 가장 높은 우선순위가 되고 가장 시급히 approximate value function 이 정확하길 바라는 곳이 바로 이들 states 와 actions 이야.
임박한 events 에 계산이 우선적으로 할당되어야 해.
메모리도 무한이 아니지? 그럼 여기 먼저 써야해.
체스에서, distinct value estimates 를 저장하기엔 너무 많은 가능한 positions 이 있어.
heuristic search 에 기반한 체스프로그램은 single position 에서 내다본 수많은 positions 에 대해 distinct estimates 를 쉽게 저장할 수 있어.
이 엄청난 메모리의 focusing 과 현재 decision 에 쏟는 계산 resources 가 heuristic search 가 그렇게 효과적인 이유야.

updates 의 distribution 이 비슷한 방식으로 변경되어 현재 state 와 그 likely successors 에 초점을 맞출 수도 있다.
limiting case(제한적인 경우?) 에 search tree를 구성하기 위해 heuristic search 의 methods 를 정확히 사용할 수 있을거다.
그러고 individual, one-step updates 를 bottom up 으로 수행하는거지.
아래 Figure 8.10 처럼.
updates 가 이 방식으로 정렬되고 tabular representation 이 사용되면,
depth-first heuristic search 에서와 같은 overall update 를 얻을 수 있다.
이런 방식으로, 모든 state-space search 는 많은 individual one-step updates 를 연결한 것으로 볼 수 있다.
그래서, deeper searches 로 관측된 성능향상은
multistep updates 사용으로 인한 게 아니다.
현재 state 에서 바로 immediately downstream 한 states 와 actions 의 updates 에 대한 집중 때문이다.
decision-time planning 은 후보 actions 과 관련된 계산에 투자해 unfocused updates 에 의존해 나오는 것보다 더 좋은 decisions 을 만들 수 있다.

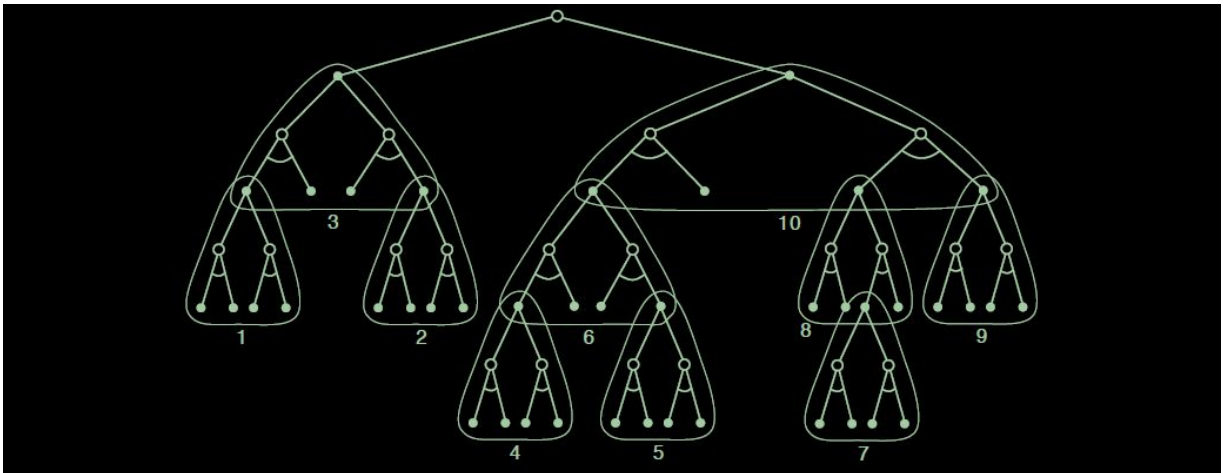


Figure 8.10: Heuristic search can be implemented as a sequence of one-step updates (shown here outlined) backing up values from the leaf nodes toward the root. The ordering shown here is for a selective depth-first search.

Figure 8.10: heuristic search 는 leaf nodes 에서 root 로 values를 back up 하는 one-step updates 의 sequences 로 적용할 수 있다.
여기서의 순서는 selective depth-first search 의 경우다.

8.10. Rollout Algorithms

rollout 알고리즘은
전부 현재 environment state 에서 시작하는 simulated trajectories 에 적용된 Monte Carlo 에 기반한 decision-time planning 알고리즘이다.
각 가능한 action 으로 시작하는 많은 simulated trajectories 의 returns 을 평균해서
주어진 policy 에 대한 action values 를 estimate 하고
주어진 policy 를 따른다.
action-value estimates 가 충분히 정확하다고 여겨질 때,
가장 높은 estimated value 를 가진 action (또는 actions 중 하나) 이 실행되고
그 후 resulting next state 에서 프로세스가 다시 시행된다.
rollout 이라는 말은,
랜덤하게 생성된 주사위의 sequences 로 게임 끝까지 여러 번 position 을
플레이해 보고 position 의 value 를 estimate 하는 데서 나왔다.
플레이어의 moves 는 주어진 policy 를 따르고.

챕터 5의 Monte Carlo 와는 다르게,
rollout 알고리즘의 목적은 주어진 policy 에서 완전한 optimal action-value function q_* 또는 action-value function q_π를 estimate 하는 게 아니다.
대신, 각 현재 state 와 주어진 policy 에 대한 action values 의 Monte Carlo estimates 를 만드는 걸 *rollout policy* 라고 한다.
decision-time planning 알고리즘처럼,
rollout 알고리즘은 이들 action-value estimates 를 즉시 사용하고 버린다.
이게 rollout 알고리즘을 상대적으로 시행하기 간단하게 만든다.
왜냐하면 모든 state-action pair 에 대해 outcomes 을 샘플링할 필요 없고
state space 나 state-action space 에 대한 function 을 approximate 할 필요가 없기 때문.

그럼 rollout 알고리즘이 달성하는 건 뭔가?
section 4.2 의 policy improvement theorem 말했던 건,
주어진 두 policies π, π' 가 identical 인데 어떤 state s에 대해서만 $\pi'(s) = a \neq \pi(s)$ 일 때,
$q_\pi(s, a) \geq v_\pi(s)$ 이면, policy π' 가 π보다 같거나 더 좋다.
이거다.
inequality 가 strict 하면 더 좋은거고.
이게 rollout 알고리즘에 적용된다.
s가 current state 고 π 가 rollout policy 인 거지.
simulated trajectories 의 return 을 평균하는 건
각 action $a' \in \mathcal{A}(s)$ 에 대해 $q_\pi(s, a')$ 의 estimates 를 만든다.
그런 다음 s 에서 이들 estimates 를 maximize 하는 action 을 선택하고 그 후에 π를 따르는 policy는 π 를 improve 하는 good candidate다.
결과는 dynamic programming 의 policy-iteration 알고리즘의 one step 과 비슷하다.
section 4.3 에서 얘기한 거지.
4.5의 asynchronous value iteration 의 one step 과 더 비슷하다.
현재 state 에 대한 action 만 바꾸거든.

다른 말로, rollout 알고리즘의 목적은 default policy 을 improve 하는거다.
optimal policy 를 찾는 게 아니라.
rollout 은 매우 효과적이다.
rollout policy 가 완전 랜덤이어도 성능이 좋을 수 있다.
하지만 확실히, improved policy 의 성능은
rollout policy 의 성능과 Monte Carlo value estimates 의 정확도에 달려있다.
rollout policy 가 더 좋아지고 value estimates 가 더 정확해질수록
rollout 알고리즘으로 나온 policy 가 더 좋아질 가능성이 높다.

이건 중요한 tradeoffs 를 안고 있다.
더 좋은 rollout policies 는 보통
좋은 value estimates 를 얻기 위해 충분한 trajectories 를 simulate 할 시간을 필요로 한다는 뜻이다.
decision-time planning methods 처럼,
rollout 알고리즘도 시간 제한을 엄격하게 한다.
rollout 알고리즘에 필요한 계산 시간은
각 decision 에서 evaluate 되어야 하는 actions 의 수와
유용한 sample returns 을 얻는 데 필요한 simulated trajectories 의 time steps 수,
rollout policy 가 결정을 내리는 데 걸리는 시간,
좋은 Monte Carlo action-value estimates 를 얻기에 필요한 simulated trajectories 수
에 달려 있다.

이들을 balancing 하는 게 중요하다.
Monte Carlo trials 이 다른 애들과 독립적이기 때문에
많은 trials 을 병렬적으로 돌리는 게 가능하다.
또다른 요령은 complete episodes 가 부족한 simulated trajectories 를 짧게 자르고,
저장된 evaluation function 으로 잘린 returns을 수정하는 방법이다.
(이전 챕터에서 truncated returns 과 updates 에 대해 얘기한 내용이 다 적용된다.)
Monte Carlo simulations 을 monitor 해서
best가 아닐 것 같은 actions 이나
현재의 best에 가까워서 선택해도 별 차이가 없는 actions 은

잘라내는 것도 가능하다.
(이러면 병렬 처리는 힘들겠지만)

우린 보통 rollout 알고리즘을 learning 알고리즘이라 생각하지 않는다.
values 나 policies 의 long-term memories 를 유지하지 않기 때문이다.
하지만, 이 알고리즘들은 강화학습의 특징을 잘 이용한다.
Monte Carlo control 에서, sample trajectories 의 모음의 returns 을 평균해서 action values 를 estimate 한다.
이 경우에 environment 의 sample model 과 simulated interactions
의 상호작용의 trajectories.
이렇게 trajectory sampling 으로 dynamic programming 의 exhaustive sweeps 를 피하는 것,
expected updates 대신 sample 에 의존해 distribution models 에 대한 필요를 없애는 면에서
강화학습 알고리즘과 닮았다.
마지막으로, rollout 알고리즘은
estimated action values 에 대해 greedy 하게 행동해 policy improvement property 를
활용한다.

8.11 Monte Carlo Tree Search(MCTS)

MCTS 는 decision-time planning 의 최근 놀라운 성공 사례다.
기본적으로 MCTS 는 rollout 알고리즘이지만,
더 높은 reward를 받는 trajectories 를 향하는 simulations 을 연속적으로 direct 하기 위해
Monte Carlo simulations 로 얻은 value estimates 를 accumulate 해서 향상시켰다.
MCTS는 컴퓨터 바둑을 grandmaster level 까지 올려놨다. 6단 정도? 2015년.
2005년엔 엄청 못하는 아마추어였는데..
기본 알고리즘의 많은 변형이 개발됐고 section 16.6 에서 볼 게 아주 critical 했어. 이세돌을 이기는 데.
MCTS 는 competitive setting 에 효과적이라고 증명됐어.
하지만 games 뿐만 아니라,
fast multistep simulation 에 충분할만큼 간단한 environment model 이 있는 경우
single-agent 의 sequential decision problems 에 효과적일 수 있다.

MCTS는 agent 의 action 을 선택할 새로운 state 를 만난후에 실행된다.
다음 state 에 대한 action 을 선택할 때 다시 실행되고.
계속 그런 식.
rollout 알고리즘에서처럼,
각 실행이 현재 state에서 terminal state 로 향하는(또는 discount 로 reward 가 return 에 미치는 영향을 무시할 수 있을 때까지) 많은 trajectories 를 simulate 하는 iterative process 다.
MCTS 의 core idea 는 현재 state에서 시작해 multiple simulations 에 연속적으로 focus 하는거다. earlier simulations 에서 high evaluations 을 받은 trajectories 의 initial portions 을 확장함으로써.
MCTS 는 한 action selection 에서 다음 선택까지 approximate value functions 이나 policies 를 유지할 필요가 없다.
하지만 많은 implementations 에서 다음 execution 에 유용할 것 같은 선택된 actions values 를 유지한다.

대부분의 경우, simulated trajectories 안의 actions 은

보통 rollout policy 라 불리는 간단한 policy 를 이용해 생성된다.
as it is for simpler rollout algorithms ?
rollout policy 와 model 둘 다 많은 계산을 필요로 하지 않을 때,
많은 simulated trajectories 가 짧은 시간에 생성될 수 있다.
모든 tabular Monte Carlo method 에서처럼,
state-action pair 의 value 는 그 pair 로부터의 (simulated) returns 의 평균으로 estimate 된다.

몇 steps 안에 도달할 가능성이 가장 큰 state-action pairs 의 subset 에 대해서만
Monte Carlo value estimates 를 유지한다.
그렇게 현재 state에 뿌리를 둔 트리를 형성하고. Fig 8.11 처럼.

MCTS 는 incrementally 트리를 확장한다.
simulated trajectories 의 결과에 기반해
유망해보이는 states 를 나타내는 nodes를 더하면서.
트리 바깥과 leaf nodes 에서 rollout policy 가 action selections 을 하는 데 쓰인다.
트리 안의 states 에서는 더 좋은 게 가능하다.
이들 states에 대해 최소한 몇 개의 actions 에 대한 value estimates 를 갖고 있고
그래서 informed policy 를 이용해 그들 중 하나를 고를 수 있다.
이걸 *tree policy* 라고 부르고, exploration 과 exploitation 을 balance 함.
예를 들어, tree policy 가 ε-greedy 나 UCB selection rule 을 이용해 actions 을 선택할 수 있다.

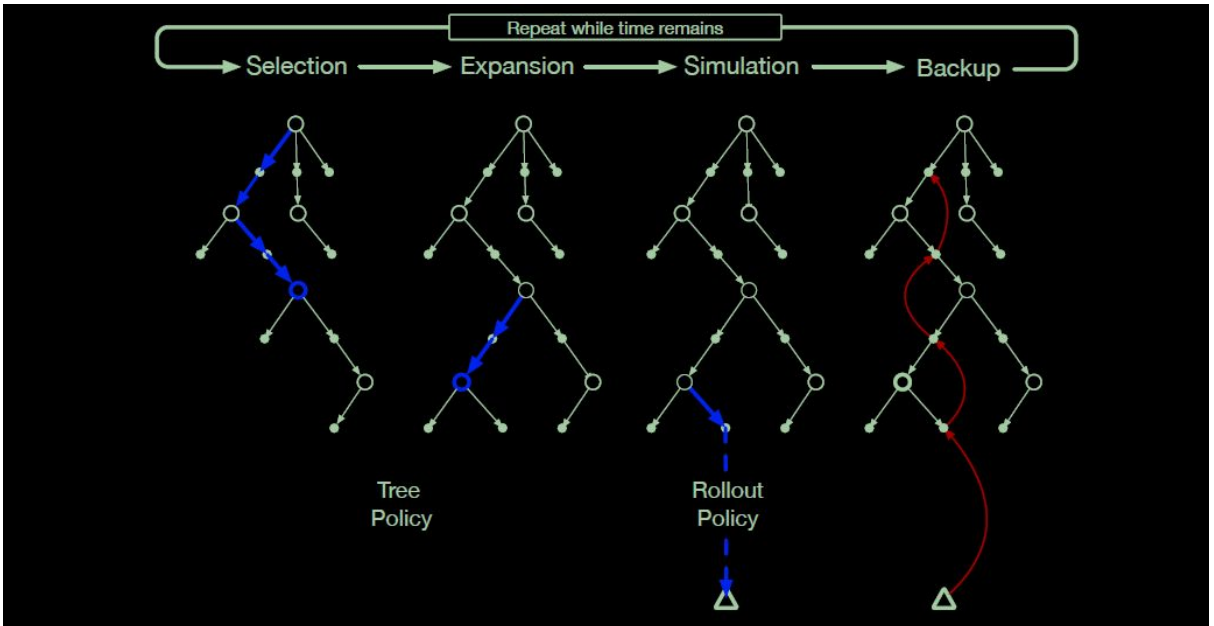


Figure 8.11: Monte Carlo Tree Search. When the environment changes to a new state, MCTS executes as many iterations as possible before an action needs to be selected, incrementally building a tree whose root node represents the current state. Each iteration consists of the four operations **Selection**, **Expansion** (though possibly skipped on some iterations), **Simulation**, and **Backup**, as explained in the text and illustrated by the bold arrows in the trees. Adapted from Chaslot, Bakkes, Szita, and Spronck (2008).

Figure 8.11: Monte Carlo Tree Search
environment 가 새로운 state로 바뀌면, action 이 선택되어야 하기 전에 MCTS 를 가능한
많은 iterations 실행한다. root node 가 현재 state를 나타내는 트리를 incrementally 만들면서.
각 iteration 은 네 가지 operations 이 있다.
Selection, Expansion(iteration 에 따라 생략 가능), Simulation, Backup.

더 구체적으로, 기본적인 MCTS의 각 iteration 을 보자.

1. Selection: root node 에서 시작해, 트리의 edges 에 붙은 actions values 에 기반한 tree policy가 트리를 가로질러 leaf node 를 선택한다.

2. Expansion: 어떤 iterations 에서, unexplored actions 를 선택한 node 에서 하나 이상의 child nodes 를 추가해 선택된 leaf node에서 트리가 확장된다.

3. Simulation: 선택된 node 에서, 또는 새로 추가된 child nodes 에서, rollout policy 로 선택된 actions 으로 complete episode 의 simulation 을 돌린다.
   처음엔 tree policy 에 의해 선택된 actions 과
   트리를 넘어가선 rollout policy 로 선택된 actions 으로
   Monte Carlo trial 이 나온다.

4. Backup: simulated episode 로 생성된 return 은,
   MCTS의 해당 iteration 에서 tree policy 에 의해 지나온 트리의 edges 에 연결된 action values 를 update 혹은 initialize 하기 위해 back up 된다.
   Fig 8.11 이 보여주고 있다.
   simulated trajectory 의 terminal state 로부터 바로
   rollout policy 가 시작하는 트리의 state-action node로의 backup을.
   (일반적으론, simulated trajectory 의 전체 return이 이 state- action node 로 backup 된다.

MCTS 는 매 time 트리의 root node 에서 시작해 더 이상 time 이 없을 때까지, 혹은 다른 computational resource 가 없어질 때까지 네 steps 을 계속 실행한다.
그러고는, 마지막으로, 트리에서 accumulated statistics 를 이용한 어떤 메커니즘에 따라 현재 state를 나타내는 root node 로부터 action 이 선택된다.
예를 들어, root state 에서 가능한 모든 actions 중 가장 큰 action value 를 갖는 애를 고를 수 있겠지.
outliers 를 피하기 위해 가장 큰 visit count 도 가능할거야.
이게 MCTS 가 실제로 선택하는 action 이야.
environment 가 새로운 state로 전환되면, MCTS 가 다시 돌아간다.
가끔 새로운 state 를 나타내는 single root node 의 트리에서 시작하기도 하지만
주로 MCTS 의 이전 실행으로 만들어진 트리에서 남은 node 의 후손을 포함한 트리로 시작한다.
모든 다른 nodes 는 버려지고, 관련된 action values 도 마찬가지로 버린다.

MCTS 는 처음에 바둑처럼 두 사람이 경쟁하는 게임을 하는 프로그램에서
움직임을 선택하는 걸 제안하도록 고안됐다.
game playing 에서, 각 simulated episode 가 one complete play of the game 인데
두 player 모두 tree와 rollout policies 에 의해 actions 을 선택하는 game 이다.
Section 16.6 이 알파고에서 쓰인 MCTS의 확장을 다루고 있다.
self-play 강화학습을 통한 deep ANN 으로 학습한 MCTS 의 Monte Carlo evaluations 을 합한 거다.

기본적으로, MCTS 가 decision-time planning 알고리즘이며
root state 부터 시작하는 simulations 에 적용된 Monte Carlo control 에 기반한다.
그러니까, rollout 알고리즘의 한 종류라는거다.

그러므로 online, incremental, sample-based value estimation 과 policy improvement 의 이점이 있다.
이걸 넘어서, tree edges 에 연결된 action-value estimates를 저장하고
강화학습의 sample updates 를 이용해 그들을 update 한다.
이건 이전에 simulate 된 high-return trajectories 에 공통인 initial segments 를 갖는 trajectories 에 Monte Carlo trials 를 집중하는 효과가 있다.
게다가, 트리를 incrementally 확장함으로써,
MCTS 가 효과적으로 lookup table 을 키워 partial action-value function 을 저장할 수 있다.
high-yielding sample trajectories 의 initial segments 에서 visit 된 state-action pairs 의 estimated values 에 메모리가 할당된다.
그래서 MCTS는 past experience로 exploration 을 guide 하는 이점을 갖는 동안
action-value function 을 globally approximate 하는 문제를 피할 수 있다.

8.12 Summary of the Chapter

planning 은 environment 의 model 이 필요하다.
distribution model 은 next states 와 가능한 actions 에 대한 rewards 의 확률로 이루어져 있다.
sample model 은 이 확률들에 따라 생성된 single transitions 과 rewards 를 만든다.
Dynamic programming 은 distribution model 이 필요하다.
expected updates 를 쓰기 때문이지. 모든 가능한 next states와 rewards 에 기대값을 계산하는 거 말이야.
반면, sample model은 sample updates 를 쓸 수 있는 environment 와의 상호작용을 simulate 하는 데 필요한 거다.
sample models 이 일반적으로 distribution models 보다 얻기 쉽다.

우린 planning optimal behavior 와 learning optimal behavior 사이의 긴밀한 관계에 초점을 맞춰봤다.
둘 다 같은 value functions 을 estimate 하는 게 연관돼 있고,
incrementally 하게 estimates 를 update 하는 게 자연스럽다.
그리고 작은 backing-up operations 이 길게 이어진다.
이게 learning 과 planning 프로세스를 통합할 수 있게 해 준다.
같은 estimated value function 을 update 하게 함으로써.
거기다가, 모든 learning methods 를 planning methods 로 간단히 바꿀 수 있다.
어떻게?
real experience 대신 simulated(model-generated) experience 를 적용하면 돼.
이 경우에 learning 과 planning 은 더 비슷해진다.
아마 다른 experience 의 source 에 작동하는 identical 알고리즘이다.

incremental planning method s를 acting 과 model-learning 과 통합하는 건 간단하다.
planning, acting, model-learning 은 circular fashion 으로 상호작용한다.(fig 8.1)
각자가 다른 애가 improve 해야하는 걸 만들어낸다.
그들 사이에 다른 interaction 은 필요도 제한도 되지 않는다.
가장 자연스러운 approach 는 모든 프로세스에 대해 asynchronously, parallel 하게 진행하는 거다.
프로세스가 computational resources 를 공유해야하면,
알아서 쪼개서 하면 된다.

이번 챕터에서 state-space planning methods 간의 다양한 dimensions 에 대해 다뤘다.
한 dimension 은 updates 의 크기에서의 variation 이다.
updates 가 작을수록 더 많은 incremental planning methods 가 가능하다.
가장 작은 updates 에는 one-step sample updates 가 있다. Dyna 에서처럼.

다른 중요한 dimension 은 updates 의 distribution 이다.
즉, search 의 초점.
prioritized sweeping 은 values 가 최근에 바뀐 states 의 predecessors 에 backward 로 초점을 둔다.
on-policy trajectory sampling 은 agent 가 environment 를 control 할 때 만날 확률이 높은 states 나 state-action pairs 에 초점을 둔다.
이게 prediction 이나 control problem 에 무관한 state space 의 부분을 계산상 skip 하게 해준다.
real-time dynamic programming, value iteration 의 on-policy trajectory sampling 버전은 이러한 전략이 conventional sweep-based policy iteration 에 비해 가지는 장점을 잘 묘사해 준다.

또한 planning 은 관련있는 states 로부터 forward 로 초점을 맞춘다.
agent-environment interaction 동안 실제로 만난 states 같은 애들.
이것의 가장 중요한 형태는 planning 이 decision time 에 수행됐을 때,
그러니까 action-selection process 의 부분으로 수행될 때다.
인공지능 분야에서 연구된 classical heuristic search 가 이것의 한 예다.
다른 예는 rollout 알고리즘과 Monte Carlo Tree Search 이다.
online, incremental, sample-based value estimation 과 policy improvement 에 이점이 있었지.

8.13 Summary of Part 1: Dimensions

이 챕터는 책의 part 1 마지막이다.
여기서 강화학습을 개별 methods 들의 collection 으로가 아니라
methods 를 가로지르는 coherent set 으로 소개하려 했단다.
각 idea 가 methods 가 다른, 하나의 dimension 으로 볼 수 있어.
그 dimensions 집합은 가능한 methods 들의 큰 공간에 걸쳐 있어.(span)
이 sections 에서, 책에서 소개된 강화학습의 관점을 요약하기 위해 methods space에서 dimensions 개념을 사용해. 뭔가 감동적

이 책에서 본 모든 methods 는 공통적인 세 가지 key ideas 가 있어.
첫 째, value functions 을 estimate 하려고 한다.
둘 째, 실제 혹은 가능한 state trajectories 를 따라 values 를 back up 해 작동한다.
셋 째, generalized policy iteration(GPI) 의 일반적인 전략을 따른다. 그러니까, approximate value function 과 approximate policy 를 유지하고 서로에 기반해 계속 improve 하려고 한다.

이들 세 ideas 가 이 책에서 커버한 주제의 핵심이야.
value functions, backing up value updates, GPI 가 아주 강력한 organizing principles 이야.
artificial 이든 natural 이든 모든 지능모델에 쓸 수 있을걸.

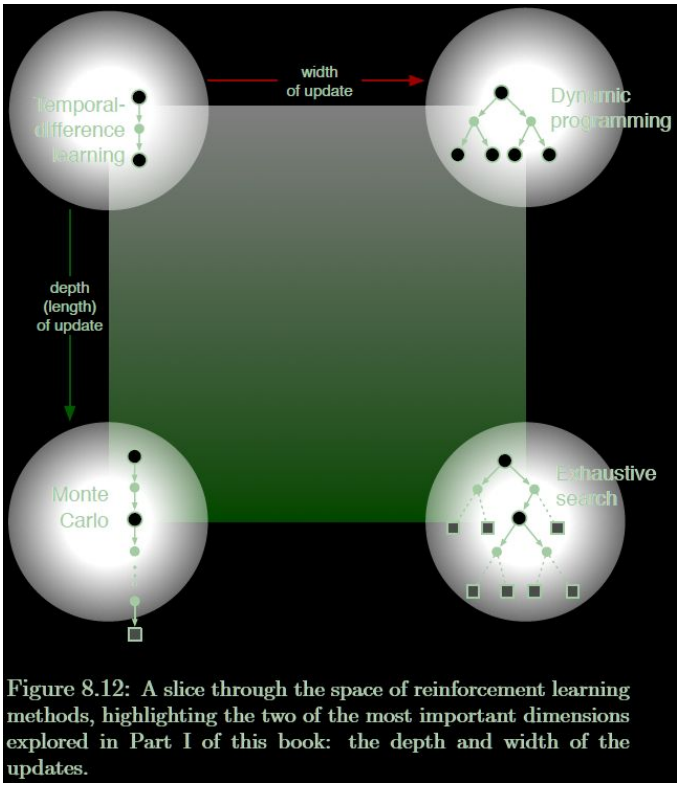


Figure 8.12: A slice through the space of reinforcement learning methods, highlighting the two of the most important dimensions explored in Part I of this book: the depth and width of the updates.

대충 depth 랑 width 가 달라진다는 뜻.

Figure 8.12 에 methods 들이 달라지는 가장 중요한 dimensions 두 개가 나와 있다.
이들 dimensions 은 value function 을 improve 하는 데 쓰는 update 의 종류와 관련이 있다.

수평 dimension 은
sample updates(sample trajectory에 기반한) 인지
expected updates(possible trajectories 의 distribution 에 기반한) 인지
나와 있다.
Expected updates 는 distribution model 이 필요한 반면
sample updates 는 sample model 만 필요하다. 또는 model 없이 actual experience 로도 수행 가능하다.(variation 의 또다른 dimension이지)

수직 dimension 은 updates 의 depth 와 대응한다.
bootstrapping 의 degree 말이지.
네 코너 중 세 코너가 values 를 estimate 하는 주요 세 가지 methods 들이다.
DP, TD, MC.
왼쪽 애들은 sample-update methods 다.
one-step TD updates 부터 full-return Monte Carlo updates 까지.
이 사이의 spectrum 은 n-step updates 에 기반한 methods 들을 포함한다.
챕터 12에서 이걸 eligibility traces 로 구현되는 λ-updates 같은
n-step updates 의 mixtures 로 확장한다.

DP methods 는 오른쪽 위 코너 끝에 있다.
one-step expected updates 를 포함하거든.
오른쪽 아래 코너는 expected updates 의 극단적인 case다.
너무 깊어서 terminal states 까지 달리는거야.

(continuing task 에선 discounting 이 깎여서 더이상 rewards 에 대한 영향을 무시할 정도까지)
이건 exhaustive search 의 case다.
이 dimension 의 중간 methods 는
heuristic search 와,
search 와 update 를 제한된 depth 까지 선택적으로 하는 관련 methods
를 포함한다.

수평 dimension 에서도 중간 methods 가 있다.
expected 와 sample updates 를 섞는 methods와,
single update 내에서 samples 과 distributions을 섞는 methods 다.
가운데 공간엔 모든 이런 중간적인 methods 를 나타낸다.

세 번째 dimension 은 on-policy 와 off-policy methods 간의 binary distinction 이다.
on-policy에선, agent 가 현재 따르고 있는 policy 에 대한 value function 을 학습하고,
off-policy에선, 현재 best 라 생각하는 다른 policy 에 대한 policy의 value function 을
학습한다.
behavior 를 생성하는 policy 는 보통 현재 best 라 생각되는 것과 다른데,
이는 explore 를 위해서다.
이 세 번째 dimension 은 Fig 8.12 에서 평면에 수직으로 나타낼 수 있다.

세 차원들에 더해, 다른 몇가지를 밝혀봤다.

Definition of return:
task 가
episodic 이니 continuing 이니,
discounted 니 undiscounted 니?

Action values vs. state values vs. afterstate values:
어떤 values 가 estimate 되어야 하나?
state values 만 estimate 되면,
action selection 을 위해 model 이든 separate policy든 필요할 거다.

Action selection / exploration:
어떻게 exploration 과 exploitation 간의 적절한 trade-off 를 보장하며 actions 을 선택할까?
우린 가장 간단한 방법만 봤다. ε-greedy, optimistic initialization of values, softmax,upper confidence bound.

Synchronous vs. asynchronous:
모든 states 에 대한 updates 가 동시에 수행되니?
아니면 특정 순서에 따라 하나씩 하니?

Real vs. simulated:
real experience 에 기반해서 update 해야할까?
아니면 simulated experience?
둘 다라면, 각각은 얼만큼?

Location of updates:
어떤 states 나 state-action pairs 가 update 되어야 할까?

model-free methods 는 실제로 만난 states 나 state-action pairs 중에서만 고를 수 있다.
model-based methods 는 임의로 고를 수 있지. 여긴 많은 가능성이 있어.

Timing of updates
updates 가 actions 을 선택하는 것의 부분으로 수행돼야할까?
아니면 그 다음에만?

Memory for updates
update 된 values 를 얼마나 오래 갖고 있어야할까?
계속 갖고 있어야할까?
아니면 heuristic search 에서처럼 action selection 을 계산하는 동안에만?

물론 이들 dimensions 이 exhaustive 하지도, mutually exclusive 하지도 않다.
개별 알고리즘은 다양하게 다르다.
예를 들어, Dyna methods 는 같은 value function 에 영향을 미치기 위해 real 과 simulated experience 를 둘 다 쓴다.
서로 다른 방식으로, 혹은 다른 state & action representations 으로 계산된 multiple value functions 을 유지하는 것도 합리적인 방법이다.
하지만, 이들 dimensions 은 많은 가능한 methods 를 설명하고 탐색하는 일관성 있는 idea set 을 구성한다.

아직 나오지 않은, 가장 중요한 dimension 은 function approximation 이다.
function approximation 은
한 극단의 tabular methods 에서, state aggregation, 선형, 비선형 methods 들에 이르기까지 다양한 가능성의 orthogonal spectrum 으로 볼 수 있다.
이 dimension 은 파트 2에서 탐험한다.

$\alpha\gamma\varepsilon\varrho\lambda\delta\tau\phi\sigma\eta\kappa$
$\pi(a|s)\ b \neq \pi$
$v_\pi(s)\ Q(s,\ a)$
$q_\pi(s,\ a)$
$ACGQRSTVWXabhfijkptnrx$
$\mathscr{A}\mathscr{P}\mathscr{T}$
$\ni\in\geq\leq\neq\doteq\Leftrightarrow\longleftarrow\ s\in\mathscr{S},\ a\in\mathscr{A}(s)\mathscr{R}\ \sum\sqrt{}$
$\blacksquare'\cdot$