서강대학교
SOGANG UNIVERSITY

강화 학습 스터디(1)

Reinforcement learning and Rubik's cube

서강대학교 수학과 김건휘

목차

- **Solving the Rubik's cube with deep reinforcement learning and search 논문 소개**
- Reinforcement learning in this paper
- Discussion

논문 소개

nature machine intelligence

ARTICLES

https://doi.org/10.1038/s42256-019-0070-z

Solving the Rubik's cube with deep reinforcement learning and search

Forest Agostinelli[1,3], Stephen McAleer[2,3], Alexander Shmakov[1,3] and Pierre Baldi[1,2*]

The Rubik's cube is a prototypical combinatorial puzzle that has a large state space with a single goal state. The goal state is unlikely to be accessed using sequences of randomly generated moves, posing unique challenges for machine learning. We solve the Rubik's cube with DeepCubeA, a deep reinforcement learning approach that learns how to solve increasingly difficult states in reverse from the goal state without any specific domain knowledge. DeepCubeA solves 100% of all test configurations, finding a shortest path to the goal state 60.3% of the time. DeepCubeA generalizes to other combinatorial puzzles and is able to solve the 15 puzzle, 24 puzzle, 35 puzzle, 48 puzzle, Lights Out and Sokoban, finding a shortest path in the majority of verifiable cases.

The Rubik's cube is a classic combinatorial puzzle that poses unique and interesting challenges for artificial intelligence and machine learning. Although the state space is exceptionally large (4.3×10^{19} different states), there is only one goal state. Furthermore, the Rubik's cube is a single-player game and a sequence of random moves, no matter how long, is unlikely to end in the goal state. Developing machine learning algorithms to deal with this property of the Rubik's cube might provide insights into learning to solve planning problems with large state spaces. Although machine learning methods have previously been applied to the Rubik's cube, these methods have either failed to reliably solve the cube[1–4] or have had to rely on specific domain knowledge[5,6]. Outside of machine learning methods, methods based on pattern databases (PDBs) have been effective at solving puzzles such as the Rubik's cube, the 15 puzzle and the 24 puzzle[7,8], but these methods can be memory-intensive and puzzle-specific.

More broadly, a major goal in artificial intelligence is to create algorithms that are able to learn how to master various environments without relying on domain-specific human knowledge. The classical $3\times3\times3$ Rubik's cube is only one representative of a larger family of possible environments that broadly share the characteristics described above, including (1) cubes with longer edges or higher dimension (for example, $4\times4\times4$ or $2\times2\times2\times2$), (2) sliding tile puzzles (for example the 15 puzzle, 24 puzzle, 35 puzzle and 48 puzzle), (3) Lights Out and (4) Sokoban. As the size and dimensions are increased, the complexity of the underlying combinatorial problems rapidly increases. For example, while finding an optimal solution to the 15 puzzle takes less than a second on a modern-day desktop, finding an optimal solution to the 24 puzzle can take days, and finding an optimal solution to the 35 puzzle is generally intractable[9]. Not only are the aforementioned puzzles relevant as mathematical games, but they can also be used to test planning algorithms[10] and to assess how well a machine learning approach may generalize to different environments. Furthermore, because the operation of the Rubik's cube and other combinatorial puzzles are deeply rooted in group theory, these puzzles also raise broader questions about the application of machine learning methods to complex symbolic systems, including mathematics. In short, for all these reasons, the Rubik's cube poses interesting challenges for machine learning.

To address these challenges, we have developed DeepCubeA, which combines deep learning[11,12] with classical reinforcement learning[13] (approximate value iteration[14–16]) and path finding methods (weighted A* search[17,18]). DeepCubeA is able to solve combinatorial puzzles such as the Rubik's cube, 15 puzzle, 24 puzzle, 35 puzzle, 48 puzzle, Lights Out and Sokoban (Fig. 1). DeepCubeA works by using approximate value iteration to train a deep neural network (DNN) to approximate a function that outputs the cost to reach the goal (also known as the cost-to-go function). Given that random play is unlikely to end in the goal state, DeepCubeA trains on states obtained by starting from the goal state and randomly taking moves in reverse. After training, the learned cost-to-go function is used as a heuristic to solve the puzzles using a weighted A* search[17–19].

DeepCubeA builds on DeepCube[20], a deep reinforcement learning algorithm that solves the Rubik's cube using a policy and value function combined with Monte Carlo tree search (MCTS). MCTS, combined with a policy and value function, is also used by AlphaZero, which learns to beat the best existing programs in chess, Go and shogi[21]. In practice, we find that, for combinatorial puzzles, MCTS has relatively long runtimes and often produces solutions many moves longer than the length of a shortest path. In contrast, DeepCubeA finds a shortest path to the goal for puzzles for which a shortest path is computationally verifiable: 60.3% of the time for the Rubik's cube and over 90% of the time for the 15 puzzle, 24 puzzle and Lights Out.

Deep approximate value iteration

Value iteration[15] is a dynamic programming algorithm[14,16] that iteratively improves a cost-to-go function J. In traditional value iteration, J takes the form of a lookup table where the cost-to-go $J(s)$ is stored in a table for all possible states s. However, this lookup table representation becomes infeasible for combinatorial puzzles with large state spaces like the Rubik's cube. Therefore, we turn to approximate value iteration[16], where J is represented by a parameterized function implemented by a DNN. The DNN is trained to minimize the mean squared error between its estimation of the cost-to-go of state s, $J(s)$, and the updated cost-to-go estimation $J'(s)$:

[1]Department of Computer Science, University of California Irvine, Irvine, CA, USA. [2]Department of Statistics, University of California Irvine, Irvine, CA, USA. [3]These authors contributed equally: Forest Agostinelli, Stephen McAleer, Alexander Shmakov. *e-mail: pfbaldi@uci.edu

- 대상 : Rubik's cube
- 방법 : Reinforcement learning
  + Deep Neural Network
  + Search

논문 소개

nature machine intelligence

ARTICLES

https://doi.org/10.1038/s42256-019-0070-z

Solving the Rubik's cube with deep reinforcement learning and search

Forest Agostinelli[1,3], Stephen McAleer[2,3], Alexander Shmakov[1,3] and Pierre Baldi[1,2*]

The Rubik's cube is a prototypical combinatorial puzzle that has a large state space with a single goal state. The goal state is unlikely to be accessed using sequences of randomly generated moves, posing unique challenges for machine learning. We solve the Rubik's cube with DeepCubeA, a deep reinforcement learning approach that learns how to solve increasingly difficult states in reverse from the goal state without any specific domain knowledge. DeepCubeA solves 100% of all test configurations, finding a shortest path to the goal state 60.3% of the time. DeepCubeA generalizes to other combinatorial puzzles and is able to solve the 15 puzzle, 24 puzzle, 35 puzzle, 48 puzzle, Lights Out and Sokoban, finding a shortest path in the majority of verifiable cases.

The Rubik's cube is a classic combinatorial puzzle that poses unique and interesting challenges for artificial intelligence and machine learning. Although the state space is exceptionally large (4.3×10^{19} different states), there is only one goal state. Furthermore, the Rubik's cube is a single-player game and a sequence of random moves, no matter how long, is unlikely to end in the goal state. Developing machine learning algorithms to deal with this property of the Rubik's cube might provide insights into learning to solve planning problems with large state spaces. Although machine learning methods have previously been applied to the Rubik's cube, these methods have either failed to reliably solve the cube[1–4] or have had to rely on specific domain knowledge[5,6]. Outside of machine learning methods, methods based on pattern databases (PDBs) have been effective at solving puzzles such as the Rubik's cube, the 15 puzzle and the 24 puzzle[7,8], but these methods can be memory-intensive and puzzle-specific.

More broadly, a major goal in artificial intelligence is to create algorithms that are able to learn how to master various environments without relying on domain-specific human knowledge. The classical $3\times3\times3$ Rubik's cube is only one representative of a larger family of possible environments that broadly share the characteristics described above, including (1) cubes with longer edges or higher dimension (for example, $4\times4\times4$ or $2\times2\times2\times2$), (2) sliding tile puzzles (for example the 15 puzzle, 24 puzzle, 35 puzzle and 48 puzzle), (3) Lights Out and (4) Sokoban. As the size and dimensions are increased, the complexity of the underlying combinatorial problems rapidly increases. For example, while finding an optimal solution to the 15 puzzle takes less than a second on a modern-day desktop, finding an optimal solution to the 24 puzzle can take days, and finding an optimal solution to the 35 puzzle is generally intractable[9]. Not only are the aforementioned puzzles relevant as mathematical games, but they can also be used to test planning algorithms[10] and to assess how well a machine learning approach may generalize to different environments. Furthermore, because the operation of the Rubik's cube and other combinatorial puzzles are deeply rooted in group theory, these puzzles also raise broader questions about the application of machine learning methods to complex symbolic systems, including mathematics. In short, for all these reasons, the Rubik's cube poses interesting challenges for machine learning.

To address these challenges, we have developed DeepCubeA, which combines deep learning[11,12] with classical reinforcement learning[13] (approximate value iteration[14–16]) and path finding methods (weighted A* search[17,18]). DeepCubeA is able to solve combinatorial puzzles such as the Rubik's cube, 15 puzzle, 24 puzzle, 35 puzzle, 48 puzzle, Lights Out and Sokoban (Fig. 1). DeepCubeA works by using approximate value iteration to train a deep neural network (DNN) to approximate a function that outputs the cost to reach the goal (also known as the cost-to-go function). Given that random play is unlikely to end in the goal state, DeepCubeA trains on states obtained by starting from the goal state and randomly taking moves in reverse. After training, the learned cost-to-go function is used as a heuristic to solve the puzzles using a weighted A* search[17–19].

DeepCubeA builds on DeepCube[20], a deep reinforcement learning algorithm that solves the Rubik's cube using a policy and value function combined with Monte Carlo tree search (MCTS). MCTS, combined with a policy and value function, is also used by AlphaZero, which learns to beat the best existing programs in chess, Go and shogi[21]. In practice, we find that, for combinatorial puzzles, MCTS has relatively long runtimes and often produces solutions many moves longer than the length of a shortest path. In contrast, DeepCubeA finds a shortest path to the goal for puzzles for which a shortest path is computationally verifiable: 60.3% of the time for the Rubik's cube and over 90% of the time for the 15 puzzle, 24 puzzle and Lights Out.

Deep approximate value iteration

Value iteration[15] is a dynamic programming algorithm[14,16] that iteratively improves a cost-to-go function J. In traditional value iteration, J takes the form of a lookup table where the cost-to-go $J(s)$ is stored in a table for all possible states s. However, this lookup table representation becomes infeasible for combinatorial puzzles with large state spaces like the Rubik's cube. Therefore, we turn to approximate value iteration[16], where J is represented by a parameterized function implemented by a DNN. The DNN is trained to minimize the mean squared error between its estimation of the cost-to-go of state s, $J(s)$, and the updated cost-to-go estimation $J'(s)$:

[1]Department of Computer Science, University of California Irvine, Irvine, CA, USA. [2]Department of Statistics, University of California Irvine, Irvine, CA, USA. [3]These authors contributed equally: Forest Agostinelli, Stephen McAleer, Alexander Shmakov. *e-mail: pfbaldi@uci.edu

- 대상 : Rubik's cube
- 방법 : Reinforcement learning
  (Value iteration)
  + Deep Neural Network
  (Build function by adjusting parameters)
  + Search
  (A* search)

Rubik's cube

- 루빅스 큐브 (Rubik's Cube) 소개

ChatGPT

정의

- 1974년 헝가리 건축가 에르뇨 루빅(Ernő Rubik) 이 발명한 3D 퍼즐
- 6면이 각기 다른 색으로 이루어진 정육면체 회전 퍼즐

구성

- 3×3×3 구조 (기본형)
- 중심 조각(움직이지 않음), 모서리 조각, 꼭짓점 조각

목적

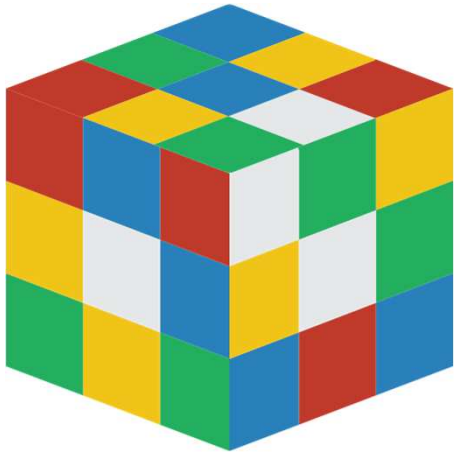
- 각 면을 하나의 색으로 맞추는 것
- 논리적 사고, 패턴 인식, 공간 지각 능력 향상

흥미로운 사실

- 최소 20회 이내의 움직임(신의 알고리즘)으로 해결 가능
- 세계기록: 3초대 해결 (스피드 큐빙)
- AI도 최적 해법 연구에 활용됨

Rubik's cube

- Rubik's cube의 구성과 표현



🧩 큐브렛(Cubelet) 구성
- Centre Cubelets (스티커 1개)
- Edge Cubelets (스티커 2개)
- Corner Cubelets (스티커 3개)

🎨 스티커와 상태 표현
- 스티커는 6가지 색상 (각 면의 색)
- 해결 상태(Goal State): 한 면의 모든 스티커가 동일한 색

⇒ 총 26개의 큐브렛 & 54개의 스티커로, 큐브렛의 스티커 조합은 유일

📣 DNN 입력 표현:
- One-hot Encoding 활용6(색) × 54(스티커) = 324차원 벡터로 표현

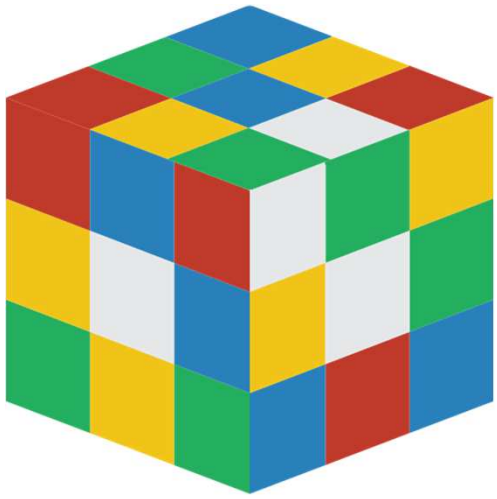
Rubik's cube

- Rubik's cube의 특징 (강화 학습 관점)

1. Exceptionally large state space
   (4.3×10^{19} different states)

2. Only one goal state

3. Sequence of random moves



논문 방법 소개

- 방법 요약

Rubik's cube를 해결할 때,
연쇄적으로 이어지는 상태 전이 구조를 탐색하는 접근을 취한다.

상태 공간이 매우 크지만, 목표까지 남은 이동 횟수를 예측하는 휴리스틱 정보가 있다면, **A* search** 알고리즘이 효과적으로 동작한다.

이 휴리스틱 정보를 강화 학습 분야의 **Value iteration**과 딥러닝 분야의 **DNN**을 결합해 학습된 모델에서 얻는다.

이렇게 학습된 함수를 A* search 알고리즘의 휴리스틱 정보로 사용함으로써, Rubik's cube의 거대한 상태공간에서도 높은 성능으로 해답을 찾을 수 있다.

Preliminary

- Basic idea

- Goal에 도달한 상태에서 시작해, 무작위 K번 섞어서 학습용 상태를 만든다.

- 'K' 라는 정보가 이 상태들이 실제로 목표 상태와 일정 거리 내에 있다는 사실을 주게 되어 네트워크가 학습하기 쉽다.

⇒ 이렇게 학습시키면, 어떤 상태가 주어졌을 때 몇 수 만 더 움직이면 Goal에 달성하는지를 근사치로 맞출 수 있다.

Preliminary

- A* search

- Dijkstra's algorithm의 변형으로,
start 와 end 사이의 최적 경로를 찾는 과정이다.

특이사항 : 각 노드로부터 end까지의 정보를 추가적으로 갖고 있을 때
사용할 수 있는 알고리즘

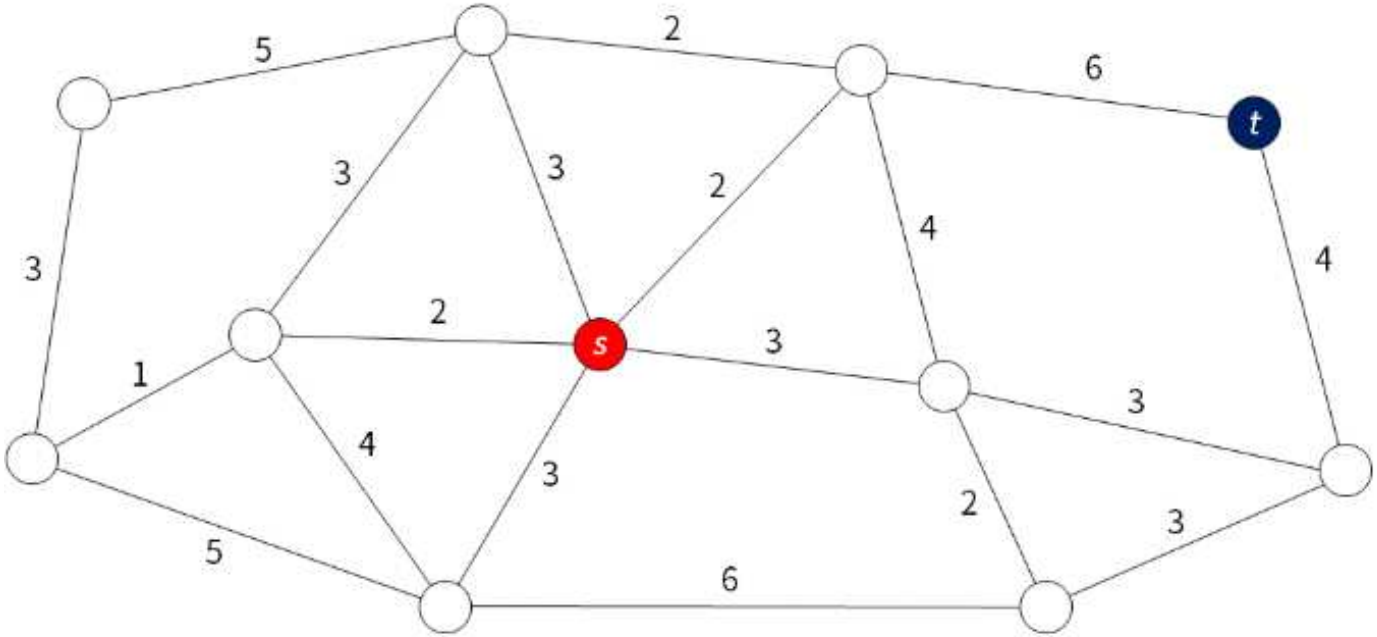
** 적절한 휴리스틱 정보란?
휴리스틱 h는 각 정점에서 목표 정점 t까지 남은 거리를 추정하는 함수로,
$h(v) \leq c(v, v') + h(v')$가 성립해야 한다. 이는 삼각 부등식과 동일한 관점에서 이해할 수 있다.

Preliminary

- A* search

- Dijkstra's algorithm의 변형으로,
start 와 end 사이의 최적 경로를 찾는 과정이다.

특이사항 : 각 노드로부터 end까지의 정보를 추가적으로 갖고 있을 때
사용할 수 있는 알고리즘

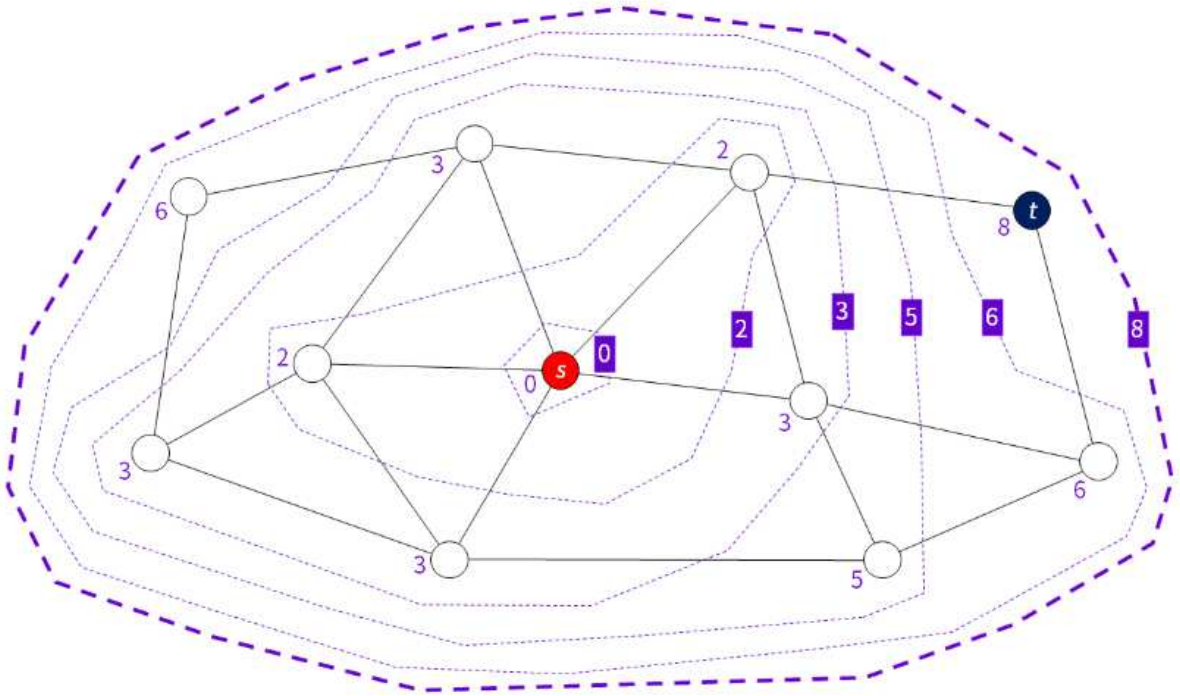


Preliminary

- A* search

- Dijkstra's algorithm의 변형으로,
start 와 end 사이의 최적 경로를 찾는 과정이다.

특이사항 : 각 노드로부터 end까지의 정보를 추가적으로 갖고 있을 때
사용할 수 있는 알고리즘

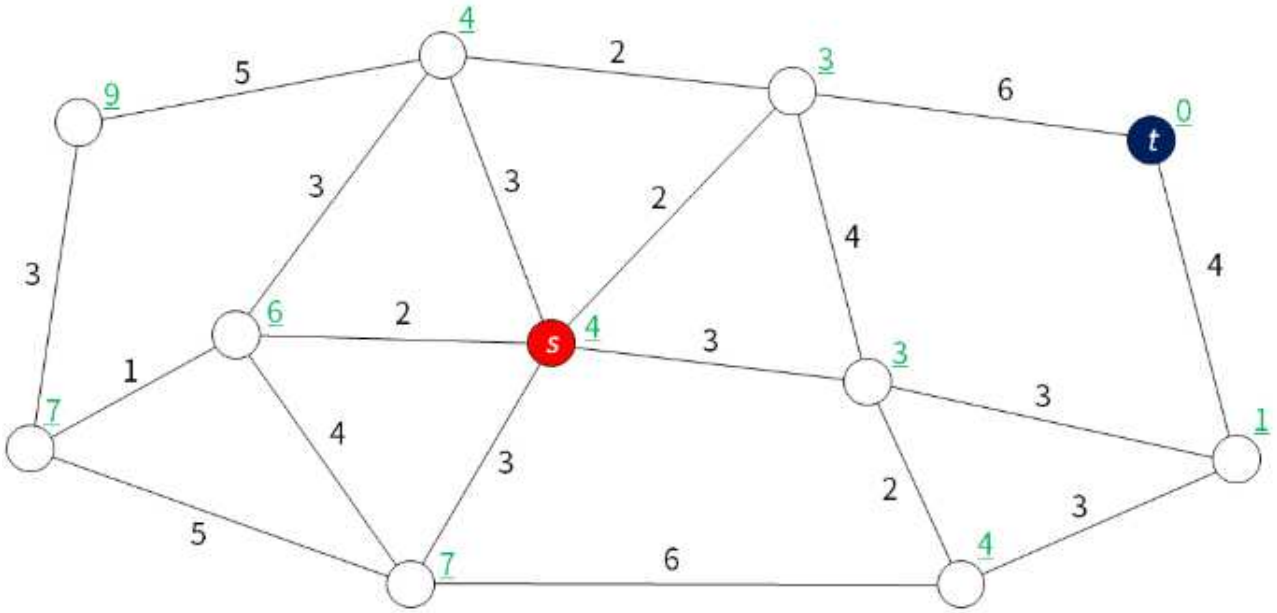


Preliminary

- A* search

- Dijkstra's algorithm의 변형으로,
start 와 end 사이의 최적 경로를 찾는 과정이다.

특이사항 : 각 노드로부터 end까지의 정보를 추가적으로 갖고 있을 때
사용할 수 있는 알고리즘

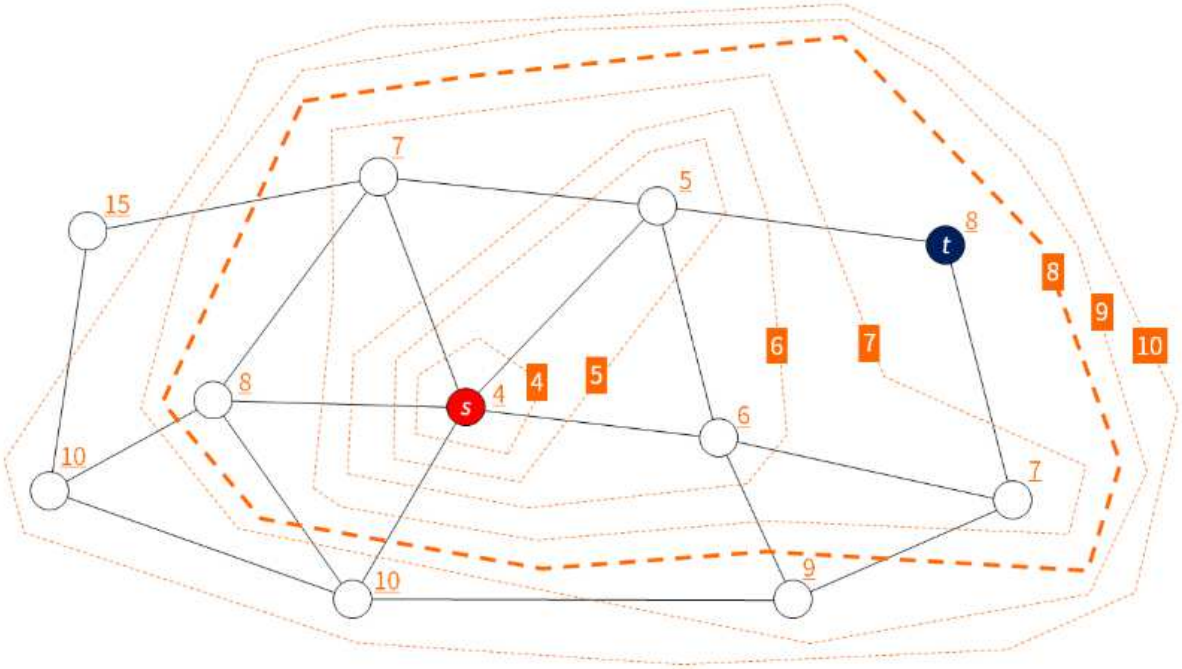


Preliminary

- A* search

- Dijkstra's algorithm의 변형으로,
start 와 end 사이의 최적 경로를 찾는 과정이다.

특이사항 : 각 노드로부터 end까지의 정보를 추가적으로 갖고 있을 때
사용할 수 있는 알고리즘



Preliminary

- Value iteration

- value iteration은 MDP를 해결하기 위한 DP의 한 방법
MDP를 해결한다는 것은 optimal policy π_*를 찾는다는 것
(DP로 MDP를 해결할 수 있을 때는 이 MDP가 Model-based일 때 가능하다.)

Value Iteration은 Model based인 MDP를 풀기 위한 DP 방법 중 하나로, Bellman optimality equation을 사용한다.

Value Iteration은 Model based인 MDP를 풀기 위한 DP 방법 중 하나로, Bellman optimality equation을 사용한다. (Policy iteration은 또 다른 방법으로, Bellman expectation equation을 사용한다고 한다)

Bellman optimality equation :

$$v_*(s) = \max_a \sum_{s',r} p(s', r \mid s, a)\,[r + \gamma v_*(s')]$$

$v_*(s)$: Optimal state-value function

- Value Iteration 특징 : 모든 state에 대한 $v_*(s)$ 값을 찾고, 이것을 사용해서 optimal policy π_*를 찾는다.
- Value Iteration의 핵심 : Bellman optimality equation의 솔루션을 어떻게 찾을 것 인가($v_*(s)$를 찾는 것)
  - Bellman optimality equation(BOE)의 특징 : maximize

Preliminary

- Value iteration

+ DNN (adjust parameters of function)

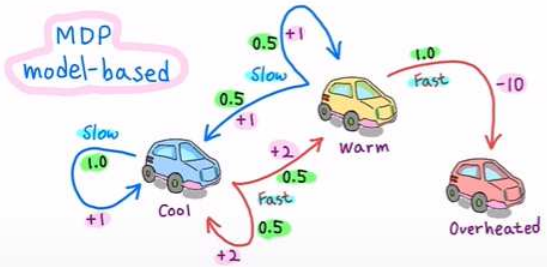
Value Iteration using full backup for DP

A robot car wants to travel far and quickly.

- 3 states: cool, warm, overheated
- 2 actions: slow, fast state transition probability
- rewards: slow = 1, fast = 2 (but −10 when overheated)

$V_{k+1}(s) = \max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma V_k(s')]$ (assume $\gamma = 1$) discount factor

- repeat until convergence



$V(s)$	cool	warm	over
V_2	3.5	2.5	0
V_1	2	1	0
V_0	0	0	0

$V_2(\text{Warm}) = \max_a \begin{Bmatrix} 0.5(1+2) + 0.5(1+1) \\ 1.0(-10+0) \end{Bmatrix}$

Value Iteration using full backup for DP

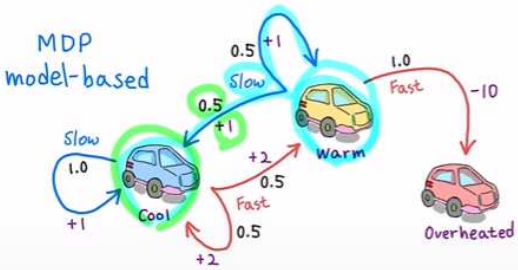
A robot car wants to travel far and quickly.

- 3 states: cool, warm, overheated
- 2 actions: slow, fast state transition probability
- rewards: slow = 1, fast = 2 (but −10 when overheated)

$V_{k+1}(s) = \max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma V_k(s')]$ (assume $\gamma = 1$)

warm, slow, cool, 0.5, 1

- repeat until convergence

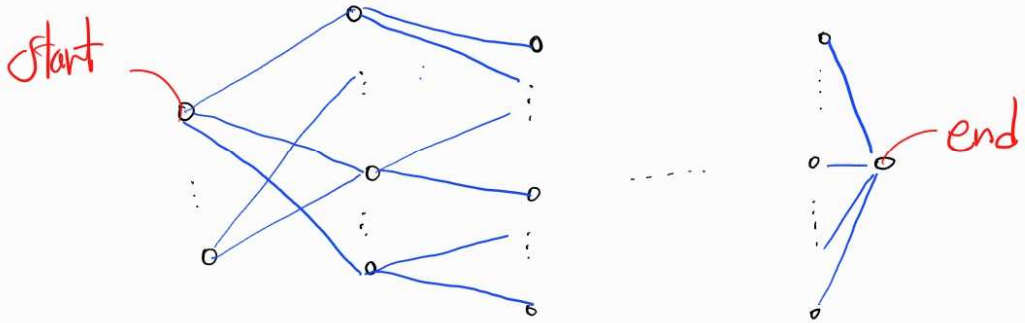


$V(s)$	cool	warm	over
V_2	3.5	2.5	0
V_1	2	1	0
V_0	0	0	0

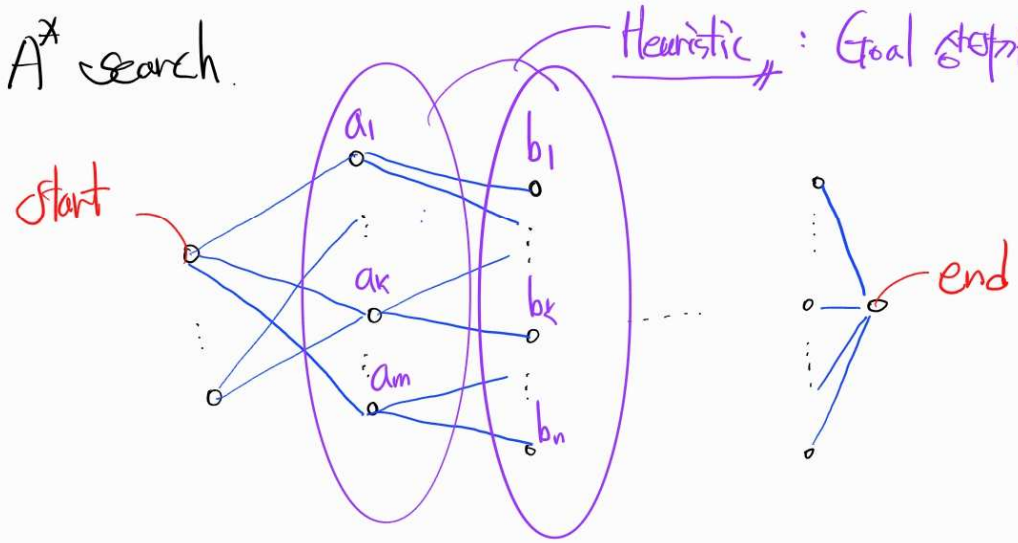
$V_2(\text{Warm}) = \max_a \begin{Bmatrix} 0.5(1+2) + 0.5(1+1) \\ 1.0(-10+0) \end{Bmatrix}$ ← slow ← fast

논문에서 제시한 방법 요약 정리

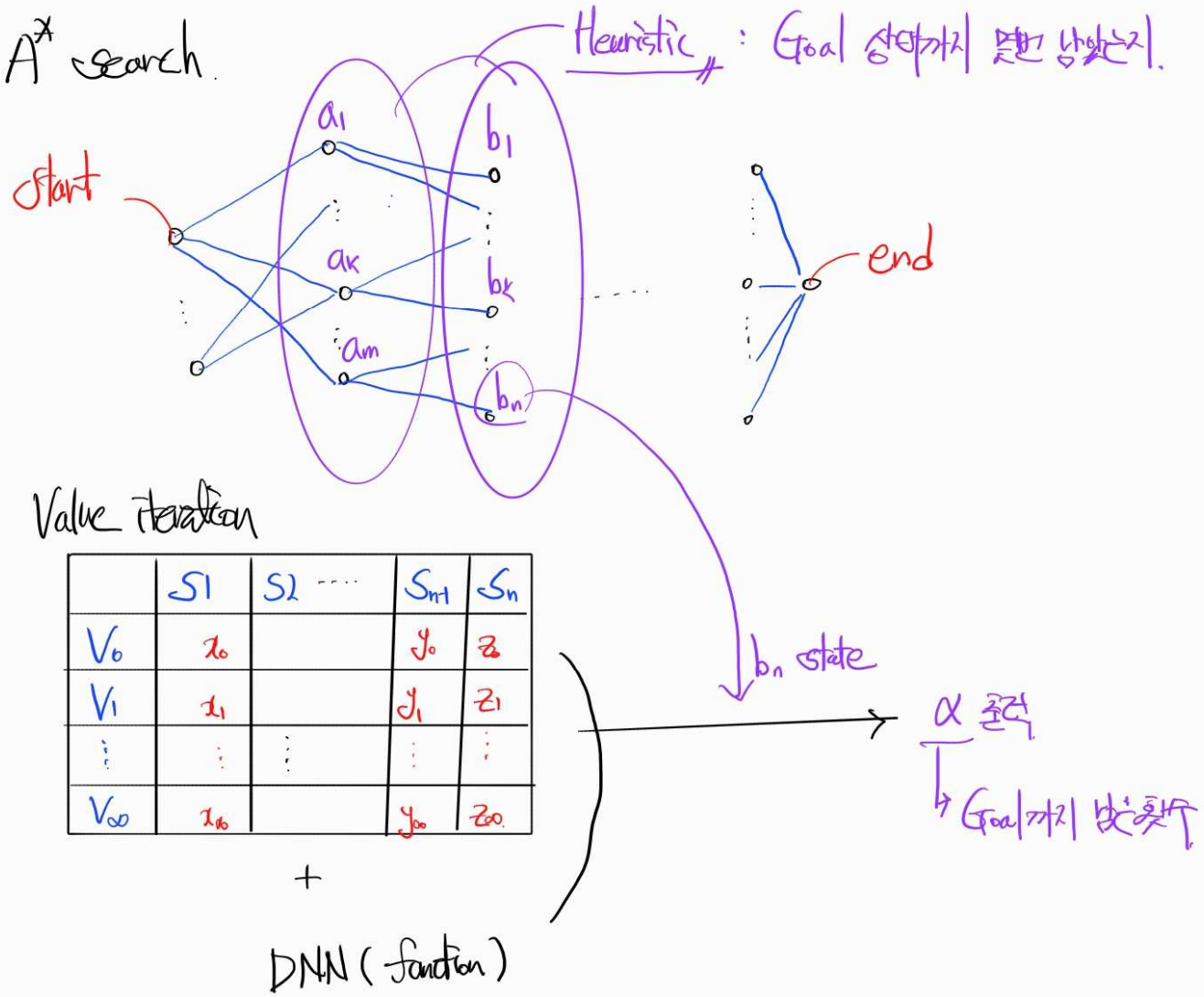
A* search



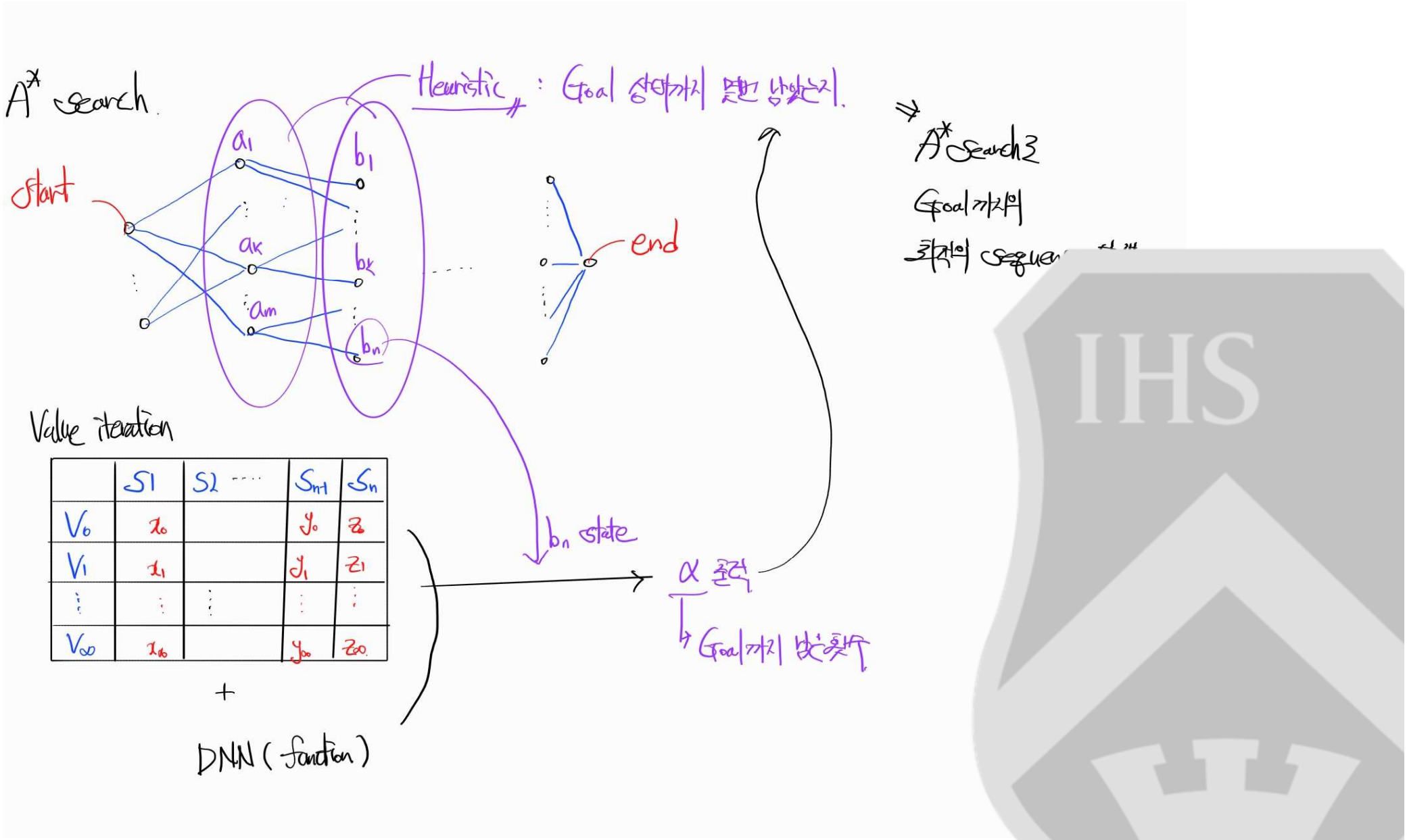
논문에서 제시한 방법 요약 정리



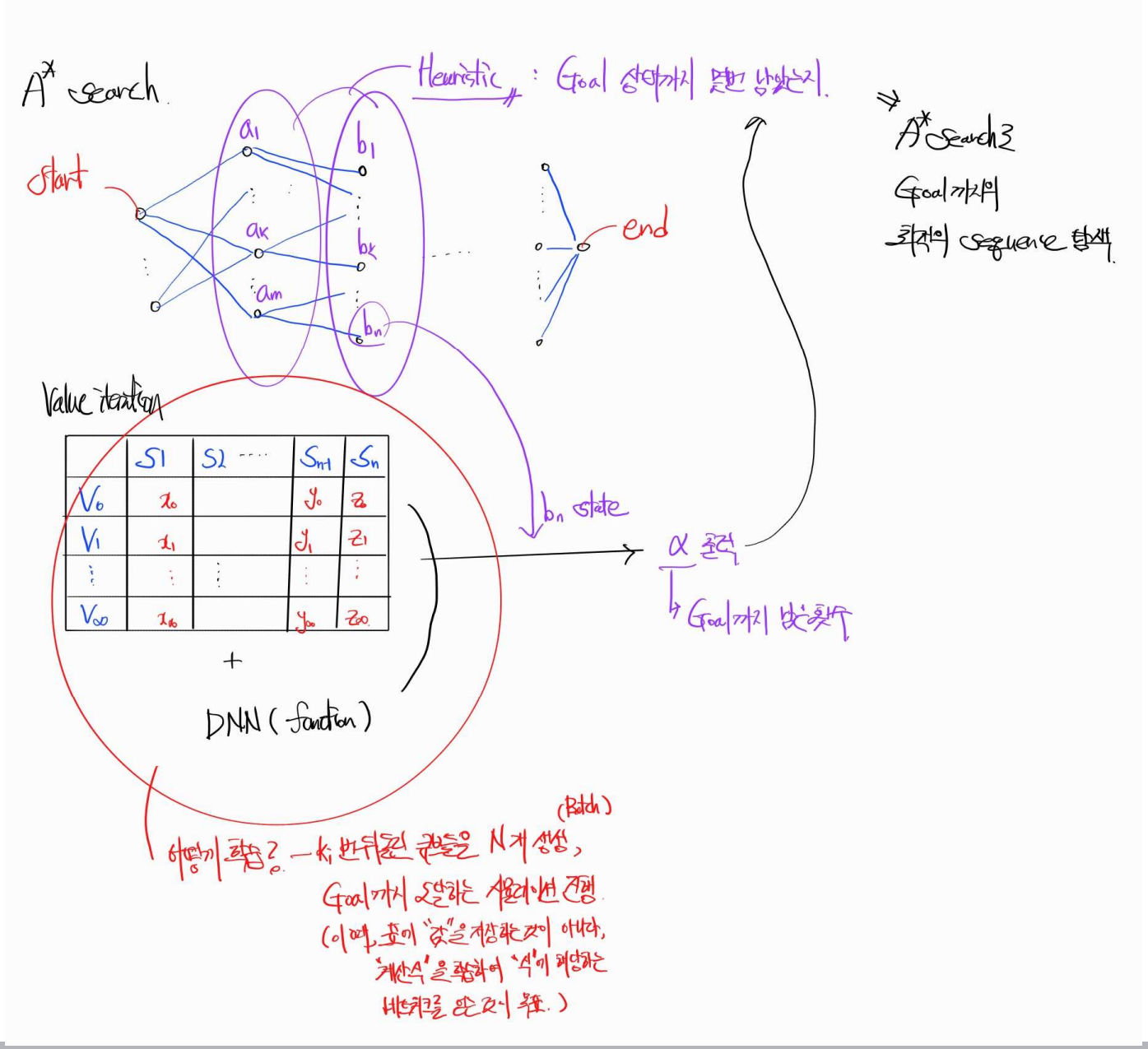
논문에서 제시한 방법 요약 정리



논문에서 제시한 방법 요약 정리



논문에서 제시한 방법 요약 정리



목차

- Solving the Rubik's cube with deep reinforcement learning and search 논문 소개
- **Reinforcement learning in this paper**
- Discussion

논문에서의 강화 학습

- Value functions

$v_\pi(s)$: 정책 π에 대한 상태 가치 함수
(State-value function for policy π)

$\Rightarrow$ 상태 s에서 시작하여, 정책 π를 따랐을 때 얻게 되는 이득의 기댓값

즉, 상태 s에 대한 정책 π의 가치평가가 가능한 함수

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right]$$

논문에서의 강화 학습

- Value functions

$q_\pi(s,a)$: 정책 π에 대한 행동 가치 함수
(Action-value function for policy π)

$\Rightarrow$ 상태 s에서 행동 a를 취한 이후 정책 π를 따랐을 때 얻게 되는 보상의 기댓값

즉, 주어진 상태에서 특정한 행동에 대한 가치평가가 가능한 함수

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A\right.$$

논문에서의 강화 학습

- Value functions

V_π를 구하거나, 알고있다 라는 것은?

A) 상태가 주어졌을 때 정책 π에 따라 의사결정한 것에 대한 보상 계산이 가능

$\Rightarrow$ 1. 궁금한 상태에 대한 평가가 가능
$\Rightarrow$ 2. 여러 정책 π마다 V_π 를 알게 된다면, 그 중에서 가장 높은 정책 π_*를 선택할 수 있다. (주어진 초기 상황 s에 대해 V_{π_k} 가 가장 큰 정책 선택 가능)

논문에서의 강화 학습

- Solving problems by using reinforcement learning

강화 학습으로 문제를 푸는 것은?

A) 장기적으로 많은 보상을 얻는 정책을 찾는다.

MDP 에서의 Optimal policy π_* 는 다음 조건을 만족시킨다.
For all $s \in S, V_{\pi_k}(s) \leq V_{\pi_*}(s) \iff \pi_k \leq \pi_*$

$\Rightarrow$ π_*에 대한 state-value function : $v_*(s) = \max_\pi v_\pi(s)$
π_*에 대한 action-value function : $q_*(s,a) = \max_\pi q_\pi(s,a)$

논문에서의 강화 학습

- Solving problems by using reinforcement learning

Bellman optimality equation 등장 배경 :

최적 정책을 따르는 상태의 가치는
그 상태에서 선택할 수 있는 행동으로부터 나오는 이득의 기댓값과 같다.

논문에서의 강화 학습

- Solving problems by using reinforcement learning

Bellman optimality equation 등장 배경 :

Optimal policy π_* 을 따르는 state-value는
s에서 선택할 수 있는 행동$a \in A$ 에서의 action-value 값과 같다.

$\Rightarrow$ π_*에 대한 state-value function : $v_*(s) = \max_\pi v_\pi(s)$
π_*에 대한 action-value function : $q_*(s,a) = \max_\pi q_\pi(s,a)$

논문에서의 강화 학습

- Solving problems by using reinforcement learning

Bellman optimality equation 등장 배경 :

Optimal policy π_* 을 따르는 state-value는
s에서 선택할 수 있는 행동$a \in A$ 에서의 action-value 값과 같다.

$$
\begin{aligned}
v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\
&= \max_a \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] \\
&= \max_a \mathbb{E}_{\pi_*}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right] \\
&= \max_a \mathbb{E}_{\pi_*}\left[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \,\middle|\, S_t = s, A_t = a\right] \\
&= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\
&= \max_{a \in \mathcal{A}(s)} \sum_{s',r} p(s', r|s, a)\left[r + \gamma v_*(s')\right].
\end{aligned}
$$

$$
\begin{aligned}
q_*(s,a) &= \mathbb{E}\left[R_{t+1} + \right. \\
&= \sum_{s',r} p(s', r|
\end{aligned}
$$

논문에서의 강화 학습

- Solving problems by using reinforcement learning

Bellman optimality equation 등장 배경 :

Optimal policy π_* 을 따르는 state-value는
s에서 선택할 수 있는 행동$a \in A$ 에서의 action-value 값과 같다.

강화 학습 목표 : π_* 를 알고 싶다
$\Rightarrow$ π_* 를 알기 위해서는, V_{π_k}를 계산해야 한다.

V_{π_k}이 V_{π_*}라면, 어떤 식을 만족시키나?

$$V_{\pi_k}(s) = \max_{a \in A} q_{\pi_*}(s, a)$$

V_π를 구하거나, 알고있다 라는 것은?

A) 상태가 주어졌을 때 정책 π에 따라 의사결정한 것에 대한 보상 계산이 가능

$\Rightarrow$ 1. 궁금한 상태에 대한 평가가 가능
$\Rightarrow$ **2. 여러 정책 π마다 V_π 를 알게 된다면, 그 중에서 가장 높은 정책 π_*를 선택할 수 있다. (주어진 초기 상황 s에 대해 V_{π_k} 가 가장 큰 정책 선택 가능)**

논문에서의 강화 학습

- Solving problems by using reinforcement learning

Bellman optimality equation 정리 :

상태 s에서 시작하여 최적 행동을 계속 취했을 때 받을 수 있는 최대 가치

논문에서의 강화 학습

- Solving problems by using reinforcement learning

Value iteration이란?

A) 가치함수 $V_k(s)$을 반복해서 갱신함으로써, 최종적으로 최적 가치 함수 $V_*(s)$에 수렴시키는 알고리즘

핵심 아이디어 :
Bellman optimality equation에 근거하여, 현재 추정치 $V_k(s)$를 사용해 다음 단계 $V_{k+1}(s)$를 구하고, 이를 모든 상태에 대해 반복

논문에서의 강화 학습

- Solving problems by using reinforcement learning

Value iteration이란?

A) 가치함수 $V_k(s)$을 반복해서 갱신함으로써, 최종적으로 최적 가치 함수 $V_*(s)$에 수렴시키는 알고리즘

1. **처음에:**
   - $V_0(s)$를 아무렇게나(0 또는 임의) 설정. 이 값은 "이 상태에서 받을 수 있는 최대 가치"에 대한 **대충의 추측** 정도.
2. **한 번의 반복(갱신):**
   - 모든 상태 s에 대해, "어떤 행동 a를 취했을 때, $\sum_{s'} P(s' \mid s,a)[R+\gamma V_k(s')]$" 값을 구하고, 그 중 최대값을 새로운 $V_{k+1}(s)$로 삼는다.
   - 즉, "현재 가치 함수로 보면 다음 상태(들)가 얼마나 가치가 있는지"를 계산해 보고, "가장 가치가 높은 행동"을 골라 그 기대값으로 s의 가치를 업데이트한다.
3. **계속 반복:**
   - k을 1씩 증가시키면서 이 과정을 반복할수록, V_k가 점점 더 **정확한** 가치로 수렴해간다.

ChatGPT

Preliminary

- Value iteration

+ DNN (adjust parameters of function)

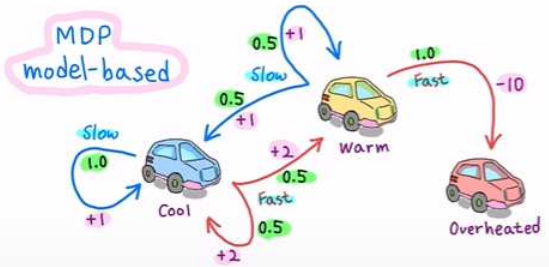
Value Iteration using full backup for DP

A robot car wants to travel far and quickly.

- 3 states: cool, warm, overheated
- 2 actions: slow, fast state transition probability
- rewards: slow = 1, fast = 2 (but −10 when overheated)

$V_{k+1}(s) = \max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma V_k(s')]$ (assume $\gamma = 1$) discount factor

- repeat until convergence



$V(s)$	cool	warm	over
V_2	3.5	2.5	0
V_1	2	1	0
V_0	0	0	0

$V_2(\text{Warm}) = \max_a \begin{Bmatrix} 0.5(1+2) + 0.5(1+1) \\ 1.0(-10+0) \end{Bmatrix}$

Value Iteration using full backup for DP

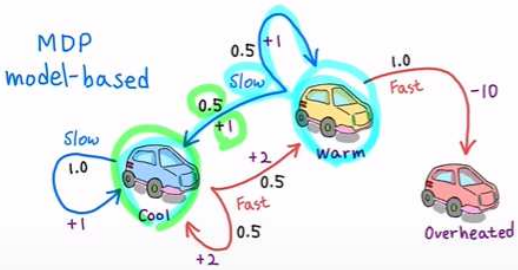
A robot car wants to travel far and quickly.

- 3 states: cool, warm, overheated
- 2 actions: slow, fast state transition probability
- rewards: slow = 1, fast = 2 (but −10 when overheated)

$V_{k+1}(s) = \max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma V_k(s')]$ (assume $\gamma = 1$)

warm, slow, cool, 0.5, 1

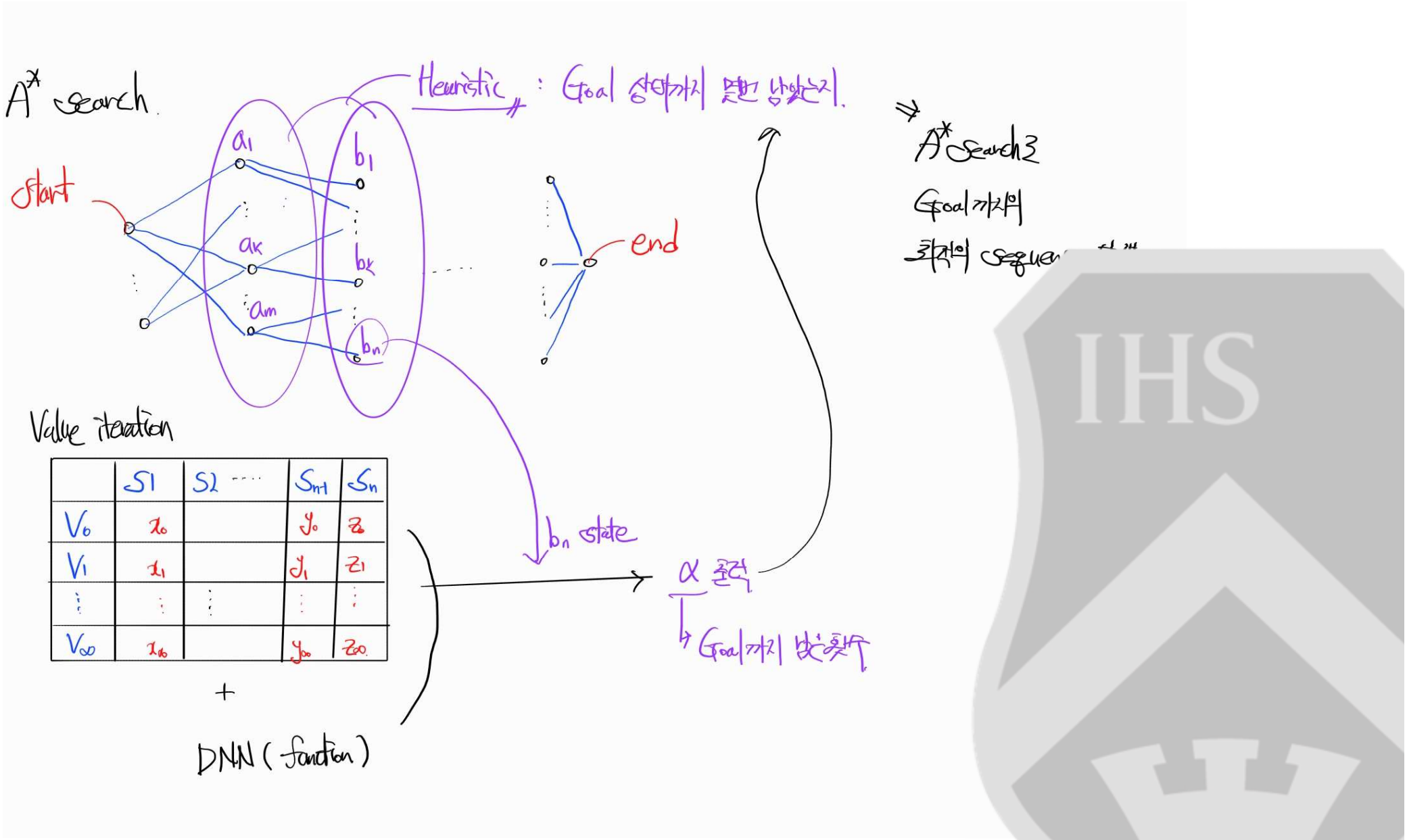
- repeat until convergence



$V(s)$	cool	warm	over
V_2	3.5	2.5	0
V_1	2	1	0
V_0	0	0	0

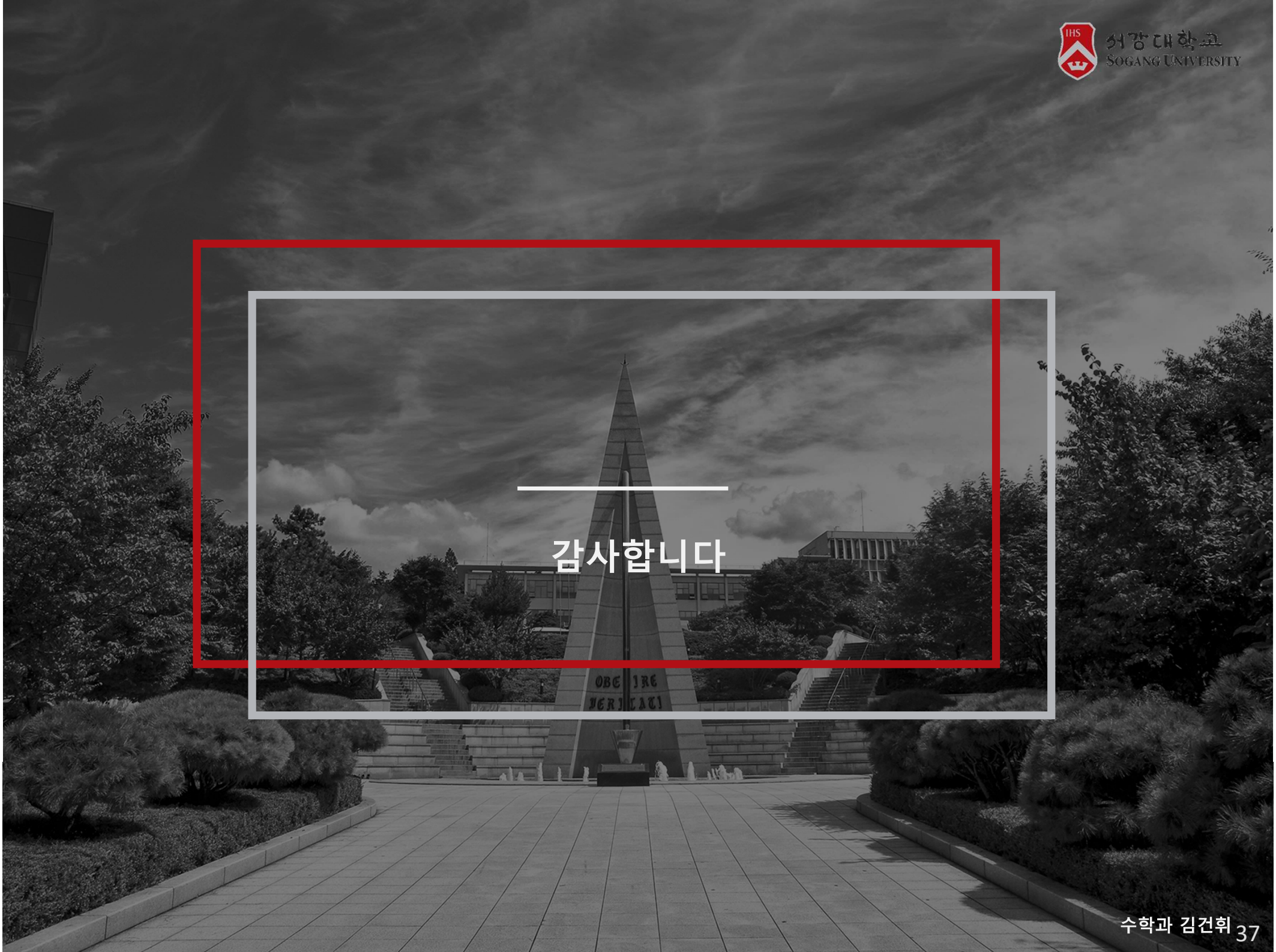
$V_2(\text{Warm}) = \max_a \begin{Bmatrix} 0.5(1+2) + 0.5(1+1) \\ 1.0(-10+0) \end{Bmatrix}$ ← slow ← fast

논문에서 제시한 방법 요약 정리



목차

- Solving the Rubik's cube with deep reinforcement learning and search 논문 소개
- Reinforcement learning in this paper
- **Discussion**


서강대학교
SOGANG UNIVERSITY
감사합니다