CSP ⇒ path가 중요한 게 아니라 가능한 solution을 찾는 게 중요

- **Node consistency**: 도메인의 모든 값이 unary 제약 위반하지 않을 때
- **Arc consistency**: D_i의 어떤 value에 관해 제약을 만족시킬 수 있는 value가 D_j에 하나도 없으면 consistency 만족 X → 해당 value를 D_i에서 제거
- **AC-3**: 제약(X_i, X_j)에 관해 보다가 arc consistency 불만족으로 인해 D_i에 변화가 생기면 X_i의 이웃인 X_k와의 제약인 (X_k, X_i)에 관해 보는 알고리즘 $O(cd^3)$

각 variable의 도메인에서 consistency 못 지키는 값을 지워 나가는 것
- **k-consistency**: $k-1$개 변수에 어떠한 consistent assignment를 하더라도 k번째 변수에 할당 가능한 consistent value가 존재하는 경우
- **Backtracking search**: 한 solution 찾기 위해 직접 value 넣어 가 보는 것
  - 어떤 variable을 먼저 뽑느냐
    (1) **Minimum Remaining Values** (할당 가능한 value가 가장 적게 남은 것 선택, *fail-first*)
    (2) **Degree heuristic** (아직 할당되지 않은 variable끼리의 제약으로 가장 많이 묶여 있는 variable 선택)
  - 어떤 value를 먼저 assign하느냐 → **Least constraining value** (다른 variable에 최대한 적은 영향을 주는 value 고르기, *fail-last*)
  - 어떻게 inference(local consistency 판단)하느냐 → **Forward checking** (어떤 variable에 value 할당하면 해당 variable을 head로 하는 제약에 관해 arc consistency 확인)
- CSP도 local search 가능
- 문제의 구조를 통해 CSP inference 최적화
  (1) Subproblem인지 확인 → $O(n/c * d^c)$ (c: subproblem의 variable 수)
  (2) **Tree 구조**일 때 → $O(nd^2)$, 문제 더 간단해짐, 위상정렬 필요
  (3) **Nearly tree-structured CSP** → tree로 만들고 싶지만 불가능해서 일부 variable을 cycle cutset으로 만듦, $O(d^c * (n-c)d^2)$ (c: cutset 크기)
  (4) Join tree algorithm → subproblem들로 묶어서 하나로 연결된 tree로 만들기, $O(nd^{w+1})$ (w: 가장 큰 subproblem 크기)

Logic Agent ⇒ Inference 하는 방법 유념하기

- Knowledge base: 세계의 지식을 표현하기에 적절한 형태의 문장의 집합
- Inference rule: KB에게 질문(ASK)했을 때 대답은 KB에게 이전에 말했던 (TELL) 것으로부터 나와야 한다.
- Logic: 결론이 도출될 수 있는 형식이 있는 언어 → syntax, semantics

→ Logic에서 모든 문장은 각각 가능한 세계에서 참 또는 거짓의 진리를 표현해야 한다. (model/possible world가 모든 문장에 관해 진리를 결정한다.)
- α is true in $m \Leftrightarrow m$이 α의 model이다.
- $M(\alpha)$: α를 만족시키는 가능한 세계(model)의 집합
- 내가 알고 있는 지식이 가능한 model의 집합이다. → 아무 것도 모르는 상태이면 모든 모델이 후보 → 지식을 쌓아 갈수록 후보 중에서 안 되는 모델이 나옴 → possible world 축소
- **Entailment**: $KB \vDash \alpha \Leftrightarrow M(KB) \subseteq M(\alpha)$ (KB가 참인 모든 세계에서 α가 참)
- Logic inference: $KB \vdash_i \alpha$ (α가 procedure i에 의해 KB로부터 도출될 수 있다.) → sound와 complete 특성 만족해야 함
- Propositional Logic: atomic sentence (명제 기호, 참, 거짓)

logic connective (논리 접속사), **literal** (명제 기호 또는 부정 붙은 명제 기호)
- $A \Rightarrow B$ (A가 true이고 B가 false일 때만 false)
- 직접 진리표(truth table) 작성해도 되지만 너무 복잡함
- Depth-first enumeration: 모든 모델을 나열하여 DFS처럼 점검하는 방식

→ KB와 α를 구성하는 sentence에 있는 symbol의 진리값을 true와 false 할당한 것 각각 추가해서 탐색

→ $KB \vDash \alpha$인지 확인하고자 KB가 현재 탐색 중인 모델에서 참일 때 α가 모델에서 참인지를 구함, 시간복잡도 $O(2^n)$, 공간복잡도 $O(n)$
- Modus Ponens: $\frac{\alpha \Rightarrow \beta, \alpha}{\beta}$ → α가 참이고 $\alpha \Longrightarrow \beta$가 참이면 β도 무조건 참
- And-elimination: $\frac{\alpha \wedge \beta}{\alpha}$ → $\alpha \wedge \beta$가 참이면 α도 무조건 참
- Logical equivalences: $\frac{\alpha \Leftrightarrow \beta}{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}$ → biconditional
- $\alpha \equiv \beta$ iff $\alpha \vDash \beta$ and $\beta \vDash \alpha$ → 두 문장 α와 β가 동일한 모델에서 참일 때, 두 문장은 서로 동치관계
- Contraposition: $(\alpha \Rightarrow \beta) \equiv (\neg\beta \Rightarrow \neg\alpha)$
- Implication elimination: $(\alpha \Rightarrow \beta) \equiv (\neg\alpha \vee \beta)$
- 한 문장이 valid 하다 = 그 문장이 모든 모델에서 참이라는 것
- 한 문장이 unsatisfiable 하다 = 어떠한 모델에서도 참이 되지 못한다는 것
- $(\boldsymbol{KB} \vDash \boldsymbol{\alpha}) \Leftrightarrow ((\boldsymbol{KB} \wedge \neg\boldsymbol{\alpha})\textbf{ is unsatisfiable})$ → $KB \Rightarrow \alpha$가 false인 걸 만족하지 못하는 상황(unsatisfiable)이면, 즉 $\neg(KB \Rightarrow \alpha)$를 true로 만드는 모델이 존재하지 않으면 $KB \Rightarrow \alpha$는 모든 모델에서 참이라는 것이고, 이는 KB가 α를 entail한다는 걸 의미함 → **모순에 의한 증명**(proof by contradiction)
- KB가 α를 entail 하는지에 관한 증명은 크게 두 가지

(1) Inference rule 적용하기 → Resolution, Forward • Backward Checking

(2) Model checking

1) Truth table enumeration → 복잡하고 오래 걸림

2) satisfiability 확인하는 방법 사용 ⇒ CSP 문제로 확장

① DPLL ② Heuristic local search (WalkSAT)
- Clause(절): disjunction of literals (literal이 OR로 연결된 것)
- $\frac{l_1 \vee \cdots \vee l_k, m}{\backslash l_1 \vee \cdots \vee \backslash l_{i-1} \vee \backslash l_{i+1} \cdots \vee \backslash l_k}$ → l_i와 m이 서로 상보(하나가 다른 것의 부정)이면 합집합 문장에서 l_i 원소 지워줌 (unit resolution)
- $\frac{l_1 \vee \cdots \vee l_k, m_1 \vee \cdots \vee m_n}{l_1 \vee \cdots \vee l_{i-1} \vee l_{i+1} \vee \backslash l_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n}$ → l_i와 m_j이 서로 상보이면, 각자 속한 합집합 문장에서 각각 원소 지워줌 (full resolution)
- **CNF**(Conjuctive Normal Form) → clause들의 논리곱으로 이루어진 문장

⇒ **모든 논리 문장은 clause들의 논리곱인 CNF와 논리적으로 동치**
- Resolution algorithm ⇒ complete 하지만 복잡도가 exponential에 가까움

(1) $KB \wedge \neg\alpha$를 CNF로 변환하기 (**resolution 사용하려면 CNF로 바꿔야 함**)

(2) Resolution → 상보 literal 갖는 각 쌍을 분해해서 새 clause를 산출하고 그것을 clause들의 집합에 추가하기

(3) 두 개의 clause가 빈 clause로 분해되는 순간 $KB \wedge \neg\alpha$를 만족하는 모델이 존재하지 않는다(unsatisfiable)는 것이므로 $KB \vDash \alpha$는 true (proof by contradiction)

(4) 더 이상 새로 분해되어서 추가되는 clause가 없으면 (기존에 있던 clause들 집합에서 더 이상의 변화가 없으면) $KB \wedge \neg\alpha$가 satisfiable 하다는 것이므로 $KB \vDash \alpha$는 false
- **Definite clause**: 정확히 한 리터럴만 positive인 논리합 문장

= 'symbol(fact)'이거나 '(conjunction of symbols) ⇒ symbol'인 경우
- Forward Backward Checking의 공통점

→ KB를 definite clauses의 conjunction으로 보자

→ KB를 이루는 Definite clause들의 전제들을 가지고 결과를 inference

→ Modus Ponens 사용 ('fact'가 true이고, 'fact이면 result이다'가 true일 때, result는 true이다) ⇒ $\frac{\alpha_1, \cdots, \alpha_n, \quad \alpha_1 \wedge \cdots \wedge \alpha_n \Rightarrow \beta}{\beta}$ (α는 참인 fact, β는 결과)

→ 복잡도가 KB의 크기에 linear하다
- **Forward checking** (전방 연쇄, 연역적)

→ 기존에 알려진 사실들로 하여금 새로운 사실을 추리 ⇒ 함의에 대한 모든 전제가 사실이면 그것의 결론을 새로운 사실로 KB에 추가

→ 참인 fact가 전제일 때 result가 참이 될 수 있는지를 계속 타고 가면서 보는 것

→ [코드] count[c]: KB를 구성하는 definite clause인 c의 전제에 있는 아직 inference 되지 않은 symbol 수

→ Goal과 필요 없는 것까지 inference함
- **Backward checking** (후방 연쇄)

→ 확인하고자 하는 사실 q로부터 내려가면서 q를 사실로서 추가하기 위해 KB에서 결론이 q인 문장의 모든 전제가 참이 될 수 있는지 확인

→ result가 참이 되게 하기 위해서 문장에서의 모든 전제가 참인지를 역방향으로 타고 가면서 보는 것

→ 무한 loop를 피하려면 새로운 subgoal이 이미 goal stack에 있는지 확인

→ 반복되는 작업 피하려면 새로운 subgoal이 이미 참 또는 거짓으로 증명되었는지 확인

→ Goal에 관련된 것만 접근하므로 linear보다 더 복잡도 낮을 수 있음
- **DPLL 알고리즘** ⇒ 어떻게 search space를 줄여나갈지에 관한 아이디어

→ 주어진 문장 s의 satisfiability를 확인하는 알고리즘 ⇒ 해당 문장을 만족하는 모델이 있는지 (만족하는 모델 없는 걸 가지고 proof by contradiction 이용 가능) ⇒ s의 CNF를 구성하는 clause이 모두 true가 되는 모델이 있는지

→ B와 C가 어떤 논리값을 가지든지 A가 true이면 $(A \vee B) \wedge (A \vee C)$는 true

→ Pure symbol: 모든 clause에서 sign이 똑같이 붙은 symbol (부정이 붙으면 모든 clause에서 부정이 붙은 상태로 나오고, 안 붙으면 모두 안 붙은 상태로 나오는 경우) ⇒ 해당 symbol에 대응되는 literal이 true가 되도록 하는 value 할당하고 그 literal들은 모두 지워줘서 s의 CNF를 구성하는 clause들이 모두 참인지를 판단하는 데 영향이 가지 않도록 함

→ Unit clause: 한 리터럴만 false인 clause ⇒ false인 리터럴만 clause에서 빼면 나머지 literal만을 가지고 clause의 참 또는 거짓을 판단할 수 있음
- **WalkSAT 알고리즘** ⇒ 마치 local search처럼

→ 현재 만족하지 않는(false인) clause 골라서 최대한 좋아지도록 만들기 ⇒ 확률에 의해 랜덤하게 두 가지 중 하나 선택하여 symbol 진리 값 뒤집기

(1) 랜덤으로 선택한 만족하지 않은 clause에서 임의의 symbol의 진리값

(2) 뒤집으면 가장 많은 clause를 참으로 만들 수 있는 symbol의 진리값
- Propositional logic의 단점: 표현력이 부족함

First Order Logic ⇒ FOL inference 과정 적용 유념

- First order logic: objects, relations, functions 포함
- FOL의 기본 구성 요소: predicates(술어), constants, functions, variables, connectives(논리 연결자), equality(=), quantifiers
- Quantifier → universal & existential
- Universal quantification: $\forall x\ P$ → 주로 ⇒와 많이 사용 → conjunction of instantiations of P
- Existential quantification: $\exists x\ P$ → 주로 ∧와 많이 사용 → disjunnction of instantiations of P
- ∀를 ∃로 바꾸려면 not을 안으로 집어넣으면서 바꿔 주기 → 예) $\forall x\ P \Leftrightarrow \neg\exists x\ \neg P$
- FOL을 propositional logic으로 바꾸려면 변수가 없는 ground sentence로 만들기 → ∀, ∃ 한정자는 어떻게 하지? ⇒ universal, existential instantiation
- Ground atomic sentence를 propositional symbol로 보자!
- **Universal instantiation**(UI): universal 한정자를 없애려면 변수를 ground term으로 대체하기 $\frac{\forall v \alpha}{\mathrm{SUBST}(\{v/g\}, \alpha)}$ → 모든 v에 관해 sentence α가 true이면, α라는 문장에서 v 대신 ground term인 g로 대입해도 true ⇒ KB에 존재하는 모든 ground term에 관해 다 적용해 봐야 한다
- **Existential instantiation**(EI): existential 한정자를 없애려면 변수를 KB에 없는 constant symbol로 대체하기 $\frac{\exists v \alpha}{\mathrm{SUBST}(\{v/k\}, \alpha)}$ → 여기서 나오는 constant symbol이 **skolem constant**
- UI는 모든 가능한 ground term에 관해 적용되어야 하고, 새로운 KB가 기존 KB와 equivalent
- EI는 한 번만 적용하면 되지만, 새로운 KB가 기존 KB와 equivalent하진 않음 → but 기존 KB가 satisfiable하면 새로운 KB도 satisfiable
- Ground sentence가 기존 KB에 관해 entail 되었으면 새로운 KB에 관해서도 entail 된다
- 모든 FOL KB는 entailment를 보존하면서 propositionalize 될 수 있다
- 문장 α가 FOL인 KB에서 entail 되었으면, propositionalize 된 KB의 유한 부분집합에서 entail 될 수 있다 → 유한 번 instantiation 하면서 propositional KB를 만들었을 때, 주어진 sentence α가 KB로부터 entail 된다고 결과가 나올 수도 있지만, 못 나올 수도 있다 (semidecidable)
- Propositionalization의 문제: 불필요한 문장들을 많이 만들어 낼 수 있다

→ 변수가 k개 쓰이는 predicate가 p개 있고, n개의 constant가 있을 때, $p \cdot n^k$ 번의 instantiation이 필요하다
- Lifting: FOL을 POL로 만들지 말고 lifted inference rule을 사용해 보자!
- **Generalized Modus Ponens**(GMP)

$\frac{p_1', \cdots, p_n', (p_1 \wedge \cdots \wedge p_n \Rightarrow q)}{\mathrm{SUBST}(\theta, q)}$ p_i'라는 n개의 사실들을 통해서 q라는 문장의 참 거짓을 판단하고 싶은데, 이를 위해 θ라는 substitution을 통해 p_i'와 p_i를 같게끔 만들어 주고 싶다
- **Unification**: 문장 α와 β가 어떤 substitution θ에 의해 같아질 때 $\mathrm{Unify}(\alpha, \beta) = \theta \quad if\ \alpha\theta = \beta\theta$ → 서로 다른 문장을 identical하게 보이도록 만들어주는 substitution을 찾는 과정
- **MGU(Most General Unifier)**: 모든 unnifiable한 표현의 쌍들은 가능한 substitution이 매우 많은데, 그중 가장 간단한 unifier
- Unification algorithm ⇒ 어렵게 생각하지 말고 흐름을 기억

(1) 두 문장 x, y가 같으면 substitution θ 반환

(2) x가 변수이면, x를 y로 대체하고 싶은데 이미 KB에 x를 val로 대체했으면 val을 y로 대체 가능한지 보고, 이미 y를 val로 대체했으면 x를 val로 대체 가능한지 보기

(3) 둘 다 compound이면 operator끼리 보고 argument끼리도 보기

(4) 둘 다 argument이면 각 argument list의 맨 앞 원소 unify 하고 나머지끼리도 보기

(5) 모든 조건 안 되면 failure (operator가 다르면 여기서 걸림)
- Standarizing: 두 문장에서 서로 관련이 없는 변수인데 이름이 같아서 충돌하는 것을 해결하고자 하나를 다른 변수의 이름으로 바꿔주는 것
- GMP는 soundness하다는 증명

$p_1', \cdots, p_n', (p_1 \wedge \cdots \wedge p_n \Rightarrow q) \vDash q\theta$ 증명 ⇒ rule 부분에 θ로 substitution 적용하는 것으로 entail 되고, 앞에 fact를 가지고 모두 ∧로 연결한 것으로 entail 된 후, 마찬가지로 각 fact 마다 θ 적용한 게 모두 ∧로 연결한 것으로 entail 된다
- GMP는 definite clause로 이루어진 KB에 관해서 complete하다
- **Forward chaining**: 알려진 사실로부터 시작해서 모든 premise가 satisfied이면 그 문장의 conclusion을 알려진 사실에 추가 (단, 이미 알려진 사실에서 변수 이름만 바뀐 것이면 새로운 사실로 추가되지 않음) ⇒ query에 관한 답을 찾거나 새로운 사실이 추가되지 않으면 종료
- Forward chaining algorithm: KB에 있는 모든 rule에 관해, fact($p_1' \cdots p_n'$)로 주어진 KB를 가지고 rule의 전제($p_1, \cdots, p_n$)과 fact를 같게끔 substitution 할 수 있는 θ를 마찬가지로 rule의 결과(q)에 적용(q')했을 때, q'이 KB에 이미 있거나 현재 돌고 있는 loop에서 나온 게 아니면 q'와 증명하고자 하는 문장인 α와 substitution → ϕ로 unify 될 수 있으면 ϕ 반환

⇒ 현재 loop에서 새로 추가된 게 없으면 실패, 있으면 기존 KB에 추가
- 증명하고자 하는 α가 entail 되지 않으면 종료되지 않고 계속 진행하게 됨

⇒ definite clause로 entail 하는 것도 semidecidable 하므로
- Forward chaining의 단점 (Forward checking과 유사)

(1) 모든 rule들을 매 iteration마다 다시 check하는 문제 발생 → incremental forward chaining (이전 iteration에서 추가되지 않은 전제는 보지 말자)

(2) 관련 없는 문장을 사실들을 많이 만들어 낼 수 있음

(3) 어떤 match에서 premise가 expensive할 수도 있음
- **Backward chaining**: 주어진 query인 q가 참인 것이 알려져 있거나, q를 결론으로 내는 rule에서 rule의 전제들이 모두 맞는지를 증명하는 과정 → depth-first AND/OR search (minimax algorithm과 유사)
- Backward chaining의 단점

(1) infinite loop에 빠질 수 있음 (증명하려고 했더니 또 증명해야 하는 경우)

(2) 반복되는 subgoal로 인해 비효율적 (caching으로 개선은 가능)
- Resolution: FOL에서도 resolution 사용 가능

$$\frac{l_1 \vee \cdots \vee l_k, \quad m_1 \vee \cdots \vee m_n}{(l_1 \vee \cdots \vee l_{i-1} \vee l_{i+1} \vee \cdots \vee l_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n)\theta}$$

→ 두 clause가 standardized apart된 상태에서 두 개의 literal이 어떠한 substitution θ에 의해 unify 될 수 있는 경우, 두 literal을 없애고 나머지 literal들을 모두 θ로 substitution 한 상태에서 OR로 연결해주면 됨
- CNF ⇒ resolution 사용하려면 문장들을 CNF로 바꿔야 한다!

(1) Biconditional elimination으로 ⇔ 기호 없애기 $(\alpha \Leftrightarrow \beta) \equiv (\alpha \Rightarrow \beta) \wedge (\alpha \Rightarrow \beta)$

(2) Implication elimination으로 ⇒ 기호 없애기 $(\alpha \Rightarrow \beta) \equiv \neg\alpha \vee \beta$

(3) ¬ 기호 모두 괄호 안으로 넣어주기 $\forall x\ P \Leftrightarrow \neg\exists x\ \neg P$

(4) Skolemization으로 ∃ quantifier 없애기 (어떤 한 변수에 dependent 한 변수로서 함수로 치환 가능) $\exists z \mathrm{Loves}(z, x) \Rightarrow \mathrm{Loves}(G(x), x)$

(5) ∀ quantifier도 drop하여 없애기 (FOL resolution에서는 그냥 없애도 됨)

(6) ∧를 괄호 밖으로, ∨를 괄호 안으로 넣어주기
- Resolution algorithm ⇒ PL에서와 마찬가지로 모순에 의한 증명!

(1) $KB \wedge \neg\alpha$를 CNF로 변환하기 (**resolution 사용하려면 CNF로 바꿔야 함**)

(2) Resolution → substitution θ를 통해 unify 될 수 있는 literal 각 쌍을 분해해서 θ 적용한 lause를 산출하고 그것을 clause들의 집합에 추가하기

(3) 두 개의 clause가 빈 clause로 분해되는 순간 $KB \wedge \neg\alpha$를 만족하는 모델이 존재하지 않는다(unsatisfiable)는 것이므로 $KB \vDash \alpha$는 true (proof by contradiction)

(4) 더 이상 새로 분해되어서 추가되는 clause가 없으면 (기존에 있던 clause들 집합에서 더 이상의 변화가 없으면) $KB \wedge \neg\alpha$가 satisfiable 하다는 것이므로 $KB \vDash \alpha$는 false
- FOL에서의 resolution algorithm은 complete → 단, KB가 찾고자 하는 문장을 entail 하지 않으면 끝나지 않을 수 있다

Local Search ⇒ 현재 상태에서 좀 더 나아질 수 있는 방향으로 최적화

- Local Search의 장점: 메모리 절약 가능, 합리적인 해결책 찾기 가능
- **Hill-climbing search** → 자신의 현재 지점에서 높아지는 방향으로 가기
- Hill-climbing 종류 ⇒ 조금 안 좋은 쪽으로 가는 것도 허용해 보자!

(1) **Stochastic HC**: 확률적으로 방향을 선택하자

(2) **First-choice HC**: 갈 수 있는 방향이 너무 많으면 랜덤으로 봐서 현 상태보다 좋은 상태가 처음 나왔을 때 바로 그 상태로 가 보는 것

(3) **Random-restart HC**: 랜덤으로 초기 상태를 여러 번 정해서 탐색
- **Simulated annealing**: 안 좋아지는 쪽으로 가는 것도 확률적으로 허용해서 변동성을 주는 방법 (좋아지는 방향이 나오면 무조건 가기) → T = 안 좋아지는 쪽으로 가는 걸 얼마나 허용할지 ⇒ $e^{-\frac{|\Delta E|}{T}}$ 확률로 안 좋은 쪽 방향 선택 → T가 점차 감소 ⇒ 초기에 안 좋은 쪽의 확률을 많이 주는 꼴
- **Local beam search**: k개의 state를 골라서 그들의 next state(successor) 중 가장 좋은 k개 선택하고 나머지 버림 → goal 찾을 때까지 반복 ⇒ dependency가 있어서 diversity가 부족
- **Genetic algorithm**: 특정 solution을 진화해 가는 과정에서 가장 optimal한 걸 가지고 진화해 가기

(1) Population: k개의 임의로 생성된 state부터 시작

(2) Fitness function: state들은 모두 evaluation 함수에 의해 결정됨

(3) Survival of the fittest: fitness function 값(quality)이 높을수록 부모로 선택될 확률 높아짐

(4) Offspring: 확률에 의해 선택한 부모를 가지고 자손 생성 → 부모의 일부를 서로 교차하는 **crossover**, 순서 바꾸기 등 무작위로 변형하는 **mutation**
- Continuous space에서도 local search를 할 수 있다

Game ⇒ 상대방은 나를 지게 하고, 나는 나를 이기기 위해 노력

• **Minimax search** → 나는 승리를 위해 최종적으로 Utility-function 값이 최대가 될 수 있는 행동을 택하고, 상대방은 나를 패배하게 만들기 위해 최종적으로 Utility-function 값이 최소가 될 수 있는 행동을 택할 것
⇒ completeness: YES (tree가 finite이면) / optimal: YES / 시간복잡도: $O(b^m)$ / 공간복잡도: $O(bm)$ (DFS와 같음)

• **Alpha-beta pruning** → minimax search의 시간복잡도 줄이기 위해
◊ Alpha: 현재까지 내가 이기기 위해 찾은 최고 점수
◊ Beta: 현재까지 상대방이 나를 지게 만들기 위해 찾은 최저 점수
◊ Alpha cut-off

- 상대방(MIN): 너를 불리하게 만들 경로의 값인 2를 찾았어. 😜
- 나(MAX): 어차피 난 이미 3을 갖는 경로를 찾아서 네가 찾은 경로는 절대 선택할 일이 없는데? 😂
- 상대방(MIN): 하... 여기서는 더 탐색해 봤자 안 되겠네... 여기서 탐색 끝내자. 😮‍💨

◊ Beta cut-off

- 나(MAX): 나를 유리하게 만들 경로를 찾았어. 😜
- 상대방(MIN): 어차피 난 이미 너를 더 불리하게 만들 경로를 찾아서 네가 찾은 경로는 절대 선택할 일이 없는데? 😂
- 나(MAX): 하... 여기서는 더 탐색해 봤자 안 되겠네... 여기서 탐색 끝내자. 😮‍💨

◊ Alpha-beta pruning은 suceessor를 어떤 순서로 check 하는지가 중요
→ perfect ordering이 되면 시간복잡도는 $O(b^{m/2})$

• Cut-off: minimax와 alpha-beta pruning은 leaf node까지는 내려가야 utility function으로 평가가 되므로 깊이가 너무 깊어지면 cut하자! ⇒ cut 할 때는 utility가 아닌 evaluation으로 추정치 반환

• **Expectminimax**: Alpha-beta pruning 할 때 무조건 조건에 맞지 않는다고 cut 하지 말고 확률적으로 보자! → MAX와 MIN 사이 chance node 껴 있음
⇒ 시간복잡도: $O(b^m n^m)$ (n: chance node 개수)

• **Monte Carlo Tree Search**(MCTS): cut-off 할 때 evaluation function 사용하는 것의 대안 → 직접 시뮬레이션을 통해 state의 값이 평가됨
(1) **Selection**: child node를 만들 node 고르는 단계 → $\mathrm{UCB1}(n) = \frac{U(n)}{N(n)} + C * \sqrt{\frac{log\, N(\mathrm{Parent}(n))}{N(n)}}$ 정책에 의해 선택됨 ($U(n)$: 노드 n을 지나가는 전체 playout에서 이긴 횟수, $N(n)$: 노드 n을 지나가는 전체 playout 수, $N(\mathrm{Parent}(n))$: 노드 n의 부모 노드를 지나가는 전체 playout 수) ⇒ 앞의 항이 exploitation (더 좋은 걸 선택하자), 뒤의 항이 exploration (좀 안 좋은 걸 탐색해 보자)
(2) **Expansion**: 선택한 노드에서 child node를 생성하여 tree를 확장 → 생성한 child node가 새로운 leaf node가 됨
(3) **Simulation**: 확장한 leaf node에서 게임이 종료될 때까지 playout → 승패가 결정되는 terminal position까지 게임을 임의로 끝까지 진행해 보기
(4) **Backpropagation**: 확장한 leaf node의 simulation 결과에 따라 조상까지 타고 올라가면서 $U(n)$과 $N(n)$을 업데이트
(5) 가장 playout 수가 많은 상태로 이동 가능한 action 선택 (그만큼 해당 방향으로 가는 경우가 많이 selection 되었다는 의미이므로)

(Uninformed) Search ⇒ 시간・공간복잡도 주의

• **Expansion**: frontier에서 전략에 의해 선택된 state(node)를 closed로
• **Frontier**(fringe, open): expansion의 후보가 되는 node
• Generated(reached) = closed(expanded) + open(fringe, frontier)
• Fringe에서 expansion하려고 node 하나 뽑고, 뽑은 node의 children을 fringe로 넣기→ expansion하려고 뽑은 node는 closed(expanded)로
• **Graph search**: repeated state를 check (반복되는 state 허용하지 X)
• **Tree search**: repeated state check하지 X (반복되는 state 허용)
• Graph search는 closed된 node의 상태를 check함
• Frontier에 있으면 무시하는 경우도 있지만, Frontier에 있는 것보다 더 좋은 게 나왔는지 확인하는 경우도 존재(예: Dijkstra's algorithm)
• **Breadth-first search**(BFS): FIFO 사용, frontier에서 뽑은 node의 child로 goal이 나오면 바로 찾은 것

Completeness	b(branch의 최대 개수)가 finite이면 YES
Optimality	step마다 동일한 cost이면 YES, 그렇지 않으면 NO
시간복잡도	Tree search: $O(b^d)$ / Graph search: $O(b^d)$이지만 작음
공간복잡도	$O(b^d)$ → 시간복잡도와 동일

• **Depth-first search**(DFS): LIFO(stack) 사용

Completeness	일반적으로 NO → Tree search: finite state일 때 YES (반복되는 state 제외) / Graph search: finite state space일 때
Optimality	NO (최상의 goal 찾지 못하고 더 깊은 goal 찾을 수 있음)
시간복잡도	$O(b^m)$ → m(최대 깊이)
공간복잡도	Tree search: $O(bm)$ → 특정 path에 관한 정보만 가지고 있으면 됨 / Graph search: $O(b^m)$ → caching을 위해

• **Uniform-cost search**: Best-first search에서 $f(n) = g(n)$인 경우

Completeness	YES (음수인 경로 없이 step-cost가 양수일 때)
Optimality	YES → evaluation function 값이 단조 증가하는 순으로 expansion 되므로
시간복잡도	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$ (최소 $b^{C^*/\epsilon}$ 만큼의 depth를 거칠 것이므로)
공간복잡도	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$ (시간복잡도와 동일)

• **Depth-limited search**(DLS) → 깊이를 제한한 DFS

Completeness	NO (goal이 있는 깊이 d보다 제한 깊이 l이 더 작으면)
Optimality	NO (제한 깊이가 작아서 goal을 못 찾는 경우)
시간복잡도	$O(b^l)$ (DFS와 유사)
공간복잡도	$O(bl)$ (DFS와 유사)

• **Iterative-deepening search**(IDS) → 반복적으로 깊이를 늘려가는 DLS
⇒ 앞 단계에서 본 node를 계속 봐서 비효율적일 수 있으나 실제 시간복잡도 관점에서 결국 영향을 주는 건 b^m ⇒ 선호되는 uninformed search

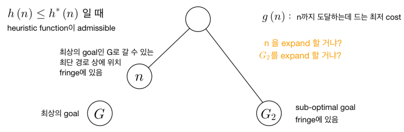
Completeness	YES (b가 유한하고, finite state space 안에 goal이 있으면)
Optimality	YES (step마다 동일한 cost이면)
시간복잡도	$O(b^d)$ (DFS와 유사)
공간복잡도	$O(bd)$ (DFS와 유사)

• **Bidirectional search** → start와 goal에서 둘 다 출발하여 찾는 경로

Completeness	YES (step마다 동일한 cost이고 두 search 모두 BFS이면)
Optimality	YES (step마다 동일한 cost이고 두 search 모두 BFS이면)
시간복잡도	$O(b^{d/2})$ ($b^{d/2} + b^{d/2} \ll b^d$)
공간복잡도	$O(b^{d/2})$ (BFS와 유사)

Informed Search ⇒ evaluation 함수에 따른 탐색 방법 유념

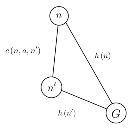
• 기본적으로 모든 search 전략은 어떠한 노드를 먼저 expansion 할 건지 그 순서에 따라 결정된다 (expansion의 후보인 fringe에서 선택하는 방법)
• **Best-first search**: $f(n)$ 값에 따라서 확장할 다음 노드를 고른다
$f(n)$을 어떻게 정의하느냐에 따라 여러 방법으로 나뉜다
(1) Greedy best-first search (2) A* search (3) Uniform cost search
($f(n)$: evaluation 함수, $g(n)$: 출발점으로부터 현재 노드로 오기까지의 cost, $h(n)$: heuristic 함수 → goal까지 가는 데 예상되는 cost 추정치)
• **Greedy best-first search**: $f(n) = h(n)$
→ expansion 할 때 이제까지 어떻게 왔는지 말고 앞으로 갈 것만 고려
→ complete 하지 않음 (단, finite space에서 반복되는 state 제외하는 graph search일 때는 complete)
→ 시간복잡도 $O(b^m)$, optimal 보장하지 않음
→ DFS와 유사할지는 몰라도 DFS는 아니다! (한쪽 방향으로 탐색하다가 heuristic cost가 더 안 좋아져서 더 좋은 heuristic cost의 다른 branch로 jump 할 수 있음 ⇒ 모두 돌아다녀야 하는 경우 존재 (공간복잡도: $O(b^m)$))
• **Uniform cost search**(= Dijkstra's algorithm): $f(n) = g(n)$
• **A* search**: $f(n) = g(n) + h(n)$
◊ **heuristic이 admissible하다** ⇔ 실제 최적의 길찾기 비용보다 같거나 작은 비용을 추정해야 한다 ⇔ **과대평가 하지 말아야 한다!** ($h(n) \le h*(n)$)
◊ heuristic이 admissible하면, tree search를 사용하는 A*는 optimal하다



$$f(G_2) = g(G_2) \quad (\because h(G_2) = 0 \quad G_2 \text{ 도 goal 이므로})$$
$$> g(G) = g(n) + \underbrace{h^*(n)}_{\text{실제 n에서 G까지의 cost}} \ge g(n) + h(n) \quad (\because h^*(n) \ge h(n))$$
$$= f(n)$$

$f(G_2) > f(n)$ 이므로 n을 expand 할 것임

◊ heuristic이 consistent하다 ⇔ 더 탐색할수록(밑으로 내려갈수록) 비용(f)이 감소하지 않아야 한다 ($h(n) \le c(n, a, n') + h(n')$)
◊ heuristic이 consistent하면, graph search를 사용하는 A*는 optimal하다



$h(n) \le c(n, a, n') + h(n')$ ⇔ h is consistent

$$g(n) + h(n) \le g(n) + c(n, a, n') + h(n')$$
$$= g(n') + h(n') = f(n')$$

$f(n) \le f(n')$ $\quad$ f is non decreasing along any path

◊ Every consistent heuristic is also admissible (역은 성립 안 함)
◊ Pathmax equation: $f(n') < f(n)$이어서 consistency 만족 못 하면 $f(n') = f(n)$으로 맞춰주자
◊ Consistence heuristic A* search에서 optimal cost를 $C*$라고 하면
(1) A*는 증가하는 f에 따라 등고선 안에 오는 state를 모두 탐색한다
(2) A*는 $f(n) < C^*$인 모든 node를 다 expand 한다
(3) A*는 $f(n) = C^*$인 일부 node를 expand 한다
(4) A*는 $f(n) > C^*$인 node를 하나도 expand 하지 않는다
⇒ 그래서 A*의 optimality가 보장(yes)되고, optimally efficient하다
⇒ $f(n) \le C^*$인 node가 무한하지 않으면 completeness(yes)
◊ 시간복잡도는 여전히 exponential이고, 모든 생성된 node를 다 메모리에 저장하므로 space가 중요한 문제가 된다 (공간복잡도가 꽤 큼)
• Weighted A* search: heuristic에 가중치를 둬서 좀 더 중요하게 생각하자
• **Memory-bounded heuristic search** ⇒ A* search에서 메모리 문제 해결
(1) **Iterative-deepening A***(IDA*): IDS와 유사, overhead 클 수 있음
→ DFS처럼 탐색하는데, depth 대신 f-cost를 늘려나가자
→ f-cost limit 안에서 goal 찾으면 반환, 못 찾으면 f-cost limit 늘리기
(2) **Recursive best-first search**(RBFS): A* search처럼 하되 이제까지 탐색한것 중에서 가장 좋은 1등 뿐만 아니라 2등(전도유망한 것)도 기억해 뒀다가 1등으로 간 곳이 2등보다 나빠지는 것 같으면 2등(전도유망한 것)으로 다시 가자
→ 공간복잡도: $O(bd)$, 반복되는 state를 check하지 못함
⇒ IDA*와 RBFS 모두 너무 적은 양의 메모리를 사용한다
(3) **Simplified Memory-bounded A***(SMA*): 사용 가능한 메모리가 다 찰 때까지 best leaf를 expand 하자
→ 메모리가 차면 가장 안 좋은 f-value를 지닌 leaf node를 삭제하자
→ 대신 잊힌 node의 f-value 값은 parent로 backup됨
→ 사용 가능한 메모리에서 complete하고 optimal함 (괜찮은 방법)
• 두 개의 admissible한 heuristic function 중에 모든 node에 관해 더 큰 값을 지니는 heuristic function을 선택하는 게 좋다
• Admissible heuristic은 좀 더 제약이 풀어진 relaxed version의 문제의 해결법의 정확한 cost로부터 나올 수 있다. (제약이 풀어진 문제의 optimal한 해결 비용이 실제 문제의 optimal한 해결 비용보다 크지는 않을 것이므로)
• Admissible heuristic은 subproblem의 해결법의 cost로부터 나올 수 있다

Intelligent Agents ⇒ 중요한 개념 위주로 정리

• **PEAS** → task environment에 관한 설명
(1) Performance measure (어떤 성능을 측정할지) (2) Environment (환경)
(3) Actuator (어떤 행동을 취할 수 있는지) (4) Sensor
• Environment type (1) Observable (환경이 agent에게 온전히 보이는지)
(2) Deterministic (어떤 action을 취했을 때 다음 state가 결정되는지) (3) static (action을 취할 당시에 환경이 변할 수 있는지) (4) Discrete (취할 수 있는 action이 finite한지, 시간적으로 input・output이 불연속적인지) (5) Single-agent (다른 agent가 있는지) (6) Sequential or episode (input을 sequence로 계속 유지하는지)
• **Simple reflex agent**: 단지 현재 인지한 것만을 바탕으로 action을 선택한다 (내부 기억이나 상태 정보가 없으므로 이전 percept는 영향 끼치지 X)
• **Model-based reflex agent**: 내부 상태를 가지므로 percept input 뿐만이 아니라 내부 상태와 함께 word knowledge를 가지고 현재 state를 update
• **Goal-based agent**: 현재 state 뿐만 아니라 미래의 action과 이를 취했을 때의 결과에 관한 바람직한 상황을 상정한다
• **Utility-based agent**: 목적 해결에 있어서 방법이 여러 개이거나 여러 목적이 있는 경우 효율성에 입각하여 행동의 일렬 구성 → 목표 도달 여부만 고려하는 goal-based와는 달리 utility를 이용해 goal에 얼마나 도달했는지
• **Learning agent**: percept에 근거하여 action을 선택하는 performance element와 현재의 수행 능력을 향상시키는 critics 존재
• 모든 agent는 learning을 통해 성능 향상 가능