CS 5100: Foundations of Artificial Intelligence

Search Problems and Solutions

Prof. Amy Sliva
October 6, 2011
Outline

• Review inference in FOL
  • Most general unifiers
  • Conversion to CNF
  • Unification algorithm
  • Backward chaining

• Search Problems
  • Uninformed search
  • Informed search (maybe...)
**Review:** Find a MGU for the following pairs (if one exists)

1. \( \text{Isa}(Fido, \text{Dog}) \)
   \( \text{Isa}(x, \text{Dog}) \)

2. \( \text{Isa}(x, \text{Dog}) \)
   \( \text{Isa}(y, z) \)

3. \( \text{Likes}(x, \text{Owner}(x)) \)
   \( \text{Likes}(\text{John}, \text{Owner}(Fido)) \)

4. \( \text{Likes}(x, \text{Owner}(y)) \)
   \( \text{Likes}(\text{John}, z) \)

5. \( \text{Likes}(x, \text{Owner}(y)) \)
   \( \text{Likes}(Fido, \text{Father}(\text{John})) \)
Review: Find a MGU for the following pairs (if one exists)

1. $\text{Isa}(Fido, \text{Dog})$
   $\text{Isa}(x, \text{Dog})$
   $\theta = \{x/Fido\}$

2. $\text{Isa}(x, \text{Dog})$
   $\text{Isa}(y,z)$
   $\theta = \{x/y, z/\text{Dog}\}$

3. $\text{Likes}(x, \text{Owner}(x))$
   $\text{Likes}(\text{John}, \text{Owner}(\text{Fido}))$
   None

4. $\text{Likes}(x, \text{Owner}(y))$
   $\text{Likes}(\text{John}, z)$
   $\theta = \{x/\text{John}, z/\text{Owner}(y)\}$

5. $\text{Likes}(x, \text{Owner}(y))$
   $\text{Likes}(\text{Fido}, \text{Father}(\text{John}))$
   None
Review: Steps in converting FOL to CNF

- Remove implications
- Move negation inward
- Standardize variable names
- Skolemize (drop existential quantifiers)
  - $\forall x \forall y \exists z \left[ \text{Knows}(x, z) \land \text{Knows}(y, z) \right]$ becomes $\forall x \forall y \left[ \text{Knows}(x, F(x,y)) \land \text{Knows}(y, F(x,y)) \right]$
  - $\forall x \exists z \forall y \left[ \text{Knows}(x, z) \land \text{Knows}(y, z) \right]$ becomes $\forall x \forall y \left[ \text{Knows}(x, F(x)) \land \text{Knows}(y, F(x)) \right]$
- Drop universal quantifiers
- Distribute $\lor$ over $\land$
Conversion to CNF Examples

1. \( \exists y \exists z \left[ \text{City}(z) \land \text{Team-from}(y, z) \land \text{Winner}(y, \text{SuperBowl}) \right] \)

2. \( \forall x \forall y \left[ \text{Student}(x) \land \text{TakingClass}(x, y) \implies \exists z \left[ \text{Grade}(z) \land \text{GradeFor}(x, y, z) \right] \right] \)

3. \( \forall w \left[ \exists z \left[ \text{House}(z) \land \left( \text{Owns}(w, z) \lor \text{Rents}(w, z) \right) \right] \implies \text{CreditWorthy}(w) \right] \)
Conversion to CNF Examples

1. \( \exists y \exists z [ \text{City}(z) \land \text{Team-from}(y, z) \land \text{Winner}(y, \text{SuperBowl}) ] \)  
   \( \text{City}(g2) \land \text{Team-from}(g1, g2) \land \text{Winner}(g1, \text{SuperBowl}) \)

2. \( \forall x \forall y [ \text{Student}(x) \land \text{TakingClass}(x, y) \Rightarrow \exists z [ \text{Grade}(z) \land \text{GradeFor}(x, y, z)] ] \)  
   \( (\neg \text{Student}(x) \lor \neg \text{TakingClass}(x, y) \lor \text{Grade}(z)) \land \)  
   \( (\neg \text{Student}(x) \lor \neg \text{TakingClass}(x, y) \lor \text{GradeFor}(x, y, z)) \)

3. \( \forall w [ \exists z [ \text{House}(z) \land (\text{Owns}(w, z) \lor \text{Rents}(w, z)) ] \Rightarrow \text{CreditWorthy}(w) ] \)  
   \( (\neg \text{House}(z) \lor \neg \text{Owns}(w, z) \lor \text{CreditWorthy}(w)) \land \)  
   \( (\neg \text{House}(z) \lor \neg \text{Rents}(w, z) \lor \text{CreditWorthy}(w)) \)
Unification Algorithm

function \textsc{Unify}(x, y, \theta) \textbf{returns} a substitution to make \( x \) and \( y \) identical

*inputs:* \( x \), a variable, constant, list, or compound

\( y \), a variable, constant, list, or compound

\( \theta \), the substitution built up so far

if \( \theta = \text{failure} \) then return failure

else if \( x = y \) then return \( \theta \)

else if Variable?(x) then return \textsc{Unify-Var}(x, y, \theta)

else if Variable?(y) then return \textsc{Unify-Var}(y, x, \theta)

else if Compound?(x) \textbf{and} Compound?(y) \textbf{and} Op(x) = Op(y) then

\hspace{1em} return \textsc{Unify}(\textsc{ARGS}[x], \textsc{ARGS}[y], \theta)

else if List?(x) \textbf{and} List?(y) then

\hspace{1em} return \textsc{Unify}(\textsc{REST}[x], \textsc{REST}[y], \textsc{Unify}(\textsc{FIRST}[x], \textsc{FIRST}[y], \theta))

else return failure
Unification Algorithm (cont.)

```plaintext
function UNIFY-VAR(var, x, \(\theta\)) returns a substitution
inputs: var, a variable
        x, any expression
        \(\theta\), the substitution built up so far

if \{var/val\} \(\in\) \(\theta\) then return UNIFY(val, x, \(\theta\))
else if \{x/val\} \(\in\) \(\theta\) then return UNIFY(var, val, \(\theta\))
else if OCCUR-CHECK?(var, x) then return failure
else return add \{var/x\} to \(\theta\) Using compose
```
Unification algorithm in action...

- Unify $\text{Likes}(y,x)$ with $\text{Likes}(x,John)$
Unification algorithm in action...

```plaintext
function UNIFY(x, y, θ) returns a substitution to make x and y identical
  inputs: x, a variable, constant, list, or compound
          y, a variable, constant, list, or compound
          θ, the substitution built up so far
  if θ = failure then return failure
  else if x = y then return θ
  else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) and COMPOUND?(y) and Op(x) = Op(y) then
    return UNIFY(ARGS[x], ARGS[y], θ)
  else if LIST?(x) and LIST?(y) then
    return UNIFY(REST[x], REST[y], UNIFY(FIRST[x], FIRST[y], θ))
  else return failure
```

call UNIFY(Likes(y,x), Likes(x,John), [ ])
return UNIFY([y,x], [x,John], [ ])
Unification algorithm in action...

function UNIFY(x, y, θ) returns a substitution to make x and y identical
inputs: x, a variable, constant, list, or compound
        y, a variable, constant, list, or compound
        θ, the substitution built up so far

if θ = failure then return failure
else if x = y then return θ
else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
else if COMPOUND?(x) and COMPOUND?(y) and Op(x) = Op(y) then
    return UNIFY(ARGS[x], ARGs[y], θ)
else if LIST?(x) and LIST?(y) then
    return UNIFY(REST[x], REST[y], UNIFY(FIRST[x], FIRST[y], θ))
else return failure

call UNIFY([y,x], [x,John], [ ])
return UNIFY([x], [John], UNIFY([y],[x],[ ]))
Unification algorithm in action...

```python
function Unify(x, y, θ) returns a substitution to make x and y identical
    inputs: x, a variable, constant, list, or compound
             y, a variable, constant, list, or compound
             θ, the substitution built up so far
    if θ = failure then return failure
    else if x = y then return θ
    else if VARIABLE?(x) then return Unify-VAR(x, y, θ)
    else if VARIABLE?(y) then return Unify-VAR(y, x, θ)
    else if COMPOUND?(x) and COMPOUND?(y) and Op(x) = Op(y) then
        return Unify(ARGs[x], ARGs[y], θ)
    else if LIST?(x) and LIST?(y) then
        return Unify(REST[x], REST[y], Unify(FIRST[x], FIRST[y], θ))
    else return failure
```

call Unify(y, x, [ ])
return Unify-VAR(y, x, [ ])
Unification algorithm in action...

```plaintext
function UNIFY-VAR(var, x, θ) returns a substitution
  inputs: var, a variable
           x, any expression
           θ, the substitution built up so far
  if {var/val} ∈ θ then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return failure
  else return add {var/x} to θ # Using compose
```

call UNIFY-VAR(y,x,[ ])
return θ = {y/x}
Unification algorithm in action...

- Unify \( Likes(y,x) \) with \( Likes(x,John) \)
  
  - First step: \( \theta = \{ y/x \} \)
Unification algorithm in action...

```plaintext
function UNIFY(x, y, θ) returns a substitution to make x and y identical
    inputs: x, a variable, constant, list, or compound
            y, a variable, constant, list, or compound
            θ, the substitution built up so far
    if θ = failure then return failure
    else if x = y then return θ
    else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
    else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
    else if COMPOND?(x) and COMPOND?(y) and Op(x) = Op(y) then
        return UNIFY(ARGS[x], ARGS[y], θ)
    else if LIST?(x) and LIST?(y) then
        return UNIFY(REST[x], REST[y], UNIFY(FIRST[x], FIRST[y], θ))
    else return failure
```
call UNIFY([y,x], [x,John], [])
return UNIFY(x, John, θ = {y/x})
Unification algorithm in action...

function Unify\( (x, y, \theta) \) returns a substitution to make \( x \) and \( y \) identical

inputs: \( x \), a variable, constant, list, or compound
\( y \), a variable, constant, list, or compound
\( \theta \), the substitution built up so far

if \( \theta = \text{failure} \) then return failure
else if \( x = y \) then return \( \theta \)
else if VARIABLE?\( (x) \) then return Unify-Var\( (x, y, \theta) \)
else if VARIABLE?\( (y) \) then return Unify-Var\( (y, x, \theta) \)
else if COMPUND?\( (x) \) and COMPUND?\( (y) \) and \( Op(x) = Op(y) \) then
  return Unify\( (\text{ARGS}[x], \text{ARGS}[y], \theta) \)
else if LIST?\( (x) \) and LIST?\( (y) \) then
  return Unify\( (\text{REST}[x], \text{REST}[y], \text{UNIFY}((\text{FIRST}[x], \text{FIRST}[y], \theta)) \)
else return failure

call Unify(x, John, \theta = \{y/x\})
return Unify-Var(x, John, \theta = \{y/x\})
Unification algorithm in action...

```plaintext
function UNIFY-VAR(var, x, θ) returns a substitution
  inputs: var, a variable
           x, any expression
           θ, the substitution built up so far
  if {var/val} ∈ θ then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return failure
  else return add {var/x} to θ # Using compose
```

call UNIFY-VAR(x,John, θ = {y/x})
Unification algorithm in action...

- Unify $\text{Likes}(y,x)$ with $\text{Likes}(x,John)$

  - First step: $\theta = \{y/x\}$
  - Second step: $\text{unify-var}(x, John, \theta)$

  - add$\{x/John\}$ to $\theta$ yields what?
    - $\text{Push}(\text{newpair}, \theta) \rightarrow \{x/John, y/x\}$
    - $\text{Compose}(\text{newpair}, \theta)$: s.t. when the lhs (L) of the new pair appears in the rhs (R) of an existing element of theta, replace R with $\text{subst}(\text{newpair}, R) \rightarrow \{x/John, y/John\}$
Unification algorithm in action...

function UNIFY-VAR(var, x, θ) returns a substitution
  inputs: var, a variable
          x, any expression
          θ, the substitution built up so far
  if {var/val} ∈ θ then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return failure
  else return add {var/x} to θ  # Using compose

call UNIFY-VAR(x, John, θ = {y/x})
return θ = {x/John, y/John}
Backward chaining algorithm

```
function FOL-BC-Ask(KB, goals, θ) returns a set of substitutions
    inputs: KB, a knowledge base
             goals, a list of conjuncts forming a query
             θ, the current substitution, initially the empty substitution { }
    local variables: ans, a set of substitutions, initially empty

    if goals is empty then return {θ}
    q' ← SUBST(θ, FIRST(goals))
    for each r in KB where STANDARDIZE-Apart(r) = ( p₁ ∧ ... ∧ pₙ ⇒ q)
        and θ' ← UNIFY(q, q') succeeds
            ans ← FOL-BC-Ask(KB, [p₁, ..., pₙ | REST(goals)], COMPOSE(θ, θ')) ∪ ans
    return ans
```
Some slight changes/corrections

• NOTE:

  \[\text{for each } r \text{ in } KB \text{ where } \text{STANDARDIZE-APART}(r) = (p_1 \land \ldots \land p_n \Rightarrow q)\]

  and \( \theta' \leftarrow \text{UNIFY}(q, q') \) succeeds

  \textbf{Should be:} for each \( r, \theta \) combination in \( KB \) where ...

• An empty \( \theta = [\ ] \) means an exact match, so \( \text{subst}(\theta, p) \) does nothing

• Why can’t we standardize apart once at the beginning?

• Remember, backward chaining is a depth-first search procedure
  • Search methods can be used to solve many problems in AI
Syntax of AND/OR trees

How do we search for backward chaining solutions?
• Facts are queried first (in the order they appear)
• Then rules in the order they appear
• Premises are also “solved” in the order they appear

Alternating node types (except for leaf nodes)
• A goal/subgoal node—**OR node** representing all the ways the goal can be proved
  • Children are rules or facts
• A rule node—**AND node** since the premises are conjoined
  • Children are subgoals
• Fact node—**leaf node** that satisfies a subgoal
Example: AND/OR tree for backward chaining process

- NOTE: Order of the KB is important

R1. $\text{House}(x) \land \text{Owns}(y, x) \Rightarrow \text{Cr}(y)$
R2. $\text{House}(x) \land \text{Rents}(y, x) \land \text{NoDebts}(y) \Rightarrow \text{Cr}(y)$
R3. $\text{Rich}(x) \Rightarrow \text{Cr}(x)$
F1. $\text{House}(H33)$
F2. $\text{House}(H21)$
F3. $\text{Owns}(\text{John}, H21)$
F4. $\text{Rents}(\text{Sally}, \text{Apt 12})$
F5. $\text{Owns}(\text{Max}, \text{Tent33})$
F6. $\text{Rich}(\text{Mike})$
F7. $\text{Rents}(\text{Jim}, H33)$

Query: $\text{Cr}(z)$ who is credit worthy??
Solutions: \([\{z = \text{John}\}, \{z = \text{Mike}\}]\)
Search problems

- Classic search problems
  - Missionaries and cannibals
  - 8-puzzle
  - 4 color map problem
  - Path finding
  - Resource allocation
  - Adversarial games extend the basic model

- Relationship to operations research
  - Factor scheduling (e.g., painting cars)
  - Logistics (e.g., delivering cars to dealers)
- Finding optimal or good schedules
Search framework for problem solving

- **Problem formulation:**
  - A set of discrete states (usually finite, but often very large) that define the problem space
  - A designated start state
  - A subset of states designated as goal states
  - A set of operators or actions the agent can perform at each state
  - A transition function $\text{Next: State} \times \text{OP} \rightarrow \text{State}$

- **Objective:**
  - Find a sequence of operators that, if applied in the start state, will lead to a goal state
    - **Optimal** solution—find the **shortest** or **least-cost** such sequence
    - Cost determined by agent’s performance measure
Explore the problem space using a search tree

- State space forms a **directed graph** or tree structure
  - Root = start state
  - Each node represents a state
  - Actions are branches—children are all the possible next-states

```plaintext
function TREE-SEARCH(problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
```
Graph search (dynamic programming)

- Graph contains at most **one copy** of each state
- Growing tree directly on state-space graph

```
function GRAPH-Search(problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
    ```
Example: The 8-puzzle

- States?
- Actions?
- Goal test?
- Path cost?

Start State

Goal State
Example: The 8-puzzle

- **States?** Locations of tiles
- **Actions?** Move blank left, right, up, down
- **Goal test?** ==Goal State (given)
- **Path cost?** 1 per move

[NOTE: optimal solution of n-puzzle family is NP-hard]
Example: Path finding

- Sample problem
  - On holiday in Romania; currently in Arad
  - Flight leaves tomorrow from Bucharest

- Formulate goal:
  - Be in Bucharest

- Formulate problem:
  - **States**—cities in Romania
  - **Actions**—drive between cities

- Find solution:
  - Sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest
Example: Path finding (cont.)
Solve path finding with tree search

(a) The initial state
Solve path finding with tree search

(b) After expanding Arad
Solve path finding with tree search

(c) After expanding Sibiu
Implementation: states vs. nodes

- A state is a (representation of) a physical configuration
- A node is a data structure constituting part of a search tree
  - Representative data structure must include state, parent node, action, path cost
  - More than one node can correspond to a single world state
Search strategies

• Search strategy defined by picking the order of node expansion

• Strategies are evaluated along the following dimensions
  1. Completeness—does it always find a solution if one exists?
  2. Time complexity—number of nodes generated
  3. Space complexity—maximum number of nodes in memory
  4. Optimality—does it always find a least-cost solution?

• Time and space complexity are measured in terms of
  • $b$—Maximum branching factor for the search tree
  • $d$—Depth of the least-cost solution
  • $m$—Maximum depth of the state space (may be $\infty$)
Uninformed search strategies

- **Uninformed** (blind) search strategies use only the information available in the problem definition

- Search strategies
  1. Breadth-first search
  2. Uniform-cost search
  3. Depth-first search
  4. Depth-limited search
  5. Iterative deepening search
Breadth-first search

• Expand **shallowest** unexpanded node
  • All nodes at a given depth expanded before any at the next level
• Implementation
  • *Frontier* is a FIFO queue, i.e., new successors go at the end
Breadth-first search

• Expand **shallowest** unexpanded node
  • All nodes at a given depth expanded before any at the next level

• Implementation
  • *Frontier* is a FIFO queue, i.e., new successors go at the end
Breadth-first search

• Expand **shallowest** unexpanded node
  • All nodes at a given depth expanded before any at the next level

• Implementation
  • *Frontier* is a FIFO queue, i.e., new successors go at the end
Breadth-first search

• Expand **shallowest** unexpanded node
  • All nodes at a given depth expanded before any at the next level

• Implementation
  • *Frontier* is a FIFO queue, i.e., new successors go at the end
Properties of breadth-first search

- Complete? Yes (if $b$ is finite)
- Time? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$
- Space? $O(b^{d+1})$—keeps every node in memory
- Optimal? Yes (if cost is non-decreasing function of depth)

- **Space** is the bigger problem (more than time)
  - Suppose $b = 10$, a node uses 1000 bytes, and 1 million nodes per second can be generated
    - Solution at depth 10 takes 3 hours and uses 10 terabytes
    - At level 12 it is 13 days and 1 petabyte; at 14 it is 3.5 years, 99 petabytes
  - **Exponential complexity** bound search problems cannot be solved by uninformed strategies—not scalable
Uniform-cost search

- Expand **least-cost** unexpanded node
- Implementation
  - *Frontier* is a queue ordered by path cost
- Equivalent to breadth-first search if step costs are all equal
  - Complete? **Yes**, if step cost $\geq \varepsilon$
  - Time? # of nodes with $g \leq$ cost of optimal solution, $O\left( b^{\text{ceiling}\left( \frac{C^*}{\varepsilon} \right)} \right)$
    where $C^*$ is the cost of the optimal solution
  - Space? # of nodes with $g \leq$ cost of optimal solution, $O\left( b^{\text{ceiling}\left( \frac{C^*}{\varepsilon} \right)} \right)$
  - Optimal? **Yes**—nodes are expanded in increasing order of $g(n)$
Depth-first search

• Expand deepest unexpanded node
• Implementation
  • *Frontier* is a LIFO queue (stack), i.e., new successors go at the front
Depth-first search

- Expand deepest unexpanded node
- Implementation
  - *Frontier* is a LIFO queue (stack), i.e., new successors go at the front
Depth-first search

- Expand deepest unexpanded node
- Implementation
  - *Frontier* is a LIFO queue (stack), i.e., new successors go at the front
Depth-first search

• Expand deepest unexpanded node
• Implementation
  • *Frontier* is a LIFO queue (stack), i.e., new successors go at the front
Depth-first search

- Expand deepest unexpanded node
- Implementation
  - *Frontier* is a LIFO queue (stack), i.e., new successors go at the front
Depth-first search

- Expand deepest unexpanded node
- Implementation
  - *Frontier* is a LIFO queue (stack), i.e., new successors go at the front
Depth-first search

- Expand deepest unexpanded node
- Implementation
  - *Frontier* is a LIFO queue (stack), i.e., new successors go at the front
Depth-first search

• Expand deepest unexpanded node
• Implementation
  • *Frontier* is a LIFO queue (stack), i.e., new successors go at the front
Depth-first search

• Expand deepest unexpanded node
• Implementation
  • *Frontier* is a LIFO queue (stack), i.e., new successors go at the front
Depth-first search

- Expand deepest unexpanded node
- Implementation
  - *Frontier* is a LIFO queue (stack), i.e., new successors go at the front
Depth-first search

- Expand deepest unexpanded node
- Implementation
  - *Frontier* is a LIFO queue (stack), i.e., new successors go at the front

![Diagram of a depth-first search tree]
Depth-first search

- Expand deepest unexpanded node
- Implementation
  - *Frontier* is a LIFO queue (stack), i.e., new successors go at the front
Properties of depth-first search

• Complete? **No**—fails in infinite depth spaces or spaces with loops (NOTE: our algorithm prevents loops)
  • Modify to avoid repeated states along path—complete in finite spaces

• Time? **$O(b^m)$**—terrible if $m$ is much larger than $d$
  • Solutions are dense, so may be much faster than breadth-first

• Space? **$O(bm)$**—linear space!

• Optimal? **No**
Depth-limited search

- Depth-first search with **depth limit** $l$
  - Nodes at depth $l$ have no successors

```
function Depth-Limited-Search(problem, limit) returns soln/fail/cutoff
  Recursive-DLS(Make-Node(InitialState[problem]), problem, limit)

function Recursive-DLS(node, problem, limit) returns soln/fail/cutoff
  cutoff-occurred? ← false
  if Goal-Test[problem](State[node]) then return Solution(node)
  else if Depth[node] = limit then return cutoff
  else for each successor in Expand(node, problem) do
    result ← Recursive-DLS(successor, problem, limit)
    if result = cutoff then cutoff-occurred? ← true
    else if result ≠ failure then return result
  if cutoff-occurred? then return cutoff else return failure
```
Iterative deepening search

- Gradually **increase** the depth limit in depth-first search
- Combines benefits of breadth-first and depth-first search

```plaintext
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure

inputs: problem, a problem

for depth ← 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result
```
Iterative deepening search
Iterative deepening search

Limit = 1
Iterative deepening search
Iterative deepening search
Iterative deepening search

- **Number of nodes** generated in iterative deepening search to depth $d$ with branching factor $b$
  
  \[ N(\text{IDS}) = (d+1)b^0 + (d)b^1 + (d-1)b^2 + ... + (3)b^{d-2} + (2)b^{d-1} + (1)b^d \]

- **Time complexity** = $O(b^d)$
- Same as breadth-first search

- **Time complexity comparison** to breadth-first search
  
  - Suppose $b = 10$ and $d = 5$
    
    $N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$
    
    $N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110$
  
  - Overhead difference = $(123,450 - 111,110) / 111,110 = 11\%$
Properties of iterative deepening

- Complete?  **Yes**
- Time?  \((d+1)b^0 + (d)b^1 + (d-1)b^2 + ... + (3)b^{d-2} + (2)b^{d-1} + (1)b^d = \mathcal{O}(b^d)\)
- Space?  **\(\mathcal{O}(bd)\)**—same as depth-first search
- Optimal?  **Yes**, if step cost = 1
Summary of uninformed search algorithms

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breadth-First</th>
<th>Uniform-Cost</th>
<th>Depth-First</th>
<th>Depth-Limited</th>
<th>Iterative Deepening</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>$O(b^{d+1})$</td>
<td>$O(b^{C*/\varepsilon})$</td>
<td>$O(b^m)$</td>
<td>$O(b^l)$</td>
<td>$O(b^d)$</td>
</tr>
<tr>
<td>Space</td>
<td>$O(b^{d+1})$</td>
<td>$O(b^{C*/\varepsilon})$</td>
<td>$O(bm)$</td>
<td>$O(bl)$</td>
<td>$O(bd)$</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Informed search

• Uses **problem-specific** knowledge beyond the problem definition to find solutions more efficiently

• Outline of topics
  • Heuristics
  • Best-first search
  • Greedy best-first search
  • A* search

• Remember, a search strategy is defined by picking the **order of node expansion**
Best-first search

- Use an **evaluation function** $f(n)$ for each node
  - Estimate of “desirability”
  - Includes a **heuristic function** to add more knowledge to the problem $h(n) = \text{estimated cost of cheapest path from state at } n \text{ to the goal}$
    - Heuristics are problem-specific
- Expand **most desirable** unexpanded node
- Implementation
  - Order nodes in the frontier in decreasing order of desirability
- Special cases
  - Greedy best-first search
  - A* search
Heuristic for path-finding

- Romania example: estimate the straight-line distance from each node to Bucharest (the goal)
Greedy best-first search

• Evaluation function
  \[ f(n) = h(n) = \text{estimate cost from } n \text{ to } goal \]
  - e.g., \( h_{SLD}(n) = \text{straight line distance from } n \text{ to Bucharest} \)

• Search strategy
  - Expand the node that appears to be closest to the goal (i.e., has the lowest \( h(n) \))
Greedy best-first search

(a) The initial state

Arad
366
Greedy best-first search

(b) After expanding Arad
Greedy best-first search

(c) After expanding Sibiu

![Diagram showing the expansion of Sibiu with distances to other cities]
Greedy best-first search

(d) After expanding Fagaras
Properties of greedy best-first search

- Complete? **No**—can get stuck in loops e.g., Iasi → Neamt → Iasi → Neamt → ...
- Time? \(O(b^m)\), but a good heuristic can give dramatic improvement
- Space? \(O(b^m)\)—keeps all nodes in memory
- Optimal? **No**
A* Search

• Avoid expanding paths that are already expensive

• **Evaluation function** $f(n) = g(n) + h(n)$
  • $g(n)$ = cost of path so far to reach $n$
  • $h(n)$ = estimated cost from $n$ to goal
  • $f(n)$ = estimated total cost of path through $n$ to goal
A* Search

(a) The initial state

- Arad
  - Distance: 366 = 0 + 366
A* Search

(b) After expanding Arad

- Arad
  - Sibiu: 393 = 140 + 253
  - Timisoara: 447 = 118 + 329
  - Zerind: 449 = 75 + 374
A* Search

(c) After expanding Sibiu

- Arad
- Fagaras
- Oradea
- Rimnicu Vilcea
- Sibiu
- Timisoara
- Zerind

Costs:
- 646 = 280 + 366
- 415 = 239 + 176
- 671 = 291 + 380
- 413 = 220 + 193
- 447 = 118 + 329
- 449 = 75 + 374
A* Search

(d) After expanding Rimnicu Vilcea

Arad

Sibiu

Timisoara

Zerind

Arad 447=118+329

Sibiu 449=75+374

Fagaras

Oradea

Rimnicu Vilcea

Craiova

Pitesti

Sibiu

646=280+366 415=239+176 671=291+380

417=317+100

526=366+160 553=300+253

591=338+253 450=450+0
A* Search

(e) After expanding Fagaras

- Arad
- Sibiu
- Fagaras
- Oradea
- Rimnicu Vilcea
- Timisoara
- Zerind

Arad
- Sibiu
  - Bucharest
  - Oradea
  - Fagaras
- Sibiu
  - Bucharest
  - Oradea
- Fagaras
  - Bucharest
  - Oradea
- Oradea
- Rimnicu Vilcea
- Rimnicu Vilcea
- Pitesti
- Pitesti
- Sibiu
A* Search

(f) After expanding Pitesti
Properties of A*

- Complete? Yes (unless there are infinitely many nodes with $f(n) < C^*$
- Time? Exponential—depends on assumptions made about state space
- Space? Keeps all nodes in memory
- Optimal? Yes
Admissible heuristics

• Heuristic $h(n)$ is **admissible** if for every node $n$, $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from $n$
  • *Never overestimates* the cost to reach the goal, i.e., it is **optimistic**

• Example
  • $h_{SLD}(n)$ never overestimates the actual road distance by using straight-line distance

• **Theorem**
  • If $h(n)$ is admissible, A* using **TREE-SEARCH** is optimal
Optimality of A* (proof)

• Suppose some suboptimal goal $G_2$ is in the frontier. Let $n$ be an unexpanded node in the frontier s.t. $n$ is on a shortest path to an optimal goal $G$

• Proof:
  • $f(G_2) > f(G)$ from above
  • $h(n) \leq h^*(n)$ since $h$ is admissible
  • $g(n) + h(n) \leq g(n) + h^*(n)$
  • $f(n) \leq f(G)$
  • Hence $f(G_2) > f(n)$, and A* will never select $G_2$ for expansion
Consistent heuristics

- Heuristic is **consistent** if for every node $n$, every successor $n'$ of $n$ generated by action $a$, $h(n) \leq c(n,a,n') + h(n')$
- If $h$ is consistent we have
  \[
  f(n') = g(n') + h(n') \\
  = g(n) + c(n,a,n') + h(n') \\
  \geq g(n) + h(n) \\
  \geq f(n)
  \]
  - i.e., $f(n)$ is non-decreasing along any path

**Theorem**
- If $h(n)$ is consistent, $A^*$ using `GRAPH-SEARCH` is optimal
Generating admissible heuristics

- Possible heuristics for the 8-puzzle
  - Generate from a relaxed problem—fewer restrictions than original
    \[ h_1(n) = \text{number of misplaced tiles} \]
    \[ h_2(n) = \text{total Manhattan distance to the goal position} \]

\[
\begin{array}{ccc}
7 & 2 & 4 \\
5 & 6 & \\
8 & 3 & 1 \\
\end{array}
\quad
\begin{array}{ccc}
1 & 2 \\
3 & 4 & 5 \\
6 & 7 & 8 \\
\end{array}
\]

\[ h_1(S) = ? \]
\[ h_2(S) = ? \]
Generating admissible heuristics

- Possible heuristics for the 8-puzzle
  - Generate from a relaxed problem—fewer restrictions than original
  - $h_1(n) = \text{number of misplaced tiles}$
  - $h_2(n) = \text{total Manhattan distance to the goal position}$

\[
\begin{align*}
  h_1(S) &= 8 \\
  h_2(S) &= 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18 \\
  h^*(S) &= 26 \text{ so both are admissible heuristics (lower bounds)}
\end{align*}
\]