Search methods

• Uninformed search methods
  – Breadth-first search (BFS)
  – Depth-first search (DFS)
  – Iterative deepening (IDA)
  – Bi-directional search
  – Uniform cost search

• Informed (or heuristic) search methods:
  – Best first search with the heuristic function
Best-first search

Best-first search
• Driven by the evaluation function $f(n)$ to guide the search.
• incorporates a **heuristic function** $h(n)$ in $f(n)$
• heuristic function measures a potential of a state (node) to reach a goal

**Special cases** (differ in the design of evaluation function):
  – **Greedy search**
    
    $f(n) = h(n)$
  
  – **A* algorithm**
    
    $f(n) = g(n) + h(n)$
  
  + **iterative deepening** version of A*: **IDA**

A* search

• The problem with the **greedy search** is that it can keep expanding paths that are already very expensive.
• The problem with the **uniform-cost search** is that it uses only past exploration information (path cost), no additional information is utilized
• **A* search**
    
    $f(n) = g(n) + h(n)$
    
    $g(n)$ - cost of reaching the state
    $h(n)$ - estimate of the cost from the current state to a goal
    $f(n)$ - estimate of the path length
• **Additional A* condition**: admissible heuristic
    
    $h(n) \leq h^*(n)$ for all $n$
Optimality of A*

• In general, a heuristic function $h(n)$:
  Can overestimate, be equal or underestimate the true distance of a node to the goal $h^*(n)$

• Admissible heuristic condition
  – Never overestimate the distance to the goal !!!

\[ h(n) \leq h^*(n) \quad \text{for all } n \]

Example: the straight-line distance in the travel problem never overestimates the actual distance

Iterative deepening algorithm (IDA)

• Based on the idea of the limited-depth search, but
• It resolves the difficulty of knowing the depth limit ahead of time.

Idea: try all depth limits in an increasing order.
That is, search first with the depth limit $l=0$, then $l=1$, $l=2$, and so on until the solution is reached

Iterative deepening combines advantages of the depth-first and breadth-first search with only moderate computational overhead
Properties of IDA

- **Completeness:** Yes. The solution is reached if it exists. (the same as BFS)
- **Optimality:** Yes, for the shortest path. (the same as BFS)
- **Time complexity:**
  \[ O(1) + O(b) + O(b^2) + \ldots + O(b^d) = O(b^d) \]
  exponential in the depth of the solution \( d \)
  worse than BFS, but asymptotically the same
- **Memory (space) complexity:**
  \[ O(db) \]
  much better than BFS

IDA*

- Solves minimum cost-path problems with heuristics
- Iterative deepening version of A*

**Idea:**

- Performs **limited-cost depth-first search** for the current evaluation function limit
  - Keeps expanding nodes in the depth-first manner up to the evaluation function limit
- Progressively increases the **evaluation function limit** (instead of the depth limit)

**Problem:** the amount by which the evaluation limit should be progressively increased
IDA*

**Problem:** the amount by which the evaluation limit should be progressively increased

**Solutions:**
(1) peak over the previous step boundary to guarantee that in the next cycle some number of nodes are expanded
(2) Increase the limit by a fixed cost increment – say $\varepsilon$

Solution 1: peak over the previous step boundary to guarantee that in the next cycle more nodes are expanded

Properties:
- the choice of the new cost limit influences how many nodes are expanded in each iteration
- Assume I choose a limit such that at least 5 new nodes are examined in the next DFS run
- What is the problem here?
  - Fix: complete the search up to the limit to find the best
**IDA**

**Solution 2: Increase the limit by a fixed cost increment** \((\varepsilon)\)

Properties:
- What is bad? Too many or too few nodes expanded – no control of the number of nodes
- How good is the first solution found? Is it optimal?
  - The solution found first may differ by \(< \varepsilon\) from the optimal solution

Cost limit = \(k \varepsilon\)

next

**Constraint satisfaction search**
Search problem

A search problem:
- **Search space (or state space):** a set of objects among which we conduct the search;
- **Initial state:** an object we start to search from;
- **Operators (actions):** transform one state in the search space to the other;
- **Goal condition:** describes the object we search for

- **Possible metric on the search space:**
  - measures the quality of the object with respect to the goal

Constraint satisfaction problem (CSP)

Two types of search:
- **path search** (a path from the initial state to a state satisfying the goal condition)
- **configuration search** (a configuration satisfying goal conditions)

Constraint satisfaction problem (CSP)

- a configuration search problem where:
  - A **state** is defined by a **set of variables and their values**
  - **Goal condition** is represented by a **set constraints on possible variable values**

Special properties of the CSP lead to special search procedures we can design to solve them
Example of a CSP: N-queens

Goal: n queens placed in non-attacking positions on the board

Variables:
- Represent queens, one for each column:
  - \( Q_1, Q_2, Q_3, Q_4 \)
- Values:
  - Row placement of each queen on the board
    \( \{1, 2, 3, 4\} \)

Constraints:
- \( Q_i \neq Q_j \) Two queens not in the same row
- \( |Q_i - Q_j| \neq |i - j| \) Two queens not on the same diagonal

Satisfiability (SAT) problem

Determine whether a sentence in the conjunctive normal form (CNF) is satisfiable (can evaluate to true)
- Used in the propositional logic (covered later)

\[
(P \lor Q \lor \neg R) \land (\neg P \lor \neg R \lor S) \land (\neg P \lor Q \lor \neg T) \ldots
\]

Variables:
- Propositional symbols (P, R, T, S)
- Values: True, False

Constraints:
- Every conjunct must evaluate to true, at least one of the literals must evaluate to true

\[
(P \lor Q \lor \neg R) \equiv True , (\neg P \lor \neg R \lor S) \equiv True , \ldots
\]
Other real world CSP problems

Scheduling problems:
– E.g. telescope scheduling
– High-school class schedule

Design problems:
– Hardware configurations
– VLSI design

More complex problems may involve:
– real-valued variables
– additional preferences on variable assignments – the optimal configuration is sought

Exercise: Map coloring problem

Color a map using k different colors such that no adjacent countries have the same color

Variables: ?

• Variable values: ?

Constraints: ?
Map coloring

Color a map using \( k \) different colors such that no adjacent countries have the same color.

Variables:
- Represent countries
  - \( A, B, C, D, E \)
- Values:
  - \( k \) different colors
    - \{Red, Blue, Green,..\}

Constraints: ?

---

Map coloring

Color a map using \( k \) different colors such that no adjacent countries have the same color.

Variables:
- Represent countries
  - \( A, B, C, D, E \)
- Values:
  - \( k \) different colors
    - \{Red, Blue, Green,..\}

Constraints: \( A \neq B, A \neq C, C \neq E \), etc
 This is an example of a problem with **binary constraints**
A formulation of the search problem:

- **States.** Assignment (partial or complete) of values to variables.
- **Initial state.** No variable is assigned a value.
- **Operators.** Assign a value to one of the unassigned variables.
- **Goal condition.** All variables are assigned, no constraints are violated.

- **Constraints** can be represented:
  - Explicitly by a set of allowable values
  - Implicitly by a function that tests for the satisfaction of constraints

Search strategies for solving CSP

![Search tree for CSP](image)
Search strategies for solving CSP

- **Maximum depth of the tree (m)**: ?
- **Depth of the solution (d)**: ?
- **Branching factor (b)**: ?

Search strategies for solving CSP

- **Maximum depth of the tree**: Number of variables in the CSP
- **Depth of the solution**: Number of variables in the CSP
- **Branching factor**: if we fix the order of variable assignments, the branch factor depends on the number of their values
Search strategies for solving CSP

• What search algorithm to use: ?
  Depth of the tree = Depth of the solution = number of vars

• What search algorithm to use: Depth first search !!!
  • Since we know the depth of the solution
  • We do not have to keep large number of nodes in queues
Search strategies for solving CSP

- **What search algorithm to use:** Depth first search !!!
  - Since we know the depth of the solution
  - We do not have to keep large number of nodes in queues

Depth-first search strategy for CSP is also referred to as **backtracking**

Constraint consistency

**Question:**
- **When to check the constraints defining the goal condition?**
- The violation of constraints can be checked:
  - at the end (for the leaf nodes)
  - for each node of the search tree during its generation or before its expansion

**Checking the constraints for intermediate nodes:**
- More efficient: cuts branches of the search tree early
Constraint consistency

Checking the constraints for intermediate nodes:
- More efficient: cuts branches of the search tree early

```
Unassigned: Q_1, Q_2, Q_3, Q_4
Assigned: Q_1 = 1
```

```
Unassigned: Q_2, Q_3, Q_4
Assigned: Q_2 = 2, Q_3 = 2
```

CS 1571 Intro to AI
M. Hauskrecht
Constraint consistency
Another way to cut the search space and tree exploration
- Current variable assignments together with constraints restrict remaining legal values of unassigned variables
- The remaining legal and illegal values of variables may be inferred (effect of constraints propagates)

Unassigned: $Q_1, Q_2, Q_3, Q_4$
Assigned: $Q_1 = 1$

Unassigned: $Q_2, Q_3, Q_4$
Assigned: $Q_1 = 2$

We know that $Q_1 \neq Q_2$. Then node representing $Q_1 = 2, Q_2 = 2$ should not be generated.
Constraint consistency

Another way to cut the search space and tree exploration

- Current variable assignments together with constraints restrict remaining legal values of unassigned variables
- The remaining legal and illegal values of variables may be inferred (effect of constraints propagates)

- To prevent “blind” exploration we can keep track of the remaining legal values, so we know when the constraints will be violated and when to terminate the search

Constraint propagation

A state (more broadly) is defined:

- by a set of assigned variables, their values and
- a list of legal and illegal assignments for unassigned variables

Legal and illegal assignments can be represented:

- equations (value assignments) and
- disequations (list of invalid assignments)

\[ A = \text{Red}, \text{Blue} \quad C \neq \text{Red} \]

Constraints + assignments can entail new equations and disequations

\[ A = \text{Red} \quad \rightarrow \quad B \neq \text{Red} \]

**Constraint propagation:** the process of inferring of new equations and disequations from existing equations and disequations
Constraint propagation

• Assign A=Red

<table>
<thead>
<tr>
<th></th>
<th>Red</th>
<th>Blue</th>
<th>Green</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td>×</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

✓ - equations  × - disequations
Constraint propagation

- Assign E = Blue

<table>
<thead>
<tr>
<th></th>
<th>Red</th>
<th>Blue</th>
<th>Green</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

Constraint propagation

- Assign E = Blue

<table>
<thead>
<tr>
<th></th>
<th>Red</th>
<th>Blue</th>
<th>Green</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Constraint propagation

- Assign F=Green

<table>
<thead>
<tr>
<th></th>
<th>Red</th>
<th>Blue</th>
<th>Green</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>✓</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>✗</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>✗</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Constraint propagation

- Assign F=Green

<table>
<thead>
<tr>
<th></th>
<th>Red</th>
<th>Blue</th>
<th>Green</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>B</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>C</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>F</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Conflict !!! No legal assignments available for B and C

Constraint propagation

- We can derive remaining legal values through propagation

<table>
<thead>
<tr>
<th></th>
<th>Red</th>
<th>Blue</th>
<th>Green</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>✓</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>C</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>✗</td>
<td></td>
</tr>
</tbody>
</table>

B=Green
C=Green
Constraint propagation

- We can derive remaining legal values through propagation

<table>
<thead>
<tr>
<th></th>
<th>Red</th>
<th>Blue</th>
<th>Green</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>B</td>
<td>x</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>C</td>
<td>x</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>D</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>x</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>F</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

B=Green  
C=Green  
F=Red