CSP, linear programming, games

Lirong Xia

Rensselaer

Spring, 2017
Project 1

➢ Admissibility must be satisfied
  • Otherwise your A* can be wrong on some instances
  • It doesn’t mean that your A* is always wrong---you might be lucky on one test

➢ Good heuristics
  • Consistent
  • Easy to compute
  • What if I use BFS to compute h* as the heuristic?
Constraint Satisfaction Problems

- Standard search problems:
  - State is a “black box”: arbitrary data structure
  - Goal test: any function over states
  - Successor function can be anything

- Constraint satisfaction problems (CSPs):
  - A special subset of search problems
  - State is defined by variables $X_i$ with values from a domain $D$ (sometimes $D$ depends on $i$)
  - Goal test is a set of constraints specifying allowable combinations of values for subsets of variables

- Allows useful general-purpose algorithms with more power than standard search algorithms
Example: Map-Coloring

- Variables: $WA, NT, Q, NSW, V, SA, T$
- Domains: $D = \{ \text{red, green, blue} \}$
- Constraints: adjacent regions must have different colors
  
  $WA \neq NT$

  $(WA, NT) \in \{(\text{red, green}), (\text{red, blue}), (\text{green, red}), \ldots\}$

- Solutions are assignments satisfying all constraints, e.g.:
  
  $\begin{cases} 
  WA = \text{red}, NT = \text{green}, Q = \text{red}, NSW = \text{green}, \\
  V = \text{red}, SA = \text{blue}, T = \text{green} 
  \end{cases}$
Constraint Graphs

- Binary CSP: each constraint relates (at most) two variables
- Binary constraint graph: nodes are variables, arcs show constraints
Backtracking Search

- Idea 1: only consider a single variable at each point
  - Variable assignments are commutative, so fix ordering
  - Consider assignments to a single variable at each step

- Idea 2: only allow legal assignments at each point
  - “Incremental goal test”

- DFS for CSPs with these two improvements is called backtracking search
Improving Backtracking

- General-purpose ideas give huge gains in speed

- Ordering:
  - Minimum remaining values (MRV)
  - least constraining value

- Filtering: Can we detect inevitable failure early?
  - forward checking
  - constraint propagation

- Structure of the problem
Filtering: Forward Checking

- Idea: keep track of remaining values for unassigned variables (using immediate constraints)
- Idea: terminate when any variable has no legal values
Problem with forward checking

- Forward checking propagates information from assigned to unassigned variables, but doesn’t provide early detection for all failures:

  - NT and SA cannot both be blue!
  - Why didn’t we detect this yet?
Today’s schedule

- CSP
  - constraint propagation
  - tree-structure constraint graph

- Search as optimization
  - linear programming

- Adversarial search: game play
Consistency of An Arc

- An arc $X \rightarrow Y$ is consistent iff for every $x$ in the tail there is some $y$ in the head which could be assigned without violating a constraint.

- Forward checking = Enforcing consistency of each arc pointing to the new assignment.
A simple form of propagation makes sure all arcs are consistent:

- If V loses a value, neighbors of V need to be rechecked!
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment
- Might be time-consuming

Delete from tail!
Use arc consistency as a subroutine

- In Backtracking search, before expanding a node (choosing a value for a variable)
- Apply arc consistency as much as possible until all arcs have been checked
  - once some values are removed for a variable, all constraints involving this variable needs to be re-checked
- Detect early failures
Limitations of Arc Consistency

- After running arc consistency:
  - Can have one solution left
  - Can have multiple solutions left
  - Can have no solutions left (and not know it)
Improving Backtracking

- General-purpose ideas give huge gains in speed

- Ordering:
  - Minimum remaining values (MRV)
  - least constraining value

- Filtering: Can we detect inevitable failure early?
  - forward checking search

- Structure of the problem
  - constraint graph is a tree
• Stage 1: moving upward, cross out the values that cannot work with the subtree below that node
• Stage 2: if a value remains at the root, there is a solution: go downward to pick a solution
Recap: CSP

- A special search problem
  - constraints presented by a graph

- Trick 1: backtracking
  - DFS with fixed order, choose one value in every step

- Trick 2: min remaining values, least constraining value

- Trick 3: early detection of failure
  - forward checking: detect consistency of the new assignment
  - constraint propagation: recursively remove illegal values

- Tractable special case: tree-structured graph
Linear programming

- Search for an assignment with highest objective value
The last battle

<table>
<thead>
<tr>
<th></th>
<th>strength</th>
<th>minerals</th>
<th>gas</th>
<th>supply</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zealot</td>
<td>1</td>
<td>100</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Stalker</td>
<td>2</td>
<td>125</td>
<td>50</td>
<td>2</td>
</tr>
<tr>
<td>Archon</td>
<td>10</td>
<td>100</td>
<td>300</td>
<td>4</td>
</tr>
</tbody>
</table>

Available resource:

<table>
<thead>
<tr>
<th></th>
<th>mineral</th>
<th>gas</th>
<th>supply</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2000</td>
<td>2000</td>
<td>30</td>
</tr>
</tbody>
</table>

How to maximize the total strength of your troop?
Computing the optimal solution

- **Variables**
  - \( x_Z \): number of Zealots
  - \( x_S \): number of Stalkers
  - \( x_A \): number of Archons

- **Objective**: maximize total strength
  \[
  \text{max } 1x_Z + 2x_S + 10x_A
  \]

- **Constraints**
  - mineral: \( 100x_Z + 125x_S + 100x_A \leq 2000 \)
  - gas: \( 0x_Z + 50x_S + 300x_A \leq 1500 \)
  - supply: \( 2x_Z + 2x_S + 4x_A \leq 30 \)
  - \( x_Z, x_S, x_A \geq 0, \text{ integers} \)
Linear programming (LP)

- **Given**
  - Variables $x$: a row vector of $m$ positive real numbers
  - Parameters (fixed)
    - $c$: a row vector of $m$ real numbers
    - $b$: a column vector of $n$ real numbers
    - $A$: an $n \times m$ real matrix

- **Solve**
  $$\text{max} \quad cx^T$$
  $$\text{s.t.} \quad Ax^T \leq b, \quad x \geq 0$$

- **Solutions**
  - $x$ is a **feasible solution**, if it satisfies all constraints
  - $x$ is an **optimal solution**, if it maximizes the objective function among all feasible solutions
General tricks

- Possibly negative variable $x$
  - $x = y - y'$
- Minimizing $cx^T$
  - $\max -cx^T$
- Greater equals to $Ax^T \geq b$
  - $-Ax^T \leq -b$
- Equation $Ax^T = b$
  - $Ax^T \geq b$ and $Ax^T \leq b$
- Strict inequality $Ax^T < b$
  - no "theoretically perfect" solution
  - $Ax^T \leq b - \varepsilon$
Integrality constraints

- Integer programming (IP): all variables are integers
- Mixed integer programming (MIP): some variables are integers
Efficient solvers

- LP: can be solved efficiently
  - if there are not too many variables and constraints

- IP/MIP: some instances might be hard to solve
  - practical solver: CPLEX free for academic use!
Game playing

- Games: Starcraft II, Pacman, Chess, Go, Poker, Texas hold’em,…
- Rich tradition of creating game-playing programs in AI
- Many similarities to search
- Many games studied
  - have two players,
  - are zero-sum: what one player wins, the other loses
  - have perfect information: the entire state of the game is known to both players at all times
- Recently more interest in other games
  - Esp. games without perfect information; e.g., poker
    - Need probability theory, game theory for such games
“Sum to 2” game

- Player 1 moves, then player 2, finally player 1 again
- Move = 0 or 1
- Player 1 wins if and only if all moves together sum to 2

Player 1’s utility is in the leaves; player 2’s utility is the negative of this