Search Strategies: Lookahead
Summary

- Introduction and Consistency Levels
- Backtracking
- Look-Ahead
Approximate Inference and Search

Need to take chances

- Complete inference (e.g., strong n-consistency) ensures no dead-end in extending partial solutions to complete solutions
- However, strong i-consistency is exponential (in the number of variables) \(\rightarrow\) not practical
- Approximate Inference is polynomial but we still need to search for a solution
- search: proceed by trial and errors
Running Example: Map-Coloring

Variables $WA, NT, Q, NSW, V, SA, T$

Domains $D_i = \{\text{red, green, blue}\}$

Constraints: adjacent regions must have different colors

e.g., $WA \neq NT$ (if the language allows this), or $(WA, NT) \in \{(\text{red, green}), (\text{red, blue}), (\text{green, red}), (\text{green, blue}), \ldots\}$
Example: Map-Coloring contd.

Solutions are assignments satisfying all constraints, e.g.,
\[
\{WA = red, NT = green, Q = red, \\
NSW = green, V = red, SA = blue, T = green\}
\]
Binary CSP: each constraint relates at most two variables
Naive choice

States are defined by the values assigned so far

- **Initial state:** the empty assignment, \( \{\} \)
- **Successor function:** assign a value to an unassigned variable that does not conflict with current assignment.
  \[ \Rightarrow \] fail if no legal assignments (not fixable!)
- **Goal test:** the current assignment is complete

1. This is the same for all CSPs! 😊
2. Every solution appears at depth \( n \) with \( n \) variables
   \[ \Rightarrow \] use depth-first search
3. Path is irrelevant, so can also use complete-state formulation
4. \( b = (n - \ell)d \) at depth \( \ell \), hence \( n!d^n \) leaves!!!
A better formulation

Variable assignments are **commutative**, i.e.,
\[ WA = red \text{ then } NT = green \] same as \[ NT = green \text{ then } WA = red \]

Only need to consider assignments to a single variable at each node

\[ \Rightarrow b = d \text{ and there are } d^n \text{ leaves} \]

◊ Depth-first search for CSPs with single-variable assignments is called **backtracking** search

◊ Backtracking search is the basic uninformed algorithm for CSPs. Can solve \( n \)-queens for \( n \approx 25 \)

◊ Variable ordering counts for performance
function Backtracking-Search \((csp)\) returns solution/failure
return Recursive-Backtracking(\{\}, \(csp\))

function Recursive-Backtracking \((assign., csp)\) returns soln/failure
if \(assign.\) is complete then return \(assign.\).
\(var \leftarrow \) Select-Unassigned-Variable(Variab\(\)les\([\)csp\)], \(assign.\), \(csp\))
for each \(value\) in Order-Domain-Values(\(var\), \(assign.\), \(csp\)) do
if \(value\) is cons. with \(assign.\) given Constraints[\(csp\)] then
add \{\(var = value\)\} to \(assign.\).
\(result \leftarrow \) Recursive-Backtracking\((assign., csp)\)
if \(result \neq failure\) then return \(result\)
remove \{\(var = value\)\} from \(assign.\).
return \(failure\)
Backtracking example
Backtracking example
Backtracking example
Backtracking example
General Goal: Reducing size of explored search space

Ingredients

- ordering variable and variables’ values
- local consistency (e.g., arc or path consistency)
- look-ahead, predict future inconsistencies
- look-back, where to backtrack
- tree decomposition, exploit problem structure
Ordering Variables: Minimum remaining values

Minimum remaining values (MRV):
choose the variable with the fewest legal values
Ordering Variables: Degree heuristic

Tie-breaker among MRV variables

Degree heuristic:
- choose the variable with the most constraints on remaining variables

MRV and DH can be applied to any CSP
Exercise: Dividing Integer

- Consider the following network $\mathcal{R}$
- Variables: $x, y, l, z$
- Domains:
  $D_x = D_y = \{2, 3, 4\}$, $D_l = \{2, 5, 6\}$, $D_z = \{2, 3, 5\}$
- Constraints: $z$ divides evenly $x, y, l$

Compute number of expanded nodes for assigning variable with different orderings:

- $d_1 = \{x, y, l, z\}$
- Use Minimum Remaining Values and degree heuristic to choose next value
Value ordering: Least constraining value

Given a variable, choose the least constraining value: the one that rules out the fewest values in the remaining variables.

Combining these heuristics makes 1000 queens feasible. Gains depend on the specific problem.
Consistency Level and Search Space

Good effects on Search Space Size

- Tighter constraints $\rightarrow$ smaller search space
- Given two equivalent network $\mathcal{R}$ and $\mathcal{R}'$
- if $\mathcal{R}' \subseteq \mathcal{R}$ then any solution path appearing in the search space of $\mathcal{R}'$ also appears in the search space of $\mathcal{R}$, for any ordering $d$.
- Higher level of consistency reduce the search space
Consistency Level and Search process

Negative effects on Searching

- Adding constraints requires more computation
- Each time a new variable is assigned need to check many more constraints
- If only binary constraints we never have more than $O(n)$ checks
- If $r$-ary constraints then we could have $O(n^{r-1})$ checks
Backtrack Free Search

Backtrack Free Network

- A network $\mathcal{R}$ is backtrack free if every leaf is a goal state.
- A DFS on a backtrack free network ensures a complete consistent assignment.
- E.g. $\mathcal{R} + \text{arc consistency} + \{z, x, y, l\} \rightarrow \text{backtrack free network}$
Look-Ahead

Look-Ahead Schemes

- Given approximate inference (arc consistency, path-consistency)
- Foresee impact of next move (which variable, which value)
- Impact: how next move restricts future assignment
- Efficient way to update information for choosing next variable/value
  - Can efficiently compute remaining legal values given current assignment
Look-Ahead Strategies

- Forward Checking
  - check unassigned variables separately
- Arc consistency look-ahead
  - propagate arc consistency
Look-ahead: Discussion

Discussion

- Incur extra cost for assigning values
  - need to propagate constraints
- Can restrict search space significantly
  - e.g., discover that a value makes a sub-problem inconsistent
  - remove values from future variables’ domains
- Usually no changes on worst case performance: trade-off between costs and benefits
Algorithm 1 Generalised Look-ahead

Require: A constraint network $\mathcal{R}$
Ensure: A solution or notification that the network is inconsistent

\begin{align*}
i & \leftarrow 1 \\
D'_i & \leftarrow D_i \\
\text{while } 1 \leq i \leq n \text{ do} \\
& x_i \leftarrow \text{SelectValue} \times \\
& \quad \text{if } x_i \text{ is null then} \\
& \qquad i \leftarrow i - 1 \\
& \quad \quad \text{Reset } D'_k \text{ for each } k > i \text{ to its value before } i \text{ was last instantiated} \\
& \text{else} \\
& \quad i \leftarrow i + 1 \\
& \quad \text{end if} \\
& \text{end while} \\
& \text{if } i \text{ is 0 then} \\
& \quad \text{return inconsistent} \\
& \text{else} \\
& \quad \text{return instantiated value for } \{x_1, \ldots, x_n\} \\
& \text{end if}
\end{align*}
Forward Checking

- most limited form of constraint propagation
- propagates the effect of a selected value to future variables separately
- if domains of one of future variables becomes empty, try next value for current variable.
Select Value Forward Checking

Algorithm 2 SelectValueForwardChecking

\[
\begin{align*}
a &\leftarrow D'_i \text{ select an arbitrary value} \\
\text{while } D'_i \neq \{\} \text{ do} & \\
\hspace{1cm} \text{for all } k, i < k \leq n \text{ do} & \\
\hspace{2cm} \text{for all } b, b \in D'_k \text{ do} & \\
\hspace{3cm} & \text{if } \langle x_{i-1}, x_i = a, x_k = b \rangle \text{ is not consistent then} \\
\hspace{4cm} & D'_k \leftarrow D'_k \setminus \{b\} \\
\hspace{1cm} & \text{end if} \\
\hspace{1cm} & \text{end for} \\
\hspace{1cm} & \text{if } D'_k = \{\} \text{ then} \\
\hspace{2cm} & \text{emptyDomain} \leftarrow \text{true} \\
\hspace{1cm} & \text{end if} \\
\hspace{1cm} & \text{end for} \\
\hspace{1cm} & \text{if } \text{emptyDomain} \text{ then} \\
\hspace{2cm} & \text{reset each } D'_k \text{ to its value before assigning } a \\
\hspace{1cm} & \text{else} \\
\hspace{2cm} & \text{return } a \\
\hspace{1cm} & \text{end if} \\
\hspace{1cm} & \text{end while} \\
& \text{return null}
\end{align*}
\]
Forward checking

Idea: Keep track of remaining legal values for unassigned variables

Terminate search when any variable has no legal values
Forward checking

Idea: Keep track of remaining legal values for unassigned variables

Terminate search when any variable has no legal values
Forward checking

**Idea:** Keep track of remaining legal values for unassigned variables

Terminate search when any variable has no legal values
Forward checking

Idea: Keep track of remaining legal values for unassigned variables

Terminate search when any variable has no legal values
Forward Checking: Example

Example (Graph Colouring Example)

- Variables: \(x_1, x_2, x_3, x_4, x_5, x_6, x_7\),
- Domains: \(D_{x_1} = \{R, B, G\}, D_{x_2} = D_{x_5} = \{B, G\}, D_{x_3} = D_{x_4} = D_{x_7} = \{R, B\}, D_{x_6} = \{R, G, Y\}\)
- Constraints: \(x_1! = x_2, x_1! = x_3, x_1! = x_4, x_1! = x_7, x_2! = x_6, x_3! = x_7, x_4! = x_5, x_4! = x_7, x_5! = x_6, x_5! = x_7\)

- \(x_1 = \text{red}\) reduces domains of \(x_3, x_4, x_7\)
- \(x_2 = \text{blue}\) no effects
- \(x_3 = \text{blue}\) (only available) makes \(x_7\) empty \(\rightarrow x_3\) dead-end
Search Strategies: Lookahead

Search for Constraint Propagation

Complexity of Forward Checking

- $O(ek^2)$
- $e_u$ consistency check for each value of each future variable $x_u$
- $k$ value for each future variables $O(e_u k)$
- $\sum_u e_u = e$ then $O(ek)$
- $k$ value for the current variable
Arc Consistency Look-Ahead

Arc Consistency Look ahead
- force **full** arc consistency on all remaining variables
- select a value for current variable \( x_i = a \)
- apply \( AC - 1 \) on all unassigned variables with \( x_i = a \)
- If a variable domain becomes empty reject current assignment
- can use \( AC - 3 \) or \( AC - 4 \) instead
Arc Consistency Look-Ahead Complexity

- Best algorithm for AC is \( AC - 4 \) complexity \( O(ek^2) \)
- worst case for Select Arc Consistency look-ahead is \( O(eK^3) \)
Example of AC Look-Ahead

Example (AC Look-Ahead for Map Colouring)
Example of AC Look-Ahead

Example (AC Look-Ahead for Map Colouring)
Example of AC Look-Ahead

Example (AC Look-Ahead for Map Colouring)
Example of AC Look-Ahead

Example (AC Look-Ahead for Map Colouring)
Exploiting problem structure in Lookahead

Definition (Cycle Cutset)

Given an undirected graph, a subset of nodes in the graph is a cycle cutset iff its removal result in an acyclic graph.

Exploiting problem structure

- Once a variable is assigned it can be removed from the graph (conditioning)
- If we remove a cycle-cutset the rest of the problem is a tree
- Can use arc consistency to solve that sub-problem
- We need to check all possible assignments of cycle-cutset variables and do arc propagation
- Complexity is still exponential but in the size of the cycle-cutset!
Exploiting problem structure: example

Example (Cycle Cut Set for Map Colouring)

**Conditioning:** instantiate a variable, prune its neighbors’ domains

Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree

Cutset size $c \implies$ runtime $O(d^c \cdot (n - c)d^2)$, very fast for small $c$
Exercise: AC Look-Ahead

AC Look-Ahead for Map Colouring

Use the AC-3 Algorithm to show that AC Look-Ahead detects an inconsistency on the partial assignment \{WA = red, V = blue\} for the Australia map colouring problem used above.
Exercise: Cycle cut set

Cycle cut set for Graph Colouring

Use the Cycle cut set algorithm to solve the graph colouring problem defined above.