Artificial Intelligence

Week 7: Backtracking

COMP30024

April 22, 2021

THE UNIVERSITY OF MELBOURNE

Backtracking

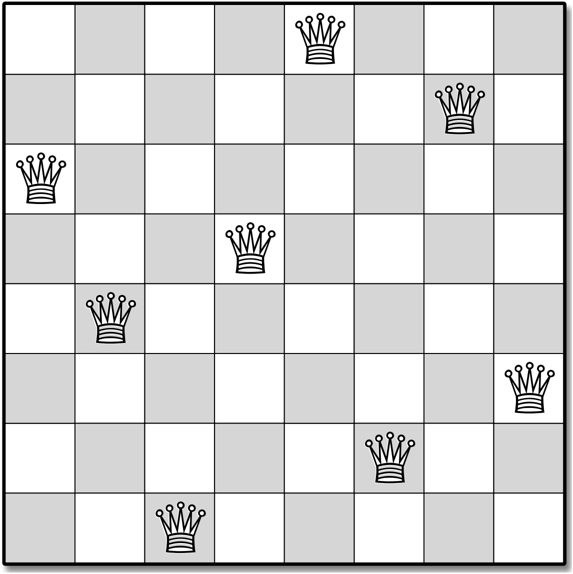
- Build sequence of decisions to solve a problem incrementally.

$$A = (a_1, a_2, \ldots a_k)$$

- To choose next solution component, a_{k+1}:
  - Recursively evaluate every possibility consistent with past decisions.
  - Choose the 'best' one, $A \leftarrow A \bigcup a_{k+1}$.
- Depth-first/recursive traversal with additional logic at each call that helps narrow the search space.

Backtracking implemented as DFS, using problem constraints to prune subtrees as early as possible.

n-Queens



n-queens

- Place n queens on an $n \times n$ chessboard with no possible captures.
- Recursive strategy:
  - Place queens on board one at a time, start with row $r = 0$.
  - To place r-th queen, try all possible squares in row r. If attacked by an earlier queen, pass, otherwise place.
  - Continue recursively, backtrack at dead-ends.
- Represent solution via recursion tree.
  - Each node represents a recursive subproblem.
  - Edges correspond to recursion calls.
  - Leaves correspond to dead-end partial solutions if $r < n$ and full solutions if $r = n$.

Backtracking

Solution finding via backtracking corresponds to DFS traversal of recursion tree.

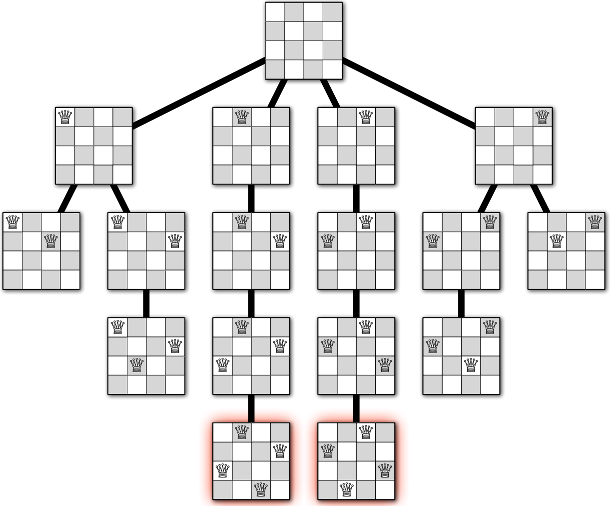


Figure 1: Backtracking execution for 4-queens problem. From Jeff Erickson's Algorithms, Ch. 2

Backtracking Pseudo-Code

- Compute all successor states to deepest partial solution, `A`.
- Extend partial solution `A` with potential successor states.
- Test if result is a solution. If not, recurse.
- Backtracking search is DFS + pruning.

```
def backtrack_dfs(A, k):
    if A = (a_1, a_2, ..., a_k) is a solution:
        return A
    else:
        k += 1
        # Enumerate all possible candidates at k+1
        # Impose some ordering criterion on trial candidates
        candidate_queue = construct_candidates(A, k)

        while candidate_queue is not None:
            A[k] = candidate_queue.pop()
            backtrack_dfs(A, k)
```

Constraint Satisfaction Problems

- Represent problem as a set of variables $X = \{X_1, \ldots X_n\}$
- Each variable can assume values in its respective domain, $D = \{D_1, \ldots, D_n\}$
- Constraints C express allowable relationships between variables.
- States in CSP defined by assignment of values variables X. CSP solved when values assigned to all X without violating constraints.
- Idea: Eliminate large regions of search space by identifying combinations of variables/values that violate constraints - backtracking!

n-Queens

- Variables: Position of queen in row r.
- Represent positions of queens through array $Q[1, \ldots, n]$. Element r of the array, $Q[r]$ indicates the position of the queen in row r.
- Domain: $Q[r] \in \{1, \ldots, n\}$.
- Constraints:
  - Let $i \neq j$ be row indexes.
  - $Q[i] \neq Q[j]$ (Same column placement forbidden).
  - $|Q[j] - Q[i]| \neq |j - i|$ (Diagonal placement forbidden).
- Constraint graph?
  - Nodes: Variables
  - Edges: Represent constraints between variables.

n-Queens

State Size?

- Stupid: 2^{n^2} (n^2 squares, 2 possible states for each).

n-Queens

State Size?

- Stupid: 2^{n^2} (n^2 squares, 2 possible states for each).
- Very Very Bad: $\binom{n^2}{n} = \frac{n^2!}{(n^2-n)!n!}$ (Permutations of n queens on $n \times n$ board).

n-Queens

State Size?

- Stupid: 2^{n^2} (n^2 squares, 2 possible states for each).
- Very Very Bad: $\binom{n^2}{n} = \frac{n^2!}{(n^2-n)!n!}$ (Permutations of n queens on $n \times n$ board).
- Very Bad: n^n (1 queen per row)

n-Queens

State Size?

- Stupid: 2^{n^2} (n^2 squares, 2 possible states for each).
- Very Very Bad: $\binom{n^2}{n} = \frac{n^2!}{(n^2-n)!n!}$ (Permutations of n queens on $n \times n$ board).
- Very Bad: n^n (1 queen per row)
- Bad: $n!$ (1 queen per row and column)

n-Queens

State Size?

- Stupid: 2^{n^2} (n^2 squares, 2 possible states for each).
- Very Very Bad: $\binom{n^2}{n} = \frac{n^2!}{(n^2-n)!n!}$ (Permutations of n queens on $n \times n$ board).
- Very Bad: n^n (1 queen per row)
- Bad: $n!$ (1 queen per row and column)
- Ok: $n(n-2)(n-4)... = \prod_{i=0}^{\lceil n/2 \rceil -1}(n-2i)$. (Each queen eliminates at least two squares in next row).

n-Queens

State Size?

- Stupid: 2^{n^2} (n^2 squares, 2 possible states for each).
- Very Very Bad: $\binom{n^2}{n} = \frac{n^2!}{(n^2-n)!n!}$ (Permutations of n queens on $n \times n$ board).
- Very Bad: n^n (1 queen per row)
- Bad: $n!$ (1 queen per row and column)
- Ok: $n(n-2)(n-4)... = \prod_{i=0}^{\lceil n/2 \rceil -1}(n-2i)$. (Each queen eliminates at least two squares in next row).
- Good: ??? No known formula for the exact number of solutions.

n-Queens

State Size?

- Stupid: 2^{n^2} (n^2 squares, 2 possible states for each).
- Very Very Bad: $\binom{n^2}{n} = \frac{n^2!}{(n^2-n)!n!}$ (Permutations of n queens on $n \times n$ board).
- Very Bad: n^n (1 queen per row)
- Bad: $n!$ (1 queen per row and column)
- Ok: $n(n-2)(n-4)... = \prod_{i=0}^{\lceil n/2 \rceil - 1}(n-2i)$. (Each queen eliminates at least two squares in next row).
- Good: ??? No known formula for the exact number of solutions.
  - (Restricting diagonals is hard)

Backtracking

```
def NQueens(Q, r):

    if r == n:
        return 1

    solutions = 0
    for i in range(n):
        # Check if placement is legal, given previous placements
        legal = True
        for j in range(r):
            if (Q[j] == i) or abs(Q[j]-i) - (r-j) == 0:
                legal = False
                # STOP!!! - Prune!
                break

        # If legal, extend recursively.
        if legal:
            Q[r] = i
            solutions += NQueens(Q,r+1)

    return solutions

Q = [None for _ in range(n)]
return NQueens(Q,0)
```

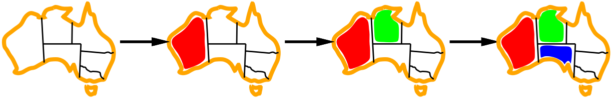
Backtracking Heuristics

- Basic idea:
  - Given partial solution $\mathtt{A} = (a_1, \ldots a_k)$:
  - Compute all successor states a_{k+1}, append to $\mathtt{A}$.
  - Test if result is solution. If not, check whether extensible to a complete solution (via recursion). If not possible, backtrack to deepest node with unexpanded children and recurse.
- Improve performance by considering:
  - Successor to generate next. In CSP: which variable to assign next.)
  - Order for values for each candidate be tried.

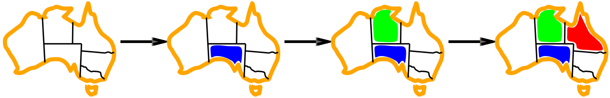
Backtracking Heuristics

Basic idea: Detect inevitable failure as soon as possible while exploring the most promising branches/edges.

- Minimum Remaining Values Heuristic
  - Choose successor with the fewest legal values. Selects successor states that are more likely to result in failure (end as dead leaves of recursion tree).

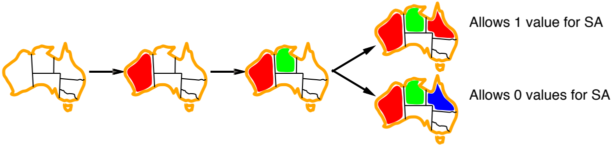


- Degree Heuristic
  - Choose successor involved in the largest number of constraints on other successor states. More constraints, lower branching factor of recursion subtree.



Backtracking Heuristics

- Least Constraining Value
  - Given a variable, assign the value that makes the fewest choices of variables for neighbouring candidates illegal.
  - Permit maximum remaining flexibility for remaining variables. More likely to find a complete solution in future.



- This heuristic seems to encourage more legal assignments, contradicting the MRV heuristic?

Backtracking Heuristics

Q6.2: Explain why it is a good heuristic to choose the variable that is most constrained but the value that is least constraining in a CSP search.

- Search tree for solutions grows exponentially, most branches are invalid combinations of assignments.

Backtracking Heuristics

Q6.2: Explain why it is a good heuristic to choose the variable that is most constrained but the value that is least constraining in a CSP search.

- Search tree for solutions grows exponentially, most branches are invalid combinations of assignments.
- MRV heuristic chooses variable most likely to cause a failure - if search fails early, backtrack and prune the search space. If current partial solution cannot be expanded into a complete solution, want to know earlier instead of wasting time on dead-ends.

Backtracking Heuristics

Q6.2: Explain why it is a good heuristic to choose the variable that is most constrained but the value that is least constraining in a CSP search.

- Search tree for solutions grows exponentially, most branches are invalid combinations of assignments.
- MRV heuristic chooses variable most likely to cause a failure - if search fails early, backtrack and prune the search space. If current partial solution cannot be expanded into a complete solution, want to know earlier instead of wasting time on dead-ends.
- Because we need to find a single solution, we want to be generous and select the value that allows the most future assignments to avoid conflict. This makes it more likely the search will find a complete solution.

Backtracking Heuristics

Q6.2: Explain why it is a good heuristic to choose the variable that is most constrained but the value that is least constraining in a CSP search.

- Search tree for solutions grows exponentially, most branches are invalid combinations of assignments.
- MRV heuristic chooses variable most likely to cause a failure - if search fails early, backtrack and prune the search space. If current partial solution cannot be expanded into a complete solution, want to know earlier instead of wasting time on dead-ends.
- Because we need to find a single solution, we want to be generous and select the value that allows the most future assignments to avoid conflict. This makes it more likely the search will find a complete solution.
- This asymmetry makes it better to use MRV prune exponentially growing search space by choosing least-promising successors first, but increase the probability of success for all successors via LCV.

Constraint Propagation

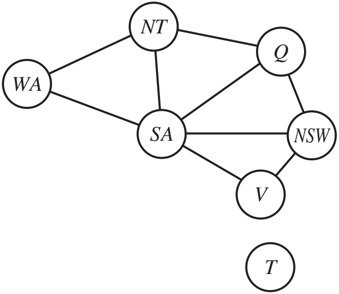
- Want to find consistent variable assignment for all variables in CSP.
- Constraint Propagation: Use constraints to eliminate illegal assignments for a variable locally.
- Enforcing local consistency in each part of the graph eventually causes inconsistent values to be eliminated throughout the graph.
- Basic idea: Reduce size of search tree when backtracking by eliminating values of the domain for each variable that violate binary constraints (arcs).

Arc Consistency

- A CSP variable is arc-consistent if every value in its domain satisfies all relevant binary constraints.
- $X \rightarrow Y$ consistent if for every value x for X $\exists$ some legal assignment y for Y (i.e. surjection).
- Arcs = variable pair (X_i, X_j). The AC-3 algorithm enumerates all possible arcs in a queue Q and makes X_i arc-consistent w.r.t X_j by reducing the domains of variables accordingly.
- If D_i unchanged, move onto next arc. Otherwise append all arcs (X_k, X_i) where X_k is adjacent to X_i.
- If any domain D_i is reduced to 0, terminate: CSP has no consistent solution.
- AC-3 returns an arc-consistent CSP that is faster to search because its variables have smaller domains.

Arc Consistency

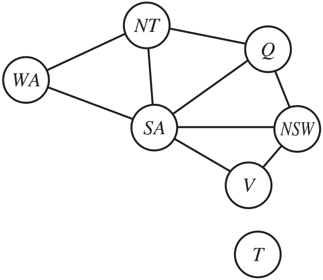
Use the AC-3 algorithm to show that arc consistency can detect the inconsistency of the partial assignment $\{WA = green, V = red\}$ for the problem of colouring the map of Australia as shown in the lectures.



- Recall we wish to assign a color to each state such that no neighbouring states share the same color.
- To detect inconsistency, want to reduce the domain of some variable in graph to zero - i.e. no possible color assignments to any state.

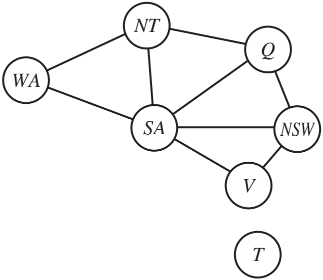
Arc Consistency

- Initial domains:
  - $WA = \{G\}$
  - $NT = \{R, G, B\}$
  - $SA = \{R, G, B\}$
  - $Q = \{R, G, B\}$
  - $NSW = \{R, G, B\}$
  - $V = \{R\}$
- Pop WA-SA arc from queue (degree heuristic).
  - $SA = \{R, B\}$
- Pop SA-V arc from queue.
  - $SA = \{B\}$
- Pop $NT - WA$.
  - $NT = \{R, B\}$
- Pop $NT - SA$.
  - $NT = \{R\}$



Arc Consistency

- Initial domains:
  - $WA = \{G\}$
  - $NT = \{R\}$
  - $SA = \{B\}$
  - $Q = \{R, G, B\}$
  - $NSW = \{R, G, B\}$
  - $V = \{R\}$
- Pop NT-Q:
  - $Q = \{G, B\}$
- Pop SA-Q:
  - $Q = \{G\}$
- No legal assignment for NSW, CSP is inconsistent, terminate.



Arc Consistency

- Time complexity for running AC-3?
  - n variables X_i.
  - E binary constraints, represented as edges.
  - D maximum domain size of each variable.
- Each variable X_i has at most D values to delete, hence each arc (X_i, X_k) can only be inserted in the queue at most D times.
- Checking consistency requires $O(D^2)$ operations (pairwise comparison).
- Hence worst case runtime for generic graph structure is $O(ED^3)$.

Arc Consistency on Trees

- A constraint graph is a tree when any two variables are connected by only one path.
- Obtain linear ordering of variables (topological sort) by:
  - Select one variable to be the root of the tree.
  - Choose ordering of variables such that each variable appears after its parent in the tree.

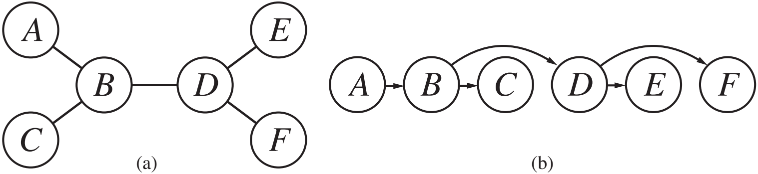


Figure 6.10 (a) The constraint graph of a tree-structured CSP. (b) A linear ordering of the variables consistent with the tree with A as the root. This is known as a **topological sort** of the variables.

Arc Consistency on Trees

- Time complexity for running AC-3 on tree-structured CSP?
  - n variables X_i.
  - E binary constraints, represented as edges.
  - D maximum domain size of each variable.
- For n variables, $E = n - 1 = O(n)$ edges/arcs in AC-3 queue.
- Checking consistency requires $O(D^2)$ operations (pairwise comparison).
- Hence worst case runtime $O(ED^2) = O(nD^2)$.
  - Note E can be up to n^2 for generic graphs!