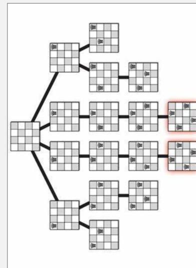
Running Time

How long does backtracking take?

Observation –

- backtracking is DFS on the *state space graph*
  - vertices = partial solutions
  - (directed) edges connect partial solutions one decision apart



https://dev.to/tharan_h5141/mastering-the-n-queen-problem-solving-complex-puzzles-with-backtracking-4814

Running Time

How long does backtracking take?

- DFS is O(n+m)
  - n = number of vertices, m = number of edges

How big is the state space graph?

- branching factor b – number of next choices
- longest path h – largest number of decisions needed to reach a base case

→ worst case $n = O(b^h)$, $m = O(b^{h+1})$

- if there are multiple paths to the same vertex, n can be much smaller – but without storing discovered vertices, repeat visits are handled the same as new visits (and storing discovered vertices takes exponential space)

This...is not good.

Key Points – Making Backtracking Practical

- recursive backtracking is generally not practical without additional effort
  - DFS is O(n+m) where $n = O(b^h)$
    - b = branching factor – number of next options for each choice
    - h = length of longest path – (maximum) number of choices made to get to a complete solution

Key Points – Making Backtracking Practical

- while reducing how much is explored is the dominating factor, it is also important to be efficient in what is done for each subproblem
  - determining whether or not to prune must be efficient
  - modify/restore rather than copying for generating subproblems and partial solutions
  - exploit clever representations

Generating New Partial Solutions and Subproblems

- making a choice typically means an incremental change to the current partial solution and subproblem
- generating the new by copying the old may be expensive
  - copying a collection takes time proportional to the size of the collection

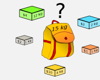
Instead, it may be more efficient to modify the current partial solution and subproblem and then undo.

```
for each legal next choice
  add choice to partial solution and remove
    from subproblem
  result = solve(modified partial solution,
                 modified subproblem)
  remove choice from partial solution and add
    to subproblem
```


Generating New Subproblems

An appropriate framing of alternatives and a clever representation may also be effective.

- e.g. 0-1 knapsack

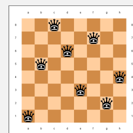


  - find subset of items to take in order to maximize value without exceeding capacity of pack
  - subproblem requires the set *S* of items remaining to consider
    - if the items left to be considered are all at one end of an array, a single index *k* + the original collection of items is sufficient to define S for the subproblem
  - the series of choices and the alternatives
    - take or not take the next item (process input)
      - S is defined by *k*+1 for the subproblems
    - which item to take next (produce output)
      - combine with considering items in some order → S is defined by *k*+1, *k*+2, *k*+3, … for the subproblems

The Effectiveness of Reducing Search

Cleverness in reducing the searching done is essential for there to be any hope of an efficient solution.

The 8 queens problem has 178,462,987,637,760 possible placements.



Tactics.

- only 16,777,216 placements with one queen per column
- only 15,720 placements when conflicts with previously-placed queens are considered
- only half as many placements when symmetry is exploited – only consider n/2 possible rows for the first column placement

Key Points – Making Backtracking Practical

- strategies effective strategies tend to be highly problem-specific
  - *reduce the size of the search space* – formulate the problem to reduce b, h
    - prefer a choice with fewer possible values to reduce *b*
      - e.g. process input “take or not take” instead of produce output “which to take next” for choose-a-subset problems
    - exploit problem characteristics to reduce the number of next possibilities (reduce *b*) without skipping possible solutions
      - e.g. *n* queens – there will be exactly one queen per column, so “place the next queen” really means choosing a row for the queen in the next column instead of choosing any of the remaining spaces
    - impose an ordering on the series of decisions to limit redundant paths
      - e.g. *n* queens – place queens in columns from left to right
      - e.g. 0-1 knapsack – for “choose next item to take”, order the items in some way and only look forward for the next choices
    - if the output size is smaller than the input size, prefer produce output to process input to reduce *h*

Key Points – Making Backtracking Practical

- strategies — effective strategies tend to be highly problem-specific
  - *reduce the amount of the search space explored* – prune branches that don't contain the (only) solution
    - don't pursue dead ends – only consider legal next possibilities (prune invalid partial solutions)
      - e.g. *n* queens – only place a queen if it doesn't attack already-placed queens
      - e.g. 0-1 knapsack – only take an item that fits in the pack
    - don't pursue solutions that are already bad (for optimization problems)
      - prune if the partial solution is worse than the best-so-far complete solution
      - prune if this branch can't lead to a better solution – *branch and bound*
    - prune if it won't eliminate all of the possible solutions
      - e.g. *n* queens – exploit symmetry: only consider rows 0..n/2-1 for the first queen placed
      - e.g. 0-1 knapsack – for process input, if the smallest remaining item doesn't fit in the pack, none of the others can be added either

pruning must be a local decision – cannot consider possible future scenarios of how specific choices would would play out in deciding whether or not to prune

Key Points – Making Backtracking Practical

- strategies — effective strategies tend to be highly problem-specific
  - *try locate good solutions quickly* – order the subproblems to explore more likely ones first
    - order the alternatives – consider the most promising alternative first
      - "most promising" must be a local decision
      - e.g. in maze solving via DFS, consider the adjacent rooms in order of straight-line distance to the goal (closest first)
      - e.g. in shortest path via DFS, consider the adjacent vertices in order of distance from the current vertex (closest first)
    - *best-first search*
      - put discovered subproblems in a priority queue ordered by cost of partial solution so far
    - *best-first search with A* heuristic*
      - put discovered subproblems in a priority queue ordered by estimated cost of the best complete solution derived from the partial solution

Branch and Bound

In optimization problems, pruning can occur if no solutions in a subtree are good enough to be the best.

Idea.

- have a function return the best possible cost of any solution derived from the current partial solution
  - if this cost is not better that the best known solution, the branch doesn't contain an optimal solution and can be pruned

Wrinkle.

- it is generally not possible to determine the best possible cost of a solution derived from a given partial solution without actually searching the subtree and finding all such solutions

Branch and Bound

Revised idea.

- have a function return *an estimate of* the best possible cost of any solution derived from the current partial solution
  - if the estimated cost is not better that the best known solution, the branch doesn't contain an optimal solution and it can be pruned

This *bound function* must be

- safe (optimistic about quality of solutions)
  - if the estimate is too optimistic (it claims better solutions than are possible), the only consequence is wasted time (unnecessary searching of a branch that doesn't actually have a better solution)
  - if the estimate is too pessimistic (it claims solutions are worse than they are), the consequence is possibly missing the optimal solution (thus getting the wrong answer)
- based only on the partial solution / current state
  - cannot consider how future choices play out – we're trying to *avoid* searching the subtree
- relatively efficient to compute
  - evaluated for every next choice

Branch and Bound

We also need to initialize the best known solution so there is something to compare the estimate to before the first solution is found.

- have a function return an estimate of the best possible cost of any solution

This function must be

- safe (pessimistic about quality of solutions)
  - if the estimate is too pessimistic (it claims the best solution is worse than it actually is), the only consequence is wasted time (unnecessary searching of branches with suboptimal solutions because they are thought to be better)
  - if the estimate is too optimistic (it claims the best solution is better than it actually is), the consequence is missing all solutions (thus getting the wrong answer) because no branch is thought to be good enough to be worth exploring
- more efficient than searching to compute

Branch and Bound

The practicality of a branch-and-bound algorithm depends on:

- the quality of the bound function
  - tighter bound means more and earlier pruning – but can't underestimate the true cost of the solutions in that subtree
- the initial solution value
  - closer to optimal means more and earlier pruning – but can't overestimate the cost of the actual best solution
- the time to compute the bound
  - better quality estimates are generally more expensive
  - expensive bound may not save enough work via pruning to be worthwhile – bound function must be computed for each alternative

(as well as the efficiency of the basic backtracking part – generating subproblems, updating partial solutions, pruning)

Branch and Bound

Design challenges:

- finding a good bound function
- finding a good initial solution value
- proving that clever bounds and initial solution values are safe

These tend to be highly problem-specific.