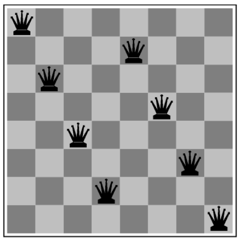
Lecture 22
Backtracking, Branch and Bound Algorithms



EECS 281: Data Structures & Algorithms

Outline

- Review
  - Constraint Satisfaction
  - Optimization
- Backtracking
  - General Form
  - *n* Queens
- Branch and Bound
  - Traveling salesperson problem

Types of Algorithm Problems

- Constraint satisfaction problems
  - Can we satisfy all given constraints?
  - If yes, how do we satisfy them?
    - Need a specific solution
  - May have more than one solution
  - Examples: sorting, puzzles, GRE/analytical
- Optimization problems
  - Must satisfy all constraints (can we?) and
  - Must minimize an objective function subject to those constraints

Types of Algorithm Problems

- Constraint satisfaction problems
  - Go over all possible solutions
  - Does a given input combination satisfy all constraints?
  - *Can stop when a satisfying solution is found*
- Optimization problems
  - Similar, except we also need to compute the objective function every time
  - *Stopping early = possible non-optimal solution*

Types of Algorithm Problems

- Constraint satisfaction problems
  - Can rely on *Backtracking algorithms*
- Optimization problems
  - Can rely on *Branch and Bound algorithms*

For particular problems, there may be much more efficient approaches, but think of these as a fallback to a more sophisticated version of a brute-force approach.

General Form: Backtracking

```
Algorithm checknode(node v)
   if (promising(v))
      if (solution(v))
         write solution*
      else
         for each node u adjacent to v
            checknode(u)
```

* Can exit here if only the existence of a solution is needed

General Form: Backtracking

`solution(v)`

- Check 'depth' of solution (constraint satisfaction)

`promising(v)`

- Different for each application

`checknode(v)`

- Called only if partial solution is both promising and not a solution

An Alternate Form: Backtracking

```
Algorithm checknode(node v)
   if (solution(v))
      write solution*
   else
      for each node u adjacent to v
         if (promising(u))
            checknode(u)
```

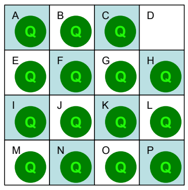
* Can exit here if only the existence of a solution is needed

Backtracking Example: *n* Queens

Can *n* queens be placed on an *n* x *n* board so that no queens are threatened?

- *n* = 1: 1 queen, 1 x 1 board
- *n* = 2: 2 queens, 2 x 2 board
- *n* = 3: 3 queens, 3 x 3 board
- *n* = 4: 4 queens, 4 x 4 board
- *n* = 5: 5 queens, 5 x 5 board
- …

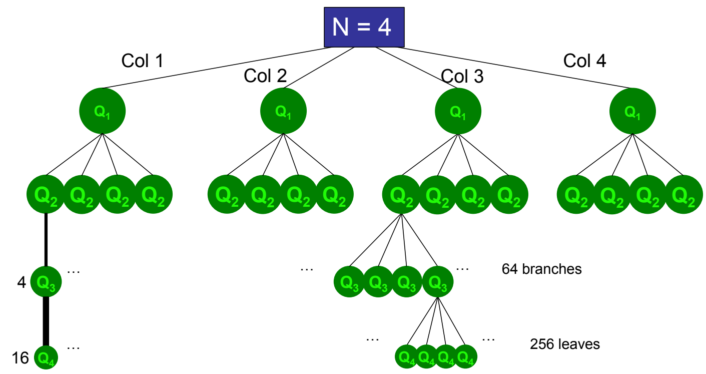
4 Queens Branches



Branches searched

1. **A**->E = vert. threat
2. A->F = diag. threat
3. **A->G**->I = vert. threat
4. A->G->J = diag. threat
5. A->G->K = 2 threats
6. A->G->L = diag. threat
7. **A->H**->I = vert. threat
8. **A->H->J**->M = 2 threats
9. A->H->J->N = 2 threats
10. A->H->J->O = diag. threat
11. A->H->J->P = 2 threats
12. A->H->K = 2 threats
13. A->H->L = vert. threat
14. **B**->E = diag. threat
15. B->F = vert. threat
16. B->G = diag. threat
17. **B->H->I**->M = vert. threat
18. B->H->I->N = 3 threats
19. **B->H->I->O = SOLUTION**

Search Tree: *n* Queens



4 Queens Recap

For 4 Queens

- Entire search tree has **256** leaves
- Backtracking enables searching of **19** branches before finding first solution
- Promising:
  - May lead to solution
- Not promising:
  - Will never lead to solution
  - Therefore should be pruned

Backtracking Elements: *n* Queens

`solution(v)`

- Check 'depth' of solution (constraint satisfaction)
- Placed queen on each row
- That is, depth = N

`checknode(v)`

- Called only if promising and not solution
- Recursive call to all positions (columns) of queen within row

Backtracking Elements: *n* Queens

`promising(row, col)`

- Called for each node of the search tree
- Assume data structures that can tell you if:
  - `column[col] // is column 'col' available`
  - `leftDiagonal[x] // is upper-left to lower-right diagonal available`
  - `rightDiagonal[y] // is upper-right to lower-left diagonal available`
- <u>*NOT*</u> promising if any of these are unavailable
  - **We'll see what 'x' and 'y' are soon…**

8 Queens: Search Space

- Brute force checks about 4.43×10^9 possibilities, including many ridiculous board configurations
- Even with sensible choices (1 queen per row), the search space is still fairly large:
  - **16,772,216** possibilities
  - **92** solutions
- How can the search space be further reduced?

Summary: Backtracking

- Backtracking allows pruning of branches that are not promising
- All backtracking algorithms have a similar form
- Often, most difficult part is determining nature of **`promising()`**

Types of Algorithm Problems

- Constraint satisfaction problems
  - Can we satisfy all given constraints?
  - If yes, how do we satisfy them?
    - Need a specific solution
  - May have more than one solution
  - Examples: sorting, puzzles, GRE/analytical
- Optimization problems
  - Must satisfy all constraints (can we?) and
  - Must minimize an objective function subject to those constraints

Types of Algorithm Problems

- Constraint satisfaction problems
  - Go over all possible solutions
  - Does a given input combination satisfy all constraints?
  - *Can stop when a satisfying solution is found*
- Optimization problems
  - Similar, except we also need to compute the objective function every time
  - *Stopping early = possible non-optimal solution*

Types of Algorithm Problems

- Constraint satisfaction problems
  - Can rely on *Backtracking algorithms*
- Optimization problems
  - Can rely on *Branch and Bound algorithms*

For particular problems, there may be much more efficient approaches, but think of these as a fallback to a more sophisticated version of a brute-force approach.

Branch-and-Bound, a.k.a. B&B

- The idea of backtracking **extended** to *optimization* problems
- You are minimizing a function with this useful property:
  - A partial solution is pruned if its cost ≥ cost of best known complete solution
  - e.g., the length of a path or tour
- If the cost of a partial solution is too big **drop this partial solution**

General Form: Branch & Bound

```
Algorithm checknode(Node v, Best currBest)
   Node u
   if (promising(v, currBest))
      if (solution(v)) then
         update(currBest)
      else
         for each child u of v
            checknode(u, currBest)
   return currBest
```


General Form: Branch & Bound

`solution()`

- Check 'depth' of solution (constraint satisfaction)

`update()`

- If new solution better than current solution, then update (optimization)

`checknode()`

- Called only if promising and not solution

General Form: Branch & Bound

`lowerbound()`

- Estimate of solution based upon
  - Cost so far, plus
  - *Under* estimate of cost remaining (aka bound)

`promising()`

- Different for each application, but must return true when **`lowerbound() < currBest`**
- A return of `false` is what causes pruning (≥)

The Key to B&B is the **Bound**

- The efficiency of B&B is based on "bounding away" (aka "pruning") unpromising partial solutions
- The earlier you know a solution is not promising, the less time you spend on it
- The more accurately you can compute partial costs, the earlier you can prune
- Sometimes it's worth spending extra effort to compute better bounds

Minimizing With B&B

- Start with an "infinity" bound
- Find first complete solution – use its cost as an upper bound to prune the rest of the search
- Measure each partial solution and calculate a lower bound estimate needed to complete the solution
- Prune partial solutions whose lower bounds exceed the current upper bound
- If another complete solution yields a lower cost – that will be the new upper bound
- When search is done, the current upper bound will be a minimal solution

Maximizing With B&B

- Start with a "zero" bound
- Find first complete solution – use its cost as a lower bound to prune the rest of the search
- Measure each partial solution and calculate an upper bound estimate needed to complete the solution
- Prune partial solutions whose upper bounds are less than the current lower bound
- If another complete solution yields a larger value – that will be the new lower bound
- When search is done, the current lower bound will be a maximal solution

Summary Branch and Bound

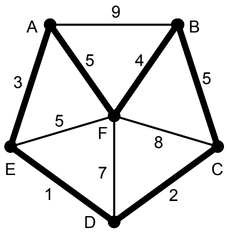
- Method to prune search space for optimization problems
- Need to keep current best solution
- Measure partial solutions and combine with **optimistic** estimates of their completions
- If estimate is not an improvement, actual cannot be either, so prune

TSP Defined

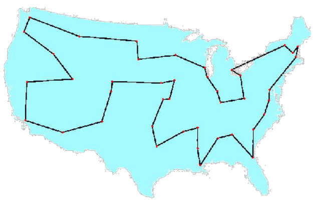
- Hamiltonian Cycle
  - Definition: Given a graph G = (V, E), find a cycle that traverses each node exactly once
  - No vertex may appear twice, except the first/last
  - Constraint satisfaction problem
- Traveling Salesperson Problem
  - Definition: Hamiltonian cycle with least weight
  - Optimization problem

TSP Illustrated

Find tour of minimum length starting and ending in same city and visiting every city exactly once



TSP: (NP) Hard Problem!

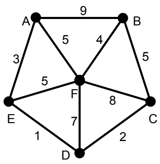


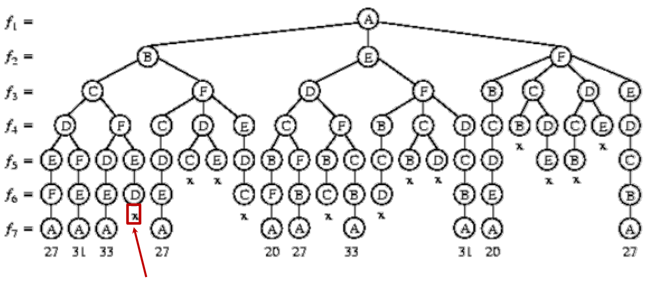
1954: n = 49

2004: n = 24978

http://www.math.uwaterloo.ca/tsp/sweden/index.html

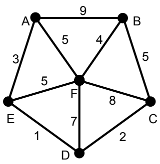
TSP with Backtracking



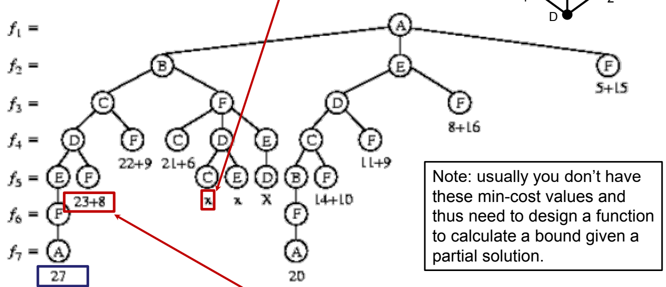


Dead end in the graph = unpromising partial solution
(all adjacent vertices are already visited)

Advantage of TSP with B&B



Dead end



Note: usually you don't have these min-cost values and thus need to design a function to calculate a bound given a partial solution.

Best solution so far

Minimum cost if we complete a cycle from this partial solution… if cost is > 27 this partial solution is not promising

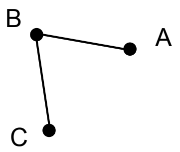
Bounding Function

- Estimate must be ≤ reality
- The bounding function must have complexity better than just continuing TSP for the *k* vertices not yet visited:
  – For instance, ***O*(*k*²)** is better than ***O*(*k*!)** for most values of *k*
- What method can we use to find the lowest cost way to connect *k* vertices together in ***O*(*k*²)** time?

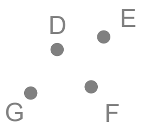
Bounding Function

- Some vertices are connected so far, some vertices are not
- For ONLY the unvisited vertices, connect them together with lowest possible cost
- Then connect the visited vertices to the unvisited
- Yes, this function considers solutions that violate constraints, but it's a LOWER bound so it's OK

Partial TSP Example



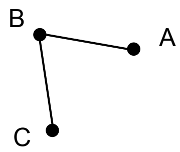
Current path: A - B - C



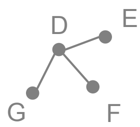
What's the best way to connect D, E, F, and G to each other?

Unvisited vertices: D, E, F, and G

Connect Unvisited Nodes Together



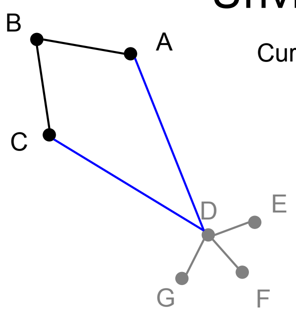
Current path: A - B - C



How many edges are we missing? A full TSP tour would have V edges (7), currently we have 5...

Unvisited vertices: D, E, F, and G

Connect Partial Tour to Unvisited



Current path: A - B - C

Connect from A-B-C to D-E-F-G in the best, cheapest, fastest way possible

Unvisited vertices: D, E, F, and G

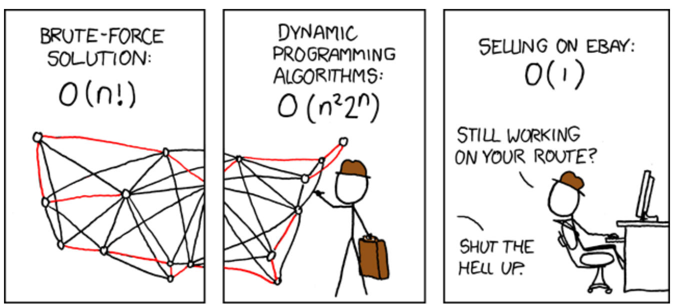
Generating Permutations

```
template <typename T>
void genPerms(vector<T> &path, size_t permLength) {
  if (permLength == path.size()) {
    // Do something with the path
    return;
  } // if
  if (!promising(path, permLength))
    return;
  for (size_t i = permLength; i < path.size(); ++i) {
    swap(path[permLength], path[i]);
    genPerms(path, permLength);
    swap(path[permLength], path[i]);
  } // for i
} // genPerms()
```


Optimal TSP With B&B

- Given *n* vertices, need to find best path out of (*n* – 1)! options, use genPerms()
- Start with upper bound that is "infinity", or better yet a fast calculation of a path that is guaranteed not shorter than optimal
- Use the upper bound to prune the rest of the search, lowering it every time a shorter, complete path is found
- Measure each partial solution, the path length of the first 1 ≤ *k* points and estimate the cheapest cost to connect the remaining *n* – *k* points, this is the lower bound
- Prune a partial solution if its lower bound exceeds the current upper bound
- If another complete path is shorter than the upper bound, save the path and replace the upper bound
- When the search is done, the current upper bound will be a shortest path

Branch and Bound & Traveling Salesperson Problem



http://xkcd.com/399

NQueens Implementation

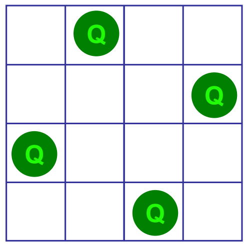
- We know that:
  - Each row will have exactly one queen
  - Each column will have exactly one queen
  - Each diagonal will have at most one queen
- Don't model the chessboard as 2D array!
  - Instead, use 1D arrays of row position, column availability and diagonal availabilities
- To simplify the presentation, we will study for size 4x4

Implementing the Chessboard

First: We need to define an array to store the location of queens placed so far

positionInRow

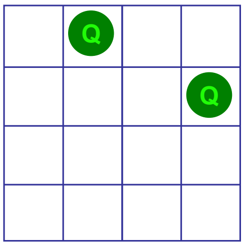
1
3
0
2



Implementing the Chessboard (cont.)

We need an array to keep track of the availability status of the column when we assign queens

Suppose that we have placed two queens

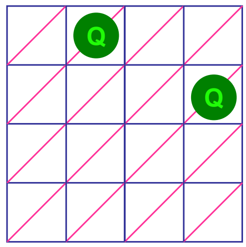


T	F	T	F

Implementing the Chessboard (cont.)

We have 7 left diagonals (2 * *N* - 1); we want to keep track of available diagonals after queens are placed (start indexing at upper left)

T
F
T
T
F
T
T

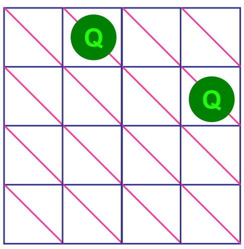


Diagonal Index = row + col

Implementing the Chessboard (cont.)

We also have 7 right diagonals (start indexing at upper right)

T
F
F
T
T
T
T



Diagonal Index = (row – col) + (n – 1)

The promising() Function

```
bool NQueens::promising(uint32_t row, uint32_t col) {
  return    column[col] == AVAILABLE
         && leftDiagonal[row + col] == AVAILABLE
         && rightDiagonal[row - col + (n - 1)] == AVAILABLE;
} // promising()
```


The Recursive putQueen() Function

```
void NQueens::putQueen(uint32_t row) {
  // Check for solution
  if (row == n) {
    cout << "Solution found" << endl;
    return;
  } // if
  // Check every column in this row
  for (uint32_t col = 0; col < n; ++col) {
    if (promising(row, col)) {
      // Make the move, and a recursive call to next move
      positionInRow[row] = col;
      column[col] = !AVAILABLE;
      leftDiagonal[row + col] = !AVAILABLE;
      rightDiagonal[row - col + (n - 1)] = !AVAILABLE;
      putQueen(row + 1);

      // Undo this move and thus backtrack
      column[col] = AVAILABLE;
      leftDiagonal[row + col] = AVAILABLE;
      rightDiagonal[row - col + (n - 1)] = AVAILABLE;
    } // if
  } // for
} // putQueen()
```

Place a piece @(row,col)

Remove piece @(row,col)

NQueens Demo

From a web browser:

`bit.ly/eecs281-nqueens-demo`

From a terminal:

`wget bit.ly/eecs281-nqueens-demo -O nqdemo.tgz`

At the command line:

```
tar xvzf nqdemo.tgz
g++ -std=c++1z -O3 *.cpp -o nqueens
./nqueens
```