Implications of NP-completeness

“I can’t find an efficient algorithm, but neither can all these famous people.”

Overview

Exhaustive search. Iterate through all elements of a search space.

Applicability. Huge range of problems (include intractable ones).

Caveat. Search space is typically exponential in size → effectiveness may be limited to relatively small instances.

Backtracking. Systematic method for examining feasible solutions to a problem, by systematically pruning infeasible ones.
Warmup: enumerate N-bit strings

**Goal.** Process all $2^N$ bit strings of length $N$.
- Maintain array $a[]$ where $a[i]$ represents bit $i$.
- Simple recursive method does the job.

```java
public class BinaryCounter {
  private int N;  // number of bits
  private int[] a;  // a[i] = i-th bit

  public BinaryCounter(int N) {
    this.N = N;
    this.a = new int[N];
    enumerate(0);
  }

  private void process() {
    for (int i = 0; i < N; i++)
      StdOut.println(a[i] + " ");
    StdOut.println();
  }

  private void enumerate(int k) {
    if (k == N)
      process();
    else {
      a[k] = 1;
      enumerate(k+1);
      a[k] = 0;
      enumerate(k+1);
    }
  }

  public static void main(String[] args) {
    int N = Integer.parseInt(args[0]);
    new BinaryCounter(N);
  }
}
```

**Remark.** Equivalent to counting in binary from 0 to $2^N - 1$.

**Warmup: enumerate N-bit strings**

**Traveling salesperson problem**

**Euclidean TSP.** Given $N$ points in the plane, find the shortest tour.

**Proposition.** Euclidean TSP is NP-hard.

**Brute force.** Design an algorithm that checks all tours.
N-rooks problem

Q. How many ways are there to place \( N \) rooks on an \( N \)-by-\( N \) board so that no rook can attack any other?

```
1 2 3 4 5 6 7
2
3
4
5
6
7
```

\( a[4] = 6 \) means the rook from row 4 is in column 6

Representation. No two rooks in the same row or column \( \Rightarrow \) permutation.

Challenge. Enumerate all \( N! \) permutations of \( N \) integers 0 to \( N-1 \).

Enumerating permutations

Recursive algorithm to enumerate all \( N! \) permutations of \( N \) elements.

- Start with permutation \( a[0] \) to \( a[N-1] \).
- For each value of \( i \):
  - swap \( a[i] \) into position 0
  - enumerate all \( (N-1)! \) permutations of \( a[1] \) to \( a[N-1] \)
  - clean up (swap \( a[1] \) back to original position)

```
public class Rooks {
    private int N;
    private int[] a; // bits (0 or 1)

    public Rooks(int N) {
        this.N = N;
        a = new int[N];
        for (int i = 0; i < N; i++)
            a[i] = i;
        enumerate(0);
    }

    private void enumerate(int k) {
        // see previous slide */

        private void enumerate(int k) {
        // place N-k rooks in a[k] to a[N-1]
        private void enumerate(int k) {
            if (k == N)
                return;
        for (int i = k; i < N; i++)
            { exch(k, i);
                enumerate(k+1);
                exch(i, k); // clean up
            }
        }
    }
}
```

Initial permutation

```
0 1 2 3
1 0 2 3
2 1 0 3
3 1 0 2
```

% java Rooks 2
0 1
1 0
% java Rooks 3
0 1 2
1 0 2
2 1 0
**N-queens problem**

**Q.** How many ways are there to place \( N \) queens on an \( N \)-by-\( N \) board so that no queen can attack any other?

```plaintext
int[] a = { 2, 7, 3, 6, 0, 5, 1, 4 };
```

**Representation.** No 2 queens in the same row or column \( \Rightarrow \) permutation.

**Additional constraint.** No diagonal attack is possible.

**Challenge.** Enumerate (or even count) the solutions. \( \leftrightarrow \) unlike N-rooks problem, nobody knows answer for \( N > 30 \)
4-queens search tree (pruned)

- "backtrack" on diagonal conflicts
- solutions

**Backtracking**

**Backtracking paradigm.** Iterate through elements of search space.
- When there are several possible choices, make one choice and recur.
- If the choice is a dead end, backtrack to previous choice, and make next available choice.

**Benefit.** Identifying dead ends allows us to prune the search tree.

**Ex.** [backtracking for N-queens problem]
- Dead end: a diagonal conflict.
- Pruning: backtrack and try next column when diagonal conflict found.

**Applications.** Puzzles, combinatorial optimization, parsing, ...

**N-queens problem: backtracking solution**

```
private boolean canBacktrack(int k)
{
    for (int i = 0; i < k; i++)
    {
        if ((a[i] - a[k]) == (k - i)) return true;
        if ((a[k] - a[i]) == (k - i)) return true;
    }
    return false;
}
```

```
private void enumerate(int k)
{
    if (k == N) {
        process();
        return;
    }
    for (int i = k; i < N; i++)
    {
        exch(k, i);
        if (!canBacktrack(k)) enumerate(k+1);
        exch(i, k);
    }
}
```

```
% java Queens 4
1 3 0 2
2 0 3 1
% java Queens 5
0 2 4 1 3
0 3 1 4 2
1 3 0 2 4
1 4 2 0 3
2 0 3 1 4
2 4 1 3 0
3 1 4 2 0
3 0 2 4 1
4 1 3 0 2
4 2 0 3 1
% java Queens 6
1 3 5 0 2 4
2 5 1 4 0 3
3 0 4 1 5 2
4 2 0 5 3 1
```

**N-queens problem: effectiveness of backtracking**

Pruning the search tree leads to enormous time savings.

<table>
<thead>
<tr>
<th>N</th>
<th>Q(N)</th>
<th>N!</th>
<th>time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>92</td>
<td>40,320</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>352</td>
<td>362,880</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>724</td>
<td>3,628,800</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>2,680</td>
<td>39,916,800</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>14,200</td>
<td>479,001,600</td>
<td>1.1</td>
</tr>
<tr>
<td>13</td>
<td>73,712</td>
<td>6,227,020,800</td>
<td>5.4</td>
</tr>
<tr>
<td>14</td>
<td>365,596</td>
<td>87,178,291,200</td>
<td>29</td>
</tr>
<tr>
<td>15</td>
<td>2,279,184</td>
<td>1,307,674,368,000</td>
<td>210</td>
</tr>
<tr>
<td>16</td>
<td>14,772,512</td>
<td>20,922,789,888,000</td>
<td>1352</td>
</tr>
</tbody>
</table>

**Conjecture.** $Q(N) \sim N! / c^N$, where $c$ is about 2.54.

**Hypothesis.** Running time is about $(N! / 2.5^N) / 43,000$ seconds.
Some backtracking success stories

TSP. Concorde solves real-world TSP instances with ~ 85K points.
- Branch-and-cut.
- Linear programming.
- ...

SAT. Chaff solves real-world instances with ~ 10K variables.
- Davis-putnam backtracking.
- Boolean constraint propagation.
- ...

**Chaff: Engineering an Efficient SAT Solver**

Matthew W. Moskewicz
Department of EECS
UIUC, Urbana-Champaign,
mwmskewicz@illinois.edu

Computer Science

ABSTRACT

Chaff is an enhanced version of the Davis-Putnam algorithm. Chaff combines an aggressive and complete treatment of constraints (X-resolution) and a careful algo-

**Counting: Java implementation**

**Goal.** Enumerate all $N$-digit base-$R$ numbers.

**Solution.** Generalize binary counter in lecture warmup.

```java
// enumerate base-R numbers in a[k] to a[N-1]
private static void enumerate(int k) {
    if (k == N) {
        process(); return;
    }

    for (int r = 0; r < R; r++) {
        a[k] = r;
        enumerate(k+1);
    }

    a[k] = 0; // cleanup not needed; why?
}
```

**Sudoku**

**Goal.** Fill 9-by-9 grid so that every row, column, and box contains each of the digits 1 through 9.

```
7 8 3
5 6 9
1 2 4
```

"Sudoku is a denial of service attack on human intellect."

— Ben Laurie (founding director of Apache Software Foundation)
**Sudoku**

**Goal.** Fill 9-by-9 grid so that every row, column, and box contains each of the digits 1 through 9.

![Sudoku puzzle](http://xkcd.com/74)

**Sudoku is (probably) intractable**

**Remark.** Natural generalization of Sudoku is NP-complete.

---

**Sudoku: brute-force solution**

**Goal.** Fill 9-by-9 grid so that every row, column, and box contains each of the digits 1 through 9.

![Sudoku brute-force solution](http://xkcd.com/74)

**Solution.** Enumerate all 81-digit base-9 numbers (with backtracking).

---

**Sudoku: backtracking solution**

**Iterate through elements of search space.**

- For each empty cell, there are 9 possible choices.
- Make one choice and recur.
- If you find a conflict in row, column, or box, then backtrack.

![Sudoku backtracking solution](http://xkcd.com/74)
Enumerating subsets: natural binary encoding

Given \( N \) elements, enumerate all \( 2^N \) subsets.
- Count in binary from 0 to \( 2^N - 1 \).
- Maintain array \( a[] \) where \( a[i] \) represents element \( i \).
- If \( 1, a[i] \) in subset; if \( 0, a[i] \) not in subset.

### Binary counter from warmup does the job.

```java
private void enumerate(int k) {
    if (k == N)
        process(); return;
    enumerate(k+1);
    a[k] = 1;
    enumerate(k+1);
    a[k] = 0;
}
```
**Quad.** Starting with empty stage, 4 characters enter and exit one at a time, such that each subset of actors appears exactly once.

<table>
<thead>
<tr>
<th>Binary</th>
<th>Subset</th>
<th>Move</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>empty</td>
<td>-</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>0</td>
<td>enter 0</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>1</td>
<td>enter 1</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>1</td>
<td>exit 1</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>2</td>
<td>enter 2</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>2 10</td>
<td>enter 0</td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>2</td>
<td>exit 1</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>2</td>
<td>exit 0</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>3 2</td>
<td>enter 3</td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>3 2 0</td>
<td>enter 0</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>3 0</td>
<td>exit 1</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>3</td>
<td>exit 0</td>
</tr>
</tbody>
</table>

**Binary reflected gray code**

**Def.** The $k$-bit binary reflected Gray code is:
- The $(k-1)$ bit code with a 0 prepended to each word, followed by
- The $(k-1)$ bit code in reverse order, with a 1 prepended to each word.

**Enumerating subsets using Gray code**

**Two simple changes to binary counter from warmup:**
- Flip $a[k]$ instead of setting it to 1.
- Eliminate cleanup.

**Standard binary counter (from warmup):**

```c
// all bit strings in a[k] to a[N-1]
private void enumerate(int k)
{
    if (k == N)
        { process(); return; }
    enumerate(k+1);
    a[k] = 1 - a[k];
    enumerate(k+1);
}
```

**Gray code binary counter:**

```c
// all bit strings in a[k] to a[N-1]
private void enumerate(int k)
{
    if (k == N)
        { process(); return; }
    enumerate(k+1);
    a[k] = 1;
    enumerate(k+1);
    a[k] = 0;
}
```

**Advantage.** Only one element in subset changes at a time.
More applications of Gray codes

3-bit rotary encoder

Chinese ring puzzle (Baguenaudier)
(move ith ring from right when bit i changes in Gray code)

8-bit rotary encoder

Towers of Hanoi
(move ith smallest disk when bit i changes in Gray code)

Scheduling

Scheduling (set partitioning). Given N jobs of varying length, divide among two machines to minimize the makespan (time the last job finishes).

<table>
<thead>
<tr>
<th>job</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.41</td>
</tr>
<tr>
<td>1</td>
<td>1.73</td>
</tr>
<tr>
<td>2</td>
<td>2.00</td>
</tr>
<tr>
<td>3</td>
<td>2.23</td>
</tr>
</tbody>
</table>

machine 0

machine 1

or, equivalently, difference between finish times

Remark. This scheduling problem is NP-complete.

Scheduling: improvements

Brute force. Enumerate $2^N$ subsets; compute makespan; return best.

Many opportunities to improve.

- Fix first job to be on machine 0.
- Maintain difference in finish times. (and avoid recomputing cost from scratch)
- Backtrack when partial schedule cannot beat best known.
- Preprocess all $2^k$ subsets of last $k$ jobs; cache results in memory.

private void enumerate(int k) {
    if (k == N) { process(); return; }
    if (canBacktrack(k)) return;
    enumerate(k+1);
    a[k] = 1 - a[k];
    enumerate(k+1);
}
Enumerating all paths on a grid

**Goal.** Enumerate all simple paths on a grid of adjacent sites.

![Grid with a path]

**Application.** Self-avoiding lattice walk to model polymer chains.

Enumerating all paths on a grid: Boggle

**Boggle.** Find all words that can be formed by tracing a simple path of adjacent cubes (left, right, up, down, diagonal).

![Boggle board]

**Backtracking.** Stop as soon as no word in dictionary contains string of letters on current path as a prefix ⇒ use a trie.

B
BA
BAX

Boggle: Java implementation

```java
private void dfs(String prefix, int i, int j) {
    if ((i < 0 || i >= N) ||
        (j < 0 || j >= N) ||
        (visited[i][j]) ||
        !dictionary.containsAsPrefix(prefix))
        return;

    visited[i][j] = true;
    prefix = prefix + board[i][j];

    if (dictionary.contains(prefix))
        found.add(prefix);

    for (int ii = -1; ii <= 1; ii++)
        for (int jj = -1; jj <= 1; jj++)
            dfs(prefix, i + ii, j + jj);

    visited[i][j] = false;
}
```

Hamilton path

**Goal.** Find a simple path that visits every vertex exactly once

**Remark.** Euler path easy, but Hamilton path is NP-complete.

![Hamilton path graph]
Knight's tour

**Goal.** Find a sequence of moves for a knight so that (starting from any desired square) it visits every square on a chessboard exactly once.

![legal knight moves](image1)

**Solution.** Find a Hamilton path in knight's graph.

---

Hamilton path: backtracking solution

**Backtracking solution.** To find Hamilton path starting at \( v \):
- Add \( v \) to current path.
- For each vertex \( w \) adjacent to \( v \):
  - find a simple path starting at \( w \) using all remaining vertices
- Clean up: remove \( v \) from current path.

**Q.** How to implement?

**A.** Depth-first search + cleanup (!)

---

Hamilton path: Java implementation

```java
public class HamiltonPath {
    private boolean[] marked; // vertices on current path
    private int count = 0; // number of Hamiltonian paths

    public HamiltonPath(Graph G) {
        marked = new boolean[G.V()];
        for (int v = 0; v < G.V(); v++)
            dfs(G, v, 1);
    }

    private void dfs(Graph G, int v, int depth) {
        marked[v] = true;
        if (depth == G.V()) count++;

        for (int w : G.adj(v))
            if (!marked[w]) dfs(G, w, depth+1);
    }
}
```

---

Exhaustive search: summary

<table>
<thead>
<tr>
<th>Problem</th>
<th>Enumeration</th>
<th>Backtracking</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-rooks</td>
<td>permutations</td>
<td>no</td>
</tr>
<tr>
<td>N-queens</td>
<td>permutations</td>
<td>yes</td>
</tr>
<tr>
<td>Sudoku</td>
<td>base-9 numbers</td>
<td>yes</td>
</tr>
<tr>
<td>Scheduling</td>
<td>subsets</td>
<td>yes</td>
</tr>
<tr>
<td>Boggle</td>
<td>paths in a grid</td>
<td>yes</td>
</tr>
<tr>
<td>Hamilton path</td>
<td>paths in a graph</td>
<td>yes</td>
</tr>
</tbody>
</table>
The longest path

The world’s longest path (Sendero de Chile): 9,700 km.
(originally scheduled for completion in 2010; now delayed until 2038)

That’s all, folks: keep searching!

Written by Dan Barrett in 1988 while a student at Johns Hopkins during a difficult algorithms take-home final