COMBINATORIAL SEARCH

- introduction
- permutations
- backtracking
- counting
- subsets
- paths in a graph
COMBINATORIAL SEARCH

- introduction
- permutations
- backtracking
- counting
- subsets
- paths in a graph
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).

![Image of a haystack with a man standing on top saying "Found it!!"]

**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

- Maintain array $a[]$ where $a[i]$ represents bit $i$.
- Simple recursive method does the job.

```java
def enumerate(int k):
    if (k == N):
        process(); return;
    enumerate(k+1);
    a[k] = 1;
    enumerate(k+1);
    a[k] = 0;
```

Remark. Equivalent to counting in binary from 0 to $2^N - 1$. 
public class BinaryCounter
{
    private int N;     // number of bits
    private int[] a;   // a[i] = ith 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.print(a[i]) + " ";
        }
        StdOut.println();
    }

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

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

% java BinaryCounter 4
0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
1 0 0 1
1 0 1 0
1 0 1 1
1 1 0 0
1 1 0 1
1 1 1 0
1 1 1 1

all programs in this lecture are variations on this theme
COMBINATORIAL SEARCH

- introduction
- permutations
- backtracking
- counting
- subsets
- paths in a graph
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?

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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 $a[i]$ back to original position)

<table>
<thead>
<tr>
<th>$N = 3$</th>
<th>0 followed by perms of 1 2 3</th>
<th>1 followed by perms of 0 2 3</th>
<th>2 followed by perms of 1 0 3</th>
<th>3 followed by perms of 1 2 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2</td>
<td>012</td>
<td>102</td>
<td>210</td>
<td>312</td>
</tr>
<tr>
<td>0 1 2</td>
<td>012</td>
<td>103</td>
<td>213</td>
<td>310</td>
</tr>
<tr>
<td>0 1 2</td>
<td>012</td>
<td>120</td>
<td>201</td>
<td>320</td>
</tr>
<tr>
<td>0 1 2</td>
<td>012</td>
<td>123</td>
<td>203</td>
<td>321</td>
</tr>
<tr>
<td>0 1 2</td>
<td>012</td>
<td>130</td>
<td>230</td>
<td>302</td>
</tr>
<tr>
<td>0 1 2</td>
<td>012</td>
<td>132</td>
<td>231</td>
<td>301</td>
</tr>
<tr>
<td>0 1 2</td>
<td>012</td>
<td>130</td>
<td>231</td>
<td>301</td>
</tr>
<tr>
<td>0 1 2</td>
<td>012</td>
<td>132</td>
<td>231</td>
<td>301</td>
</tr>
</tbody>
</table>
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 $a[i]$ back to original position)

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

    for (int i = k; i < N; i++)
    {
        exch(k, i);
        enumerate(k+1);
        exch(i, k); // clean up
    }
}
```
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 exch(int i, int j)
    {  int t = a[i]; a[i] = a[j]; a[j] = t;  }

    public static void main(String[] args)
    {
        int N = Integer.parseInt(args[0]);
        new Rooks(N);
    }
}
4-rooks search tree
COMBINATORIAL SEARCH

- introduction
- permutations
- backtracking
- counting
- subsets
- paths in a graph
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?

![Chessboard with queens placed](image)

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

**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. Unlike \( N \)-rooks problem, nobody knows answer for \( N > 30 \).
4-queens search tree

diagonal conflict on partial solution: no point going deeper

solutions
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

```java
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;
}

// place N-k queens in a[k] to a[N-1]
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);
    }
}
```

The code above uses backtracking to solve the N-queens problem. It checks if placing a queen at a particular position leads to a diagonal violation. If it does, it backtracks by swapping the positions of the queens to find a different solution. This process is repeated until all possible solutions are found.

```
% 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.** \( Q(N) \sim \frac{N!}{c^N} \), where \( c \) is about 2.54.

**Hypothesis.** Running time is about \( \left( \frac{N!}{2.5^N} \right) / 43,000 \) 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 variable.
- Davis-Putnam backtracking.
- Boolean constraint propagation.
- ...

---

### Chaff: Engineering an Efficient SAT Solver

Matthew W. Moskewicz  
Conor F. Madigan  
Ying Zhao, Lintao Zhang, Sharad Malik  
Department of EECS  
Department of EECS  
Department of Electrical Engineering  
UC Berkeley  
muskewcz@alumni.princeton.edu  
cmadigan@mit.edu  
(yingzhao, lintaoz, sharadj)@ee.princeton.edu

**ABSTRACT**

Boolean Satisfiability is probably the most studied of combinatorial optimization/search problems. Significant effort has been devoted to trying to provide practical solutions to this problem for problem instances encountered in a range of applications in Electronic Design Automation (EDA), as well as in Artificial Intelligence (AI). This study has culminated in the many publicly available SAT solvers (e.g. GRASP [8], POSIT [5], SATO [13], relsat [2], WALKSAT [9]) have been developed, most employing some combination of two main strategies: the Davis-Putnam (DP) backtrack search and heuristic local search. Heuristic local search techniques are not guaranteed to be complete (i.e. they are not guaranteed to find a satisfying assignment if one exists or prove unsatisfiability); as a
COMBINATORIAL SEARCH

- introduction
- permutations
- backtracking
- counting
- subsets
- paths in a graph
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.

“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.

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

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

```
3 1 5
6 7 8
2 4 9
8 2 6
1 5 7
4 9 3
7 8 4
5 6 2
9 3 1
```
Sudoku is (probably) intractable

Remark. Natural generalization of Sudoku is NP-complete.

http://xkcd.com/74
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.

**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 grid with backtracking solution highlighted](image)
private void enumerate(int k) {
    if (k == 81) {
        process(); return; }
    if (a[k] != 0) {
        enumerate(k+1); return; }
    for (int r = 1; r <= 9; r++) {
        a[k] = r;
        if (!canBacktrack(k)) {
            enumerate(k+1);
        } else {
            a[k] = 0; break;
        }
    }
}

% more board.txt
7 0 8 0 0 0 3 0 0
0 0 2 0 1 0 0 0
5 0 0 0 0 0 0 0 0
0 4 0 0 0 0 2 6
3 0 0 0 8 0 0 0 0
0 0 0 1 0 0 9 0
0 9 0 6 0 0 0 0 4
0 0 0 7 0 5 0 0
0 0 0 0 0 0 0 0

% java Sudoku < board.txt
7 2 8 9 4 6 3 1 5
9 3 4 2 5 1 6 7 8
5 1 6 7 3 8 2 4 9
1 4 7 5 9 3 8 2 6
3 6 9 4 8 2 1 5 7
8 5 2 1 6 7 4 9 3
2 9 3 6 1 5 7 8 4
4 8 1 3 7 9 5 6 2
6 7 5 8 2 4 9 3 1
COMBINATORIAL SEARCH

- introduction
- permutations
- backtracking
- counting
- subsets
- paths in a graph
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.

<table>
<thead>
<tr>
<th>i</th>
<th>binary</th>
<th>subset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 0</td>
<td>empty</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0 0 1 0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0 0 1 1</td>
<td>1 0</td>
</tr>
<tr>
<td>4</td>
<td>0 1 0 0</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>0 1 0 1</td>
<td>2 0</td>
</tr>
<tr>
<td>6</td>
<td>0 1 1 0</td>
<td>2 1</td>
</tr>
<tr>
<td>7</td>
<td>0 1 1 1</td>
<td>2 1 0</td>
</tr>
<tr>
<td>8</td>
<td>1 0 0 0</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>1 0 0 1</td>
<td>3 0</td>
</tr>
<tr>
<td>10</td>
<td>1 0 1 0</td>
<td>3 1</td>
</tr>
<tr>
<td>11</td>
<td>1 0 1 1</td>
<td>3 1 0</td>
</tr>
<tr>
<td>12</td>
<td>1 1 0 0</td>
<td>3 2</td>
</tr>
<tr>
<td>13</td>
<td>1 1 0 1</td>
<td>3 2 0</td>
</tr>
<tr>
<td>14</td>
<td>1 1 1 0</td>
<td>3 2 1</td>
</tr>
<tr>
<td>15</td>
<td>1 1 1 1</td>
<td>3 2 1 0</td>
</tr>
</tbody>
</table>
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;
}
```
Digit out: Samuel Beckett play

**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 0</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 1</td>
<td>enter 2</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>2 1 0</td>
<td>enter 0</td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>2 0</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 1 1 1</td>
<td>3 2 1 0</td>
<td>enter 1</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>3 2 1</td>
<td>exit 0</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>3 1</td>
<td>exit 2</td>
</tr>
<tr>
<td>1 0 1 1</td>
<td>3 1 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**

**Ruler function**

33
Digression: Samuel Beckett play

**Quad.** Starting with empty stage, 4 characters enter and exit one at a time, such that each subset of actors appears exactly once.

“faceless, emotionless one of the far future, a world where people are born, go through prescribed movements, fear non-being even though their lives are meaningless, and then they disappear or die.” — Sidney Homan
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.

**Gray code binary counter**

```java
// 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);
}
```

**standard binary counter (from warmup)**

```java
// 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;
}
```

```
0 0 0
0 0 1
0 1 1
0 1 0
1 1 0
1 1 1
1 0 1
1 0 0
```

same values since no cleanup

```
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1
```

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

3-bit rotary encoder

8-bit rotary encoder

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

Chinese ring puzzle (Baguenaudier)
(move ith ring from right when bit i changes in Gray code)
Scheduling (set partitioning). Given $N$ jobs of varying length, divide among two machines to minimize the makespan (time the last job finishes).

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.  
  factor of 2 speedup
- Maintain difference in finish times.
  (and avoid recomputing cost from scratch)
  factor of $N$ speedup (using Gray code order)
- Backtrack when partial schedule cannot beat best known.
  huge opportunities for improvement on typical inputs
- Preprocess all $2^k$ subsets of last $k$ jobs;
  reduces time to $2^{N-k}$ at cost of $2^k$ memory
  cache results in memory.

```java
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);
}
```
COMBINATORIAL SEARCH

- introduction
- permutations
- backtracking
- counting
- subsets
- paths in a graph
Enumerating all paths on a grid

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

**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).

---

**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
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.
**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.

**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 (!)
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);
      marked[v] = false;
   }
}
## 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!

Woh-oh-oh-oh, find the longest path!
Woh-oh-oh-oh, find the longest path!

If you said P is NP tonight,
There would still be papers left to write.
I have a weakness;
I'm addicted to completeness,
And I keep searching for the longest path.

The algorithm I would like to see
Is of polynomial degree.
But it's elusive:
Nobody has found conclusive
Evidence that we can find a longest path.

I have been hard working for so long.
I swear it's right, and he marks it wrong.
Some how I'll feel sorry when it's done: GPA 2.1
Is more than I hope for.

Garey, Johnson, Karp and other men (and women)
Tried to make it order N log N.
Am I a mad fool
If I spend my life in grad school,
Forever following the longest path?

Woh-oh-oh-oh, find the longest path!
Woh-oh-oh-oh, find the longest path!
Woh-oh-oh-oh, find the longest path.

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