Combinatorial Search

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

Overview

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


Applicability. Huge range of problems (include NP-hard ones).

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

Caveat to the caveat. Backtracking may prune search space to reasonable size, even for relatively large instances.

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

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

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

Warmup: enumerate N-bit strings

```
public class Counter
{
   private int N;   // number of bits
   private int[] a; // bits (0 or 1)
   public Counter(int N)
   {
      this.N = N;
      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 n)
   {
      if (n == N)
      {  process(); return;  }
      enumerate(n+1);
      a[n] = 1;
      enumerate(n+1);
      a[n] = 0;
   }  
}
```

```
% java Counter 4
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
```

all programs in this lecture are variations on this theme
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?

Representation. No two rooks in the same row or column ⇒ permutation.

Challenge. Enumerate all N! permutations of 0 to N-1.

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

Enumerating permutations

Recursive algorithm to enumerate all N! permutations of size N.

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

```java
private void enumerate(int n)
{
   if (n == N)
   {  process(); return;  }

   for (int i = n; i < N; i++)
   {
      exch(n, i);
      enumerate(n+1);
      exch(i, n);
   }
}
```

```java
% java Rooks 4
0 1 2 3
0 1 3 2
0 2 1 3
0 2 3 1
0 3 2 1
0 3 1 2
1 0 2 3
1 0 3 2
1 2 0 3
1 2 3 0
1 3 2 0
1 3 0 2
2 0 1 3
2 0 3 1
2 1 0 3
2 1 3 0
2 3 0 1
2 3 1 0
3 0 2 1
3 0 1 2
3 1 0 2
3 1 2 0
3 2 0 1
3 2 1 0
```

Recursive algorithm to enumerate all N! permutations of size N.

- Start with permutation 0 to N-1.
- For each value of i
  - swap i into position 0
  - enumerate all (N-1)! arrangements of a[1..N-1]
  - clean up (swap i and 0 back into position)
public class Rooks {
    private int N; // number of bits
    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 n) {
        /* as before */
    }

    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]);
        Rooks rooks = new Rooks(N);
    }
}

N-rooks problem: back-of-envelope running time estimate

Studying slow way to compute N! but good warmup for calculations.

% java Rooks 7 | wc -l
5040
% java Rooks 8 | wc -l
40320
% java Rooks 9 | wc -l
362880
% java Rooks 10 | wc -l
3628800
% java Rooks 25 | wc -l
...  

Hypothesis. Running time is about 2(N! / 8!) seconds.
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?

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Representation. Solution is a permutation: \( a[i] \) = column of queen in row \( i \).

Additional constraint. No diagonal attack is possible

Challenge. Enumerate (or even count) the solutions.

N-queens problem: backtracking solution

Backtracking paradigm. Iterate through elements of search space.
• When there are N 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 row when diagonal conflict found.
N-queens problem: backtracking solution

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

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

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>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>120</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>720</td>
</tr>
<tr>
<td>7</td>
<td>40</td>
<td>5,040</td>
</tr>
<tr>
<td>8</td>
<td>92</td>
<td>40,320</td>
</tr>
<tr>
<td>9</td>
<td>352</td>
<td>362,880</td>
</tr>
<tr>
<td>10</td>
<td>724</td>
<td>3,628,800</td>
</tr>
<tr>
<td>11</td>
<td>2,680</td>
<td>39,916,800</td>
</tr>
<tr>
<td>12</td>
<td>14,200</td>
<td>479,001,600</td>
</tr>
<tr>
<td>13</td>
<td>73,712</td>
<td>6,227,020,800</td>
</tr>
<tr>
<td>14</td>
<td>365,596</td>
<td>87,178,291,200</td>
</tr>
</tbody>
</table>

Hypothesis. Running time is about \(N! / 2.5^N\) / 43,000 seconds.

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

N-queens problem: How many solutions?

<table>
<thead>
<tr>
<th>N</th>
<th>Q(N)</th>
<th>N!</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>73,712</td>
<td>6,227,020,800</td>
</tr>
<tr>
<td>14</td>
<td>365,596</td>
<td>87,178,291,200</td>
</tr>
</tbody>
</table>

Hypothesis. Running time is about \(N! / 2.5^N\) / 43,000 seconds.
Counting: Java implementation

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

Solution. Generalize binary counter in lecture warmup.

```java
private static void enumerate(int n)
{  if (n == N)  { process(); return; }  
   for (int r = 0; r < R; r++)
   {  a[n] = r;  enumerate(n+1);  
   }  a[n] = 0;  
}
```

enumerate $N$-digit base-$R$ numbers

% java Counter 3 2
000
001
010
011
100
101
110
111

% java Count 2 4
00
01
02
03
10
11
12
13
20
21
22
23
30
31
32
33

Remark. Natural generalization is NP-hard.

Counting application: Sudoku

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

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

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

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

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.

backtrack on 3, 4, 5, 7, 8, 9
private static void solve(int cell) {
    if (cell == 81) {
        show(board); return;
    }
    if (board[cell] != 0) {
        solve(cell + 1); return;
    }
    for (int n = 1; n <= 9; n++) {
        if (!backtrack(cell, n)) {
            board[cell] = n;
            solve(cell + 1);
        }
    }
    board[cell] = 0;
}

private static void solve(int cell) {
    if (cell == 81) {
        show(board); return;
    }
    if (board[cell] != 0) {
        solve(cell + 1); return;
    }
    for (int n = 1; n <= 9; n++) {
        if (!backtrack(cell, n)) {
            board[cell] = n;
            solve(cell + 1);
        }
    }
    board[cell] = 0;
}

Enumerating subsets: natural binary encoding

Given N items, enumerate all 2^N subsets.
• Count in binary from 0 to 2^N - 1.
• Bit i represents item i.
• If 0, in subset; if 1, not in subset.

<table>
<thead>
<tr>
<th>i</th>
<th>binary</th>
<th>subset</th>
<th>complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 0</td>
<td>empty</td>
<td>4 3 2 1</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 1</td>
<td>1</td>
<td>4 3 2</td>
</tr>
<tr>
<td>2</td>
<td>0 0 1 0</td>
<td>2</td>
<td>4 3 1</td>
</tr>
<tr>
<td>3</td>
<td>0 0 1 1</td>
<td>3</td>
<td>4 3</td>
</tr>
<tr>
<td>4</td>
<td>0 1 0 0</td>
<td>5</td>
<td>4 2 1</td>
</tr>
<tr>
<td>5</td>
<td>0 1 0 1</td>
<td>6</td>
<td>4 2</td>
</tr>
<tr>
<td>6</td>
<td>0 1 1 0</td>
<td>7</td>
<td>4 1</td>
</tr>
<tr>
<td>7</td>
<td>0 1 1 1</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>1 0 0 0</td>
<td>9</td>
<td>3 2 1</td>
</tr>
<tr>
<td>9</td>
<td>1 0 0 1</td>
<td>10</td>
<td>3 2</td>
</tr>
<tr>
<td>10</td>
<td>1 0 1 0</td>
<td>11</td>
<td>3 1</td>
</tr>
<tr>
<td>11</td>
<td>1 0 1 1</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>1 1 0 0</td>
<td>13</td>
<td>2 1</td>
</tr>
<tr>
<td>13</td>
<td>1 1 0 1</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>1 1 1 0</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>1 1 1 1</td>
<td>empty</td>
<td>4 3 2 1</td>
</tr>
</tbody>
</table>

Binary counter from warmup does the job.

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

Given N items, enumerate all 2^N subsets.
• Count in binary from 0 to 2^N - 1.
• Maintain a[i] where a[i] represents item i.
• If 0, a[i] in subset; if 1, a[i] not in subset.
**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.

<table>
<thead>
<tr>
<th>code</th>
<th>subset</th>
<th>move</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>empty</td>
<td></td>
</tr>
<tr>
<td>0 0 1</td>
<td>1</td>
<td>enter 1</td>
</tr>
<tr>
<td>0 1 1</td>
<td>2</td>
<td>enter 2</td>
</tr>
<tr>
<td>0 0 1</td>
<td></td>
<td>exit 1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>3</td>
<td>enter 3</td>
</tr>
<tr>
<td>0 1 1</td>
<td></td>
<td>exit 2</td>
</tr>
<tr>
<td>0 0 0</td>
<td>4</td>
<td>enter 4</td>
</tr>
<tr>
<td>0 1 0</td>
<td></td>
<td>exit 3</td>
</tr>
<tr>
<td>1 0 0</td>
<td></td>
<td>exit 4</td>
</tr>
<tr>
<td>1 0 1</td>
<td></td>
<td>enter 1</td>
</tr>
<tr>
<td>1 1 1</td>
<td></td>
<td>enter 2</td>
</tr>
<tr>
<td>1 1 0</td>
<td></td>
<td>enter 3</td>
</tr>
<tr>
<td>1 0 1</td>
<td></td>
<td>enter 4</td>
</tr>
<tr>
<td>1 1 1</td>
<td></td>
<td>exit 2</td>
</tr>
<tr>
<td>1 1 0</td>
<td></td>
<td>exit 3</td>
</tr>
<tr>
<td>1 0 1</td>
<td></td>
<td>exit 4</td>
</tr>
<tr>
<td>1 0 0</td>
<td></td>
<td>exit 1</td>
</tr>
</tbody>
</table>

**Binary reflected gray code**

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

**Beckett: Java implementation**

```java
public static void moves(int n, boolean enter)
{
    if (n == 0) return;
    moves(n-1, true);
    if (enter) StdOut.println("enter " + n);
    else StdOut.println("exit  " + n);
    moves(n-1, false);
}
```

```java
% java Beckett 4
enter 1
enter 2
exit  1
enter 3
enter 1
exit  2
exit  1
enter 4
```

**More applications of gray codes**

- 3-bit rotary encoder
- Chinese ring puzzle
- 8-bit rotary encoder
- Towers of Hanoi
- Chinese ring puzzle
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 enumeration

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

standard binary (from warmup)

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

Remark. Only one item changes at a time.

Scheduling (full implementation)

```java
public class Scheduler
{
    private int N; // Number of jobs.
    private int[] a; // Subtask assignments.
    private int[] b; // Best assignment.
    private int[] jobs; // Job lengths.
    private Scheduler(double[] jobs)
    {
        this.N = jobs.length;
        this.jobs = jobs;
        a = new int[N];
        b = new int[N];
        enumerate(0);
    }
    private int[] best()
    {
        return b;
    }
    private void enumerate(int n)
    {
        if (n == N)
        {
            process(); return;
        }
        enumerate(n+1);
        a[n] = 1 - a[n];
        enumerate(n+1);
    }
    private double[] cost(int[] a)
    {  
        double[] time = finish(a);
        return Math.abs(time[0] - time[1]);
    }
    public double[] finish(int[] a)
    {
        double[] time = new double[2];
        for (int i = 0; i < N; i++)
            time[a[i]] += jobs[i];
        return time;
    }

    public static void main(String[] args)
    {  
        /* create Scheduler, print results */
        trace of
        public class Scheduler
        {
            private int N; // Number of jobs.
            private int[] a; // Subtask assignments.
            private int[] b; // Best assignment.
            private int[] jobs; // Job lengths.
            private Scheduler(double[] jobs)
            {
                this.N = jobs.length;
                this.jobs = jobs;
                a = new int[N];
                b = new int[N];
                enumerate(0);
            }
            private int[] best()
            {
                return b;
            }
            private void enumerate(int n)
            {
                if (n == N)
                {
                    process(); return;
                }
                if (backtrack(n)) return;
                enumerate(n+1);
                a[n] = 1 - a[n];
                enumerate(n+1);
            }
            private void process()
            {
                if (cost(a) < cost(b))
                {
                    process(); return;
                }
            }
            public static void main(String[] args)
            {  
                /* create Scheduler, print results */
            }
        }
    }
```

Scheduling (set partitioning). Given \( n \) jobs of varying length, divide among two machines to minimize the 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>

or, equivalently, difference between finish times

```java
public double[] finish(int[] a)
{
    double[] time = new double[2];
    for (int i = 0; i < N; i++)
        time[a[i]] += jobs[i];
    return time;
}
```

Many opportunities (details omitted).

• Fix last job on machine 0 (quick factor-of-two improvement).
• Backtrack when partial schedule cannot beat best known
  (check total against goal: half of total job times)

```java
private void enumerate(int n)
{
    if (n == N-1)
    {  
        process(); return;
    }
    if (backtrack(n)) return;
    enumerate(n+1);
    a[n] = 1 - a[n];
    enumerate(n+1);
}
```

• Process all \( 2^k \) subsets of last \( k \) jobs, keep results in memory,
  (reduces time to \( 2^{N-k} \) when \( 2^k \) memory available).
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 square, it visits every square on a chessboard exactly once.

Solution. Find a Hamilton path in knight’s graph.

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. Add cleanup to DFS (!!)
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;
        // length of current path (recursion depth)
        if (depth == G.V()) count++;
        for (int w : G.adj(v))
            if (!marked[w]) dfs(G, w, depth+1);
        marked[v] = false; // clean up
    }
}
```

Backtracking 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>Hamilton path</td>
<td>paths in a graph</td>
<td>yes</td>
</tr>
</tbody>
</table>

The longest path

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.

Recorded by Dan Barrett in 1988
while a student at Johns Hopkins
during a difficult algorithms final