ADVANCED RECURSION AND BACKTRACKING

Lecture notes of the course "*Programming Techniques*"

Lê Hồng Phương[1]

[1]Department of Mathematics, Mechanics and Informatics
Hanoi University of Science, VNUH
<*phuonglh@gmail.com*>

11/2012

Content

Content

Content

Content

Backtracking

- In many problems, the best way to find a solution is to try all possibilities.
- This is always slow, but there are standard tools that we can use to help.
- Exhaustive search usually uses **divide and conquer** technique represented by recursive functions with **backtracking** technique.

Backtracking

Backtracking is a general technique for finding all or some solutions that

- incrementally builds candidates to the solutions;
- abandons partial candidates if they can not possibly be completed to a valid solution.

Backtracking is an important technique for solving *constraint satisfaction problems* (crosswords, Sudoku and many other puzzles). It is also the basis of logic programming languages like Prolog.

Backtracking

Backtracking is a general technique for finding all or some solutions that

- incrementally builds candidates to the solutions;
- abandons partial candidates if they can not possibly be completed to a valid solution.

Backtracking is an important technique for solving *constraint satisfaction problems* (crosswords, Sudoku and many other puzzles). It is also the basis of logic programming languages like Prolog.

Backtracking

General principles:

- Backtracking with binary strings, k-ary strings
- Knapsack problems
- Generating permutations and combinations

Useful tips: pruning to speed up, how to use backtracking technique

Backtracking

General principles:

- Backtracking with binary strings, k-ary strings
- Knapsack problems
- Generating permutations and combinations

Useful tips: pruning to speed up, how to use backtracking technique

Content

Content

Bit-string generation

A bit-string is an array of n bits. Bit-strings are often used to represent sets.

Problem

Generate (enumerate) all bit-strings of length n.

For example, $n = 3$:

$$000, 001, 010, 011, 100, 101, 110, 111$$

How many bit-strings of length n?

Bit-string generation

```
int a[100];
int n = 3;

void printArray() {
  int i;
  for (i = 0; i < n; i++)
    printf("%d", a[i]);
  printf("\n");
}

int main(int argc, char **argv) {
  binaryStrings(n-1);
  return 0;
}
```

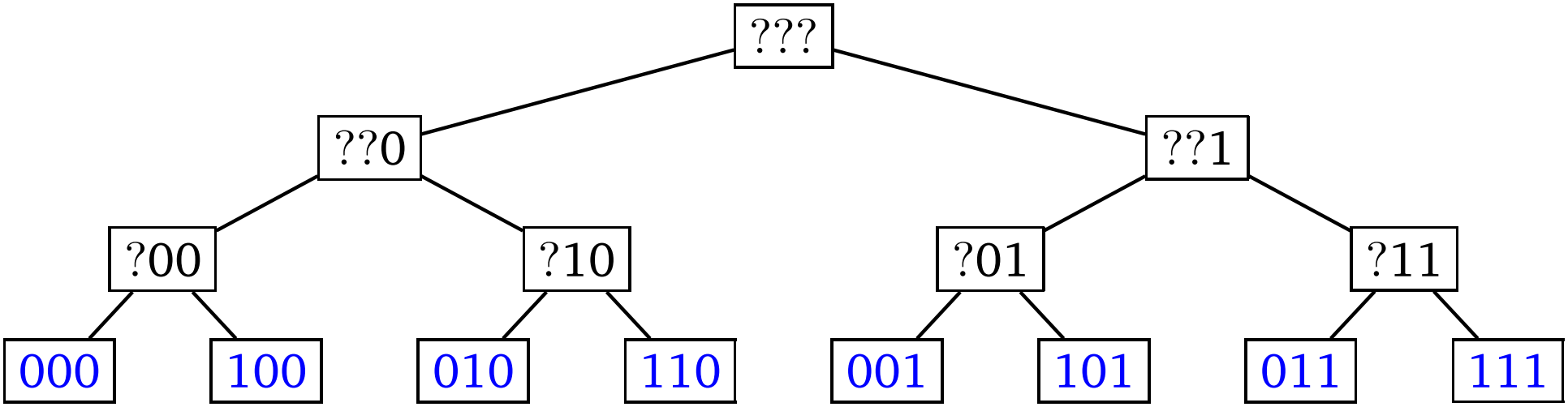
Bit-string generation

The backtracking function:

```
void binaryStrings(int m) {
  if (m < 0) {
    printArray();
  } else {
    a[m] = 0;
    binaryStrings(m-1);
    a[m] = 1;
    binaryStrings(m-1);
  }
}
```

Bit-string generation

Backtracking imposes a tree structure on the solution space.



Backtracking does a preorder traversal on this tree and processes the leaves.

Bit-string generation

What is the time complexity of this algorithm?

- Let $T(n)$ be the running time of `binaryStrings(n)`. We have:

$$T(n) = \begin{cases} c & \text{if } n = 1, \\ 2T(n-1) + d & \text{if } n > 1, \end{cases}$$

  where c and d are constants.

- Solve this recurrence relation using repeated substitutions:

$$T(n) = (c+d)2^{n-1} - d.$$

In terms of complexity notation, $T(n) = O(2^n)$. This means that the algorithm is optimal. Why?

Bit-string generation

What is the time complexity of this algorithm?

- Let $T(n)$ be the running time of `binaryStrings(n)`. We have:

$$T(n) = \begin{cases} c & \text{if } n = 1, \\ 2T(n-1) + d & \text{if } n > 1, \end{cases}$$

  where c and d are constants.

- Solve this recurrence relation using repeated substitutions:

$$T(n) = (c+d)2^{n-1} - d.$$

In terms of complexity notation, $T(n) = O(2^n)$. This means that the algorithm is optimal. Why?

k-ary string generation

- A more general string generation problem: we want to generate all the strings of n numbers drawn from $0..k-1$.
- How many k-ary strings of length n?
- We can use the similar backtracking algorithm:
  - The array `a[]` keep k-ary strings instead of bit strings.
  - Call `karyStrings(n)`

k-ary string generation

- A more general string generation problem: we want to generate all the strings of n numbers drawn from $0..k-1$.
- How many k-ary strings of length n?
- We can use the similar backtracking algorithm:
  - The array `a[]` keep k-ary strings instead of bit strings.
  - Call `karyStrings(n)`

k-ary string generation

```
void karyStrings(int m) {
  if (m < 0) {
    printArray();
  } else {
    int i = 0;
    for (i = 0; i < k; i++) {
      a[m] = i;
      karyStrings(m-1);
    }
  }
}
```

Example: $n = 4, k = 2$.

Content

Enumerating permutations

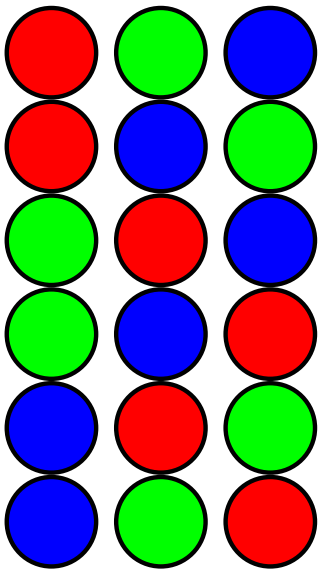
- An arrangement (or ordering) of a set of objects is called a **permutation**.
- In a permutation, the order that we arrange the objects is important.
- Permutations occur in almost every domain of mathematics.
- Here, we consider permutations in combinatorics.

Enumerating permutations

Problem

Generate (enumerate) all permutations of a set of n elements.

- n elements: $n!$ possibilities
- If elements are a, b, c then 6 possible permutations are: $abc, acb, bac, bca, cab, cba$



Enumerating permutations

A simple but inelegant solution for $n = 3$:

```
#define N 3

int main(int argc, char **argv) {
  char a[] = "abc";
  int i, j, k;
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      for (k = 0; k < N; k++)
        if (i != j && j != k && k != i)
          printf("%c%c%c\n", a[i], a[j], a[k]);
  return 0;
}
```

The variable N is hardwired everywhere!

Enumerating permutations

- The program for $n = 3$ above cannot be generalized to deal with general case. Try with $n = 6$.
- Recall that if elements are a, b, c then 6 possible permutations are: $abc, acb, bac, bca, cab, cba$.
- **Key idea:** we can reduce enumerating permutations of n elements to enumerating permutations of $n - 1$ elements and use backtracking technique.

Enumerating permutations

- The program for $n = 3$ above cannot be generalized to deal with general case. Try with $n = 6$.
- Recall that if elements are a, b, c then 6 possible permutations are: $abc, acb, bac, bca, cab, cba$.
- **Key idea:** we can reduce enumerating permutations of n elements to enumerating permutations of $n - 1$ elements and use backtracking technique.

Enumerating permutations

Permutations of *abcde* are one of the following:

1. end with *a* preceded by one of 4! permutations of *bcde*;
2. end with *b* preceded by one of 4! permutations of *acde*;
3. end with *c* preceded by one of 4! permutations of *abde*;
4. end with *d* preceded by one of 4! permutations of *abce*;
5. end with *e* preceded by one of 4! permutations of *abcd*;

So, we can reduce permutations of 5 elements to enumerating permutations of 4 elements.

Enumerating permutations

```
void enumerate(char a[], int n) {
  int i;
  if (i == 0)
    printf("%s\n", a);
  else {
    for (i = 0; i < n; i++) {
      swap(a, i, n - 1);
      enumerate(a, n - 1);
      swap(a, i, n - 1);
    }
  }
}
```

Enumerating permutations

```
void swap(char a[], int i, int j) {
  char c;
  c = a[i];
  a[i] = a[j];
  a[j] = c;
}

int main(int argc, char **argv) {
  char a[] = "abcde";
  enumerate(a, 5);
  return 0;
}
```

Content

Knapsack problem

Problem

Given n objects of weight $s_1, s_2, \ldots, s_n$ and a natural number L, find a subset of the objects that has total weight exactly L.

Example: $n = 3, s_1 = 4, s_2 = 5, s_3 = 2, L = 6$. The answer is $\{s_1, s_3\}$.

- Use a bit-string to represent objects. Set $a[i] = 1$ if object i is to be used.
- To solve the problem, we search exhaustively all possible bit-strings $a[1..n]$ and test for a fit.

Knapsack problem

Problem

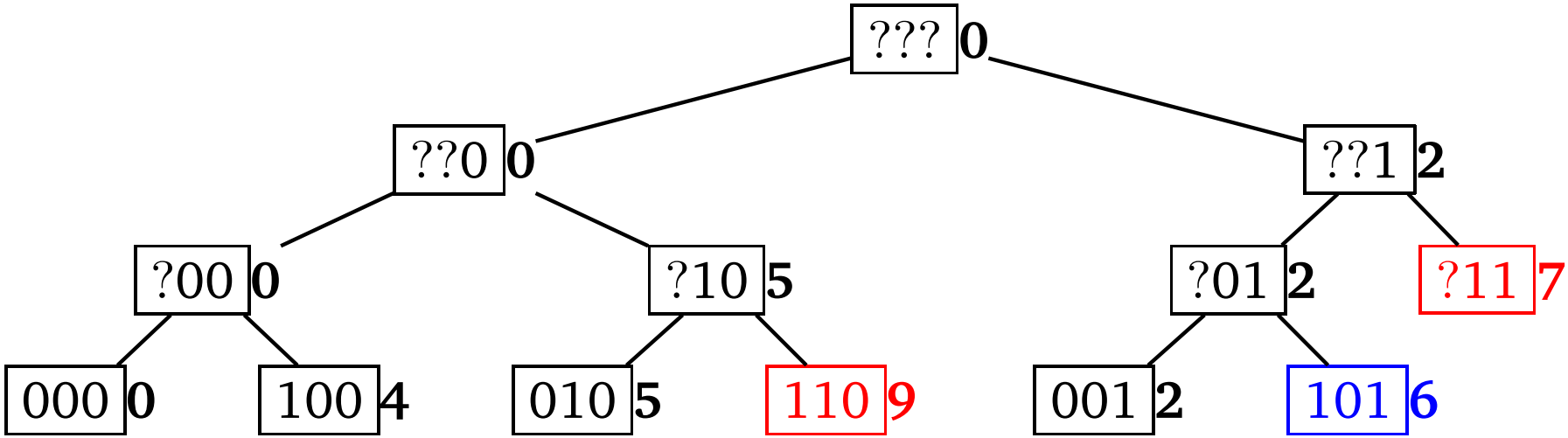
Given n objects of weight $s_1, s_2, \ldots, s_n$ and a natural number L, find a subset of the objects that has total weight exactly L.

Example: $n = 3, s_1 = 4, s_2 = 5, s_3 = 2, L = 6$. The answer is $\{s_1, s_3\}$.

- Use a bit-string to represent objects. Set $a[i] = 1$ if object i is to be used.
- To solve the problem, we search exhaustively all possible bit-strings $a[1..n]$ and test for a fit.

Knapsack problem

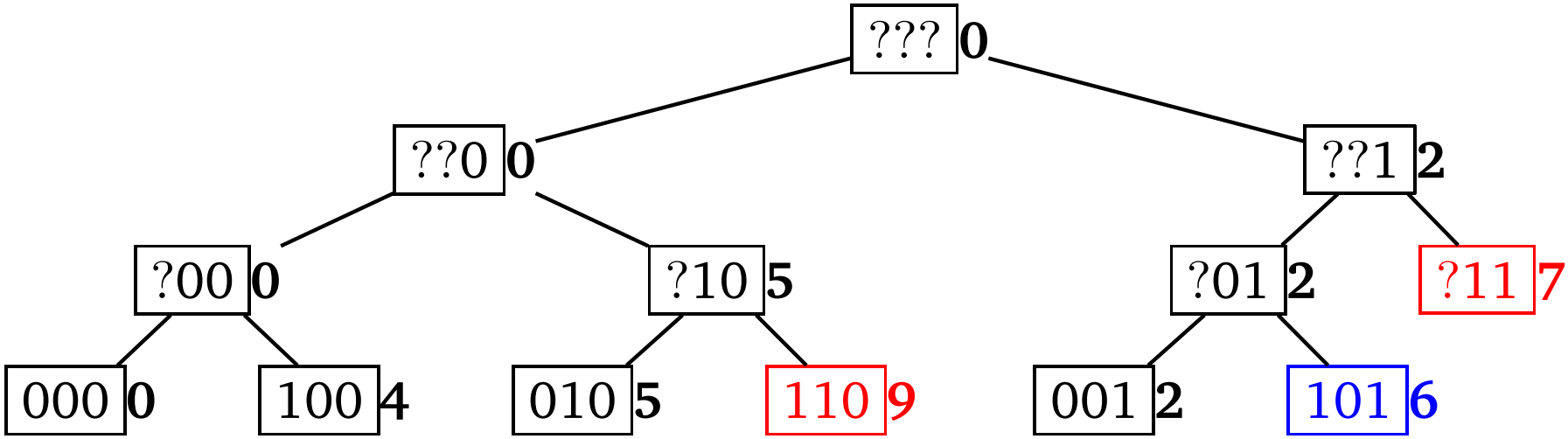
Recall the search space:



- Red branches are pruned to speed up the search.
- Blue strings are accepted: 101 6

Knapsack problem

Recall the search space:



- Red branches are pruned to speed up the search.
- Blue strings are accepted: 101 6

Capacity used

Knapsack problem

- Use the bit-string generation algorithm. Let t be the weight remaining.
- For each object m, there are two cases:
  - Case 1: $a[m] = 0$ is always legal;
  - Case 2: $a[m] = 1$ is illegal if $s_m > t$; prune here
- Call `knapsack`(n, L) to print all solutions to the knapsack problem with n objects and knapsack of weight L.

Knapsack problem

```
void knapsack(int m, int t) {
  if (m == 0) {
    if (t == 0) printArray();
  } else {
     a[m] = 0;
     knapsack(m-1, t);
     if (s[m] <= t) {
       a[m] = 1;
       knapsack(m-1, t - s[m]);
     }
   }
}
```

Note that we index the arrays `a` and `s` from 1 to n.

Content

Exercises

Exercise 1. Draw the tree structure of the solution space of the k-ary string generation algorithm when

1. $n = 2, k = 3$
2. $n = 3, k = 3$

Exercise 2. Given n packs of candy, each pack contains a number of candies. Find a way to split these packs into two parts so that the difference of numbers of candies in the two parts is least.

Exercise 3. Implement the algorithms for enumerating permutations, test your program with different n.

Exercises

Exercise 4. How many different number-plates for cars can be made if each number-plate contains five of the digits 0 to 9 preceded by a letter A to Z (for example: A 004.62), assuming that

- no repetition of digits is allowed?
- repetition of digits is allowed?

Exercise 5. Write a program which can give a random number- plate for a car as specified in Exercise 4.

Exercises

Exercise 6. Implement the knapsack algorithm. Test your program with different data.

Exercise 7. (*Eight queens puzzle*) The eight queens puzzle is the problem of placing 8 chess queens on an 8*x*8 chessboard so that no two queens attach each other. Write a program which gives all solutions to the problem.