

CH-231-A

Algorithms and Data Structures

ADS

Lecture 38

Dr. Kinga Lipskoch

Spring 2020

Backtracking: Motivation¹

Example Sudoku solving

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

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

¹Source of slides: Steven Skiena, Lecture slides, Stony Brook University

Solving Sudoku

- ▶ Solving Sudoku puzzles involves a form of exhaustive search of possible configurations.
- ▶ However, exploiting constraints to rule out certain possibilities for certain positions enables us to prune the search to the point people can solve Sudoku by hand.
- ▶ **Backtracking is a general algorithm** which can be used to implement exhaustive search programs correctly and efficiently.

Backtracking Technique

- ▶ Backtracking is a systematic method to iterate through all the possible configurations of a search space.
- ▶ It is a general algorithm/technique which must be customized for each individual application.
- ▶ In the general case, we will model our solution as a vector $a = (a_1, a_2, \dots, a_n)$, where each element a_i is selected from a finite ordered set S_i .
- ▶ Such a vector might represent an arrangement where a_i contains the i^{th} element of the permutation.
- ▶ Or the vector might represent a given subset S , where a_i is true if and only if the i^{th} element of the universe is in S .

The Idea of Backtracking

- ▶ At each step in the backtracking algorithm, we start from a given partial solution, $a = (a_1, a_2, \dots, a_k)$, and try to extend it by adding another element at the end.
- ▶ After extending it, we must test whether what we have so far is a solution.
- ▶ If not, we must then check whether the partial solution is still potentially extendible to some complete solution.
- ▶ If so, recur and continue. If not, we delete the last element from a and try another possibility for that position, if one exists.

Recursive Backtracking

```
1 Backtrack(a, k)
2   if a is a solution
3     print(a)
4   else {
5     k = k + 1
6     compute S[k]
7     while S[k] != empty do
8       a[k] = an element in S[k]
9       S[k] = S[k] - a[k]
10      Backtrack(a, k)
11   }
```

Backtracking and DFS

- ▶ Backtracking is just depth-first search on an implicit graph of configurations.
- ▶ Backtracking can easily be used to iterate through all subsets or permutations of a set.
- ▶ Backtracking ensures correctness by enumerating all possibilities.
- ▶ For backtracking to be efficient, we must **prune** the search space.

Implementation

```
1 bool finished = FALSE; /* all solutions? */
2 backtrack(int a[], int k, data input) {
3     int c[MAXCANDIDATES]; /* cand. next pos. */
4     int ncandidates; /* next pos. cand. count */
5     int i; /* counter */
6     if (is_a_solution(a, k, input))
7         process_solution(a, k, input);
8     else {
9         k = k+1;
10        construct_candidates(a, k, input, c,
11                             &ncandidates);
12        for (i=0; i<ncandidates; i++) {
13            a[k] = c[i];
14            backtrack(a, k, input);
15            if (finished) return; /* term. early */
16        }
17    }
```



Is a Solution?

- ▶ `is_a_solution(a, k, input)`
- ▶ This boolean function tests whether the first k elements of vector a are a complete solution for the given problem.
- ▶ The last argument, `input`, allows us to pass general information into the routine.

Construct Candidates

- ▶ `construct_candidates(a, k, input, c, &ncandidates);`
- ▶ This routine fills an array `c` with the complete set of possible candidates for the k^{th} position of `a`, given the contents of the first $k - 1$ positions.
- ▶ The number of candidates returned in this array is denoted by `ncandidates`.

Process Solution

- ▶ `process_solution(a, k)`
- ▶ This routine prints, counts, or somehow processes a complete solution once it is constructed.
- ▶ Backtracking ensures correctness by enumerating all possibilities. It ensures efficiency by never visiting a state more than once.
- ▶ Because a new candidates array `c` is allocated with each recursive procedure call, the subsets of not-yet-considered extension candidates at each position will not interfere with each other.

Constructing all Subsets (1)

- ▶ How many subsets are there of an n -element set?
- ▶ To construct all 2^n subsets, set up an array/vector of n elements, where the value of a_i is either true or false, signifying whether the i^{th} item is or is not in the subset.
- ▶ To use the notation of the general backtrack algorithm, $S_k = (\text{true}, \text{false})$, and v is a solution whenever $k \geq n$.
- ▶ What order will this generate the subsets of $\{1, 2, 3\}$?

$(1) \rightarrow (1, 2) \rightarrow (1, 2, 3)^* \rightarrow$
 $(1, 2, -)^* \rightarrow (1, -) \rightarrow (1, -, 3)^* \rightarrow$
 $(1, -, -)^* \rightarrow (1, -) \rightarrow (1) \rightarrow$
 $(-) \rightarrow (-, 2) \rightarrow (-, 2, 3)^* \rightarrow$
 $(-, 2, -)^* \rightarrow (-, -) \rightarrow (-, -, 3)^* \rightarrow$
 $(-, -, -)^* \rightarrow (-, -) \rightarrow (-) \rightarrow ()$

Constructing all Subsets (2)

- ▶ We can construct the 2^n subsets of n items by iterating through all possible 2^n length- n vectors of *true* or *false*, letting the i^{th} element denote whether item i is or is not in the subset.
- ▶ Using the notation of the general backtrack algorithm, $S_k = (\text{true}, \text{false})$, and a is a solution whenever $k \geq n$.

Constructing all Subsets (3)

```
1 is_a_solution(int a[], int k, int n) {
2     return (k == n); /* is k == n? */
3 }
4 construct_candidates(int a[], int k, int n, int
5     c[], int *ncandidates) {
6     c[0] = TRUE;
7     c[1] = FALSE;
8     *ncandidates = 2;
9 }
10 process_solution(int a[], int k) {
11     int i; /* counter */
12     print("(");
13     for (i=1; i<=k; i++)
14         if (a[i] == TRUE)
15             print(i);
16     print(")");}
```

Main Routine: Subsets

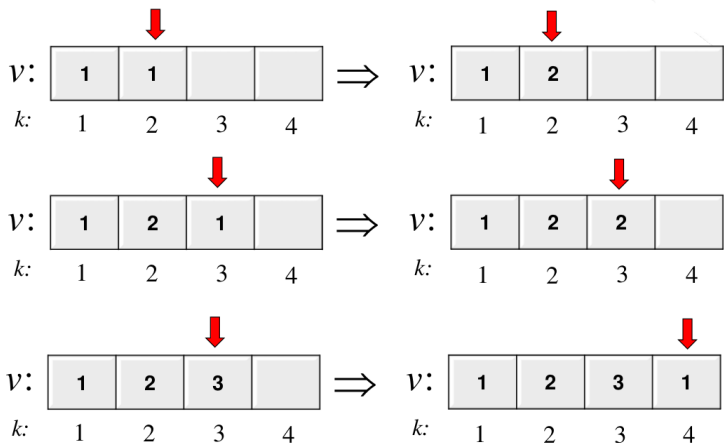
- ▶ Finally, we must instantiate the call to backtrack with the corresponding arguments.

```
1  generate_subsets(int n) {  
2      int a[NMAX]; /* solution vector */  
3      backtrack(a, 0, n);  
4  }
```

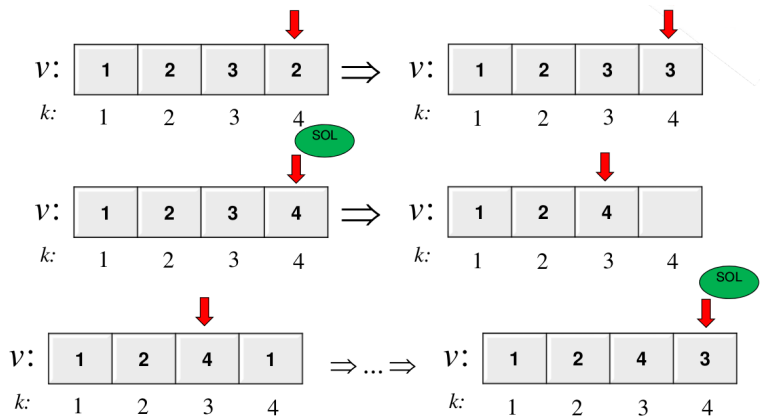
Iterative Backtracking

- ▶ Stack data structure v to store solution(s).
- ▶ Top index of the stack is k .
- ▶ Algorithm iterates, adding/modifying/deleting values on the top of stack
 - ▶ Initialize value on the top of stack - $\text{Init}(k)$
 - ▶ Modify value on the top of stack - $\text{Successor}(k)$
 - ▶ Validate value on the top of stack - $\text{Valid}(k)$
 - ▶ If value on the top of stack valid, we may have a solution - $\text{Solution}(k)$, if yes print - $\text{Print}(k)$
 - ▶ 3 possibilities of stack index:
 - ▶ No change - k
 - ▶ Add new value - $k++$
 - ▶ Go down on stack if value on top not good - $k--$

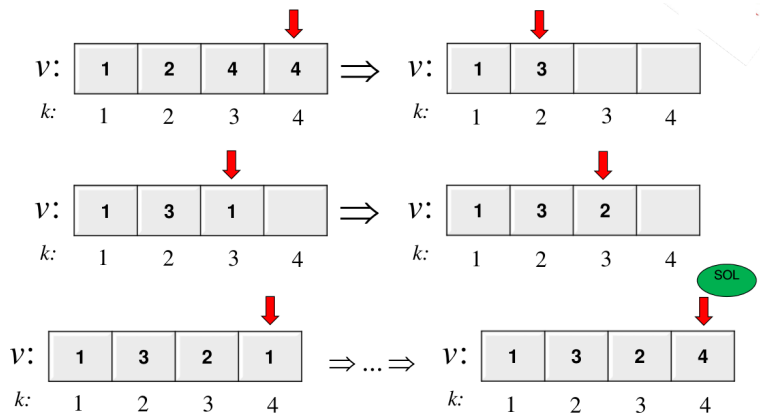
Permutations Example $n = 4$ (1)



Permutations Example $n = 4$ (2)



Permutations Example $n = 4$ (3)



Init and Successor Functions

```
1 void Init(int k) { // k index top of stack
2     v[k]=0; // init top of stack
3 }
4
5 int Succesor(int k) {
6     if (v[k]<n) { // top can increase
7         v[k]++; // increment top
8         return 1;
9     }
10    else
11        // no increase is possible on top
12        return 0;
13 }
```

Valid, Solution and Print Functions

```
1 int Valid(k) {
2     for (i=1;i<k;i++) // check if value on top
3         if (v[i]==v[k]) return 0; // is different
4         // from earlier values in the stack
5     return 1;
6 }
7 int Solution(k) {
8     return (k==n);
9 }
10 void Print() {
11     printf("%d : ",++countSol);
12     for (i=1;i<=n;i++)
13         printf("%d ",v[i]);
14     printf("\n");
15 }
```

Main Iterative Function

```
1 void Back(int n) {
2     k=1; Init(k);
3     while (k>0) { // stack not empty
4         isS=0; isV=0;
5         if (k<=n) // position valid
6             do {
7                 isS=Successor(k);
8                 if (isS) isV=Valid(k);
9             } while (isS && !isV); // s. but not valid
10        if (isS) // successor and valid
11            if (Solution(k))
12                Print();
13            else { // not a solution
14                k++; Init(k); }
15        else // no successor for top
16            k--; }}
```