CH08-320201

Algorithms and Data Structures

ADS

Lecture 26

Dr. Kinga Lipskoch

Spring 2019

Backtracking: Motivation[1]

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

[1] Source 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 $a = (a_1, a_2, ..., a_n)$, 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, $a = (a_1, a_2, ..., a_k)$, 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

```
Backtrack(a, k)
  if a is a solution
    print(a)
  else {
    k = k +1
    compute S[k]
    while S[k] != empty do
      a[k] = an element in S[k]
      S[k] = S[k] - a[k]
      Backtrack(a, k)
  }
```

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

```
bool finished = FALSE; /* all solutions? */
backtrack(int a[], int k, data input) {
  int c[MAXCANDIDATES]; /* cand. next pos. */
  int ncandidates; /* next pos. cand. count */
  int i; /* counter */
  if (is_a_solution(a, k, input))
    process_solution(a, k, input);
  else {
    k = k+1;
    construct_candidates(a, k, input, c,
      &ncandidates);
    for (i=0; i<ncandidates; i++) {
      a[k] = c[i];
      backtrack(a, k, input);
      if (finished) return; /* term. early */
    }}}
```

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`$^{\text{th}}$ 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, $S_k = (true, false)$, 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, $S_k = (true, false)$, and a is a solution whenever $k \geq n$.

Constructing all Subsets (3)

```
is_a_solution(int a[], int k, int n) {
  return (k == n); /* is k == n? */
}
construct_candidates(int a[], int k, int n, int
    c[], int *ncandidates) {
  c[0] = TRUE;
  c[1] = FALSE;
  *ncandidates = 2;
}
process_solution(int a[], int k) {
  int i; /* counter */
  print("(");
  for (i=1; i<=k; i++)
    if (a[i] == TRUE)
      print(i);
  print(")");}
```

Main Routine: Subsets

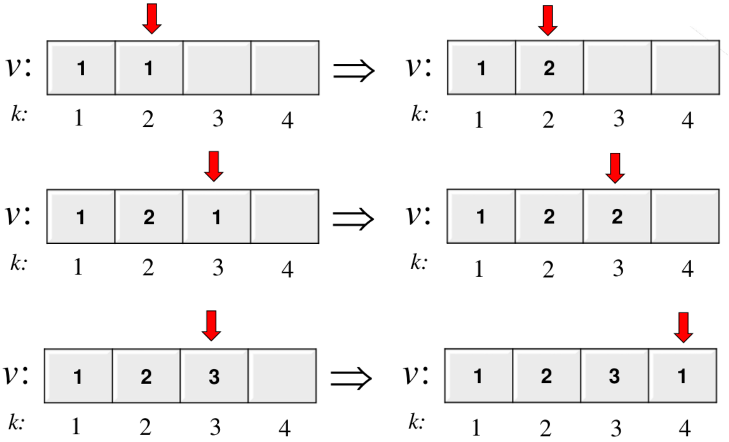
- Finally, we must instantiate the call to backtrack with the corresponding arguments.

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

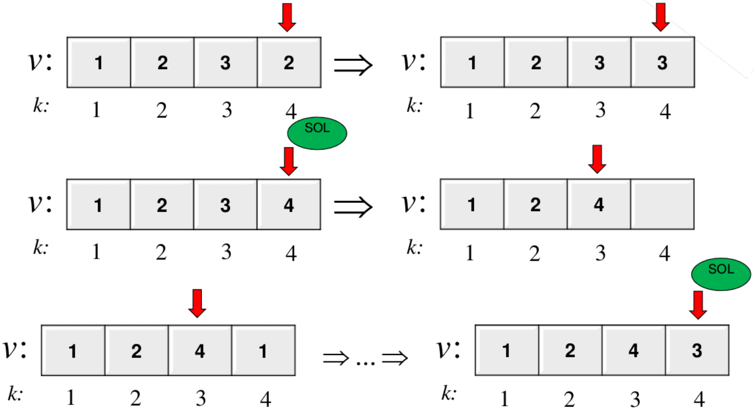
Iterative Backtracking

- 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 - `Init(k)`
  - Modify value on the top of stack – `Successor(k)`
  - Validate value on the top of stack – `Valid(k)`
  - If value on the top of stack valid, we may have a solution – `Solution(k)`, if yes print – `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--`

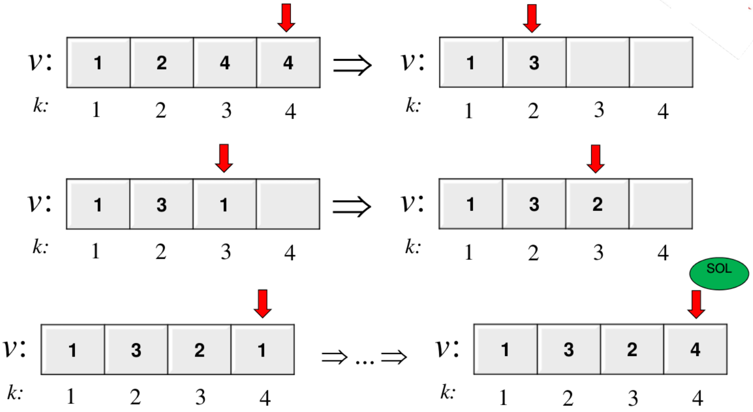
Permutations Example $n = 4$ (1)



Permutations Example $n = 4$ (2)



Permutations Example $n = 4$ (3)



Init and Successor Functions

```
void Init(int k) { // k index top of stack
  v[k]=0; // init top of stack
}

int Succesor(int k) {
  if (v[k]<n) { // top can increase
    v[k]++; // increment top
    return 1;
  }
  else
    // no increase is possible on top
    return 0;
}
```

Valid, Solution and Print Functions

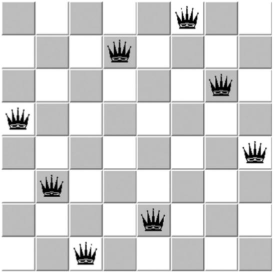
```
int Valid(k) {
  for (i=1;i<k;i++) // check if value on top
    if (v[i]==v[k]) return 0; // is different
                // from earlier values in the stack
  return 1;
}
int Solution(k) {
  return (k==n);
}
void Print() {
  printf("%d : ",++countSol);
  for (i=1;i<=n;i++)
    printf("%d ",v[i]);
  printf("\n");
}
```

Main Iterative Function

```
void Back(int n) {
  k=1; Init(k);
  while (k>0) { // stack not empty
    isS=0; isV=0;
    if (k<=n) // position valid
      do {
        isS=Succesor(k);
        if (isS) isV=Valid(k);
      } while (isS && !isV); // s. but not valid
    if (isS) // successor and valid
      if (Solution(k))
        Print();
      else { // not a solution
        k++; Init(k); }
    else // no successor for top
      k--; }}
```

The Eight-Queens Problem

- The eight queens problem is a classical puzzle of positioning eight queens on an 8×8 chessboard such that no two queens threaten each other.
- This a classical textbook backtracking problem.



Eight Queens: Representation

- What is concise, efficient representation for an *n*-queens solution, and how big must it be?
- Since no two queens can occupy the same column, we know that the *n* columns of a complete solution must form a permutation of *n*.
- By avoiding repetitive elements, we reduce our search space to just $8! = 40,320$ quick for any reasonably fast machine.
- The critical routine is the candidate constructor.
- We repeatedly check whether the k^{th} square on the given row is threatened by any previously positioned queen.
- If so, we move on, but if not we include it as a possible candidate.
- Algorithm can find the $365,596$ solutions for $n = 14$ in minutes.