Backtracking

Busca combinatorial
Subconjuntos
Permutações
Backtracking com Poda
Sudoku

Busca combinatorial

- Busca pela solução ótima
  - Ex: tour de robô otimizado
    - Buscar um ciclo que visita cada ponto em um circuito e que tenha o menor comprimento possível
- Processador atual pode calcular bilhões de instruções por segundo
  - Uma operação comum pode consumir algumas centenas de instruções
  - Então, para problemas em que para cada item é preciso apenas realizar operações simples, pode-se processar até alguns milhões de itens por segundo
  - Pouco ou muito?

Busca combinatorial

- Permutações e combinações
  - 10! = 3.628.800
  - 11! = 39.916.800
  - ...
- Busca por força bruta
  - Listar todas possíveis soluções de um problema de busca combinatorial
  - Técnicas backtracking e pruning (corte) para acelerar a busca através das soluções

Backtracking

- Forma sistemática de iterar por todas as possíveis configurações de um espaço de busca
  - Configurações podem representar
    - Todas possíveis formas de combinar objetos (permutações)
    - Todas possíveis formas de construir uma coleção de combinações (subconjuntos).
    - Enumerar todos caminhos entre dois vértices de um grafo
    - Enumerar todas árvores geradoras de um grafo
    - etc

Backtracking

- Gerar cada possível solução exatamente uma vez
  - Conjunto de soluções $a = (a_1, a_2, \dots, a_n)$
  - Cada elemento a_i é selecionado de um conjunto ordenado S_i
- A cada passo em um algoritmo de backtracking, tentamos estender uma solução parcial $a = (a_1, a_2, \dots, a_k)$ adicionando outro elemento ao final de a
  - Testar se a é uma solução
  - Se a não for solução, verificar se a solução parcial ainda pode ser estendida a uma outra potencial solução

Backtracking

- A busca por backtracking constrói uma árvore de soluções parciais
  - Cada vértice representa uma solução parcial
  - Se existe uma aresta de x para y, então y foi criado a partir de extensão da solução parcial com término em x.
- Processo de construção da árvore
  - Percurso em profundidade na árvore

Algoritmo básico por backtracking

```
Backtrack-DFS(A, k)
    if A = (a_1, a_2, ..., a_k) is a solution, report it.
    else
        k = k + 1
        compute S_k
        while S_k ≠ ∅ do
            a_k = an element in S_k
            S_k = S_k − a_k
            Backtrack-DFS(A, k)
```

Algoritmo básico por backtracking

```
bool finished = FALSE; /* found all solutions yet? */

backtrack(int a[], int k, data input){
    int c[MAXCANDIDATES]; /* candidates for next position */
    int ncandidates; /* next position candidate 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];
            make_move(a,k,input);
            backtrack(a,k,input);
            unmake_move(a,k,input);
            if (finished) return; /* terminate early */
        }
    }
}
```

Algoritmo básico por backtracking

- Mantém algumas informações localmente para cada chamada recursiva
  - Vetor com novos candidatos c
  - Operações de chamadas futuras não interferem os dados locais

Algoritmo básico por backtracking

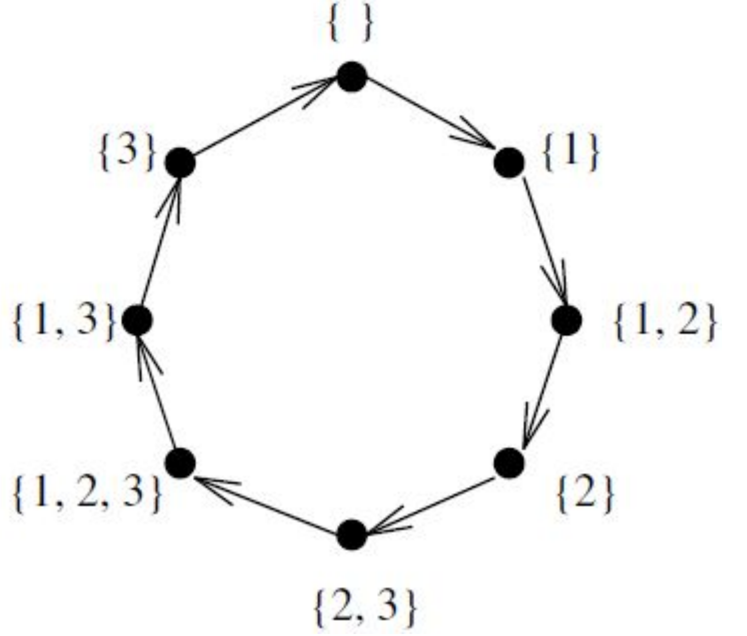
- Funções
  - `is_a_solution(a,k,input)`: teste se os primeiros k elementos do vetor a formam uma solução para um dado problema. Input, por exemplo, pode ser o tamanho (N) da solução do problema.
  - `construct_candidates(a,k,input,c,ncandidates)`: constrói o vetor c com todos possíveis candidatos para a k-éssima posição de a.
  - `process_solution(a,k,input)`: processa (imprime/conta, etc) uma solução
  - `make_move(a,k,input)`: acrescenta um novo elemento à solução
  - `unmake_move(a,k,input)`: desfaz a adição do novo elemento para a busca por novas soluções que não incluam o elemento escolhido no passo atual.

Gerando subconjuntos

- Problema: gerar todos subconjuntos existentes em um conjunto A de n elementos?
  Ex: $A = \{1, .. , n\}$
  $n = 1 \;\; => \;\;$ 2 subconjuntos: $\{\} \; e \; \{1\}$
  $n = 2 \;\; => \;\;$ 4 subconjuntos: $\{\}, \{1\}, \{2\}, \{1,2\}$
  $n = 3 \;\; => \;\;$ 8 subconjuntos:
  $\{\}, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}$
  ... $=> \;\; 2^n$ subconjuntos

Gerando subconjuntos

- Entrada: {1,2,3}
- Saída



Gerando subconjuntos

- Vetor soluções $a = (a_1, a_2, \ldots, a_n)$ de n elementos, onde a_i é verdadeiro ou falso, indicando se o elemento está ou não no subconjunto
- $Sk = (\text{verdadeiro}, \text{falso})$
  - a é solução quando $k = n$

```
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 */
   printf("{");
   for (i=1; i<=k; i++)
      if (a[i] == TRUE) printf(" %d",i);
         printf(" }\n");
}
```

Gerando subconjuntos

- Chamada inicial

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

  – Em qual ordem os subconjuntos são gerados por esta solução?

Permutações

- Quantas permutações possíveis de $\{1, .., n\}$?
  - Primeira posição: n escolhas possíveis distintas
  - Segunda posição: uma vez escolhido um a_1, existem $n-1$ candidatos
  - Portanto, $n! = \prod_{i=1}^{n} i$ permutações distintas
- Ideia
  - Construir um vetor com n posições que indicam quais são os $i-1$ elementos que aparecem na solução parcial
  - Construir o vetor de candidatos para a próximo posição apenas com os elementos que ainda não aparecem na solução parcial

```
construct_candidates(int a[], int k, int n, int c[],
int *ncandidates){
  int i; /* counter */
  bool in_perm[NMAX]; /* who is in the permutation? */

  for (i=1; i<NMAX; i++) in_perm[i] = FALSE;
  for (i=0; i<k; i++) in_perm[ a[i] ] = TRUE;

  *ncandidates = 0;
  for (i=1; i<=n; i++)
    if (in_perm[i] == FALSE) {
      c[ *ncandidates] = i;
      *ncandidates = *ncandidates + 1;
    }
}
```

Permutações

- Nesta solução, a cada chamada da função backtrack, além dessa função alocar um vetor local de tamanho n para armazenar os candidatos de cada etapa e percorrê-lo, a função de construção cria outro vetor de tamanho n, percorre-o duas vezes e ainda percorre o vetor de solução parcial
- É possível fazer com que todas as chamadas recursivas utilizem o mesmo vetor que indica quais elementos são possíveis candidatos para ser adicionados à solução parcial?
  - Exercício: reescreva a função recursiva para que essa utilize menos memória e realize menos operações que a versão apresentada

Poda de busca

- Backtracking garante corretude pois enumera todas as possibilidades
- Poda em árvore de busca
  - Em alguns casos é possível cortar galhos da árvore de busca quando prosseguir a busca a partir de uma solução parcial não leva a uma solução procurada.

Sudoku

- Quebra-cabeça com 3x3 setores de 3x3 posições preenchidos com dígitos de 1 a 9 e posições vazias.
  - Solução: Cada linha, coluna e setor contém os dígitos de 1 a 9 sem repetição ou remoções.

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

							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

Sudoku

- Solução por backtracking
  - Espaço de busca: sequência de posições vazias que devem ser preenchidas com números válidos.
  - Candidatos para os quadrados (i, j): números de 1 a 9 que ainda não apareceram na linha i, coluna j ou no setor 3x3 que contém (i, j)
  - Vetor move com a sequências de pontos (x, y) preenchidos até o passo atual

```
#define DIMENSION 9 /* 9*9 board */
#define NCELLS DIMENSION*DIMENSION /* 81 cells in a 9*9 problem */
typedef struct {
   int x, y;
} point;
typedef struct {
   int m[DIMENSION+1][DIMENSION+1]; /* matrix of board contents */
   int freecount; /* how many open squares remain? */
   point move[NCELLS+1]; /* how did we fill the squares? */
} boardtype;
```

- `construct_candidates()`
  - `next_square()`: escolhe a posição a ser preenchida
  - `possible_values()`: verifica quais são os possíveis valores para a posição

```
construct_candidates(int a[], int k, boardtype *board, int c[],
int *ncandidates)
{
   int x,y; /* position of next move */
   int i; /* counter */
   bool possible[DIMENSION+1]; /* what is possible for the square */
   next_square(&x,&y,board); /* which square should we fill next? */
   board->move[k].x = x; /* store our choice of next position */
   board->move[k].y = y;
   *ncandidates = 0;
   if ((x<0) && (y<0)) return; /*error condition, no moves possible */
   possible_values(x,y,board,possible);
   for (i=0; i<=DIMENSION; i++)
      if (possible[i] == TRUE) {
         c[*ncandidates] = i;
         *ncandidates = *ncandidates + 1;
      }
}
```

- Atualização da tabela
  - O preenchimento da tabela deve permitir que o algoritmo desfaça o movimento
    - `make_move()`
    - `unmake_move()`

```
make_move(int a[], int k, boardtype *board)
{
   fill_square(board->move[k].x, board->move[k].y, a[k], board);
}
unmake_move(int a[], int k, boardtype *board)
{
   free_square(board->move[k].x, board->move[k].y, board);
}
```

- Verificando a solução
  - Verifica contador de posições vazias

```
is_a_solution(int a[], int k, boardtype *board)
{
   if (board->freecount == 0)
      return (TRUE);
   else return(FALSE);
}
```

- Processar a solução
  - Imprime o quadro e termina o processo

```
process_solution(int a[], int k, boardtype *board)
{
   print_board(board);
   finished = TRUE;
}
```

- Escolha de uma posição vazia
  - Função next_square() escolhe uma posição vazia
  - A ordem das posições importa?
  - Escolha arbitrária: a primeira, última ou posição vazia aleatória

- Seleção por restrição: checar cada uma das posições vazias (i, j) e verificar quantos candidatos restam para cada uma. Escolher aquela com o menor número de candidatos.
  - Se alguma posição tiver apenas um candidato, então forçamos a escolha dessa posição. Assim, reduzimos as possibilidades das outras posições sem ter que testar esse candidato para outras posições
  - Redução do espaço de busca!
  - Se a posição mais restrita possui 2 possibilidades: probabilidade de acerto de 1/2 >> 1/9 de probalidade de acerto na posição menos restrita.
  - Escolha de número que não leva a uma solução: em algum momento não haverá candidatos para uma certa posição
    - Fazer backtracking.

Exercícios

1) Considere o problema das 8 rainhas: em um tabuleiro 8x8 de xadrez é possível encontrar diferentes formas de posicionar 8 rainhas de forma que nenhuma rainha consiga atacar outra rainha em apenas 1 movimento. A rainha é uma peça de xadrez que a cada movimento pode se movimentar múltiplas casas e em diferentes direções: vertical, horizontal e em diagonal. Projete um algoritmo por backtracking que encontre uma solução para o problema das 8 rainhas de forma eficiente.

Exercícios

2) Escreva um algoritmo por backtracking que encontre o número mínimo de moedas para retornar *n* centavos de troco para qualquer conjunto *D* de diferentes valores de moedas disponíveis, sendo que *D* sempre inclui a moeda de 1 centavo.

Referências

- Skiena, The Algorithm Design Manual, 2nd ed.
  - Seções 7.1 – 7.3