# 04-630
# Data Structures and Algorithms for Engineers

## Lecture 20: Algorithm Design Strategies II

*Adopted and Adapted from Material by:*

*David Vernon: vernon@cmu.edu ; www.vernon.eu*

## Agenda

- Classes of algorithms
  - Iteration
  - Recursion
  - Brute force
  - Divide and conquer
  - Greedy algorithms
  - Dynamic programming
  - Combinatorial search and backtracking
  - Branch and bound

## Rules of Thumb: algorithm design

1. Handle repetitive tasks through iteration. [resource limited situations]
2. Iterate elegantly though recursion. [elegance & simplicity]
3. Use brute force when you are lazy but powerful. [computationally expensive even for small input sizes]
4. Test bad options then backtrack. [adds intelligence to brute force]
5. Save time with heuristics for a reasonable way out.
6. Divide and conquer your toughest opponents.
7. Identify old issues dynamically not to waste energy again.
8. Bound your problem so the solution doesn't escape.

F. F. Wladston, Computer Science Distilled: Learn the Art of Solving Computational Problems. Code Energy LLC (2017)

## Agenda

- Classes of algorithms
  - Brute force
  - Divide and conquer
  - Greedy algorithms
  - Dynamic programming
  - Combinatorial search and backtracking
  - Branch and bound

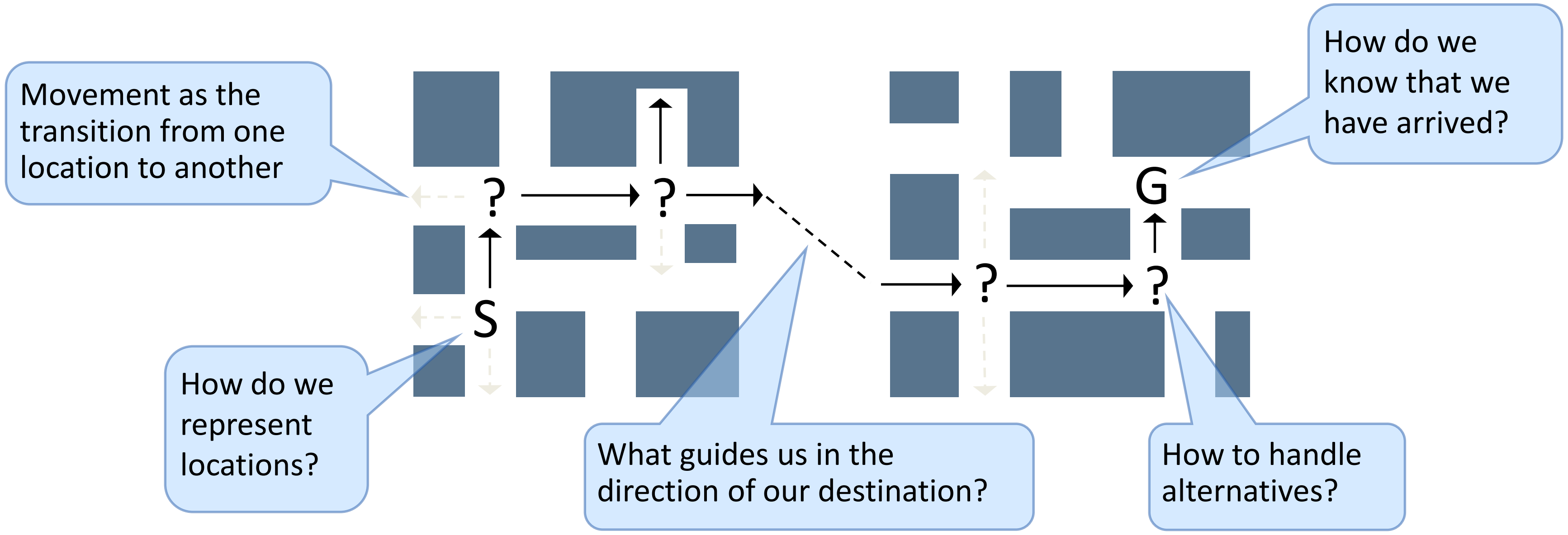
## Combinatorial Search / State Space Search

We can find optimal solutions to many problems using exhaustive search technique

- However, the complexity can be huge, so we need to be careful
- If the complexity is $O(2^n)$ it will be feasible to consider problems where $n < 40$
- If the complexity is $O(n!)$ it will be feasible to consider problems where $n < 20$

# Combinatorial Search / State Space Search

- Solving problems through the **systematic search** for solutions in a (large) **state space**
- The general idea is to incrementally **extend partial solutions until a complete solution is obtained**


Movement as the transition from one location to another
How do we know that we have arrived?
G
?
?
?
?
S
How do we represent locations?
What guides us in the direction of our destination?
How to handle alternatives?

## Combinatorial Search / State Space Search

- Search is the systematic process of
  - choosing one of many possible alternatives,
  - saving the rest in case the alternative selected first does not lead to the goal
- Search can be viewed as the construction and traversal of **search trees**

# Combinatorial Search / State Space Search

Characterization of the state space

- The **initial state** (e.g., a location)
- A **set of operators** which take us from one state to another state (e.g., drive straight, turn left, … )
- A **goal-test** which decides when the goal is reached (e.g., comparing locations)
  - Explicit states (e.g., a specific address)
  - Abstractly described states (e.g., any sector office)

## Combinatorial Search / State Space Search

Characterization of the state space

- A **description of a solution** (e.g., the address, the path between locations or the moves used)
  - The search path (e.g., the shortest path between your home and your office)
  - Just the final state (e.g., the sector office)

## Combinatorial Search / State Space Search

Characterization of the state space

- A **cost function** (e.g., time, money, distance or number of moves):
  *true cost* for going from start to where we are now +
  *estimated cost* for going from where we are now to the nearest goal

  - Search cost, the cost for concluding that a certain operator should be used (e.g., the time it takes to ask someone for directions or thinking about a move) +

  - Path cost, the cost for using an operator (e.g., the energy it takes to walk or time)

# Combinatorial Search / State Space Search

Reminder of the potential size of state spaces

Propositional satisfiability problem (SAT):

- Decide if there is an assignment to the variables of a propositional formula that satisfies it:

$$f = (\bar{x}_2 + \bar{x}_4)(x_3 + x_4)(\bar{x}_3 + x_4)(x_1 + x_2)(\bar{x}_1 + \bar{x}_3)$$

- 100 variables → $2^{100} \sim 10^{30}$ combinations
  1000 evaluations/second →
  **31,709,791,983,764,586,504 years** required to evaluate all combinations

## Combinatorial Search / State Space Search

Reminder of the potential size of state spaces

Traveling salesman problem (TSP)

- Given a number of cities along with the cost of travel between each pair of them, find the cheapest way of visiting all the cities exactly once and returning to the starting point
- There are 2 identical tours for each permutation of $n$ cities → the number of tours are $n!/(2n) = (n-1)!/2$ ...
  - divide by n if we don't care where we start
  - divide by 2 if we don't care which direction we take the tour
- A 50-city TSP therefore has about **$3*10^{62}$ potential solutions**

## Agenda

- Classes of algorithms
  - Brute force
  - Divide and conquer
  - Greedy algorithms
  - Dynamic programming
  - Combinatorial search and backtracking
  - Branch and bound

## Backtracking

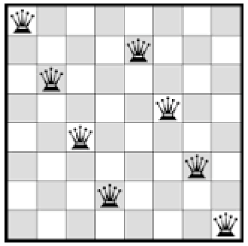
- A systematic method to iterate through all the possible configurations of a search space
  - All possible arrangements of object: permutations
  - All possible ways of building a collection of objects: subsets
  - Generation of all possible spanning trees of a graph
  - Generation of all possible paths between two vertices
  - …
- Exhaustive search … check each solution generated to see if it is the required solution (satisfies some optimality criterion)
- General technique
  - Must be customized for each individual application

## Backtracking

- Based on the construction of a state space tree
  - nodes represent states,
  - root represents the initial state
  - one or more leaves are goal states
  - each edge represents the application of an operator
- The solution is found by expanding the tree until a goal state is found

## Backtracking

- Examples of problems that can be solved using backtracking:
  - Puzzles (e.g., eight queens puzzle, crosswords, Sudoku)
  - Combinatorial optimization problems (e.g., parsing and layout problems)
  - Logic programming languages such as Icon, Planner and Prolog, which use backtracking internally to generate answers



## Backtracking

- Generate each possible configuration exactly once
- Avoiding repetitions and not missing configurations means we must define a systematic generation order
- Let the solution be a vector

  $\boldsymbol{a} = (a_1, a_2, \ldots a_n)$

  where each element is selected from a finite ordered set $S_i$

  – For example, $\boldsymbol{a}$ might represent a permutation and $a_i$ might be the $i^{th}$ element of the permutation

  – For example, $\boldsymbol{a}$ might be a subset $S$, and $a_i$ would be true if and only if the $i^{th}$ element of the universal set is in $S$

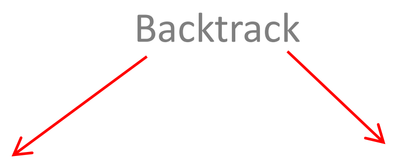
  – For example, $\boldsymbol{a}$ might be a sequence of moves in a game or a path in a graph, where $a_i$ contains the $i^{th}$ event in the sequence

# Backtracking

- At each step, start from a partial solution

  $\boldsymbol{a} = (a_1, a_2, \ldots a_i)$

- Try to extend it by adding another element at the end
- After extending, test whether what we have so far is a solution
- If it is, use it (e.g., check to see if it’s the best solution so far)
- If it isn’t, check to see whether it can be extended to form a complete solution
- If it can, continue with recursion
- If it can’t, delete the last element from $a$ and try another possibility from that position if it exists



# Backtracking

- Backtracking constructs a tree of partial solutions
  - Each vertex represents one partial solution
  - There is an edge from one node $x$ to node $y$ if node $y$ was created by advancing from $x$
  - Constructing the solutions can be viewed as doing a depth-first traversal of the backtrack tree
- Backtracking ensures correctness by enumerating all possibilities
- Backtracking ensures efficiency by never visiting a state more than once

# Backtracking

Backtracking as a depth-first traversal

```
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 $S_k \neq \emptyset$ do
            $a_k$ = an element in $S_k$
            $S_k = S_k - a_k$
            Backtrack-DFS($A, k$)
```

# 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; // k==1 => we need to choose a1, ...
      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 */
      }
   }
}
```

This backtracking code is based on the examples in S. Skiena, The Algorithm Design Manual, 2008.

## Backtracking

Note how recursion yields an elegant and easy implementation of the backtracking algorithm

- The new candidates array `c` is allocated with each recursive procedure call
- Consequently, the not-yet-considered extension candidates at each position don't interfere with each other

## Backtracking

The application-specific parts are dealt with in functions

```
1.  is_a_solution(a,k,input)
2.  construct_candidates(a,k,input,c,&ncandidates)
3.  process_solution(a,k,input)
4.  make_move(a,k,input)
5.  unmake_move(a,k,input)
```

## Backtracking

is_a_solution(a,k,input)

- Boolean function
- Tests whether the first *k* elements of vector a form a complete solution for the given problem
- The argument input allows us to pass general information to the routine
- We could use it to specify *n*, the size of a target solution, e.g. when constructing permutations or subsets of *n* elements

# Backtracking

construct_candidates(a,k,input,c,&ncandidates)

– 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 given by ncandidates

– Again, input may be used to pass auxiliary information

## Backtracking

process_solution(a,k,input)

– Prints, counts, or otherwise processes a complete solution once it is constructed

## Backtracking

make_move(a,k,input)

unmake_move(a,k,input)

- These functions enable us to modify a data structure in response to the latest move
- or clean up this data structure if we decide to take back the move
- You could build such a data structure from scratch from the solution a if required but it can be more efficient to do it this way if the changes involved in a move can be easily undone

## Backtracking

- Many combinatorial optimization problems require the enumeration of all subsets / permutations of some set (and testing each enumeration for optimality / success)

- Being able to compute the number of subset / permutations is far easier than enumerating them

  – There are $n!$ permutations of $n$ elements

  – There are $2^n$ subsets of $\mathrm{n}$ elements

- Recall earlier comments on the exponential size of a state space

## Backtracking

- Pruning
  - Backtracking ensures correctness by enumerating all possibilities
  - Enumerating all $n!$ permutations of $n$ vertices of a graph and selecting the best one certainly yields the correct algorithm to find the optimal travelling salesman tour
    - For each permutation, check to see if the tour exists in the graph (do the edges exist?)
    - If so, add all the weights and see if it is the best solution

## Backtracking

- Pruning
  - But it is very wasteful to construct all the permutations first and then analyze them later
    - For example, if the search starts at vertex $v_1$ and if $(v_1, v_2)$ is not in the graph
    - The next $(n\text{-}2)!$ permutations enumerated starting with $\mathbf{v_2}$ would be a complete waste of effort
    - Much better to prune the search after $(v_1, v_2)$ and continue next with $(v_1, v_3)$
    - By restricting the set of next elements to reflect only moves that are legal / valid from the current partial configuration, we significantly reduce the search complexity

## Backtracking

- Pruning
  - Is the technique of cutting off the search the instant we have established that a partial solution cannot be extended into a full solution
  - Combinatorial searches, when augmented with tree pruning techniques, can be used to find the optimal solution of small optimization problems
    - The actual size depends on the problem
    - Typical size limit are somewhere from 15 to 50 items

## Agenda

- Classes of algorithms
  - Brute force
  - Divide and conquer
  - Greedy algorithms
  - Dynamic programming
  - Combinatorial search and backtracking
  - Branch and bound

## Branch-and-Bound

- In backtracking, we used depth-first search with pruning to traverse the state space

- But we can achieve better performance for many problems using breadth-first search with pruning

- This approach is known as branch-and-bound

  – The implicit stack in depth first search is  replaced by an explicit queue in breadth first search

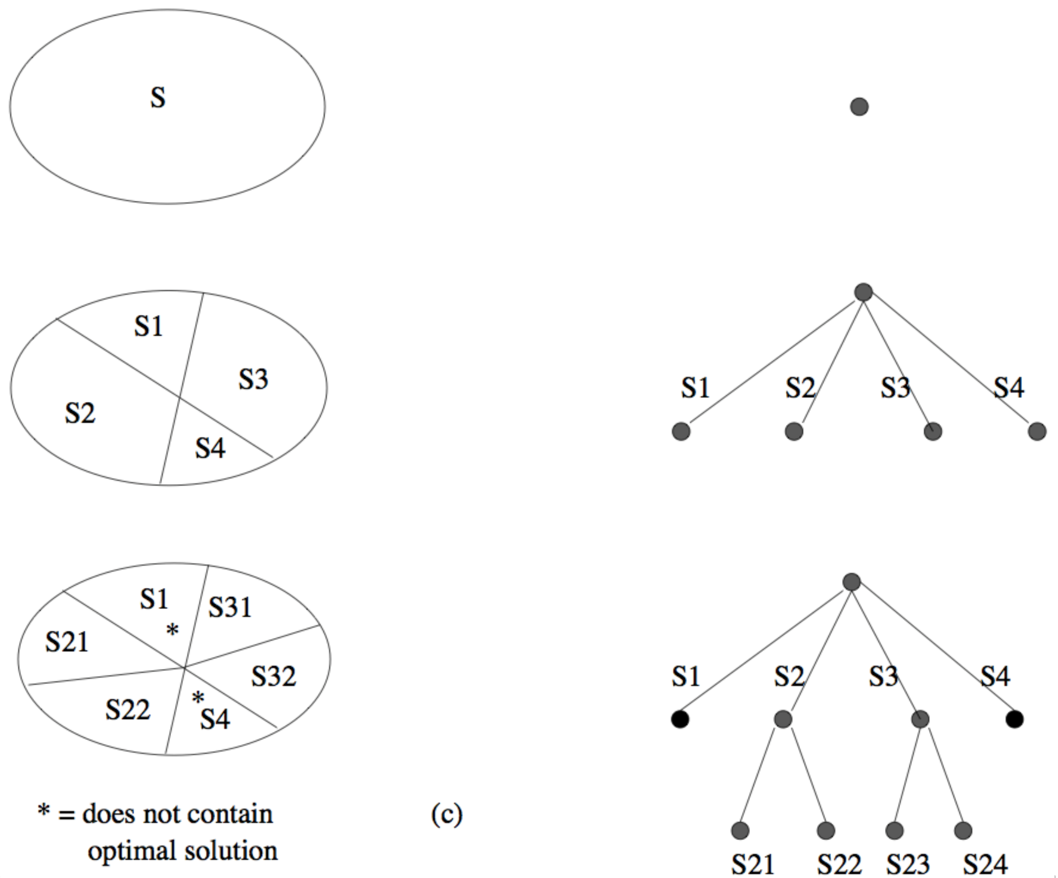
  – If we use a priority queue, we have a **best-first** traversal of the state space

## Branch-and-Bound

- Advantage of using breadth-first (or best-first) search:
  - When a node (i.e. a partial solution) that is judged to be promising (i.e. a possible candidate for a full solution) when it is first encountered and placed in the queue, it may no longer be promising when it is removed
  - If it is no longer promising, it is discarded and the evaluation and testing of its children (i.e. remainder of the solution) is avoided
- Branch-and-bound is by far the most widely used tool for solving large scale NP-hard combinatorial optimization problems
- However, it is an algorithm paradigm that has be be customized for each specific problem type

# Branch-and-Bound

- Use bounds for the function to be optimized & the value of the current best solution to limit the search space



## Summary

- *Several strategies* exist.

- *Choice* of strategy should be guided by:
  - *available resources* (CPU, memory etc.),
  - *problem size*, and
  - *time complexity*.

- Ultimate *goal* is to develop *optimal solutions*.