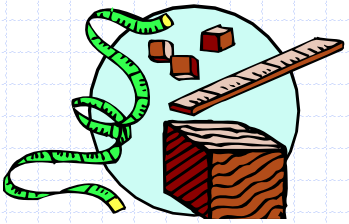
Presentation for use with the textbook, Algorithm Design and Applications, by M. T. Goodrich and R. Tamassia, Wiley, 2015

Approximation Algorithms



1

1

Bike Tour

- Suppose you decide to ride a bicycle around Ireland
  - you will start in Dublin
  - the goal is to visit Cork, Galway, Limerick, and Belfast before returning to Dublin
- What is the best itinerary?
  - how can you minimize the number of kilometers yet make sure you visit all the cities?

	Belfast	Cork	Dublin	Galway	Limerick
Belfast	–				
Cork	425	–			
Dublin	167	257	–		
Galway	306	209	219	–	
Limerick	323	105	198	105	–

2

Optimal Tour

- If there are only 5 cities it's not too hard to figure out the optimal tour
  - the shortest path is most likely a "loop"
  - any path that crosses over itself will be longer than a path that travels in a big circle

3

Exhaustive Search: Too Many Tours

- There is a problem with the exhaustive search strategy
  - the number of possible tours of a map with n cities is $(n - 1)! / 2$
  - $n!$ is the product $n \times (n - 1) \times (n - 2) \ldots \times 2 \times 1$
- The number of tours grows incredibly quickly as we add cities to the map

#cities	#tours
5	12
6	60
7	360
8	2,520
9	20,160
10	181,440

The number of tours for 25 cities:

310,224,200,866,619,719,680,000

4

The Traveling Salesman

- Computer scientists call the problem of finding an optimal path between n points the traveling salesman problem (TSP)
- The TSP is a famous problem
  - first posed by Irish mathematician W. R. Hamilton in the 19th century
  - intensely studied in operations research and other areas since 1930

This tour of 13,500 US cities was generated by an advanced algorithm that used several "tricks" to limit the number of possible tours

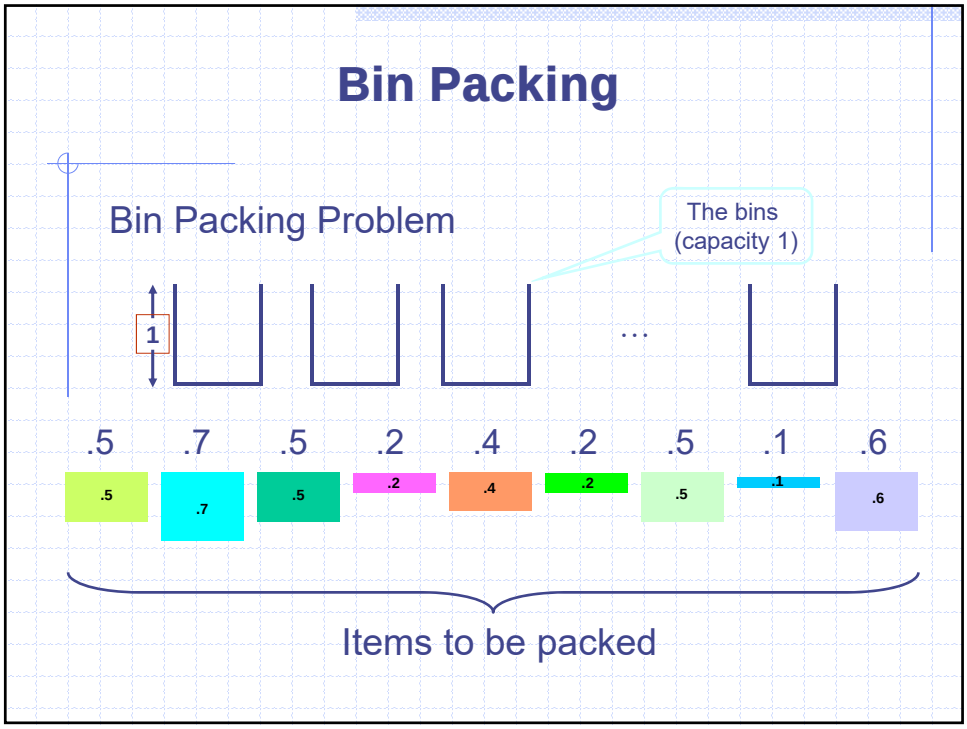
Required 5 "CPU-years" http://www.tsp.gatech.edu/

5

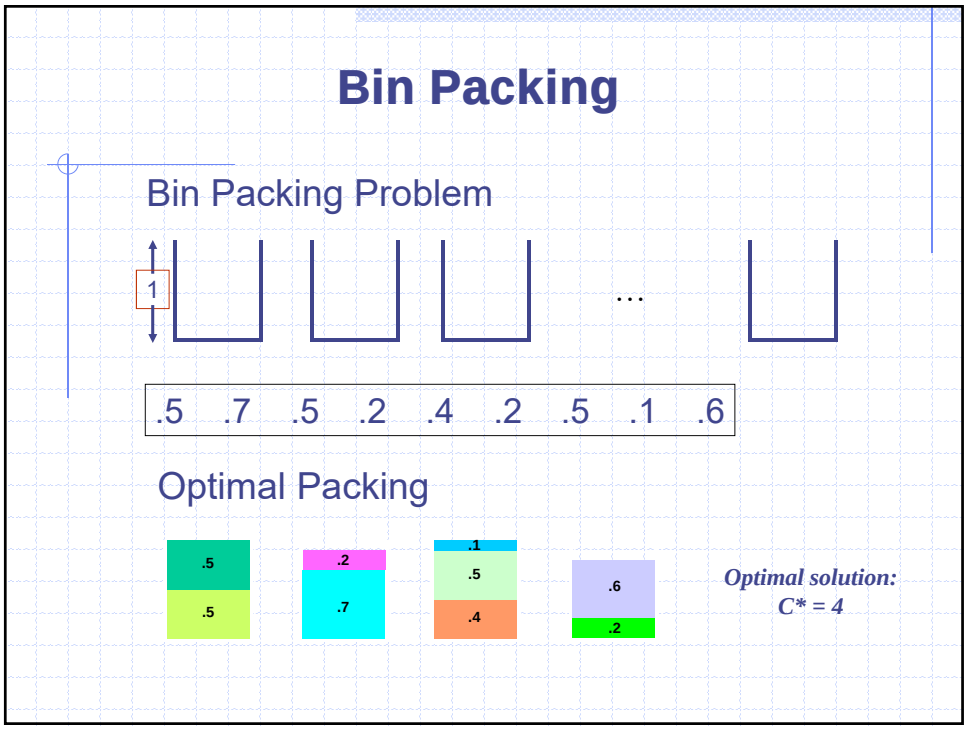
Dealing with Hard Combinatorial Problems (CP)

1. For many Combinatorial Problems (CP), not all instances are hard to solve. Some special cases can be solved easily, e.g, graph coloring is hard, but trees can be two-colored easily.
2. For small problems, it may be possible to solve them using backtracking search or branch-and-bound algorithms.
3. Look for good-enough approximate solutions.
4. Use random algorithms to find solutions with high probability.

6


Bin Packing
Bin Packing Problem
The bins
(capacity 1)
1
...
.5 .7 .5 .2 .4 .2 .5 .1 .6
Items to be packed

7

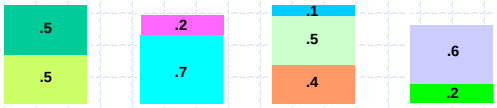

Bin Packing
Bin Packing Problem
1
...
.5 .7 .5 .2 .4 .2 .5 .1 .6
Optimal Packing
Optimal solution:
C* = 4

8

First Fit Packing Algorithm

Bin Packing Problem

.5 .7 .5 .2 .4 .2 .5 .1 .6



Optimal solution:
C = 4*

First Fit Packing Algorithm (FF): For each item s_i, find the first bin (from left to right) which can hold s_i.

FF solution: C = 5

$$\frac{C}{C^*} \leq 2$$

9

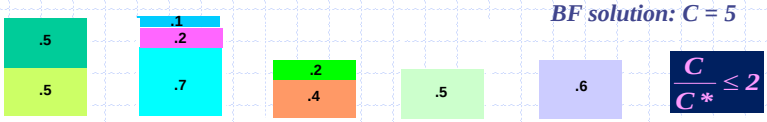
Best Fit Packing Algorithm

Bin Packing Problem

.5 .7 .5 .2 .4 .2 .5 .1 .6

Best Fit Packing Algorithm (BF): For each item s_i, find the most-full bin which can hold s_i.

BF solution: C = 5



$$\frac{C}{C^*} \leq 2$$

10

First Fit Decreasing Packing Algorithm

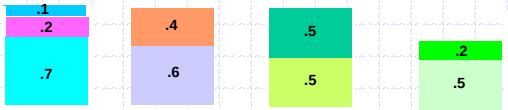
Bin Packing Problem

.5 .7 .5 .2 .4 .2 .5 .1 .6

First Fit Decreasing (FFD): First sort the items in decreasing order, then apply FF.

.7 .6 .5 .5 .5 .4 .2 .2 .1

FFD solution: C = C = 4*



$$\frac{C}{C^*} \le 2$$

11

Best Fit Decreasing Packing Algorithm

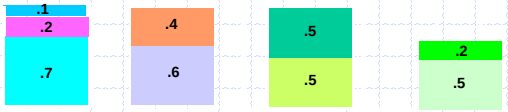
Bin Packing Problem

.5 .7 .5 .2 .4 .2 .5 .1 .6

Best Fit Decreasing (BFD): First sort the items in decreasing order, then apply BF.

.7 .6 .5 .5 .5 .4 .2 .2 .1

BFD solution: C = C = 4*



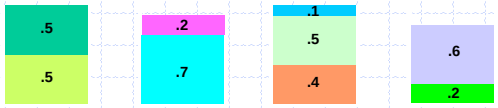
$$\frac{C}{C^*} \le 2$$

12

Next Fit Packing Algorithm

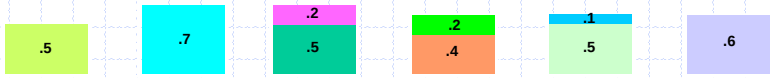
Bin Packing Problem

.5 .7 .5 .2 .4 .2 .5 .1 .6



C = 4*

Next Fit Packing Algorithm: If the current bin cannot hold s_i, start a new bin (good for online application).



C = 6

13

Approximation Terminology

- Given an instance I of an optimization problem, let C^* = OPT(I) be the cost of an optimal solution, and let C be the cost of the solution of an approximation algorithm. The algorithm has an *approximation ratio* of $\rho(n)$ if, for all solutions $\max(C/C^*, C^*/C) \leq \rho(n)$.
- We say that an approximation algorithm with an approximation ration of $\rho(n)$ is a $\rho(n)$-*approximation algorithm*.

14

Approximation Terminology

- An *approximation scheme* is an approximation algorithm that takes an instance and an $\varepsilon > 0$, and produces a $(1+\varepsilon)$ approximation ($\rho(n) = 1+\varepsilon$).
- If an approximation scheme runs in polynomial time in the size of its input when given any ε, we say it is a *polynomial-time approximation scheme*. Note that the running time could still increase rapidly as ε decreases.

15

Approximation Algorithms

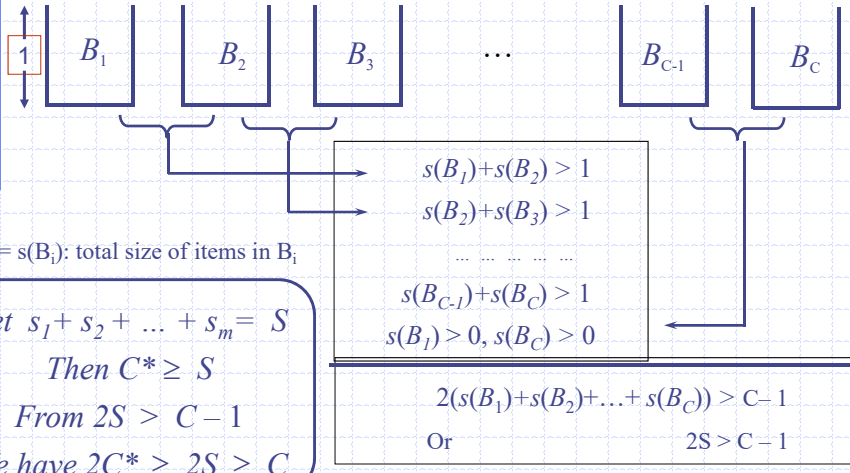
Not always optimal solution,
but with some performance guarantee
(eg, no worst than *twice the optimal*)

Even though
***we don't know what the optimal solution is*!!!**

16

Approximate Bin Packing Algorithms:

C* = OPT(I), C is the approximate cost



$s_i = s(B_i)$: total size of items in B_i

$s(B_1)+s(B_2) > 1$

$s(B_2)+s(B_3) > 1$

... ...

$s(B_{C-1})+s(B_C) > 1$

$s(B_1) > 0, s(B_C) > 0$

Let $s_1 + s_2 + \ldots + s_m = S$

Then $C^* \geq S$

From $2S > C - 1$

We have $2C^* \geq 2S \geq C$

$2(s(B_1)+s(B_2)+\ldots+s(B_C)) > C - 1$

Or $2S > C - 1$

17

Approximate Bin Packing Algorithms

Theorem

- First-Fit: $C \leq 17/10C^* + 2$
- First-Fit Decreasing: $C \leq 11/9C^* + 4$

18

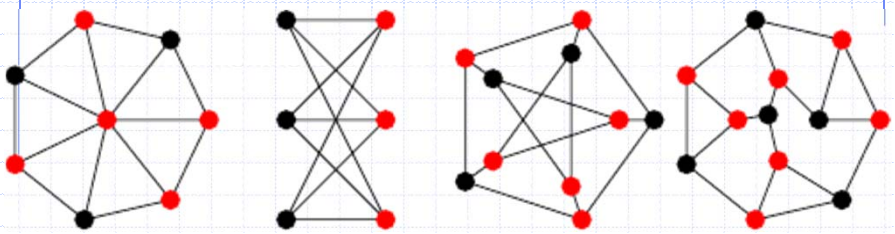
Vertex Cover Problem

- Let G=(V, E). The subset S of V that meets every edge of E is called the vertex cover.
- The Vertex Cover problem is to find a vertex cover of the minimum size. It is a hard combinatorial optimization problem (NP-complete).

19

19

Examples of vertex cover



Red points are in vertex cover.

20

20

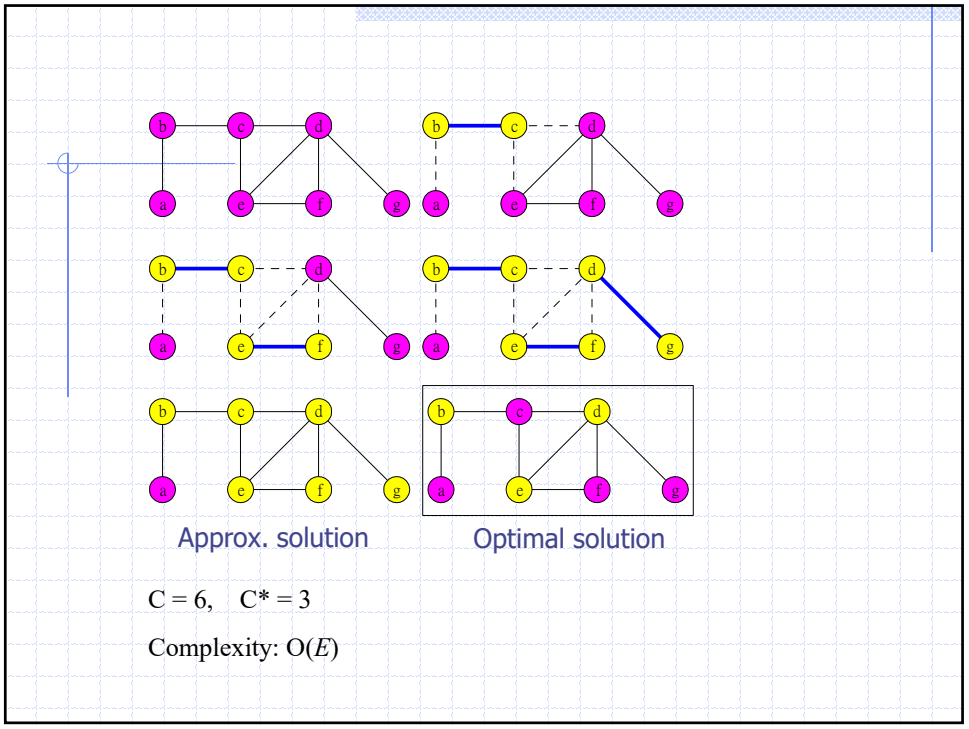
Approximating Vertex Cover

- The following algorithm is a 2-approximation algorithm for the vertex-cover optimization problem:
  - Choose an edge e from G.
  - Add both e's endpoints (say u and v) to the vertex cover.
  - Remove all edges incident to u and v.
  - Repeat while there are edges left.

21

```
APPROX_VERTEX_COVER(G)
1  C ← φ
2  E' ← E(G)
3  while E' ≠ φ
4      do let (u,v) be an arbitrary edge of E'
5          C ← C ∪ {u,v}
6          remove from E' every edge incident on either u or v
7  return C
```

22



C = 6, C* = 3

Complexity: O(E)

23

C*: the size of optimal solution
C: the size of approximate solution
A: the set of edges selected in step 4

Theorem

APPROX_VERTEX_COVER has ratio bound of 2.

Proof.

Let A be the set of selected edges.

$C = 2 | A |$ When one edge is selected, 2 vertices are added into C.

$| A | \leq C^*$ No two edges in A share a common endpoint due to step 6.

$\Rightarrow C \leq 2C^*$

24

Vertex Cover: Greedy Algorithm 2

Idea: Keep finding a vertex which covers the maximum number of edges.

Greedy Algorithm 2:

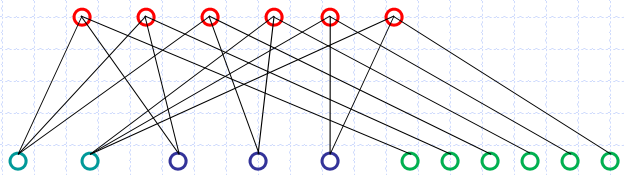
1. Find a vertex v with maximum degree.
2. Add v to the solution and remove v and all its incident edges from the graph.
3. Repeat until all the edges are covered.

How good is this algorithm?

25

25

Vertex Cover: Greedy Algorithm 2



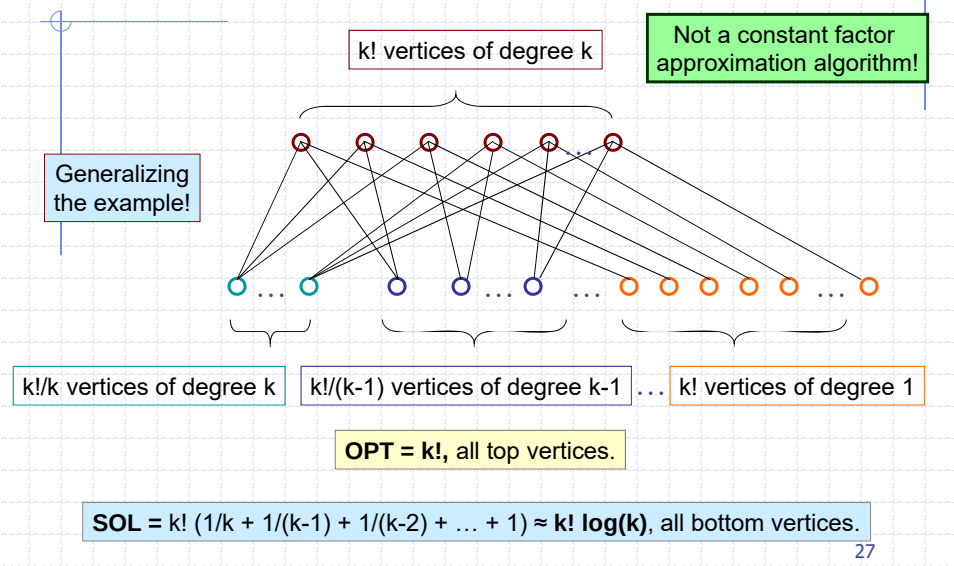
OPT = 6, all red vertices.

SOL = 11, if we are unlucky in breaking ties.
First we might choose all the green vertices.
Then we might choose all the blue vertices.
And then we might choose all the orange vertices.

26

26

Vertex Cover: Greedy Algorithm 2



27

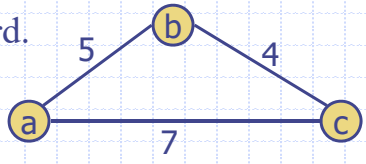
27

Euclidian Traveling Salesman Problem

- The costs in the traveling salesman problem do not need to represent distances (e.g., they may be things like airline fares).
- If the costs do represent geometric distances, they obey the *triangle inequality* for all vertices *u*, *v*, and *w*:

$$c(u,w) \leq c(u,v) + c(v,w)$$

- The problem remains very hard.



28

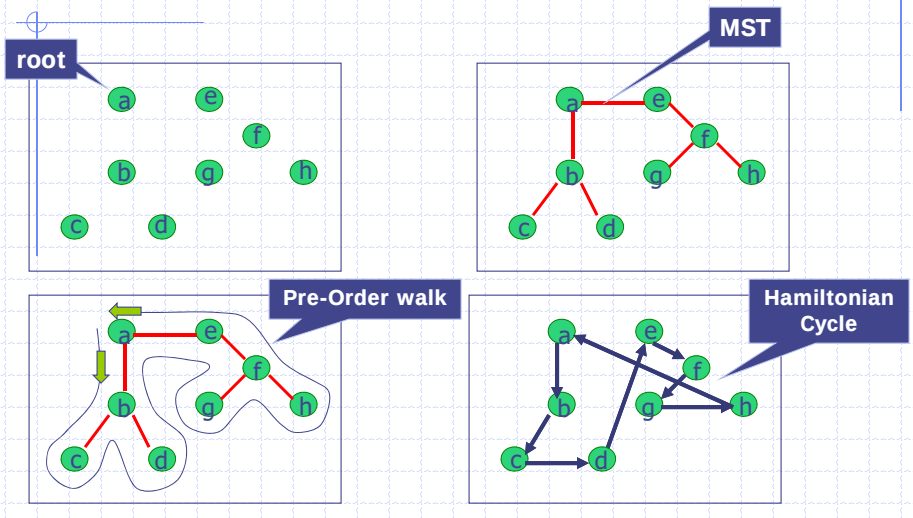
Euclidian Traveling Salesman Problem

Euclidian Traveling Salesman Problem is the Traveling Salesman Problem with the Triangle Inequality.

- The following is a 2-approximation of TSP with the triangle inequality:
  - compute a minimum spanning tree T of the weighted clique
  - order the vertices according to a preorder walk on T
  - return this order as the TSP tour

29

Euclidian Traveling Salesman Problem



30

30

Euclidian Traveling Salesman Problem

APPROX-TSP-TOUR(G, W)

1 select a vertex **r** ∈ V[**G**] to be root.
2 compute a **MST** for **G** from root r using Prim Alg.
3 **L** = list of vertices in preorder walk of that **MST**.
4 **return L** as the Hamiltonian cycle.

31

31

Traveling salesman problem

- This is polynomial-time 2-approximation algorithm. (Why?)
  - Because:
    - APPROX-TSP-TOUR is $O(V^2)$
    - $C(MST) \le C(H^*)$ — **Optimal**
      $C(W) = 2C(MST)$
      $C(W) \le 2C(H^*)$ — **Pre-order**
      $C(H) \le C(W)$
      $C(H) \le 2C(H^*)$ — **Solution**

H*: optimal soln
W: Preorder walk

H: approx soln & triangle inequality

32

32

Solving a maze

- Given a maze, find a path from start to finish
- At each intersection, you have to decide between three or fewer choices:
  - Go straight
  - Go left
  - Go right
- You don't have enough information to choose correctly
- Each choice leads to another set of choices
- One or more sequences of choices may (or may not) lead to a solution
- Many types of maze problem can be solved with backtracking

33

33

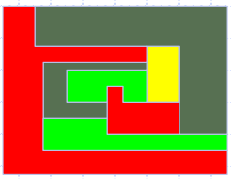
Backtracking

- Suppose you have to make a series of *decisions,* among various *choices,* where
  - You don't have enough information to know what to choose.
  - Each decision leads to a new set of choices
  - Some sequence of choices (possibly more than one) may be a solution to your problem
- Backtracking is a methodical way of trying out various sequences of decisions, until you find one that "works".

34

34

Coloring a map



- You wish to color a map with not more than four colors
  - red, yellow, green, blue
- Adjacent countries must be in different colors
- You don't have enough information to choose colors
- Each choice leads to another set of choices
- One or more sequences of choices may (or may not) lead to a solution
- Many coloring problems can be solved with backtracking

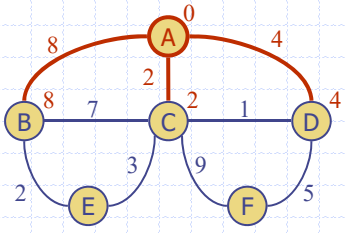
35

35

Longest Path Problem

Given two nodes A and B, find the longest (simple) path from A to B.

Solution: Starting from A, using backtrack search.



36

Real and virtual graphs

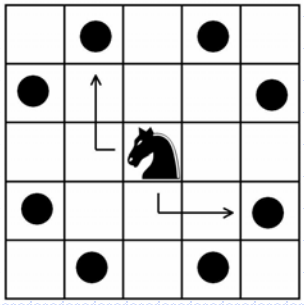
- There is a type of data structure called a graph
  - But we are **not** using it here
- If we diagram the sequence of choices we make, the diagram looks like a tree
  - In fact, we did just this in DFS algorithm.
  - Backtracking algorithm works in general in virtual graph (implicitly defined by the starting state and the consequence relation induced by decisions.
  - Our backtracking algorithm "sweeps out a tree" in "problem state space".

37

37

Knight Puzzle

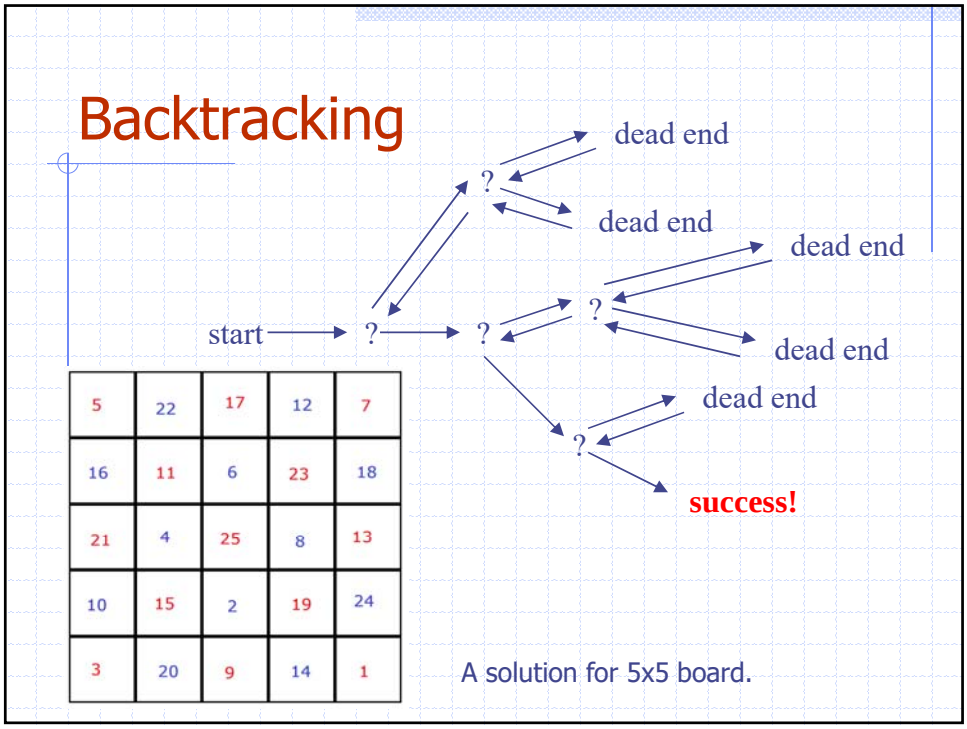
- In this puzzle, move the knight legally in the board.
- The object is to go to each position exactly once.



- You don't have enough information to jump correctly
- Each choice leads to another set of choices
- One or more sequences of choices may (or may not) lead to a solution
- Many kinds of puzzle can be solved with backtracking search.

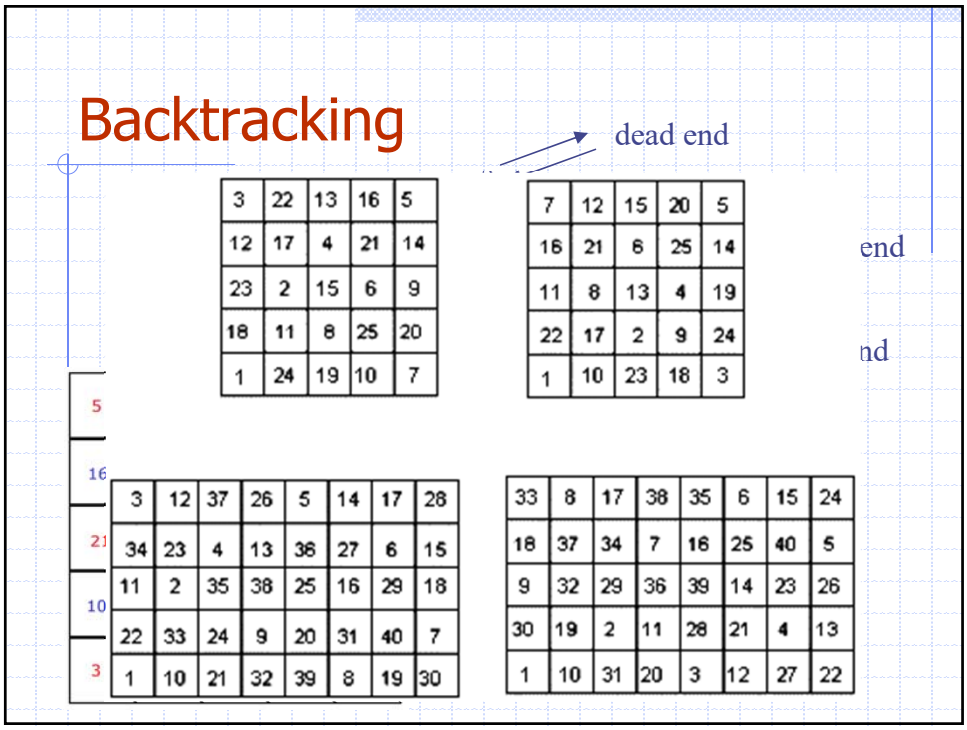
38

38



5	22	17	12	7
16	11	6	23	18
21	4	25	8	13
10	15	2	19	24
3	20	9	14	1

39



3	22	13	16	5			
12	17	4	21	14			
23	2	15	6	9			
18	11	8	25	20			
1	24	19	10	7			

7	12	15	20	5			
16	21	6	25	14			
11	8	13	4	19			
22	17	2	9	24			
1	10	23	18	3			

5							
16							
21							
10							
3							

3	12	37	26	5	14	17	28
34	23	4	13	36	27	6	15
11	2	35	38	25	16	29	18
22	33	24	9	20	31	40	7
1	10	21	32	39	8	19	30

33	8	17	38	35	6	15	24
18	37	34	7	16	25	40	5
9	32	29	36	39	14	23	26
30	19	2	11	28	21	4	13
1	10	31	20	3	12	27	22

40

The backtracking algorithm

- Backtracking is really quite simple--we "explore" each node, as follows:
- To "explore" node N:

proc Explore(N)

1. If N is a goal node, return "success"
2. If N is a leaf node, return "failure"
3. For each child C of N,
   - 3.1. result = Explore(C)
   - 3.2. If result was "success", return "success"
4. Return "failure"

41

DFS vs Backtracking Search

- DFS visits each node once and marks the node once it's visited. Hence the linear time complexity.
- Backtracking Search in general does not mark nodes, and may visit a node many times.
- Example: What algorithm to use?
  - Find a path in a graph: DFS
  - Find the shortest path in a graph: Dijkstra's
  - Find the longest simple path in a graph: Backtracking

42

Full example: Map coloring

- The Four Color Theorem states that any map on a plane can be colored with no more than four colors, so that no two countries with a common border are the same color
- For most maps, finding a legal coloring is easy
- For some maps, it can be fairly difficult to find a legal coloring
- We will develop a complete Java program to solve this problem

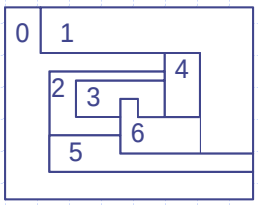
43

Data structures

- We need a data structure that is easy to work with, and supports:
  - Setting a color for each country
  - For each country, finding all adjacent countries
- We can do this with two arrays
  - An array of "colors", where `countryColor[i]` is the color of the i^{th} country
  - A ragged array of adjacent countries, where `map[i][j]` is the j^{th} country adjacent to country i
    - Example: `map[5][3]==8` means the 3^{th} country adjacent to country 5 is country 8

44

Creating the map




```
int map[][];

void createMap() {
   map = new int[7][];
   map[0] = new int[] { 1, 4, 2, 5 };
   map[1] = new int[] { 0, 4, 6, 5 };
   map[2] = new int[] { 0, 4, 3, 6, 5 };
   map[3] = new int[] { 2, 4, 6 };
   map[4] = new int[] { 0, 1, 6, 3, 2 };
   map[5] = new int[] { 2, 6, 1, 0 };
   map[6] = new int[] { 2, 3, 4, 1, 5 };
}
```

45

45

Setting the initial colors

```
static final int NONE = 0;
static final int RED = 1;
static final int YELLOW = 2;
static final int GREEN = 3;
static final int BLUE = 4;

int mapColors[] = { NONE, NONE, NONE, NONE,
                    NONE, NONE, NONE };
```

46

46

The main program

(The name of the enclosing class is ColoredMap)

```
public static void main(String args[]) {
    ColoredMap m = new ColoredMap();
    m.createMap();
    boolean result = m.explore(0, RED);
    System.out.println(result);
    m.printMap();
}
```

47

47

The backtracking method

```
boolean explore(int country, int color) {
    if (country >= map.length) return true;
    if (okToColor(country, color)) {
        mapColors[country] = color;
        for (int i = RED; i <= BLUE; i++) {
            if (explore(country + 1, i)) return true;
        }
    }
    return false;
}
```

48

48

Checking if a color can be used

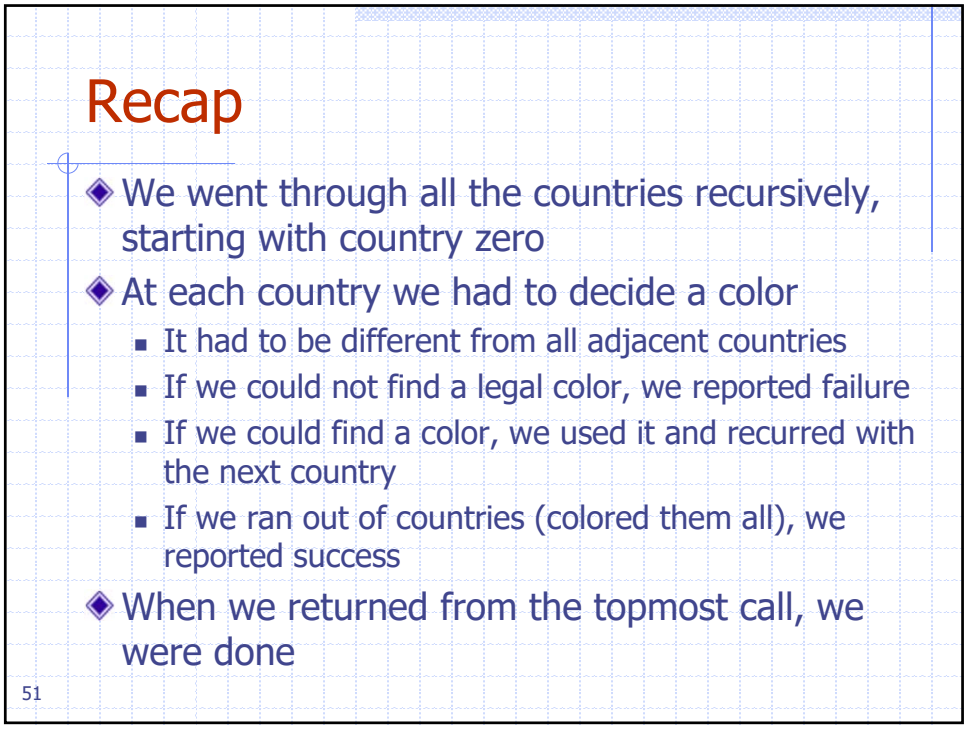
```
boolean okToColor(int country, int color) {
    for (int i = 0; i < map[country].length; i++) {
        int ithAdjCountry = map[country][i];
        if (mapColors[ithAdjCountry] == color) {
            return false;
        }
    }
    return true;
}
```

49

Printing the results

```
void printMap() {
    for (int i = 0; i < mapColors.length; i++) {
        System.out.print("map[" + i + "] is ");
        switch (mapColors[i]) {
            case NONE:   System.out.println("none");   break;
            case RED:    System.out.println("red");    break;
            case YELLOW: System.out.println("yellow"); break;
            case GREEN:  System.out.println("green");  break;
            case BLUE:   System.out.println("blue");   break;
        }
    }
}
```

50


Recap
We went through all the countries recursively, starting with country zero
At each country we had to decide a color
It had to be different from all adjacent countries
If we could not find a legal color, we reported failure
If we could find a color, we used it and recurred with the next country
If we ran out of countries (colored them all), we reported success
When we returned from the topmost call, we were done
51

51