Algorithms (2IL15)

3th Quartile, 2014

www.win.tue.nl/~kbuchin/teaching/2IL15/

Lecturer: Kevin Buchin (MF 6.093, k.a.buchin@tue.nl)
Organization of the course

Similar to Datastructures

• homework exercises

• tutorials (for help in solving homework + discussing solutions)

• minimum score needed for homework to be admitted to exam

• registration via OASE mandatory (at the latest today), register for group this week (Tuesday-Thursday)

Literature: same as Datastructures

Organization of the course

Part I: Techniques for optimization
- backtracking exercises
- greedy algorithms exercises
- dynamic programming I exercises
- dynamic programming II exercises

homework set A.1
homework set A.2

Part II: Graph algorithms
- shortest paths I exercises
- shortest paths II exercises
- max flow exercises
- matching exercises

homework set B.1
homework set B.2

Part III: Selected topics
- NP-hardness I exercises
- NP-hardness II exercises
- approximation algorithms exercises
- linear programming exercises

homework set C.1
homework set C.2
## Planning

Below is the (preliminary) schedule for the course. Please note that the schedule may change during the course. Deadlines are indicated in red. Homework exercises will become due in due time.

<table>
<thead>
<tr>
<th>Part of the course</th>
<th>week</th>
<th>lectures + tutorials</th>
<th>Slides / Homework</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>Mon</strong></td>
<td></td>
</tr>
<tr>
<td>Optimization</td>
<td></td>
<td><strong>Fri</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lecture 2: Greedy algorithms. Chapter 16, Sections 16.1—16.3. The chapter refers several times to dynamic programming (Chapter 15), which we will discuss later. Upon first reading, you may skip the parts referring to dynamic programming.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Feb 10 – 14</td>
<td><strong>Mon</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lecture 3: Dynamic programming I. Chapter 15, sections 15.1 + 15.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Fri</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lecture 4: Dynamic programming II. Chapter 15, sections 15.3 – 15.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Fri</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tutorial 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Fri</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tutorial 2</td>
<td></td>
</tr>
<tr>
<td>Graph Algorithms</td>
<td>Feb 17 – 21</td>
<td><strong>Mon</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Thu</strong></td>
<td>23:59 deadline for Homework Set A (= Homeworks A.1 and A.2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Fri</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lecture 6: All-Pairs Shortest paths. Chapter 25, except 25.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Fri</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tutorial 3: Discussion of solutions to Homework Set A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Feb 24 – 28</td>
<td><strong>Mon</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lecture 7: Max flow (part 1). Chapter 26 until (and excluding) “Analysis of Ford-Fulkerson”.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Fri</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lecture 8: Max flow (continued) + Max matching and other applications. Rest of Section 26.2 + section 26.3.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Fri</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tutorial 4</td>
<td></td>
</tr>
<tr>
<td>Carnival</td>
<td>March 3 – 7</td>
<td>no lectures or tutorial</td>
<td></td>
</tr>
<tr>
<td></td>
<td>March 10 – 14</td>
<td><strong>Mon</strong></td>
<td>12:00 deadline for Homework Set B (= Homeworks B.1 and B.2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Mon</strong></td>
<td>Lecture 9: NP-completeness. Chapter 34, sections 34.1—34.4. [NB The book sometimes contains slightly more details than I expect you to know. Have a look at the slides to see what you should know.]</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Fri</strong></td>
<td>Lecture 10: NP-completeness, part II. Section 34.5.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Fri</strong></td>
<td>Tutorial 5: Discussion of solutions to Homework Set B</td>
</tr>
<tr>
<td></td>
<td>March 17 – 21</td>
<td><strong>Mon</strong></td>
<td>Lecture 11: Approximation algorithms: Sections 35.1, 35.2, 35.5.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Fri</strong></td>
<td>Lecture 12: Linear programming: Sections 29.1, 29.2</td>
</tr>
</tbody>
</table>

and so on …: see the course webpage
Grading

Homework (no copying, solutions submitted in pdf by email to instructor)
- six sets: two sets for each of the three parts
  (but: handed in per part, so only three deadlines)
- best four sets count, but at least one per part
- maximum homework score: $4 \times 10 = 40$ points

Exam
- need at least 20 points for Homework
- maximum score = 10 points

Final grade
- need at least 5 points for exam, otherwise FAIL
- if final exam at least 5: final grade = \( \frac{(\text{homework} + 4 \times \text{exam})}{8} \)
Part I: Techniques for optimization
Optimization problems

- for each instance there are (possibly) multiple valid solutions
- goal is to find an **optimal solution**

- minimization problem:
  associate cost to every solution, find **min-cost solution**

- maximization problem:
  associate profit to every solution, find **max-profit solution**
Optimization problems: examples

Traveling Salesman Problem

- input = set of $n$ cities with distances between them
- valid solution = tour visiting all cities
- cost = length of tour

![Traveling Salesman Problem Graph]

```
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>-</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td>-</td>
<td>15</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>
```
Optimization problems: examples

Traveling Salesman Problem
- input = set of \( n \) cities with distances between them
- valid solution = tour visiting all cities
- cost = length of tour

Knapsack
- input = \( n \) items, each with a weight and a profit, and value \( W \)
- valid solution = subset of items whose total weight is \( \leq W \)
- profit = total profit of all items in subset

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>weight</td>
<td>5</td>
<td>7</td>
<td>5</td>
<td>8</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>profit</td>
<td>6</td>
<td>7</td>
<td>4.5</td>
<td>10</td>
<td>14</td>
<td>5</td>
</tr>
</tbody>
</table>

solutions:
1,2,6: weight 18, profit 18

\( W = 18 \)
Optimization problems: examples

Traveling Salesman Problem
- input = set of $n$ cities with distances between them
- valid solution = tour visiting all cities
- cost = length of tour

Knapsack
- input = $n$ items, each with a weight and a profit, and value $W$
- valid solution = subset of items whose total weight is $\leq W$
- profit = total profit of all items in subset

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>weight</td>
<td>5</td>
<td>7</td>
<td>5</td>
<td>8</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>profit</td>
<td>6</td>
<td>7</td>
<td>4.5</td>
<td>10</td>
<td>14</td>
<td>5</td>
</tr>
</tbody>
</table>

solutions:
- 1,2,6: weight 18, profit 18
- 2,5: weight 18, profit 21
- etcetera

$W = 18$
Optimization problems: examples

Traveling Salesman Problem
- input = set of \( n \) cities with distances between them
- valid solution = tour visiting all cities
- cost = length of tour

Knapsack
- input = \( n \) items, each with a weight and a profit, and value \( W \)
- valid solution = subset of items whose total weight is \( \leq W \)
- profit = total profit of all items in subset

Linear Programming
minimize: \( c_1 x_1 + \cdots + c_n x_n \)
subject to: \( a_{1,1} x_1 + \cdots + a_{1,n} x_n \leq b_1 \)
\[ a_{m,1} x_1 + \cdots + a_{m,n} x_n \leq b_m \] even hard to find any solution!
Techniques for optimization

optimization problems typically involve making choices

- **backtracking**: just try all solutions
  - can be applied to almost all problems, but gives very slow algorithms
  - try all options for first choice,
    for each option, recursively make other choices

- **greedy algorithms**: construct solution iteratively, always make choice that seems best
  - can be applied to few problems, but gives fast algorithms
  - only try option that seems best for first choice (greedy choice), recursively make other choices

- **dynamic programming**
  - in between: not as fast as greedy, but works for more problems
Today: backtracking + how to (slightly?) speed it up

Example 1: Traveling Salesman Problem (TSP)

given: $n$ cities and the (non-negative) distances between them

Input: matrix $\text{Dist}[1..n,1..n]$, where $\text{Dist}[i,j] =$ distance from $i$ to $j$

goal: find shortest tour visiting all cities and returning to starting city

Output: permutation of $\{1,\ldots,n\}$ such that visiting cities in that order gives min-length tour

choices: what is first city to visit?
what is second city to visit?
...
what is last city to visit?
Backtracking for TSP:

- first city is city 1
- try all remaining cities as next city
  for each option for next city, recursively try all ways to finish the tour

  for each recursive call:
  - remember which choices we already made
    ( = part of the tour we fixed in earlier calls)
  - and which choices we still need to make
    ( = remaining cities, for which we need to decide visiting order)

parameters of algorithm

- when all choices have been made:
  compute length of tour, compare to length of shortest tour found so far
Parameters: 

- $R$ = sequence of already visited cities (initially: $R = \text{city 1}$)
- $S$ = set of remaining cities (initially: $S = \{ 2, \ldots, n \}$)

We want to compute a shortest tour visiting all cities in $R \cup S$, under the condition that the tour starts by visiting the cities from $R$ in the given order.

**Algorithm** $TSP_{\text{BruteForce1}} (R, S)$

1. if $S$ is empty all choices have been made
2. then $minCost \leftarrow \text{length of the tour represented by } R$
3. else $minCost \leftarrow \infty$
4. for each city $i$ in $S$ try all remaining cities as next city
5. do Remove $i$ from $S$, and append $i$ to $R$. $i$ is next city
6. $minCost \leftarrow \min(minCost, TSP_{\text{BruteForce1}} (R, S))$
7. Reinsert $i$ in $S$, and remove $i$ from $R$. undo choice
8. return $minCost$

recursively compute best way to make remaining choices
Parameters: \( R \) = sequence of already visited cities (initially: \( R = \text{city 1} \))
\( S \) = set of remaining cities (initially: \( S = \{ 2, \ldots, n \} \))

We want to compute a shortest tour visiting all cities in \( R \cup S \), under the condition that the tour starts by visiting the cities from \( R \) in the given order.

**Algorithm** \( TSP_{BruteForce1} (R, S) \)

1. if \( S \) is empty  all choices have been made
2. recursively compute best way to make remaining choices
3. return \( \text{minCost} \)
Parameters: $R =$ sequence of already visited cities (initially: $R =$ city 1)
$S =$ set of remaining cities (initially: $S = \{2, \ldots, n\}$)

Possible implementation: $R$ and $S$ are linked lists

Analysis: $n_R =$ size of $R$, $n_S =$ size of $S$

Algorithm $TSP\_BruteForce1 (R, S)$

1. if $S$ is empty
2. then $minCost \leftarrow$ length of the tour represented by $R$ $O(n_R)$
3. else $minCost \leftarrow \infty$
4. for each city $i$ in $S$
5. do Remove $i$ from $S$, and append $i$ to $R$. $O(1)$
6. $minCost \leftarrow \min(minCost, TSP\_BruteForce1 (R, S))$
7. Reinsert $i$ in $S$, and remove $i$ from $R$. $O(1)$
8. return $minCost$

$T(n_R, n_S) = n_S \cdot (O(1) + T(n_R+1, n_S-1))$ with $T(n_R, 0) = O(n_R)$
Parameters: \( R = \) sequence of already visited cities (initially: \( R = \) city 1) \\
\( S = \) set of remaining cities (initially: \( S = \{2, \ldots, n\} \) )

Possible implementation: \( R \) and \( S \) are linked lists

Analysis: \( n_R = \) size of \( R \), \( n_S = \) size of \( S \)

**Algorithm** \( TSP_{\text{BruteForce1}} (R, S) \)

1. if \( S \) is empty
2. then \( \text{minCost} \leftarrow \) length of the tour represented by \( R \) \( \quad \Theta(n_R) \)
3. else \( \text{minCost} \leftarrow \infty \)
4. for each city \( i \) in \( S \)
5. do Remove \( i \) from \( S \), and append \( i \) to \( R \). \( \quad \Theta(1) \)
6. \( \text{minCost} \leftarrow \min(\text{minCost}, TSP_{\text{BruteForce1}} (R, S)) \)
7. Reinsert \( i \) in \( S \), and remove \( i \) from \( R \). \( \quad \Theta(1) \)
8. return \( \text{minCost} \)

\[ T(n_R, n_S) = n_S \cdot (\Theta(1) + T(n_R+1, n_S-1)) \quad \text{with} \quad T(n_R, 0) = \Theta(n_R) \]
$R$ = sequence of already visited cities (initially: $R$ = city 1)
$S$ = set of remaining cities (initially: $S$ = \{ 2, \ldots, n \} )

Alternative implementation:
- $A[1..n]$ = array of cities, parameter $1 \leq k \leq n$

**Algorithm** $TSP\_BruteForce1 ~(A,k~)$
1. $n \leftarrow \text{length}[A]$
2. \hspace{1em} if $k = n$
3. \hspace{2em} then $\text{minCost} \leftarrow \text{length of the tour represented by } A[1..n]$
4. \hspace{1em} else $\text{minCost} \leftarrow \infty$
5. \hspace{2em} for $i \leftarrow k+1$ to $n$
6. \hspace{3em} do Swap $A[i]$ and $A[k+1]$
7. 
8. \hspace{1em} $\text{minCost} \leftarrow \text{min}(\text{minCost}, TSP\_BruteForce1 ~(A,k+1~))$
9. 
10. return $\text{minCost}$
$R =$ sequence of already visited cities (initially: $R =$ city 1)
$S =$ set of remaining cities (initially: $S = \{2, \ldots, n\}$)

Alternative implementation: ▪ $A[1..n] =$ array of cities, parameter $1 \leq k \leq n$

improvement: maintain $\text{lengthSoFar} =$ length of initial part given by $A[1] \ldots A[k]$

Algorithm $\text{TSP\_BruteForce1} (A,k)$
1. $n \leftarrow \text{length}[A]$
2. if $k = n$
3. then $\text{minCost} \leftarrow \text{length of the tour represented by } A[1..n]$
4. else $\text{minCost} \leftarrow \infty$
5. for $i \leftarrow k+1$ to $n$
6. do Swap $A[i]$ and $A[k+1]$
7. $\text{newLength} \leftarrow \text{lengthSoFar} + \text{Dist}[A[k],A[1]]$
8. $\text{minCost} \leftarrow \min(\text{minCost}, \text{TSP\_BruteForce1}(A,k+1))$
9. Swap $A[k+1]$ and $A[i]$
10. return $\text{minCost}$
\( R = \) sequence of already visited cities (initially: \( R = \) city 1)
\( S = \) set of remaining cities (initially: \( S = \{2, \ldots, n\} \))

Alternative implementation:  
- \( A [1..n] = \) array of cities, parameter \( 1 \leq k \leq n \)
  - \( A [1..k] \) contains \( R \), \( A [k+1 \ldots n] \) contains \( S \)

improvement: maintain \( \text{lengthSoFar} = \) length of initial part given by \( A[1] \ldots A[k] \)

\textbf{Algorithm} \( \text{TSP\_BruteForce1} (A,k,) \)
\begin{enumerate}
  \item \( n \leftarrow \text{length}[A] \)
  \item \textbf{if} \( k = n \)
    \begin{itemize}
      \item \( \text{lengthSoFar} + \text{Dist} [A[n], A[1]] \)
    \end{itemize}
  \item \textbf{then} \( \text{minCost} \leftarrow \text{length of the tour represented by } A[1..n] \)
  \item \textbf{else} \( \text{minCost} \leftarrow \infty \)
\end{enumerate}

Old running time: \( O(n) \) for all permutations of \( 2, \ldots n \), so \( O(n!) \) in total

New running time: \( O((n-1)!) \) still very very slow
A [1..n] = array of cities, parameter 1 ≤ k ≤ n
A [1..k] contains initial part of tour, A [k+1 .. n] contains remaining cities

pruning: don’t recurse if initial part cannot lead to optimal tour

Algorithm TSP_BruteForce1 (A,k, lengthSoFar, )
1. \( n \leftarrow \text{length}[A] \)
2. \( \text{if } k = n \)
3. \( \text{then } \text{minCost} \leftarrow \text{min} (\text{minCost}, \text{lengthSoFar} + \text{Dist} [A[n], A[1]]) \)
4. \( \text{else } \text{minCost} \leftarrow \infty \)
5. \( \text{for } i \leftarrow k+1 \text{ to } n \)
6. \( \text{do } \text{Swap } A[i] \text{ and } A[k+1] \)
7. \( \text{newLength} \leftarrow \text{lengthSoFar} + \text{Dist} [A[k], A[k+1]] \)
8. \( \text{if } \text{newLength} \geq \text{minCost} \text{ then skip } \)
9. \( \text{else } \text{minCost} \leftarrow \text{min}(\text{minCost}, \text{TSP_BruteForce1}(A,k+1)) \)
10. \( \text{Swap } A[k+1] \text{ and } A[i] \)
11. \( \text{return } \text{minCost} \)
Intermezzo: Queens Problem (on NxN board)

- backtracking: not only for optimization problems
- parameters of algorithm? running time?
backtracking + pruning (gives \textit{branch-and-bound} algorithm)

Example 2: 1-dimensional clustering

given: $X = \text{set of } n \text{ numbers (points in 1D), parameter } k$

goal: find partitioning of $S$ into $k$ clusters of minimum cost

1, 3, 4, 6: average = 3.5, cost = 6

- cost of single cluster: sum of distance to cluster average
- cost of total clustering: sum of costs of its clusters
backtracking + pruning (gives \textit{branch-and-bound} algorithm)

Example 2: 1-dimensional clustering

given: $X =$ set of $n$ numbers (points in 1D), parameter $k$
assume points are sorted from left to right
goal: find partitioning of $S$ into $k$ clusters of minimum cost

choices to be made
- cluster 1 always starts at point 1
- we have a choice where to start clusters 2, ..., $k$
Backtracking for 1-dimensional clustering:

- first cluster starts at point 1

- try all remaining points as starting point for next cluster
  for each option, recursively try all ways to finish the clustering

  for each recursive call:
  - remember which choices we already made
    ( = starting points we fixed in earlier calls)
  - and which choices we still need to make
    ( = starting points for remaining clusters)

parameters of algorithm

- when all choices have been made:
  compute cost of clustering, compare to best clustering found so far
Parameters: $X_{[1..n]} = \text{sorted set of numbers (points in 1-dimensional space)}$

$k = \text{desired number of clusters}$, $A_{[1..k]} = \text{starting points}$

$m = \text{number of starting points that have already been fixed}$

We want to compute an optimal partitioning of $X$ into $k$ clusters, under the condition that the starting points of clusters $1,..,m$ are as given by $A[1..m]$.

**Algorithm** *Clustering* $(X,k,A,m)$

1. **if** $m = k$
2. **then** $\text{minCost} \leftarrow \text{total cost of clustering represented by } A[1..k]$
3. **else** $\text{minCost} \leftarrow \infty$
4. **for** $i \leftarrow A[m] + 1 \text{ to } n - (k-m-1)$
5. **do** $A[m+1] \leftarrow i$
6. $\text{minCost} \leftarrow \min(\text{minCost}, \text{Clustering} (X,k,A,m+1))$
7. $A[m+1] \leftarrow \text{nil}$
8. **return** $\text{minCost}$

**number of clusterings:** $\binom{n}{k} = O(n^k)$

**running time:** $O(n^{k+1})$
Parameters: \( X[1..n] \) = sorted set of numbers (points in 1-dimensional space)
\( k \) = desired number of clusters,
\( A[1..k] \) = starting points
\( m \) = number of starting points that have already been fixed

**Pruning:** do not recurse if solution cannot get better than current best

**Algorithm** Clustering \((X,k,A,m, \text{minCost, costSoFar})\)
1. if \( m = k \)
2. then \( \text{minCost} \leftarrow \min(\text{minCost, costSoFar} + \text{cost last cluster}) \)
3. else \( \text{minCost} \leftarrow \infty \)
4. for \( i \leftarrow A[m] + 1 \) to \( n - (k-m-1) \)
5. do \( A[m+1] \leftarrow i \)
   \( \text{newCost} \leftarrow \text{costSoFar} + \text{cost} \ m\text{-th cluster} \)
   if \( \text{newCost} \geq \text{minCost} \) then skip
6. else \( \text{minCost} \leftarrow \min(\text{minCost, Clustering}(X,k,A,m+1)) \)
7. \( A[m+1] \leftarrow \text{nil} \)
8. return \( \text{minCost} \)

\( \text{minCost, newCost} \)
Parameters: \( X[1..n] \) = sorted set of numbers (points in 1-dimensional space)

\( k \) = desired number of clusters, \( A[1..k] \) = starting points

\( m \) = number of starting points that have already been fixed

Algorithm Clustering ← (\( X, k, A, m, \) ←minCost, ←costSoFar)

1. if \( m = k \) then

2. \( \text{minCost} ← \text{min}(\text{minCost}, \text{costSoFar} + \text{cost last cluster}) \)

3. else

4. for \( i ← A[m] + 1 \) to \( n - (kXmX1) \) do

5. \( A[m+1] ← i \)

6. \( \text{newCost} ← \text{costSoFar} + \text{cost} mX \text{th cluster} \)

7. if \( \text{newCost} ≥ \text{minCost} \) then skip

8. \( \text{else minCost} ← \text{min}(\text{minCost}, \text{Clustering} (X,k,A,m+1)) \)

9. \( A[m+1] ← \text{nil} \)

10. return \( \text{minCost} \)

NOTE:

This problem can actually be solved much more efficiently, for example using dynamic programming (as we will see later).
Intermezzo: Map Coloring problem

- adjacent regions should have different colors
- parameters of algorithm? running time?
- what if we want to check if 2 colors are enough?
Summary

- backtracking solves optimization problem by generating all solutions
- this is done best using a recursive algorithm
- backtracking is typically very slow
- speed can be improved using pruning, but it is usually hard to prove anything about how much the running time improves (and algorithm usually remains slow)
- there is a trade-off: more complicated pruning tests may result in fewer recursive calls, but pruning itself becomes more expensive