10.1 Heuristics

Heuristics

In practice, solving a problem in the optimal or most accurate way may require more computational resources than are available or feasible. Algorithms implemented for such problems often use a ***heuristic***: A technique that willingly accepts a non-optimal or less accurate solution in order to improve execution speed.

PARTICIPATION ACTIVITY 10.1.1: Introduction to the knapsack problem.

Animation content:

`undefined`

Animation captions:

1. A knapsack is a container for items, much like a backpack or bag. Suppose a particular knapsack can carry at most 30 pounds worth of items.
2. Each item has a weight and value. The goal is to put items in the knapsack such that the weight <= 30 pounds and the value is maximized.
3. Taking a 20 pound item with an 8 pound item is an option, worth $142.
4. If more than 1 of each item can be taken, 2 of item 1 and 1 of item 4 provide a better option, worth $145.
5. Trying all combinations will give an optimal answer, but is time consuming. A heuristic algorithm may choose a simpler, but non-optimal approach.

PARTICIPATION ACTIVITY 10.1.2: Heuristics.

1) A heuristic is a way of producing an optimal solution to a problem.

○ True

○ False

2) A heuristic technique used for numerical computation may sacrifice accuracy to gain speed.

○ True

○ False

PARTICIPATION ACTIVITY 10.1.3: The knapsack problem.

Refer to the example in the animation above.

1) Which of the following options provides the best value?

○ 5 6-pound items

○ 2 6-pound items and 1 18-pound item

○ 3 8-pound items and 1 6-pound item.

2) The optimal solution has a value of $162 and has one of each item: 6-pound, 8-pound, and 18-pound.

○ True

○ False

3) Which approach would guarantee finding an optimal solution?

○ Taking the largest item that fits in the knapsack repeatedly until no more items will fit.

○ Taking the smallest item repeatedly until no more items will fit in the knapsack.

○ Trying all combinations of items and picking the one with maximum value.

Heuristic optimization

A ***heuristic algorithm*** is an algorithm that quickly determines a near optimal or approximate solution. Such an algorithm can be designed to solve the ***0-1 knapsack problem***: The knapsack problem with the quantity of each item limited to 1.

A heuristic algorithm to solve the 0-1 knapsack problem can choose to always take the most valuable item that fits in the knapsack's remaining space. Such an algorithm uses the heuristic of choosing the highest value item, without considering the impact on the remaining choices. While the algorithm's simple choices are aimed at optimizing the total value, the final result may not be optimal.

PARTICIPATION ACTIVITY 10.1.4: Non-optimal, heuristic algorithm to solve the 0-1 knapsack.

Animation content:

`undefined`

Animation captions:

1. The item list is sorted and the most valuable item is put into the knapsack first.
2. No remaining items will fit in the knapsack.
3. The resulting value of $95 is inferior to taking the 12 and 8 pound items, collectively worth $102.
4. The heuristic algorithm sacrifices optimality for efficiency and simplicity.

PARTICIPATION ACTIVITY 10.1.5: Heuristic algorithm and the 0-1 knapsack problem.

1) Which is not commonly sacrificed by a heuristic algorithm?

- ○ speed
- ○ optimality
- ○ accuracy

2) What restriction does the 0-1 knapsack problem have, in comparison with the regular knapsack problem?

- ○ The knapsack's weight limit cannot be exceeded.
- ○ At most 1 of each item can be taken.
- ○ The value of each item must be less than the item's weight.

3) Under what circumstance would the Knapsack01 function not put the most

valuable item into the knapsack?

- ○ The item list contains only 1 item.
- ○ The weight of the most valuable item is greater than the knapsack's limit.

Self-adjusting heuristic

A ***self-adjusting heuristic*** is an algorithm that modifies a data structure based on how that data structure is used. Ex: Many self-adjusting data structures, such as red-black trees and AVL trees, use a self-adjusting heuristic to keep the tree balanced. Tree balancing organizes data to allow for faster access.

Ex: A self-adjusting heuristic can be used to speed up searches for frequently-searched-for list items by moving a list item to the front of the list when that item is searched for. This heuristic is self-adjusting because the list items are adjusted when an search is performed.

PARTICIPATION ACTIVITY 10.1.6: Move-to-front self-adjusting heuristic.

Animation content:

```
undefined
```

Animation captions:

1. 42 is at the end of a list with 8 items. A linear search for 42 compares against 8 items.
2. The move-to-front heuristic moves 42 to the front after the search.
3. Another search for 42 now only requires 1 comparison. 42 is left at the front of the list.
4. A search for 64 compares against 8 items and moves 64 to the front of the list.
5. 42 is no longer at the list's front, but a search for 42 need only compare against 2 items.

PARTICIPATION ACTIVITY 10.1.7: Move-to-front self-adjusting heuristic.

Suppose a move-to-front heuristic is used on a list that starts as (56, 11, 92, 81, 68, 44).

1) A first search for 81 compares against how many list items?

- ○ 0
- ○ 3
- ○ 4

2) A subsequent search for 81 compares against how many list items?

- ○ 1
- ○ 2
- ○ 4

3) Which scenario results in faster searches?

- ○ Back-to-back searches for the same key.
- ○ Every search is for a key different than the previous search.

How was this section? Provide feedback

10.2 Greedy algorithms

Greedy algorithm

A ***greedy algorithm*** is an algorithm that, when presented with a list of options, chooses the option that is optimal at that point in time. The choice of option does not consider additional subsequent options, and may or may not lead to an optimal solution.

PARTICIPATION ACTIVITY 10.2.1: MakeChange greedy algorithm.

Animation content:

`undefined`

Animation captions:

1. The change making algorithm uses quarters, dimes, nickels, and pennies to make change equaling the specified amount.

2. The algorithm chooses quarters as the optimal coins, as long as the remaining amount is >= 25.
3. Dimes offer the next largest amount per coin, and are chosen while the amount is >= 10.
4. Nickels are chosen next. The algorithm is greedy because the largest coin <= the amount is always chosen.
5. Adding one penny makes 91 cents.
6. This greedy algorithm is optimal and minimizes the total number of coins, although not all greedy algorithms are optimal.

PARTICIPATION ACTIVITY 10.2.2: Greedy algorithms.

1) If the MakeChange function were to make change for 101, what would be the result?

- ○ 101 pennies
- ○ 4 quarters and 1 penny
- ○ 3 quarters, 2 dimes, 1 nickel, and 1 penny

2) A greedy algorithm is attempting to minimize costs and has a choice between two items with equivalent functionality: the first costing $5 and the second costing $7. Which will be chosen?

- ○ The $5 item
- ○ The $7 item
- ○ The algorithm needs more information to choose

3) A greedy algorithm always finds an optimal solution.

- ○ True
- ○ False

Fractional knapsack problem

The ***fractional knapsack problem*** is the knapsack problem with the potential to take each item a fractional number of times, provided the fraction is in the range [0.0, 1.0]. Ex: A 4 pound,

$10 item could be taken 0.5 times to fill a knapsack with a 2 pound weight limit. The resulting knapsack would be worth $5.

While a greedy solution to the 0-1 knapsack problem is not necessarily optimal, a greedy solution to the fractional knapsack problem is optimal. First, items are sorted in descending order based on the value-to-weight ratio. Next, one of each item is taken from the item list, in order, until taking 1 of the next item would exceed the weight limit. Then a fraction of the next item in the list is taken to fill the remaining weight.

Figure 10.2.1: FractionalKnapsack algorithm.

```
FractionalKnapsack(knapsack, itemList, itemListSize) {
   Sort itemList descending by item's (value / weight) ratio
   remaining = knapsack->maximumWeight
   for each item in itemList {
      if (item->weight <= remaining) {
         Put item in knapsack
         remaining = remaining - item->weight
      }
      else {
         fraction = remaining / item->weight
         Put (fraction * item) in knapsack
         break
      }
   }
}
```

PARTICIPATION ACTIVITY 10.2.3: Fractional knapsack problem.

Suppose the following items are available: 40 pounds worth $80, 12 pounds worth $18, and 8 pounds worth $8.

1) Which item has the highest value-to-weight ratio?

- 40 pounds worth $80
- 12 pounds worth $18
- 8 pounds worth $8

2) What would FractionalKnapsack put in a 20-pound knapsack?

- One 12-pound item and one 8-pound item
- One 40-pound item
- Half of a 40-pound item

3) What would FractionalKnapsack put in

a 48-pound knapsack?

- ○ One 40-pound item and one 8-pound item
- ○ One 40-pound item and 2/3 of a 12-pound item

Activity selection problem

The ***activity selection problem*** is a problem where 1 or more activities are available, each with a start and finish time, and the goal is to build the largest possible set of activities without time conflicts. Ex: When on vacation, various activities such as museum tours or mountain hikes may be available. Since vacation time is limited, the desire is often to engage in the maximum possible number of activities per day.

A greedy algorithm provides the optimal solution to the activity selection problem. First, an empty set of chosen activities is allocated. Activities are then sorted in ascending order by finish time. The first activity in the sorted list is marked as the current activity and added to the set of chosen activities. The algorithm then iterates through all activities after the first, looking for a next activity that starts after the current activity ends. When such a next activity is found, the next activity is added to the set of chosen activities, and the next activity is reassigned as the current. After iterating through all activities, the chosen set of activities contains the maximum possible number of non-conflicting activities from the activities list.

PARTICIPATION ACTIVITY 10.2.4: Activity selection problem algorithm.

Animation content:

`undefined`

Animation captions:

1. Activities are first sorted in ascending order by finish time. The set of chosen activities initiall has the activity that finishes first.
2. The morning mountain hike does not start after the history museum tour finishes and is not added to the chosen set of activities.
3. The boat tour is the first activity to start after the history museum tour finishes, and is the "greedy" choice.
4. Hang gliding and the fireworks show are chosen as 2 additional activities.
5. The maximum possible number of non-conflicting activities is 4, and 4 have been chosen.

PARTICIPATION ACTIVITY 10.2.5: ActivitySelection algorithm.

1) The fireworks show and the night movie both finish at 9 PM, so the sorting algorithm could have swapped the order of the 2. If the 2 were swapped, the number of chosen activities would not be affected.

   - ○ True
   - ○ False

2) Changing snorkeling's _____ would cause snorkeling to be added to the chosen activities.

   - ○ start time from 3 PM to 4 PM
   - ○ finish time from 5 PM to 4 PM

3) Regardless of any changes to the activity list, the activity with the _____ will always be in the result.

   - ○ earliest start time
   - ○ earliest finish time
   - ○ longest length

How was this section? **Provide feedback**

10.3 Dynamic programming

Dynamic programming overview

Dynamic programming is a problem solving technique that splits a problem into smaller subproblems, computes and stores solutions to subproblems in memory, and then uses the stored solutions to solve the larger problem. Ex: Fibonacci numbers can be computed with an iterative approach that stores the 2 previous terms, instead of making recursive calls that recompute the same term many times over.

PARTICIPATION ACTIVITY 10.3.1: FibonacciNumber algorithm: Recursion vs. dynamic programming.

Animation content:

`undefined`

Animation captions:

1. The recursive call hierarchy of FibonacciNumber(4) shows each call made to FibonacciNumber.
2. Several terms are computed more than once.
3. The iterative implementation uses dynamic programming and stores the previous 2 terms at time.
4. For each iteration, the next term is computed by adding the previous 2 terms. The previous and current terms are also updated for the next iteration.
5. 3 loop iterations are needed to compute FibonacciNumber(4). Because the previous 2 terms are stored, no term is computed more than once.

PARTICIPATION ACTIVITY 10.3.2: FibonacciNumber implementation.

1) If the recursive version of FibonacciNumber(3) is called, how many times will be FibonacciNumber(2) be called?

- ○ 1
- ○ 2
- ○ 3

2) Which version of FibonacciNumber is faster for large term indices?

- ○ Recursive version
- ○ Iterative version
- ○ Neither

3) Which version of FibonacciNumber is more accurate for large term indices?

- ○ Recursive version
- ○ Iterative version
- ○ Neither

4) The recursive version of FibonacciNumber has a runtime

complexity of O(1.62^N). What is the runtime complexity of the iterative version?

- ○ O(N)
- ○ O(N^2)
- ○ O(1.62^N)

PARTICIPATION ACTIVITY 10.3.3: Dynamic programming.

1) Dynamic programming avoids recomputing previously computed results by storing and reusing such results.

- ○ True
- ○ False

2) Any algorithm that splits a problem into smaller subproblems is using dynamic programming.

- ○ True
- ○ False

Longest common substring

The ***longest common substring*** algorithm takes 2 strings as input and determines the longest substring that exists in both strings. The algorithm uses dynamic programming. An N x M integer matrix keeps track of matching substrings, where N is the length of the first string and M the length of the second. Each row represents a character from the first string, and each column represents a character from the second string.

An entry at i, j in the matrix indicates the length of the longest common substring that ends at character i in the first string and character j in the second. An entry will be 0 only if the 2 characters the entry corresponds to are not the same.

The matrix is built one row at a time, from the top row to the bottom row. Each row's entries are filled from left to right. An entry is set to 0 if the two characters do not match. Otherwise, the entry at i, j is set to 1 plus the entry in i - 1, j - 1.

PARTICIPATION ACTIVITY 10.3.4: Longest common substring algorithm.

Animation content:

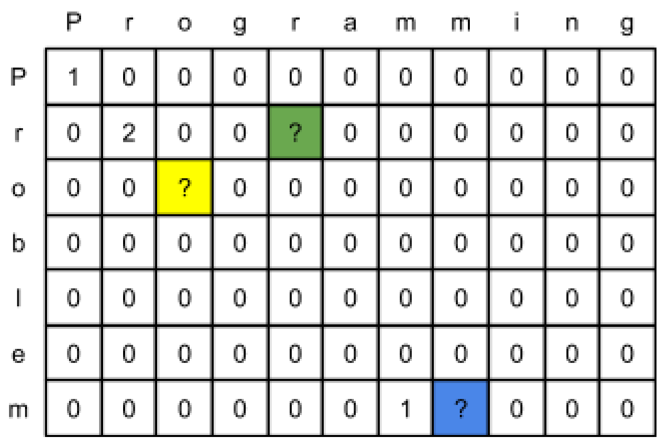
`undefined`

Animation captions:

1. Comparing "Look" and "zyBooks" requires a 7x4 matrix.
2. In the first row, 0 is entered for each pair of mismatching characters.
3. In the next row, 'o' matches in 2 entries. In both cases the upper-left value is 0, and 1 is entered into the matrix.
4. Two matches for 'o' exist in the next row as well, with the second having a 1 in the upper-left entry.
5. The character 'k' matches once in the last row and an entry of 2 + 1 = 3 is entered.
6. The maximum entry in the matrix is the longest common substring's length. The maximum entry's row index is the substring ending index in the first string.

PARTICIPATION ACTIVITY 10.3.5: Longest common substring matrix.

Consider the matrix below for the two strings "Programming" and "Problem".



	P	r	o	g	r	a	m	m	i	n	g
P	1	0	0	0	0	0	0	0	0	0	0
r	0	2	0	0	🟩 ?	0	0	0	0	0	0
o	0	0	🟨 ?	0	0	0	0	0	0	0	0
b	0	0	0	0	0	0	0	0	0	0	0
l	0	0	0	0	0	0	0	0	0	0	0
e	0	0	0	0	0	0	0	0	0	0	0
m	0	0	0	0	0	0	1	🟦 ?	0	0	0

1) What should be the value in the green cell?

- ○ 0
- ○ 1
- ○ 2

2) What should be the value in the yellow cell?

- ○ 1
- ○ 2
- ○

3

3) What should be the value in the blue cell?

- ○ 0
- ○ 1
- ○ 2

4) What is the longest common substring?

- ○ Pr
- ○ Pro
- ○ mm

Longest common substring algorithm complexity

The longest common substring algorithm operates on two strings of length N and M. For each of the N characters in the first string, M matrix entries are computed, making the runtime complexity O($N \cdot M$). Since an N x M integer matrix is built, the space complexity is also O($N \cdot M$).

Common substrings in DNA

A real-world application of the longest common substring algorithm is to find common substrings in DNA strings. Ex: Common substrings between 2 different DNA sequences may represent shared traits. DNA strings consist of characters C, T, A, and G.

PARTICIPATION ACTIVITY 10.3.6: Finding longest common substrings in DNA.

Animation content:

`undefined`

Animation captions:

1. Finding common substrings in DNA strings can be used for detecting things such as geneti disorders or for tracing evolutionary lineages.

2. DNA strings are very long, often billions of characters. Dynamic programming is crucial to obtaining a reasonable runtime.
3. Optimizations can lower memory usage by keeping only the following in memory: previous row data and largest matrix entry information.

PARTICIPATION ACTIVITY 10.3.7: Common substrings in DNA.

1) Which cannot be a character in a DNA string?

- ○ A
- ○ B
- ○ C

2) If an animal's DNA string is available, a genetic disorder might be found by finding the longest common substring from the DNA of another animal _____ the disorder.

- ○ with
- ○ without

3) When computing row X in the matrix, what information is needed, besides the 2 strings?

- ○ Row X - 1
- ○ Row X + 1
- ○ All rows before X

PARTICIPATION ACTIVITY 10.3.8: Longest common substrings - critical thinking.

1) If the largest entry in the matrix were the only known value, what could be determined?

- ○ The starting index of the longest common substring within either string
- ○ The character contents of the common substring

- ○ The length of the longest common substring

2) Suppose only the row and column indices for the largest entry in the matrix were known, and not the value of the largest or any other matrix entry. What can be determined in O(1)?

- ○ Only the longest common substring's ending index within either string
- ○ The longest common substring's starting and ending indices within either string

Optimized longest common substring algorithm complexity

The longest common substring algorithm can be implemented such that only the previously computed row and the largest matrix entry's location and value are stored in memory. With this optimization, the space complexity is reduced to O(N). The runtime complexity remains O($N \cdot M$).

How was this section? Provide feedback