HOMEWORK ASSIGNMENT
Problem 2

Draghici Bogdan
The Faculty of ACE, Craiova
Second Year, Calculatoare romana
CR 2.2.A

May 12, 2020

Contents

1 Problem statement

The Knuth Sequence - start with a number, apply a sequence of factorial, square root, and floor operations, and arrive at any desired positive integer.

a. Write a detailed formulation for this search problem.

b. Identify a suitable search algorithm for this task and explain your choice.

1.1 Formulation for the search problem

As the problem states, we are given 2 numbers and must find an equation composed only of factorial, square root and floor operations, that applied to the first number, will result the second one.
So we have to search for a path from the initial number, to the target number, where each node is represented by a series of operations applied on the initial number.
As we explore the possible numbers that can be created from the initial one, we must keep in mind that we also have to create new, unexplored numbers, from the one already created.

1.2 Suitable algorithm search for the problem

As the problem does **NOT** specify that the solution must be the shortest path between those numbers (as few operations applied to transform the first number into the second one), I chose to find instead the path with the smallest cost, given by an heuristic.
The algorithm I used to solve this problem is **A* Search**, one of the best algorithms used for searching, as it intelligently chooses the best path at every step. This property makes it also a greedy algorithm. **Note that** I could have easily implement a BFS without any heuristic, but it would have run slowly on the tests and it would have not been such big of a challenge.
A* Search combines the cost to reach a node (G) with the heuristic cost needed to achieve the desired node (H), and the sum of those 2 costs is noted F. In my case, the cost G is represented by the sum of minimum distances of all pairs in the path to reach a number, starting from the initial one. The cost H has a similar formulation, such that it is represented by the direct distance from any number to the target number. So my implementation will find the path with the smallest fluctuation of the numbers.

2 Pseudo-code description of the algorithm

2.1 Structures used

In order to solve the given problem, I used the following structures:

1. a map which stores the cost to reach from the starting number to any other number created: *costG*,
2. a set of tuples that stores the combined cost of a number F, the number itself, and the previous operation applied to reach it: *distF*,
3. an array of 2 elements of pairs that stores the newly created numbers and the operations used to reach those numbers: *newNum[2]*,
4. a map, that has as keys and values pairs of numbers and operations, and acts as an array of parents to find the predecessor of any pair (number, operation): *prevNum*,
5. a stack of pairs that stores the path of numbers and operations made to reach the targeted number: *sol*.

2.2 The main algorithm

Function AStar(input, output)

```
1.  read initial, target
2.  initialize costG[initial] = 0, boolean isOne = false
3.  prevNum[pair(initial,"Initial")] = pair(0,"0"),
4.  add tuple(absolute value(initial - target), initial,"Initial") to distF
5.  while distF not empty do
6.      tuple (combinedCostF,current,operation) = first tuple of distF
7.      remove first tuple of distF
8.      if current < 2 and isOne = true
9.          go to next tuple of distF
10.     else if current = 1
11.             isOne = true
12.     if current = target
13.         push pair(current, position) in sol
14.         while prevNum[pair(current,operation)] != pair(0,"0") do
15.             pair aux = prevNum[pair(current,operation)]
16.             push aux in sol
17.             current = first element of pair aux
```

```
18.                 operation = second element of pair aux
19.             break
20.         first element of pair newNum[0] = square root of current
21.         second element of pair newNum[0] = "Root"
22.         if current is not an integer
23.             first element of pair newNum[1] = floor of current
24.             second element of pair newNum[1] = "Floor"
25.             else if current <= 1750
26.                     first element of pair newNum[1] = fact(current)
27.                     second element of pair newNum[1] = "Factorial"
28.                 else
29.                     first element of pair newNum[1] = -1
30.         for i=0, i<2, execute
31.             if first of newNum[i] != -1 and first of newNum[i] not found
    in costG
32.                 costH = abs(target - first of newNum[i])
33.                 costG[first of newNum[i]] = costG[current] + abs(current
    - first of newNum[i])
34.                 insert tuple(costG[first of newNum[i]] + costH, first of
    newNum[i], second of newNum[i]) in distF
35.                 prevNum[newNum[i]] = pair(current, operation)
36. if sol not empty
37.         write "The path between 'initial' and 'target' is:"
38.         while sol not empty do
39.             write " 'second of top of sol' 'first of top of sol' "
40.             remove top of sol
41.         write "The path cost is 'costG[target]' "
42. else
43.         write "Can not find a path between 'initial' and 'target' "
```

First we read the initial number and target number. I also initialize the cost for the initial number to 0, and the previous pair from the initial number to be(0,"0"). Boolean variable *isOne* is used to see if the exact number 1 has already been processed. I also insert the initial number in the set *distF*, as this will be the first one to be processed.
On line 5 starts the repetitive structure that stores the whole algorithm used, and it will run till it finds the path, or the set *distF* is empty, meaning all possible numbers have been processed and no path was found. The algorithm extracts the first tuple from *distF* and remove it. Note that because it's a set, it will sort the tuple by the first parameter, *combinedCostF* in our case, which will always give us at it's first position the minimum cost to reach a

new number. The condition from line 8 is used to minimize the time of the algorithm, by jumping at the next tuple if the current number is smaller than 2 and the number 1 has already been processed, so we no longer need to search for a solution in interval [1,2).
If we have reached the targeted number, this will be signalled by the condition on line 12, and will begin the process of memorizing the path in the stack *sol*, using the map of pairs *prevNum* to backtrack the previous numbers till we reach the initial one (lines 13 - 18). The first solution has the best cost, because in order to process this number, it had to be the first in the set *distF* that is sorted ascending by the cost. So we exit the repetitive structure and go straight to line 36.
On lines 20-21 we create the square root of the number and memorize this operation's name in the pair *newNum[0]*. Next we can either create the factorial of the current number if it is an integer, or create the floor of the current non integer number, and the result will be stored in pair *newNum[1]*. Note that if the number is an integer, but it is greater than 1750, we can not create the factorial, because it will exceed the range used (approx. 10^{4932}, and 1750! is approx. 10^{4918}).
Next we see for both numbers created if they do not have a cost attributed. In affirmative case, we create the heuristic cost *costH* to reach the target as being the linear distance between the *newNum[i]* and *target*, *costG* will take the value of the cost to reach the previous number plus cost to reach the *newNum[i]* number from current number, which is also the linear distance between them.
We insert the newly created tuple for the path in the distF, which has as arguments the cost F = G + H, the newly created number, and the operation used to obtain to this number.
Outside the repetitive structure which represents the whole algorithm, we have a condition that can tell us if there is a path that contains numbers $< 10^{4932}$, or if it is required some larger numbers to reach the target. In case it found a path, we write it by sequentially extracting the top of the stack *sol*, and then also writing the cost for this path.

2.3 Data generation function

For the generation of the input data sets I used the rand() function from the library math.h, and also a function implemented by myself, that generates a random number from a given range.

Function random(min, max)

1. counter = max - min, number = 0
2. **while** counter != 0 **do**
3. number = number * RAND_MAX + rand()
4. counter = counter RAND_MAX
5. return number % (max - min + 1) + min

RAND_MAX is a constant which represents the biggest value that can be generated using the function rand(), and it's value is 32767.
On line 2 I initialized counter with the length of the range and the variable in which the number will be created with 0. On line 3 is a repetitive structure that will be executed for log(RAND_MAX) counter times, in order to obtain a number in the range. On line 4, the *number* is multiplied by RAND_MAX and also added a new random value generated by the rand() function. On line 5 the counter is divided by the maximum value that can be taken by a number generated in rand().
So, when the *number* is multiplied by RAND_MAX, the *counter* is divided by RAND_MAX, in order to get to the given range as quickly as possible. After the repetitive structure ends, the *number* created might exceed the range length, so we must do the rest of the division with the range length+1, then add the first value of the range in order to obtain a number between min and max.
For the input, I created a function called generator that generates the input for a given number of tests, using the function random previously explained.
Function generator(tests)

1. **for** i = 1, tests, **execute**
2. initial = random(1, max1)
3. target = random(1, max2)
4. **write in test i** initial, target

I used a repetitive structure in order to write in the specific file the input test, and for each test I generate the *initial* and the *target* numbers, which are written in the input files. I observed that for a big *target* number it will rarely find a path, so I limited it to the interval [1,200] (max2 = 200).

3 Application outline

3.1 The high level architectural overview of the application

The architecture is, in my opinion, quite complex for this algorithm. My implementation for A* Search algorithm contains a set of tuples as it's core, and uses many other structures. Although it does not guarantee to find the shortest path(correctness is approx. 40%) on account of the heuristic used, because it is a combination of greedy and Dijkstra, it always finds the path with the lowest cost for this heuristic, very fast, in less than 0.15s per test.

3.2 The specification of the input format

The input for this algorithm is represented by only 2 numbers, that are stored in the input files in this order: initial number which must be transformed on the first line, and the target number that we must reach on the second line.

3.3 The specification of the output format

In the output file is written on it's first line if the target number can be obtained by applying the operations to the initial one. In affirmative case, the next lines will each contain an operation and the number resulted by applying that operation, the last number being the target one. After the path has been written, the last line in the output file will contain the cost of the path.

3.4 The list of all the modules in the application and their description

The modules I have used are:

1. main.cpp, that controls the generation and execution of the algorithm for all tests

2. generator.h, generator.cpp, used for generating and storing the input data

3. AStar.h, AStar.cpp, contains the main algorithm and the function for creating the factorial of a number

3.5 The list of all functions in the application

1. in module generator we have the function **random** and the function **generator**, that have already been described

2. in module AStar is the **main algorithm** also described previously, and the function **fact** that gets as a parameter a long double variable, creates it's factorial, and returns that value

4 Results and Conclusion

4.1 Experiments and Results

As it can be seen in the figure, the algorithm is time efficient, as it finds a path(if there is any) between the 2 numbers in less than 0.15s for any test.

Initial	Target	Time(s)
284370690	204	0.102
192224298	89	0.013
276487692	77	0.009
276487692	215	0.016
984185000	177	0.000
16804900	29	0.007
789716000	56	0.007
674199964	95	0.035
336236696	132	0.036
411892469	23	0.002
123516938	88	0.024
3	5	0.000
23	1904	0.001
1,67962E+14	1099566	0.013

Figure 1: Execution Time

I also manually created, using the calculator, the inputs: 3 - 5, 23 - 1904 and 167961600000000 - 1099566, and the algorithm gave the exact same paths that I used.

```
Initial 3
Factorial 6
Factorial 720
Root 26.832815729997476356105678974018
Root 5.180040128222702905391727767892
Floor 5
```

Figure 2: Result 3 - 5

```
Initial 23
Factorial 25852016738884976640000
Root 160785623545.40587669610977172852
Root 400980.82690498541677470711874776
Floor 400980
Root 633.22981610154776410670507402756
Root 25.164058021343611663486750096297
Root 5.016378975051985808931293009393 9
Floor 5
Factorial 120
Root 10.954451150103322269041550285351
Floor 10
Factorial 3628800
Root 1904.940943966505225160013026197 7
Floor 1904
```

Figure 3: Result 23 - 1904

```
Initial 167961600000000
Root 12960000
Root 3600
Root 60
Factorial 8.32098711274139014353091756154 68e+081
Root 9.12194448171078825879311774608 02e+040
Root 302025569806776264928
Root 17378882869.930859422311186790466
Root 131828.99100702720399169720622012
Root 363.08262283814575857587847451668
Floor 363
Factorial 1.88951743757027941419927227326 54e+773
Root 4.34685798890449078608761131898 82e+386
Root 2.08491198588921033286118582503 29e+193
Root 4.56608364563025754151757665823 44e+096
Root 2.13683963966186698542626252825 1e+048
Root 1461793295805486647083008
Root 1209046440714.9489835500717163086
Root 1099566.47853367600896490330342 2
Floor 1099566
```

Figure 4: Result 167961600000000 - 1099566

4.2 Summary of achievements

By making this problem, I learned how to implement A* Search algorithm, how to use heuristics to solve problems, and how to extend as much as I can the range in which a number can fluctuate(current up to 10^{4930}) till it finds a path.

4.3 Brief description of the most challenging achievement

The most challenging achievement was to find a somehow good heuristic to use in my problem, in such a way that my solution will achieve a solution fast and accurate. (note that my scope was to use an heuristic to find very fast a path between the two numbers, that also has the smallest cost for the heuristic used).

4.4 Future directions for extending the lab homework

I hope that in the future I will be able to implement the A* algorithm with a better heuristic, in such a way that not only it will find the path very fast, but also with as few intermediary numbers as possible.

5 References

1. https://classroom.google.com/u/1/c/NjE5MTY5NDk3ODda/m/NjM4Njg4Njk3MDha/details
2. https://classroom.google.com/u/1/c/NjE5MTY5NDk3ODda/m/NTM1NjU5NTE3NDda/details
3. https://www.youtube.com/watch?v=ySN5Wnu88nE
4. https://www.geeksforgeeks.org/a-search-algorithm/
5. https://stackoverflow.com/questions/27501160/pass-a-reference-to-stdifstream-as-parameter
6. https://en.wikipedia.org/wiki/Extended_precision