Fundamentals of Programming 2

Backtracking Algorithms

Arkadiusz Chrobot

Department of Computer Science

June 16, 2019

Outline

1. Introduction
2. The Water Jug Problem
3. Analysis of the Problem
4. The Algorithm
   - The Single Solution Version
   - The Many Solutions Version
5. The Implementation
6. Summary

Introduction

Backtracking algorithms are applied for solving a class of problems so defined that the input data and the goal or at least its characteristics are given, but the way of achieving the goal is unknown. The examples of such issues are the chess problems such as the eight queens problem or the knight's tour.

There is no efficient, dedicated way of solving such problems. Often the only feasible solution is using the "try and error" strategy. Since it is a tedious task it is beneficial to apply a computer for it, by adjusting and implementing a backtracking algorithm for the particular problem.

The Water Jug Problem

The application of backtracking algorithm can be demonstrated with the use of a quite simple problem named “the water jug problem”:

Definition

The Water Jug Problem: Having a two jugs of capacities, respectively, four and three liters, measure exactly two liters of water. The problem is solved when any of the jugs contains the desired amount of water.

It is not necessary to use a computer for solving such a problem. Even using a pen and a peace of paper should be enough. Nevertheless, its simplicity is an advantage — it is easier to apply the backtracking algorithm for solving it.

Analysis of the Problem

Let's consider the problem closer. There are available two jugs and a source of unlimited water. The task is to measure two liters of water, by filling the jugs with this liquid, pouring their contents between them or empting them. Thus, the number of actions that can be done to the jugs is limited. Moreover, in one step only one action can be performed. A more detailed analysis shows that each of the actions results in leaving in the jugs a discrete amount of water, i.e. a one that can be expressed with the use of natural numbers. Hence, the amount of the water in both jugs, or the *state* of the jugs can be described with a pair of such numbers. Many such states can be found while trying to solve the problem. They form a *discrete solution space*. Some of the states are the one that are sought for — one of the jugs contains two liters of water, hence they satisfy the *goal condition*. If the definition of the problem stated that the desired state is the only one, in which, for example, the four liter jug contains two liters of water, then the state would be a single *goal* state.

Analysis of the Problem

The transition from one state to another is possible only by performing an *operation* (action) on the jugs, that changes the amount of water in them (the aforementioned filling, empting and pouring). However, all of the operations cannot be applied to each possible state, for example it is impossible to pour water from an empty jug. Thus a state has to fulfill a specific conditions that allows an operation to be performed on it. The next two slides contain a list of *operators*, which are a formal notation for describing all possible operations that can be performed on jugs. The "→" symbol means a transition from one state of jugs to another, cause by the operation. The `j3` and `j4` variables denote the amount of water in the three and four liter jug respectively. The expression on the right side of the "→" symbol defines the condition that has to be satisfied by the initial state, so that the operation described by the operator could be performed. The expression on the left side of the symbol describes the state of the jugs after the operation is done (the next state).

Analysis of the Problem

Operators

1: Fill the four liter jug

$(j4, j3 | j4 < 4) \rightarrow (4, j3)$

2: Fill the three liter jug

$(j4, j3 | j3 < 3) \rightarrow (j4, 3)$

3: Empty the four liter jug

$(j4, j3 | j4 > 0) \rightarrow (0, j3)$

4: Empty the three liter jug

$(j4, j3 | j3 > 0) \rightarrow (j4, 0)$

Analysis of the Problem

Operators

5: Pour the water from the three liter jug to the four liter jug, to fill the latter

$(j4, j3 | j4 + j3 \geq 4 \wedge j3 > 0) \rightarrow (4, j3 - (4 - j4))$

6: Pour the water from the four liter jug, to the three liter jug to fill the latter

$(j4, j3 | j4 + j3 \geq 3 \wedge j4 > 0) \rightarrow (j4 - (3 - j3), 3)$

7: Pour the water from the three liter jug to the four liter jug, to empty the former

$(j4, j3 | j4 + j3 \leq 4 \wedge j3 > 0) \rightarrow (j3 + j4, 0)$

8: Pour the water from the four liter jug to the three liter jug, to empty the former

$(j4, j3 | j4 + j3 \leq 3 \wedge j4 > 0) \rightarrow (0, j3 + j4)$

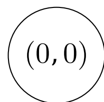
The Algorithm

As it has already been stated there is a space of states that describe the amount of the water in the jugs and the transitions between those states are possible with the use of operations defined by the operators. This space can be expressed as a directed graph where the vertices are the states of jugs and the edges are operations that make possible to leave one state and enter another. So, finding the solution of the water jug problem reduces to finding a path in the graph leading from the initial vertex, which describes the state where the jugs are empty to one of the *goal vertices* describing the state where one of the jugs contains exactly two liters of water. To find such a path the DFS algorithm can be applied. There is however one issue left — the structure of the graph is mostly unknown. Only the initial vertex and the set of operators that can form the edges of the graph are known. However, this information is all that is needed to generate the whole graph. On the other hand, building the whole graph is unnecessary.

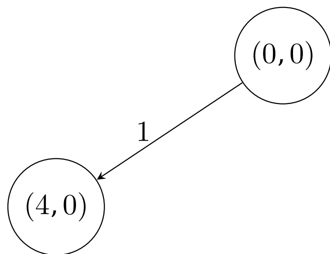
The Algorithm

It is enough to generate the vertices that belong to the currently explored path in the graph. If one of the new vertices repeats itself then it is necessary to *backtrack* to the previous vertex and try to create another vertex (a different state) that belongs to the path. Making new vertices should be stopped after a vertex that corresponds to one of the states that satisfy the goal condition. The resulting path is the solution of the water jug problem. This is the essence of backtracking algorithm. The next slide contains an animation that (partially) illustrates how the algorithm works.

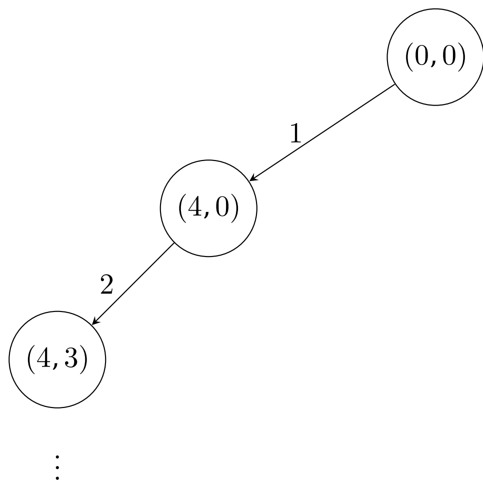
The Algorithm



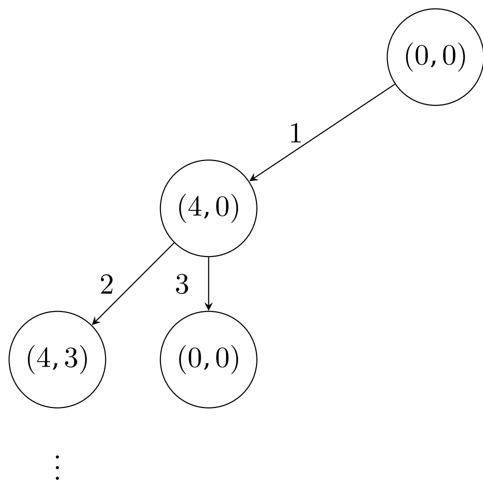
The Algorithm



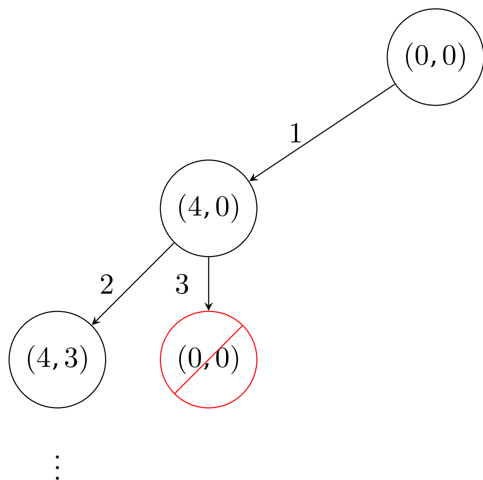
The Algorithm



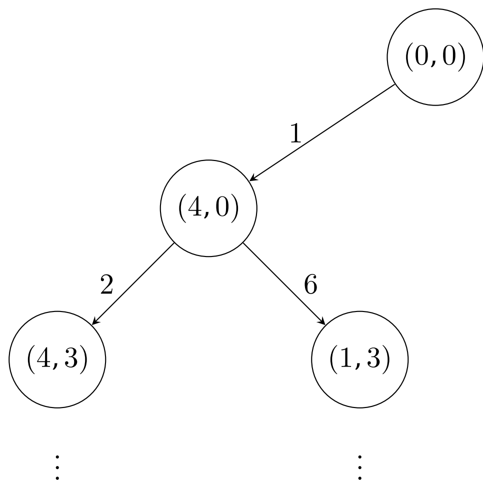
The Algorithm



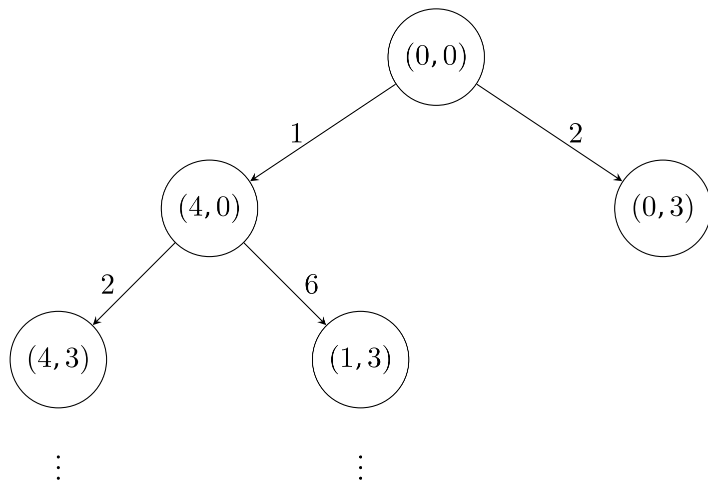
The Algorithm



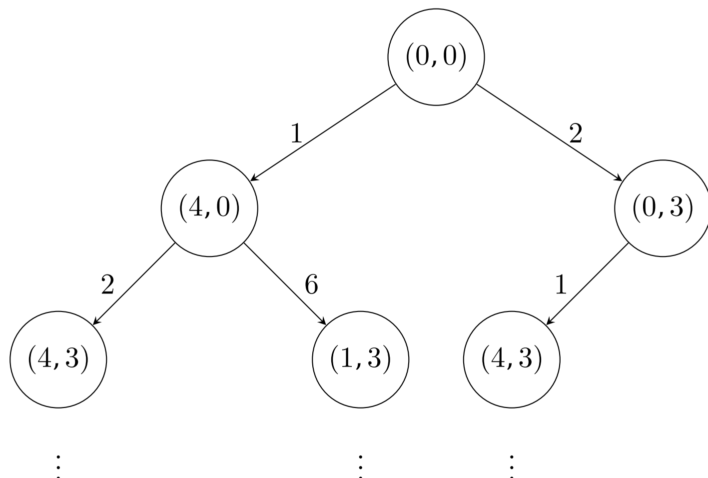
The Algorithm



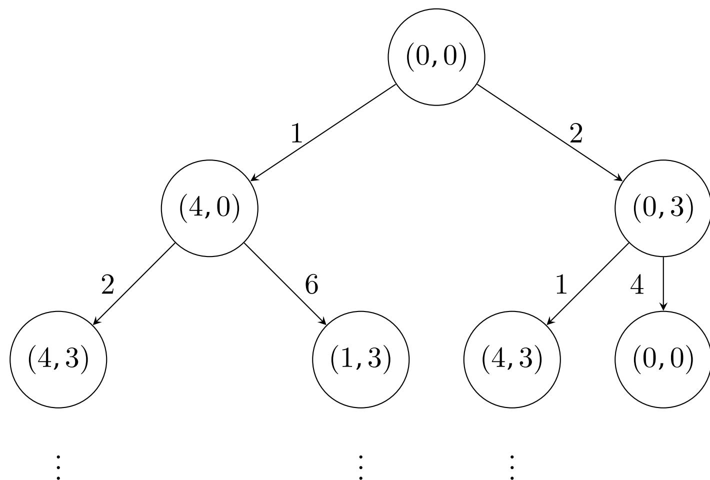
The Algorithm



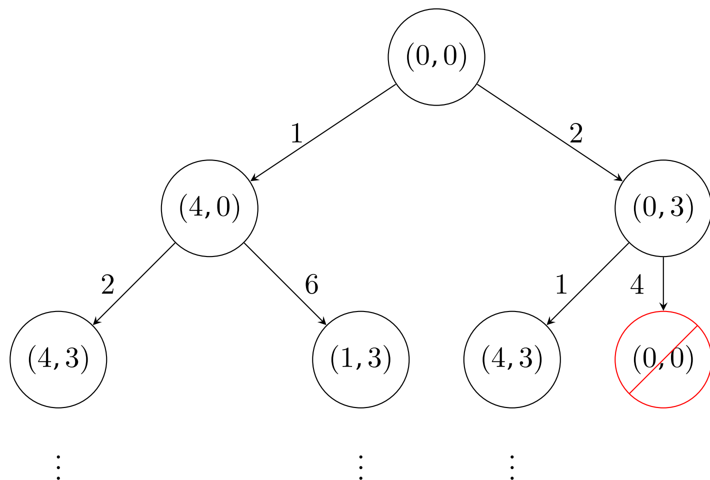
The Algorithm



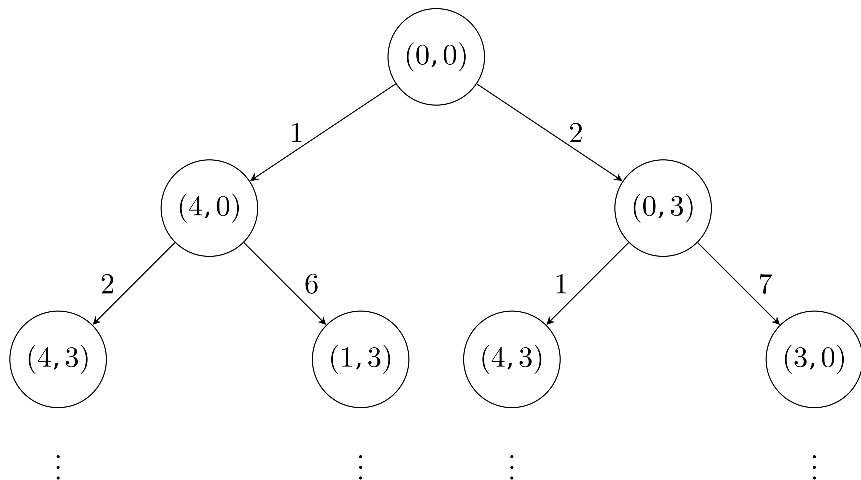
The Algorithm



The Algorithm



The Algorithm



The Algorithm

The Single Solution Version

To easier implement the described algorithm it is expressed in a pseudocode which is an intermediate notation between a natural language and the computer code. The version of the algorithm presented on the next slide finds only one solution of the water jug problem.

The Algorithm

The Single Solution Version

The Pseudocode for the Single Solution Version

```
find_solution(path)
{
    for each(operator){
        if(can_apply(operator, last_state(path))){
            new_state = operator(last_state(path));
            if(has_not_been(path, new_state)){
                add(path, new_state);
                if(goal_condition(new_state))
                    print(path);
                else
                    find_solution(path);
            } else
                remove(new_state);
        }
    }
}
```

The Algorithm

The Many Solutions Version

From the water jug problem definition it can be deduced that the problem has more than one solution. Thus, the question arises how to modify the algorithm from the previous slide, so that it can find all possible solutions to the problem. It occurs that to this end the concept of "backtracking" has to be expanded. The algorithm should go back to the previously generated vertex not only when the new one repeats itself in the path, but always when the exploration of one of the path associated with the previous vertex is finished. In that case the algorithm can examine all the possible paths associated with each of the vertices, which allows it to find all possible solutions to the problem. It means that the pseudocode from the previous slide has to be supplemented with one additional statement that removes the last vertex from the path when the algorithm returns form a recursive call or prints the solution (the path). The modification is shown in the pseudocode presented in the next slide.

The Algorithm

The Many Solutions Version

The Pseudocode for The Many Solutions Version

```
find_solution(path)
{
    for each(operator){
        if(can_apply(operator, last_state(path))){
            new_state = operator(last_state(path));
            if(has_not_been(path, new_state)){
                add(path, new_state);
                if(goal_condition(new_state))
                    print(path);
                else
                    find_solution(path);
                remove_last_state(path);
            } else
                remove(new_state);
        }
    }
}
```

The Backtracking Algorithm — Implementation

In the next slides is presented a source code of a program that implements the backtracking algorithm that finds all solutions to the water jug problem. It is quite complex program, but each part of it is commented to make it easier to understand.

The Backtracking Algorithm — Implementation

Inclusion of the Header Files and Definitions

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <stdbool.h>
4
5   #define NUMBER_OF_OPERATORS 8
6
7   enum operator_index {NONE = -1, FILL_4_LITTERS, FILL_3_LITERS,
8           EMPTY_4_LITTERS, EMPTY_3_LITERS, POUR_3_FILL_4,
9           POUR_4_FILL_3, POUR_AND_EMPTY_4, POUR_AND_EMPTY_3};
10
11  struct jugs_states {
12      unsigned int jug_3_liters_state, jug_4_liters_state;
13  };
```

The Backtracking Algorithm — Implementation

Inclusion of the Header Files and Definitions

The lines no. 1–3 of the code from the previous slide contain statements that include the header files to the program. In the 5th line is defined a constant that defines the number of the operators used for solving the water jug problem. Since the operators are stored in an array, the constant also defines the number of elements of that array. The lines no. 7–9 contain a definition of an enumerated type. A variable of this type is used as an index for the array of operators used for creating the vertices of the graph. The operators are numerated starting from zero, but the first element of the enumerated type has a value `-1`. It is used for marking that no operator has been used for creating a vertex, and the vertex in question is the one that describes the initial state. The lines no. 11–12 contain definition of a structure type, which fields are used for storing the information about the amount of water in both of the jugs.

The Backtracking Algorithm — Implementation

Definitions of Data Types

```
1  struct queue_node {
2      struct jugs_states states;
3      enum operator_index operator_number;
4      struct queue_node *next;
5  };
6
7  struct queue_pointers {
8      struct queue_node *head, *tail;
9  } queue;
10
11 typedef bool (*operator_condtion_function_pointer)
12                                         (struct jugs_states state);
13
14 typedef struct jugs_states(*operator_function_pointer)
15                                               (struct jugs_states state);
```

The Backtracking Algorithm — Implementation

Definitions of Data Types

The lines no. 1–5 from the previous slide contain a definition of the base type of a queue used for storing the currently explored path of the graph. It is an input-restricted double-ended queue. The `state` field in the definition of the type is used for storing information about the state of jugs, the `operator_number` field stores the number of the operator used for creating the state. The `next` field is a pointer to a next element of the queue. A pointers structure for this queue is defined in lines no. 7–9. A function pointer type is defined in the lines 11–12. A pointer of this type can point a function that takes as an argument a structure that describes a state and return a value of the `bool` type, which informs if an operator can be applied to a specified state. In other words this function is responsible for evaluating the condition on the left side of the "$\rightarrow$" symbol in the formal definition of the operator.

The Backtracking Algorithm — Implementation

Definitions of Data Types

The lines no. 14–15 contain a definition of another function pointer type. This time the pointer can point a function that generates a new states basing on its argument, which is a previous state. It means that the function implements an operator. Its behaviours corresponds to the part of the formal definition of the operator that is located on the right side of the "$\rightarrow$" symbol.
The next five slides contains definitions of such aforementioned functions. Please observe, that their code corresponds to the formal definitions of operators presented in the earlier slides.

The Backtracking Algorithm — Implementation

The Operators Functions

```
1   bool can_fill_4_liters_jug(struct jugs_states state)
2   {
3       return state.jug_4_liters_state<4;
4   }
5
6   struct jugs_states fill_4_liters_jug(struct jugs_states state)
7   {
8       state.jug_4_liters_state = 4;
9       return state;
10  }
11
12  bool can_fill_3_liters_jug(struct jugs_states state)
13  {
14      return state.jug_3_liters_state<3;
15  }
```

The Backtracking Algorithm — Implementation

The Operators Functions

```
1  struct jugs_states fill_3_liters_jug(struct jugs_states state)
2  {
3      state.jug_3_liters_state = 3;
4      return state;
5  }
6
7  bool can_empty_4_liters_jug(struct jugs_states state)
8  {
9      return state.jug_4_liters_state>0;
10 }
11
12 struct jugs_states empty_4_liters_jug(struct jugs_states state)
13 {
14     state.jug_4_liters_state = 0;
15     return state;
16 }
```

The Backtracking Algorithm — Implementation

The Operators Functions

```
1  bool can_empty_3_liters_jug(struct jugs_states state)
2  {
3      return state.jug_3_liters_state>0;
4  }
5
6  struct jugs_states empty_3_liters_jug(struct jugs_states state)
7  {
8      state.jug_3_liters_state = 0;
9      return state;
10 }
11
12 bool can_fill_up_4_liters_jug_with_3_liters
13                                         (struct jugs_states state)
14 {
15     return state.jug_4_liters_state + state.jug_3_liters_state
16                                 >= 4 && state.jug_3_liters_state>0;
17 }
```

The Backtracking Algorithm — Implementation

The Operators Functions

```
1  struct jugs_states empty_3_liters_jug_to_4_liters
2                                                  (struct jugs_states state)
3  {
4      state.jug_4_liters_state = state.jug_3_liters_state +
5                                                      state.jug_4_liters_state;
6      state.jug_3_liters_state = 0;
7      return state;
8  }
9
10 bool can_empty_4_liters_jug_to_3_liters(struct jugs_states state)
11 {
12     return state.jug_3_liters_state +
13         state.jug_4_liters_state <= 3 &&
14         state.jug_4_liters_state > 0;
15 }
```

The Backtracking Algorithm — Implementation

The Operators Functions

```c
struct jugs_states empty_4_liters_jug_to_3_liters
                                        (struct jugs_states state)
{
    state.jug_3_liters_state = state.jug_3_liters_state +
                                        state.jug_4_liters_state;
    state.jug_4_liters_state = 0;
    return state;
}
```

The Backtracking Algorithm — Implementation

The Operators Array

```
1  struct operator_structure {
2      operator_condtion_function_pointer is_condition_fullfiled;
3      operator_function_pointer get_next_state;
4  } operators[NUMBER_OF_OPERATORS] = {
5      [FILL_4_LITTERS] = {
6          .is_condition_fullfiled = can_fill_4_liters_jug,
7          .get_next_state = fill_4_liters_jug
8      },
9      [FILL_3_LITERS] = {
10         .is_condition_fullfiled = can_fill_3_liters_jug,
11         .get_next_state = fill_3_liters_jug
12     },
13     [EMPTY_4_LITTERS] = {
14         .is_condition_fullfiled = can_empty_4_liters_jug,
15         .get_next_state = empty_4_liters_jug
16     },
```

The Backtracking Algorithm — Implementation

The Operators Array

```
1        [EMPTY_3_LITERS] = {
2            .is_condition_fullfiled = can_empty_3_liters_jug,
3            .get_next_state = empty_3_liters_jug
4        },
5        [POUR_3_FILL_4] = {
6            .is_condition_fullfiled = can_fill_up_4_liters_jug_with_3_liters,
7            .get_next_state = fill_up_4_liters_jug_with_3_liters
8        },
9        [POUR_4_FILL_3] = {
10           .is_condition_fullfiled = can_fill_up_3_liters_jug_with_4_liters,
11           .get_next_state = fill_up_3_liters_jug_with_4_liters
12       },
13       [POUR_AND_EMPTY_4] = {
14           .is_condition_fullfiled = can_empty_4_liters_jug_to_3_liters,
15           .get_next_state = empty_4_liters_jug_to_3_liters
16       },
```

The Backtracking Algorithm — Implementation

The Operators Array

```
1         [POUR_AND_EMPTY_3] = {
2             .is_condition_fullfiled =
3                                     can_empty_3_liters_jug_to_4_liters,
4             .get_next_state = empty_3_liters_jug_to_4_liters
5         }
6     };
```

The Backtracking Algorithm — Implementation

The Operators Array

The three previous slides contain declaration and initialisation of the operators array. It is an array of structures of pointers to a function. A concept from the object oriented programming is applied here — the pointed functions can be regarded as a methods of an object. The type of the elements of the array is defined in the lines no. 1–4 in the first of the described slides. It defines a structure which which both fields are pointers to a function. The type of the pointer has been defined earlier in the program. The array is declared in the same slide in the 4th line. The rest of the lines, also in the other slides, initialises elements of the array. To this end a *designated initialiser* is applied, which allows for assigning a value to a given element of the array, by placing its index in brackets and using an assignment operator. For example the 6th element of an array of ten elements of the `int` type can be initialised using the designated initialiser as follows:

```
int array[10] = {[5]=7};
```

The Backtracking Algorithm — Implementation

The Operators Array

The rest of the elements of the example array get the value of `0`. In the initialisation of elements of the operators array the elements of the `operator_index` enumerated type are used. Since its an array of pointers structures, each field of each element has assigned an address of a function associated with a specified operator.

The Backtracking Algorithm — Implementation

The `enqueue()` and `dequeue()` Functions

```
1   void enqueue(struct queue_pointers *queue,
2                                            struct queue_node *new_node)
3   {
4       queue->tail->next = new_node;
5       queue->tail = new_node;
6   }
7
8   void dequeue(struct queue_pointers *queue)
9   {
10      if(queue->head) {
11          struct queue_node *tmp = queue->head->next;
12          free(queue->head);
13          queue->head=tmp;
14          if(tmp==NULL)
15              queue->tail = NULL;
16      }
17  }
```

The Backtracking Algorithm — Implementation

The `enqueue()` and `dequeue()` Functions

The previous slide contains definitions of functions that, respectively, add and remove a new element from a queue where the currently explored path in the graph is stored. The `enqueue()` function, defined in the lines no. 1–6, doesn't return any value, but takes two arguments: an address of the queue pointers structure and an address of the new element that it adds at the tail of the queue. The function assumes that the queue is not empty. i.e. it has at list one element. The function in the 4th line assigns the address of the new element in the `next` field of the last element of the queue, then it assigns the same address to the pointer to the last element of the queue (5th line) and terminates. The `dequeue()` function removes an element at the head of the queue and it is defined in the same way as in the program from the previous lecture that demonstrates the DFS algorithm.

The Backtracking Algorithm — Implementation

The `remove_queue()` Function

```
1  void remove_queue(struct queue_pointers *queue)
2  {
3      while(queue->head)
4          dequeue(queue);
5  }
```

The Backtracking Algorithm — Implementation

The `remove_queue()` Function

The `remove_queue()` function removes the whole queue from the computer memory and it is defined in the same way as in the previous lecture.

The Backtracking Algorithm — Implementation

The `already_been()` Function

```
1   bool already_been(struct queue_pointers queue,
2                                         struct queue_node *new_node)
3   {
4       while(queue.head) {
5           if(queue.head->states.jug_4_liters_state ==
6                   new_node->states.jug_4_liters_state &&
7                   queue.head->states.jug_3_liters_state ==
8                            new_node->states.jug_3_liters_state)
9               return true;
10          queue.head = queue.head->next;
11      }
12      return false;
13  }
```

The Backtracking Algorithm — Implementation

The `already_been()` Function

The `already_been()` function checks if the new element describes a state (a vertex of the graph) that has already been found. It returns a value of the `bool` type. If it is `true` then it means that the state described by the new element has been already discovered, if `false` then it is an original state. To this end the function usses the `while` loop to iterate the queue and compare the state stored in elements of the queue with the state stored in the new element (lines no. 5–8). If one of the former elements stores the same state as the new one, the function returns `true` and terminates (line no. 9). If none of the elements stores the same state as the new one, then the `while` loop terminates after checking all the elements of the queue and the function returns `false` (12th line).

The Backtracking Algorithm — Implementation

The `remove_tail()` Function

```
1  void remove_tail(struct queue_pointers *queue)
2  {
3      if(queue->head) {
4          if(queue->head == queue->tail) {
5              free(queue->head);
6              queue->head = queue->tail = NULL;
7              return;
8          }
9          struct queue_node *node = queue->head;
10         while(node->next!=queue->tail)
11             node = node->next;
12         free(queue->tail);
13         node->next = NULL;
14         queue->tail = node;
15     }
16 }
```

The Backtracking Algorithm — Implementation

The `remove_tail()` Function

The `remove_tail()` function removes the last element from the queue. It is necessary in order to allow the backtracking algorithm to find a new solutions of the water jug problem. The `remove_tail()` function takes the address of the queue pointers structure as its only argument and doesn't return any value. In the 4th line it checks by comparing the values of the `head` and `tail` pointers, if the queue has only a single element. If so, it frees the memory allocated to that element (5th line), assignes the `NULL` value to the queue pointers (6th line) and terminates (7th line). If the queue has more than one element then the function searches in the `while` loop for the last but one element (lines no. 10–11). The element stores in the `next` field an address of the last element of the queue. After the former element is found the function removes the last one (12th line), assignes the `NULL` value to the `next` field of the new last element of the queue (13th line) and assigns an address of the element in the `tail` pointer of the queue (14th line).

The Backtracking Algorithm — Implementation

The print_solution() Function

```
1  void print_solution(struct queue_pointers queue)
2  {
3      static unsigned char solution_number;
4      unsigned char step = 1;
5      char * operators_description[NUMBER_OF_OPERATORS] = {
6          "Fill the 4 liter jug.",
7          "Fill the 3 liter jug.",
8          "Empty the 4 liter jug.",
9          "Empty the 3 liter jug.",
10         "Pour the water from the 3 liter jug to the 4 liter jug,\
11  to fill the latter.",
12         "Pour the water from the 4 liter jug to the 3 liter jug,\
13  to fill the latter.",
14         "Pour the water from the 3 liter jug to the 4 liter jug,\
15  to empty the former.",
16         "Pour the water from the 4 liter jug to the 3 liter jug,\
17  to empty the former."
18     };
```

The Backtracking Algorithm — Implementation

The print_solution() Function

```
1         printf("Solution no. %hhu:\n",++solution_number);
2         while(queue.head) {
3             enum operator_index operator =
4                     queue.head->operator_number;
5             if(operator!=NONE) {
6                 printf("Step number %hhu:\n",step++);
7                 printf("%s\n",operators_description[operator]);
8             } else
9                 puts("Initial state:");
10            printf("Water level in the 4 liter jug: %u and 3 liter\
11            jug: %u\n",
12                    queue.head->states.jug_4_liters_state,
13                    queue.head->states.jug_3_liters_state);
14            getchar();
15            queue.head = queue.head->next;
16        }
17        puts("THE END");
18    }
```

The Backtracking Algorithm — Implementation

The `print_solution()` Function

The two previous slides contain definition of the `print_solution()` function, which displays the details of a single solution to the water jug problem found by the program. This task mainly consists of interpreting the data from the path stored in the queue. The function takes the structure of queue pointers as an argument and returns no value. A static local variable for numerating the solutions printed by the function is declared in the 3rd line. All static variables are initialised by default with the `0` value and are not destroyed between subsequent calls of a function. Another variable is declared in the 4th line. This one is used for numerating the steps of a single solution (subsequent actions) and it is an ordinary local variable, initiated with the value `1`. The lines 5–18 contain declaration and initialisation of an array of strings that describe in English the actions defined by the operators.

The Backtracking Algorithm — Implementation

The `print_solution()` Function

In the 1st line of the second slide the described function prints a message informing which discovered solution it will display on the screen. The number of the solution is calculated by applying the pre-increment operator to the `solution_number` variable. Next in the `while` loop the function iterates the elements of the queue and for each of them assigns the operator number stored in the element to the `operator` variable and then compares it with the value of the `NONE` element of the `operator_index` enumerated type. If they are not equal then the function prints the value of the `step` variable incremented by one (6th line) and the description of the operator. Otherwise it displays a message informing that it is printing information about the initial state. The next actions performed by the function are common for the initial state and the rest of the states.

The Backtracking Algorithm — Implementation

The `print_solution()` Function

The state of the water in jugs that is stored by the queue element currently visited by the loop is displayed on the screen in the lines no. 10–13 of the second slide with the source code of the `print_solution()` function. Next, the function stops until the user presses any key on the keyboard (14th line). After that the loop visits the next element of the queue (15 th line). When the loop terminates, the function displays a message that informs the user that it has finished printing a single found solution to the water jug problem and terminates.

The Backtracking Algorithm — Implementation

The create_new_state() Function

```
1  struct queue_node *create_new_state(struct jugs_states state,
2                                      enum operator_index operator_number)
3  {
4      struct queue_node *new_node = (struct queue_node *)
5                                      malloc(sizeof(struct queue_node));
6      if(new_node) {
7          new_node->states = state;
8          new_node->operator_number = operator_number;
9          new_node->next = NULL;
10     }
11     return new_node;
12 }
```

The Backtracking Algorithm — Implementation

The `create_new_state()` Function

The `create_new_state()` function creates a new element of the queue that describes a newly generated state of the water in jugs. It takes as arguments the structure that describes the new state and the number of the operator that has been used for creating the state. The function returns address of the element of the queue that stores both of that data. It allocates memory for the new element of queue in the lines no. 4–5. If the allocation is successful then the condition in the conditional statement in the 6th line is satisfied and the function commences the initialisation of the new element's fields. It assigns the state of the water in jugs to a field of the element in the 7th line. The number of the operator is assigned to another field in the 8th line. Finally, the `next` field of the element is assigned the `NULL` value in the 9th line. The function returns in the 11th line the address of the new element of the queue or the `NULL` value, if it has been unable to create such an element, and terminates.

The Backtracking Algorithm — Implementation

The `initialize_queue()` Function

```
1  void initialize_queue(struct queue_pointers *queue)
2  {
3      struct queue_node *first_state = (struct queue_node *)
4                                   malloc(sizeof(struct queue_node));
5      if(first_state) {
6          first_state->states.jug_4_liters_state =
7                          first_state->states.jug_3_liters_state = 0;
8          first_state->operator_number = NONE;
9          first_state->next = NULL;
10         queue->head = queue->tail = first_state;
11     }
12 }
```

The Backtracking Algorithm — Implementation

The `initialize_queue()` Function

The `initialize_queue()` function initialises the queue by adding to it a first element which describes the initial vertex of the path that in turn describes a state in which both the jugs are empty. Since the function is invoked before the `enqueue()` function, the latter can skip testing if the queue has at least a single element. The described function takes as an argument the address of the queue pointers structure and returns no value. In the lines no. 3–4 it allocates memory for the first element of the queue. If the allocation is successful, then the condition in the conditional statement from the 5th line is satisfied. In this case the function initialises the `state` field of this element, so that it describes a sate in which both jugs contain `0` liters of water (lines no. 6–7), assigns the value of the `NONE` element of the `operator_index` enumerated type to the `operator_number` field and assigns the `NULL` value to the `next` field of the element (9th line). Finally, the function assigns an address of the first element to the queue pointers (10th line).

The Backtracking Algorithm — Implementation

The search() Function

```
1     void search(struct queue_pointers *queue, const struct operator_structure operators[])
2     {
3         enum operator_index operator_index;
4         for(operator_index=FILL_4_LITTERS; operator_index<=POUR_AND_EMPTY_3; operator_index++) {
5             if(queue->tail) {
6                 if(operators[operator_index].is_condition_fullfiled(queue->tail->states)) {
7                     struct queue_node *new_state =
8                     create_new_state(operators[operator_index].get_next_state(queue->tail->states),
9                                                                                    operator_index);
10                    if(new_state) {
11                        if(!already_been(*queue,new_state)) {
12                            enqueue(queue,new_state);
13                            if(queue->tail->states.jug_4_liters_state == 2 ||
14                                            queue->tail->states.jug_3_liters_state == 2)
15                                print_solution(*queue);
16                            else
17                                search(queue,operators);
18                            remove_tail(queue);
19                        } else
20                            free(new_state);
21                    }
22                }
23            }
24        }
25    }
```

The Backtracking Algorithm — Implementation

The `search()` Function

The definition of the `search()` function is based upon the pseudocode that describes the backtracking algorithm that finds all solutions of the water jug problem and that has been presented in the beginning of the lecture. The function returns no value and takes as arguments the address of the queue pointers structure and the operators array. Please observe, that the latter argument is passed by a constant parameter. In the 3rd line of the function a local variable is declared that serves as the `for` loop counter. The loop iterates over the operators array. In each iteration it first checks if there is a last element in the queue (5th line). If so, then it tests if the operator specified by the `operator_index` variable can be applied to the state described by that element in order to create a new state. If so, then the function calls the `create_new_state()` function to create a new element of the queue, which describes this new state (lines no. 7–9).

The Backtracking Algorithm — Implementation

The `search()` Function

If the creation of the new element is successful, what is checked in the 10th line, then the function tests if the state stored in the element hasn't been added to the queue earlier. To this end it calls the `already_been()` function and negates the returned result (11th line). If it turns up that that the state has been already created then the function removes the new element (20th line) and begins a new iteration of the `for` loop. However, if the state has not been created earlier then the function adds the new element to the queue (12th line) and checks if it describes a state that satisfies the goal condition (lines no. 13–14). If so, then the function invokes the `print_solution()` function, to display the information about the found solution. Otherwise the function calls itself recursively (17th line), to check which of the operators can be applied to the state described by the newly added to the queue element and what new states can be created that way.

The Backtracking Algorithm — Implementation

The `search()` Function

Regardless if the function has called itself recursively or it has displayed the details of a found solution, the last element of the queue is removed (18th line), so a new iteration of the `for` loop can check if another operators can be applied to the previous element of the queue and if other solutions of the water jug problem can be found that way.

The Backtracking Algorithm — Implementation

The main() Function

```
1  int main(void)
2  {
3      initialize_queue(&queue);
4      search(&queue,operators);
5      remove_queue(&queue);
6      return 0;
7  }
```

The Backtracking Algorithm — Implementation

The `main()` Function

Only three of the earlier defined functions have to be invoked in the `main()` function, for the program to display information about all possible solutions to the water jug problem. As the first one is called in the 3rd line the `initialize_queue()` function to create the queue by adding to it an element that describes the initial state of the water jugs. The `search()` function, that finds all possible solutions to the problem and prints their details on the screen is invoked in the 4th line. Finally, in the 5th line the `remove_queue()` function is called to destroy the queue, in case it would have some elements left after the `search()` function finished its task.

Summary

The presented program finds 10 solutions of the water jug problem. Some of them are suboptimal, i.e. force to perform some unnecessary steps. Indeed the backtracking algorithm finds all possible solutions of the problem without evaluating their quality. It means it uses a *brute force* approach. To make it chose only the optimal solutions, the heuristic functions should be applied that would evaluate the legitimacy of each step. Initially, the backtracking algorithms were applied only to the Artificial Intelligence problems, but nowadays they are used for solving many other issues outside this discipline, like: finding the extrema of multi-variable functions or constructing parts of compilers called *parsers*.

Questions

?

The End

Thank You For Your Attention!