Podstawy Programowania 2

Algorytmy z nawrotami

Arkadiusz Chrobot

Zakład Informatyki

9 czerwca 2019

Plan

Wstęp

Algorytmy z nawrotami, nazywane także algorytmami z powrotami (ang. *backtracking algorithm*) służą do rozwiązywania problemów, które w swojej definicji zawierają dane wejściowe oraz charakterystykę poszukiwanego rezultatu, a poszukiwaną informacją jest ciąg czynności, które należy wykonać, aby ze wspomnianych danych uzyskać oczekiwany wynik. Przykładami takich problemów są znane zadania szachowe, takie jak problem ośmiu hetmanów lub problem skoczka szachowego.
Dla tego typu problemów nie istnieją dedykowane, efektywne sposoby rozwiązywania, pozostaje jedynie zastosowanie metody „prób i błędów". Ponieważ takie działanie jest żmudne, to warto wykorzystać do jego wykonania komputer, czyli dostosować i zaimplementować algorytm z nawrotami do rozwiązywanego problemu.

Zastosowanie algorytmów z nawrotami zostanie przedstawione na przykładzie bardzo prostego problemu, o nazwie „problem dzbanów z wodą” (ang. *water jug problem*):

Definicja

Problem dzbanów z wodą: Mając dwa dzbany wody, o pojemności, odpowiednio, cztery i trzy litry, należy odmierzyć dwa litry wody. Zakładamy, że rozwiązanie jest znalezione, jeśli w **dowolnym** dzbanie jest żądana ilość wody.

Jego rozwiązanie nie wymaga użycia komputera, można je uzyskać nawet bez pomocy kartki papieru i długopisu. Niemniej jednak prostota tego problemu jest zaletą - łatwiej będzie zastosować do jego rozwiązania algorytm z nawrotami.

Analiza problemu

Zastanówmy się nad istotą tego problemu. Dane są dwa dzbany. Napełniając je wodą i przelewając ją między dzbanami mamy odmierzyć dwa litry wody. Każdy z dzbanów można wypełnić wodą, opróżnić z wody lub przelać wodę z jednego dzbana do drugiego, a więc liczba czynności, które można w tym problemie wykonać jest ograniczona. Dodatkowo, po zastanowieniu się, dojedziemy do wniosku, że ilość wody w dzbanie jest wielkością dyskretną, tzn. można ją wyrazić liczbami naturalnymi. Ilość wody w obu dzbanach, czyli *stan* dzbanów można zatem opisać parą liczb naturalnych. Rozwiązując problem dzbanów z wodą można znaleźć wiele takich stanów, z zatem tworzą one *dyskretną przestrzeń rozwiązań*. Część z tych stanów odpowiada pożądanej sytuacji - jeden z dzbanów zawiera dwa litry wody, a zatem spełniają one *warunek celu* (ang. *goal condition*). Gdybyśmy w definicji problemu określili, że interesuje nas jedynie stan, w którym np. dzban czterolitrowy zawiera dwa litry wody, a dzban trzylitrowy jest pusty, to byłby to stan docelowy (ang. *goal*).

Analiza problemu

Przejścia między stanami są możliwe tylko poprzez wykonanie *operacji* na dzbanach, które zmieniają ilość wody w nich zawartych (wspomniane napełnianie, opróżnianie i przelewanie). Czynności tych nie da się jednak wykonać dla każdego możliwego stanu, np. nie można opróżnić pustego dzbana. Zatem stan musi spełniać określone warunki, aby można było na nim wykonać daną operację. Dwa kolejne slajdy zawierają listę *operatorów*, które są formalnym zapisem wszystkich operacji jakie można wykonać w opisywanym problemie na dzbanach. Symbol „→” oznacza przejście z jednego stanu dzbanów do drugiego, jakie odpowiada wykonywanej czynności. Zmienne `j3` i `j4` oznaczają odpowiednio stan wody w trzylitrowym i czterolitrowym dzbanie. Zapis po prawej stronie symbolu „→” definiuje warunek, jaki musi spełnić stan początkowy, aby operator mógł być zastosowany. Zapis po lewej stronie tego symbolu określa stan uzyskany po wykonaniu czynności opisywanej przez operator.

Analiza problemu

Operatory

1: Napełnij dzban czterolitrowy

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

2: Napełnij dzban trzylitrowy

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

3: Opróżnij dzban czterolitrowy

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

4: Opróżnij dzban trzylitrowy

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

Analiza problemu

Operatory

5: Przelej wodę z dzbana trzylitrowego do czterolitrowego, wypełniając ten ostatni całkowicie

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

6: Przelej wodę z dzbana czterolitrowego do trzylitrowego, wypełniając ten ostatni całkowicie

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

7: Przelej wodę z dzbana trzylitrowego do czterolitrowego, tak aby opróżnić ten pierwszy

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

8: Przelej wodę z dzbana czterolitrowego do trzylitrowego, tak aby opróżnić ten pierwszy

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

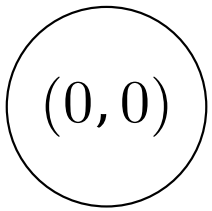
Algorytm

Ustaliliśmy, że istnieje przestrzeń stanów opisujący zawartość wody w obu dzbanach, a przejścia między tymi stanami możliwe są za pomocą operacji wyrażonych przy użyciu operatorów. Tę przestrzeń możemy opisać grafem skierowanym, którego wierzchołkami będą poszczególne stany, a krawędziami operacje prowadzące od stanu bieżącego do stanu następnego. Rozwiązanie problemu dzbanów z wodą sprowadza się zatem do znalezienia ścieżki w tym grafie, która prowadzi od wierzchołka początkowego, opisującego puste dzbany do jednego z wierzchołków końcowych (ang. *goal vertex*), odpowiadającego stanowi, w którym jeden z dzbanów zawiera dwa litry wody. Do znalezienia tej ścieżki można zastosować algorytm DFS. Pozostaje jednak pewna przeszkoda - nie mamy tego grafu podanego w jawnej postaci, nie wiemy, jak jest zbudowany. Znamy tylko jeden z jego wierzchołków, który odpowiada stanowi początkowemu i zbiór operatorów, które mogą stanowić krawędzie między wierzchołkami tego grafu. Oznacza to, że możemy ten graf wygenerować, ale okazuje się to niepotrzebne.

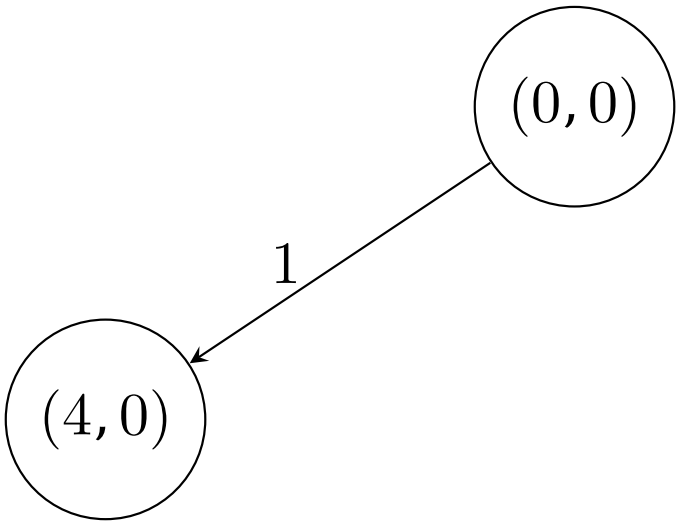
Algorytm

Wystarczy tylko, abyśmy generowali wierzchołki należące do bieżąco badanej ścieżki w tym grafie. Jeśli któryś z nich się powtórzy, to należy wrócić do jego poprzednika i znaleźć innego jego sąsiada (wygenerować inny stan). Tworzenie kolejnych stanów należy zakończyć po znalezieniu wierzchołka spełniającego warunek celu. Otrzymana ścieżka będzie stanowiła rozwiązanie problemu dzbanów z wodą. Na tym polega właśnie działanie algorytmu z nawrotami. Następny slajd zawiera animację ilustrującą częściowo jego działanie.

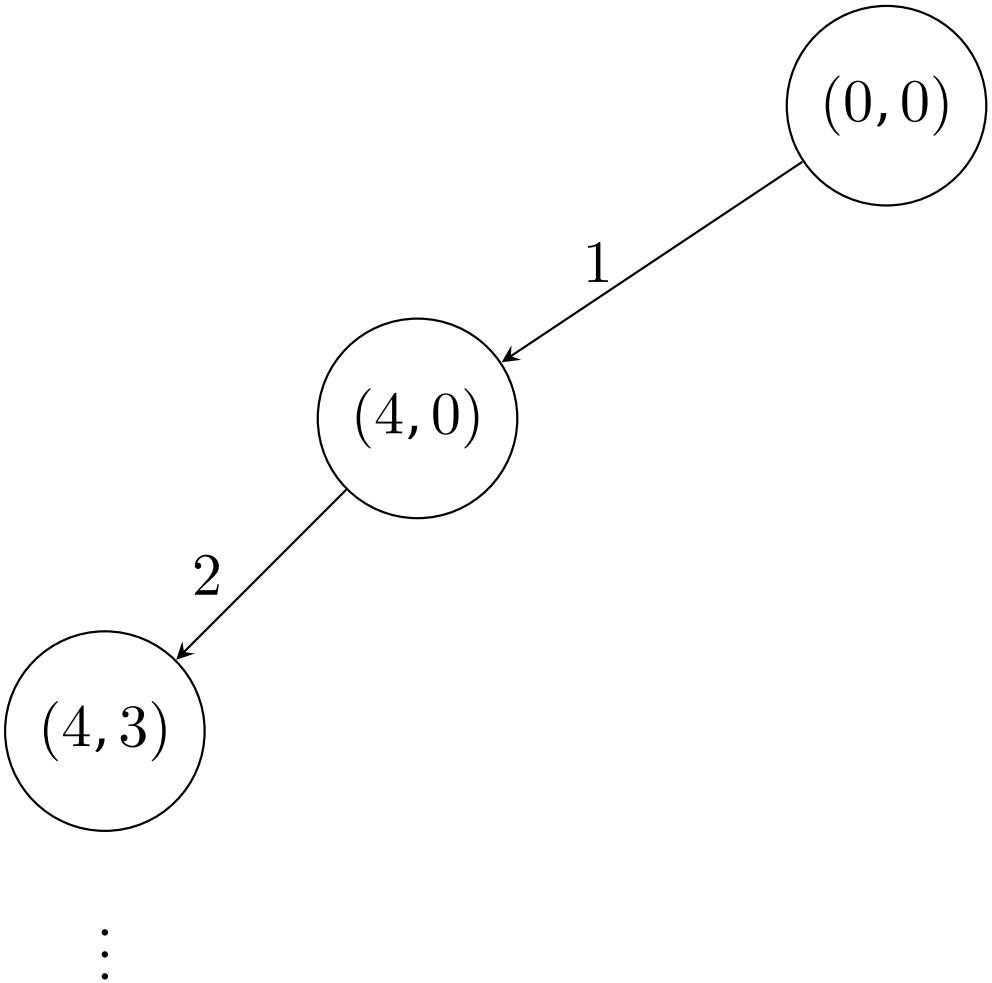
Algorytm



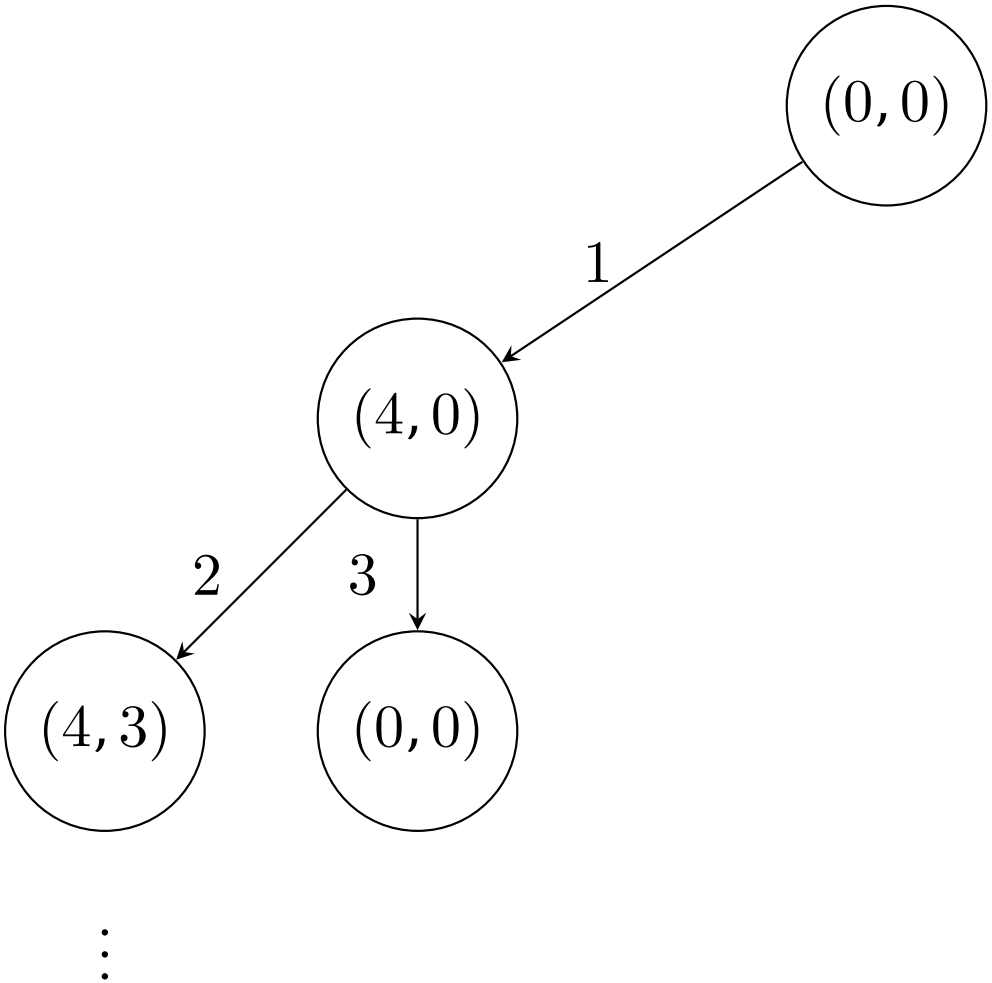
Algorytm



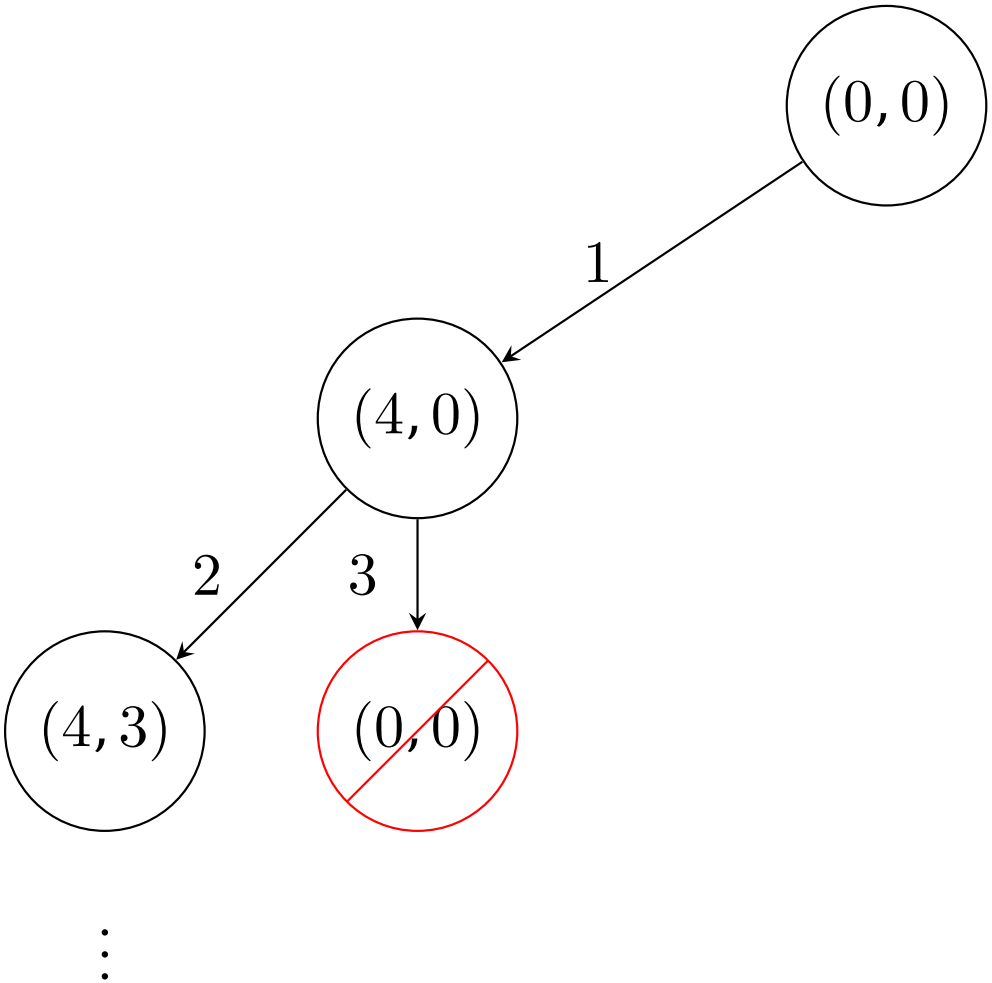
Algorytm



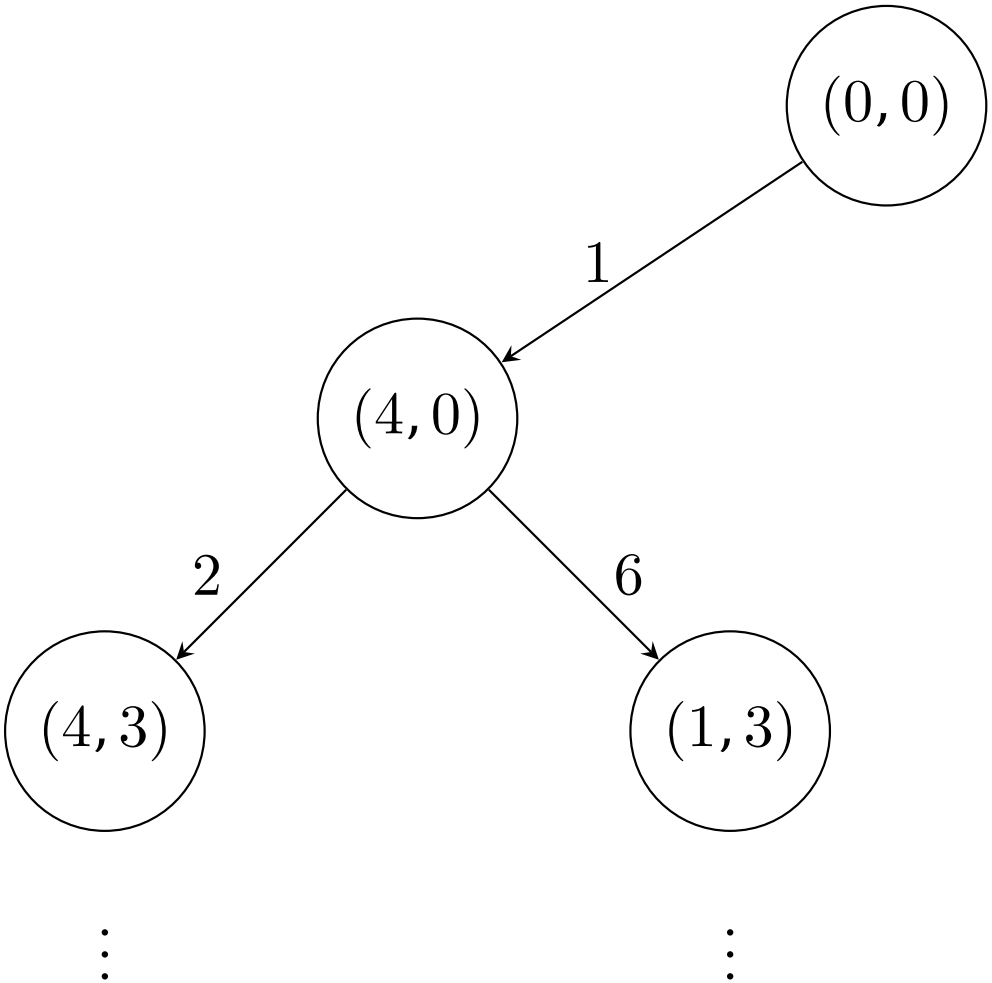
Algorytm



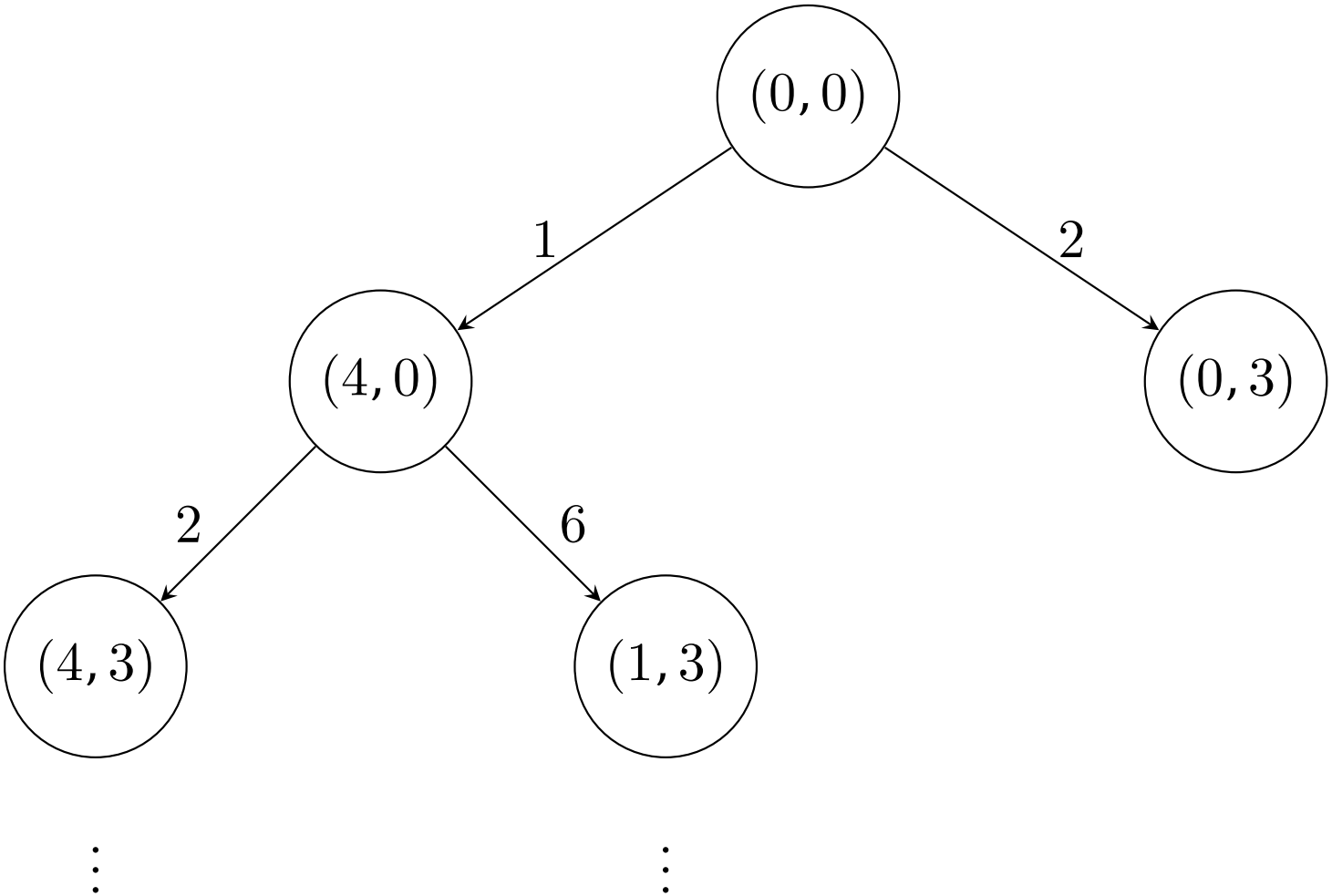
Algorytm



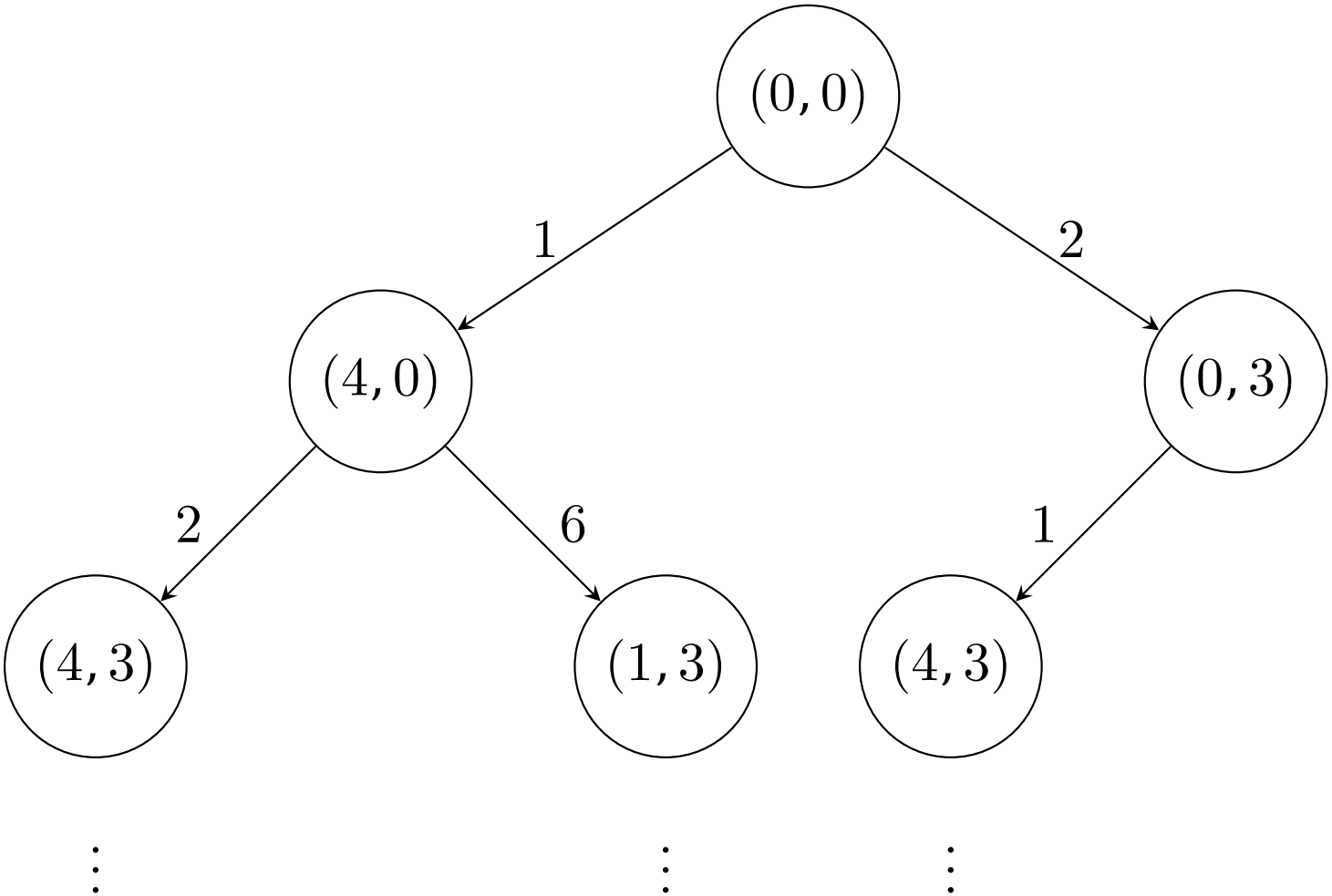
Algorytm



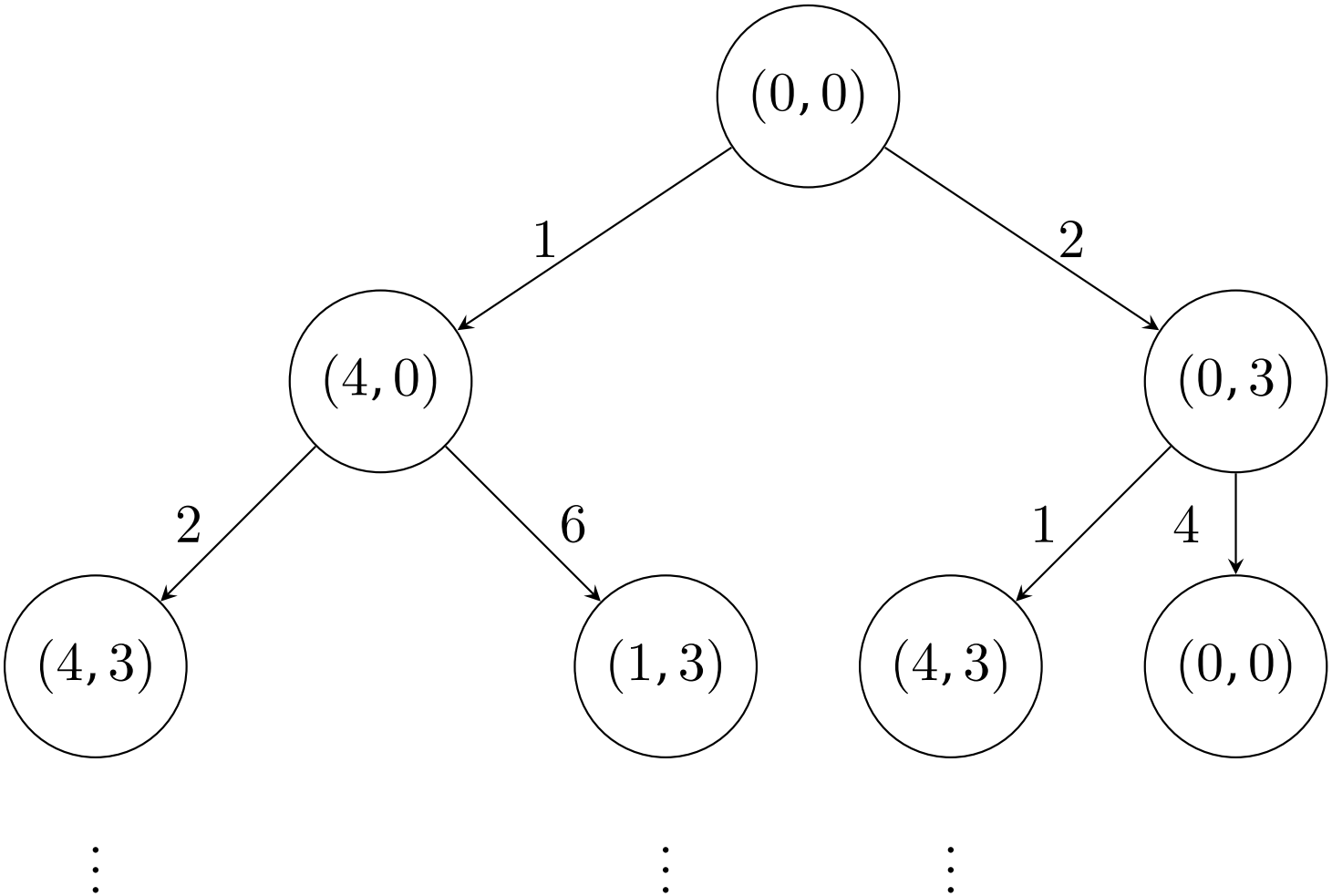
Algorytm



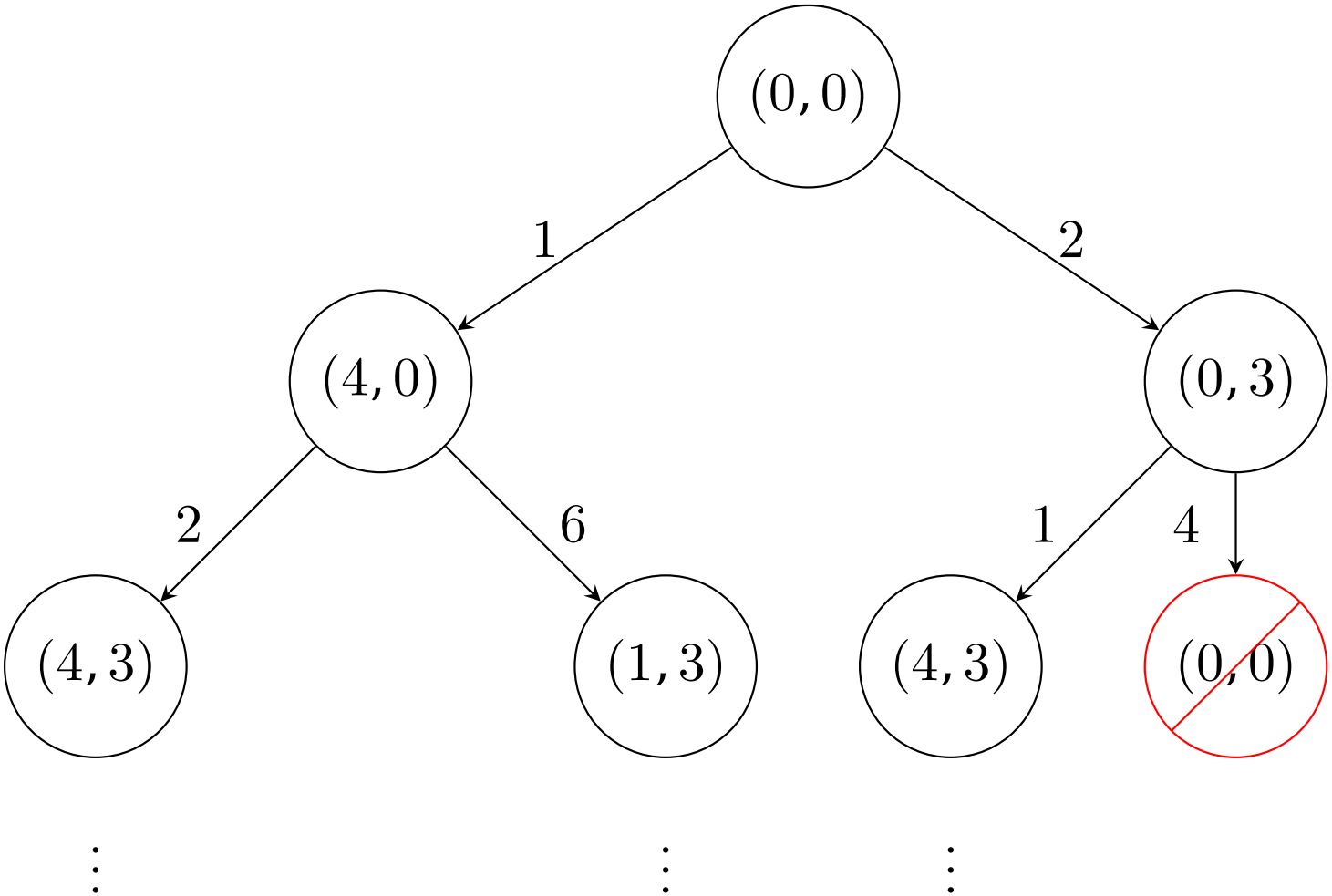
Algorytm



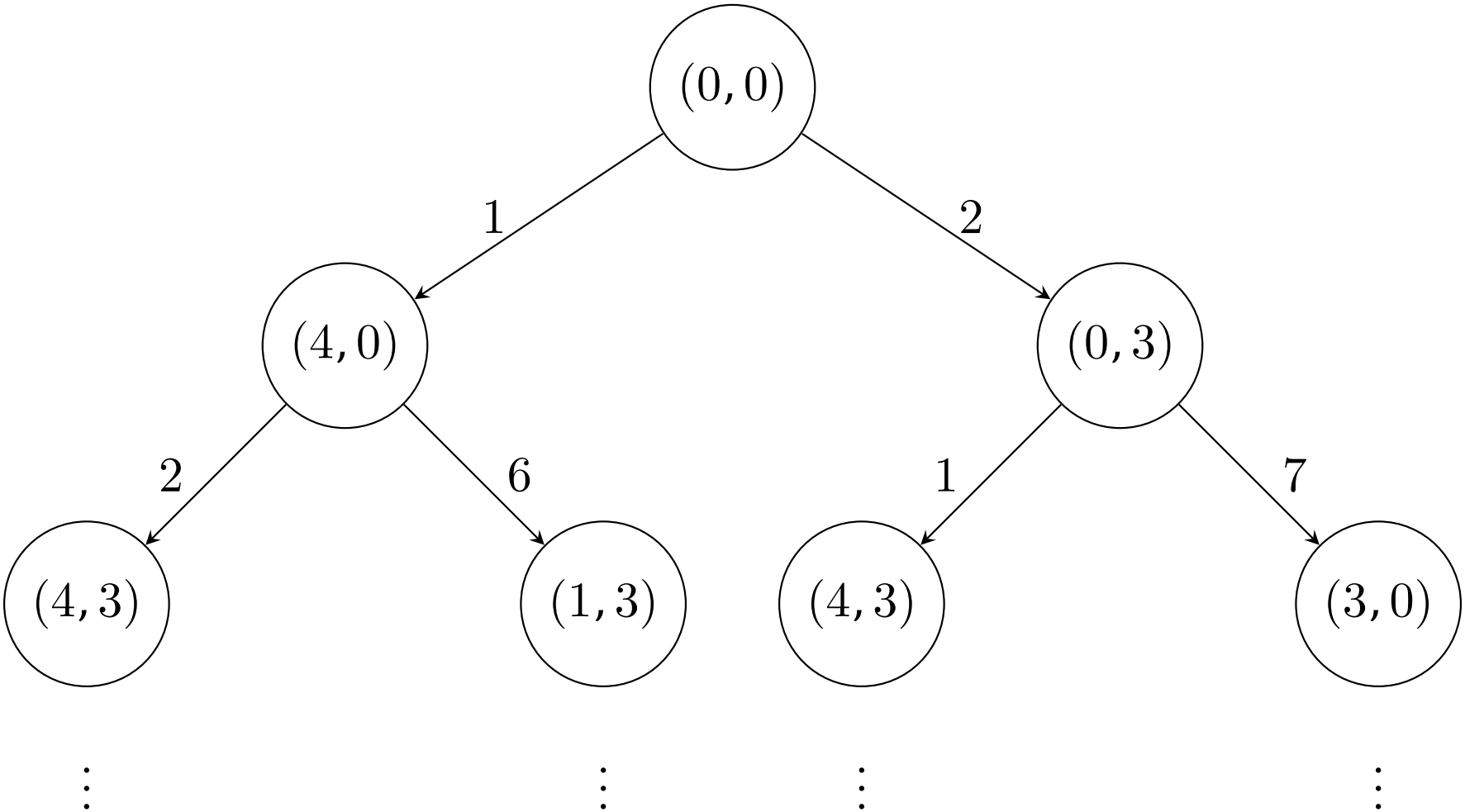
Algorytm



Algorytm



Algorytm



Algorytm

Wersja z jednym rozwiązaniem

Aby łatwiej było wyrazić opisany algorytm w postaci programu komputerowego, zapiszemy go w postaci pseudokodu, czyli notacji pośredniej między zapisem algorytmu zrozumiałym dla człowieka, a zrozumiałym dla komputera. Przedstawiona wersja będzie jednak znajdywała tylko jedno rozwiązanie problemu dzbanów z wodą.

Algorytm

Wersja z jednym rozwiązaniem

Pseudokod dla jednego rozwiązania

```
znajdź_rozwiązanie(ścieżka)
{
    dla(każdy operator){
        jeśli(można_zastosować(operator,ostatni_stan(ścieżka))){
            nowy_stan = operator(ostatni_stan(ścieżka));
            jeśli(jeszcze_nie_wystąpił(ścieżka,nowy_stan)){
                dodaj(ścieżka,nowy_stan);
                jeśli(warunek_celu(nowy_stan))
                    wypisz(ścieżka);
                w_przeciwnym_przypadku
                    znajdź_rozwiązanie(ścieżka);
            } w_przeciwnym_przypadku
                usuń(nowy_stan);
        }
    }
}
```

Algorytm

Wersja z wieloma rozwiązaniami

Możemy wnioskować na podstawie jego definicji, że problem dzbanów z wodą może mieć wiele rozwiązań. Pojawia się zatem pytanie jak przekształcić algorytm z poprzedniego slajdu, aby znajdywał on wszystkie rozwiązania. Okazuje się, że musimy w tym celu rozszerzyć pojęcie „nawrotu” algorytmu (inaczej „powrotu”). Powinien on wracać do poprzednio wygenerowanego wierzchołka nie tylko wtedy, gdy napotka stan, który wystąpił już wcześniej na badanej ścieżce, ale zawsze po zbadaniu ścieżki związanej z którymkolwiek z jego sąsiadów. W ten sposób będzie on badań wszystkie ścieżki związane z wszystkimi sąsiadami danego wierzchołka, co pozwoli mu znaleźć wszystkie rozwiązania problemu. Oznacza to, że zapis algorytmu z poprzedniego slajdu należy uzupełnić o instrukcję usuwającą, po powrocie z rekurencyjnego wywołania, ostatni wierzchołek z bieżąco badanej ścieżki. Modyfikacja ta została wykonana w pseudokodzie zaprezentowanym na następnym slajdzie.

Algorytm

Wersja z wieloma rozwiązaniami

Pseudokod dla wielu rozwiązań

```
znajdź_rozwiązania(ścieżka)
{
    dla(każdy operator){
        jeśli(można_zastosować(operator,ostatni_stan(ścieżka))){
            nowy_stan = operator(ostatni_stan(ścieżka));
            jeśli(jeszcze_nie_wystąpił(ścieżka,nowy_stan)){
                dodaj(ścieżka,nowy_stan);
                jeśli(warunek_celu(nowy_stan))
                    wypisz(ścieżka);
                w_przeciwnym_przypadku
                    znajdź_rozwiązania(ścieżka);
                usuń_ostatni_stan(ścieżka);
            } w_przeciwnym_przypadku
                usuń(nowy_stan);
        }
    }
}
```

Algorytm z nawrotami - implementacja

Na kolejnych slajdach zostanie zaprezentowany kod źródłowy programu implementującego algorytm z nawrotami znajdujący wszystkie rozwiązania problemu dzbanów z wodą. Jest to dosyć rozbudowany program, ale każdy jego fragment jest odpowiednio opisany, celem ułatwienia jego zrozumienia.

Algorytm z nawrotami - implementacja

Włączenie plików nagłówkowych oraz definicje

```
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 };
```

Algorytm z nawrotami - implementacja

Włączenie plików nagłówkowych oraz definicje

Wiersze 1-3 z poprzedniego slajdu zawierają instrukcje włączające pliki nagłówkowe do kodu programu. W wierszu nr 5 zdefiniowana jest stała określająca liczbę operatorów, które będą używane do rozwiązania problemu dzbanów z wodą. Ponieważ operatory te zostaną zapisane w tablicy, to ta stała będzie również wyznaczała liczbę jej elementów. Wiersze 7-9 zawierają definicję typu wyliczeniowego. Zmienna tego typu będzie używana do iterowania po tablicy operatorów, ale również jego elementy będą używane samodzielnie do numerowania operatorów, na podstawie których powstały określone wierzchołki grafu. Operatory będą numerowane od zera, ale pierwszy element tego zbioru ma wartość `-1`. Będzie on służył do zaznaczenia, że dany stan nie powstał przy użyciu operatora i to oznaczenie będzie dotyczyło wyłącznie stanu początkowego. W wierszach 11-12 zdefiniowany jest typ struktury, której pola będą zawierały informację o ilości wody w obu dzbanach.

Algorytm z nawrotami - implementacja

Definicje typów danych

```
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);
```

Algorytm z nawrotami - implementacja

Definicje typów danych

Na poprzednim slajdzie wiersze 1-5 zawierają definicję typu bazowego kolejki używanej do zapisywania bieżąco analizowanej ścieżki grafu. Będzie to kolejna z ograniczonym wejściem. Pole `states` w typie jest strukturą opisującą stan dzbanów, pole `operator_number` będzie zawierało numer operatora, za pomocą którego uzyskano ten stan. Pole `next` jest wskaźnikiem na następny element kolejki. Wiersze 7-9 zawierają definicję struktury wskaźników tej kolejki. W wierszach 11-12 znajduje się definicja typu wskaźnika na funkcję, która będzie jako argument przyjmowała strukturę opisującą stan i zwracała wartość typu `bool` informującą, czy dla tego stanu można zastosować określony operator. Innymi słowy ta funkcja będzie odpowiedzialna za sprawdzenie, warunku, który znajduje się po lewej stronie symbolu „$\rightarrow$” w formalnym zapisie operatora. Wiersze 14-15 zawierają definicję typu wskaźnika na funkcję, która na podstawie przekazanego jej jako argument stanu wygeneruje nowy stan, a wiec będzie to funkcja implementująca określony operator.

Algorytm z nawrotami - implementacja

Definicje typów danych

Jej zachowanie będzie zgodne z częścią zapisu formalnego tego operatora znajdującą się po prawej stronie symbolu „→”.
Kolejne pięć slajdów zawiera definicje takich funkcji, dla każdego z operatorów. Proszę zwrócić uwagę, że ich kod jest zgodny z przedstawionym wcześniej zapisem formalnym każdego operatora, który implementują.

Algorytm z nawrotami - implementacja

Funkcje operatorów

```
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 }
```

Algorytm z nawrotami - implementacja

Funkcje operatorów

```
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 }
```

Algorytm z nawrotami - implementacja

Funkcje operatorów

```
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 }
```

Algorytm z nawrotami - implementacja

Funkcje operatorów

```
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 }
```

Algorytm z nawrotami - implementacja

Funkcje operatorów

```
1  struct jugs_states empty_4_liters_jug_to_3_liters
2                                              (struct jugs_states state)
3  {
4      state.jug_3_liters_state = state.jug_3_liters_state +
5                                               state.jug_4_liters_state;
6      state.jug_4_liters_state = 0;
7      return state;
8  }
```

Algorytm z nawrotami - implementacja

Tablica operatorów

```
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     },
```

Algorytm z nawrotami - implementacja

Tablica operatorów

```
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      },
```

Algorytm z nawrotami - implementacja

Tablica operatorów

```
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   };
```

Algorytm z nawrotami - implementacja

Tablica operatorów

Poprzednie trzy slajdy zawierają deklarację i inicjację tablicy operatorów. Jest to tablica struktur, które są wskaźnikami na funkcję. Został tu zatem wykorzystany element techniki obiektowej - wskazywane funkcje będą pełniły rolę podobną do metod obiektu. Typ elementów tablicy zostały zdefiniowany na pierwszym slajdzie w wierszach 1-4. Określa on strukturę, której obydwa pola są wskaźnikami na funkcje, których typy został opisany wcześniej. Sama tablica jest zadeklarowana na tym samym slajdzie w wierszu nr 4. Reszta wierszy, również na pozostałych slajdach, to inicjacja poszczególnych elementów tablicy. Wykorzystano tu *inicjację oznaczoną*, która pozwala wskazać element tablicy, któremu ma być przypisana określona wartość, poprzez umieszczenie jego indeksu w nawiasach kwadratowych i użycie operatora przypisania. Przykładowo zainicjowanie elementu o indeksie `5` w dziesięcioelementowej tablicy elementów typu `int` będzie zapisane następująco:

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

Algorytm z nawrotami - implementacja

Tablica operatorów

Pozostałe elementy przykładowej tablicy uzyskają wartość `0`. W przypadku tablicy operatorów jako indeksy elementów wykorzystano elementy typu wyliczeniowego `operator_index`. Ponieważ jest to tablica struktur wskaźników, to każdemu polu, każdego elementu jest przypisywany adres odpowiedniej funkcji związanej z określonym operatorem.

Algorytm z nawrotami - implementacja

Funkcje `enqueue()` i `dequeue()`

```
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 }
```

Algorytm z nawrotami - implementacja

Funkcje `enqueue()` i `dequeue()`

Poprzedni slajd zawiera definicje funkcji służących do dodawania nowego elementu i usuwania elementu z czoła kolejki, w której zapisana jest badana ścieżka grafu. Funkcja `enqueue()` zdefiniowana w wierszach 1-6 nie zwraca żadnej wartości, ale przyjmuje dwa argumenty wywołania: adres struktury zawierającej wskaźniki kolejki i adres nowego elementu, który doda na końcu tej kolejki. Funkcja ta zakłada, że kolejka nie jest pusta, tzn. zawiera co najmniej jeden element. W wierszu nr 4, w polu `next` ostatniego elementu kolejki funkcja ta zapisuje adres nowego elementu, a następnie ten sam adres zapisuje we wskaźniku ostatniego elementu kolejki (wiersz nr 5) i kończy działanie. Funkcja `dequeue()` została zdefiniowana podobnie, jak w programie demonstrującym działanie algorytmu DFS, który został zaprezentowany na poprzednim wykładzie.

Algorytm z nawrotami - implementacja

Funkcja `remove_queue()`

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

Algorytm z nawrotami - implementacja

Funkcja `remove_queue()`

Funkcja `remove_queue()` służy do likwidowania kolejki i została ona zdefiniowana tak samo jak w programach zaprezentowanych na poprzednim wykładzie.

Algorytm z nawrotami - implementacja

Funkcja `already_been()`

```
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 }
```

Algorytm z nawrotami - implementacja

Funkcja `already_been()`

Zdaniem funkcji `already_been()` jest sprawdzenie, czy nowy element kolejki nie opisuje stanu, który już wystąpił na ścieżce. Przyjmuje ona dwa argumenty wywołania: strukturę wskaźników kolejki oraz adres nowego elementu. Zwraca wartość typu `bool`. Jeśli będzie nią `true`, to znaczy, że stan opisywany przez nowy element już wystąpił, a jeśli `false`, to że jeszcze się nie pojawił. Aby to ustalić funkcja przegląda w pętli `while` wszystkie elementy kolejki, które dotychczas były do niej dodane i porównuje zapisane w nich stany ze stanem zapisanym w nowym elemencie (wiersze 5-8). Jeśli wystąpi zgodność, to funkcja zakończy swe działanie zwracając wartość `true` (wiersz nr 9). Jeśli zaś nie będzie zgodności, to pętla zakończy się po sprawdzeniu wszystkich elementów kolejki i funkcja zwróci wartość `false` (wiersz nr 12).

Algorytm z nawrotami - implementacja

Funkcja `remove_tail()`

```
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 }
```

Algorytm z nawrotami - implementacja

Funkcja `remove_tail()`

Zadaniem funkcji `remove_tail()` jest usunięcie ostatniego elementu w kolejce. Czynność ta jest niezbędna do tego, aby algorytm z nawrotami mógł znaleźć dodatkowe możliwości rozwiązania badanego problemu. Funkcja `remove_tail()` przyjmuje jako argument wywołania adres struktury wskaźników kolejki i nie zwraca żadnej wartości. W wierszu 4 sprawdza ona, porównując wartości wskaźników `head` i `tail`, czy kolejka zawiera tylko jeden element. Jeśli tak jest, to funkcja zwalnia pamięć przydzieloną na ten element (wiersz nr 5), przypisuje wskaźnikom kolejki wartość `NULL` (wiersz nr 6) i kończy swe działanie (wiersz nr 7). Jeśli kolejka ma więcej elementów, to funkcja w pętli `while` (wiersze 10-11) wyszukuje przedostatni z nich (zawiera on w polu `next` adres ostatniego elementu kolejki). Po zlokalizowaniu tego elementu funkcja usuwa ostatni element (wiersz nr 12), zapisuje wartość `NULL` w polu `next` dotychczas przedostatniego elementu (wiersz nr 13) i zapisuje jego adres we wskaźniku `tail` kolejki (wiersz nr 14).

Algorytm z nawrotami - implementacja

Funkcja `print_solution()`

```
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           "Napełnij dzban czterolitrowy.",
7           "Napełnij dzban trzylitrowy.",
8           "Opróżnij dzban czterolitrowy.",
9           "Opróżnij dzban trzylitrowy.",
10          "Przelej wodę z dzbana trzylitrowego, do czterolitrowego,\
11                          tak aby wypełnić ten ostatni całkowicie.",
12          "Przelej wodę z dzbana czterolitrowego, do trzylitrowego,\
13                                  tak aby wypełnić ten ostatni całkowicie.",
14          "Przelej wodę z dzbana trzylitrowego, do czterolitrowego,\
15                                  tak aby opróżnić ten pierwszy całkowicie.",
16          "Przelej wodę z dzbana czterolitrowego, do trzylitrowego,\
17                                  tak aby opróżnić ten pierwszy całkowicie."
18      };
```

Algorytm z nawrotami - implementacja

Funkcja `print_solution()`

```
1       printf("Rozwiązanie nr %hhu:\n",++solution_number);
2       while(queue.head) {
3           enum operator_index operator =
4                                           queue.head->operator_number;
5           if(operator!=NONE) {
6               printf("Krok numer %hhu:\n",step++);
7               printf("%s\n",operators_description[operator]);
8           } else
9               puts("Stan początkowy:");
10          printf("Woda w dzbanie czterolitrowym: %u \
11                  i trzylitrowym: %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("KONIEC");
18  }
```

Algorytm z nawrotami - implementacja

Funkcja `print_solution()`

Poprzednie dwa slajdy zawierają kod źródłowy funkcji o nazwie `print_solution()`, której zadaniem jest wypisanie jednego ze znalezionych przez program rozwiązań problemu dzbanów z wodą. Działanie to sprowadza się do odpowiedniego zinterpretowania zapisanej w kolejce ścieżki w grafie stanowiącym przestrzeń rozwiązań. Jako argument wywołania funkcja przyjmuje strukturę wskaźników tej kolejki i nie zwraca żadnej wartości. W wierszu nr 3 zadeklarowana jest statyczna zmienna lokalna, która będzie służyła do numerowania kolejnych rozwiązań wypisywanych przez funkcję. To, że jest ona statyczna, oznacza, że jest automatycznie inicjowana wartością `0` oraz, że nie jest ona niszczona między kolejnymi wywołaniami opisywanej funkcji. Wiersz nr 4 zawiera deklarację zmiennej służącej do numerowania kolejnych kroków rozwiązania (kolejnych wykonywanych czynności). Jest to zwykła zmienna lokalna, zainicjowana liczbą `1`. Wiersze 5-18 zawierają deklarację i inicjację tablicy ciągów znaków będących słownymi opisami poszczególnych operatorów.

Algorytm z nawrotami - implementacja

Funkcja `print_solution()`

W wierszu nr 1 na drugim slajdzie opisywana funkcja wypisuje komunikat informujący o tym, które znalezione rozwiązanie będzie wypisywać. Liczbę tę uzyskuje zwiększając przy pomocy operatora preinkrementacji wartość zmiennej `solution_number`. Następnie, w pętli `while` funkcja iteruje po kolejnych elementach umieszczonych w kolejce i dla każdego z nich zapamiętuje w zmiennej `operator` zapisany w nim numer operatora, a następnie porównuje go z elementem `NONE` typu wyliczeniowego `operator_index`. Jeśli nie są one sobie równe, to wypisuje na ekranie powiększoną o jeden wartość zmiennej `step` (wiersz nr 6) oraz opis operatora, określonego wartością zmiennej `operator`. W przeciwnym przypadku wypisuje komunikat, że wydrukuje informacje o stanie początkowym. Kolejne czynności wykonywane w tej funkcji są wspólne zarówno dla stanu początkowego, jak i pozostałych.

Algorytm z nawrotami - implementacja

Funkcja `print_solution()`

W wierszach 10-13 na drugim slajdzie z kodem źródłowym funkcji `print_solution()`, wypisywany jest na ekran stan wody w dzbanach, zapisany w bieżąco przeglądanym przez pętlę elemencie kolejki, a następnie wstrzymywane jest działanie funkcji, do czasu naciśnięcia przez użytkownika dowolnego klawisza klawiatury (wiersz nr 14), po czym pętla przechodzi do kolejnego elementu kolejki (wiersz nr 15). Po jej zakończeniu funkcja wypisuje na ekranie monitora informację, że jest to już koniec opisu pojedynczego znalezionego rozwiązania problemu dzbanów z wodą i kończy swe działanie.

Algorytm z nawrotami - implementacja

Funkcja `create_new_state()`

```
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 }
```

Algorytm z nawrotami - implementacja

Funkcja `create_new_state()`

Funkcja `create_new_state()` tworzy nowy element kolejki, opisujący nowy, wygenerowany stan wody w dzbanach. Jako argumenty wywołania przyjmuje ona strukturę opisującą ten nowy stan, oraz numer operatora, który posłużył do jego uzyskania, a zwraca adres elementu, kolejki, który zawiera obie te informacje. W wierszach 4-5 funkcja przydziela pamięć na nowy element kolejki. Jeśli ten przydział się powiedzie, to będzie spełniony warunek w instrukcji warunkowej z wiersza nr 6 i funkcja przystąpi do inicjacji pól nowego elementu. W wierszu nr 7 zapisze w nim stan wody w dzbanach, w wierszu nr 8 numer operatora, a w wierszu nr 9 nada polu `next` tego elementu wartość `NULL`. Funkcja kończy swe działania w wierszu nr 11 zwracając adres nowego elementu kolejki lub wartość `NULL`, jeśli nie udało się go utworzyć.

Algorytm z nawrotami - implementacja

Funkcja `initialize_queue()`

```
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 }
```

Algorytm z nawrotami - implementacja

Funkcja `initialize_queue()`

Funkcja `initialize_queue()` odpowiedzialna jest za inicjację kolejki, czyli dodanie do niej pierwszego elementu, który opisuje stan początkowy na ścieżce, a więc stan, w którym oba dzbany są puste. Ponieważ jest ona wywoływana przed `enqueue()` to w tej ostatniej mogliśmy założyć, że kolejka zawiera co najmniej jeden element i nie badać, czy jest pusta. Opisywana funkcja przyjmuje jako argument wywołania adres struktury wskaźników kolejki i nie zwraca żadnej wartości. W wierszach 3-4 przydziela ona pamięć na pierwszy element kolejki. Jeśli ten przydział się powiedzie, to będzie spełniony warunek w instrukcji warunkowej z wiersza nr 5. W takim wypadku funkcja zainicjuje pole `state` tego elementu, tak aby opisywało ono sytuację, w której oba dzbany zawierają `0` litrów wody (wiersze 6-7), zapisze jako numer operatora wartość `NONE` typu wyliczeniowego `operator_index`, a następnie zainicjuje pole wskaźnikowe `next` elementu wartością `NULL` (wiersz nr 9). Na koniec (wiersz nr 10) funkcja zapisze adres pierwszego elementu we wskaźnikach kolejki.

Algorytm z nawrotami - implementacja

Funkcja `search()`

```
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  }
```

Algorytm z nawrotami - implementacja

Funkcja `search()`

Kod funkcji `search()` powstał w oparciu o pseudokod zaprezentowany na początku wykładu, który opisuje algorytm z nawrotami znajdujący wszystkie rozwiązania problemu dzbanów z wodą. Funkcja ta nie zwraca żadnej wartości, jako argumenty wywołania przyjmuje adres struktury wskaźników kolejki i tablicę operatorów. Proszę zwrócić uwagę, że ten drugi argument jest przekazywany przez stałą. W wierszu nr 3 tej funkcji zadeklarowana jest zmienna lokalna, która posłuży jako zmienna sterująca pętli `for`. Ta pętla iteruje po całej tablicy operatorów. W każdej iteracji sprawdza ona najpierw, czy istnieje ostatni element kolejki (wiersz nr 5) jeśli tak, to sprawdza, czy można zastosować operator określany przez zmienną `operator_index` dla tego stanu opisywanego przez ostatni element kolejki, aby wygenerować nowy stan (wiersz nr 6). Jeśli tak, to funkcja wywołuje ona `create_new_state()`, aby utworzyć nowy element kolejki, który będzie opisywał nowy stan (wiersze 7-9).

Algorytm z nawrotami - implementacja

Funkcja `search()`

Jeśli utworzenie tego elementu się powiedzie, co jest sprawdzane w wierszu nr 10, to funkcja sprawdza, czy zapisany w nim stan nie wystąpił już wcześniej w kolejce, wywołując w tym celu funkcję `already_been()` i negując wartość przez nią zwracaną (wiersz nr 11). Jeśli okaże się, że tak, to funkcja usunie nowy element (wiersz nr 20) i przejdzie do kolejnej iteracji pętli `for`. Jeśli zaś ten stan jeszcze nie wystąpił, to funkcja doda nowy element do kolejki (wiersz nr 12) i sprawdzi, czy nie opisuje on stanu spełniającego warunek celu (wiersze 13-14). Jeśli ten warunek będzie spełniony to funkcja uruchomi `print_solution()`, aby ta wypisała informację o znalezionym rozwiązaniu. Jeśli zaś nie, to funkcja wywoła się rekurencyjnie (wiersz nr 17), aby zbadać jakie operatory mogą być zastosowane dla stanu opisanego przez nowo dodany do kolejki element i jakie nowe stany mogą w ten sposób zostać wygenerowane.

Algorytm z nawrotami - implementacja

Funkcja `search()`

Niezależnie od tego, czy nastąpił powrót z wywołania rekurencyjnego, czy znaleziono stan spełniający warunek celu, funkcja `search()` usuwa z kolejki ostatni dodany do niej element (wiersz nr 18), aby kolejna iteracja pętli `for` mogła sprawdzić, czy dla jego poprzednika można zastosować inne z pozostałych operatorów i czy w ten sposób można znaleźć inne rozwiązania problemu dzbanów z wodą.

Algorytm z nawrotami - implementacja

Funkcja `main()`

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

Algorytm z nawrotami - implementacja

Funkcja `main()`

W funkcji `main()`, aby program wypisał na ekranie informacje o wszystkich możliwych rozwiązaniach problemu dzbanów z wodą wystarczy wywołać trzy ze zdefiniowanych wcześniej funkcji. Na początek wywoływana jest w wierszu nr 3 funkcja `initialize_queue()` celem stworzenia kolejki i dodania do niej elementu opisującego stan początkowy dzbanów z wodą. W wierszu nr 4 wywoływana jest funkcja `search()`, która wyszukuje możliwe rozwiązania i wypisuje informacje o nich na ekran monitora. W wierszu nr 5 wywoływana jest funkcja `remove_queue()` celem usunięcia kolejki, jeśli zawierała by ona jeszcze jakieś elementy pozostałe po działaniach, jakie przeprowadzała na niej funkcja `search()`.

Podsumowanie

Przedstawiony program znajduje 10 rozwiązań problemu dzbanów z wodą. Niektóre z nich są nieoptymalne, tzn. zawierają zbędne kroki. Rzeczywiście, algorytm z nawrotami znajduje wszystkie możliwe rozwiązania przedstawionego problemu, bez ich oceniania. Stosuje on zatem przeszukiwanie siłowe (ang. *brute force*). Aby wybierał on tylko optymalne rozwiązania, należałby go wyposażyć w funkcje heurystyczne, które oceniałby zasadność wykonywanych posunięć. Algorytmy z nawrotami pierwotnie służyły do znajdywania rozwiązań problemów z zakresu sztucznej inteligencji. Obecnie stosowane są one nie tylko w tej dziedzinie, ale również wykorzystuje się je np.: do znajdowania ekstremów złożonych funkcji wielu zmiennych lub do konstrukcji *parserów*, czyli elementów kompilatorów.

Pytania

?

KONIEC

Dziękuję Państwu za uwagę!