Изкуствен интелект - летен семестър, 2023/2024 учебна година

Тема 2, част 2:
Търсене на път до определена цел

Примерни задачи от реалния свят: намиране на път (при routing в компютърни мрежи, при планиране на пътувания с автомобилен/самолетен транспорт и др.), задача за търговския пътник, навигация на роботи, задачи за планиране и конструиране (асемблиране) на сложни обекти и др.

Характеристики на алгоритмите за търсене:

- пълнота
- оптималност
- сложност
  - по време ≈ брой изследвани възли
  - по памет ≈ максимален размер на фронта на търсенето

Необходимост от избягване на повторения и зацикляне.

Параметри на търсенето:

- дълбочина на „най-плитката“ цел – *d* (или максимална „дълбочина“ на пространството/графа на състоянията – *m*)
- коефициент на разклонение (разклоненост) на ГС – *b*

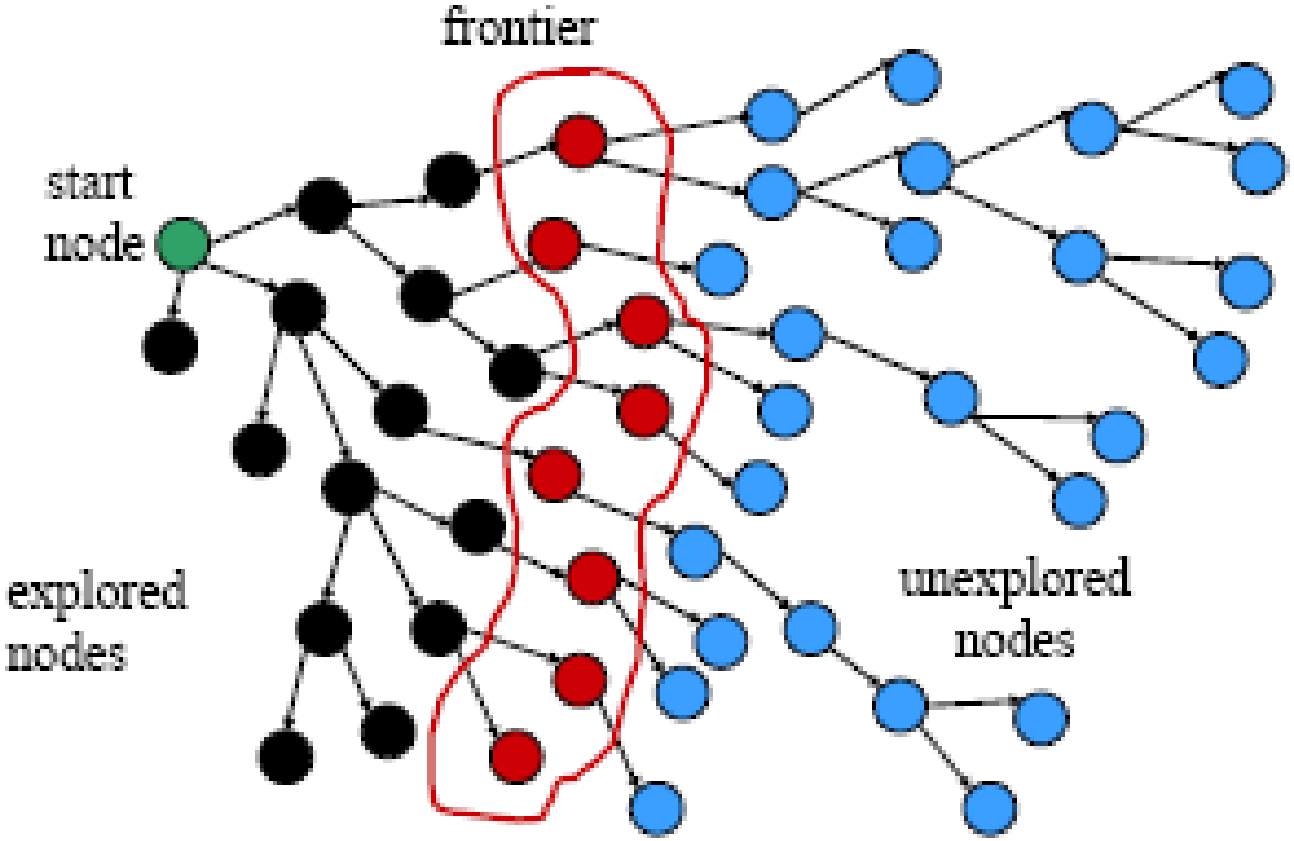
Методи за неинформирано („сляпо“) търсене

Обща характеристика: пълно изчерпване по твърда (фиксирана отнапред) стратегия. Прилагат се, когато липсва специфична информация за предметната област и оценяващата функция може само да провери дали дадено състояние е целево или не е.
Програмна реализация: чрез използване на работна памет (списък Open/frontier/fringe на т. нар. открити възли или списък от натрупани/изминати пътища, започващи от началния възел – нарича се още ***фронт на търсенето***).

Общ алгоритъм за неинформирано търсене

Тръгва се от началния възел, като на всяка стъпка фронтът на търсенето се разширява в посока към неизследваните възли, докато се достигне до целеви възел.

Начинът, по който се разширява фронтът, както и правилата за избор на конкретен елемент на фронта, от който ще продължи неговото разширяване, определят ***стратегията на търсене*** (***search strategy***).


frontier
start
node
explored
nodes
unexplored
nodes

Неинформирано търсене в дърво

Неформално описание на общия алгоритъм за неинформирано търсене в дърво

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
```

Практическа реализация – представяне на възлите на графа на състоянията

В следващите дефиниции на функции за търсене (написани на псевдокод) ще предполагаме, че всеки възел от ГС е структура от данни с 5 компонента:

STATE: състоянието от пространството, на което съответства разглежданият възел

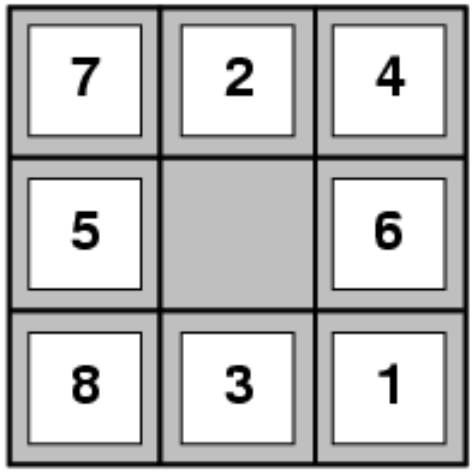
PARENT: възелът от ГС, като наследник на който е генериран разглежданият възел

ACTION: операторът, с помощта на който е бил генериран разглежданият възел

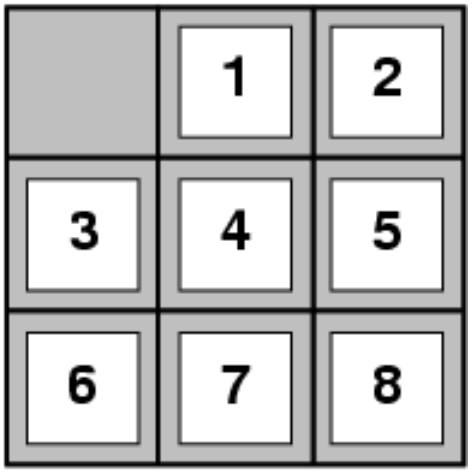
PATH-COST: цената (традиционно означавана с *g(n)*) на пътя от началното състояние до състоянието, съответно на разглеждания възел (в съответствие с указателите към родителските възли)

DEPTH: броят на стъпките, от които е съставен пътят от началното състояние до състоянието, съответно на разглеждания възел

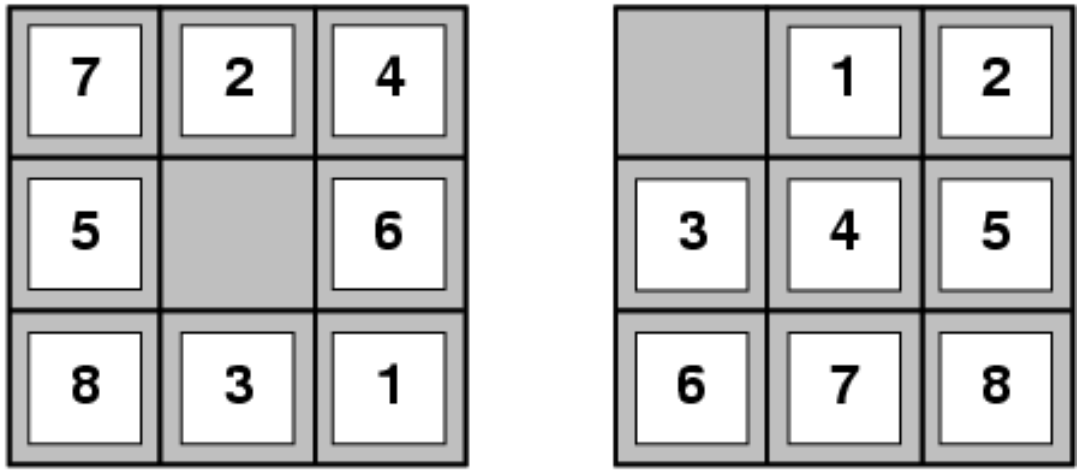
Пример: задача за 8-те плъзгащи се плочки (the 8-puzzle problem)



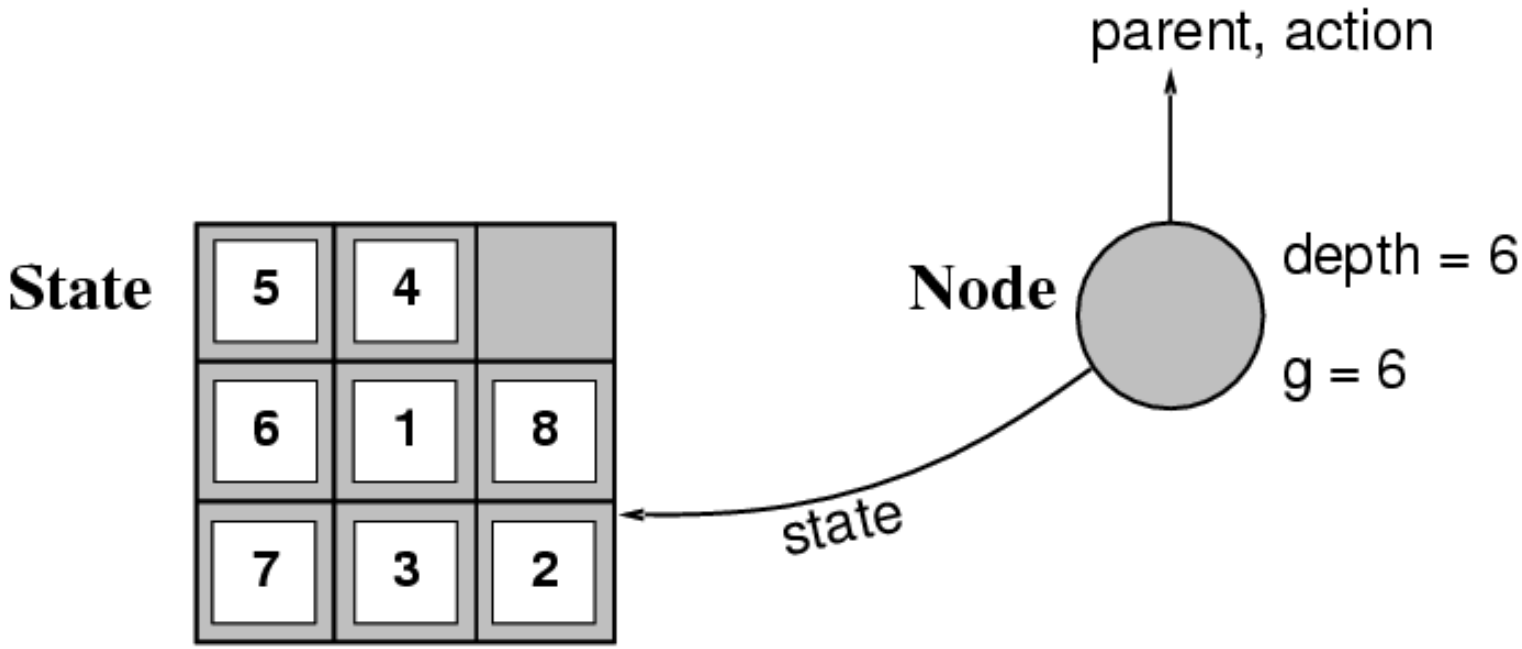
Start State



Goal State



Състояния: разположения (конфигурации) на 8-те плочки
Оператори: премествания на празното поле (left, right, up, down)
Проверка за достигане на целта: проверка за съвпадение с даденото целево състояние
Цена на пътя: 1 за всяко преместване


parent, action
State
5
4
6
1
8
7
3
2
Node
depth = 6
g = 6
state

- Всяко състояние (state) представя конкретна конфигурация на плочките;
- Всеки възел от ГС (node) се представя като структура от данни, която съдържа компоненти (полета), описващи състоянието, родителя, използвания оператор, цената на съответната част от пътя и дълбочината на вложение;
- Функцията EXPAND генерира нови възли, като за целта попълва съответните компоненти (полета) и в частност използва функцията SUCCESSOR-FN за създаване на съответните състояния.

Формално описание на общия алгоритъм за неинформирано търсене в дърво

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

```
function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

Описание на действието на използваните функции:

- FIRST(*list*) връща първия елемент на *list*
- REMOVE-FRONT(*list*) връща FIRST(*list*) и го изтрива от *list*
- INSERT(*element*, *list*) добавя *element* към *list* и връща получения списък
- INSERTALL(*elements*, *list*) добавя множество от елементи към даден списък и връща като резултат получения списък
- SOLUTION(*node*) връща като резултат поредицата от оператори, довели от началното до целевото състояние (в съответствие с указателите към родителските възли)

Забележка. Функцията TREE-SEARCH се изпълнява със стойност на аргумента *fringe*, равна на празен списък.

Неинформирано търсене в граф

Формално описание на общия алгоритъм за неинформирано търсене в граф

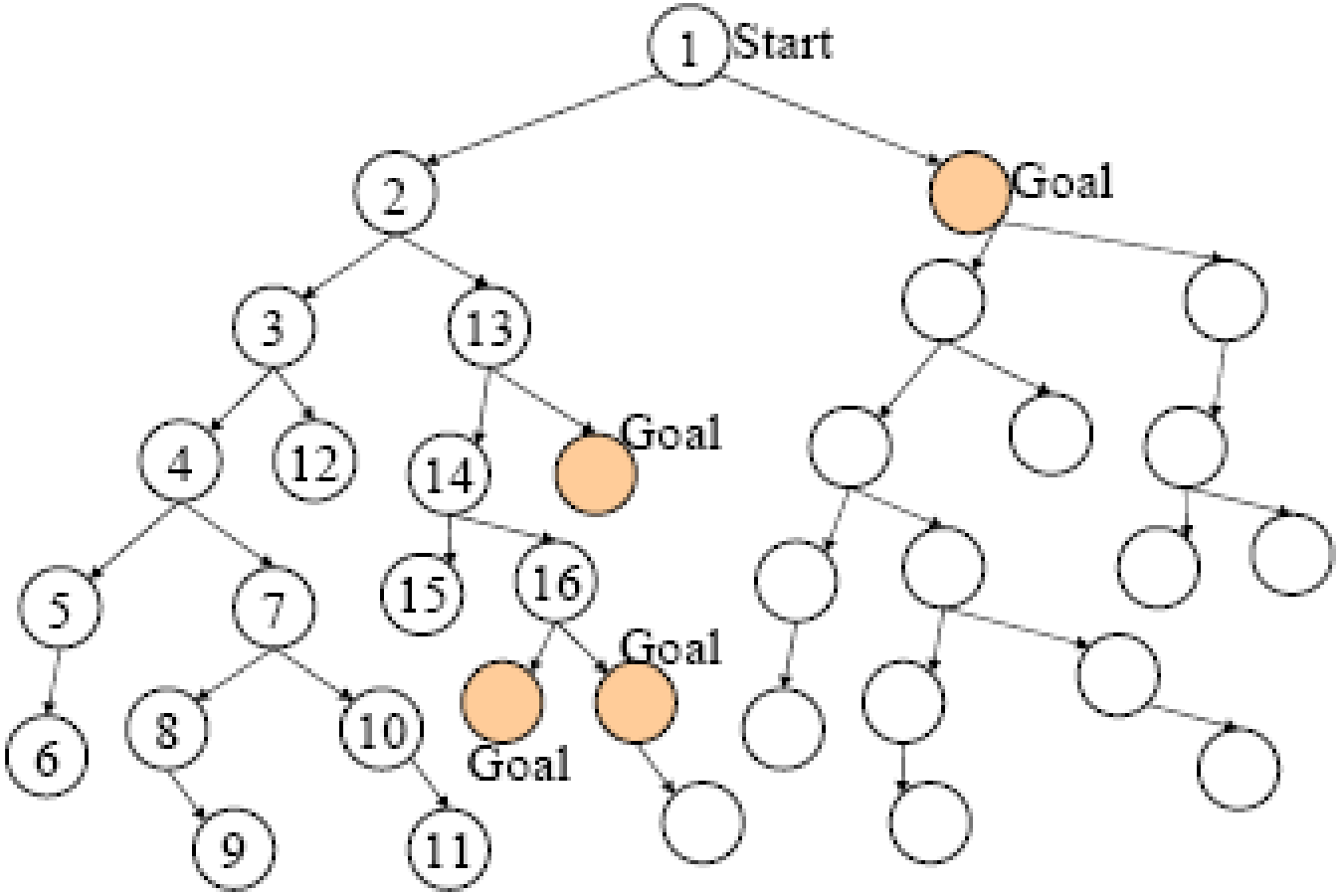
```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

Стратегии за неинформирано търсене

1. **Изчерпване (търсене) в дълбочина** (***depth-first search***) - добавяне на новите възли (новия възел) в началото на списъка Open/fringe.

   fringe = **стек** (LIFO, put successors at front)

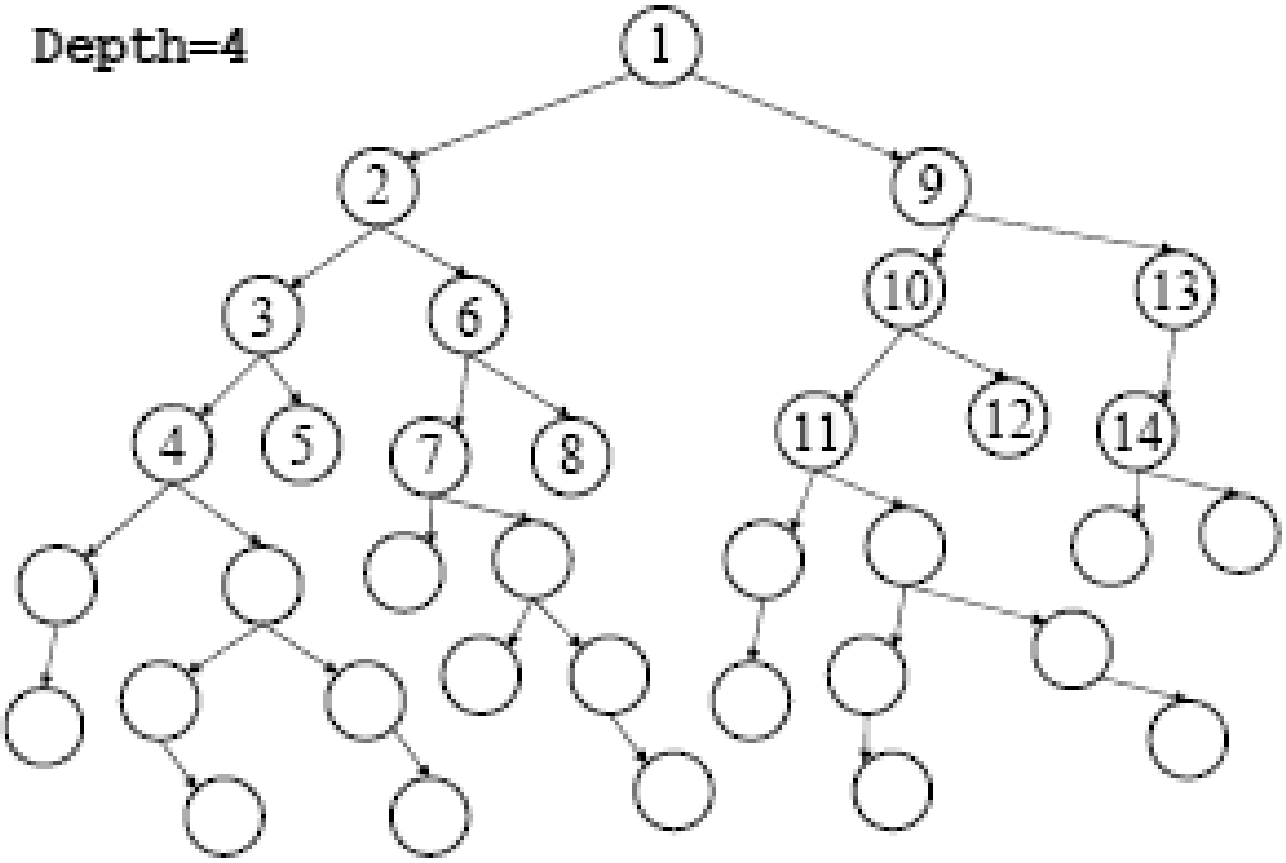
Търсенето е евтино (линейно по отношение на необходимата памет), но не е нито пълно, нито оптимално (завършва и е пълно, когато графът на състоянията е краен или поне не съществуват безкрайно дълги ациклични пътища в него).


1
Start
2
Goal
3
13
4
12
14
Goal
5
7
15
16
Goal
6
8
10
Goal
9
11

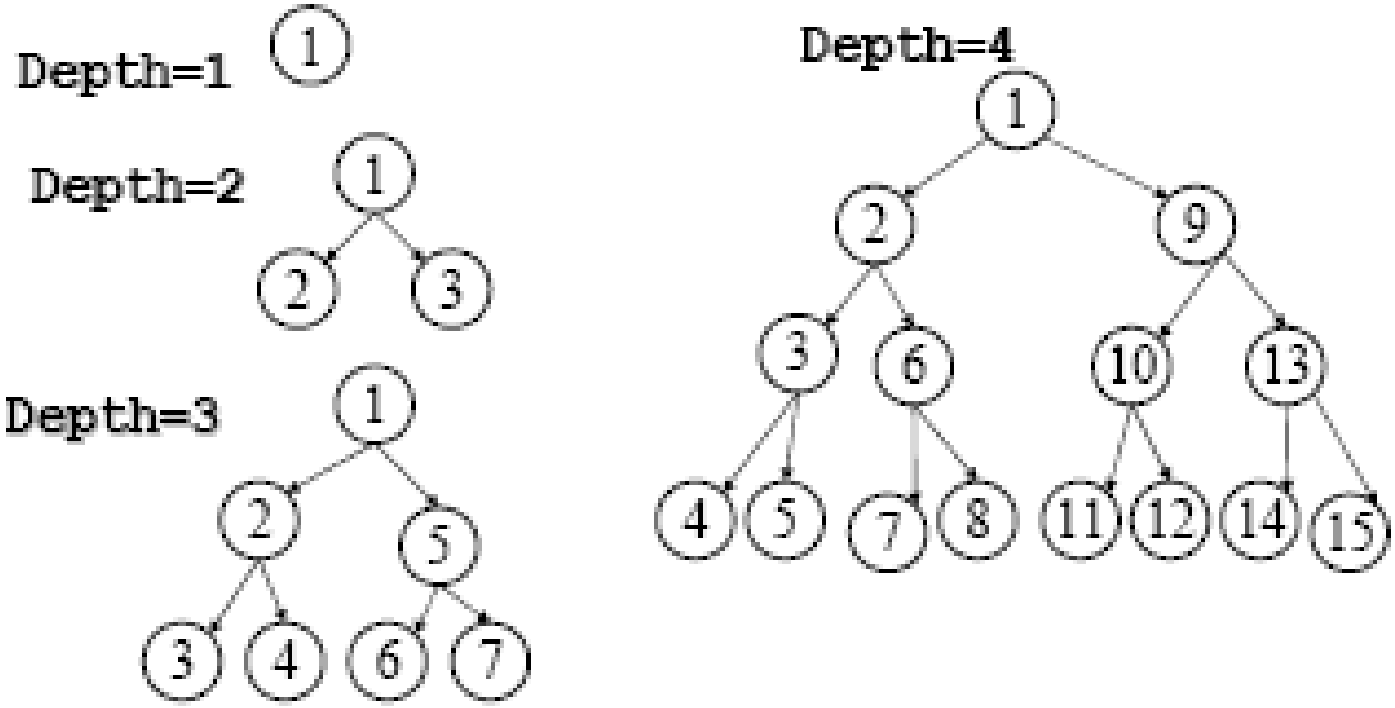
Построяване на пътя като списък от последователни състояния

- Фронтът на търсенето е списък от пътища (отначало е списък от един път, който включва само началното състояние).
- При търсенето в дълбочина фронтът се обработва като стек.
- Ако фронтът има вида [p1, p2, … , pn], то:
  - Избира се p1. Ако p1 е довел до целта, ***край***.
  - Ацикличните пътища $p1_1$, $p1_2$, … , $p1_k$, които разширяват (продължават) p1, се добавят в началото на фронта (преди p2), т.е. фронтът придобива вида [$p1_1$, $p1_2$, … , $p1_k$, p2, … , pn]. Преминава се към следващата стъпка от цикъла.

Вариант 1: търсене в дълбочина до определено ниво (depth-bound search, depth-limited search).



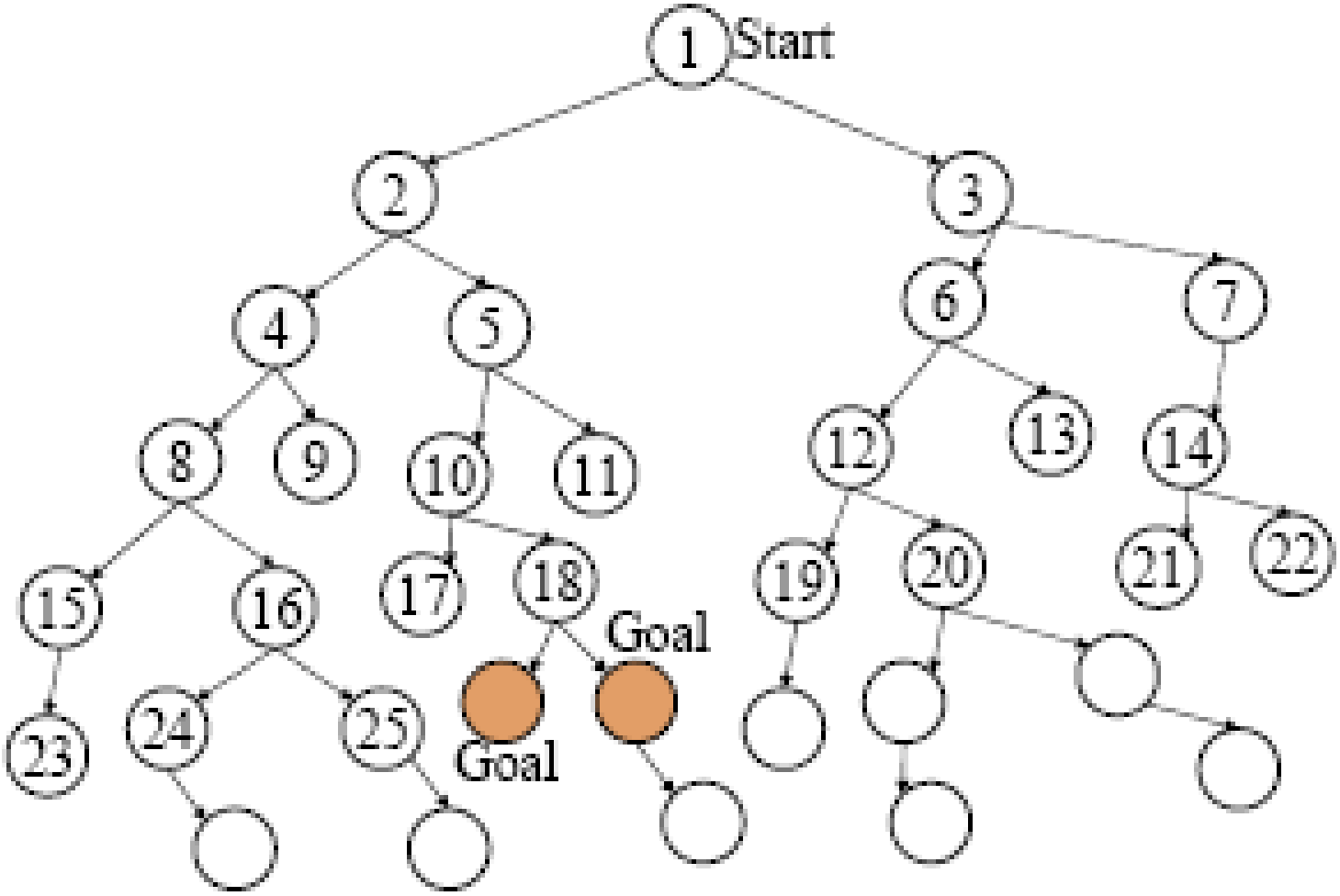
Вариант 2: итеративно търсене по нива (iterative deepening).



2. **Изчерпване (търсене) в широчина** (***breadth-first search***) - добавяне на новите възли в края на списъка Open/fringe.

fringe = **опашка** (FIFO, put successors at end)

Търсенето е пълно (когато всеки възел от ГС има краен брой наследници) и оптимално, но е скъпо (експоненциално по отношение на необходимата памет).


1
Start
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
Goal
23
24
25
Goal

Построяване на пътя като списък от последователни състояния

- Фронтът на търсенето е списък от пътища (отначало е списък от един път, който включва само началното състояние).
- При търсенето в широчина фронтът се обработва като опашка.
- Ако фронтът има вида [p1, p2, … , pn], то:
  - Избира се p1. Ако p1 е довел до целта, ***край***.
  - Ацикличните пътища $p1_1$, $p1_2$, … , $p1_k$, които разширяват (продължават) p1, се добавят в края на фронта (след pn), т.е. фронтът придобива вида [p2, … , pn, $p1_1$, $p1_2$, … , $p1_k$]. Преминава се към следващата стъпка от цикъла.

Вариант: търсене с равномерна цена на пътя (uniform-cost search) - сортиране на списъка Open/fringe според цената на изминатия път.

Търсенето в широчина е частен случай на търсене с равномерна цена на пътя.

Оценка на сложността. Пространствената сложност (сложността по памет) на търсенето в дълбочина е O(bm), докато пространствената сложност на търсенето в широчина е O(b^{d+1}). Времевата сложност на търсенето в дълбочина е O(b^m), а времевата сложност на търсенето в широчина е O(b^{d+1}). При задачи, които имат много решения, търсенето в дълбочина може да бъде по-бързо от търсенето в широчина, тъй като има добър шанс да се намери решение след изследване на малка част от цялото ПС.

Търсенето в ограничена дълбочина (търсенето в дълбочина до определено ниво) има сложност, подобна на тази на търсенето в дълбочина (изисква $O(b^l)$ време и $O(bl)$ памет, където l е максималната дълбочина на изследване). Итеративното търсене по нива има времева сложност $O(b^d)$ и пространствена сложност $O(bd)$, където d е максималната дълбочина на изследване. Това е най-предпочитаният метод за търсене, когато ПС е с голям размер и няма никаква информация за дълбочината на решението.

Резюме на алгоритмите за неинформирано търсене

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

Означения:

b – коефициент на разклонение на ГС

d – дълбочина на най-плитката цел

m – максимална дълбочина на ГС (може да бъде ∞)

l – максимална дълбочина на изследване

*C** – цена на оптималния път (предполага се, че всички стъпки имат цена ≥ ε, ε>0)