



# »Лекционен курс »Изкуствен интелект



# Неинформирано (Сляпо) търсене

- » Стратегиите използват само необходимата информация за представяне на проблема
  - > **Всичкото**, което могат да правят е генериране на наследници и различаване на целево състояние от нецелеви
  - > Отделните стратегии се различават по **последователността**, в която се разширяват възлите



# Методи

## » Три базови метода:

- > Търсене в широчина
- > Търсене с еднакви разходи
- > Търсене в дълбочина
  - + Лимитирано търсене в дълбочина

# Търсене в широчина

- » Първо се разширява началния възел (корена)
  - > След него всички възли генерирани от корена и т.н.
  - > Разширяват се всички възли на едно ниво, преди да се премине към следващото ниво
- » Инстанция на генетичния Graph-Search алгоритъм, където **най-плиткият** неразширен възел се избира за разширение
- » Намира най-плитките решения
- » Опашката за граничните възли
  - > **FIFO**
  - > Новите („по-дълбоки“) възли отиват в края на опашката



# Bread-first-search

```
function Breadth-First-Search (problem) returns решение или грешка
  node ← възел със State = problem.Initial-State; path-Cost = 0;
  if problem.Goal-Test(node.State) then return Solution(node);
  frontier ← една FIFO опашка с node като единствен елемент;
  explored ←  $\emptyset$ ;
  loop do
    if Empty(frontier) then return грешка;
    node ← Pop(frontier); /* избира най-плиткия възел от frontier */
    for each action in problem.Actions(node.State) do
      child ← Child-Node(problem, node, action);
      if (child.State  $\notin$  explored)  $\vee$  (child.State  $\notin$  frontier) then
        if problem.Goal-Test(child.State) then return Solution(child);
        frontier ← Insert(child, frontier);
  end do
```

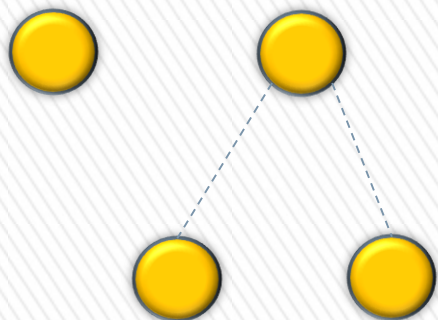
- В сравнение с генетичния алгоритъм има малко подобрене: целевият тест се прилага, когато се генерира един възел, а не когато е избран за разширение
- Премахва възлите, съдържащи се в множеството на граничните или изследваните възли



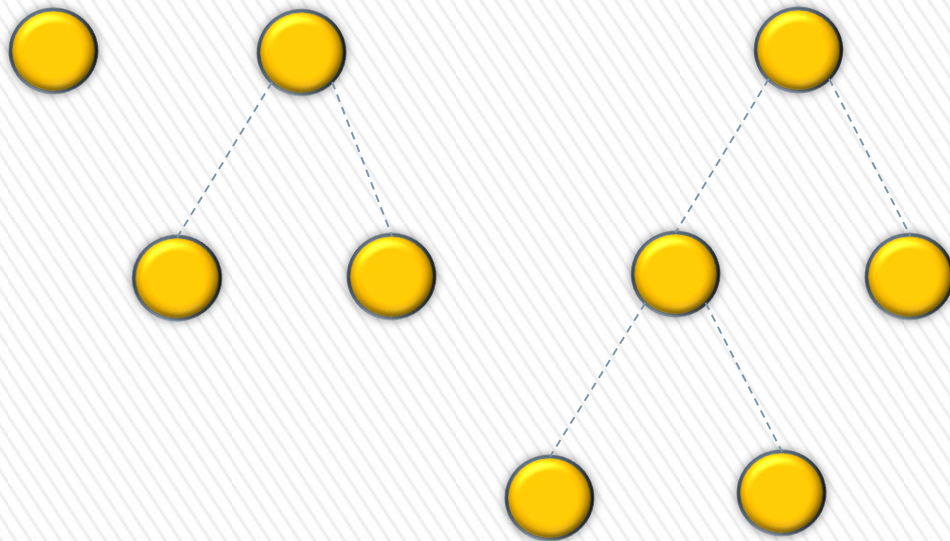
# Пример



# Пример

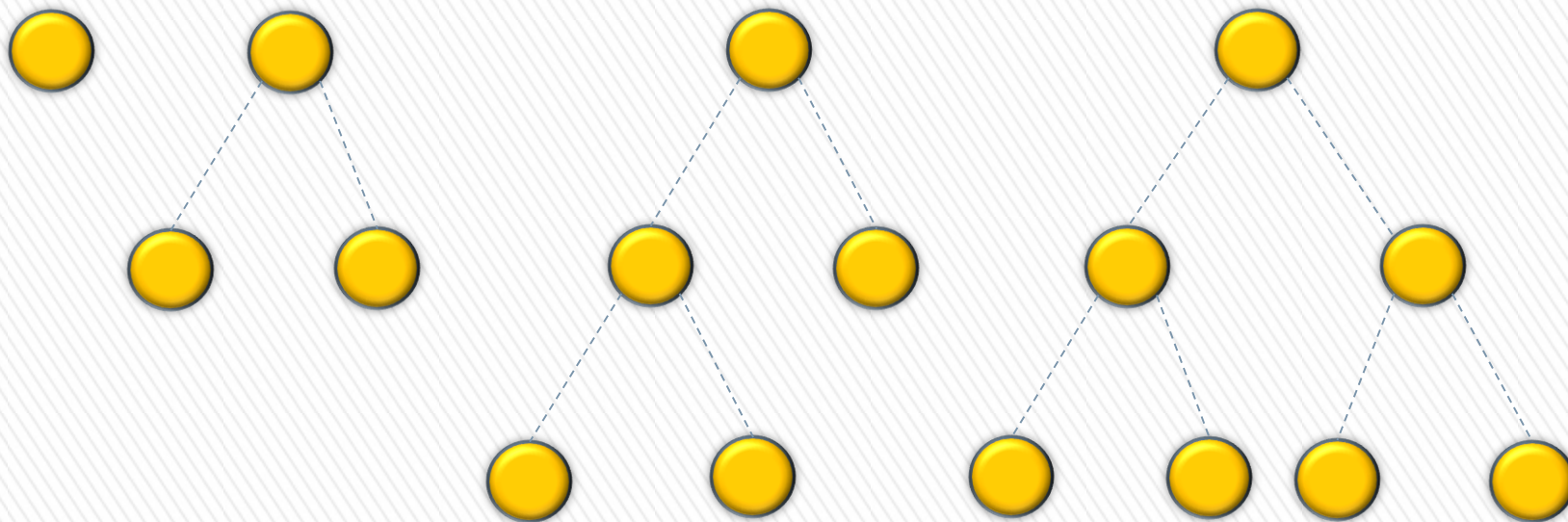


# Пример





# Пример



# Оценка на алгоритъма

## » Пълен:

- > Когато **най-плиткият** целеви възел се намира на **крайна** дълбочина  $d$ , алгоритъмът го намира, след като е генерирал всичките по-плитки възли

## » Оптимален:

- > Най-плиткият възел **не е** непременно най-оптималният
- > Търсенето в широчина е оптимално, когато разходите за пътя представят ненамаляваща функция на дълбочината на възела

## » Времева и паметна комплексност:

- > При  $b$  наследника на всяко състояние на дълбочина  $d$ :  $O(b^{d+1})$
- > Проблемът с паметта е **най-сериозен** (по-сериозен от този с времето)



# Пример

- $b = 10$
- 1 възел = 1000 байта

*Големият проблем е  
паметта !*

d	Възли	Време	Памет
2	1100	0.11 мсек	107 килобайта
4	111100	11 мсек	10.6 мегабайта
6	$10^7$	1.1 сек	1 гигабайт
8	$10^9$	2 мин	103 гигабайта
10	$10^{11}$	3 ч	10 терабайта
12	$10^{13}$	13 ден	1 петабайт
14	$10^{15}$	3.5 год	99 петабайт
16	$10^{16}$	350 год	10 ексабайта

# Търсене с еднакви разходи

- » Търсенето в широчина е **оптимално**, когато всички разходи за отделните стъпки са еднакви
  - > Понеже винаги разширява **най-плиткия** неразширен възел
- » С едно просто разширение имаме алгоритъм, който е оптимален за всяка функция на разходите за стъпки
  - > Вместо да разширява най-плиткия възел, търсенето с еднакви разходи разширява възела  $n$  с най-малки разходи за път  $g(n)$
  - > Граничните възли се съхраняват като опашка с приоритети, сортирани по  $g$



# Uniform-cost-search

```
function Uniform-Cost-Search (problem) returns решение или грешка
  node ← възел със State = problem.Initial-State;
  path-Cost = 0;
  frontier ← една приоритетна опашка, сортирана по Path-Cost;
  explored ←  $\emptyset$ ;
  loop do
    if Empty(frontier) then return грешка;
    node ← Pop(frontier); /* избира възел с най-малки разходи от frontier */
    if problem.Goal-Test(node.State) then return Solution(node);
    node.State добавяме към explored;
    for each action in problem.Actions(node.State) do
      child ← Child-Node(problem, node, action);
      if (child.State  $\notin$  explored)  $\vee$  (child.State  $\notin$  frontier) then
        frontier ← Insert(child, frontier)
      else if child.State  $\in$  frontier (с по-висок Path-Cost) then
        заместваме този frontier възел с child
    end do
```

# Разлики с търсене в широчина

» Освен сортирането, още две разлики с търсенето в широчина:

- > Целевият тест се прилага при избора на възела за разширение (както при оригиналния генетичен алгоритъм), а не когато се създава възел
  - + Понеже първият генериран целеви възел може да лежи на субоптимален път
- > Допълнителен тест за случая, когато е намерен по-добър път за един моментно намиращ се в границата възел

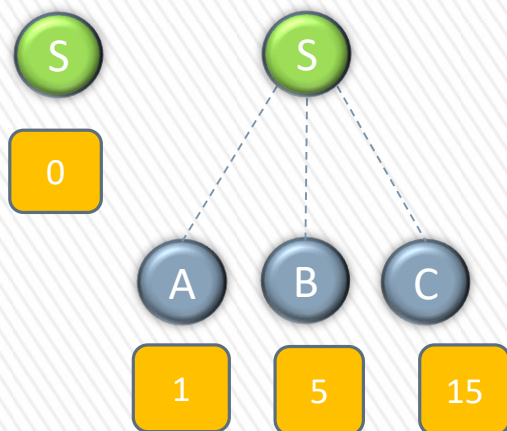




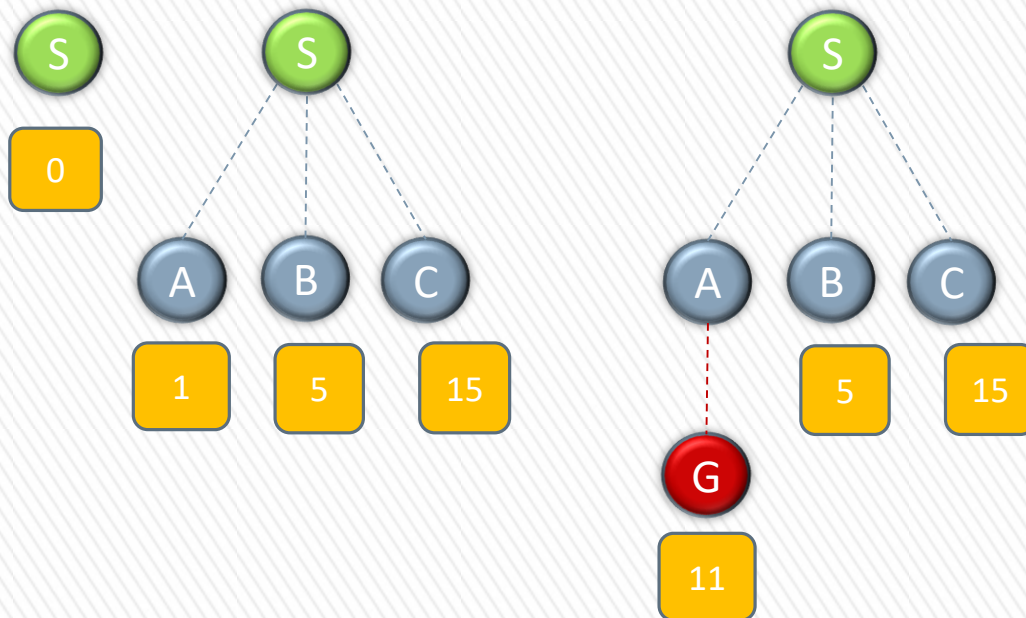
# Пример



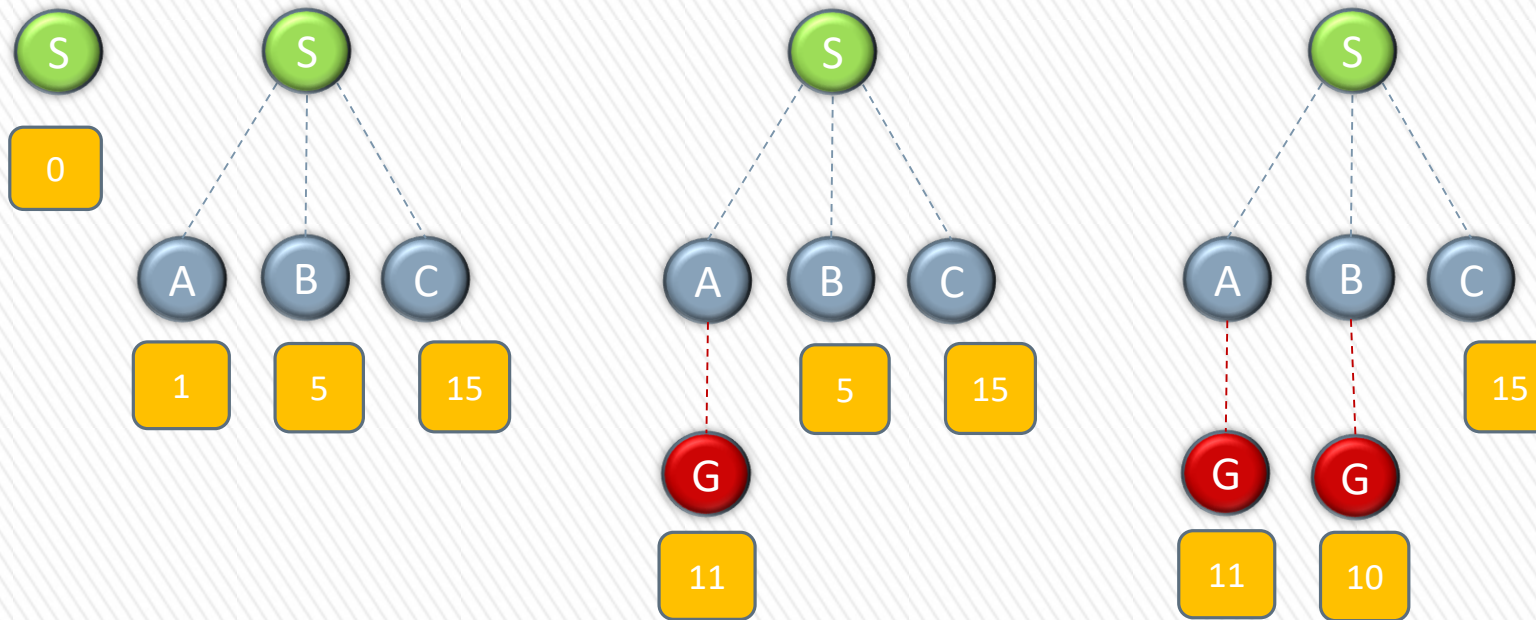
# Пример



# Пример



# Пример



# Търсене първо в дълбочина

- » Винаги разширява „най-дълбокия“ възли в актуалната гранична област на дървото за търсене
  - > Инстанция на генетичния Graph-Search алгоритъм, където се използва LIFO структура за съхраняване на границата
  - > Последно генерираният възел се избира за разширение
- » Имплементира се с рекурсивна функция

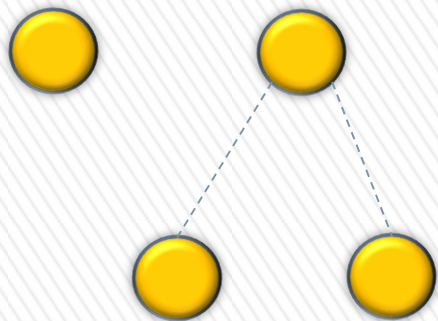


# Пример

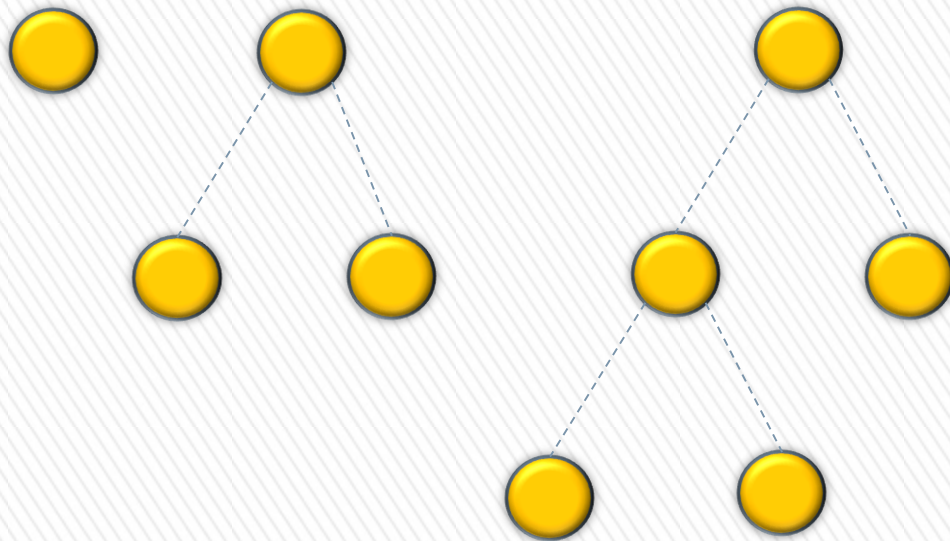




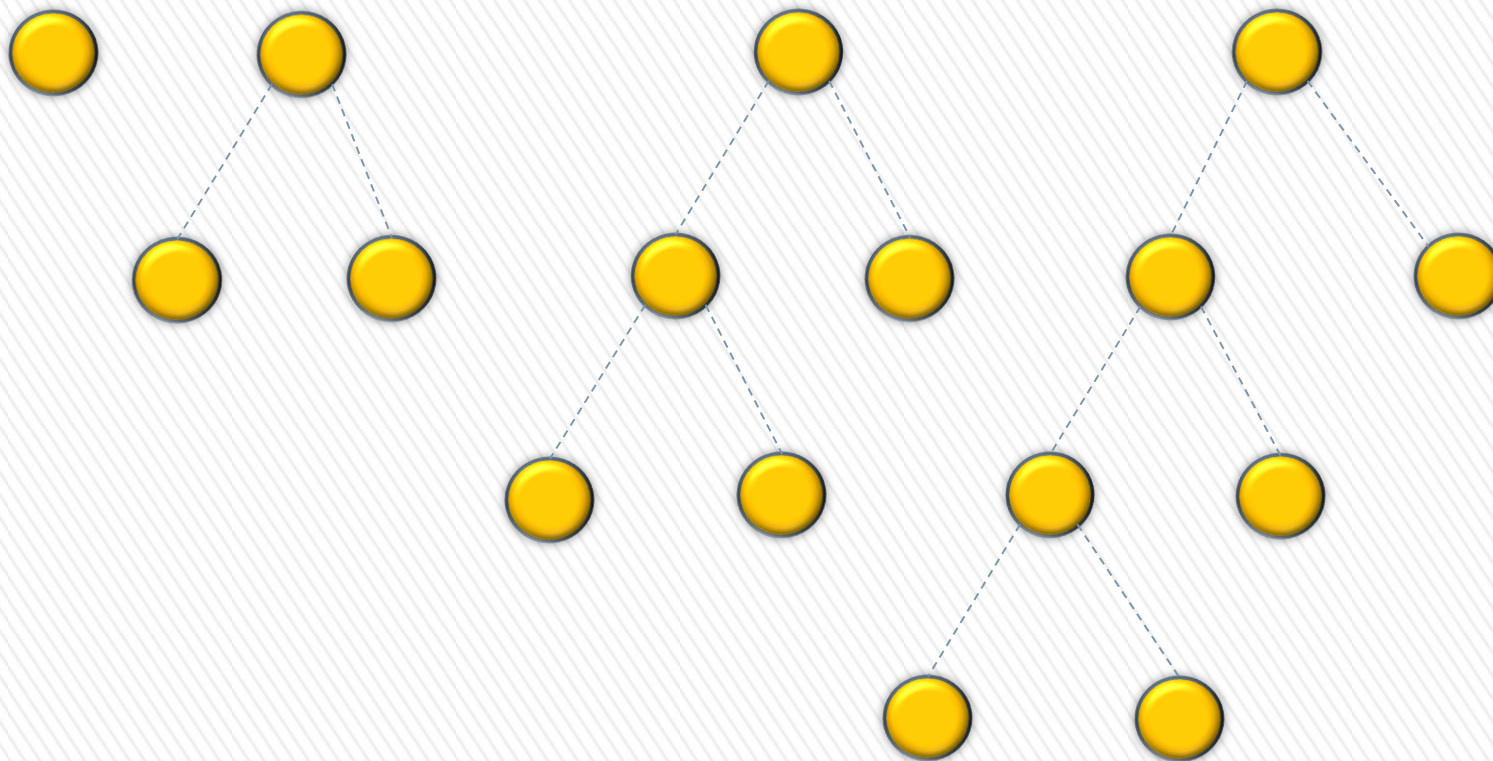
# Пример



# Пример



# Пример



# Оценка на алгоритъма

- » Пълен – трябва да се различава за каква структура се прилага:
  - > При графи – пълен в крайни ПС
  - > При дървета – непълен
- » Оптимални: и за двете структури не е оптимален
- » Времева и паметна комплексност: основното предимство на алгоритъма
- » Вариант на търсенето в дълбочина: търсене с възврат
  - > Изисква още по-малко памет



# Depth-limited-search

```
function Depth-Limited-Search (problem, limit) returns решение или грешка/cutoff  
  return Recursive-DLS(Make-Node(problem, Initial-State), problem, limit)
```

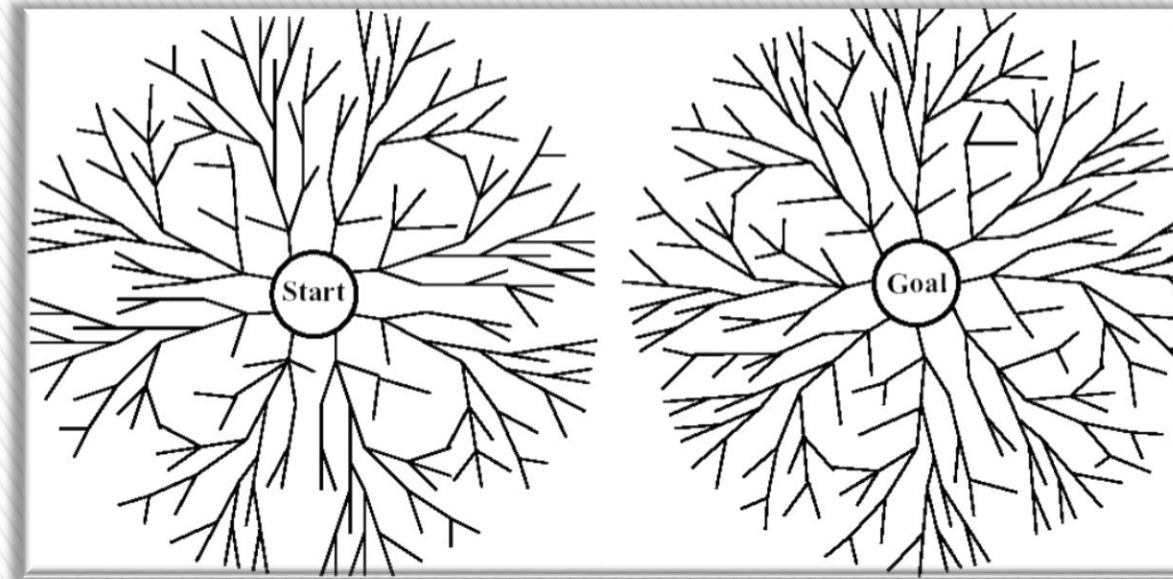
```
function Recursive-DLS(node, problem, limit) return решение или грешка/cutoff  
  if problem.Goal-test(node.State) then return Solution(node)  
  else if limit = 0 then return cutoff  
  else  
    cutoff occurred?  $\leftarrow$  false;  
    for each action in problem.Actions(node.State) do  
      child  $\leftarrow$  Child-Node(problem, node, action);  
      result  $\leftarrow$  Recursive_DLS(child, problem, limit -1);  
      if result = cutoff then cutoff occurred?  $\leftarrow$  true;  
      else result  $\neq$  failure then return result  
  if cutoff occurred? then return cutoff else return failure
```

Търсенето може да завърши с два типа грешки:

- *failure* – няма решение
- *cutoff* – няма решение в ограничението за дълбочина

# Бидирекционално в широчина

- » Започвайки от старта и целта „паралелно“ търсене до срещане



*Разход: от двете страни половин дълбочина*







Благодаря за вниманието!