



»Лекционен курс »Интелигентни системи



Неинформирано
търсене



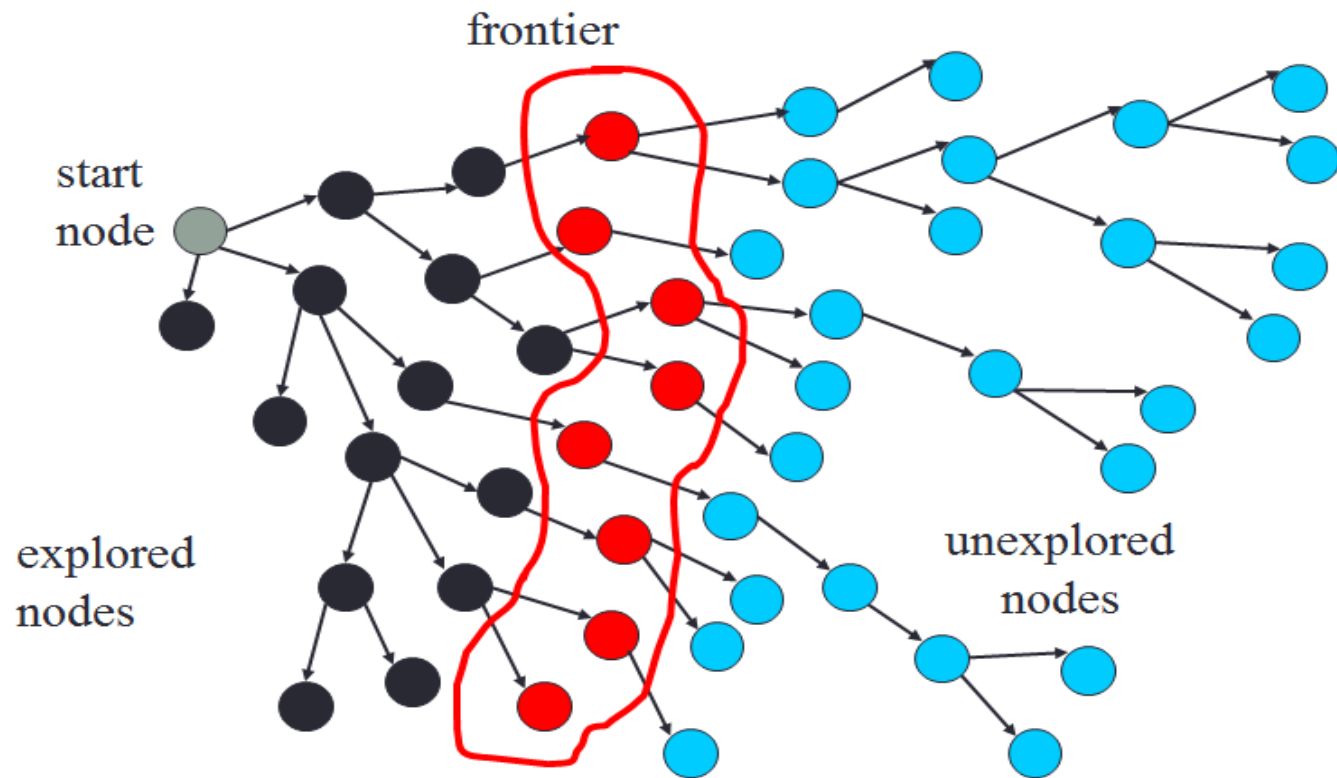
Неинформирано (Сляпо) търсене

- » 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



Общ алгоритъм за търсене:

- даден е граф
- начални състояния
- целеви състояния
- последователно да се изследват пътищата от началните състояния
- Поддържа се фронт/граница (frontier) от пътищата, които са били изследвани
- По време на процеса на търсене фронта се разширява в посока към неизследваните възли, докато се достигне до целеви възел



Предполагаме, че след като алгоритъма за търсене намери един път, може да му бъде зададено да търси още решения и тогава процесът трябва да продължи.

Начинът, по който фронта се разширява и това точно коя стойност от фронта се избира дефинира стратегията на търсене (search strategy).



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



Търсене в широчина

- » При търсенето в ширина фронтът се обработва като опашка
- » Ако фронта е $[p_1, p_2, \dots, p_n]$
 - > избира се p_1
 - > пътищата $p_1', p_1'', \dots, p_1^{(k)}$, които разширяват p_1 се добавят в края на опашката (след p_n), т.е. $[p_2, \dots, p_n, p_1', p_1'', \dots, p_1^{(k)}]$ и се обработват едва след като всички пътища p_2, \dots, p_n , се изследват
- » Намира най-краткия път



Bread-first-search

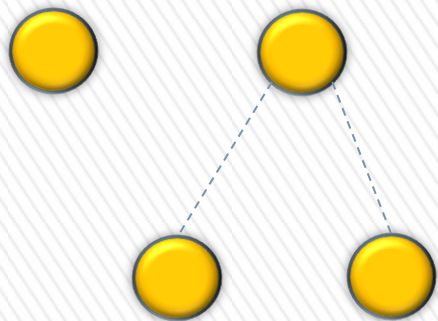
```
function Breadth-First-Search (problem) returns решение или грешка
  node  $\leftarrow$  възел със State = problem.Initial-State; path-Cost = 0;
  if problem.Goal-Test(node.State) then return Solution(node);
  frontier  $\leftarrow$  една FIFO опашка с node като единствен елемент;
  explored  $\leftarrow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

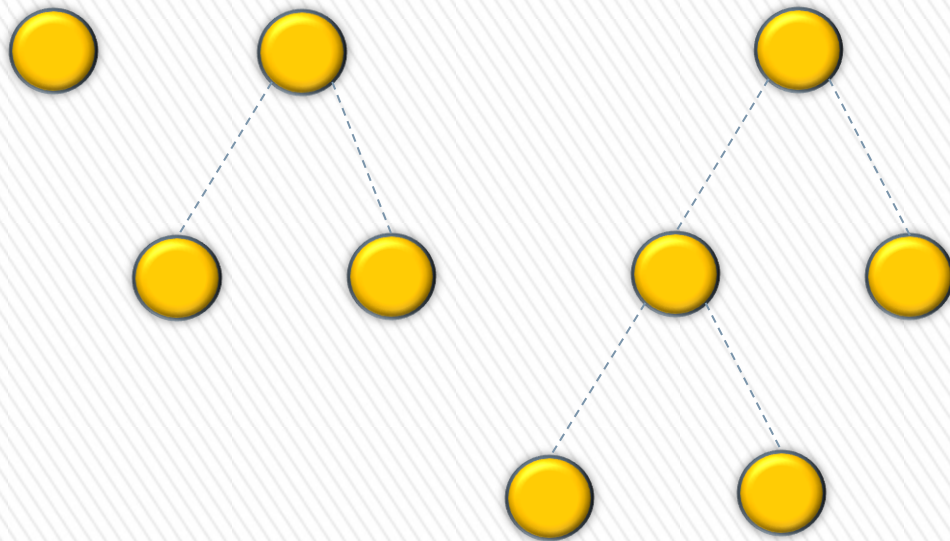
Пример



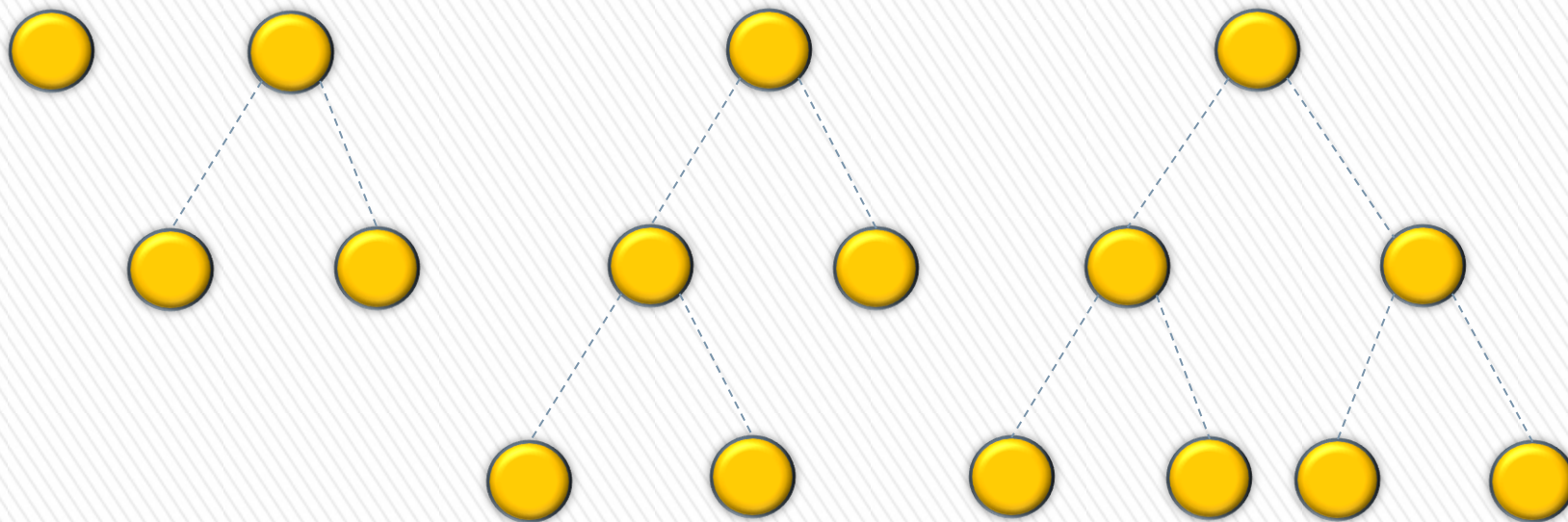
Пример



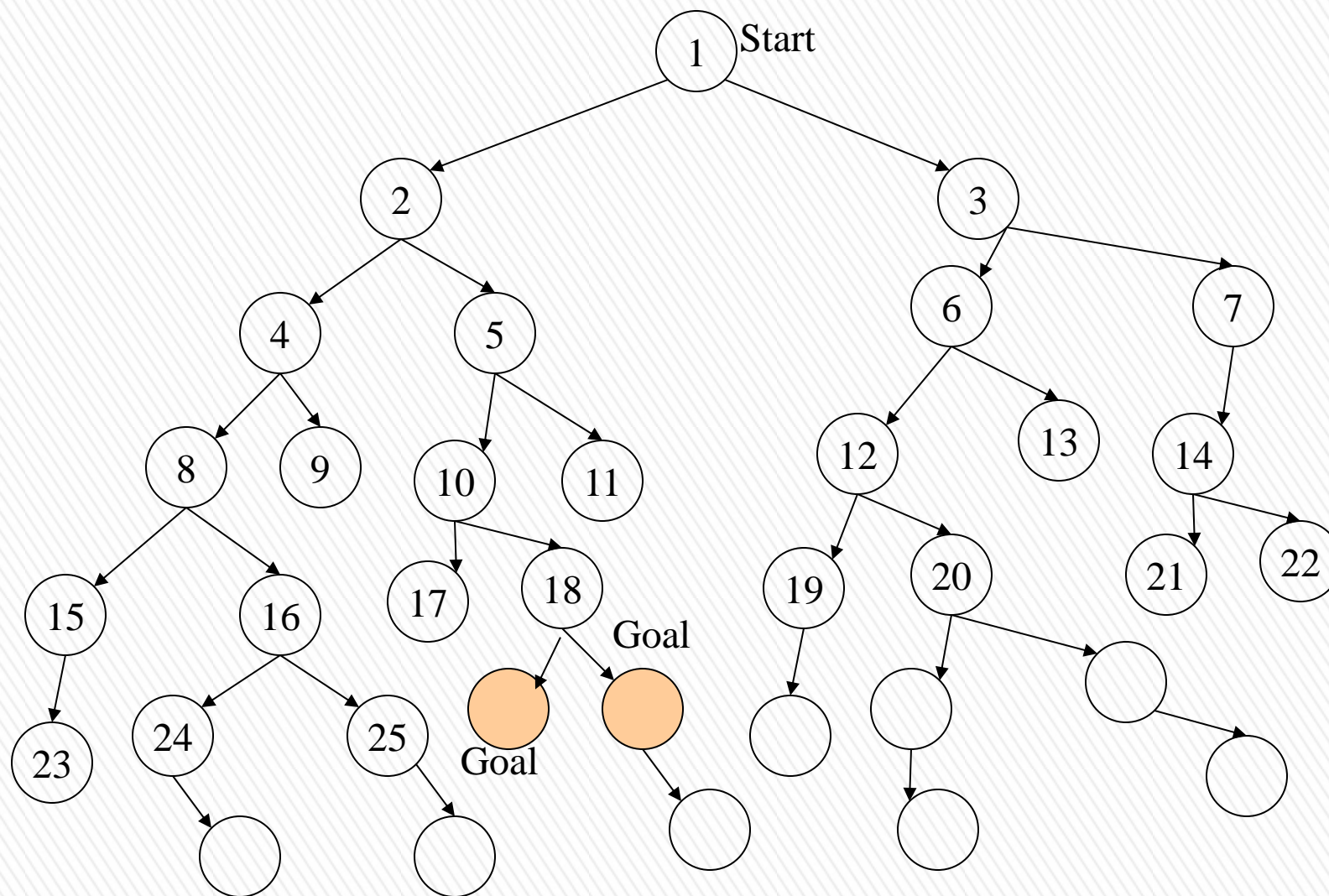
Пример



Пример



Търсене в широчина



Търсене в широчина

- » При търсенето в ширина фронтът се обработва като **опашка**
- » Ако фронта е $[p_1, p_2, \dots, p_n]$
 - > избира се p_1
 - > пътищата $p_1', p_1'', \dots, p_1^{(k)}$, които разширяват p_1 се добавят в края на опашката (след p_n), т.е. $[p_2, \dots, p_n, p_1', p_1'', \dots, p_1^{(k)}]$ и се обработват едва след като всички пътища p_2, \dots, p_n , се изследват
- » Намира най-краткия път



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  $\leftarrow$  възел със State = problem.Initial-State;
  path-Cost = 0;
  frontier  $\leftarrow$  една приоритетна опашка, сортирана по Path-Cost;
  explored  $\leftarrow \emptyset$ ;
  loop do
    if Empty(frontier) then return грешка;
    node  $\leftarrow$  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  $\leftarrow$  Child-Node(problem, node, action);
      if (child.State  $\notin$  explored)  $\vee$  (child.State  $\notin$  frontier) then
        frontier  $\leftarrow$  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:

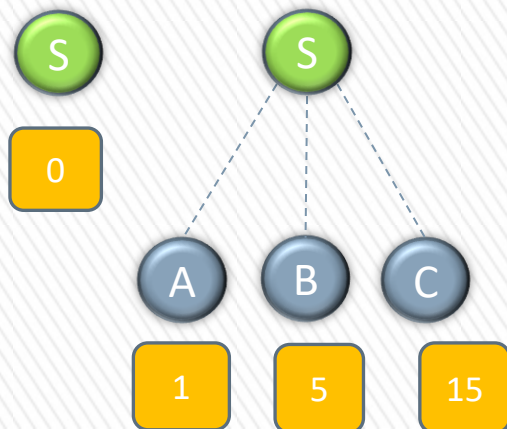
- > Целевият тест се прилага при избора на възела за разширение (както при оригиналния генетичен алгоритъм), а не когато се създава възел
 - + Понеже първият генериран целеви възел може да лежи на субоптимален път
- > Допълнителен тест за случая, когато е намерен по-добър път за един моментно намиращ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



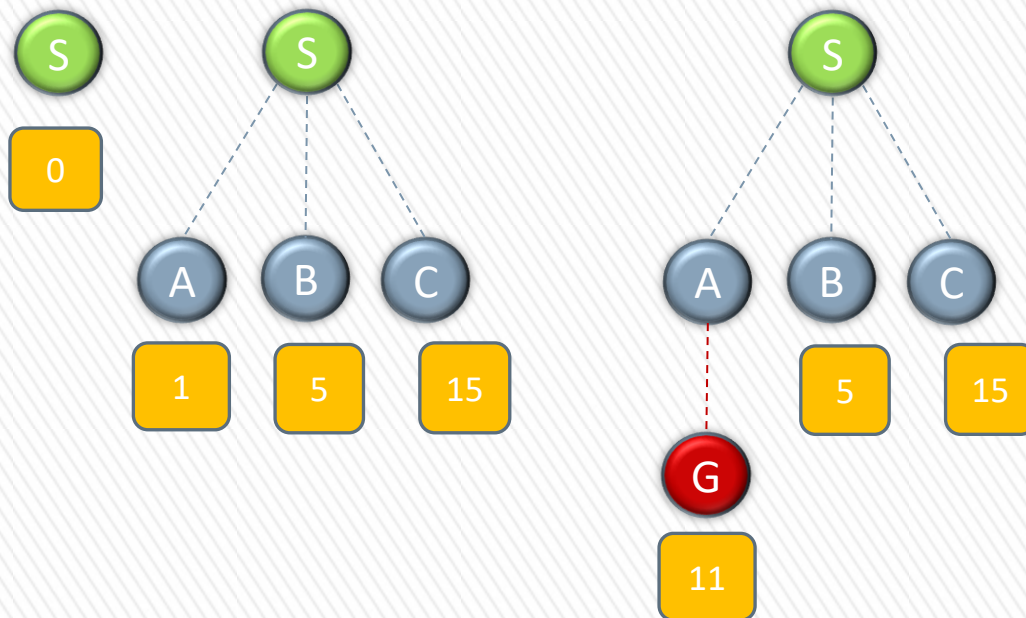
Пример



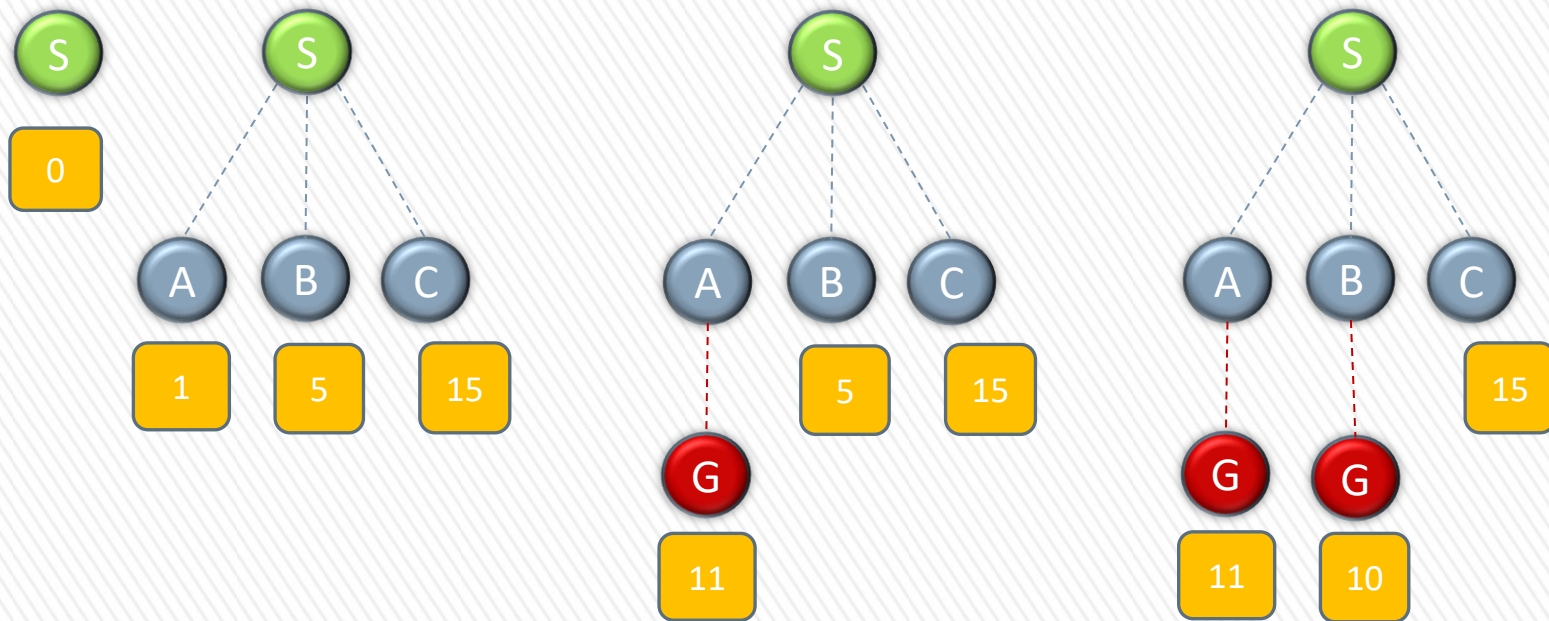
Пример



Пример



Пример



Търсене първо в дълбочина

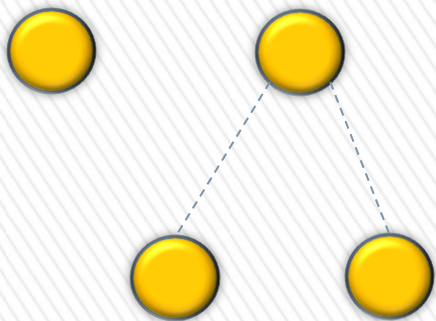
- » Винаги разширява „най-дълбокия“ възел в актуалната гранична област на дървото за търсене
 - > Инстанция на генетичния Graph-Search алгоритъм, където се използва LIFO структура за съхраняване на границата
 - > Последно генерираният възел се избира за разширение
- » Имплементира се с рекурсивна функция



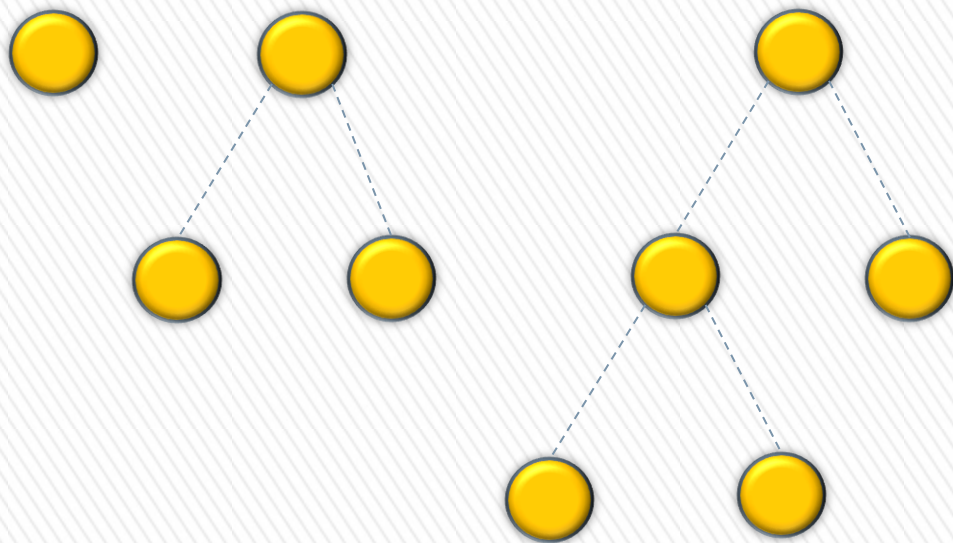
Пример



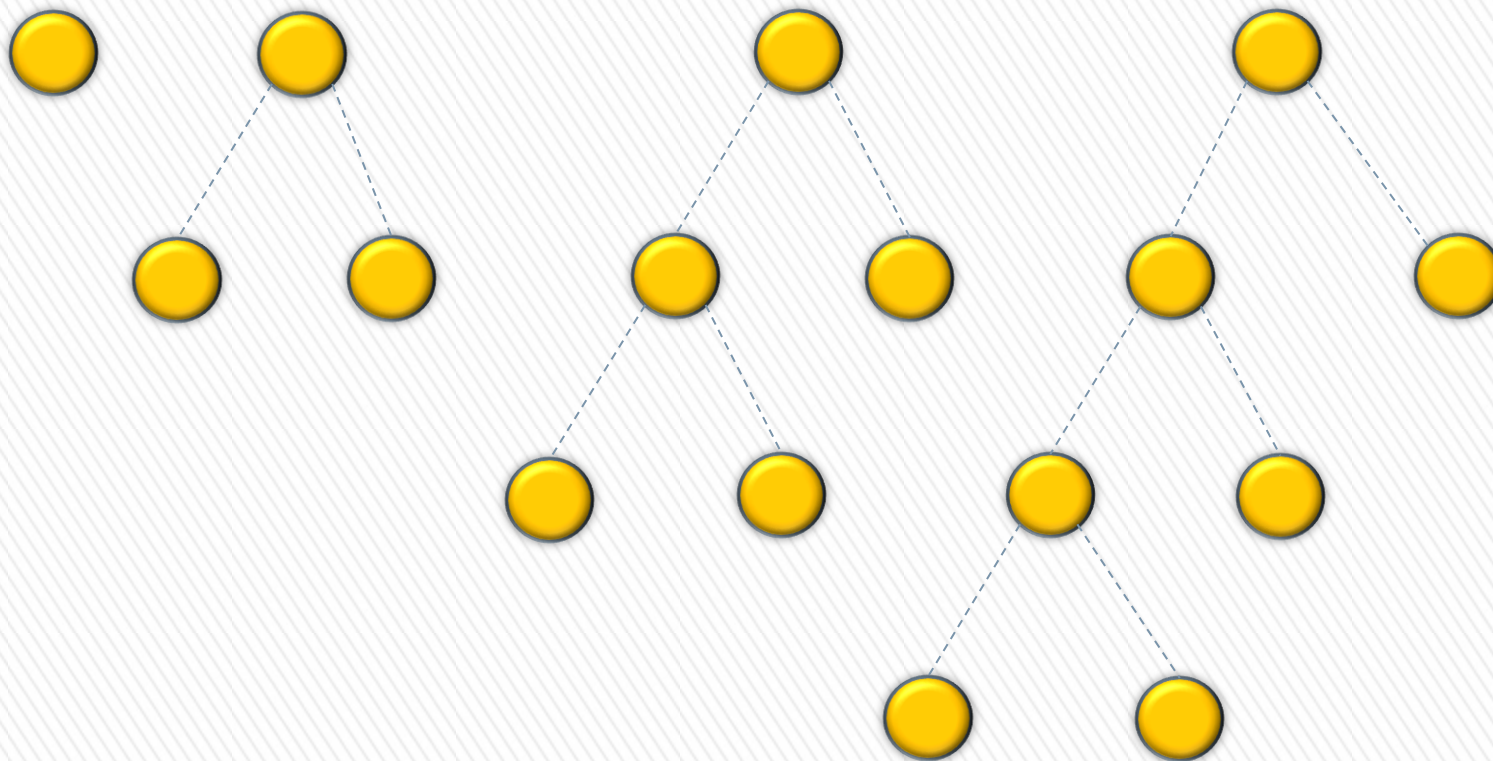
Пример



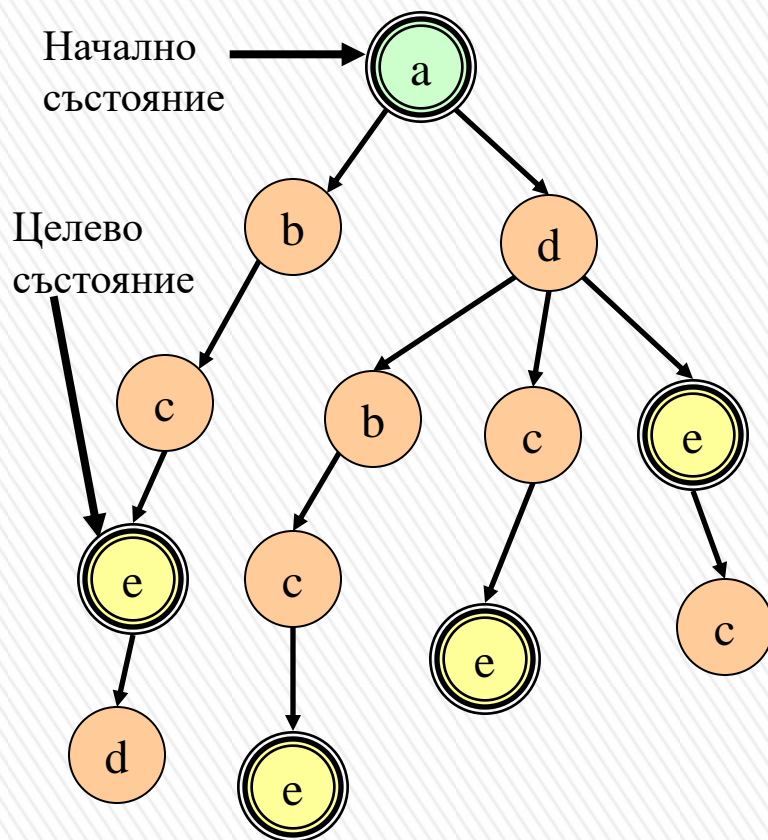
Пример



Пример



Представяне на фронта при Търсене в дълбочина



[[a]]

[[b a] [d a]]

[[c b a] [d a]]

[[e c b a] [d a]]

I-во решение:

[e c b a] => [a b c e]

Преудовлетворяване:

[[d a]]

[[b d a] [c d a] [e d a]]

[[c b d a] [c d a] [e d a]]

[[e c b d a] [c d a] [e d a]]

II-ро решение:

[e c b d a] => [a d b c e]

Преудовлетворяване:

[[c d a] [e d a]]

[[e c d a] [e d a]]

III-то решение:

[e c d a] => [a d c e]

Преудовлетворяване:

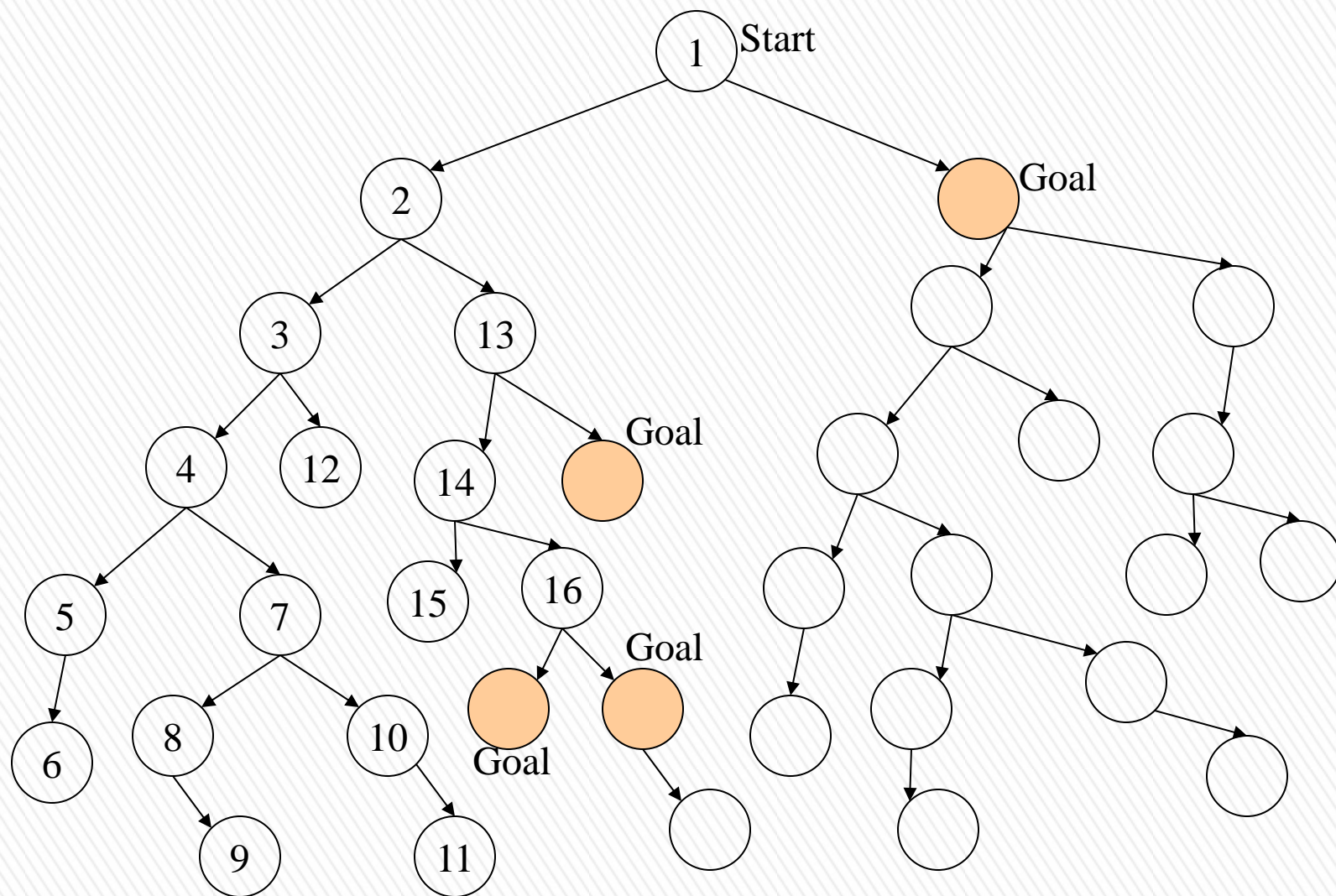
[[e d a]]

IV-то решение:

[e d a] => [a d e]



Търсене в дълбочина



Търсене в дълбочина

- » При търсенето в дълбочина фронтът се обработва като стек
- » Ако фронта е $[p_1, p_2, \dots]$
 - > избира се p_1
 - > пътищата $p_1', p_1'', \dots, p_1^{(k)}$, които разширяват p_1 се добавят в началото на стека (преди p_2), т.е. $[p_1', p_1'', \dots, p_1^{(k)}, p_2, \dots]$
 - > p_2 се обработва едва след като всички пътища, които са продължение на p_1 са били изследвани



Оценка на алгоритъма

- » Пълен – трябва да се различава за каква структура се прилага:
 - > При графи – пълен в крайни ПС
 - > При дървета – непълен
- » Оптимален: 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)
```

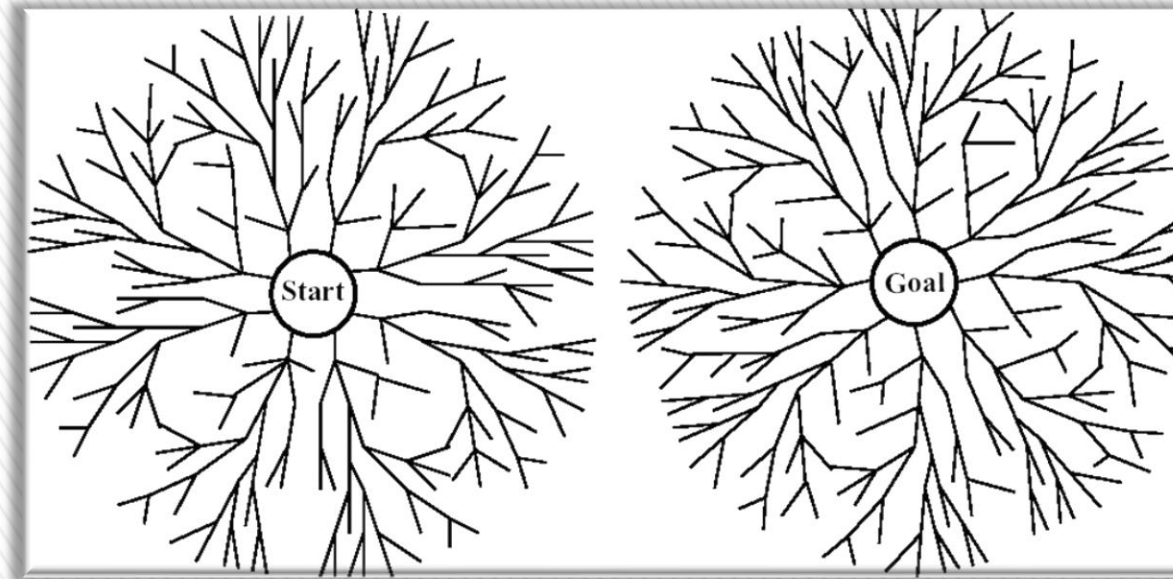
```
function Recursive-DLS(node, problem, limit) return решение или грешка/cutoff  
  if problem.Goal-test(node.State) then return Solution(node)  
  else if limit = 0 then return cutoff  
  else  
    cutoff occurred?  $\leftarrow$  false;  
    for each action in problem.Actions(node.State) do  
 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



Регистрация

`https://tinyurl.com/2ahxod2t`





Благодаря за вниманието!