



»Лекционен курс



»Интелигентни системи

Търсене на решения > 1

Дърво на търсене

- » След като е формулиран един **проблем**, той трябва да бъде **решен**
- » Понеже решенията са последователност от действия, алгоритмите за търсене (АТ) разглеждат различни възможни **последователности от действия** (ПД)
- » Дърво на търсене - възможните ПД, започващи от началното състояние, където:
 - > Възли – състояния от РС на проблема
 - > Ребра - действия
 - > Корен – началното състояние



Обща характеристика на търсене

» Търсенето започва от **началното състояние**

- > Първо се тества дали това е целево състояние

» **Разширяване** на актуалното състояние

- > Прилагане на всяко възможно действие върху актуалното състояние
- > Генериране ново множество от състояния
- > Въвеждаме нови ребра в дървото на търсене

» Същност на търсене

- > Една опция се преследва по-нататък, докато другите се остават за следващо проследяване, ако първото не доведе до решение



Пример



Пример



Пример



Гранични възли

» Възел-лист

- > Няма възли наследници

» Гранични възли

- > Множеството на всички възли, налични за разширение в определен момент от процеса на търсене
- > Разширяването на граничните възли продължава дотогава, докато е намерено решение или няма повече състояние за разширяване



Tree-search

```
function Tree-Search (problem) returns решение или грешка
  Инициализиране граничните възли с началното състояние на problem;

  loop do
    if (гранично множество ==  $\emptyset$ ) then return грешка;
    Избира един възел-лист и го отстранява от граничното множество;
    if (възелът съдържа целево състояние) then return съответно решение;
    Разширяване възела;
    Добавяне на резултиращите възли към граничното множество;
  end do
```

- Това е обща генетична структура за всички алгоритми за търсене
- Алгоритмите се различават по използваната стратегия за търсене, т.е. избор на следващо за разширяване състояние

Особеност



Циклични пътища

- » Цикличните пътища правят пълното дърво на търсене **безкрайно**, въпреки че пространството на търсене се състои само от няколко десетки състояния
 - > Понеже няма граници за броя пъти за преминаване през цикъла
- » Циклите могат да причинят определени алгоритми да не успяват и дори решими проблеми да станат нерешими
- » За радост, обикновено не е необходимо да разглеждаме циклични пътища
 - > Понеже разходите за път са сумарни и разходите за отделните стъпки са положителни стойности, следва че един цикличен път до едно състояние никога не е по-добър от същия път с премахнат цикъл
- » Цикличните пътища са специален случай на общата концепция за **излишни пътища**
 - > Съществуват, когато има повече от един път



Особеност



Избягване на излишни пътища

- » В определени случаи е възможно проблемът да бъде поначало така дефиниран, че да избягва излишни пътища
 - > Напр., 8-те дами
- » В други случаи излишните пътища са **неизбежни**
 - > Напр., всички проблеми, където действията са обратими
 - + Търсене на маршрути, игрите с подреждане на блокчета



Решение на повторението

- » Алгоритмите, които забравят проследяването, са „прокълнати“ да го повтарят
- » Когато забелязваме, къде сме били, можем да избегнем изследването на излишни пътища
- » За целта можем да разширим Tree-Search алгоритъма с една структура данни, наречена **изследвано множество**, което помни разширените възли
 - > Новосъздадените възли, които съвпадат с вече съществуващите възли (в изследваното множество или граничното множество) могат да бъдат премахвани, вместо добавяни към граничното множество



Graph-search

```
function Graph-Search (problem) returns решение или грешка
  Инициализиране граничните възли с началното състояние на problem;
  Изследвано множество =  $\emptyset$ ;
  loop do
    if (гранично множество ==  $\emptyset$ ) then return грешка;
    Избира един възел-лист и го отстранява от граничното множество;
    if (възелът съдържа целево състояние) then return съответно решение;
    Добавяне възела към изследваното множество;
    Разширяване възела;
    Добавяне на резултиращите възли към граничното множество
      (само ако не се съдържат в изследваното множество);
  end do
```

- Конструираното дърво на търсене съдържа максимално по един екземпляр за всяко състояние
- Дървото може да се разширява директно върху графа на ПС

Граница като разделител

» Алгоритъмът Graph-Search има още едно приятно свойство:

границата е един разделител

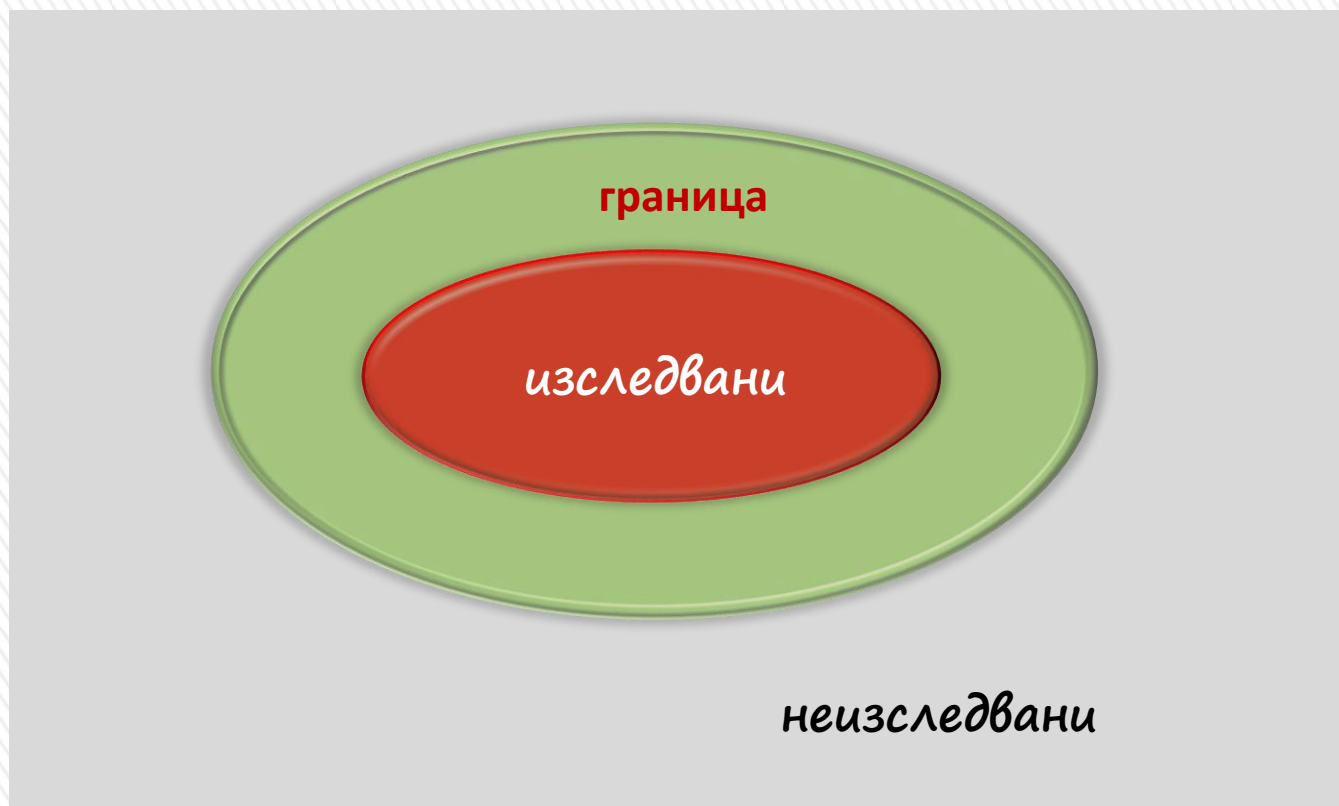
- > Разделя графа на състоянията на две области:
 - + Изследвани състояния
 - + Неизследвани състояния
- > Всеки път от началното състояние към едно неизследвано състояние трябва да минава през едно състояние от границата

» Всяка стъпка в процеса на търсене:

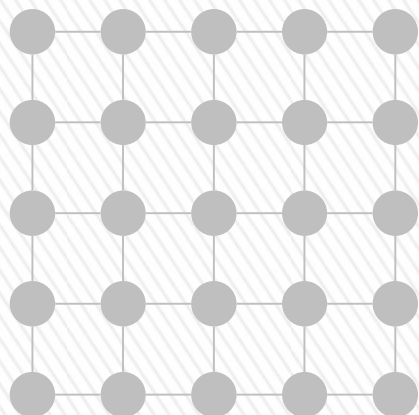
- > Премества едно състояние от границата в областта на изследваните състояния
- > Премества някои неизследвани състояния в границата
- > Така алгоритъмът изследва систематично и последователно състоянията в ПС докато намери решение



Граница като разделител



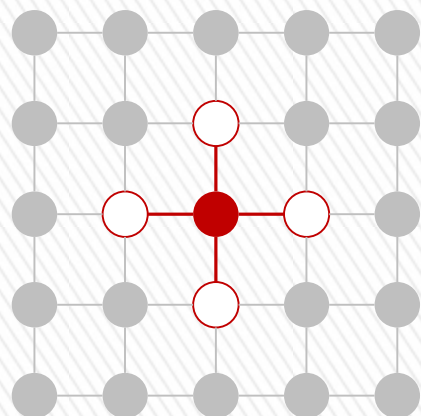
Пример



„Правоъгълна решетка“

- Особено важен пример за компютърните игри
- Всяко състояние има 4 наследника
- Дърво на търсене с дълбочина d , включващо повторяеми състояния, има 4^d листа
- Само $2d^2$ различни състояния за d стъпки
- За $d = 20$, около 1 милиард възли, но само 800 различни състояния
- Проследяването на излишни пътища може да превърне един решим в нерешим проблем

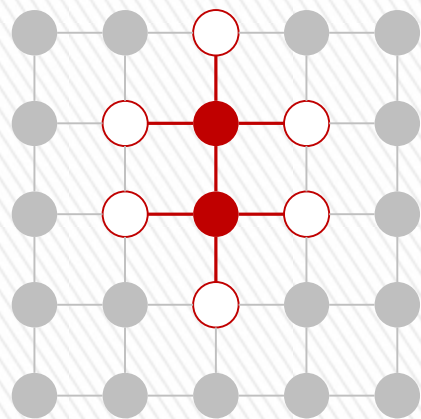
Пример



„Правоъгълна решетка“

- Граница (белите възли) винаги разделя изследвана област (червените възли) от неизследваната област (сивите възли)

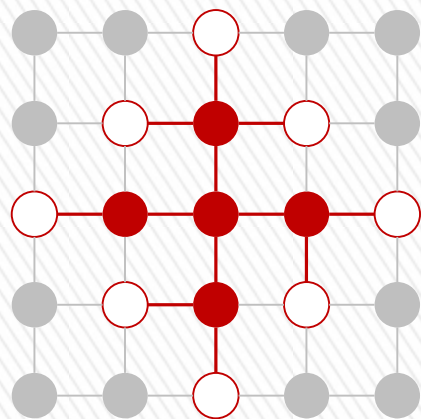
Пример



„Правоъгълна решетка“

- Граница (белите възли) винаги разделя изследвана област (червените възли) от неизследваната област (сивите възли)

Пример



„Правоъгълна решетка“

- Граница (белите възли) винаги разделя изследвана област (червените възли) от неизследваната област (сивите възли)

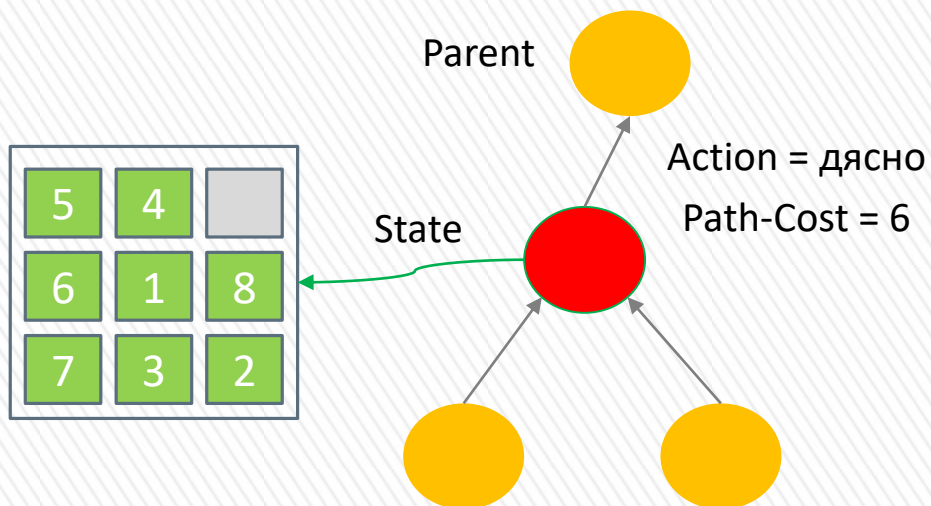
Инфраструктура на алгоритмите

- » За да могат да проследяват конструирането на дървото на търсене, алгоритмите за търсене използват за всеки възел **п структура от данни**, състояща се от четири компонента
- » Използвайки компонентите на един родителски възел лесно могат да се изчислят компонентите на възли-наследници



Структура

- **n.State**: състояние в ПС, кореспондиращо на възела;
- **n.Parent**: възел в ПС, генерирал актуалния възел;
- **n.Action**: действие, приложено в родителския възел за генериране на настоящия;
- **n.Path-Cost**: разходи от началното състояние до възела, даден чрез указателя на родителския възел (обикновено означаваме с $g(n)$)



Child-node

```
function Child-Node (problem, parent, action) returns един възел  
  return един възел с:  
    State = problem.Result(parent.State, action);  
    Parent = parent;  
    Action = action;  
    Path-Cost = parent.path-Cost + problem.Step-Cost(parent.State, action)
```


Възли и състояния

- » До сега, не правехме ясно разграничение между възли и състояния
- » При задаване на детайлните алгоритми това разграничение е съществено
 - > **Възел:** управляваща структура за представяне дървото на търсене
 - + Възлите са разположени върху определени пътища
 - > **Състояние:** кореспондира с конфигурации в света
 - + Не се разполагат върху пътища
 - > Два различни възела могат да съдържат едно и също състояние, когато това състояние е генерирано през два различни пътя за търсене



Разполагане на възлите

» След като разполагаме с възли, трябва да намерим място, където да ги съхраняваме

> Границата трябва да се съхранява в някаква форма, така че алгоритмите за търсене лесно да могат да намират следващия за разширяване възел

+ В зависимост от избраната стратегия - такава структура може да бъде:

– FIFO (опашка)

– LIFO (стек)

– С приоритети

> Изследваното множество може да бъде реализирано посредством използване на **хеш-техники** – позволяват ефективен контрол на повтаряемите състояния

+ Въвеждане и търсене се извършва в приблизително константно време

» Канонични форми: логически еквивалентни състояния кореспондират с еднакви структури данни



Операции с опашки

» Empty(queue)

- > Връща true, ако опашката е празна

» Pop(queue)

- > Връща и премахва първия елемент от опашката

» Insert(element, queue)

- > Добавя елемент и връща резултираща опашка



Критерии за оценяване на стратегиите

» Комплексност

- > Ако съществува решение гарантирано ще го намери

» Времева комплексност

- > Изразходвано време за намиране на решение

» Пространствена комплексност

- > Необходима памет за намиране на решение

» Оптималност

- > Намира най-доброто от възможните решения





Благодаря за вниманието!